POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Parareal-accelerated gradient flow iterations for optimal control problems

APSC Course Project - Mathematical Engineering

Author: **Fiammetta Artioli**

Student ID: 10659207
Advisor: Prof. Gabriele Ciaramella
Academic Year: 2023-24

Contents

Introduction

The aim of the project is to introduce and implement in $C++$ a novel computational framework for the numerical solution of optimal control problems. This new framework is based on a parareal acceleration of gradient-flow type iterations.
K. Tanabe in [1] proposed a numerical method to solve the constrained function maximization problem. This method, named Continuous Gradient-Projection Method with Enforced Constraint Restoration (CGPECR), expresses the iterative rule used to obtain the solution as an autonomous differential problem that requires to solve a linear system of equations. In sections 1.1 and 1.2 we introduce an optimal control problem in continuous form and its discretization with the finite elements method. In section 1.3, instead, we describe CGPECR and we show how it can be used to obtain successive iterates of the optimal state and optimal control of the discrete problem, which can be interpreted as two gradient flow curves of the descent directions.
D. Kingma and J. Ba, in [4] introduced Adam, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. In section 1.4, we describe how Adam can be used as an alternative rule to obtain successive approximations of the solution of the discrete optimal control problem.
J.-L. Lions, Y. Maday, and G. Turinici in [2] introduced the Parareal algorithm. Parareal allows one to perform the parallel in time integration of ordinary differential equations, by breaking the time-evolution problem into a series of independent evolution problems on smaller time intervals.
M. J. Gander and S. Vandewalle in [3] proved a relation between the Parareal algorithm and the multiple shooting methods. In section 1.5, we describe the Parareal algorithm and how it can be applied to accelerate the integration of the autonomous system reported in section 1.3. In sections 1.6 and 1.7, instead, we propose respectively the ParaFlow and ParaFlowS algorithms, which are both based on the Parareal, but they adapt it to the purpose of accelerating the resolution process of discrete optimal control problems.
In chapter 2, we describe the details of the code implementation in C++. For the finite element method we have relied upon the *deal.ii* library [5].
Lastly, in chapter 3 we report the results obtained by applying the Gradient Flow,

Parareal, ParaFlow and ParaFlowS algorithms to a discrete optimal control problem, focusing in particular on the number of iterations required by each algorithm to reach the solution.

1 | Mathematical Formulation

The modelling and simulation of complex systems plays an important role in physics, engineering, medicine and in other disciplines. Very often, mathematical models of complex systems result in partial differential equations (PDEs) as for example heat flow, diffusion and elastic deformation. In most applications the ultimate goal is not only the mathematical modelling and numerical simulation of the complex system, but rather the optimization or optimal control of the considered process. Therefore, an important class of problems in optimization results from optimal control applications, which consist of an evolutionary or equilibrium system that includes a control mechanism and a functional modeling the purpose of the control.
In this work, we consider an optimal control problem governed by a semilinear elliptic differential equation and characterized by a quadratic cost functional.

1.1. Continuous Problem

We consider the optimal control problem

$$\min_{y,u} \quad J(y,u) := \frac{1}{2}\|y - y_d\|_{L^2}^2 + \frac{\nu}{2}\|u\|_{L^2}^2 \tag{1.1}$$

subject to the constraint

$$\int_\Omega \nabla y \cdot \nabla v + \gamma\varphi(y)v = \int_\Omega uv \qquad \forall v \in H_0^1(\Omega), \tag{1.2}$$

where $\Omega \subseteq \mathbb{R}^2$, $y \in H_0^1(\Omega)$, $u \in H_0^1(\Omega)$ and $y_d \in L^2(\Omega)$. Here, $\varphi : \mathbb{R} \to \mathbb{R}$ is a nonlinear function and the constants $\nu \in \mathbb{R}$, $\gamma \in \mathbb{R}$ are such that $\nu > 0$, $\gamma > 0$. Moreover, y is the state vector, y_d the desired state, u represents the control vector, and ν is a regularization parameter used to tune the weight of the L^2-norm of u. Therefore, minimizing the cost functional $J(y,u)$ is equivalent to minimizing the deviation of y from the desired state y_d, while reducing the norm of the control u to apply.

The Karush-Kuhn-Tucker conditions for the optimal control problem above [6] are

$$\begin{aligned} \int_\Omega \nabla y \cdot \nabla v + \gamma\varphi(y)v &= \int_\Omega uv && \forall v \in H_0^1(\Omega), \\ \int_\Omega \nabla p \cdot \nabla v + \gamma\varphi'(y)pv &= \int_\Omega (y - y_d)v && \forall v \in H_0^1(\Omega), \\ \nu u &= p && a.e. \quad in \quad \Omega. \end{aligned} \tag{1.3}$$

1.2. Discrete Problem

To computationally address the optimal control problem (1.1)-(1.2), we discretize the domain Ω with quadrilateral elements to obtain a mesh $\mathcal{T}_h$ that defines a discrete domain Ω_h such that $\bar{\Omega}_h = \cup_{K\in\mathcal{T}_h} K$. As discrete approximation space V_h we choose the one constructed with bilinear functions:

$$V_h = \{v \in C^0(\bar{\Omega}_h) : v|_{\partial\Omega_h} = 0, \forall K \in \mathcal{T}_h, v|_K \in Q_1(K)\} \subset H_0^1(\Omega_h). \tag{1.4}$$

This choice of Lagrangian $Q1$ finite elements has been made for implementation purposes, as it is the easiest to be used in the *deal.ii* library, but it does not affect the methodology reported hereinafter.

The discretized problem, obtained as Galerkin's approximation of the continuous problem's weak formulation, is the following

$$\min_{y_h,u_h} \quad J(y_h, u_h) := \frac{1}{2}(\mathbf{y}_h - \mathbf{y}_{d_h})^T M(\mathbf{y}_h - \mathbf{y}_{d_h}) + \frac{\nu}{2}\mathbf{u}_h^T M\mathbf{u}_h \tag{1.5}$$

subject to the constraint

$$K\mathbf{y}_h + \gamma M\varphi(\mathbf{y}_h) = M\mathbf{u}_h, \tag{1.6}$$

where y_h, u_h, $y_{d_h} \in V_h \subset H_0^1(\Omega)$ and matrices K and M are, respectively, the global stiffness and mass matrices obtained as

$$\begin{aligned} K = [k_{ij}] \qquad k_{ij} &= \int_\Omega \nabla v_j \nabla v_i d\Omega && i,j = 1,\dots,N_h, \\ M = [m_{ij}] \qquad m_{ij} &= \int_\Omega v_j v_i d\Omega && i,j = 1,\dots,N_h, \end{aligned} \tag{1.7}$$

where $\{v_i\}$ is a basis in V_h and N_h is the number of nodes of $\mathcal{T}_h$.

1.3. Numerical Solution

To solve the problem we have implemented CGPECR [1], which is a numerical method for nonlinear constrained optimization based on a geometric approach.
In this section, we report a description of the method and of its application to the discrete optimal control problem above.

1.3.1. Preliminary Definitions

Defining $\mathbf{x} = (\mathbf{x_1}, \mathbf{x_2}) := (\mathbf{y}_h, \mathbf{u}_h)$, we can express the discrete problem as

$$\min_{\mathbf{x}} \quad J(\mathbf{x})$$

subject to

$$g(\mathbf{x}) := K\mathbf{x_1} + \gamma M\varphi(\mathbf{x_1}) - M\mathbf{x_2} = 0.$$

We can define the feasible set for the solution $\mathbf{x}$ as $V_g := \{\mathbf{x} \in \mathbb{R}^{N_n} \times \mathbb{R}^{N_n} : g(\mathbf{x}) = 0\}$. If $\nabla g \neq \mathbf{0}$, then V_g is a differentiable manifold and J is a differentiable function on the manifold V_g. Therefore, the discrete problem consists in minimizing J on the manifold V_g.
Moreover, if we denote by $T_{\mathbf{x}}$ the tangent space of the manifold V_g at a feasible point $\mathbf{x}$, then $T_{\mathbf{x}}$ can be represented by the null space $N(\nabla g(\mathbf{x})) := \{z \in \mathbb{R}^{N_n} \times \mathbb{R}^{N_n} : \nabla g(\mathbf{x})^T \mathbf{z} = \mathbf{0}\}$.

1.3.2. Continuous Gradient-Projection Method with Enforced Constraint Restoration (CGPECR)

According to the Gradient-Projection method, successive approximations $\mathbf{x}^{(k)}$ are generated by the rule

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\phi(\mathbf{x}^{(k)}), \tag{1.8}$$

where $\phi(\mathbf{x}) := -P(\mathbf{x})\nabla J(\mathbf{x})$, and $P(\mathbf{x})$ is the orthogonal projection on the null space $N(\nabla g(\mathbf{x}))$. If the constraint g is highly nonlinear, as it happens to be in our case, the feasible manifold V_g cannot be traced well by these approximations. To address this difficulty, different methods to compute successive approximations $\mathbf{x}^{(k)}$ are presented in [1]. For instance, the Continuous Gradient-Projection method introduces the following autonomous system

$$d\mathbf{x}/dt = \phi(\mathbf{x}) = -P(\mathbf{x})\nabla J(\mathbf{x}), \tag{1.9}$$

where the set

$$\{\mathbf{x} \in V_g : \phi(\mathbf{x}) = 0\} \tag{1.10}$$

of equilibrium points coincides with the set of critical points of J on V_g to which the solution of the minimization problem belongs. Moreover, this method associates with the vector field $\phi(\mathbf{x})$ a differentiable mapping $\Lambda(\mathbf{x})$ such that

$$\begin{bmatrix} I & \nabla^T g(\mathbf{x}) \\ \nabla g(\mathbf{x}) & 0 \end{bmatrix} \begin{bmatrix} \phi(\mathbf{x}) \\ \Lambda(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} -\nabla J(\mathbf{x}) \\ 0 \end{bmatrix}. \tag{1.11}$$

The Continuous Gradient-Projection method can still be improved by incorporating in (1.11) a self-correcting feature for the restoration of violation of constraints. While in the method above $\Lambda(\mathbf{x})$ is determined by the equation $dg(\mathbf{x})/dt = 0$, in CGPECR one uses $dg(\mathbf{x})/dt = -g(\mathbf{x})$. The obtained autonomous system is

$$d\mathbf{x}/dt = \phi(\mathbf{x}) = -P(\mathbf{x})\nabla J(\mathbf{x}) = -\nabla J(\mathbf{x}) - \nabla^T g(\mathbf{x})\Lambda(\mathbf{x}) \tag{1.12}$$

with

$$\begin{bmatrix} I & \nabla^T g(\mathbf{x}) \\ \nabla g(\mathbf{x}) & 0 \end{bmatrix} \begin{bmatrix} \phi(\mathbf{x}) \\ \Lambda(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} -\nabla J(\mathbf{x}) \\ -g(\mathbf{x}) \end{bmatrix} \tag{1.13}$$

where, once again, the set

$$\{\mathbf{x} \in V_g : \phi(\mathbf{x}) = 0\} \tag{1.14}$$

of equilibrium points coincides with the set of critical points of J on V_g to which the solution of the minimization problem belongs.

1.3.3. Application to the Discrete Problem

If we apply CGPECR to the discrete optimal control problem (1.5)-(1.6), we obtain the autonomous system

$$\begin{cases} \frac{dy_h}{dt} = \phi_1(y_h, u_h), \\ \frac{du_h}{dt} = \phi_2(y_h, u_h), \end{cases} \tag{1.15}$$

where ϕ_1 and ϕ_2 solve the system

$$\begin{bmatrix} I & 0 & \tilde{K}(y_h) \\ 0 & I & -M \\ \tilde{K}(y_h) & -M & 0 \end{bmatrix} \begin{bmatrix} \phi_1(y_h, u_h) \\ \phi_2(y_h, u_h) \\ \Lambda(y_h, u_h) \end{bmatrix} = \begin{bmatrix} -\nabla_{y_h} J(y_h, u_h) \\ -\nabla_{u_h} J(y_h, u_h) \\ -g(y_h, u_h) \end{bmatrix}, \tag{1.16}$$

with

$$\begin{cases} \tilde{K}(y_h) := K + \gamma M \varphi'(\mathbf{y}_h) \\ g(y_h, u_h) := K\mathbf{y}_h + \gamma M \varphi(\mathbf{y}_h) - M\mathbf{u}_h \\ \nabla_{y_h} J(y_h, u_h) = M(\mathbf{y}_h - \mathbf{y}_{d_h}) \\ \nabla_{u_h} J(y_h, u_h) = \nu M \mathbf{u}_h \end{cases} . \tag{1.17}$$

Given the initial conditions $y_h^{(0)}$ and $u_h^{(0)}$, successive approximations of the solution vectors $y_h^{(k)}$ and $u_h^{(k)}$ from (1.15) are two gradient flow curves that can be obtained with the following update rule

$$\begin{cases} y_h^{(k+1)} = y_h^{(k)} + \alpha^{(k)} \phi_1(y_h^{(k)}, u_h^{(k)}) \\ u_h^{(k+1)} = u_h^{(k)} + \alpha^{(k)} \phi_2(y_h^{(k)}, u_h^{(k)}) \end{cases} . \tag{1.18}$$

At each iteration k, the descent directions ϕ_1 and ϕ_2 are obtained by solving the linear system (1.16), while $\alpha^{(k)}$ is the time step used to discretize the time derivatives of y_h and u_h with the Explicit Euler scheme.

As stopping criterion used to determine if $y_h^{(k)}$ and $u_h^{(k)}$ are close enough to the solution, we can check that the L^2-norms of ϕ_1 and ϕ_2 are below certain tolerances tol_1 and tol_2. Indeed, finding a solution in the set (1.14) of equilibrium points is equivalent to solving the KKT conditions system associated with the discrete problem (1.5)-(1.6) [1].

