SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Heat Equation

Report for the course of
High Performance Scientific Computing in Aerospace

Author: **Giorgio Miani**

Student ID: 10685401
Advisor: Prof. Franco Auteri
Academic Year: 2023-24

Contents

1 | Introduction

Consider the following problem: we must control a satellite in order to land it on an asteroid and to drill its surface. To do this, we need an infra-red camera that takes a picture of the asteroid, and we have to compare it with a synthetic image simulated, in order to choose where to safely land on the asteroid. How can we generate such images artificially? We do that by understanding how the asteroid would irradiate infra-red light, know how the camera captures the infra-red light and reconstruct everything, hence our final aim is to reconstruct the thermal map of the asteroid.
When solving a mathematical problem in scientific computing there are some crucial steps to follow in order to reach a reliable solution in the fastest way:

- **Mathematical formulation of the problem**: find a set of equations that can describe my problem;
- **Approximate study to evaluate**:
  (a) `Scale of quantities involved`: e.g. how much time does it take to solve the problem?
  (b) `Any simplifications one could make to the problem`: solve the problem in the easiest way to save time and money;
- **Discretisation of the problem**: both spatial and temporal discretisation using different techniques;
- **Implementation of a program**;
- **Validation and profiling of the program**: look for bugs, errors and possible optimizations;
- **Use the application to solve the problem**.

Note that the validation phase is often the most troublesome since we may have very subtle and hidden bugs.
In the following chapters we will analyze the proccess that brought to the resolution of the problem, starting from its mathematical formulation with the appropriate simplifications and looking at different methods of discretisation, then we will see the code implementations

of the resolution, from the simplest case to the 3D complete and parallelized program, and finally we will check some results of the simulations.

2 | Mathematical Formulation

Let's start from the principles: we need a function that describes the temperature of the asteroid, therefore we need the **heat equation**.
We have:

- heat exchanged Q
- specific heat of the asteroid c

And there is the **energy conservation equation**:

$$de = Tds - pdv \tag{2.1}$$

We also know that:

$$dh = de + pdv + vdp \tag{2.2}$$

So we get that

$$de + pdv = dh - vdp \tag{2.3}$$

$$\partial q = dh - vdp \tag{2.4}$$

Since the asteroid may change its volume due to the heat exchange with the sun and there is no constraint that makes its pressure change, we will consider the specific heat at constant pressure c_p. Hence the second right term of the equation 2.4 simplifies:

$$\partial q = dh \tag{2.5}$$

Now we calculate the entalpy on a small volume of our problem:

$$dh = \frac{d}{dt}\int_V \rho c_p T dV \tag{2.6}$$

From the previous equation we know that the variation in enthalpy is equal to the heat exchanged, thus the flux of heat through the surface of the volume becomes:

$$dh = \frac{d}{dt}\int_V \rho c_p T dV = \oint_S -\mathbf{q}\cdot\mathbf{n} dS + \int_V f dV \tag{2.7}$$

Where f in as external heat source which we will neglect for now, but can turn out to be useful in the future for controlling the results.
Fourier law tells us that

$$q = -k\nabla\mathbf{T} \tag{2.8}$$

Merging everything we get the heat equation:

$$\frac{d}{dt}\int_V \rho c_p T dV = \oint_S k\nabla\mathbf{T}\cdot\mathbf{n}dS + \int_V f dV \tag{2.9}$$

After this we can transform everything into volume integrals using divergence theorem and obtain:

$$\frac{d}{dt}\int_V \rho c_p T dV = \int_V \nabla\cdot(k\nabla\mathbf{T})dV + \int_V f dV \tag{2.10}$$

Which can be rewritten as:

$$\int_V (\frac{\partial}{\partial t}(\rho c_p T) - \nabla\cdot(k\nabla\mathbf{T}) - f)dV \tag{2.11}$$

Generally is not true that if an integral is equal to zero, the function integrated is zero too, expect for one case when we the volume is an arbitrary volume which is our case.
Finally we can write our heat equation as:

$$\frac{\partial(\rho c_p T)}{\partial t} - \nabla\cdot(k\nabla\mathbf{T}) - f = 0 \tag{2.12}$$

Now we just need to define the boundary conditions. Irradiation depends on the incident angle, the material and finally the difference in temperature between the two bodies, it is an hard non linear problem but we will simplify it to better solve it.

2.1. Approximate Study

We consider the asteroid with constant coefficents in a simple box. We assume to know the boundary temperature T_B, thus we can write the equation as:

$$\begin{cases} \rho c_p \dfrac{\partial T}{\partial t} - k\nabla^2 T - f = 0 \\ T_s = T_B \end{cases} \tag{2.13}$$

However, we are dealing with a dimensional problem which is not good to see the properties of the problem.
So we need to remove the dimensionalities, but how do we do that?

The first step is to adimensionalise constants:

$$\frac{\partial T}{\partial t} - \kappa \nabla^2 T = \bar{f} \tag{2.14}$$

Where $\kappa = \frac{k}{\rho c_p}$ and $\bar{f} = \frac{f}{\rho c_p}$.
Then we have to adimensionalise time, temperature and space, therefore we define:

$$t = t^* \bar{t},\ T = T^* \bar{T},\ x = L\bar{x},\ y = L\bar{y},\ z = L\bar{z}$$

We substitute these variables in our heat equation and we get:

$$\frac{T^*}{t^*}\frac{\partial \bar{T}}{\partial \bar{t}} - \kappa \frac{T^*}{L^2} \nabla^2 \bar{T} = \bar{f} \tag{2.15}$$

We can finally rewrite everything as:

$$\frac{\partial \bar{T}}{\partial \bar{t}} - \frac{\kappa t^*}{L^2} \nabla^2 \bar{T} = \frac{t^*}{T^*} \bar{f} \tag{2.16}$$

In this way when solving a problem we can choose L such that $\frac{\kappa t^*}{L^2} = 1$:

$$\frac{\partial \bar{T}}{\partial \bar{t}} - \nabla^2 \bar{T} = \bar{f} \tag{2.17}$$

2.2. Spatial Discretization

Now that our mathematical formulation is complete we need to discretize our model in order to be able to solve it numerically. In this chapter we will see how to discretize its spatial domain.

There are different techniques of discretization, each one differs from the others in accuracy, simplicity and flessibility, here below there is a summary of the most common ones:

Method	Pros
Finite Difference Method	Easy to implement, fast
Finite Volume Method	Geometric flessibility
Finite Element Method	General, Geometric flessibility
Spectral Methods	High accuracy
Spectral Element Methods	General, Geometric flessibility, Accuracy

Table 2.1: Pros of Spatial Discretization Methods

Method	Cons
Finite Difference Method	Accuracy, Simple geometries, Complicated mathematic
Finite Volume Method	Complicated mathematic, Hard to implement
Finite Element Method	Slow, Hard to implement
Spectral Methods	Simple domains
Spectral Element Methods	Slow, Hard to implement

Table 2.2: Cons of Spatial Discretization Methods

We decide to use the Finite Difference Methods thanks to its simplicity and speed.
This method approximates derivatives with finite differences, hence we divide the domain into a finite number of elements, where my solution si defined, and we use a polynomial function to interpolate and calculate the differences. In this way for example I can approximate a derivative using a straight line between two points.
Depending on the degree of the polynomial the method has different order of accuracy which can be calculated through Taylor expansion, for example using three points to interpolate and approximate a first derivative produces an accuracy of the second order.
Moreover before implementing we have to verify that the norm of A_h^{-1} is less than a constant C to ensure that the method is stable and consistent.
If we consider the 1D Time Independent case:

$$-\frac{d^2T}{dx^2} = f(x) \tag{2.18}$$

we can approximate the secord order derivative with a parable in the following way:

$$-\frac{T_{i-1} - 2T_i + T_{i+1}}{\Delta x^2} = f_i \tag{2.19}$$

and prove that we get an accuracy of the second order.

