One-space dimensional Heat Conduction

Computational Methods and C++ report

M.Sc. in Software Engineering for Technical Computing

Aman Makkar

S321839

20th December 2019

Contents

Abstract

This report has been compiled based on the work carried out using various numerical schemes to find a solution of a partial differential equation. Alongside these schemes, the analytical solution has also been provided to be able to compare the solutions. Explicit schemes such as Du-Fort Frankel and Richardson methods have been computed, and for the implicit scheme, methods such as Laasonen Simple Implicit and Crank-Nicholson methods have been computed. The purpose is to be able to differentiate between Stable and Unstable schemes and how errors differ depending on step sizes, and compared to the analytical solution for a better understanding of the methods useful for Partial Differential Equations.

Nomenclature

Time step	Δt
Space step	Δx
Surface Temperature	T_{sur}
Initial Temperature	T_{in}
Diffusivity	D
Time	T
Space	x
First derivative in time	$\frac{df}{dt}$
First derivative in space	$\frac{df}{dx}$

1. Introduction

The purpose of this report is to examine the application of numerical schemes to solve partial differential equations. The finite difference method is applied, which includes discretising on a series of nodes and the Taylor series expansions for partial differential equations (PDE). The advantage of using the finite difference scheme is that it is easy to understand and it is relatively easy to code when compared to the other two methods, i.e. finite volume and finite element (Collis, 2005). Computational fluid dynamics (CFD) has been used along with experimental fluid mechanics within the industry for design purposes, for instance it can be used determine lift and drag of an Aircraft. Due to the advancements made in computer hardware and the improvements that are made consistently, CFD is at the forefront to compute numerical analysis for PDEs (Hoffman, 2000).

2. Problem statement

A wall exists with a thickness of 31cm that is infinite in all other directions, it has the initial temperature of $38℃$, on both sides of the wall the temperature has been increased and maintained at $148℃$. The wall is made of Nickel steel and has the diffusivity (D) of $93\ \mathrm{cm}^2/\mathrm{hr}$. Time steps are between 0.0 to 0.5, with the value of x to be plotted for all time steps.

$$T_{in} = 38℃$$

$$T_{sur} = 148\ ℃$$

$$D = 93\ cm^2/hr$$

$$\Delta x\ = 0.05$$

$$\Delta t = 0.01$$

The analytical solution equation has been provided as the following:

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

3. Numerical Methods

The purpose of using numerical methods is to be able to solve mathematical problems using arithmetic methods. Prior to the invention of computers, engineers and scientists solved analytical/exact equations, however most of this work was for linear problems. As most of the problems are non-linear, these methods did not carry any value. Graphs were also used to be able to solve these problems however, the results were not always precise and to plot these graphs could take a relatively long period of time. Manual calculations were also carried out however, this was also a very long, tedious and time-consuming task. (Chapra & Canale, 2015).

There are various methods that currently exist that can be used to solve these problems using computers, these include Finite difference, Finite volume and Finite element. For the purposes of this report, the finite difference method has been used to carry out the solutions and obtain the results.

However, there are a few things that need to be kept in mind when solving the numerical methods on a computer, one of them being errors. There are two sources of errors that occur in the finite difference method, i.e. Truncation error and Round off error. Truncation error can be calculated as a difference between the exact solution and the numerical solution, with an assumption that there is no round off, this is referred to as the Big-O notation. The round off errors occur due to computers only being able to contain floating-point numbers to certain precision, which leads to small round off errors in the solution. Although the round off errors can usually be negligible in a solution, there can be large errors due to the source code/algorithms that do not take this into account (Collis, 2005).

4. Methodology

For the purposes of this work, four different methods will be used to compute the solution. These methods are DuFort Frankel method and Richardson method that are Explicit schemes, Laasonen method and Crank Nicholson method that are Implicit schemes. The time step (Δt) will be used as a constant 0.01 and the space step (Δx) will be kept constant at 0.05. Furthermore, for the Laasonen Simple implicit method, additional time steps will be computed to compare the accuracy and computation time of the solution and how it differs depending on the time step.

4.1. Explicit Schemes

In order to compute the Explicit schemes, two boundary conditions and an initial condition has to be provided i.e. previous time step or space step is required in order to compute for the next time step (Hoffman, 2000). In this scheme the equations are discretised in such a way that there is only one unknown and furthermore, the stability conditions have to be satisfied as it is conditionally stable.

4.1.1. Richardson Method

The Richardson method uses central time and central space to approximate the solution. This method is unconditionally unstable and has the accuracy of $[(\Delta t)^2, (\Delta x)^2]$. The model equation for the Richardson method is (Hoffman, 2000):

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

In the heat equation, the unknown is f_i^{n+1} and therefore the equation above then equates to:

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

A further breakdown and an explanation of the Richardson method has been provided in the Appendix.

4.1.2. DuFort Frankel Method

DuFort Frankel method was introduced after the Richardson method as the purpose of this method was to make the Richardson method stable, by replacing f_i^n with the average time expression, i.e. $\frac{f_i^{n+1}+f_i^{n-1}}{2}$ (Tannehil, Anderson, & Pletcher, 1997). When applying this change to the heat equation, we get the following:

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

To simplify the above equation for f_i^{n+1}:

$$f_i^{n+1}=f_i^{n-1}-\frac{2\Delta tD}{(\Delta x)^2}(f_{i+1}^n-f_i^{n+1}-f_i^{n-1}+f_{i-1}^n)$$

Based on the von Neumann analysis it is determined that this method is unconditionally stable, which is a rarity to have in an Explicit scheme as Explicit schemes require less computational time to run equating to lower costs however, unfortunately duFort does face a disadvantage. When the equation is expanded using the Taylor series, at the third term of this expansion this error starts to increase exponentially making it inconsistent (Collis, 2005). The duFort Frankel method has the accuracy of $(\Delta x^2, \Delta t^2, \left(\frac{\Delta t}{\Delta x}\right)^2)$.