The steps to follow to obtain a numerical solution to the discrete problem with the method just described can be summarized in Algorithm 1.1:

Algorithm 1.1 Gradient Flow

1: Given the initial guesses $y_h^{(0)}$ and $u_h^{(0)}$
2: Assemble the linear system with $y_h^{(0)}$ and $u_h^{(0)}$
3: Solve the linear system to obtain ϕ_1 and ϕ_2
4: Update vectors $y_h^{(0)}$ and $u_h^{(0)}$ with the new ϕ_1 and ϕ_2
5: **while** $\|\phi_1\|_2 > tol_1$ or $\|\phi_2\|_2 > tol_2$ **do**
6: Assemble the linear system with the current $y_h^{(k)}$ and $u_h^{(k)}$
7: Solve the linear system to obtain ϕ_1 and ϕ_2
8: Update vectors $y_h^{(k)}$ and $u_h^{(k)}$ with the new ϕ_1 and ϕ_2
9: **end while**

1.4. Adam Algorithm

As an alternative to the update rule (1.18), based on the Explicit Euler discretization, we can use the *Adam* (Adaptive Moment Estimation) algorithm [4].
The update rule proposed by the algorithm takes into account not only the descent directions ϕ_1 and ϕ_2 at step k, but also their previous values through two *momentum* terms, one related to the descent directions and the other related to its square values.
More specifically, for a generic vector z that satisfies $dz/dt = \phi(z(t))$, successive approximations of the solution $z^{(k)}$, starting from an initial guess $z^{(0)}$ and the momentum terms $m_z^{(0)} = 0$ and $v_z^{(0)} = 0$, can be obtained as

$$m_z^{(k+1)} = \beta_1 m_z^{(k)} + (1-\beta_1)\phi(z^{(k)}), \tag{1.19}$$
$$v_z^{(k+1)} = \beta_2 v_z^{(k)} + (1-\beta_2)\phi(z^{(k)})^2, \tag{1.20}$$
$$\hat{m}_z = \frac{m_z^{(k+1)}}{1-\beta_1}, \tag{1.21}$$
$$\hat{v}_z = \frac{v_z^{(k+1)}}{1-\beta_2}, \tag{1.22}$$
$$z^{(k+1)} = z^{(k)} + \alpha \frac{\hat{m}_z}{\sqrt{\hat{v}_z} + \epsilon}, \tag{1.23}$$

where the vector $\frac{\hat{m}_z}{\sqrt{\hat{v}_z}+\epsilon}$ is obtained by performing element-wise operations and $\epsilon = 10^{-8}$ is a value added to guarantee numerical stability. Moreover, β_1 and β_2 are the forgetting factors for ϕ and its square value.
In our case, we can apply this update rule to (1.15) and obtain successive approximations of vectors $y_h^{(k)}$ and $u_h^{(k)}$ as

$$\begin{aligned} y_h^{(k+1)} &= y_h^{(k)} + \alpha\tilde{\phi}_1(\hat{m}_y, \hat{v}_y), \\ u_h^{(k+1)} &= u_h^{(k)} + \alpha\tilde{\phi}_2(\hat{m}_u, \hat{v}_u), \end{aligned} \tag{1.24}$$

where

$$\begin{aligned} m_y^{(k+1)} &= \beta_1 m_y^{(k)} + (1-\beta_1)\phi_1(y_h^{(k)}, u_h^{(k)}), \\ m_u^{(k+1)} &= \beta_1 m_u^{(k)} + (1-\beta_1)\phi_2(y_h^{(k)}, u_h^{(k)}), \\ v_y^{(k+1)} &= \beta_2 v_y^{(k)} + (1-\beta_2)\phi_1(y_h^{(k)}, u_h^{(k)})^2, \\ v_u^{(k+1)} &= \beta_2 v_u^{(k)} + (1-\beta_2)\phi_2(y_h^{(k)}, u_h^{(k)})^2, \\ \hat{m}_y &= \frac{m_y^{(k+1)}}{1-\beta_1}, \quad \hat{m}_u = \frac{m_u^{(k+1)}}{1-\beta_1}, \\ \hat{v}_y &= \frac{v_y^{(k+1)}}{1-\beta_2}, \quad \hat{v}_u = \frac{v_u^{(k+1)}}{1-\beta_2}, \end{aligned} \tag{1.25}$$

and $\tilde{\phi}_1(\hat{m}_y, \hat{v}_y)$ and $\tilde{\phi}_2(\hat{m}_u, \hat{v}_u)$ are the projections respectively of $\frac{\hat{m}_y}{\sqrt{\hat{v}_y}+\epsilon}$ and $\frac{\hat{m}_u}{\sqrt{\hat{v}_u}+\epsilon}$ on the manifold V_g, which are necessary since the descent directions $\tilde{\phi}_1$ and $\tilde{\phi}_2$ computed with the moment terms do not necessarily lead to solutions $y_h^{(k+1)}$ and $u_h^{(k+1)}$ that satisfy the constraint g. The projections require the solution of an additional linear system at each iteration and can be performed as in (1.12)

$$\begin{aligned}\tilde{\phi}_1(\hat{m}_y, \hat{v}_y) &= \frac{\hat{m}_y}{\sqrt{\hat{v}_y}+\epsilon} - \nabla^T g(y_h, u_h)\tilde{\Lambda}(\hat{m}_y, \hat{v}_y),\\ \tilde{\phi}_2(\hat{m}_u, \hat{v}_u) &= \frac{\hat{m}_u}{\sqrt{\hat{v}_u}+\epsilon} - \nabla^T g(y_h, u_h)\tilde{\Lambda}(\hat{m}_y, \hat{v}_y),\end{aligned} \tag{1.26}$$

where $\tilde{\phi}_1$, $\tilde{\phi}_2$ and $\tilde{\Lambda}$ solve the system

$$\begin{bmatrix} I & 0 & \tilde{K}(y_h) \\ 0 & I & -M \\ \tilde{K}(y_h) & -M & 0 \end{bmatrix} \begin{bmatrix} \tilde{\phi}_1(\hat{m}_y, \hat{v}_y) \\ \tilde{\phi}_2(\hat{m}_y, \hat{v}_y) \\ \tilde{\Lambda}(\hat{m}_y, \hat{v}_y) \end{bmatrix} = \begin{bmatrix} \frac{\hat{m}_y}{\sqrt{\hat{v}_y}+\epsilon} \\ \frac{\hat{m}_u}{\sqrt{\hat{v}_u}+\epsilon} \\ -g(y_h, u_h) \end{bmatrix}. \tag{1.27}$$

1.5. Parareal Algorithm

The iterative method reported above, regardless of the update rule used, can require a large number of steps to converge to the solution, especially when the mesh used to discretize the domain Ω is particularly refined. Therefore, to speed up the descent process towards the solution we have implemented the Parareal algorithm [2], a parallel-in-time integration method. The Parareal algorithm is a numerical strategy used to solve evolution problems in parallel, by approximating solutions later in time before having fully accurate approximations from earlier times. In particular, it is used to compute the numerical solution for general systems of ordinary differential equations of the form

$$\mathbf{u}' = \mathbf{f}(\mathbf{u}), \quad \mathbf{u}(0) = \mathbf{u}_0, \quad t \in [0, T]. \tag{1.28}$$

Parareal is defined using two propagation operators:

- The coarse solver G, which provides a rough approximation to $\mathbf{u}(t)$, solution of (1.28)
- The fine solver F, which provides a more accurate approximation to $\mathbf{u}(t)$, solution of (1.28)

In particular, given an initial condition $\mathbf{u}(t_1) = \mathbf{u}_1$, the operators $G(t_2, t_1, \mathbf{u}_1)$ and $F(t_2, t_1, \mathbf{u}_1)$ give respectively a rough and more accurate solution of $\mathbf{u}(t_2) = \mathbf{u}_2$. They both propagate the initial value over the time interval, however operator G does so at a much lower numerical accuracy and therefore computational cost with respect to the fine solver F.
The algorithm starts with an initial approximation $\mathbf{U}_n^0$, $n = 0, 1, ..., N$ at time $t_0, t_1, ..., t_N$ given by the sequential computation of $U_{n+1}^0 = G(t_{n+1}, t_n, \mathbf{U}_n^0)$, with $\mathbf{U}_0^0 = \mathbf{u}_0$, and then performs for $k = 0, 1, 2, ...$ the correction iteration

$$\mathbf{U}_{n+1}^{k+1} = G(t_{n+1}, t_n, \mathbf{U}_n^{k+1}) + F(t_{n+1}, t_n, \mathbf{U}_n^k) - G(t_{n+1}, t_n, \mathbf{U}_n^k). \tag{1.29}$$

As the number of iterations $k \to \infty$, the approximated values $\mathbf{U}_n$ will have achieved the accuracy of the operator F.
We will now report the derivation of the Parareal formulation based on the multiple shooting method applied to (1.28).

1.5.1. Derivation of the Parareal algorithm

In the multiple shooting method the time interval $[0, T]$ is partitioned into N subintervals, determined by the time-points $0 = t_0 < t_1 < ... < t_N = T$. A system of N separate initial value problems is posed

$$\begin{cases} \mathbf{u}_0' = \mathbf{f}(\mathbf{u}_0), \quad \mathbf{u}_0(0) = \mathbf{U}_0, \\ \mathbf{u}_1' = \mathbf{f}(\mathbf{u}_1), \quad \mathbf{u}_1(t_1) = \mathbf{U}_1, \\ \qquad \vdots \\ \mathbf{u}_{N-1}' = \mathbf{f}(\mathbf{u}_{N-1}), \quad \mathbf{u}_{N-1}(t_{N-1}) = \mathbf{U}_{N-1}, \end{cases} \tag{1.30}$$

together with the matching conditions

$$\mathbf{U}_0 - \mathbf{u}_0 = 0, \quad \mathbf{U}_1 - \mathbf{u}_0(t_1, \mathbf{U}_0) = 0, \quad \ldots, \quad \mathbf{U}_N - \mathbf{u}_{N-1}(T, \mathbf{U}_{N-1}) = 0. \tag{1.31}$$

These matching conditions form a nonlinear system of equations

$$\mathbf{U}^{k+1} = \mathbf{U}^k - J_F^{-1}(\mathbf{U}^k)\mathbf{F}(\mathbf{U}^k), \tag{1.32}$$

that can be solved with Newton's method which leads to

$$\mathbf{F}(\mathbf{U}) = 0, \quad \mathbf{U} = (\mathbf{U}_0, \mathbf{U}_1, ..., \mathbf{U}_N)^T, \tag{1.33}$$

where J_F denotes the Jacobian of $\mathbf{F}$. The Newton update $J_F^{-1}(\mathbf{U}^k)\mathbf{F}(\mathbf{U}^k)$ can be written as

$$\begin{bmatrix} I & & & \\ -\frac{\partial \mathbf{u}_0}{\partial \mathbf{U}_0}(t_1, \mathbf{U}_0^k) & I & & \\ & -\frac{\partial \mathbf{u}_1}{\partial \mathbf{U}_1}(t_2, \mathbf{U}_1^k) & I & \\ & & \ddots & \\ & & -\frac{\partial \mathbf{u}_{N-1}}{\partial \mathbf{U}_{N-1}}(t_{n-1}, \mathbf{U}_{N-1}^k) & I \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{U}_0^k - \mathbf{u}_0 \\ \mathbf{U}_1^k - \mathbf{u}_0(t_1, \mathbf{U}_0^k) \\ \mathbf{U}_2^k - \mathbf{u}_1(t_2, \mathbf{U}_1^k) \\ \vdots \\ \mathbf{U}_N^k - \mathbf{u}_{N-1}(T, \mathbf{U}_1^k) \end{bmatrix}$$

and, therefore, (1.32) can be rearranged to obtain the basic recurrence of the multiple shooting method applied to initial value problems,

$$\begin{cases} \mathbf{U}_0^{k+1} = \mathbf{u}_0, \\ \mathbf{U}_{n+1}^{k+1} = \mathbf{u}_n(t_{n+1}, \mathbf{U}_n^k) + \frac{\partial \mathbf{u}_n}{\partial \mathbf{U}_n}(t_{n+1}, \mathbf{U}_n^k)(\mathbf{U}_n^{k+1} - \mathbf{U}_n^k). \end{cases} \tag{1.34}$$

If one approximates the subinterval solves by $\mathbf{u}_n(t_{n+1}, \mathbf{U}_n^k) = F(t_{n+1}, t_n, \mathbf{U}_n^k)$ and the second term of the right-hand side of (1.34) in a finite difference way with the coarse solver G

$$\frac{\partial \mathbf{u}_n}{\partial \mathbf{U}_n}(t_{n+1}, \mathbf{U}_n^k)(\mathbf{U}_n^{k+1} - \mathbf{U}_n^k) = G(t_{n+1}, t_n, \mathbf{U}_n^{k+1}) - G(t_{n+1}, t_n, \mathbf{U}_n^k), \tag{1.35}$$

then the multiple shooting algorithm (1.30) and the Parareal algorithm (1.29) coincide.

1.5.2. Parallel Computations

Instead of using a single processor to solve (1.28), as classical time-stepping methods do, the Parareal algorithm makes use of N processors. Specifically, the N processors are used to solve in parallel N smaller initial value problems. Indeed, if the time interval $[0, T]$ is split in N subintervals, the coarse solver G can be used by a unique processor to compute in *series* a rough approximation of $N-1$ initial conditions, one for each subinterval. Subsequently, once each one of the $N-1$ initial conditions are sent to the remaining $N-1$ processors, each processor can use it as initial value to propagate in *parallel* the fine solver F over its subinterval. The values obtained from the fine solvers can then be used to update the parareal solution values sequentially by the first processor as in Equation (1.29). This iteration can be repeated until

$$\|\mathbf{U}_j^k - \mathbf{U}_j^{k-1}\| < \epsilon \quad \forall j < N \tag{1.36}$$

for some $\epsilon > 0$.

