COMPUTATIONAL METHODS & C++

Heat Conduction Equation

December 11, 2017

António Pedro Araújo Fraga

Student ID: 279654

Cranfield University

M.Sc. in Software Engineering for Technical Computing

Contents

Abstract

Four numerical schemes were applied to compute a solution for a parabolic partial differential equation, the heat conduction equation. Two different types of schemes were used, explicit and implicit, and their solutions were evaluated. It could be observed the different behaviours of unstable and stable schemes. Step size variations of a stable method were studied as well. The obtained solutions were compared to the problem analytical solution in order to have a better understanding on these different behaviours.

Table 1: Nomenclature

Diffusivity	D
First derivative in time	$\frac{\partial f}{\partial t}$
First derivative in space	$\frac{\partial f}{\partial x}$
Time grid position	n
Space grid position	i
Function at time and space grid position	f_i^n
Time step	Δt
Space step	Δx
Time value	t
Space value	x
Analytical function at specific space and time values	$f(x,t)$
Initial Temperature	T_{in}
Surface Temperature	T_{sur}

Introduction

Numerical methods are used to obtain an approximated solution to problems with no given analytical solution. These methods can be used in order to save computational time, therefore they can obtain results which are similar to the real solution more efficiently. Four different schemes were applied to compute an approximated solution to a **Parabolic Partial Differential Equation**, in this case the heat conduction equation.

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

This condition had to be satisfied on a grid in space and time, which means the problem has a structured mesh type, and therefore can be represented as a grid of two dimensions. The previous equation could be written in its discretized form for each method.

Problem definition

A few initial or boundary conditions were set, including the heat conduction equation. An existing wall with **1 *ft*** thick had an initial temperature of **100ºF** and the surface temperatures at both sides were suddenly increased and maintained to **300ºF**. It is also known that the wall is composed of nickel steel (40% Ni) with a Diffusivity of **0.01** ft^2/h.

Since the wall has a 1 *ft* thickness, the problem space domain could be restricted between **0** and **1**, and the diffusivity value, which is considered constant, could be set to **0.01**. The time domain was restricted between **0** to **0.5**:

$$x \in [0,1], t \in [0,0.5]$$

$$T_{in} = 100, T_{sur} = 300$$

$$D = 0.01$$

The initial boundaries can be formalized in mathematical expressions:

$$f(x,0) = T_{in}$$

$$f(0,t) = T_{sur}$$

$$f(1,t) = T_{sur}$$

The analytical solution of this problem was given by the following expression:

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

Numerical analysis

Numerical analysis is the study of the obtained solution. Criticism is very important on this phase, since the solutions are evaluated. Digital computers have problems with round-off errors, and since values were truncated, problems with discretization errors may appear. There are some definitions related with this study: stability, convergence and approximation [1, 2].

A method is declared stable if the error doesn't grow as time advances. Theoretically, conditions that make a scheme becomes stable or unstable can be known.

Approximation can be verified by comparing the computed solution with the analytical solution, and check if there is an approximation at all.

Convergence is defined by how well the computed solution approximates to the analytical solution. This can vary with a change in the number of **time steps** or **space steps**. A smaller number of steps can lead to a bigger error, whereas a bigger number of steps can lead to a considerably more time expensive solution. Every method could be developed using **Taylor Series** [3]. This series were developed for **n** terms. Thus, every method has a given approximation factor, that could be represented in the **Big Oh** annotation. The error related with this approximation is called **Truncation Error** [2].

A quantitative analysis can be done by comparing solutions of each method. By calculating the **norms** of the error matrix, one can conclude which method is more accurate. The **error matrix** can be calculated by subtracting each cell of the **analytical** solution matrix to the cells of a method matrix. When the result is a matrix of small values, the error is small. Whenever a norm is calculated, one is able to translate an error matrix into a single value:

- **One Norm** - Which is given by adding the absolute values of each cell of the matrix.
- **Two Norm** - Which is obtained by adding the squares of every value in the matrix.
- **Uniform Norm** - Which represents the biggest error, or the maximum value in the matrix.

The second norm "punishes" the biggest values, and "regards" the lowest ones. Notice that a the square of a value between 0 and 1 is lower than the given value. Whereas the square of a value bigger than 1 is higher. Therefore, this norm is a good quality indicator.

A stencil could also be developed for each method, which relates the several grid points, revealing the dependencies for a calculation in a more graphical way.

Procedures

Four different schemes/methods were used to compute a solution for the given problem, two of them are explicit schemes, **Richardson**, **DuFort-Frankel**, and two of them are implicit schemes, **Laasonen Simple Implicit** and **Crank-Nicholson**. The space step was maintained at **0.05** *ft*, and the time step took the value of **0.01** *h*, studying every solutions in intervals

of **0.1** hours from **0.0** to **0.5**. The **Laasonen Simple Implicit** solution was also studied with different time steps, always maintaining the same space step, $\Delta x = 0.05$:

- $\Delta t = 0.01$
- $\Delta t = 0.025$
- $\Delta t = 0.05$
- $\Delta t = 0.1$

As referred, considering the initial equation, these methods can be written in its discretized form.

Explicit Schemes

This type of schemes rely only on the previous time steps to calculate the current time step solution. In the case of both used methods, they were relying in known values of the **n - 1** and **n** time steps to calculate a value for the **n + 1** time step. Thereby, the second time step can not be calculated by these methods, because there's no possible value for a negative time step. A different method, for the same equation, with two levels of time steps was used in order to overcome this situation, the **Forward in Time and Central in Space** scheme. It's known that this method is **conditionally stable**, and its stability condition is given by[3],

$$\frac{D\Delta t}{(\Delta x)^2} \leq 0.5$$

Therefore, considering $\Delta t = 0.01$, $\Delta x = 0.05$, and $D = 0.1$, this method is declared stable. It's important to have a stable solution for the first iteration, since it is a major influence on the overall solution[4]. Therefore, this iteration could be calculated with the following expression,

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

Richardson

The Richardson method can be applied by having a central in time and central in space scheme. Regarding to stability issues, this method is unconditionally unstable. This method is of order **O**(Δx^2**,** Δt^2**)** [3]. Following the heat conduction equation, the expression could be represented at **Figure 2** and could be written as following:

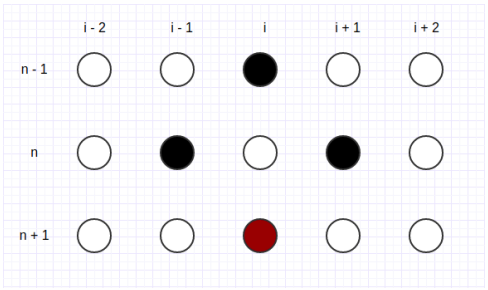


Figure 1: DuFort-Frankel's method stencil.

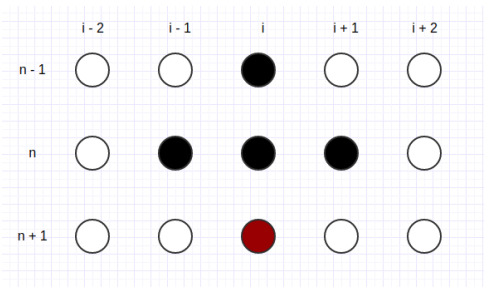


Figure 2: Richardson's method stencil.

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

Which corresponds to,

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

DuFort-Frankel

The DuFort-Frankel scheme can be applied by having central differences in both derivatives, but to prevent stability issues, the space derivative term f_i^n can be written as the average value of f_i^{n+1} and f_i^{n-1}. The method stencil can be observed at **Figure 1**. Therefore this method is of order $\mathbf{O}(\Delta x^2, \Delta t^2, (\frac{\Delta t}{\Delta x})^2)$ [3], it is declared as unconditionally stable and it may be formulated as follows:

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

Which is equivalent to,

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

Implicit Schemes

In other hand, implicit schemes rely not only on lower time steps to calculate a solution, but also on the current time step known values. Each time step solution can often be solved by applying the Thomas Algorithm, which is an algorithm that can solve tridiagonal matrix systems, $Ax = r$ [5]. This algorithm is a special case of the LU decomposition, with a better performance. The matrix A can be decomposed in a lower triangular matrix L and an

upper triangular matrix U, therefore $A = LU$[5]. This algorithm consists of two steps, the downwards phase where the equation $Lp = r$ is solved and the upwards phase, solving $Ux = p$[5], obtaining a solution for x.

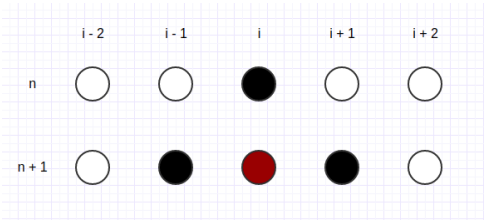


Figure 3: Laasonen's method stencil.

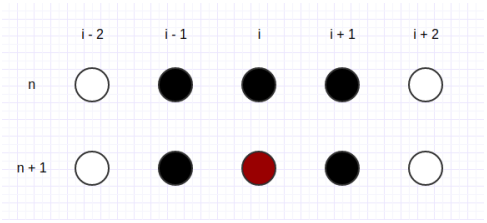


Figure 4: Crank-Nicholson's method stencil.

Laasonen Simple Implicit

The time derivative is considered forward in time. Central difference is used in space derivative, and the scheme is of order $\mathbf{O}(\Delta x, \Delta t^2)$[3], and unconditionally stable. Concluding, the below equation could be established, and it could be represented at **Figure 3**:

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

Assuming that $c = \frac{\Delta t D}{(\Delta x)^2}$, the equation could be represented as:

$$(1-2c)f_i^{n+1} = f_i^n + c\left[f_{i+1}^{n+1} + f_{i-1}^{n+1}\right]$$

The values of the first and last space position of each time step are known, they are represent by the T_{sur} value. Therefore, in every second and penultimate space step, two terms of the previous equation could be successfully inquired. For the second space step, the equation could be divided by having the unknown terms in the left side and the known terms in the right side:

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

And the same could be done for the penultimate space step:

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

For every other space steps with unknown values, the expression could be generalized as:

$$(1-2c)f_i^{n+1} - c\left[f_{i+1}^{n+1} + f_{i-1}^{n+1}\right] = f_i^n$$

Considering that the maximum number of space steps is **m**, the previous expressions could form a system of linear equations, $A.x = r$:

$$\begin{bmatrix} (1-2c) & -c & 0 & 0 & \dots & 0 & 0 \\ -c & (1-2c) & -c & 0 & \dots & 0 & 0 \\ 0 & -c & (1-2c) & -c & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & -c & (1-2c) \end{bmatrix} \begin{bmatrix} f_1^{n+1} \\ f_2^{n+1} \\ f_3^{n+1} \\ \dots \\ f_{\mathbf{m}-1}^{n+1} \end{bmatrix} = \begin{bmatrix} f_1^n + cf_0^{n+1} \\ f_2^n \\ f_3^n \\ \dots \\ f_{\mathbf{m}-1}^n + cf_{\mathbf{m}}^{n+1} \end{bmatrix}$$

Crank-Nicholson

The time derivative is considered forward in time, and the space derivative can be replaced by the average of central differences in time steps **n** and **n + 1**. The method is of order **O**(Δx^2**,** Δt^2**)** [3], is declared unconditionally stable and it could be represented at **Figure 4**. Thus:

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

In this method, the coefficient had a new value, $c = \frac{1}{2}\frac{\Delta tD}{(\Delta x)^2}$, and assuming that $p = f_{i+1}^n + f_{i-1}^n$, the equation could be written as follows,

$$(1-2c)f_i^{n+1} = (1-2c)f_i^n + c\left[f_{i+1}^{n+1} + f_{i-1}^{n+1} + p\right]$$

Following the same logical principles of the previous scheme, some expressions could be generalized for the second,

$$(1-2c)f_i^{n+1} - cf_{i+1}^{n+1} = (1-2c)f_i^n + c\left[f_{i-1}^{n+1} + p\right]$$

, penultimate,

$$(1-2c)f_i^{n+1} - cf_{i+1}^{n+1} = (1-2c)f_i^n + c\left[f_{i+1}^{n+1} + p\right]$$

, and every other space steps with unknown values.

$$(1-2c)f_i^{n+1} - c\left[f_{i+1}^{n+1} + f_{i-1}^{n+1}\right] = (1-2c)f_i^n + cp$$

Thus, a tridiagonal matrix system is obtained,

$$\begin{bmatrix} (1-2c) & -c & 0 & 0 & \dots & 0 & 0 \\ -c & (1-2c) & -c & 0 & \dots & 0 & 0 \\ 0 & -c & (1-2c) & -c & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & -c & (1-2c) \end{bmatrix} \begin{bmatrix} f_1^{n+1} \\ f_2^{n+1} \\ f_3^{n+1} \\ \dots \\ f_{\mathbf{m}-1}^{n+1} \end{bmatrix} = \begin{bmatrix} (1-2c)f_1^n + c\left[f_0^{n+1} + p\right] \\ (1-2c)f_2^n + cp \\ (1-2c)f_3^n + cp \\ \dots \\ (1-2c)f_{\mathbf{m}-1}^n + c\left[f_{\mathbf{m}}^{n+1} + p\right] \end{bmatrix}$$

Solution Design

The code was first planned with an initial structure and suffered incremental upgrades. A **method** class was created, being a prototype with multiple inheritance, containing three sub classes: **Analytical**, **Implicit** and **Explicit**. Therefore, the **Implicit** class is an Abstract class as well. This class has three sub classes, representing the three explicit methods used in this problem. Similarly, the **Implicit** class is also an abstract class, having two implicit methods classes as sub classes. The previously described inheritance structure can be more easily visualized on **Figure 5**.

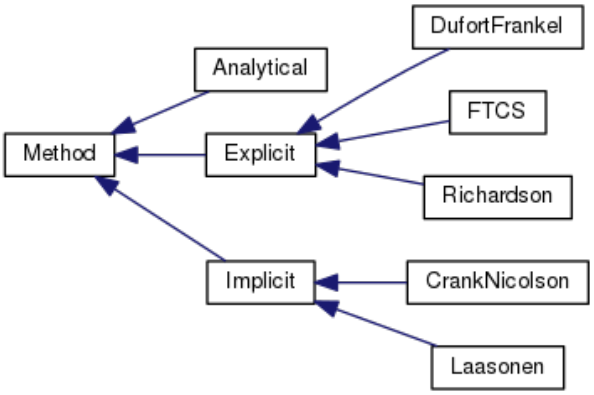


Figure 5: Method Class inheritance diagram.

A Method class contains a **Problem** object. The **Problem** class represents the Heat Conduction problem, containing informations about the time and space steps, the solution and initial conditions.

An **Input and Output Manager** class was developed so that the code related with plots and tables exportations could be separated from the logical source code. This class was developed with several methods regarding data interpretation and structuration in order to easily export plot charts. A **gnuplot c++ library** was used, therefore the gnuplot syntax could

be directly used from the c++ code, cutting down the need of developing external bash scripts for this specific purpose.

Despite the referred classes, a header file with useful **macros** was declared. This file contains information about which conditions to test, like the initial temperature and the surface temperature. Therefore, if for some reason, one of this values changes, it can be easily corrected.

The **Matrix** and **Vector** classes, which were provided in the c++ module were reused to represent a solution matrix or a solution vector of a certain iteration.

The several objects in this structure could be instantiated in the main file, calling methods to compute the several solutions and to export their plot charts. The previously described classes can be represented in the **Figure 6** diagram.

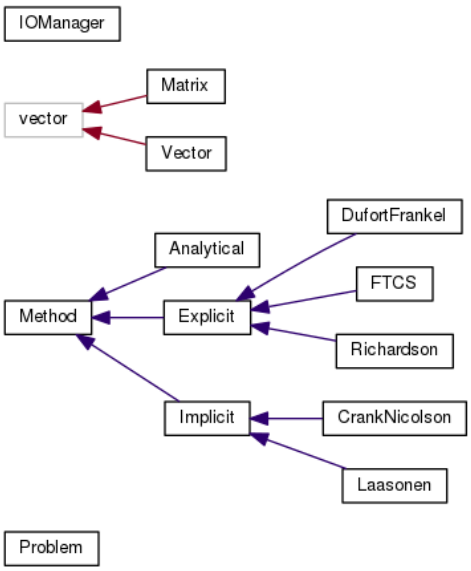


Figure 6: Class Diagram.

Results & Discussion

The results of the four methods, **Richardson**, **DuFort-Frankel**, **Laasonen Simple Implicit** and **Crank-Nicholson** can be seen in the following figures/tables. These results were used to analyze each solution quantitatively and qualitatively. In most of the plot charts, the obtained solution was compared to the analytical solution so that it would be possible to

realize whether the solution was a good approximation or not. Notice that the next results are regarding to the "default" values of time and space steps, $\Delta t = 0.01$ and $\Delta x = 0.05$.

