UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques
i Informàtica

DOBLE GRAU DE MATEMÀTIQUES I ENGINYERIA INFORMÀTICA

Treball final de grau

Numerical Techniques for Robot Path Planning: Artificial Potential Fields and Proper Generalized Decomposition

Autor: Pau Hernando Marmol

Director:	**Dr. Eloi Puertas**
	Dr. Arturo Vieiro
Realitzat a:	**Departament de Matemàtiques i Informàtica**
Barcelona,	**10 de juny de 2024**

Contents

Abstract

One of the most important tasks in the mobile robot navigation field is the planning of a collision-free path from a starting point to a target point. This project introduce the concept of Artificial Potential Field (APF) as a real time global path planner method and how it is modelled using the Poisson equation. To solve it, a recently developed numerical technique called Proper Generalized Decomposition (PGD) is considered, since it makes the resolution of the Poisson equation feasible for real-time calculations.

To illustrate the properties of those methods, a simulation with a virtual robot on a virtual world has been produced. The Construct AI, a free online platform, has enabled us to develop the necessary code for this, based on the Robot Operating System (ROS) framework. This tools are used on the Robotics subject at *Universitat de Barcelona*, and that has proved to be very useful, because I had at my disposal some documentation and repositories that enabled me to avoid having to do all the settings from scratch.

This work tries to be an accessible introduction to this topics, and it can serve as a basis for future multiple extensions, as we will comment at the end. The advantages and the projection of this approach inside the path planning area make it a candidate for become the future of robot navigation.

Resum

Una de les tasques més importants en el món de la navegació de robots mòbils és la planificació d'un camí lliure de col·lisions des d'un punt inicial fins a un punt final. En aquest projecte s'introdueix el concepte d' *Artificial Potential Field* (APF) com a un planificador de camins global i com aquest és modelat utilitzant l'equació de Poisson. Per resoldre-la, s'ha considerat un mètode numèric desenvolupat recentment anomenat *Proper Generalized Decomposition* (PGD), doncs fa que la resolució de l'equació de Poisson sigui factible de calcular a temps real.

Per il·lustrar les propietats d'aquests mètodes, s'ha realitzat una simulació amb un robot i un món virtual. *The Construct AI*, una plataforma en línia gratuïta, ens ha permès desenvolupar el codi necessari per aconseguir-ho. Aquesta és l'eina que s'utilitza a la classe de robòtica de la Universitat de Barcelona, fet que ha resultat extremadament útil, doncs he tingut a la meva lliure disposició certa documentació i repositoris que m'han permès no haver de configurar tot des de zero.

Aquest treball pretén ser una introducció accessible d'aquests tòpics, i pot servir com a base per a futures extensions del projecte, tal com es comenta al final. Els avantatges i la projecció d'aquesta nova perspectiva dins del camp de la planificació de camins la converteixen en una clara candidata per esdevenir el futur de la navegació de robots.

Introduction

One of the core challenges in robotics is to plan a path free of collisions from an initial to a target position without collisions. There are several ways to approach this problem: sampling-based planners, interval-based planners, potential-field-based techniques, [10, 12] etc.

In this project, we will focus on the Artificial Potential Field method (APF). This method was first presented by O. Khatib in 1985 in *The International Journal of Robotics Research* [13]. The APF generates an artificial potential field that guides the trajectory of the robot. The target position originates an attractive force which makes the mobile robot move towards it and the obstacles generate repulsive forces to avoid them. Its computation is fast, making it a very good choice for real-time applications. However, repulsive fields generally create local minima and the robot may not reach the goal even if a solution exist. For avoiding that, harmonic functions [14, 5], have been used to generate artificial potential fields. Harmonic functions have some valuable properties [2, 22], including the min-max principle, which prevent the appearance of deadlocks. We will see that applying some potential flow theory, one can adapt a Poisson equation to have the harmonic properties while defining a proper source term. In spite of this, there have not been many attempts of using harmonic functions for path planning, since this functions cannot be computed in closed form and the the computational burden of discrete approximations is really high.

But, a short time ago, an original technique called proper generalized decomposition (PGD) was developed to give an approximation of the solutions of nonlinear convex variational problems [8, 1]. One of the main advantages of this method is the capability of transform high dimensional problems into a series of decoupled one-dimensional problems. This allows us to work on high dimensional spaces, being able to compute all the possibles solutions and parameters, such as the combinations of goals and targets.

The objective of this project is use the PGD algorithm to approximate a solution of the Poisson equation that will give us a path from a source point to a target point. To do so, this project is divided into four main goals:

- Understand what is an Artificial potential field, its main problem and how to avoid it. For doing that, we introduce the concept of Harmonic function, boundary conditions and the weak formulation of the Poisson equation.
- Explain how does PGD works, and how its separated representation is constructed, step by step.
- Relate the PGD methodology with the path planning scope.
- Build the robot application to generate a simulation and test it.

Each goal, is covered by a chapter, and one can also an appendix. The dissertation is organised as follows:

- **Chapter 2:** Defines the Artificial Potential Field, as well as the harmonic functions, which are the solution of the main disadvantage of the APF. Other important concepts are explained here, such as boundary conditions, potential flow theory and how that lead to the Poisson equation. In this chapter, also is defined the weak formulation as well as how they should be constructed in order for everything to work properly.
- **Chapter 3:** Defines the Poisson equation and shows how we can approximate a solution through the PGD algorithm. It's described for Dirichlet and Neumann boundary conditions.
- **Chapter 4:** Applies the PGD for robot path planning, defining a source term and computing the PGD-Vademecum. It also shows the approach of this project, which is slightly different from this one. It also includes the plots of the numerical results.
- **Chapter 5:** Defines the framework used for building the robot application, as well as its Filesystem and Computational Graph. Details the steps followed for making the robot work as expected and illustrates it.
- **Appendix:** Contain all the code of the various PGD implementations, so it can be referenced throughout the work to facilitate understanding.

All the code developed in Chapter 4 and Chapter 5 can be found here[1], as well as the *.csv* files used for plotting purposes.

[1] `https://github.com/phernama21/tfg`

Chapter 1

Path planners: The Artificial Potential Field

1.1 Artificial potential field

The Artificial Potential Field, first presented by O. Khatib [13], is a method to navigate the robot from the source to the goal following the flow that defines a suitable constructed potential field.

Definition 1.1. *A* gradient system *or* potential field *[17] on an open set* $\Omega \subset \mathbb{R}^n$ *is a system of differential equations of the form*

$$\dot{q} = -\nabla U(q), \quad q \in \Omega,$$

where $U : \Omega \to \mathbb{R}$ *is a* $\mathcal{C}^2(\Omega)$ *potential function and*

$$\nabla U = \left(\frac{\partial U}{\partial x_1}, ..., \frac{\partial U}{\partial x_n}\right)$$

is the gradient vector field, $\nabla U : \Omega \to \mathbb{R}^n$*, of* U*.*

This approach treats the robot, represented as a point q in the configuration space ($C - space$), as a particle under the influence of an artificial potential field U. In order to make the robot be attracted towards its goal configuration, while being repulsed from the obstacles, U is constructed as the sum of two elementary potential functions:

$$U(q) = U_{att}(q) + U_{rep}(q), \tag{1.1}$$

where U_{att} is the attractive potential associated with the goal configuration q_{goal} and U_{rep} is the repulsive potential, associated with the $C - obstacle$ region (see definition in section 1.1.2) . With this conventions, $\vec{F}$ (the force generated by this potential field) is the sum of two vectors:

$$\vec{F}_{att} = -\vec{\nabla} U_{att} \quad and \quad \vec{F}_{rep} = -\vec{\nabla} U_{rep}, \tag{1.2}$$

which are called the attractive and repulsive forces, respectively.

1.1.1 Attractive potential

There exists a lot of ways to model our attractive fields, but we can define the two most common as follows. Let $\rho_{goal}(q)$ be the Euclidean distance $\|q - q_{goal}\|$, and $\xi_q, \xi_c > 0$ scaling factors. The subindex q and c here refer to quadratic and conical cases.

Quadratic potential:

$$U_{att}(q) = \frac{1}{2}\rho^2_{goal}(q)\xi_q$$

Conical potential:

$$U_{att}(q) = \rho_{goal}(q)\xi_c$$

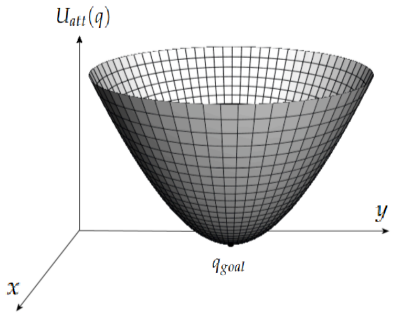

Figure 1.1: Graphical representation of a 2-dimensional quadratic potential field

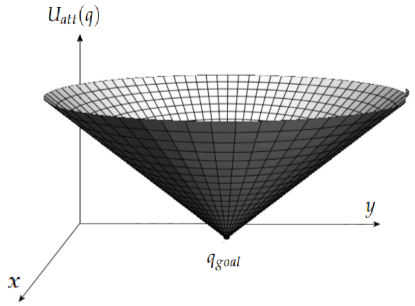

Figure 1.2: Graphical representation of a 2-dimensional conical potential field

In the previous expressions, $\rho_{goal}(q)$ is the Euclidean distance $\|q - q_{goal}\|$. The quadratic potential generates a force $\vec{F}_{att}$ which converges linearly towards 0 when the robot's configuration approaches the goal, but increases indefinitely as ρ_{goal} increases. On the other hand, the conical potential generates a constant amplitude force $\vec{F}_{att}$ for configurations far away from the goal. Therefore, a convenient solution is to combine the two profiles: conical away from q_{goal} and quadratic close to q_{goal}. So we define a distance parameter d. The value of distance parameter depends on the physical characteristics of the robot, since factors such as speed, braking capacity or ground friction, will led us to chose a larger or smaller value. Once we choose d depending on the robot features we consider the potential function

$$U_{att} = \begin{cases} \frac{1}{2}\rho^2_{goal}(q)\xi_q & \text{if} \quad \rho \leq d, \\ \rho_{goal}(q)\xi_c & \text{if} \quad \rho > d. \end{cases} \tag{1.3}$$

For the sake of the continuity of $\vec{F}_{att}$ we need the condition $2\xi_c = \rho_{goal}(d)\xi_q$

The artificial attractive force deriving from U_{att} is

$$\vec{F}_{att} = \begin{cases} -\xi_a(q - q_{goal}) & \text{if } \rho \leq d, \\ -\frac{\xi_b(q-q_{goal})}{\|q-q_{goal}\|} & \text{if } \rho > d. \end{cases} \tag{1.4}$$

1.1.2 Repulsive potential

The main idea underlying the definition of the repulsive potential is to create a potential barrier around the obstacles region which cannot be traversed by the robot, and is modelled as being inversely proportional to the distance from the obstacle. Note that while the attractive potential is applied only by the goal, the repulsive potential is applied by each obstacle. We do name the $C - obstacle$ region as CO and assume that has been partitioned in convex components CO_i. Then, each CO_i defines a repulsive field.

Even though this methodology naturally fades away, we define the repulsive potential of each obstacle U_{rep_i} with finite support. We choose a scalar value $\eta_{CO_i} > 0$ that depends on the condition of the obstacle and the goal point of the robot, and is usually taken to be less than half of the minimum of the distances between the obstacles and the shortest length from the destination to the obstacles. The repulsive potential field is described by;

$$U_{rep_i} = \begin{cases} \frac{1}{2}k_{CO_i}\left(\frac{1}{\eta_i(q)} - \frac{1}{\eta_{CO_i}}\right)^2 & \text{if } \eta_i(q) \leq \eta_{CO_i}, \\ 0 & \text{if } \eta_i(q) > \eta_{CO_i}. \end{cases} \tag{1.5}$$

Here, $k_{CO_i} > 0$ denotes the constant associated with the repulsive potential and $\eta_i(q) = min_{q' \in CO_i}\|q - q'\|$. As for the constants ξ_q, ξ_c in the attractive potential, the constant k_{CO_i} also depends on the robot features.

The resulting repulsive force is

$$\vec{F}_{rep_i} = -\nabla U_{rep_i} = \begin{cases} -\frac{k_{CO_i}}{\eta_i^2(q)}\left(\frac{1}{\eta_i(q)} - \frac{1}{\eta_{CO_i}}\right)\nabla\eta_i(q) & \text{if } \eta_i(q) \leq \eta_{CO_i}, \\ 0 & \text{if } \eta_i(q) > \eta_{CO_i}. \end{cases} \tag{1.6}$$

The cumulative repulsion is the repulsion from all obstacles region CO_i. It can be modelled taking the addition from all p obstacles at a distance of η_{CO_i} or less:

$$U_{rep} = \sum_{i=1}^{p} U_{rep_i} \quad \text{and} \quad \vec{F}_{rep} = \sum_{i=1}^{p} \vec{F}_{rep_i}. \tag{1.7}$$

The complete potential field $U = U_{att} + U_{rep}$ construction is illustrated in figure 1.3.

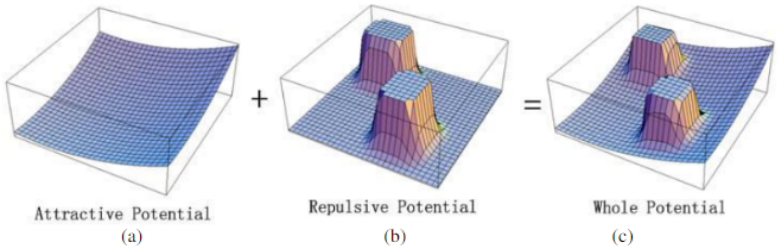

Figure 1.3: (a) The attractive potential without obstacles (b) The repulsive potential representing the obstacles (c) Combination of the two fields to get the resulting whole potential [6]

1.1.3 Problems with potential fields

The APF seems to be a good choice of the algorithm due to its ability to tackle obstacles in real-time, lack of a need of a preknown map, ability to work in partially known environments and ability to work in highly dynamic environments. In addition to its use in robot path planning, it has also been found to have many other functions in areas such as autonomous vehicles navigation [11], multi-robot systems [15], surgical robots [23] ...

The major problem with the APF is its lack of completeness, since the robot can get stuck at a local minima, and even if there exists a path, the algorithm shall not find it. This happens if there exists a region where the attractive and repulsive forces cancel each other out, and therefore the resultant force is 0, as we can see on Figure 1.4.

There exists multiple workarounds for this problem:

- **Best-first algorithm**

  Consists in building a discretized representation of the *C-free space*, i.e the $C - space$ excluding the $C - obstacle$ region, using a grid, and associate to each free cell of the grid a value U_0 at its centroid. Next, build a tree T rooted at q_{start} start point: at each iteration, select the leaf of T with minimum value of U_t and add as children its adjacent cells that are not in T. Planning stops when q_s is reached or no further cells can be added to T. At a local minimum, best-first will fill its basin of attraction until it finds a way out since it will continue expanding the search space from that point, exploring neighboring cells to see if there is a path leading to a lower potential value. This algorithm is resolution complete, which means that it guarantees finding a solution, if it exists, if the discretization of the search space is fine enough. However, its complexity is exponential in the dimension of *C-free space*, hence it is only applicable in low-dimensional spaces. Furthermore, in environments with

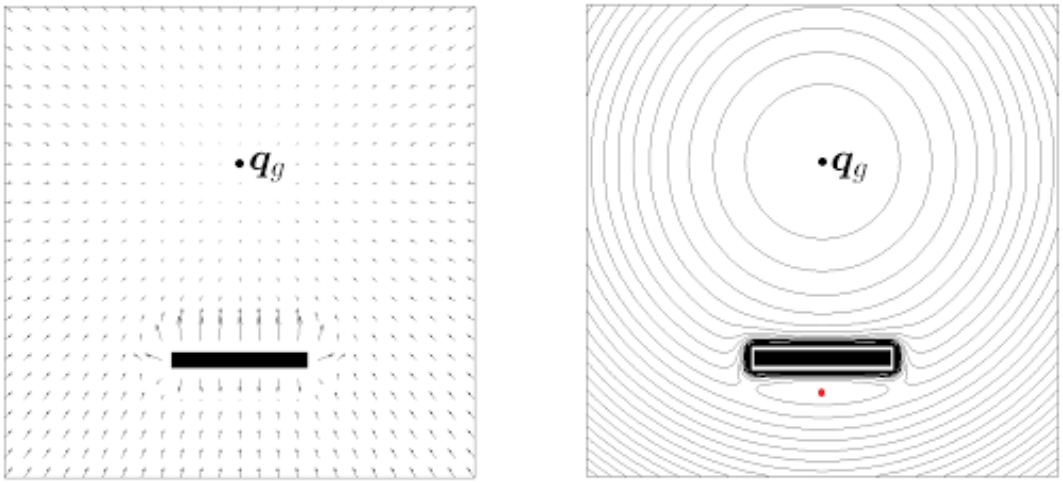

Figure 1.4: Goal g_g and a C-obstacle. Left: the vector field corresponding to an APF. Right: the level curves of the potential U where can see a local minimum marked with a red point in the plot

high obstacle density or complex obstacle shapes, the execution time can be slower.

- **Navigation functions**

  Since path generated by best-first algorithms doesn't avoid local minima (instead we seek a way to escape from it) we search a different approach: build navigation functions, i.e potentials without local minima. For example, we can define the potential as an harmonic function, solution which will be discussed in depth in section 1.2.1.

- **Vortex fields**

  The idea is to replace the repulsive action (which is the responsible for appearance of local minima) with an action forcing the robot to go around the $C - obstacle$.

  For example, if we assume $C = \mathbb{R}^2$ we can define the vortex field for CO_i as

$$\vec{F}_{vort} = \pm \begin{pmatrix} \frac{\partial U_{rep_i}}{\partial y} \\ -\frac{\partial U_{rep_i}}{\partial x} \end{pmatrix}$$

  where the the repulsive force generated by the potential field is now given by the vortex force $\vec{F}_{vort}$ .The intensity of the field remains the same, only the direction changes. If CO_i is convex, as we assumed before, the vortex sense

(clockwise or counter-clockwise) can be always chosen in such way that the total field has no local minima. In particular, the vortex sense should be chosen depending on the entrance point of the robot in the area of influence of the CO_i.

1.2 Partial differential equations

Throughout this paper we will explore the workaround that includes building functions without local minima, also called navigation functions. For doing this, first we need to introduce the *Laplace* and *Poisson* partial differential equation.

1.2.1 Harmonic functions

Equilibrium problems in two-dimensional and higher space, give rise to elliptic partial differential equations. A prototype is the famous Laplace's equation:

$$\Delta\phi = \nabla^2\phi = \sum_{i=1}^{n} \frac{\partial^2\phi}{\partial x_i^2} = 0 \tag{1.8}$$

Remark 1.2. Δ is the Laplacian operator, also denoted as ∇^2.

This equation holds for the steady temperature in an isotropic medium, characterizes gravitational or electrostatic potentials at points of empty space, and describes the velocity potential of an irrotational, incompressible fluid flow as we will see. An harmonic function $\phi \in \mathcal{C}^2$ on a domain $\Omega \in \mathbb{R}^n$ is a function which satisfies Laplace's equation.

Properties of the Harmonic function

The first important property of a harmonic function is the principle of superposition, which follows from the linearity of the Laplace equation. That is, if ϕ_1 and ϕ_2 are harmonic, then a linear combination of ϕ_1 and ϕ_2 is also harmonic and a solution of Laplace equation .

There are other important properties of harmonic functions we want to remark, but for explaining them we shall introduce some notation following [2], [22]:

Let Ω be an open bounded subset of $\mathbb{R}^n$. An open ball centered at a of radius r is defined as $B(a,r) = \{x \in \Omega : \|x-a\| < r\}$; its closure is the closed ball $\overline{B}(a,r)$; the unit ball $B(0,1)$ is denoted by B and its closure is $\overline{B}$. A *dashed integral* stands for an averaged integral, that is $\fint_\Omega f \quad d\mu = \mu(\Omega)^{-1} \int_\Omega f \quad d\mu$, where μ is a measure in $\mathbb{R}^n$.

Proposition 1.3. (Spherical means) *Let $u \in \mathcal{C}^2(\Omega)$. Then, for $x \in \Omega$ and $r > 0$ with $B(x,r) \subset \Omega$, we have*

$$\frac{d}{d\rho} \fint_{\partial B(x,\rho)} u(y) \quad d\mathcal{H}^{n-1}(y) = \frac{\rho}{n} \fint_{B(x,\rho)} \Delta u(y) \quad dy, \tag{1.9}$$

for all $\rho \in (0,r)$, where $d\mathcal{H}^{n-1}(y)$[1] indicates that the integration is being performed over the $(n-1)$ dimensional boundary of $B(x,\rho)$.

Proof. Fix $x \in \Omega$ and $r > 0$ such that $B(x,r) \subset \Omega$. Then for $0 < \rho < r$ consider the spherical means of u over $\partial B(x,\rho) \subset \Omega$. Upon translating and rescaling by introducing $y = x - \rho\xi$, we can write

$$\fint_{\partial B(x,\rho)} u(y) \quad d\mathcal{H}^{n-1}(y) = \fint_{\partial B(0,1)} u(x+\rho\xi) \quad d\mathcal{H}^{n-1}(\xi).$$

This is actually the definition of spherical means. Using that we obtain the following equality

$$\tfrac{d}{d\rho} \fint_{\partial B(x,\rho)} u(x) \quad d\mathcal{H}^{n-1}(y) = \frac{d}{d\rho} \fint_{\partial B(0,1)} u(x+\rho\xi) \quad d\mathcal{H}^{n-1}(\xi)$$

Now we apply the chain rule $\frac{d}{d\rho} u(x) = \langle \nabla u(x+\rho\xi), \xi \rangle$ and we get

$$\tfrac{d}{d\rho} \fint_{\partial B(0,1)} u(x+\rho\xi) \quad d\mathcal{H}^{n-1}(\xi) = \fint_{\partial B(0,1)} \langle \nabla u(x+\rho\xi), \xi \rangle \quad d\mathcal{H}^{n-1}(\xi)$$

By rescaling back $y = x + \rho\xi$, it follows

$$\fint_{\partial B(0,1)} \langle \nabla u(x+\rho\xi), \xi \rangle \quad d\mathcal{H}^{n-1}(\xi) = \fint_{\partial B(x,\rho)} \langle \nabla u(y), \frac{y-x}{\rho} \rangle \quad d\mathcal{H}^{n-1}(y)$$

Observing that $\langle \nabla u(y), \frac{y-x}{\rho} \rangle$ is the directional derivative in the direction $\nu = \frac{y-x}{\rho}$, which is the outward-pointing unit normal vector to the surface $\partial B(x,\rho)$,

$$\fint_{\partial B(x,\rho)} \langle \nabla u(y), \frac{y-x}{\rho} \rangle \quad d\mathcal{H}^{n-1}(y) = \fint_{\partial B(x,\rho)} \frac{\partial u}{\partial \nu}(y) \quad d\mathcal{H}^{n-1}(y)$$

Finally, using the divergence theorem, we get the required identity

$$\fint_{\partial B(x,\rho)} \frac{\partial u}{\partial \nu}(y) \quad d\mathcal{H}^{n-1}(y) = \frac{\rho}{n} \fint_{B(x,\rho)} \Delta u(y) \quad dy$$

□

[1] $\mathcal{H}^{n-1}$ is the $(n-1)$ dimensional Hausdorff measure [16]. Note that his measure is the volume on the previous dimension. As we will work in $\mathbb{R}^2$, we are talking about the line integral.