1.5.3. Application to Optimal Control Problems

The Parareal algorithm can be used to accelerate the numerical solution of (1.15). In this case however, our interest does not vert towards finding the exact value of the solutions y_h and u_h at each time instant in an interval $[0, T]$, but rather in finding y_h and u_h that satisfy the convergence criteria based on the L^2-norms of ϕ_1 and ϕ_2, reported at the end of section 1.3.3. Moreover, we do not know a priori how many iterations will be required to converge to the solution, so the value of T is unknown. Therefore, to adapt the Parareal algorithm to our problem we proceeded in the following way. We chose a certain value T_1 and ran the Parareal algorithm considering the time interval $[0, T_1]$ until its convergence condition (1.36) was satisfied. If the L^2-norm convergence criterion was not satisfied, we applied the Parareal algorithm to the same ordinary differential equation (1.15) but on a different time interval $[T_1, T_2]$, considering as initial values for y_h and u_h the ones previously obtained at T_1. This procedure was iterated on different time intervals $[T_j, T_{j+1}]$ until the L^2-norm convergence criterion was satisfied. For simplicity, once T_1 was chosen, successive values T_j have been taken equal to jT_1 so that all intervals had length T_1.

The pseudocode of this procedure is reported in Algorithm 1.2.

Algorithm 1.2 Parareal for Gradient Flow

Given T_1, the initial values $\mathbf{y}_h^{(0)}$ and $\mathbf{u}_h^{(0)}$ and N processors P_i, $i = 0, \dots, N-1$
Set $\mathbf{U}_0^0 = (\mathbf{y}_h^{(0)}, \mathbf{u}_h^{(0)})$, $\Delta = T_1/N$, $t_i = i\Delta$, $i = 0, \dots, N-1$, $k = 0$ and $j = 0$
Processor P_0 propagates $\mathbf{U}_{i+1}^0 = G(t_{i+1}, t_i, \mathbf{U}_i^0)\ \forall i$
while $\|\phi_1\|_2 > tol_1$ or $\|\phi_2\|_2 > tol_2$ **do**
    Processor P_0 sends $\mathbf{U}_{i+1}^k = G(t_{i+1}, t_i, \mathbf{U}_i^k)\ \forall i = 0, \dots, N-2$ respectively to processors $P_1, \dots, P_{N-1}$
    Processors P_i propagate in parallel $F(t_{i+1}, t_i, \mathbf{U}_i^k)\ \forall i$
    Each processor $P_i\ \forall i = 1, \dots, N-1$ sends the obtained $\mathbf{U}_{i+1}^k$ to processor P_0
    Processor P_0 performs the correction iteration (1.29) to obtain $\mathbf{U}_{i+1}^{k+1}\ \forall i$
    Set $k = k + 1$
    if $\|\mathbf{U}^k - \mathbf{U}^{k-1}\|_\infty < \epsilon$ **then**
        Set $\mathbf{U}_0^k = \mathbf{U}_N^k$, $j = j + 1$, $t_i = jT_1 + i\Delta$, $i = 0, \dots, N-1$
        Processor P_0 propagates $\mathbf{U}_{i+1}^k = G(t_{i+1}, t_i, \mathbf{U}_i^k)\ \forall i$
    end if
end while

1.5.4. Choice of Operators G and F

Operators G and F must solve numerically equation (1.15) on each subinterval with a different level of accuracy and, hence, computational cost. For what concerns the type of solver we can choose among two types of explicit solvers, one implementing the Explicit Euler update rule in (1.18) and the other implementing the Adam update rule in (1.24) [1]. Instead, for what concerns the accuracy, we can vary the operator's accuracy by choosing different step sizes α for the update rule. Therefore, for G and F we can either combine the same type of solver but with different step size or we can also use two different solver types.

Since explicit solvers are subject to stability issues, the step size α chosen for operators G and F has to be small enough to avoid divergence. However, since G is the coarse operator, its step size α should be chosen as the largest possible in order to reduce the number of iterations of the update rule to perform on each subinterval and therefore its computational cost. If the step size α of G becomes comparable to that of the fine solver F, the computational cost of both operators becomes similar. Moreover, we remind that in our case we are more interested in reaching the solution with the minimum number of iterations, rather than obtaining accurate results at every time step.

Therefore, since the advantage of applying the Parareal algorithm to accelerate the solution finding process relies on the fact that the operations carried out in series (therefore the ones performed by operator G) have a low computational cost, in this case the direct application of the Parareal algorithm does not result convenient.

To improve the performance of the Parareal algorithm for this problem we present the following algorithm.

1.6. ParaFlow

The ParaFlow algorithm is based upon the Parareal Algorithm 1.2, but adapts it to the purpose of solving optimal control problems, therefore aiming at a fast convergence towards the solution rather than at an accurate solution of the whole gradient flow trajectory.

Parareal solves equation (1.15) on a time interval $[0, T]$ and defines the operators G and F based on T and on the number of processes N. Indeed, each operator F is assigned to a subinterval of length T/N while operator G operates on the whole time interval $[0, T]$ and provides initial values for the N initial value problems solved by each solver F.

In the ParaFlow algorithm, instead, operators G and F are defined in terms of update

[1] Other explicit solvers can be used, however we postpone the testing to future work.

rule iterations N_G and N_F to perform. More specifically, operator G undertakes a unique descent step ($N_G = 1$) per processor, while the number of iterations of F, N_F, can be chosen based on the problem. Since G operates in series, the total number of update rule iterations per ParaFlow iteration will be equal to $N_G N$. This choice is first of all possible since the initial value problem that we are solving is not bound to a particular time interval $[0, T]$, while it is mainly driven by the aim to reduce the computations performed in series by G.
In addition, unlike the Parareal algorithm, the ParaFlow algorithm does not check the convergence condition (1.36) before moving on to the next interval on which it operates. Instead, at each iteration k, it uses the last value obtained by the correction step (1.29), as initial value $\mathbf{U}_0^{k+1}$ propagated in the next iteration $k+1$.

The pseudocode of the ParaFlow is given in Algorithm 1.3.

Algorithm 1.3 ParaFlow

Given N_F, the initial values $\mathbf{y}_h^{(0)}$ and $\mathbf{u}_h^{(0)}$ and N processors P_i, $i = 0, \dots, N-1$
Set $N_G = 1$, $\mathbf{U}_0^0 = (\mathbf{y}_h^{(0)}, \mathbf{u}_h^{(0)})$, $i = 0, \dots, N-1$ and $k = 0$
Processor P_0 propagates operator $G(N_G, \mathbf{U}_i^0)$ $\forall i$ to obtain $\mathbf{U}_{i+1}^0$
while $\|\phi_1\|_2 > tol_1$ or $\|\phi_2\|_2 > tol_2$ **do**
    Processor P_0 sends $\mathbf{U}_{i+1}^k = G(N_G, \mathbf{U}_i^k)$ $\forall i = 0, \dots, N-2$ respectively to processors $P_1, \dots, P_{N-1}$
    Processors P_i propagate in parallel $F(N_F, \mathbf{U}_i^k)$ $\forall i$
    Each processor P_i $\forall i = 1, \dots, N-1$ sends the obtained $\mathbf{U}_{i+1}^k$ to processor P_0
    Processor P_0 performs the correction iteration (1.29) to obtain $\mathbf{U}_{i+1}^{k+1}$ $\forall i$
    Set $\mathbf{U}_0^{k+1} = \mathbf{U}_N^{k+1}$
    Processor P_0 propagates operator $G(N_G, \mathbf{U}_i^{k+1})$ $\forall i$
    Set $k = k+1$
end while

Monitoring the progress in terms of achieved cost functional J of the operators F at each iteration of Algorithm 1.3, we observed that the different operators obtain descent trajectories that are overlapping to a great extent. To improve the performance of the algorithm on this point we propose the following alternative.

1.7. ParaFlowS

The ParaFlowS (ParaFlow in Series) algorithm is based upon the ParaFlow Algorithm 1.3 but introduces a different approach for the initialization of operator F which no longer

requires the fine solvers to operate in parallel.
At each iteration k, ParaFlow initializes N (as the number of processors) fine solvers F with $\mathbf{U}_{i+1}^k \ \forall i = 0, \dots, N-1$ obtained by the correction iteration (1.29). Since operator G performs a unique iteration per operator F, these initial conditions used to initialize the different fine solvers F are similar one another and therefore the N parallel trajectories overlap to a great extent.
To solve this issue, the ParaFlowS algorithm proposes to use a unique solver F which is initialized at each iteration with the initial value among $\mathbf{U}_{i+1}^k \ \forall i = 0, \dots, N-1$ that achieves the minimum value of the cost functional J. In this framework, N no longer represents the number of processors but it is an additional parameter to be optimized. Operator G and the correction iteration (1.29) are instead defined as in the ParaFlow algorithm.
The pseudocode of the ParaFlowS is given in Algorithm 1.4.

Algorithm 1.4 ParaFlowS

Given N_F, N and the initial values $\mathbf{y}_h^{(0)}$ and $\mathbf{u}_h^{(0)}$
Set $N_G = 1$, $\mathbf{U}_0^0 = (\mathbf{y}_h^{(0)}, \mathbf{u}_h^{(0)})$, $i = 0, \dots, N-1$ and $k = 0$
Propagate operator $G(N_G, \mathbf{U}_i^0) \ \forall i$ to obtain $\mathbf{U}_{i+1}^0 \ \forall i$
while $\|\phi_1\|_2 > tol_1$ or $\|\phi_2\|_2 > tol_2$ **do**
    Find $\mathbf{U}_{\tilde{i}+1}^k = \underset{\forall i=0,\dots,N-1}{\operatorname{argmin}} \ J(\mathbf{U}_{i+1}^k)$
    Propagate $F(N_F, \mathbf{U}_{\tilde{i}+1}^k)$
    Perform the correction iteration in (1.29) by using the same $F(N_F, \mathbf{U}_{\tilde{i}+1}^k) \ \forall i$ to obtain $\mathbf{U}_{i+1}^{k+1} \ \forall i$
    Set $k = k + 1$
end while

2 | Implementation and C++ code

In this chapter, we report and analyze the C++ library that implements the algorithms described in Chapter 1, employed to find a numerical solution of the discrete optimal control problem (1.5)-(1.6). The library is based on the `deal.ii` [5] library from which it collects core structures and algebraic methods for the finite elements method implementation.

2.1. General Code Structure

The library is composed of different template classes which can be distinguished in two main groups:

- The **core** classes, which implement fundamental methods on which all the numerical algorithms rely on;
- The **numerical methods** classes, which implement the GradientFlow, ParaReal, ParaFlow and ParaFlowS algorithms.

All the classes can be specialized based on `<unsigned int dim>`, which can be chosen as equal to 2 or 3 and denotes the space dimension.

2.2. Core Classes

The **core** classes, which can be found in the `include/utils/` folder, are the following:

- The `LinearSystem` class;
- The `DescentStepBase` abstract class and its derived classes `DescentStepEuler` and `DescentStepAdam`.

Given an initial condition, the two derived classes are used to propagate successive approximations of vectors y_h and u_h according to the corresponding update rules illustrated in chapter 1. At each step, the different descent directions are computed with the vectors ϕ_1 and ϕ_2, obtained by solving the linear system (1.16), which is represented by an object

of the `LinearSystem` class.

2.2.1. LinearSystem

The class `LinearSystem<unsigned int dim>` relies on the `deal.ii` library and represents the linear system (1.16)

$$\underbrace{\begin{bmatrix} I & 0 & \tilde{K}(y_h) \\ 0 & I & -M \\ \tilde{K}(y_h) & -M & 0 \end{bmatrix}}_{A} \begin{bmatrix} \phi_1(y_h, u_h) \\ \phi_2(y_h, u_h) \\ \Lambda(y_h, u_h) \end{bmatrix} = \begin{bmatrix} -\nabla_{y_h} J(y_h, u_h) \\ -\nabla_{u_h} J(y_h, u_h) \\ -g(y_h, u_h) \end{bmatrix} \tag{2.1}$$

where

$$\begin{cases} \tilde{K}(y_h) := K + \gamma M \varphi'(y_h) \\ g(y_h, u_h) := K y_h + \gamma M \varphi(y_h) - M u_h \\ \nabla_{y_h} J(y_h, u_h) = M(y_h - y_{d_h}) \\ \nabla_{u_h} J(y_h, u_h) = \nu M u_h \end{cases} . \tag{2.2}$$

The main purpose of the class is to first assemble matrix A and the right-hand side vector and then to obtain vectors ϕ_1 and ϕ_2 by solving the linear system. A brief explanation of the implemented methods is reported here below.

First of all, the domain Ω, assumed to be equal to the square of side $[-1, 1]$, is discretized with the `make_grid()` method, generating a rectangular grid characterized by a number of nodes that depends on the number of times it is refined (`grid_refinement` attribute). Indeed, the initial grid has 2 elements on each side of the square domain. Refining the grid 2 times leads to a grid with $2^2 = 4$ elements per side, refining it 3 times leads to a grid with $2^3 = 8$ elements per side and so on. The chosen finite elements to describe the shape functions are the bi-linear Lagrange elements, denoted by the attribute `fe` of type `dealii::FE_Q`.

After having enumerated the degrees of freedom on the mesh with the `dealii::DofHandler` object, we initialize with zeros the node coefficient vectors of y_h and u_h, represented respectively with the `dealii::Vector` objects `y_vec` and `u_vec`. The size `vector_size` of `y_vec` and `u_vec` is equal to the number of degrees of freedom of the mesh, which coincides with $(\textit{number of elements per side} + 1)^2$ in our case, since we are using Q1 Lagrange elements.

