Universidad
Carlos III de Madrid

Fast Marching Methods in path and motion planning: improvements and high-level applications

Javier V. Gómez González

Ph.D. Thesis

Department of Systems Engineering and Automation
Leganés, Madrid, Spain, November 2015

Fast Marching Methods in path and motion planning: improvements and high-level applications

Candidate
Javier V. Gómez González

Advisers
Luis Moreno Lorente
Santiago Garrido Bullón

Review Committee

President: Carlos Balaguer _____

Secretary: Antonio Giménez _____

Vocal: Isabel Ribeiro _____

Grade:

Leganés, Madrid, Spain, November 20$^{\text{th}}$, 2015

Preface

Path planning is defined as the process to establish the sequence of states a system must go through in order to reach a desired state. Additionally, motion planning (or trajectory planning) aims to compute the sequence of motions (or actions) to take the system from one state to another. In robotics path planning can refer for instance to the waypoints a robot should follow through a maze or the sequence of points a robotic arm has to follow in order to grasp an object. Motion planning is considered a more general problem, since it includes kinodynamic constraints.

As motion planning is a more complex problem, it is often solved in a two-level approach: path planning in the first level and then a control layer tries to drive the system along the specified path. However, it is hard to guarantee that the final trajectory will keep the initial characteristics.

The objective of this work is to solve different path and motion planning problems under a common framework in order to facilitate the integration of the different algorithms that can be required during the nominal operation of a mobile robot. Also, other related areas such as motion learning are explored using this framework. In order to achieve this, a simple but powerful algorithm called Fast Marching will be used. Originally, it was proposed to solve optimal control problems. However, it has became very useful to other related problems such as path and motion planning.

Since Fast Marching was initially proposed, many different alternative approaches have been proposed. Therefore, the first step is to formulate all these methods within a common framework and carry out an exhaustive comparison in order to give a final answer to: which algorithm is the best under which situations?

This Thesis shows that the different versions of Fast Marching Methods become useful when applied to motion and path planning problems. Usually, high-level problems as motion learning or robot formation planning are solved with completely different algorithms, as the problem formulation are mixed. Under a common framework, task integration becomes much easier bringing robots closer to everyday applications.

The Fast Marching Method has also inspired modern probabilistic methodologies, where computational cost is enormously improved at the cost of bounded, stochastic variations on the resulting paths and trajectories. This Thesis also explores these novel algorithms and their performance.

Acknowledgments

Many people deserve to appear in these acknowledgments for both personal and professional reasons.

Firstly, I want to thank my girlfriend Ruth for making my brain fly away from its logical prison and my family for supporting me along these years, both emotionally and economically (otherwise it would have been impossible to continue this Thesis). They should appear as coauthors of this Thesis.

I want to hugely thank Diego Rodriguez-Losada. Former professor that has become a close friend and a mentor. He was one of my principal sources of inspiration for my decisions during the last years. I do not forget Miguel Hernando, Pablo San Segundo, Alberto Valero, and the Biicode team, who kept me close to the real world and taught me how cool is to work in something you love with people you like.

I want to thank all the colleagues I met during my research. Alberto Vale and Pedro Lima, from Lisbon, I really enjoyed working with them. Frode Eika Sandness, from Oslo, for acting as a mentor and showing a great interest in my evolution as researcher. Finally, Marco Pavone, Edward Schmerling, Joseph Starek and the Stanford Autonomous Systems Lab, for giving me the incredible opportunity to be a visitor in their lab. It is a experience I will never forget.

And last, but not least, I want to thank my lab mate, David Álvarez, for lending me the money to buy bread when I forgot it, and listening to me when my code did not compile.

Contents

List of Tables

List of Figures

Chapter 1

Introduction to motion planning

The path planning problem is considered solved by some researchers. They argue that the current existing solutions are good enough to solve most of the problems that require robot motion. However, it is still one of the most active fields in robotics research. In fact, in the last editions of the most important robotic conferences (IROS, ICRA, RSS) path and motion planning are still among the topics with more accepted papers. Therefore, many researchers are still focusing their efforts on the path planning problem, trying to come up with better, faster and more general planning frameworks or modelling the real world in a more complex way so that motion planning do not require an intense computational effort.

Hence, it turns out that the path planning problem is not as solved as many people think. It is true that there exist very good solutions but none of them can be considered as a *general* framework. In fact some approaches only perform well in specific cases or are not able to adapt to new models of robots or environments. For example, those algorithms which perform well in low dimensions become much slower when increasing the dimensionality of the problem. Analogously, fastest algorithms provide stochastic solutions with few guarantees about their solutions. Other planners are difficult to tune if the configuration space does not follow an Euclidean metric. Furthermore, robotics is moving towards small, more agile robots with fewer computational capabilities. Thus, even faster solutions are desired. Not to mention problems with kinodynamic constraints, in which state-of-the-art planners could take hundreds of seconds to come up with a satisfactory solution.

It is important to remark the differences between path planning and motion planning. Path planning tries to compute purely geometric paths which go from one state to another one. Typically, states are only positions and orientations. However, motion or trajectory planning also computes dynamic properties along the path, such as velocities, accelerations, etc. Although there is a distinction between path and motion planning, sometimes these terms are interchangeably used as motion planning can be solved by just incrementing the dimensionality of the configuration space and applying path planning techniques, with the proper feasibility checks. Most of the work of this Thesis builds upon a trajectory planner called Fast Marching Square (FM^2), which provides a path and a velocities profile. FM^2 is applied to many different problems but some of the solutions proposed do not take into account the velocities profile, dealing only with the path planning part of the problem.

1.1 Context and motivation

As mentioned before, in order to consider a path planning algorithm "good" it has to perform well in many different cases, because it is not desirable to have different path planning algorithms for every different situation. This would require to distinguish those different situations and implement many different algorithms, which is complex

and very time consuming.

In this context, this work focuses on analyzing the capabilities of the Fast Marching Method (FMM) and its variations when applied to path and motion planning. The FM^2 algorithm will be the focus which, in parallel development to this Thesis, has shown its great adaptability to several motion planning problems such as robot formation motion planning [9], grasp planning [10], socially-aware path planning [11], etc.

This Thesis focuses on the FM^2 adaptability, pushing the algorithm towards new problems, such as motion learning. It is also deeply examined how the algorithm performs in real-world scenarios where the path planning is critical. Furthermore, new variants to the FM^2 are proposed which bring it closer to problems where its application was not practical, such as high-dimensional problems. For example, the application of cost-to-go heuristics with the purpose of boosting the planning computation time is studied.

Although an in-depth explanation of the contributions is contained in the next Chapter, Section 2.5, they are outlined here as well. Firstly, this Thesis addresses a comparison of the different Fast Methods available in the literature, which are the basis of the algorithms contained in this Thesis. Then, high-level applications such as path planning and motion learning are studied from the perspective of these Fast Methods. Lastly, latest trends in stochastic motion planning are explored maintaining the Fast Methods perspective.

1.2 Document structure

The document is divided in 8 Chapters and an Appendix apart from this introduction. The first 3 Chapters are introductory and provide the required background to understand this Thesis and its contribution. The following Chapters (except the last one), contain specific methodologies and algorithms. Thus, a more detailed formulation and state of the art is given in each one of them. They also contain their own results section.

Chapter 2 formalizes the path planning problem and provides a detailed state of the art of path and motion algorithms. It also summarizes the previous work in FM^2.

Chapter 3 introduces the Eikonal equation and how it is solved by means of the Fast Marching Method.

Chapter 4 carries out a exhaustive description of the Fast Marching Method and the variations proposed in the last years. Nine different Fast Marching-based algorithms are detailed and experimentally compared under a common mathematical formulation.

Chapter 5 builds on top of Chapters 4 and 3, as it introduces the Fast Marching Square (FM^2) motion planning algorithm, which relies on the Fast Marching Method.

Three new variants are included, focusing on improving the FM^2 performance and output quality.

Chapter 6 extensively analyzes the performance of FM^2 in a real-world scenario: the ITER project, where the path planning is a critical step in a process where a failure could suppose a environmental disaster. The problem is described and the results of applying FM^2 are analyzed.

Chapter 7 proposes a novel motion learning algorithm, the Fast Marching Learning (FML) method. Based on FM^2, this algorithm is able to accurately replicate and generalize taught motions given by an expert. It is deeply tested and compared with a state-of-the art method.

Chapter 8 introduces a novel, bi-directional, asymptotically optimal, sampling-based path planning method based on the Fast Marching Method which clearly outperforms its counterparts.

Then, Chapter 9 outlines the main conclusions extracted from this Thesis and its results, pointing out the most promising ideas in this area.

Finally, Appendix A is attached, which details the formulation of n-dimensional gridmaps and specifies how to generalize operations such as neighbor extraction in these grids.

Chapter 2

State of the Art

Path planning has been a very active field since the days humankind started to think about artificial intelligence. Although currently there are very good approaches, there is not any algorithm able to satisfy all the requirements that a path planning algorithm needs to satisfy: low computational complexity, reliability, completeness, robustness, smooth paths, optimal solutions, safety, etc.

The algorithms which are very fast usually provide non-smooth paths, so a smoothing step is required afterward. On the other hand, the algorithms whose solutions are smooth are usually fast in 2 dimensions (or even 3) but their usefulness decrease as the dimensionality is increased.

2.1 Problem formulation

The path planning problem formulation is clearly described in the literature. This Section details the formal definition based on the one given in [12]. Note that only the path planning formulation is detailed. This Thesis focuses in both path and motion planning with FM^2. However, the motion planning algorithm can be understood as the path planning algorithm with the only difference that the output is no longer a path, but a trajectory: a sequence of actions and durations, or the combination of a path plus a velocity reference to traverse the path.

2.1.1 Notation

This Section describes the mathematical notation to be used along the Thesis.

Consider the Euclidean space in N dimensions, i.e., $\mathbb{R}^N$. A ball of radius $r > 0$ centered at $\bar{x} \in \mathbb{R}^d$ is defined as $B(\bar{x};\ r) := \{x \in \mathbb{R}^d \,|\, \|x - \bar{x}\| < r\}$. Given a subset $\mathcal{X}$ of $\mathbb{R}^N$, its boundary is denoted by $\partial\mathcal{X}$ and its closure is denoted by $\mathrm{cl}(\mathcal{X})$.

2.1.2 Path planning problem formulation

Let $\mathcal{X} = [0,\ 1]^N$ be the configuration space, where the number of dimensions, N, is an integer larger than or equal to two. $\mathcal{X}_{\text{obs}}$ represents the obstacle set (or region), while $\mathcal{X}_{\text{free}}$ contains the obstacle-free space, being both mutually exclusive. Usually, it is considered $\mathcal{X}_{\text{obs}}$ such that $\mathcal{X} \setminus \mathcal{X}_{\text{obs}}$ is an open set and thus $\mathcal{X}_{\text{free}} = \mathrm{cl}(\mathcal{X} \setminus \mathcal{X}_{\text{obs}})$. In other words, the boundary of the space belongs to the obstacle set $\partial\mathcal{X} \subset \mathcal{X}_{\text{obs}}$. As this is necessary in some mathematical demonstrations, it has not noticeable consequences in practice.

The initial condition (or state) $\mathcal{X}_{\text{s}}$ is an element of $\mathcal{X}_{\text{free}}$, and the goal region $\mathcal{X}_{\text{goal}}$ is an open subset of $\mathcal{X}_{\text{free}}$. A path planning problem is fully defined by a triplet $(\mathcal{X}_{\text{free}}, \mathcal{X}_{\text{s}}, \mathcal{X}_{\text{goal}})$. A d-dimensional function $\sigma : [0, 1] \to \mathbb{R}^d$ is called a *path* if it is continuous and has bounded variation (see [13, Section 2.1] for a formal definition). As

the path is defined as a continuous function on a one-dimensional, bounded domain, bounded variation can be understood as finite length.

In order to be useful the path requires to be collision-free. More formally, a path is denoted as *collision-free* if $\sigma(\tau) \in \mathcal{X}_{\text{free}}$ for all $\tau \in [0, 1]$. Furthermore, a collision-free path for the planning problem $(\mathcal{X}_{\text{free}}, \mathcal{X}_{\text{s}}, \mathcal{X}_{\text{goal}})$ is said *feasible path* if $\sigma(0) = \mathcal{X}_{\text{s}}$, and $\sigma(1) \in \text{cl}(\mathcal{X}_{\text{goal}})$.

The goal $\mathcal{X}_{\text{goal}}$ region is expected to be a "well-behaved" set, that is, it has at least an arbitrarily minimum volume and its boundary has bounded curvature. Formally, $\mathcal{X}_{\text{goal}}$ is said to be *regular* if $\exists \xi > 0$ such that $\forall x \in \partial\mathcal{X}_{\text{goal}}$, there exists a ball in the goal region $B(\bar{x}; \xi) \subseteq \mathcal{X}_{\text{goal}}$ such that $x \in \partial B(\bar{x}; \xi)$. Although this requirement might seem also too formal, in fact it is critical to guarantee that it is possible to find a solution. Planning algorithms require these assumptions in order to identify the goal region. Non-bounded curvature on goal regions or infinitesimal volume cause the path planning problem to be not well defined.

2.1.3 Optimal path planning problem formulation

So far, the problem definition does not give preference for a solution path over the rest of feasible paths. Therefore, a metric is required in order to sort the paths and select the more convenient depending on the application.

Let Σ be the set of all feasible paths. A cost function for the planning problem $(\mathcal{X}_{\text{free}}, \mathcal{X}_{\text{s}}, \mathcal{X}_{\text{goal}})$ is a function $c : \Sigma \to \mathbb{R}_{\geq 0}$ that assigns a non-negative value to each feasible path. The cost function is often required to satisfy the triangle inequality, and this is assumed along this Thesis unless stated otherwise. Some examples of this case are *arc length* of σ with respect to the Euclidean metric in $\mathcal{X}$, time required to traverse the whole path, or energy consumed by the system.

The optimal path planning problem is then defined as follows:

> **Optimal path planning problem**: Given a path planning problem $(\mathcal{X}_{\text{free}}, \mathcal{X}_{\text{s}}, \mathcal{X}_{\text{goal}})$ with a regular goal region and an arc cost function $c : \Sigma \to \mathbb{R}_{\geq 0}$, find a feasible path σ^* such that $c(\sigma^*) = \min\{c(\sigma) : \sigma \text{ is feasible}\}$. If no such path exists, report failure.

2.2 Classification of the path planning algorithms

There is a huge literature in path planning. However, most of the path planning algorithms can be classified as follows:

- *Geometric methods*: The environment is described as a set of polygons and, from their properties, the paths are computed as a sequence of geometric primitives such as lines, arcs, splines, etc. The most common approach of this class

are those based on *visibility graphs* [14, 15]. The visibility graph of a set of non-intersecting polygonal obstacles in the plane is an undirected graph whose vertices are the vertices of the obstacles and whose edges are pairs of vertices such that the open line segment between each two vertices does not intersect any of the obstacles. Although this approach is very intuitive and provide optimal paths in 2D (in terms of path length), its extension to more dimensions is not trivial and optimality is lost. Also, to determine the visible portions of the map is a complex and computationally intense problem.

Other common geometric approach is to compute the path using the Delaunay triangulation of the environment [16]. This is one of the most widely-employed triangulation algorithms because it minimizes the sum of all the angles of the triangles in the triangulation. Although its complexity is $\mathcal{O}(n \log n)$ [17], its main drawback is that the triangulation is not unique and it can lead to very weird paths, as detailed in Chapter 6.

- *Graph- and tree-based methods*: This is the class with most research effort during the recent years. In this class, the environment is modeled by the state of the robot with respect to environment. A graph is built in which every node is one state of the robot (a state can be for instance a pose or velocities and accelerations). The transitions between states are modeled as costs and the path chosen is the one that minimizes the total cost of reaching a goal state from a current state.

  There exist several subclasses within the graph search method, depending on how the graph is constructed and also how the costs are assigned. One of the possible classifications is:

  - Grid-based methods: characterized by discretization of the space in grid cells. The most common grid representations are with rectangular or triangular cells. This discretization can lead to a loss of accuracy, but this issue is overcome by choosing an appropriate cell size. Every cell of the space is a node of a graph, and it is connected with its neighbours (4 or 8-connectivity in 2D, depending on the algorithm) and the cost of travelling from one node to other can be set on many different ways. Once the costs are set, graph search algorithms can be applied in order to choose the path which allows to reach the goal point with the minimum possible cost.

    Within this group, the typical graph search algorithms can be found, such Dijkstra [18] or A* [19]. Modifications of these have been already proposed, such as D* [20], which efficiently reuses information from previous steps

to avoid redundant computations in dynamically changing scenarios or D*-lite [21] which simplifies D*.

Finally, wavefront propagation methods, such as Fast Marching Method (FMM) [22] which is the focus of this Thesis (see Chapter 4), lies also in this group. In this case, the cost for each node is related with the time a propagating wave takes to reach that node. Its formulation allows to guarantee a convergence to the optimal path in terms of traversal time as the grid is refined. When the propagation velocity is constant, the path becomes also the optimal in terms of length.

– Sampling-based methods: this type of algorithms incrementally searches in the space for a solution using a collision detection algorithm [23]. The most extended algorithms of this group are those based on rapidly exploring random trees (RRTs) [24]. The branches of these trees are randomly created from the initial point of the trajectory. Another common approach is the Probabilistic RoadMaps (PRM) [25]. PRM creates a roadmap (set of connections) among a set of points randomly sampled.

  The main problem of sampling-based algorithms is their stochasticity. Most of the times the computed paths are far from the optimal one, are neither safe nor smooth. However, since RRT* and PRM* were proposed [26], this problem has vanished and thus this family of algorithms is the most widely used in practical applications. Finding faster approaches is a very active field nowadays, and many more asymptotically-optimal algorithms have been proposed, for instance BIT* [27] and RRT# [28]. A detailed state of the art is given in the next Section and in Chapter 8.

- *Artificial potential fields methods:* These algorithms rely in a grid representation but are conceptually different from other grid-based methods.Artificial potential fields can be included in this group. Conceptually, they represent the robot as a punctual electric charge and the goal point as a charge of opposite sign [29]. Thus, the robot is attracted by the goal. Obstacles are modeled by charges of the same sign of the robot, so that it is repelled and collisions are avoided. Although they are conceptually simple and easy to implement, their main drawback is that their are prone to local minima and oscillatory paths.

Figure 2.1 summarizes this classification. It is important to remark that this classification might be considered not unique, as other criteria could be chosen: stochasticity, optimality, etc. For example, artificial potential methods are usually implemented using a grid discretization. A* models the environment as a grid as well, but it is a graph search algorithm. In the same way, RRT and FMM could be in the same group considering that both work on a continuous space representation. However, this classification was chosen as it is the most common and the easiest to understand.

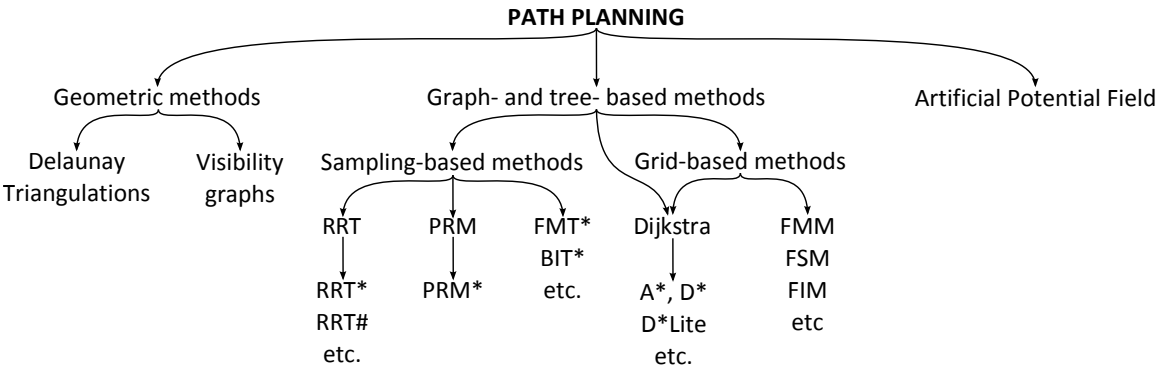


Figure 2.1: Classification of the current path planning approaches.

In the next sections the state of the art of path planning research is summarized as it is the main topic of this Thesis.

2.3 Latest trends in path planning research

Many researchers consider that the path planning problem as a solved one. However, most (if not all) current approaches perform poorly when they are applied to problems in which different assumptions are made, even if these are considered simpler. Current research efforts focus on more realistic environment representations and reformulating the path planning problem in order to get it closer to the real problem robots have to face when working among humans.

For instance, during recent years many researchers have focused on path planning in dynamic environments with high uncertainty, due to sensor noise or also due to a movement of other agents in the surroundings of the robot or crowded places. Also, navigation in large environments, in which all the obstacles are not modelled.

For instance, in dynamic, uncertain environments (DUEs), robots must work in close proximity with many other moving agents, whose future actions and reactions are difficult to predict accurately. Robot motion planning in dynamic environments has recently received substantial attention because of the Defense Advanced Research Project Agency (DARPA) Urban Challenge [30] and growing interest in service and assistive robots (see, e.g., [31] and [32]). In urban environments, traffic rules define the expected behaviors of the dynamic agents and constrain expected future locations of moving objects. In other applications, agent behaviors are less well defined, and the prediction of their future trajectories is more uncertain.

When the future locations of moving agents are known, the two common approaches are to add a time dimension to the configuration space, or to separate the spatial and temporal planning problems [23]. When the future locations are unknown,

the planning problem is solved locally (via reactive planners) [33], [34] or in conjunction with a global planner that guides the robot toward a goal [35]. The work in [36] presents an initial approach to a general framework that integrates planning, prediction an estimation, incorporating as well the effect of anticipated future measurements in the motion planning process.

Another important problem which still remains open are those systems with high dimensional configuration spaces. These approaches are usually based on RRT, with smoothing steps [37]. Most of these approaches are just modifications of the RRT basic algorithm to apply it to specific cases: Transition-based RRT (T-RRT) [38] which includes a state transitions cost, Manhattan-like RRT (ML-RRT) [39] which incorporates a mix of active and passive parameters, or MLT-RRT [40] which is a combination of the aforementioned. Other approaches simply focus on dealing with much higher dimensionality, but under very specific contexts [41].

Other interesting problems have been faced as well such path planning for multi-section continuum arms [42] or maximum planning coverage minimizing energy [43].

This Thesis focuses on optimal path planning methods where the cost metric is arbitrarily defined, usually under the assumption that this metric satisfies the triangle inequality. This problem has been mainly solved by means of deterministic methods, using graph-based algorithms [23] or through Fast Marching-based approaches (see Section 2.4 for a detailed state of the art). However, Karaman and his colleagues were the firsts to design asymptotically-optimal, sampling-based algorithms (the main drawback of this family of algorithms) proposing the RRG, PRM* and RRT* algorithms [26]. This boosted the research in this field, as these algorithms provide optimality guarantees while maintaining the key properties of sampling-based methods: fastness and dimensional scalability.

After Karaman's groundbreaking work, many new approaches have been proposed following the same formulation and mathematical framework. Informed-RRT* [44] uses the best current best path cost to prematurely reject samples before inserting them into the tree, boosting RRT* convergence. CForest [45] proposes a simple but powerful paralellizing framework for incremental, asymptotically-optimal, sampling- and tree-based algorithms, integrated into the most used path planning library, the OMPL [46, 47], by the author of this Thesis. Kinodynamic RRT* versions have been also proposed [48, 49].

In fact, Janson et al [12] went a step further and proposed the Fast Marching Trees (FMT*): an asymptotically-optimal, sampling-based path planning algorithm inspired in the Fast Marching Method. This novel approach relaxes the mathematical requirements and thus provides faster convergence towards the optimum path than its counterparts. In fact, this method has been rapidly adopted and improved by the community, proposing anytime versions [50], kinodynamic versions [51, 52], and bidirectional versions, detailed in this Thesis in Chapter 8. Also inspiring new algorithms

as the Batch Informed Trees [27].

Last but not least, Yershov proposed a very different approach to bringing together Fast Marching and sampling-based methods [53]. His approach implements a modified Fast Marching Method over a simplicial mesh incrementally refined. Thus, the convergence to the optimum is fast as continuous spaces are volumetrically discretized, while previous approaches are graph-based, and thus the whole space is reduced to a set of uni-dimensional connections among states, requiring infinite time to cover the whole space.