2.3. Temporal Discretization

2.3.1. MultiPoints Methods

Now that we have discretized the spatial domain we need to discretize the temporal domain, too. For the temporal Discretization we used the Linear MultiPoints Methods family:

$$\begin{aligned} \sum_{i=0}^{r} a_i u^{n+i} = \Delta t; \qquad & \sum_{i=0}^{r} b_i f(t_{n+i}\Delta t, u^{n+i}) \\ \rho(\xi) = \sum_{i=0}^{r} a_i \xi^i; \qquad & \sigma(\xi) = \sum_{i=0}^{r} b_i \xi^i \end{aligned} \tag{2.20}$$

Where r is the order of the method, a_i and b_i are the coefficients of the method.
This family of methods is based on the idea of using multiple points in time to approximate the solution of the problem.
The main advantage of this method is that it is possible to use a higher order of accuracy in time without increasing the computational cost.
The main disadvantage is that the method is implicit and so we need to solve a linear system at each time step.
Based on the value of a_i and b_i we have two main families:

- **Adams Methods**: $a_r = 1$, $a_{r-1} = -1$, $a_i = 0$ for i <r-1;
- **BDF**: b_i =0 for $i \neq$r, b_r =1;

Here we won't go into the details of each method, but it is worth of mention the Explicit Euler method:

$$\frac{u^{n+1} - u^n}{\Delta t} = f(t_n, u^n) \tag{2.21}$$

and also Cranck-Nicolson method, which is an Adam method with $r = 1$ and $b_1 = b_0 = \frac{1}{2}$:

$$\frac{u^{n+1} - u^n}{\Delta t} = \frac{1}{2} f(t_{n+1} + \Delta t, u^{n+1}) + \frac{1}{2} f(t_n + \Delta t, u^n) \tag{2.22}$$

2.3.2. Zero Stability and Absolute Stability

Then we define a parameter to evaluate each method and choose the best one for our problem. First we consider the zero stability for the LMMs methods as:

$$\rho(\xi) = 0 => |\xi| \leq 1 \tag{2.23}$$

Hence, we know all Adam methods work for a sufficiently-small Δt; however, zero-stability is not enough, we need a stronger condition, called absolute stability, which is defined in the following way:

a LMM is A-stable if the roots of the following equation

$$\rho(\xi) - z\sigma(\xi) = 0 \tag{2.24}$$

where z $= \Delta t\lambda < 0$, are $|\xi| \leq 1$.
If we take the 1D Time Dependent equation:

$$\frac{\partial T}{\partial t} - \frac{\partial^2 T}{\partial x^2} = f(x,t) \tag{2.25}$$

and we discretize it with Explicit Euler (2.21) and using the second order Finite Difference Method for the spatial domain, we get:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} - \frac{T_{i-1}^n - 2T_i^n + T_{i+1}^n}{\Delta x^2} = f_i^n \tag{2.26}$$

Where i is the index for the spatial domain and n is the index for the temporal one.
If we apply the stability condition to this equation we get that it is stable if $\Delta t < \frac{\Delta x^2}{2}$, which is a very strong constraint, that's why explicit method are not usually used.
On the other hand if we apply the condition to the Cranck-Nicolson method we get that the method is A-stable and has an accuracy of the second order.

2.4. Complete Problem

Finally we considered the 3D problem with spatial and time discretization. First we applied Cranck-Nicolsol (2.22) for the temporal domain, since this method has better properties than Explicit Euler:

$$\frac{T^{n+1} - T^n}{\Delta t} - \bigtriangledown^2 \frac{T^{n+1} + T^n}{2} = f^{n+\frac{1}{2}} \tag{2.27}$$

And then we apply the Finite Difference Method of the second order for the spatial domain:

$$\begin{aligned} \frac{1}{\Delta t}T_{i,j,k}^{n+1} - \frac{1}{2\Delta x^2}(T_{i-1,j,k}^{n+1} + T_{i+1,j,k}^{n+1} + T_{i,j-1,k}^{n+1} + T_{i,j+1,k}^{n+1} + T_{i,j,k-1}^{n+1} + T_{i,j,k+1}^{n+1}) + \frac{6}{2\Delta x^2}T_{i,j,k}^{n+1} = \\ = \frac{1}{\Delta t}T_{i,j,k}^{n} + \frac{1}{\Delta x^2}(T_{i-1,j,k}^{n} + T_{i+1,j,k}^{n} + T_{i,j-1,k}^{n} + T_{i,j+1,k}^{n} + T_{i,j,k-1}^{n} + T_{i,j,k+1}^{n}) - \frac{6}{2\Delta x^2}T_{i,j,k}^{n} + f_{i,j,k}^{n+\frac{1}{2}} \end{aligned} \tag{2.28}$$

3 | Code Implementation

Here we present the implementation of the equations seen in the Mathematical formulation chapter, with their respective resolutions. We will start with the 1D case, introduce the temporal dependecy and then we will move to the 3D case. We will conclude with the parallelization of the code using different techniques.
Note that in each code we use functions to estimate the time of execution, in order to compare it with either the number of points, the language used on the parallelization or method adopted. Moreover we will build our problems starting from a manufactured solution in order to check the correctness of our code by comparing it with the computed one.

3.1. 1D Time-Independet Heat Equation

We start from the simpliest case, which is the 1-dimensional time independent Heat Equation (2.18).
We consider our domain of length $L = 1.0$ and we will use a grid of n points, so the step will be $\Delta x = \frac{L}{n+1}$.
We take the discretized equation seen in the chapters before:

$$-\frac{T_{i-1} - 2T_i + T_{i+1}}{\Delta x^2} = f_i \tag{3.1}$$

First we need to build our manufactured solution and we choose

$$T(x) = \sin(x) \tag{3.2}$$

and then we can calculate the source term as:

$$f(x) = -\sin(x) \tag{3.3}$$

Finally our equation becomes:

$$-\frac{T_{i-1} - 2T_i + T_{i+1}}{\Delta x^2} = -\sin(x) \tag{3.4}$$

Now we need to construct the matrix of the equation, we will use the following notation:

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & \cdots & 0 \\ a_2 & b_2 & c_2 & \ddots & & \vdots \\ 0 & a_3 & b_3 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & c_{n-2} & 0 \\ \vdots & & \ddots & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ \vdots \\ \vdots \\ T_n \end{bmatrix} = \begin{bmatrix} -\sin(x_1) \\ -\sin(x_2) \\ \vdots \\ \vdots \\ \vdots \\ -\sin(x_n) \end{bmatrix} \tag{3.5}$$

We get a tridiagonal matrix, with $a_i = c_i = \frac{1}{-\Delta x^2}$ area the upper and lower coefficents, and $b_i = \frac{2}{\Delta x^2}$ is the main diagonal.
So we save the coefficents in two variables a and b, we build the source term f and we add the boundary conditions $T_0 = \frac{\sin(0)}{\Delta x^2}$ and $T_n = \frac{\sin(1.0)}{\Delta x^2}$ to the forcing term at the boundaries. Then we start solving the system using thomas algorithm, which is a simplified version of the Gaussian Elimination that works well with tridiagonal matrices.

```
for(int i = 1; i < n; i++){
    w = a/b(i - 1);
    b(i) -= w*a;
    f(i) -= w*f(i-1);
}

solution(n - 1) = f(n - 1)/b(n-1);
for(int i = n - 2; i >= 0; i-- ){
    solution(i) = (f(i) - a*solution(i + 1))/b(i);
}
```

At the end we calculate the error by comparing the computed solution with the real one

$$error = \frac{(\sum_{i=1}^{n}(T_i - T_i^*)^2)^{\frac{1}{2}}}{n} \tag{3.6}$$

and we analyze how the error converges with the number of points and how the execution time scales.

3.2. 1D TD Heat Equation

Now we introduce the temporal dependecy, thus we use the equation (2.18). We will use the same domain and the same grid as before, but now we will add the $\Delta t = \frac{1.0}{Nt}$, where Nt

is the number of temporal steps. Note that, since we are using Explicit Euler, we have to choose $\Delta t < \frac{\Delta x^2}{2}$.
We get:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} - \frac{T_{i-1}^n - 2T_i^n + T_{i+1}^n}{\Delta x^2} = f_i^n \tag{3.7}$$

In this case we are using an explicit method, hence we just simply write

$$T_i^{n+1} = T_i^n + \frac{\Delta t}{\Delta x^2}(T_{i-1}^n - 2T_i^n + T_{i+1}^n) + \Delta t f_i^n \tag{3.8}$$

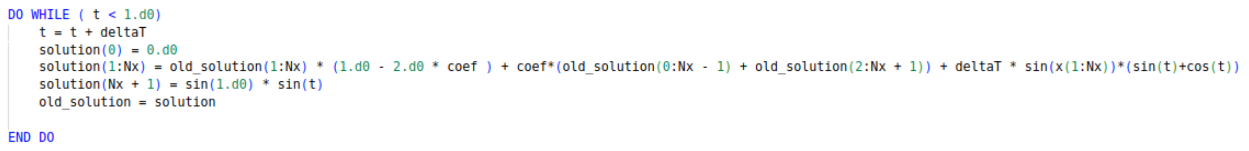
We choose as manufactured solution:

$$T(x, t) = \sin(x)\sin(t) \tag{3.9}$$

and we calculate the forcing term:

$$f(x, t) = \sin(x)(\cos(t) + \sin(t)) \tag{3.10}$$

We iterate over each time step and update the solution using the old one computed in the previous step.



```
DO WHILE ( t < 1.d0)
    t = t + deltaT
    solution(0) = 0.d0
    solution(1:Nx) = old_solution(1:Nx) * (1.d0 - 2.d0 * coef ) + coef*(old_solution(0:Nx - 1) + old_solution(2:Nx + 1)) + deltaT * sin(x(1:Nx))*(sin(t)+cos(t))
    solution(Nx + 1) = sin(1.d0) * sin(t)
    old_solution = solution

END DO
```

Figure 3.1: Implementation for the 1D TD Heat Equation.

