UNIVERSITY OF STRASBOURG

MASTER CSMI

Internship Report : Parareal method and data assimilation for PDEs with Feel++

Authors:
Melissa Aydogdu
Frédérique Lecourtier

Supervisors:
Christophe Prudhomme
Luca Berti

Date: August 21, 2022

Contents

1 Introduction

This internship is the continuation of a project realised in th platform Cemosis . During the project the main goals were to implement a parallel-in-time resolution method for the Lorenz system, to realize the data assimilation using the EnKF method using a function already implemented in the library Filterpy. For this we also had to implement several methods to solve numerically the Lorenz system (like RK4). During the project, all these methods were implemented or used in Python.

1.1 Presentation of Cemosis

This project is managed by Cemosis which is the "Centre de Modélisation et de Simulation de Strasbourg" (Strasbourg Modeling and Simulation Center). Cemosis is hosted by the Institute of Advanced Mathematical Research (IRMA) and was created in January 2013. Cemosis work is focused on the numarical simulation and mathematical modeling of different phenomena. They use and develop tools in the fields of:

- **MSO** - Modeling Simulation and Optimization
- **DS** - Data Science, Big Data, Smart Data
- **HPC** - High Performance Computing, Parallel Computing, Cloud Computing
- **SI** - Signal and Image processing

They work with researchers and engineers of other research centers and companies.
For more informations, refer to the cemosis website.

1.2 Context

Cemosis relies currently on the team Modeling and Control of the IRMA and is developing competences and projects in the energy sector of buildings. Nowadays it is important to reduce the energy consumption of buildings in order to move to a more ecologic and economic lifestyle. For this we need to know how to simulate and model buildings using fast simulation methods such as parallel calculation and data integration because our models used to simulate buildings are never perfect, and even less in the case of existing buildings where we have little information. In fact, in order to perform simulations over long periods, such as a year, we would also have to take into consideration phenomena such as radiative exchanges and convective effects, which therefore require a PDE model if we want a spatial discretization.

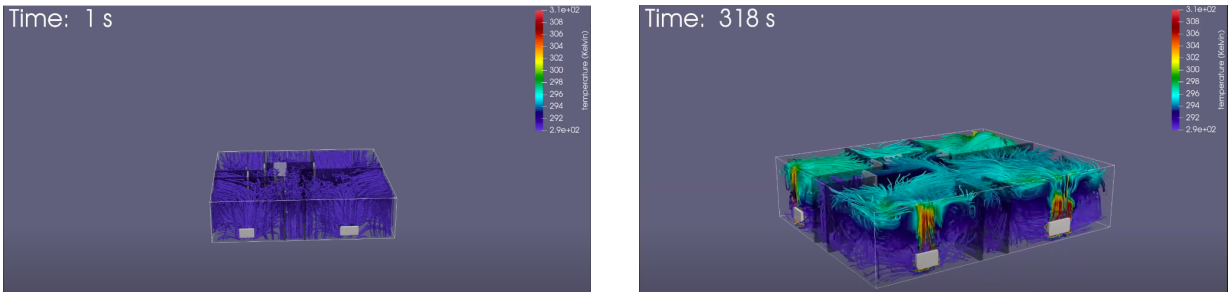


Figure 1.1: Example of a building simulation with Feel++

1.3 Goals of the Intership

The main objective of the intership is to implement the Parareal method and data assimilation for PDEs with Feel++.

Objectives for the common part:

1. To read the following article about the Heat equation (Chapter 11) : [14]
2. To set up a project on Github :
   - Organisation of the repository (Python library, cmake in C++ ...)
   - Create issues to see the progress of the project
   - Set up the CI : build, test, documentation

   Github repository : https://github.com/master-csmi/2022-m1-lorenz

Objectives for parareal method:

1. Implement the parareal method in C++ and :
   - Test the method (with oscillator)
   - Check convergence and stability results
   - Check speed-up and efficiency
2. Implement the resolution of the heat equation in C++ with Feel++
   ⇒ Implement the resolution of the Laplace equation in C++ with Feel++
3. Use the previous implementation of the heat equation with the parareal method
4. Check the convergences/accuracies of the method

Objectives for the Data assimilation:

1. Write a class for the EnKF in C++, inspired by the EnKF written in Python (FilterPy); Test the algorithm.
2. To read the following article :
    - Fundamentals Of Building Performance Simulation By Beausoleil-Morrison
3. Understand the heat equation and what are the phenomena involved in the modification of the temperature of a building (conduction, convection, radiation).
4. Write the mathematical problem to be solved if we want to simulate an office then realize the simulation using Feel++ toolboxes.
5. Introduce data assimilation using the sensor and correct the simulation.

2 Differential Equations

In Mathematics, ordinary differential equations (ODE) are equations that involve derivatives of one-variable functions, and partial differential equations (PDE) are equations that imposes relations between the various partial derivatives of a multivariable function. The difference between ODEs and PDEs is that for ODEs the unknown functions depend only on one variable, whereas for PDEs the unknown functions may depend on several independent variables. Differential equations are an important object of study in both pure and applied mathematics. They are used to build mathematical models of physical and biological evolution processes, for example for the study of radioactivity, celestial mechanics, weather or population dynamics... During the project we had already used the Lorenz system and the harmonic oscillator, for the internship we also used these two differential equations but we worked with two other equations: the heat equation and the Laplacian.

2.1 ODE: Harmonic oscillator

A harmonic oscillator is an ideal oscillator whose evolution over time is described by a sinusoidal function, whose frequency depends only on the characteristics of the system and whose amplitude is constant.This mathematical model describes the evolution of any physical system in the vicinity of a stable equilibrium position, which makes it a transversal tool used in many fields: mechanics, electricity and electronics, optics.

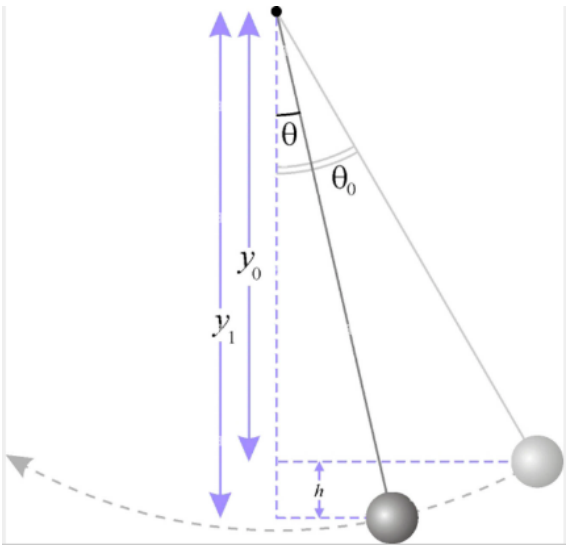


Figure 2.1: Simple pendulum

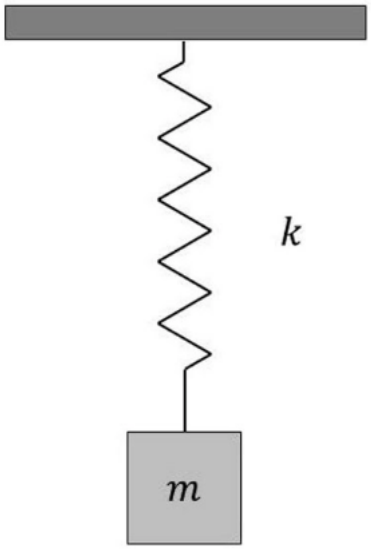


Figure 2.2: Spring/mass system

Let's consider a mass-spring system of the following form :

$$\frac{d^2x}{dt^2} + \omega_0^2 x = 0 \quad \Longleftrightarrow \quad \frac{d^2x}{dt^2} = -\omega_0^2 x \quad \text{with} \quad \omega_0 = \sqrt{\frac{k}{m}}. \tag{1}$$

k and m are the spring stiffness and the suspended mass respectively. We are interested in this equation because its exact solution is known and is of the form :

$$x(t) = x_0 \cos(\omega_0 t + \phi_0).$$

First of all, the numerical methods such as Runge Kutta order 4 allow us to solve first order differential equations. But the harmonic oscillator equation (1) is a second order equation. We will therefore start by making a simple change of variable which will allow us to replace this second order differential equation by a system of two first order equations. We pose :

$$\frac{dx}{dt} = v \quad \Rightarrow \quad \frac{d^2x}{dt^2} = \frac{dv}{dt}.$$

As a result the equation becomes :

$$\begin{cases} \dfrac{dx}{dt} = v \\ \dfrac{dv}{dt} = -\omega_0^2 x \end{cases}$$

Let's take an example to understand how we can determine the exact solution from the initial conditions that we will take. For example if we take $\omega_0 = 5$ and $(x(0), v(0)) = (0, 1)$, we have :

$$\begin{cases} x(0) = 0 \\ v(0) = 1 \end{cases} \Rightarrow \begin{cases} x_0 \cos(\phi_0) = 0 \\ -x_0 \omega_0 \sin(\phi_0) = 1 \end{cases} \Rightarrow \begin{cases} x_0 = \dfrac{-1}{5} \\ \phi_0 = \dfrac{\pi}{2} \end{cases}$$

And thus the solutions of the equation are of the form :

$$x(t) = \frac{-1}{5} \cos(5t + \frac{\pi}{2}).$$

2.2 ODE: Lorenz system

The Lorenz system is a simplified three-variable model to investigate atmospheric convection. This model has had important repercussions in showing the possible limits on the ability to predict long-term climate and weather evolution. One of the important characteristics of the Lorenz system is that it is a chaotic system, which means that this type of system is roughly defined by sensitivity to initial conditions: infinitesimal differences in initial conditions of the system result in large differences in behavior.

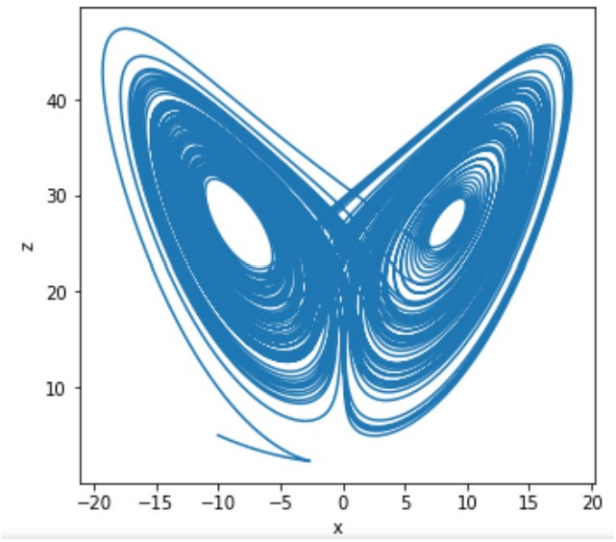


Figure 2.3: butterfly wing pattern

The Lorenz system defines a 3 dimensional trajectory by differential equations with 3 parameters.

$$\begin{cases} x' &= \sigma(y-x) \\ y' &= x(r-z)-y \\ z' &= xy-bz \end{cases}$$

Here, x is proportional to the rate of convection, y is related to the horizontal temperature variation, and z is the vertical temperature variation. We have also three parameters all strictly positive:

- $\sigma > 0$ relates to the Prandtl number. This number is a dimensionless quantity that puts the viscosity of a fluid in correlation with the thermal conductivity;
- $r > 0$ relates to the Rayleigh number, it is a control parameter, representing the temperature difference between the top and bottom of the tank;
- $b > 0$ relates to the physical dimensions of the layer of fluid uniformly heated from below and cooled from above.

We can see that this system is non-linear, because in the second differential equation ($\frac{dy}{dt}$) we can see the term xz and in the third differential equation ($\frac{dz}{dt}$) we have xy. The three differential equations form a coupled system.
Let us now determine the fixed points of the Lorenz system. These are the points such that $X' = 0$.

$$X' = (x', y', z') = 0 \quad \Rightarrow \quad \begin{cases} x' = \sigma(y-x) & = 0 \\ y' = x(r-z)-y & = 0 \\ z' = xy-bz & = 0 \end{cases} \quad \Rightarrow \quad \begin{cases} x = y \\ (r-1-z)x = 0 \\ x^2 = bz \end{cases}$$

- If $x = 0$: $\quad y = 0$ and $z = 0$.
- If $x \neq 0$ and $r > 1$: $\quad \begin{cases} z = r - 1 \\ x = y = \pm\sqrt{b(r-1)} \end{cases}$

We deduce that the fixed points of the Lorenz system are: $(0,0,0)$ for all values of the parameters. And for $r > 1$, there is also a pair of fixed points $(\sqrt{b(r-1)}, \sqrt{b(r-1)}, r-1)$ and $(-\sqrt{b(r-1)}, -\sqrt{b(r-1)}, r-1)$.

2.3 PDE: Laplacian equation

The Laplace equation is a second-order partial differential equation . This equation is a basic PDE that arises in the heat and diffusion equations. It is a useful method for determining electric potentials in space or the free region. It is often written as :

$$\Delta f = 0 \qquad \text{or} \qquad \nabla^2 f = 0$$

with Δ the Laplace operator. We can define the Laplace operator as follows: $\Delta = \nabla \cdot \nabla$ where $\nabla\cdot$ divergence operator and ∇ is the gradient operator.
We can write the problem this way:

$$\begin{cases} -\Delta u = f & \Omega \\ u = g & \Gamma_D \\ \dfrac{\partial u}{\partial n} = h & \Gamma_N \\ \dfrac{\partial u}{\partial n} + u = l & \Gamma_R \end{cases}$$

Where Ω corresponds to our domain, Γ_D corresponds to the Dirichlet boundary condition, Γ_N to the Neumann boundary condition and Γ_R to the Robin boundary condition.

2.4 PDE: Heat equation

Heat transfer is the process of energy transfer resulting from a temperature difference.
Thermal analysis is undertaken to predict temperatures and heat transfer in and around bodies. This information can then be used to model temperature-dependent phenomena, such as heat-induced stresses or the effect on fluid flow in the case of a solidifying metal. Heat flow has been classified into three different modes: conduction, convection and radiation.

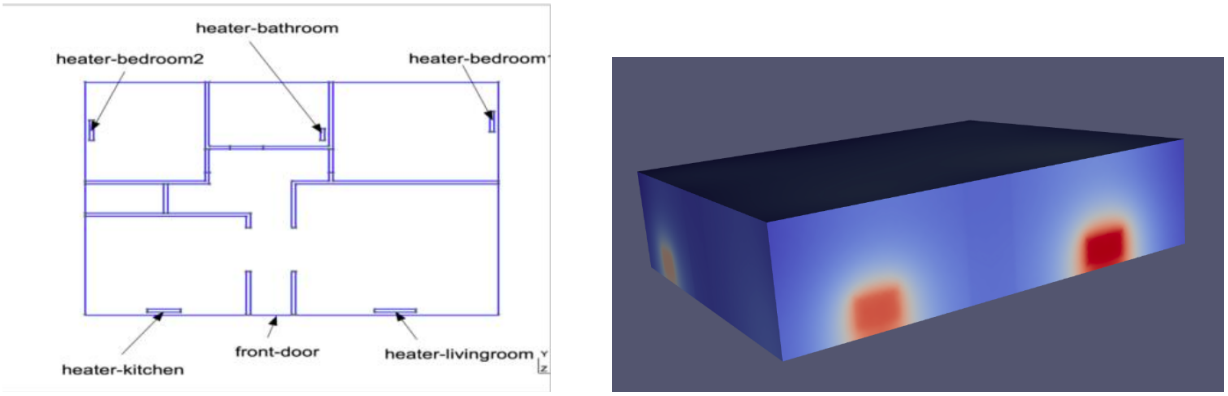


Figure 2.4: Simulation of the temperature of a building using the heat equation with Feel++

The heat equation with convective effects can be written as:

$$\rho C_p((\frac{\partial T}{\partial t}) + u.\nabla T) - \nabla.(k\nabla T) = Q$$

and it must be completed with boundary conditions and initial conditions.

Notation	Quantity	Unit
ρ	density	$Kg.m^{-3}$
C_p	Specific heat	$J/KgC = J/KgK$
k	Conductivity	$W/mC = W/mK$
u	Fluid velocity	$m.s^{-1}$
T	Temperature	K or C
t	Time	s.

Table 1: Parameters for the heat equation

The values of the chosen parameters are typical for air.