Proposition 1.4. (The Mean-Value Property) *Suppose Ω is connected, u is real valued and harmonic on Ω. Then*

$$u(x) = \fint_{\partial B(x,r)} u(y) \quad dy \tag{1.10}$$

$$u(x) = \fint_{B(x,r)} u(y) \quad dy \tag{1.11}$$

for $\forall x \in \Omega$ and $r > 0$ such that $B(x,r) \subset \Omega$.

Proof. Fix $x \in \Omega$ and $r > 0$ such that $B(x,r) \subset \Omega$. Since Ω is open, there exists $s > r$ such that $\overline{B}(x,r) \subset B(x,s) \subset \Omega$. Hence, referring to (1.9) and using that u is harmonic, it follows that for $\rho \in (0,s)$,

$$\tfrac{d}{d\rho} \fint_{\partial B(x,\rho)} u(y) \quad d\mathcal{H}^{n-1}(y) = 0.$$

Integrating the above with respect to ρ from 0 to r we get

$$\int_0^r \frac{d}{d\rho} \fint_{\partial B(x,r)} u(y) d\mathcal{H}^{n-1}(y) = 0$$

By the Fundamental Theorem of Calculus, this expression becomes

$$\fint_{\partial B(x,r)} u(y) \quad d\mathcal{H}^{n-1}(y) - \fint_{\partial B(x,0)} u(y) \quad d\mathcal{H}^{n-1}(y) = 0$$

Since $B(x,0)$ is just the point x, we have

$$\fint_{\partial B(x,r)} u(y) \quad d\mathcal{H}^{n-1}(y) = u(x)$$

Starting with 1.10 (with ρ in place of r) and by the definition of the spherical mean, we can rewrite it as

$$u(x) = \tfrac{1}{\omega_{n-1}\rho^{n-1}} \int_{\partial B(x,\rho)} u(y) \quad d\mathcal{H}^{n-1}(y)$$

where ω_{n-1} is the surface area of the unit $(n-1)$ sphere. Multiplying both sides by $n\omega_n\rho^{n-1}$ we obtain

$$n\omega_n\rho^{n-1}u(x) = \tfrac{n\omega_n\rho^{n-1}}{\omega_{n-1}\rho^{n-1}} \int_{\partial B(x,\rho)} u(y) \quad d\mathcal{H}^{n-1}(y)$$

by integrating with respect of ρ in interval $[0,r]$ and knowing that $n\omega_n = \omega_{n-1}$ it follows

$$\int_0^r n\omega_n\rho^{n-1}u(x) = \int_0^r \int_{\partial B(x,\rho)} u(y) \quad d\mathcal{H}^{n-1}(y)$$

Recognize that the right part is actually the integral of $u(y)$ over the volume of the ball $B(x,r)$ so we have

$$\omega_n r^n u(x) = \int_{B(x,r)} u(y) \quad d\mathcal{H}^{n-1}(y)$$

Dividing both sides by $w_n r^n$ gives

$$u(x) = \frac{1}{\omega_n r^n} \int_{B(x,r)} u(y) \quad d\mathcal{H}^{n-1}(y) = \fint_{B(x,r)} u(y) \quad dy$$

□

Proposition 1.5. (The Maximum Principle) *Suppose Ω is connected, u is real valued and harmonic on Ω, and u has a maximum or a minimum in Ω. Then u is constant.*

Proof. Suppose u attains a maximum at $a \in \Omega$. Choose $r > 0$ such that $\overline{B}(a,r) \subset \Omega$. If u were less than $u(a)$ at some point of $B(a,r)$, then the continuity of u would show that the average of u over $B(a,r)$ is less or equal than $u(a)$, contradicting (1.10). Therefore u is constant on $B(a,r)$, proving that the set were u attains its maximum is open in Ω. Because this set is also closed in Ω (again by continuity of u), it must be all of Ω (by connectivity). Thus u is constant on Ω, as desired. If u attains a minimum in Ω, we can apply this argument to $-u$. □

The unique properties of harmonic functions make them ideal for constructing artificial potential fields for obstacle avoidance [9]. These properties ensure that the potential field does not have local minima, which can cause the robot to get stuck. By appropriately defining the source term function, as explained in Section 3.3, we can create a field where the goal is the only global minima and no other critical point is generated when adding obstacles.

1.2.2 Boundary conditions

The different kinds of contour conditions imposed to Laplace's equation have a critical importance in the solution of the equation and the quality of the trajectory that will follow the robot. The following forms of the Dirichlet and Neumann boundary conditions will be used

- **Dirichlet**

  In Dirichlet's conditions case, the boundary is maintained at a constant value higher than the goal point. As the boundary value is fixed, the vector field is normal to the boundary. The Dirichlet boundary condition is

$$\phi|_{\partial\Omega} = c, \quad c \in \mathbb{R}.$$

One requires to define a proper source term f as described on 3.3, so our PGD approximation ϕ satisfies $c > \phi(q_{goal})$. This solution tends to have precision problems, though. Flat regions can develop resulting in very small (but necessarily nonzero) gradients, requiring higher precision in generating the solution trajectory.

- **Neumann**

  Neumann's conditions constrain the normal component of the gradient to be zero at the boundaries. As there is no normal component of fluid flow, the condition forces the flow to be tangential to the boundary. The Neumann boundary condition is

  $$\nabla\phi|_{\partial\Omega} = c, \quad c \in \mathbb{R}.$$

  In our case we need $c = 0$ so that any trajectory leave the domain. When $c > 0$ the boundary push the trajectories inwards, so it could also be an option. In Neumann's conditions case, the descent towards $\partial\Omega_{free}$ is smooth and continuous, with a slope not close to zero and, because of that, the trajectory calculation is more numerically stable than in Dirichlet case.

1.2.3 Potential flow theory

We will approach the path planning problem as a mathematical model describing the flow of an inviscid incompressible fluid. Assuming a steady irrotational flow in the three-dimensional Euclidean space ($\mathbb{R}^3$), the velocity field V vanishes

$$\nabla \times V = 0 \tag{1.12}$$

As a consequence, the velocity is the gradient of a scalar (potential) function ϕ, $V = -\nabla\phi$. Furthermore, when the fluid is incompressible, the velocity field must satisfy $\operatorname{div} V = 0$. By joining the two previous expressions, we get

$$\nabla^2\phi = 0 \tag{1.13}$$

so the potential is solution of the Laplace equation, hence ϕ is harmonic inside any domain $\Omega \in \mathbb{R}^3$. To force the flow to reach the target we choose to pass through a crucial step. A localized fluid source (or a sink) can be modeled by a Dirac term (δ) added to the right hand side of (1.13). Assuming a unit amount of fluid injected at point S during a unit of time and the same unit withdrawn at point T, the velocity potential is now solution of the Poisson equation:

$$-\nabla^2\phi = \delta_S - \delta_T, \tag{1.14}$$

where δ_S means the fluid source and δ_T the target sink.

This equation must be complemented by appropriate boundary conditions. The fluid cannot flow through the boundaries, so it must satisfy a condition expressed by $V \cdot n = 0$ (n being a normal vector to the boundary Γ). So, on the boundary Γ, the potential must verify:

$$-\phi \cdot n = 0 \tag{1.15}$$

which amounts to the Neumann boundary condition, which we will see applied later:

$$\frac{\partial \phi}{\partial n}\Big|_{\Gamma} = 0 \tag{1.16}$$

1.2.4 Poisson equation: The weak formulation

Given that our approach leads us to solve a particular Poisson equation (1.14), for now on we will focus on solving this type of PDE's, starting from some basic examples and making them more complex.

Let us consider the Poisson problem posed in a domain Ω, an relatively compact subset of $\mathbb{R}^d, d \geq 1$ supplemented with homogeneous Dirichlet boundary conditions:

$$\begin{aligned} -\Delta u(x) &= f(x), \quad \forall x \in \Omega \\ u(x) &= 0, \quad \forall x \in \partial\Omega \end{aligned} \tag{1.17}$$

with $f \in \mathcal{C}^0(\overline{\Omega}), \quad \overline{\Omega} = \partial\Omega \cup \Omega$.

Definition 1.6. A *classical solution* (or strong solution) of equation (1.17) is a function $u \in \mathcal{C}^2(\Omega)$ that satisfies both conditions.

We may want to relax the pointwise regularity (i.e. continuity) required to ensure the existence of the classical derivative to the (weaker) existence of the distributional derivative. The strong formulation requires solutions to be twice differentiable, and we may search for solutions that do not possess this degree of smoothness. For example, if we model the source term f with delta Dirac functions (3.3) the solution is typically not twice differentiable. The weak formulation, which will be defined later, also allows the incorporation of boundary conditions into the solution process by means of choosing the proper test functions. It is also very suitable when using numerical methods, like the finite element method, as can be seen in 2.3.1. The weak formulation can be discretized, leading to a system of algebraic equations that can be solved numerically with higher stability than the strong formulation. For this reasons, for now on we will work with the weak solution.

Let's consider the Lebesgue space

$$L^n(\Omega) = \{u : \int_\Omega |u(x)|^n dx < \infty\}$$

and $L^n_{loc}(\Omega) = \{\phi : \mathbb{R}^n \to \mathbb{R}| \quad \phi \in L^1(K) \text{ for all compact sets } K \subset \Omega\}$.

Definition 1.7. (Weak derivatives) *A function $u \in L^1_{loc}(\mathbb{R}^n)$ is* weakly differentiable *with respect to x_i if there exists a function $g_i \in L^1_{loc}(\mathbb{R}^n)$ such that*

$$\int_{\mathbb{R}^n} u\partial_i\phi \quad dx = -\int_{\mathbb{R}^n} g_i\phi \quad dx, \forall\phi \in \mathcal{C}^\infty_c(\mathbb{R}^n)$$

where $\mathcal{C}^\infty_c(\mathbb{R}^n) := \{\phi : \mathbb{R}^n \to \mathbb{R}| \quad \phi \in \mathcal{C}^\infty(\mathbb{R}^n), and\ \phi\ has\ compact\ support\}$. The function g_i is called the weak ith-partial derivative *of u and is denoted by $\partial_i u$.*

Let $u \in C^2(\Omega)$ be a classical solution to (1.17) and let us test the equation against any smooth function $\varphi \in \mathcal{C}^\infty_c(\Omega)$, also called test function.

$$-\int_\Omega \Delta u(x)\varphi(x) \quad dx = \int_\Omega f(x)\varphi(x) \quad dx \tag{1.18}$$

Since $u \in \mathcal{C}^2(\Omega)$, Δu is well defined. Integrating by parts, the left hand reads:

$$-\int_\Omega \Delta u(x)\varphi(x) \quad dx = -\int_{\partial\Omega} \nabla u(x)\cdot n\varphi(x) \quad ds + \int_\Omega \nabla u(x)\nabla\varphi(x) \quad dx \tag{1.19}$$

Since φ has compact support in Ω, it vanishes on the boundary $\partial\Omega$, consequently the boundary integral is zero, thus the distributional formulation reads

$$\int_\Omega \nabla u(x)\cdot\nabla\varphi(x) \quad dx = \int_\Omega f(x)\varphi(x) \quad dx \quad ,\forall\varphi \in C^\infty_c(\Omega) \tag{1.20}$$

Definition 1.8. (Topological dual space) The topological dual space $\mathcal{V}'$ of a normed vector space $\mathcal{V}$ is the vector space of continuous linear forms on $\mathcal{V}$ equipped with the norm:

$$\|f\|_{\mathcal{V}'} = \sup_{x\in\mathcal{V}, x\neq 0} \frac{f(x)}{\|x\|_\mathcal{V}}$$

Consider H and $\mathcal{V}$ as normed function spaces yet to be defined, both satisfying regularity constraints and for H boundary condition constraints.

A **weak formulation** of (1.17) consists in finding $u \in H$, given $f \in \mathcal{V}'$, such that

$$\int_\Omega \nabla u\cdot\nabla v \quad dx = \int_\Omega fv \quad dx \quad ,\forall v \in \mathcal{V} \tag{1.21}$$

Any solution of 1.21 is a **weak solution** of the Dirichlet problem 1.17.

Provided that the weak solution to (1.21) belongs to $\mathcal{C}^2(\Omega)$ then the second derivatives exist in the classical sense. Consequently the integration by parts can be performed the other way around and the weak solution is indeed a classical solution.

Functional settings

Since (1.21) involves first order derivatives, then we should consider a solution in Sobolev space

$$H^1(\Omega) = \{u \in L^2(\Omega) : Du \in L^2(\Omega)\},$$

endowed with the Sobolev norm $\|\cdot\|_{H^1} = \langle\cdot,\cdot\rangle^{1/2}_{H^1(\Omega)}$ defined from the scalar product that gives us a Hilbert structure,

$$\langle u, v\rangle_{H^1(\Omega)} = \int_\Omega uv \quad dx + \int_\Omega \nabla u \cdot \nabla v \quad dx$$

Moreover, the solution should satisfy the boundary condition of the strong form of the PDE problem. The homogeneous Dirichlet condition is embedded in the function space of the solution: u vanishing on the boundary $\partial\Omega$ yields that we should seek u in $H^1_0(\Omega)$, the compact support subspace of $H^1(\Omega)$.

Choice of test space

In order to give sense to the solution in a Hilbert-Sobolev space we need to choose the test function φ itself in the same kind of space. If we chose $\varphi \in H^1_0(\Omega)$ then by definition, we can construct a sequence $(\varphi_n)_{n\in\mathbb{N}}$ of functions in $C^\infty_c(\Omega)$ converging in $H^1_0(\Omega)$ to φ,

$$\|\varphi_n - \varphi\|_{H^1(\Omega)} \to 0 \quad \text{, as } n \to +\infty$$

For the sake of completeness, we note that we can pass to the limit in the formulation, term by term for any partial derivative:

$$\int_\Omega \partial^i u \cdot \partial^i \varphi_n \to \int_\Omega \partial^i u \cdot \partial^i \varphi$$

as $\partial^i \varphi_n \to D_i\varphi$ in $L^2(\Omega)$, and

$$\int_\Omega f\varphi_n \to \int_\Omega f\varphi, \quad \forall f \in \mathcal{V}$$

as $\varphi_n \to \varphi$ in $L^2(\Omega)$. Consequently the weak formulation (1.21) is satisfied if $\varphi \in H^1_0(\Omega)$.

Choice of solution space

The determination of the function space is guided by two main factors: the regularity of the solution and the boundary conditions.

Firstly, if u is a classical solution then it belongs to $C^2(\Omega)$ which implies that $u \in L^2(\Omega)$ and $\partial^i u \in L^2(\Omega)$, thus $u \in H^1_0(\Omega)$.

Secondly the solution should satisfy the Dirichlet boundary condition on $\partial\Omega$. This requirement is fulfilled by the following trace theorem:

Lemma 1.9. *(Trace Theorem) Let Ω be a bounded open subset of $\mathbb{R}^d$ with piecewise C^1 boundary, then there exists a linear application $\gamma : H^1(\Omega) \to L^2(\partial\Omega)$ continuous on $H^1(\Omega)$.*

From the trace theorem, defining $\gamma(u) = u|_{\partial\Omega}$, we know that $Ker(\gamma) = H_0^1(\Omega)$ and $\gamma(u) = 0$, so we can conclude that $u \in H_0^1(\Omega)$ to satisfy the boundary conditions.

With all that, we conclude that $H = \mathcal{V} = H_0^1(\Omega)$ and the weak formulation for (1.17) search $u \in H_0^1(\Omega)$ satisfying:

$$\int_\Omega \nabla u \cdot \nabla v \quad dx = \int_\Omega fv \quad dx \quad ,\forall v \in H_0^1(\Omega), \quad f \in \mathcal{V}'. \tag{1.22}$$

More generally, we can define the problem as finding u satisfying

$$a(u, v) = L(v) \quad ,\forall v \in \mathcal{V} \tag{1.23}$$

with $a(\cdot, \cdot)$ a continuous bilinear form on $\mathcal{V} \times \mathcal{V}$ and $L(\cdot)$ a continuous linear form on $\mathcal{V}$.

In our previous case (1.22), the bilinear form corresponds to

$$\begin{aligned} a : \quad & \mathcal{V} \times \mathcal{W} \to \mathbb{R} \\ & (u, v) \mapsto \int_\Omega \nabla u \cdot \nabla v \quad dx \end{aligned}$$

and the linear form to

$$\begin{aligned} L : \quad & \mathcal{V} \to \mathbb{R} \\ & v \mapsto \int_\Omega fv \quad dx \end{aligned}$$

Proposition 1.10. (Continuity) *A bilinear form $a(\cdot, \cdot)$ is continuous on $\mathcal{V} \times \mathcal{W}$ if there exists a positive constant real number M such that*

$$a(v, w) \leq M\|v\|_V\|w\|_W \quad ,\forall (v, w) \in \mathcal{V} \times \mathcal{W}$$

The continuity of these two forms comes directly from that they are respectively the inner-product in $H_0^1(\Omega)$, and the L^2 inner-product with $f \in L^2(\Omega)$: Using the Cauchy-Schwartz inequality in the context of L^2 inner product, we get:

$$|a(u,v)| = \left| \int_\Omega \nabla u \cdot \nabla v \quad dx \right| \leq \left(\int_\Omega |\nabla u|^2 dx \right)^{\frac{1}{2}} \left(\int_\Omega |\nabla v|^2 dx \right)^{\frac{1}{2}}$$

We note that for $u \in H_0^1(\Omega)$, the H^1 norm is defined as:

$$\|u\|_{H^1(\Omega)} = \left(\|u\|^2_{L^2(\Omega)} + \|\nabla u\|^2_{L^2(\Omega)} \right)^{\frac{1}{2}}$$

Since u vanishes on the boundary, the $H_0^1(\Omega)$ can be effectively denominated by:

$$\|u\|_{H^1(\Omega)} = \|\nabla u\|_{L^2(\Omega)}$$

Therefore, we have :

$$|a(u,v)| \leq \|\nabla u\|_{L^2(\Omega)} \|\nabla v\|_{L^2(\Omega)} = \|u\|_{H^1(\Omega)} \|v\|_{H^1(\Omega)}$$

1.2.5 Well-posedness

In the usual sense, a problem is well-posed if it admits a unique weak solution which is bounded in the V -norm by the data (forcing term, boundary conditions) which are independent on the solution. In this particular case of the Poisson problem the bilinear form $a(\cdot,\cdot)$ is the natural scalar product in $H_0^1(\Omega)$, thus it defines a norm in $H_0^1(\Omega)$.

Theorem 1.11. (Riesz Representation theorem) *For a continuous linear function ϕ on a Hilbert space H, there exists a unique $u \in H$ such that $\phi(v) = \langle u, v \rangle, \quad \forall v \in H$. Furthermore, $\|u\|_H = \|\phi\|_H$*

This result, first announced in [20], ensures directly the existence and uniqueness of a weak solution as soon as $a(\cdot,\cdot)$ is a scalar product and L is continuous for $\|\cdot\|_a$. If the bilinear form $a(\cdot,\cdot)$ is not symmetric then the previous theorem does not apply. Therefore, we can guarantee that our algorithm will be able to find a path that goes from the source point to the target point avoiding the boundaries of our domain and, in this way, the obstacles.

Chapter 2

PGD as a Solution of the Poisson Equation

To illustrate how does PGD work, we will begin with a simple case study, which we shall progressively make more complex until the equation to be solved gives us the APF.

One of the main advantages of this method, and one of the main reasons for choosing it over another algorithm, is the capability of transform high dimensional problems into a series of decoupled one-dimensional problems formulated in each domain Ω. We will see a two dimensional example on this work, but when adding more complexity factors like obstacles (for example in [7]), this property becomes very relevant since makes the execution time feasible.

In this chapter, we will follow the structure of the second unit of [3], while adding some annotations and pseudo-code for making it more detailed and easier to understand.

2.1 The Poisson equation

Consider the solution of the Poisson equation

$$\Delta u(x,y) = f(x,y), \quad u, f \in H_0^1(\Omega) \tag{2.1}$$

in a two-dimensional rectangular domain $\Omega = \Omega_x \times \Omega_y = (0,L) \times (0,H)$, with homogeneous Dirichlet boundary conditions for the unknown field $u(x,y)$, i.e $u(x,y)\Big|_{\partial\Omega} = 0$. Furthermore, we assume that the source term f is constant over the domain Ω.

We can write (2.1) in a weak formulation. For all suitable test functions $u^* \in H_0^1(\Omega)$, its weighted residual form reads

$$\int_{\Omega_x \times \Omega_y} u^* \cdot (\Delta u(x,y) - f) \quad dx \cdot dy = 0 \tag{2.2}$$

Now, our main goal is to obtain a Proper Generalized Decomposition [1] approximate solution to (2.1) in the separated form

$$u(x,y) = \sum_{i=1}^{N} X_i(x) \cdot Y_i(y) \tag{2.3}$$

We will do it iterating over three basis steps: enrichment, alternating direction and stopping criterion. Just below we have a pseudo-code that will help us to understand how this PGD works, from a high level perspective. Throughout the detailed explanation, we will also refer to the actual code used on each process. The whole code can be found on the appendix of this work (A).

```
public static void main(String[] args) {
    //Declare your empty solution
    std::vector<std::vector<double>> solution = createMatrix();
    //Loop until convergence of the solution
    while(!solutionConverges(solution)){
        //Add another iteration step (n) to the enrichment process
        Y_n^0 = randomVector()
        X_n^1 = computeAlternating(Y_n^0)
        //Compute the alternating direction scheme
        while(!stoppingCriterionEnrichmentProcess()){
            Y_n^i = computeAlternating(X_n^{i-1})
            X_n^i = computeAlternating(Y_n^i)
        }
        solution += X_n^final · Y_n^final
    }
}
```

2.2 Progressive Construction of the Separated Representation

At each enrichment step n $(n \geq 1)$, we have already computed the $n-1$ first terms of the PGD approximation (2.3):

$$u^{n-1}(x,y) = \sum_{i=1}^{n-1} X_i(x) \cdot Y_i(y) \tag{2.4}$$

We want to compute the next pair of terms $X_n(x)$, $Y_n(y)$ to obtain the enriched PGD solution

$$u^n(x,y) = u^{n-1}(x,y) + X_n(x) \cdot Y_n(y) = \sum_{i=1}^{n-1} X_i(x) \cdot Y_i(y) + X_n(x) \cdot Y_n(y) \tag{2.5}$$

For computing those terms, that are unknown at the current step n, an iterative scheme is used. The iterative scheme that fits our model more closely is the alternating direction strategy, detailed in section 2.2.2. We will use the index p to denote a particular iteration of the alternating scheme. This scheme consists in computing $X_n^p(x)$ from $Y_n^{p-1}(y)$, and then $Y_n^p(y)$ from $X_n^p(x)$. An arbitrary initial guess Y_n^0 is specified so start the iterative process and proceed until reaching a fixed point within a desired tolerance ϵ.

$$\frac{\|X_n^p(x) \cdot Y_n^p(y) - X_n^{p-1}(x) \cdot Y_n^{p-1}(y)\|}{\|X_n^{p-1}(x) \cdot Y_n^{p-1}(y)\|} < \epsilon \tag{2.6}$$

where $\|\cdot\|$ is a suitable norm. We can see the implementation in A.1.