4.2. Implicit Schemes

Unlike Explicit schemes, the Implicit schemes can have multiple unknowns however, they are considered to be unconditionally stable for all time steps and they are the preferred choice of scheme to be used for Parabolic equations. The model equation can be discretised as the following for the Implicit schemes (Hoffman, 2000):

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

4.2.1. Laasonen Simple Implicit Method

The Laasonen method is forward in time and central in space. This method is unconditionally stable and has the first order accuracy of $O[\Delta t,(x)^2]$. The model equation for the Laasonen simple implicit method is (Tannehil, Anderson, & Pletcher, 1997):

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

The above can be simplified as below for f_i^n:

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

This leads to a set of Tridiagonal system of linear algebraic equations that need to be solved for the Laasonen implicit method.

4.2.2. Crank - Nicolson Method

This method is unconditionally stable that leads to a set of Tridiagonal system of linear algebraic equations, it is of order $O[(\Delta t)^2, (\Delta x)^2$. In order to provide this accuracy, at the midpoint of the time increment, difference approximations are developed. In order to do this, the first derivative is approximated by (Chapra & Canale, 2015):

$$\frac{df}{dt} \cong \frac{f_{i+1}^n - f_i^n}{\Delta t}$$

The model equation for the Crank – Nicolson method is (Hoffman, 2000):

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

5. Source Code

In order to be able to write the source code for the above methods, it was decided that in the first instance, all the above methods will be written as functions within one “main” function. The initial code was in the following hierarchy:

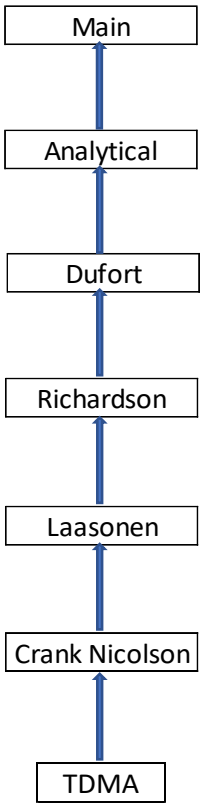

Fig 1: Initial Code design

Once the initial code was written and some initial results were obtained, the code was broken into various functions and all the functions were stored in their own header files. These header files are then called into the main file, with all the schemes then run through the main function.

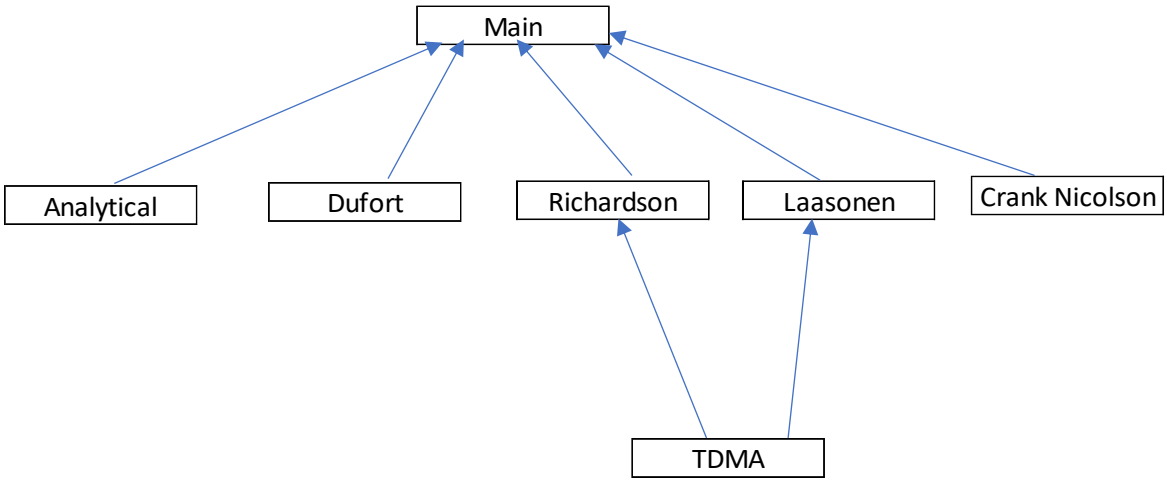

Fig 2: Final code design

In order for the TDMA header file to be used for both Laasonen and Crank Nicolson method, the "FILE_FOO_SEEN" condition was used, also referred to as a "wrapper". The compiler calls up an error if the same header file is used twice, when the wrapper is used the condition is set to false and the pre-processor skips over the contents of the file and does not call an error (Cogswell, Turkanis, & Diggins, 2005). Chrono header file was also used, the purpose is to be able to measure the time it takes to run the program, these timings were plotted in a graph and have been compared in the "Discussions" section below.

6. Discussions and Results

The solution to the given problem statement was carried out, and this included obtaining and comparing results for the DuFort Frankel, Richardson, Crank Nicolson and Laasonen methods. An Analytical solution was obtained to be able to compare the results for the above methods, to be able to determine which method is suitable for this particular heat equation. Norm 1 and Norm 2 have been plotted for all the schemes and the required computation time has also been calculated and plotted for the Laasonen method at different time steps sizes.

6.1 Qualitative Analysis

The Richardson method has provided results that have not been possible to plot on a graph as the results oscillate between very high to very low temperatures. As discussed above, this method is unconditionally unstable and the results were expected to be unstable for this method. Furthermore, this method also uses FTCS method (Forward time/Central Space) to be solved, the FTCS method equation is (Hoffman, 2000):

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

The above equation is considered to be stable if the diffusion number is $\frac{D(\Delta t)}{(\Delta x)^2} \leq \frac{1}{2}$, however for the given problem this ratio would equate to 372, which would mean that this would lead to an unstable result. Therefore, the oscillating result for this method is as expected.