2.5 Runge-Kutta

During our project and internship we used the Runge-Kutta method of order 4 to solve ordinary differential equations. Runge-Kutta techniques are one-step numerical schemes for solving ordinary differential equations. They are among the most popular methods because of their ease of implementation and accuracy.
We consider $f : [0;T] \times \mathbb{R}^n \to \mathbb{R}^n$ a continuous function. For $X_0 \in \mathbb{R}^n$, the problem is to find $X \in C^1([0,T],\mathbb{R}^n)$ solution for the differential equation:

$$\begin{cases} X' = f(t,X), \\ X(0) = X_0. \end{cases}$$

After discretizing the problem in time, we can use the Runge Kutta method of order 4 to solve the ODE:

$$X_{n+1} = X_n + \frac{\Delta t}{6}\left(K_1 + 2K_2 + 2K_3 + K_4\right),$$

where

$$\begin{cases} K_1 = f(t_n, X_n), \\ K_2 = f\left(t_n + \dfrac{\Delta t}{2}, X_n + \dfrac{1}{2}K_1\Delta t\right), \\ K_3 = f\left(t_n + \dfrac{\Delta t}{2}, X_n + \dfrac{1}{2}K_2\Delta t\right), \\ K_4 = f\left(t_n + \Delta t, X_n + K_3\Delta t\right). \end{cases}$$

3 Parareal method

Parareal method is a parallel-in-time integration method which was introduced in 2001 by Lions, Maday and Turinici [13]. Parareal computes the numerical solution for multiple time steps in parallel and it is categorized as a parallel-across-the-steps method.

3.1 Explanation

We consider an initial value problem of the form

$$\begin{cases} X' = f(t, X), & t_0 \leq t \leq T, \\ X(t_0) = X_0. \end{cases}$$

3.1.1 Time decomposition

Parareal method needs a decomposition of the time interval $[t_0, T]$ into P slices $[t_j, t_{j+1}]$ with $j \in \{0, \ldots, P-1\}$, where P is the number of process units. As we want to parallelize the algorithm, each time slice is assigned to one process.
We denote by F an integrator which is of high accuracy and G which is of low accuracy. Also, F will be very expensive in terms of calculation but very accurate, and G will be very cheap but very imprecise. We denote by Δt_F the time step and by Δt_G the coarse time step.
To have the right number of points in total (i.e. the sum of the number of points of each interval is equal to the number of points between t_0 and T), we are not going to cut the interval in equal parts. However, the t_j will have to be multiples of Δt_G and the final point of the fine integrator on each process should be equal to the final point of the coarse integrator.
To do this, let's do the following:

- We compute the number of coarse time steps n_G and the number of fine time steps n_F between t_0 and T as well as the ratio between the two $r = n_F / n_G$.
- We do the Euclidean division of n_G by the number of processes P, which gives us the number of coarse time steps on each subinterval. Then we distribute equally to each process the rest of the division.
- We compute the number of fine time steps on each sub-interval using r such that the final time of the fine integrator and the final time of the coarse integrator are the same on each sub-interval.
- We then compute the t_j, i.e. the initial times of each subinterval, from the number of coarse time steps per process.

For example, taking $P = 3$ processes and the following parameters :

$$\Delta t_G = 0.1, \quad \Delta t_F = 0.01, \quad t_0 = 0, \quad T = 10$$

We have :

$$n_G = 100, n_F = 1000, r = 1000/100 = 10$$

Thus we can calculate the number of coarse time steps per subinterval :

$$tab_G = [34, 33, 33]$$

Multiplying by r, we obtain the number of fine time steps per subinterval :

$$tab_G = [340, 330, 330]$$

And we deduce the following t_j :

$$times = [0, 3.4, 6.7, 10]$$

3.1.2 Principle of parareal method

We denote by U_j^k, $j \in \{0, \dots, P\}$ the initial point at time t_j and at iteration k. We also denote $F(U_{j-1}^k)$, $j \in \{1, \dots, P\}$ the result of the fine integrator between t_{j-1} and t_j which starts from the initial point U_{j-1}^k at iteration k and respectively $G(U_{j-1}^k)$, $j \in \{1, \dots, P\}$ the result of the coarse integrator between t_{j-1} and t_j which starts from the initial point U_{j-1}^k at iteration k. Then, a series of steps (*see Figure 2*) is performed until the solution of the system converges.

At iteration $k = 0$:

- Step 1 (*see 3.1.a*) : At iteration $k = 0$, we have an initial point $U_0^0 = X_0$.
- Step 2 (*see 3.1.b*) : We start by applying the function G on all intervals $[t_j, t_{j+1}]$ and we denote by $U_j^0 = G(U_{j-1}^0)$ the values of G at t_j.
  Note that this part of the method must be done sequentially because if we parallelize the task, the process j should wait until the process $j-1$ has finished before starting.
- Step 3 (*see 3.1.c*) : We can then calculate from each U_j^0 the fine solution between t_j and t_{j+1} : $F(U_j^0)$. This is an operation that must be parallelized.

At iteration $k = 1$:

- Step 4 (*see 3.1.d*) : We can then continue to iteration $k = 1$ where we need the values $G(U_j^0)$ and $F(U_j^0)$ calculated at the previous iteration ($k = 0$).
  We will also keep the initial point at time t_0 : $U_0^1 = U_0^0$.
- Step 5 (*see 3.1.e*) : We can then calculate $G(U_0^1)$ which allows us to obtain the point U_1^1 by the following formula:

  $$U_j^1 = G(U_{j-1}^1) + (F(U_{j-1}^0) - G(U_{j-1}^0)).$$

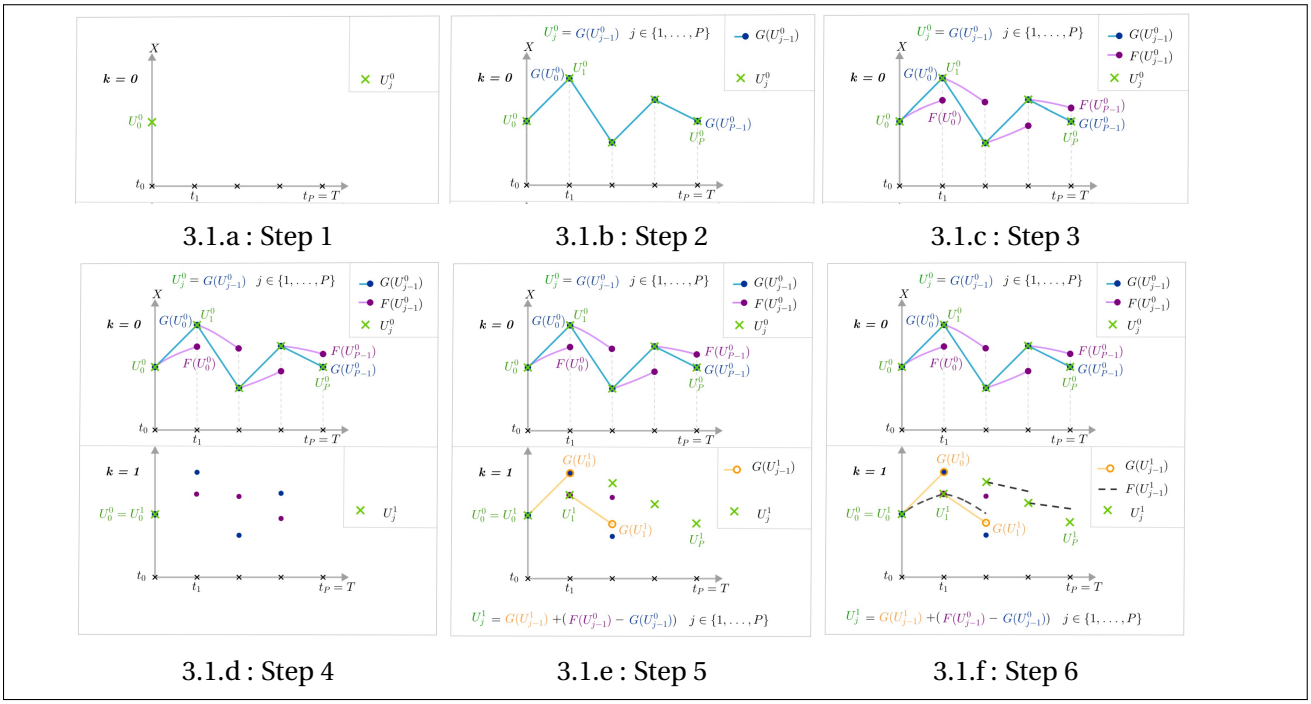
  Note that due to $U_0^1 = U_0^0$, we have $G(U_0^1) = G(U_0^0)$ and therefore $U_1^1 = F(U_0^0)$
  We then compute in the same way the following $G(U_j^1)$ and the associated U_{j+1}^1 points.
  This step can be done sequentially for the same reason as in step 2.

- Step 6 (*see 3.1.f*) : We can then calculate from each U_j^1 the fine solution between t_j and t_{j+1} : $F(U_j^1)$. This is an operation that must be parallelized.
  Note that due to $U_0^1 = U_0^0$, we also have $F(U_0^1) = F(U_0^0)$.

Then we repeat steps 3 to 6 until $U_j^k - U_j^{k-1} \to 0 \quad \forall j \in \{0, \dots, P-1\}$.
We have at iteration k:

$$U_j^k = G(U_{j-1}^k) + (F(U_{j-1}^{k-1}) - G(U_{j-1}^{k-1}))$$

The following figure (Figure 2) illustrates the previous steps :



3.1.a : Step 1 | 3.1.b : Step 2 | 3.1.c : Step 3

3.1.d : Step 4 | 3.1.e : Step 5 | 3.1.f : Step 6

Figure 3.1: Parareal method

Remarks : At iteration k.

- We have : $U_0^k = U_0^0 = X_0 \quad \forall k.$

- Note that the first 2 steps are always done because there can't be convergence with only one value. So, for example, if we take only one process and we apply the parareal method, we will have at the first iteration ($k = 0$) only one initial point $U_0^0 = X_0$ and we will compute the fine and coarse solution only on this point. We can then go to the next iteration, which will be exactly the same, and the algorithm will stop immediately. Indeed, there will be convergence between U_0^0 and U_0^1 (because they are equal), moreover there is obviously convergence between the 2 solutions. However, there is still one extra iteration ($k = 1$) and the computation of the coarse solution is also useless because it is not used to update the other initial points due to the fact that there is only one.

- We can also notice that the calculations which are done on the last interval $[t_{P-1}, t_P]$ are not used because the points U_P^k are not useful for the method. On this interval, one could then compute the fine solution only if there is convergence.

3.1.3 Algorithm

We will present in this section the algorithm to represent the parareal method. We will have :

input : t_0 : initial time ; T : final time ; U_0^0 : initial point ;
F : fine integrator ; G : coarse integrator ;
Δt_F : fine time step ; Δt_G : coarse time step ;
P : number of processes
output: sol : solution with parareal method

This is the initialization of the parareal method:

```
1  k←0;
2  init_pts←[U_0^0];
3  coarse←[];
4  fine←[];
5  for j ← 1 to P do
6      coarse[j-1]←G(init_points[j-1])[-1];
7      init_pts[j]←coarse[j];
8  end
9  U_j^k ← send init_pts[j] to each proc j;
10 do in parallel
11     tab_F← F(U_j^k) // fine solution on each sub-interval
12 end
13 fine[j]← recv tab_F[-1] from each proc j;
```

Algorithm 1: initialization

And this is the entire algorithm for the parareal method (with the initialization) :

```
initialization();
while not converge do
    for j ← 1 to P do
        val_G←G(init_points[j-1])[-1];
        init_pts[j]←val_G+fin[j-1]-coarse[j-1];
        coarse[j-1]←val_G;
    end
    U_j^k ← send init_pts[j] to each proc j;
    do in parallel
        tab_F← F(U_j^k) // fine solution on each sub-interval
    end
    fine[j]← recv tab_F[-1] from each proc j;
end
// Get the solution between y_0 and T
//  sol(t_0, T) = [sol(t_0, t_1), ..., sol(t_{P-1}, t_P)]
sol ← send the final point from the last process;
return sol
```

Algorithm 2: parareal

3.1.4 Order of parareal method

We say that the previous method is of order k (see [10]) if there is a constant c_k such that :

$$\forall j \in \{0,\dots,P-1\} \qquad |U_j^k - U_{ex}(t_j)| + \max_{t\in[t_j,t_{j+1}]} |U_k(t) - U_{ex}(t)| \leq c_k(\Delta t_G)^k \tag{2}$$

where U_j^k is the initial point at t_j and iteration k, $U_{ex}(t_j)$ is the exact solution at the same time (at t_j and iteration k), U_k is the solution between t_0 and T at iteration k and U_{ex} is the exact solution between t_0 and T. Note that we calculate the maximum just between t_j and t_{j+1}.

In the following, we will note :

$$\mathscr{E}(j,k) = |U_j^k - U_{ex}(t_j)| + \max_{t\in[t_j,t_{j+1}]} |U_k(t) - U_{ex}(t)|$$

3.2 Application in Python and C++

During the project, we have already implemented the parareal method in Python and obtained some results that will be presented in this section. One of the objectives of the internship was to implement it in C++ using MPI. In this section, as for the project, we will only focus on the following ODEs : Harmonic oscillator (see Section 2.1) and Lorenz system (see Section 2.2).

3.2.1 Results

After implementing the parareal method in C++ and Python we saved the results at each iteration in csv files to see the results. For the results of the parareal method implemented in Python as for the one implemented in C++, the reading of the csv files and the display of the results is done with Python.
For the two integrators of the Parareal method, we will take Runge Kutta order 4 method (see Section 2.5) with a small time step for F and a larger one for G.

- **Harmonic oscillator :**
  We want to apply the parareal method on the harmonic oscillator. With the previous example of the Section 2.1, we have the following parameters with 3 process :

$$x(0) = 0, \quad v(0) = 1, \quad \omega_0 = 5, \quad x_0 = \frac{-1}{5}, \quad \phi_0 = \frac{\pi}{2}$$

  We then obtain the following result with the parareal method in Python according to the iterations of the method :

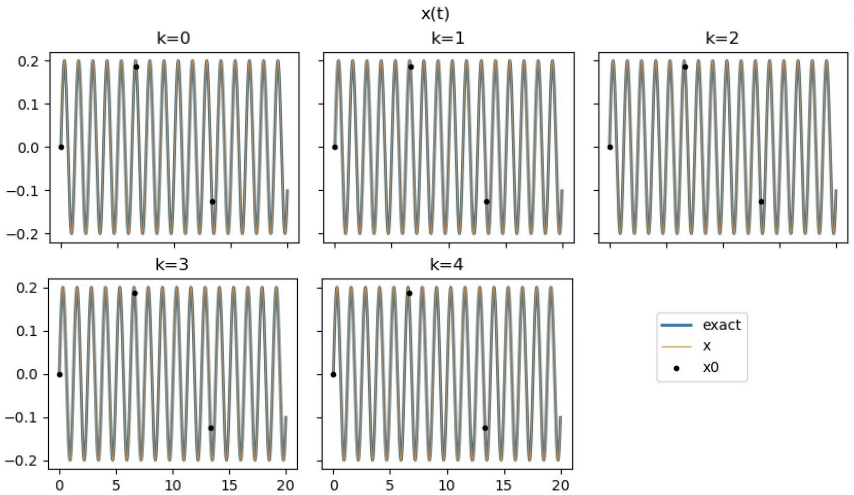


Figure 3.2: Exact solution vs Approximate solution for given parameters (in **Python**)

We see that the system seems to converge in 5 iterations, i.e. between the iterations $k = 4$ and $k = 5$ the points U_n^k are equal to almost one ϵ.

- **Lorenz system :**

  We choose the following parameters :

$$\sigma = 10, \quad b = \frac{8}{3}, \quad r = 28, \quad X_0 = (5,5,5)$$

$$t_0 = 0, \quad T = 20, \quad P = 4, \quad \Delta t_G = 0.1, \quad \Delta t_F = 0.01$$

  For this choice of parameters, we have a butterfly wing pattern (see Figure 4.13) and we have the three following curves (see Figure 3.4) for the representation of the 3 variables x, y and z as functions of time.