In a particular enrichment step n, the PGD approximation $u^{n,p}$ obtained at iteration p reads as

$$u^{n,p}(x,y) = u^{n-1}(x,y) + X_n^p(x) \cdot Y_n^p(y) \tag{2.7}$$

When the fixed point is good enough we end this iterative process with the assignments $X_n(x) \leftarrow X_n^p(x)$ and $Y_n(x) \leftarrow Y_n^p(x)$.

The enrichment process itself stops when an appropriate measure of error $\epsilon(n)$ becomes small becomes small enough. Several stopping criteria are suitable, but as we shall argument later, in our particular case this choice will not matter at all, since the robot itself will already generate an error of a higher order than a bad choice of the norm.

2.2.1 Stopping Criterion for the Enrichment Process

A first stopping criterion is associated with the relative weight of the newly computed term within the PGD expansion. Thus, $\epsilon(n)$ is usually given by

$$\epsilon(n) = \frac{\|X_n(x) \cdot Y_n(y)\|}{\|u^n(x,y)\|} = \frac{\|X_n(x) \cdot Y_n(y)\|}{\|\sum_{i=1}^n X_i(x) \cdot Y_i(y)\|} \tag{2.8}$$

This criterion involves the computation of $n+1$ M-dimensional vector products and, despite is not has a high computational cost, we can avoid it by with a similar but less expensive criterion. Keep in mind that depending on the chosen norm, the computational cost can be increased. For instance, for the L^2-norm we have

$$\|X_n(x)\cdot Y_n(y)\|_2 = \left(\int_{\Omega_x\times\Omega_y}(X_n(x))^2\cdot(Y_n(x))^2 \quad dx\cdot dy\right)^{1/2}$$
$$= \left(\int_{\Omega_x}(X_n(x))^2 \quad dx\right)^{1/2}\cdot\left(\int_{\Omega_y}(Y_n(x))^2 \quad dy\right)^{1/2} \tag{2.9}$$

we can see that using this norm involves $2+n\cdot(n+1)$ one-dimensional integrals. An alternative is

$$\epsilon(n) = \frac{\|X_n(x)\cdot Y_n(y)\|}{\|X_1(x)\cdot Y_1(y)\|}. \tag{2.10}$$

This criterion involves way less operations and the level of the error precision we need is not particularly high. Since the final goal is to work with a robot, and it has precision error by itself, it is not so important to be very accurate. Hence, the chosen stopping criterion is 2.10, see implementation in lines [87-94] of A.1.

2.2.2 Alternating Direction Strategy

An alternating direction strategy is a computational technique usually used to solve partial differential equations (PDEs) and optimization problems [19, 18]. The basic idea is to break a complex problem into simpler sub-problems that can be solved more easily by alternating between different directions or dimensions. On this example, we will break down a two-dimensional problem searching at each step the solution for a single one-dimensional direction (alternating between the $x-direction$ and the $y-direction$). The workflow is as follows:

$$Y_n^0 \longrightarrow X_n^1 \longrightarrow Y_n^1 \longrightarrow X_n^2 \longrightarrow Y_n^2 \longrightarrow ... \longrightarrow X_n^p \longrightarrow Y_n^p$$

where Y_n^i and X_n^j denote the $i-th$ and $j-th$ iteration of the alternating direction strategy on the $n-th$ step of the enrichment process. The whole alternating direction iterative process can be found between the lines [257-282] of A.1.

Each iteration of the alternating direction scheme consists in the following two steps:

1. **Calculating $X_n^p(x)$ from $Y_n^{p-1}(y)$.** In this case, the approximation reads

$$u^{n,p} = \sum_{i=1}^{n-1} X_i(x)\cdot Y_i(y) + X_n^p(x)\cdot Y_n^{p-1}(y) \tag{2.11}$$

where all functions are known except $X_n^p(x)$. The most intuitive choice for the weight function u^* in the weighted residual formulation (2.2) is

$$u^* = X_n^*(x)\cdot Y_n^{p-1}(y) \tag{2.12}$$

which amounts to select the Galerkin weighted residual form of the Poisson equation. Injecting (2.11) and (2.12) into (2.2), we obtain

$$\begin{aligned}
&\int_{\Omega_x \times \Omega_y} X_n^* \cdot Y_n^{p-1} \cdot \left(\frac{\partial^2 X_n^p}{\partial x^2} \cdot Y_n^{p-1} + X_n^p \cdot \frac{\partial^2 Y_n^{p-1}}{\partial y^2} \right) \quad dx \cdot dy \\
&= - \int_{\Omega_x \times \Omega_y} X_n^* \cdot Y_n^{p-1} \cdot \sum_{i=1}^{n-1} \left(\frac{\partial^2 X_i}{\partial x^2} \cdot Y_i + X_i \cdot \frac{\partial^2 Y_i}{\partial y^2} \right) \quad dx \cdot dy \\
&+ \int_{\Omega_x \times \Omega_y} X_n^* \cdot Y_n^{p-1} \cdot f \quad dx \cdot dy
\end{aligned} \tag{2.13}$$

Note that all functions depending on y are already known, so we can compute the following one-dimensional integrals over Ω_y:

$$\begin{cases}
\alpha^x & = \int_{\Omega_y} \left(Y_n^{p-1}(y) \right)^2 \quad dy \\
\beta^x & = \int_{\Omega_y} Y_n^{p-1}(y) \cdot \frac{\partial^2 Y_n^{p-1}}{\partial y^2} \quad dy \\
\gamma_i^x & = \int_{\Omega_y} Y_n^{p-1}(y) \cdot Y_i(y) \quad dy \\
\delta_i^x & = \int_{\Omega_y} Y_n^{p-1}(y) \cdot \frac{\partial^2 Y_i}{\partial y^2} \quad dy \\
\xi^x & = \int_{\Omega_y} Y_n^{p-1}(y) \cdot f \quad dy
\end{cases} \tag{2.14}$$

Then, the previous equation (2.12) becomes

$$\begin{aligned}
&\int_{\Omega_x} X^* \cdot \left(\alpha^x \cdot \frac{\partial^2 X_n^p}{\partial x^2} + \beta^x \cdot X_n^p \right) \quad dx \\
&= - \int_{\Omega_x} X_n^* \cdot \sum_{i=1}^{n-1} \left(\gamma_i^x \cdot \frac{\partial^2 X_i}{\partial x^2} + \delta_i^x \cdot X_i \right) + \int_{\Omega_x} X^* \cdot \xi^x \quad dx
\end{aligned} \tag{2.15}$$

This is the weighted residual form of a one-dimensional problem defined over Ω_x. We will use the finite element method to obtain the function X_n^p we are looking for. In this particular example, that is two-dimensional, we will return to the corresponding strong formulation

$$\alpha^x \cdot \frac{\partial^2 X_n^p}{\partial x^2} + \beta^x \cdot X_n^p = - \sum_{i=1}^{n-1} \left(\gamma_i^x \cdot \frac{\partial^2 X_i}{\partial x^2} + \delta_i^x \cdot X_i \right) + \xi^x \tag{2.16}$$

Then the operator $\frac{\partial^2 X_n^p}{\partial x^2}$ is discretized along a one-dimensional mesh, leading for a linear system of equations.We can solve it numerically by means of the Householder QR decomposition [21]. For the Householder QR decomposition implementation, a C++ library named *Eigen* (`https://gitlab.com/libeigen/eigen`) is used. The code can be found on lines [188-228] of A.1.

2. **Calculating $Y_n^p(y)$ from $X_n^p(x)$.** In fact, the procedure is completely analog from what we have just done. Indeed, we simply exchange the roles played by all relevant functions of x and y.

   Now, the approximation reads as

$$u^{n,p} = \sum_{i=1}^{n-1} X_i(x) \cdot Y_i(y) + X_n^p(x) \cdot Y_n^p(y) \tag{2.17}$$

   where the function sought is $Y_n^p(y)$.

   The Galerkin formulation (2.2) is obtained with the switched choice

$$u^*(x,y) = X_n^p(x) \cdot Y_n^*(y) \tag{2.18}$$

   Then, by introducing (2.15) and (2.15) into (2.2), we get

$$\begin{aligned} &\int_{\Omega_x \times \Omega_y} X_n^p \cdot Y_n^* \cdot \left(\frac{\partial^2 X_n^p}{\partial x^2} \cdot Y_n^p + X_n^p \cdot \frac{\partial^2 Y_n^p}{\partial y^2} \right) \quad dx \cdot dy \\ &= - \int_{\Omega_x \times \Omega_y} X_n^p \cdot Y_n^* \cdot \sum_{i=1}^{n-1} \left(\frac{\partial^2 X_i}{\partial x^2} \cdot Y_i + X_i \cdot \frac{\partial^2 Y_i}{\partial y^2} \right) \quad dx \cdot dy \\ &+ \int_{\Omega_x \times \Omega_y} X_n^p \cdot Y_n^* \cdot f \quad dx \cdot dy \end{aligned} \tag{2.19}$$

   This time all function of x are known, so we can compute the integrals over Ω_x to obtain

$$\begin{cases} \alpha^y &= \int_{\Omega_x} \left(X_n^p(x) \right)^2 \quad dx \\ \beta^y &= \int_{\Omega_x} X_n^p(x) \cdot \frac{\partial^2 X_n^p(x)}{\partial x^2} \quad dx \\ \gamma_i^y &= \int_{\Omega_x} X_n^p(x) \cdot X_i(x) \quad dx \\ \delta_i^y &= \int_{\Omega_x} X_n^p(x) \cdot \frac{\partial^2 X_i}{\partial x^2} \quad dx \\ \xi^y &= \int_{\Omega_x} X_n^p(x) \cdot f \quad dx \end{cases} \tag{2.20}$$

   Then by replacing on (2.18) we obtain

$$\begin{aligned} &\int_{\Omega_y} Y^* \cdot \left(\alpha^y \cdot \frac{\partial^2 Y_n^p}{\partial y^2} + \beta^y \cdot Y_n^p \right) \quad dy \\ &= - \int_{\Omega_y} Y_n^* \cdot \sum_{i=1}^{n-1} \left(\gamma_i^y \cdot \frac{\partial^2 Y_i}{\partial y^2} + \delta_i^y \cdot Y_i \right) + \int_{\Omega_y} Y_n^* \cdot \xi^y \quad dy \end{aligned} \tag{2.21}$$

   As before, we have thus obtained the weighted residual form of an elliptic problem defined over Ω_y whose solution is the function $Y_n^p(y)$. We can transform this expression into the strong formulation

$$\alpha^y \cdot \frac{\partial^2 Y_n^p}{\partial y^2} + \beta^y \cdot Y_n^p = - \sum_{i=1}^{n-1} \left(\gamma_i^y \cdot \frac{\partial^2 Y_i}{\partial y^2} + \delta_i^y \cdot Y_i \right) + \xi^y, \tag{2.22}$$

and we integrate by reducing it to a linear system after discretizing $\frac{\partial^2 Y_n^p}{\partial y^2}$ on discrete mesh and using Householder QR decomposition again.

We have thus completed iteration p at enrichment step n. We must realize that the original two-dimensional Poisson equation defined over $\Omega = \Omega_x \times \Omega_y$ has been transformed thanks to PGD into a series of decoupled one-dimensional problems formulated in each Ω_i.

2.3 Taking into Account Neumann Boundary Conditions

Previously, the only conditions we specified were Dirichlet boundary conditions. We will divide the domain boundary and force a flux or Neumann condition along each part of the domain boundary, and then unify those fluxes using the principle of superposition explained at section 1.2.1 :

$$\begin{cases} u(x=0,y) & = 0 \\ u(x=L,y) & = 0 \\ u(x,y=0) & = 0 \\ \frac{\partial u}{\partial u}|_{x,y=H} & = q \end{cases} \tag{2.23}$$

The objective is to integrate by parts the weighted residual form (2.2) and implement the flux condition as a so-called natural boundary condition:

$$-\int_{\Omega_x \times \Omega_y} \nabla u^* \cdot \nabla u \quad dx \cdot dy = \int_{\Omega_x \times \Omega_y} u^* \cdot f \quad dx \cdot dy - \int_{\Omega_x} u^*(x, y=H) \cdot q \quad dx \tag{2.24}$$

or more explicitly:

$$\begin{aligned} &\int_{\Omega_x \times \Omega_y} \left(\frac{\partial u^*}{\partial x} \cdot \frac{\partial u^*}{\partial x} + \frac{\partial u^*}{\partial y} \cdot \frac{\partial u^*}{\partial y} \right) \quad dx \cdot dy \\ &= -\int_{\Omega_x \times \Omega_y} u^* \cdot f \quad dx \cdot dy + \int_{\Omega_x} u^*(x, y=H) \cdot q \quad dx \end{aligned} \tag{2.25}$$

This is the starting point from which a PGD solution can be sought in the separated form

$$u(x,y) = \sum_{i=1}^{N} X_i(x) \cdot Y_i(y) \tag{2.26}$$

The PGD solution procedure then readily follows as described in the first case of study. The modified alternating direction can be found between the lines [269-294] of A.2. At enrichment step n, one iteration p of the alternating direction strategy amounts to the following computations:

1. **Calculating $X_n^p(x)$ from $Y_n^{p-1}(y)$.** At this stage, the PGD approximation is given by

$$u^{n,p} = \sum_{i=1}^{n-1} X_i(x) \cdot Y_i(y) + X_n^p(x) \cdot Y_n^{p-1}(y) \tag{2.27}$$

where X_n^p is the unknown function.

Using Galerkin's method, we select the following weight function

$$u^*(x,y) = X_n^*(x) \cdot Y_n^{p-1}(y) \tag{2.28}$$

Inserting (2.27) and (2.28) into (2.25), we obtain

$$\begin{aligned}
&\int_{\Omega_x \times \Omega_y} \left(\frac{\partial X_n^*}{\partial x} \cdot \frac{\partial X_n^p}{\partial x} \cdot \left(Y_n^{p-1} \right)^2 + X_n^* \cdot X_n^p \cdot \left(\frac{\partial Y_n^{p-1}}{\partial y} \right)^2 \right) \quad dx \cdot dy \\
= &- \int_{\Omega_x \times \Omega_y} \sum_{i=1}^{n-1} \left(\frac{\partial X_n^*}{\partial x} \cdot \frac{\partial X_i}{\partial x} \cdot Y_n^{p-1} \cdot Y_i + X_n^* \cdot X_i \cdot \frac{\partial Y_n^{p-1}}{\partial y} \cdot \frac{\partial Y_i}{\partial y} \right) \quad dx \cdot dy \\
&- \int_{\Omega_x \times \Omega_y} X_n^* \cdot Y_n^{p-1} \cdot f \quad dx \cdot dy + \int_{\Omega_x} X_n^* \cdot Y_n^{p-1}(x, y = H) \cdot q \quad dx
\end{aligned} \tag{2.29}$$

In the above expression, all functions of the coordinate y are known, and we can evaluate the corresponding one-dimensional integrals:

$$\begin{cases}
\alpha^x & = \int_{\Omega_y} \left(Y_n^{p-1}(y) \right)^2 \quad dy \\
\beta^x & = \int_{\Omega_y} \left(\frac{\partial Y_n^{p-1}(y)}{\partial y} \right)^2 \quad dy \\
\gamma_i^x & = \int_{\Omega_y} Y_n^{p-1}(y) \cdot Y_i(y) \quad dy \\
\delta_i^x & = \int_{\Omega_y} \frac{\partial Y_n^{p-1}(y)}{\partial y} \cdot \frac{\partial Y_i(y)}{\partial y} \quad dy \\
\xi^x & = \int_{\Omega_y} Y_n^{p-1}(y) \cdot f \quad dy \\
\mu^x & = Y_n^{p-1}(y = H) \cdot q
\end{cases} \tag{2.30}$$

As before, we thus obtain the weighted residual form of an elliptic problem for $X_n^p(x)$ defined over Ω_x:

$$\begin{aligned}
&\int_{\Omega_x} \left(\frac{\partial X_n^*}{\partial x} \cdot \frac{\partial X_n^p}{\partial x} \cdot \alpha^x + X_n^* \cdot X_n^p \cdot \beta^x \right) \quad dx = \\
&- \int_{\Omega_x} \sum_{i=1}^{n-1} \left(\frac{\partial X_n^*}{\partial x} \cdot \frac{\partial X_i}{\partial x} \cdot \gamma_i^x + X_n^* \cdot X_i \cdot \delta_i^x \right) \quad dx \\
&\qquad - \int_{\Omega_x} X_n^* \cdot \xi^x \quad dx + \int_{\Omega_x} X_n^* \cdot \mu^x \quad dx
\end{aligned} \tag{2.31}$$

The finite element method, which will be explained below, can then be used to discretize this one dimensional problem, with the remaining Dirichlet condition $X_n^p(x=0) = X_n^p(x=L) = 0$.

2. **Calculating $Y_n^p(y)$ from $X_n^p(x)$.** Here again, the second step of iteration p simply mirrors the first one with an exchange of role between x and y coordinates.

   The current PGD approximation reads

$$u^{n,p} = \sum_{i=1}^{n-1} X_i(x) \cdot Y_i(y) + X_n^p(x) \cdot Y_n^p(y) \tag{2.32}$$

   where $Y_n^p(y)$ is the only unknown function.

   Selecting the Galerkin method,

$$u^*(x,y) = X_n^p(x) \cdot Y_n^*(y) \tag{2.33}$$

   we introduce (2.32) and (2.33) into (2.25) to obtain

$$\begin{aligned} &\int_{\Omega_x \times \Omega_y} \left(\left(\frac{\partial X_n^p}{\partial x} \right)^2 \cdot Y_n^* \cdot Y_n^p + (X_n^p)^2 \cdot \frac{\partial Y_n^*}{\partial y} \cdot \frac{\partial Y_n^p}{\partial y} \right) \quad dx \cdot dy = \\ -&\int_{\Omega_x \times \Omega_y} \sum_{i=1}^{n-1} \left(\frac{\partial X_n^p}{\partial x} \cdot \frac{\partial X_i}{\partial x} \cdot Y_n^* \cdot Y_i + X_n^p \cdot X_i \cdot \frac{\partial Y_n^*}{\partial y} \cdot \frac{\partial Y_i}{\partial y} \right) \quad dx \cdot dy \\ &- \int_{\Omega_x \times \Omega_y} X_n^p \cdot Y_n^* \cdot f \quad dx \cdot dy + \int_{\Omega_x} X_n^p \cdot Y_n^*(x, y=H) \cdot q \quad dx \end{aligned} \tag{2.34}$$

   Now, all functions of x are known, and we can compute the integrals

$$\begin{cases} \alpha^y &= \int_{\Omega_x} \left(X_n^p(x) \right)^2 \quad dx \\ \beta^y &= \int_{\Omega_x} \left(\frac{\partial X_n^p(x)}{\partial x} \right)^2 \quad dy \\ \gamma_i^y &= \int_{\Omega_x} X_n^p(x) \cdot X_i(x) \quad dx \\ \delta_i^y &= \int_{\Omega_x} \frac{\partial X_n^p(x)}{\partial x} \cdot \frac{\partial X_i(x)}{\partial x} \quad dx \\ \xi^y &= \int_{\Omega_x} X_n^p(x) \cdot f \quad dx \\ \mu^y &= \int_{\Omega_x} X_n^p(x) \cdot q \quad dx \end{cases} \tag{2.35}$$

   We thus obtain the weighted residual form of an elliptic problem for $Y_n^p(y)$

defined over Ω_y:

$$\begin{aligned} &\int_{\Omega_y} \left(\frac{\partial Y_n^*}{\partial y} \cdot \frac{\partial Y_n^p}{\partial y} \cdot \alpha^y + Y_n^* \cdot Y_n^p \cdot \beta^y \right) \quad dy \\ = &- \int_{\Omega_y} \sum_{i=1}^{n-1} \left(\frac{\partial Y_n^*}{\partial y} \cdot \frac{\partial Y_i}{\partial y} \cdot \gamma_i^y + Y_n^* \cdot Y_i \cdot \delta_i^y \right) \quad dy \\ &- \int_{\Omega_y} Y_n^* \cdot \xi^y \quad dy + Y_n^*(y = H) \cdot \mu^y \end{aligned} \tag{2.36}$$

Here again, we can use the finite element method to discretize this one-dimensional problem, with the remaining Dirichlet conditions $Y_n^p(y = 0) = 0$

2.3.1 The Finite Element Method

The Finite Element Method (FEM) is a numerical technique used for finding approximate solutions to boundary value problems for partial differential equations [24].