The literature in path and motion planning, specially for sampling-based methods is quite vast. This state of the art has focused on the works closely related to this Thesis and some other examples. A recent, more detailed review can be found in [54].

2.4 Fast Marching Methods in path planning: previous work

The Fast Marching Method (FMM), detailed in Chapter 4, is the central algorithm of this Thesis. This Section details its road map during the last years of FMM applied to path and motion planning. The application of the Fast Marching Method in path planning is not novel but recent. At the beginning, the Fast Marching Method was used to find paths within the Voronoi diagram [55]. Later, Fast Marching was combined with the Extended Voronoi Transform (EVT) in what was called Voronoi Fast Marching (VFM) [56]. This approach has been applied to exploration of cluttered environments [57] and to 2D robot formations [58]. An improvement of the VFM was the Fast Marching Square (FM^2) planning method [59].

The VFM and FM^2 have been applied to other planning problems such as: 3D robot formations [9], smooth planning for non-holonomic robots [60, 61], simultaneous robot localization and mapping [62], planning in outdoor environments [63], [64], RRT path smoothing [65], tube skeletons [66], grasp planning [10], socially-aware motion planning [11], underwater vehicle planning [67], formations of unmanned surface vehicles [68], and many others. As described in previous section, FMM has also inspired sampling-based algorithms such as FMT* [12] and simplicial-based methods [53].

2.5 Open Issues and Contributions

Despite the great amount of work available in the topic, there are still plenty of open problems to be addressed. Furthermore, although some of the problems are, in practice, solved, latest robotic trends have made researchers to revisit some of the approaches already accepted by the community.

To be more specific, this Thesis studies, among others, three of these cases. Although there are many sound approaches to motion learning, most of them suffer from the main drawback: stochasticity. That is, a good learning is probable but not guaranteed and it mostly depends on the demonstrations given to be learned. Another example is that, when building real-world robotic applications, some of the latest planning approaches are not robust enough to be used without any supervision. Therefore, more basic, stable approaches are used that guarantee a minimum quality of the solutions. Finally, the last example is the constant search for more computationally-efficient algorithms. Robots are becoming smaller and thus their computational capabilities are shrank. Thus, more efficient algorithms are required so that automony is not affected.

The main contribution of this Thesis is to address those examples aforementioned and therefore to push the application of Fast Marching-based methods to robotic applications. Three specific contributions can be identified:

- Fast wavefront methods comparison: in Chapter 4 the complete family of sequential, isotropic wavefront propagation methods, called Fast Methods, is introduced and compared, both qualitatively and quantitatively. Although these methods are widely used nowadays, there is a lack of a deep comparison among all the algorithms.

- High-level applications: Chapter 6 proposes a FM^2 application to the ITER project. This research lead to a more robust solution to the motion planned problem proposed by the ITER project, of critical relevance for the future of humankind. The Fast Marching Learning algorithm is proposed in Chapter 7. Up to author's knowledge, it is the first non-Bayesian motion learning algorithm. Therefore, a novel approach motion learning is proposed.

- Improve FMT*: In Chapter 8 a resampling strategy is proposed to improve the FMT* algorithm. FMT* main drawback is its batch design, where the number of samples to be used is a parameter to be set by the user. In some cases, the sampled set is not enough to find a feasible solution and thus FMT* would return failure. With the proposed resampling strategy this drawback no longer exists, drastically increasing success rates. It is presented together with the Bi-Directional FMT*, a work carried out in collaboration with the Autonomous Systems Lab from Stanford University, where this bidirectional approach was initially proposed.

Chapter 3

Eikonal Equation and the Fast Marching Method

3.1 Introduction

The Fast Marching Method (FMM) has been extensively applied since it was firstly proposed in 1995 [69] as a solution to isotropic control problems using first-order semi-Langragian discretizations on Cartesians grids. Their main field of application are robotics [70, 71, 68] and computer vision [72], mainly medical image segmentation [73, 74]. However, it has proven to be useful in many other applications such as tomography [75] or seismology [76].

The first approach was proposed by Tsitsiklis [69], but the most popular solution was given few months later by Sethian [22] using first-order upwind-finite differences in the context of isotropic front propagation. Differences and similarities between both works can be found in [77].

FMM was originally proposed to simulate a wavefront propagation through a regular discretization of the space. However, many different approaches have been proposed, extending these methods to other discretizations and formulations. For a more detailed history of Fast Marching methods, interested readers are referred to [78].

This Chapter introduces FMM, its formulation and its application to path planning, which is the main topic of this Thesis. In particular, next Section provides an intuitive explanation to FMM. Then, Section 3.3 details the formulation and how to solve the Eikonal equation in n-dimensions. Section 3.4 details the FMM and, finally, Section 3.5 outlines how the FMM can be applied to path planning.

3.2 Introduction to Fast Marching Methods

The FMM can be intuitively understood considering the expansion of a wave. If a stone is thrown into a pond a wavefront is originated, and this wave expands with a circle shape around the point where the stone fell. In this example the fluid is always water, thus the wave expansion velocity is always the same, and thus the wavefront is circular. Instead, if this experiment is repeated mixing water and oil, it would be observed that the wave expands at different speeds in each medium. As a consequence, the wavefront will not be circular anymore. If another point on the fluid is considered (a target point), the wavefront will arrive to that point after a certain time. The path that the wavefront has followed from the origin to the target point will be the shortest path (in terms of time), considering that the traveling speed along the path is the expansion velocity of the wavefront (which differs depending on the fluid). This path can be computed by using gradient descent (following the direction of maximum change) from any given point. The smoother the variations are in the wave propagation velocities, the smoother the path will be. These concepts are plotted in Figure 3.1. In few words, FMM computes the arrival times of the wavefront

from a starting point to all the points of the reachable space.

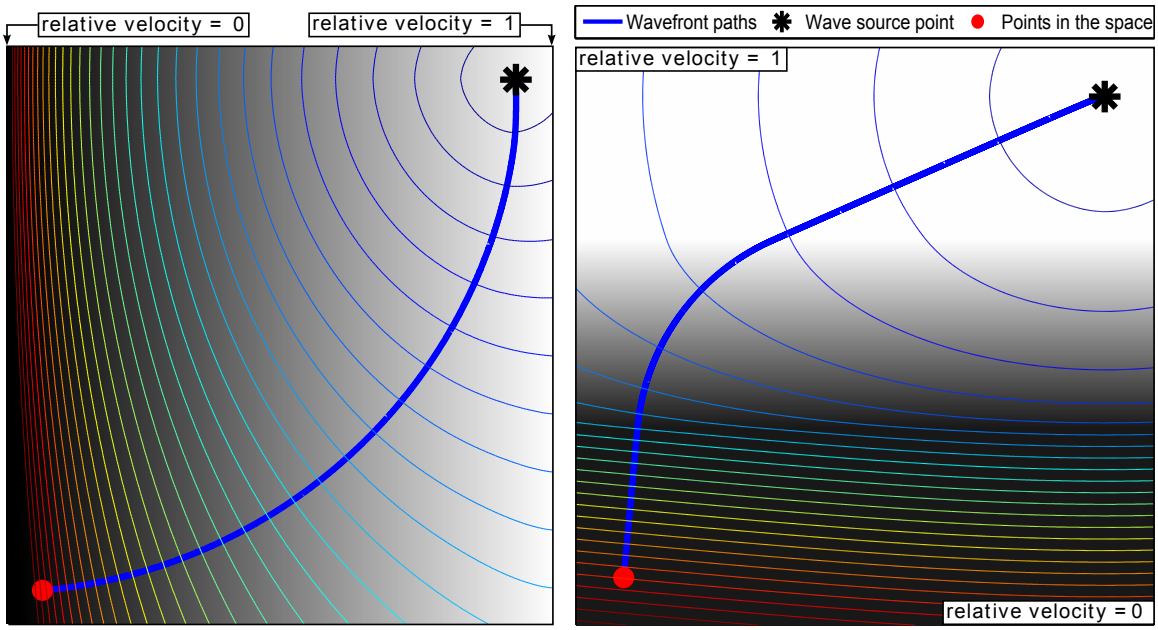
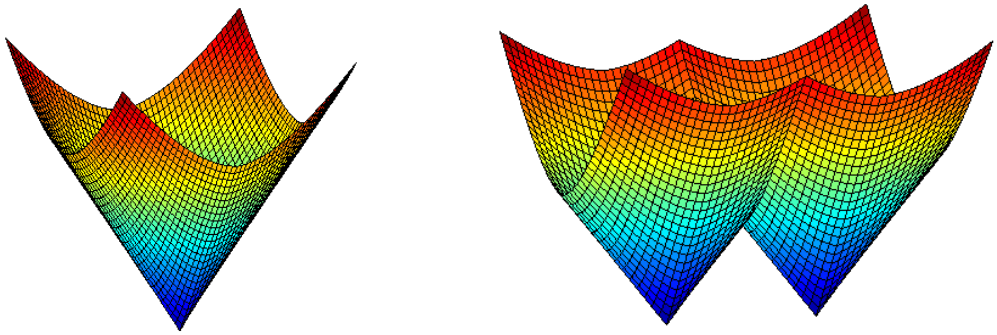


Figure 3.1: Examples of a wave propagation through media with different velocities.

The FMM is a particular case of Level Set Methods, initially developed by Osher and Sethian [79]. The FMM assumes that the wave propagation velocity is always non-negative, and thus the wavefront will never contract. Usually, propagation velocities are defined by the environment or by higher-level algorithmic layers so that it is easy to satisfy this assumption. Therefore, if only one wave source is used FMM ensures that the time-of-arrival map will have only one minimum (both local and global) at the start point. Section 3.5 details how FMM deals when obstacles are present in the environment.

Many different wave sources can be set. In that case, there will be as many minima as starting points, each one of them located at these wave sources. An example is shown in Figure 3.2 with a 3D interpretation of the arrival times represented in the Z axis.

From a high-level perspective, FMM computes the arrival time for one point at every iteration. This is done efficiently by selecting the non-visited points with lower value calculated so far. When evaluating a point, its value is increased taking into account current propagation velocity and only the neighbors with lower times in each dimension. As velocities are always non-negative current value will be at least as high as the neighbor with lower value, and thus no local minima can appear. However, saddle points are possible but they are not a problem as gradient descent (used to



(a) One wave source (b) Two wave sources

Figure 3.2: 3D representation of the FMM output in a 2D grid. The arrival time is shown in the Z axis .

compute the paths) will be able to continue without problems. In this case, it means that two solution with the same propagation time are possible and which one is chosen uniquely depends on the gradient descent implementation.

3.3 Problem Formulation

Formally, FMM is built to solve the nonlinear boundary value problem[1]. That is, given a domain $\mathcal{X}$ and a velocity field function $F : \mathcal{X} \to \mathbb{R}_+$ which represents the local speed of the motion, drive a system from a starting set $\mathcal{X}_s \subset \mathcal{X}$ to a goal set $\mathcal{X}_{goal} \subset \mathcal{X}$ through the fastest possible path. The Eikonal equation computes the minimum time-of-arrival function $T(\mathbf{x})$ as follows:

$$\begin{aligned} \left|\nabla T(\mathbf{x})\right| F(\mathbf{x}) = 1,\ \mathcal{X} \subset \mathbb{R}^N \\ T(\mathbf{x}) = 0, \mathbf{x} \subset \mathcal{X}_s \end{aligned} \tag{3.1}$$

Once solved, $T(\mathbf{x})$ represents a distance (time-of-arrival) field containing the time it takes to go from any point $\mathbf{x}$ to the closest point in $\mathcal{X}_s$ following the velocities on $F(\mathbf{x})$.

It is assumed, without loss of generality, that the domain is a unit hypercube of N dimensions: $\mathcal{X} = [0, 1]^N$. The domain is represented with a rectangular Cartesian

[1]This problem formulation closely follows [80]

grid $\mathcal{X} \subset \mathbb{R}^N$, containing the discretizations of the functions $F(\mathbf{x})$ and $T(\mathbf{x})$, $\mathcal{F}$ and $\mathcal{T}$ respectively. The grid points $\mathbf{x}_{\mathrm{ij}} = (x_i, y_i), \mathbf{x}_{\mathrm{ij}} \in \mathcal{X}$ represents the point $\mathbf{x} = (x, y)$ in the space corresponding to a cell (i, j) of the grid (for the 2D case). For notation simplicity, $T_{\mathrm{ij}} = T(\mathbf{x}_{\mathrm{ij}}) \approx T(\mathbf{x}), T_{\mathrm{ij}} \in \mathcal{T}$, that is, T_{ij} represents an approximation to the real value of the function $T(\mathbf{x})$. Analogously, $F_{\mathrm{ij}} = F(\mathbf{x}_{\mathrm{ij}}) \approx F(\mathbf{x}), F_{\mathrm{ij}} \in \mathcal{F}$. The set of Von-Neumann neighbors (4-connectivity in 2D) of grid point $\mathbf{x}_{\mathrm{ij}}$ is denoted as $\mathcal{N}(\mathbf{x}_{\mathrm{ij}})$. For a general grid of N-dimensions, cells will be referred by their indices (or keys) i as $\mathbf{x}_{\mathrm{i}}$, since a flat representation is more efficient for such datastructure. More details about n-dimensional grids formulation and representation through flat arrays is given in Appendix A.

3.3.1 n-Dimensional Discrete Eikonal Equation

In this Section the most common first-order discretization of the Eikonal equation is detailed. There exist many other first-order and higher-order approaches on grids, meshes and manifolds [81, 82, 83, 84].

The discrete Eikonal equation is derived in 2D for better understanding. The most common first-order discretization is given in [79], which uses an upwind-difference scheme to approximate partial derivatives of $T(\mathbf{x})$ ($D_{\mathrm{ij}}^{\pm x}$ represents the one-sided partial difference operator in direction $\pm x$ and Δx and Δy are the grid spacing in the x and y directions):

$$\begin{aligned} T_x(\mathbf{x}) &\approx D_{\mathrm{ij}}^{\pm x} T = \tfrac{T_{i\pm1,j} - T_{\mathrm{ij}}}{\pm \Delta_{\mathrm{x}}} \\ T_y(\mathbf{x}) &\approx D_{\mathrm{ij}}^{\pm y} T = \tfrac{T_{i,j\pm1} - T_{\mathrm{ij}}}{\pm \Delta_{\mathrm{y}}} \end{aligned} \tag{3.2}$$

$$\left\{ \begin{array}{c} \max(D_{\mathrm{ij}}^{-x} T, 0)^2 + \min(D_{\mathrm{ij}}^{+x} T, 0)^2 + \\ \max(D_{\mathrm{ij}}^{-y} T, 0)^2 + \min(D_{\mathrm{ij}}^{+y} T, 0)^2 \end{array} \right\} = \frac{1}{F_{\mathrm{ij}}^2} \tag{3.3}$$

Simpler but less accurate solution to Equation 3.3 is proposed in [85]:

$$\left\{ \begin{array}{c} \max(D_{\mathrm{ij}}^{-x} T, -D_{\mathrm{ij}}^{+x} T, 0)^2 + \\ \max(D_{\mathrm{ij}}^{-y} T, -D_{\mathrm{ij}}^{+y} T, 0)^2 \end{array} \right\} = \frac{1}{F_{\mathrm{ij}}^2} \tag{3.4}$$

Replacing Equation 3.2 in Equation 3.4 and letting

$$\begin{aligned} T &= T_{i,j} \\ T_{\mathrm{x}} &= \min(T_{i-1,j}, T_{i+1,j}) \\ T_{\mathrm{y}} &= \min(T_{i,j-1}, T_{i,j+1}) \end{aligned} \tag{3.5}$$

the Eikonal Equation can be rewritten for a discrete 2D space as:

$$\max\left(\frac{T-T_{\mathrm{x}}}{\Delta_{\mathrm{x}}},0\right)^2+\max\left(\frac{T-T_{\mathrm{y}}}{\Delta_{\mathrm{y}}},0\right)^2=\frac{1}{F_{\mathrm{ij}}^2} \tag{3.6}$$

Since it is assumed that the speed of the front is positive ($F > 0$), T must be greater than T_{x} and T_{y} whenever the front wave has not already visited the point at coordinates (i, j). Therefore, it is safe to simplify Equation 3.6 to:

$$\left(\frac{T-T_{\mathrm{x}}}{\Delta_{\mathrm{x}}}\right)^2+\left(\frac{T-T_{\mathrm{y}}}{\Delta_{\mathrm{y}}}\right)^2=\frac{1}{F_{\mathrm{ij}}^2} \tag{3.7}$$

Equation 3.7 is a regular quadratic equation of the form $aT^2 + bT + c = 0$, where:

$$\begin{aligned} a &= \Delta_{\mathrm{x}}^2+\Delta_{\mathrm{y}}^2 \\ b &= -2(\Delta_{\mathrm{y}}^2T_{\mathrm{x}}+\Delta_{\mathrm{x}}^2T_{\mathrm{y}}) \\ c &= \Delta_{\mathrm{y}}^2T_{\mathrm{x}}^2+\Delta_{\mathrm{x}}^2T_{\mathrm{y}}^2-\tfrac{\Delta_{\mathrm{x}}^2\Delta_{\mathrm{y}}^2}{F_{\mathrm{ij}}^2} \end{aligned} \tag{3.8}$$

In order to simplify the notation for the n-dimensional case, let us assume that the grid is composed by (hyper)cubic cells, that is, $\Delta_{\mathrm{x}} = \Delta_{\mathrm{y}} = \Delta_{\mathrm{z}} = \cdots = h$. Let us denote T_{d} as the generalization of T_{x} or T_{y} for dimension d, up to N dimensions. F denotes the propagation velocity for point with coordinates $(i, j, k, \dots)$. Operating and simplifying terms, the discretization of the Eikonal is a quadratic equation with parameters:

$$\begin{aligned} a &= N \\ b &= -2\sum_{d=1}^{N}T_{\mathrm{d}} \\ c &= (\sum_{d=1}^{N}T_{\mathrm{d}}^2)-\tfrac{h^2}{F^2} \end{aligned} \tag{3.9}$$

3.3.2 Solving the nD discrete Eikonal equation

Wavefront propagation follows causality. That is, in order to reach a point with higher time of arrival, it should firstly travel through neighbors of such point with smaller values. The opposite would imply a jump in time continuity and therefore the solutions would be erroneous.

The proposed Eikonal solution (quadratic equation with parameters of Equation 3.9) does not guarantee the causality of the resulting distance map, as F and h can

have arbitrary values. Therefore, before accepting a solution as valid its causality has to be checked. For instance, in 2D the Eikonal is solved as:

$$T = \frac{T_\mathrm{x} + Ty}{2} + \frac{1}{2}\sqrt{\frac{2h^2}{F^2} - \big(T_\mathrm{x} - T_\mathrm{y}\big)^2} \tag{3.10}$$

called the *two-sided* update, as both parents T_x and T_y are taken into account. The solution is only accepted if $T \geq \max\big(T_\mathrm{x}, T_\mathrm{y}\big)$. The *upwind condition* [78] shows that:

$$T \geq \max\big(T_\mathrm{x}, T_\mathrm{y}\big) \Longleftrightarrow \big|T_\mathrm{x} - T_\mathrm{y}\big| \leq \frac{h}{F} \tag{3.11}$$

If this condition fails, the *one-sided update* is applied instead:

$$T = \min\big(T_\mathrm{x}, T_\mathrm{y}\big) + \frac{h}{F} \tag{3.12}$$

This is a top-down approach: the parents are iteratively discarded until a causal solution is found. To generalize Equation 3.11 is complex. Therefore, a bottom-up approach is chosen: Equation 3.12 is solved and parents are iteratively included until the time of the next parent is higher than the current solution: $T_k > T$. The procedure is detailed in Algorithms 1 and 2. The MinTDim() function returns the minimum time of the neighbors in a given dimension (left and right for $dim = 1$, bottom and top for $dim = 2$, etc.). The experiments found this approach more robust for 3 or more dimensions with negligible impact on the computational performance.

Algorithm 1 Solve Eikonal Equation

```
1:  procedure SolveEikonal(x_i, T, F)
2:      a ← N
3:      for dim = 1 : N do
4:          min_T ← MinTDim(dim)
5:          if min_T ≠ ∞ and min_T < T_i then
6:              T_values.push(min_T)
7:          else
8:              a ← a − 1
9:      if a = 0 then                ▷ Fast Sweeping Method can cause this situation.
10:         return ∞
11:     T_values ← Sort(T_values)
12:     for dim = 1 : a do
13:         T̃_i ← SolveNDims(x_i, dim, T_values, F)
14:         if dim = a or T̃_i < T_values,dim+1 then
15:             break
16:     return T̃_i
```

Algorithm 2 Solve Eikonal for n dimensions

procedure SOLVENDIMS($\mathbf{x}_\mathrm{i}, dim, T_{\texttt{values}}, \mathcal{F}$)
  if $dim = 1$ **then**
    return $T_{\texttt{values},1} + \frac{h}{F_\mathrm{i}}$
  $sumT \leftarrow \sum_{i=1}^{dim} T_{\texttt{values},i}$
  $sumT^2 \leftarrow \sum_{i=1}^{dim} T^2_{\texttt{values},i}$
  $a \leftarrow dim$
  $b \leftarrow -2sumT$
  $c \leftarrow sumT^2 - \frac{h^2}{F_\mathrm{i}}$
  $q \leftarrow b^2 - 4ac$
  if $q < 0$ **then** ▷ Non-causal solution
    return ∞
  else
    return $\frac{-b+sqrt(q)}{2a}$

3.4 Fast Marching Method

The Fast Marching Method (FMM) [22] is the most common Eikonal solver. It can be classified as a label-setting, Dijkstra-like algorithm [18]. It uses a first-order upwind-finite difference scheme to simulate an isotropic front propagation. The main difference with Dijkstra's algorithm is the operation carried out on every node. Dijkstra's algorithm is designed to work on graphs. Therefore, the value for every node $\mathbf{x}_\mathrm{i}$ only depends on one parent $\mathbf{x}_\mathrm{j}$, following the Bellman's optimality principle [86]:

$$T_\mathrm{i} = \min_{\mathbf{x}_\mathrm{i} \in \mathcal{N}(\mathbf{x}_\mathrm{i})} (c_{ij} + T_\mathrm{j}) \tag{3.13}$$

In other words, a node $\mathbf{x}_\mathrm{i}$ is connected to the parent $\mathbf{x}_\mathrm{j}$ in its neighborhood $\mathcal{N}(\mathbf{x}_\mathrm{i})$ which minimizes (or maximizes) the function value (in this case T_i) composed by the value of T_j plus the addition of the cost of *traveling* from $\mathbf{x}_\mathrm{j}$ to $\mathbf{x}_\mathrm{i}$, represented as c_{ij}.

The FMM follows Bellman's optimality principle but the value for every node is computed following first-order upwind discretization of the Eikonal equation, which is described in detail in Section 3.3. This discretization takes into account the spatial representation (i.e. a rectangular grid) and the value of all the causal upwind neighbors. Thus, the time-of-arrival field computed by FMM is more accurate than Dijkstra's.

The algorithm labels the cells in three different sets: 1) `Frozen`: those cells which value is computed and cannot change, 2) `Unknown`: cells with no value assigned, to

be evaluated, and 3) `Narrow` band (or just `Narrow`): frontier between `Frozen` and `Unknown` containing those cells with a value assigned that can be improved. These sets are mutually exclusive, that is, a cell cannot belong to more than one of them at the same time. The implementation of the `Narrow` set is a critical aspect of FMM. A more detailed discussion will be carried out in Section 4.2.1.

The procedure is detailed in Algorithm 3. Initially, all points[2] in the grid belong to the `Unknown` set with infinite arrival time. The initial points (wave sources) are assigned a value 0 and introduced in `Frozen` (lines 2-7). Then, the main FMM loop starts by choosing the element with minimum arrival time from `Narrow` (line 9). All its non-`Frozen` neighbors are evaluated: for each of them the Eikonal is solved and the new arrival time value is kept if it is improved. In case the cell is in `Unknown`, it is transferred to `Narrow` (lines 10-16). Finally, the previously chosen point from `Narrow` is transferred to `Frozen` (lines 17 and 18) and a new iteration starts until the `Narrow` set is empty. The arrival times map $\mathcal{T}$ is returned as the result of the procedure.