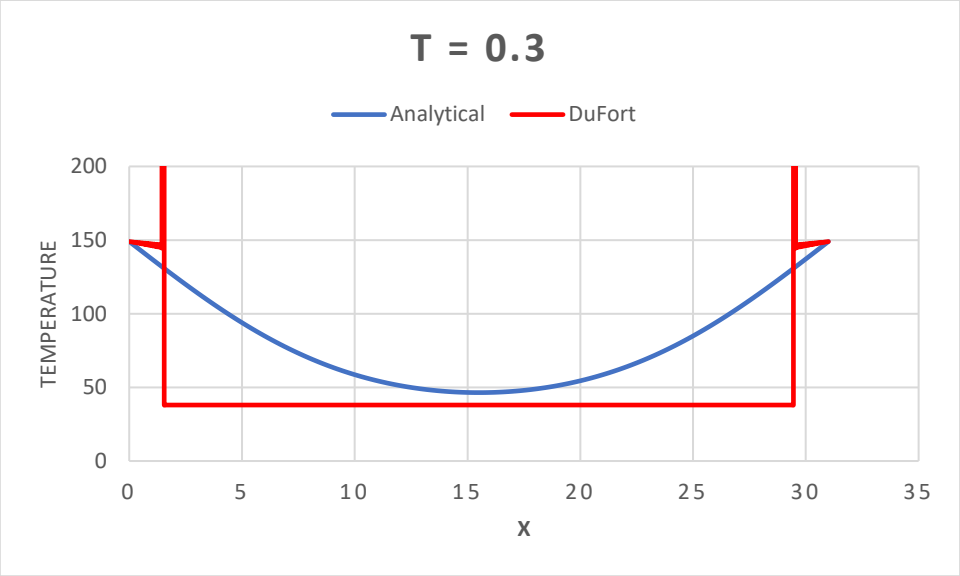

Fig 3: Du Fort Frankel at T = 0.3

As the Dufort Frankel method is a modification of the Richardson method and it also requires the FTCS method to obtain a solution, the diffusion number needs to be $\frac{D(\Delta t)}{(\Delta x)^2} \leq \frac{1}{2}$, and as seen above that is not the case and the result is expected to be unstable. As can be seen from Fig 3 above, when comparing the Analytical solution to the Numerical solution the results are not very similar, as due to the conditions provided it is an unstable method.