Let's consider the $x - direction$ problem of finding X_n^p in a discrete one-dimensional mesh of M uniformly distributed elements where h is the distance between the mesh nodes (the $y - direction$ problem is analog). Starting from the weighted residual form of the elliptic problem (2.31), we aim to describe X_n^p as a linear combination of a discrete basis $\{\phi_1, ..., \phi_M\}$ of our one-dimensional function space $H_0^1(\Omega_x)$. We will define ϕ_i to be the hat functions given as follows:

$$\phi_i(x) = \begin{cases} \frac{x - x_{i-1}}{h}, & x_{i-1} \leq x < x_i \\ \frac{x_{i+1} - x}{h}, & x_i \leq x < x_{i+1} \\ 0, \quad \text{otherwise} \end{cases} \tag{2.37}$$

Hence, we get $X_n^p = \sum_{i=1}^M \phi_i x_i$. By inserting it into (2.31) the left side of the equation becomes

$$\int_{\Omega_x} \frac{\partial X_n^*}{\partial x} \cdot \left(\sum_{i=1}^M \phi_i' x_i \right) \cdot \alpha^x + X_n^* \cdot \left(\sum_{i=1}^M \phi_i x_i \right) \cdot \beta^x \quad dx, \tag{2.38}$$

that can be rewritten as

$$\sum_{i=1}^M \left(\alpha^x \int_{\Omega_x} \phi_i' \cdot \frac{\partial X_n^*}{\partial x} \quad dx \cdot x_i \right) + \sum_{i=1}^M \left(\beta^x \int_{\Omega_x} X_n^* \cdot \phi_i \quad dx \cdot x_i \right) \tag{2.39}$$

As the only requirement for the test functions is that it have to belong to H_0^1, we

can take X_n^* as the basis functions ϕ_i and get M equations

$$\begin{gathered}\sum_{i=1}^{M}\left(\alpha^x\int_{\Omega_x}\phi_i'\cdot\phi_1' \quad dx\cdot x_i\right)+\sum_{i=1}^{M}\left(\beta^x\int_{\Omega_x}\phi_1\cdot\phi_i \quad dx\cdot x_i\right)\\ \sum_{i=1}^{M}\left(\alpha^x\int_{\Omega_x}\phi_i'\cdot\phi_2' \quad dx\cdot x_i\right)+\sum_{i=1}^{M}\left(\beta^x\int_{\Omega_x}\phi_2\cdot\phi_i \quad dx\cdot x_i\right)\\ \vdots\\ \sum_{i=1}^{M}\left(\alpha^x\int_{\Omega_x}\phi_i'\cdot\phi_M' \quad dx\cdot x_i\right)+\sum_{i=1}^{M}\left(\beta^x\int_{\Omega_x}\phi_M\cdot\phi_i \quad dx\cdot x_i\right)\end{gathered} \tag{2.40}$$

As

$$\int_{\Omega_x}\phi_j'\cdot\phi_i' \quad dx=\begin{cases}\frac{2}{h}, & \text{if } j=i\\ -\frac{1}{h}, & \text{if } j=i\pm1\\ 0, & \text{otherwise}\end{cases} \tag{2.41}$$

and

$$\int_{\Omega_x}\phi_j\cdot\phi_i \quad dx=\begin{cases}h, & \text{if } j=i\\ 0, & \text{otherwise}\end{cases} \tag{2.42}$$

we can write (2.40) in the matrix form:

$$A=\begin{pmatrix}\frac{2}{h}\alpha^x+h\beta^x & -\frac{1}{h}\alpha^x & 0 & 0 & \cdots & 0\\ -\frac{1}{h}\alpha^x & \frac{2}{h}\alpha^x+h\beta^x & -\frac{1}{h}\alpha^x & 0 & \cdots & 0\\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots\\ 0 & \cdots & 0 & -\frac{1}{h}\alpha^x & \frac{2}{h}\alpha^x+h\beta^x & -\frac{1}{h}\alpha^x\\ 0 & \cdots & 0 & 0 & -\frac{1}{h}\alpha^x & \frac{2}{h}\alpha^x+h\beta^x\end{pmatrix} \tag{2.43}$$

Of the elements on the right-hand side of the equation (2.31), the only one affected by the FEM is the one containing the summation.We want to get rid of the $\frac{\partial X_n^*}{\partial x}$ terms, as we don't know how to compute it against $\frac{\partial X_i}{\partial x}$ and X_i. First, we extract the summation out of the integral as follows

$$\begin{gathered}-\sum_{i=1}^{n-1}\int_{\Omega_x}\left(\frac{\partial X_n^*}{\partial x}\cdot\frac{\partial X_i}{\partial x}\cdot\gamma_i^x+X_n^*\cdot X_i\cdot\delta_i^x\right) \quad dx=\\ -\sum_{i=1}^{n-1}\left(\int_{\Omega_x}\frac{\partial X_n^*}{\partial x}\cdot\frac{\partial X_i}{\partial x}\cdot\gamma_i^x \quad dx+\int_{\Omega_x}X_n^*\cdot X_i\cdot\delta_i^x \quad dx\right),\end{gathered}$$

and, integrating by parts the left integral with $u=\frac{\partial X_i}{\partial x}$ and $\partial v=\frac{\partial X_n^*}{\partial x}$, we obtain

$$-\sum_{i=1}^{n-1}\left(\gamma_i^x\cdot X_n^*\cdot\frac{\partial X_i}{\partial x}-\gamma_i^x\int_{\Omega_x}X_n^*\cdot\frac{\partial^2X_i}{\partial x} \quad dx+\delta\int_{\Omega_x}X_n^*\cdot X_i\right). \tag{2.44}$$

Now we are facing again a linear equation system that can be solved as before, with the Householder QR decomposition. The code of all this FEM method can be found between the lines [194-228] on A.2.

Chapter 3

Applying PGD for Robot Path Planning

The preceding section has presented a simple example application of the resolution of the Poisson equation using PGD where a two dimensional space is decomposed in X and Y. (Chinesta et al., 2013; Chinesta et al., 2014) demonstrate that parameters in a model can be set as additional coordinates when using the PGD approach. In this chapter, a path planning technique is presented where these additional parameters are all the possible combination of the start and target position, and are included in the source term of the Poisson equation (2.1)

3.1 Definition of the Source Term

Until now, we assumed the source term f was constant. Now, we will consider it as a non uniform source term $f(\Omega_{\underline{X}}, \Omega_{\underline{S}}, \Omega_{\underline{T}})$, where $\Omega_{\underline{X}} = \Omega_x \times \Omega_y$, $\Omega_{\underline{S}} = \Omega_{s_x} \times \Omega_{s_y}$ and $\Omega_{\underline{T}} = \Omega_{t_x} \times \Omega_{t_y}$. The start and target points S and T are defined by means of Gaussian models with mean and a variance. In these models, $s = (s_x, s_y)$ and $t = (t_x, t_y)$ are the mean values located in specific points $X = (x, y)$ in each separated space $\Omega_{\underline{S}}, \Omega_{\underline{T}}$ and r is its variance. Gaussian models are used instead of Delta Dirac models because they provide much better results in a PGD-Vademecum than Delta Dirac model, as explained at [4]. In order to define the source term, we must construct this two matrices first:

$$
\begin{aligned}
f(X,S) &= \begin{pmatrix} f(x_1,s_1) & \cdots & f(x_1,s_N) \\ \vdots & \ddots & \vdots \\ f(x_N,s_1) & \cdots & f(x_N,s_N) \end{pmatrix} \\
g(X,T) &= \begin{pmatrix} f(x_1,t_1) & \cdots & f(x_1,t_N) \\ \vdots & \ddots & \vdots \\ f(x_N,t_1) & \cdots & f(x_N,t_N) \end{pmatrix}
\end{aligned}
\tag{3.1}
$$

One can find the creation of the $f(X,S)$ and $g(X;T)$ matrices between the lines [417-452] of A.3 Applying the Single Value Decomposition (SVD) method to these matrices, the result is the decomposition of the source term in the form:

$$
\begin{aligned}
f(X,S) &= \sum_{j=1}^{F} \alpha_j^S \cdot F_j^S(X) \cdot G_j^S(S) \\
g(X,T) &= \sum_{j=1}^{F} \alpha_j^T \cdot F_j^T(X) \cdot G_j^T(T)
\end{aligned}
\tag{3.2}
$$

Thus, the Poisson equation to be solved is of the form:

$$
\Delta u(x,y) = f(X,S) + g(X,T) \tag{3.3}
$$

3.2 Computation of the PGD-Vademecum

For all suitable test functions u^*, we can write the weak form 2.2 as

$$
\int_{\Omega_{\underline{X},\underline{S},\underline{T}}} u^* \cdot (\Delta u - f) \quad d\Omega_{\underline{X},\underline{S},\underline{T}} = 0 \tag{3.4}
$$

where $f = f(X,\underline{S}) + g(X,\underline{T})$. Now, the equation 2.2 reads as

$$
\int_{\Omega_{\underline{X},\underline{S},\underline{T}}} \nabla u^* \cdot \nabla u \quad d\Omega_{\underline{X},\underline{S},\underline{T}} = \int_{\Omega_{\underline{X},\underline{S},\underline{T}}} u^* \cdot f \quad d\Omega_{\underline{X},\underline{S},\underline{T}} - \int_{\Omega_{\underline{X},\underline{S},\underline{T}}} u^*(x, y = \Gamma) \cdot q \quad d\Omega_{\underline{X},\underline{S},\underline{T}} \tag{3.5}
$$

where the solution will take the form

$$
u(X,S,T) = \sum_{i=1}^{N} R_i(X) \cdot W_i(S) \cdot K_i(T) \tag{3.6}
$$

We shall follow then the same steps seen on the previous, building an enriched solution

$$u^{n-1}(X,S,T) = \sum_{i=1}^{n-1} R_i(X) \cdot W_i(S) \cdot K_i(T) \tag{3.7}$$

where each enrichment step is given by

$$u^n = u^{n-1} + R(X) \cdot W(S) \cdot K(T) \tag{3.8}$$

One of the main advantages of PGD is the capability of decompose a high dimensional problem into a combination of rank one functions

$$R(X) \cdot W(S) \cdot K(T) = R_1(x) \cdot R_2(y) \cdot W_1(s_x) \cdot W_2(s_y) \cdot K_1(t_x) \cdot K_2(t_y) \tag{3.9}$$

The test functions u^* live then in the linear space of functions

$$R(X) \cdot W(S) \cdot K^*(T) + R(X) \cdot W^*(S) \cdot K(T) + R^*(X) \cdot W(S) \cdot K(T)$$

where $K^*(T)$ is orthogonal to $K(T)$, $W^*(S)$ is orthogonal to $W(S)$ and $R^*(X)$ is orthogonal to R(X). On [4] an alternating direction algorithm is used to construct the separated representation.

3.3 Our own approach

Setting two additional coordinates for computing the PGD for all possible combinations for the start and target positions brings us a lot of value when working on dynamic environments. But since dynamic environments are not really the goal of this project (could be a nice extension though), and it highly increases the complexity of the algorithm, we will modify this implementation to get our own approach. In fact, we won't need all possible combinations, we only aim to get the path from one single goal to one single target. So, instead of adding two coordinates, we will simply consider the source term as a known non-uniform source $h(x,y)$. Actually, that means to get the appropriate combination of columns from 3.1. Indeed, for a sought source and target points, s_i and t_j, we will compute $h(x,y) = f(X,S)|_{s_i} - g(X,S)|_{t_j}$, where $f(X,S)|_{s_i}$ denotes the $i-th$ column of the matrix $f(X,S)$, corresponding to the source term s_1. Same applies to $g(X,S)|_{t_j}$ for the target point t_j.

Our goal is to obtain a separated representation of h in the form

$$h(x,y) = \sum_{j=1}^{\mathcal{F}} H_k^x(x) \cdot H_j^y(y). \tag{3.10}$$

There are several methods to achieve that, but since PGD is one of the main topics of this work, I find it appropriate to use it also for calculating a separated form approximation of h. In this case, no derivatives are envolved on the algebraic problem of finding $u(x,y)$:

$$u(x,y) = h(x,y), \quad (x,y) \in \Omega = \Omega_x \times \Omega_y. \tag{3.11}$$

The corresponding weighted residual form reads

$$\int_{\Omega_x \times \Omega_y} u* \cdot (u(x,y) - h(x,y)) \quad dx \cdot dy = 0, \quad \forall u^* \in H_0^1(\Omega). \tag{3.12}$$

As is now customary, we shall build an enriched solution as in (2.4) and solve each iteration by means of the alternating direction scheme. First, we compute X_n^p using $u* = X_n^* \cdot Y_n^{p-1}$ by solving

$$\int_{\Omega_x \times \Omega_y} X_n^* \cdot Y_n^{p-1} \cdot (X_n^p \cdot Y_n^{p-1} - h(x,y)) \quad dx \cdot dy = 0, \tag{3.13}$$

and then compute Y_n^p using $u* = Y_n^* \cdot X_n^p$,

$$\int_{\Omega_x \times \Omega_y} X_n^p \cdot Y_n^* \cdot (X_n^p \cdot Y_n^p - h(x,y)) \quad dx \cdot dy = 0. \tag{3.14}$$

The strong forms of (3.13) and (3.14) thus yield

$$X_n^p = \frac{\int_{\Omega_y} Y_n^{p-1} \cdot h \quad dy}{\int_{\Omega_y} (Y_n^{p-1})^2 \quad dy}, \tag{3.15}$$

and

$$Y_n^p = \frac{\int_{\Omega_x} X_n^p \cdot h \quad dx}{\int_{\Omega_x} (X_n^p)^2 \quad dx}. \tag{3.16}$$

Now we must translate it into the discrete analog forms of (3.15) and (3.16). As the points of the mesh are uniformly distributed in both domains Ω_x and Ω_y, via numerical integration we have

$$X_n^p = \frac{H(X,Y)^T \cdot Y_n^{p-1}}{(Y_n^{p-1})^T \cdot Y_n^{p-1}}, \tag{3.17}$$

and

$$Y_n^p = \frac{H(X,Y) \cdot X_n^p}{(X_n^p)^T \cdot X_n^p}, \tag{3.18}$$

respectively, where $H(X,Y)$ is the matrix form of $h(x,y)$. The separated form computation of h can be found between the lines [506-512] of A.3.

Once we have (3.10), following the next notation,

$$\epsilon_j^x = \int_{\Omega_y} Y_n^{p-1}(y) \cdot H_j^y(y) \quad dy, \tag{3.19}$$

it is easy to note that (2.15) has became

$$\begin{aligned} &\int_{\Omega_x} X_n^* \cdot \left(\alpha^x \cdot \frac{\partial^2 X_n^p}{\partial x^2} + \beta^x \cdot X_n^p \right) \quad dx \\ = &- \int_{\Omega_x} X_n^* \cdot \sum_{i=1}^{n-1} \left(\gamma_i^x \cdot \frac{\partial^2 X_i}{\partial x^2} + \delta_i^x \cdot X_i \right) + \int_{\Omega_x} X_n^* \cdot \left(\sum_{j=1}^{\mathcal{F}} \xi_j^x \cdot H_j^x(x) \right) \quad dx. \end{aligned} \tag{3.20}$$

Similarly, with the definition

$$\epsilon_j^y = \int_{\Omega_x} X_n^p(x) \cdot H_j^x(x) \quad dx. \tag{3.21}$$

the equation (2.21) becomes

$$\begin{aligned} &\int_{\Omega_y} Y_n^* \cdot \left(\alpha^y \cdot \frac{\partial^2 Y_n^p}{\partial y^2} + \beta^y \cdot Y_n^p \right) \quad dy \\ = &- \int_{\Omega_y} Y_n^* \cdot \sum_{i=1}^{n-1} \left(\gamma_i^y \cdot \frac{\partial^2 Y_i}{\partial y^2} + \delta_i^y \cdot Y_i \right) + \int_{\Omega_y} Y_n^* \cdot \left(\sum_{j=1}^{\mathcal{F}} \xi_j^y \cdot H_j^y(y) \right) \quad dy. \end{aligned} \tag{3.22}$$

We can transform this expressions into the strong formulation and resolve it via discretization and Householder QR decomposition as we did previously. The implementation of all this process can be found between the lines [212-246] of A.3.

3.4 Numerical results

3.4.1 Basic example

We will start with a basic example of Poisson equation (1.17) of which we know the analytical form. This example was mainly to ensure the PGD algorithm worked correctly. We considered a two-dimensional rectangular domain $\Omega = \Omega_x \times \Omega_y = (0,2) \times (0,1)$. The source term f is constant set to $f = 1$. The exact solution for $u(x,y)$ is:

$$u_{ex}(x,y) = \sum_{\substack{m \geq 1 \\ m \text{ odd}}} \sum_{\substack{n \geq 1 \\ n \text{ odd}}} \frac{64}{\pi^4 nm(4n^2 + m^2)} \sin\left(\frac{m\pi x}{2}\right) \sin(n\pi y). \tag{3.23}$$

The solution is represented with an $M \times M$ grid, where $M = 101$. The error tolerance is $\epsilon = 10^{-6}$ and the maximum iteration for the enrichment step and

alternating direction scheme are $max_n = 20$, $max_p = 30$, respectively. The full code can be found in [A.1]. In Fig 3.1 we show the PGD approximation of the solution of the problem. In Fig 3.2 the error between our PGD approximated solution and the analytical solution 3.23 is shown.

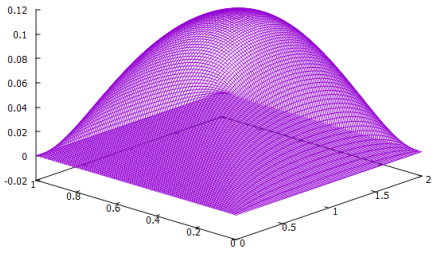

Figure 3.1: Reconstructed PGD solution of (2.1) with $f = 1$

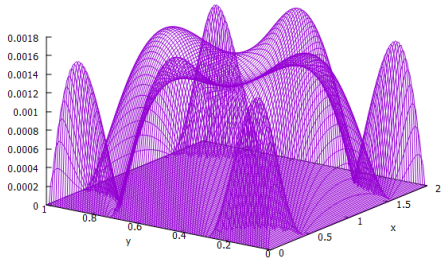

Figure 3.2: Absolute difference between the PGD approximation and the analytical solution 3.23
(computing for $1 \leq m, n \leq 101$)

3.4.2 Computing navigation path

Once we get our PGD algorithm working, the next step is to use it for finding a path between two points. We will use the same values as before for the variables M, max_n and max_p. Now, we also have to consider the maximum iteration for the enrichment step of the PGD computation for the separated form of the source term f, $max_f = 20$. We considered a two-dimensional rectangular domain $\Omega = \Omega_x \times \Omega_y = (0,7) \times (0,5)$. On the section A.3 of the appendix, we can see how the function f is computed using a Gaussian model. Then a separated version is estimated via PGD. The Figure 3.3 shows the result of this estimated version for a source point $S_p = (2,1)$, a target point $T_p = (5,4)$ and a variance $r = 1.2$. The Figure 3.5, on the other hand, shows the solution $u(x,y)$ by applying the just computed f to the Poisson equation. The Figure 3.6 shows the vector field corresponding to the solution u.

As we can see, the source and target points seems a bit displaced. This is due to the Dirichlet boundary condition. Since the flux is forced to disappear on the boundary, the minimum and maximum of the fields are slightly modified. To avoid that, we can modify the variance r value to make the Gaussian distribution more concentrated on the desired points. On the Figure 3.7 we will use a variance

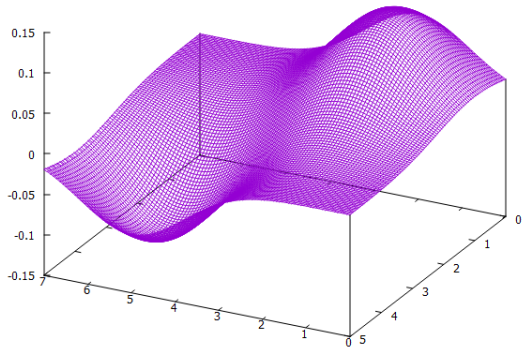

Figure 3.3: Separated form estimation of source term function f for source point $S_p = (2,1)$ and a target point $T_p = (5,4)$

$r = 0.1$ and we can clearly see the difference, and check that now the source and target points keep the desired values. On Figure 3.8 the resulting vector field is plotted.

Finally, we compute the interpolated path using the Euler integration method of the ode $\dot{q} = -\nabla U(q)$ where U is the obtained PGD approximation for the potential. This method approximates the solution by taking small steps along the direction of the gradient. We can see the code in lines [530-563] and the results on the Figure 3.4. Note that Euler method has a local error $\mathcal{O}(h^2)$ but, as we will explain later in 4.3.3, the step size h in the integration process has to be small enough so the default global navigation system of the ROS package we use don't interfere in the robot path.

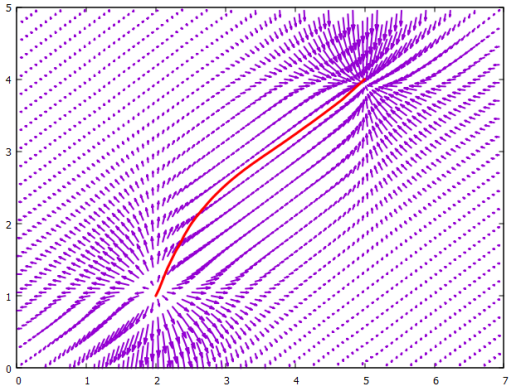

Figure 3.4: Interpolated path from source point $S_p = (2,1)$, target point $T_p = (5,4)$

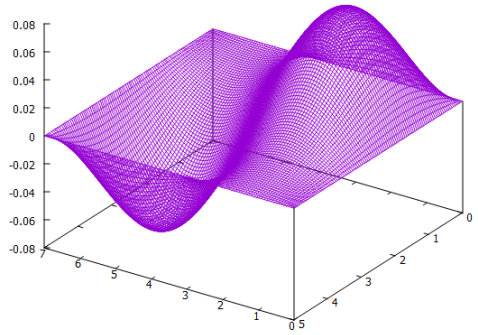

Figure 3.5: PGD estimation of the solution $u(x,y)$ for source point $S_p = (2,1)$, target point $T_p = (5,4)$ and variance $r = 1.2$

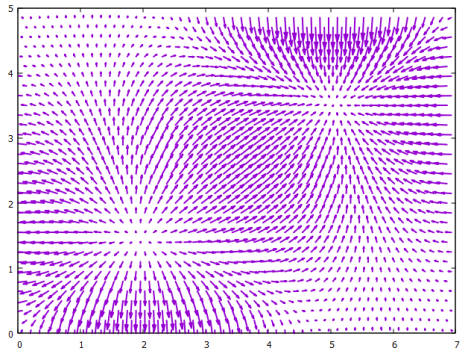

Figure 3.6: Vector field given by the PGD approximation $u(x,y)$ for source point $S_p = (2,1)$, target point $T_p = (5,4)$ and variance $r = 1.2$

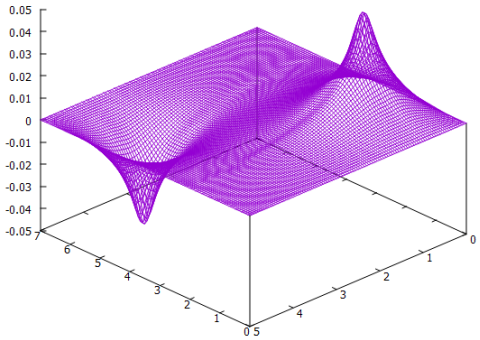

Figure 3.7: PGD approximation of the solution $u(x,y)$ for source point $S_p = (2,1)$, target point $T_p = (5,4)$ and variance $r = 0.1$

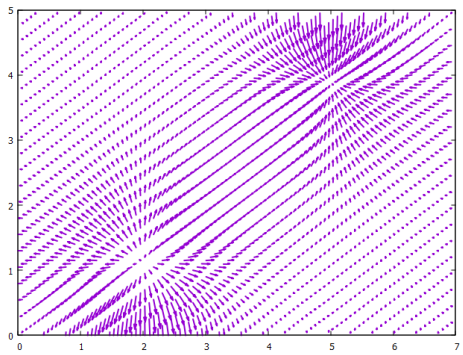

Figure 3.8: Vector field given by the PGD approximation of $u(x,y)$ for source point $S_p = (2,1)$, target point $T_p = (5,4)$ and variance $r = 0.1$

Chapter 4

Building a robot application

Once we already have the path the robot should take, the last step is to give that information to the robot itself. I've decided to use the Robot Operating System (ROS) tool, since it's the framework used in the Robotics subject at *Universitat de Barcelona*, whereby I have at my disposal some repositories and nodes that will help me not to do all the settings from scratch. Moreover, in the event of wanting to continue this work in the future, I can even transfer everything I have done so far to the physical plane, since I also have physical and functional robots at my disposal.

