HIGH PERFORMANCE TECHNICAL COMPUTING

Heat Conduction Equation

January 31, 2018

António Pedro Araújo Fraga

Student ID: 279654

Cranfield University

M.Sc. in Software Engineering for Technical Computing

Contents

Abstract

Three numerical schemes were applied to compute a solution for a parabolic partial differential equation, the heat conduction equation. It was used an explicit scheme and two implicit schemes. The solutions were computed both sequentially and in parallel making use of **MPI**, **M**essage **P**assing **I**nterface technology. Two pairs of **time** and **space** steps were studied, along with their effect on the solving system. **Speed-ups** of solutions computed with several processes were analysed, and their values were discussed.

Table 1: Nomenclature

Speedup	S
Speedup with **p** processes	S_p
Forward in Time Central in Space scheme	FTCS
Sequential execution time	T
Execution time in parallel with **p** processes	T_p
Lower index of space step computed by a process	$lower$
Upper index of space step computed by a process	$upper$
Time step	Δt
Space step	Δx

Introduction

Numerical methods are used to obtain an approximated solution for a given problem. The exact solution of those problems are usually computed in **Nondeterministic Polynomial Time**, therefore an approximated solution is often accepted. The approximation factor is often related with the number of space steps that a time step is split on. A high number of space steps leads to a more accurate solution, keeping in mind the existence of **round-off** errors [1].

Although, decreasing the value of space steps results in a more "computational hungry" process. Which means that the time used to compute a solution increases, unless several processes are used to compute the final solution. Parallel computing is simple if it's done with **independent** data. In this case, there's no need to exchange data between workers. Sometimes the methods used to compute the solution have dependencies between space steps calculations. Thus, unless a process communicates with different processes, it's not possible to compute an approximated solution.

The challenge is to minimize the communication overhead. To achieve that, one has to keep in mind how to perform communications. Deciding **when** and **how** to **exchange** data between processes.

Memory management plays an important role on **efficiency** as well. A program is split into **segments**, and each segment is split into **pages**. The size of each segment depends of the compiler. Each of them contains important data like the **stack**, **heap**, and actual data. But

each **page** is divided into equal parts, **4kb** on UNIX based systems. Therefore, whenever a program accesses an address, a page is loaded into **cache**, which is a very fast type of memory close to the **CPU**. This means that accessing memory within the same page won't add the overhead of moving the data from the main memory to cache again. Such knowledge might make one think of how to write specific parts of the code.

Problem definition

The problem is defined in the previously developed **report**[1] for **Computational Methods** & **C++** modules. It is intended to examine the application of distributed memory parallel programming techniques on the referred problem.

Analysis

The analysis of **parallel programming** can be done by measuring times of execution. One should measure the time of execution with a single process, comparing it with the execution time in parallel with **p** processors. Therefore, the **Speedup** concept can be defined[2],

$$Speedup(P) = \frac{Time_{Sequential}}{Time_{parallel}(P)}$$

It is important to define the concept of **Theoretical Speedup** as well. Whenever a program is computed in parallel, the optimal speedup is obtained by taking the number of processes used to compute a solution. Often, the **Optimal Speedup** is difficult to be achieved. This happens because the solution is computed in a distributed system. Processes, by default, are not able to access data from different processes easily. One must use a form of **IPC**, or **I**nter **P**rocess **C**ommunication. The **MPI** technology offers an **API** to achieve that.

Procedures

Every solution was computed on **Delta**, a supercomputer located at Cranfield University. The computational time was measured by using the **MPI Wtime** call, and assuring that every process was entering and leaving the computation of a given method at the same time.

Three different schemes/methods were used to compute a solution for the given problem, sequentially and in parallel. One of them is an explicit schemes, **Forward in Time, Central in Space**, and two of them are implicit schemes, **Laasonen Simple Implicit** and **Crank-Nicholson**. Two pairs **space** and **time** steps were studied,

- $\Delta t = 0.1$ and $\Delta x = 0.5$
- $\Delta t = 0.001$ and $\Delta x = 0.005$

It was seen in the previous work[1] that these schemes can be written in its discretized form.

A **time step** can be defined as an array, divided by the number of space steps to be computed. In a sequential algorithm, the process handles the entire domain. Since it was intended to parallelize the computational method, the **time step** can be divided into **p almost equal** parts, with **p** being the number of processes available. Notice that having **more** processes than space steps is an exception. Thereby, the **lower** and **upper** bounds of space steps to be computed by a process can be determined by it's ranking.

$$lower = \frac{ranking \times number_{SpaceSteps}}{number_{Processes}} \qquad upper = \frac{(ranking+1) \times number_{SpaceSteps}}{number_{Processes}} - 1$$

Defining these values as integers will avoid any kind of **floating point** values.

We can not, however, define these rules if it happens to have more available processes than space steps. In this case, only the number of processes corresponding to the number of space steps can be used. Therefore, it was created a new **communication world** if this condition was met, using only the necessary processes to compute the solution.

Notice that the process with the **rank** = 0, computes the first window of values, whereas the process with **rank** = 1 computes the second window of values. The same idea is replicated to the next processes and windows.

Explicit Scheme

It is known that these type of schemes rely only on values from the previous time steps to compute the solution[1]. Therefore, once one has access to this values, it is possible to compute the values for the current time step. In this cases, processes that need values from different workers, can obtain them by using the MPI API.

Forward in Time Central in Space

The stencil for this scheme can be observed on **Figure 1**. When computing the **first** time step for this scheme, the previous time step values correspond to the **initial conditions**. This means that no data had to be exchanged during the calculations of that iteration. Every process had knowledge of these conditions.

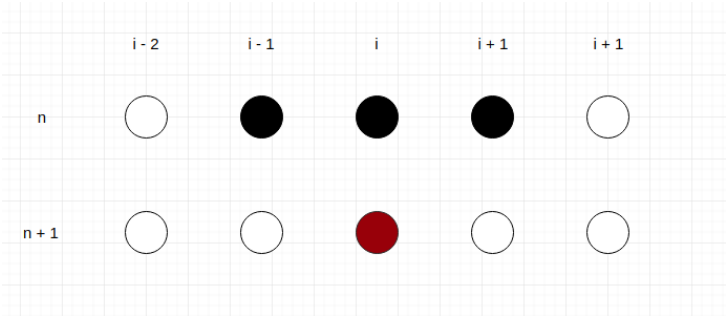


Figure 1: Forward in Time Central in Space method stencil.

However, the next time steps calculations of each process required values that didn't exist in their memory. **Non-blocking** communication was used to exchange dependable data. It was known that the next iteration to be computed by those processes needed the first time step value from the processes that were responsible to compute the next window of values. Therefore, whenever a process finished to compute the first value of an iteration, was responsible to send that value to the previous process. With the exception of the **root** process, that knows the value from the **Initial Conditions**. By this time, the processes could expect to receive a value from the next process, so the same **Non-blocking** communication was used to obtain it. **Figure 2** contains a graphic representation of the sending messages on the first iteration.