3.5 Path planning with the Fast Marching Method

Analyzing the FMM formulation given in 3.3, it can be seen that there are many common components with the path planning problem formulation detailed in Chapter 2. Therefore, it is possible to solve the path planning method with FMM, but also with any of the algorithms detailed in the next Chapter as they all share the same formulation.

The configuration space $\mathcal{X}$ corresponds with the domain $\mathcal{X}$ of the FMM (reason why they are named the same). $\mathcal{X}_{\text{obs}}$ corresponds to the subset of $\mathcal{X}$ and represents those points in the space in which the wave cannot propagate. Although theoretically obstacles and zero-velocity cells are different, in practice they are treated the same as they output the same infinite value as the wave never reach such point. Consequently, $\mathcal{X}_{\text{free}}$ contains the rest of the cells.

For path planning, it is assumed that a $\mathcal{X}_{\text{goal}} \subset \mathcal{X}_{\text{free}}$ occupies at least one cell of the configuration space discretization. In other words, a well-behaved goal set will be in practice at least of the size of a cell. Analogously, the start set $\mathcal{X}_{\text{s}} \subset \mathcal{X}_{\text{free}}$ is also represented by at least one cell.

In order to compute the path, the wave is propagated from $\mathcal{X}_{\text{s}}$ to $\mathcal{X}_{\text{goal}}$, obtaining a time-of-arrival map $\mathcal{T}$. Applying gradient descent over $\mathcal{T}$ from $\mathcal{X}_{\text{goal}}$, it is satisfied that $\sigma(1) \in \text{cl}(\mathcal{X}_{\text{goal}})$. Gradient descent will compute the path to the unique minimum of $\mathcal{T}$ and therefore $\sigma(0) = \mathcal{X}_{\text{s}}$. In terms of implementation, this will actually return the path inverted, as its waypoints will travel from $\mathcal{X}_{\text{goal}}$ to $\mathcal{X}_{\text{s}}$. Thus the wave is often propagated from $\mathcal{X}_{\text{goal}}$ or the path is inverted. However, this has no effect in

[2]From now on, point, cell or node will indistinctly used to refer to each element of the grid.

Algorithm 3 Fast Marching Method

procedure FMM($\mathcal{X}, \mathcal{T}, \mathcal{F}, \mathcal{X}_\mathrm{s}$)
Initialization:
  $\texttt{Unknown} \leftarrow \mathcal{X}, \texttt{Narrow} \leftarrow \emptyset, \texttt{Frozen} \leftarrow \emptyset$
  $T_\mathrm{i} \leftarrow \infty \ \forall \mathbf{x}_\mathrm{i} \in \mathcal{X}$
  for $\mathbf{x}_\mathrm{i} \in \mathcal{X}_\mathrm{s}$ **do**
    $T_\mathrm{i} \leftarrow 0$
    $\texttt{Unknown} \leftarrow \texttt{Unknown} \backslash \{\mathbf{x}_\mathrm{i}\}$
    $\texttt{Narrow} \leftarrow \texttt{Narrow} \cup \{\mathbf{x}_\mathrm{i}\}$

Propagation:
  while $\texttt{Narrow} \neq \emptyset$ **do**
    $\mathbf{x}_\mathrm{min} \leftarrow \arg\min_{\mathbf{x}_\mathrm{i} \in \texttt{Narrow}} \{T_\mathrm{i}\}$ ▷ `Narrow` top operation.
    for $\mathbf{x}_\mathrm{i} \in (\mathcal{N}(\mathbf{x}_\mathrm{min}) \cap \mathcal{X} \backslash \texttt{Frozen})$ **do** ▷ For all neighbors not in Frozen.
      $\widetilde{T}_\mathrm{i} \leftarrow \textsc{SolveEikonal}(\mathbf{x}_\mathrm{i}, \mathcal{T}, \mathcal{F})$
      if $\widetilde{T}_\mathrm{i} < T_\mathrm{i}$ **then**
        $T_\mathrm{i} \leftarrow \widetilde{T}_\mathrm{i}$ ▷ `Narrow` increase operation if $\mathbf{x}_\mathrm{i} \in$ `Narrow`.
      if $\mathbf{x}_\mathrm{i} \in$ `Unknown` **then** ▷ `Narrow` push operation.
        $\texttt{Narrow} \leftarrow \texttt{Narrow} \cup \{\mathbf{x}_\mathrm{i}\}$
        $\texttt{Unknown} \leftarrow \texttt{Unknown} \backslash \{\mathbf{x}_\mathrm{i}\}$
    $\texttt{Narrow} \leftarrow \texttt{Narrow} \backslash \{\mathbf{x}_\mathrm{min}\}$ ▷ `Narrow` pop operation: add to Frozen.
    $\texttt{Frozen} \leftarrow \texttt{Frozen} \cup \{\mathbf{x}_\mathrm{min}\}$
  return $\mathcal{T}$

the path, as isotropic FMM are being considered.

Regarding optimality, the cost function is defined as $c = \mathcal{T}$. That is, the time of arrival of each cell actually represents its cost from the start point. FMM guarantees that the path returned by gradient descent are optimal, as every cell has the lowest possible value T_{i} assigned, and therefore there is not better alternative to reach that cell.

Concretely, the maximum gradient direction is computed applying the Sobel operator over the grid map.

$$grad_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \star \mathcal{T} \qquad grad_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \star \mathcal{T} \tag{3.14}$$

For tracing the path between the initial and the goal points the maximum gradient direction has to be follow starting at the initial point. The path is computed iteratively. $grad_{ix}$ and $grad_{iy}$ are computed at every point p_i. From p_i is computed p_{i+1} (Equation 3.15) successively until the minimum is reached. The step size ($step$) is user-defined. Thus one advantage of FMM is that the extracted paths have a large number of points, a useful feature when implementing path following on a real robot. As the goal point is located at the global minima it is always reached (whenever there is path).

$$\begin{aligned} mod_i &= \sqrt{grad_{ix}^2 + grad_{iy}^2} \\ alpha_i &= \arctan(\tfrac{grad_{iy}}{grad_{ix}}) \\ p_{(i+1)x} &= p_{ix} + step \cdot \cos(alpha_i) \\ p_{(i+1)y} &= p_{iy} + step \cdot \sin(alpha_i) \end{aligned} \tag{3.15}$$

Figure 3.3 shows an example of a path planning problem solved with FMM in 2D. The velocity map $\mathcal{F}$ is a binary map: every cell in $\mathcal{X}_{\text{free}}$ has velocity 1 (white) and $\mathcal{X}_{\text{obs}}$ is represented with cells with zero velocity (black). Usually, obstacles are dilated in order to provide minimum safety guarantees, but this is very application dependent.

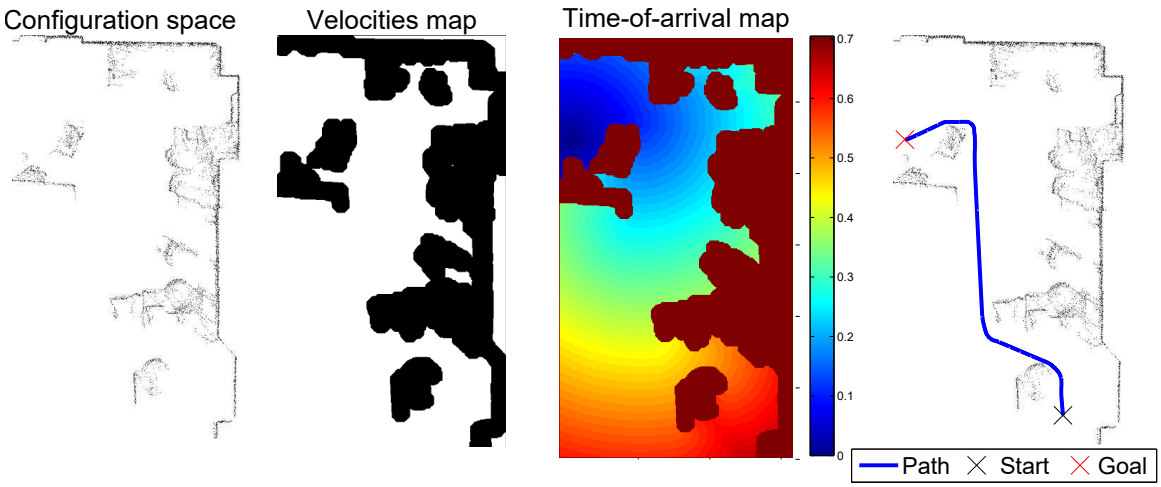


Figure 3.3: Example of a path planning problem solved with Fast Marching.

The main drawbacks of Fast Marching-based planning methods are: 1) robot dimensions are not explicitly taken into account, and 2) kinematic constraints are not taken into account. 1) is partially solved in practice by obstacle dilation as shown in Figure 3.3. Sometimes, if enough computation resources are available, the obstacle set in the grid already takes the shape of the robot into account, commonly by using the maximum radius of the robot projection on the ground or by including a third dimension to the configuration space in order to represent the yaw (heading angle) of the robot. However, 2) is still an open issue. Some approaches to include kinodynamic constraints have been proposed. For instance, [87] uses a two-step approach which first computes a geodesic-based path initialization (Fast Marching can be thought as a geodesic finder) and then an optimization procedure based on Bézier curves is applied to satisfy robot's kinodynamic constraints. Or [88] which samples the space mixing a wavefront-propagation schema and forward-simulation of the system's kinematics and dynamics. However, these approaches represent a mixture of problems and further testing is required before using them in real robotic applications.

Chapter 4

Fast Methods

4.1 Introduction

Although this Thesis is focusing on FMM, many other different approaches exists, providing the same (or very similar) output. There are referred as Fast Methods, as they are inspired in FMM but more computationally efficient.

In this Chapter, nine sequential single-threaded, isotropic, grid-based Fast Methods are detailed: Fast Marching Method (FMM), Fibonacci-Heap FMM (FMMFib), Simplified FMM (SFMM), Untidy FMM (UFMM), Group Marching Method (GMM), Fast Iterative Method (FIM), Fast Sweeping Method (FSM), Lock Sweeping Method (LSM) and Double Dynamic Queue Method (DDQM). All these algorithms provide exactly the same solution except for UFMM and FIM, which have bounded errors. However, the question of which one is the best for which applications is still open. For example, [2] compares only FMM and FSM in spite of the fact that GMM and UFMM were already published. Survey [6] mentions most of the algorithms but only compares FMM and SFMM. A more recent work compares FMM, FSM and FIM in 2D [4]. However, FIM is paralellized and implemented in CUDA providing a biased comparison. Figure 4.1 schematically shows the comparisons among algorithms carried out in the literature. Methods such as UFMM have been barely compared to their counterparts while others as FMM and FSM are compared in many works despite the fact that it is well known when each perform better: FSM is faster in simple environments with constant velocity. In addition, results from one work cannot be directly extrapolated to other works since the performance of these methods highly depend on their implementation.

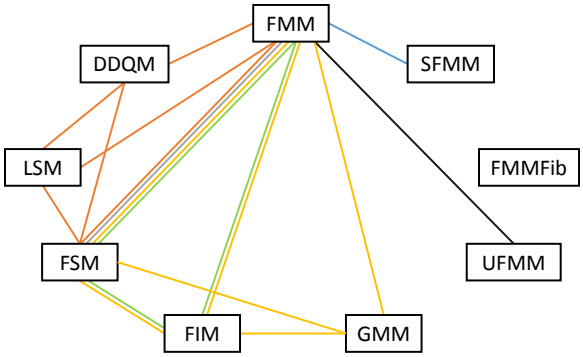


Figure 4.1: Comparisons among algorithms. Colors refer to different works: orange [1], gray [2], yellow [3], green [4], black [5], and blue [6].

This Chapter has three different contributions: 1) Based on previous works, a common formulation and notation is given for all the algorithms following the previous Chapter formulation. This way, it is possible to easily understand the working principles and mathematical formulation. More detailed formulations are available in

the literature but this Chapter focuses on a practical perspective, as the rest of this Thesis. 2) A recent survey of the work on designing sequential Fast Methods during the last years. Parallel or high-accuracy approaches are not taken into account as these fields are dense enough to fill another survey. 3) Extensive and systemic comparison among the mentioned methods with experiments designed taking into account their applications and results previously reported.

The two-scale methods are proposed in [78], namely Fast Marching-Sweeping Method (FMSM), Heap Cell Method (HCM), and Fast Heap Cell Method (FHCM). They combine the FMM and FSM in order obtain the best features of both algorithms, dividing the grid into two different levels and performing marching on a coarser scale and then sweeping on a finer scale. However, these methods have not been included in this analysis for different reasons: 1) HCM and FHCM performance depend on the discretization of the coarse grid, where the optimal parameter depends on the velocities profile. Furthermore, FHCM includes additional error. 2) FMSM error is not mathematically bounded. Thus, the comparison with other Fast Methods becomes more complex. Additionally, an efficient n-dimensional implementation of these methods is overly complex, even given the fact that they are based on other Fast Methods.

Additionally, single-pass methods proposed in [89] have not been included in this Chapter because, as the authors acknowledge, it is not always possible to know in advance which method among the proposed should be used. This is an important drawback for practical applications such as robotics.

This Chapter is organized as follows: next Section includes Fast Marching-like algorithms, Section 4.3 Fast Sweeping-based, and Section 4.4 contains other Fast Methods. The benchmark and its results are included in Section 4.5, followed by a discussion in Section 4.6. Finally, Section 4.7 outlines the conclusions and proposes future works.

4.2 Fast Marching Methods

4.2.1 Binary and Fibonacci Heaps

FMM requires the implementation of the `Narrow` set to have four different operations: 1) Push: to insert a new element to the set, 2) Increase: to reorder an element already existing in the set which value has been improved, 3) Top: retrieve the element with minimum value, and 4) Pop: remove the element with minimum value. As stated before, this is the most critical aspect of the FMM implementation. The most efficient way to implement `Narrow` is by using a min-heap data structure. A heap is an ordered tree in which every parent is ordered with respect to its children. In a min-heap, the minimum value is at the root of the tree and the children have higher values. This is

satisfied for any parent node of the tree.

Among all the existing heaps, FMM is usually implemented with a binary heap [90]. However, the Fibonacci Heap [91] has a better amortized time for Increase and Push operations, but it has additional computational overhead with respect to other heaps. For relatively small grids, where the narrow band is composed by few elements and the performance is still far from its asymptotic behavior, the binary heap performs better. Table 4.1 summarizes the time complexities for these heaps[1] (the priority queue will be detailed in Section 4.2.2). Note that n is the number of cells in the map, as the worst case is to have all the cells in the heap.

Table 4.1: Summary of amortized time complexities for common heaps used in FMM (n is the number of elements in the heap).

	Push	Increase	Top	Pop
Fibonacci	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
Binary	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
Priority Queue	$\mathcal{O}(\log n)$	–	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

Each cell is pushed and popped at most once in the heap. For each loop, the top of `Narrow` is accessed ($\mathcal{O}(1)$), the Eikonal is solved for at most 2^N neighbors ($\mathcal{O}(1)$ for a given N), these cells are pushed or increased ($\mathcal{O}(\log n)$ in the worst case), and finally the top cell is popped ($\mathcal{O}(\log n)$). Therefore each loop is at most $\mathcal{O}(\log n)$. Since this loop is executed at most n times, the total FMM complexity is $\mathcal{O}(n \log n)$, where n represents the total number of cells of the grid in the worst case scenario.

4.2.2 Simplified Fast Marching Method

The Simplified Fast Marching Method (SFMM) [6] is a relatively unknown variation of the standard FMM but with an impressive performance. SFMM, detailed in Algorithm 4, is a reduced version of FMM where `Narrow`, implemented as a simple priority queue, can contain different instances of the same cell with different values. Additionally, it can happen that the same cell belongs to `Narrow` and `Frozen` at the same time. The simplification occurs since no Increase operation is required. Every time a cell has an updated value, it is pushed to the priority queue. Once it is popped and inserted in `Frozen`, the remaining instances in the queue are simply ignored.

The advantage is that all the increase operations are substituted by push operations. Although both have the same computational complexity, the constant for push is much lower (increase requires removal and Push operations). Note that the computational complexity is maintained, $\mathcal{O}(n \log n)$.

[1] http://bigocheatsheet.com/

Algorithm 4 Simplified Fast Marching Method

procedure SFMM($\mathcal{X}, \mathcal{T}, \mathcal{F}, \mathcal{X}_\mathrm{s}$)
Initialization as FMM (Algorithm 3)

Propagation:
  while `Narrow` $\neq \emptyset$ **do**
    $\mathbf{x}_\mathrm{min} \leftarrow \arg\min_{\mathbf{x}_\mathrm{i} \in \texttt{Narrow}} \{T_\mathrm{i}\}$ ▷ `Narrow` top operation.
    if $\mathbf{x}_\mathrm{min} \in$ `Frozen` **then**
      `Narrow` $\leftarrow$ `Narrow`$\backslash\{\mathbf{x}_\mathrm{min}\}$
    else
      for $\mathbf{x}_\mathrm{i} \in (\mathcal{N}(\mathbf{x}_\mathrm{min}) \cap \mathcal{X} \backslash$ `Frozen`$)$ **do** ▷ For all neighbors not in Frozen.
        $\widetilde{T}_\mathrm{i} \leftarrow$ SolveEikonal$(\mathbf{x}_\mathrm{i}, \mathcal{T}, \mathcal{F})$
        if $\widetilde{T}_\mathrm{i} < T_\mathrm{i}$ **then** ▷ Update arrival time.
          $T_\mathrm{i} \leftarrow \widetilde{T}_\mathrm{i}$
          `Narrow` $\leftarrow$ `Narrow` $\cup \{\mathbf{x}_\mathrm{i}\}$ ▷ `Narrow` push operation.
        if $\mathbf{x}_\mathrm{i} \in$ `Unknown` **then**
          `Unknown` $\leftarrow$ `Unknown`$\backslash\{\mathbf{x}_\mathrm{i}\}$
      `Narrow` $\leftarrow$ `Narrow`$\backslash\{\mathbf{x}_\mathrm{min}\}$ ▷ `Narrow` pop operation.
      `Frozen` $\leftarrow$ `Frozen` $\cup \{\mathbf{x}_\mathrm{min}\}$
  return $\mathcal{T}$

4.2.3 Untidy Fast Marching Method

The Untidy Fast Marching Method (UFMM) [5, 92] follows exactly the same procedure as FMM. However, a special heap structure is used which reduces the overall computational complexity to $\mathcal{O}(n)$: the untidy priority queue.

This untidy priority queue is closer to a look-up table than to a tree. It assumes that the $\mathcal{F}$ values are bounded, hence the $\mathcal{T}$ values are also bounded. The untidy queue, depicted in Figure 4.2, is a circular array which divides the maximum range of $\mathcal{T}$ into a set of k consecutive buckets. Each bucket contains an unordered list of cells with similar T_{i} value. The threshold values for each bucket evolve with the algorithm, trying to maintain an uniform distribution of the elements in `Narrow` among the buckets.

Since the index of the corresponding bucket can be analytically computed, Push is $\mathcal{O}(1)$ as well as Top. Pop and Increase operation are, in average, $\mathcal{O}(1)$ as long as $\#_{buckets} < \mathcal{O}(n)$. Therefore, the total UFMM complexity is $\mathcal{O}(n)$. However, since elements within a bucket are not sorted (FIFO strategy is applied in each bucket), errors are being introduced in the final result. In fact, it is shown that the accumulated additional error is bounded by $\mathcal{O}(h)$ (h is the cell size), which is the same order of magnitude as in the original FMM.

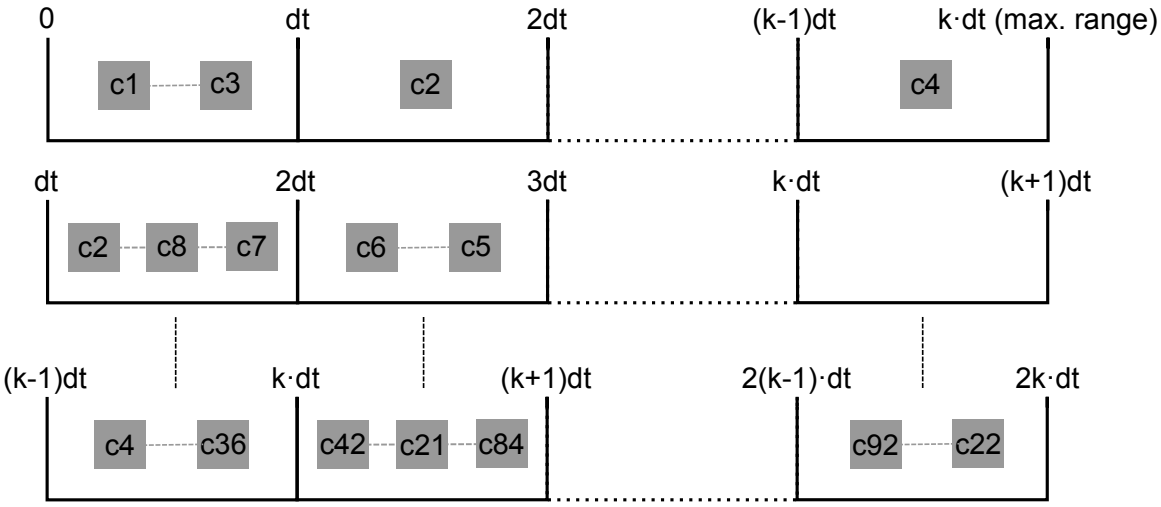


Figure 4.2: Untidy priority queue representation. Top: first iteration, the four neighbors of the initial point are pushed. Middle: the first bucket became empty, so the circular array advances one position. Cell c2 is first evaluated because it was the first pushed in the bucket. Bottom: after a few iterations, a complete loop on the queue is about to be completed.

4.3 Fast Sweeping Methods

The Fast Sweeping Method (FSM) [93, 94] is an iterative algorithm which computes the time-of-arrival map by successively *sweeping* (traversing) the whole grid following a specific order. FSM performs Gauss-Seidel iterations in alternating directions. These directions are chosen so that all the possible characteristic curves of the solution to the Eikonal are divided into the possible quadrants (or octants in 3D) of the environment. For instance, a bi-dimensional grid has 4 possible Gauss-Seidel iterations (the combinations of traversing x and y dimensions forwards and backwards): are North-East, North-West, South-East and South-West, as shown in Figure 4.3.

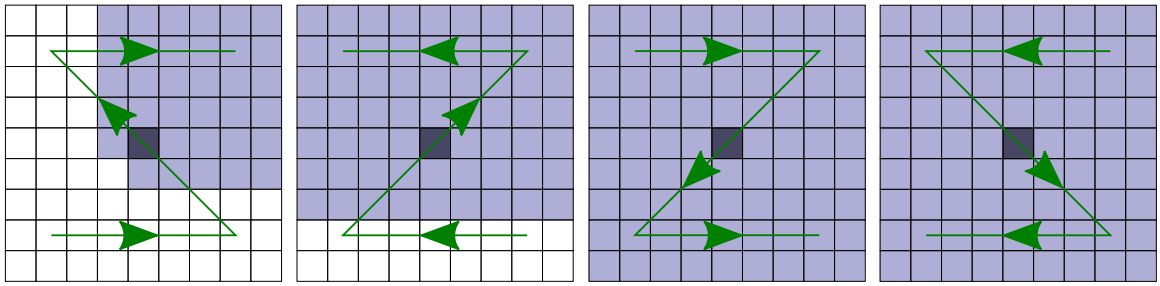


Figure 4.3: FSM sweep directions in 2D represented with arrows. The darkest cell is the initial point and the shaded cells are those analyzed by the current sweep (time improved or maintained).

The FSM is a simple algorithm: it performs sweeps until no value is improved. In each sweep, the Eikonal equation is solved for every cell. However, to generalize this algorithm to N-dimensions is complex. Up to author's knowledge, there are only 2D and 3D implementations. However, Algorithm 5 introduces a N-dimensional version. Sweeping directions are denoted as a binary array `SweepDirs` with elements 1 or -1, with 1 (-1) meaning forwards (backwards) traversal in that dimension. This array is initialized to 1 (North-East in the 2D case or North-East-Top in 3D) and the grid is initialized as in FMM (lines 2-5). The main loop updates `SweepDirs` and a sweep is performed in the new direction (lines 8-9).