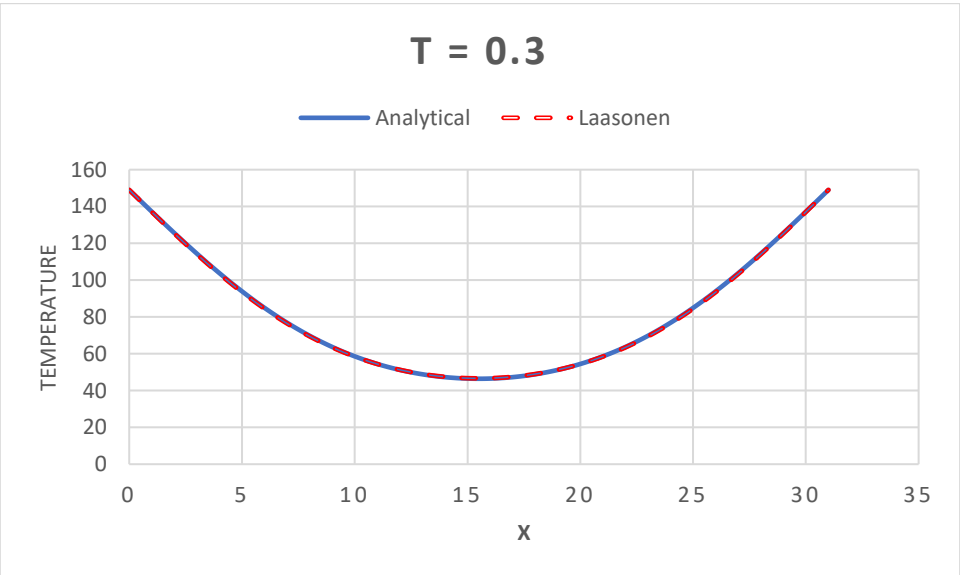

Fig 4: Laasonen T = 0.3

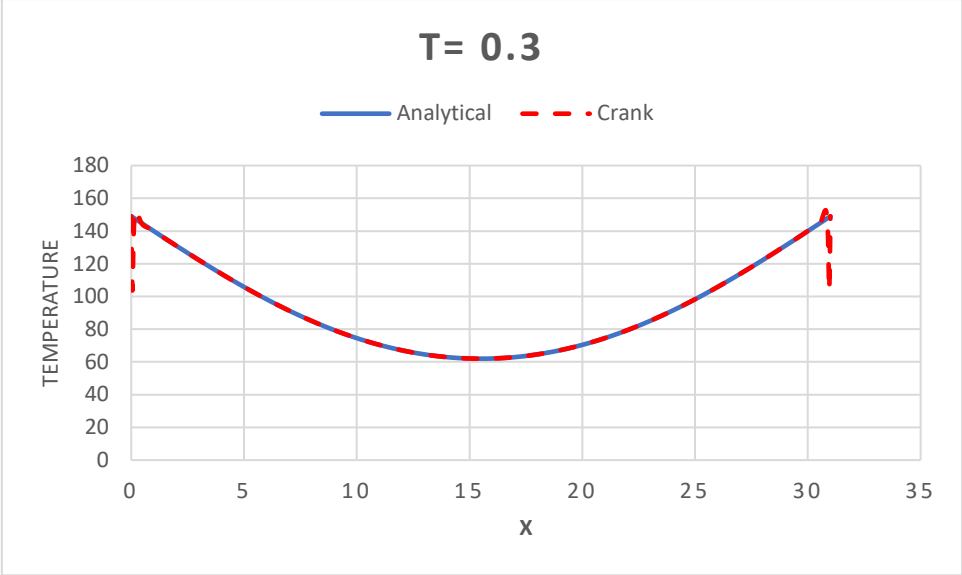

Fig 5: Crank Nicolson T = 0.3

Both the Laasonen and the Crank Nicolson method are Implicit schemes, i.e. at least two unknowns. Both of these methods are unconditionally stable as shown above and based on the theory it is expected that both of these methods would provide better results when compared to Explicit schemes. As can be seen from Fig 4 and Fig 5, when the numerical and analytical results are compared, the resulting plotted graphs provide results that are very similar to the analytical solution.

6.2 Quantitative Analysis

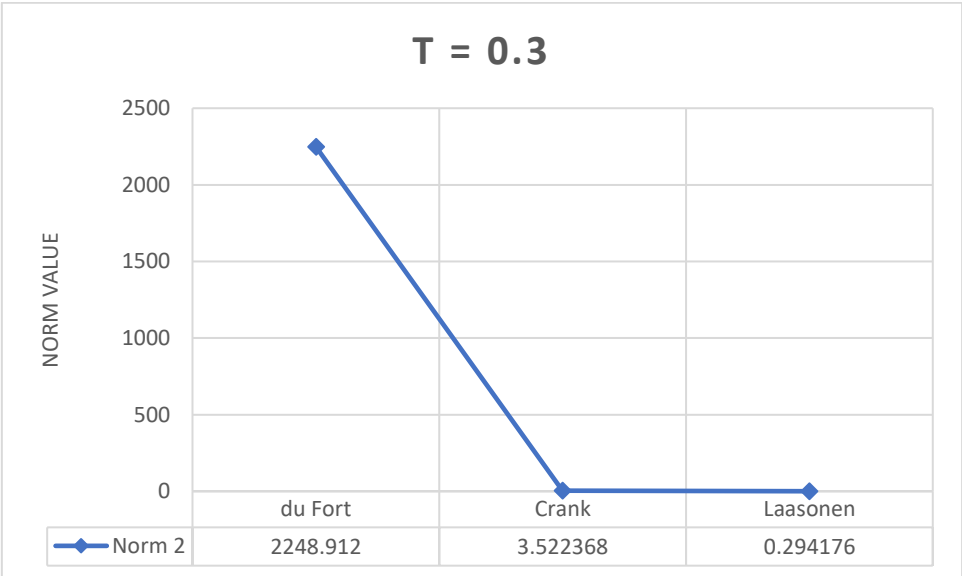

Fig 6: Norm 2 comparison of the methods, T = 0.3

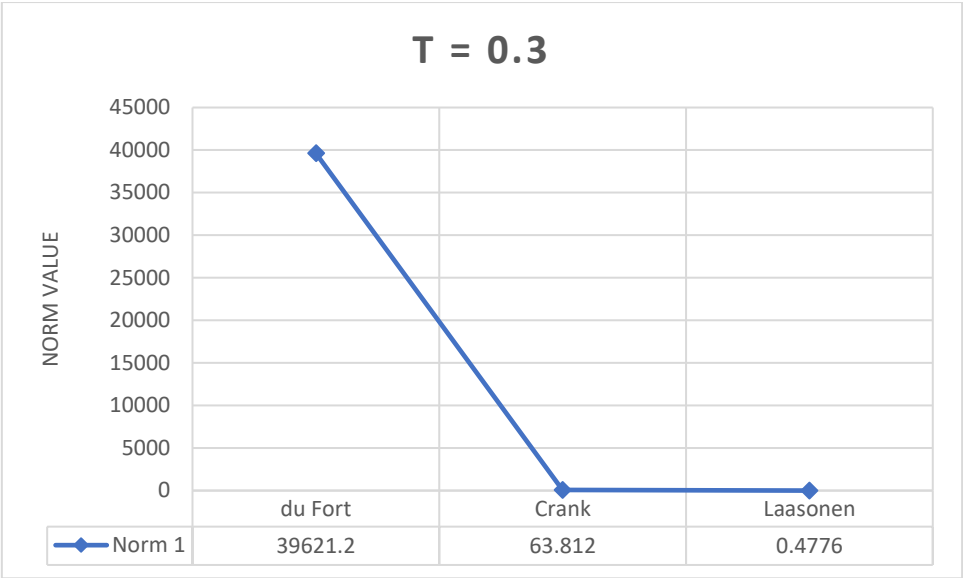

Fig 7: Norm 1 comparison of the methods, T = 0.3

Referring to Fig 6 and Fig 7, Norm 1 and Norm 2 have been plotted to show the difference between the methods in terms of the error. The purpose of using a Norm is that it is possible to express the size of the error with one number, making it easier to understand and compare. As can be seen in the figures above for this particular problem, the Laasonen method provided the closest answer to the analytical solution or the smallest error.