The next step consists in the assembly of the mass and stiffness matrices M (`mass_matrix` attribute) and K (`stiffness_matrix` attribute) in (1.7), represented by `dealii::SparseMatrix` objects which store the data of sparse matrices in the compressed

row storage (CRS) format. A `dealii::DynamicSparsityPattern` object, containing the possible non-zero entries of matrices M and K, is used to initialize them, while the quadrature formula chosen for the evaluation of the integrals on each cell is a Gauss formula with two quadrature points in each direction. The method `assemble_matrices()` initially computes in a small matrix the contributions of each cell to the matrices M and K, by taking into account the degrees of freedom on that specific cell. When the computations on that cell are finished, then the contributions are transferred to to the global matrices.
Afterwards, the method `assemble_KTilde()` can be used to assemble matrix $\tilde{K}(y_h)$ (2.2), once again represented by a `dealii::SparseMatrix` object, which can be initialized with the same `dealii::DynamicSparsityPattern` object `sparsity_pattern` used for M and K. To compute $\tilde{K}(y_h)$, we need to specify φ, a nonlinear function of y_h defined as $\varphi(y_h) = \exp(y_h)$, and its diagonal Jacobian matrix, $\varphi'(y_h)$, which are both defined as lambda functions in the method `set_phi()`. In this framework, we are considering the simplified case in which we assume that $\varphi(y_h)$ and $\varphi'(y_h)$ can be approximated by directly evaluating φ and φ' at the node coefficients of y_h. Therefore, the values of the `dealii::Vector` objects `phi_vec` and `Jacobian_phi` can be directly computed by using `y_vec` and `u_vec` in the method `set_phi()`.
It is now possible to assemble matrix A, represented with a `dealii::SparseMatrix` object named `A_matrix`. First of all, it has to be initialized with the `dealii::DynamicSparsityPattern` object created in the `set_global_sparsity_pattern()` method, which copies the entries of `sparsity_pattern` in correspondence of the $\tilde{K}(y_h)$ and $-M$ blocks of A, and adds additional entries on the diagonal corresponding to its two identity matrices. In a second moment, with the method `assemble_A()` the values of each entry are assigned based on the matrices $\tilde{K}(y_h)$ and $-M$.
Furthermore, we can set the values of the right-hand side vector, represented by the `dealii::Vector` object `rhs_vec`, with the method `assemble_rhs()`, which evaluates g (2.2) and the gradients of J with respect to y_h and u_h, given `y_vec` and `u_vec`.
The most significant method implemented by this class is the `solve()` method. Once `A_matrix` and `rhs_vec` have been assembled and the boundary conditions have been applied, this method solves the linear system by means of the `dealii::SolverMinRes` object which implements the Minimal Residual Method, a numerical method for the iterative solution of a symmetric system of equations as the one we are considering.

We will now give a brief overview of the implemented public methods.
The class constructor takes as input three parameters: γ and ν, which are used to define the discrete optimal control problem and N, the number of times the grid is refined.

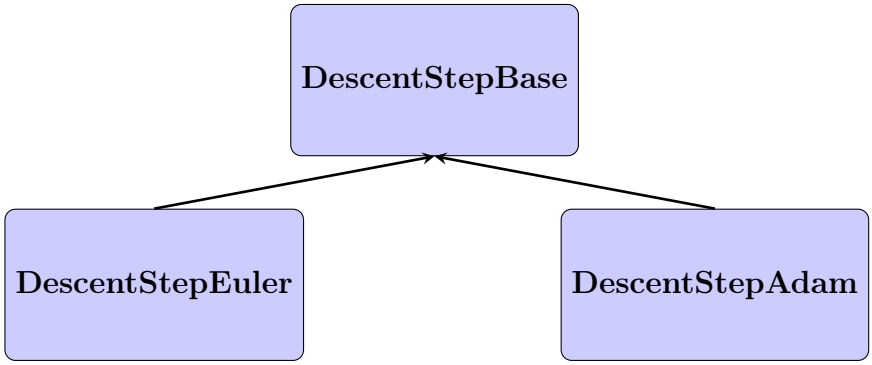
```
template <unsigned int dim>
LinearSystem<dim>::LinearSystem(const double gamma_val, const double
    nu_val, const unsigned int N)
:
  //...
  , grid_refinement(N)
  , nu(nu_val)
  , gamma(gamma_val)
{
  setup_linear_system();
  //...
}
```

Listing 2.1: LinearSystem<dim>::LinearSystem

During the object construction, the constructor executes `setup_linear_system()` which runs the methods required to discretize the domain Ω, to assemble the mass and stiffness matrices M and K and to set up `global_sparsity_pattern`. It is important to note that these methods do not depend on the node coefficients values `y_vec` and `u_vec`.
Other significant public methods of the class are the following:

- `update_vectors`, which allows to update the node coefficients `y_vec` and `u_vec`;
- `solve_system()`, which first assembles the objects that depend on `y_vec` and `u_vec` (i.e. `K_Tilde`, `A_Matrix` and `rhs_vec`) and then executes `solve()` to compute the solution of the system;
- `get_phi1()` and `get_phi2()`, which return respectively the first two components of the solution vector, `phi1_vec` and `phi2_vec`;
- `evaluate_J()` and `evaluate_g()`, which return respectively the cost functional J and the L^2-norm of g defined in (2.2), both obtained with the current `y_vec` and `u_vec`;
- `output_result_vectors()`, which allows to export `y_vec` and `u_vec` in VTK format to be visualized with a tool as, for example, VisIt;
- `projected_moments`, which allows to compute the projections on the manifold V_g of the two descent directions in input, necessary for the Adam update rule in (1.24).

2.2.2. DescentStep



DescentStepBase

The class `DescentStepBase<unsigned int dim>` is an abstract class which implements fundamental methods to obtain successive approximations of the solution vectors y_h and u_h of the autonomous system (1.15). The class has a pure virtual method `vectors_iteration_step()` that specifies the update rule used to compute $y_h^{(k+1)}$ and $u_h^{(k+1)}$ from $y_h^{(k)}$ and $u_h^{(k)}$ which can be of *Explicit Euler* or *Adam* type and is implemented respectively in the two derived classes `DescentStepEuler` and `DescentStepAdam`. The class interface is the following:

```
template <unsigned int dim>
class DescentStepBase
{
public:

  DescentStepBase();
  DescentStepBase(const double gamma_val, const double nu_val, const
   unsigned int N);
  virtual void set_initial_vectors(const Vector<double>& y0, const
   Vector<double>& u0);
  void set_step_size(const double step_sz);
  const Vector<double>& get_y_vec() const { return y_vec; };
  const Vector<double>& get_u_vec() const { return u_vec; };
  void run();
  void run(const unsigned int n_iter);
  //...

protected:
  bool converged() const;
  //...
```

```
  void descent_step();
  virtual void vectors_iteration_step() = 0;

  LinearSystem<dim>          linear_system;
  //...
  Vector<double>             y_vec;
  Vector<double>             u_vec;
  Vector<double>             phi1;
  Vector<double>             phi2;
  double                    step_size;
  //...
};
```

Listing 2.2: class DescentStepBase<dim>

The `descent_step()` method computes $y_h^{(k+1)}$ and $u_h^{(k+1)}$ from $y_h^{(k)}$ and $u_h^{(k)}$ by solving the linear system associated to $y_h^{(k)}$ and $u_h^{(k)}$ with the `solve_system()` method of the `linear_system` attribute, by extracting the solution subvectors $\phi_1(y_h^{(k)}, u_h^{(k)})$ and $\phi_2(y_h^{(k)}, u_h^{(k)})$ respectively in `phi1` and `phi2`, and by running `vectors_iteration_step()` with the updated `phi1` and `phi2`. When $y_h^{(k+1)}$ and $u_h^{(k+1)}$ have been computed, the attributes `y_vec` and `u_vec` of `linear_system` are updated for the next iteration.
The most relevant method of this class is the `run` method which has been overloaded to either accept no parameters in input or to accept `n_iter` as input parameter. In the first case, the method iterates the computation of `descent_step()` until the convergence criteria based on the L^2-norms of ϕ_1 and ϕ_2 (checked by the `converged()` method) is met. In the second case, instead, it computes `descent_step()` `n_iter` times if the convergence criteria is not met beforehand.

DescentStepEuler

The class `DescentStepEuler<unsigned int dim>` inherits from `DescentStepBase<unsigned int dim>` and overrides the virtual method `vectors_iteration_step()` by implementing the Explicit Euler update rule (1.18) as update rule to compute $y_h^{(k+1)}$ and $u_h^{(k+1)}$ starting from $y_h^{(k)}$ and $u_h^{(k)}$ and given the descent directions $\phi_1(y_h^{(k)}, u_h^{(k)})$ and $\phi_2(y_h^{(k)}, u_h^{(k)})$. The implemented method is the following:

```
template<unsigned int dim>
void DescentStepEuler<dim>::vectors_iteration_step()
{
  this->y_vec.add(this->step_size, this->phi1);
  this->u_vec.add(this->step_size, this->phi2);
```

```
}
```

Listing 2.3: DescentStepEuler<dim>::vectors_iteration_step()

DescentStepAdam

The class `DescentStepAdam<unsigned int dim>` inherits from `DescentStepBase<unsigned int dim>` and overrides the virtual method `vectors_iteration_step()` by implementing the Adam update rule (1.24) as update rule to compute $y_h^{(k+1)}$ and $u_h^{(k+1)}$ starting from $y_h^{(k)}$ and $u_h^{(k)}$ and given the descent directions $\phi_1(y_h^{(k)}, u_h^{(k)})$ and $\phi_2(y_h^{(k)}, u_h^{(k)})$. The first and second order momentum terms are represented respectively by the `dealii::Vector` objects `m_y`, `m_u` and `v_y`, `v_u`. Instead, the parameters β_1, β_2 and ϵ of the Adam update rule are represented by the attributes `beta_1`, `beta_2` and `epsilon`. The implemented method is the following:

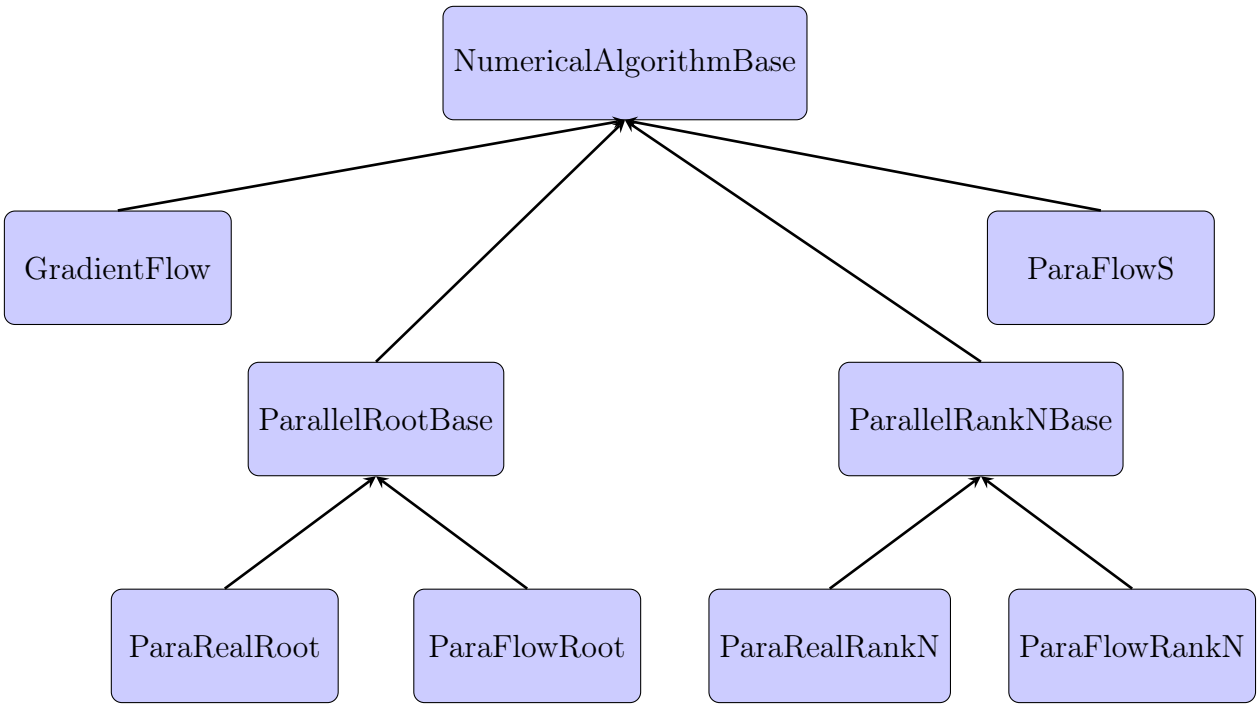
```
template<unsigned int dim>
void DescentStepAdam<dim>::vectors_iteration_step()
{
  //The first and second order momentum terms are updated for y with the
    new descent direction phi1
  m_y *= beta1;
  m_y.add(1-beta1, this->phi1);

  Vector<double> phi1_squared(this->phi1);
  phi1_squared.scale(this->phi1);
  v_y *= beta2;
  v_y.add(1-beta2, phi1_squared);

  Vector<double> m_hat(m_y);
  Vector<double> v_hat(v_y);

  m_hat /= (1-beta1);
  v_hat /= (1-beta2);

  //The descent direction is computed but it still requires the
   projection on Vg
  Vector<double> temp_y(this->dim_vec);
  for (unsigned int i=0; i<this->dim_vec; i++)
    temp_y(i) = m_hat(i)/(std::sqrt(v_hat(i))+eps);

  //The first and second order momentum terms are updated for u with the
    new descent direction phi2
  m_u *= beta1;
  m_u.add(1-beta1, this->phi2);
```

```
27
28    Vector<double> phi2_squared(this->phi2);
29    phi2_squared.scale(this->phi2);
30    v_u *= beta2;
31    v_u.add(1-beta2, phi2_squared);
32
33    Vector<double> m_hat_u(m_u);
34    Vector<double> v_hat_u(v_u);
35
36    m_hat_u /= (1-beta1);
37    v_hat_u /= (1-beta2);
38
39    //The descent direction is computed but it still requires the
       projection on Vg
40    Vector<double> temp_u(this->dim_vec);
41    for (unsigned int i=0; i<this->dim_vec; i++)
42      temp_u(i) = m_hat_u(i)/(std::sqrt(v_hat_u(i))+eps);
43
44    //The computed descent directions are projected on Vg by solving the
       linear system that substitutes them to the gradient of -J in the rhs
       vector
45    this->linear_system.projected_moments(temp_y, temp_u);
46
47    Vector<double> proj_y(this->linear_system.get_projection_vec1());
48    Vector<double> proj_u(this->linear_system.get_projection_vec2());
49
50    //y_vec and u_vec are updated with the descent directions
51    this->y_vec.add(this->step_size, proj_y);
52    this->u_vec.add(this->step_size, proj_u);
53
54  }
```