Following the same mindset, whenever a process reached the last value to be computed, it was sending that information to the process responsible for the next window. Knowing that

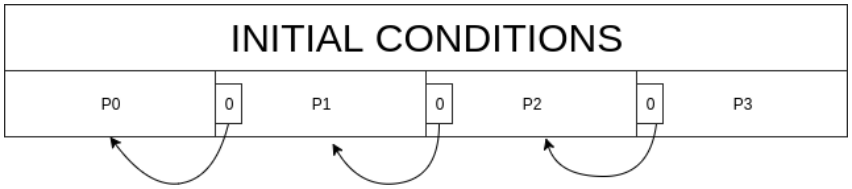


Figure 2: Processes sending first value to the previous worker.

it would have to receive a value from the previous process. Notice that the **ranking** of the receiving process could be discovered by incrementing its own ranking, and the ranking of the sending process was found by decrementing it. The process with the highest ranking was not sending or receiving any value from the next worker, because there was none. It could also access to the surface temperature value, a **known** and constant variable. The graphical representation of this phase can be observed on **Figure 3**.

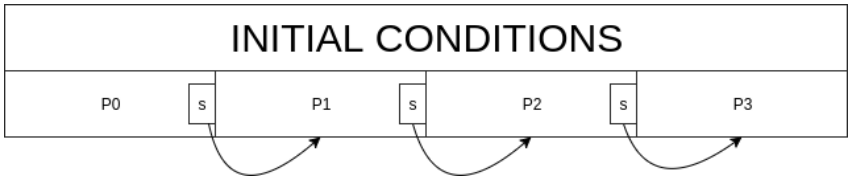


Figure 3: Processes sending first value to the previous worker.

The advantage of using **Non-blocking** communication, is that processes didn't have to go into the **waiting** state whenever they were sending or receiving messages. The **MPI Wait** call was used by the time processes were about to use those values. For the next iteration, whenever a process was computing the first and last value of its own window, it had to wait for the receiving value. The process would enter into a **waiting** state if the request was not fulfilled by that time.

This mechanism of communication was used in nearly every time step calculations, **except** for the **first** and **last** one, when there was no need of communications.

Implicit Schemes

Implicit schemes rely on both previous and current time step in order to calculate the desired value[1]. The methods stencil can be observed on **Figure 4** and **Figure 5**.

The time step computation of these schemes requires that a special form of a linear system of equations to be solved. This system $Ex = b$ contains a **tridiagonal matrix**, the

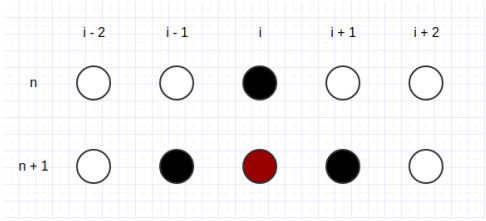


Figure 4: Laasonen's method stencil.

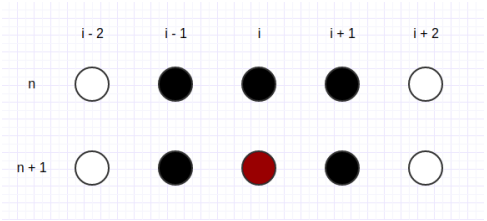


Figure 5: Crank-Nicholson's method stencil.

matrix **E**, which can be divided into two different matrices, **A** and **S**[4]. In case of a sequential code, the **Thomas Algorithm** is the most efficient solution. With a parallel code, the **spike** algorithm can be used to solve dependencies among processes. The results can be achieved by splitting the computation methodology into three different phases.

- Solve **Ay = b** in parallel.
- Gather the top and bottom values of **y** into one process and solve dependencies, computing a vector **x**. Broadcast the solution.
- Solve **Sx = y** in parallel.

Since the matrix **A** is a **tridiagonal** matrix, it can be divided into several smaller matrices, a **block tridiagonal matrix**[3], like seen in **Figure 6**. Each process was responsible to solve a smaller linear system of equations, obtaining its own **y** array.

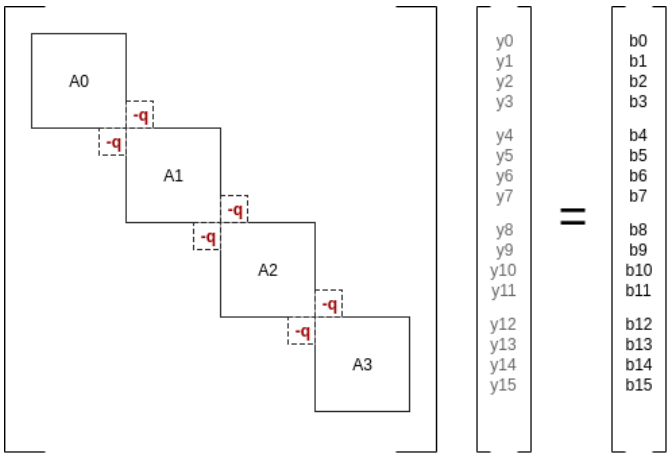


Figure 6: Division of main matrix in blocks.

This implies that several variables are not used in this phase (**-q**). Thus, two different arrays were created[4], **v'** and **w'**,

$$v' = \begin{bmatrix} 0 \\ 0 \\ \cdots \\ -q \end{bmatrix} \qquad w' = \begin{bmatrix} -q \\ \cdots \\ 0 \\ 0 \end{bmatrix}$$

These arrays were used to compute the **spikes** arrays. They can be computed by solving **Av = v'** and **Aw = w'**, creating two different arrays, **v** and **w**. And since **A** is a tridiagonal matrix, they can be computed with **Thomas Algorithm** as well. These arrays are useful to create the matrix **S** of the last phase of the **spike** algorithm.

In order to solve the **Sx = y**, one has to solve a special form of this equation first. Involving every **top** and **bottom** element of the **y** array of each process, forming the array **y'**. This information was gathered by the **root** process. In order to solve dependencies between processes, the **S'x' = y'** system was solved. Note that the v_t, v_b, w_t and w_b values are the **top** and **bottom** values of both **v** and **w** arrays, the **spike arrays**, that were previously calculated.

$$\begin{bmatrix} 1 & 0 & v_t & & & & & \\ 0 & 1 & v_b & & & & & \\ & w_t & 1 & 0 & v_t & & & \\ & w_b & 0 & 1 & v_b & & & \\ & & & w_t & 1 & 0 & v_t & \\ & & & w_b & 0 & 1 & v_b & \\ & & & & & w_t & 1 & 0 \\ & & & & & w_b & 0 & 1 \end{bmatrix} \begin{bmatrix} x'0_t \\ x'0_b \\ x'1_t \\ x'1_b \\ x'2_t \\ x'2_b \\ x'3_t \\ x'3_b \end{bmatrix} = \begin{bmatrix} y'0_t \\ y'0_b \\ y'1_t \\ y'1_b \\ y'2_t \\ y'2_b \\ y'3_t \\ y'3_b \end{bmatrix}$$

This system of equations can be solved by using the **Gaussian Elimination** method, and the **x'**, can be *broadcasted* to the rest of the processes.

The last phase consists of solving the **Sx = y** system, that can be defined in **Figure 7**.