6.3 Laasonen Method – Comparing various Δt

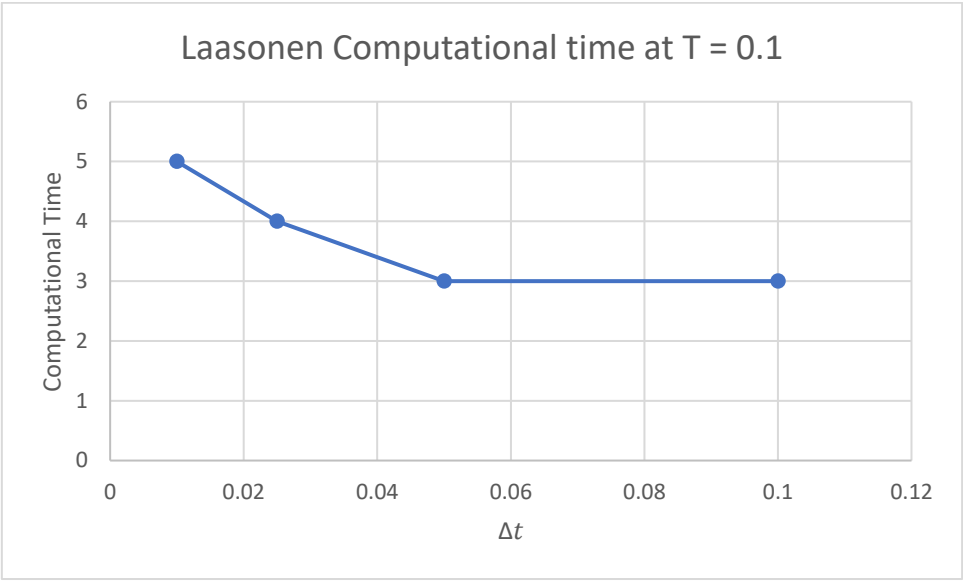

Fig 8: Laasonen Method Computational time at various Δt

The Laasonen method is an implicit scheme that is unconditionally stable, which lead to very accurate results depending on the time step and space step used. To compare how the computational time and accuracy would defer, the solution was run for various Δt, i.e. 0.01, 0.025. 0.05 and 0.1. As it can be seen from the graph above the computational time started to decrease as the Δt got bigger. A further

comparison has been made below in Fig 9 to see if the accuracy is affected depending on the value of Δt.

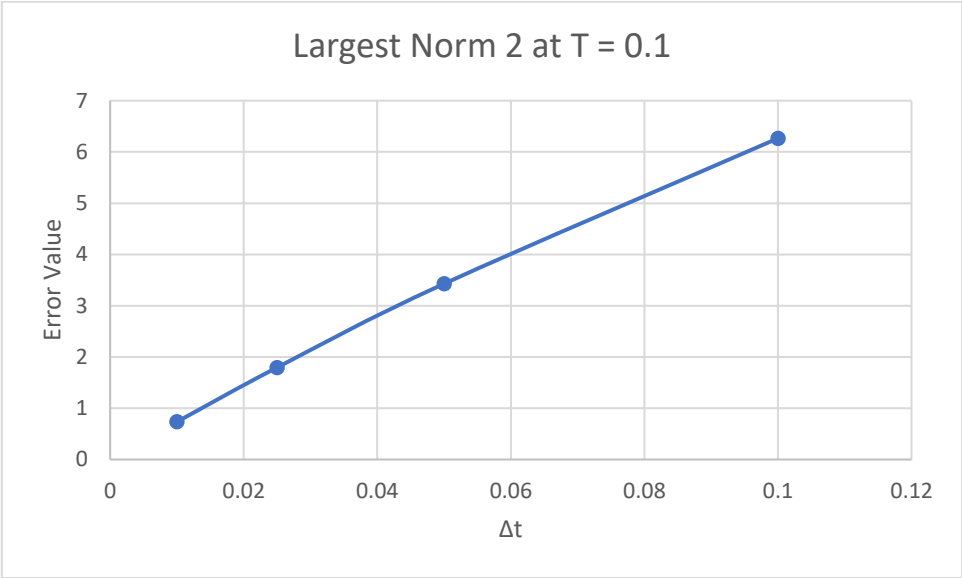

Fig 9: Laasonen Error comparison at various Δt

As can be seen from Fig 9, that as the value of Δt has got larger, the error has also increased exponentially. Therefore, when comparing the computational time to the error value, although the computational time has decreased with the larger Δt, the error value has increased. Therefore, when using the Laasonen scheme a compromise has to be made between the acceptable value of error and the amount of computational time required to run a solution.

7. Future work

A thorough comparison has been made in this report between the Explicit and Implicit schemes, for future work the source code will be written as classes rather than functions. C++ has evolved from the C programming language and one of the strongest points and the reasons for using C++ is object orientated programming (OPP) (Deitel & Deitel, 2015). The purpose of using OPP is that it is easy to reuse and modular which leads to better productivity rather than rewriting the code for every application.

8. Conclusion

After comparing the results above, it can be seen that for this particular problem the implicit schemes have provided more accurate results when compared to the explicit schemes. The Richardson method is always unstable however the DuFort Frankel method, although can provide accurate results depending on the parameters provided, in this instance have not been accurate due to the diffusion number not meeting the condition of being ≤ 0.5. Therefore, based on the literature review carried out and the explanation provided above for these two methods, the results were expected to be inaccurate and unstable, and as expected the graphs plotted prove that to be the case.

Implicit schemes have been more accurate in this instance with comparisons made between the Laasonen method and Crank Nicholson method. The Laasonen method has provided the most accurate results for this particular problem and the parameters provided. Further comparisons made between various Δt, it has been observed that although the larger Δt requires less computational time

to run, it increases the error value providing a more inaccurate answer. Therefore, a fair compromise needs to be made between the computational time and an acceptable error value.

9. Bibliography

Burg, C. O., & Erwin, T. (2009). *Application of Richardson Extrapolation to the Numerical Solution of Partial Differential Equations.* Texas: American Institute of Aeronautics and Astronautics.