Table 2: Richardson method error table.

x / t	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
0.1	1.05136e+06	631856	123707	8417.73	300.81	8417.73	123707	631856	1.05136e+06
0.2	1.33245e+11	1.39e+11	7.02854e+10	2.06123e+10	6.93136e+09	2.06123e+10	7.02854e+10	1.39e+11	1.33245e+11
0.3	2.14659e+16	2.74969e+16	1.97012e+16	9.88337e+15	5.98161e+15	9.88337e+15	1.97012e+16	2.74969e+16	2.14659e+16
0.4	3.91917e+21	5.60267e+21	4.87086e+21	3.31281e+21	2.58429e+21	3.31281e+21	4.87086e+21	5.60267e+21	3.91917e+21
0.5	7.74272e+26	1.19047e+27	1.18021e+27	9.72231e+26	8.60626e+26	9.72231e+26	1.18021e+27	1.19047e+27	7.74272e+26

By examining **Table 2**, it could be concluded that the solution given by the Richardson method was considerably different from the analytical solution. This was due to the fact that this method is declared as **unconditionally unstable**. As referred before, when a method is declared unstable, the error grows as the time advances. The error growth was responsible for obtaining a different solution, or a solution to a different problem. The mathematical calculations regarding the stability and accuracy properties of this method can be found under the appendix section.

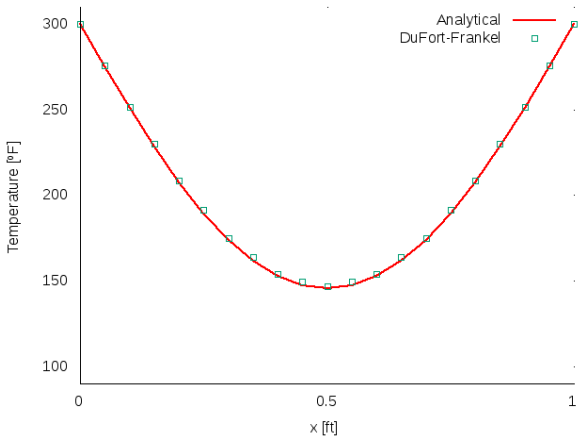


Figure 7: DuFort-Frankel's solution at $t = 0.5$.

When looking at **Figure 7**, it can be observed that the DuFort-Frankel solution is quite approximated to the real solution. This scheme, as it could be observed at **Figure 10**, is more time efficient comparing to the implicit unconditionally stable methods, the only disadvantage is the fact that it requires a different method for the first iteration.

Similarly of what could be concluded on DuFort-Frankel results, by observing **Figure 8** and **Figure 9**, it can also be deducted that these are good solutions. These schemes, Crank-Nicholson and Laasonen, are unconditionally stable as well. Therefore good results were expected.

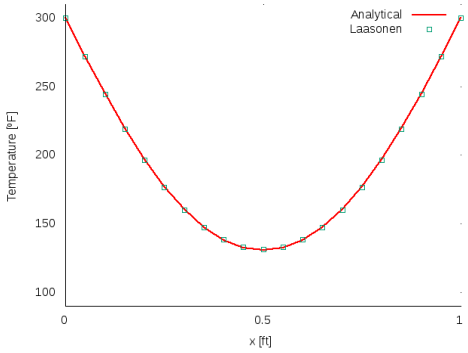


Figure 8: Laasonen's solution at $t = 0.4$.

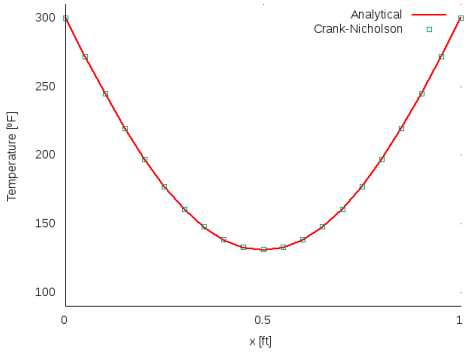


Figure 9: Crank-Nicholson's solution at $t = 0.4$.

In other hand, when a quantitative analysis was done, it could be seen that the Crank Nicholson scheme is more accurate than the Laasonen and DuFort-Frankel methods. By looking at **Figure 10**, it can be observed that the second norm value of the **Error matrix** of this scheme is smaller than the values obtained by the other methods **Error Matrices**.

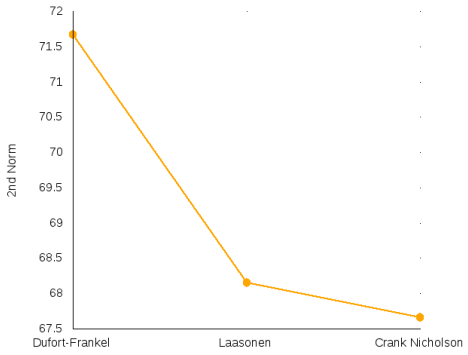


Figure 10: 2nd norm values of Error Matrices.

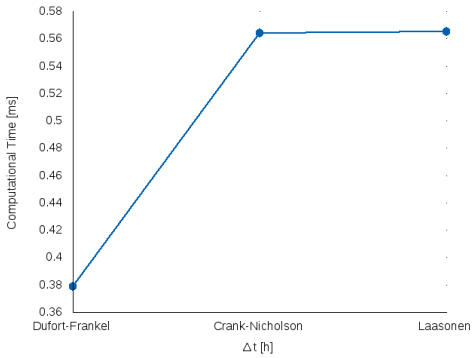


Figure 11: Computational times of stable methods.

Laasonen Implicit Scheme: study of time step variation

Laasonen Implicit Scheme is an unconditionally stable scheme to solve Parabolic Partial Differential Equations. Therefore, with the right time and space step, there's almost no error related to the development of its results throughout the time advancement.

A reduction on these steps led to a higher computational time, since there's more calculations to be made. Whereas steps with higher values led to more inaccurate results[7]. This phenomenon could be explained with a concept that was introduced earlier, the **truncation error**[3]. This error can only be avoided with exact calculations, but can be reduced by applying a larger number of smaller intervals or steps. As referred before, different results of this method were studied by changing the time step size. The space step was maintained, $\Delta x = 0.05$.

Table 3: Laasonen method error table for the several Δt at $t = 0.5$

x / Δt	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
0.01	0.288694	0.385764	0.255427	0.0405061	-0.0611721	0.0405061	0.255427	0.385764	0.288694
0.025	0.738044	1.0344	0.805442	0.368491	0.157551	0.368491	0.805442	1.0344	0.738044
0.05	1.53627	2.15669	1.71375	0.864487	0.457364	0.864487	1.71375	2.15669	1.53627
0.1	3.29955	4.49523	3.46045	1.7082	0.898726	1.7082	3.46045	4.49523	3.29955

Table 3 and **figure 12** could support the previous affirmations. While observing **table 3**, it could be seen that the error is larger for bigger time steps, as it was expected. Whereas when observing **figure 12**, it can be identified a reduction in computational time as the **time step** becomes larger.

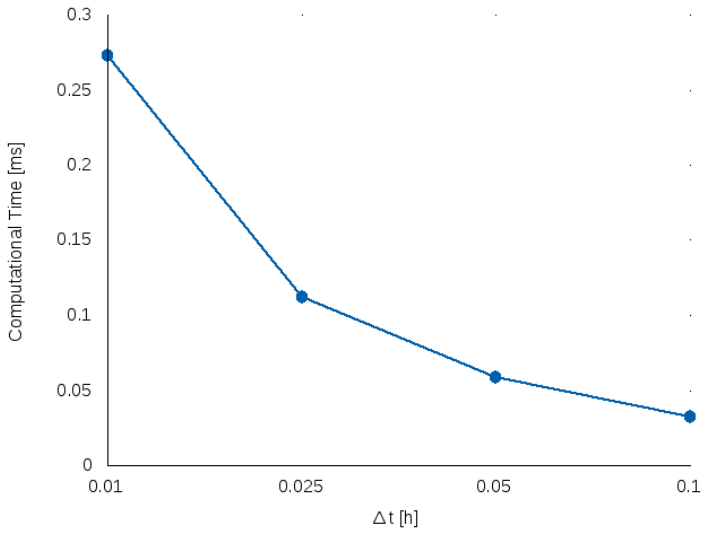


Figure 12: Laasonen method computational times for the several Δt.

Conclusions

The obtained results could support the theoretical concepts. Unstable methods demonstrated an error growth through the time progress. The **Forward in Time, Central in Space** explicit scheme was stable with the given initial conditions, therefore it could support a good solution for the explicit stable scheme, **DuFort-Frankel**. As referred, the solution of the DuFort-Frankel method strongly depends on the first iteration solution.

It could be observed that smaller steps can lead to a time expensive solution, whereas larger steps lead to an error increase. Stable methods could give a good solution with the right time and space steps, but by analysing the second norm value, it was concluded that the Crank-Nicholson method is more accurate. This is due to the fact that this method has a better approximation order.

It is important to have a balance between the two problems (time and approximation), a method should be computed in an acceptable time, and still obtain a good result. In realistic scenarios the problem solution is not known, therefore error estimates are impractical. The used step size should be small as possible, as long as the solution is not dominated with round-off errors. The solution must be obtained with a number of steps that one has time to compute.

References

[1] Gilberto E. Urroz, July 2004, *Convergence, Stability, and Consistency of Finite Difference Schemes in the Solution of Partial Differential Equations,* Available at: <http://ocw.usu.edu/Civil_and_Environmental_Engineering/Numerical_Methods_in_Civil_Engineering/StabilityNumericalSchemes.pdf> [Accessed 2 October 2017]

[2] S. Scott Collis, April 26, 2005, *An Introduction to Numerical Analysis for Computational Fluid Mechanics,* [Accessed 2 October 2017]

[3] Klaus A. Hoffman, Steve T. Chiang, August 2000, *Computational Fluid Dynamics, Volume 1* [Accessed 27 October 2017]

[4] Richard H. Pletcher, Jhon C. Tannehill, Dale A. Anderson, 2013, *Computational Fluid Mechanics and Heat Transfer, Third Edition*, [Accessed 27 October 2017]

[5] W. T. Lee, *Tridiagonal Matrices: Thomas Algorithm*, Available at: <https://www3.ul.ie/wlee/ms6021_thomas.pdf> [Accessed 28 October 2017]

[6] *Error in Euler's Method*, Available at: <http://www.math.unl.edu/~gledder1/Math447/EulerError> [Accessed 2 November 2017]

[7] John D. Cook, February 22 2008, *Step size for numerical differential equations*, Available at: <https://www.johndcook.com/NumericalODEStepSize.pdf> [Accessed 3 November 2017]

[8] B.J.P. Kaus, *Explicit versus implicit Finite Difference Schemes*, Available at: <https://www.geowiss.uni-mainz.de/Dateien/Finite_Differerence_Timpl_expl.pdf> [Accessed 4 November 2017]

[9] Markus Schmuck, *Numerical Methods for PDEs*, Available at: <http://www.macs.hw.ac.uk/~ms713/lecture_9.pdf> [Accessed 10 November 2017]

Appendices

Richardson Method Analysis

Accuracy

Every term can be developed with Taylor expansions:

$$f(x+\Delta t) = f_i^n + \Delta t \left(\frac{\partial f}{\partial t}\right)_i^n + \frac{\Delta t^2}{2}\left(\frac{\partial^2 f}{\partial t^2}\right)_i^n + O(\Delta t^3)$$

$$f(x-\Delta t) = f_i^n - \Delta t \left(\frac{\partial f}{\partial t}\right)_i^n + \frac{\Delta t^2}{2}\left(\frac{\partial^2 f}{\partial t^2}\right)_i^n + O(\Delta t^3)$$

$$f(x+\Delta x) = f_i^n + \Delta x \left(\frac{\partial f}{\partial x}\right)_i^n + \frac{\Delta x 2}{2}\left(\frac{\partial^2 f}{\partial x^2}\right)_i^n + \frac{\Delta x 3}{3}\left(\frac{\partial^3 f}{\partial x^3}\right)_i^n + O(\Delta x^4)$$

$$f(x-\Delta x) = f_i^n - \Delta x \left(\frac{\partial f}{\partial x}\right)_i^n + \frac{\Delta x^2}{2}\left(\frac{\partial^2 f}{\partial x^2}\right)_i^n - \frac{\Delta x 3}{3}\left(\frac{\partial^3 f}{\partial x^3}\right)_i^n + O(\Delta x^4)$$

By replacing every term expansion in the **Richardson's Method equation**, one should obtain the **Heat Conduction Equation**.

Starting by the left side of the **Richardson's Method equation**,

$$\frac{f_i^{n+1} - f_i^{n-1}}{2\Delta t}$$

it can be converted into,

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

which can be translated to,

$$\left(\frac{\partial f}{\partial t}\right)_i^n + O(\Delta t^2)$$

When looking at the right side of the equation,

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

one can replace the terms by their expansions as well,

$$D\frac{f_i^n+\Delta x\left(\frac{\partial f}{\partial x}\right)_i^n+\frac{\Delta x^2}{2}\left(\frac{\partial^2 f}{\partial x^2}\right)_i^n+\frac{\Delta x3}{3}\left(\frac{\partial^3 f}{\partial x^3}\right)_i^n+O(\Delta x^4)-2f_i^n+f_i^n-\Delta x\left(\frac{\partial f}{\partial x}\right)_i^n+\frac{\Delta x^2}{2}\left(\frac{\partial^2 f}{\partial x^2}\right)_i^n-\frac{\Delta x3}{3}\left(\frac{\partial^3 f}{\partial x^3}\right)_i^n+O(\Delta x^4)}{(\Delta x)^2}$$

which can be translated to,

$$D\left(\frac{\partial^2 f}{\partial x^2}\right)_i^n+O(\Delta x^2)$$

Therefore the obtained equation is of order of $O(\Delta x^2,\Delta t^2)$,

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

Stability

The Richardson Scheme is given by,

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

Using the von-Neumann analysis one can write:

$$f_i^n=\xi^n\times e^{j\omega i}$$

and by inserting this expression in the scheme, assuming that $q=D\frac{\Delta t}{(\Delta x)^2}$, one obtains,

$$\xi^2+8q\sin^2(\tfrac{1}{2}\omega)\xi-1=0$$

which is a quadratic expression, thus it can be solved with,

$$\xi=-4q\sin^2\tfrac{\omega}{2}\pm\sqrt{1+16q^2\sin^4\tfrac{\omega}{2}}$$

assuming that $r=4q\sin^2\frac{\omega}{2}$:

$$\xi=-r\pm\sqrt{1+r^2}$$

one will obtain $|\xi|>1$ when $r>0$, and because stability requires that $|\xi|\leq 1$, the Richardson scheme is **unconditionally unstable**.

Source Code

main.cpp

```
#include <iostream>
#include "methods/analytical.h"
#include "methods/explicit/dufort_frankel.h"
#include "methods/explicit/richardson.h"
#include "methods/implicit/laasonen.h"
#include "methods/implicit/crank_nicolson.h"
#include "io/iomanager.h"

/*
  Main file - creates a problem, solving it with the different methods, ending
      by exporting the results.
*/
int main() {
  IOManager io_manager;
  Problem default_problem(DELTA_T, DELTA_X);
  Analytical analytical(default_problem);
  DufortFrankel dufort_frankel(default_problem);
  Richardson richardson(default_problem);
  CrankNicolson crank_nicolson(default_problem);

  std::vector<Method*> solutions = {&analytical, &richardson, &dufort_frankel,
      &crank_nicolson};

  for (int index = 0; index < DELTA_T_LASSONEN.size(); index++) {
    double delta_t = DELTA_T_LASSONEN[index];
    Problem laasonen_problem = Problem(delta_t, DELTA_X);
    Laasonen * laasonen = new Laasonen(laasonen_problem);
    solutions.push_back(laasonen);
  }

  for (int index = 0; index < solutions.size(); index++) {
    solutions[index]->compute();
    if (solutions[index]->get_name() != ANALYTICAL) {
```

```
      solutions[index]->compute_norms(analytical.get_solution());
    }
  }

  std::vector<Method*> methods(solutions.begin() + 1, solutions.end());
  io_manager.export_outputs(solutions[0], methods);

  return 0;
}
```