Therefore, the solution can be obtained with[4],

$$\begin{cases} X_0 = Y_0 - V X_1^t \\ X_j = Y_j - V X_{j+1}^t - W X_{j-1}^b \\ X_{P-1} = Y_{P-1} - W X_{P-2}^b \end{cases}$$

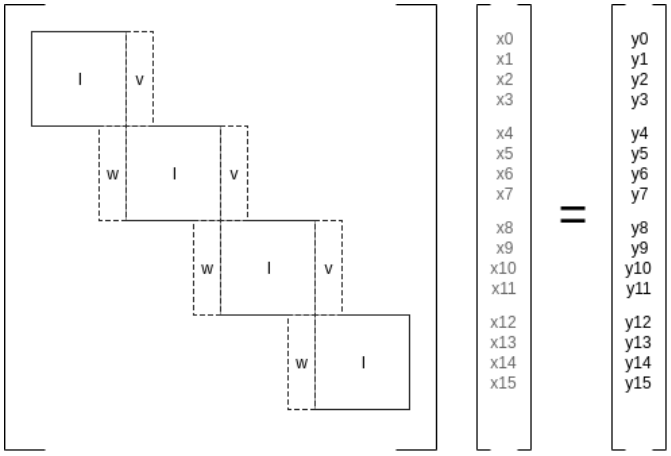


Figure 7: Sx = y system of equations.

Results & Discussion

In this section the results of the experiments were analysed. Starting by observing **Figure 8**, it can be seen that the solution of the **explicit** scheme is not accurate at these conditions. Notice that this solution was computed by **4** processes, with a $\Delta t = 0.001$ and $\Delta x = 0.005$. This phenomenon can be explained by making the stability analysis of this method[1]. On this conditions, this method is unstable, therefore it can not present an approximated solution. It can be seen that the implicit schemes, however, were able to present good results.

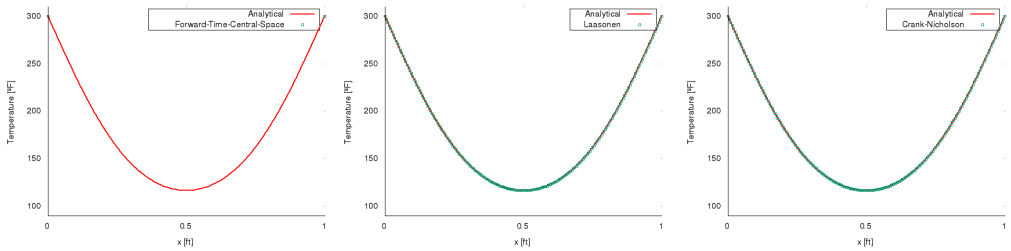


Figure 8: FTCS, Laasonen and Crank-Nicholson solutions calculated with 4 processors, at t = 0.4, with $\Delta t = 0.001$ and $\Delta x = 0.005$.

As referred, it was possible to measure the computation time of a given method. The results for a sequential calculations were quite similar to the previous results[1]. The execution time of a solution for $\Delta t = 0.001$ **and** $\Delta x = 0.005$ did was stable as the number of processes were increased. This happened because the **software** implementation allowed only one process to work. Notice that under these conditions, each time step contains **three** different values. **Two** of those values are known, as they are considered as **initial conditions**.

Therefore, there was only one value to be computed for every time step.

The results of a parallel execution for the $\Delta t = 0.001$ **and** $\Delta x = 0.005$ can be observed on **Figure 9**. Under these conditions, **199** values had to be computed for every time step. By observing the figure, and comparing the sequential solution execution time of the methods, we can see a continuous improvement of the **Forward in Time, Central in Space** scheme. This improvement couldn't be replicated by the implicit schemes. The execution time of these schemes started to increase when the solution was calculated by more than **four** processes.

It was referred that the dependencies among the several processes computing the **Forward in Time, Central in Space** solution were handled using **non-blocking** communication. Whereas the implicit schemes were using a blocking communication to solve their dependencies. A blocking communication was used because the **second** phase of the **spike** algorithm required a sequential flow of data. Thereby, one can conclude that this phase was a major bottleneck while measuring the execution time of these solutions.

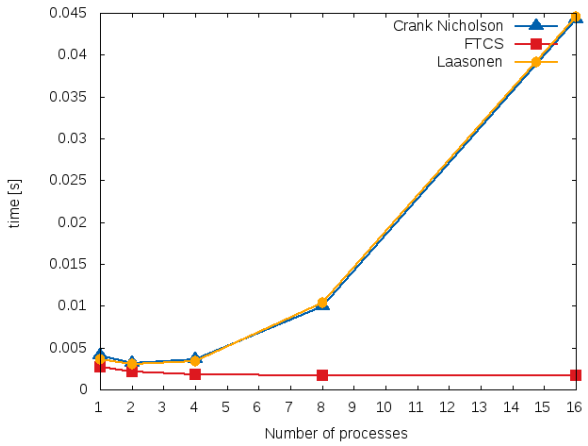


Figure 9: Comparison of execution times.

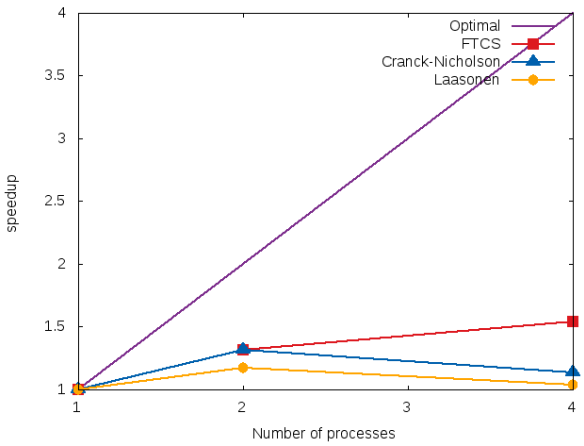


Figure 10: Speedup of each method.

However, while observing **Figure 10**, it could be seen that the **speedup** value of the **Forward in Time, Central in Space** scheme is far from its optimal value. In fact this value is significantly low. A larger **speedup** value would be seen while computing a larger grid. In such small system, the bottleneck of communications is high, and therefore one can not obtain big improvements. An experiment with a system of **thousands** of elements in each time step would contribute to a later perceived bottleneck. Clarifying, the bottleneck would only be visible with a higher number of processes.

The implemented **tridiagonal** solver was compared to the **Lapack** library **tridiagonal** solver. While observing **Figure 11**, it can be seen that the execution time of the implemented solver was faster than the Lapack one. The **Thomas algorithm** has a **linear** time complexity,

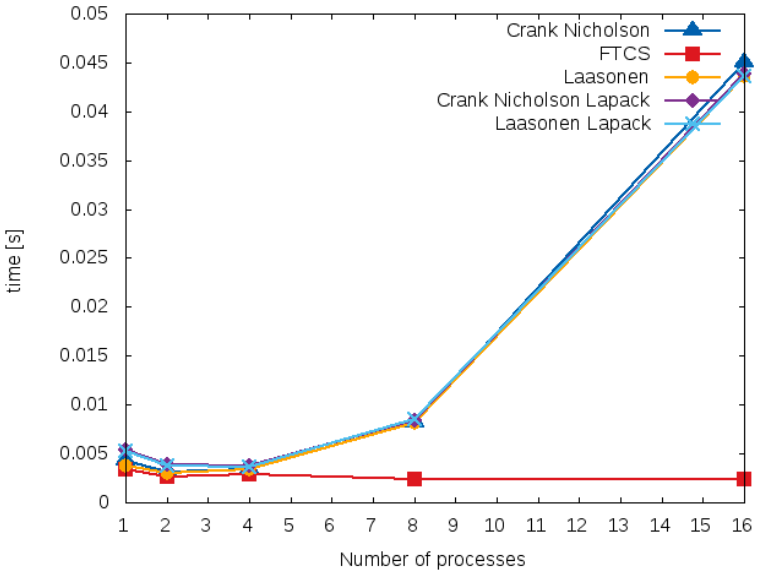


Figure 11: Comparison of execution times with one process.

making it one of the fastest algorithms in sequential code. The significantly slow time obtained by the Lapack implementation may have to do with the architecture where the code is running. It can be possibly explained with **memory management** matters as well. It's also possible that these methods would behave better with a larger grid.