Listing 2.4: DescentStepAdam<dim>::vectors_iteration_step()

2.3. Numerical Methods Classes

In this section we describe the **numerical methods** classes, that can be found in the `include/numerical_methods/` folder, and which implement the algorithms described in chapter 1 for the numerical solution of the discrete optimal control problem (1.5)-(1.6). An abstract class `NumericalAlgorithmBase<unsigned int dim>` provides common methods to all the classes implementing the different algorithms, which inherit from it. The class hierarchy can be represented as follows:



2.3.1. NumericalAlgorithmBase

The class `NumericalAlgorithmBase<unsigned int dim>` is an abstract class which declares two pure virtual methods: the `run()` and `get_numerical_method_params(const std::string& filename)` methods. The derived classes must override the first method to provide the specific algorithm that characterizes them, while instead they override the second method to read from file the different parameters specific to each method, relying on the *nlohmann-json* library.

The core structure of the class is the following:

```
template<unsigned int dim>
class NumericalAlgorithmBase
{
public:
  NumericalAlgorithmBase(){};
  NumericalAlgorithmBase(const std::string& filename);
  NumericalAlgorithmBase(const double gamma_val, const double nu_val,
   const unsigned int N);
  virtual void run() = 0;

protected:
  virtual void get_numerical_method_params(const std::string& filename)
   = 0;
```

```
12   std::unique_ptr<DescentStepBase<dim>> create_GF(const GFStepType
      GFMethod);
13   void get_linear_system_params(const std::string& filename);
14
15   double  gamma;
16   double  nu;
17   unsigned int  grid_refinement;
18 };
```

Listing 2.5: class NumericalAlgorithmBase<unsigned int dim>

Other than the default constructor, the class provides two different constructors to obtain in two different ways the parameters for the initialization of the `LinearSystem<dim>` class (i.e. `gamma`, `nu` and `grid_refinement`), which are common to all numerical methods. While the first constructor allows to directly provide them as input, the second one reads them from a `.json` file by means of the `get_linear_system_params` method which requires to enter the name of the file in which they are stored.
The `create_GF` method, instead, is provided to all classes to return a `std::unique_ptr` pointer to one of the classes derived from the `DescentStepBase<dim>` class, based on the chosen update rule that will be specified on the parameter file of each numerical method. The code is the following:

```
1  template<unsigned int dim>
2  std::unique_ptr<DescentStepBase<dim>> NumericalAlgorithmBase<dim>::
       create_GF(const GFStepType GFMethod)
3  {
4    switch(GFMethod)
5    {
6          case GFStepType::EULER:
7              return std::make_unique<DescentStepEuler<dim>>(gamma, nu,
       grid_refinement);
8          case GFStepType::ADAM:
9              return std::make_unique<DescentStepAdam<dim>>(gamma, nu,
       grid_refinement);
10         default:
11             std::cerr << "Solver not implemented.\n" << std::endl;
12             return nullptr;
13   }
14 }
```

Listing 2.6: NumericalAlgorithmBase<dim>::create_GF

2.3.2. GradientFlow

The `GradientFlow<unsigned int dim>` class inherits from the `NumericalAlgorithmBase<dim>` class and implements Algorithm 1.1. Its class interface is the following:

```
template<unsigned int dim>
class GradientFlow: public NumericalAlgorithmBase<dim>
{
public:
  GradientFlow(const std::string& linear_system_filename, const std::
   string& ParaFlowS_params_filename);
  GradientFlow(const double gamma_val, const double nu_val, const
   unsigned int N_grid, const std::string& GF_params_filename);
  void run() override;

private:
  void get_numerical_method_params(const std::string& filename) override
   ;

  std::unique_ptr<DescentStepBase<dim>> gf;
  GFStepType          MethodOperatorGF;
  double              step_size;

};
```

Listing 2.7: class GradientFlow<unsigned int dim>

The input parameters that characterize this method are the update rule to use and its step size, which are both read from file with the `get_numerical_method_params(const std::string& filename)` method, overriden as follows:

```
template<unsigned int dim>
void GradientFlow<dim>::get_numerical_method_params(const std::string&
   filename)
{
  std::ifstream file(filename);
  json parameters;
  file >> parameters;
  file.close();

  step_size = parameters["step_size"];
  std::string solver = parameters["Solver"];

  if (solver == "Euler")
    MethodOperatorGF = GFStepType::EULER;
  else if (solver == "Adam")
```

```
15      MethodOperatorGF = GFStepType::ADAM;
16    else
17      std::cerr << "Solver not implemented.\n" << std::endl;
18  }
```

Listing 2.8: GradientFlow<dim>::get_numerical_method_params

The two constructors take as input the file name with the numerical method's parameters and can either directly take as input the parameters for the `LinearSystem<dim>` class or the file name in which they are stored. Once the update rule type has been read, the constructor initializes `gf` accordingly and sets the specified step size. For this class, the overriden `run()` method simply executes the `DescentStepBase<dim>::run()` method, as specified in Algorithm 1.1.

2.3.3. Parallel Computing with MPI

The ParaReal and ParaFlow algorithms require to perform computations in parallel. For this purpose, this library leverages the Message Passing Interface (MPI) and a number of utility functions that rely on it, declared in the `dealii::Utilities::MPI` namespace. Based on how the different processors operate among them, we distinguish them in *Root* processor and *RankN* processors. More specifically, the *Root* processor is the only processor that can send and receive messages from all the other processors and is identified as processor 0. All the other processors, defined as *RankN* processors, can instead exchange messages only with the *Root* and therefore not among themselves.

To implement Algorithms 1.2 and 1.3 it is necessary to discern between the operations computed by the *Root* and the ones computed by all the other processors. Indeed, the flow of the algorithms is managed by the *Root*, while the *RankN* processors instead perform computations only based on the messages that they receive from the *Root*. This different behavior leads to the need of creating two different abstract classes, `ParallelRootBase<unsigned int dim>` and `ParallelRankNBase<unsigned int dim>` that inherit from the `NumericalAlgorithmBase<unsigned int dim>` class and from which the classes specified for both algorithms derive.

ParallelRootBase

The `ParallelRootBase<unsigned int dim>` class inherits from the `NumericalAlgorithmBase<dim>` class and is an abstract class itself since it does not override the `run()` and the `get_numerical_method_params` methods. It is a class providing the common attributes and methods used by the *Root* processor for both the ParaReal

and ParaFlow algorithms. A struct with the following types is defined as

```
struct VectorTypes{
    using ArrayType = std::array<Vector<double>, 2>;
    using VectorArrayType = std::vector<ArrayType>;
    using TupleType = std::tuple<ArrayType, bool>;
    using FutureArrayType = Utilities::MPI::Future<ArrayType>;
    using FutureTupleType = Utilities::MPI::Future<TupleType>;
}
```

Listing 2.9: struct VectorTypes

while the class interface is the following:

```
template<unsigned int dim>
class ParallelRootBase: public NumericalAlgorithmBase<dim>
{
public:
  using VT = VectorTypes;
  ParallelRootBase(const std::string& linear_system_filename);
  ParallelRootBase(const double gamma_val, const double nu_val, const
   unsigned int N_grid);

protected:
  bool check_convergence();

  MPI_Comm                                  mpi_communicator;
  const unsigned int                        n_mpi_processes;
  const unsigned int                        this_mpi_process;

  bool                                   converged;

  std::unique_ptr<DescentStepBase<dim>> gf_G;
  std::unique_ptr<DescentStepBase<dim>> gf_F;

  GFStepType                    MethodOperatorG;
  GFStepType                    MethodOperatorF;
  double                        step_size_G;
  double                        step_size_F;
  unsigned int                   n_iter_G;
  unsigned int                   n_iter_F;
  VT::VectorArrayType                  F_vectors;
  VT::VectorArrayType                  G_old_vectors;
  VT::VectorArrayType                  G_new_vectors;
  VT::VectorArrayType                  new_yu_vectors;

```

```
  std::vector<bool>                 converged_vec;
  unsigned int                        converged_rank;
};
```

Listing 2.10: class ParallelRootBase<unsigned int dim>

The object `dealii::Utilities::MPI::MPI_Comm` is the communicator, while the attributes `n_mpi_processes` and `this_mpi_processes` represent respectively the number of processors in the communicator and the rank of the present MPI process. The attributes `MethodOperatorG` and `MethodOperatorF` represent the chosen update rule type for the operators G and F, while `gf_G` and `gf_F` are the `std::unique_ptr` pointers to the corresponding derived classes of `DescentStepBase<dim>`. The attributes `F_vectors`, `G_old_vectors` and `G_new_vectors` have size `n_mpi_processes` and contain respectively the values propagated by operator F at iteration k, operator G at iteration k and operator G at iteration $k+1$. The values obtained with the correction iteration at step $k+1$ are instead stored in `new_yu_vectors`.

ParallelRankNBase

The `ParallelRankNBase<unsigned int dim>` class inherits from the `NumericalAlgorithmBase<dim>` class and is an abstract class itself since it overrides the `run()` method but not the `get_numerical_method_params` method. It is a class providing the common attributes and methods used by the *RankN* processors for both the ParaReal and ParaFlow algorithms. The class interface is the following:

```
class ParallelRankNBase: public NumericalAlgorithmBase<dim>
{
public:
  using VT = VectorTypes;
  ParallelRankNBase(const std::string& linear_system_filename);
  ParallelRankNBase(const double gamma_val, const double nu_val, const
   unsigned int N_grid);
  void run() override;

protected:
  MPI_Comm                                mpi_communicator;
  const unsigned int                        n_mpi_processes;
  const unsigned int                        this_mpi_process;

  bool                 converged;
  bool                   convergence_F;

  std::unique_ptr<DescentStepBase<dim>> gf_F;
```

```
18
19    GFStepType                MethodOperatorF;
20    double                    step_size_F;
21    unsigned int                n_iter_F;
22
23    VT::ArrayType                              initial_time_vectors;
24    VT::ArrayType                              final_time_vectors;
25
26  }
```

Listing 2.11: class ParallelRankNBase<unsigned int dim>

We can observe that, unlike the `ParallelRootBase<unsigned int dim>` class, this class only requires operator F, whose update rule type depends on `MethodOperatorF`. Moreover, it cannot access the values obtained by the correction iterations. Indeed, it only presents attributes `initial_time_vectors` and `final_time_vectors` which store respectively the initial values for operator F, received at each iteration by the *Root*, and the values obtained propagating operator F with these initial conditions, that will be sent back to the *Root*. The overriden `run()` method is the following:

```
1  template<unsigned int dim>
2  void ParallelRankNBase<dim>::run()
3  {
4    while(!converged)
5    {
6        // Receive initial conditions from Root
7        VT::FutureArrayType future = Utilities::MPI::irecv<VT::ArrayType>(
     mpi_communicator, root);
8        initial_time_vectors = future.get();
9        // Set received values as initial conditions of operator F
10       gf_F->set_initial_vectors(initial_time_vectors[0],
     initial_time_vectors[1]);
11       //Propagate operator F
12       gf_F->run(n_iter_F);
13       convergence_F = std::get<0>(gf_F->convergence_info());
14       final_time_vectors[0] = gf_F->get_y_vec();
15       final_time_vectors[1] = gf_F->get_u_vec();
16       // Send tuple with the vectors obtained by operator F and its
     convergence to the root
17       VT::TupleType tuple_to_send = std::make_tuple(final_time_vectors,
     convergence_F);
18       Utilities::MPI::isend(tuple_to_send, mpi_communicator, root);
19       // Receive convergence result from the root
20       Utilities::MPI::Future<bool> future1 = Utilities::MPI::irecv<bool
```

```
    >(mpi_communicator, root);
21      converged = future1.get();
22  }
23 }
```

Listing 2.12: ParallelRankNBase<dim>::run()