Chapra, S. C., & Canale, R. P. (2015). *Numerical methods for Engineers.* New York: McGraw-Hill Education.

Cogswell, J., Turkanis, J., & Diggins, C. D. (2005). *C++ cookbook.* Sebastopol : O'Reilly Media.

Collis, S. S. (2005). *AN INTRODUCTION TO NUMERICAL ANALYSIS FOR.* Albuquerque: Sandia National Laboratories.

Davis, M. (2011). *Imperial University.* Retrieved from http://wwwf.imperial.ac.uk/~mdavis/FDM11/LECTURE_SLIDES2.PDF

Deitel, H., & Deitel, P. (2015). *C++How to program 10/e.* Boston: Pearson.

Hellevik, L. R. (2018). *Numerical Methods for Engineers.* Trondheim.

Hoffman, K. (2000). *Computational Fluid Dynamics - Volume I.* Wichita: Engineering Education system.

Silva, J. B., Romão, E. C., & de Moura, L. F. (2008). A COMPARISON OF TIME DISCRETIZATION METHODS IN THE SOLUTION OF A. *Brazilian Conference on Dynamics, Control and Applications* (pp. 1-6). São Paulo: DINCON.

Tannehil, J. C., Anderson, D. A., & Pletcher, R. H. (1997). *Computational Fluid Mechanincs and Heat Transfer.* Texas: Taylor & Francis.

Appendix A

Richardson Method

The Richardson method is an explicit scheme, it is unconditionally unstable and has the accuracy of in order of O ($\Delta x^2, \Delta t^2$). An in-depth study has been carried out in the accuracy and stability of this method and below are the mathematical calculations.

Stability

Von Neumann Analysis will be used to get a further understanding of the stability of the Richardson scheme (Hellevik, 2018) :

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

Simplifying the above for f_i^{n+1}:

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

Von Neumann Analysis:

$$f_i^n \rightarrow E_i^n = G^n e^{j\beta x_i}$$

Using Von Neumann analysis:

$$G^{n+1}e^{j.\beta y_i} = G^{n-1}e^{j.\beta y_i} + 2D[G^n e^{j.\beta y_i - 1} - 2G^n e^{j.\beta y_i} + G^n e^{j.\beta y_i + 1}]$$

Dividing the above with $G^{n-1}e^{i\beta y_j}$ and $y_j = j\,.\,h$

$$G^2 = 1 + 4DG\,.\,(\cos(\delta) - 1)$$

Using derivation on the above:

$$2G\frac{dg}{d\delta} = 4D\,[(\cos(\delta) - 1)\frac{dg}{d\delta} - Gsin(\delta)]$$

Simplifying the above, as $\delta = 0$ and $G_{1,2} = \pm 1$ and as long as $\delta = \pm\pi$ it would lead to the equation below:

$$G_{1,2} = -4D \pm \sqrt{1 + (4D)^2}$$

The above would obtain $G_2 > 1$, making the Richardson scheme unconditionally unstable.

Accuracy

The Richardson method is expressed by the equation and further work is carried out below to explain the accuracy of this method (Hoffman, 2000):

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

Using the Taylor expansion, the left side of the equation above can be expressed as:

$$\frac{f_i^n\,\Delta t\left(\frac{df}{dt}\right)_i^n + \frac{\Delta t^2}{2}\left(\frac{d^2f}{dt^2}\right)_i^n + O(\Delta t)^3 - f_i^n + \Delta t\left(\frac{df}{dt}\right)_i^n - \frac{\Delta t^2}{2}\left(\frac{d^2f}{dt^2}\right)_i^n + O(\Delta t)^3}{2\Delta t}$$

Using the Taylor expansion, the right side of the equation can be expressed as:

$$D\,\frac{f_i^n + \Delta x\left(\frac{df}{dx}\right)_i^n + \frac{\Delta x^3}{3}\left(\frac{d^3f}{dx^3}\right)_i^n + O(\Delta x^4) - 2f_i^n + f_i^n - \Delta x\left(\frac{df}{dx}\right)_i^n + \frac{\Delta x^2}{2}\left(\frac{d^2f}{dx^2}\right)_i^n - \frac{x^3}{3}\left(\frac{d^3f}{dx^3}\right)_i^n + O(\Delta x^4)}{(\Delta x)^2}$$

Simplifying the equation confirms that the accuracy of the Richardson method is order of $O(\Delta x^2, \Delta t^2)$.

$$\left(\frac{df}{dt}\right)_i^n - D\left(\frac{d^2f}{dx^2}\right)_i^n = O(\Delta x^2, \Delta t^2)$$

Appendix B

Source Code

Analytical.h

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>
#include <fstream>

using namespace std;

const double pi = 3.1415;

double analytical_solution(double exact[], double dx, double dt, int nodes, double
time) {

	ofstream ofs("Analytical.csv");

	double series;
	int acc = 100;