Conclusions

In conclusion, computational power may not be the more important factor when dealing with High Performance Computing.

It was seen that memory plays an important role on this area as well. In order to build an efficient solver, one must take attention of **how** to deal with the several **data structures** used to obtain a solution.

The overhead of communication among processes is often a major bottleneck that one must minimize as much as possible. Is also known that a solution with independent data can be more efficient than systems which rely on information spread across the solution. Whenever the number of communications between processes is significantly high, it is expected to obtain less efficient solvers.

The use of distributed systems becomes advantageous when dealing with large problems. Problems that often require a high number of calculations within a process. Those calculations can often be distributed by workers in order to minimize execution times. Bigger solution tend to minimize the communication bottleneck as well. Whenever the ratio between number of communications and work load per process is minimized, one can expect a better performance.

Like referred before, building a system requires a "near to optimal" level of synchronization. It is implied to think **when**, **how** and **what** to exchange between workers.

References

[1] António Pedro Fraga, December 2017, *Heat Conduction Equation, C++ & Computational Methods*, Available at: <http://pedrofraga.me/heat-conduction-equation.pdf> [Accessed 24 January 2018]

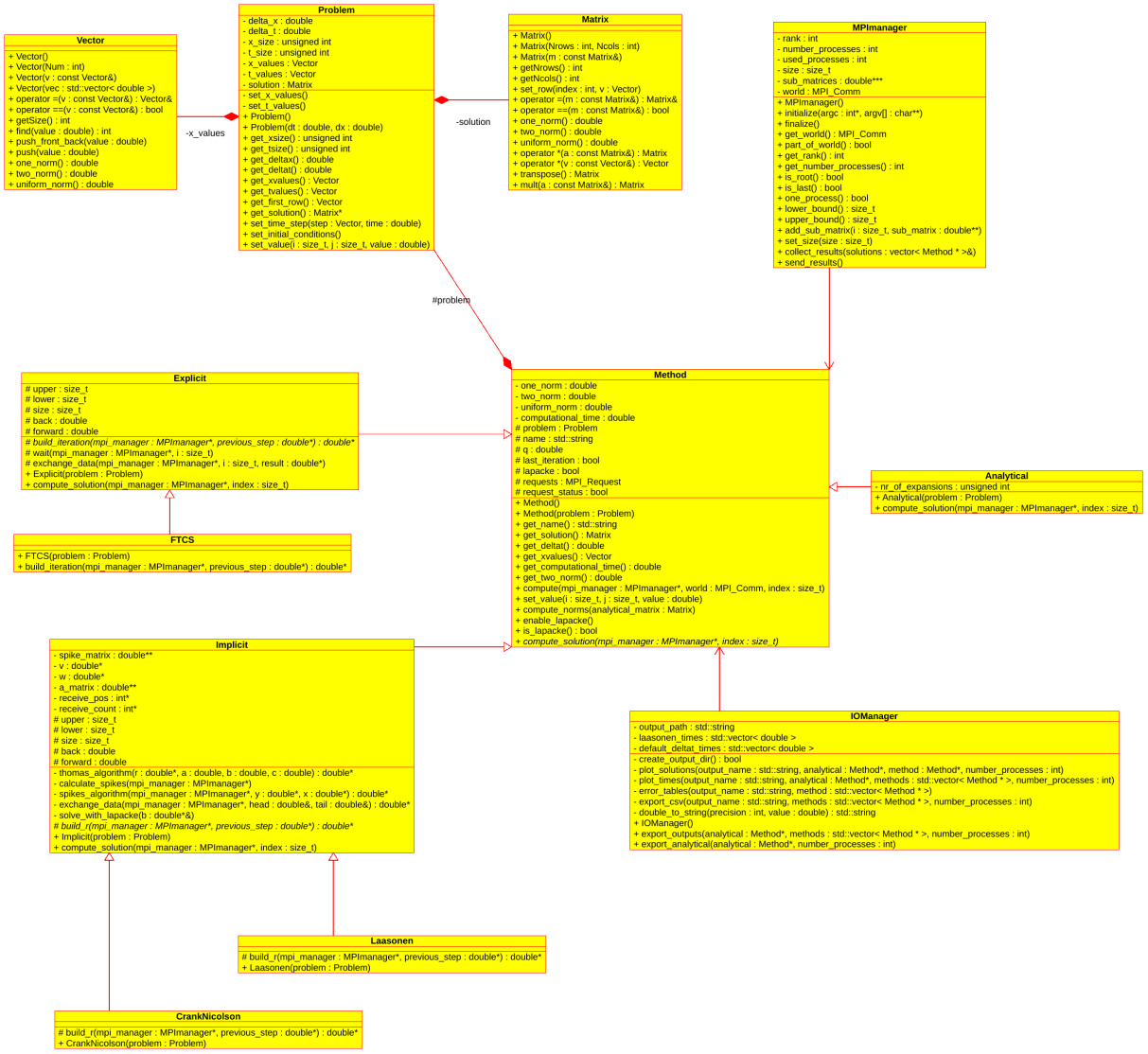
[2] Multiple authors, May 2017, *A Comprehensive Linear Speedup Analysis for Asynchronous Stochastic Parallel Optimization from Zeroth-Order to First-Order*, Available at: <http://xrlian.com/res/Asyn_Comprehensive/paper.pdf> [Accessed 24 January 2018]

[3] Li-Wen Chang, 2014, *Scalable Parallel Tridiagonal Algorithms with Diagonal Pivoting and their optimization for Many-Core Architectures*, Available at: <http://impact.crhc.illinois.edu/shared/Papers/Chang-Thesis-2014.pdf> [Accessed 25 January 2018]

[4] Eric Polizzi, Ahmed H. Sameh, February 2016, *A parallel hybrid banded system solver: the SPIKE algorithm*, Available at: <http://impact.crhc.illinois.edu/shared/Papers/Chang-Thesis-2014.pdf> [Accessed 25 January 2018]