2.3.4. Parareal

ParaRealRoot

The `ParaRealRoot<unsigned int dim>` class inherits from the `ParallelRootBase<dim>` class and overrides the `run()` and the `get_numerical_method_params` methods by implementing Algorithm 1.2 and by specifying to read from file the parameters that characterize the Parareal algorithm, which are the time interval length T, the solvers to use for operators G and F and their step sizes. Therefore, this class presents the additional attribute `global_T` and the method `set_number_iter()` which computes the number of iterations to perform by operators G and F. Moreover, it defines also the `check_pr_interval_convergence()` method to check the Parareal convergence criterion (1.36), used to determine if it is necessary to shift the time interval taken into account. The class interface is the following:

```
1  template<unsigned int dim>
2  class ParaRealRoot: public ParallelRootBase<dim>
3  {
4  public:
5    using VT = VectorTypes;
6    ParaRealRoot(const std::string& linear_system_filename, const std::
       string& ParaReal_params_filename);
7    ParaRealRoot(const double gamma_val, const double nu_val, const
       unsigned int N_grid, const std::string& ParaReal_params_filename);
8    void run() override;
9
10 private:
11   void get_numerical_method_params(const std::string& filename) override
       ;
12   bool check_pr_interval_convergence();
13   void set_number_iter();
14
15   double                          global_T;
16   double                          epsilon=1e-3;
17   VT::VectorArrayType             old_yu_vectors;
18   VT::ArrayType                   initial_vectors_F;
```

```
};
```

Listing 2.13: class ParaRealRoot<unsigned int dim>

while the code for the `run()` method is:

```
template<unsigned int dim>
void ParaRealRoot<dim>::run()
{
  unsigned int total_n_it=0; //number of iterations in series
  std::cout << "Initialization" << std::endl;
  //Initial conditions are computed by propagating operator G
  for (unsigned int i=0; i<this->n_mpi_processes; i++)
  {
    this->gf_G->run(this->n_iter_G);
    Vector<double> y = this->gf_G->get_y_vec();
    Vector<double> u = this->gf_G->get_u_vec();
    this->G_new_vectors[i][0] = y;
    this->G_new_vectors[i][1] = u;
    this->new_yu_vectors[i][0] = y;
    this->new_yu_vectors[i][1] = u;
    this->gf_G->output_iteration_results();
  }
  this->G_old_vectors = this->G_new_vectors;
  old_yu_vectors = this->new_yu_vectors;
  total_n_it += this->n_iter_G*this->n_mpi_processes;

  unsigned int it=0;
  while(!this->converged)
  {
      std::cout << "Iteration n: " << it+1 << std::endl;
      //Root sends initial conditions to all other ranks
      for (unsigned int rank=1; rank<this->n_mpi_processes; rank++)
      {
        Utilities::MPI::isend(this->new_yu_vectors[rank-1], this->
   mpi_communicator, rank);
      }
      // Root propagates its F operator
      this->gf_F->set_initial_vectors(initial_vectors_F[0],
   initial_vectors_F[1]);
      this->gf_F->run(this->n_iter_F);
      this->F_vectors[0][0] = this->gf_F->get_y_vec();
      this->F_vectors[0][1] = this->gf_F->get_u_vec();
      this->converged_vec[0] = std::get<0>(this->gf_F->convergence_info
   ());
```

```
37        // Root receives F operator and convergence results from all other
      ranks
38        for (unsigned int rank=1; rank<this->n_mpi_processes; rank++)
39        {
40          VT::FutureTupleType future = Utilities::MPI::irecv<VT::TupleType
     >(this->mpi_communicator, rank);
41          VT::TupleType fut_tuple = future.get();
42          this->F_vectors[rank] = std::get<0>(fut_tuple);
43          this->converged_vec[rank] = std::get<1>(fut_tuple);
44        }
45        total_n_it += this->n_iter_F;
46        //Root computes convergence and sends result to other ranks
47        this->converged = this->check_convergence();
48        for (unsigned int rank=1; rank<this->n_mpi_processes; rank++)
49        {
50          Utilities::MPI::isend(this->converged, this->mpi_communicator,
     rank);
51        }
52
53        if(!this->converged)
54        {
55       // Update the values of y and u obtained from Root (operators G
      cancel out for the root since G_old and G_new are computed with the
      same initial condition)
56    this->new_yu_vectors[0][0] = this->F_vectors[0][0];
57    this->new_yu_vectors[0][1] = this->F_vectors[0][1];
58
59          this->G_old_vectors = this->G_new_vectors;
60          for (unsigned int rank=1; rank<this->n_mpi_processes; rank++)
61          {
62            // Update G
63            this->gf_G->set_initial_vectors(this->new_yu_vectors[rank
     -1][0], this->new_yu_vectors[rank-1][1]);
64            this->gf_G->run(this->n_iter_G);
65            this->G_new_vectors[rank][0] = this->gf_G->get_y_vec();
66            this->G_new_vectors[rank][1] = this->gf_G->get_u_vec();
67            // Perform correction iteration
68            Vector<double> y_temp(this->G_new_vectors[rank][0]);
69            y_temp -= this->G_old_vectors[rank][0];
70            y_temp += this->F_vectors[rank][0];
71            Vector<double> u_temp(this->G_new_vectors[rank][1]);
72            u_temp -= this->G_old_vectors[rank][1];
73            u_temp += this->F_vectors[rank][1];
74            this->new_yu_vectors[rank][0] = y_temp;
```

```
75          this->new_yu_vectors[rank][1] = u_temp;
76        }
77        total_n_it += this->n_iter_G*(this->n_mpi_processes-1);
78
79        // Output results of this iteration process
80        for (unsigned int rank=0; rank<this->n_mpi_processes; rank++)
81        {
82          this->gf_G->set_initial_vectors(this->new_yu_vectors[rank][0],
    this->new_yu_vectors[rank][1]);
83          this->gf_G->output_iteration_results();
84        }
85      }
86      // Check the ParaReal convergence criterion
87      if (!this->converged && check_pr_interval_convergence())
88      {
89        //Time interval is shifted by imposing U_0=U_N
90        this->new_yu_vectors[0] = this->new_yu_vectors[this->
   n_mpi_processes-1];
91        this->gf_G->set_initial_vectors(this->new_yu_vectors[0][0], this
   ->new_yu_vectors[0][1]);
92        initial_vectors_F[0] = this->new_yu_vectors[0][0];
93        initial_vectors_F[1] = this->new_yu_vectors[0][1];
94        this->gf_F->set_initial_vectors(initial_vectors_F[0],
   initial_vectors_F[1]);
95        //Compute initial conditions on the new interval with G
96        for (unsigned int i=0; i<this->n_mpi_processes; i++)
97        {
98          this->gf_G->run(this->n_iter_G);
99          Vector<double> y = this->gf_G->get_y_vec();
100         Vector<double> u = this->gf_G->get_u_vec();
101         this->G_new_vectors[i][0] = y;
102         this->G_new_vectors[i][1] = u;
103         this->new_yu_vectors[i][0] = y;
104         this->new_yu_vectors[i][1] = u;
105       }
106       total_n_it += this->n_iter_G*this->n_mpi_processes;
107     }
108     old_yu_vectors = this->new_yu_vectors;
109     it++;
110 }
111
112 std::cout << "Final results " << std::endl;
113 std::cout << "ParaReal converged in: " << total_n_it << " iterations"
    << std::endl;
```

```
114   this->gf_G->set_initial_vectors(this->F_vectors[this->converged_rank
       ][0], this->F_vectors[this->converged_rank][1]);
115   this->gf_G->output_iteration_results(); //prints J and norm of g
116   this->gf_G->output_results_vectors(); //outputs obtained vectors y and
        u to .vtk files for visualization
117 }
```

Listing 2.14: ParaRealRoot<dim>::run()

ParaRealRankN

The `ParaRealRankN<unsigned int dim>` class inherits from the `ParallelRankNBase<dim>` class and overrides the `get_numerical_method_params` method by specifying to read from file the parameters that characterize the Parareal algorithm for the *RankN* processors which are just the time interval length T, the solver to use for operator F and its step size since operator G is not used. The class interface is the following:

```
1  template<unsigned int dim>
2  class ParaRealRankN: public ParallelRankNBase<dim>
3  {
4  public:
5    using VT = VectorTypes;
6    ParaRealRankN(const std::string& linear_system_filename, const std::
       string& ParaReal_params_filename);
7    ParaRealRankN(const double gamma_val, const double nu_val, const
       unsigned int N_grid, const std::string& ParaReal_params_filename);
8
9
10 private:
11   void get_numerical_method_params(const std::string& filename) override
       ;
12   void set_number_iter();
13
14   double  global_T;
15
16 };
```

Listing 2.15: class ParaRealRankN<unsigned int dim>

2.3.5. ParaFlow

ParaFlowRoot

The `ParaFlowRoot<unsigned int dim>` class inherits from the `ParallelRootBase<dim>` class and overrides the `run()` and the `get_numerical_method_params` methods by implementing Algorithm (1.3) and by specifying to read from file the parameters that characterize the ParaFlow algorithm. These parameters are the solvers to use for operators G and F, their step sizes and the number of iterations N_G and N_F that they have to perform (instead of the time interval length T). The class interface is the following:

```
template<unsigned int dim>
class ParaFlowRoot: public ParallelRootBase<dim>
{
public:
  using VT = VectorTypes;
  ParaFlowRoot(const std::string& linear_system_filename, const std::
    string& ParaFlow_params_filename);
  ParaFlowRoot(const double gamma_val, const double nu_val, const
    unsigned int N_grid, const std::string& ParaFlow_params_filename);
  void run() override;

private:
  void get_numerical_method_params(const std::string& filename) override
    ;
};
```

Listing 2.16: class ParaFlowRoot<unsigned int dim>

while the code for the `run()` method is:

```
template<unsigned int dim>
void ParaFlowRoot<dim>::run()
{
  unsigned int total_n_it=0;
  std::cout << "Initialization" << std::endl;
  for (unsigned int i=0; i<this->n_mpi_processes; i++)
  {
    this->gf_G->run(this->n_iter_G);
    Vector<double> y = this->gf_G->get_y_vec();
    Vector<double> u = this->gf_G->get_u_vec();
    this->G_new_vectors[i][0] = y;
    this->G_new_vectors[i][1] = u;
    this->new_yu_vectors[i][0] = y;
    this->new_yu_vectors[i][1] = u;
```

```
15      this->gf_G->output_iteration_results();
16    }
17    this->G_old_vectors = this->G_new_vectors;
18    total_n_it += this->n_iter_G*this->n_mpi_processes;
19
20    unsigned int it=0;
21    while(!this->converged)
22    {
23        std::cout << "Iteration n: " << it+1 << std::endl;
24        //Root send initial conditions to all the other ranks
25        for (unsigned int rank=1; rank<this->n_mpi_processes; rank++)
26        {
27          Utilities::MPI::isend(this->new_yu_vectors[rank-1], this->
     mpi_communicator, rank);
28        }
29        //Root propagates its F operator
30        this->gf_F->run(this->n_iter_F);
31        this->F_vectors[0][0] = this->gf_F->get_y_vec();
32        this->F_vectors[0][1] = this->gf_F->get_u_vec();
33        this->converged_vec[0] = std::get<0>(this->gf_F->convergence_info
     ());
34        //Root receives F operator results and convergence results from
     all the other ranks
35        for (unsigned int rank=1; rank<this->n_mpi_processes; rank++)
36        {
37          VT::FutureTupleType future = Utilities::MPI::irecv<VT::TupleType
     >(this->mpi_communicator, rank);
38          VT::TupleType fut_tuple = future.get();
39          this->F_vectors[rank] = std::get<0>(fut_tuple);
40          this->converged_vec[rank] = std::get<1>(fut_tuple);
41        }
42        total_n_it += this->n_iter_F;
43
44        //Root computes convergence and sends the result to the other
     ranks
45        this->converged = this->check_convergence();
46        for (unsigned int rank=1; rank<this->n_mpi_processes; rank++)
47        {
48          Utilities::MPI::isend(this->converged, this->mpi_communicator,
     rank);
49        }
50
51        if(!this->converged)
52        {
```

```
53        // Perform correction iteration
54    // Update the value of y and u obtained from root
55    this->new_yu_vectors[0][0] = this->F_vectors[0][0];
56    this->new_yu_vectors[0][1] = this->F_vectors[0][1];
57
58        this->G_old_vectors = this->G_new_vectors;
59        for (unsigned int rank=1; rank<this->n_mpi_processes; rank++)
60        {
61          // Update G
62          this->gf_G->set_initial_vectors(this->new_yu_vectors[rank
    -1][0], this->new_yu_vectors[rank-1][1]);
63          this->gf_G->run(this->n_iter_G);
64          this->G_new_vectors[rank][0] = this->gf_G->get_y_vec();
65          this->G_new_vectors[rank][1] = this->gf_G->get_u_vec();
66
67          //Update vectors y and u with correction iteration
68          Vector<double> y_temp(this->G_new_vectors[rank][0]);
69          y_temp -= this->G_old_vectors[rank][0];
70          y_temp += this->F_vectors[rank][0];
71          Vector<double> u_temp(this->G_new_vectors[rank][1]);
72          u_temp -= this->G_old_vectors[rank][1];
73          u_temp += this->F_vectors[rank][1];
74          this->new_yu_vectors[rank][0] = y_temp;
75          this->new_yu_vectors[rank][1] = u_temp;
76        }
77        total_n_it += this->n_iter_G*(this->n_mpi_processes-1);
78
79        // Output results of this iteration process
80        for (unsigned int rank=0; rank<this->n_mpi_processes; rank++)
81        {
82          this->gf_G->set_initial_vectors(this->new_yu_vectors[rank][0],
     this->new_yu_vectors[rank][1]);
83          this->gf_G->output_iteration_results();
84        }
85      }
86      if(!this->converged)
87      {
88            //Interval shift by imposing U_0 = U_N
89        this->new_yu_vectors[0] = this->new_yu_vectors[this->
    n_mpi_processes-1];
90        this->gf_G->set_initial_vectors(this->new_yu_vectors[0][0], this
    ->new_yu_vectors[0][1]);
91        this->gf_F->set_initial_vectors(this->new_yu_vectors[0][0], this
    ->new_yu_vectors[0][1]);
```

```
92
93          for (unsigned int i=0; i<this->n_mpi_processes; i++)
94          {
95        this->gf_G->run(this->n_iter_G);
96        Vector<double> y = this->gf_G->get_y_vec();
97        Vector<double> u = this->gf_G->get_u_vec();
98        this->G_new_vectors[i][0] = y;
99        this->G_new_vectors[i][1] = u;
100       this->new_yu_vectors[i][0] = y;
101       this->new_yu_vectors[i][1] = u;
102         }
103         total_n_it += this->n_iter_G*this->n_mpi_processes;
104       }
105       it++;
106   }
107 ...
108 }
```

Listing 2.17: ParaFlowRoot<dim>::run()

The main difference between the method above and the `ParaRealRoot<dim>::run()` method is the fact that in the ParaFlow version, the interval on which the operators are propagated is shifted at each iteration without having to check if the ParaReal convergence criterion (1.36) is satisfied. Therefore, the results of the correction iteration are only used to compute the initial condition of the next interval, while the initial values provided to the F operators are determined by operator G through the initialization propagation on each new interval.