At the end we compare and analyze the error to check the correctness of our code and how it behaves when increasing the number of points.

3.3. 3D TD Heat Equation

Then we move to the 3D Time-Dependent Equation, which is the one we are interested in. We take the equation 2.28 seen in the Complete Problem section and we implement it in three steps:

- We write the code to construct the matrix for the problem we are solving;
- We implement a first code to solve it;
- We try to improve our solver implementing Douglas method;

3.3.1. Matrix Constrution

Since now we are dealing with 3 dimensions we need to structure our matrix in a different way.
We will use three indices o, p and q to identify the position of each point in the grid and we will use a mapping function to map the 3D point to a 1D array, defined in the following way:

```
integer function findex(o,p,q, N)
    integer, intent (in) :: o, p, q , N
    findex = o + (p - 1) * N + (q - 1) * N * N
end function findex
```

We will show as an example how the matrix sparsity structure would look like for a $3 \times 3 \times 3$ grid.

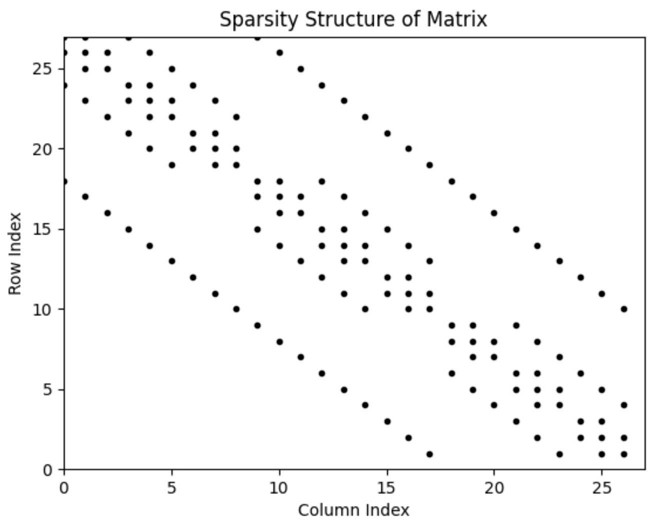


Figure 3.2: Sparsity structure of the 3D matrix.

3.3.2. Code

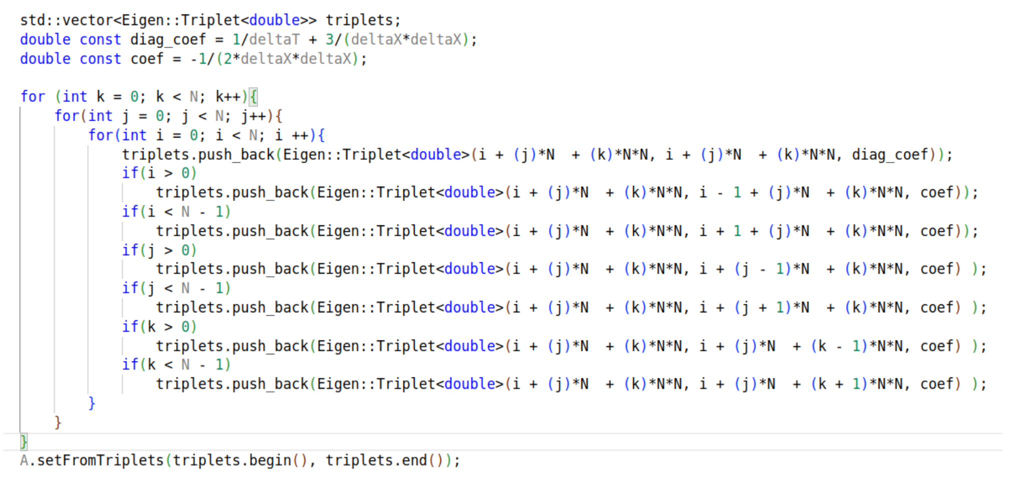
Then we will write the code to solve the problem, we will use the following manufactured solution:

$$T(x, y, z) = \sin(x)\sin(y)\sin(z)\sin(t) \tag{3.11}$$

Which produces the forcing term:

$$f(x, y, z, t) = \sin(x)\sin(y)\sin(z)(\cos(t) + 3\sin(t)) \tag{3.12}$$

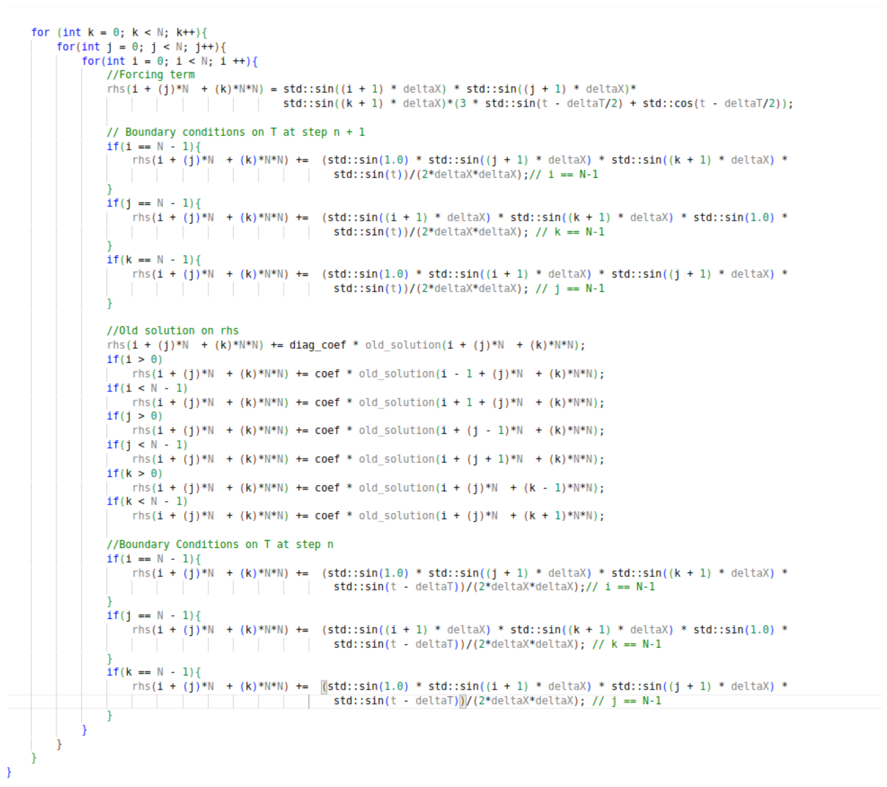
We implement the code in C++ using the Eigen library to handle the matrix and the linear system.
First we construct the matrix using Eigen Triplets to save memory and time:



```cpp
std::vector<Eigen::Triplet<double>> triplets;
double const diag_coef = 1/deltaT + 3/(deltaX*deltaX);
double const coef = -1/(2*deltaX*deltaX);

for (int k = 0; k < N; k++){
    for(int j = 0; j < N; j++){
        for(int i = 0; i < N; i ++){
            triplets.push_back(Eigen::Triplet<double>(i + (j)*N  + (k)*N*N, i + (j)*N  + (k)*N*N, diag_coef));
            if(i > 0)
                triplets.push_back(Eigen::Triplet<double>(i + (j)*N  + (k)*N*N, i - 1 + (j)*N  + (k)*N*N, coef));
            if(i < N - 1)
                triplets.push_back(Eigen::Triplet<double>(i + (j)*N  + (k)*N*N, i + 1 + (j)*N  + (k)*N*N, coef));
            if(j > 0)
                triplets.push_back(Eigen::Triplet<double>(i + (j)*N  + (k)*N*N, i + (j - 1)*N  + (k)*N*N, coef) );
            if(j < N - 1)
                triplets.push_back(Eigen::Triplet<double>(i + (j)*N  + (k)*N*N, i + (j + 1)*N  + (k)*N*N, coef) );
            if(k > 0)
                triplets.push_back(Eigen::Triplet<double>(i + (j)*N  + (k)*N*N, i + (j)*N  + (k - 1)*N*N, coef) );
            if(k < N - 1)
                triplets.push_back(Eigen::Triplet<double>(i + (j)*N  + (k)*N*N, i + (j)*N  + (k + 1)*N*N, coef) );
        }
    }
}
A.setFromTriplets(triplets.begin(), triplets.end());
```

Figure 3.3: Implementation for the construction of the matrix.