The virtual environment used for developing the ROS application is provided by The Construct AI[1], a free online platform designed to create and simulate robotic applications. One of its advantages is that it runs entirely in the cloud, eliminating the need for installing and configuring ROS and associated tools on your local machine and thus saving us a lot of time. It also has an integrated IDE that includes tools for writing and testing code, visualizing robot movements, and debugging applications.

4.1 Robot Operating System (ROS)

Robot Operating System[2] (ROS) is a flexible open source framework for creating robot software. It provides different tools, libraries and conventions to simplify the task of setting up and creating a complex and robust robot behaviours.

Over its advantages, we can emphasize the modularity and reusability, which will be discussed in the next subsection, and also the communication structure, that enables seamless interaction between different software component in the robot system.

[1] `https://www.theconstruct.ai/`

[2] `https://ros.org/`

4.2 How does ROS work?

Actually, ROS is more than a development framework. We can refer to ROS as a meta-operating system, since it offers not only tools and libraries but even OS-like functions, such as hardware abstraction, package management, and a developer toolchain. Like a real operating system, ROS files are organized on the hard disk in a particular manner. On this section, we will summarize the two main levels of concept ROS has: the Filesystem and the Computational Graph.

4.2.1 ROS Filesystem Level

Similar to an operating system, ROS files are also organized on the hard disk in a particular fashion. In this level, we can see how these files are organized on the disk. The figure 4.1 shows how ROS files and folder are organized on the disk:

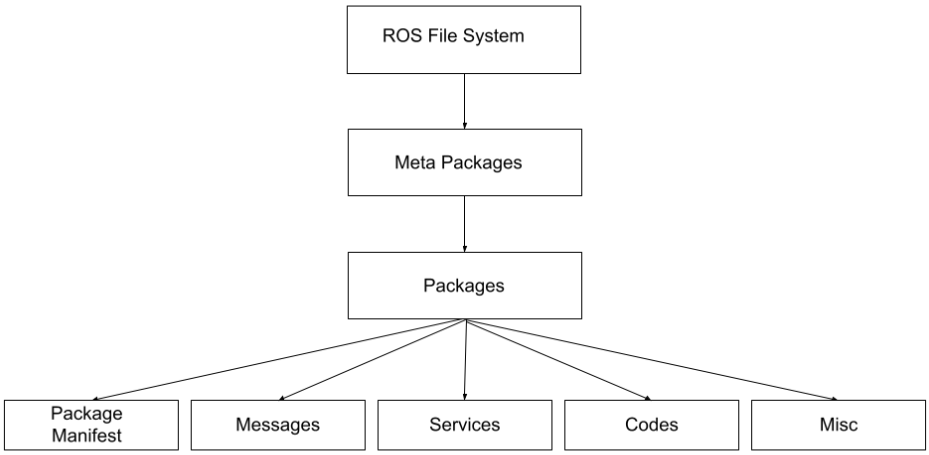

Figure 4.1: Graph representing the ROS file system hierarchy

Here are the explanations for each block in the filesystem:

- **Packages:**

  The ROS packages are the most basic unit of the ROS software. They contain libraries, executables, scripts, configuration files and other resources needed to perform an specific task. Packages are the atomic build item and release item in the ROS software. A typical structure of an ROS package is shown in figure 4.2.

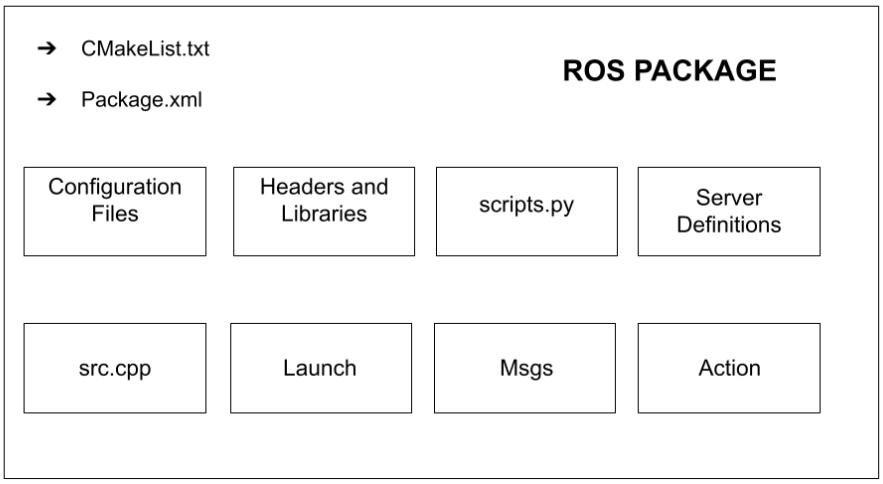

Figure 4.2: Typical structure of a ROS package

- **Metapackages:**

  One or more related packages which can be loosely grouped together. In principle, metapackages are virtual packages that don't contain any source code or typical files usually found in packages. . Most commonly metapackages are used as a backwards compatible place holder for converted rosbuild Stacks.

- **Package manifest:**

  Provide metadata about a package, including its name, version, description, license information, dependencies, and other meta information like exported packages.

- **Repositories:**

  A collection of packages which share a common Version Control System (VCS). Packages which share a VCS share the same version and can be released together using the catkin[3] release automation tool bloom[4]. Catkin is the official build system for ROS and it simplifies the process of building ROS packages by managing dependencies and helping with complex build configurations. On the other hand, bloom is a release automation tool used to prepare ROS packages for release into the ROS ecosystem as a Debian packages.

[3] `https://wiki.ros.org/catkin`
[4] `https://wiki.ros.org/bloom`

- **Message types:**

  Message descriptions, stored in *my_package/msg/MyMessageType.msg*, define the data structures for messages sent in ROS.

- **Services types:**

  Service descriptions, stored in *my_package/srv/MyServiceType.srv*, define the request and response data structures for services in ROS.

4.2.2 ROS Computation Graph Level

The Computation Graph is the peer-to-peer network of ROS processes that are processing data together. This network uses a number of process called ROS nodes. Each concept in the graph is contributed to this graph in different ways. The figure 4.3 exemplifies how the nodes communicate.

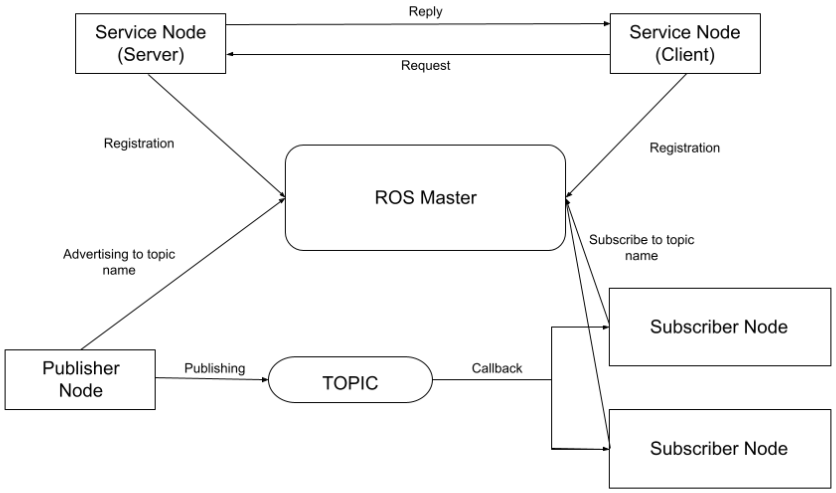

Figure 4.3: Workflow of ROS main communication processes: Topics and Services

- **Nodes:** Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. Each node is an an individual programs or processes that performs computations, handles data, and communicates with other nodes. A ROS node is written with the use of a ROS client library, such as roscpp or rospy.

Using nodes can make the system fault tolerant. Even if a node crashes, an entire robot system can still work. Nodes also reduce the complexity and increase debugability.

- **Master:** The ROS Master provides name registration and lookup to the rest of the nodes. Nodes will not be able to find each other, exchange messages, or invoke services without a ROS Master.

- **Parameter Server:** The parameter server allows you to keep the data to be stored in a central location. All nodes can access and modify these values. Parameter server is a part of ROS Master.

- **Messages:** ROS nodes communicate with each other by publishing messages to a topic. Messages are simply a data structure containing the typed field, which can hold a set of data and that can be sent to another node. There are standard primitive types (integer, floating point, Boolean, and so on) and these are supported by ROS messages. We can also build our own message types using these standard types.

  Nodes can also exchange information using service calls. Services are also messages, the service message definitions are defined inside the srv file.

- **Topics:** Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption. The communication using topics are unidirectional, if we want to implement request/response such as communication, we have to switch to ROS services.

- **Services:** In some robot applications, a publish/subscribe model will not be enough if it needs a request/response interaction. The publish/subscribe model is a kind of one-way transport system and when we work with a distributed system, we might need a request/response kind of interaction.

  Request / reply is done via services, which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply.

- **Bags:** Bags are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, which can be difficult to collect but is necessary for developing and testing robot algorithms. Bags are very useful features when we work with complex robot mechanisms.

4.3 ROS Navigation stack

There are many packages and stacks some of which are used in this project for simulation, kinematic designs and so on, such as Gazevo or RViz. However, we will focus on the navigation stack because is the module which will be directly affected by our previous work of computing the path.

4.3.1 A general view

The Navigation Stack is fairly simple on a conceptual level. It takes in information from odometry and sensor streams and outputs velocity commands to send to a mobile base. Use of the Navigation Stack on an arbitrary robot, however, is a bit more complicated. From a higher point of view, Robot Navigation can be broken down into the following interrelated subproblems, as we can see on the figure 4.4. Each item has his own functionality:

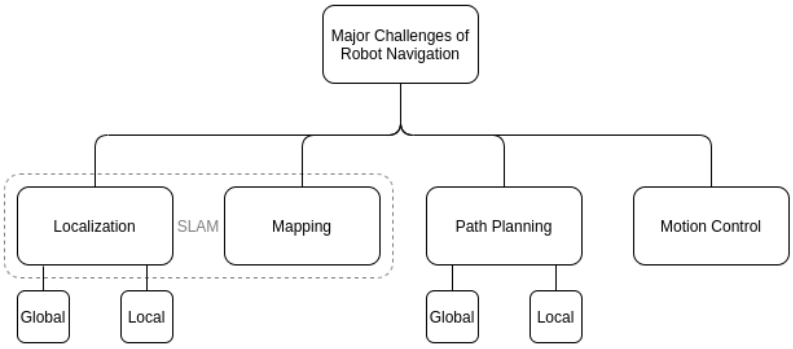

Figure 4.4: The navigation stack

- **Localization:** The robot needs to know where it is.
- **Mapping:** The robot should be able to build a virtual representation of its environment.
- **Path Planning:** The robot needs to be able to plan a route.

- **Motion Control:** The robot has to able to follow a planned route correctly.

4.3.2 Path planning

Depending on the previous knowledge of the environment, path planning can be either online or offline, although sometimes these methods are called static or dynamic planning. In any case, the distinction being made refers to whether the entire path is calculated before the motion begins, with a previously existing map of the environment, or incrementally, during motion using recent sensor information. The path planner created in this projects is an offline path planning, but as it can be extended to compute offline all the possible configurations of the environment, it can also work as an online planner, with the advantage that you do not need to recalculate the path.

Path planning can also be classified into holonomic path planning and non-holonomic path planning, depending on if kinemetic constraints are considered or not. If the generated path also considers constraints on velocity and acceleration, the term kinodynamic path planning is used.

Also, based on the decomposition method of the environment used, path planning algorithms can be classified into deterministic planners and probabilistic planners. Figure 4.5 exemplifies the path planning classification through a graph.

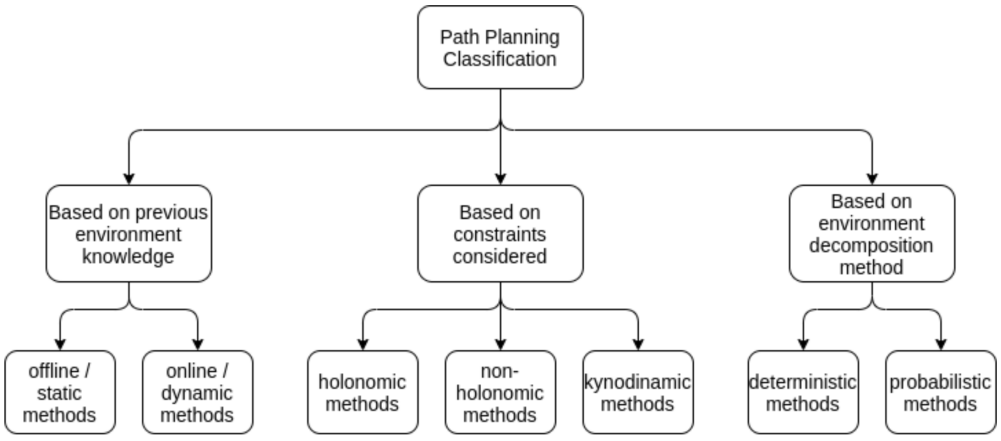

Figure 4.5: Types of path planning classified according to environment knowledge, constraints considered and environment decomposition.

With the many different types of algorithms that exist, there are also many pros and cons to each solution.

But why restrict yourself to using just one path planning method? For instance, both offline and online planning capabilities are very important. Using an existing

map to find the shortest path to a goal is just as valuable as being able to react to unexpected obstacles in that path.

A common way of satisfying the requirements of a robust autonomous navigation system is to use a two-level planning architecture.

In such systems, a global path planner is paired with a local path planner and both work in a complementary manner.

The global path planner is concerned with long range planning and uses the available map information, which can be slow, but is key to finding the most efficient path to a distant goal. It is not concerned with the robot's dynamics or how to avoid unexpected obstacles, which are left to the local path planner.

In this way, each planner deals with only one set of concerns: finding a traversable path to a distant goal, and following that path while reacting to unforeseen situations like the appearance of obstacles.

ROS already provides a local and global navigation using the move_base node[5]. This node links together a global and local path planner via the interfaces of the nav_core node[6], the nav_core::BaseGlobalPlanner and nav_core::BaseLocalPlanner.

On this project, we are going to focus not on replacing the current global path planning but to supply this navigation node a list of destination goals with a step size small enough so the path we computed previously does not get influenced by this interfaces.

A possible future continuation for this project can be swapping this planner already designed by our PGD-computed path planner, so given a goal point the navigation stack gets the path the robot should follow from our PGD algorithm, instead of getting a list of waypoints.

4.3.3 Inside our navigation system

We will follow the same order as in the figure 4.4. So, for the localization and mapping we are using SLAM (Simultaneous Localization and Mapping). SLAM is a technique used in robotics to explore and map an unknown environment while estimating the pose of the robot itself. As it moves all around, it will be acquiring structured information of the surroundings by processing the raw data coming from its sensors. It already exists a SLAM package (`https://wiki.ros.org/slam_gmapping`) so we will only have to modify his configurable parameters in order to improve it's performance.

But, to be able to generate the virtual map, first we need to create the world. In robotics research, always before working with a real robot, we simulate the robot behaviour in a virtual environment close to the real one. Gazebo is an open

[5] `https://wiki.ros.org/move_base`

[6] `https://wiki.ros.org/nav_core`

source 3D robotics simulator that includes an ODE physics engine and OpenGL rendering, and supports code integration for closed-loop control in robot drives. Gazebo has a "Building editor" tool where we can easily generate and save our world. When a model is created with "Building Editor", this path is saved in gazebo environment and you can use it in the future. In this way, you can construct your world adding different models created previously. Figure 4.6 shows the interface Gazebo provides to create and export a custom world.

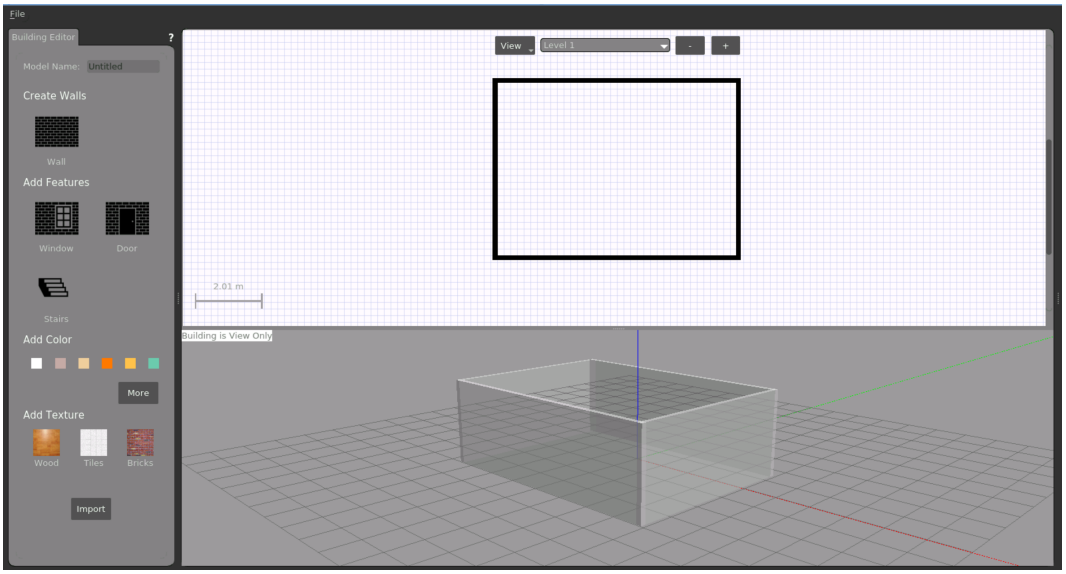

Figure 4.6: Gazebo building editor tool interface.

Once we've created the world we have to spawn the robot into the just designed virtual world. This robot was already designed with the exact same features as the real robot to which we have access. The robot virtual model has been generated with a tool named RVIZ and we can spawn it via *.launch* files. Figure 4.7 shows the robot spawned on the 3D virtual world visualisation using RVIZ. When the robot bring up has been properly carried out, we use the *slam_gmapping* node (`http://wiki.ros.org/gmapping` and the teleoperation package (`https://wiki.ros.org/teleop_twist_keyboard`) in order to move the robot around the world while mapping all the environment. Once the map is finished, it is saved in local directory.

After finished the mapping we want to make the robot go from one initial point to another. To achieve this, the robot needs to know which is its *POSE* within the map. The AMCL[7] (Adaptive Monte Carlo Localization) package provides the amcl node, which uses the MCL system in order to track the localization of a robot

[7] `http://wiki.ros.org/amcl`

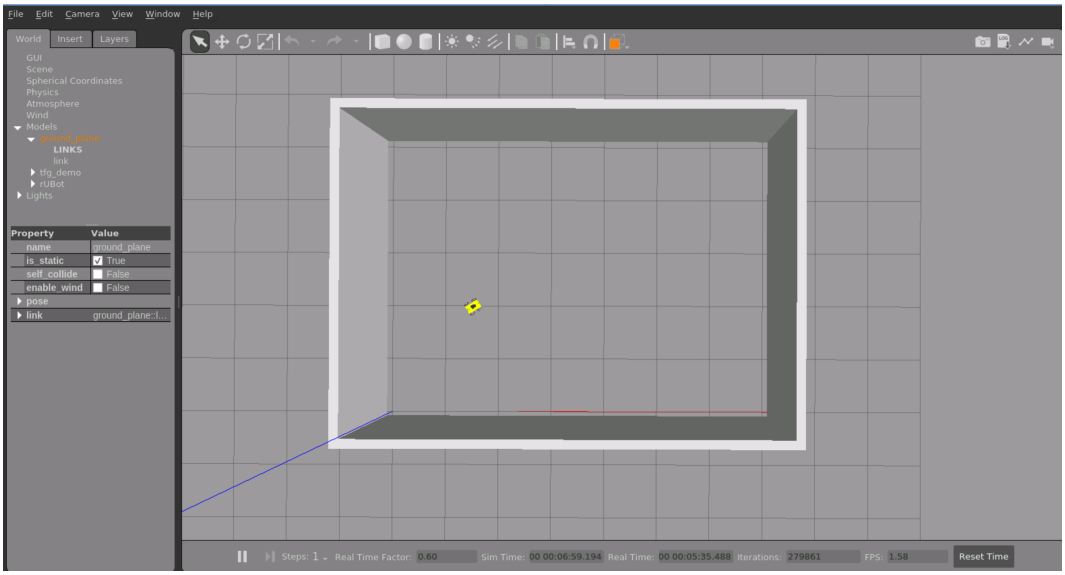

Figure 4.7: RVIz interface where the world and the robot is rendered and visualized.