ParaFlowRankN

The `ParaFlowRankN<unsigned int dim>` class inherits from the `ParallelRankNBase<dim>` class and overrides the `get_numerical_method_params` method by specifying to read from file the parameters that characterize the ParaFlow algorithm for the *RankN* processors which are just the solver to use for operator F, its step size and the number of iterations that it has to perform. The class interface is the following:

```
1 template<unsigned int dim>
2 class ParaFlowRankN: public ParallelRankNBase<dim>
3 {
4 public:
5   using VT = VectorTypes;
6   ParaFlowRankN(const std::string& linear_system_filename, const std::
     string& ParaFlow_params_filename);
```

```
  ParaFlowRankN(const double gamma_val, const double nu_val, const
   unsigned int N_grid, const std::string& ParaFlow_params_filename);

private:
  void get_numerical_method_params(const std::string& filename) override
   ;

};
```

Listing 2.18: class ParaFlowRankN<unsigned int dim>

2.3.6. ParaFlowS

The `ParaFlowS<unsigned int dim>` class inherits from the `NumericalAlgorithmBase<dim>` class and overrides both the `run()` method by implementing Algorithm (1.4) and the `get_numerical_method_params` method. We recall that ParaFlowS is based on the ParaFlow algorithm but no longer requires to perform parallel computations. Therefore, the only difference between the parameters of this algorithm and the ones of ParaFlow is the additional parameter N, which no longer represents the number of processors but it has to be specified in the parameter file. The class interface is the following:

```
template<unsigned int dim>
class ParaFlowS: public NumericalAlgorithmBase<dim>
{
public:
  using VT = VectorTypes;
  ParaFlowS(const std::string& linear_system_filename, const std::string
   & ParaFlowS_params_filename);
  ParaFlowS(const double gamma_val, const double nu_val, const unsigned
   int N_grid, const std::string& ParaFlowS_params_filename);
  void run() override;

private:
  void get_numerical_method_params(const std::string& filename) override
   ;

  bool                                converged;

  std::unique_ptr<DescentStepBase<dim>> gf_G;
  std::unique_ptr<DescentStepBase<dim>> gf_F;
  GFStepType                          MethodOperatorG;
  GFStepType                          MethodOperatorF;

  unsigned int                        N;
```

```
  double                          step_size_G;
  double                          step_size_F;
  unsigned int                    n_iter_G;
  unsigned int                    n_iter_F;

  VT::ArrayType                     F_vectors;
  VT::VectorArrayType               G_old_vectors;
  VT::VectorArrayType               G_new_vectors;
  VT::VectorArrayType               new_yu_vectors;

  std::vector<double>               J_eval;
}
```

Listing 2.19: class ParaFlowS<unsigned int dim>

In this class we introduce the vector `J_eval`, which stores at each iteration the evaluations of the cost functional for the values obtained by performing the correction iteration. The value achieving the minimum of the cost functional is used to initialize operator F at the next iteration. The implementation of the `run()` method is the following:

```
template<unsigned int dim>
void ParaFlowS<dim>::run()
{
  unsigned int total_n_it=0;
  std::cout << "Initialization" << std::endl;
  for (unsigned int i=0; i<N; i++)
  {
    gf_G->run(n_iter_G);
    J_eval[i] = gf_G->evaluate_J();
    Vector<double> y = gf_G->get_y_vec();
    Vector<double> u = gf_G->get_u_vec();
    G_new_vectors[i][0] = y;
    G_new_vectors[i][1] = u;
    new_yu_vectors[i][0] = y;
    new_yu_vectors[i][1] = u;
    gf_G->output_iteration_results();
  }
  G_old_vectors = G_new_vectors;
  gf_G->output_results_vectors();
  total_n_it += n_iter_G*N;

  unsigned int idx_minJ;
  unsigned int local_convergence_iter(n_iter_F);

  unsigned int it=0;
```

```
26  while(!converged)
27  {
28      std::cout << "Iteration n: " << it+1 << std::endl;
29      //We find the index of the values that minimize the cost
     functional
30      idx_minJ = std::min_element(J_eval.begin(), J_eval.end()) - J_eval
     .begin();
31      gf_F->set_initial_vectors(new_yu_vectors[idx_minJ][0],
     new_yu_vectors[idx_minJ][1]);
32      gf_F->run(n_iter_F);
33      gf_F->output_iteration_results();
34      total_n_it += n_iter_F;
35
36      J_eval[0] = gf_F->evaluate_J();
37      F_vectors[0] = gf_F->get_y_vec();
38      F_vectors[1] = gf_F->get_u_vec();
39      converged = std::get<0>(gf_F->convergence_info());
40
41      if(converged)
42      {
43        local_convergence_iter = std::get<1>(gf_F->convergence_info());
44        total_n_it = total_n_it - n_iter_F + local_convergence_iter;
45      }
46      else
47      {
48        // Update the value of y and u of i=0
49        new_yu_vectors[0][0] = F_vectors[0];
50        new_yu_vectors[0][1] = F_vectors[1];
51
52        G_old_vectors = G_new_vectors;
53        for (unsigned int i=1; i<N; i++)
54        {
55          // Update G
56          gf_G->set_initial_vectors(new_yu_vectors[i-1][0],
     new_yu_vectors[i-1][1]);
57          gf_G->run(n_iter_G);
58          G_new_vectors[i][0] = gf_G->get_y_vec();
59          G_new_vectors[i][1] = gf_G->get_u_vec();
60
61          // Perform correction iteration
62          Vector<double> y_temp(G_new_vectors[i][0]);
63          y_temp -= G_old_vectors[i][0];
64          y_temp += F_vectors[0];
65          Vector<double> u_temp(G_new_vectors[i][1]);
```

```
66            u_temp -= G_old_vectors[i][1];
67            u_temp += F_vectors[1];
68            new_yu_vectors[i][0] = y_temp;
69            new_yu_vectors[i][1] = u_temp;
70
71            gf_G->set_initial_vectors(new_yu_vectors[i][0], new_yu_vectors
    [i][1]);
72            gf_G->output_iteration_results();
73            J_eval[i] = gf_G->evaluate_J();
74
75          }
76          total_n_it += n_iter_G*(N-1);
77        }
78        it++;
79    }
80 //...
81 }
```

Listing 2.20: ParaFlowS<dim>::run()

2.4. Dependencies, Installation and Execution

Before installing the library, it is necessary to check that the dependencies that guarantee its correct functioning are met. First of all, the library can be used only on a linux-based machine with CMake $\geq$ 3.25.1 and GNU bash $\geq$ 5.2.15. Moreover, since the library essentially relies on the *deal.II*[1] library, it's necessary to guarantee that it is installed with a version $\geq$ 9.5.0, which can be found in the `mk`[2] module, `2024.0` version. Finally, to view and analyze the numerical solutions it is possible to use VisIt[3] $\geq$ 3.3.3.

Once all dependencies are met, this library can be downloaded from GitHub using the following bash command:

```
git clone https://github.com/fartiolii/pacs-project.git
```

Since it is only a header library, the previous command already provides the library installation.

The `include/` folder contains the header files for the core and numerical methods template classes explained above, while in the `Examples/` folder we can find the `Test 1/`, `Test`

[1] https://www.dealii.org/
[2] https://github.com/pcafrica/mk/releases
[3] https://visit-dav.github.io/visit-website

`2/` and `Grid Refinement/` subfolders, containing the files to compile and run the tests explained in detail in chapter 3.

2.4.1. Test 1 and Test 2

The folders `Test 1/` and `Test 2/` contain a `config_params.json` file where the parameters characterising the optimal control problem (γ and ν) are defined, along with the number of grid refinements to apply to the mesh of the discretized domain Ω. Moreover, for each numerical method described above, we provide a folder containing the respective `.cc` file to run the test, a `CMakeLists.txt` file and another `.json` file to specify the parameters characterising the numerical method.
The steps to undertake to compile and run a test for each algorithm from its corresponding folder are the following:

```
cmake -DDEAL_II_DIR=/path/to/dealii .
```

```
make release
```

```
make run
```

The algorithms that are enabled to perform parallel computations, which are Parareal and ParaFlow, can be run in parallel on multiple processors respectively with the following commands:

```
mpirun -np number_of_processors ./test_ParaReal
```

and

```
mpirun -np number_of_processors ./test_ParaFlow
```

Finally, execution files can be removed with the command

```
make clean
```

while to clean the directory from all output files generated you can run

```
make distclean
```

2.4.2. Grid Refinement

The `Grid Refinement/` folder contains the files to compile and run each one of the four algorithms described above, to solve the optimal control problem characterized by the same parameters γ and ν of Test 2, while varying the refinement of the mesh used to discretize the domain Ω. In particular, the mesh refinements that have been tested are the following:

Mesh refinements

Number of mesh refinements	**2**	**3**	**4**	**5**	**6**
Number of nodes N_h	25	81	289	1089	4225

As before, you can find a folder for each numerical method containing a `.cc` file to run the test, a `CMakeLists.txt` file and a `.json` file to specify the parameters of the method. Once again, parameters γ and ν are read from the `config_params.json` file, common to all numerical methods, while instead the number of grid refinements is directly specified in the `.cc` file. Finally, the steps to compile and run a test for each algorithm are the ones reported in section 2.4.1.

3 | Numerical Results

In this chapter, we report the results obtained by applying the numerical methods described in chapter 1 to solve the discrete optimal control problem (1.5)-(1.6) with different values of parameters γ and ν. We focus in particular on the number of iterations that each numerical method requires to converge to the solution.

All tests have been run considering $\varphi(y) = \exp\{y\}$ and $\Omega = [-1, 1]^2$ which has been discretized using Q_1 elements and $N_h = 1089$ nodes. Moreover, the initial value vectors $\mathbf{y}_h^{(0)}$ and $\mathbf{u}_h^{(0)}$ have been chosen with all elements set to 0, while the desired state has been set to $\mathbf{y}_d = 1.0$ in Ω.
For what concerns the values of step size α used both for the Explicit Euler and the Adam update rules, they have been chosen as the ones that led to the fastest convergence towards the solution. In addition, for the Adam update rule we have chosen $\beta_1 = 0.95$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$, while the value of ϵ for the convergence criteria of the Parareal algorithm has been set to 10^{-3}. For the stopping criterion on the L^2-norms of ϕ_1 and ϕ_2, instead, we considered $tol_1 = 5 * 10^{-5}$ and $tol_2 = 1 * 10^{-4}$, which resulted as the best compromise between the achieved minimum value of the cost functional J and the need to avoid infinite oscillations of the value of J around the minimum.

Lastly, we point out that the total number of iterations computed with the Parareal and ParaFlow algorithms is intended as the total number of iterations computed in series. Indeed, the number of computations performed in parallel by the operators F is counted just once for each Parareal or ParaFlow iteration.

3.1. Test 1

This test considers the discrete optimal control problem with the following parameters:

$$\begin{cases} \nu = 0.05 \\ \gamma = 0.8 \end{cases} \tag{3.1}$$

First of all, the Gradient Flow algorithm has been tested by using both update rules. The number of iterations required to obtain the solution vectors ($\mathbf{y}_h$ and $\mathbf{u}_h$) and the cost functional evaluated at the optimal $\mathbf{y}_h$ and $\mathbf{u}_h$ are reported in the table

Gradient Flow

Update rule	**Explicit Euler**	**Adam**
Minimum J achieved	1.665669	1.665679
Total number of Iterations	**11785**	**259**
Step size α	1.5	0.2

.

In second place, we have tested the Parareal algorithm, combining different choices of update rules for operators F and G, and optimizing accordingly their step sizes and the time interval length T. The number of iterations reported in the table below are the number of iterations of the update rule computed in series by executing the algorithm on 4 processors.

Parareal

Operator G	**Explicit Euler**	**Explicit Euler**	**Adam**
Operator F	**Explicit Euler**	**Adam**	**Adam**
Minimum J achieved	1.665669	1.665680	1.665681
Total number of Iterations	**27340**	**455**	**495**
Time interval length T	100	75	50
Operator F step size	1.0	0.1	0.1
Operator G step size	1.5	1.5	1.0
Number of Processors N	4	4	4

In third place, we have tested the ParaFlow algorithm, by using the Explicit Euler or the Adam update rules for both operators F and G, and optimizing accordingly their step sizes and the number of update rule iterations to perform. Once again, the number of iterations reported in the table below are the number of iterations of the update rule computed in series by executing the algorithm on 4 processors.

ParaFlow

Operator G	**Explicit Euler**	**Adam**
Operator F	**Explicit Euler**	**Adam**
Minimum J achieved	1.665669	1.665720
Total number of Iterations	**3966**	**500**
Operator F step size	1.5	0.1
Operator G step size	1.5	1.0
Number of Iterations of F	20	50
Number of Iterations of G	1	12
Number of Processors N	4	4

Lastly, we have tested the ParaFlowS algorithm, combining different choices of update rules for operators F and G and optimizing accordingly their step sizes, the number of update rule iterations to perform and the additional parameter N.