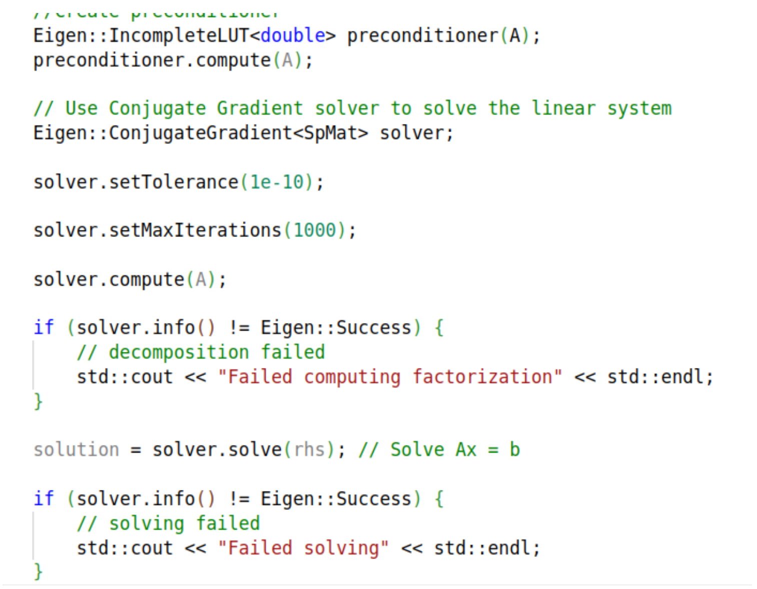
Where diag_coef are the diagonal coefficients which correspond to the coefficient of $T^{n+1}_{i,j,k}$, coef are the off-diagonal coefficients which correspond to the coefficient of $T^{n+1}_{i\pm1,j,k}$, $T^{n+1}_{i,j\pm1,k}$, $T^{n+1}_{i,j,k\pm1}$. Then we assemble the right hand side of the equation using the manufactured solution for the boundary conditions and the forcing term:



```cpp
for (int k = 0; k < N; k++){
    for(int j = 0; j < N; j++){
        for(int i = 0; i < N; i ++){
            //Forcing term
            rhs(i + (j)*N  + (k)*N*N) = std::sin((i + 1) * deltaX) * std::sin((j + 1) * deltaX)*
                                        std::sin((k + 1) * deltaX)*(3 * std::sin(t - deltaT/2) + std::cos(t - deltaT/2));

            // Boundary conditions on T at step n + 1
            if(i == N - 1){
                rhs(i + (j)*N  + (k)*N*N) +=  (std::sin(1.0) * std::sin((j + 1) * deltaX) * std::sin((k + 1) * deltaX) *
                                                std::sin(t))/(2*deltaX*deltaX);// i == N-1
            }
            if(j == N - 1){
                rhs(i + (j)*N  + (k)*N*N) +=  (std::sin((i + 1) * deltaX) * std::sin((k + 1) * deltaX) * std::sin(1.0) *
                                                std::sin(t))/(2*deltaX*deltaX); // k == N-1
            }
            if(k == N - 1){
                rhs(i + (j)*N  + (k)*N*N) +=  (std::sin(1.0) * std::sin((i + 1) * deltaX) * std::sin((j + 1) * deltaX) *
                                                std::sin(t))/(2*deltaX*deltaX); // j == N-1
            }

            //Old solution on rhs
            rhs(i + (j)*N  + (k)*N*N) += diag_coef * old_solution(i + (j)*N  + (k)*N*N);
            if(i > 0)
                rhs(i + (j)*N  + (k)*N*N) += coef * old_solution(i - 1 + (j)*N  + (k)*N*N);
            if(i < N - 1)
                rhs(i + (j)*N  + (k)*N*N) += coef * old_solution(i + 1 + (j)*N  + (k)*N*N);
            if(j > 0)
                rhs(i + (j)*N  + (k)*N*N) += coef * old_solution(i + (j - 1)*N  + (k)*N*N);
            if(j < N - 1)
                rhs(i + (j)*N  + (k)*N*N) += coef * old_solution(i + (j + 1)*N  + (k)*N*N);
            if(k > 0)
                rhs(i + (j)*N  + (k)*N*N) += coef * old_solution(i + (j)*N  + (k - 1)*N*N);
            if(k < N - 1)
                rhs(i + (j)*N  + (k)*N*N) += coef * old_solution(i + (j)*N  + (k + 1)*N*N);

            //Boundary Conditions on T at step n
            if(i == N - 1){
                rhs(i + (j)*N  + (k)*N*N) +=  (std::sin(1.0) * std::sin((j + 1) * deltaX) * std::sin((k + 1) * deltaX) *
                                                std::sin(t - deltaT))/(2*deltaX*deltaX);// i == N-1
            }
            if(j == N - 1){
                rhs(i + (j)*N  + (k)*N*N) +=  (std::sin((i + 1) * deltaX) * std::sin((k + 1) * deltaX) * std::sin(1.0) *
                                                std::sin(t - deltaT))/(2*deltaX*deltaX); // k == N-1
            }
            if(k == N - 1){
                rhs(i + (j)*N  + (k)*N*N) +=  (std::sin(1.0) * std::sin((i + 1) * deltaX) * std::sin((j + 1) * deltaX) *
                                                std::sin(t - deltaT))/(2*deltaX*deltaX); // j == N-1
            }
        }
    }
}
```

Figure 3.4: Implementation for the construction of the RHS.

As we can see, for each step in my spatial discretization, first I add the forcing term, then the boundary conditions of the solution at the step $n + 1$, the solution at the step n and finally the boundary conditions of the solution at the step n.
Finally we solved it using the Conjugate Gradient Method implemeted in Eigen library.



```
//create preconditioner
Eigen::IncompleteLUT<double> preconditioner(A);
preconditioner.compute(A);

// Use Conjugate Gradient solver to solve the linear system
Eigen::ConjugateGradient<SpMat> solver;

solver.setTolerance(1e-10);

solver.setMaxIterations(1000);

solver.compute(A);

if (solver.info() != Eigen::Success) {
    // decomposition failed
    std::cout << "Failed computing factorization" << std::endl;
}

solution = solver.solve(rhs); // Solve Ax = b

if (solver.info() != Eigen::Success) {
    // solving failed
    std::cout << "Failed solving" << std::endl;
}
```

Figure 3.5: Implementation for the CG method.

At last we evaluate the error and the time of execution of the code.

3.3.3. Douglas

Since we were not totally satisfied with the performance of the solver, we will try to manually implement a specific method for the problem. In Douglas Method we transform our heat equation into a system of three equations, one for each direction:

$$\begin{cases} (\Delta_x^2 - \dfrac{2}{\Delta t})T^*_{n+1} = -(\Delta_x^2 + 2\Delta_y^2 + 2\Delta_z^2 + \dfrac{2}{\Delta t})T_n \\ (\Delta_y^2 - \dfrac{2}{\Delta t})T^{**}_{n+1} = \Delta_y^2 T_n - \dfrac{2}{\Delta t}T^*_{n+1} \\ (\Delta_z^2 - \dfrac{2}{\Delta t})T_{n+1} = \Delta_z^2 T_n - \dfrac{2}{\Delta t}T^{**}_{n+1} \end{cases} \tag{3.13}$$