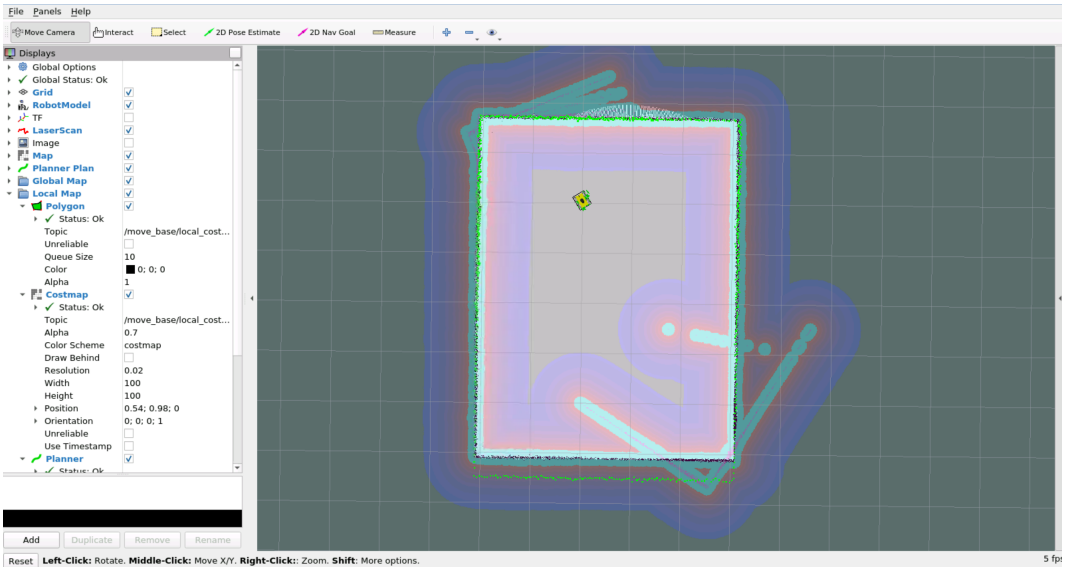

Figure 4.8: Visualization of the virtual map generated by the *slam_gmapping* package..

moving in a 2D space. This node subscribes to the data of the laser, the laser-based map, and the transformations of the robot, and publishes its estimated position in the map. This AMCL node is also highly customizable and we can configure many parameters in order to improve its performance. First, we set up an initial pose by using the 2D Pose Estimate tool (which published that pose to the /initialpose topic). The message type is *"PoseWithCovarianceStamped"*

```
def init_pose():
    rospy.init_node('pub_initpose_node', anonymous=True)
    pub = rospy.Publisher('/initialpose', PoseWithCovarianceStamped,
        queue_size=10)
    #Define whatever initial pose you want, for example (4,3)
    initial_pose = create_initpose(4, 3, radians(180))
    rate = rospy.Rate(1) # 1hz has to be low value
    pub.publish(initial_pose)
    rate.sleep()
```

For sending a goal, we use the move_base ROS Node which itself uses SimpleActionServer, with a single navigation goal. To communicate with this node, the SimpleActionClient interface is used. The message type goal of the pose is *"geometry_msgs/PoseStamped"*. The move_base node tries to achieve a desired pose by combining a global and a local motion planners, as explained in 4.3.2. Since we need to send not a single goal but a sequence of goals which outline the final path, we will use a *.yaml* file to define the waypoints. We have to specify the waypoints as pose in (x,y,w) values and create a new create_pose_stamped(position_x, position_y, rotation_z) function. The "waypoints.yaml" file will shall take the following form:

```
#waypoint.yaml file
goal1: {"x": -0.5, "y": 0.8, "w": 90}
goal2: {"x": -0.5, "y": -0.5, "w": 180}
goal3: {"x": -0.5, "y": -1.3, "w": 180}

...
```

And load this file as a parameters in our *.launch* file to use it in our navigation function:

```
def movebase_client():
    client = actionlib.SimpleActionClient('move_base',MoveBaseAction)
    client.wait_for_server()
    waypoints = []
    with open(rospy.get_param("~waypoints_file")) as file:
```

```
            waypoints_data = yaml.load(file, Loader=yaml.FullLoader)
        # Process loaded waypoints
        for goal_data in waypoints_data.values():
            goal_pose = create_pose_stamped(goal_data['x'],
                goal_data['y'], radians(goal_data['w']))
            waypoints.append(goal_pose)
        #we send a goal for each waypoint
        for wp in waypoints:
            max_attempts = 3
            for attempt in range(max_attempts):
                client.send_goal(wp)
                wait = client.wait_for_result(rospy.Duration(100))
                if wait:
                    rospy.loginfo("Goal execution done!")
                    break # Goal reached successfully, exit loop
                else:
                    rospy.logwarn("Failed to reach goal, retrying...")
```

In this way, the robot will follow the desired path as we are sending the trajectory points with a small spacing, not letting the global path planner already integrated on the move_base node to compute a different flow. On Figure 4.9 we can see how the different nodes communicate with each other. The map_server node provides the virtual map to the move_base which ,in turn, interact with the move_base_waypoints node sending the status position and retrieving back the next goal. It also sends the velocity to the real world environment (gazebo) and fetch the localisation information from the amcl node, as we explained before.

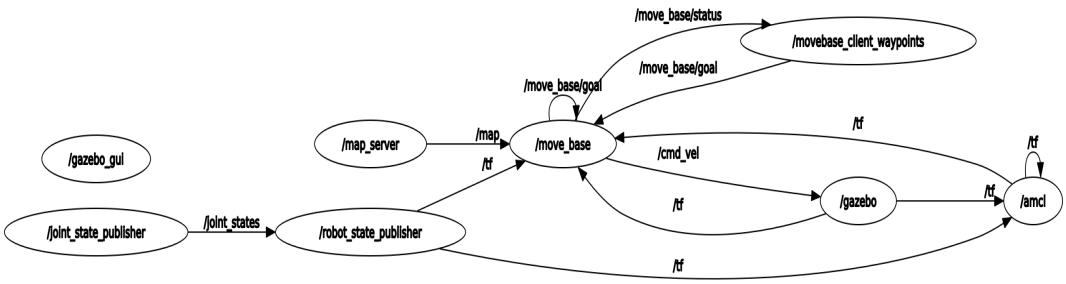

Figure 4.9: ROS computation graph that shows the topics, nodes, and packages used in our project. It is generated via `rqt_graph`(`https://wiki.ros.org/rqt_graph`)

Project Planning

Effective project planning is crucial for the successful execution and completion of any project. It involves defining the project objectives and determining the timeline required. Figure 4.10 is a Gantt chart that reflects the temporal planning throughout the project. There are some nuances that I would like to highlight. As we can see, it has been a priority to separate and parallelise the tasks related to the PGD and the ROS parts. Also, we can clearly see that the implementation of the PGD algorithm has been divided into three part that have been developed linearly: the Dirichlet, the Neumann and the Path Planner. Finally, it may surprise that the investigation of the navigation stack has been realised so late. This is because I wanted to assist to the Robotic class related to this topic, and the agenda for this course was defined this way.

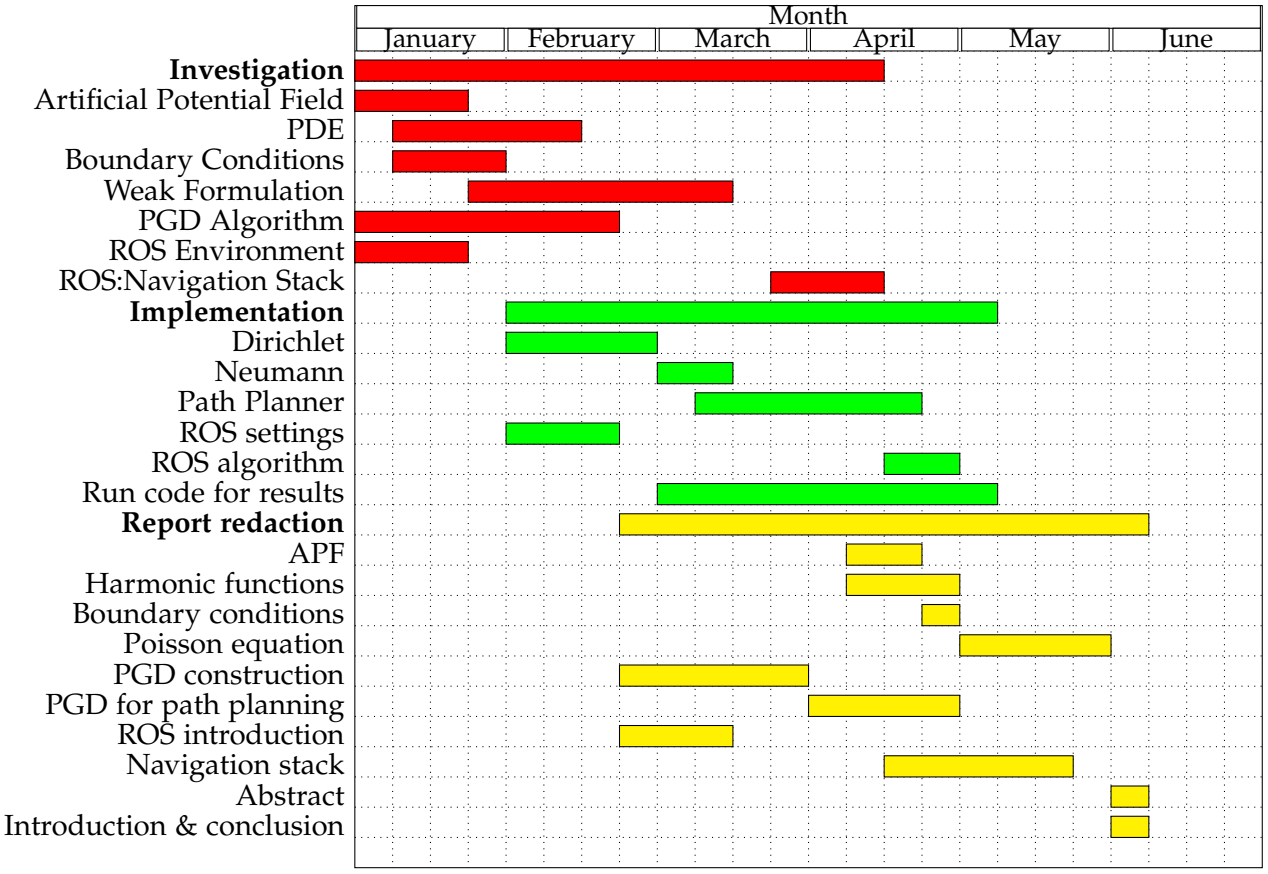

Figure 4.10: **Gantt Chart**. Reflects the temporal planning followed throughout the project making

Conclusion

This project was inspired by an article of MDPI journal [7],which quickly caught my attention as it dealt with two subjects that I had not been able to study throughout my university degree as I mentioned on the abstract. That is why the first objective was to be able to create a model able to find a free-collision path for dynamic environments. It quickly realised that accomplish this while being detailed, formal and rigorous, would take more time than I had available, since a lot of new complex concepts involving different areas have never studied at an advanced level such as physics were introduced. Despite that, restrict the initial conditions to only source and target points, has turned out to be equally interesting and enriching.

In addition to the initial theoretical investigation of the topic, the practical side illustrate the potential of this method for real life applications. Actually, it has been used in multiples areas [11, 15, 23].

The main contributions lie on giving a mathematical foundation to the existing papers about the APF, the application of harmonic function to robotics and the use of PGD for solving Poisson equation [3, 8, 14], which were written from a more engineering perspective. The other fundamental contribution is to illustrate it by means of the building a robot application with the same framework and same robot model as the used in *Universitat de Barcelona*, thus providing an opportunity to continue with this work. It is the author's opinion that this project merges in a very successful way both worlds: mathematics and computer science.

As mentioned on the abstract, this work provides the perfect basis for future extensions in both areas. As for the mathematically oriented part, an interesting future continuation would be integrate dynamic objects to the Poisson equation, as originally intended. On the other hand, as the robotic virtual model of the robot application has been modelled with the same features as the physical ones we have on *Universitat de Barcelona*, translating all this work to the physical realm by leaving the simulation environment and test it on a real environment can be a good experiment.

Appendix

A PGD Code

In this appendix one can find the code used for computing the $u(x, y)$ function by applying the PGD algorithm, as well as the vector field associated and the interpolated path between the source and target points.

A.1 Dirichlet condition

This is the basic example of computing the Poisson equation (2.1) with a constant source term f and considering only Dirichlet boundary conditions.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <Eigen/Dense>
#include <random>

//Settings
using Vector = std::vector<double>;
using Matrix = std::vector<Vector>;




//CONSTANTS
const int M = 101, max_p = 30, max_n = 20; //Mesh points
const double epsilon = 1e-6; //Error tolerance
const Vector f (M,1); // f function
int N = -1; //current iteration of the solution
const int a_y = 0, b_y = 1, a_x = 0, b_x = 2; //Domain

// Create and initialize the matrix with zeros
Matrix createMatrix(int m, int n) {
```

```
23          Matrix matrix(m, Vector(n, 0.0));
24          return matrix;
25      }
26
27      // Print the given matrix
28      void printMatrix(const Matrix& matrix, const std::string& name) {
29          std::cout << "Matrix: " << name << std::endl;
30          for (const auto& row : matrix) {
31              for (double value : row) {
32                  std::cout << value << " ";
33              }
34              std::cout << std::endl;
35          }
36      }
37
38      // Print the given vector
39      void printVector(const Vector& myVector, const std::string& name) {
40          std::cout << "Vector: " << name << std::endl;
41          for (const auto& element : myVector) {
42              std::cout << element << " ";
43          }
44          std::cout << std::endl;
45      }
46
47      // Compute the outer product of two vectors
48      Matrix outerProduct(const Vector& vector1, const Vector& vector2) {
49          // Get the sizes of the vectors
50          size_t size1 = vector1.size();
51          size_t size2 = vector2.size();
52
53          // Create an MxM matrix filled with zeros
54          Matrix resultMatrix(size1, Vector(size2, 0));
55
56          // Compute the outer product
57          for (size_t i = 0; i < size1; ++i) {
58              for (size_t j = 0; j < size2; ++j) {
59                  resultMatrix[i][j] = vector1[i] * vector2[j];
60              }
61          }
62
63          return resultMatrix;
64      }
65
66      // Compute the 1-norm of the given matrix
```

```
67  double computeNorm(const Matrix& matrix) {
68      // Get the dimensions of the matrix
69      size_t rows = matrix.size();
70      size_t cols = matrix[0].size();
71
72      // Initialize the 1-norm to a negative value
73      double norm = -1.0;
74
75      // Iterate over columns and calculate the absolute column sum
76      for (size_t j = 0; j < cols; ++j) {
77          double columnSum = 0.0;
78          for (size_t i = 0; i < rows; ++i) {
79              columnSum += std::abs(matrix[i][j]);
80          }
81          // Update the 1-norm if the current column sum is larger
82          norm = std::max(norm, columnSum);
83      }
84      return norm;
85  }
86
87  // Check if the solution has converged based on a tolerance criterion
88  bool checkSolutionConvergence(const Matrix& X, const Matrix& Y) {
89      if (N == -1) return false;
90      Matrix m1 = outerProduct(X[N], Y[N]);
91      Matrix m2 = outerProduct(X[0], Y[0]);
92
93      return (computeNorm(m1) / computeNorm(m2)) < epsilon;
94  }
95
96  // Compute the integral of a function
97  double computeIntegral(const Vector& function, int a, int b) {
98      double h = static_cast<double>(b - a) / (2.0 * (M - 1));
99      double integral = 0.0;
100     for (int i = 0; i < M; ++i) {
101         if (i == 0 || i == M - 1) {
102             integral += function[i];
103         } else {
104             integral += 2 * function[i];
105         }
106     }
107     return integral * h;
108 }
109
110 // Compute the derivative of a function
```

```
111     Vector computeDerivative(const Vector& function, int a, int b) {
112         Vector der(M, 0.0);
113         double h = static_cast<double>(b - a) / (M - 1);
114         for (int i = 0; i < M; ++i) {
115             if (i == 0) {
116                 der[i] = (function[i] - 2 * function[i + 1] + function[i +
                        2]) / (h * h);
117             } else if (i == M - 1) {
118                 der[i] = (function[i - 1] - 2 * function[i] + function[i +
                        1]) / (h * h);
119             } else {
120                 der[i] = (function[i - 2] - 2 * function[i - 1] +
                        function[i]) / (h * h);
121             }
122         }
123         return der;
124     }
125
126     // Compute the product of two vectors element-wise
127     Vector vectorProduct(const Vector& v1, const Vector& v2) {
128         Vector product(M, 0.0);
129         for (int i = 0; i < M; ++i) {
130             product[i] = v1[i] * v2[i];
131         }
132         return product;
133     }
134
135     // Compute the product of a scalar and a vector
136     Vector scalarProduct(double scalar, const Vector& v1) {
137         Vector product(M, 0.0);
138         for (int i = 0; i < M; ++i) {
139             product[i] = v1[i] * scalar;
140         }
141         return product;
142     }
143
144     // Compute the sum of two vectors
145     Vector vectorSum(const Vector& v1, const Vector& v2) {
146         Vector sum(M, 0.0);
147         for (int i = 0; i < M; ++i) {
148             sum[i] = v1[i] + v2[i];
149         }
150         return sum;
151     }
```

```
152
153     // Compute the difference between two matrices
154     Matrix matrixSub(const Matrix& m1, const Matrix& m2) {
155         Matrix sub = createMatrix(M, M);
156         for (int i = 0; i < M; ++i) {
157             for (int j = 0; j < M; ++j) {
158                 sub[i][j] = m1[i][j] - m2[i][j];
159             }
160         }
161         return sub;
162     }
163
164     // Compute the sum of two matrices
165     Matrix matrixSum(const Matrix& m1, const Matrix& m2) {
166         Matrix sum = createMatrix(M, M);
167         for (int i = 0; i < M; ++i) {
168             for (int j = 0; j < M; ++j) {
169                 sum[i][j] = m1[i][j] + m2[i][j];
170             }
171         }
172         return sum;
173     }
174
175     //Compute the sum used in the EDO
176     Vector computeSum(const Matrix& m1, const Matrix& m2, const Vector
            &previous, int a1, int b1,int a2, int b2){
177         double gamma, delta;
178         Vector suma (M, 0.0f);
179         for (int i = 0; i < N; ++i){
180             gamma = computeIntegral(vectorProduct(previous, m1[i]), a1, b1);
181             delta = computeIntegral(vectorProduct(previous,
                        computeDerivative(m1[i],a1,b1)), a1, b1);
182             suma = vectorSum(suma ,vectorSum(scalarProduct(gamma,
                        computeDerivative(m2[i],a2,b2)) , scalarProduct(delta,
                        m2[i])));
183         }
184         return suma;
185     }
186
187     //Solve the system of equations using the given parameters
188     Vector solveSystem(double alpha, double beta, double xi, const Vector
            &summation,int a, int c ){
189         Eigen::MatrixXd A(M,M);
190         A.setZero();
```

```
191         double h = (c-a)/(double)(M-1);
192         Eigen::VectorXd b(M);
193         b.setConstant(xi);
194         for (int i = 0; i < M ; ++i){
195             if(i==0 or i == M-1){
196                 b(i) = 0.0f;
197             }else{
198                 b(i) -= summation[i];
199             }
200             for (int j = 0; j < M ; ++j){
201                 if(i==j){
202                     if(i == 0 or i == M-1 ){
203                         A(i,j) = 1.0f;
204                     }else{
205                         A(i,j) = -2*alpha/(h*h) + beta;
206                     }
207                 }
208                 else if((i == j-1 or i == j+1) and i != 0 and i!=M-1 ){
209                     if (j!=0 and j != M-1){
210                         A(i,j) = alpha/(h*h);
211                     }
212                 }
213             }
214         }
215         Eigen::VectorXd x = A.colPivHouseholderQr().solve(b);
216         Vector solution(x.data(), x.data() + x.size());
217 
218         return solution;
219 
220     }
221 
222     //Solve the one-dimensional EDO for each alternating direction step
223     Vector computeEDO(Vector &previous, const Matrix& m1,const Matrix& m2,
            int a1, int b1, int a2, int b2){
224         Vector squared_previous;
225         std::transform(previous.begin(), previous.end(),
                std::back_inserter(squared_previous),
226                           [](double x) { return x * x; });
227         double alpha = computeIntegral(squared_previous, a1,b1);
228         double beta = computeIntegral(vectorProduct(previous,
                computeDerivative(previous,a1,b1)),a1,b1);
229         double xi = computeIntegral(vectorProduct(previous, f), a1 , b1);
230         Vector summation = computeSum(m1, m2, previous, a1 , b1, a2, b2);
231 
```

```
232         return solveSystem(alpha, beta, xi, summation, a2,b2);
233
234     }
235
236     //Check the error of the alternating direction iteration step
237     bool checkTolerance(const Vector &current_X, const Vector &current_Y,
            const Vector &previous_X, const Vector &previous_Y){
238         double numerator = computeNorm(matrixSub(outerProduct(current_X,
                current_Y) , outerProduct(previous_X,previous_Y)));
239         double denominator = computeNorm(outerProduct(previous_X,
                previous_Y));
240         return (numerator/denominator) < epsilon;
241     }
242
243     //Generate a random vector to start the alternatingdirection process
244     Vector generateRandomVector(int m, double minVal, double maxVal) {
245         std::random_device rd;
246         std::mt19937 gen(rd());
247         std::uniform_real_distribution<double> dis(minVal, maxVal);
248
249         Vector randomVector;
250         for (int i = 0; i < m; ++i) {
251             double randomValue = dis(gen);
252             randomVector.push_back(randomValue);
253         }
254         return randomVector;
255     }
256
257     //Compute the alternating direction method
258     void alternatingDirection(Matrix& X, Matrix& Y,int iteration) {
259         Vector previous_Y (M, 1.0f);
260         previous_Y[0] = 0.0f;
261         previous_Y[M-1] = 0.0f;
262         Vector current_Y(M, 0.0f);
263         Vector previous_X(M, 0.0f);
264         Vector current_X(M, 0.0f);
265         current_X = computeEDO(previous_Y, Y, X,a_y, b_y, a_x , b_x);
266         current_Y = computeEDO(current_X, X , Y , a_x , b_x,a_y, b_y);
267         previous_X = current_X;
268         int p = 1;
269
270         while(!checkTolerance(current_X, current_Y, previous_X, previous_Y)
                and p < max_p ){
271             previous_Y = current_Y;
```

```
272             previous_X = current_X;
273             current_X = computeEDO(previous_Y, Y, X,a_y, b_y, a_x , b_x);
274             current_Y = computeEDO(current_X, X , Y , a_x , b_x,a_y, b_y);
275             p+=1;
276 
277         }
278 
279         X[N] = current_X;
280         Y[N] = current_Y;
281 
282     }
283 
284     //Export matrix for plotting purposes
285     void exportMatrixToCSV(const Matrix& matrix, const std::string&
        filename) {
286         std::ofstream outFile(filename);
287         double h_x = (b_x - a_x) / (double)(M-1);
288         double h_y = (b_y - a_y) / (double)(M-1);
289         if (!outFile.is_open()) {
290             std::cerr << "Error: Unable to open file " << filename <<
                std::endl;
291             return;
292         }
293 
294         for (size_t j = 0; j < matrix[0].size(); ++j) {
295             for (size_t i = 0; i < matrix.size(); ++i) {
296                 outFile << a_x + i*h_x << " " << a_y + j*h_y << " " <<
                    matrix[i][j]<< "\n";
297             }
298         }
299         outFile.close();
300     }
301 
302     //Compute the gradient fields generated by the solution u(x,y)
303     std::tuple<Matrix, Matrix> computeGradient(const Matrix& function){
304         int rows = function.size();
305         int cols = function[0].size();
306         Matrix gradient_x = createMatrix(rows,cols);
307         Matrix gradient_y = createMatrix(rows,cols);
308 
309         // Compute the gradient using central differences
310         for (int i = 1; i < rows - 1; ++i) {
311             for (int j = 1; j < cols - 1; ++j) {
312                 // Compute partial derivatives with respect to x and y
```