The GetSweepDirs() procedure (see Algorithm 6) is in charge of generating the appropriate Gauss-Seidel iteration directions. If a 3D `SweepDirs` $= [1, 1, 1]$ vector is given, the following sequence will be generated:

$$\begin{array}{lll} 1-[-1,-1,-1] & 4-[1,1,-1] & \\ 2-[1,-1,-1] & 5-[-1,-1,1] & 7-[-1,1,1] \\ 3-[-1,1,-1] & 6-[1,-1,1] & 8-[1,1,1] \end{array} \tag{4.1}$$

Note that this sequence creates a sweep pattern which is not exactly the same as detailed in the literature, but it is equally valid as the same directions are visited and the same number of sweeps are required to cover the whole grid.

Finally, the SWEEP() procedure (see Algorithm 7) recursively generates the Gauss-Seidel iterations following the traversal directions specified by the corresponding value of `SweepDirs` (line 4). Each recursive level traverses the whole corresponding dimension. The extent of dimension n is denoted as $\mathcal{X}_{\mathrm{n}}$. Once the most inner loop is reached, the corresponding cell is evaluated and its value updated if necessary (lines 7-11).

Algorithm 5 Fast Sweeping Method

1: **procedure** FSM($\mathcal{X}, \mathcal{T}, \mathcal{F}, \mathcal{X}_{\mathrm{s}}$)
Initialization.
2: `SweepDirs` $\leftarrow [1, \ldots, 1]$ ▷ Initialize sweeping directions.
3: $T_{\mathrm{i}} \leftarrow \infty \; \forall \mathbf{x}_{\mathrm{i}} \in \mathcal{X}$
4: **for** $\mathbf{x}_{\mathrm{i}} \in \mathcal{X}_{\mathrm{s}}$ **do**
5: $T_{\mathrm{i}} \leftarrow 0$

Propagation:
6: `stop` $\leftarrow$ `False`
7: **while** `stop` $\neq$ `True` **do**
8: `SweepDirs` $\leftarrow$ GETSWEEPDIRS($\mathcal{X}$, `SweepDirs`)
9: `stop` $\leftarrow$ SWEEP($\mathcal{X}, \mathcal{T}, \mathcal{F}$, `SweepDirs`, N)
10: **return** $\mathcal{T}$

Algorithm 6 Sweep directions algorithm

1: **procedure** GETSWEEPDIRS($\mathcal{X}$, `SweepDirs`)
2: **for** $i = 1 : N$ **do**
3: `SweepDirs`$_{\mathrm{i}} \leftarrow$ `SweepDirs`$_{\mathrm{i}} + 2$
4: **if** `SweepDirs`$_{\mathrm{i}} \leq 1$ **then**
5: **break** ▷ Finish For loop.
6: **else**
7: `SweepDirs`$_{\mathrm{i}} \leftarrow -1$
8: **return** `SweepDirs`

The FSM carries out as many grid traversals as necessary until the value T_{i} for every cell has converged. Since no ordering is implied, the evaluation of each cell is $\mathcal{O}(1)$. As there are n cells, the total computational complexity of FSM is $\mathcal{O}(n)$. However, note that the constants highly depend on the velocity function $F(\mathbf{x})$. In the

Algorithm 7 Recursive sweeping algorithm

```
1: procedure SWEEP(𝒳, 𝒯, ℱ, SweepDirs, n)
2:     stop ← True
3:     if n > 1 then
4:         for i ∈ 𝒳_n following SweepDirs_n do
5:             stop ← SWEEP(𝒳, 𝒯, ℱ, SweepDirs, n − 1)
6:     else
7:         for i ∈ 𝒳_1 following SweepDirs_1 do
8:             T̃_i ← SOLVEEIKONAL(x_i, 𝒯, ℱ)        ▷ x_i is the corresponding cell.
9:             if T̃_i < T_i then
10:                T_i ← T̃_i
11:                stop ← False
12:    return stop
```

case of an empty environment with constant $F(\mathbf{x})$, only 2^N sweeps will be required as the characteristic directions are straight lines. However, for environment with obstacles or complex velocities functions, where the characteristics directions change frequently, the number of sweeps required can be much higher and therefore FSM will take longer to return a solution. Note as well that the $\mathcal{T}$ returned by FSM is exactly the same as all the FMM-like algorithms (except UFMM).

4.3.1 Lock Sweeping Methods

The Lock Sweeping Method (LSM) [1] is a natural improvement over FSM. The FSM might spend computation time recomputing T_i even if none of the neighbors of $\mathbf{x}_\mathrm{i}$ has improved their value since the last sweep. LSM labels a cell as *unlocked* if any of its neighbors has changed and thus its value can be improved. Otherwise, the cell is labeled as *locked* and it will be skipped.

The LSM procedure is detailed in Algorithm 8. It is basically the same as FSM but with the addition of tracking if a point is locked (`Frozen`) or unlocked (`Narrow`). Analogously, the LOCKSWEEP() (see Algorithm 9) procedure is similar to SWEEP() with two differences: 1) if a point is not unlocked it is skipped (see line 8), and 2) neighbors of $\mathbf{x}_\mathrm{i}$ are unlocked if the new value T_i is better that their current value.

Note that the asymptotic computational complexity of FSM is kept, $\mathcal{O}(n)$. The number of required sweeps is also maintained. However, in practice it turns out that most of the cells are locked during a sweep. Therefore, the computation time saved is important.

Algorithm 8 Lock Sweeping Method

procedure LSM($\mathcal{X}, \mathcal{T}, \mathcal{F}, \mathcal{X}_s$)
Initialization.
  $\texttt{Frozen} \leftarrow \mathcal{X}, \texttt{Narrow} \leftarrow \emptyset$
  $T_i \leftarrow \infty \ \forall \mathbf{x}_i \in \mathcal{X}$
  $\texttt{SweepDirs} \leftarrow [1, \ldots, 1]$ ▷ Initialize sweeping directions.
  for $\mathbf{x}_i \in \mathcal{X}_s$ **do**
    $T_i \leftarrow 0$
    for $\mathbf{x}_j \in \mathcal{N}(\mathbf{x}_i)$ **do** ▷ Unlocking neighbors of starting cells.
      $\texttt{Frozen} \leftarrow \texttt{Frozen} \backslash \{\mathbf{x}_j\}$
      $\texttt{Narrow} \leftarrow \texttt{Narrow} \cup \{\mathbf{x}_j\}$

Propagation:
  $\texttt{stop} \leftarrow \texttt{False}$
  while $\texttt{stop} \neq \texttt{True}$ **do**
    $\texttt{SweepDirs} \leftarrow$ getSweepDirs($\mathcal{X}, \texttt{SweepDirs}$)
    $\texttt{stop} \leftarrow$ LockSweep($\mathcal{X}, \mathcal{T}, \mathcal{F}, \texttt{SweepDirs}, N$)
  return $\mathcal{T}$

4.4 Other Fast Methods

4.4.1 Group Marching Method

The Group Marching Method (GMM) [95] is an FMM-based Eikonal solver which solves a group of grid points in `Narrow` at once, instead of sorting them in a heap structure.

Consider a front propagating. At a given time, `Narrow` will be composed by the set of cells belonging to the wavefront. GMM selects a group G out of `Narrow` composed by the global minimum and the local minima in `Narrow`. Then, every neighboring cell to G is evaluated and added to `Narrow`. These points in G have to be chosen carefully so that causality is not violated since GMM does not sort the `Narrow` set. For that, GMM selects those points following Equation 4.2:

$$G = \{\mathbf{x}_i \in \texttt{Narrow} : T_i \leq \min(T_{\texttt{Narrow}}) + \delta_\tau\} \tag{4.2}$$

where

$$\delta_\tau = \frac{1}{\max(\mathcal{F})} \tag{4.3}$$

In the original GMM work [95] $\delta_\tau = \frac{h}{\max(\mathcal{F})\sqrt{N}}$. However, Equation 4.3 is referred

Algorithm 9 Recursive sweeping algorithm

procedure LockSweep($\mathcal{X}, \mathcal{T}, \mathcal{F}, \texttt{SweepDirs}, n$)
  $\texttt{stop} \leftarrow \texttt{True}$
  if $n > 1$ **then**
    for $i \in \mathcal{X}_{\mathrm{n}}$ following $\texttt{SweepDirs}_{\mathrm{n}}$ **do**
      $\texttt{stop} \leftarrow$ LockSweep($\mathcal{X}, \mathcal{T}, \mathcal{F}, \texttt{SweepDirs}, n-1$)
  else
    for $i \in \mathcal{X}_1$ following $\texttt{SweepDirs}_1$ **do**
      if $\mathbf{x}_{\mathrm{i}} \in \texttt{Narrow}$ **then**
        $\widetilde{T}_{\mathrm{i}} \leftarrow$ SolveEikonal($\mathbf{x}_{\mathrm{i}}, \mathcal{T}, \mathcal{F}$) ▷ $\mathbf{x}_{\mathrm{i}}$ is the corresponding cell.
        if $\widetilde{T}_{\mathrm{i}} < T_{\mathrm{i}}$ **then**
          $T_{\mathrm{i}} \leftarrow \widetilde{T}_{\mathrm{i}}$
          $\texttt{stop} \leftarrow \texttt{False}$
          for $\mathbf{x}_{\mathrm{j}} \in \mathcal{N}(\mathbf{x}_{\mathrm{i}})$ **do**
            if $T_{\mathrm{i}} < T_{\mathrm{j}}$ **then** ▷ Add improvable neighbors to Narrow.
              $\texttt{Frozen} \leftarrow \texttt{Frozen} \backslash \{\mathbf{x}_{\mathrm{j}}\}$
              $\texttt{Narrow} \leftarrow \texttt{Narrow} \cup \{\mathbf{x}_{\mathrm{j}}\}$
        $\texttt{Narrow} \leftarrow \texttt{Narrow} \backslash \{\mathbf{x}_{\mathrm{i}}\}$ ▷ Add $\mathbf{x}_{\mathrm{i}}$ to Frozen.
        $\texttt{Frozen} \leftarrow \texttt{Frozen} \cup \{\mathbf{x}_{\mathrm{i}}\}$
  return stop

in [3]. Although this second formula is not mathematically proven, the results for the original δ_τ are much worse than FMM in most of the cases, reaching one order of magnitude of difference.

If the time difference between two adjacent cells is larger than δ_τ, their values will barely affect each other since the wavefront propagation direction is more perpendicular than parallel to the line segment formed by both cells. However, the downwind points (those to be evaluated in future iterations) can be affected by both adjacent cells. Therefore, points in G are evaluated twice to avoid instabilities.

GMM is detailed in Algorithm 10. Its initialization is FMM-like. Note that δ_τ depends on the maximum velocity value in the grid. The main loop updates the threshold T_m every iteration. Firstly, it carries out a reverse traversal through the selected points, computing and updating their value (lines 17-21). Then, lines 22-31 perform a forward traversal with the same operations as the reverse traversal but updating the `Narrow` and `Frozen` sets in the same way as FMM.

Note that GMM returns the same solution as FMM. GMM evaluates twice every node before inserting it in `Frozen` while FMM only evaluates it once. However, GMM does not require any sorting. Therefore, GMM is an $\mathcal{O}(n)$ iterative algorithm that converges in only 2 iterations (traversals). The value of δ_τ can be modified: higher δ_τ would require more iterations to converge. However, smaller δ_τ will require also 2 traversals but the group G will be composed by fewer cells. As GMM authors point out, GMM can be interpreted as an intermediary point between FMM ($\delta_\tau = 0$) and a purely iterative method [96] ($\delta_\tau = \infty$).

Additionally, a generalized N-dimensional implementation is straightforward.

4.4.2 Double Dynamic Queue Method

The Double Dynamic Queue Method (DDQM) [1] is inspired in LSM but resembles to GMM. DDQM is conceptually simple. `Narrow` is divided into two non-sorted FIFO queues: one with cells to be evaluated sooner and the other one with cells to be evaluated later. Every iteration takes an element from the first queue and evaluates it. If the time is improved, the neighboring cells with higher time are unlocked and added to the first or second queue depending on the value of the cell updated. Once the first queue is empty, queues are swapped and the algorithm continues. The purpose is to achieve a pseudo-ordering of the cells, so that cells with lower value are evaluated first.

Since queues are not sorted, it could require to solve many times the same cell until its value converges. DDQM dynamically computes the threshold value depending on the number of points inserted in each queue, trying to reach an equilibrium. The original paper includes an important analysis about this threshold update. Initially,

Algorithm 10 Group Marching Method

procedure GMM($\mathcal{X}, \mathcal{T}, \mathcal{F}, \mathcal{X}_\mathrm{s}$)
Initialization:
  $\texttt{Unknown} \leftarrow \mathcal{X}, \texttt{Narrow} \leftarrow \emptyset, \texttt{Frozen} \leftarrow \emptyset$
  $T_\mathrm{i} \leftarrow \infty \ \forall \mathbf{x}_\mathrm{i} \in \mathcal{X}$
  $\delta_\tau \leftarrow \frac{1}{\max(\mathcal{F})}$
  for $\mathbf{x}_\mathrm{i} \in \mathcal{X}_\mathrm{s}$ **do**
    $T_\mathrm{i} \leftarrow 0$
    $\texttt{Unknown} \leftarrow \texttt{Unknown} \backslash \{\mathbf{x}_\mathrm{i}\}$
    $\texttt{Frozen} \leftarrow \texttt{Frozen} \cup \{\mathbf{x}_\mathrm{i}\}$
    for $\mathbf{x}_\mathrm{j} \in \mathcal{N}(\mathbf{x}_\mathrm{i})$ **do** ▷ Adding neighbors of starting points to Narrow.
      $T_\mathrm{i} \leftarrow \textsc{SolveEikonal}(\mathbf{x}_\mathrm{j}, \mathcal{T}, \mathcal{F})$
      if $T_\mathrm{i} < T_\mathrm{m}$ **then**
        $T_\mathrm{m} \leftarrow T_\mathrm{i}$
      $\texttt{Unknown} \leftarrow \texttt{Unknown} \backslash \{\mathbf{x}_\mathrm{i}\}$
      $\texttt{Narrow} \leftarrow \texttt{Narrow} \cup \{\mathbf{x}_\mathrm{i}\}$

Propagation:
  while $\texttt{Narrow} \neq \emptyset$ **do**
    $T_\mathrm{m} \leftarrow T_\mathrm{m} + \delta_\tau$
    for $\mathbf{x}_\mathrm{i} \in (\texttt{Narrow} \leq T_\mathrm{m})$ `REVERSE` **do** ▷ Reverse traversal.
      for $\mathbf{x}_\mathrm{j} \in (\mathcal{N}(\mathbf{x}_\mathrm{i}) \cap \mathcal{X} \backslash \texttt{Frozen})$ **do**
        $\widetilde{T}_\mathrm{i} \leftarrow \textsc{SolveEikonal}(\mathbf{x}_\mathrm{j}, \mathcal{T}, \mathcal{F})$
        if $\widetilde{T}_\mathrm{i} < T_\mathrm{i}$ **then**
          $T_\mathrm{i} \leftarrow \widetilde{T}_\mathrm{i}$
    for $\mathbf{x}_\mathrm{i} \in (\texttt{Narrow} \leq T_\mathrm{m})$ `FORWARD` **do** ▷ Forward traversal.
      for $\mathbf{x}_\mathrm{j} \in (\mathcal{N}(\mathbf{x}_\mathrm{i}) \cap \mathcal{X} \backslash \texttt{Frozen})$ **do**
        $\widetilde{T}_\mathrm{i} \leftarrow \textsc{SolveEikonal}(\mathbf{x}_\mathrm{j}, \mathcal{T}, \mathcal{F})$
        if $\widetilde{T}_\mathrm{i} < T_\mathrm{i}$ **then**
          $T_\mathrm{i} \leftarrow \widetilde{T}_\mathrm{i}$
        if $\mathbf{x}_\mathrm{i} \in \texttt{Unknown}$ **then**
          $\texttt{Unknown} \leftarrow \texttt{Unknown} \backslash \{\mathbf{x}_\mathrm{i}\}$
          $\texttt{Narrow} \leftarrow \texttt{Narrow} \cup \{\mathbf{x}_\mathrm{i}\}$
      $\texttt{Narrow} \leftarrow \texttt{Narrow} \backslash \{\mathbf{x}_\mathrm{i}\}$
      $\texttt{Frozen} \leftarrow \texttt{Frozen} \cup \{\mathbf{x}_\mathrm{i}\}$
  return $\mathcal{T}$

the step value of the threshold is increased every iteration and is computed as:

$$step = \frac{1.5hn}{\sum_i F_{\mathrm{i}}} \tag{4.4}$$

where n is the total number of cells in the grid. Originally, this step was proposed as $step = \frac{1.5n}{h\sum_i \frac{1}{F_{\mathrm{i}}}}$. However, step should have time units and this expression has $[t^{-1}]$ units (probably an error due to the ambiguity of using speed F or slowness $f = \frac{1}{F}$). Therefore, Equation 4.4 is proposed.

Every time the first queue is empty, UpdateStep() (see Algorithm 11) is called, with the value of the current *step*, c_1, and c_{total} which are the number of cells inserted in the first queue and the total number of cells inserted, correspondingly. *step* is modified so that the number of cells inserted in the first queue is between 65% and 75% of the total inserted cells. This is a conservative approach, since the closer this percentage is to 50% the faster DDQM is. However, the penalization for percentages lower than 50% is much important than for higher percentages.

Note that the step is increased by a factor 1.5 but decreased by a factor of 2. This makes *step* to converge to a value instead of overshooting around the optimal value. Dividing by a larger number causes the first queue to become empty earlier. Thus, next iteration will finish faster and a better *step* value can be computed.

Algorithm 11 DDQM Threshold Increase

```
1: procedure UpdateStep(step, c_1, c_total)
2:     m ← 0.65
3:     M ← 0.75
4:     Perc ← 1
5:     if c_1 > 0 then
6:         Perc ← c_1 / c_total
7:     if Perc ≤ m then
8:         step ← step * 1.5
9:     else if Perc ≥ M then
10:        step ← step / 2
11:    return step
```

DDQM is detailed in Algorithm 12. As in LSM, points are locked (`Frozen`) or unlocked (`Narrow`). Initialization sets all points as frozen except the neighbors of the start points, which are added to the first queue (lines 2-12). While first queue is not empty, its front element is extracted and evaluated (lines 14-16). If its value is improved, all its locked neighbors with higher value are unlocked and added to its corresponding queue.

Algorithm 12 Double Dynamic Queue Method

procedure DDQM($\mathcal{X}, \mathcal{T}, \mathcal{F}, \mathcal{X}_\mathrm{s}$)
Initialization:
  $\texttt{Frozen} \leftarrow \mathcal{X}, \texttt{Narrow} \leftarrow \emptyset$
  $\mathcal{Q}_1 \leftarrow \emptyset, \mathcal{Q}_2 \leftarrow \emptyset, c_1 \leftarrow 0, c_\mathrm{total} \leftarrow 0$
  $step = \frac{1.5hn}{\sum_i F_\mathrm{i}}$ ▷ n is the total number of cells.
  $th \leftarrow step$
  $T_\mathrm{i} \leftarrow \infty \ \forall \mathbf{x}_\mathrm{i} \in \mathcal{X}$
  for $\mathbf{x}_\mathrm{i} \in \mathcal{X}_\mathrm{s}$ **do**
    $T_\mathrm{i} \leftarrow 0$
    for $\mathbf{x}_\mathrm{j} \in \mathcal{N}(\mathbf{x}_\mathrm{i})$ **do**
      $\mathcal{Q}_1 \leftarrow \mathcal{Q}_1 \cup \{\mathbf{x}_\mathrm{j}\}$
      $\texttt{Unknown} \leftarrow \texttt{Unknown} \backslash \{\mathbf{x}_\mathrm{i}\}$
      $\texttt{Narrow} \leftarrow \texttt{Narrow} \cup \{\mathbf{x}_\mathrm{i}\}$
Propagation:
  while $\mathcal{Q}_1 \neq \emptyset$ **or** $\mathcal{Q}_2 \neq \emptyset$ **do**
    while $\mathcal{Q}_1 \neq \emptyset$ **do**
      $\mathbf{x}_\mathrm{i} \leftarrow \mathcal{Q}_1.\textsc{front}()$ ▷ Extracts the front element.
      $\widetilde{T}_\mathrm{i} \leftarrow \textsc{SolveEikonal}(\mathbf{x}_\mathrm{i}, \mathcal{T}, \mathcal{F})$
      if $\widetilde{T}_\mathrm{i} < T_\mathrm{i}$ **then**
        $T_\mathrm{i} \leftarrow \widetilde{T}_\mathrm{i}$
        for $\mathbf{x}_\mathrm{j} \in (\mathcal{N}(\mathbf{x}_\mathrm{i}) \cap \texttt{Frozen})$ **do**
          if $T_\mathrm{i} < T_\mathrm{j}$ **then** ▷ Add improvable neighbors to queue.
            $\texttt{Frozen} \leftarrow \texttt{Frozen} \backslash \{\mathbf{x}_\mathrm{j}\}$
            $\texttt{Narrow} \leftarrow \texttt{Narrow} \cup \{\mathbf{x}_\mathrm{j}\}$
            $c_\mathrm{total} \leftarrow c_\mathrm{total} + 1$
            if $T_\mathrm{i} \leq th$ **then**
              $\mathcal{Q}_1 \leftarrow \mathcal{Q}_1 \cup \{\mathbf{x}_\mathrm{j}\}$
              $c_1 \leftarrow c_1 + 1$
            else
              $\mathcal{Q}_2 \leftarrow (\mathcal{Q}_2 \cup \{\mathbf{x}_\mathrm{j}\})$
      $\texttt{Narrow} \leftarrow \texttt{Narrow} \backslash \{\mathbf{x}_\mathrm{i}\}$
      $\texttt{Frozen} \leftarrow \texttt{Frozen} \cup \{\mathbf{x}_\mathrm{i}\}$
    $step \leftarrow \textsc{UpdateStep}(step, c_1, c_\mathrm{total})$
    $\textsc{swap}(\mathcal{Q}_1, \mathcal{Q}_2)$
    $c_1 \leftarrow 0, c_\mathrm{total} \leftarrow 0$
    $th \leftarrow th + step$
  return $\mathcal{T}$

In the original work, three methods were proposed: 1) single-queue (SQ), and therefore simpler algorithm, 2) two-queue static (TQS), where the *step* is not updated, and 3) two-queue dynamic (called DDQM). SQ and TQS slightly improve DDQM in some experiments, but when DDQM improves SQ and TQS (for instance environments with noticeable speed changes) the difference can reach one order of magnitude. Therefore, DDQM was included instead of SQ and TQS since it has shown a more adaptive behaviour. In any case, any of these methods return the same solution as FMM.

In the worst case, the whole grid is contained in both queues and traversed many times during the propagation. However, since queue insertion and deletion are $\mathcal{O}(1)$ operations, the overall complexity is $\mathcal{O}(n)$. Note that SWAP() can be efficiently implemented in $\mathcal{O}(1)$ as a circular binary index, or updating references (or pointers). There is not need for a real swap operation.

4.4.3 Fast Iterative Method

The Fast Iterative Method (FIM) [3] is based on the iterative method proposed by [96] but inspired in FMM. It also resembles to DDQM (concretely to its single queue variant). It iteratively evaluates every point in `Narrow` until it converges. Once a node has converged its neighbors are inserted into `Narrow` and the process continues. *Narrow* is implemented as a non-sorted list. The algorithm requires a convergence parameter ϵ: if T_i is improved less than ϵ, it is considered converged. FIM has also been proposed for triangulated surfaces [97].

FIM is designed to be efficient for parallel computing, since all the elements in `Narrow` can be evaluated simultaneously. However, this Chapter focuses on its sequential implementation in order to have a fair comparison with other methods.

Algorithm 13 details FIM. Its initialization is the same as FMM. Then, for each element in `Narrow`, its value is updated (lines 11-12). If the value difference is less than ϵ, the neighbors are evaluated and added to `Narrow` in case their value is improved (lines 13-19). Since `Narrow` is a list, the new elements should be inserted just before the point being currently evaluated, $\mathbf{x}_\mathrm{i}$. Finally, this point is removed from `Narrow` and labeled as `Frozen` (lines 20 and 21).