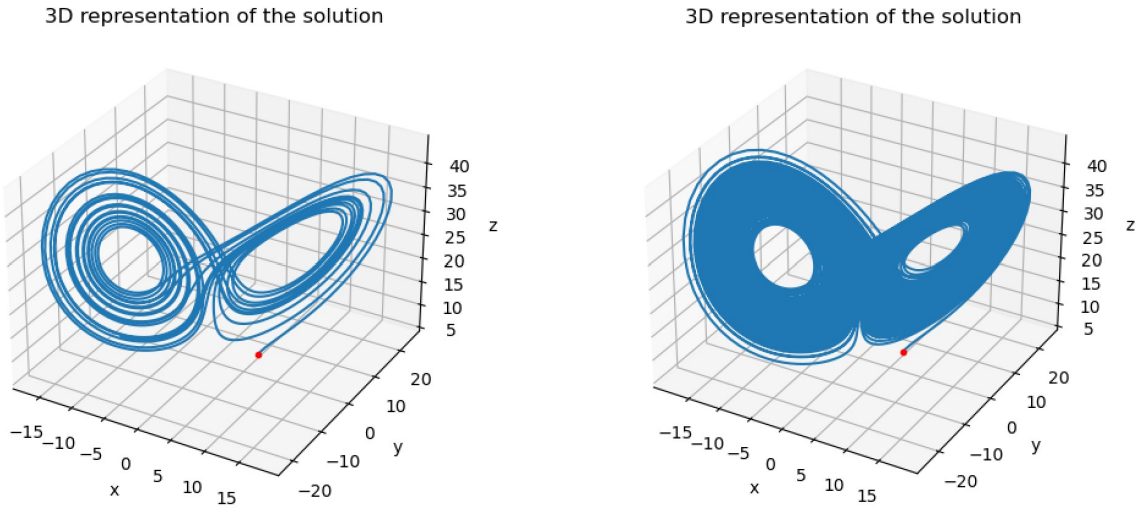


Figure 3.3: Solution 3D of the Lorenz system with given parameters ($T = 20$ and $T = 200$)

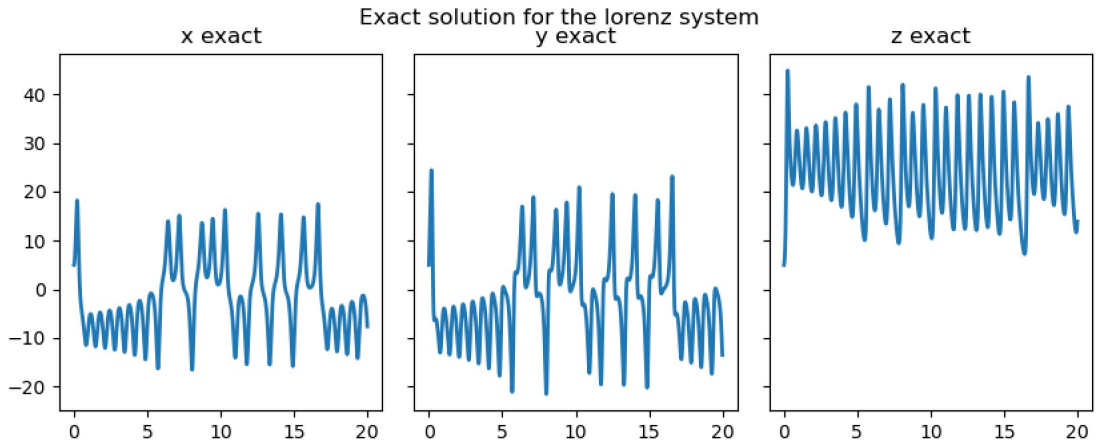


Figure 3.4: Solution of the Lorenz system with given parameters

We will now apply the parareal method in **C++** with the previous parameters for different numbers of processes (from 1 to 4 processes : see Figures 3.5, 3.6, 3.7 and 3.8). We are going to be interested here only in the variable x.

- With 1 process we see (as explained in the "Remarks" of the section 3.1.2) that there are only two iterations and that they are similar (because they start from the same initial point and there is no other initial point).

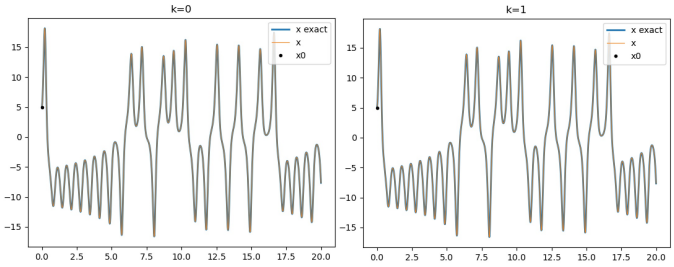


Figure 3.5: Parareal method on the Lorenz system with 1 process

- With two processes we see that the solution converges in 2 iterations.

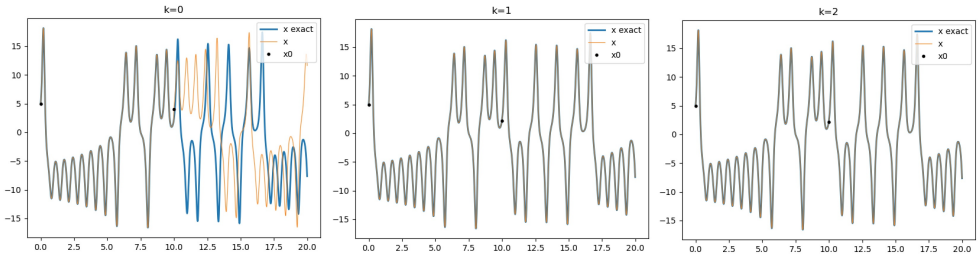


Figure 3.6: Parareal method on the Lorenz system with 2 processes

- With three processes we see that the solution converges in 3 iterations.

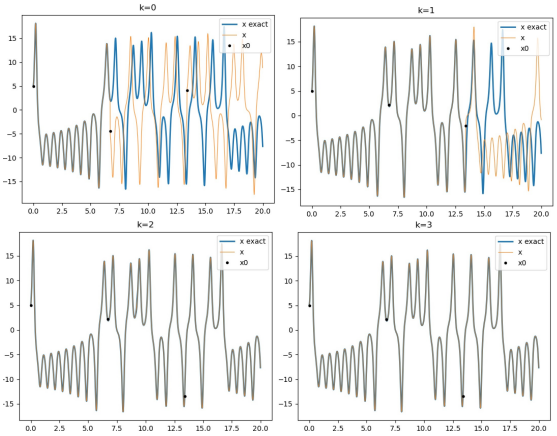


Figure 3.7: Parareal method on the Lorenz system with 3 processes

- With four processes we see that the solution converges in 4 iterations.

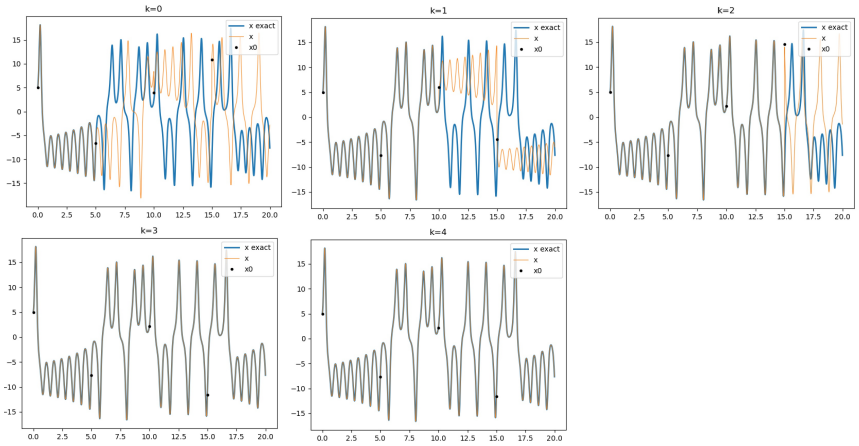


Figure 3.8: Parareal method on the Lorenz system with 4 processes

3.2.2 Speed-up

The main objective of parallel methods is to reduce execution time by simultaneously performing tasks that can be done at the same time. In this section, we will study the speed-up achieved by the parareal method when the number of processes is increased. We will look at the speed-up with the Python implementation (done during the project) and the C++ implementation.

- **Method implemented in Python :**
  The table below (Table 5) contains the execution time first of the iterative method with RK4 (the sequential method) and the execution time for different numbers of processes (parareal method) and this for various fine and coarse time steps by taking the following parameters for the Lorenz system :

$$\sigma = 10, \quad b = \frac{8}{3}, \quad r = 28, \quad X_0 = (5,5,5), \quad t_0 = 0, \quad T = 200$$

Δt	Seq	1 proc	2 proc	3 proc	4 proc
F : 0.0025 G : 0.025	20s	24s	11s	12s	15s
F : 0.00125 G : 0.0125	1m26	1m40	1m13	47s	52s
F : 0.001 G : 0.01	1m44	2m55	1m26	1m9	1m24

Table 2: Execution time with **Lorenz system** for various time steps in **Python**.

Observations :

- It seems that the time in sequential and with the parareal method with 1 process is not the same. Moreover for the different steps calculated, we have that the execution time with 1 process is greater than for the sequential method. This can be explained by 2 main observations. First of all, we have seen previously that there are necessarily 2 iterations which are done on $[t_0, T]$ contrary to the sequential method where there is only one. This problem is easily solved if we compute only the fine and coarse solutions at the first iteration (because the point U_0^k is identical for all k). Thereafter, the parareal method also computes the coarse solution which is not needed at all.
- It seems that the time for the parareal method with P processes is not the time of the sequential method divided by P. It seems obvious that (with intervals of the same length for each process), the time for the parareal method is only divided by P if there is only one iteration, which is never the case.
- It seems that if we increase the number of processes, the time does not necessarily decrease (as between $P = 3$ and $P = 4$ for $\Delta t_F = 0.001$ and $\Delta t_G = 0.01$), so we have to find the right balance between the parameters Δt_F, Δt_G, $[t_0, T]$ and P.

- **Method implemented in C++ :**
  First of all an interesting observation about the implementation of the parareal method in C++ is that it is much faster than Python (see Table 3).
  For example let's take the following parameters for the Lorenz system :

$$\sigma = 10, \quad b = \frac{8}{3}, \quad r = 28, \quad X_0 = (5,5,5)$$

$$t_0 = 0, \quad P = 4, \quad \Delta t_G = 0.01, \quad \Delta t_F = 0.001$$

  Here are the execution times obtained in Python and C++ with differents final time T :

Python : $T = 200$	C++ : $T = 200000$
1m24s	1m15s

Table 3: Difference of the execution times between Python and C++

It can be seen that these times are of the same order while the problem in C++ is 1000 times larger than that in Python. This implies that, to see a speed-up, we must focus on much larger problems. For this we will take into account three parameters

- the size of the interval: T (we will always take $t_0 = 0$)
- the precision of the fine time step : Δt_F
- the size of the problem to solve with the parareal method (in our case the Lorenz system)

We will consider below two different problems :

- **First problem : Harmonic oscillator.** (see Section 2.1)
  We will consider the harmonic oscillator with the following parameters :

$$x(0) = 0, \quad v(0) = 1, \quad \omega_0 = 5, \quad x_0 = \frac{-1}{5}, \quad \phi_0 = \frac{\pi}{2}$$

  As the problem here is a rather small one we will increase its equations tenfold to be able to see a speed-up without having to take T too large. In other words, instead of having 2 equations and thus 2 unknowns during the resolution we will have $2 * d$ equations where d is a chosen integer.
  For the parareal method, we will take the following parameters (where T and d will change):

$$t_0 = 0, \quad T = \cdot, \quad \Delta t_G = 0.01, \quad \Delta t_F = 0.001, \quad d = \cdot$$

  Example 1 : We start by taking $T = 10000$ and $d = 100$, this is the execution times obtained :

·	Seq	1 proc	2 proc	3 proc	4 proc
T : 10000 d : 100	17s	36s	30s	29s	24s

Table 4: Example 1 for the **harmonic oscillator**.

  It seems that when we apply the parareal method with only 1 process it is slower than with 2, 3 or 4 processes. However the time taken by the parareal method with 1 process is almost 2 times slower than with RK4 in sequential. However, the correct solution is obtained in all 5 cases:

$$x(T) = -0.199968 = \frac{-1}{5}\cos\left(5T + \frac{\pi}{2}\right)$$

  Example 2 : We will now double the number of equations ($d = 200$) while keeping the same final time ($T = 10000$) :

·	Seq	1 proc	2 proc	3 proc	4 proc
T : 10000 d : 200	28s	59s	49s	48s	41s

Table 5: Example 2 for the **harmonic oscillator**.

We make the same observation as before concerning the speed-up. However, we can see that by doubling the work to be provided by our methods, the execution times are not doubled. However, as in the previous example, we get the right solution in all 5 cases:

$$x(T) = -0.199968 = \frac{-1}{5}\cos\left(5T + \frac{\pi}{2}\right)$$

Example 3 : Using the first example and doubling the final time ($T = 20000$), we obtain :

·	Seq	1 proc	2 proc	3 proc	4 proc
T : 20000 d : 100	34s	1m10s	58s	58s	1m

Table 6: Example 3 for the **harmonic oscillator**.

We can make the same observations as before and the solution obtained is correct:

$$x(T) = 0.00714986 = \frac{-1}{5}\cos\left(5T + \frac{\pi}{2}\right)$$

Explanation :
As explained in the examples it seems that the parareal method in C++ is not faster than the sequential version (with RK4) but on the contrary slower. This could be explained by the initialization which is in fact an additional iteration :

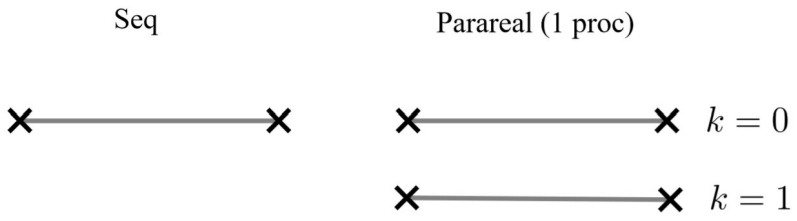


Figure 3.9: Explanation : Sequential vs Parareal with 1 process

However, we can see that the parareal method with 2, 3 or 4 processes is faster than with a single process but still does not seem to be faster than the sequential method. Moreover as presented before there is a speed-up of the method with Python.
For that we can make several hypothesis which will be presented in the Section 3.2.5.

- **Second problem : Lorenz system.** (see Section 2.2) We will now consider the Lorenz system with the following parameters :

$$\sigma = 10, \quad b = \frac{8}{3}, \quad r = 28, \quad X_0 = (5,5,5)$$

As for the harmonic oscillator, the problem here is rather small, so we will increase the number of equations tenfold to be able to see an acceleration without having to take T too large. In other words, instead of having 3 equations and thus 3 unknowns during the resolution, we will have $3 * d$ equations where d is a chosen integer.
For the parareal method, we will take the following parameters (where T and d will change):

$$t_0 = 0, \quad T = \cdot, \quad \Delta t_G = 0.01, \quad \Delta t_F = 0.001, \quad d = \cdot$$

Example 1 : We start by taking $T = 200$ and $d = 10000$, this is the execution times obtained :

·	Seq	1 proc	2 proc	3 proc	4 proc
T : 200 d : 10000	53s	1m46s	1m29s	1m25s	1m27s

Table 7: Example 1 for the **Lorenz system**.

It seems as for the harmonic oscillator that when we apply the parareal method with only 1 process, it is slower than with 2, 3 or 4 processes. However the time taken by the parareal method with 1 process is almost 2 times slower than with RK4 in sequential. The results also seem to be correct in all cases since they are similar to those obtained by RK4 in sequential:

$$(x(T), y(T), z(T)) = (-5.65157, -1.7475, 28.839)$$

Example 2 : We now divide by 2 the size of the problem ($d = 5000$) while keeping the same $T = 200$:

·	Seq	1 proc	2 proc	3 proc	4 proc
T : 200 d : 5000	25s	53s	44s	42s	44s

Table 8: Example 2 for the **Lorenz system**.

We can make the same observations as before but we notice this time that the size of the problem being divided by 2 the execution times are also divided by 2 in the 5 cases. We also obtain the same results:

$$(x(T), y(T), z(T)) = (-5.65157, -1.7475, 28.839)$$

Example 3 : We will now look at a completely different case where we take a larger final time $T = 500$ and fewer equations $d = 1000$. Here are the execution times obtained:

·	Seq	1 proc	2 proc	3 proc	4 proc
T : 500 d : 1000	13s	27s	22s	23s	22s

Table 9: Example 3 for the **Lorenz system**.