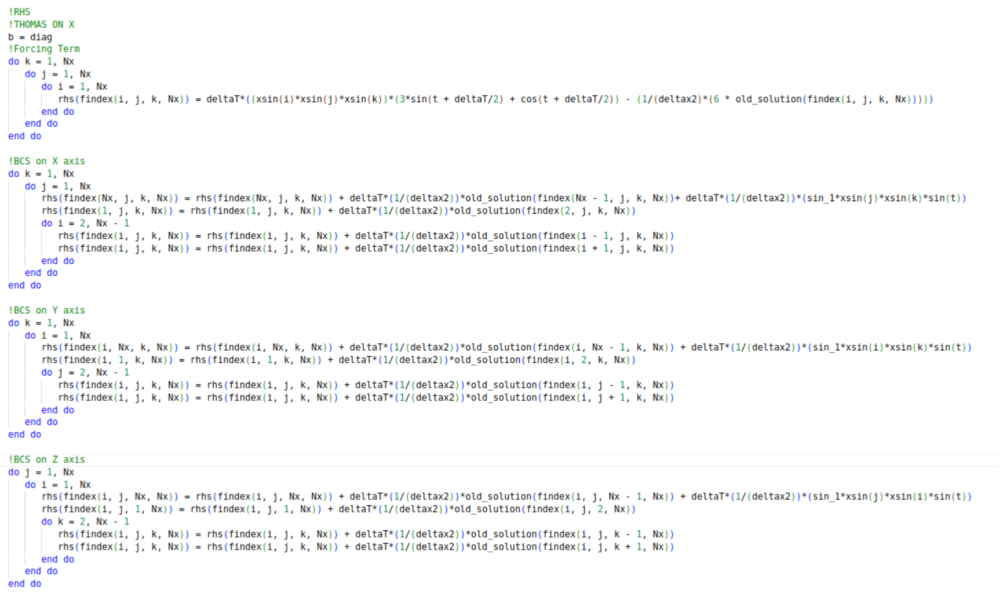
Which can be then trasformed into the following system of equations:

$$\begin{cases} (1-\frac{\Delta t}{2}\Delta_x^2)(T^* - T_n) = \Delta t(\Delta_x^2 + 2\Delta_y^2 + 2\Delta_z^2)T_n + \Delta t f^{n+\frac{1}{2}} \\ (1-\frac{\Delta t}{2}\Delta_y^2)(T^{**} - T_n) = T^* - T_n \\ (1-\frac{\Delta t}{2}\Delta_x^2)(T_n - T_{n+1}) = T^{**} - T_n \\ T_{n+1} = T_n - (T_n - T_{n+1}) \end{cases} \tag{3.14}$$

It is evident that each equation solves the system along a different direction:

- The first equation discretizes the domain along the X axis only and solves it (although the right hand side is computed discretizing along all the directions);
- The second equation takes as RHS the solution of the first equation and solves the system along the y direction;
- The third equation takes as RHS the solution of the second equation and solves the system along the z direction.
- The last step is necessary to get the new solution at the step $n + 1$ from the solution of the Z axis system.

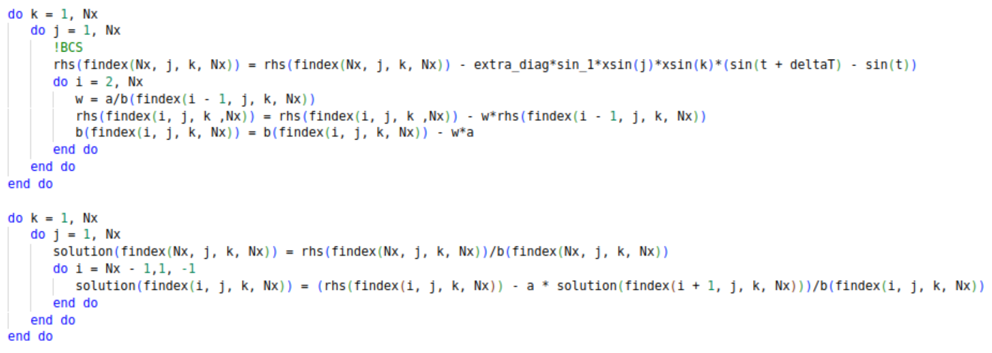
Hence we can sequentially solve the system along each direction, and improve the performance since the systems I'm solving are n times n, instead of n^3 times n^3. Note that we need to add the Boundary conditions for the axis along which we are solving the system. We implement the code in fortran this time, so as before we construct the matrix using the mapping function seen at the beginnig of the section and adding the boundary conditions for each direction:



```fortran
!RHS
!THOMAS ON X
b = diag
!Forcing Term
do k = 1, Nx
   do j = 1, Nx
      do i = 1, Nx
         rhs(findex(i, j, k, Nx)) = deltaT*((xsin(i)*xsin(j)*xsin(k))*(3*sin(t + deltaT/2) + cos(t + deltaT/2)) - (1/(deltax2)*(6 * old_solution(findex(i, j, k, Nx)))))
      end do
   end do
end do

!BCS on X axis
do k = 1, Nx
   do j = 1, Nx
      rhs(findex(Nx, j, k, Nx)) = rhs(findex(Nx, j, k, Nx)) + deltaT*(1/(deltax2))*old_solution(findex(Nx - 1, j, k, Nx))+ deltaT*(1/(deltax2))*(sin_1*xsin(j)*xsin(k)*sin(t))
      rhs(findex(1, j, k, Nx)) = rhs(findex(1, j, k, Nx)) + deltaT*(1/(deltax2))*old_solution(findex(2, j, k, Nx))
      do i = 2, Nx - 1
         rhs(findex(i, j, k, Nx)) = rhs(findex(i, j, k, Nx)) + deltaT*(1/(deltax2))*old_solution(findex(i - 1, j, k, Nx))
         rhs(findex(i, j, k, Nx)) = rhs(findex(i, j, k, Nx)) + deltaT*(1/(deltax2))*old_solution(findex(i + 1, j, k, Nx))
      end do
   end do
end do

!BCS on Y axis
do k = 1, Nx
   do i = 1, Nx
      rhs(findex(i, Nx, k, Nx)) = rhs(findex(i, Nx, k, Nx)) + deltaT*(1/(deltax2))*old_solution(findex(i, Nx - 1, k, Nx)) + deltaT*(1/(deltax2))*(sin_1*xsin(i)*xsin(k)*sin(t))
      rhs(findex(i, 1, k, Nx)) = rhs(findex(i, 1, k, Nx)) + deltaT*(1/(deltax2))*old_solution(findex(i, 2, k, Nx))
      do j = 2, Nx - 1
         rhs(findex(i, j, k, Nx)) = rhs(findex(i, j, k, Nx)) + deltaT*(1/(deltax2))*old_solution(findex(i, j - 1, k, Nx))
         rhs(findex(i, j, k, Nx)) = rhs(findex(i, j, k, Nx)) + deltaT*(1/(deltax2))*old_solution(findex(i, j + 1, k, Nx))
      end do
   end do
end do

!BCS on Z axis
do j = 1, Nx
   do i = 1, Nx
      rhs(findex(i, j, Nx, Nx)) = rhs(findex(i, j, Nx, Nx)) + deltaT*(1/(deltax2))*old_solution(findex(i, j, Nx - 1, Nx)) + deltaT*(1/(deltax2))*(sin_1*xsin(j)*xsin(i)*sin(t))
      rhs(findex(i, j, 1, Nx)) = rhs(findex(i, j, 1, Nx)) + deltaT*(1/(deltax2))*old_solution(findex(i, j, 2, Nx))
      do k = 2, Nx - 1
         rhs(findex(i, j, k, Nx)) = rhs(findex(i, j, k, Nx)) + deltaT*(1/(deltax2))*old_solution(findex(i, j, k - 1, Nx))
         rhs(findex(i, j, k, Nx)) = rhs(findex(i, j, k, Nx)) + deltaT*(1/(deltax2))*old_solution(findex(i, j, k + 1, Nx))
      end do
   end do
end do
```

Figure 3.6: Implementation for the construction of the matrix in Fortran.

Then we start solving our system using Douglas method: first we solve it on the x axis, then on the y axis and finally on the z axis.



```fortran
do k = 1, Nx
   do j = 1, Nx
      !BCS
      rhs(findex(Nx, j, k, Nx)) = rhs(findex(Nx, j, k, Nx)) - extra_diag*sin_1*xsin(j)*xsin(k)*(sin(t + deltaT) - sin(t))
      do i = 2, Nx
         w = a/b(findex(i - 1, j, k, Nx))
         rhs(findex(i, j, k ,Nx)) = rhs(findex(i, j, k ,Nx)) - w*rhs(findex(i - 1, j, k, Nx))
         b(findex(i, j, k, Nx)) = b(findex(i, j, k, Nx)) - w*a
      end do
   end do
end do

do k = 1, Nx
   do j = 1, Nx
      solution(findex(Nx, j, k, Nx)) = rhs(findex(Nx, j, k, Nx))/b(findex(Nx, j, k, Nx))
      do i = Nx - 1,1, -1
         solution(findex(i, j, k, Nx)) = (rhs(findex(i, j, k, Nx)) - a * solution(findex(i + 1, j, k, Nx)))/b(findex(i, j, k, Nx))
      end do
   end do
end do
```

Figure 3.7: Implementation for the Douglas method on the X Direction.

Above is shown the implementation for Douglas on the X direction, in case of Y and Z we just need to change the direction along which we are solving the system.
Finally we update the solution at the step $n + 1$ and we iterate over the time steps.

```fortran
old_solution = solution + old_solution
```

Figure 3.8: Solution update.