	for (int i = 0; i < nodes; i++) {
		series = 0;
		for (int m = 1; m <= acc; m++) {
			series += (exp(-93.0 * time * pow((m * pi) / 31.0, 2)* dt) *
((1.0 - pow(-1.0, m)) / (m * pi)) * sin((m * pi * i * dx) / 31.0));
		}
		exact[i] = 149.0 - 222.0 * series;
	}
	cout << "The analytical solution is: " << endl;
	cout << "At Time = " << fixed << setprecision(4) << time << " :" << endl;
	cout << "--------------------" << endl;
	for (int i = 0; i < nodes; i++) {
		cout << fixed << setprecision(4) << "Node #" << i << " at (x = " << (i *
dx) << ") = " << exact[i] << endl;
		ofs << fixed << setprecision(4) << "Node #" << i << " at (x = " << (i *
dx) << ") = " << exact[i] << endl;
	}
	return 0;
}
```

DuFort.h

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>
#include <fstream>

using namespace std;

double duFort(double** numerical, double dx, double dt, double time, int nodes, double
r) {
	for (int n = 0; n <= (time / dt); n++) {
		numerical[0][n] = 149.0;   //Boundary Condition at -40 and all n (node
#0)
		numerical[nodes - 1][n] = 149.0; //Boundary Condition at +40 and all n
(node #100)
	}

	//Initial Conditions #1
	for (int i = 1; i < (nodes - 1); i++) {
		numerical[i][0] = 38.0; // Initial condition at n = 0
	}

	//Initial Conditions #2
	for (int i = 1; i < (nodes - 1); i++) {
		numerical[i][1] = r * numerical[i - 1][0] + (1.0 - (2.0 * r)) *
numerical[i][0] + r * numerical[i + 1][0]; //Initial Conditon at n = 1
	}

	for (int n = 1; n < (time / dt); n++) {
		for (int i = 1; i < (nodes - 1); i++) {
			numerical[i][n + 1] = ((2.0 * r) / (1.0 + 2.0 * r)) * numerical[i
- 1][n] + ((2.0 * r) / (1.0 + 2.0 * r)) * numerical[i + 1][n] + ((1.0 - 2.0 * r) /
(1.0 + 2.0 * r)) * numerical[i][n - 1];
		}
	}
	cout << "The solution for the duFort Frankel scheme is: " << endl;
	cout << "At Time = " << time << endl;

	int n = time / dt;
	for (int i = 0; i < nodes; i++) {
		cout << fixed << setprecision(4) << "Node #" << i << " at (x = " << (i *
dx) << ") = " << numerical[i][n] << endl;
	}
	return 0;
}
```

Richadson.h

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>
#include <fstream>

using namespace std;

double Richardson(double** numerical, double dx, double dt, double time, int nodes,
double r) {
	for (int n = 0; n <= (time / dt); n++) {
		numerical[0][n] = 149.0;   //Boundary Condition at -40 and all n at
node# 0
		numerical[nodes - 1][n] = 149.0; //Boundary Condition at +40 and all n
at node# 100
	}
	//Initial Conditions #1
	for (int i = 1; i < (nodes - 1); i++) {
		numerical[i][0] = 38.0; //Initial Condition at n = 0, for all i, except
node i = 0 and i = nodes
	}
	//Initial Condition #2
	for (int i = 1; i < (nodes - 1); i++) {
		numerical[i][1] = r * numerical[i - 1][0] + (1.0 - (2.0 * r)) *
numerical[i][0] + r * numerical[i + 1][0]; //Initial Condition at n = 1, for all i,
except node node i = 0 and i = nodes
	}
	for (int n = 1; n < (time / dt); n++) {
		for (int i = 1; i < (nodes - 1); i++) {
			numerical[i][n + 1] = 2 * r * numerical[i - 1][n] - 4 * r *
numerical[i][n] + 2 * r * numerical[i + 1][n] + numerical[i][n - 1];
		}
	}
	cout << "The solution for Richardson scheme is: " << endl;
	cout << "At Time = " << time<< endl;
	int n = time / dt;
	for (int i = 0; i < nodes; i++) {
		cout << fixed << setprecision(4) << "Node #" << i << " at (x = " << (i *
dx) << ") = " << numerical[i][n] << endl;
	}
	return 0;
}
```

Laasonen.h

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>
#include <fstream>
#include "TDMA.h"

using namespace std;

double Laasonen(double** numerical, double dx, double dt, double time, int nodes,
double r) {

	ofstream ofs("Laasonen.csv");

	for (int n = 0; n <= (time / dt); n++) {
		numerical[0][n] = 149.0;
		numerical[nodes - 1][n] = 149.0;
	}

	for (int i = 1; i < (nodes - 1); i++) {
		numerical[i][0] = 38.0;
	}

	double* lower_diagnol;
	double* main_diagnol;
	double* upper_diagnol;
	double* b;

	lower_diagnol = new double[nodes - 2];
	main_diagnol = new double[nodes - 2];
	upper_diagnol = new double[nodes - 2];
	b = new double[nodes - 2];

	for (int k = 0; k < nodes - 2; k++) {
		lower_diagnol[k] = 0;
		main_diagnol[k] = 0;
		upper_diagnol[k] = 0;
		b[k] = 0;
	}

	for (int k = 1; k < nodes - 2; k++) {
		lower_diagnol[k] = -r;
	}

	for (int k = 0; k < nodes - 2; k++) {
		main_diagnol[k] = 1 + 2 * r;
	}

	double* numerical_old;
	numerical_old = new double[nodes - 2];

	for (int n = 0; n < time / dt; n++) {
		for (int k = 0; k < nodes - 3; k++) {
			upper_diagnol[k] = -r;
		}

		for (int k = 0; k < nodes - 2; k++) {
			if (k == 0) {
				b[k] = numerical[k + 1][n] + r * 149;
```

```
                        }
                        else if (k == nodes - 3) {
                                b[k] = numerical[k + 1][n] + r * 149;
                        }
                        else {
                                b[k] = numerical[k + 1][n];
                        }
                }

                TDMA(lower_diagnol, main_diagnol, upper_diagnol, b, nodes);
                for (int i = 0; i < nodes - 2; i++) {
                        numerical[i + 1][n + 1] = b[i];
                }
        }

        cout << "The solution for Laasonen equation is: " << endl;
        cout << "At Time = " << time << " :" << endl;
        int n = time / dt;
        for (int i = 0; i < nodes; i++) {
                cout << fixed << setprecision(4) << "Node #" << i << " at (x = " << (i *
dx) << ") = " << numerical[i][n] << endl;
                ofs << fixed << setprecision(4) << "Node #" << i << " at x =, " << (i *
dx) << " ,= " << numerical[i][n] << endl;
        }
        return 0;
}
```