As far as execution times are concerned, there are not really any new observations to make. However, the results are different this time. Indeed, in the first 3 cases (sequential and parareal with 1 and 2 processes) we obtain the same results:

$$(x(T), y(T), z(T)) = (-3.33169, -4.35437, 18.3289)$$

But when we apply the parareal method with 3 or 4 processes we get completely different results. Below is what we get for 3 and 4 processes respectively:

$$3 \text{ proc} : (x(T), y(T), z(T)) = (-0.935968, 0.240612, 21.186)$$

$$4 \text{ proc} : (x(T), y(T), z(T)) = (8.18609, 1.32531, 33.8032)$$

Moreover we can observe that by keeping the same parameters as before but by refining the fine and coarse time steps the results change when using RK4 sequentially. We will take $\Delta t_F = 0.0001$, this is what we obtain :

$$(x(T), y(T), z(T)) = (-0.46317, -2.72402, 22.8864)$$

Explanation :
To try to understand why we can obtain very different results by changing the number of parameters or the precision of the time steps, we have to look at the following new example:
Example 4 : We will consider the Lorenz system with the following parameters :

$$\sigma = 10, \quad b = \frac{8}{3}, \quad r = 28, \quad X_0 = (5, 5, 5)$$

$$t_0 = 0, \quad T = 500, \quad \Delta t_G = 0.1, \quad \Delta t_F = 0.01, \quad d = 1$$

Here we are only interested in the use of parareal with 1 and 2 processes.

→ Applying the parareal method with 1 process, we obtain the results :

$$(x(T), y(T), z(T)) = (-9.63764, -14.9937, 19.9616)$$

These are the same results as with RK4 in sequential.
By taking 2 processes for the parareal method, we obtain very different results:

$$(x(T), y(T), z(T)) = (-14.3485, -15.7886, 33.3809)$$

→ Using the *diff* command of linux to compare 2 files, we realized that the difference that totally changes the final results of the method with 2 processes is a rounding to time $t = 274,04s$. Indeed at this time, we have the following results with 1 and 2 processes :

$$1 \text{ proc} : (x, y, z) = (-13.0404, -9.45555, 36.4169)$$

$$2 \text{ proc} : (x, y, z) = (-13.0405, -9.45558, 36.4169)$$

→ It is a very small difference but 10 seconds later, at time $t = 284,04s$, the results are already starting to be less and less close:

$$1 \text{ proc} : (x, y, z) = (-0.102688, 0.234586, 16.7023)$$

$$2 \text{ proc} : (x, y, z) = (-0.043548, 0.325784, 16.7179)$$

→ This small difference makes that 100 seconds later at $t = 374,04s$, the results are not at all the same :

$$1 \text{ proc} : (x, y, z) = (13.0111, 11.062, 34.8919)$$

$$2 \text{ proc} : (x, y, z) = (-12.7836, -15.694, 29.2703)$$

→ This is how at time T, we obtain totally false results with 2 processes which is probably due to a rounding made at the initial time t_1 of this sub-interval. That is to say that $(x, y, z)(t_1)$ with 1 process is the right value but that $(x, y, z)(t_1) = U_1^1$ is wrongly computed, possibly because of the coarse integrator whose time step is not precise enough.

Thus it seems that the more values to be calculated in the interval, the more errors can accumulate and therefore propagate to give false results. This is why the parareal method is perhaps not adapted to the Lorenz system due to its sensitivity to initial conditions (see section 2.2) which makes that small errors can lead the results diverge totally from the real solution.

3.2.3 Efficiency

The efficiency of a program in parallel allows to test the robustness of the code. By taking a problem of size T and P processes we obtain a certain execution time t_e. We will say that the problem is efficient if by taking a problem of size $2 * T$ and $2 * P$ processes the new execution time is in the range $[0.9t_e, 1.1t_e]$. Here we will try to determine if the method implemented in C++ is efficient. Let's go back to Examples 1 and 3 of the harmonic oscillator in the Section 3.2.2. So the parameters are as follows in the 2 examples:

$$x(0) = 0, \quad v(0) = 1, \quad \omega_0 = 5, \quad x_0 = \frac{-1}{5}, \quad \phi_0 = \frac{\pi}{2}$$
$$t_0 = 0, \quad T = \cdot, \quad \Delta t_G = 0.01, \quad \Delta t_F = 0.001, \quad d = 100$$

In Example 1 we have $T = 10000$ and in Example 3 we double the final time so $T = 20000$, we then get the following execution times:

	First case	Second case
Example 1 $T = 10000$	Seq : 17s 1 proc : 36s	2 proc : 30s
Example 2 $T = 20000$	2 proc : 58s	4 proc : 1m

Table 10: Two tests of the efficiency of the parareal method in C++.

It would seem that the method is not efficient and that doubling its size and the number of processes increases the execution time significantly.

Explanation/Hypothesis :
We will now try to understand why the method seems not to be efficient :

- In the following, we will consider $t_0 = 0$.
  First case : We will try to understand why the method is not effective in the first case, here is a scheme:

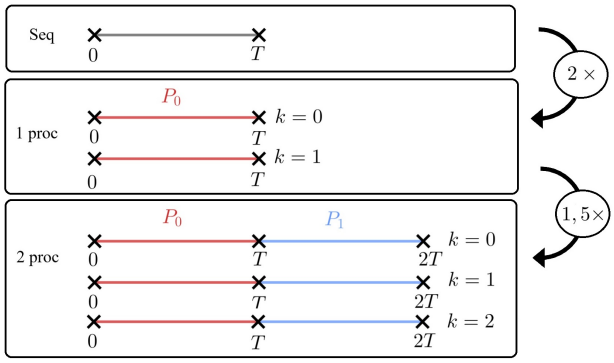


Figure 3.10: Explanation : Sequential vs Parareal with 1 process

We see in the diagram above that RK4 in sequential solves the problem with the fine integrator between 0 and T, i.e. over T seconds.
The parareal method with 1 process performs the initialization ($k = 0$) where it solves the problem with the fine integrator over T seconds and performs a first iteration ($k = 1$) where it solves the same problem once again : it therefore solves the problem over $2 * T$ seconds and should therefore take twice as much time to execute as the sequential version.
To check if the method is efficient, we want to solve the problem between 0 and $2T$ with 2 processes. It solves three times the problem on $2 * T$ with 2 processes (because 2 iterations and the initialization) thus $3 * 2 * T$ seconds and each process solves the problem on $3 * T$ seconds and thus it should take 1.5 times more time than the parareal method with 1 process and 3 times more time than in sequential.
Second case : We will try now to understand why the method is not effective in the second case, here is a scheme:

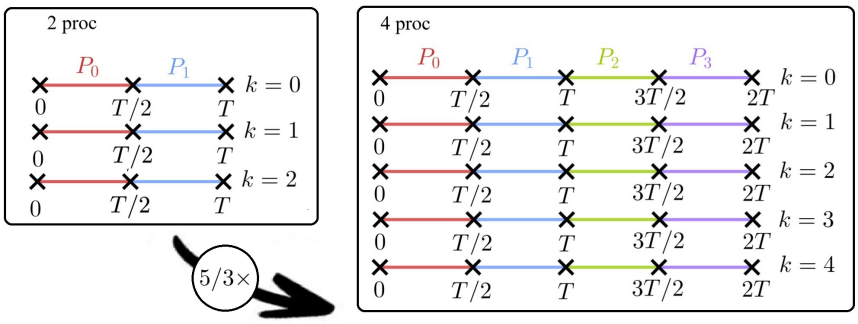


Figure 3.11: Explanation : Sequential vs Parareal with 1 process

This time we want to solve the problem with the parareal method on 2 processes between 0 and T or T seconds. Each process solves the problem on $3 * T/2$ seconds because of the 2 iterations and the initialization.
To check if the method is efficient, we want to solve the problem on $2 * T$ seconds with 4 processes. Each process then solves the problem on $T/2$ seconds 5 times because of the 4 iterations and the initialization thus $5 * T/2$ seconds. So the parareal method with 4 processes should take 5/3 times more time than the method with 2 processes.

- Then, it is also necessary to take into account that the method also uses the coarse integrator which must be used sequentially. These sequential parts also slow down the program and therefore has an impact. Indeed if the fine integrator is used between 0 and T and we double the size of the problem it will have to be computed between 0 and $2 * T$ seconds, which will take twice more time.

- One last point to take into account, even if it is smaller than the others, is that increasing the size of the problem also increases the number of operations and communications to do.

3.2.4 Order of the method

Let us now check the order of convergenc (see Section 3.1.4) of the method on the harmonic oscillator with the following parameters :

$$x(0) = 0, \quad v(0) = 1, \quad \omega_0 = 5, \quad x_0 = \frac{-1}{5}, \quad \phi_0 = \frac{\pi}{2}$$

$$t_0 = 0, \quad T = 200, \quad \Delta t_G = 0.01, \quad \Delta t_F = 0.001, \quad d = 1$$

In the following graph, the blue curve represents the maximum on j of $\mathscr{E}(j,k)$ according to the iterations k and the orange curve represents the $(\Delta t_G)^k$.

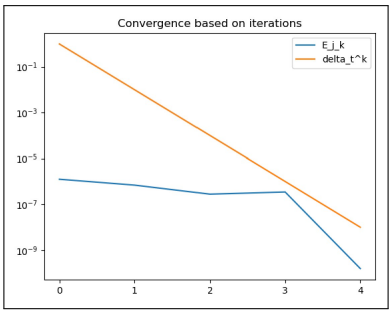


Figure 3.12: Plot of $\mathscr{E}(j,k)$ and $(\Delta t_G)^k$ in **Python**.

We can see that the blue curve is below the orange curve for all the k iterations and so for the parameters fixed previously on the oscillator, the method is of order k.

3.2.5 Conclusion for parareal method

- A first improvement for the implementation of the method could be to make a separate case for $P = 1$. For this case, we would only compute the fine solution once on $[t_0, T]$.

- As said before, one possibility to improve the performances would be to compute only the fine and coarse solutions between t_0 and t_1 at the first iteration because they are the only ones that never change during the iterations.

- In the same way, we could never compute the coarse solution between t_{P-1} and t_P because we never use it and we can compute the fine solution on this interval only at the last iteration (i.e. when the solution converges) because we need this information only to obtain the final solution of the system.

- We could see what happens if we take a lot more intervals that are spread over the different processes. Because here we are limited as the number of processes is equal to the number of intervals. We can ask ourselves if we would have had an improvement for the speed of the method or on the contrary if we would have had many more iterations because the initial points would be much less precise.

3.3 Solving PDEs with Feel++

In this section our goal will be to apply the parareal method to the resolution of the heat equation (see Section 2.4) using Feel++ which is a solver developed by Cemosis (see Section 1.1) that uses the finite element method.
To solve the heat equation, we will use the finite element method which is a numerical method for solving PDEs and which is based on the approximation of the solution in appropriate Hilbert spacess (L^2 or H^1 wery often) which are spaces of infinite dimension. For these spaces, there are bases of functions which allow to make a projection of the equations on finite dimension subspaces and to reduce the problem to find the solution to the projected equation. Before starting to solve the heat equation by the parareal method using Feel++, we will first try to use Feel++ to solve the Laplace equation [9].

3.3.1 Laplace equation

We are interested in this section in the conforming finite element approximation of the Laplacian problem :

$$\begin{cases} -\Delta u = f & \Omega \\ u = g & \Gamma_D \\ \dfrac{\partial u}{\partial n} = h & \Gamma_N \\ \dfrac{\partial u}{\partial n} + u = l & \Gamma_R \end{cases} \tag{3}$$

We deduce the following weak formulation:

$$\begin{aligned} -\int_\Omega \Delta u v &= \int_\Omega f v \\ \int_\Omega \nabla u \cdot \nabla v - \int_{\partial\Omega} \frac{\partial u}{\partial n} v &= \int_\Omega f v \\ \int_\Omega \nabla u \cdot \nabla v + \int_{\Gamma_R} uv &= \int_\Omega f v + \int_{\Gamma_N} hv + \int_{\Gamma_R} lv \end{aligned}$$

Thus we try to find $u : \Omega \mapsto \mathbb{R}$ such that $u \in H^1_{g,\Gamma_D}(\Omega)$ and

$$\int_\Omega \nabla u \cdot \nabla v + \int_{\Gamma_R} uv = \int_\Omega f v + \int_{\Gamma_N} hv + \int_{\Gamma_R} lv \quad \forall v \in H^1_{0,\Gamma_D}(\Omega)$$

i.e. find $u : \Omega \mapsto \mathbb{R}$ such that $u \in H^1_{g,\Gamma_D}(\Omega)$ and

$$a(u,v) = l(v) \quad \forall v \in H^1_{0,\Gamma_D}(\Omega)$$

with

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v + \int_{\Gamma_R} uv, \quad l(v) = \int_\Omega f v + \int_{\Gamma_N} hv + \int_{\Gamma_R} lv$$

The Galerkin method is then used to determine the approximate variational problem. Let (φ_i) a basis of V_h, this means $u_h(x) = \sum_{j=1}^{N_h} u_j \varphi_j(x)$ and then the approximate variational problem is written :
Find $u_h \in V_h$, $a(u_h, v_h) = l(v_h) \quad \forall v_h \in V_h$
and then :

$$\begin{aligned} & a(u_h, v_h) = l(v_h) \\ \iff & \sum_{j=1}^{N_h} u_j a(\varphi_j, v_h) = l(v_h) \qquad \forall v_h \in V_h \\ \iff & \sum_{j=1}^{N_h} u_j a(\varphi_j, \varphi_i) = l(\varphi_i) \qquad \forall i \in \{1, \dots, N_h\} \\ \iff & A u_h = b \end{aligned}$$

with

$$A = (A_{i,j})_{i,j} = (a(\varphi_i, \varphi_j))_{i,j} \quad \text{et} \quad b = (b_i)_i = (l(\varphi_i))_i$$

Implementation with Feel++:
We start by loading the mesh and defining all the elements we will need for the resolution. We can then define the bilinear form a and the linear form l of the variational formulation as well as the boundary conditions (Dirchlet, Neumann and Robin) by imposing by default Dirichlet conditions (see [9]).

In the following we will consider the following geometry and the following mesh:

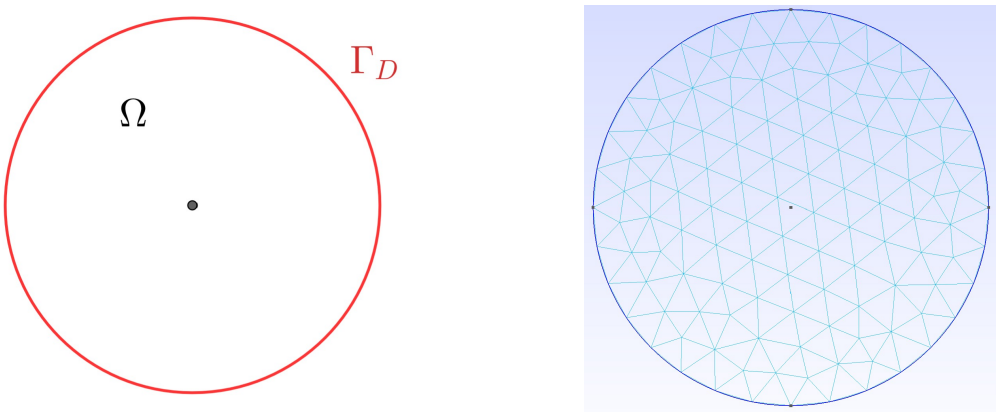


Figure 3.13: Geometry considered with its mesh (with Dirichlet)

Example 1 :
We consider :

$$\begin{cases} -\Delta u = f & \Omega \\ u = g & \Gamma_D \end{cases} \quad \text{with} \quad u_{exact} = g = x^2 + y^2, \quad f = -\Delta u_{exact} = -4$$

We visualize the result with Paraview and we have the expected convergence results with $P^1_{c,h}$:

$$||u-u_h||_{L^2} \sim h^2 \qquad ||u-u_h||_{H^1} \sim h^1$$

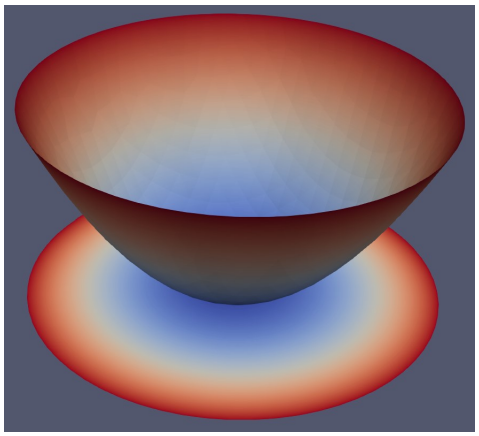


Figure 3.14: Result (with Paraview)

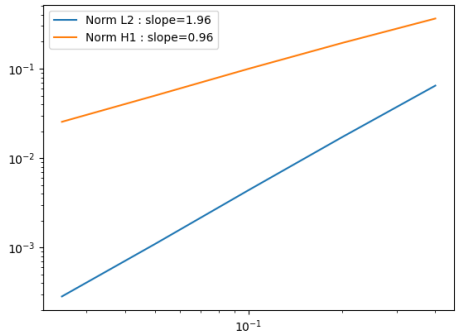


Figure 3.15: Convergence order for the Laplacian problem with $P^1_{c,h}$.