3.4. Parallel 3D TD Heat Equation

Lastly, once our code to compute the complete heat equation is ready we start to parallelize it in order to improve further its performances. From a profiling of the previous programes it can be shown that most of the performances are spent on solving the linear system, thus we will try to parallelize this step in particular to improve our codes.
We will explain three different parallelization techniques: 2D Pencil Decomposition, Pipeline, Schur, but we will go into the implementation details of the first one only.

3.4.1. 2D pencil Decomposition

The key idea is to divide the 3D domain into a grid of 2D Pencils equal to the number of processors used.

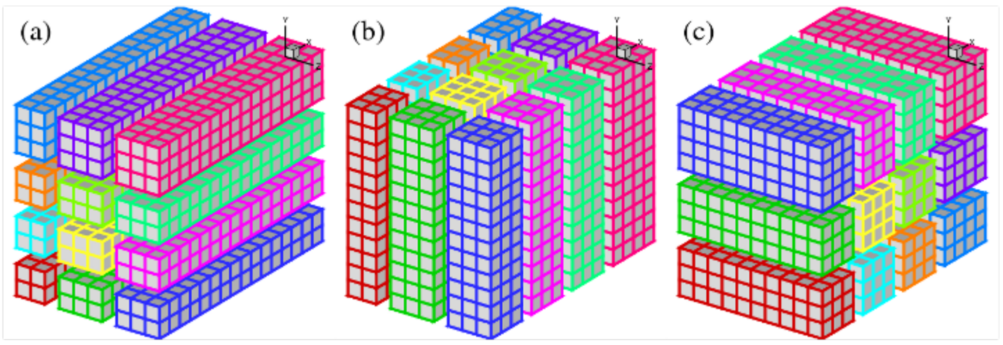


Figure 3.9: Pencils

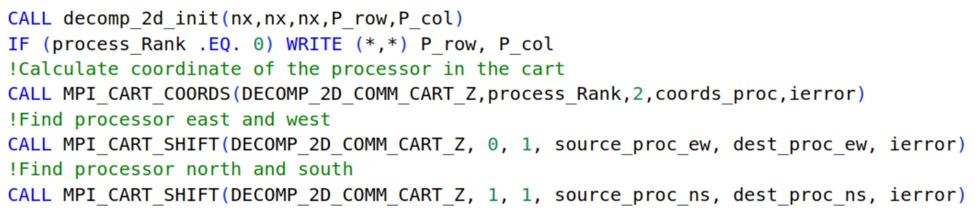
Initially every Pencil is directed along the Z axis, this way every processor can perform Thomas along that specific direction.
Then we rotate the Pencils so they are directed along the Y axis, and we perform Thomas along the Y axis.
Finally we rotate the Pencils so they are directed along the X axis, and we perform Thomas along the X axis.
At the end, processors have to exchange elements at the boundaries of the Pencils with the neighbours processors. Then we iterate over the time steps and we repeat the process.
Finally every processor will have a subsection of the final solution.
Let's have a look at how the code is implemented.

Libraries We need two libraries:

- `2decomp_fft`: allows to decompose the domain into pencils and to rotate them;

- `MPI`: allows to parallelize the code and to more easily handle the decomposition.

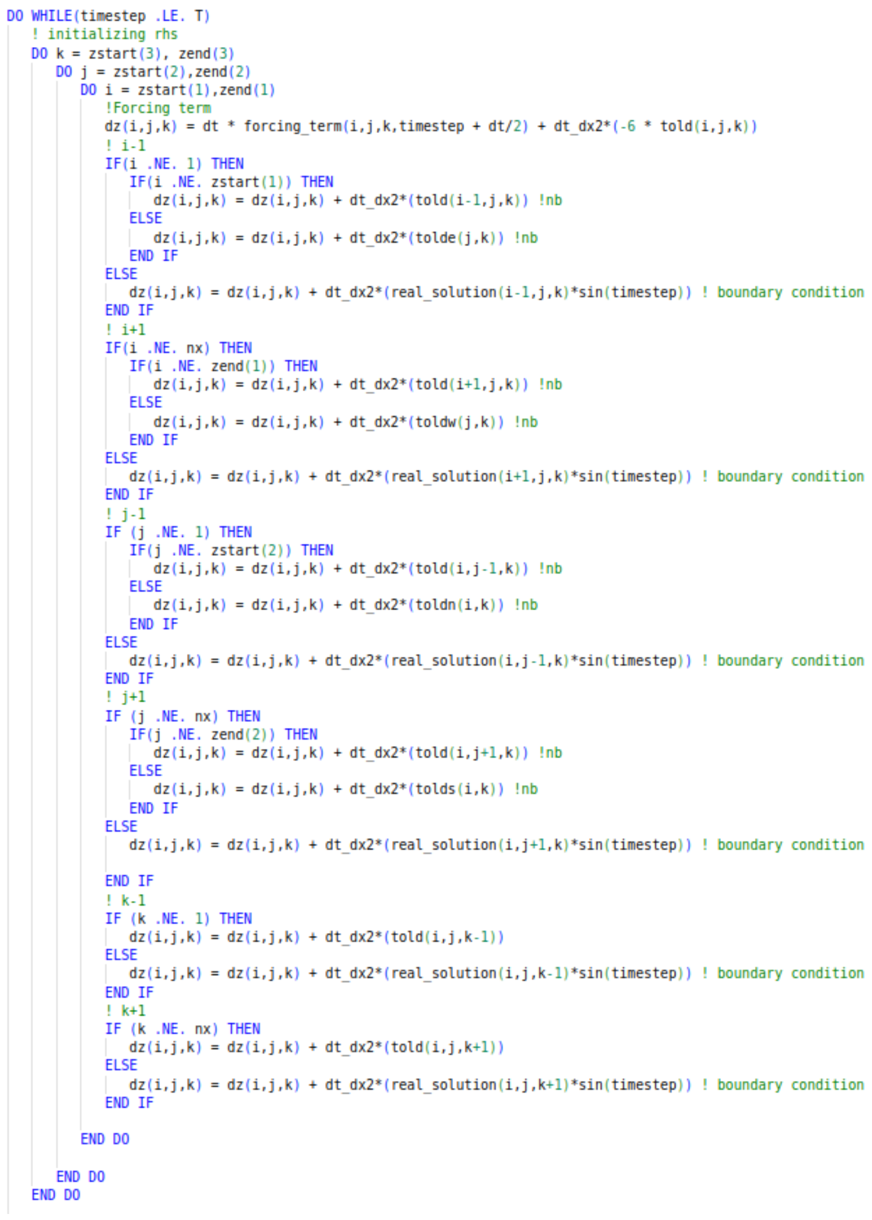
Initialization Then we initialize the `2decomp_fft` library and we setup the `MPI_Cart` grid. Usign the `MPI_CART_COORD` function to define the coordinates for each processor on the 2D grid and the `MPI_CART_SHIFT` function to define the neighbours of each processor.



```fortran
CALL decomp_2d_init(nx,nx,nx,P_row,P_col)
IF (process_Rank .EQ. 0) WRITE (*,*) P_row, P_col
!Calculate coordinate of the processor in the cart
CALL MPI_CART_COORDS(DECOMP_2D_COMM_CART_Z,process_Rank,2,coords_proc,ierror)
!Find processor east and west
CALL MPI_CART_SHIFT(DECOMP_2D_COMM_CART_Z, 0, 1, source_proc_ew, dest_proc_ew, ierror)
!Find processor north and south
CALL MPI_CART_SHIFT(DECOMP_2D_COMM_CART_Z, 1, 1, source_proc_ns, dest_proc_ns, ierror)
```

Figure 3.10: Initialization

Matrix Construction We start building the matrix of the system. As before we use the same mapping function