A node can be added several times to `Narrow` during FIM execution, since every time an upwind (parent) neighbor is updated, the node can improve its value. In the worst case, `Narrow` contains the whole grid and the loop would go through all the points several times. Operations on the list are $\mathcal{O}(1)$. Therefore, the overall computational complexity of FIM is $\mathcal{O}(n)$.

For a small enough ϵ (depending on the environment), FIM will return the same solution as FMM. However, it can be speed up allowing small errors bounded by ϵ.

Algorithm 13 Fast Iterative Method

procedure FIM($\mathcal{X}, \mathcal{T}, \mathcal{F}, \mathcal{X}_\mathrm{s}, \epsilon$)
Initialization:
  $\texttt{Frozen} \leftarrow \mathcal{X}, \texttt{Narrow} \leftarrow \emptyset$
  $T_\mathrm{i} \leftarrow \infty \ \forall \mathbf{x}_\mathrm{i} \in \mathcal{X}$
  for $\mathbf{x}_\mathrm{i} \in \mathcal{X}_\mathrm{s}$ **do**
    $T_\mathrm{i} \leftarrow 0$
    for $\mathbf{x}_\mathrm{j} \in (\mathcal{N}(\mathbf{x}_\mathrm{i}) \cap \texttt{Unknown})$ **do**
      $\texttt{Frozen} \leftarrow \texttt{Frozen} \backslash \{\mathbf{x}_\mathrm{i}\}$
      $\texttt{Narrow} \leftarrow \texttt{Narrow} \cup \{\mathbf{x}_\mathrm{i}\}$

Propagation:
  while $\texttt{Narrow} \neq \emptyset$ **do**
    for $\mathbf{x}_\mathrm{i} \in \texttt{Narrow}$ **do**
      $\widetilde{T}_\mathrm{i} \leftarrow T_\mathrm{i}$
      $T_\mathrm{i} \leftarrow \textsc{SolveEikonal}(\mathbf{x}_\mathrm{i}, \mathcal{T}, \mathcal{F})$
      if $\left|T_\mathrm{i} - \widetilde{T}_\mathrm{i}\right| < \epsilon$ **then**
        for $\mathbf{x}_\mathrm{j} \in (\mathcal{N}(\mathbf{x}_\mathrm{i}) \cap \texttt{Frozen})$ **do**
          $\widetilde{T}_\mathrm{j} \leftarrow \textsc{SolveEikonal}(\mathbf{x}_\mathrm{j}, \mathcal{T}, \mathcal{F})$
          if $\widetilde{T}_\mathrm{j} < T_\mathrm{j}$ **then**
            $T_\mathrm{j} \leftarrow \widetilde{T}_\mathrm{j}$
            $\texttt{Frozen} \leftarrow \texttt{Frozen} \backslash \{\mathbf{x}_\mathrm{j}\}$
            $\texttt{Narrow} \leftarrow \texttt{Narrow} \cup \{\mathbf{x}_\mathrm{j}\}$ ▷ Insert in the list just before $\mathbf{x}_\mathrm{i}$
        $\texttt{Narrow} \leftarrow \texttt{Narrow} \backslash \{\mathbf{x}_\mathrm{i}\}$
        $\texttt{Frozen} \leftarrow \texttt{Frozen} \cup \{\mathbf{x}_\mathrm{i}\}$
  return $\mathcal{T}$

4.5 Experimental Comparison

4.5.1 Experimental setup

In order to give an impartial and meaningful comparison, all the algorithms have been implemented from scratch, in C++11 using the Boost.Heap library[2]. An automatic benchmarking application has also been created so that the experiments are carried out and evaluated in the most systematic possible way.

This implementation is focused on time performance and compiled using G++ 4.9.2 with optimizations flag -Ofast. However, no special optimizations have been included. All algorithms use the same primitive functions for grid and cell computations. The times reported correspond to an Ubuntu 14.04 64 bits computer running in a Dual Core 3.3 GHz with 4Gb of RAM. However, all experiments were carried out in one core. Only propagation times are taken into account. The computation time used in the initialization has been omitted since it can be done offline, besides it is similar for all algorithms and represents a little percentage of the total computation time.

Since the algorithms are deterministic, the deviation in the computation time between different runs is theoretically 0. In fact, this deviation mostly depends on the OS scheduler and not on the algorithm, as this will perform the exact same number of operations in all runs. However, the results shown are the mean of 10 runs for every algorithm, so that the deviation of the results is practically 0.

For UFMM, the default parameters are a maximum range of $\mathcal{T}$ of 2 units and 1000 buckets (the checkerboard experiment required different parameters, see Section 4.5.1). The ϵ parameter for FIM is set to 0 (actually 10^{-47} to provide robust 64bit double comparison).

Although error analysis is not in the scope of this Chapter, it can be compared among the existing papers since it is implementation-independent. UFMM errors are reported in those experiments with non-constant velocity. Usually, L_1 and L_∞ norms of the error are reported. Most of the works compute norm L_1 as:

$$|\mathcal{T}|_1 = \sum_{\mathbf{x}_\mathrm{i} \in \mathcal{X}} |T_\mathrm{i}| \tag{4.5}$$

where $\mathcal{X}$ is treated as a regular vector. However, following [78], numerical solutions are treated as elements of L_p spaces (generalization of the p-norm to vector spaces), where L_1 norm is defined as an integral over the function. The result is a norm closely related to its physical meaning and independent of the cell size. L_1 is numerically integrated over the domain and therefore computed as (assuming hypercubic cells

[2]The source code is available at https://github.com/jvgomez/fastmarching

and grids):

$$|\mathcal{T}|_1 = \sum_{\mathbf{x}_\mathrm{i} \in \mathcal{X}} |T_\mathrm{i} h^N| = h^N \sum_{\mathbf{x}_\mathrm{i} \in \mathcal{X}} |T_\mathrm{i}| \tag{4.6}$$

Four different experiments have been carried out, which represent the most characteristic cases for the Fast Methods. They have been chosen so that the advantages and disadvantages of each algorithm are remarked. By combining these problems it is possible to get close to any situation. These experiments were chosen attending also to the most common situations tested in the literature.

Empty map

This experiment is designed to show the performance of the methods in the most basic situation, where most of the algorithms perform best. An empty map with constant velocity represents the simplest possible case for the Fast Methods. In fact, analytical methods could be implemented by computing the euclidean distance from every point to the initial point. However, it is interesting because it shows the performance of the algorithms on open spaces which, in a real application, can be part of large environments.

The same environment is divided into a different number of cells to study how the algorithms behave as the number of cell increases. Composed by an empty 2D, 3D and 4D hyper-cubical environment of size $[0, 1]^N$, with $N = 2, 3, 4$. Constant velocity $F_\mathrm{i} = 1$ on $\mathcal{X}$. The wavefront starts at the center of the grid.

The number of cells was chosen so that an experiment has the same (or as close as possible) number of cells in all dimensions. For instance, a 50x50 2D grid has 2500 cells. Therefore, the equivalent 3D grid is 14x14x14 (2744) and in 4D is 7x7x7x7 (2401). This way, it is possible to also analyze the performance of the algorithms for a different number of dimensions. Thus, the following number of cells for each dimension for 2D grid have been chosen:

$$2D : \{50, 100, 200, 400, 800, 1000, 1500, 2000, 2500, 3000, 4000\}$$

Consequently, the 3D and 4D cells are:

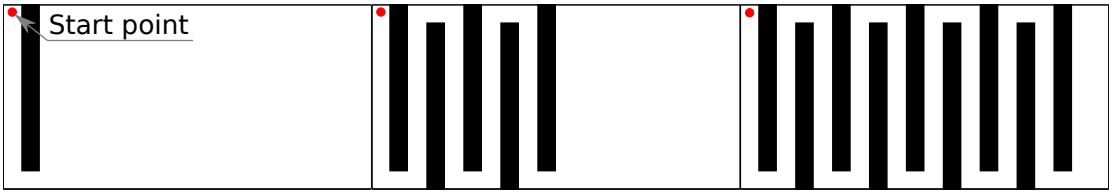
$$\begin{aligned} 3D :&\{14, 22, 34, 54, 86, 100, 131, 159, 184, 208, 252\} \\ 4D :&\{7, 10, 14, 20, 28, 32, 39, 45, 450, 55, 63\} \end{aligned}$$

Alternating barriers

In this case, the objective is to analyze how the algorithms behave with obstacles ($F_\mathrm{i} = 0$) in a constant velocity environment ($F_\mathrm{i} = 1$). The obstacles cause the

characteristics to change.

The experiment contains a 2D environment of constant size $[0, 1]$x$[0, 2]$ discretized in a 1000x2000 grid. A variable number of alternating barriers are equally distributed along the longest dimension. The number of barriers goes from 0 to 9. Examples are shown in Figure 4.4. Analogously, in 3D a $[0, 1]$x$[0, 1]$x$[0, 2]$ environment represented by a 100x100x200 is chosen, with equally-distributed alternating barriers (from 0 to 9) along the z axis. The wavefront starts in all cases close to a corner of the map.



(a) 1 barrier. (b) 5 barriers. (c) 9 barriers.

Figure 4.4: 2D alternating barriers environments.

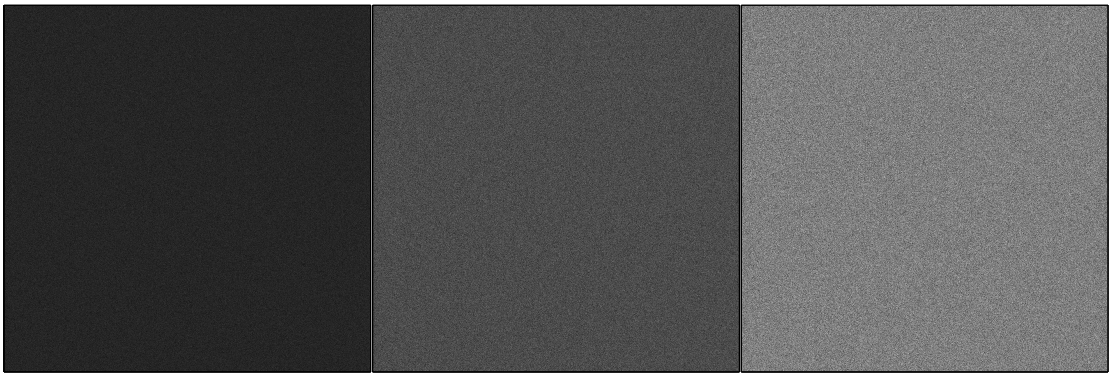
Random velocities

This experiment aims to test the performance of the algorithms with random velocities (similar to noisy images as in the case of medical computer vision). It creates a 2D, 3D and 4D environment of size $[0, 1]^N$ with $N = 2, 3, 4$ discretized in a 2000x2000 grid in 2D, 159x159x159 in 3D and 45x45x45x45 in 4D. These discretizations are chosen so that it is possible to compare directly with the empty map problem for the corresponding grid sizes. The wavefront starts in the center of the grid.

Additionally, the maximum velocity is increased from 10 to 100 (in steps of 10 units) to analyze how the algorithms behave with increasing velocity changes. 2D examples are shown in Figure 4.5.

Checkerboard

The random velocities experiment already tested changes in velocities. However, those are high-frequency changes because it is unlikely to have 2 adjacent cells with the same velocitiy. In this experiment low-frequency changes are studied. The same environment and discretizations as in random velocities are now divided like a checkerboard, alternating minimum and maximum velocities. Analogously, the maximum velocity is increased from 10 to 100, while the minimum velocity is always 1. There are 10 checkerboard divisions on each dimension. The wavefront starts in the center of the grid. 2D examples are shown in Figure 4.6.



(a) Max. velocity = 30 (b) Max. velocity = 60. (c) Max. velocity = 100.

Figure 4.5: 2D random velocities environments. Lighter color means faster wave propagation.

In this case, UFMM in 3D and 4D performed very poorly with the default parameters. Additional tests not reported show that the best parameters for UFMM are approximately 1000 buckets with a maximum range of 0.01 in 3D. In the 4D case, 20000 buckets and maximum range of 0.025 are used.

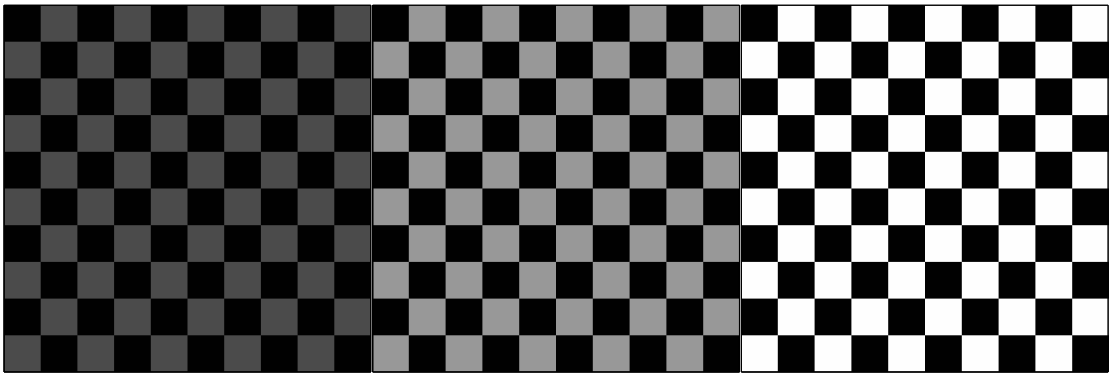
4.5.2 Results

Empty map

An example of the time-of-arrival field computed by FMM is shown in Figure 4.7. Note that all algorithms provide the same exact solution in this case. The higher the resolution the better the accuracy.

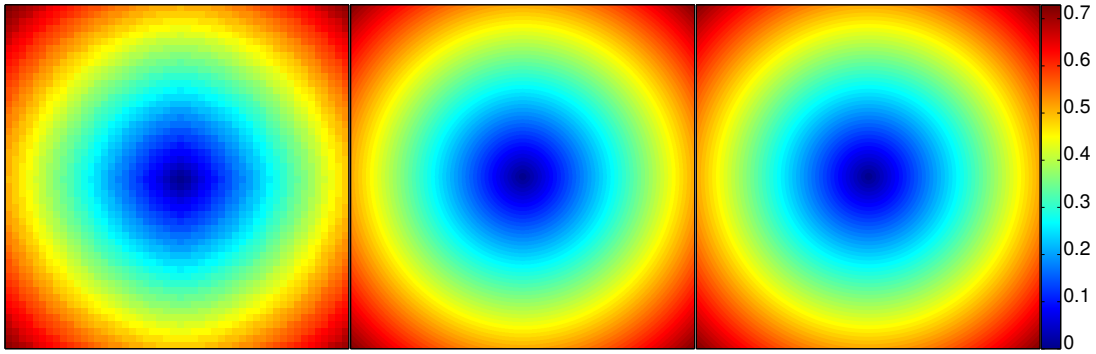
The results for the empty map experiment are shown in Figure 4.8 for 2D, Figure 4.9 for 3D, and Figure 4.10 for 4D. In all cases 2 plots are included: raw computation times for each algorithm, and time ratios computed as:

$$ratio = \frac{\text{Alg. Time}}{\text{FMM Time}} \tag{4.7}$$



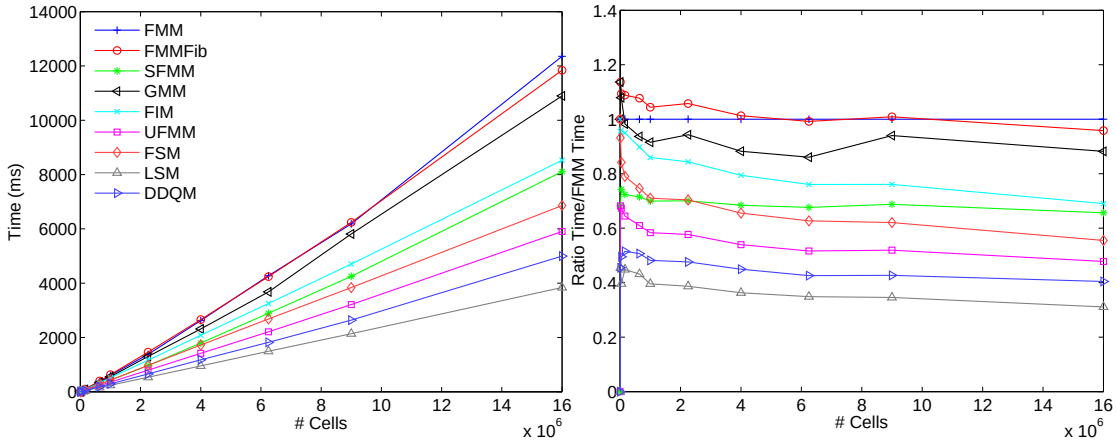
(a) Max. velocity = 30 (b) Max. velocity = 60. (c) Max. velocity = 100.

Figure 4.6: 2D checkerboard environments. Lighter color means faster wave propagation.



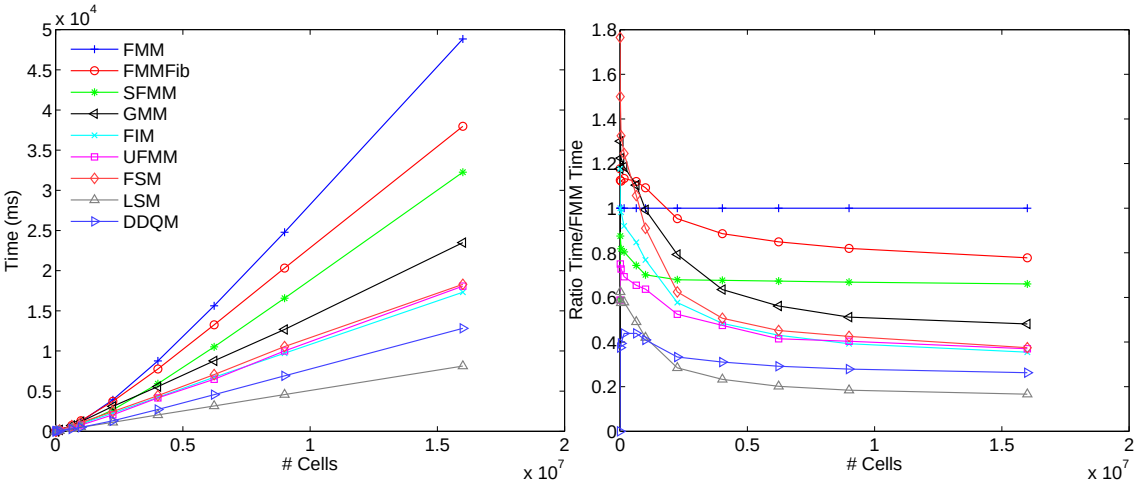
(a) 50x50 (b) 800x800 (c) 4000x4000

Figure 4.7: Example of the resulting time-of-arrival maps applying FMM to the empty environment in 2D.



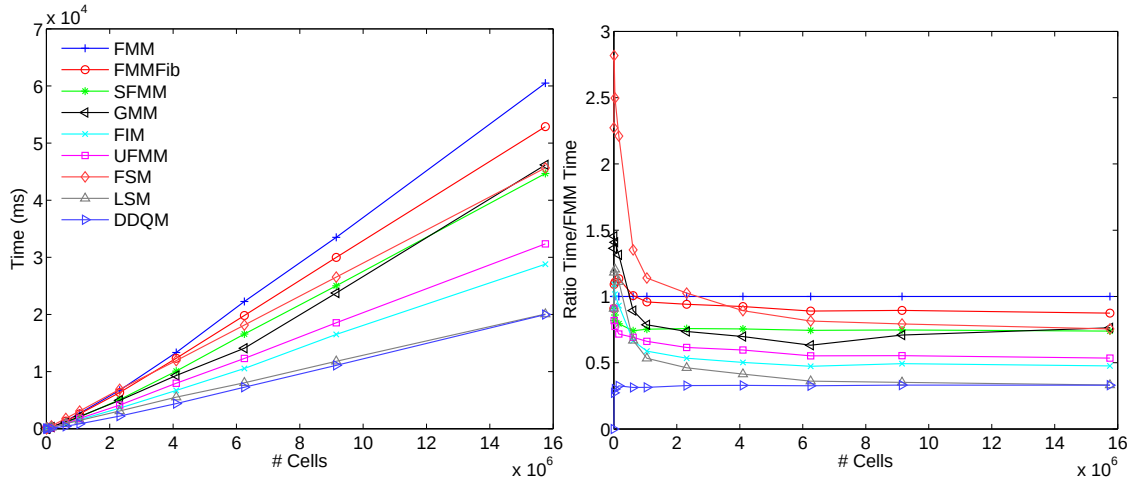
(a) Computation times. (b) Time ratios against FMM.

Figure 4.8: Computation times and ratios for the empty map environment in 2D.



(a) Computation times. (b) Time ratios against FMM.

Figure 4.9: Computation times and ratios for the empty map experiment in 3D.



(a) Computation times. (b) Time ratios against FMM.

Figure 4.10: Computation times and ratios for the empty map experiment in 4D.

In all cases, both LSM and DDQM are the fastest algorithms. DDQM tends to perform better in smaller grids, specially en 4D. This was an expected result since this is the case in which they perform less sweeps (queue recomputations in DDQM). Besides, FSM is slower than UFMM in all cases, and than FIM in 3D and 4D. As velocity is constant all over the grid, UFMM provides the same solution as other methods.

GMM slightly improves FMM and FMMFib in 2D. However in higher dimensions this improvement becomes larger, but it always performs worse than UFMM, FIM, DDQM and LSM. In a previous comparison between GMM and FMM [3], GMM was

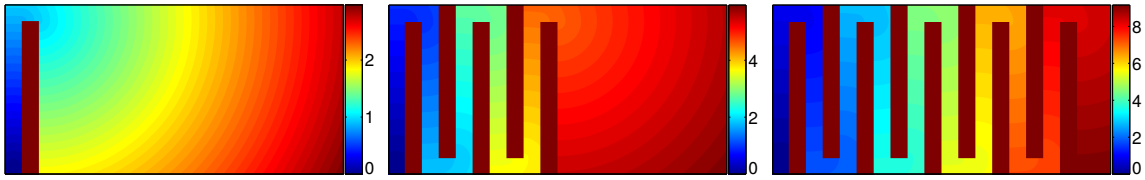
about 50% faster than FMM in all cases than FMM. In this results GMM is at most 40% better. This difference is attributed to implementation, as the heaps for FMM and FMMFib are highly optimized. FIM is always faster than GMM as it only needs one iteration trough the narrow band, while GMM always performs two.

SFMM results are of special interest since it is a minor modification of FMM but, however, highly outperforms FMM in all cases, and even FIM in 2D and small 3D grids. In 4D FIM becomes faster.

As expected, FMMFib is worse than FMM for almost all sizes in the 2D case. However, when dimensions increase FMMFib quickly outperforms FMM as the number of elements in the narrow band increases exponentially with the number of dimensions, therefore the better amortized times of Fibonacci heap become useful.

Alternating barriers

An example of FMM results in some of the alternating barriers environment is shown in Figure 4.11, while performance results (times and ratios) for 2D and 3D are shown in Figure 4.12 and Figure 4.13 correspondingly.

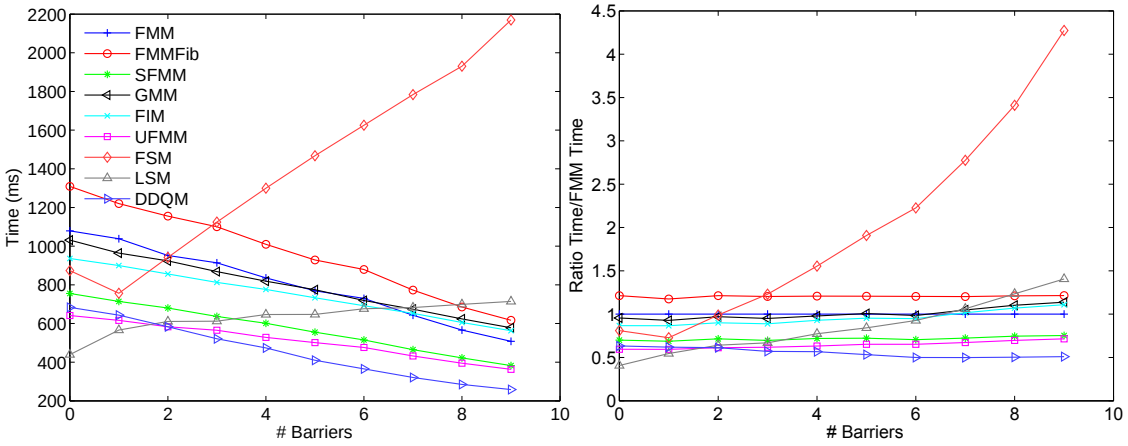


(a) 1 barrier. (b) 5 barriers. (c) 9 barriers.