Example 2 :
We consider :

$$\begin{cases} -\Delta u = f & \Omega \\ u = g & \Gamma_D \end{cases} \quad \text{with} \quad u_{exact} = g = x^3 + y^3, \quad f = -\Delta u_{exact} = -(6x+6y)$$

We visualize the result with Paraview and we have the expected convergence results with $P^2_{c,h}$:

$$||u-u_h||_{L^2} \sim h^3 \qquad ||u-u_h||_{H^1} \sim h^2$$

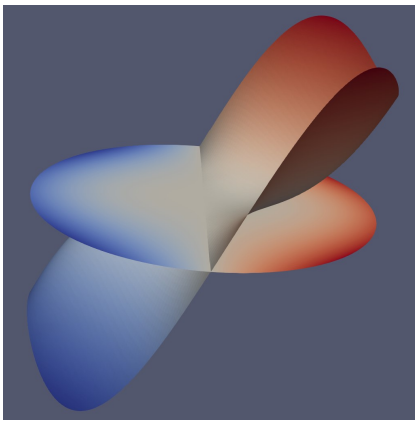


Figure 3.16: Result (with Paraview)

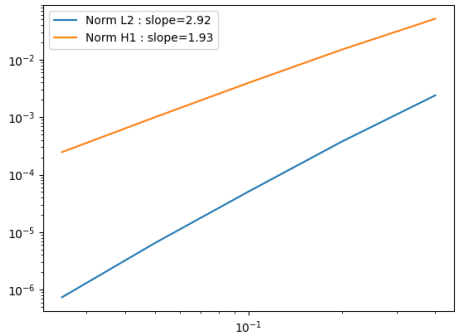


Figure 3.17: Convergence order for the Laplacian problem with $P^2_{c,h}$.

Example 3 :
We consider :

$$\begin{cases} -\Delta u = f & \Omega \\ u = g & \Gamma_D \end{cases} \quad \text{with} \quad u_{exact} = g = x^4 + y^4, \quad f = -\Delta u_{exact} = -(12x^2 + 12y^2)$$

We visualize the result with Paraview and we have the expected convergence results with $P^3_{c,h}$:

$$||u - u_h||_{L^2} \sim h^4 \qquad ||u - u_h||_{H^1} \sim h^3$$

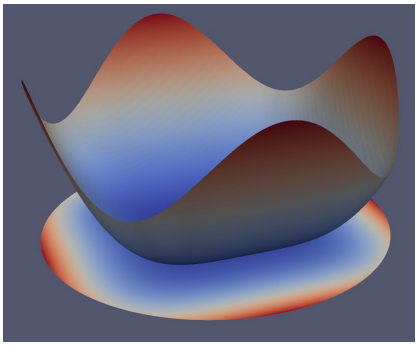


Figure 3.18: Result (with Paraview)

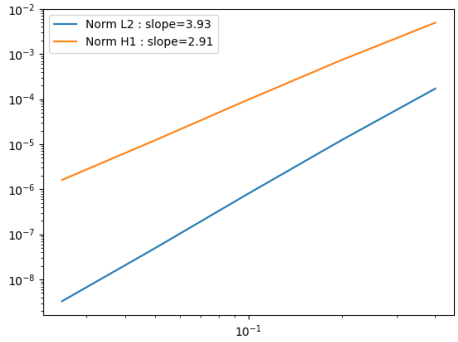


Figure 3.19: Convergence order for the Laplacian problem with $P^3_{c,h}$.

3.3.2 Heat equation

We now want to apply the parareal method to the solution of the heat equation using Feel++. We will only present here the method without presenting numerical results due to time constraints. As presented in the part 2.4, the heat equation with convective effects can be written as:

$$\rho C_p((\frac{\partial T}{\partial t}) + u.\nabla T) - \nabla.(k\nabla T) = Q$$

and it must be completed with boundary conditions and initial conditions.
In this part, we will focus on the heat conduction equation which is written :

$$\rho C_p \frac{\partial T}{\partial t} - \nabla.(k\nabla T) = Q$$

with $\rho = C_p = k = 1$ This gives us :

$$\frac{\partial T}{\partial t} - \Delta T = Q$$

Since we already note T the final time in the parareal method, we will note now on the temperature u and not T. To stay consistent with the Laplace equation, we will note $f := Q$.
Adding the boundary conditions, we obtain :

$$\begin{cases} \dfrac{\partial u}{\partial t} - \Delta u = f & (t_0, T) \times \Omega \\ u = 0 & (t_0, T) \times \partial\Omega \\ u = u_0 & \{0\} \times \Omega \end{cases} \tag{4}$$

We obtain the following weak formulation :
Find $u : (t_0, T) \times \Omega \mapsto \mathbb{R}$ such that $u(t, \cdot) \in H^1_{g,\Gamma_D}(\Omega)$ and

$$\int_\Omega \frac{\partial u}{\partial t}(t,x)v(x)dx + \int_\Omega \nabla u(t,x) \cdot \nabla v(x)dx = \int_\Omega f(t,x)v(x)dx \quad \forall v \in H^1_{0,\Gamma_D}(\Omega)$$

for almost every $t \in (0, T)$.
The major difference with the Laplace equation is that we must, in addition to the spatial discretization, perform a temporal discretization:

$$\int_\Omega \frac{\partial u}{\partial t}(t,x)v(x)dx \simeq \int_\Omega \frac{u^{n+1}(x) - u^n(x)}{\Delta t} v(x)dx$$

We then obtain :

$$\frac{1}{\Delta t}\int_\Omega u^{n+1}(x)v(x)dx + \int_\Omega \nabla u^{n+1}(t,x) \cdot \nabla v(x)dx = \int_\Omega f(t,x)v(x)dx + \frac{1}{\Delta t}\int_\Omega u^n(x)v(x)dx$$

We then pose :

$$a(u^{n+1}, v) = \frac{1}{\Delta t}\int_\Omega u^{n+1}(x)v(x)dx + \int_\Omega \nabla u^{n+1}(t,x) \cdot \nabla v(x)dx$$

$$l(v) = \int_\Omega f(t,x)v(x)dx + \frac{1}{\Delta t}\int_\Omega u^n(x)v(x)dx$$

Method :
For the numerical solution, we have been provided with a class **Heat** allowing the solution of the heat equation with Feel++. When instantiating this class, the constructor allows to define the bilinear form a and the linear form l of the variational formulation as well as the boundary conditions (Dirchlet, Neumann and Robin).
The next objective is therefore to apply the parareal method to the solution of the heat equation using this class. For this, we use the master-slave principle [*].

[*] https://www.cs.sjsu.edu/~pearce/oom/patterns/behavioral/masterslave.htm

We place ourselves in the following test case:

→ The spatial domain is partitioned into 2 sub-domains.

→ We also partition the temporal domain into 2 sub-domains, which makes 2*2 processes for the fine integrator and 2 processes for the coarse integrator, so 6 processes in all. We can then run in parallel with 6 processes.

For that, we will separate our 6 processes in 3 groups numbered by color $\in \{0,1,2\}$. Each of them has 2 processes with a local rank between 0 and 1. The first group (color=0) is the master and manages the coarse integration between t_0 and T : each process of this group deals with a partition of the domain on $[t_0, T]$. The second and third groups are the slaves and handle the fine integration: the second group (color=1) handles the fine integration between t_0 and $T/2$ (with each of its processes handling a part of the spatial domain) and similarly the third group handles the fine integration between $T/2$ and T. Here is a small diagram:

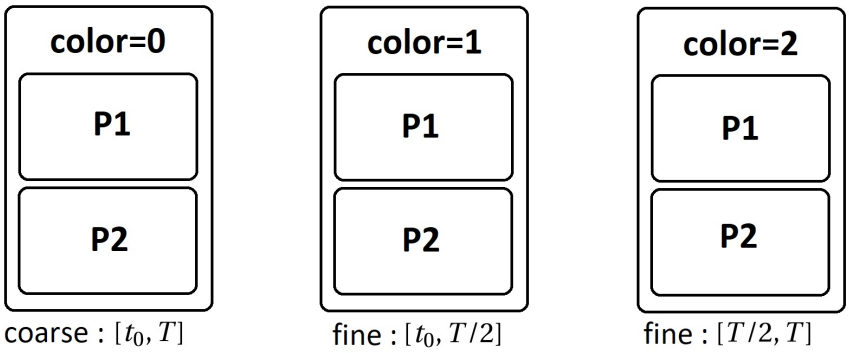


Figure 3.20: Explanation of the test case

To be able to apply the parareal method, we need to communicate between the processes of the same spatial domain, i.e. the processes of the same local rank in each group must be able to communicate with each other:

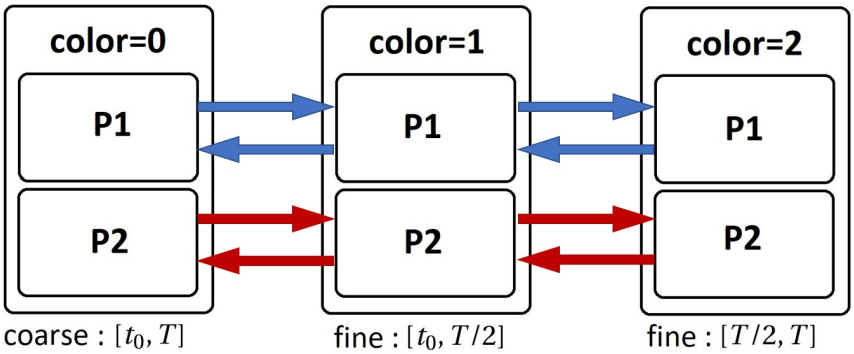


Figure 3.21: Communication between groups

For that we will use the group class of boostmpi [*] which allows to create a new communicator containing only the processes of this group. Here we are going to group the processes which manage the same spatial domains, i.e. the processes of the same local rank :

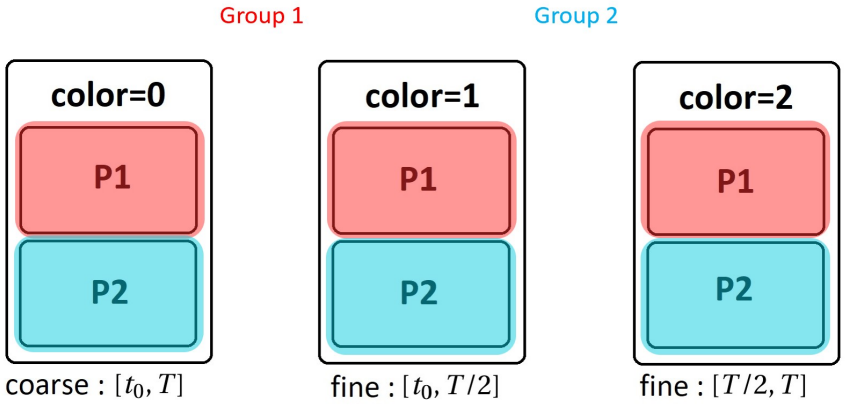


Figure 3.22: Group for communications

Result :
Let $t_0 = 0$ and $T = 1$.
We consider the following geometry :

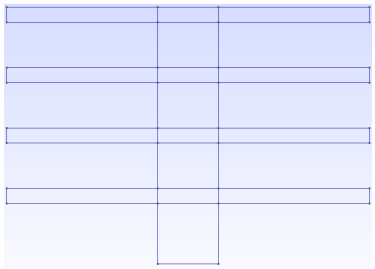


Figure 3.23: Geometry considered

and the following mesh :

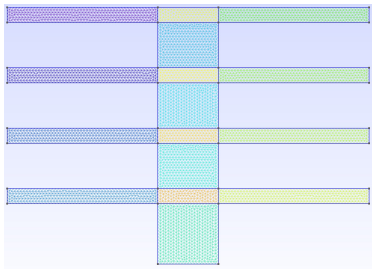


Figure 3.24: Mesh

[*] https://www.boost.org/doc/libs/1_62_0/doc/html/boost/mpi/group.html

Concerning the results, we think that there is a problem with the initial point in group 2, it seems that it does not take into account the final value of group 1 but that it takes 0 as initial point. Here is what we get for $t = 0.50s$ (final time of group 1) and at $t = 0.51s$ (a fine time step after the initial time of group 2):

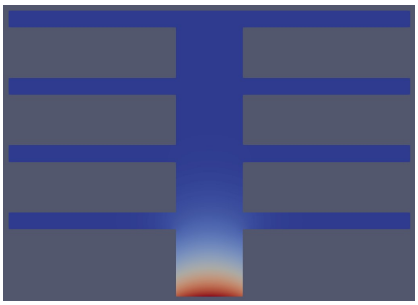


Figure 3.25: Solution : 0.50s
(visualization with Paraview)

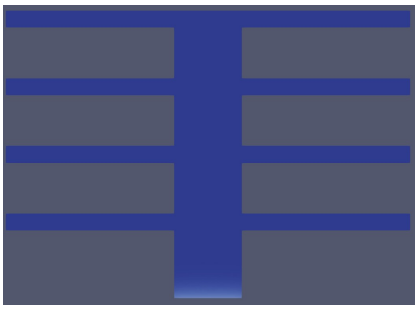


Figure 3.26: Solution : 0.51s
(visualization with Paraview)

This last part is still under development.

4 Data assimilation

4.1 Introduction

Data assimilation is nowadays widely used to make predictions in complex systems, for example in weather forecasting or ocean simulation. Data assimilation is a method that combines observations with the output of a model to improve a prediction. The main idea is to combine information from our model and from observations, in order to have a more reliable analysis. Sometimes the output of the model may not be in the same space as the observations. This is something that we need to consider while doing our assimilation. This means that there will be uncertainties coming from the model or from the observations, as were asthese uncertainties coming from our input that will also translate into uncertainties in our output. At the end of the data assimilation we will obtain an output which will be an estimate of the unknown state variables. The best estimate is searched for as a linear combination of the background estimate and the observations:

$$x^a = Lx^b + Ky^0$$

Data assimilation methods are often split into two families: statistical methods and a variational methods. For the internship we will be focusing on the statistical methods.

4.2 Statistical approach

4.2.1 Kalman filter

The Kalman filter method consists in looking for an analyzed state x^a . This analysis will be a linear combination of the outputs of the model and observations. To explain this method let's consider that we observe a single quantity, an estimation of a scalar quantity at a point in space. For example we are observing the temperature in the middle of the room, and the model also outputs the temperature in the middle of the room. We will then have :

$$x^a = x^b + K(y - x^b)$$

with $x^a = \begin{pmatrix} x_1^a \\ \vdots \\ x_n^a \end{pmatrix}$ the analysis, $x^b = \begin{pmatrix} x_1^b \\ \vdots \\ x_n^b \end{pmatrix}$ the state background or model output, $y = \begin{pmatrix} y_1 \\ \vdots \\ y_p \end{pmatrix}$ the observation and K the gain matrix of size $p \times p$.
To simplify let's consider that we are trying to find the true state in the scalar setting (1D), we suppose that the true state x^t exists so:

$$x^a - x^t = x^b - x^t + K(y - x^t - x^b + x^t).$$

Let's define the errors:

$$\epsilon^a = x^a - x^t,$$
$$\epsilon^b = x^b - x^t,$$
$$\epsilon^y = y - x^t.$$

So we will have:

$$\epsilon^a = \epsilon^b + K(\epsilon^y - \epsilon^b).$$

If we have many realisations of these error, then we can write:

$$<\epsilon^a>=<\epsilon^b>+K(<\epsilon^y>-<\epsilon^b>).$$

We want to have the analysis error variance as low as possible .So we want to minimize $<(\epsilon^a)^2>$ with respect to K, this will give us:

$$<(\epsilon^a)^2>=<(\epsilon^b)^2>+K^2<(\epsilon^y-\epsilon^b)^2>+2K<\epsilon^b(\epsilon^y-\epsilon^b)^2>,$$

$$2K<(\epsilon^y)^2+(\epsilon^b)^2>-2<(\epsilon^b)^2>=0.$$

We assume that the errors in the background and observation are uncorrelated,wh,ch leads to

$$K=\frac{<(\epsilon^b)^2>}{<(\epsilon^b)^2>+<(\epsilon^y)^2>}\Rightarrow K=\frac{(\sigma^b)^2}{(\sigma^b)^2+(\sigma^y)^2},$$

where $(\sigma^y)^2$ is the observations error variance and $(\sigma^b)^2$ is the background or model error variance.
If we have $(\sigma^y)^2=0$, $K=1$ and $x^a=y$ this means that the observation are perfect.
And if $(\sigma^b)^2=0$, $K=0$ and $x^a=x^b$ this is equivalent to ignoring the observations.