Appendices

Class Diagram



Source Code

main.cpp

```
#include <iostream>
#include "io/iomanager.h"
#include "methods/analytical.h"
#include "methods/explicit/forward_t_central_s.h"
#include "methods/implicit/laasonen.h"
#include "methods/implicit/crank_nicolson.h"
#include "mpi/mpimanager.h"

/*
  Main file - creates a problem, solving it with the different methods, ending
      by exporting the results.
*/
int main(int argc, char * argv[]) {

  MPImanager * mpi_manager = new MPImanager();
  mpi_manager->initialize(&argc, &argv);

  for (size_t i = 0; i < 2; i++) {
    for (size_t j = 0; j < 2; j++) {
      IOManager io_manager;
      Problem * default_problem = new Problem(DELTA_T[i], DELTA_X[i]);
      mpi_manager->set_size(default_problem->get_xsize() - 1);

      if (mpi_manager->part_of_world()) {

        size_t lower = mpi_manager->lower_bound(), upper =
            mpi_manager->upper_bound();

        Analytical * analytical = new Analytical(*default_problem);
        FTCS * ftcs = new FTCS(*default_problem);
        Laasonen * laasonen = new Laasonen(*default_problem);
        CrankNicolson * crank_nicolson = new CrankNicolson(*default_problem);
```

```
      if (j == 1) {
        laasonen->enable_lapacke();
        crank_nicolson->enable_lapacke();
      }

      std::vector<Method*> solutions = {analytical, laasonen, crank_nicolson,
          ftcs};

      for (size_t index = 0; index < solutions.size(); index++) {
        solutions[index]->compute(mpi_manager, mpi_manager->get_world(),
            index);
      }

      if (mpi_manager->is_root()) {
        std::cout << "Collecting results." << std::endl;
        mpi_manager->collect_results(solutions);
        io_manager.export_analytical(solutions[0],
            mpi_manager->get_number_processes());
        std::vector<Method*> methods(solutions.begin() + 1, solutions.end());
        io_manager.export_outputs(solutions[0], methods,
            mpi_manager->get_number_processes());
      } else {
        std::cout << "Sending results." << std::endl;
        mpi_manager->send_results();
      }
    }
  }

  MPI_Barrier(MPI_COMM_WORLD);
 }

 mpi_manager->finalize();

 return 0;
}
```

mpi/mpimanager.h

```
#ifndef MPI_MANAGER_H //include guard
#define MPI_MANAGER_H

#include <mpi.h>
#include "../variants/utils.h"
#include "../methods/method.h"
#include <vector>
using namespace std;

class MPImanager {
private:
  int rank;
  int number_processes;
  int used_processes;
  size_t size;
  double *** sub_matrices;
  MPI_Comm world;
public:
  MPImanager();
  void initialize(int *argc, char ** argv[]);
  void finalize();
  MPI_Comm get_world();
  bool part_of_world();
  int get_rank();
  int get_number_processes();
  bool is_root();
  bool is_last();
  bool one_process();
  size_t lower_bound();
  size_t upper_bound();
  void add_sub_matrix(size_t i, double ** sub_matrix);
  void set_size(size_t size);

  void collect_results(vector<Method*> &solutions);
  void send_results();
```

```
};

#endif
```

mpi/mpimanager.cpp

```
#include "mpimanager.h"


MPImanager::MPImanager() {}

void MPImanager::initialize(int *argc, char ** argv[]) {
  MPI_Init(argc, argv);
  MPI_Comm_size(MPI_COMM_WORLD, &number_processes);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
}

void MPImanager::set_size(size_t size) {
  this->size = size;

  if ((int)size < number_processes) {
    MPI_Group world_group;
    MPI_Comm_group(MPI_COMM_WORLD, &world_group);

    MPI_Group new_group;
    int ranges[3] = {(int)size, number_processes - 1, 1 };
    MPI_Group_range_excl(world_group, 1, &ranges, &new_group);

    MPI_Comm_create(MPI_COMM_WORLD, new_group, &world);

    used_processes = size;
  } else {
    world = MPI_COMM_WORLD;
    used_processes = number_processes;
  }

  size_t lower = lower_bound(), upper = upper_bound();
```

```
  sub_matrices = alloc3d(SOLUTIONS_NR, (NUMBER_TIME_STEPS - 1), (upper - lower
      + 1));


}

MPI_Comm MPImanager::get_world() {
  return world;
}

bool MPImanager::part_of_world() {
  return rank < (int)used_processes;
}

void MPImanager::finalize() {
  free(sub_matrices);
  MPI_Finalize();
}

int MPImanager::get_rank() {
  return rank;
}

int MPImanager::get_number_processes() {
  return used_processes;
}

bool MPImanager::is_root() {
  return rank == 0;
}

bool MPImanager::is_last() {
  return rank == used_processes - 1;
}

bool MPImanager::one_process() {
  return 0 == used_processes;
```

```
}

size_t MPImanager::lower_bound() {
  return rank * size / used_processes;
}

size_t MPImanager::upper_bound() {
  return (rank + 1) * size / used_processes - 1;
}

void MPImanager::collect_results(vector<Method*> &solutions) {

  for (int p = 0; p < used_processes; p++) {
    size_t lower = p * size / used_processes, upper = (p + 1) * size /
        used_processes - 1;
    size_t count = solutions.size() * (NUMBER_TIME_STEPS - 1) * (upper - lower
        + 1);
    double buffer[count];

    if (p != 0) {
      MPI_Recv(buffer, count, MPI_DOUBLE, p, p, world, MPI_STATUS_IGNORE);
    }

    for (size_t i = 0; i < solutions.size(); i++) {
      for (size_t j = 0; j < (size_t)NUMBER_TIME_STEPS - 1; j++) {
        for (size_t k = 0; k <= upper - lower; k++) {
          solutions[i]->set_value(j + 1, k + lower + 1, p != 0 ? buffer[i *
              ((size_t)NUMBER_TIME_STEPS - 1) * (upper - lower + 1) + j * (upper
              - lower + 1) + k] : sub_matrices[i][j][k]);
        }
      }
    }

  }
}

void MPImanager::send_results() {
```

```
  size_t lower = lower_bound(), upper = upper_bound();
  size_t count = SOLUTIONS_NR * ((size_t)NUMBER_TIME_STEPS - 1) * (upper -
      lower + 1);
  double buffer[count];

  for (size_t i = 0; i < SOLUTIONS_NR; i++) {
    for (size_t j = 0; j < (size_t)NUMBER_TIME_STEPS - 1; j++) {
      for (size_t k = 0; k <= upper - lower; k++) {
        buffer[i * ((size_t)NUMBER_TIME_STEPS - 1) * (upper - lower + 1) + j *
            (upper - lower + 1) + k] = sub_matrices[i][j][k];
      }
    }
  }

  MPI_Send(buffer, count, MPI_DOUBLE, 0, rank, world);
}

void MPImanager::add_sub_matrix(size_t i, double ** sub_matrix) {
  sub_matrices[i] = sub_matrix;
}
```

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();

  void set_value(size_t i, size_t j, double value);
};

#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);
  }
}

void Problem::set_value(size_t i, size_t j, double value) {
  if (NUMBER_TIME_STEPS <= i) {
    std::cout << "i = " << i << " and size = " << NUMBER_TIME_STEPS <<
        std::endl;
  } else if (x_size + 1 <= j) {
    std::cout << "j = " << j << " and size = " << x_size + 1 << std::endl;
  } else {
    solution[i][j] = value;
  }
}