ParaFlowS

Operator G	**Explicit Euler**	**Explicit Euler**	**Adam**
Operator F	**Explicit Euler**	**Adam**	**Adam**
Minimum J achieved	1.665654	1.665688	1.665691
Total number of Iterations	**559**	**194**	**138**
Operator F step size	1.5	0.2	0.1
Operator G step size	1.5	1.5	1.0
Number of Iterations of F	15	100	100
Number of Iterations of G	1	1	1
N	8	2	2

3.2. Test 2

This test, instead, considers the discrete optimal control problem with the following parameters:

$$\begin{cases} \nu = 0.01 \\ \gamma = 1.0 \end{cases} \tag{3.2}$$

Once again, the Gradient Flow algorithm has been tested by using both update rules. The number of iterations required to obtain the solution vectors ($\mathbf{y}_h$ and $\mathbf{u}_h$) and the cost functional evaluated at the optimal $\mathbf{y}_h$ and $\mathbf{u}_h$ are reported in the table

Gradient Flow

	Explicit Euler	**Adam**
Minimum J achieved	1.076377	1.076360
Total number of Iterations	**45202**	**242**
Step size α	1.5	0.5

.

The Parareal algorithm, instead, has been tested combining different choices of update rules for operators F and G, and optimizing accordingly their step sizes and the time interval length T. The number of iterations reported in the table below are the number of iterations of the update rule computed in series by executing the algorithm on 4 processors.

Parareal

Operator G	**Explicit Euler**	**Explicit Euler**	**Adam**
Operator F	**Explicit Euler**	**Adam**	**Adam**
Minimum J achieved	1.076377	1.076266	1.076267
Total number of Iterations	**99034**	**455**	**495**
Time interval length T	100	50	50
Operator F step size	1.0	0.1	0.1
Operator G step size	1.5	1.5	1.0
Number of Processors N	4	4	4

Next, the ParaFlow algorithm has been tested by using the Explicit Euler or the Adam update rules for both operators F and G, and optimizing accordingly their step sizes and the number of update rule iterations to perform. Once again, the number of iterations reported in the table below are the number of iterations of the update rule computed in series by executing the algorithm on 4 processors.

ParaFlow

Operator G	**Explicit Euler**	**Adam**
Operator F	**Explicit Euler**	**Adam**
Minimum J achieved	1.076377	1.076405
Total number of Iterations	**15252**	**634**
Operator F step size	1.5	0.1
Operator G step size	1.5	1.0
Number of Iterations of F	20	50
Number of Iterations of G	1	12
Number of Processors N	4	4

Lastly, we have tested the ParaFlowS algorithm, combining different choices of update rules for operators F and G and optimizing accordingly their step sizes, the number of update rule iterations to perform and the additional parameter N.

ParaFlowS

Operator G	**Explicit Euler**	**Explicit Euler**	**Adam**
Operator F	**Explicit Euler**	**Adam**	**Adam**
Minimum J achieved	1.076320	1.076418	1.076447
Total number of Iterations	**427**	**143**	**141**
Operator F step size	1.5	0.2	0.2
Operator G step size	1.5	1.5	0.1
Number of Iterations of F	15	50	50
Number of Iterations of G	1	1	1
N	8	2	2

3.3. Results Discussion

We have used the four numerical methods presented in Chapter 1 to solve the discrete optimal control problem with two different sets of parameters γ and ν. For each numerical method, we have tested the *Explicit Euler* update rule, the *Adam* update rule and a combination of the two, where possible. The results of the GradientFlow algorithm are used as a benchmark to compare the performance, on Tests 1 and 2, of the other three

methods with respect to both update rules.
First of all, considering the *Explicit Euler* update rule for both operators G and F, we can observe that the number of iterations required by the Parareal algorithm is more than two times greater than the one required by the GradientFlow algorithm for both tests. Indeed, as highlighted at the end of section 1.5, the Parareal algorithm does not perform well on this type of problems. If instead we consider the performance of the ParaFlow algorithm with the same solvers G and F as before, we can notice that the number of iterations required has been reduced for both tests by almost three times when compared to the GradientFlow algorithm. Indeed, even if the ParaFlow is defined similarly to the Parareal, continuously updating the initial values $\mathbf{U}_0^{k+1}$ at each iteration with the last value obtained by the correction iteration, has proved to considerably speed up the convergence process when using the *Explicit Euler* update rule.
Lastly, we can observe that the ParaFlowS algorithm, when using the *Explicit Euler* update rule for G and F, performs significantly better than all the other algorithms, with the additional advantage that it does not require to perform parallel computations. In particular, if we compare the results of the GradientFlow and the ParaFlowS algorithms, we can see that the ParaFlowS reduces the number of required iterations of the GradientFlow by a factor of 20 in the first test and by a factor of 100 in the second.

Considering the *Adam* update rule, instead, we can notice that the GradientFlow algorithm requires a really low number of iterations for both tests, when compared to the ones required by the same algorithm with the *Explicit Euler* update rule. Indeed, it achieves a very fast convergence to the solution by leveraging the first and second moments of the descent directions. As before, both by using the *Adam* update rule for both operators G and F, or the *Explicit Euler* one for G and the *Adam* one for F, the number of iterations required by the Parareal algorithm increases with respect to the GradientFlow one. In this case however, the ParaFlow algorithm is not able to improve the performance of the Parareal algorithm. Indeed, it actually performs worse in both tests and it converges only if the number of iterations of operator G is greater than 10. This behavior can be explained by the fact that the *Adam* update rule strongly relies on past descent values through the momentum terms. Continuously shifting the interval on which the operators propagate the initial values, as the ParaFlow algorithm does, is not beneficial to this update rule type especially in the initial phase, when the successive approximations of the solution vectors are still far from convergence values.
Lastly, we can observe that both by using the *Adam* update rule for both operators G and F, or the *Explicit Euler* one for G and the *Adam* one for F, the number of iterations required by the ParaFlowS algorithm is lower, for both tests, when compared to all other

algorithms, and therefore even in the GradientFlow case.
Moreover, to conclude, if we compare the behavior of the *Explicit Euler* and the *Adam* update rules, especially when chosen as operators F for the ParaFlowS algorithm, we can notice that the *Explicit Euler* one benefits from frequent correction iterations, which led to the choice of a small number of iterations for F and a large N, while instead the *Adam* update rule performs better with a large number of subsequent iterations and benefits from the correction step closer to convergence, which threfore led to the choice of a large number of iterations for F and a small N.

3.4. Grid Refinement Tests

In this section we report the results obtained by testing the four numerical methods to solve the optimal control problem characterized by parameters $\gamma = 1.0$ and $\nu = 0.01$ of Test 2, by varying the refinement of the mesh used to discretize the domain Ω, as described in section 2.4.2. Without loss of generality, the results reported hereinafter take into account only the Explicit Euler update rule for the Gradient Flow algorithm and for the operators G and F of the other three algorithms.

Therefore, by running the Gradient Flow algorithm with Explicit Euler update rule and step size $\alpha = 1.5$ we have obtained the following results:

Gradient Flow

Number of mesh refinements	**2**	**3**	**4**	**5**	**6**
Number of nodes N_h	25	81	289	1089	4225
Total number of iterations	1949	4889	14421	45202	138091

Furthermore, by running the Parareal algorithm in parallel on 4 processors with the Explicit Euler update rule for both operators G and F, step sizes respectively equal to $\alpha = 1.5$ and $\alpha = 1.0$ and $T = 100$ we have obtained the following results:

Parareal

Number of mesh refinements	**2**	**3**	**4**	**5**
Number of nodes N_h	25	81	289	1089
Total number of iterations	5303	11915	33331	99034

Instead, by running the ParaFlow algorithm in parallel on 4 processors with the Explicit Euler update rule for both operators G and F, a number of iterations per operator respectively equal to $N_G = 1$ and $N_F = 20$ and step sizes both equal to $\alpha = 1.5$ we have obtained the following results:

ParaFlow

Number of mesh refinements	**2**	**3**	**4**	**5**
Number of nodes N_h	25	81	289	1089
Total number of iterations	672	1644	4857	15252

Lastly, by running the ParaFlowS algorithm with the Explicit Euler update rule for both operators G and F, a number of iterations per operator respectively equal to $N_G = 1$ and $N_F = 15$, both step sizes equal to $\alpha = 1.5$ and $N = 8$, we have obtained the following results:

ParaFlowS

Number of mesh refinements	**2**	**3**	**4**	**5**	**6**
Number of nodes N_h	25	81	289	1089	4225
Total number of iterations	346	581	669	427	955

For all algorithms except ParaFlowS, we can observe that the number of iterations required to converge to the solution increases substantially as the number of nodes increases. For the ParaFlowS algorithm, instead, the number of iterations required by using a mesh with 1089 nodes is lower if compared to the one required by using meshes with 81 or 289 nodes. This behavior can be explained by the fact that the parameters used for ParaFlowS have been optimized based on the problem that employs the mesh with 1089 nodes. It is therefore possible that, by using different parameters, the number of iterations required by using the mesh with 81 and 289 nodes can be reduced.

3.5. Scalability Tests

Scalability tests have been performed for the Parareal and ParaFlow algorithms, which leverage parallel computations, by varying the number of processors on which they are executed. As a metric to compare the computational times, in absence of hardware with more than 4 processors, we have reported the number of iterations of update rule in series that each algorithm requires to compute in the different cases.

3.5.1. Strong Scalability

Strong scalability has been tested by executing the algorithms with a different number of processors, while using a mesh to discretize Ω with the same number of nodes $N_h = 1089$.

By considering the optimal control parameters $\gamma = 1.0$ and $\nu = 0.01$ of Test 2, $T = 100$ and the Explicit Euler update rule for both operators G and F with step size respectively equal to $\alpha = 1.5$ and $\alpha = 1.0$, the obtained results with the Parareal algorithm are the following:

Parareal

Number of processors	**1**	**2**	**4**	**8**	**16**	**32**
Total number of Iterations	113214	101553	99034	93336	91916	91401

Instead, the results obtained with the ParaFlow algorithm, considering the same optimal control parameters $\gamma = 1.0$ and $\nu = 0.01$ of Test 2 and the Explicit Euler update rule for both operators G and F, with a number of iterations respectively equal to $N_G = 1$ and $N_F = 20$ and both step sizes equal to $\alpha = 1.5$ are reported as follows:

ParaFlow

Number of processors	**1**	**2**	**4**	**8**	**16**	**32**
Total number of Iterations	47481	25989	15252	9898	7176	5862

We can observe that, although the number of iterations required to converge to the solution decreases for both algorithms as the number of processors increases, this reduction is very low for Parareal, whereas it is more significant for ParaFlow, though not being linear.

3.5.2. Weak Scalability

Weak scalability has been tested by executing the algorithms with a different number of processors, while also varying the number of nodes N_h of the mesh used to discretize the domain Ω. More specifically, we have reported the results obtained by increasing by a factor of 4 the number of processors and by a factor of $\sim$4 the number of nodes N_h. The limited number of tests executed in this case is due to the limited flexibility in the choice of N_h, constrained by the number of times that the mesh is refined.
Moreover, when evaluating the results of this test, we must also consider that by increasing the number of nodes N_h, the computational time of a single update rule iteration is higher, due to an increase in the dimension of the linear system that is solved iteratively with the Minimal Residual Method.

By considering the optimal control parameters $\gamma = 1.0$ and $\nu = 0.01$ of Test 2, $T = 100$ and the Explicit Euler update rule for both operators G and F with step size respectively equal to $\alpha = 1.5$ and $\alpha = 1.0$, the obtained results with the Parareal algorithm are the following:

Parareal

Number of processors	**2**	**8**	**32**
Number of mesh refinements	4	5	6
Number of nodes N_h	289	1089	4225
Total number of Iterations	33930	93336	263450

Instead, the results obtained with the ParaFlow algorithm, considering the same optimal control parameters $\gamma = 1.0$ and $\nu = 0.01$ of Test 2 and the Explicit Euler update rule for both operators G and F, with a number of iterations respectively equal to $N_G = 1$ and $N_F = 20$ and both step sizes equal to $\alpha = 1.5$ are reported as follows:

ParaFlow

Number of processors	**2**	**8**	**32**
Number of mesh refinements	4	5	6
Number of nodes N_h	289	1089	4225
Total number of Iterations	8302	9898	17897

We can observe that Parareal is not weakly scalable, as the number of iterations required increases substantially with the number of processors and of nodes N_h. The number of iterations required by ParaFlow, instead, has a slight increase in the first case and a greater one in the second. Therefore, also in this case, we cannot affirm that the algorithm is weakly scalable.

References

[1] K. Tanabe, A geometric method in nonlinear programming, Journal of Optimization Theory and Applications 2 (30), 181-185 (1980)
[2] J.-L. Lions, Y. Maday, and G. Turinici, A "parareal" in time discretization of PDE's, C. R. Acad. Sci. Paris S´er. I Math., 332 (2001), pp. 661–668.
[3] M. J. Gander and S. Vandewalle, Analysis of the parareal time-parallel time-integration method. SIAM Journal on Scientific Computing, 29 (2):556–578, 2007
[4] D. Kingma and J. Ba, Adam: A Method for Stochastic Optimization. Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)
[5] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin and D. Wells, The deal.II finite element library: design, features, and insights Computers & Mathematics with Applications, vol. 81, pages 407-422, 2021.
[6] M. Ulbrich, Optimization methods in Banach spaces, in Optimization with PDE Constraints, ed. by Y. Borenstein, A. Moraglio, pp. 97–156 (Springer, Berlin, 2009)
[7] Borzì, A. and Schulz, V. (2011) Computational Optimization of Systems Governed by Partial Differential Equations. SIAM, Philadelphia.