```
313             double df_dx = (function[i][j + 1] - function[i][j - 1]) /
                    2.0; // Central difference for x
314             double df_dy = (function[i + 1][j] - function[i - 1][j]) /
                    2.0; // Central difference for y
315
316             // Assign the derivatives to gradient matrices
317             gradient_x[i][j] = df_dx;
318             gradient_y[i][j] = df_dy;
319         }
320     }
321     return {gradient_x, gradient_y};
322 }
323
324 //Plot the vector field for graphic examples
325 void plotVectorField(const Matrix& matrix){
326     std::tuple<Matrix, Matrix> fields = computeGradient(matrix);
327 }
328
329 int main() {
330     Matrix solution = createMatrix(M, M);
331     Matrix X = createMatrix(max_n, M);
332     Matrix Y = createMatrix(max_n, M);
333
334     //Loop until convergence
335     while(N < (max_n - 1) && !checkSolutionConvergence(X,Y)){
336         N+=1;
337         alternatingDirection(X,Y, N);
338
339     }
340
341     //COMPUTE SOLUTION
342     for (int i = 0; i < N; ++i){
343         solution = matrixSub(solution, outerProduct(X[i],Y[i]));
344     }
345     //Plot vector fields
346     plotVectorField(solution);
347     return 0;
348 }
```

A.2 Neumann condition

This subsection is the continuation of A.1, but adding Neumann boundary condition to the Dirichlet previous ones. On this example, we are still considering a constant source term f.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <Eigen/Dense>
#include <random>

//Settings
using Vector = std::vector<double>;
using Matrix = std::vector<Vector>;

// CONSTANTS
const int M = 101, max_p = 30, max_n = 20; // Mesh points
const double epsilon = 1e-6; // Error tolerance
const double q = 3;
const Vector f (M,0); // f function

// Global variables
int N = -1; // Current iteration of the solution
const int a_y = 0, b_y = 1, a_x = 0, b_x = 1;

// Create and initialize the matrix with zeros
Matrix createMatrix(int m, int n) {
    Matrix matrix(m, Vector(n, 0.0));
    return matrix;
}

// Print the given matrix
void printMatrix(const Matrix& matrix, const std::string& name) {
    std::cout << "Matrix: " << name << std::endl;
    for (const auto& row : matrix) {
        for (double value : row) {
            std::cout << value << " ";
        }
        std::cout << std::endl;
    }
}

// Print the given vector
```

```
40  void printVector(const Vector& myVector, const std::string& name) {
41      std::cout << "Vector: " << name << std::endl;
42      for (const auto& element : myVector) {
43          std::cout << element << " ";
44      }
45      std::cout << std::endl;
46  }
47
48  // Compute the outer product of two vectors
49  Matrix outerProduct(const Vector& vector1, const Vector& vector2) {
50      // Get the sizes of the vectors
51      size_t size1 = vector1.size();
52      size_t size2 = vector2.size();
53
54      // Create an MxM matrix filled with zeros
55      Matrix resultMatrix(size1, Vector(size2, 0));
56
57      // Compute the outer product
58      for (size_t i = 0; i < size1; ++i) {
59          for (size_t j = 0; j < size2; ++j) {
60              resultMatrix[i][j] = vector1[i] * vector2[j];
61          }
62      }
63
64      return resultMatrix;
65  }
66
67  // Compute the 1-norm of the given matrix
68  double computeNorm(const Matrix& matrix) {
69      // Get the dimensions of the matrix
70      size_t rows = matrix.size();
71      size_t cols = matrix[0].size();
72
73      // Initialize the 1-norm to a negative value
74      double norm = -1.0;
75
76      // Iterate over columns and calculate the absolute column sum
77      for (size_t j = 0; j < cols; ++j) {
78          double columnSum = 0.0;
79          for (size_t i = 0; i < rows; ++i) {
80              columnSum += std::abs(matrix[i][j]);
81          }
82          // Update the 1-norm if the current column sum is larger
83          norm = std::max(norm, columnSum);
```

```
84          }
85          return norm;
86      }
87
88      // Check if the solution has converged based on a tolerance criterion
89      bool checkSolutionConvergence(const Matrix& X, const Matrix& Y) {
90          if (N == -1) return false;
91          Matrix m1 = outerProduct(X[N], Y[N]);
92          Matrix m2 = outerProduct(X[0], Y[0]);
93
94          return (computeNorm(m1) / computeNorm(m2)) < epsilon;
95      }
96
97      // Compute the integral of a function
98      double computeIntegral(const Vector& function, int a, int b) {
99          double h = static_cast<double>(b - a) / (2.0 * (M - 1));
100         double integral = 0.0;
101         for (int i = 0; i < M; ++i) {
102             if (i == 0 || i == M - 1) {
103                 integral += function[i];
104             } else {
105                 integral += 2 * function[i];
106             }
107         }
108         return integral * h;
109     }
110
111     // Compute the derivative of a function
112     Vector computeDerivative(const Vector& function, int a, int b) {
113         Vector der(M, 0.0);
114         double h = static_cast<double>(b - a) / (M - 1);
115         for (int i = 0; i < M; ++i) {
116             if (i == 0) {
117                 der[i] = (function[i] - 2 * function[i + 1] + function[i +
                        2]) / (h * h);
118             } else if (i == M - 1) {
119                 der[i] = (function[i - 1] - 2 * function[i] + function[i +
                        1]) / (h * h);
120             } else {
121                 der[i] = (function[i - 2] - 2 * function[i - 1] +
                        function[i]) / (h * h);
122             }
123         }
124         return der;
```

```
125     }
126
127     // Compute the product of two vectors element-wise
128     Vector vectorProduct(const Vector& v1, const Vector& v2) {
129         Vector product(M, 0.0);
130         for (int i = 0; i < M; ++i) {
131             product[i] = v1[i] * v2[i];
132         }
133         return product;
134     }
135
136     // Compute the product of a scalar and a vector
137     Vector scalarProduct(double scalar, const Vector& v1) {
138         Vector product(M, 0.0);
139         for (int i = 0; i < M; ++i) {
140             product[i] = v1[i] * scalar;
141         }
142         return product;
143     }
144
145     // Compute the sum of two vectors
146     Vector vectorSum(const Vector& v1, const Vector& v2) {
147         Vector sum(M, 0.0);
148         for (int i = 0; i < M; ++i) {
149             sum[i] = v1[i] + v2[i];
150         }
151         return sum;
152     }
153
154     // Compute the difference between two matrices
155     Matrix matrixSub(const Matrix& m1, const Matrix& m2) {
156         Matrix sub = createMatrix(M, M);
157         for (int i = 0; i < M; ++i) {
158             for (int j = 0; j < M; ++j) {
159                 sub[i][j] = m1[i][j] - m2[i][j];
160             }
161         }
162         return sub;
163     }
164
165     // Compute the sum of two matrices
166     Matrix matrixSum(const Matrix& m1, const Matrix& m2) {
167         Matrix sum = createMatrix(M, M);
168         for (int i = 0; i < M; ++i) {
```

```
169             for (int j = 0; j < M; ++j) {
170                 sum[i][j] = m1[i][j] + m2[i][j];
171             }
172         }
173         return sum;
174     }
175
176     // Compute the sum used as an EDO parameter
177     Vector computeSum(const Matrix& m1, const Matrix& m2, const Vector&
        previous,const Vector& previous_derivative , int a1, int b1, int
        a2, int b2) {
178         double gamma, delta;
179         Vector test_derivative(M, -(b2-a2)/static_cast<double>(M-1));
180         Vector suma(M, 0.0);
181         Vector integralVector(M, 0.0);
182         for (int i = 0; i < N; ++i) {
183             gamma = computeIntegral(vectorProduct(previous, m1[i]), a1, b1);
184             delta = computeIntegral(vectorProduct(previous_derivative,
                computeDerivative(m1[i], a1, b1)), a1, b1);
185
186             std::fill(integralVector.begin(),
                integralVector.end(),computeIntegral(computeDerivative(computeDerivative(m2[i],a2
                ));
187             suma = vectorSum(suma, vectorSum(scalarProduct(gamma,
                computeDerivative(m2[i], a2, b2)), scalarProduct(-gamma,
                integralVector)));
188             std::fill(integralVector.begin(),
                integralVector.end(),computeIntegral(scalarProduct(delta,
                m2[i]),a2,b2 ));
189             suma = vectorSum(suma, integralVector);
190         }
191         return suma;
192     }
193
194     // Solve the system of equations using the given parameters
195     Vector solveSystem(double alpha, double beta, double xi, Vector&
        summation, int a, int c) {
196         Eigen::MatrixXd A(M, M);
197         Vector xiVector(M, xi);
198         A.setZero();
199         double h = (c - a) / static_cast<double>(M - 1);
200         Eigen::Map<Eigen::VectorXd> b(summation.data(), M);
201         for (int i = 0; i < M; ++i) {
202             if (i == 0 || i == M - 1) {
```

```
203             b(i) = 0.0;
204         } else {
205             b(i) -= computeIntegral(xiVector,a,c);
206         }
207 
208         for (int j = 0; j < M; ++j) {
209             if (i == j) {
210                 if (i == 0 || i == M - 1) {
211                     A(i, j) = 1.0;
212                 } else {
213                     A(i, j) = 2 * alpha / h + beta * h;
214                 }
215             } else if ((i == j - 1 || i == j + 1) && i != 0 && i != M -
                    1) {
216                 if (j != 0 && j != M - 1) {
217                     A(i, j) = -1.0f * alpha / h;
218                 }
219             }
220         }
221     }
222 
223     Eigen::VectorXd x = A.colPivHouseholderQr().solve(b);
224 
225     Vector solution(x.data(), x.data() + x.size());
226 
227     return solution;
228 }
229 
230 // Compute the EDO (Ordinary Differential Equation) for each
        alternating direction step
231 Vector computeEDO(Vector& previous, const Matrix& m1, const Matrix& m2,
        int a1, int b1, int a2, int b2) {
232     Vector squared_previous;
233     Vector previous_derivative = computeDerivative(previous,a1,b1);
234     Vector squared_previous_derivative;
235     std::transform(previous.begin(), previous.end(),
            std::back_inserter(squared_previous),
236                    [](double x) { return x * x; });
237     std::transform(previous_derivative.begin(),
            previous_derivative.end(),
            std::back_inserter(squared_previous_derivative),
238                    [](double x) { return x * x; });
239     double alpha = computeIntegral(squared_previous, a1, b1);
240     double beta = computeIntegral(squared_previous_derivative, a1, b1);
```

```
241         double xi = computeIntegral(vectorProduct(previous, f), a1, b1);
242         double mu = previous[M-1] * q;
243         Vector summation = computeSum(m1, m2, previous,previous_derivative,
                a1, b1, a2, b2);
244 
245         return solveSystem(alpha, beta, xi, summation, a2, b2);
246     }
247 
248     // Check if the tolerance criterion is met
249     bool checkTolerance(const Vector& current_X, const Vector& current_Y,
            const Vector& previous_X, const Vector& previous_Y) {
250         double numerator = computeNorm(matrixSub(outerProduct(current_X,
                current_Y), outerProduct(previous_X, previous_Y)));
251         double denominator = computeNorm(outerProduct(previous_X,
                previous_Y));
252         return (numerator / denominator) < epsilon;
253     }
254 
255     // Generate a random vector to start the alternatingdirection process
256     Vector generateRandomVector(int m, double minVal, double maxVal) {
257             std::random_device rd;
258         std::mt19937 gen(rd());
259         std::uniform_real_distribution<double> dis(minVal, maxVal);
260 
261         Vector randomVector;
262         for (int i = 0; i < m; ++i) {
263             double randomValue = dis(gen);
264             randomVector.push_back(randomValue);
265         }
266         return randomVector;
267     }
268 
269     // Alternating Direction Method to solve the system
270     void alternatingDirection(Matrix& X, Matrix& Y, int iteration) {
271         Vector previous_Y(M, 1.0);
272         previous_Y[0] = 0.0;
273         previous_Y[M - 1] = 0.0;
274         Vector current_Y(M, 0.0);
275         Vector previous_X(M, 0.0);
276         Vector current_X(M, 0.0);
277         current_X = computeEDO(previous_Y, Y, X, a_y, b_y, a_x, b_x);
278         current_Y = computeEDO(current_X, X, Y, a_x, b_x, a_y, b_y);
279         previous_X = current_X;
280         int p = 1;
```

```
281
282         while (!checkTolerance(current_X, current_Y, previous_X,
                previous_Y) && p < max_p) {
283             previous_Y = current_Y;
284             previous_X = current_X;
285             current_X = computeEDO(previous_Y, Y, X, a_y, b_y, a_x, b_x);
286             current_Y = computeEDO(current_X, X, Y, a_x, b_x, a_y, b_y);
287             p += 1;
288         }
289         X[N] = current_X;
290         for (int i = 0; i < M; ++i) {
291             X[N] = current_X;
292             Y[N] = current_Y;
293         }
294     }
295     // Export the matrix to a CSV file
296     void exportMatrixToCSV(const Matrix& matrix, const std::string&
            filename) {
297         std::ofstream outFile(filename);
298         if (!outFile.is_open()) {
299             std::cerr << "Error: Unable to open file " << filename <<
                    std::endl;
300             return;
301         }
302
303         for (size_t j = 0; j < matrix[0].size(); ++j) {
304             for (size_t i = 0; i < matrix.size(); ++i) {
305                 if (i == matrix.size() - 1) {
306                     outFile << matrix[i][j] << "\n";
307                 } else {
308                     outFile << matrix[i][j] << ",";
309                 }
310             }
311         }
312
313         outFile.close();
314     }
315
316     // Compute the gradient fields of the solution u(x,y)
317     std::tuple<Matrix, Matrix> computeGradient(const Matrix& function) {
318         int rows = function.size();
319         int cols = function[0].size();
320         Matrix gradient_x = createMatrix(rows, cols);
321         Matrix gradient_y = createMatrix(rows, cols);
```

```
322
323         // Compute the gradient using central differences
324         for (int i = 1; i < rows - 1; ++i) {
325             for (int j = 1; j < cols - 1; ++j) {
326                 // Compute partial derivatives with respect to x and y
327                 double df_dx = (function[i][j + 1] - function[i][j - 1]) /
                        2.0; // Central difference for x
328                 double df_dy = (function[i + 1][j] - function[i - 1][j]) /
                        2.0; // Central difference for y
329
330                 // Assign the derivatives to gradient matrices
331                 gradient_x[i][j] = df_dx;
332                 gradient_y[i][j] = df_dy;
333             }
334         }
335         return { gradient_x, gradient_y };
336     }
337
338     // Main function
339     int main() {
340         Matrix solution = createMatrix(M, M);
341         Matrix X = createMatrix(max_n, M);
342         Matrix Y = createMatrix(max_n, M);
343
344         // Loop until convergence
345         while (N < (max_n - 1) && !checkSolutionConvergence(X, Y)) {
346             N += 1;
347             alternatingDirection(X, Y, N);
348         }
349
350         // COMPUTE SOLUTION
351         for (int i = 0; i < N; ++i) {
352             solution = matrixSum(solution, outerProduct(X[i], Y[i]));
353         }
354
355         return 0;
356     }
```

A.3 Path Planner

On this final subsection we are computing the path between the source point and the target point. To achieve that, we first model a non-constant source term f by means of Gaussian models. Then, we obtain the separated representation of f applying again the PGD strategy. Later we get the $u(x,y)$ function as explained in 3.3. To conclude, we compute the vector field associated to u and trace the path by interpolating a streamline. The last step is to export this just created path to an *.yaml* file so we can pass it as a parameter to our robot.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <Eigen/Dense>
#include <random>

//SETTINGS
using Vector = std::vector<double>;
using Matrix = std::vector<Vector>;
struct Point {
    double x;
    double y;
};

//CONSTANTS
const int M = 101, max_p = 30, max_n = 20,max_f=50; //Mesh points
const double epsilon = 1e-6; //Error tolerance
const double variance = 0.1; //Variance r of the Gaussian model
Point source, target; //Source and target points
int N = -1, F=-1; //current iteration of the solution
const int a_y = 0, b_y = 5, a_x = 0, b_x = 7; //Domain

//Create and initialise the matrix with zeros
Matrix createMatrix(int m, int n) {
    Matrix matrix(m, Vector(n, 0.0f));
    return matrix;
}

//Print the given matrix
void printMatrix(const Matrix& matrix,const std::string &name ) {
    std::cout << "Matrix: " << name << std::endl;
    for (const auto& row : matrix) {
        for (double value : row) {
```

```
35              std::cout << value << " ";
36          }
37          std::cout << std::endl;
38      }
39  }
40
41  //Print the given vector
42  void printVector(Vector &myVector,const std::string &name){
43      std::cout << "Vector: "<< name << std::endl;
44      for (const auto &element : myVector) {
45          std::cout << element << " ";
46      }
47
48      std::cout << std::endl;
49  }
50
51  //Compute the outer product of two vectors
52  Matrix outerProduct(const Vector& vector1, const Vector& vector2) {
53      // Get the sizes of the vectors
54      size_t size1 = vector1.size();
55      size_t size2 = vector2.size();
56
57      // Create an MxM matrix filled with zeros
58      Matrix resultMatrix(size1, Vector(size2, 0));
59
60      // Compute the outer product
61      for (int i = 0; i < M; ++i) {
62          for (int j = 0; j < M; ++j) {
63              resultMatrix[i][j] = vector1[i] * vector2[j];
64          }
65      }
66      return resultMatrix;
67  }
68
69  //Compute the 1-norm of the given matrix
70  double computeNorm(const Matrix& matrix) {
71      // Get the dimensions of the matrix
72      size_t rows = matrix.size();
73      size_t cols = matrix[0].size();
74
75      // Initialize the 1-norm to a negative value
76      double norm = -1.0f;
77
78      // Iterate over columns and calculate the absolute column sum
```

```
79          for (int j = 0; j < M; ++j) {
80              double columnSum = 0.0f;
81              for (int i = 0; i < M; ++i) {
82                  columnSum += std::abs(matrix[i][j]);
83              }
84              // Update the 1-norm if the current column sum is larger
85              norm = std::max(norm, columnSum);
86          }
87          return norm;
88      }
89
90      //Check if the solution has converged based on the stopping criterion
91      bool checkSolutionConvergence(const Matrix& X, const Matrix& Y) {
92          if(N==-1)return false;
93          Matrix m1 = outerProduct(X[N],Y[N]);
94          Matrix m2 = outerProduct(X[0],Y[0]);
95
96          return (computeNorm(m1) / computeNorm(m2)) < epsilon;
97      }
98
99      //Compute the discrete integral of a function
100     double computeIntegral(const Vector &function, int a, int b) {
101         double h = static_cast<float>(b - a) / (double)(2.0f *(M-1));
102         double integral = 0.0f;
103         for (int i = 0; i < M; ++i){
104             if (i == 0 or i == M-1){
105                 integral += function[i];
106             }else{
107                 integral += 2 * function[i];
108             }
109
110         }
111         return integral * h;
112     }
113
114     //Compute the derivative of a function
115     Vector computeDerivative(const Vector &function, int a, int b){
116         Vector der(M, 0.0f);
117         double h = static_cast<float>(b - a)/(M-1);
118         for (int i = 0; i < M; ++i){
119             if ( i == 0 ){
120                 der[i] = (function[i] - 2*function[i + 1] + function[i + 2]
                        ) / (h*h);
121             }else if(i == M-1){
```

```
122             der[i] = (function[i-1] - 2*function[i] + function[i + 1] )
                    / (h*h);
123         }else{
124             der[i] = (function[i-2] - 2*function[i-1] + function[i] ) /
                    (h*h);
125         }
126     }
127     return der;
128 }
129
130 //Compute the product of two vectors element-wise
131 Vector vectorProduct(const Vector &v1,const Vector &v2 ) {
132     Vector product(M, 0.0f);
133     for (int i = 0; i < M; ++i){
134         product[i] = v1[i] * v2[i];
135
136     }
137     return product;
138 }
139
140 //Compute the scalar product of two vectors
141 double scalarVectorProduct(const Vector& v1, const Vector& v2) {
142     double result;
143     for (int i = 0; i < M; ++i) {
144         result += v1[i] * v2[i];
145     }
146     return result;
147 }
148
149 //Compute the product of an scalar and a vector
150 Vector scalarProduct(double scalar,const Vector &v1) {
151     Vector product(M, 0.0f);
152     for (int i = 0; i < M; ++i){
153         product[i] = v1[i] * scalar;
154
155     }
156     return product;
157 }
158
159 //Compute the sum of two vectors
160 Vector vectorSum(const Vector &v1,const Vector &v2) {
161     Vector sum(M, 0.0f);
162     for (int i = 0; i < M; ++i){
163         sum[i] = v1[i] + v2[i];
```

```
164
165         }
166         return sum;
167     }
168
169     //Compute the difference between two vectors
170     Vector vectorSub(const Vector& v1, const Vector& v2) {
171         Vector sum(M, 0.0);
172         for (int i = 0; i < M; ++i) {
173             sum[i] = v1[i] - v2[i];
174         }
175         return sum;
176     }
177
178     //Compute the difference between two matrices
179     Matrix matrixSub(const Matrix& m1,const Matrix& m2) {
180         Matrix sub = createMatrix(M,M);
181         for (int i = 0; i < M; ++i){
182             for(int j = 0; j < M; ++j){
183                 sub[i][j] = m1[i][j] - m2[i][j];
184             }
185         }
186         return sub;
187     }
188
189     //Compute the sum of two matrices
190     Matrix matrixSum(const Matrix& m1,const Matrix& m2) {
191         Matrix sum = createMatrix(M,M);
192         for (int i = 0; i < M; ++i){
193             for(int j = 0; j < M; ++j){
194                 sum[i][j] = m1[i][j] + m2[i][j];
195             }
196         }
197         return sum;
198     }
199
200     //Compute the sum element of the EDO
201     Vector computeSum(const Matrix& m1, const Matrix& m2, const Vector
        &previous, int a1, int b1,int a2, int b2){
202         double gamma, delta;
203         Vector suma (M, 0.0f);
204         for (int i = 0; i < N; ++i){
205             gamma = computeIntegral(vectorProduct(previous, m1[i]), a1, b1);
206             delta = computeIntegral(vectorProduct(previous,
```