Now that we have explained the method for finding x^a in the 1D case, let's try to generalize our formula in a multi-dimensional case:

$$\begin{cases} x^a=(I-KH)x^b+Ky^0=x^b+K(y^0-H(x^b)), \\ K=BH^T(HBH^T+R)^{-1}. \end{cases}$$

With K the gain or weight matrix, $(y^0-H(x^b))$ the innovation and H the linear operation of the observations. This formulation is called the Best Linear Unbiased Estimator (BLUE) or least squares analysis. The principle of the Kalman filter is based on this formulation. Here is

a small figure which illustrates the Kalman filter.

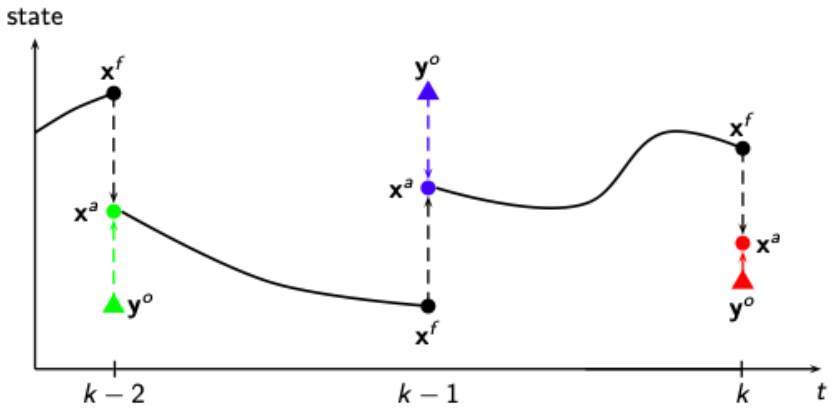


Figure 4.1: Kalman filter

The general idea consists in estimating the state at time k from an estimate at time $k-1$ and measurements at time k. We do the estimation in two steps:

- Prediction of the state from the evolution model
- Correction of the prediction from the measurements

4.2.2 Kalman filter algorithm

Let us precise the notations we will use.
Let $n \in \mathbf{N}$ be the state's size, $k \in \mathbf{N}$, $x_k^f, x_k^a \in \mathbf{R^n}$ and $P_k^f, P_k^a \in \mathbf{R^{n \times n}}$:

- $k \in \mathbf{N}$ the time index
- $x_k^f \in \mathbf{R^n}$ forecast state (background), forecast error covariance matrix $P_k^f \in \mathbf{R^{n \times n}}$
- $x_k^a \in \mathbf{R^n}$ analyzed state (result of the assimilation process), analysis error covariance matrix $P_k^a \in \mathbf{R^{n \times n}}$

Let us define the operators:

- model operator $M_{k,k+1}(x_k^t)$ model error $\eta_{k,k+1}$, covariance matrix $Q_k \in \mathbf{R^{n \times n}}$
- observation operator $H_k(x^t) : \mathbf{R^n} \rightarrow \mathbf{R^n}$, observation error $\epsilon^0 \in \mathbf{R^n}$, covariance matrix $R_k \in \mathbf{R^{n \times n}}$

The hypotheses necessary for the application of the Kalman filter are:

- Model and observations operators $M_{k,k+1}$ and H_k are linear.
- Errors are unbiased, Gaussian, independent and white in time. For example for the observation we will have $< \epsilon_k^0 \epsilon_j^{0T} >= 0$ if $k \neq j$:.

So finally we obtain the Kalman filter algorithm.

1 Initialization: x_0^f and P_0^f are given, equal to x^b and B;
2 BLUE:

$$K_k = (H_k P_k^f)^T [H_k (H_k P_k^f)^T + R_k]^{-1},$$
$$x_k^a = x_k^f + K_k (y_k^0 - H_k x_k^f),$$
$$P_k^a = (I - K_k H_k) P_k^f;$$

Forecast step:

$$x_{k+1}^f = M_{k,k+1} x_k^a,$$
$$P_{k+1}^f = M_{k,k+1} P_k^a M_{k,k+1}^T + Q_k$$

Algorithm 3: Kalman Filter

4.3 Ensemble Kalman Filter

4.3.1 Explanation of the method

We have seen so far two methods to do data assimilation, these methods are valid only for linear systems, but the Lorenz system is non-linear, that's why we will introduce the Ensemble Kalman Filter method which works well for non-linear systems. The ENKF method consists in using the Kalman filter method in high dimension and compare P by a set of states $x_1, x_2, .., x_m$. So we can approximate the moments of the error by the moments of the sample. For all the samples, we have:

$$x_i^a = x_i^f + K[y - h(x_i^f)]$$

with $h(x_i^f)$ the observation operator.
To begin with we can estimate the forecast error covariance matrix as:

$$P^f = \frac{1}{m-1} \sum_{i=1}^{m} (x_i^f - \bar{x}^f)(x_i^f - \bar{x}^f)^T \quad with \;\; \bar{x}^f = \frac{1}{m} \sum_{i=1}^{m} x_i^f .$$

We can factorized the forecast error covariance matrix by:

$$P^f = X_f X_f^T$$

where X_f is an $n \times m$ matrix whose columns are the normalized anomalies or normalized perturbations,

$$[X_f]_i = \frac{x_i^f - \bar{x}^f}{\sqrt{m-1}}$$

We can also define the Kalman gains:

$$K = P^f H^T (HP^f H^T + R)^{-1}$$

In addition, we have:

$$\bar{x}^a = \frac{1}{m}\sum_{i=1}^{m} x_i^a \quad , \quad [X_a]_i = \frac{x_i^a - \bar{x}^a}{\sqrt{m-1}}.$$

Now we can calculate the normalized anomalies X_i^a :

$$X_i^a = X_i^f + K(0 - HX_i^f) = (I_n - KH)X_i^f$$

with $X_i^f \equiv [X_f]_i$.
We can write the covariance matrix of the analysis errors as:

$$P^a = (I_n - KH)P^f(I_n - KH)^T + KRK^T = (In - KH)P^f.$$

A better choice for the rest may be to add a disruption to the observation: $y \rightarrow y_i + \bar{u}_i$ where u_i is drawn from the Gaussian distribution $u_i \sim \mathcal{N}(0, R)$, $\bar{u}$ will be the mean of the sampled u_i, then we can define the innovation perturbations as :

$$[Y_f]_i = \frac{Hx_i^f - u_i - H\bar{x}^f + \bar{u}}{\sqrt{m-1}}.$$

Finally we can modify the posterior anomalies:

$$X_i^a = X_i^f - KY_i^f = (I_n - KH)X_i^f + \frac{K(u_i - \bar{u})}{\sqrt{m-1}}.$$

4.3.2 Ensemble Kalman Filter Algorithm

input : For k=0,...,K: the observation error covariance matrices R_k, the observation models H_k, the forward models M_k.

1 Initialize the ensemble $\left\{x^f_{i,0}\right\}_{=1,...,m}$;

2 **for** *k=0,...,K* **do**

3 Draw a statistically consistent observation set:
for i=1,...,m: $y_{i,k} = y_k + u_i$, with $u_i \sim \mathcal{N}(0, R_k)$;

4 Compute the ensemble means
$\bar{x}^f_k = \frac{1}{m}\sum_{i=1}^m x^f_{i,k}$, $\bar{u} = \frac{1}{m}\sum_{i=1}^m u_i, \bar{y}^f_k = \frac{1}{m}\sum_{i=1}^m H_k(x^f_{i,k})$
and the normalized anomalies
$[X_f]_{i,k} = \frac{x^f_{i,k} - \bar{x}^f_k}{\sqrt{m-1}}$, $[Y_f]_{i,k} = \frac{H_k(x^f_{i,k}) - u_i - \bar{y}^f_k + \bar{u}}{\sqrt{m-1}}$;

5 Compute the gain: $K_k = X^f_k (Y^f_k)^T \left\{Y^f_k (Y^f_k)^T\right\}^{-1}$;

6 Update of the ensemble:
for i=1,...,m: $x^a_{i,k} = x^f_{i,k} + K_k(y_{i,k} - H_k(x^f_{i,k}))$;

7 Compute the ensemble forecast:
for i=1,...,m: $x^f_{i,k+1} = M_{k+1}(x^a_{i,k})$

8 **end**

Algorithm 4: Ensemble Kalman Filter

4.4 Comparison of Python results with C++

During the project we used a function already implemented in the library Filterpy to perform data assimilation. In the internship we had to implement it in C++. In order to verify the results we obtained with the C++ implementation, we wrote in csv files our observation, model and analized state for each time step and then read them back in Python. Having both results (C++ and Python) in Python makes the comparison easier.
For this case, the model and the observation will be the Lorenz system solved with RK4, we chose to take exactly the same initial point but different parameters for the observations and the model. For (σ, r, b) we have chosen to take $(12., 6., 12.)$ for the observation and $(10., 6., 10.)$ for the model. For the initial condition we have taken for both $(-10, 10, 25)$.
The Lorenz system:

$$\begin{cases} x' &= \sigma(y - x) \\ y' &= x(r - z) - y \\ z' &= xy - bz \end{cases}$$

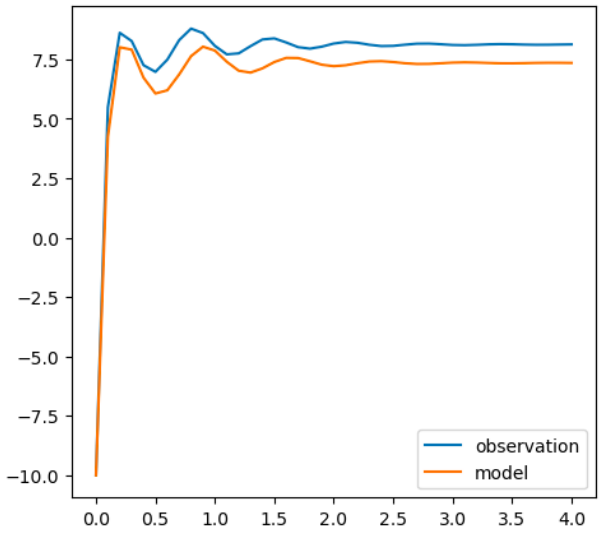


Figure 4.2: Lorenz system example, $(\sigma, r, b) = (12., 6., 12.)$, $X_0 = (-10., -10., 25.)$ for observation and $(\sigma, r, b) = (10., 6., 10.)$, $X_0 = (-10., 10., 25.)$ for the model

Let's take

$$P = \begin{pmatrix} 0.1 & 0. & 0. \\ 0. & 0.1 & 0. \\ 0. & 0. & 0.1 \end{pmatrix}, Q = \begin{pmatrix} 0.1 & 0. & 0. \\ 0. & 0.1 & 0. \\ 0. & 0. & 0.1 \end{pmatrix}, R = \begin{pmatrix} 0.01 & 0. & 0. \\ 0. & 0.01 & 0. \\ 0. & 0. & 0.01 \end{pmatrix}.$$

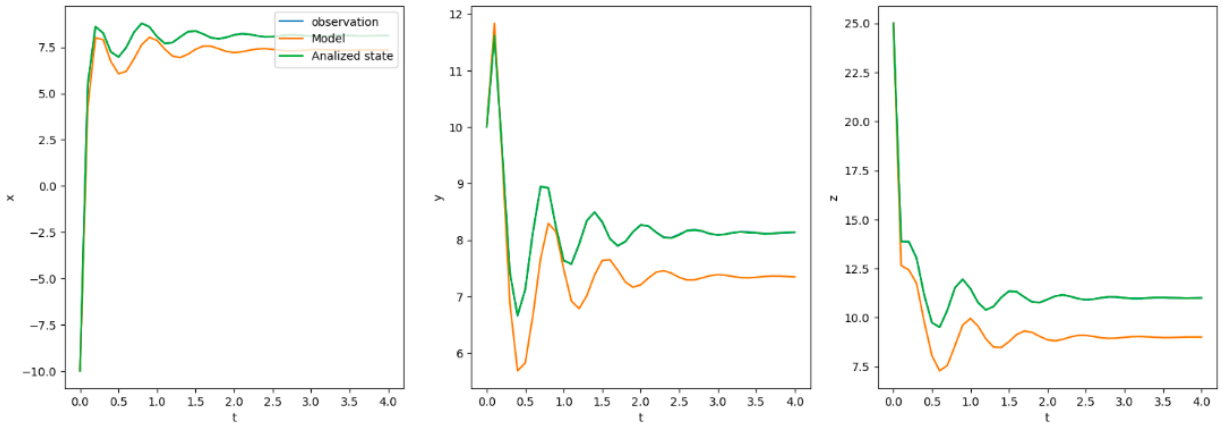


Figure 4.3: Curve of the model, observation and analyzed states with Filterpy

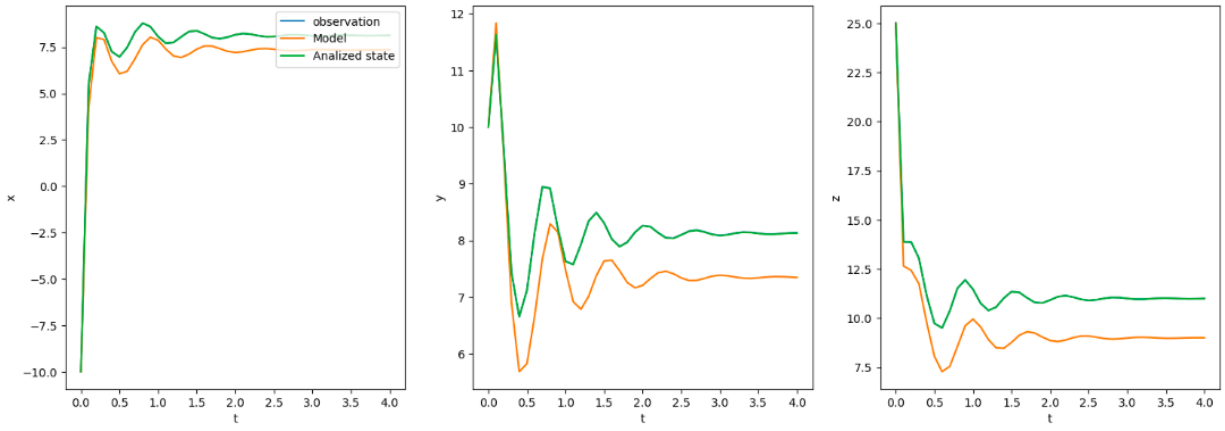


Figure 4.4: Curve of the model, observation and analyzed states with C++

First of all let's try to understand the impact of the covariance matrices associated with the error in the algorithm. For this case we took a covariance matrix associated to the observation R smaller than the other matrices. This refers to the fact that our observations are very precise, and are much closer to the exact solution. For Q, the covariance matrix associated to the model, we have also chosen diagonal matrix, with 0.1 on all the entries, since we solve it with the method of RK4 this one remains quite accurate. And for P, we have chosen diagonal matrix, with 0.1 on all the entries.
We expect to see that our green curve is close to the blue curve which corresponds to the observations. On the other hand we notice that the values of the states after the data assimilation are very close almost similar to the observation ones.
We can also notice that our results obtained with Filterpy and with the implementation made with C++ are very similar. This proves us that our implementation in C++ seems to work properly.

4.5 Integrate data assimilation to Feel++ toolboxes

4.5.1 The context

For the second part of the internship, our goal was to integrate the Ensemble Kalman Filter algorithm to Feel++ toolboxes. For this we had to realize a simulation of an office located in the university of Strasbourg, at the UFR of mathematics on the 2nd floor. This simulation will be the model to be used for data assimilation, and it will return the temperature of the office for a given time and at each point of the room.
To perform data assimilation, apart from a numerical model (obtained with Feel++) we also need observations. For the observation we have 10 sensors in the office that collect the temperature every hour.