variants/problem.h

```
#ifndef PROBLEM_H //include guard
#define PROBLEM_H

#include "../variants/utils.h" //relevant utils
#include "../grid/matrix.h" // declare the structure of the matrix object

/**
*  A Problem class to structure relevant information related with the problem
*
* The Problem class provides:
* \n-basic constructors for creating a Problem object.
* \n-acessor methods to retrieve valuable information
* \n-mutator methods to change the solution system
*/
class Problem {
private:
  double delta_x; /**< Private double delta_x. Space step of the solution. */
  double delta_t; /**< Private double delta_t. Time step of the solution. */

  unsigned int x_size; /**< Private unsigned int x_size. Space size of the
      solution. */
  unsigned int t_size; /**< Private unsigned int t_size. Time size of the
      solution. */
```

```
  Vector x_values; /**< Private Vector x_values. Space correspondent value for
      each column index. */
  Vector t_values; /**< Private Vector t_values. Time correspondent value for
      each row index. */

  Matrix solution; /**< Private Matrix solution. Matrix containing the computed
      solution. */

  // PRIVATE MUTATOR METHODS

  /**
  * Normal private set method.
  * Intialize Vector x_values with the correct values.
  * @see x_values
  */
  void set_x_values();

  /**
  * Normal private set method.
  * Intialize Vector t_values with the correct values.
  * @see t_values
  */
  void set_t_values();

public:
  // CONSTRUCTORS
  /**
  * Default constructor. Intialize an empty Problem object
  * @see Problem(double dt, double dx)
  */
  Problem();

  /**
  * Intialize Problem object with specific time and space steps
  * @see Problem()
  * @param dt Time step to assign
  * @param dx Space step to assign
```

```
* @exception out_of_range ("space step can't be negative or zero")
* @exception out_of_range ("time step can't be negative or zero")
*/
Problem(double dt, double dx);

// PUBLIC ACCESSOR METHODS
/** Normal public get method that returns an unsigned int, the number of
    columns of the solution
* @return unsigned int. The number of columns of the solution.
*/
unsigned int get_xsize();

/** Normal public get method that returns an unsigned int, the number of rows
    of the solution
* @return unsigned int. The number of rows of the solution.
*/
unsigned int get_tsize();

/** Normal public get method that returns a double, the space step value of
    the solution
* @return double. The space step value of the solution.
*/
double get_deltax();

/** Normal public get method that returns a double, the time step value of
    the solution
* @return double. The time step value of the solution.
*/
double get_deltat();

/** Normal public get method that returns a Vector, containing the space
    values in each column
* @return Vector. The space values in each column.
*/
Vector get_xvalues();
```

```
  /** Normal public get method that returns a Vector, containing the time
      values in each row
  * @return Vector. The time values in each row.
  */
  Vector get_tvalues();

  /** Normal public get method that returns a Vector, containing the initial
      boundaries in the first row of the solution
  * @return Vector. The initial boundaries in the first row of the solution.
  */
  Vector get_first_row();

  /** Normal public get method that returns a Matrix, containing the solution
      solution.
  * @return Matrix*. The solution solution.
  */
  Matrix *get_solution();

  // PUBLIC MUTATOR METHODS

  /**
  * Normal public set method.
  * replace a row of the solution for a given Vector.
  * @param step Vector conatining the new values.
  * @param time Corresponding row to be replaced
  */
  void set_time_step(Vector step, double time);

  /**
  * Normal public set method.
  * set the problem initial boundaries.
  */
  void set_initial_conditions();
};

#endif
```

variants/problem.cpp

```
#include "problem.h"

// CONSTRUCTORS
/*=
 *Default constructor
 */
Problem::Problem() {}



/*
 * Alternate constructor - creates a problem with a specific time and space step
 */

Problem::Problem(double dt, double dx) {
  //check the input
  if (dx <= 0) throw std::out_of_range("space step can't be negative or zero");
  if (dt <= 0) throw std::out_of_range("time step can't be negative or zero");

  // set time and space steps
  delta_x = dx;
  delta_t = dt;

  // set time and space size
  x_size = (int)(THICKNESS / delta_x);
  t_size = (int)(TIMELIMIT / delta_t);

  // initializes the solution and set space value in each column and time value
      in each row
  solution = Matrix(NUMBER_TIME_STEPS, x_size + 1);
  set_x_values();
  set_t_values();

  // set the initial boundaries
  set_initial_conditions();
}
```

```
//PRIVATE MUTATOR METHODS

/*
* private mutator method - set space value in each column
*/
void Problem::set_x_values() {
  x_values = Vector(x_size + 1);
  for (unsigned int index = 0; index <= x_size; index++) {
    x_values[index] = delta_x * index;
  }
}

/*
* private mutator method - set time value in each column
*/
void Problem::set_t_values() {
  t_values = Vector(NUMBER_TIME_STEPS);
  for (unsigned int time = 0; time < NUMBER_TIME_STEPS; time++) {
    t_values[time] = double(time) / 10.0;
  }
}

//PUBLIC MUTATOR METHODS

/*
* public mutator method - set initial boundaries
*/
void Problem::set_initial_conditions() {
  for (unsigned int index = 1; index < x_size; index++) {
    solution[0][index] = INITIAL_TEMPERATURE;
  }
  for (unsigned int time = 0; time < NUMBER_TIME_STEPS; time++) {
    solution[time][0] = solution[time][x_size] = SURFACE_TEMPERATURE;
  }
}
```

```
/*
* public mutator method - set solution row
*/
void Problem::set_time_step(Vector step, double time) {

  //checks if value is in vector
  int position = t_values.find(time);
  if (position != -1) {
    solution.set_row(position, step);
  }
}

//PUBLIC ACCESSOR METHODS

/*
* public accessor method - get first row of the solution
*/
Vector Problem::get_first_row() {
  return solution[0];
}

/*
* public accessor method - get number of columns of the solution
*/
unsigned int Problem::get_xsize() {
  return x_size;
}

/*
* public accessor method - get number of rows of the solution
*/
unsigned int Problem::get_tsize() {
  return t_size;
}

/*
* public accessor method - get space step
```

```
*/
double Problem::get_deltax() {
  return delta_x;
}

/*
* public accessor method - get time step
*/
double Problem::get_deltat() {
  return delta_t;
}

/*
* public accessor method - get Vector of corresponding space values for each
    column index
*/
Vector Problem::get_xvalues() {
  return x_values;
}

/*
* public accessor method - get Vector of corresponding time values for each row
    index
*/
Vector Problem::get_tvalues() {
  return t_values;
}

/*
* public accessor method - get solution solution
*/
Matrix *Problem::get_solution() {
  return &solution;
}
```

variants/utils.h

```
#ifndef UTILS_H // include guard
#define UTILS_H

#include <cmath> // PI calculation
#include <string> // string usage
#include <vector>

const double DELTA_T = 0.01; /**< Macro double. The default time step. */
const double DELTA_X = 0.05; /**< Macro double. The default space step. */

const std::vector<double> DELTA_T_LASSONEN = {0.01, 0.025, 0.05, 0.1}; /**<
    Macro double. Time steps to study in Laasonen Implicit Scheme. */

const double DIFUSIVITY = 0.1; /**< Macro double. The default value of
    difusivity. */
const double THICKNESS = 1.0; /**< Macro double. The default value of
    thickness. */
const double TIMELIMIT = 0.5; /**< Macro double. The default value of time
    limit. */

const double SURFACE_TEMPERATURE = 300.0; /**< Macro double. The default
    surface temperature. */
const double INITIAL_TEMPERATURE = 100.0; /**< Macro double. The default
    initial temperature. */

const double NUMBER_TIME_STEPS = 6.0; /**< Macro double. The default limit of
    time steps. 0, 0.1, 0.2, 0.3, 0.4, 0.5 */
const unsigned int NUMBER_OF_EXPANSIONS = 20; /**< Macro unsigned int. Number
    of expansions to calculate the analytical solution sum expansion. */

const double PI = std::atan(1) * 4; /**< Macro double. Approximated value of
    PI. */

const std::string OUTPUT_PATH = "../outputs"; /**< Macro string. Default
    outputs path. */
```

```
const std::string ANALYTICAL = "Analytical"; /**< Macro string. Forward in Time
    and Central in Space method name. */
const std::string FORWARD_TIME_CENTRAL_SPACE = "Forward Time Central Space";
    /**< Macro string. Forward in Time and Central in Space method name. */
const std::string RICHARDSON = "Richardson"; /**< Macro string. Richardson
    method name. */
const std::string DUFORT_FRANKEL = "DuFort-Frankel"; /**< Macro string.
    DuFort-Frankel method name. */
const std::string LAASONEN = "Laasonen"; /**< Macro string. Laasonen method
    name. */
const std::string CRANK_NICHOLSON = "Crank-Nicholson"; /**< Macro string.
    Crank-Nicholson method name. */

#endif
```

methods/analytical.h

```
#ifndef ANALYTICAL_H //include guard
#define ANALYTICAL_H

#include "method.h" // inheritance

/**
*  An Analytical class to compute the solution with standard procedures
*  \n The implementation is derived from the Method Object
*
* The Analytical class provides:
* \n-a basic constructor for an object,
* \n-a method to compute a solution with the correct procedures
*/
class Analytical: public Method {
  unsigned int nr_of_expansions; /**< Private unsigned int nr_of_expansions.
      Limit of expansions to do in the sum used to compute the solution. */
public:
  // CONSTRUCTORS

  /**
```

```
  * Default constructor. Intialize a Analytical object
  */
  Analytical(Problem problem);

  // MUTATOR METHODS

  /**
  * Normal public method.
  * compute the solution with specific given rules
  */
  void compute_solution();
};

#endif
```

methods/analytical.cpp

```
#include "analytical.h"


// CONSTRUCTORS
/*=
 *Default constructor, creates an object to compute an analytical solution for
     the given parabolic problem.
 */
Analytical::Analytical(Problem problem)
: Method(problem) {
  name = ANALYTICAL;
  this->nr_of_expansions = NUMBER_OF_EXPANSIONS;
}

/*
 * Computes the analytical solution
 */
void Analytical::compute_solution() {
  Matrix * solution = problem.get_solution();
  Vector t_values = problem.get_tvalues();
```

```
  Vector x_values = problem.get_xvalues();
  unsigned int x_size = problem.get_xsize();

  // iterates through the solution columns
  for (unsigned int t = 0; t < NUMBER_TIME_STEPS; t++) {

    // iterates through the solution rows
    for (unsigned int x = 0; x <= x_size; x++) {
      double sum = 0.0;
      // expansions
      for (unsigned int m = 1; m <= this->nr_of_expansions; m++) {
        double m_double = (double)m;
        sum += exp(-DIFUSIVITY * pow(m_double * PI / THICKNESS, 2) *
            t_values[t]) * (1 - pow(-1, m_double)) / (m_double * PI) *
            sin(m_double * PI * x_values[x] / THICKNESS);
      }
      // assigns the correct value to a position
      (*solution)[t][x] = SURFACE_TEMPERATURE + 2.0 * (INITIAL_TEMPERATURE -
          SURFACE_TEMPERATURE) * sum;
    }
  }
}
```

methods/method.h

```
#ifndef METHOD_H // include guard
#define METHOD_H

#include "../variants/problem.h" // declare the problem structure


/**
*  A Method class to structure information used to solve the problem
*
* The Method class provides:
* \n-basic constructors for creating a Method object.
* \n-acessor methods to retrieve valuable information
```

```
* \n-mutator methods to change the problem grid system
*/
class Method {
private:
  double one_norm;
  double two_norm;
  double uniform_norm;

  double computational_time; /**< Private double computational_time. Elapsed
      time throughout the solution computation. */
protected:
  Problem problem; /**< Protected Problem problem. Space step of the solution.
      */
  std::string name; /**< Protected string name. Name of the method. */
  double q; /**< Protected double q. A coeficient which value depends of way
      the equation is written, it may vary from method to method. */
public:
  // CONSTRUCTORS

  /**
  * Default constructor. Intialize a Method object
  * @see Method(Problem problem)
  */
  Method();

  /**
  * Alternate constructor. Initializes a Method with a given parabolic problem.
  * @see Method()
  */
  Method(Problem problem);

  // PUBLIC ACCESSOR METHODS

  /**
  * Normal public get method.
  * get the method name
  * @return string. Method name.
```

```
*/
std::string get_name();

/**
* Normal public get method.
* get the solution grid
* @return Matrix. Computed solution grid.
*/
Matrix get_solution();

/**
* Normal public get method.
* get the time step of the solution
* @return double. Solution time step.
*/
double get_deltat();

/**
* Normal public get method.
* get x values vector
* @return Vector. x values Vector.
*/
Vector get_xvalues();

/**
* Normal public get method.
* get the elapsed time value to compute a solution
* @return double. Elapsed time throughout the computation.
*/
double get_computational_time();

/**
* Normal public get method.
* get the second norm
* @return double. Second norm value.
*/
double get_two_norm();
```

```
  /**
  * Normal public method.
  * Keeps track of the time to compute a solution
  */
  void compute();

  void compute_norms(Matrix analytical_matrix);

  // PUBLIC MUTATOR METHODS

  /**
  * A pure virtual member.
  * compute the solution following the rules of a given method.
  */
  virtual void compute_solution() = 0;


};

#endif
```

methods/method.cpp

```
#include "method.h"

// CONSTRUCTORS
/*=
 *Default constructor
 */
Method::Method() {}

/*
 * Alternate constructor - creates a method with a given problem
 */
Method::Method(Problem problem) {
  this->problem = problem;
```

```
}
/*
 * public method - compute a solution keeping track of spent time
 */
void Method::compute() {
  clock_t begin = clock();
  compute_solution();
  clock_t end = clock();
  computational_time = double(end - begin) * 1000 / CLOCKS_PER_SEC;
}

/*
 * public method - compute norms of the error matrix;
 */
void Method::compute_norms(Matrix analytical_matrix) {
  Matrix method_matrix = get_solution();
  unsigned int rows = method_matrix.getNrows(), cols = method_matrix.getNcols();
  Matrix error_matrix(rows, cols);

  for (unsigned int i = 0; i < rows; i++) {
    for (unsigned int j = 0; j < cols; j++) {
      error_matrix[i][j] = analytical_matrix[i][j] - method_matrix[i][j];
    }
  }

  one_norm = error_matrix.one_norm();
  two_norm = error_matrix.two_norm();
  uniform_norm = error_matrix.uniform_norm();
}

// PUBLIC ACCESSOR METHODS

/*
 * public accessor method - get the method name
 */
std::string Method::get_name() {
  return name;
```

```
}

/*
 * public accessor method - get the two norm value
 */
double Method::get_two_norm() {
  return two_norm;
}

/*
 * public accessor method - get the solution grid
 */
Matrix Method::get_solution() {
  return (*problem.get_solution());
}

/*
 * public accessor method - get the solution time step
 */
double Method::get_deltat() {
  return problem.get_deltat();
}

/*
 * public accessor method - get x values vector
 */
Vector Method::get_xvalues() {
  return problem.get_xvalues();
}

/*
 * public accessor method - get computational time
 */
double Method::get_computational_time() {
  return computational_time;
}
```

methods/explicit/explicit.h

```
#ifndef EXPLICIT_H //include guard
#define EXPLICIT_H

#include "../method.h" // declare that the Method class exists (inheritance)

/**
*  An explicit method class that contains default methods that only explicit
    methods use
*  \n The implementation is derived from the Method class
*
* The Explicit class provides:
* \n-a basic constructor for creating an explicit method object.
* \n-a method to compute a solution following explicit methods rules
*/
class Explicit: public Method {
protected:
  // PROTECTED METHODS

  /**
  * A pure virtual member.
  * Build the solution of the next time step, using the previous time step and
     the next time step solutions
  * @param previous_step A vector containing the previous time step solution.
  * @param current_step A vector containing the current time step solution.
  * @return Vector. A vector representing the next time step solution.
  */
  virtual Vector build_iteration(Vector current_step, Vector previous_step) = 0;
public:
  // CONSTRUCTORS

  /**
  * Default constructor.
  */
  Explicit(Problem problem);
  // PUBLIC METHODS
```

```
  /**
  * Normal public method.
  * Calculates a solution for the given problem by populating the solution grid
      with the correct values.
  */
  void compute_solution();
};

#endif
```

methods/explicit/explicit.cpp

```
#include "explicit.h"
#include "forward_t_central_s.h" // method to use in the first iteration

// CONSTRUCTORS
/*=
 *Default constructor, method to solve a problem with explicit procedures
 */
Explicit::Explicit(Problem problem) : Method(problem) {
  double delta_t = problem.get_deltat(), delta_x = problem.get_deltax();
  q = (2 * delta_t * DIFUSIVITY) / pow(delta_x, 2);
}

// METHODS

/*
* public method - compute a solution using explicit procedures
*/
void Explicit::compute_solution() {
  FTCS ftcs(problem);
  Vector current_step, previous_step, next_step;
  unsigned int t_size = problem.get_tsize(), x_size = problem.get_xsize();
  double delta_t = problem.get_deltat(), time;
  current_step = next_step = Vector(x_size + 1);
  // iterate through the several time steps
```

```
  for (int i = 1; i <= t_size; i++) {
    // if is the first iteration then the previous step is known (initial
        conditions)
    // and the current-step may be obtained with the forward in time and
        central in space method, which only requires the previous step to
        calculate the current time step solution
    if (i == 1) {
      previous_step = problem.get_first_row();
      current_step = ftcs.build_iteration(Vector(0), previous_step);
    }

    // use the current and previous time steps to calculate the next time step
        solution
    next_step = build_iteration(current_step, previous_step);
    previous_step = current_step;
    current_step = next_step;
    time = delta_t * (double)i;
    // save solution if time step == 0.1, 0.2, 0.3, 0.4 or 0.5
    problem.set_time_step(current_step, time);
  }
}
```