//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[2] = { 0.1, 0.001 }; /**< Macro double. The default time
    step. */
const double DELTA_X[2] = { 0.5, 0.005 }; /**< 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. */

const size_t SOLUTIONS_NR = 4;
```

```
double ***alloc3d(int l, int m, int n);
double **alloc2d(int rows, int cols);
double * gaussian_elimination(double ** &a, double * &b, size_t n);

#endif
```

methods/analytical.h

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

#include "method.h" // inheritance
#include "../mpi/mpimanager.h"

/**
*  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(MPImanager *mpi_manager, size_t index);
};

#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(MPImanager *mpi_manager, size_t index) {
  Vector t_values = problem.get_tvalues();
  Vector x_values = problem.get_xvalues();
  size_t lower = mpi_manager->lower_bound(), upper = mpi_manager->upper_bound();

  double** sub_matrix = alloc2d(NUMBER_TIME_STEPS - 1, upper - lower + 1);
  lower++; upper++;

  // iterates through the solution columns
  for (unsigned int t = 1; t < NUMBER_TIME_STEPS; t++) {
```

```
    // iterates through the solution rows
    double * current_step = (double*) malloc(((upper - lower) + 1) *
        sizeof(double));
    for (unsigned int x = lower; x <= upper; 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
      current_step[x - lower] = SURFACE_TEMPERATURE + 2.0 *
          (INITIAL_TEMPERATURE - SURFACE_TEMPERATURE) * sum;
    }
    sub_matrix[t - 1] = current_step;
  }

  mpi_manager->add_sub_matrix(index, sub_matrix);
}
```

methods/method.h

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

#include "../variants/problem.h" // declare the problem structure
#include <mpi.h>
#include <vector>

class MPImanager;

using namespace std;


/**
```

```
*  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. */
  bool last_iteration;

  bool lapacke;

  MPI_Request requests[4];
  bool request_status[4];
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(MPImanager *mpi_manager, MPI_Comm world, size_t index);

  void set_value(size_t i, size_t j, double value);

  void compute_norms(Matrix analytical_matrix);

  void enable_lapacke();

  bool is_lapacke();

  // PUBLIC MUTATOR METHODS

  /**
  * A pure virtual member.
  * compute the solution following the rules of a given method.
  */
  virtual void compute_solution(MPImanager *mpi_manager, size_t index) = 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;
  this->last_iteration = false;
  this->lapacke = false;

  for (size_t i = 0; i < 4; i++)
    this->request_status[i] = false;
}

/*
 * public method - compute a solution keeping track of spent time
 */
void Method::compute(MPImanager *mpi_manager, MPI_Comm world, size_t index) {
  double start = 0.0, end = 0.0;

  MPI_Barrier(world);
  start = MPI_Wtime();

  compute_solution(mpi_manager, index);

  MPI_Barrier(world);
```

```
  end = MPI_Wtime();

  computational_time = double(end - start);
}

/*
 * 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;
}

void Method::set_value(size_t i, size_t j, double value) {
  problem.set_value(i, j, value);
}

void Method::enable_lapacke() {
  this->lapacke = true;
```

```
}


bool Method::is_lapacke() {
  return lapacke;
}
```

methods/explicit/explicit.h

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

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

/**
*  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 double* build_iteration(MPImanager *mpi_manager, double*
      previous_step) = 0;
  void wait(MPImanager * mpi_manager, size_t i);
  void exchange_data(MPImanager * mpi_manager, size_t i, double * result);

  size_t upper, lower, size;
  double back, forward;
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(MPImanager *mpi_manager, size_t index);
};

#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(MPImanager *mpi_manager, size_t index) {
  size_t t_size = problem.get_tsize();
  upper = mpi_manager->upper_bound(), lower = mpi_manager->lower_bound();
  back = mpi_manager->is_root() ? SURFACE_TEMPERATURE : INITIAL_TEMPERATURE,
      forward = mpi_manager->is_last() ? SURFACE_TEMPERATURE :
      INITIAL_TEMPERATURE;

  size = upper - lower;
  Vector t_values = problem.get_tvalues();

  double delta_t = problem.get_deltat(), time;
  double * current_step = (double*) malloc((size + 1) * sizeof(double)), *
      previous_step = (double*) malloc((size + 1) * sizeof(double));


  double ** sub_matrices = alloc2d(NUMBER_TIME_STEPS - 1, size + 1);
  // iterate through the several time steps
  for (size_t i = 1; i <= t_size; i++) {
    // if is the first iteration then the previous step is known (initial
        conditions)
    if (i == 1) {
      for (size_t t = 0; t <= size; t++) {
        previous_step[t] = INITIAL_TEMPERATURE;
      }
    } else if (i == t_size) {
      last_iteration = true;
    }
    // use the current and previous time steps to calculate the next time step
        solution
```

```
    current_step = build_iteration(mpi_manager, previous_step);
    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
    int position = t_values.find(time);
    if (position != -1) {
      sub_matrices[position - 1] = current_step;
    }
  }


  mpi_manager->add_sub_matrix(index, sub_matrices);
}


void Explicit::wait(MPImanager * mpi_manager, size_t i) {
  int rank = mpi_manager->get_rank();
  MPI_Status status;

  if (i == 0) {
    if (!mpi_manager->is_root() && request_status[0]) {
      MPI_Wait(&requests[0], &status);
    }
    if (!mpi_manager->is_last() && request_status[1]) {
      MPI_Wait(&requests[1], &status);
    }
  }

  if (i == size) {
    if (!mpi_manager->is_last() && request_status[2]) {
      MPI_Wait(&requests[2], &status);
    }
    if (!mpi_manager->is_root() && request_status[3]) {
      MPI_Wait(&requests[3], &status);
    }
```

```
  }
}

void Explicit::exchange_data(MPImanager *mpi_manager, size_t i, double *
    result) {
  if (mpi_manager->one_process()) return;
  int rank = mpi_manager->get_rank();
  MPI_Comm world = mpi_manager->get_world();
  if (i == 0) {
    if (!mpi_manager->is_root()) {
      MPI_Isend(&result[i], 1, MPI_DOUBLE, rank - 1, 0, world, &requests[0]);
      if (!request_status[0]) request_status[0] = true;
    }
    if (!mpi_manager->is_last()) {
      MPI_Irecv(&forward, 1, MPI_DOUBLE, rank + 1, 0, world, &requests[1]);
      if (!request_status[1]) request_status[1] = true;
    }
  }
  if (i == size) {
    if (!mpi_manager->is_last()) {
      MPI_Isend(&result[i], 1, MPI_DOUBLE, rank + 1, 0, world, &requests[2]);
      if (!request_status[2]) request_status[2] = true;
    }
    if (!mpi_manager->is_root()) {
      MPI_Irecv(&back, 1, MPI_DOUBLE, rank - 1, 0, world, &requests[3]);
      if (!request_status[3]) request_status[3] = true;
    }
  }
}
```

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.
  */
  double* build_iteration(MPImanager *mpi_manager, double* 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
*/
double* FTCS::build_iteration(MPImanager *mpi_manager, double* previous_step) {
  int rank = mpi_manager->get_rank();
  double * result = (double*) malloc((size + 1) * sizeof(double));
  for (size_t i = 0; i <= size; i++) {
    wait(mpi_manager, i);

    double back_space = i == 0 ? back : previous_step[i - 1],
    forward_space = i >= size ? forward : previous_step[i + 1];
    result[i] = previous_step[i] + q / 2.0 * (forward_space - 2.0 *
        previous_step[i] + back_space);

    if (!last_iteration) exchange_data(mpi_manager, i, result);
  }

  return result;
}
```

methods/implicit/implicit.h

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

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