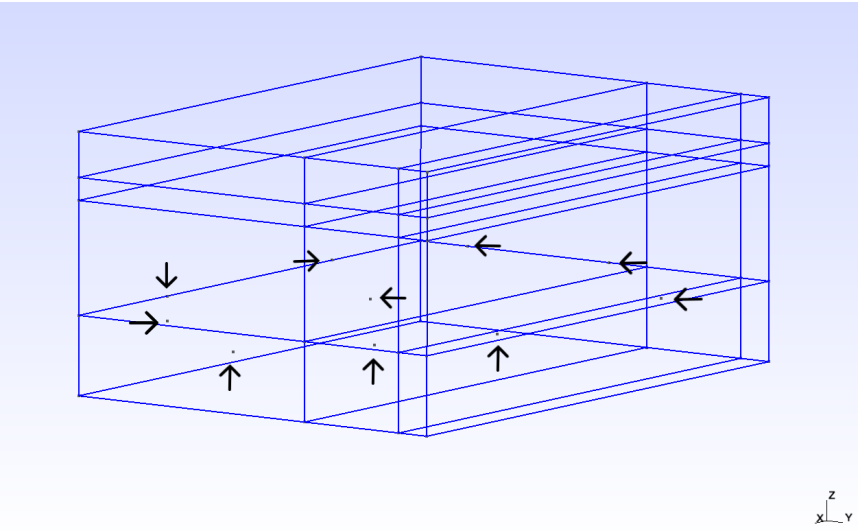


Figure 4.5: Mesh of the office with the 10 sensors

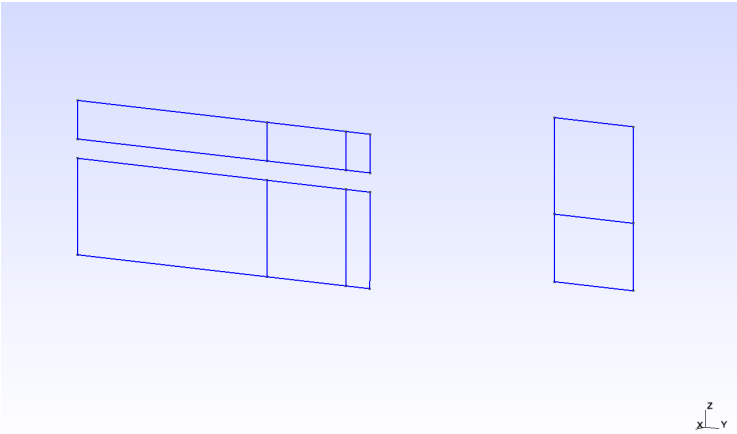


Figure 4.6: Mesh of the office with the 10 sensors:View of the windows and door

4.5.2 Heat flux

In order to simulate the evolution of the temperature in the office, we must first understand the different types of heat transfer. Heat transfer is one of the modes of internal energy exchange between two systems. Heat goes always from a hot body to a cold one and it takes place until the two systems or two bodies are at the same temperature. When the two bodies are at the same temperature, we call this " thermal equilibrium", the two systems are at the same temperature and there is no more heat transfer between the two systems.

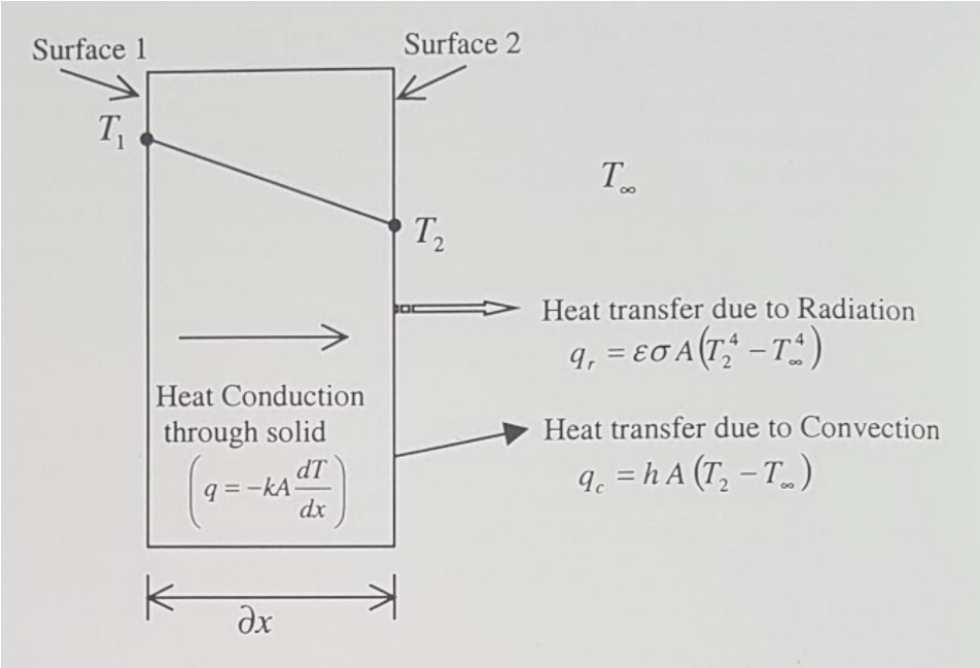


Figure 4.7: Three different modes of heat transfer

There are three kinds of heat transfer: the first is conduction, it is the heat transfer that takes place when two solid bodies are in contact.

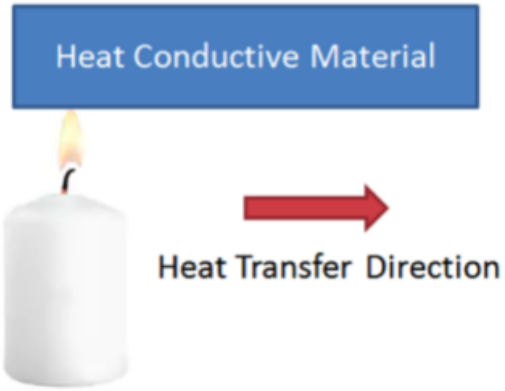


Figure 4.8: Conduction

This is the simplest type of heat transfer and the law that applies to conduction is Fourier's law, which states that the heat flux is of the form $Q = -k\nabla T$. In a building, this kind of heat transfer takes place inside walls, or between any pair of solid objects at different temperatures. The second type of heat transfer is convection. Convection occurs when a fluid participates in the heat transfer.

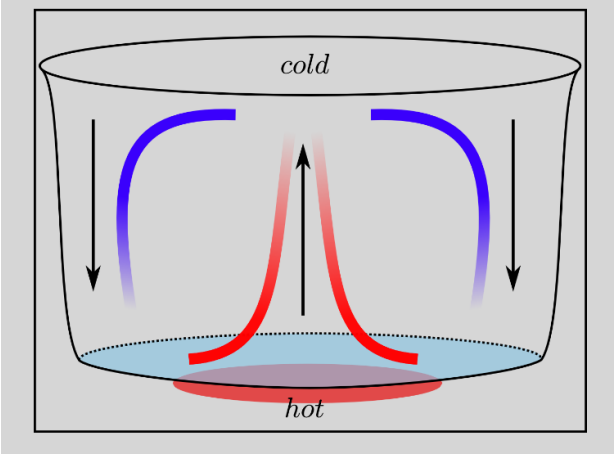


Figure 4.9: Convection

In convection, heat is transferred by the motion of matter. In a building, for example, convective heat exchange takes place when hot/cold air masses enter from the window. An another example can be when we heat water, the hot water which has a lower density at the bottom will go up, and conversely the cold water which is at the top will go down and we will see the appearance of convection rolls.
The third type of heat transfer is the radiation.

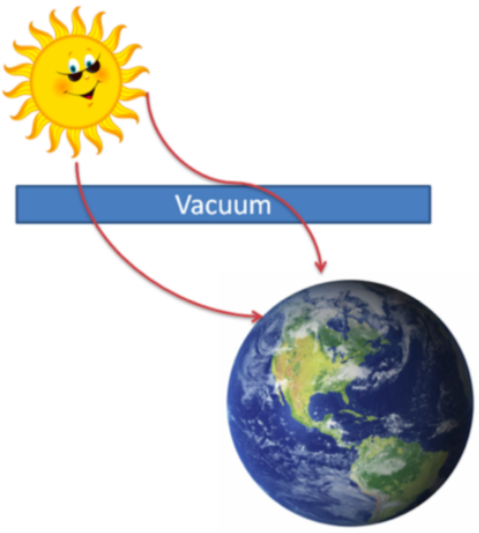


Figure 4.10: Radiation

Radiation is the only energy transfer that takes place in vacuum. It is the light which will transport the energy, it will be transported either as an electromagnetic wave or as a photon. In building simulations, the most important source of heat from radiation is the sun, whose beams hit the building opaque walls and enter the interior space through transparent surfaces. Consequently, any object emits a radiation.

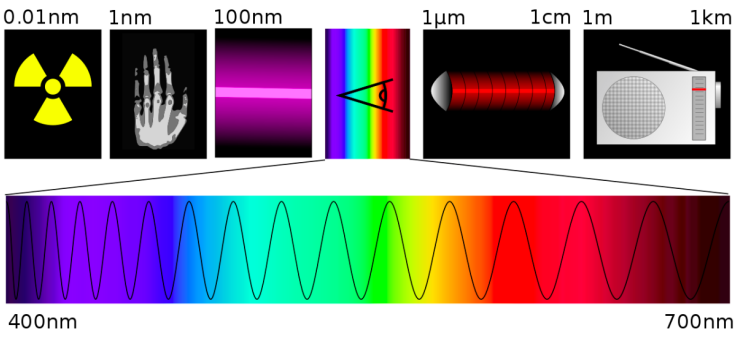


Figure 4.11: Electromagnetic spectrum

4.5.3 The simulation using Feel++ toolboxes

Feel++ is a powerful tool for numerical simulations. Using JSON and CFG files, we can configure models by defining the required parameters like physical quantities, material properties, constants. It is possible to solve several types of problems like stationary or time dependent, linear or non linear. It's also possible with Feel++ to realize solutions with several cores for larger problems.

To study the heat flow in the office we want to simulate, we must first study the heat equation with convective effects.

- Heat equation with convective effects.

$$\rho C_p((\frac{\partial T}{\partial t}) + u \cdot \nabla T) - \nabla \cdot (k \nabla T) = Q$$

  Which is completed with boundary conditions, initial value and with the parameters below.

ρ	Air density	$Kg.m^{-3}$	1.125
C_p	Specific heat	J/KgK	1004
k	Conductivity	W/mK	0.025
u	Fluid velocity	$m.s^{-1}$	unknown

Parameters for the heat equation

The real unknown in our case will be the temperature, but we can see that there is a second unknown in this equation which is the fluid velocity. This unknown comes from the fact that we have chosen to work with the heat equation with convective effects. In order to find this value it will be necessary to use other equations, which are the Navier-Stokes equations.

In fluid mechanics, the Navier-Stokes equations are nonlinear partial differential equations that describe the motion of fluids in the continuous state. They apply to the motion of air in the atmosphere, ocean currents, the flow of water in a pipe, and many other fluid flow phenomena.

- Equation of the air motion (Navier-Stokes).

$$\begin{cases} \rho(\frac{\partial u}{\partial t} + u \cdot \nabla u) - \nabla \cdot (\mu \nabla u) + \nabla P = -\rho_0 \beta (T - T_{ref}) g \\ \nabla \cdot u = 0 \end{cases}$$

ρ	fluid density	$Kg.m^{-3}$	1.125
u	fluid velocity	$m.s^{-1}$	unknown
β	coefficient of thermal expansion	K^{-1}	0.0034
μ	dynamic viscosity	$Pa.s$	$1.81e^{-5}$
g	gravitational acceleration	$m.s^{-2}$	9.8

Parameters for the equation of the air motion

In order to simulate our office we have to solve two different equations, first we have to solve the Navier-Stokes equations in order to find the dynamic viscosity and reinject this value in the heat equation to finally find the temperature.
To achieve this we can first use the Heat Transfer and Fluids toolboxes to find the dynamic viscosity μ, then use the Heat Transfer toolbox by injecting the value found by applying the Fluid toolbox to find the temperature in the room for each time step.
However, for the rest of the internship, in order to facilitate the problem and to save time, we have chosen to solve the heat equation without convection effect.

- Heat equation without convection effect.

$$\rho C_p(\frac{\partial T}{\partial t}) - \nabla.(k\nabla T) = Q$$

  Which is completed with boundary conditions, initial value and the values of the chosen parameters are typical for air.

We chose to take a time step of 10 minutes for the simulation and we put radiation coming from the windows. For the outside temperature we put it at 27,85 degrees celsius, and we initialize the temperature inside the room to 20 celsius.

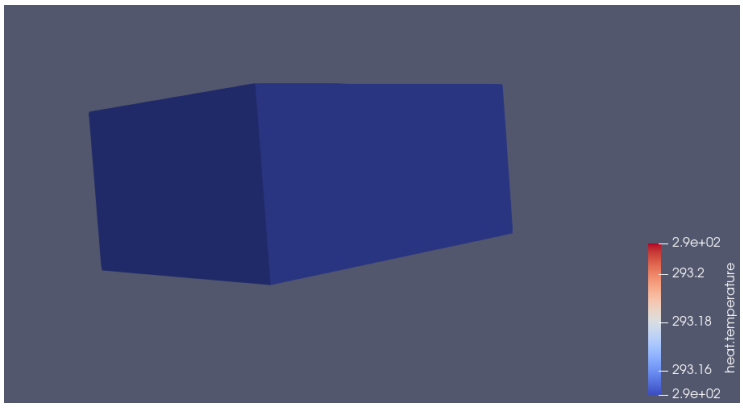


Figure 4.12: Simulation realized with the toolbox heat of Feel++ (time= 0)

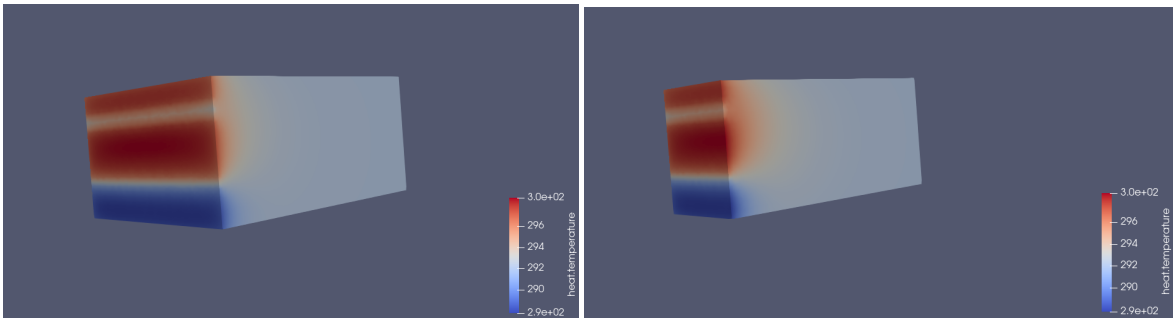


Figure 4.13: Simulation realized with the toolbox heat of Feel++ (time= 1h.)

We notice that at the beginning of the simulation the office is at 20 degrees everywhere, which means that the simulation has taken into account the initial condition that we had given. We notice that after one hour, we have radiation coming from the windows. We can see the heat source beginning to warm up the room. This is due to the fact that during the simulation we had set the radiation factor for the windows. We can also see that the heating of the room is done very slowly. The reason for this is that we have ignored the convective effects.

4.5.4 Including data assimilation to our simulation

In the previous sections we have seen how to build our model using the toolboxes available in Feel++. We have also seen how to get the observations from the 10 sensors located in the office. Now let's try to understand how to apply our Ensemble Kalman Filter class to correct our simulation with our sensor data. To do this let's first try to understand how to use our ENKF class.
The Ensemble Kalman Filter class is composed of 3 major functions.
The first one is the initialization. This one takes in arguments all the necessary parameters in order to initialize them. It takes:

- the dimension of our model, in our case it will be of size 10 because we have 10 sensors in our office;
- the dimension of the vector containing our observations which will also be of size 10 for the same reason;
- we will have a vector X of size 10 which will represent the initial state for our model;
- the matrix P which will be the covariance matrix associated to the analyzed state;
- our time step dt. For our case we have chosen to perform the data assimilation every hour because the observations are available for each hour;
- a function hx which takes a state as argument, and it allows us to pass from the model space to the observation space. For our example they are the same dimension;
- and finally a function fx, which takes as arguments the time, the state, the index of the sample to be simulated and the time step. It returns the state for the next time step;

The second function in our class is the predict function. To summarize, this function will predict the state of each sample for the next time step. This is where our fx function is used. The last function, named update takes as argument an observation vector z, and calculates the analyzed state.

The delicate part to realize the data assimilation was to create the fx function. In the case of Lorenz it was obvious, this function was only applying RK4 to find the state for the next time step. For our case, we have to apply the simulation using as initial point our state in argument. But to implement this function, we had to pay attention to several points.