methods/explicit/dufort_frankel.h

```
#ifndef DUFORT_FRANKEL_H //include guard
#define DUFORT_FRANKEL_H

#include "explicit.h" // declare that the Explicit class exists (inheritance)

/**
*  A DufortFrankel method class that contains an iteration builder.
*  \n This builder is used to calculate a solution using the Dufort-Frankel
    mathod.
*
* The DufortFrankel class provides:
* \n-a basic constructor for creating a DufortFrankel method object.
* \n-a method to compute a solution of the current iteration
```

```
*/
class DufortFrankel: public Explicit {
protected:
  // PROTECTED METHODS

  /**
  * Normal protected method.
  * Calculate a next time step solution requiring a previous time step and a
      current time step solution.
  * @param current_step A vector representing the current time step solution.
  * @param previous_step A vector representing the previous time step solution.
  * @return Vector. The computed solution.
  */
  Vector build_iteration(Vector current_step, Vector previous_step);
public:
  // CONSTRUCTORS

  /**
  * Default constructor.
  */
  DufortFrankel(Problem problem);
};

#endif
```

methods/explicit/dufort_frankel.cpp

```
#include "dufort_frankel.h"

// CONSTRUCTORS
/*=
 *Default constructor, method to compute an explicit solution.
 */
DufortFrankel::DufortFrankel(Problem problem)
: Explicit(problem) {
  name = DUFORT_FRANKEL;
}
```

```
/*
* Normal public method - compute a solution for the current time step using the
    Dufort-Frankel method.
*/
Vector DufortFrankel::build_iteration(Vector current_step, Vector
    previous_step) {
  unsigned int size = previous_step.getSize();
  Vector result(size);
  result[0] = result[size - 1] = SURFACE_TEMPERATURE;
  for (unsigned int i = 1; i < size - 1; i++) {
      result[i] = ((1.0 - q) * previous_step[i] + q * (current_step[i + 1] +
          current_step[i - 1])) / (1.0 + q);
  }
  return result;
}
```

methods/explicit/richardson.h

```
#ifndef RICHARDSON_H //include guard
#define RICHARDSON_H

#include "explicit.h" // declare that the Explicit class exists (inheritance)

/**
*  A Richardson method class that contains an iteration builder.
*  \n This builder is used to calculate a solution using the Richardson method.
*
* The Richardson class provides:
* \n-a basic constructor for creating a Richardson method object.
* \n-a method to compute a solution of the current iteration
*/
class Richardson: public Explicit {
protected:
  // PROTECTED METHODS

  /**
```

```
  * Normal protected method.
  * Calculate a next time step solution requiring a previous time step and a
      current time step solution.
  * @param current_step A vector representing the current time step solution.
  * @param previous_step A vector representing the previous time step solution.
  * @return Vector. The computed solution.
  */
  Vector build_iteration(Vector current_step, Vector previous_step);
public:
  // CONSTRUCTORS

  /**
  * Default constructor.
  */
  Richardson(Problem problem);
};

#endif
```

methods/explicit/richardson.cpp

```
#include "richardson.h"

// CONSTRUCTORS
/*=
 *Default constructor, method to compute an explicit solution.
 */
Richardson::Richardson(Problem problem)
: Explicit(problem) {
  name = RICHARDSON;
}

/*
* Normal public method - compute a solution for the current time step using the
    Richardson method.
*/
Vector Richardson::build_iteration(Vector current_step, Vector previous_step) {
```

```
  unsigned int size = previous_step.getSize();
  Vector result(size);
  result[0] = result[size - 1] = SURFACE_TEMPERATURE;
  for (unsigned int i = 1; i < size - 1; i++) {
      result[i] = previous_step[i] + q * (current_step[i + 1] - 2.0 *
          current_step[i] + current_step[i - 1]);
  }
  return result;
}
```

methods/explicit/forward_t_central_s.h

```
#ifndef FORWARD_TIME_CENTRAL_SPACE_H //include guard
#define FORWARD_TIME_CENTRAL_SPACE_H

#include "explicit.h" // declare that the Explicit class exists (inheritance)

/**
*  A FTCS method class that contains an iteration builder.
*  \n This builder is used to calculate the first iteration of explicit
    methods, since it only requires the previous step solution to do it.
*
* The FTCS class provides:
* \n-a basic constructor for creating a FTCS method object.
* \n-a method to compute the current iteration
*/
class FTCS: public Explicit {
public:
  // CONSTRUCTORS

  /**
  * Default constructor.
  */
  FTCS(Problem problem);

  // PUBLIC METHODS
```

```
  /**
  * Normal public method.
  * Calculate a solution requiring only the previous time step solution.
  * @param current_step A vector with size 0, it's not required in this method.
  * @param previous_step A vector representing the previous time step solution
  * @return Vector. The computed solution.
  */
  Vector build_iteration(Vector current_step, Vector previous_step);
};

#endif
```

methods/explicit/forward_t_central_s.cpp

```
#include "forward_t_central_s.h"

// CONSTRUCTORS
/*=
 *Default constructor, method to solve a the first iteration of explicit
    methods.
 */
FTCS::FTCS(Problem problem)
: Explicit(problem) {
  name = FORWARD_TIME_CENTRAL_SPACE;
}

/*
* Normal public method - compute the first iteration of explicit methods
*/
Vector FTCS::build_iteration(Vector current_step, Vector previous_step) {
  unsigned int size = previous_step.getSize();
  Vector result(size);
  result[0] = result[size - 1] = SURFACE_TEMPERATURE;
  for (unsigned int i = 1; i < size - 1; i++) {
    result[i] = previous_step[i] + q / 2.0 * (previous_step[i + 1] - 2.0 *
        previous_step[i] + previous_step[i - 1]);
  }
```

```
  return result;
}
```

methods/implicit/implicit.h

```
#ifndef IMPLICIT_H //include guard
#define IMPLICIT_H

#include "../method.h" // declare that the Method class exists (inheritance)


/**
*  An implicit method class that contains default methods that only implicit
    methods use
*  \n The implementation is derived from the Method class
*
* The Implicit class provides:
* \n-a basic constructor for creating an implicit method object.
* \n-a method to compute a solution following implicit methods rules
*/
class Implicit: public Method {
private:
  // PRIVATE METHODS

  /**
  * Normal private method.
  * Calculates the current time step with Tomas Algorithm.
  * Giving the A.x = r, in which A is a matrix, whereas b and r are vectors, it
     calculates the b vector, since A and b are known variables.
  * @see build_r(Vector previous_step)
  * @param r Vector calculated by the build_r method.
  * @param a Lower diagonal value of the tridiagonal matrix
  * @param b Center diagonal value of the tridiagonal matrix
  * @param c Upper diagonal value of the tridiagonal matrix
  * @return Vector. Vector that represents the current time step solution.
  */
  Vector thomas_algorithm(Vector r, double a, double b, double c);
```

```
protected:
  // PROTECTED METHODS

  /**
  * A pure virtual member.
  * Build the r vector in a linear system of A.x = r in which A is a matrix,
     whereas b and r are vectors.
  * \n This method is used to compute a solution using the thomas algorithm,
     which can be used in a triadiogonal matrix.
  * @param previous_step A vector containing the previous time step solution.
  * @return Vector. The r vector, which can be used in to calculate the current
     time step solution with Tomas Algorithm.
  */
  virtual Vector build_r(Vector previous_step) = 0;
public:
  // CONSTRUCTORS

  /**
  * Default constructor.
  */
  Implicit(Problem problem);

  // PUBLIC METHODS

  /**
  * Normal public method.
  * Calculates a solution for the given problem by populating the solution grid
     with the correct values.
  */
  void compute_solution();
};

#endif
```

methods/implicit/implicit.cpp

```
#include "implicit.h"
```

```
// CONSTRUCTORS
/*=
 *Default constructor, method to solve a problem with implicit procedures
 */
Implicit::Implicit(Problem problem) : Method(problem) {
  double delta_t = problem.get_deltat(), delta_x = problem.get_deltax();
  q = delta_t * DIFUSIVITY / pow(delta_x, 2);
}

// METHODS

/*
* public normal method - compute a solution using implicit procedures
*/
void Implicit::compute_solution() {
  Vector previous_step, current_step, r, t_values = problem.get_tvalues();
  unsigned int t_size = problem.get_tsize();
  double delta_t = problem.get_deltat(), time;

  // iterate through the several time steps
  for (int i = 1; i <= t_size; i++) {
    // if is the first iteration then the previous step is known (initial
       conditions)
    if (i == 1) { previous_step = problem.get_first_row(); }
    // build r vector
    r = build_r(previous_step);
    // use the r vector to calculate the current time step solution with the
       thomas algorithm
    current_step = thomas_algorithm(r, -q, (1.0 + 2.0 * q), -q);
    current_step.push_front_back(SURFACE_TEMPERATURE);
    previous_step = current_step;
    time = delta_t * (double)i;
    // save solution if time step == 0.1, 0.2, 0.3, 0.4 or 0.5
    problem.set_time_step(current_step, time);
  }
}
```

```
/*
* private normal method - thomas algorithm to compute a given time step solution
*/
Vector Implicit::thomas_algorithm(Vector r, double a, double b, double c) {
  unsigned int size = r.getSize();
  Vector p(size), y(size - 1), x(size);
  // building the y and p vectors (forward phase)
  for (unsigned int i = 0; i < size; i++) {
    if (i == 0) {
      y[i] = c / b;
      p[i] = r[i] / b;
    } else if (i == size - 1) {
      p[i] = (r[i] - a * p[i - 1]) / (b - a * y[i - 1]);
    } else {
      y[i] = c / (b - a * y[i - 1]);
      p[i] = (r[i] - a * p[i - 1]) / (b - a * y[i - 1]);
    }
  }
  // building the x vector in A.x = r linear equations system (backwards phase)
  x[size - 1] = p[size - 1];
  for (int i = size - 2; i >= 0; i--) {
    x[i] = p[i] - y[i] * x[i + 1];
  }
  return x;
}
```

methods/implicit/crank_nicolson.h

```
#ifndef CRANK_NICOLSON_H //include guard
#define CRANK_NICOLSON_H

#include "implicit.h" // declare that the Implicit class exists (inheritance)

/**
*  A CrankNicolson method class that contains a r vector builder.
```

```
*  \n This builder is used to calculate the r vector in A.x = r linear equation
    system.
*
* The CrankNicolson class provides:
* \n-a basic constructor for creating a CrankNicolson method object.
* \n-a method to compute the r vector.
*/
class CrankNicolson: public Implicit {
protected:
  // PROTECTED METHODS

  /**
  * Normal protected method.
  * get the number of rows
  * @param previous_step Vector representing the solution of the previous time
      step.
  * @return Vector. r vector to be used in A.x = r
  */
  Vector build_r(Vector previous_step);
public:
  // CONSTRUCTORS

  /**
  * Default constructor.
  */
  CrankNicolson(Problem problem);
};

#endif
```

methods/implicit/crank_nicolson.cpp

```
#include "crank_nicolson.h"

// CONSTRUCTORS
/*=
 *Default constructor, with a given problem.
```

```
 */
CrankNicolson::CrankNicolson(Problem problem) : Implicit(problem) {
  name = CRANK_NICHOLSON;
  // q = delta_t * DIFUSIVITY / (pow(delta_x, 2) * 2) so we multiply it by 0.5
  q *= 0.5;
}
// PROTECTED METHODS

/*
* protected method - build the r vector in A.x = r
*/
Vector CrankNicolson::build_r(Vector previous_step) {
  unsigned int size = previous_step.getSize() - 2, j = 0;
  Vector r(size);
  for (unsigned int i = 0; i < size; i++) {
    j = i + 1;
    r[i] = (i == 0 || i == size - 1) ? previous_step[j] + q *
        (SURFACE_TEMPERATURE + previous_step[j - 1] - 2.0 * previous_step[j] +
        previous_step[j + 1])
    : previous_step[j] + q * (previous_step[j - 1] - 2.0 * previous_step[j] +
        previous_step[j + 1]);
  }
  return r;
}
```

methods/implicit/laasonen.h

```
#ifndef LAASONEN_H //include guard
#define LAASONEN_H

#include "implicit.h" // declare that the Implicit class exists (inheritance)

/**
*  A Laasonen method class that contains a r vector builder.
*  \n This builder is used is used to calculate the r vector in A.x = r linear
    equation system.
*
```

```
* The Laasonen class provides:
* \n-a basic constructor for creating a Laasonen method object.
* \n-a method to compute the r vector.
*/
class Laasonen: public Implicit {
protected:
  // PROTECTED METHODS

  /**
  * Normal protected method.
  * get the number of rows
  * @param previous_step Vector representing the solution of the previous time
      step.
  * @return Vector. r vector to be used in A.x = r
  */
  Vector build_r(Vector previous_step);
public:
  // CONSTRUCTORS

  /**
  * Default constructor.
  */
  Laasonen(Problem problem);
};

#endif
```

methods/implicit/laasonen.cpp

```
#include "laasonen.h"

// CONSTRUCTORS
/*=
 *Default constructor, with a given problem.
 */
Laasonen::Laasonen(Problem problem) : Implicit(problem) {
  name = LAASONEN;
```

```
}

// PROTECTED METHODS

/*
* protected method - build the r vector in A.x = r
*/
Vector Laasonen::build_r(Vector previous_step) {
  unsigned int size = previous_step.getSize() - 2, j = 0;
  Vector r(size);
  for (unsigned int i = 0; i < size; i++) {
    j = i + 1;
    r[i] = (i == 0 || i == size - 1) ? q * SURFACE_TEMPERATURE +
        previous_step[j] : previous_step[j];
  }
  return r;
}
```

io/iomanager.h

```
#ifndef IO_MANAGER_H //Include guard
#define IO_MANAGER_H

#include "../libs/gnuplot-iostream.h" // lib to be able to use gnuplot from c++
#include "../methods/method.h" // provides knowledge about method objects
    structure
#include <vector>

/**
*  An input/output manager class to handle plot exportations and future
    implementations of input handling
*
* The IOManager class provides:
* \n-plot method which compares the analytical solution with a set of given
    methods, ploting them with a custom configuration using gnuplot
*/
class IOManager {
```

```
private:
  std::string output_path; /**< Private string output_path. Contains the ouput
     directory path name. */
  std::vector<double> laasonen_times;  /**< Private Vector laasonen_times.
     Contains the computation time of each laasonen solution, with a different
     time step. */
  std::vector<double> default_deltat_times; /**< Private Vector
     default_deltat_times. Contains the computation time of each method
     solution, with a time step of 0.01. */

  // PRIVATE PLOT METHODS

  /**
  * Method to create ouput folder if the folder does not exist
  * @return bool. true if successfull, false if not
  */
  bool create_output_dir();

  /**
  * Exports a plot chart that compares the analytical solution to any other
     solution using gnuplot
  * @param string output_name File name to be exported
  * @param Method* analytical The analytical solution
  * @param Method* method Any method solution
  */
  void plot_solutions(std::string output_name, Method * analytical, Method *
     method);

  /**
  * Exports a plot with Laasonen delta t variation computational times
  */
  void plot_laasonen_times();


  /**
  * Exports a plot with four methods computational times
  */
```

```
  void plot_default_deltat_times();

  /**
  * Exports a plot that compares the norms of each solution
  * @param string output_name File name to be exported
  * @param vector<Method*> vector of methods to plot the second norm
  */
  void error_tables(std::string output_name, std::vector<Method*> method);

  // AUX METHODS

  /**
  * Converts a double to a string with a precison of 2 decimal places
  * @param double value Number to be converted
  * @param int precision Precision to have
  * @return string. String containing the converted number
  */
  std::string double_to_string(int precision, double value);
public:
  // CONSTRUCTORS

  /**
  * Default constructor. Initialize an IOManager object.
  */
  IOManager();

  // PUBLIC OUTPUTS METHODS

  /**
  * Exports outputs regarding plots images and error tables for each computed
      solution, comparing them to the analytical solution
  * @param Method* analytical The analytical solution
  * @param vector<Method*> methods Vector containing the solutions
  */
  void export_outputs(Method * analytical, std::vector<Method*> methods);
};
```

```
#endif
```