```
/**
*  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.
  */
  double * thomas_algorithm(double * r, double a, double b, double c);
  void calculate_spikes(MPImanager * mpi_manager);
  double * spikes_algorithm(MPImanager * mpi_manager, double * y, double * x);
  double * exchange_data(MPImanager *mpi_manager, double &head, double &tail);
  void solve_with_lapacke(double * &b);
  double ** spike_matrix;
  double * v, * w;
  double ** a_matrix;
  int * receive_pos, * receive_count;
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 double * build_r(MPImanager * mpi_manager, double * previous_step) =
      0;

  size_t upper, lower, size;
  double back, forward;
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(MPImanager *mpi_manager, size_t index);
};

#endif
```

methods/implicit/implicit.cpp

```
#include "implicit.h"

extern "C" {
     void dgttrs_(char * trans, int *n, int *nrhs, double *dl, double *d,
          double *du, double *du2, int *ipivot, double *b, int *ldb, int *info)
              ;
     void dgttrf_(int * n, double *low, double *diag, double *up1,
              double *up2, int *ipivot, int *info);
}

// 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(MPImanager *mpi_manager, size_t index) {
  Vector t_values = problem.get_tvalues();
  size_t t_size = problem.get_tsize();

  upper = mpi_manager->upper_bound(), lower = mpi_manager->lower_bound();
  size = upper - lower;

  double delta_t = problem.get_deltat(), time;
  double * current_step = (double*) malloc((size + 1) * sizeof(double)), *
      previous_step = (double*) malloc((size + 1) * sizeof(double)),
  *r, *x;
```

```
back = mpi_manager->is_root() ? SURFACE_TEMPERATURE : INITIAL_TEMPERATURE,
    forward = mpi_manager->is_last() ? SURFACE_TEMPERATURE :
    INITIAL_TEMPERATURE;
v = (double*) malloc((size + 1) * sizeof(double)), w = (double*) malloc((size
    + 1) * sizeof(double));
std::fill_n(&v[0], size + 1, 0.0);
std::fill_n(&w[0], size + 1, 0.0);

v[size] = -q;
w[0] = -q;

if (!lapacke) {
  v = thomas_algorithm(v, -q, (1.0 + 2.0 * q), -q);
  w = thomas_algorithm(w, -q, (1.0 + 2.0 * q), -q);
} else {
  solve_with_lapacke(v);
  solve_with_lapacke(w);
}

double ** sub_matrices = alloc2d(NUMBER_TIME_STEPS - 1, size + 1);

if (mpi_manager->is_root() && !mpi_manager->one_process()) {
  calculate_spikes(mpi_manager);
}

// iterate through the several time steps
for (size_t i = 1; i <= t_size; i++) {
  // if is the first iteration then the previous step is known (initial
     conditions)
  if (i == 1) {
    for (size_t t = 0; t <= size; t++) {
      previous_step[t] = INITIAL_TEMPERATURE;
    }
  }

  // build r vector
```

```
    r = build_r(mpi_manager, previous_step);

    // use the r vector to calculate the current time step solution
    if (!lapacke) {
      r = thomas_algorithm(r, -q, (1.0 + 2.0 * q), -q);
    } else {
      solve_with_lapacke(r);
    }

    x = exchange_data(mpi_manager, r[0], r[size]);

    if (x != NULL) {
      current_step = spikes_algorithm(mpi_manager, r, x);
    } else {
      current_step = r;
    }

    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
    int position = t_values.find(time);
    if (position != -1) {
      sub_matrices[position - 1] = current_step;
    }

  }

  mpi_manager->add_sub_matrix(index, sub_matrices);
}

/*
* private normal method - thomas algorithm to compute a given time step solution
*/
double * Implicit::thomas_algorithm(double * r, double a, double b, double c) {
  double * x = (double*) malloc((size + 1) * sizeof(double)), y[size], p[size +
      1];
```

```
  // building the y and p vectors (forward phase)
  for (size_t i = 0; i <= size; i++) {
    if (i == 0) {
      y[i] = c / b;
      p[i] = r[i] / b;
    } else if (i == size) {
      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)
  for (int i = size; i >= 0; i--) {
    x[i] = (size_t)i == size ? p[i] : p[i] - y[i] * x[i + 1];
  }
  return x;
}

void Implicit::calculate_spikes(MPImanager * mpi_manager) {
  size_t y_size = mpi_manager->get_number_processes() * 2, npes =
      mpi_manager->get_number_processes();
  receive_count = (int*) malloc(npes * sizeof(int));
  receive_pos = (int*) malloc(npes * sizeof(int));

  std::fill_n(&receive_count[0], npes, 2);

  for (size_t i = 0; i < npes; i++) {
    receive_pos[i] = i * 2;
  }


  spike_matrix = alloc2d(y_size, y_size);
  std::fill_n(&spike_matrix[0][0], y_size * y_size, 0.0);

  for (size_t i = 0; i < y_size; i++) {
    spike_matrix[i][i] = 1.0;
```

```
    if (i != 0 && i != y_size - 1) {
      if (i % 2 == 0) {
        spike_matrix[i - 2][i] = v[0];
        spike_matrix[i - 1][i] = v[size];
      } else {
        spike_matrix[i + 2][i] = w[size];
        spike_matrix[i + 1][i] = w[0];
      }
    }
  }

}

void Implicit::solve_with_lapacke(double * &b) {
  int n, nrhs, ldb, info = 0;
  n = size + 1; nrhs = 1;
  ldb = n;

  int ipiv1[n]; double du2[n], dl[n], d[n], du[n];
  fill_n(&dl[0], n, -q); fill_n(&du[0], n, -q); fill_n(&d[0], n, (1.0 + 2.0 *
     q));

  char trans = 'N';

  dgttrf_(&n, dl, d, du, du2, ipiv1, &info);
  dgttrs_(&trans, &n, &nrhs, dl, d, du, du2, ipiv1, &b[0], &ldb, &info);

  if (info != 0) {
    cout << "LAPACKE: something wrong happened." << endl;
  }

}

double * Implicit::spikes_algorithm(MPImanager *mpi_manager, double * y, double
    * x) {
  double * current_step = (double *) malloc(sizeof(double) * (size + 1));
  size_t y_size = mpi_manager->get_number_processes() * 2;
```

```
  size_t s_index = mpi_manager->get_rank() * 2;


  for (size_t i = 0; i <= size; i++) {
    if (mpi_manager->is_root()) {
      current_step[i] = y[i] - v[i] * x[2];
    } else if (mpi_manager->is_last()) {
      current_step[i] = y[i] - w[i] * x[y_size - 3];
    } else {
      current_step[i] = y[i] - v[i] * x[s_index + 2] - w[i] * x[s_index - 1];
    }
  }
  return current_step;
}


double * Implicit::exchange_data(MPImanager *mpi_manager, double &head, double
    &tail) {
  if (mpi_manager->one_process()) return NULL;