Figure 4.11: Example of the resulting time-of-arrival maps applying FMM to some of the alternating barriers environment in 2D.

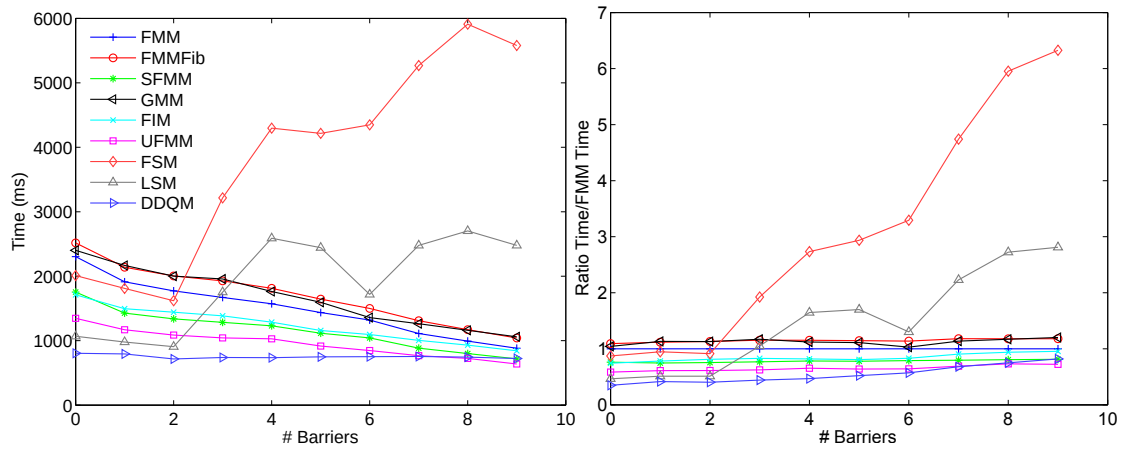
In this case, results are very similar to the empty map experiment. However, as the number of barriers increases and the environment becomes more complex, the characteristics changes their directions more often and thus more sweeps are required. Therefore, FSM and LSM decrease their performance despite the fact that the more barriers the less cells are to evaluate. This is the reason why all other algorithms tend to lower times as the number of barriers increases. However, LSM is still faster than some algorithms in most cases, as its computational overhead is low even though it performs many sweeps. DDQM also suffers from map complexity, however it affects much less and only in 3D.

UFMM provides again the same solution as other methods. It is the second fastest algorithm behind DDQM, closely followed by SFMM and FIM.



(a) Computation times. (b) Time ratios against FMM.

Figure 4.12: Computation times and ratios for the alternating barriers experiment in 2D.

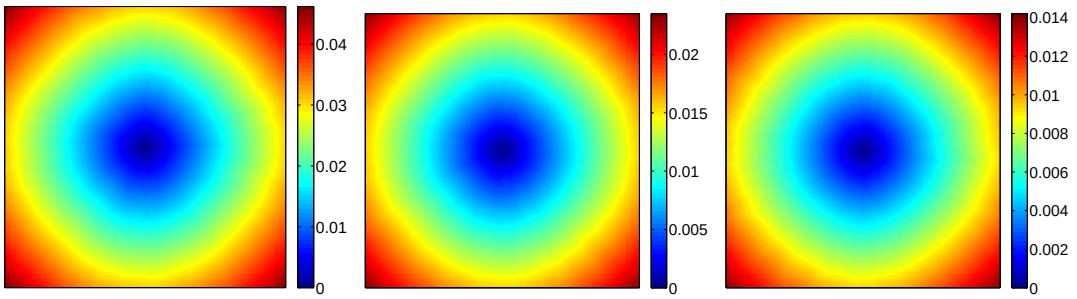


(a) Computation times. (b) Time ratios against FMM.

Figure 4.13: Computation times and ratios for the alternating barriers experiment in 3D.

Random velocities

The output of the FMM for the random velocity map is apparently close to the one of the empty map (Figure 4.14), but with the wavefronts slightly distorted because of the velocity changes.



(a) Max. velocity=30 (b) Max. velocity=60 (c) Max. velocity=100

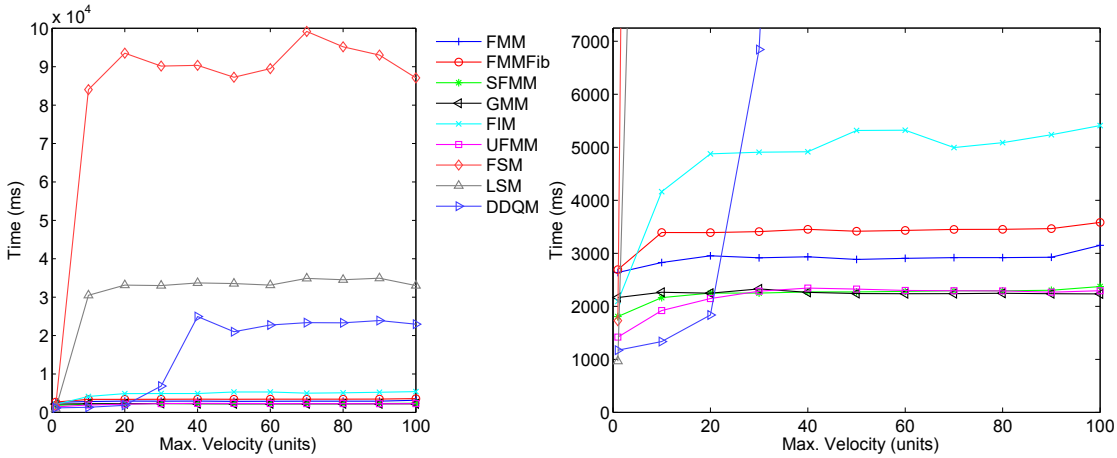
Figure 4.14: Example of the resulting time-of-arrival maps applying FMM to the random velocities environment in 2D.

Although the results on the time-of-arrival map are slightly different from the results obtained in the empty map, the performance of the algorithms is highly modified. 2D, 3D and 4D results are shown respectively in Figure 4.15, Figure 4.16, and Figure 4.17. In this case, raw computation times are shown together with a zoomed view of the fastest algorithms to make the analysis easier. Note that all methods become slower with non-constant velocities. The reason is that the narrow band tends to have more elements in this cases.

Some algorithms become unstable (in terms of asymptotic computational time) with non-uniform velocities: FSM, LSM and DDQM. DDQM is able to maintain the fastest time for slight velocity changes. But when these are sharper the double-queue threshold becomes unstable. However, for a high number of dimensions this effect vanishes (its computation time is barely modified with the number of dimensions) and DDQM becomes the fastest algorithms together with FIM and GMM.

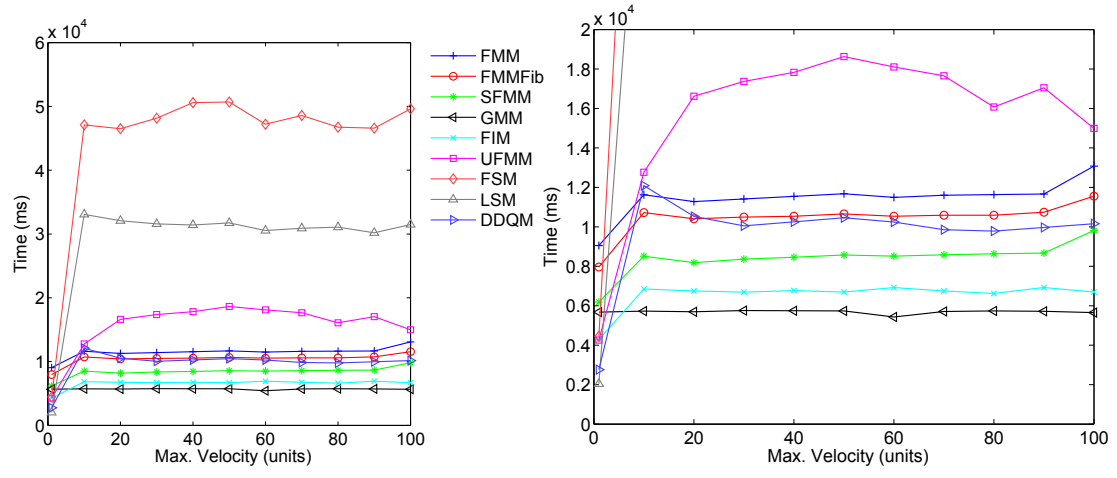
SFMM provides one of the best performances across all dimensions. And FIM becomes relatively faster in 3D and 4D. However, in most of the cases GMM is the fastest algorithm. FIM requires now multiple iterations to converge to the solution while GMM guarantees convergence with only 2 iterations.

Finally, UFMM requires special attention as it does not return the same solution than the other Fast Methods. Its performance is highly affected by the number of dimensions. The main reason is the election of the parameters: they were experimentally chosen to optimize 2D performance. However, these parameters are no longer useful in other number of dimensions. UFMM parameter tuning is therefore considered to be a complex task.



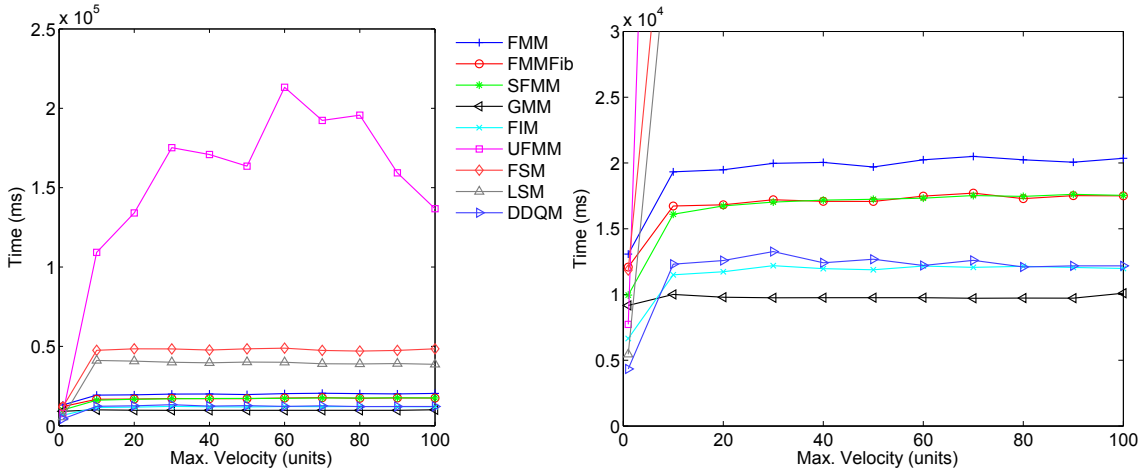
(a) Computation times. (b) Zoomed view.

Figure 4.15: Computation times and ratios for the random velocities experiment in 2D.



(a) Computation times. (b) Zoomed view.

Figure 4.16: Computation times and ratios for the random velocities experiment in 3D.



(a) Computation times. (b) Zoomed view.

Figure 4.17: Computation times and ratios for the random velocities experiment in 4D.

Table 4.2 summarizes the largest errors for this experiment. As the number of dimensions is increased, the error decreases while the computation time increases exponentially. Therefore, by properly tuning parameters for 3D and 4D better times could be achieved while keeping a negligible error in most practical cases.

Table 4.2: Largest L_1 and L_∞ errors for UFMM in the random velocities experiment.

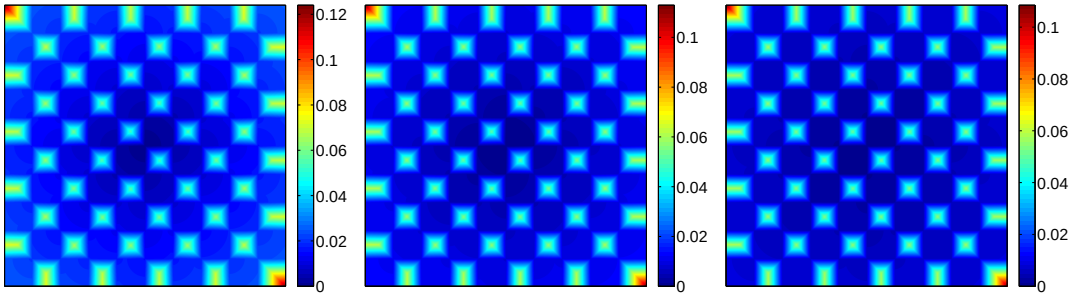
	2D	3D	4D
L_1	10^{-3}	$1.3 \cdot 10^{-10}$	$6.9 \cdot 10^{-12}$
L_∞	$4.8 \cdot 10^{-3}$	10^{-6}	10^{-7}

Checkerboard

Finally, different time-of-arrival map returned by FMM applied to the checkerboard map are shown in Figure 4.18. Numerical results of the computation times are included in Figure 4.19 for 2D, Figure 4.20 for 3D, and Figure 4.21 for 4D.

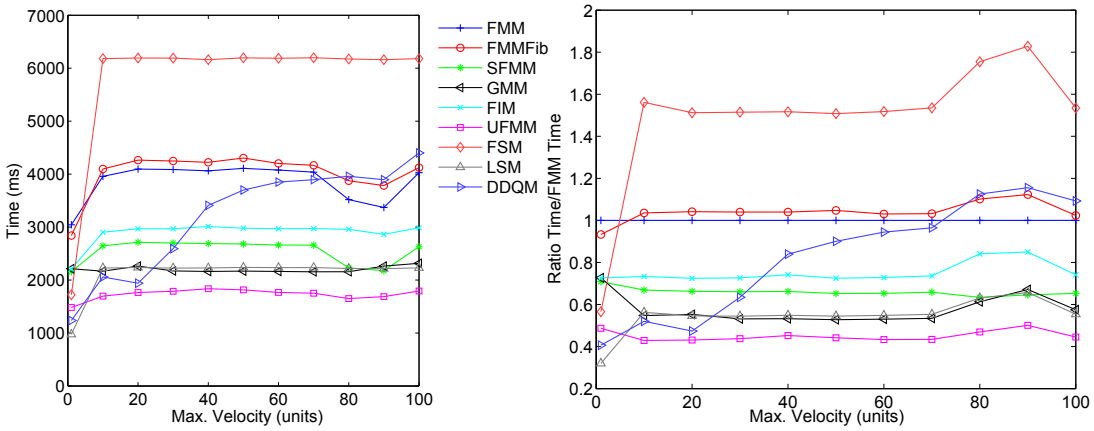
The results are relatively close to the random velocities experiment. However, the differences for FSM, LSM and DDQM are much smaller. In fact, DDQM presents a poor performance in 2D, but in 3D and 4D it becomes the fastest algorithm for higher velocity modifications.

In both 3D and 4D, GMM and FIM spend practically the same time. Since the environment is structured, FIM does not require too many iterations to compute the final value.



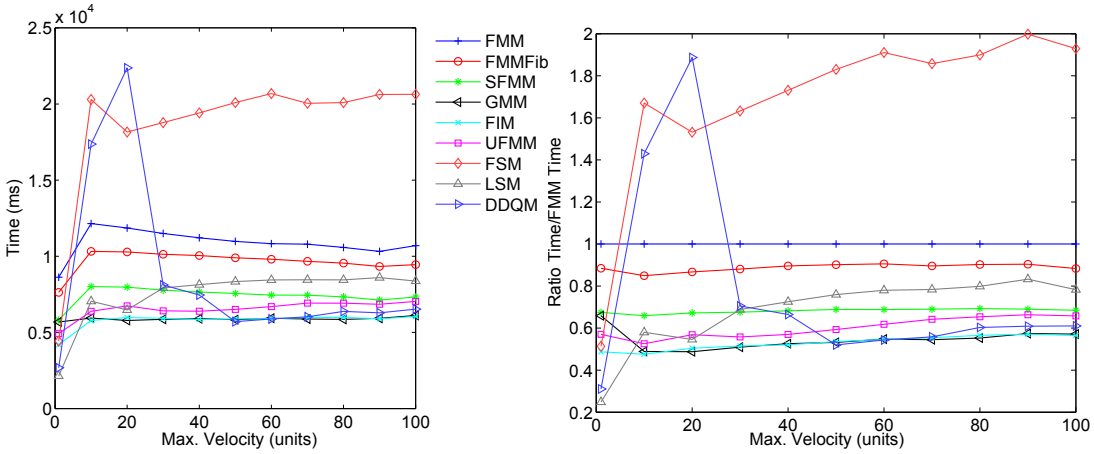
(a) Max. velocity=30 (b) Max. velocity=60 (c) Max. velocity=100

Figure 4.18: Example of the resulting time-of-arrival maps applying FMM to the checkerboard environment in 2D.



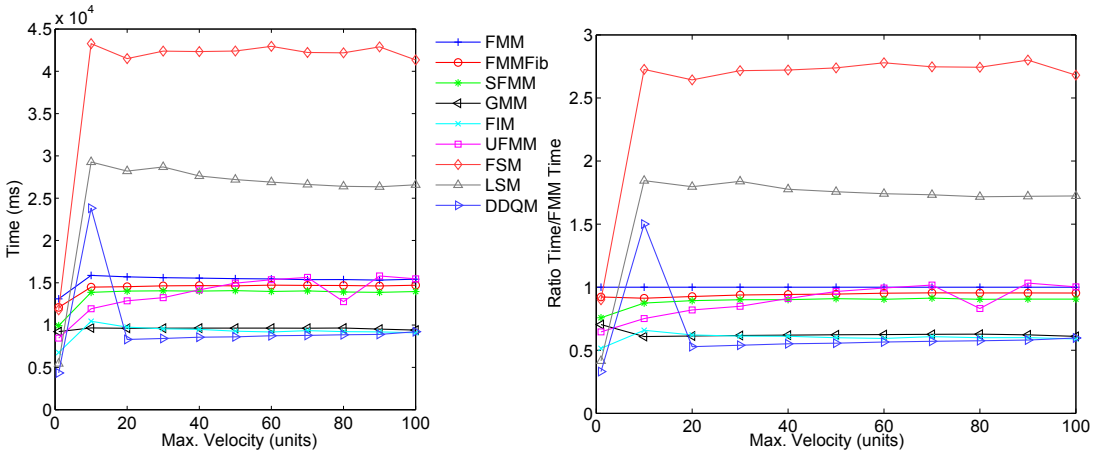
(a) Computation times. (b) Time ratios against FMM.

Figure 4.19: Computation times for the checkerboard experiment in 2D.



(a) Computation times. (b) Time ratios against FMM.

Figure 4.20: Computation times for the checkerboard experiment in 3D.



(a) Computation times. (b) Time ratios against FMM.

Figure 4.21: Computation times for the checkerboard experiment in 4D.

UFMM errors are shown in Table 4.3. In this case, UFMM becomes worse with the number of dimensions but it is among the fastest algorithms in lower dimensions.

The differences between this experiments and random velocities is that the environment presents a well-defined structure, and locally acts as a constant velocity environment, as in empty map experiment.

Table 4.3: Largest L_1 and L_∞ errors for UFMM in the checkerboard experiment.

	2D	3D	4D
L_1	$1.7 \cdot 10^{-7}$	$1.2 \cdot 10^{-9}$	$1.9 \cdot 10^{-10}$
L_∞	$2.5 \cdot 10^{-6}$	$5 \cdot 10^{-7}$	10^{-6}

4.6 Discussion

With the four experiments designed, the main characteristics of the Fast Methods have been shown. Any other environment can be thought as a combination of free space with obstacles and high-frequency or low-frequency velocity changes of different magnitude. Note also that the grid sizes covered by the experiments vary from extremely small to extremely big grids. In practice, it is hard to find applications requiring more than 16 million cells.

FIM results can be speeded up for non-constant velocity problems if larger errors are allowed. UFMM can be probably improved as well. However, our experience is that the configuration of its parameters is complex and requires a deep knowledge of the environment to be applied on.

Several conclusions can be extracted from the conducted experiments: 1) There is no practical reasons to use FMM or FMMFib as SFMM is faster in all cases with the same behaviour as its counterparts. 2) If a sweep-based method is required, LSM should be always chosen, as it greatly outperforms FSM. 3) In problems with constant velocity DDQM should be chosen, as it has shown the best performance for the empty map and the alternating barriers environments. 4) For variable velocities, but simple scenarios, GMM is the algorithm to choose. 5) UFMM is hard to tune and the result has errors. Also, it has been outperformed in most of the cases by DDQM in constant velocity scenarios, or by SFMM or FIM in experiments with variable velocities. 6) There is not a clear winner for complex scenarios with variable velocity. UFMM can perform well in all cases if tuned properly. Otherwise, SFMM is a safe choice, specially in cases where there is not too much information about the environment.

If a goal point is selected, cost-to-go heuristics could be applied [98], and thus enormously affect the results. Heuristics for FMM, FMMFib and SFMM are straightforward. They would improve the results in most of the cases. They could also be applied to UFMM. However, it is not clear if they can be applied to other Fast Methods. It is also of interest the solution of anisotropic problems [99], solved only by FMM-based methods.

Since all of the Fast Methods share the same formulation and the output is practically the same, all of them can be used as path planners following the same criteria as detailed in previous Chapter. Note that most of the Fast Methods would return

the same exact path for a given path planning problem, since the computed time-of-arrival maps $\mathcal{T}$ are equal. Methods which introduce errors such as UFMM, and potentially FIM, will introduce slight variations in the paths, which in most cases will be negligible.

If different velocities field $\mathcal{F}$ are provided, the resulting path can change its shape. However, the time optimality will be always kept, as the wave and thus the path will be computed according the best time of arrival. This is discussed deeper in the next Chapter.

4.7 Conclusions

Along this Chapter the main Fast Methods have been introduced under a common mathematical framework, focusing on a practical point of view.

The main purpose is closing the discussion about which method should be used in which case, as this is the first exhaustive comparison of the main Fast Methods (up to author's knowledge).

The code is publicly available as well as the automatic benchmark programs. This code has been deeply tested and it can serve as a base for future algorithm design, as it provides all the tools required to easily implement and compare novel Fast Methods.

The future work focuses in 3 different aspects: develop the analogous work for parallel Fast Methods [100], study the application of these methods to anisotropic problems and also to the new Fast Marching-based solutions focused on path planning applications [12, 53]. Finally, the combination of UFMM and SFMM seems straightforward and it would presumably outperform both algorithms.

Chapter 5

Fast Marching Square motion planning algorithm

5.1 Motivation

The path calculated by FMM is the shortest in terms of length but it might not be safe due to its proximity to obstacles or even not feasible due to robot kinematic constraints. In practice, this aspect also causes the path not to be the shortest in time, as the robot must move slowly when it is close to the obstacles in order to decrease damages if a collision occurs because of reasons external to planning such as sensor or motor failures. The usual solution is to expand obstacles before calculating the path, or to compute artificial repulsive potentials. However, these classic methods generate other problems such as local minima or deletion of actually feasible paths.

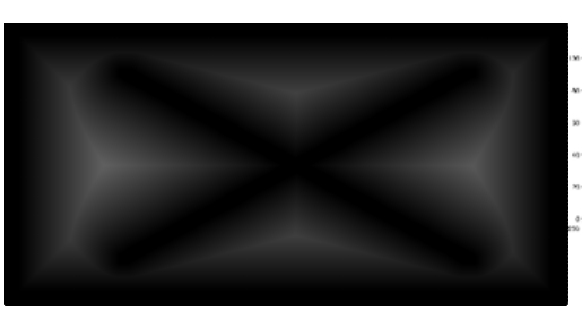
In this Chapter a robust, efficient and safe motion planner is proposed. More concretely, four different methods based on Fast Marching are introduced: the Fast Marching Square (FM^2) method for path planning, two heuristic modifications of it, FM^{2*} and Greedy FM^{2*}, and the Directional Fast Marching Square Method (DFM^2).