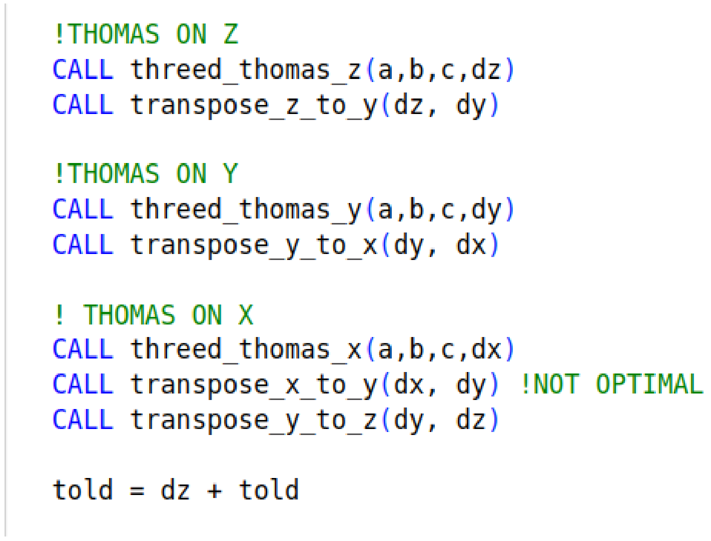
```fortran
DO WHILE(timestep .LE. T)
    ! initializing rhs
    DO k = zstart(3), zend(3)
        DO j = zstart(2),zend(2)
            DO i = zstart(1),zend(1)
                !Forcing term
                dz(i,j,k) = dt * forcing_term(i,j,k,timestep + dt/2) + dt_dx2*(-6 * told(i,j,k))
                ! i-1
                IF(i .NE. 1) THEN
                    IF(i .NE. zstart(1)) THEN
                        dz(i,j,k) = dz(i,j,k) + dt_dx2*(told(i-1,j,k)) !nb
                    ELSE
                        dz(i,j,k) = dz(i,j,k) + dt_dx2*(tolde(j,k)) !nb
                    END IF
                ELSE
                    dz(i,j,k) = dz(i,j,k) + dt_dx2*(real_solution(i-1,j,k)*sin(timestep)) ! boundary condition
                END IF
                ! i+1
                IF(i .NE. nx) THEN
                    IF(i .NE. zend(1)) THEN
                        dz(i,j,k) = dz(i,j,k) + dt_dx2*(told(i+1,j,k)) !nb
                    ELSE
                        dz(i,j,k) = dz(i,j,k) + dt_dx2*(toldw(j,k)) !nb
                    END IF
                ELSE
                    dz(i,j,k) = dz(i,j,k) + dt_dx2*(real_solution(i+1,j,k)*sin(timestep)) ! boundary condition
                END IF
                ! j-1
                IF (j .NE. 1) THEN
                    IF(j .NE. zstart(2)) THEN
                        dz(i,j,k) = dz(i,j,k) + dt_dx2*(told(i,j-1,k)) !nb
                    ELSE
                        dz(i,j,k) = dz(i,j,k) + dt_dx2*(toldn(i,k)) !nb
                    END IF
                ELSE
                    dz(i,j,k) = dz(i,j,k) + dt_dx2*(real_solution(i,j-1,k)*sin(timestep)) ! boundary condition
                END IF
                ! j+1
                IF (j .NE. nx) THEN
                    IF(j .NE. zend(2)) THEN
                        dz(i,j,k) = dz(i,j,k) + dt_dx2*(told(i,j+1,k)) !nb
                    ELSE
                        dz(i,j,k) = dz(i,j,k) + dt_dx2*(tolds(i,k)) !nb
                    END IF
                ELSE
                    dz(i,j,k) = dz(i,j,k) + dt_dx2*(real_solution(i,j+1,k)*sin(timestep)) ! boundary condition

                END IF
                ! k-1
                IF (k .NE. 1) THEN
                    dz(i,j,k) = dz(i,j,k) + dt_dx2*(told(i,j,k-1))
                ELSE
                    dz(i,j,k) = dz(i,j,k) + dt_dx2*(real_solution(i,j,k-1)*sin(timestep)) ! boundary condition
                END IF
                ! k+1
                IF (k .NE. nx) THEN
                    dz(i,j,k) = dz(i,j,k) + dt_dx2*(told(i,j,k+1))
                ELSE
                    dz(i,j,k) = dz(i,j,k) + dt_dx2*(real_solution(i,j,k+1)*sin(timestep)) ! boundary condition
                END IF

            END DO

        END DO
    END DO
```

Figure 3.11: Matrix Construction

We need to add the forcing term and the old solution at the neighbours spatial steps. We also need to check if the cell is at the boundaries of the pencil, so we need to take the value from the neighbours processor, or if the cell is at the boundaries of the entire domain we need to add the boundaries conditions, and we must do this step for each direction.
Note that along Z direction there is no need to exchange values with the other processors since at this step every Pencil spans along the whole Z axis.

Thomas Algorithm Then we start solving the system using Thomas Algorithm.
Similar to the non parallel case with Douglas, we have to solve Thomas along each direcction,

but this time only for a smaller domain for the other two directions.

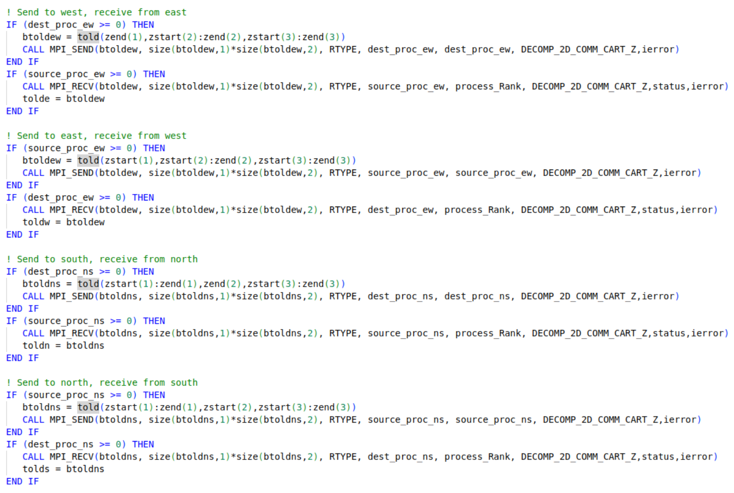


```
!THOMAS ON Z
CALL threed_thomas_z(a,b,c,dz)
CALL transpose_z_to_y(dz, dy)

!THOMAS ON Y
CALL threed_thomas_y(a,b,c,dy)
CALL transpose_y_to_x(dy, dx)

! THOMAS ON X
CALL threed_thomas_x(a,b,c,dx)
CALL transpose_x_to_y(dx, dy) !NOT OPTIMAL
CALL transpose_y_to_z(dy, dz)

told = dz + told
```

Figure 3.12: System solving

So, we start with Thomas on the Z axis, we perfom a rotation of the Pencils using `transpose_z_to_y` from the `2decomp_fft` library. We repeat the process for the Y axis and finally for the X axis. Then we return to the Z axis and we update the new solution.

Boundaries exchange At the end we need to exchange the boundaries of the Pencils with the neighbours processors.
We use the coordinates defined at the initialization section, each processor has to switch values with 2/4 processor depending on its position.



```
! Send to west, receive from east
IF (dest_proc_ew >= 0) THEN
   btoldew = told(zend(1),zstart(2):zend(2),zstart(3):zend(3))
   CALL MPI_SEND(btoldew, size(btoldew,1)*size(btoldew,2), RTYPE, dest_proc_ew, dest_proc_ew, DECOMP_2D_COMM_CART_Z,ierror)
END IF
IF (source_proc_ew >= 0) THEN
   CALL MPI_RECV(btoldew, size(btoldew,1)*size(btoldew,2), RTYPE, source_proc_ew, process_Rank, DECOMP_2D_COMM_CART_Z,status,ierror)
   tolde = btoldew
END IF

! Send to east, receive from west
IF (source_proc_ew >= 0) THEN
   btoldew = told(zstart(1),zstart(2):zend(2),zstart(3):zend(3))
   CALL MPI_SEND(btoldew, size(btoldew,1)*size(btoldew,2), RTYPE, source_proc_ew, source_proc_ew, DECOMP_2D_COMM_CART_Z,ierror)
END IF
IF (dest_proc_ew >= 0) THEN
   CALL MPI_RECV(btoldew, size(btoldew,1)*size(btoldew,2), RTYPE, dest_proc_ew, process_Rank, DECOMP_2D_COMM_CART_Z,status,ierror)
   toldw = btoldew
END IF

! Send to south, receive from north
IF (dest_proc_ns >= 0) THEN
   btoldns = told(zstart(1):zend(1),zend(2),zstart(3):zend(3))
   CALL MPI_SEND(btoldns, size(btoldns,1)*size(btoldns,2), RTYPE, dest_proc_ns, dest_proc_ns, DECOMP_2D_COMM_CART_Z,ierror)
END IF
IF (source_proc_ns >= 0) THEN
   CALL MPI_RECV(btoldns, size(btoldns,1)*size(btoldns,2), RTYPE, source_proc_ns, process_Rank, DECOMP_2D_COMM_CART_Z,status,ierror)
   toldn = btoldns
END IF

! Send to north, receive from south
IF (source_proc_ns >= 0) THEN
   btoldns = told(zstart(1):zend(1),zstart(2),zstart(3):zend(3))
   CALL MPI_SEND(btoldns, size(btoldns,1)*size(btoldns,2), RTYPE, source_proc_ns, source_proc_ns, DECOMP_2D_COMM_CART_Z,ierror)
END IF
IF (dest_proc_ns >= 0) THEN
   CALL MPI_RECV(btoldns, size(btoldns,1)*size(btoldns,2), RTYPE, dest_proc_ns, process_Rank, DECOMP_2D_COMM_CART_Z,status,ierror)
   tolds = btoldns
END IF
```

Figure 3.13: Boundaries exchange

First each processor sends to his neighbour at the east side (if present) and it receives from the west side the values at the boundaries. And viceversa it sends to the west side and receives from the east side.
Then it does the same thing for the north and south side, all this process it computed simultaneously by all processors, in order that at the end they have the value they are facing at the boundaries.
At the end as always we compute the error and the time of execution.