  MPI_Comm world = mpi_manager->get_world();
  size_t y_size = mpi_manager->get_number_processes() * 2;
  double * y = (double *) malloc(sizeof(double) * y_size), g[2] = { head, tail
      }, *x = (double *) malloc(sizeof(double) * y_size);

  MPI_Gatherv(g, 2, MPI_DOUBLE, &y[0], receive_count, receive_pos, MPI_DOUBLE,
      0, world);

  if (mpi_manager->is_root()) {
    x = gaussian_elimination(spike_matrix, y, y_size);
  }

  MPI_Bcast(&x[0], y_size, MPI_DOUBLE, 0, world);

  size_t index = mpi_manager->get_rank() * 2;
  if (!mpi_manager->is_root()) {
    back = x[index - 1];
```

```
  }
  if (!mpi_manager->is_last()) {
    forward = x[index + 2];
  }
  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
  */
  double * build_r(MPImanager * mpi_manager, double * 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
*/
double * CrankNicolson::build_r(MPImanager * mpi_manager, double *
    previous_step) {

  double * r = (double*) malloc((size + 1) * sizeof(double));
  for (size_t i = 0; i <= size; i++) {
    if (i == 0 && mpi_manager->is_root()) {
      r[i] = previous_step[i] + q * (2 * SURFACE_TEMPERATURE - 2.0 *
          previous_step[i] + previous_step[i + 1]);
    } else if (i == size && mpi_manager->is_last()) {
```

```
      r[i] = previous_step[i] + q * (2 * SURFACE_TEMPERATURE + previous_step[i
          - 1] - 2.0 * previous_step[i]);
    } else if (i == 0) {
      r[i] = previous_step[i] + q * (back - 2.0 * previous_step[i] +
          previous_step[i + 1]);
    } else if (i == size) {
      r[i] = previous_step[i] + q * (previous_step[i - 1] - 2.0 *
          previous_step[i] + forward);
    } else {
      r[i] = previous_step[i] + q * (previous_step[i - 1] - 2.0 *
          previous_step[i] + previous_step[i + 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
  */
  double * build_r(MPImanager * mpi_manager, double * 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
*/
double * Laasonen::build_r(MPImanager * mpi_manager, double * previous_step) {
```

```
  double * r = (double*) malloc((size + 1) * sizeof(double));
  for (size_t i = 0; i <= size; i++) {
    if (i == 0 && mpi_manager->is_root()) {
      r[i] = q * SURFACE_TEMPERATURE + previous_step[i];
    } else if (i == size && mpi_manager->is_last()) {
      r[i] = q * SURFACE_TEMPERATURE + previous_step[i];
    } else {
      r[i] = previous_step[i];
    }
  }
  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, int number_processes);

void plot_times(std::string output_name, Method * analytical,
    std::vector<Method*> methods, int number_processes);

/**
* 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);

void export_csv(std::string output_name, std::vector<Method*> methods, int
    number_processes);

// 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, int
      number_processes);


  void export_analytical(Method * analytical, int number_processes);
};

#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, int number_processes) {
  //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 = double_to_string(3, methods[index]->get_deltat());
    output_name = output_path + '/' + name;
    output_name += "dt=" + deltat_string;
    std::cout << "Exporting " << name << " method outputs... ";
    plot_solutions(output_name, analytical, methods[index], number_processes);
    std::cout << "Finished!" << std::endl;
  }
  plot_times(output_path, analytical, methods, number_processes);
  export_csv(output_path, methods, number_processes);
  //std::vector<Method*> error_vector(methods.begin() + 1, methods.begin() + 4);
  //error_tables(output_name, error_vector);
}
/*
* 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, int number_processes) {
  // 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();
  std::string lapacke_str = method->is_lapacke() ? "_lapacke" : "";
  // 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 << "p=" << number_processes << lapacke_str << ".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 0.5 lw 1" << std::endl;
  }
}

void IOManager::plot_times(std::string output_name, Method * analytical,
    std::vector<Method*> methods, int number_processes) {
  Gnuplot gp;
  std::vector<double> times;
  times.push_back(analytical->get_computational_time());

  for (unsigned int i = 0; i < methods.size(); i++) {
    times.push_back(methods[i]->get_computational_time());
  }

  std::string lapacke_str = methods[1]->is_lapacke() ? "_lapacke" : "";
```

```
  std::string deltat_string = double_to_string(3, methods[0]->get_deltat());

  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 \"time [s]\";set xlabel
      \"\"; set term png; set xtics (\"Analytical\" 0, \"Laasonen\" 1, \"Crank
      Nicholson\" 2, \"FTCS\" 3)\n";
  gp << "set output \"" << output_path << "/timesdt=" << deltat_string << "p="
      << number_processes << lapacke_str << ".png\";\n";
  gp << "plot" << gp.file1d(times) << " notitle with linespoint ls 1" <<
      std::endl;
}

void IOManager::export_analytical(Method * analytical, int number_processes) {
  // Object to export plots
  Gnuplot gp;

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

  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 \"./outputs/analytical" + time_str;
    gp << "p=" << number_processes << ".png\";\n";
    gp << "plot" << gp.file1d(analytical_matrix[index]) << "with lines title
       \"Analytical\" lw 2 lt rgb \"red\""
      << std::endl;
  }
}
```

```
void IOManager::export_csv(std::string output_name, std::vector<Method*>
    methods, int number_processes) {
  double time = methods[0]->get_deltat();
  std::string time_str = double_to_string(3, time);
  std::string lapacke_str = methods[1]->is_lapacke() ? "_lapacke" : "";
  std::ofstream out;
  for (size_t i = 0; i < methods.size(); i++) {
    std::string name = output_name + "/csvs/" + methods[i]->get_name() + "dt="
        + time_str + lapacke_str + ".csv";
    out.open(name, std::ios_base::app);
    out << number_processes << " " << methods[i]->get_computational_time() <<
        std::endl;
    out.close();
  }
}

/*
* 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 (unsigned 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 : public 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 {
  size_t n = size();
  double result = 0.0;
  for (size_t index = 0; index < n; index++) {
    result += std::abs((*this)[index]);
  }
  return result;
}

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

```
/*
* uniform (infinity) norm
*/
double Vector::uniform_norm() const
{
  size_t n = size();
  double result = -DBL_MAX;
  for (size_t 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 : public 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) {
  size_t vector_size = v.getSize();

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

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

// NORMS
/*
 * 1 norm
 */
double Matrix::one_norm() const
{
  size_t nrows = getNrows();
  size_t ncols = getNcols();
  Matrix transpose = this->transpose();
  Vector * sizes = new Vector(nrows);
  for (size_t 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
{
  size_t nrows = getNrows();
  size_t ncols = getNcols();

  double result = 0.0;
  for (size_t i = 0; i < nrows; i++) {
    for (size_t j = 0; j < ncols; j++) {
      result += pow((*this)[i][j], 2);
    }
  }
  return sqrt(result);
}

/*
 * uniform (infinity) norm
 */
double Matrix::uniform_norm() const
{
  size_t nrows = getNrows();
  size_t ncols = getNcols();
  Vector * sizes = new Vector(nrows);
  for (size_t 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;
}
```