- First of all, in order to perform simulations using toolboxes in Feel++, we need to give it as initial condition a h5 file where the temperature of the office at each mesh point is stored. When we run simulations in Feel++, the toolboxes automatically generate h5 files for each time step. These files are only readable by Feel++. But the tricky part is that our vector X given as argument in the function fx stores only the temperature at the points where the sensors are located. But in order to realize the simulation we need

instead a vector field where are stored the temperature at each point of the office. We have to modify our h5 file in order to integrate the temperature values at the sensors positions (which are in the X vector). The approach we choose is to apply Gaussian $\frac{1}{\sigma\sqrt{2\pi}} \exp^{-\sum \frac{\|x-\mu\|}{2\sigma^2}}$ around our sensors for a given radius to make a more general and accurate correction.

- After modifying our h5 file with the Gaussians around the sensors, we need to perform the simulation to get our state for the next time step. Therefore thanks to the toolboxes in Feel++, by giving as initial condition our modified h5 file we will be able to get our state for the next time step. This will also be stored as a h5 file, and to pass from the temperature field to point values, we chose to average the finite element field around the sensors.

- The last important point was the fact that we applied the fx function to each sample. As explained before, our Ensemble Kalman Filter applies the Kalman Filter to each sample. But in our case, we have to give him a h5 in order to apply the simulation. The solution we found for this problem is to put in argument to the fx function the sample index. To avoid the files to be overwrite at each simulation, we have created directories for each sample to store the h5 files. And when we will apply our fx function, this one will be able to get the h5 file corresponding to the right sample.

Taking into account all these points, we have implemented the function fx. However after several debugging attempts we did not manage to get it to execute correctly. This function seems to work without returning any error message, but we noticed that we had incorrect values at different places in our vector. Due to the time , we couldn't figure out where the problem was coming from. We think that maybe our Gaussian calculations are falsifying the results, but to be sure, it will be necessary to look in detail at the fx function.

5 Conclusion

Our goal during the project was to implement a parallel time resolution method for the Lorenz system, and to realize the data assimilation using the EnKF method. For the internship which is a continuation of the project, we had to implement these two methods in C++ in order to integrate and test them with Feel++. We had to implement the parareal method and the data assimilation in order to test them with ODEs like the harmonic oscillator or the Lorenz system but also with PDEs like the heat equation or the Laplacian. Since the implementation was already done in Python during the project, we could check the results obtained with C++ and compare them with those obtained in Python.
For the parareal part, we had to implement the method in C++ with MPI after having implemented it in Python during the project. We also had to check the speed-up of the method and still couldn't see it. However, we have put forward some hypotheses that could explain why we did not have any speed-up in the test cases unlike in Python. We also checked the efficiency of the method and concluded that it was not efficient with the current implementation but in the same way as for the speed-up there are some hypothesis about the reasons. The next step of the project was to apply the parareal method to the resolution of the heat equation in C++ with Feel++ but we could not succeed, or verify the speed-up. For the future, we should review the current implementation of the parareal method in C++ by using non-blocking communications for example. We should also finish solving the heat equation with parareal and continue to test it by checking its correctness.
For the data assimilation part, we first had to realize the implementation of the Ensemble Kalman Filter algorithm in C++ based on the one already done in Python in the FilterPy library. We were able to compare the results by applying it to the Lorenz system to verify our algorithm. After making sure that our EnKF in C++ was working properly, we had to understand the heat equation in order to perform a thermal simulation of an office located at the UFR. Furthermore, we had data from the sensors in the office at our disposition. Using the simulation performed with Feel++ as the model and the data as the observations, we had to perform data assimilation to correct the model. We noticed that there were some incorrect values during this part, but due to lack of time we did not manage to correct our algorithm. For the future, it would be interesting to look further into the problem we had during the realization of the data assimilation. Another point to reconsider will be the simulation made with Feel++, as explained before, when we realized the simulation with the toolboxes available in Feel++, we had ignored the convective effects, so maybe we could try to do the simulation by introducing the convective effects.

Bibliography

References

[2] University Corporation for Atmospheric Research. *The Lorenz 63 model and its relevance to data assimilation*. URL: https://docs.dart.ucar.edu/en/latest/guide/lorenz-63-model.html (visited on 04/03/2022).

[3] Ian Beausoleil-Morrison. "Fundamentals of Building Performance Simulation". In: (), p. 410.

[4] Eric Blayo. *An introduction to data assimilation*. en. URL: https://www.eccorev.fr/IMG/pdf/Assimilationdonnees_EBlayo.pdf.

[10] Jacques-Louis Lions, Yvon Maday, and Gabriel Turinici. *Résolution d'EDP par un schéma en temps pararéel*. en. Apr. 2001. DOI: 10.1016/S0764-4442(00)01793-6. URL: https://hal.archives-ouvertes.fr/hal-00798372/file/CRAS_01_lions_maday_turinici.pdf (visited on 05/22/2022).

[11] Y. Maday. *The 'Parareal in Time' Algorithm*. en. Ed. by F. Magoulès. Stirlingshire, UK, May 2010. DOI: 10.4203/csets.24.2. URL: https://www.ljll.math.upmc.fr/publications/2008/R08030.pdf (visited on 05/22/2022).

[12] Maëlle Nodet. *Introduction to Data Assimilation*. en. URL: https://team.inria.fr/airsea/files/2012/03/Nodet_Intro_DataAssimilation.pdf.

[13] *Parareal*. en. Page Version ID: 1047894968. Oct. 2021. URL: https://en.wikipedia.org/w/index.php?title=Parareal&oldid=1047894968 (visited on 05/22/2022).

[14] Alfio Quarteroni and Alberto Valli. *Numerical Approximation of Partial Differential Equations*. Red. by R. L. Graham, J. Stoer, and R. Varga. Vol. 23. Springer Series in Computational Mathematics. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994. ISBN: 978-3-540-85267-4 978-3-540-85268-1. DOI: 10.1007/978-3-540-85268-1. URL: https://link.springer.com/content/pdf/10.1007/978-3-540-85268-1.pdf (visited on 08/17/2022).

[15] S Ricci. *Data Assimilation training course @ CEMRACS Introduction and variational algorithms*. en. URL: http://smai.emath.fr/cemracs/cemracs16/images/DA_Ricci_CEMRACS2016.pdf.

[16] John Stockie and Ken Wong. *Laboratory #6: A Simple Model of the Unpredictability of Weather: The Lorenz Equations*. en. URL: https://ftp.emc.ncep.noaa.gov/mmb/sref/Doc/lorenz_fcst.pdf.

[17] *THREE DIMENSIONAL SYSTEMS, Lecture 6 : The Lorenz Equations*. URL: https://www2.physics.ox.ac.uk/sites/default/files/profiles/read/lect6-43147.pdf (visited on 04/03/2022).

Documentation

[1] *Antora Documentation : Antora Docs*. URL: https://docs.antora.org/antora/latest/ (visited on 08/14/2022).

[5] *cmake-presets(7) — CMake 3.24.0 Documentation*. URL: https://cmake.org/cmake/help/latest/manual/cmake-presets.7.html#test-preset (visited on 08/14/2022).

[6] *Doxygen: Doxygen*. URL: https://doxygen.nl/index.html (visited on 08/14/2022).

[7] *Eigen: Getting started*. URL: https://eigen.tuxfamily.org/dox/GettingStarted.html (visited on 08/17/2022).

[8] *Feel++ documentation :: Feel++ Docs*. URL: https://docs.feelpp.org/feelpp/0.109/index.html (visited on 08/17/2022).

[9] *Laplacian : Feel++ Docs*. URL: https://docs.feelpp.org/user/latest/cpp/laplacian.html (visited on 08/14/2022).

[18] *Welcome — Sphinx documentation*. URL: https://www.sphinx-doc.org/en/master/# (visited on 08/14/2022).

A Organisation of the repository

For the organisation of the repository, we created several directories (Figure A.1):

- **src :** In this directory there is all the source code of the project. We have a first *python* subdirectory which contains the python codes for the data assimilation and parareal parts, these files were created during the M1 project. Then we have a second subdirectory *cpp* which contains the C++ codes of the same 2 parts.
- **examples :** Here there are examples of how to use python and C++ codes.
- **tests :** In this directory there are python and C++ tests of the code. For python tests we will use the pytest tool and for C++ we will use ctest (see Appendix B).
- **docs :** This directory gathers all the files that are used to generate the documentation of the project (see Appendix C). The *sphinx* and *doxygen* directories enable respectively to document the python code and the C++ code (thanks to the sphinx [18] and doxygen [6] tools). The *antora* directory contains some explanations of the project which are accessible online (the antora [1] tool was used). For example it explains : the differential equations concerned by this project, the data assimilation methods, the parareal method... The documentations generated by the previous tools are available directly in the GitHub repository via a Continuous Integration (CI) that has been set up (see Appendix D). The directories *gantt*, *meeting*, *presentation* and *report* contain all the latex files and images used for the presentations and reports requested in the context of the internship.
- **cmake/build :** The directory *cmake* contains cmake configuration files which are used for the compilation of the C++ project (see Appendix B). The directory *build* contains all the files generated by the compilation of the C++ code.

```
2022-m1-lorenz
├── build
├── cmake
├── docs
│   ├── antora
│   ├── doxygen
│   ├── gantt
│   ├── meeting
│   ├── presentation
│   │   ├── internship
│   │   └── project
│   ├── report
│   │   ├── internship
│   │   └── project
│   └── sphinx
├── examples
│   ├── cpp
│   │   ├── enkf
│   │   ├── feelpp
│   │   └── parareal
│   └── python
│       ├── enkf
│       └── parareal
├── src
│   ├── cpp
│   │   ├── EigenRand
│   │   ├── enkf
│   │   ├── feelpp
│   │   │   ├── heat
│   │   │   └── laplacian
│   │   ├── feelpp_heat
│   │   └── parareal
│   └── python
│       └── lorenz
│           ├── enkf
│           └── parareal
└── tests
    ├── cpp
    └── python
```

Figure A.1: Repository tree

B Compile and test

As explained in the previous section, the project includes both C++ and Python code. Let's see how to use these source codes :

- **Python :**
  Here all methods are grouped in a module called *lorenz*. To execute the examples, we first need to add the path to the source code in the PYTHONPATH variable using the following command at the root of the project :

  ```
  export PYTHONPATH=$PWD/src/python
  ```

  To run the examples for the ENKF part you need to go to "examples/python/enkf" and run the jupyter notebook named "`kalman_filter.ipynb`".
  To run the examples for the parareal method with 2 processes, go to "examples/python/-parareal" and then execute :

  ```
  mkdir data_parareal
  mpirun -n 2 python3 examples_parareal.py [0,1,2]
  ```

  To run the tests we use the pytest tool. To do this, go to "tests/python" and run the following command :

  ```
  pytest
  ```

- **C++ :**
  For the C++ code, we decided to write a CMake* with presets [5]. A recurrent problem of CMake users is to share settings with other people for common ways of configuring a project. A solution to this is to define a "CMakePresets.json" file at the root of the project allowing to define different compilation modes.
  We defined 3 presets : *default* is the default compilation mode, *dbg* the mode to debug the code and *doc* to generate the documentation (see Appendix C).
  These are the commands to configure and build in default mode :

  ```
  cmake --preset default
  cmake --build --preset default
  ```

  The executables generated by the last command will then be stored in the "build/default/bin" directory. There you will find :

  - **enkf.e :** In order to execute the examples for the ENKF part it is necessary to download a github repository which is used to generate real vectors on a multivariate normal distribution Download EigenRand.
    To run the examples :

*Minimum version required : 3.21

```
./data_assim_heat.e --config-file
    /home/u4/csmi/2020/aydogdu/2022-m1-lorenz/src
    /cpp/feelpp_heat/bureau228_stage/bureau228.cfg
```

- **parareal.e :** To run in parallel with 4 processes :

```
mpirun -n 4 ./build/default/bin/parareal.e
```

This applies the parareal method to the Lorenz system with parameters defined in the code and displays the number of iterations of the method as well as the execution time.

- **laplacian.e :** To run in sequential:

```
./build/default/bin/laplacian.e
  --config-file <cfg_filename>
```

To run in parallel with 4 processes:

```
mpirun -n 4 ./build/default/bin/laplacian.e
  --config-file <cfg_filename>
```

This allows to solve the Laplace equation from a given geometry by the finite element method by using Feel++ [9].

- **heat.e :**
This solves in parallel the heat equation from a given geometry with the parareal method by using Feel++.
We will place ourselves in the following test case.
The spatial domain is partitioned into 2 sub-domains using the command :

```
feelpp_mesh_partitioner --dim=2 --part 2
  --ifile <geo_filename>
```

We also partition the temporal domain into 2 sub-domains, which makes 2*2 processes for the fine integrator and 2 processes for the coarse integrator, so 6 processes in all. We can then run in parallel with 6 processes :

```
mpirun -np 6 ./build/default/bin/heat.e
  --config-file <cfg_filename>
```

For more details see Section 3.3.2.

For C++ tests, we use the ctest tool in the following way:

```
ctest --preset default
```

C Documentation

To document the project as well as possible, we have used different tools which we will talk about here. As said before, all the files concerning the documentation are in the *docs* directory.
Before presenting these tools, it is important to note that we have set up a Continuous Integration in order to automatically generate this documentation when a push is made on GitHub (see Section D).
Here is the link to the documentation : https://master-csmi.github.io/2022-m1-lorenz/.
To document the project we used 3 tools:

- **Sphinx**[18] : To document the Python code.
- **Doxygen**[6] : To document C++ code.
- **Antora**[1] : To explain some parts/methods of the project (and to be able to keep it online).

We will now see how to generate the documentation locally using these 3 tools. For that, there are 2 methods : the first one is to generate "manually" the html files and the second one is to use the *docs* preset of our CMake.

- **Manual generation :**
  - **Antora :** Go to "docs/antora", then execute :

    ```
    npm install
    npm run dev
    npm run serve
    ```

    Then click on the following link (in the "site-dev.yml" file) :
    http://127.0.0.1:8080/lorenz/1.0.0/index.html.
    If we modify the documentation files, we just have to re-run the second line which will regenerate the documentation and then reload our browser window.
  - **Sphinx :** Go to "docs/sphinx", then execute :

    ```
    make html
    npm run sphinx
    ```

  - **Doxygen :** Go to "docs/doxygen", then execute :

    ```
    doxygen Doxyfile-dev.in
    npm run doxygen
    ```

- **With CMake :**
  ⚠ This method is only available for Sphinx and Doxygen documentations. Indeed, CMake is configured in order that Antora documentation is generated from the files uploaded online on Github. This configuration is due to CI (see Section D). This means that local modifications will have no effect with this method.

  Here is how to use the CMake *docs* preset to generate the Sphinx and Doxygen documentations (by placing at the root of the project) :

  ```
  cmake --preset docs
  cmake --build --preset docs
  ```

  Then in the same way as before, execute one of the following commands at the root of the project :

  ```
  npm run sphinx
  ```

  or

  ```
  npm run doxygen
  ```

D Github actions

As said before we have set up a CI (Continuous Integration) using GitHub actions. We added a badge at the beginning of the readme of our project to see when the build passed and when the build failed. Here is what it looks like in both cases :

Figure D.1: Build passing

Figure D.2: Build failing

For more details you have to go to the action tab of the repository where are detailed all the tasks of each commit. Here is a preview of the CI for one commit :

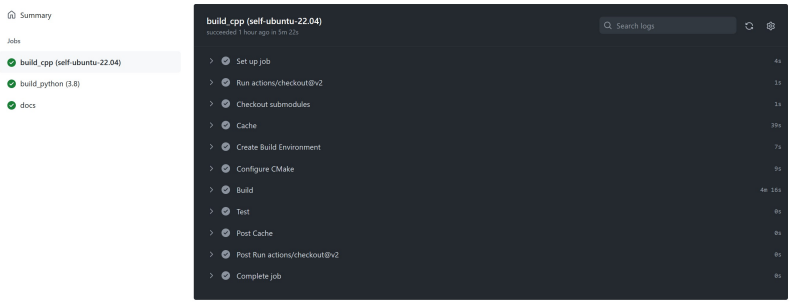


Figure D.3: Tab actions in GitHub (for one commit)

Here are the jobs realized as soon as we push in the main branch :

- Compile C++ code using CMake default preset then use ctest to check if the tests pass.
- Test the Python code with pytest.
- Generate the documentation (Sphinx, Doxygen and Antora) using the docs preset of the CMake. Then, deploy the 3 documentations in the *gh-pages* branch (in 3 respective directories: *build_sphinx*, *build_doxygen* and *build_antora)*. An html file has been created manually at the root of this branch in order to group the 3 documentations :

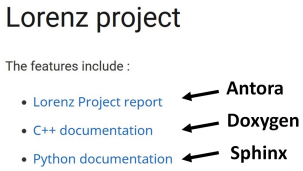


Figure D.4: Main page of the documentation

For more details about the documentation see Section C.