Crank_Nicholson.h

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>
#include <fstream>
#include "TDMA.h"

using namespace std;

double Crank_Nicholson(double** numerical, double dx, double dt, double time, int
nodes, double r) {
	for (int n = 0; n <= (time / dt); n++) {
		numerical[0][n] = 149.0;
		numerical[nodes - 1][n] = 149.0;
	}

	for (int i = 1; i < (nodes - 1); i++) {
		numerical[i][0] = 38.0;
	}

	double* lower_diagnol;
	double* main_diagnol;
	double* upper_diagnol;
	double* b;

	lower_diagnol = new double[nodes - 2];
	main_diagnol = new double[nodes - 2];
	upper_diagnol = new double[nodes - 2];
	b = new double[nodes - 2];

	for (int k = 0; k < nodes - 2; k++) {
		lower_diagnol[k] = 0;
		main_diagnol[k] = 0;
		upper_diagnol[k] = 0;
		b[k] = 0;
	}

	for (int k = 1; k < nodes - 2; k++) {
		lower_diagnol[k] = -(r / 2);
	}

	for (int k = 0; k < nodes - 2; k++) {
		main_diagnol[k] = 1 + r;
	}

	double d = r / 2;
	double e = 1 - r;
	double f = r / 2;

	double* numerical_old;
	numerical_old = new double[nodes - 2];

	for (int n = 0; n < time / dt; n++) {
		for (int k = 0; k < nodes - 3; k++) {
			upper_diagnol[k] = -(r / 2);
		}

		for (int k = 0; k < nodes - 2; k++) {
			if (k == 0) {
```

```
                        b[k] = d * numerical[k][n] + e * numerical[k + 1][n] + f *
numerical[k + 2][n] + (r / 2) * 149;
                    }
                    else if (k == nodes - 3) {
                        b[k] = d * numerical[k][n] + e * numerical[k + 1][n] + f *
numerical[k + 2][n] + (r / 2) * 149;
                    }
                    else {
                        b[k] = d * numerical[k][n] + e * numerical[k + 1][n] + f *
numerical[k + 2][n];
                    }
                }

                TDMA(lower_diagnol, main_diagnol, upper_diagnol, b, nodes);
                for (int i = 0; i < nodes - 2; i++) {
                    numerical[i + 1][n + 1] = b[i];
                }
        }

        cout << "The solution for Crank Nocholson equation is: " << endl;
        cout << "At Time = " << time << " :" << endl;
        int n = time / dt;
        for (int i = 0; i < nodes; i++) {
                cout << fixed << setprecision(4) << "Node #" << i << " at (x = " << (i *
dx) << ") = " << numerical[i][n] << endl;
        }
        return 0;
}
```

TDMA.h

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>
#include <fstream>

using namespace std;

#ifndef FILE_FOO_SEEN //To be able to use the TDMA header file twice, once in Laasonen
and once in Crank_Nicholson
#define FILE_FOO_SEEN

int TDMA(double* lower_diagonal, double* main_diagonal, double* upper_diagonal,
double* b, int nodes) {
	nodes = nodes - 2;
	nodes--;
	upper_diagonal[0] /= main_diagonal[0];
	b[0] /= main_diagonal[0];

	for (int i = 1; i < nodes; i++) {
		upper_diagonal[i] /= main_diagonal[i] - lower_diagonal[i] *
upper_diagonal[i - 1];
		b[i] = (b[i] - lower_diagonal[i] * b[i - 1]) / (main_diagonal[i] -
lower_diagonal[i] * upper_diagonal[i - 1]);
	}

	b[nodes] = (b[nodes] - lower_diagonal[nodes] * b[nodes - 1]) /
(main_diagonal[nodes] - lower_diagonal[nodes] * upper_diagonal[nodes - 1]);

	for (int i = nodes; i-- > 0;) {
		b[i] -= upper_diagonal[i] * b[i + 1];
	}
	return 0;
}
#endif
```

Main.cpp

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>
#include <fstream>
#include "Analytical.h"
#include "duFort.h"
#include "Richardson.h"
#include "Laasonen.h"
#include "Crank_Nicholson.h"
#include <algorithm>
#include <chrono>

using namespace std;
using namespace std::chrono;

int main() {

	auto start = high_resolution_clock::now();

	double L = 31.0; //setting the length
	double dx = 0.05; //setting delta x
	int nodes = L / dx + 1;
	double dt; //delta t, to be entered by the user
	double time; //to be entered by the user
	double* real; //dynamic memory

	cout << "Enter a value for delta t: " << endl;
	cin >> dt;
	cout << "Enter the time for output:" << endl;
	cin >> time;

	real = new double[nodes];
	double r = (93.0 * dt) / (pow(dx, 2));

	double** numerical = new double* [nodes];
	for (int i = 0; i < nodes; i++) numerical[i] = new double[int((time / dt) +
1)];

	analytical_solution(real, dx, dt, nodes, time);

	duFort(numerical, dx, dt, time, nodes, r);

	Richardson(numerical, dx, dt, time, nodes, r);

	Laasonen(numerical, dx, dt, time, nodes, r);

	Crank_Nicholson(numerical, dx, dt, time, nodes, r);

	for (int i = 0; i < nodes; i++) delete[] numerical[i];
	delete[] numerical; //deleting the matrix set above to free up memory

	auto stop = high_resolution_clock::now();
	auto duration = duration_cast<seconds>(stop - start);
	cout << duration.count();

	return 0;
}
```