io/iomanager.cpp

```
#include "iomanager.h"

// CONSTRUCTORS
/*=
 * Default constructor
 */

IOManager::IOManager() {
  output_path = OUTPUT_PATH;
}

// PLOT METHODS

/*
* private output method - creates output folder
*/

bool IOManager::create_output_dir() {
  char answer = '.';

  // if plot folder exists ask if the user wants to replace the previous results
  if(boost::filesystem::exists(output_path)) {
    while (answer != 'y' && answer != 'Y' && answer != 'n' && answer != 'N') {
      std::cout << "Output folder already exists, this program might overwrite
          some files, procceed? [Y:N] - ";
      answer = std::getchar();
      std::cout << std::endl;
    }
    return answer == 'y' || answer == 'Y' ? true : false;
  } else {
    if(boost::filesystem::create_directory(output_path)) {
      std::cout << "Output folder was created!" << std::endl;
      return true;
```

```
    } else {
      std::cout << "Output folder couldn't be created! Not exporting outputs."
          << std::endl;
      return false;
    }
  }
}

/*
* public output method - iterates through all the solutions in order to export
    them in varied formats
*/

void IOManager::export_outputs(Method * analytical, std::vector<Method*>
    methods) {
  if (!create_output_dir()) return;
  std::cout << "Exporting outputs to " <<
      boost::filesystem::canonical(output_path, ".") << std::endl;
  std::string name, output_name, deltat_string;
  for (unsigned int index = 0; index < methods.size(); index++) {
    name = (*methods[index]).get_name();
    deltat_string = name == LAASONEN ? double_to_string(3,
        methods[index]->get_deltat()) : double_to_string(2,
        methods[index]->get_deltat());
    output_name = output_path + '/' + name;
    if (name == LAASONEN) {
      output_name += "dt=" + deltat_string;
      laasonen_times.push_back(methods[index]->get_computational_time());
    }
    if (! (name == LAASONEN && methods[index]->get_deltat() != 0.01) && name !=
        RICHARDSON) {
      default_deltat_times.push_back(methods[index]->get_computational_time());
    }
    std::cout << "Exporting " << name << " method outputs... ";
    plot_solutions(output_name, analytical, methods[index]);
    std::cout << "Finished!" << std::endl;
  }
```

```
  std::vector<Method*> error_vector(methods.begin() + 1, methods.begin() + 4);
  error_tables(output_name, error_vector);
  plot_default_deltat_times();
  plot_laasonen_times();
}

/*
* private plot method - Exports a plot chart which compares the analytical
    solution with a given solution
*/

void IOManager::plot_solutions(std::string output_name, Method * analytical,
    Method * method) {
  // Object to export plots
  Gnuplot gp;

  // methods solutions
  Matrix analytical_matrix = analytical->get_solution(), method_matrix =
      method->get_solution();
  unsigned int rows = method_matrix.getNrows();
  unsigned int cols = method_matrix.getNcols();
  double time;
  std::string time_str, name = method->get_name();

  // defining gnuplot configuration with the correct syntax
  gp << "set key on box; set tics scale 0; set border 3; set ylabel
      \"Temperature [F]\";set xlabel \"x [ft]\"; set yrange [90:310]; set term
      png; set xtics (\"0\" 0, \"0.5\"" << cols / 2 << ", \"1\"" << cols - 1 <<
      ")\n";
  for (unsigned int index = 1; index < rows; index++) {
    time = (double)index / 10.0;
    time_str = double_to_string(1, time);

    gp << "set output \"" << output_name << "t=" << time_str;
    gp << ".png\";\n";
    gp << "plot" << gp.file1d(analytical_matrix[index]) << "with lines title
        \"Analytical\" lw 2 lt rgb \"red\","
```

```
      << gp.file1d(method_matrix[index]) << "with points title \"" << name <<
         "\" pt 17 ps 1 lw 1" << std::endl;
  }
}

/*
* private plot method - Exports a plot chart with laasonen solutions
    computational times
*/

void IOManager::plot_laasonen_times() {
  // Object to export plots
  Gnuplot gp;

  gp << "set tics scale 0; set border 3; set style line 1 lc rgb '#0060ad' lt 1
      lw 2 pt 7 pi -1 ps 1.5; set clip two; set ylabel \"Computational Time
      [ms]\";set xlabel \"Delta t [h]\"; set term png; set xtics (\"0.01\" 0,
      \"0.025\" 1, \"0.05\" 2, \"0.1\" 3)\n";
  gp << "set output \"" << output_path << "/laasonen_times.png\";\n";
  gp << "plot" << gp.file1d(laasonen_times) << " notitle with linespoint ls 1"
      << std::endl;
}

/*
* private plot method - Exports a plot chart with solutions computational times
*/

void IOManager::plot_default_deltat_times() {
  // Object to export plots
  Gnuplot gp;

  gp << "set tics scale 0; set border 3; set style line 1 lc rgb '#0060ad' lt 1
      lw 2 pt 7 pi -1 ps 1.5; set clip two; set ylabel \"Computational Time
      [ms]\";set xlabel \"Delta t [h]\"; set term png; set xtics
      (\"Dufort-Frankel\" 0, \"Crank-Nicholson\" 1, \"Laasonen\" 2)\n";
  gp << "set output \"" << output_path << "/default_deltat_times.png\";\n";
```

```
  gp << "plot" << gp.file1d(default_deltat_times) << " notitle with linespoint
      ls 1" << std::endl;
}

/*
* private table method - Exports an error table to a .lsx file which compares
    the analytical solution with a given solution
*/
void IOManager::error_tables(std::string output_name, std::vector<Method*>
    methods) {

  // Object to export plots
  Gnuplot gp;
  std::vector<double> norms;

  for (int i = 0; i < methods.size(); i++) {
    norms.push_back(methods[i]->get_two_norm());
  }
  std::swap(norms[1],norms[2]);


  gp << "set tics scale 0; set border 3; set style line 1 lc rgb '#FFA500' lt 1
      lw 2 pt 7 pi -1 ps 1.5; set clip two; set ylabel \"2nd Norm\";set xlabel
      \"\"; set term png; set xtics (\"Dufort-Frankel\" 0, \"Laasonen\" 1,
      \"Crank Nicholson\" 2)\n";
  gp << "set output \"" << output_path << "/norms.png\";\n";
  gp << "plot" << gp.file1d(norms) << " notitle with linespoint ls 1" <<
      std::endl;
}

// AUX METHODS

/*
* aux method - Converts a double into a string
*/

std::string IOManager::double_to_string(int precision, double value) {
```

```
  std::stringstream stream;
  stream << std::fixed << std::setprecision(precision) << value;
  return stream.str();
}
```

grid/vector.h

```
#ifndef VECTOR_H //Include guard
#define 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
#include <cmath> // use of existing mathematical methods
#include <float.h> // provides double maximum value
#include <algorithm> // to use a method which finds a given value in a vector


/**
*  A vector class for data storage of a 1D array of doubles
*  \n The implementation is derived from the standard container vector
    std::vector
*  \n We use private inheritance to base our vector upon the library version
    whilst
*  \nallowing usto expose only those base class functions we wish to use - in
    this
*  \ncase the array access operator []
*
* The Vector class provides:
* \n-basic constructors for creating vector obcjet from other vector object,
or by creating empty vector of a given size,
* \n-input and oput operation via >> and << operators using keyboard or file
* \n-basic operations like access via [] operator, assignment and comparision
*/
class Vector : private std::vector<double> {
  typedef std::vector<double> vec;
```

```
public:
  using vec::operator[]; // elevates the array access operator inherited from
      std::vector
             // to public access in Vector
  // CONSTRUCTORS
  /**
  * Default constructor. Intialize an empty Vector object
  * @see Vector(int Num)
  * @see Vector(const Vector& v)
  */
    Vector(); // default constructor


  /**
  * Explicit alterative constructor takes an intiger.
  * it is explicit since implicit type conversion int -> vector doesn't make
      sense
  * Intialize Vector object of size Num
  * @see Vector()
  * @see Vector(const Vector& v)
  * @exception invalid_argument ("vector size negative")
  */
    explicit Vector(int Num /**< int. Size of a vector */);

  /**
  * Copy constructor takes an Vector object reference.
  * Intialize Vector object with another Vector object
  * @see Vector()
  * @see Vector(int Num)
  */
    Vector(const Vector& v);

    /**
  * Copy constructor takes an vector<double> object reference.
  * Intialize Vector object with an vector<double> object
  * @see Vector()
  * @see Vector(int Num)
```

```
* @see Vector(const Vector& v)
*/
  Vector(std::vector<double> vec);

// OVERLOADED OPERATORS
/**
* Overloaded assignment operator
* @see operator==(const Vector& v)const
* @param v Vector to assign from
* @return the object on the left of the assignment
*/
  Vector& operator=(const Vector& v /**< Vecto&. Vector to assign from */);

/**
* Overloaded comparison operator
* returns true if vectors are the same within a tolerance (1.e-07)
* @see operator=(const Vector& v)
* @see operator[](int i)
* @see operator[](int i)const
* @return bool. true or false
* @exception invalid_argument ("incompatible vector sizes\n")
*/
bool operator==(const Vector& v /**< Vector&. vector to compare */
  ) const;


// ACCESSOR METHODS
/** Normal get method that returns integer, the size of the vector
* @return int. the size of the vector
*/
int getSize() const;

//AUX METHODS
/**
* Method to find the value index in a vector
* @param value Value to find
* @return int. -1 if value was not found or the value index otherwise
```

```
*/
int find(double value);


/**
* Method to push a value to the first and last position of a Vector
* @param value Value to insert
*/
void push_front_back(double value);

/**
* Method to push a value to the last position of a Vector
* @param value Value to be pushed
*/
void push(double value);

// THREE NORMS
/**
* Normal public method that returns a double.
* It returns L1 norm of vector
* @see two_norm()const
* @see uniform_norm()const
* @return double. vectors L1 norm
*/
double one_norm() const;

/**
* Normal public method that returns a double.
* It returns L2 norm of vector
* @see one_norm()const
* @see uniform_norm()const
* @return double. vectors L2 norm
*/
double two_norm() const;

/**
* Normal public method that returns a double.
```

```
* It returns L_max norm of vector
* @see one_norm()const
* @see two_norm()const
* @exception out_of_range ("vector access error")
* vector has zero size
* @return double. vectors Lmax norm
*/
double uniform_norm() const;


// KEYBOARD/SCREEN INPUT AND OUTPUT
/**
* Overloaded istream >> operator. Keyboard input
* if vector has size user will be asked to input only vector values
* if vector was not initialized user can choose vector size and input it
    values
* @see operator>>(std::ifstream& ifs, Vector& v)
* @see operator<<(std::ostream& os, const Vector& v)
* @see operator<<(std::ofstream& ofs, const Vector& v)
* @return std::istream&. the input stream object is
* @exception std::invalid_argument ("read error - negative vector size");
*/
  friend std::istream& operator >> (std::istream& is, /**< keyboard input
      straem. For user input */
  Vector& v /**< Vector&. vector to write to */
  );


/**
* Overloaded ifstream << operator. Display output.
* @see operator>>(std::istream& is, Vector& v)
* @see operator>>(std::ifstream& ifs, Vector& v)
* @see operator<<(std::ofstream& ofs, const Vector& v)
* @return std::ostream&. the output stream object os
*/
friend std::ostream& operator<<(std::ostream& os, /**< output file stream */
  const Vector& v /**< vector to read from */
```

```
    );

  /**
  * Overloaded ifstream >> operator. File input
  * the file output operator is compatible with file input operator,
  * ie. everything written can be read later.
  * @see operator>>(std::istream& is, Vector& v)
  * @see operator<<(std::ostream& os, const Vector& v)
  * @see operator<<(std::ofstream& ofs, const Vector& v)
  * @return ifstream&. the input ifstream object ifs
  * @exception std::invalid_argument ("file read error - negative vector size");
  */
  friend std::ifstream& operator >> (std::ifstream& ifs, /**< input file
      straem. With opened matrix file */
    Vector& v /**< Vector&. vector to write to */
    );


  /**
  * Overloaded ofstream << operator. File output.
  * the file output operator is compatible with file input operator,
  * ie. everything written can be read later.
  * @see operator>>(std::istream& is, Vector& v)
  * @see operator>>(std::ifstream& ifs, Vector& v)
  * @see operator<<(std::ostream& os, const Vector& v)
  * @return std::ofstream&. the output ofstream object ofs
  */
  friend std::ofstream& operator<<(std::ofstream& ofs, /**< outputfile stream.
      With opened file */
    const Vector& v /**< Vector&. vector to read from */
    );
};

#endif
```

grid/vector.cpp

```
#include "vector.h"

// 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];
}

/*
* Copy constructor from different type
*/
Vector::Vector(std::vector<double> vec) : std::vector<double>() {
```

```
  (*this).resize(vec.size());
    // copy the data members (if vector is empty then num==0)
  std::size_t i;
    for (i=0; i<vec.size(); i++) (*this)[i]=vec[i];
}

/*
* accessor method - get the size
*/
int Vector::getSize() const
{
  return size();
}

/*
* aux method - pushes a value to the begining and ending of a vector
*/
void Vector::push_front_back(double value) {
  this->insert(this->begin(), value);
  this->push_back(value);
}

/*
* aux method - pushes a value to the vector
*/
void Vector::push(double value) {
  this->push_back(value);
}

/*
* aux method - finds a given value
*/
int Vector::find(double value) {
  std::vector<double>::iterator it = std::find((*this).begin(), (*this).end(),
      value);
  if (it != (*this).end()) {
    return std::distance((*this).begin(), it);
```

```
  } else {
    return -1;
  }
}

// OVERLOADED OPERATORS
/*
* Operator= - assignment
*/
Vector& Vector::operator=(const Vector& copy)
{
  if (this != &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 vector
      sizes\n");
  std::size_t i;
  for (i = 0; i<size(); i++)  if (fabs((*this)[i] - v[i]) > 1.e-07) { return
      false; }
  return true;
}

// NORMS
/*
* 1 norm
```

```
*/
double Vector::one_norm() const {
  int n = size();
  double result = 0.0;
  for (unsigned int index = 0; index < n; index++) {
    result += std::abs((*this)[index]);
  }
  return result;
}

/*
* 2 norm
*/
double Vector::two_norm() const
{
  int n = size();
  double result = 0.0;
  for (unsigned int index = 0; index < n; index++) {
    result += pow((*this)[index], 2);
  }
  return sqrt(result);
}

/*
* uniform (infinity) norm
*/
double Vector::uniform_norm() const
{
  int n = size();
  double result = -DBL_MAX;
  for (unsigned int index = 0; index < n; index++) {
    double value = std::abs((*this)[index]);
    if (value > result) result = value;
  }
  return result;
}
```

```
// INPUT AND OUTPUT
/*
* keyboard input , user friendly
*/
std::istream& operator>>(std::istream& is, Vector& v)
{
  if (!v.size()) {
    int n;

    std::cout << "input the size for the vector" << std::endl;
    is >> n;
    //check input sanity
    if(n < 0) throw std::invalid_argument("read error - negative vector size");

    // prepare the vector to hold n elements
    v = Vector(n);
  }
  // input the elements
  std::cout << "input "<< v.size() <<" vector elements" << std::endl;
  std::size_t i;
  for (i=0; i<v.size(); i++) is >> v[i];

    // return the stream object
    return is;
}

/*
* file input - raw data, compatible with file writing operator
*/
std::ifstream& operator>>(std::ifstream& ifs, Vector& v)
{
    int n;

    // read size from the file
    ifs >> n;
    //check input sanity
```

```
    if(n < 0) throw std::invalid_argument("file read error - negative vector
        size");

    // prepare the vector to hold n elements
    v = Vector(n);

    // input the elements
    for (int i=0; i<n; i++) ifs >> v[i];

    // return the stream object
    return ifs;
}

/*
* screen output, user friendly
*/
std::ostream& operator<<(std::ostream& os, const Vector& v)
{
    if (v.size() > 0) {
    std::size_t i;
        for (i=0; i<v.size(); i++) os << v[i] << " ";
        os << std::endl;
    }
    else
    {
        os << "Vector is empty." << std::endl;
    }
    return os;
}

/*
* file output - raw data, comaptible with file reading operator
*/
std::ofstream& operator<<(std::ofstream& ofs, const Vector& v)
{
    //put vector size in first line (even if it is zero)
    ofs << v.size() << std::endl;
```

```
    //put data in second line (if size==zero nothing will be put)
  std::size_t i;
    for (i=0; i<v.size(); i++) ofs << v[i] << " ";
    ofs << std::endl;
    return ofs;
}
```