```
                computeDerivative(m1[i],a1,b1)), a1, b1);
207             suma = vectorSum(suma ,vectorSum(scalarProduct(gamma,
                    computeDerivative(m2[i],a2,b2)) , scalarProduct(delta,
                    m2[i])));
208         }
209         return suma;
210     }
211
212     //Solve the system of equations using the given parameters
213     Vector solveSystem(double alpha, double beta, const Vector& xi, const
            Vector &summation,int a, int c ){
214         Eigen::MatrixXd A(M,M);
215         A.setZero();
216         double h = (c-a)/(double)(M-1);
217         Eigen::VectorXd b(M);
218         b.setConstant(0);
219         for (int i = 0; i < M ; ++i){
220             if(i==0 or i == M-1){
221                 b(i) = 0.0f;
222             }else{
223                 b(i) += xi[i] - summation[i];
224             }
225             for (int j = 0; j < M ; ++j){
226                 if(i==j){
227                     if(i == 0 or i == M-1 ){
228                         A(i,j) = 1.0f;
229                     }else{
230                         A(i,j) = -2*alpha/(h*h) + beta;
231                     }
232                 }
233                 else if((i == j-1 or i == j+1) and i != 0 and i!=M-1 ){
234                     if (j!=0 and j != M-1){
235                         A(i,j) = alpha/(h*h);
236                     }
237                 }
238             }
239         }
240         Eigen::VectorXd x = A.colPivHouseholderQr().solve(b);
241
242         Vector solution(x.data(), x.data() + x.size());
243
244         return solution;
245
246     }
```

```
247
248     //Solve the one-dimensional EDO for each alternating direction step
249     Vector computeEDO(Vector &previous, const Matrix& m1,const Matrix& m2,
250                                 const Matrix& function1,const Matrix&
                                        function2, int a1, int b1, int a2, int
                                        b2){
251         Vector squared_previous;
252         std::transform(previous.begin(), previous.end(),
                std::back_inserter(squared_previous),
253                         [](double x) { return x * x; });
254         double alpha = computeIntegral(squared_previous, a1,b1);
255         double beta = computeIntegral(vectorProduct(previous,
                computeDerivative(previous,a1,b1)),a1,b1);
256         Vector vectorXi(M, 0.0);
257         for(int i = 0; i < F; ++i){
258             vectorXi = vectorSum(vectorXi,
                    scalarProduct(computeIntegral(vectorProduct(previous,
                    function2[i]),a1,b1),function1[i]));
259         }
260         Vector summation = computeSum(m1, m2, previous, a1 , b1, a2, b2);
261
262         return solveSystem(alpha, beta, vectorXi, summation, a2,b2);
263
264
265     }
266
267     //Check the error of the alternating direction step
268     bool checkTolerance(const Vector &current_X, const Vector &current_Y,
            const Vector &previous_X, const Vector &previous_Y){
269         double numerator = computeNorm(matrixSub(outerProduct(current_X,
                current_Y) , outerProduct(previous_X,previous_Y)));
270         double denominator = computeNorm(outerProduct(previous_X,
                previous_Y));
271         return (numerator/denominator) < epsilon;
272     }
273
274     //Generate a random vector to start the alternating direction process
275     Vector generateRandomVector(int m, double minVal, double maxVal) {
276         std::random_device rd;
277         std::mt19937 gen(rd());
278         std::uniform_real_distribution<double> dis(minVal, maxVal);
279
280         Vector randomVector;
281         for (int i = 0; i < m; ++i) {
```

```
282             double randomValue = dis(gen);
283             randomVector.push_back(randomValue);
284         }
285         return randomVector;
286     }
287 
288     //Execute the alternating direction process
289     void alternatingDirection(Matrix& X, Matrix& Y,
290                               Matrix& f_x,Matrix& f_y,int iteration) {
291         Vector previous_Y = generateRandomVector(M,-3,3);
292         previous_Y[0] = 0.0f;
293         previous_Y[M-1] = 0.0f;
294         Vector current_Y(M, 0.0f);
295         Vector previous_X(M, 0.0f);
296         Vector current_X(M, 0.0f);
297         current_X = computeEDO(previous_Y, Y, X,f_x,f_y, a_y, b_y, a_x,
                b_x);
298         current_Y = computeEDO(current_X, X, Y,f_y,f_x, a_x, b_x, a_y, b_y);
299         previous_X = current_X;
300         int p = 1;
301 
302         while(!checkTolerance(current_X, current_Y, previous_X, previous_Y)
                and p < max_p ){
303             previous_Y = current_Y;
304             previous_X = current_X;
305             current_X = computeEDO(previous_Y, Y, X,f_x,f_y, a_y, b_y, a_x,
                    b_x);
306             current_Y = computeEDO(current_X, X, Y,f_y,f_x, a_x, b_x, a_y,
                    b_y);
307             p+=1;
308 
309         }
310         X[N] = current_X;
311         Y[N] = current_Y;
312     }
313 
314     //Compute product between a matrix and a vector
315     Vector productMatrixVector(Matrix& matrix,Vector& vector, bool
            transposed){
316         Vector result(M, 0.0);
317         for (int i = 0; i < M; ++i) {
318             for (int j = 0; j < M; ++j) {
319                 if (transposed){
320                     result[i] += matrix[j][i] * vector[j];
```

```
321             }else{
322                 result[i] += matrix[i][j] * vector[j];
323             }
324         }
325     }
326     return result;
327 }
328
329 //Compute the adittion of the previous computed steps of the
        alternating direction strategy
330 Vector sumPrevious( Matrix& previousMatrixA,Matrix&
        previousMatrixB,Vector& previousVector ){
331     Vector result(M, 0.0);
332     for(int i = 0; i < F; ++i){
333         result = vectorSum(result,
                scalarProduct(scalarVectorProduct(previousVector,previousMatrixB[i]),
                previousMatrixA[i]));
334     }
335     return result;
336 }
337
338 //ompute the alternating direction matrices for the source term f
339 void alternatingDirectionSourceTerm(Matrix& function,Matrix& X, Matrix&
        Y, int iteration){
340     Vector previous_Y(M, 1.0);
341     previous_Y[0] = 0.0;
342     previous_Y[M - 1] = 0.0;
343     Vector current_Y(M, 0.0);
344     Vector previous_X(M, 0.0);
345     Vector current_X(M, 0.0);
346     current_X = scalarProduct(1.0 /
            scalarVectorProduct(previous_Y,previous_Y), vectorSub(
            productMatrixVector(function,
            previous_Y,false),sumPrevious(X,Y,previous_Y)));
347     current_Y = scalarProduct(1.0 /
            scalarVectorProduct(current_X,current_X),
348     vectorSub(productMatrixVector(function,current_X,true),sumPrevious(Y,X,current_X)));
349     previous_X = current_X;
350     int p = 1;
351     while (!checkTolerance(current_X, current_Y, previous_X,
            previous_Y) && p < max_p) {
352         previous_Y = current_Y;
353         previous_X = current_X;
354         current_X = scalarProduct(1.0 /
```

```
                scalarVectorProduct(previous_Y,previous_Y), vectorSub(
                productMatrixVector(function,
                previous_Y,false),sumPrevious(X,Y,previous_Y) ));
355             current_Y = scalarProduct(1.0 /
                    scalarVectorProduct(current_X,current_X),vectorSub(productMatrixVector(
356             function, current_X,true),
357             sumPrevious(Y,X,current_X)));
358             p += 1;
359         }
360         X[F] = current_X;
361         Y[F] = current_Y;
362
363     }
364
365     //Gaussian Model for a particular point, a mean and a variance
366     double gaussian2D(double x, double y, double mean_x, double mean_y) {
367         double exponent = -((x - mean_x) * (x - mean_x) / (2 * variance *
                variance) +
368                             (y - mean_y) * (y - mean_y) / (2 * variance *
                                variance));
369
370         return exp(exponent) / (2 * M_PI * variance * variance);
371     }
372
373     //Compute an specific ource term modelled by the Gaussian distribution
374     void computeUniqueF(Matrix& matrix){
375         double h_x = (b_x - a_x) / (double)(M-1);
376         double h_y = (b_y - a_y) / (double)(M-1);
377         std::cout << "Enter X-component of the Source: \n";
378         std::cin >> source.x;
379
380         std::cout << "Enter Y component of the Source: \n";
381         std::cin >> source.y;
382         std::cout << "\n\nYou entered: (" << source.x << "," << source.y<<
                ")" << std::endl;
383         // Ask the user to input values
384         std::cout << "Enter X component of the Target: \n";
385         std::cin >> target.x;
386
387         std::cout << "Enter Y component of the Target: \n";
388         std::cin >> target.y;
389
390         std::cout << "\n\nYou entered: (" << target.x << "," << target.y<<
                ")" << std::endl;
```

```
391         for(int i = 0; i < M; ++i){
392             for (int j = 0; j < M; ++j) {
393                 matrix[i][j] = gaussian2D(a_x + i*h_x,a_y +
                        j*h_y,source.x,source.y) -
394                             gaussian2D(a_x + i*h_x,a_y + j*h_y,target.x,target.y);
395             }
396         }
397     }
398
399     //Compute the source term for all possible combinations of source terms
400     void computeF(Matrix& matrix) {
401         int x,y,s1,s2;
402         double h_x = (b_x - a_x) / (double)(M-1);
403         double h_y = (b_y - a_y) / (double)(M-1);
404         for (int i = 0; i < M*M; ++i) {
405             for (int j = 0; j < M*M; ++j) {
406                 x = j / M;
407                 y = j % M;
408
409                 s1 = i / M;
410                 s2 = i % M;
411
412                 matrix[i][j] = gaussian2D(a_x + x*h_x,a_y + y*h_y, a_x +
                        s1*h_x,a_y + s2*h_y );
413             }
414         }
415     }
416
417     //Compute the two matrices F and G of all possible goals and target
            combinations
418     void  computeFlow(Matrix& function) {
419         Matrix F = createMatrix(M*M, M*M);
420         Matrix G = createMatrix(M*M, M*M);
421
422         // Ask the user to input values
423         std::cout << "Enter X component of the Source: \n";
424         std::cin >> source.x;
425
426         std::cout << "Enter Y component of the Source: \n";
427         std::cin >> source.y;
428         std::cout << "\n\nYou entered: " << source.x << " and " << source.y
                << std::endl;
429         // Ask the user to input values
430         std::cout << "Enter X component of the Target: \n";
```

```
431         std::cin >> target.x;
432 
433         std::cout << "Enter Y component of the Target: \n";
434         std::cin >> target.y;
435 
436         std::cout << "\n\nYou entered: " << target.x << " and " << target.y
                << std::endl;
437         computeF(F);
438         computeF(G);
439 
440         int s_position = source.x * M + source.y;
441         int t_position = target.x * M + target.y;
442         int x,y;
443 
444         // Loop over the matrix
445         double h_x = (b_x - a_x) / (double)(M-1);
446         double h_y = (b_y - a_y) / (double)(M-1);
447         for (int i = 0; i < M*M; ++i){
448             x = i / M;
449             y = i % M;
450             function[x][y] = F[s_position][i] - G[t_position][i];
451         }
452     }
453 
454     //Export a given matrix to .csv
455     void exportMatrixToCSV(const Matrix& matrix, const std::string&
            filename) {
456         std::ofstream outFile(filename);
457         double h_x = (b_x - a_x) / (double)(M-1);
458         double h_y = (b_y - a_y) / (double)(M-1);
459         if (!outFile.is_open()) {
460             std::cerr << "Error: Unable to open file " << filename <<
                    std::endl;
461             return;
462         }
463 
464         // Transpose the matrix while writing to the file
465         for (size_t j = 0; j < matrix[0].size(); ++j) {
466             for (size_t i = 0; i < matrix.size(); ++i) {
467                 outFile << (a_x + i*h_x) << " " << (a_y + j*h_y) << " " <<
                        matrix[i][j]/10 <<"\n";
468             }
469             outFile <<"\n";
470         }
```

```
471
472         outFile.close();
473     }
474
475     //Compute the gradient fields generated by the solution u(x,y)
476     std::pair<Matrix, Matrix> computeGradient(const Matrix& function){
477         double h_x = (b_x - a_x) / (double)(M-1);
478         double h_y = (b_y - a_y) / (double)(M-1);
479         Matrix gradient_x = createMatrix(M,M);
480         Matrix gradient_y = createMatrix(M,M);
481
482         // Compute the gradient using central differences
483         for (int i = 1; i < M-1; ++i) {
484             for (int j = 1; j < M-1; ++j) {
485                 // Compute partial derivatives with respect to x and y
486                 double df_dx = (function[i - 1][j] - function[i + 1][j]) /
                        (2.0*h_x); // Central difference for x
487                 double df_dy = (function[i][j - 1] -function[i][j + 1] ) /
                        (2.0*h_y); // Central difference for y
488
489                 // Assign the derivatives to gradient matrices
490                 gradient_x[i][j] = df_dx;
491                 gradient_y[i][j] = df_dy;
492                 double norm = std::sqrt(df_dx * df_dx + df_dy * df_dy);
493             }
494         }
495         return std::make_pair(gradient_x, gradient_y);
496     }
497
498     //Unify the separated form of the source term for testing purposes
499     void unifyFunction(Matrix& function,Matrix& f_x,Matrix& f_y){
500         for (int i = 0; i < F ; ++i){
501             function = matrixSum(function, outerProduct(f_x[i], f_y[i]));
502         }
503
504     }
505
506     //Get the separated form of the non-constant source term
507     void separateF(Matrix& function,Matrix& f_x, Matrix& f_y ){
508         while (F < (max_f - 1) && !checkSolutionConvergence(f_x, f_y)) {
509             F += 1;
510             alternatingDirectionSourceTerm(function,f_x, f_y, F);
511         }
512     }
```

```
513
514     //Calculate the distance of two 2-D points
515     double calculateDistance(const Point& P1, const Point& P2) {
516         return std::sqrt((P2.x - P1.x) * (P2.x - P1.x) + (P2.y - P1.y) *
                (P2.y - P1.y));
517     }
518
519     //Get the first point of the path
520     Point findPointAtDistance(const Point& P1, const Point& P2, double h) {
521         double distance = calculateDistance(P1, P2);
522         double unitVectorX = (P2.x - P1.x) / distance;
523         double unitVectorY = (P2.y - P1.y) / distance;
524         Point P3;
525         P3.x = P1.x + h * unitVectorX;
526         P3.y = P1.y + h * unitVectorY;
527         return P3;
528     }
529
530     //Interpolate the path
531     std::vector<Point> interpolatePath(const Matrix& gradient_x, const
            Matrix& gradient_y, Point start, double step_size, int num_steps) {
532         std::vector<Point> path;
533         path.push_back({source.x,source.y});
534         path.push_back(start);
535
536         double x = start.x;
537         double y = start.y;
538         double h_x = (b_x - a_x) / (double)(M-1);
539         double h_y = (b_y - a_y) / (double)(M-1);
540
541         for (int step = 0; step < num_steps; ++step) {
542             int i = static_cast<int>(x / h_x);
543             int j = static_cast<int>(y / h_y);
544
545             if (i < 0 || i >= M || j < 0 || j >= M) {
546                 break; // Out of bounds
547             }
548
549             double dx = gradient_x[i][j];
550             double dy = gradient_y[i][j];
551
552             x += step_size * dx;
553             y += step_size * dy;
554
```

```
555             path.push_back({x, y});
556             if (std::sqrt((x - target.x) * (x - target.x) + (y - target.y) *
                    (y - target.y)) < step_size) {
557                 path.push_back({target.x, target.y});
558                 break;
559             }
560         }
561
562         return path;
563     }
564
565     //Export the path to an .yml file
566     void exportPath(std::vector<Point> path) {
567         std::ofstream outFile("waypoints.yaml");
568         if (!outFile.is_open()) {
569             std::cerr << "Error: Unable to open file " << "vectorfield" <<
                    std::endl;
570             //return;
571         }
572         for (size_t i = 0; i < path.size(); ++i) {
573             outFile <<"goal"<< i <<": {\"x\":"<< path[i].x << ", \"y\": " <<
                    path[i].y << ", \"w\": 90}"<< "\n";
574
575         }
576         outFile.close();
577
578     }
579
580     //Plot vector field for graphic examples
581     void plotVectorField(const Matrix& matrix){
582         std::pair<Matrix, Matrix> fields = computeGradient(matrix);
583         // Define the step size and the first point
584         double stepSize = 0.5;
585         Point firstPoint = findPointAtDistance(source,target,stepSize);
586
587         int maxIterations = 300;
588         std::vector<Point> path = interpolatePath(fields.first,
                fields.second, firstPoint, stepSize, maxIterations);
589         //Export our computed path to .yaml file to work with ROS
590         exportPath(path);
591
592     }
593
594     int main() {
```

```
595
596         Matrix function = createMatrix(M, M);
597         Matrix new_function = createMatrix(M, M);
598         computeUniqueF(function);
599         Matrix f_x = createMatrix(M,M);
600         Matrix f_y = createMatrix(M,M);
601         separateF(function,f_x,f_y);
602         //Test the new function is the one we desire
603         unifyFunction(new_function, f_x, f_y);
604         Matrix solution = createMatrix(M, M);
605         Matrix X = createMatrix(max_n, M);
606         Matrix Y = createMatrix(max_n, M);
607
608         // Loop until convergence
609         while (N < (max_n - 1) && !checkSolutionConvergence(X, Y)) {
610             N += 1;
611             alternatingDirection(X, Y,f_x,f_y, N);
612         }
613
614         // COMPUTE SOLUTION
615         for (int i = 0; i < N; ++i) {
616             solution = matrixSub(solution, outerProduct(X[i], Y[i]));
617         }
618         //Plot vector field and export path
619         plotVectorField(solution);
620
621         return 0;
622     }
```

Bibliography

[1] A. Ammar, B. Mokdad, F. Chinesta, and R. Keunings, *A new family of solvers for some classes of multidimensional partial differential equations encountered in kinetic theory modeling of complex fluids*, Journal of Non-Newtonian Fluid Mechanics **139** (2006), no. 3, 153–176.

[2] Sheldon Axler, Paul Bourdon, and Wade Ramey, *Basic properties of harmonic functions*, pp. 1–29, Springer New York, New York, NY, 2001.

[3] Francisco Chinesta, Roland Keunings, and Adrien Leygue, *Pgd solution of the poisson equation*, pp. 25–46, Springer International Publishing, Cham, 2014.

[4] Francisco Chinesta, Adrien Leygue, Felipe Bordeu, Jose V. Aguado, Elias CUETO, David Gonzalez, I. Alfaro, Amine Ammar, and Antonio Huerta, *Pgd-based computational vademecum for efficient design, optimization and control*, Archives of Computational Methods in Engineering **20** (2013).

[5] Christopher Connolly and Roderic Grupen, *The application of harmonic functions to robotics*, Journal of Robotic Systems **10** (1993), 931 – 946.

[6] Sabudin elia nadira, Rosli Omar, and Che Ku Nor Hailma, *Potential field methods and their inherent approaches for path planning*, **11** (2016), 10801–10805.

[7] Antonio Falco, L. Hilario, Nicolas Montes, Marta Mora, and Enrique Nadal, *A path planning algorithm for a dynamic environment based on proper generalized decomposition*, Mathematics **8** (2020).

[8] Antonio Falco and Anthony Nouy, *Proper generalized decomposition for nonlinear convex problems in tensor banach spaces*, Numerische Mathematik **121** (2011).

[9] Santiago Garrido, Luis Moreno, Dolores Blanco, and Fernando Martín, *Robotic motion using harmonic functions and finite elements*, Journal of Intelligent and Robotic Systems **59** (2010), 57–73.

[10] David González, Joshué Pérez, Vicente Milanés, and Fawzi Nashashibi, *A review of motion planning techniques for automated vehicles*, IEEE Transactions on Intelligent Transportation Systems **17** (2016), no. 4, 1135–1145.

[11] Hu Hongyu, Zhang Chi, Sheng Yuhuan, Zhou Bin, and Gao Fei, *An improved artificial potential field model considering vehicle velocity for autonomous driving*, vol. 51, 2018, 5th IFAC Conference on Engine and Powertrain Control, Simulation and Modeling E-COSM 2018, pp. 863–867.

[12] Lydia E. Kavraki and Steven M. LaValle, *Chapter 5 motion planning*, 2007.

[13] Oussama Khatib, *Real-time obstacle avoidance for manipulators and mobile robots*, The International Journal of Robotics Research **5** (1985), 90 – 98.

[14] J.-O. Kim and P.K. Khosla, *Real-time obstacle avoidance using harmonic potential functions*, IEEE Transactions on Robotics and Automation **8** (1992), no. 3, 338–349.

[15] Fethi Matoui, B. Boussaid, and Abdelkrim Mohamed Naceur, *Distributed path planning of a multi-robot system based on the neighborhood artificial potential field approach*, vol. 95, 07 2018, p. 003754971878544.

[16] Frank Morgan, *Geometric measure theory*, fifth ed., Elsevier/Academic Press, Amsterdam, 2016, A beginner's guide, Illustrated by James F. Bredt. MR 3497381

[17] Hirsch M.W, *Differential equations, dynamical systems, and linear algebra*, (1974).

[18] N. Rusli N.A Halif, *Alternating direction implicit (adi) method for solving two dimensional (2-d) transient heat equation*, ASM Sci. J, Special Issue 6, 2019, pp. 28–33.

[19] D. W. Peaceman and H. H. Rachford, Jr., *The numerical solution of parabolic and elliptic differential equations*, Journal of the Society for Industrial and Applied Mathematics **3** (1955), no. 1, 28–41.

[20] PF. Riesz, *Sur les opérations fonctionnelles linéares*, Comptes Rendus Acad. Sci, 1909, pp. 974–977.

[21] Erhard Schmidt, *Zur theorie der linearen und nichtlinearen integralgleichungen. i. teil: Entwicklung willkürlicher funktionen nach systemen vorgeschriebener*, Mathematische Annalen **63** (1907), 433–476.

[22] Ali Taheri, *Harmonic Functions and the Mean-Value Property*, Function Spaces and Partial Differential Equations: Volume 1 - Classical Analysis, Oxford University Press, 07 2015.

[23] Yu Wang, Gang Zhu, Yajun Liu, Jingwei Zhao, and Wei Tian, *Use of artificial potential field theory to determine the spatial position of a navigation-guided orthopedic surgical robot*, 05 2020, pp. 11–17.

[24] O.C. Zienkiewicz, R. Taylor, and J.Z. Zhu, *Weak forms and finite element approximation: 1-d problems*, pp. 47–92, 12 2013.