3.4.2. Pipelined

In this technique the domain is divided into cubes instead of pencils and the parallelization is implemented on all directions.
Since in thomas algorithm the for loops contain dependecies between iterations, in particular each iteration needs a value from the iteration before, we need to construct a pipeline enviroment between the processors in order to pass these values. Hence for each direction we obtain a full parallelization in the other two directions, while in the current direction we implement a pipeline execution order where each processor waits for the processor before or after, depending if we are executing the backward or the forward loop of the Thomas algorithm.

3.4.3. Schur

The key idea is to divide the system matrix, as well as the right hand side and the solution, into a number of overlapping submatrices equal to the number of processor used.
The common elements between the submatrices are moved to the right and bottom side of the matrix and at the end of the diagonal. Hence we will have n submatrices A, one for each processor, and n matrices D and E with the common elements, and a final matrix B called Schur complement.
Suppose we have two processors and a generic system matrix:

$$\begin{bmatrix} A^- & c_i^- & 0 \\ a_i^- & b_i & c_i^+ \\ 0 & a_i^+ & A^+ \end{bmatrix} \begin{bmatrix} X^- \\ x_i \\ X^+ \end{bmatrix} = \begin{bmatrix} f^- \\ f_i \\ f^+ \end{bmatrix}$$

we can rearrange it as:

$$\begin{bmatrix} A^- & 0 & c_i^- \\ 0 & A^+ & a_i^+ \\ a_i^- & c_i^+ & b_i \end{bmatrix} \begin{bmatrix} X^- \\ X^+ \\ x_i \end{bmatrix} = \begin{bmatrix} f^- \\ f^+ \\ f_i \end{bmatrix}$$

Then the matrix can be rewritten as:

$$\begin{bmatrix} A^- & 0 & D_s^- \\ 0 & A^+ & D_s^+ \\ E_s^- & E_s^+ & b_i \end{bmatrix}$$

And finally transformed in the following way:

$$\begin{bmatrix} A^- & 0 & D_s^- \\ 0 & A^+ & D_s^+ \\ 0 & 0 & B_i \end{bmatrix}$$

Where:

$$B_i = b_i - E_s^-(A^-)^{-1}D_s^- - E_s^+(A^+)^{-1}D_s^+$$

is called Schurs complement and can be proven to be, not only smaller, but also tridiagonal. Finally from here we can easily calculate the solution:

$$\begin{cases} X_i = B_i^{-1}(f_i - E_s^-(A^-)^{-1}f^- - E_s^+(A^+)^{-1}f^+) \\ \hfill X^+ = (A^+)^{-1}(f^+ - D_s^+X_i) \\ \hfill X^- = (A^-)^{-1}(f^- - D_s^-X_i) \end{cases}$$

In the general case with n processors we:

- first assemble Schur complement and solve x_i globally;
- we send x_i to all processors;
- every processor locally solves X^j;

4 | Results

4.1. 1D Time Independet

For the 1-dimensional Time independent heat equation, we computed the error and we estimated the time execution for $10^2, 10^3, 10^4, 10^5, 10^6$ and 10^7 grid points. The results are shown in the following table:

Grid points	L_2 Norm Error /points	Time [s]/points
10^2	3.53189×10^{-8}	4.3536×10^{-7}
10^3	1.13202×10^{-10}	3.97014×10^{-7}
10^4	3.4015×10^{-13}	3.66991×10^{-7}
10^5	2.13598×10^{-13}	3.83695×10^{-7}
10^6	2.62698×10^{-13}	3.74883×10^{-7}
10^7	4.22898×10^{-11}	3.81862×10^{-7}

Table 4.1: Error and time execution for 1D Time Independet

If we look at the time execution divided by the number of points, we clearly see that it remains almost constant, so the algorithm scales well with the number of points.
While if we plot the error:

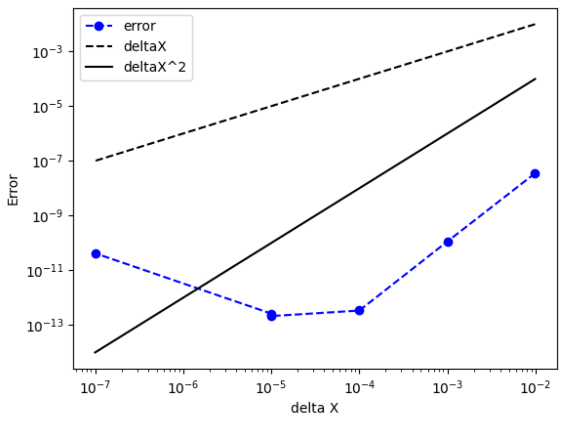


Figure 4.1: Error for 1D Time Independet

We see that for smaller number of points the error converges as Δx^2 as expected from theory (since we are using second order Finite Difference Method), while, when the number of points increases (around 10^6) the error starts increasing too.

4.2. 1D Time Dependent

For the Time Dependent 1-dimensional case, we again computed the L_2 norm error divided by the number of points in space and time and then we calculated the execution time, but this time for a smaller number of points since the computational cost is higher. We choose 10, 20, 40, 80 and 160 points in space, while the points in time were calculated as $\Delta t = \frac{\Delta x^2}{2}$. We got the following results:

Spatial Points (Nx)	Temporal Points (Nt)	L_2 Norm Error /points	Total Time [s]/points
10	400	0.000175611	2.28876e-07
20	1600	3.38828e-05	1.72873e-07
40	6400	6.25633e-06	1.78064e-07
80	25600	1.13029e-06	1.78536e-07
160	102400	2.01992e-07	1.79468e-07

Table 4.2: Error and Total Time for different Spatial and Temporal Points

Again the execution time scales well with the number of points, while the error converges as Δx^2, as it is more evident if we plot it:

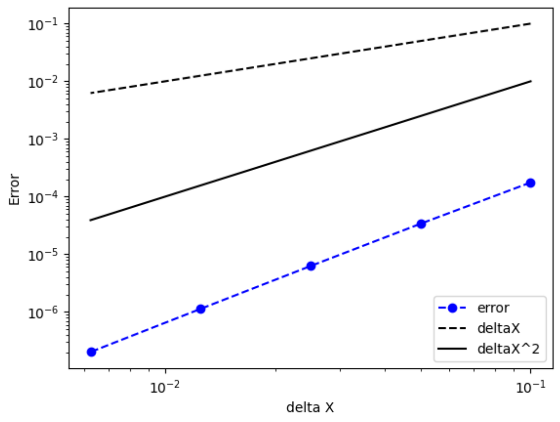


Figure 4.2: Error for 1D Time Dependent

4.3. 3D Time Dependent

We computed the execution time to solve the linear system for both the Conjugate Gradient taken from the `Eigen` library, and the Doublas method directly implemented.
Below are shown the results for fixed Nt and various number of spatial points:

Points	CG Solving Time [s]	Douglas Solving Time [s]
10	1.19784	0.181014001
30	78.7525	6.99920058e-02
40	259.537	8.84680003e-02
100	/	2.21069384
200	/	20.9538994
300	/	76.2972488

Table 4.3: Comparison of CG and Douglas solving time

As expected, it is clear that Douglas increases a lot the performances with respect to other solvers.

4.4. Parallelization

Finally we tested the three parallelization techniques.
Since for the Schur technique we only have the 1DTI case, we compared how the Pencil and the Pipeline techniques scale with the number of processors.
We used N spatial points equal to 100 for each direction, Nt equal to 100 and we compared 1, 2, 4 processors.

Processors	Pencil solving time	Pipeline solving time
1	12.449375s	13.9847s
2	6.897545s	7.41955s
4	4.115876s	4.1417s

Table 4.4: Comparison of Pencil and Pipeline solving time for different number of processors

As we can see the Pencil technique scales well from 1 to 2 processors, but it loses performances when we increase to 4 processors. On the other hand, the Pipeline technique scales worse and takes more time to solve, probably due to the bigger overhead that it contains.