grid/matrix.h

```
#ifndef MATRIX_H //include guard
#define MATRIX_H

#include <iostream> //generic IO
#include <fstream> //file IO
#include <stdexcept> //provides exceptions
#include "vector.h" //we use Vector in Matrix code



/**
*  A matrix class for data storage of a 2D array of doubles
*  \n The implementation is derived from the standard container vector
    std::vector
*  \n We use private inheritance to base our vector upon the library version
    whilst
*  \nallowing usto expose only those base class functions we wish to use - in
    this
*  \ncase the array access operator []
*
* The Matrix class provides:
* \n-basic constructors for creating a matrix object from other matrix object,
* \nor by creating empty matrix of a given size,
* \n-input and oput operation via >> and << operators using keyboard or file
* \n-basic operations like access via [] operator, assignment and comparision
*/
class Matrix : private std::vector<std::vector<double> > {
  typedef std::vector<std::vector<double> > vec;
```

```
public:
  using vec::operator[]; // make the array access operator public within Matrix

  // CONSTRUCTORS

  /**
  * Default constructor. Intialize an empty Matrix object
  * @see Matrix(int Nrows, int Ncols)
  * @see Matrix(const Matrix& m)
  */
    Matrix();

  /**
  * Alternate constructor.
  * build a matrix Nrows by Ncols
  * @see Matrix()
  * @see Matrix(const Matrix& m)
  * @exception invalid_argument ("matrix size negative or zero")
  */
    Matrix(int Nrows /**< int. number of rows in matrix */, int Ncols /**< int.
        number of columns in matrix */);

  /**
  * Copy constructor.
  * build a matrix from another matrix
  * @see Matrix()
  * @see Matrix(int Nrows, int Ncols)
  */
  Matrix(const Matrix& m /**< Matrix&. matrix to copy from */);

  // ACCESSOR METHODS

  /**
  * Normal public get method.
  * get the number of rows
  * @see int getNcols()const
  * @return int. number of rows in matrix
```

```
*/
int getNrows() const; // get the number of rows

/**
* Normal public get method.
* get the number of columns
* @see int getNrows()const
* @return int. number of columns in matrix
*/
int getNcols() const; // get the number of cols

// MUTATOR METHODS

/**
* Normal public set method.
* replace a row with a given vector
* @param index Index of row to mutate
* @param v New vector
* @exception out_of_range ("index out of range.\n")
* @exception out_of_range ("vector size is different from matrix columns
    number.\n")
*/
void set_row(int index, Vector v);

// OVERLOADED OPERATOR

/**
* Overloaded assignment operator
* @see operator==(const Matrix& m)const
* @return Matrix&. the matrix on the left of the assignment
*/
Matrix& operator=(const Matrix& m /**< Matrix&. Matrix to assign from */); //
    overloaded assignment operator


/**
* Overloaded comparison operator
```

```
* returns true or false depending on whether the matrices are the same or not
* @see operator=(const Matrix& m)
* @return bool. true or false
*/
bool operator==(const Matrix& m /**< Matrix&. Matrix to compare to */
  ) const; // overloaded comparison operator


// NORMS
/**
* Normal public method that returns a double.
* It returns L1 norm of matrix
* @see two_norm()const
* @see uniform_norm()const
* @return double. matrix L1 norm
*/
double one_norm() const;

/**
* Normal public method that returns a double.
* It returns L2 norm of matrix
* @see one_norm()const
* @see uniform_norm()const
* @return double. matrix L2 norm
*/
double two_norm() const;

/**
* Normal public method that returns a double.
* It returns L_max norm of matrix
* @see one_norm()const
* @see two_norm()const
* @return double. matrix L_max norm
*/
double uniform_norm() const;
```

```
// MULTIPLICATION, COMPARISON METHODS and TRANSPOSE METHODS

/**
* Overloaded *operator that returns a Matrix.
* It Performs matrix by matrix multiplication.
* @see operator*(const Matrix & a) const
* @exception out_of_range ("Matrix access error")
* One or more of the matrix have a zero size
* @exception std::out_of_range ("uncompatible matrix sizes")
* Number of columns in first matrix do not match number of columns in second
    matrix
* @return Matrix. matrix-matrix product
*/
//
Matrix operator*(const Matrix & a /**< Matrix. matrix to multiply by */
  ) const;

/**
* Overloaded *operator that returns a Vector.
* It Performs matrix by vector multiplication.
* @see operator*(const Matrix & a)const
* @exception std::out_of_range ("Matrix access error")
* matrix has a zero size
* @exception std::out_of_range ("Vector access error")
* vector has a zero size
* @exception std::out_of_range ("uncompatible matrix-vector sizes")
* Number of columns in matrix do not match the vector size
* @return Vector. matrix-vector product
*/
//
Vector operator*(const Vector & v /**< Vector. Vector to multiply by */
  ) const;


/**
* public method that returns the transpose of the matrix.
* It returns the transpose of matrix
```

```
* @return Matrix. matrix transpose
*/
Matrix transpose() const;

Matrix mult(const Matrix& a) const;

/**
* Overloaded istream >> operator.
* Keyboard input
* if matrix has size user will be asked to input only matrix values
* if matrix was not initialized user can choose matrix size and input it
    values
* @see operator<<(std::ofstream& ofs, const Matrix& m)
* @see operator>>(std::istream& is, Matrix& m)
* @see operator<<(std::ostream& os, const Matrix& m)
* @exception std::invalid_argument ("read error - negative matrix size");
* @return std::istream&. The istream object
*/
 friend std::istream& operator >> (std::istream& is, /**< Keyboard input
     stream */
  Matrix& m /**< Matrix to write into */
  );// keyboard input


/**
* Overloaded ostream << operator.
* Display output
* if matrix has size user will be asked to input only matrix values
* if matrix was not initialized user can choose matrix size and input it
    values
* @see operator>>(std::ifstream& ifs, Matrix& m)
* @see operator>>(std::istream& is, Matrix& m)
* @see operator<<(std::ostream& os, const Matrix& m)
* @return std::ostream&. The ostream object
*/
friend std::ostream& operator<<(std::ostream& os, /**< Display output stream
    */
```

```
    const Matrix& m /**< Matrix to read from*/
    );// screen output


  /**
  * Overloaded ifstream >> operator. File input
  * the file output operator is compatible with file input operator,
  * ie. everything written can be read later.
  * @see operator>>(std::ifstream& ifs, Matrix& m)
  * @see operator<<(std::ofstream& ofs, const Matrix& m)
  * @see operator<<(std::ostream& os, const Matrix& m)
  * @return std::ifstream&. The ifstream object
  */
  friend std::ifstream& operator >> (std::ifstream& ifs, /**< Input file stream
      with opened matrix file */
    Matrix& m /**< Matrix to write into */
    );// file input

  /**
  * Overloaded ofstream << operator. File output
  * the file output operator is compatible with file input operator,
  * ie. everything written can be read later.
  * @see operator>>(std::ifstream& ifs, Matrix& m)
  * @see operator<<(std::ofstream& ofs, const Matrix& m)
  * @see operator>>(std::istream& is, Matrix& m)
  * @exception std::invalid_argument ("file read error - negative matrix size");
  * @return std::ofstream&. The ofstream object
  */
  friend std::ofstream& operator<<(std::ofstream& ofs,
    const Matrix& m /**< Matrix to read from*/
    );// file output
};

#endif
```

grid/matrix.cpp

```
#include "matrix.h"

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

/*
 * Alternate constructor - creates a matrix with the given values
 */
Matrix::Matrix(int Nrows, int Ncols) : std::vector<std::vector<double> >()
{
    //check input
    if(Nrows < 0 || Ncols < 0) throw std::invalid_argument("matrix size
        negative");

  // set the size for the rows
  (*this).resize(Nrows);
  // set the size for the columns
  for (int i = 0; i < Nrows; i++) (*this)[i].resize(Ncols);

  // initialise the matrix to contain zero
  for (int i = 0; i < Nrows; i++)
    for (int j = 0; j < Ncols; j++) (*this)[i][j] = 0;
}


/*
 * Copy constructor
 */
Matrix::Matrix(const Matrix& m) : std::vector<std::vector<double> >()
{
  // set the size of the rows
  (*this).resize(m.size());
  // set the size of the columns
```

```
  std::size_t i;
  for (i = 0; i < m.size(); i++) (*this)[i].resize(m[0].size());

  // copy the elements
  for (int i = 0; i < m.getNrows(); i++)
    for (int j = 0; j < m.getNcols(); j++)
      (*this)[i][j] = m[i][j];
}

// ACCESSOR METHODS
/*
* accessor method - get the number of rows
*/
int Matrix :: getNrows() const
{
  return size();
}

/*
* accessor method - get the number of columns
*/
int Matrix::getNcols() const
{
  return (*this)[0].size();
}

// MUTATORS METHODS

/*
* mutator method - set new values to a row
*/

void Matrix::set_row(int index, Vector v) {
  int vector_size = v.getSize();

  //catches invalid arguments
```

```
  if (index < 0 || index >= vector_size) throw std::out_of_range("index out of
      range.");
  if (vector_size != (*this).getNcols()) throw std::out_of_range("vector size
      is different from matrix columns number.");

  for (unsigned int i = 0; i < vector_size; i++) {
    (*this)[index][i] = v[i];
  }
}

// NORMS
/*
 * 1 norm
 */
double Matrix::one_norm() const
{
  int nrows = getNrows();
  int ncols = getNcols();
  Matrix transpose = this->transpose();
  Vector * sizes = new Vector(nrows);
  for (unsigned int index = 0; index < nrows; index++) {
    Vector * vector = (Vector*)&transpose[index];
    (*sizes)[index] = vector->one_norm();
  }
  return sizes->uniform_norm();
}

/*
 * 2 norm
 */
double Matrix::two_norm() const
{
  int nrows = getNrows();
  int ncols = getNcols();

  double result = 0.0;
  for (unsigned int i = 0; i < nrows; i++) {
```

```
    for (unsigned int j = 0; j < ncols; j++) {
      result += pow((*this)[i][j], 2);
    }
  }
  return sqrt(result);
}

/*
 * uniform (infinity) norm
 */
double Matrix::uniform_norm() const
{
  int nrows = getNrows();
  int ncols = getNcols();
  Vector * sizes = new Vector(nrows);
  for (unsigned int index = 0; index < nrows; index++) {
    Vector * vector = (Vector*)&((*this)[index]);
    (*sizes)[index] = vector->one_norm();
  }
  return sizes->uniform_norm();
}

// OVERLOADED OPERATORS
/*
* Operator= - assignment
*/
Matrix& Matrix::operator=(const Matrix& m)
{ if (this != &m) {
    (*this).resize(m.size());
    std::size_t i;
    std::size_t j;
    for (i = 0; i < m.size(); i++) (*this)[i].resize(m[0].size());

    for (i = 0; i<m.size(); i++)
      for (j = 0; j<m[0].size(); j++)
        (*this)[i][j] = m[i][j];
  }
```

```
  return *this;
}

/*
* Operator* multiplication of a matrix by a matrix
*/
Matrix Matrix::operator*(const Matrix& a) const {

  int nrows = getNrows();
  int ncols = getNcols();

  // catch invalid matrices
  if (nrows <= 0 || ncols <= 0) { throw std::out_of_range("Matrix access
      error"); }
  if (a.getNrows() <= 0 || a.getNcols() <= 0) { throw std::out_of_range("Matrix
      access error"); }

  //if the matrix sizes do not match
  if (ncols != a.getNrows()) throw std::out_of_range("matrix sizes do not
      match");

  //  matrix to store the result
  Matrix mmult = Matrix(getNrows(), a.getNcols());

  //matrix multiplication
  for (int i = 0;i<getNrows();i++) {
    for (int j = 0;j<a.getNcols();j++) {
      for (int k = 0;k<getNcols();k++) {
        mmult[i][j] += ((*this)[i][k] * a[k][j]);
      }
    }
  }
  return mmult;
}


/*
```

```
* Operator* multiplication of a matrix by a vector
*/
Vector Matrix::operator*(const Vector& v) const {
  int nrows = getNrows();
  int ncols = getNcols();

  // catch invalid matrix, vector
  if (nrows <= 0 || ncols <=0 ) { throw std::out_of_range("Matrix access
      error"); }
  if (v.getSize() <= 0) { throw std::out_of_range("Vector access error"); }

  //if the matrix sizes do not match
  if (ncols != v.getSize()) throw std::out_of_range("matrix sizes do not
      match");

  // matrix to store the multiplication
  Vector res(nrows);

  // perform the multiplication
  for (int i = 0;i<nrows;i++)
    for (int j = 0;j<ncols;j++) res[i] += ((*this)[i][j] * v[j]);

  // return the result
  return res;
}


/*
 * Operator== comparison function, returns true if the given matrices are the
     same
 */
bool Matrix::operator==(const Matrix& a) const {
  int nrows = getNrows();
  int ncols = getNcols();

  //if the sizes do not match return false*
  if ( (nrows != a.getNrows()) || (ncols != a.getNcols()) ) return false;
```

```
  //compare all of the elements
  for (int i=0;i<nrows;i++) {
    for (int j=0;j<ncols;j++) {
      if (fabs((*this)[i][j] - a[i][j]) > 1.e-07) { return false; }
    }
  }

  return true;
}


// OTHER METHODS
/*
 * Transpose of the matrix
 */
Matrix Matrix::transpose() const
{
  int nrows = getNrows();
  int ncols = getNcols();

  // matrix to stoore the transpose
  Matrix temp(ncols, nrows);

  for (int i=0; i < ncols; i++)
    for (int j=0; j < nrows; j++)
      temp[i][j] = (*this)[j][i];

  return temp;
}

Matrix Matrix::mult(const Matrix& m) const
{
  return (*this) * m;
}
```

```
// INPUT AND OUTPUT FUNCTIONS
/*
 * keyboard input , user friendly
 */
std::istream& operator>>(std::istream& is, Matrix& m) {

  int nrows, ncols;

  // test to see whether the matrix m is empty
  if (!m.getNrows()) {
    std::cout << "input the number of rows for the matrix" << std::endl;
    is >> nrows;
    std::cout << "input the number of cols for the matrix" << std::endl;
    is >> ncols;
    //check input
    if(nrows < 0 || ncols < 0) throw std::invalid_argument("read error -
        negative matrix size");

    // prepare the matrix to hold n elements
    m = Matrix(nrows, ncols);
  }
    // input the elements
    std::cout << "input "<< m.getNrows() * m.getNcols() << " matrix elements"
        << std::endl;
  for (int i = 0; i < m.getNrows(); i++)
    for (int j=0; j< m.getNcols(); j++) is >> m[i][j];

    // return the stream object
    return is;
}

/*
 * screen output, user friendly
 */
std::ostream& operator<<(std::ostream& os, const Matrix& m) {
```

```
  // test to see whether there are any elements
    if (m.getNrows() > 0) {
        os << "The matrix elements are" << std::endl;
        for (int i=0; i<m.getNrows();i++) {
      for (int j=0;j<m.getNcols();j++) {
        os << m[i][j] << " ";
      }
      os << "\n";
    }
        os << std::endl;
    }
    else
    {
        os << "Matrix is empty." << std::endl;
    }
    return os;
}

/*
* file input - raw data, compatible with file writing operator
*/
std::ifstream& operator>>(std::ifstream& ifs, Matrix& m) {

  int nrows, ncols;

    // read size from the file
    ifs >> nrows; ifs>> ncols;
    //check input sanity
    if(nrows < 0 || ncols < 0) throw std::invalid_argument("file read error -
        negative matrix size");

    // prepare the vector to hold n elements
    m = Matrix(nrows, ncols);

    // input the elements
    for (int i=0; i<nrows; i++)
    for (int j=0; j<ncols; j++) ifs >> m[i][j];
```

```
    // return the stream object
    return ifs;
}


/*
* file output - raw data, comaptible with file reading operator
*/
std::ofstream& operator<<(std::ofstream& ofs, const Matrix& m) {
    //put matrix rownumber in first line (even if it is zero)
    ofs << m.getNrows() << std::endl;
  //put matrix columnnumber in second line (even if it is zero)
  ofs << m.getNcols() << std::endl;
    //put data in third line (if size==zero nothing will be put)
    for (int i=0; i<m.getNrows(); i++) {
    for (int j=0; j<m.getNcols(); j++) ofs << m[i][j] << " ";
    ofs << std::endl;
  }
    return ofs;
}
```

Heat Conduction Equation

Generated by Doxygen 1.8.11

Contents

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

Chapter 2

Class Index

2.1 Class List

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

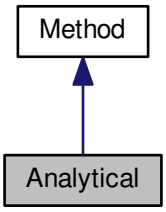
Chapter 3

Class Documentation

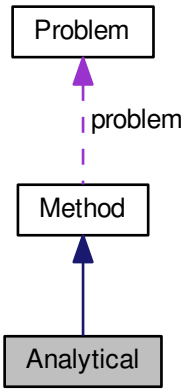
3.1 Analytical Class Reference

`#include <analytical.h>`

Inheritance diagram for Analytical:



Collaboration diagram for Analytical:



Public Member Functions

- Analytical (Problem problem)
- void compute_solution ()

Additional Inherited Members

3.1.1 Detailed Description

An Analytical class to compute the solution with standard procedures
The implementation is derived from the Method Object

The Analytical class provides:
-a basic constructor for an object,
-a method to compute a solution with the correct procedures

3.1.2 Constructor & Destructor Documentation

3.1.2.1 Analytical::Analytical (Problem *problem*)

Default constructor. Intialize a Analytical object

3.1.3 Member Function Documentation

3.1.3.1 void Analytical::compute_solution ( ) [virtual]

Normal public method. compute the solution with specific given rules

Implements Method.

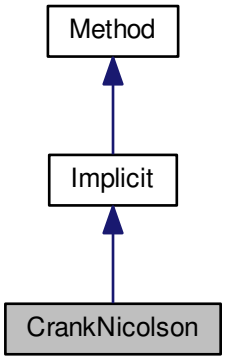
The documentation for this class was generated from the following files:

- methods/analytical.h
- methods/analytical.cpp

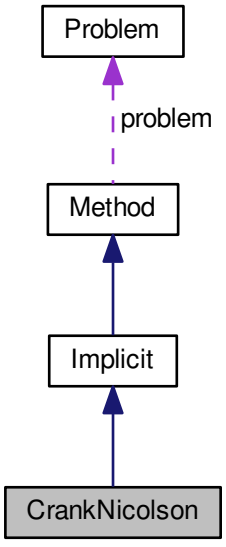
3.2 CrankNicolson Class Reference

`#include <crank_nicolson.h>`

Inheritance diagram for CrankNicolson:



Collaboration diagram for CrankNicolson:



Public Member Functions

- CrankNicolson (Problem problem)

Protected Member Functions

- Vector build_r (Vector previous_step)

Additional Inherited Members

3.2.1 Detailed Description

A CrankNicolson method class that contains a r vector builder.
This builder is used to calculate the r vector in A.x = r linear equation system.

The CrankNicolson class provides:
-a basic constructor for creating a CrankNicolson method object.
-a method to compute the r vector.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 CrankNicolson::CrankNicolson (Problem *problem*)

Default constructor.