These methods are considered robust as they are deterministic, it is easy to predict their output and they never produce unexpected results or failures. Also, from an engineering point of view, their memory usage is clearly upper-bounded by the size of the grid representing the environment and the amount of time required in order to get a solution is also accurately predictable as it mainly depends on the grid size. Efficient as they will analyze the environment and return a path with a relatively light algorithm, specially in the new proposed versions. And safe because they keep paths away from obstacles and it is theoretically impossible for the trajectory to collide with obstacles.

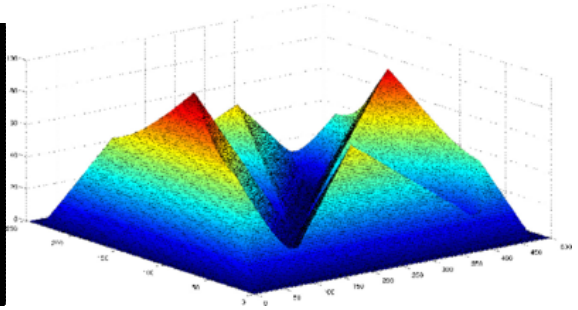
5.2 Fast Marching Square

Let us take an evidence gridmap in which obstacles are labeled as 0 and free space as 1. The Fast Marching Method can be applied to this map considering all the obstacles to be wave sources. In the previous Chapter, there was just one wave source (at the target point). Here all the obstacles are a source of the wave, and hence, several waves are being expanded at the same time. The map resulting from applying this wave expansion to a simple map with a cross-shaped obstacle can be seen in Figure 5.1. This resulting map is denoted as *fast marching gridmap* (FMGridMap); it represents a potential field of the original map. As cells get further from the obstacles, the computed T_{i} value is greater. This map can be seen as a *slowness map*: T_{i} value can be considered to be proportional to the maximum allowed speed of the robot at each point. It can be appreciated that speeds are lower when the cell is close to the obstacles, and greater when far away from them. In fact, a robot whose speed at each point is given by the T_{i} value will never collide, as $T_{\mathrm{i}} \to 0$ when approaching the obstacles. Making an appropriate scaling of the FMGridMap cell values to the robot

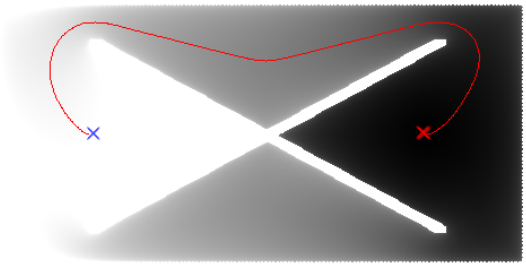
allowed speeds, a *slowness map* is obtained, that provides a safe speed for the robot at any point of the environment. In Figure 5.1 d) the speed profile is shown. In the image it is clear that speed becomes greater far from the obstacles.



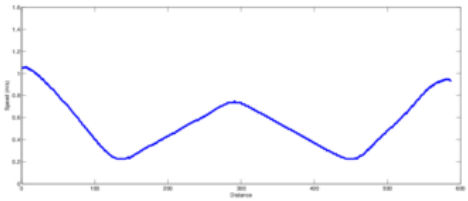
(a) FMGridMap



(b) 3D Representation



(c) Second Fast Marching and Path



(d) Speed Profile

Figure 5.1: FM^2 steps.

The path is calculated as in the previous Chapter, so that the robot is still considered as a holonomic point. But instead of taking a constant value for the propagation velocity $\mathcal{F}$, the value of the *slowness map* is used. If a wave is expanded from one point of the grid, considering that the expansion speed $F_i = T_i$, being T_i the value of the FMGridMap at cell i, the expansion speed will depend on the position. Given the Fast Marching properties, the obtained trajectory is the fastest path (in time) assuming the robot moves at the maximum allowed speed at every point.

5.2.1 FM^2: The Saturated Variation

In Fig. 5.1 c) it can be appreciated that the computed path is not the logical/optimal trajectory one would expect. The FM^2 computed path, as it has been presented, tries to keep the trajectory as far as possible from obstacles. This computed trajectory is similar to the path computed with the Voronoi diagram [101]. But there are environments in which there is no need to follow a trajectory so far away from obstacles, as a

smaller distance may be safe enough to navigate. To solve this a saturated variation of the FMGridMap is implemented. When the first fast marching has been computed, the FMGridMap is first scaled and then saturated.

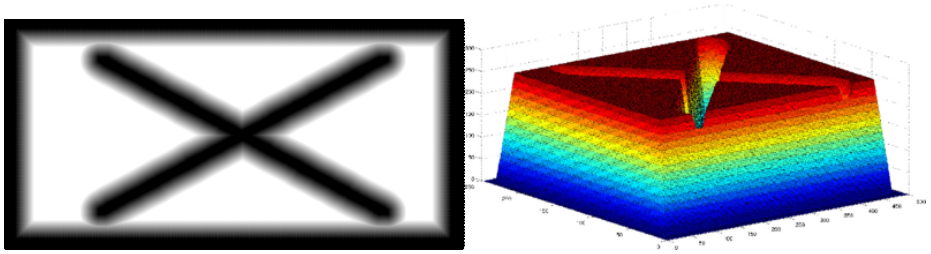
The map is scaled according to two configuration parameters:

- Maximum allowed speed, which is the maximum control speed the robot may receive.

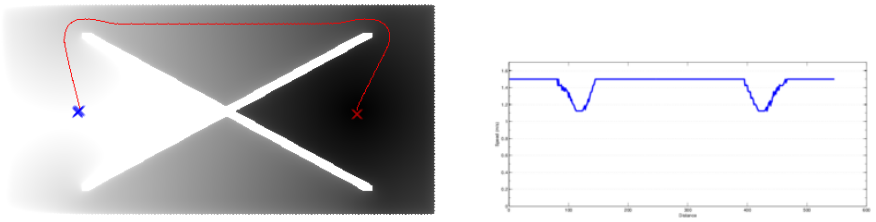
- Safe distance, which is the distance from the closest obstacle at which the maximum speed can be reached.

Finally the map is saturated to the maximum allowed speed. After this scaling and saturation process the slowness map provides the maximum speed for all the points that are farther than the safe distance from the obstacles and the control speed varying form 0 (at obstacles) to the maximum speed (at safe distance) for the rest of points.

Figure 5.2 shows the saturated variation of Figure 5.1 with the new computed trajectory. It can appreciated that now the path and speed profile are as expected.



(a) Saturated FMGridMap (b) 3D Representation



(c) Second Fast Marching and Path (d) Speed Profile

Figure 5.2: Saturated FM^2 steps.

5.3 FM^{2*}: Fast Marching Square Star

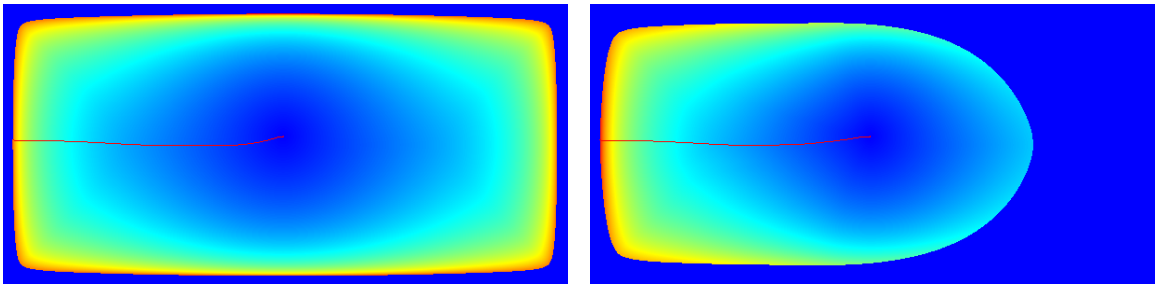
Consider a rectangular map in which a trajectory among two points on a free space must be computed. Figure 5.3 a) shows the FM^2 wave expansion originated from the target point. As it can be appreciated the wave grows concentrically around the target point until it reaches the initial point. The FM^{2*} (to be read as FM^2 star, where *star* indicates the heuristic version, analogous to A* and Dijkstra) method is an extension of the FM^2 method. It tries to reduce the total number of expanded cells (wave expansion) by incorporating a heuristic estimate of the cost to get to the goal from a given point.

The FM^{2*} algorithm's principle is the same as for the FM^2 algorithm. The only difference is the function used to sort the *narrow band* queue. The FM^2 sorts the *narrow band* cells in increasing T_{i} order, so that in each iteration, the first element in the queue (lowest T_{i}) becomes *frozen* and it is expanded. In FM^{2*} the algorithm uses the cost-to-come T_{i}, which is known, and the *optimal cost-to-go*, that is, the minimum time the robot would employ to reach the target. This implies that the *narrow band* queue is sorted by estimates of the optimal cost from the given cell to the target. Whenever the optimal cost-to-go is an underestimate of the real cost-to-go the algorithm will still work. In fact, if the optimal cost-to-go is assigned to be 0, the FM^{2*} algorithm is equivalent to the FM^2 algorithm. If the estimation is greater than the real cost-to-go, the FM^{2*} algorithm could take more computational steps than the FM^2 to find the path and the path could be not the shortest.

In this problem, the optimal cost-to-go (optimal time to reach the target) would happen if the robot went directly towards the goal at maximum speed. This cost-to-go is given by the Cartesian distance (minimum distance) divided by the maximum speed the robot can reach. It is known that the real cost-to-go will always be greater than this computed value. So, the *narrow band* queue is ordered according to the value T_{i}^*:

$$T_{\mathrm{i}}^* = T_{\mathrm{i}} + \frac{cartesian\_distance\_to\_target}{robot\_max\_speed} \tag{5.1}$$

These two methods are analogous to Dijkstra and A* in path-finding over graphs. The result for FM^{2*} is shown in Figure 5.3 b). The original method explores all the space at the same distance in time from the source point until the goal is reached. However, FM^{2*} focuses the space exploration on those areas which are closer to the goal point, so that the wavefront is biased. The consequence is that only a portion of the space is explored and thus the computation time is improved.



(a) FM^2 wave expansion and path. (b) FM^{2*} wave expansion and path.

Figure 5.3: Comparison between FM^2 and FM^{2*}

5.4 Greedy Fast Marching Square Star

The previous Section improved FM^2 by including an Euclidean-based cost-to-go heuristic, as the distance was computed according the Euclidean metric. However, the wave propagation does not take place in an Euclidean space, but in a time-based metric.

Equation 5.1 does not take into account the velocity propagation in the current cell. Therefore, it will give higher priority in the narrow band to points which are closer to obstacles, and thus have a lower propagation velocity, but are closer in terms of Euclidean distance to the goal. Once evaluated, these new cells will be inserted at very end of the narrow band as they will have high arrival time. The consequence is the evaluation of cells which are not likely to belong to the final solution.

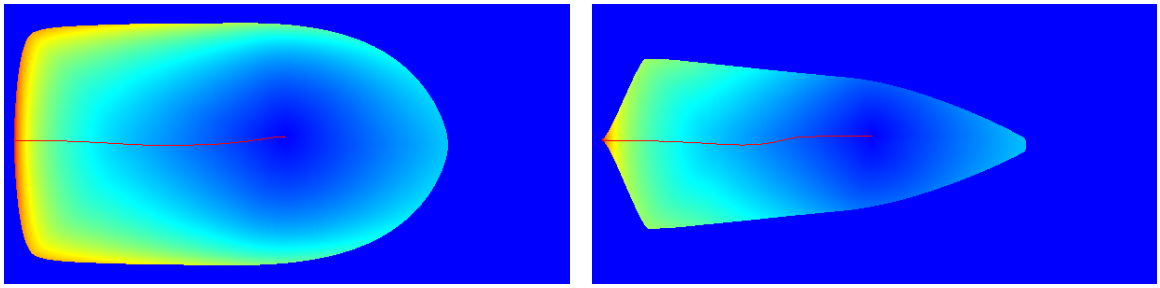
Thus, the greedy variation of FM^{2*} takes into account the velocity of the cell being computed when computing the cost-to-go heuristic:

$$T_{\mathrm{i}}^{*} = T_{\mathrm{i}} + \frac{cartesian\_distance\_to\_target}{F_{\mathrm{i}}} \tag{5.2}$$

Applying this heuristic, cells with higher velocity will be given preference even if they are far from the goal point. And cells with same velocity will be sorted giving priority to those with smaller distance to the goal, and thus greatly improving the computational time with negligible impact on the final result. A comparison against FM^{2*} in an empty environment is shown in Figure 5.4. Greedy FM^{2*} returns a path close to FM^2 and FM^{2*} but even less space is explored by the wave front and thus less computation effort is required.

5.5 Results

FM^2 and its two heuristic versions are compared. A 2D map representing a complex environment is chosen, with a combination of empty spaces, corridors and small rooms



(a) Wave Expansion and Path with FM^{2*} (b) Wave Expansion and Path with Greedy FM^{2*}

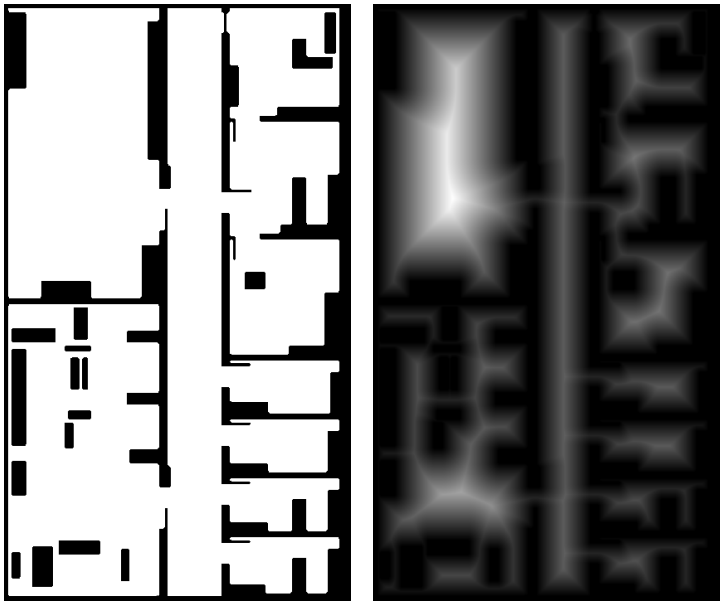
Figure 5.4: Comparison between FM^{2*} and Greedy FM^{2*}.

(Figure 5.5). Table 5.1 reports the computation times of the second FM^2 wave using FM^2 with Binary heap (FM^2 for simplicity), FM^{2*} and Greedy FM^{2*} (G-FM^{2*}), and the corresponding for SFMM, SFMM-based FM^2 (SFM^2), SFM^{2*}, and G-SFM^{2*}. for the path queries included in Figures 5.6 to 5.10. These figures show the final propagated wave of FM^2 with Binary heap, as the propagation is exactly the same for SFMM. Experiments were run in an dual-core machine 2.2 GHz with 4 GB of RAM, running Ubuntu 14.04 64bits. In these images it is possible to appreciate that the wavefront always explores much less space for the Greedy FM^{2*}, while FM^{2*} implies just a little improvement. However, in all cases the paths are almost identical (differences would not be meaningful in a real robotic application).

Table 5.1: Time (ms) comparison for the proposed FM^2 variants.

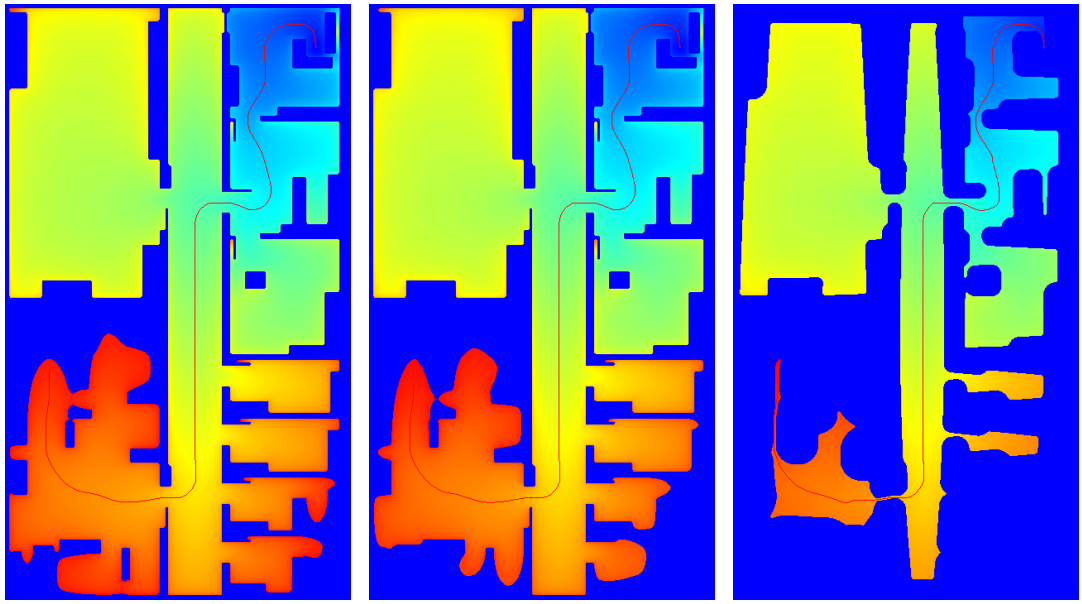
Exp.	FM^2	SFM^2	FM^{2*}	SFM^{2*}	G-FM^{2*}	G-SFM^{2*}
1	91	60	68	51	2	1
2	142	96	137	106	93	62
3	150	100	89	66	14	11
4	30	24	24	17	6	6
5	91	82	81	66	50	35

In some cases SFM^2 is even faster than FM^{2*}. However, G-SFM^{2*} is clearly faster in all cases while keeping the main FM^2 properties: safety and smoothness, as the path is minimally modified.



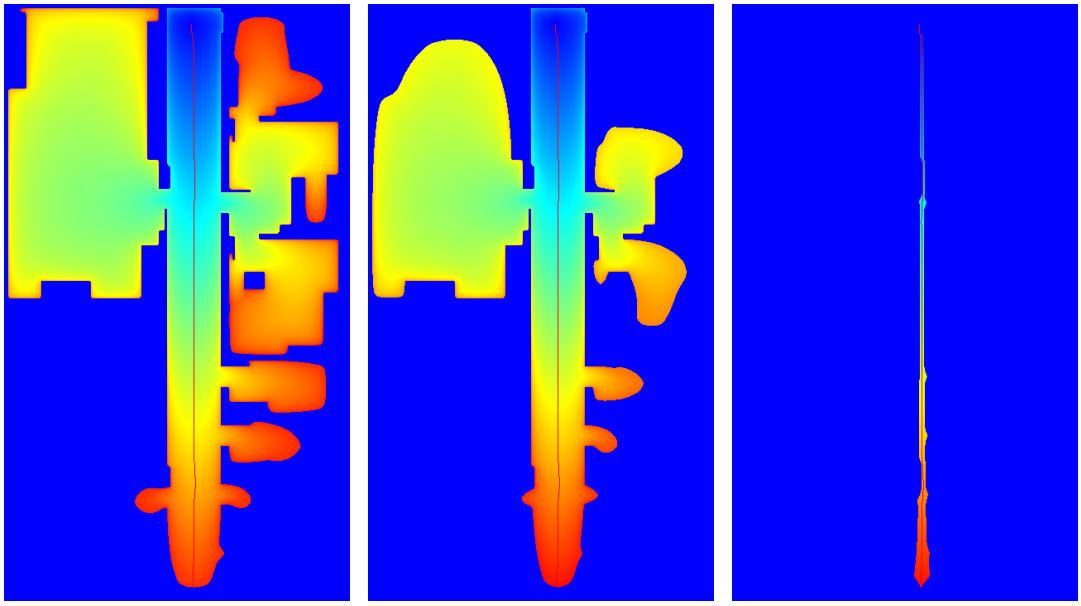
(a) Environment map. (b) Velocities map, $\mathcal{F}$.

Figure 5.5: Map used to experiment FM^2 and its variants.



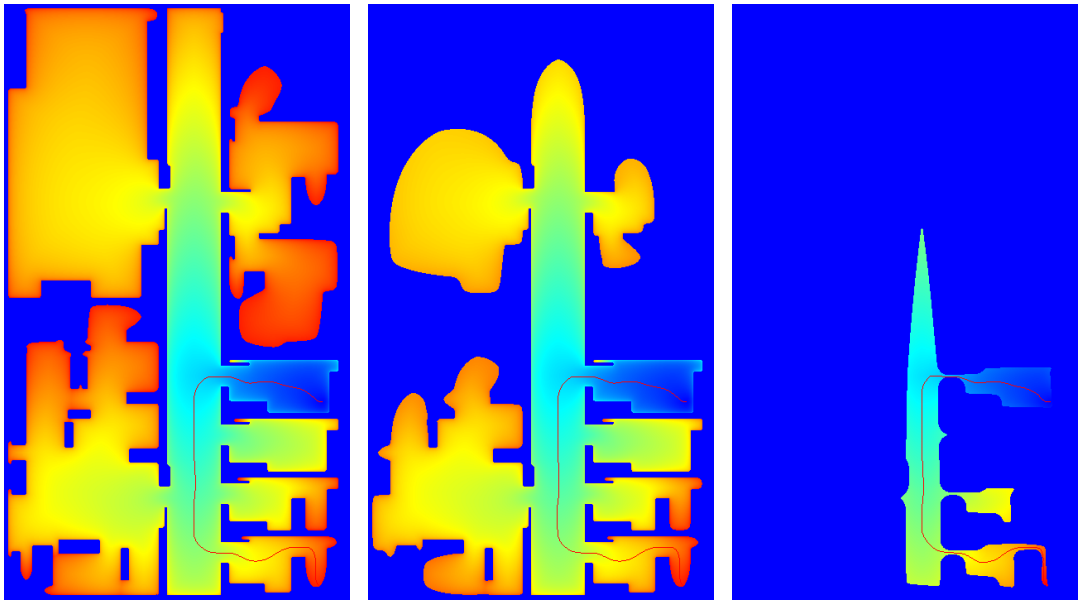
(a) FM^2 Binary heap. (b) FM^{2*}. (c) Greedy FM^{2*}.

Figure 5.7: Wave propagation comparison: experiment 2.



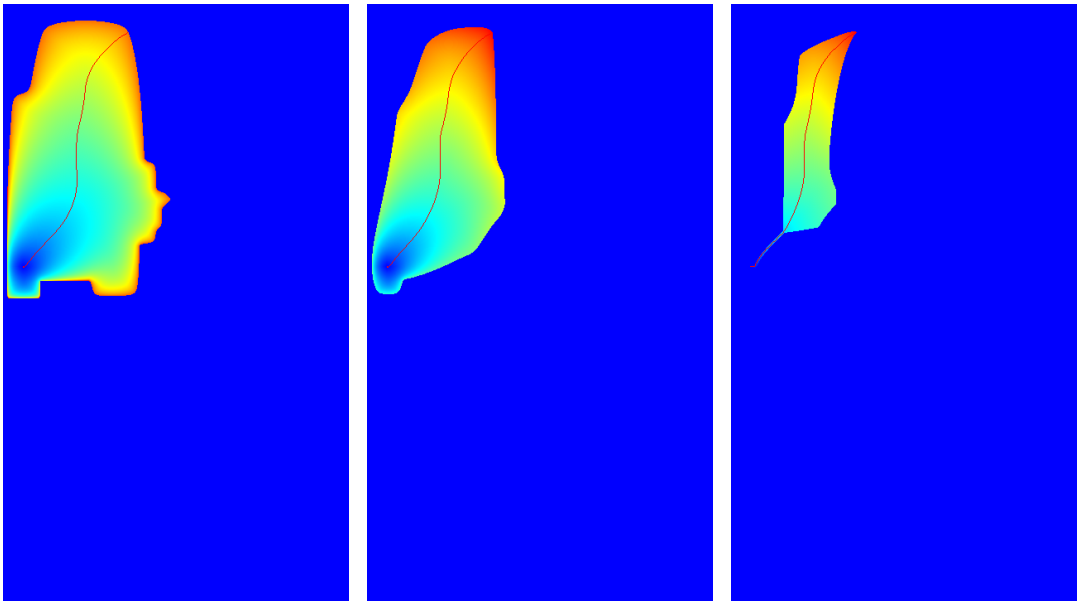
(a) FM^2 Binary heap. (b) FM^{2*}. (c) GreedyFM^{2*}.