3.2.3 Member Function Documentation

3.2.3.1 Vector CrankNicolson::build_r (Vector *previous_step*) [protected], [virtual]

Normal protected method. get the number of rows

Parameters

previous_step	Vector representing the solution of the previous time step.

Returns

Vector. r vector to be used in A.x = r

Implements Implicit.

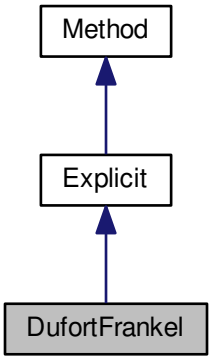
The documentation for this class was generated from the following files:

- methods/implicit/crank_nicolson.h
- methods/implicit/crank_nicolson.cpp

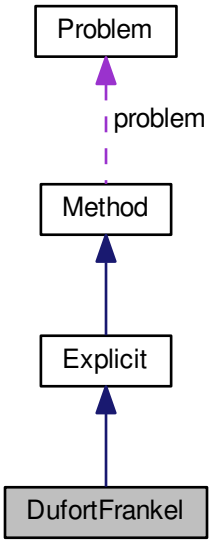
3.3 DufortFrankel Class Reference

`#include <dufort_frankel.h>`

Inheritance diagram for DufortFrankel:



Collaboration diagram for DufortFrankel:



Public Member Functions

- DufortFrankel (Problem problem)

Protected Member Functions

- Vector build_iteration (Vector current_step, Vector previous_step)

Additional Inherited Members

3.3.1 Detailed Description

A DufortFrankel method class that contains an iteration builder.
This builder is used to calculate a solution using the Dufort-Frankel mathod.

The DufortFrankel class provides:
-a basic constructor for creating a DufortFrankel method object.
-a method to compute a solution of the current iteration

3.3.2 Constructor & Destructor Documentation

3.3.2.1 DufortFrankel::DufortFrankel (Problem *problem*)

Default constructor.

3.3.3 Member Function Documentation

3.3.3.1 Vector DufortFrankel::build_iteration (Vector *current_step*, Vector *previous_step*) [protected], [virtual]

Normal protected method. Calculate a next time step solution requiring a previous time step and a current time step solution.

Parameters

current_step	A vector representing the current time step solution.
previous_step	A vector representing the previous time step solution.

Returns

Vector. The computed solution.

Implements Explicit.

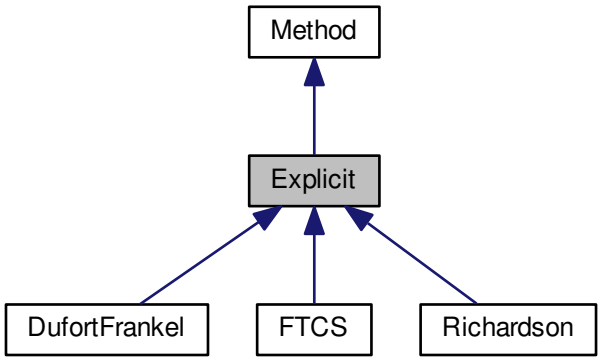
The documentation for this class was generated from the following files:

- methods/explicit/dufort_frankel.h
- methods/explicit/dufort_frankel.cpp

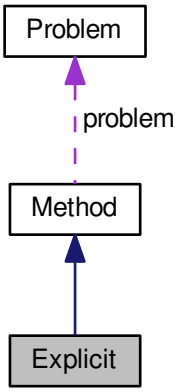
3.4 Explicit Class Reference

`#include <explicit.h>`

Inheritance diagram for Explicit:



Collaboration diagram for Explicit:



Public Member Functions

- Explicit (Problem problem)
- void compute_solution ()

Protected Member Functions

- virtual Vector build_iteration (Vector current_step, Vector previous_step)=0

Additional Inherited Members

3.4.1 Detailed Description

An explicit method class that contains default methods that only explicit methods use
The implementation is derived from the Method class

The Explicit class provides:
-a basic constructor for creating an explicit method object.
-a method to compute a solution following explicit methods rules

3.4.2 Constructor & Destructor Documentation

3.4.2.1 Explicit::Explicit (Problem *problem*)

Default constructor.

3.4.3 Member Function Documentation

3.4.3.1 virtual Vector Explicit::build_iteration (Vector *current_step*, Vector *previous_step*) [protected], [pure virtual]

A pure virtual member. Build the solution of the next time step, using the previous time step and the next time step solutions

Parameters

previous_step	A vector containing the previous time step solution.
current_step	A vector containing the current time step solution.

Returns

Vector. A vector representing the next time step solution.

Implemented in FTCS, DufortFrankel, and Richardson.

3.4.3.2 void Explicit::compute_solution ( ) [virtual]

Normal public method. Calculates a solution for the given problem by populating the solution grid with the correct values.

Implements Method.

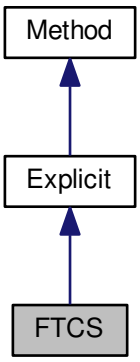
The documentation for this class was generated from the following files:

- methods/explicit/explicit.h
- methods/explicit/explicit.cpp

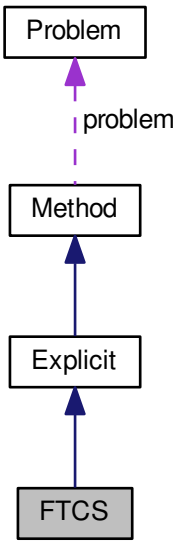
3.5 FTCS Class Reference

`#include <forward_t_central_s.h>`

Inheritance diagram for FTCS:



Collaboration diagram for FTCS:



Public Member Functions

- FTCS (Problem problem)
- Vector build_iteration (Vector current_step, Vector previous_step)

Additional Inherited Members

3.5.1 Detailed Description

A FTCS method class that contains an iteration builder.
This builder is used to calculate the first iteration of explicit methods, since it only requires the previous step solution to do it.

The FTCS class provides:
-a basic constructor for creating a FTCS method object.
-a method to compute the current iteration

3.5.2 Constructor & Destructor Documentation

3.5.2.1 FTCS::FTCS (Problem *problem*)

Default constructor.

3.5.3 Member Function Documentation

3.5.3.1 Vector FTCS::build_iteration (Vector *current_step,* Vector *previous_step*) [virtual]

Normal public method. Calculate a solution requiring only the previous time step solution.

Parameters

current_step	A vector with size 0, it's not required in this method.
previous_step	A vector representing the previous time step solution

Returns

Vector. The computed solution.

Implements Explicit.

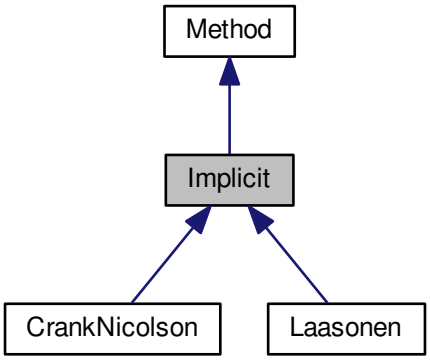
The documentation for this class was generated from the following files:

- methods/explicit/forward_t_central_s.h
- methods/explicit/forward_t_central_s.cpp

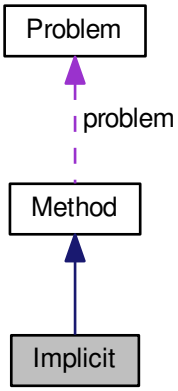
3.6 Implicit Class Reference

`#include <implicit.h>`

Inheritance diagram for Implicit:



Collaboration diagram for Implicit:



Public Member Functions

- Implicit (Problem problem)
- void compute_solution ()

Protected Member Functions

- virtual Vector build_r (Vector previous_step)=0

Additional Inherited Members

3.6.1 Detailed Description

An implicit method class that contains default methods that only implicit methods use
The implementation is derived from the Method class

The Implicit class provides:
-a basic constructor for creating an implicit method object.
-a method to compute a solution following implicit methods rules

3.6.2 Constructor & Destructor Documentation

3.6.2.1 Implicit::Implicit (**Problem** *problem*)

Default constructor.

3.6.3 Member Function Documentation

3.6.3.1 virtual **Vector** Implicit::build_r (**Vector** *previous_step*) [protected], [pure virtual]

A pure virtual member. Build the r vector in a linear system of A.x = r in which A is a matrix, whereas b and r are vectors.
This method is used to compute a solution using the thomas algorithm, which can be used in a triadiogonal matrix.

Parameters

previous_step	A vector containing the previous time step solution.

Returns

Vector. The r vector, which can be used in to calculate the current time step solution with Tomas Algorithm.

Implemented in CrankNicolson, and Laasonen.

3.6.3.2 void Implicit::compute_solution ( ) [virtual]

Normal public method. Calculates a solution for the given problem by populating the solution grid with the correct values.

Implements Method.

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

- methods/implicit/implicit.h
- methods/implicit/implicit.cpp

3.7 IOManager Class Reference

`#include <iomanager.h>`

Public Member Functions

- IOManager ()
- void export_outputs (Method *analytical, std::vector< Method * > methods)

3.7.1 Detailed Description

An input/output manager class to handle plot exportations and future implementations of input handling

The IOManager class provides:
-plot method which compares the analytical solution with a set of given methods, ploting them with a custom configuration using gnuplot

3.7.2 Constructor & Destructor Documentation

3.7.2.1 IOManager::IOManager ( )

Default constructor. Initialize an IOManager object.

3.7.3 Member Function Documentation

3.7.3.1 void IOManager::export_outputs (Method * *analytical*, std::vector< Method * > *methods*)

Exports outputs regarding plots images and error tables for each computed solution, comparing them to the analytical solution

Parameters

*Method**	analytical The analytical solution
vector<Method>*	methods Vector containing the solutions

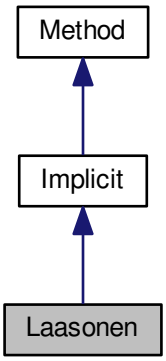
The documentation for this class was generated from the following files:

- io/iomanager.h
- io/iomanager.cpp

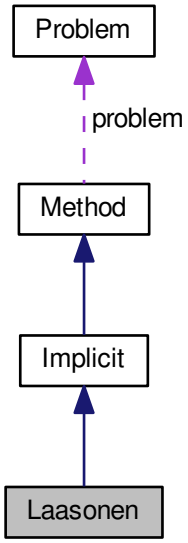
3.8 Laasonen Class Reference

`#include <laasonen.h>`

Inheritance diagram for Laasonen:



Collaboration diagram for Laasonen:



Public Member Functions

- Laasonen (Problem problem)

Protected Member Functions

- Vector build_r (Vector previous_step)

Additional Inherited Members

3.8.1 Detailed Description

A Laasonen method class that contains a r vector builder.
This builder is used is used to calculate the r vector in A.x = r linear equation system.

The Laasonen class provides:
-a basic constructor for creating a Laasonen method object.
-a method to compute the r vector.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 Laasonen::Laasonen (Problem *problem*)

Default constructor.

3.8.3 Member Function Documentation

3.8.3.1 Vector Laasonen::build_r (Vector *previous_step*) [protected], [virtual]

Normal protected method. get the number of rows

Parameters

previous_step	Vector representing the solution of the previous time step.

Returns

Vector. r vector to be used in A.x = r

Implements Implicit.

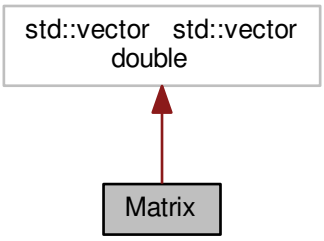
The documentation for this class was generated from the following files:

- methods/implicit/laasonen.h
- methods/implicit/laasonen.cpp

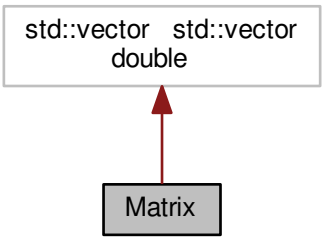
3.9 Matrix Class Reference

```
#include <matrix.h>
```

Inheritance diagram for Matrix:



Collaboration diagram for Matrix:



Public Member Functions

- Matrix ()
- Matrix (int Nrows, int Ncols)
- Matrix (const Matrix &m)
- int getNrows () const
- int getNcols () const
- void set_row (int index, Vector v)
- Matrix & operator= (const Matrix &m)
- bool operator== (const Matrix &m) const
- double one_norm () const
- double two_norm () const
- double uniform_norm () const
- Matrix operator∗ (const Matrix &a) const
- Vector operator∗ (const Vector &v) const
- Matrix transpose () const
- Matrix **mult** (const Matrix &a) const

Friends

- std::istream & operator>> (std::istream &is, Matrix &m)
- std::ostream & operator<< (std::ostream &os, const Matrix &m)
- std::ifstream & operator>> (std::ifstream &ifs, Matrix &m)
- std::ofstream & operator<< (std::ofstream &ofs, const Matrix &m)

3.9.1 Detailed Description

A matrix class for data storage of a 2D array of doubles
The implementation is derived from the standard container vector std::vector
We use private inheritance to base our vector upon the library version whilst usto expose only those base class functions we wish to use - in this the array access operator []

The Matrix class provides:
-basic constructors for creating a matrix object from other matrix object, by creating empty matrix of a given size,
-input and oput operation via >> and << operators using keyboard or file
-basic operations like access via [] operator, assignment and comparision

3.9.2 Constructor & Destructor Documentation

3.9.2.1 Matrix::Matrix ( )

Default constructor. Intialize an empty Matrix object

See also

Matrix(int Nrows, int Ncols)
Matrix(const Matrix& m)

3.9.2.2 Matrix::Matrix (int *Nrows,* int *Ncols*)

Alternate constructor. build a matrix Nrows by Ncols

See also

Matrix()
Matrix(const Matrix& m)

Exceptions

invalid_argument	("matrix size negative or zero")

Parameters

Nrows	int. number of rows in matrix
Ncols	int. number of columns in matrix

3.9.2.3 Matrix::Matrix (const Matrix & *m*)

Copy constructor. build a matrix from another matrix

See also

Matrix()
Matrix(int Nrows, int Ncols)

Parameters

m	Matrix&. matrix to copy from

3.9.3 Member Function Documentation

3.9.3.1 int Matrix::getNcols ( ) const

Normal public get method. get the number of columns

See also

int getNrows()const

Returns

int. number of columns in matrix

3.9.3.2 int Matrix::getNrows ( ) const

Normal public get method. get the number of rows

See also

int getNcols()const

Returns

int. number of rows in matrix

3.9.3.3 double Matrix::one_norm ( ) const

Normal public method that returns a double. It returns L1 norm of matrix

See also

two_norm()const
uniform_norm()const

Returns

double. matrix L1 norm

3.9.3.4 Matrix Matrix::operator∗ (const Matrix & *a*) const

Overloaded ∗operator that returns a Matrix. It Performs matrix by matrix multiplication.

See also

operator∗(const Matrix & a) const

Exceptions

out_of_range	("Matrix access error") One or more of the matrix have a zero size
std::out_of_range	("uncompatible matrix sizes") Number of columns in first matrix do not match number of columns in second matrix

Returns

Matrix. matrix-matrix product

Parameters

a	Matrix. matrix to multiply by

3.9.3.5 Vector Matrix::operator∗ (const Vector & *v*) const

Overloaded ∗operator that returns a Vector. It Performs matrix by vector multiplication.

See also

operator∗(const Matrix & a)const

Exceptions

std::out_of_range	("Matrix access error") matrix has a zero size
std::out_of_range	("Vector access error") vector has a zero size
std::out_of_range	("uncompatible matrix-vector sizes") Number of columns in matrix do not match the vector size

Returns

Vector. matrix-vector product

Parameters

v	Vector. Vector to multiply by

3.9.3.6 Matrix & Matrix::operator= (const Matrix & *m*)

Overloaded assignment operator

See also

operator==(const Matrix& m)const

Returns

Matrix&. the matrix on the left of the assignment

Parameters

m	Matrix&. Matrix to assign from

3.9.3.7 bool Matrix::operator== (const Matrix & *m*) const

Overloaded comparison operator returns true or false depending on whether the matrices are the same or not

See also

operator=(const Matrix& m)

Returns

bool. true or false

Parameters

m	Matrix&. Matrix to compare to

3.9.3.8 void Matrix::set_row (int *index*, Vector *v*)

Normal public set method. replace a row with a given vector

Parameters

index	Index of row to mutate
v	New vector

Exceptions

out_of_range	("index out of range.\n")
out_of_range	("vector size is different from matrix columns number.\n")

3.9.3.9 Matrix Matrix::transpose ( ) const

public method that returns the transpose of the matrix. It returns the transpose of matrix

Returns

Matrix. matrix transpose

3.9.3.10 double Matrix::two_norm ( ) const

Normal public method that returns a double. It returns L2 norm of matrix

See also

one_norm()const
uniform_norm()const

Returns

double. matrix L2 norm

3.9.3.11 double Matrix::uniform_norm ( ) const

Normal public method that returns a double. It returns L_max norm of matrix

See also

one_norm()const
two_norm()const

Returns

double. matrix L_max norm

3.9.4 Friends And Related Function Documentation

3.9.4.1 std::ostream& operator<< (std::ostream & *os,* const Matrix & *m*) [friend]

Overloaded ostream << operator. Display output if matrix has size user will be asked to input only matrix values if matrix was not initialized user can choose matrix size and input it values

See also

operator>>(std::ifstream& ifs, Matrix& m)
operator>>(std::istream& is, Matrix& m)
operator<<(std::ostream& os, const Matrix& m)

Returns

std::ostream&. The ostream object

Parameters

os	Display output stream
m	Matrix to read from

3.9.4.2 std::ofstream& operator<< (std::ofstream & *ofs,* const Matrix & *m*) [friend]

Overloaded ofstream << operator. File output the file output operator is compatible with file input operator, ie. everything written can be read later.

See also

operator>>(std::ifstream& ifs, Matrix& m)
operator<<(std::ofstream& ofs, const Matrix& m)
operator>>(std::istream& is, Matrix& m)

Exceptions

std::invalid_argument	("file read error - negative matrix size");

Returns

std::ofstream&. The ofstream object

Parameters

m	Matrix to read from

3.9.4.3 std::istream& operator>> (std::istream & *is,* Matrix & *m*) [friend]

Overloaded istream >> operator. Keyboard input if matrix has size user will be asked to input only matrix values if matrix was not initialized user can choose matrix size and input it values

See also

operator<<(std::ofstream& ofs, const Matrix& m)
operator>>(std::istream& is, Matrix& m)
operator<<(std::ostream& os, const Matrix& m)

Exceptions

std::invalid_argument	("read error - negative matrix size");

Returns

std::istream&. The istream object

Parameters

is	Keyboard input stream
m	Matrix to write into

3.9.4.4 std::ifstream& operator>>(std::ifstream & *ifs,* Matrix & *m*) [friend]

Overloaded ifstream >> operator. File input the file output operator is compatible with file input operator, ie. everything written can be read later.

See also

operator>>(std::ifstream& ifs, Matrix& m)
operator<<(std::ofstream& ofs, const Matrix& m)
operator<<(std::ostream& os, const Matrix& m)

Returns

std::ifstream&. The ifstream object

Parameters

ifs	Input file stream with opened matrix file
m	Matrix to write into

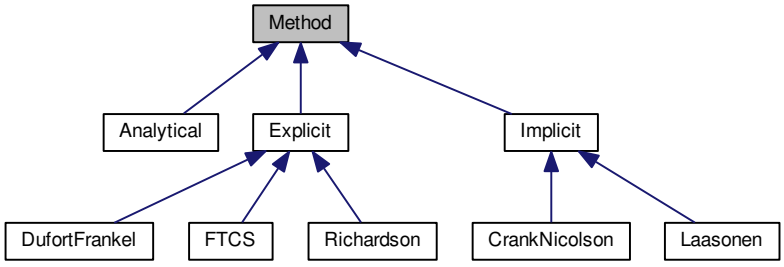
The documentation for this class was generated from the following files:

- grid/matrix.h
- grid/matrix.cpp

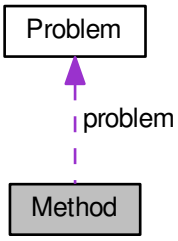
3.10 Method Class Reference

`#include <method.h>`

Inheritance diagram for Method:



Collaboration diagram for Method:



Public Member Functions

- Method ()
- Method (Problem problem)
- std::string get_name ()
- Matrix get_solution ()
- double get_deltat ()
- Vector get_xvalues ()
- double get_computational_time ()
- double get_two_norm ()
- void compute ()
- void **compute_norms** (Matrix analytical_matrix)
- virtual void compute_solution ()=0

Protected Attributes

- Problem problem
- std::string name
- double q

3.10.1 Detailed Description

A Method class to structure information used to solve the problem

The Method class provides:
-basic constructors for creating a Method object.
-acessor methods to retrieve valuable information
-mutator methods to change the problem grid system

3.10.2 Constructor & Destructor Documentation

3.10.2.1 Method::Method ( )

Default constructor. Intialize a Method object

See also

Method(Problem problem)

3.10.2.2 Method::Method (Problem *problem*)

Alternate constructor. Initializes a Method with a given parabolic problem.

See also

Method()

3.10.3 Member Function Documentation

3.10.3.1 void Method::compute ( )

Normal public method. Keeps track of the time to compute a solution

3.10.3.2 virtual void Method::compute_solution ( ) [pure virtual]

A pure virtual member. compute the solution following the rules of a given method.

Implemented in Implicit, Explicit, and Analytical.

3.10.3.3 double Method::get_computational_time ( )

Normal public get method. get the elapsed time value to compute a solution

Returns

double. Elapsed time throughout the computation.

3.10.3.4 double Method::get_deltat ( )

Normal public get method. get the time step of the solution

Returns

double. Solution time step.

3.10.3.5 std::string Method::get_name ( )

Normal public get method. get the method name

Returns

string. Method name.

3.10.3.6 Matrix Method::get_solution ( )

Normal public get method. get the solution grid

Returns

Matrix. Computed solution grid.

3.10.3.7 double Method::get_two_norm ( )

Normal public get method. get the second norm

Returns

double. Second norm value.

3.10.3.8 Vector Method::get_xvalues ( )

Normal public get method. get x values vector

Returns

Vector. x values Vector.

3.10.4 Member Data Documentation

3.10.4.1 std::string Method::name [protected]

Protected string name. Name of the method.

3.10.4.2 Problem Method::problem [protected]

Protected Problem problem. Space step of the solution.

3.10.4.3 double Method::q [protected]

Protected double q. A coeficient which value depends of way the equation is written, it may vary from method to method.

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

- methods/method.h
- methods/method.cpp

3.11 Problem Class Reference

`#include <problem.h>`

Public Member Functions

- Problem ()
- Problem (double dt, double dx)
- unsigned int get_xsize ()
- unsigned int get_tsize ()
- double get_deltax ()
- double get_deltat ()
- Vector get_xvalues ()
- Vector get_tvalues ()
- Vector get_first_row ()
- Matrix * get_solution ()
- void set_time_step (Vector step, double time)
- void set_initial_conditions ()

3.11.1 Detailed Description

A Problem class to structure relevant information related with the problem

The Problem class provides:
-basic constructors for creating a Problem object.
-acessor methods to retrieve valuable information
-mutator methods to change the solution system

3.11.2 Constructor & Destructor Documentation

3.11.2.1 Problem::Problem ( )

Default constructor. Intialize an empty Problem object

See also

Problem(double dt, double dx)

3.11.2.2 Problem::Problem (double *dt*, double *dx*)

Intialize Problem object with specific time and space steps

See also

Problem()

Parameters

dt	Time step to assign
dx	Space step to assign

Exceptions

out_of_range	("space step can't be negative or zero")
out_of_range	("time step can't be negative or zero")

3.11.3 Member Function Documentation

3.11.3.1 double Problem::get_deltat ( )

Normal public get method that returns a double, the time step value of the solution

Returns

double. The time step value of the solution.

3.11.3.2 double Problem::get_deltax ( )

Normal public get method that returns a double, the space step value of the solution

Returns

double. The space step value of the solution.

3.11.3.3 Vector Problem::get_first_row ( )

Normal public get method that returns a Vector, containing the initial boundaries in the first row of the solution

Returns

Vector. The initial boundaries in the first row of the solution.

3.11.3.4 Matrix * Problem::get_solution ( )

Normal public get method that returns a Matrix, containing the solution solution.

Returns

Matrix*. The solution solution.

3.11.3.5 unsigned int Problem::get_tsize ( )

Normal public get method that returns an unsigned int, the number of rows of the solution

Returns

unsigned int. The number of rows of the solution.

3.11.3.6 Vector Problem::get_tvalues ( )

Normal public get method that returns a Vector, containing the time values in each row

Returns

Vector. The time values in each row.

3.11.3.7 unsigned int Problem::get_xsize ( )

Normal public get method that returns an unsigned int, the number of columns of the solution

Returns

unsigned int. The number of columns of the solution.

3.11.3.8 Vector Problem::get_xvalues ( )

Normal public get method that returns a Vector, containing the space values in each column

Returns

Vector. The space values in each column.

3.11.3.9 void Problem::set_initial_conditions ( )

Normal public set method. set the problem initial boundaries.

3.11.3.10 void Problem::set_time_step (Vector *step*, double *time*)

Normal public set method. replace a row of the solution for a given Vector.

Parameters

step	Vector conatining the new values.
time	Corresponding row to be replaced

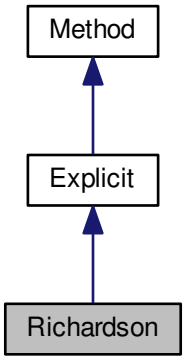
The documentation for this class was generated from the following files:

- variants/problem.h
- variants/problem.cpp

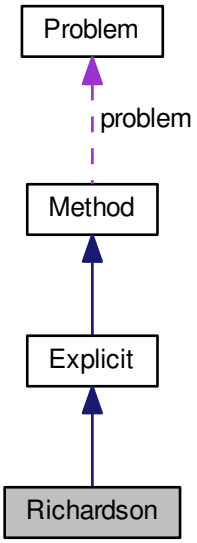
3.12 Richardson Class Reference

`#include <richardson.h>`

Inheritance diagram for Richardson:



Collaboration diagram for Richardson:



Public Member Functions

- Richardson (Problem problem)

Protected Member Functions

- Vector build_iteration (Vector current_step, Vector previous_step)

Additional Inherited Members

3.12.1 Detailed Description

A Richardson method class that contains an iteration builder.
This builder is used to calculate a solution using the Richardson method.

The Richardson class provides:
-a basic constructor for creating a Richardson method object.
-a method to compute a solution of the current iteration

3.12.2 Constructor & Destructor Documentation

3.12.2.1 Richardson::Richardson (Problem *problem*)

Default constructor.

3.12.3 Member Function Documentation

3.12.3.1 Vector Richardson::build_iteration (Vector *current_step,* Vector *previous_step*) [protected], [virtual]

Normal protected method. Calculate a next time step solution requiring a previous time step and a current time step solution.

Parameters

current_step	A vector representing the current time step solution.
previous_step	A vector representing the previous time step solution.

Returns

Vector. The computed solution.

Implements Explicit.

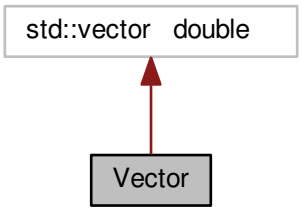
The documentation for this class was generated from the following files:

- methods/explicit/richardson.h
- methods/explicit/richardson.cpp

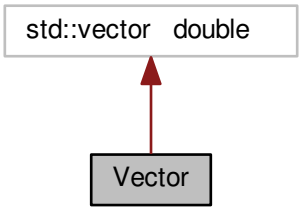
3.13 Vector Class Reference

`#include <vector.h>`

Inheritance diagram for Vector:



Collaboration diagram for Vector:



Public Member Functions

- Vector ()
- Vector (int Num)
- Vector (const Vector &v)
- Vector (std::vector< double > vec)
- Vector & operator= (const Vector &v)
- bool operator== (const Vector &v) const
- int getSize () const
- int find (double value)
- void push_front_back (double value)
- void push (double value)
- double one_norm () const
- double two_norm () const
- double uniform_norm () const

Friends

- std::istream & operator>> (std::istream &is, Vector &v)
- std::ostream & operator<< (std::ostream &os, const Vector &v)
- std::ifstream & operator>> (std::ifstream &ifs, Vector &v)
- std::ofstream & operator<< (std::ofstream &ofs, const Vector &v)

3.13.1 Detailed Description

A vector class for data storage of a 1D array of doubles
The implementation is derived from the standard container vector std::vector
We use private inheritance to base our vector upon the library version whilst usto expose only those base class functions we wish to use - in this the array access operator []

The Vector class provides:
-basic constructors for creating vector obcjet from other vector object, or by creating empty vector of a given size,
-input and oput operation via >> and << operators using keyboard or file
-basic operations like access via [] operator, assignment and comparision

3.13.2 Constructor & Destructor Documentation

3.13.2.1 Vector::Vector ( )

Default constructor. Intialize an empty Vector object

See also

Vector(int Num)
Vector(const Vector& v)

3.13.2.2 Vector::Vector (int *Num*) [explicit]

Explicit alterative constructor takes an intiger. it is explicit since implicit type conversion int -> vector doesn't make sense Intialize Vector object of size Num

See also

Vector()
Vector(const Vector& v)

Exceptions

invalid_argument	("vector size negative")

Parameters

Num	int. Size of a vector

3.13.2.3 Vector::Vector (const Vector & *v*)

Copy constructor takes an Vector object reference. Intialize Vector object with another Vector object

See also

Vector()
Vector(int Num)

3.13.2.4 Vector::Vector (std::vector< double > *vec*)

Copy constructor takes an vector<double> object reference. Intialize Vector object with an vector<double> object

See also

Vector()
Vector(int Num)
Vector(const Vector& v)

3.13.3 Member Function Documentation

3.13.3.1 int Vector::find (double *value*)

Method to find the value index in a vector

Parameters

value	Value to find

Returns

int. -1 if value was not found or the value index otherwise

3.13.3.2 int Vector::getSize ( ) const

Normal get method that returns integer, the size of the vector

Returns

int. the size of the vector

3.13.3.3 double Vector::one_norm ( ) const

Normal public method that returns a double. It returns L1 norm of vector

See also

two_norm()const
uniform_norm()const

Returns

double. vectors L1 norm

3.13.3.4 Vector & Vector::operator= (const Vector & *v*)

Overloaded assignment operator

See also

operator==(const Vector& v)const

Parameters

v	Vector to assign from

Returns

the object on the left of the assignment

Parameters

v	Vecto&. Vector to assign from

3.13.3.5 bool Vector::operator== (const Vector & *v*) const

Overloaded comparison operator returns true if vectors are the same within a tolerance (1.e-07)

See also

operator=(const Vector& v)
operator[ ](int i)
operator[ ](int i)const

Returns

bool. true or false

Exceptions

invalid_argument	("incompatible vector sizes\n")

Parameters

v	Vector&. vector to compare

3.13.3.6 void Vector::push (double *value*)

Method to push a value to the last position of a Vector

Parameters

value	Value to be pushed

3.13.3.7 void Vector::push_front_back (double *value*)

Method to push a value to the first and last position of a Vector

Parameters

value	Value to insert

3.13.3.8 double Vector::two_norm ( ) const

Normal public method that returns a double. It returns L2 norm of vector

See also

one_norm()const
uniform_norm()const

Returns

double. vectors L2 norm

3.13.3.9 double Vector::uniform_norm ( ) const

Normal public method that returns a double. It returns L_max norm of vector

See also

one_norm()const
two_norm()const

Exceptions

out_of_range	("vector access error") vector has zero size

Returns

double. vectors Lmax norm

3.13.4 Friends And Related Function Documentation

3.13.4.1 std::ostream& operator<< (std::ostream & *os,* const Vector & *v*) [friend]

Overloaded ifstream << operator. Display output.

See also

operator>>(std::istream& is, Vector& v)
operator>>(std::ifstream& ifs, Vector& v)
operator<<(std::ofstream& ofs, const Vector& v)

Returns

std::ostream&. the output stream object os

Parameters

os	output file stream
v	vector to read from

3.13.4.2 std::ofstream& operator<< (std::ofstream & *ofs,* const Vector & *v*) [friend]

Overloaded ofstream << operator. File output. the file output operator is compatible with file input operator, ie. everything written can be read later.

See also

operator>>(std::istream& is, Vector& v)
operator>>(std::ifstream& ifs, Vector& v)
operator<<(std::ostream& os, const Vector& v)

Returns

std::ofstream&. the output ofstream object ofs

Parameters

ofs	outputfile stream. With opened file
v	Vector&. vector to read from

3.13.4.3 std::istream& operator>> (std::istream & *is,* Vector & *v*) [friend]

Overloaded istream >> operator. Keyboard input if vector has size user will be asked to input only vector values if vector was not initialized user can choose vector size and input it values

See also

operator>>(std::ifstream& ifs, Vector& v)
operator<<(std::ostream& os, const Vector& v)
operator<<(std::ofstream& ofs, const Vector& v)

Returns

std::istream&. the input stream object is

Exceptions

std::invalid_argument	("read error - negative vector size");

Parameters

is	keyboard input straem. For user input
v	Vector&. vector to write to

3.13.4.4 std::ifstream& operator>> (std::ifstream & *ifs,* Vector & *v*) [friend]

Overloaded ifstream >> operator. File input the file output operator is compatible with file input operator, ie. everything written can be read later.

See also

operator>>(std::istream& is, Vector& v)
operator<<(std::ostream& os, const Vector& v)
operator<<(std::ofstream& ofs, const Vector& v)

Returns

ifstream&. the input ifstream object ifs

Exceptions

std::invalid_argument	("file read error - negative vector size");

Parameters

ifs	input file straem. With opened matrix file
v	Vector&. vector to write to

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

- grid/vector.h
- grid/vector.cpp

Index