Figure 5.6: Wave propagation comparison: experiment 1.



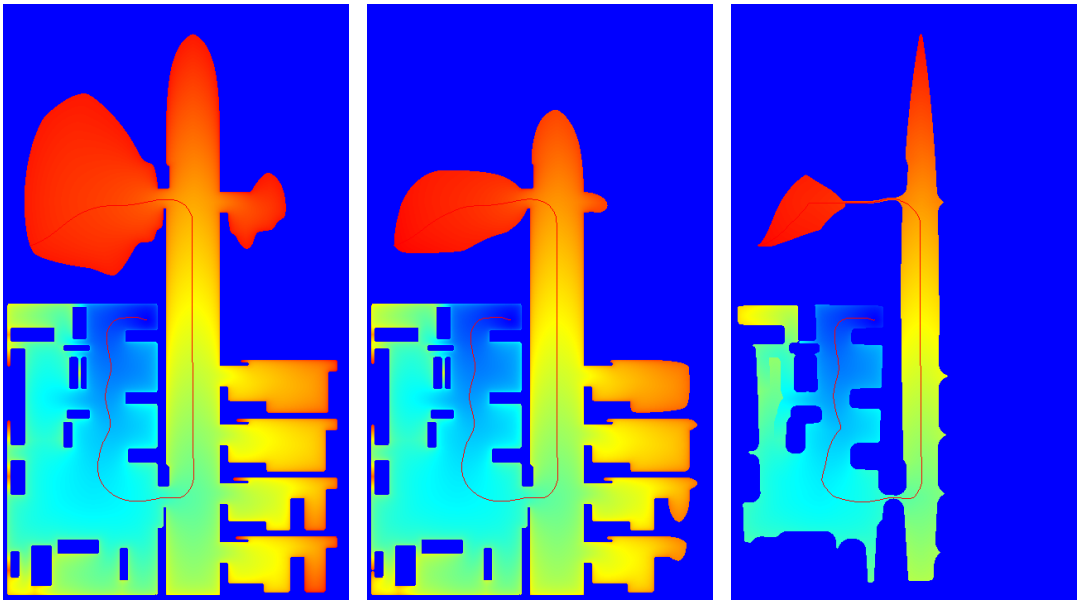
(a) FM^2 Binary heap. (b) FM^{2*}. (c) Greedy FM^{2*}.

Figure 5.8: Wave propagation comparison: experiment 3.



(a) FM^2 Binary heap. (b) FM^{2*}. (c) Greedy FM^{2*}.

Figure 5.9: Wave propagation comparison: experiment 4.

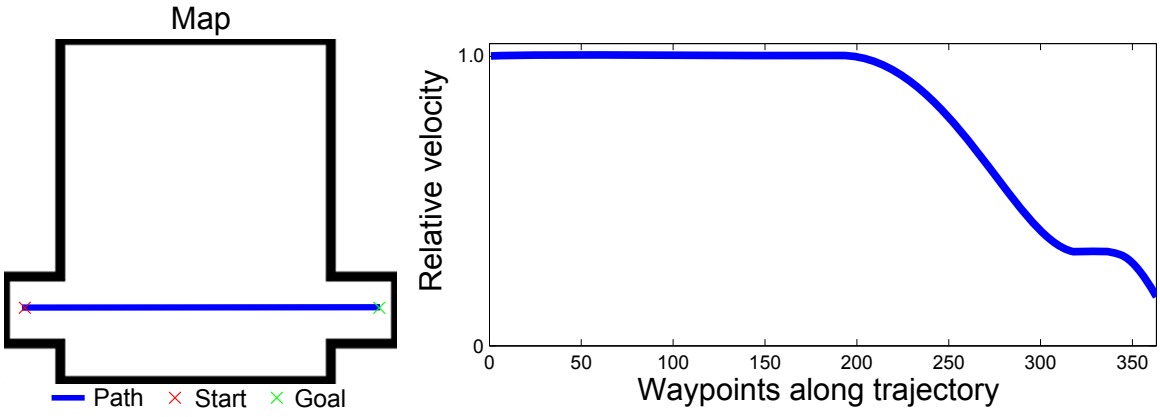


(a) FM^2 Binary heap. (b) FM^{2*}. (c) Greedy FM^{2*}.

Figure 5.10: Wave propagation comparison: experiment 5.

5.6 Directional Fast Marching Square

The Directional Fast Marching Square (DFM^2) tries to solve one of the most important problems of FM^2: in environments with large, clear areas the FM^2 trajectory will be not as expected: it will steer towards the center of these areas and the velocities profile will command a slow velocity even if the path is getting away from obstacles. Figure 5.11 shows a clear example of this problem. Velocity reference is slow at the beginning of the path, where the path is safe with respect to obstacles and, once the room is reached, the path is bended towards the center of the room, increasing its length. Figure 5.12 shows the desired result for an efficient algorithm: maximum velocity when obstacles do not represent a danger and paths closer to its optimal length. This problem is partially solved with the saturated variation of FM^2. However, there are cases in which the robot can be close to an obstacle, in the not-saturated area of the velocity map, and getting farther for this obstacle. While maximum speed is safe, the reference velocity will still increase as the robot gets farther from the obstacle. DFM^2 also aims to solve this case.



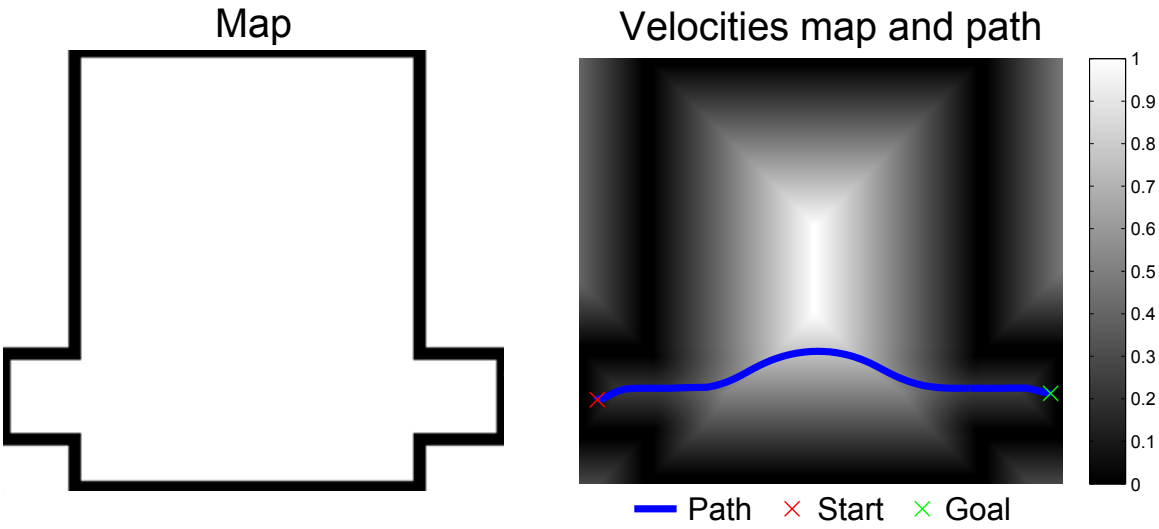
(a) FM^2 desired result.

(b) FM^2 desired velocities profile.

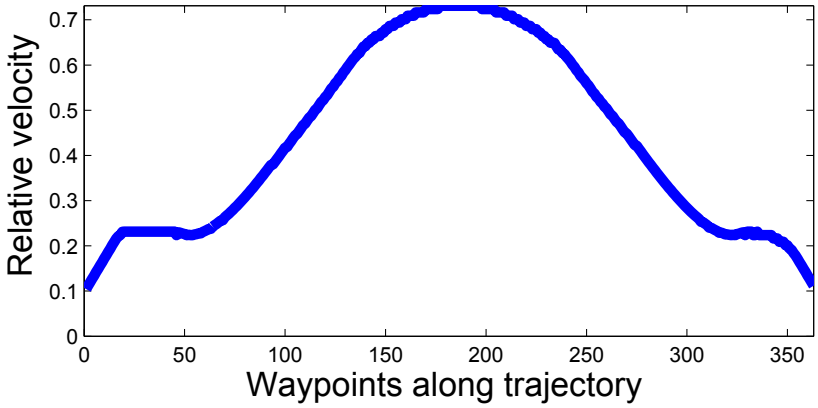
Figure 5.12: Desired FM^2 results.

DFM^2 tries to solve these two problems at once. For that, the velocities of the next cell to evaluate and current cell are compared. If the velocity in the following cell is lower than the velocity of the current cell the wave is then approaching to an obstacle. However, the path in that cell will be getting farther from that obstacle, as the path travels in the opposite direction than the wave front. Therefore, in this case it does not make sense to follow the reference velocity given by the velocity map, but use the maximum velocity.

More formally, let F_{i} and F_{j} be the velocities for the current and next cell, correspondingly. The Eikonal equation for the next cell is solved using the propagation



(a) Map and FM^2 result.



(b) FM^2 velocities profile.

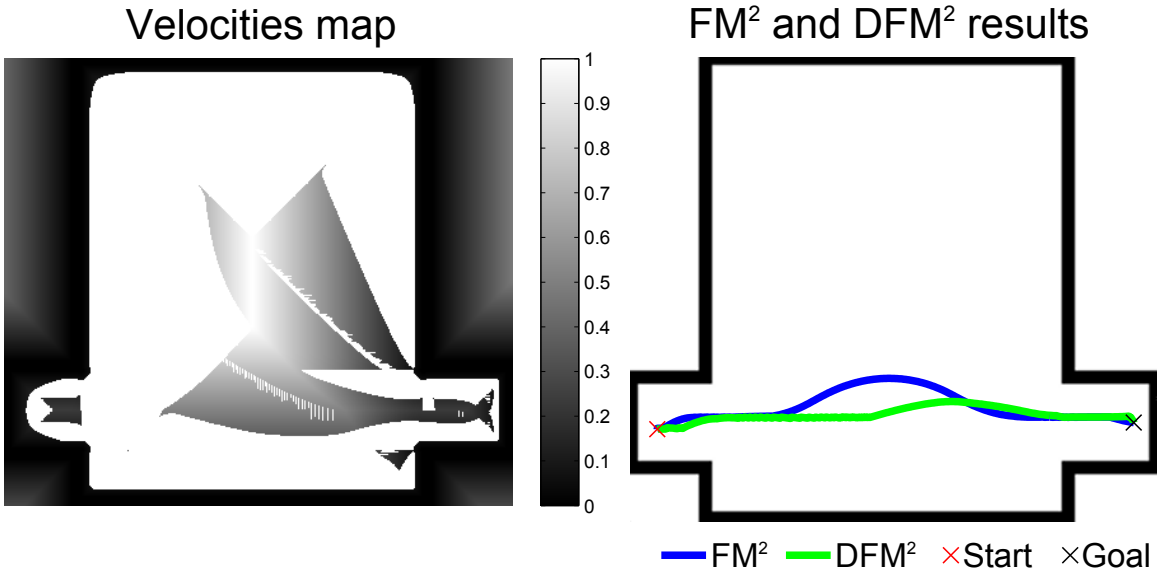
Figure 5.11: FM^2 drawback example.

velocity F_{prop} computed as:

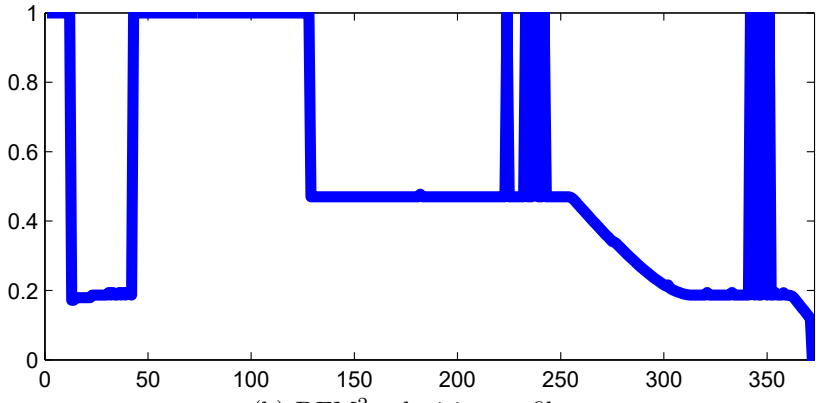
$$F_{\text{prop}} = \begin{cases} F_{\text{max}}, & \text{if } F_{\text{i}} > F_{\text{j}} \\ F_{\text{j}}, & \text{otherwise} \end{cases} \tag{5.3}$$

However, if the wave is propagated using F_{prop} its behaviour will be unpredictable. Therefore, a two-layer approach is required: the wave is propagated and as usual, using F_{j}, but F_{prop} is computed and stored to apply gradient descent and compute the velocities profile. This approach is obviously slower than FM^2 but the objective of DFM^2 is to have better quality paths. Figure 5.13 shows the result in the example map given in previous figures. The final velocity map completely depends on the path

query and it is very sensitive to small changes in the original velocity map. However, the resulting path is clearly shorter and the velocities profile is closer to the desired result. However, the velocities profile would require a post-processing step (not in the focus of this Thesis) since it presents sharp changes unreliable for a real system. In fact, DFM^2 looks to behave like FMM when getting farther from obstacles, but like FM^2 when getting closer.



(a) DFM^2 velocity map and result.



(b) DFM^2 velocities profile.

Figure 5.13: DFM^2 results.

However, DFM^2 results in more complex scenarios are not this satisfactory, since its behaviour in cluttered environments is close to FMM in most of the cases. However, it has been shown that the idea is sound and can be improved by taking into account

the propagation direction and the velocities gradient. A more detailed analysis about DFM^2 and possible improvements is included in [102].

5.7 Conclusions

FM^2 and its variations have been explained in detail. FM^2 computes two wave expansions over the gridmap. The first expansion computes the FMGridMap, a slowness map that provides the maximum allowed speed at each point of the map. This slowness map is used to compute the second expansion, form the target point to the initial point. As a result of the second expansion, the trajectory is computed (using the maximum gradient direction). This solution provides both a path (waypoint) and the control speed at each point. As a result, this trajectory is safe and optimal in time. The FM^2 computes paths that tend to navigate far from obstacles, but this situation is not always necessary.

Two variations to the FM^2 to avoid this problem were presented: the saturated FM^2 and DFM^2. Also, to reduce the computation time, other two heuristics were proposed to improve FM^2: FM^{2*} and Greedy FM^{2*}. The experimental analysis shows that the computation time is reduced while providing a trajectory that are practically the same.

As discussed in previous sections, these algorithms do not take explicitly into account robot's dimensions and its kinematic constraints. FM^2 and their versions rely on a basic obstacle dilation pre-processing step in order to solve the first problem. Given the smoothness of the solutions, in most cases output trajectories can be directly used in robots kinematically constrained. However, a validation step should be included. This issue is addressed in [67], where the velocity field is iteratively modified in order ensure that the minimum turning radius is satisfied.

Chapter 6

Fast Marching Square applied to the ITER project

DISCLAIMER: Research contained in this Chapter was developed jointly between the author of this Thesis and members of the Instituto de Plasmas e Fusão Nuclear (IPFN), Instituto Superior Técnico, University of Lisbon. Specifically, the Fast Marching Square experimentation and results comparison represent the authors' collective contribution. The remaining CDT, optimization algorithm and the Trajectory Evaluator and Simulator (TES) simulation were performed by members of IPFN.

6.1 Introduction

Path planning is one of the key issues for hazardous transport operations using autonomous mobile robots. Not only for scenarios of disaster, but also when working in experimental scenarios testing new sources of energy where human being is not allowed. In particular, the International Thermonuclear Experimental Reactor (ITER) is a worldwide research experiment that aims to explore nuclear fusion as a viable source of energy for the coming years. The ITER project aims to make the long-awaited transition from experimental studies of plasma physics to full-scale electricity-producing fusion power plants. The largest experimental tokamak nuclear fusion reactor, depicted in Figure 6.1, will be located at the Cadarache facility, in the south of France.

Besides the major scientific objective of exploring the nuclear fusion as a source of energy, ITER aims to demonstrate that the future fusion power plants can be safely and effectively maintained through Remote Handling (RH) techniques, due to restrictions on human being in activated areas. The RH approach must be from the outset as flexible as possible with minimum reliance on the tokamak configuration, such as in ITER, [103].

The top level maintenance functions of RH in ITER are the exchange of blanket segments, divertor cassettes, perform in-vessel inspection and recovery tasks, allow remote maintenance of ex-vessel systems including heating and current drive systems, ex-vessel transfer casks and servo manipulators. In particular, the maintenance functions of ex-vessel transfer cask has a relevant reliance not only with the reactor, but with the entire power plant. Transportation of equipment for storage, refurbishment and repair requires vehicles of transportation that navigate along corridors of the power plant. A transport cask system (simply identified as cask) is required to accomplish the maintenance operations that includes transportation. Pre-computed

paths assuming the well-known scenarios are expected for nominal operations. However, during the maintenance operations, new paths must be computed. For instance, when a cask fails, another rescue cask has to dock into the first one, remove the activated load and then drive it to the maintenance area.

The cask, depicted at the bottom of Figure 6.1, is a large and complex unit to transport heavy and contaminated components between the two main buildings of ITER: the Tokamak Building (TB), where the reactor is installed, and the Hot Cell Building (HCB) for refurbishment and storage. The geometry of the cask and its payload vary according to the components to be transported and hence, different cask typologies are expected. As a reference, the largest cask dimensions are 8.5m x 2.62m x 3.62m (length x width x height) and the total weight can reach up to 100 tons.

The maintenance operations of transportation require the cask to move throughout the cluttered environments of the TB and the HCB, equivalent to drive a truck under 30 cm of safety margins to the closest walls and pillars. The constrained space may also rise a logistic problem, where multiple vehicles have to move and different paths must be computed. Because of this, some of the choices are subject to be changed. This Chapter focuses in the most probable ITER scenario as ITER as not officially accepted and adopted these methodologies.

The most probable kinematics of the cask are equivalent to a rhombic like vehicle, with two drivable and steerable wheels. Given this configuration, proposed in [104, 105], the cask has a higher maneuverability in confined spaces than the traditional cars with Ackerman or tricycle configurations [106].

From previous work of RH in ITER, the optimized paths would most probably be implemented on the scenario using buried wired systems [104]. Presently, the buried wired systems are being superseded by other systems, as a line painted on the floor or, simply, by a virtual path. These systems are used in several Automatic Guided Vehicles (AGV) applications [107, 108, 109]. In this navigation methodology, the vehicles would follow the path by using a line guidance approach: both wheels following the same path. The proposed planning methodology returns directly the path to be followed by the center of the wheels and not the one corresponding to the center of the vehicle (identified as the free roaming, out of the scope of this Chapter).

A nominal operation of the vehicle for a specified environment determines a motion between two configurations (2D points with specific orientations). The first step of this planning methodology is to find an initial geometric path, i.e. a set of 2D points, connecting the initial and final configurations. The previous implemented approach was based on the Constrained Delaunay Triangulation (CDT) [7, 8]. This solution presents some limitations in terms of path smoothness that are solved with the next stages of the algorithm. In complex scenarios, the geometric representation results in a huge number of triangles with rough initial paths still far from the optimal

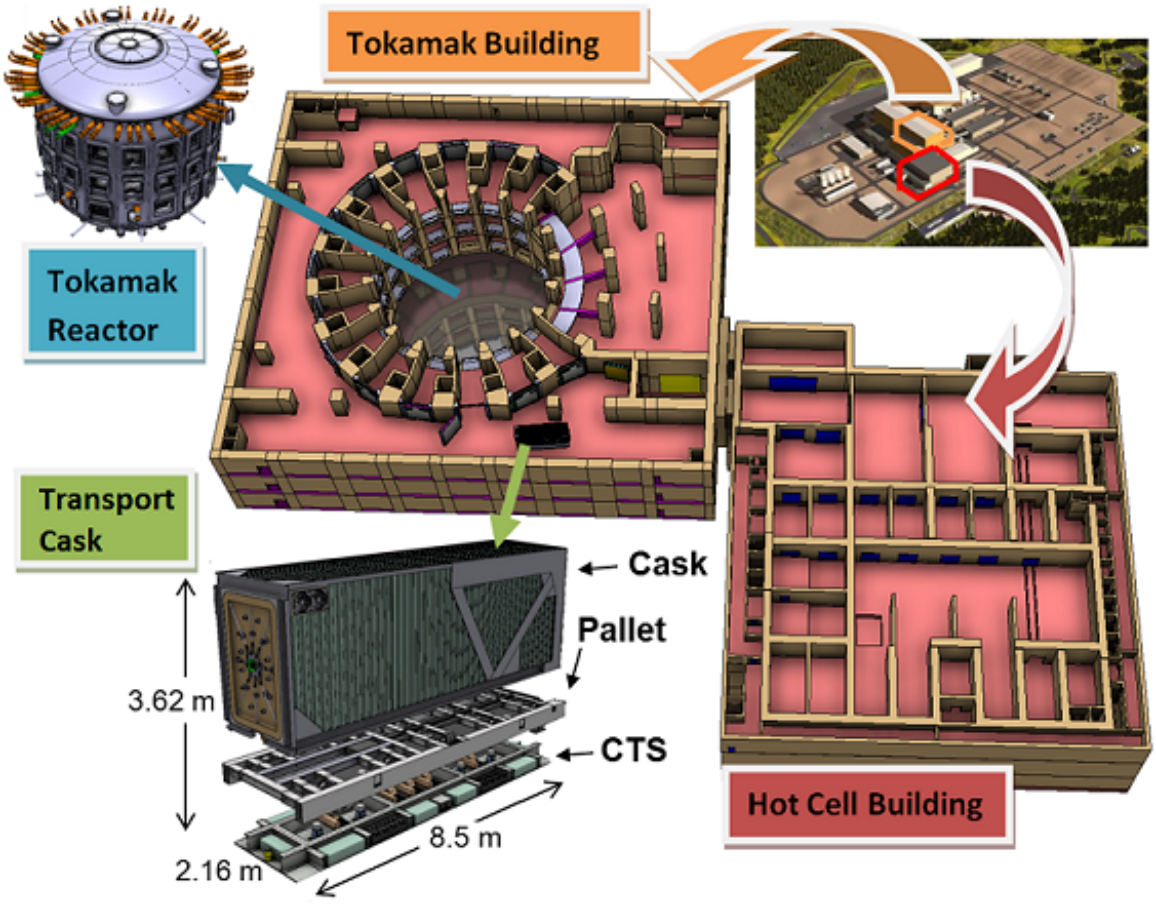


Figure 6.1: The ITER tokamak and the scientific buildings and facilities that will house the ITER experiments in Cadarache, South of France.

one, yielding to a computational effort [110]. The FM^2 is an alternative approach for the initialization as it provides an initial path closer to the optimal solution and in a shorter period of time, resulting in an improvement of the computational effort. Therefore, in this Chapter an in-depth analysis of the FM^2 performance when applied to such a complex and critical problem as ITER.

Previous works have already addressed the application of the FMM to kinematically constrained systems. One of the first approaches is an iterative method which in every iteration computes a different path [67]. If this path does not satisfy the kinematic constraints, the obstacles are smoothly dilated, so that the next computed path will have smoother, larger curves. This process is repeated until a valid path was found. Another different approach is to compute an initial path with FMM and then propagate a second FMM wave within a tube in the initial path surroundings [111, 71]. In this second wave expansion, the FMM is modified so that neighbors of the grid cell are no longer computing according Von Neumann neighborhood, but are computed by propagating the system with different input actions. The main drawback of this problem is its computational complexity. Ryo et al. [112] proposed a new Hamilton-Jacobi formulation for computing optimal trajectories for systems with limited curvature. This formulation has been successfully applied to Dubin's and ReedsShepp car models. However, the whole formulation needs to be done from scratch for every different kinematic system.

Previous work in this topic also includes the application of FM^2 in a 3-dimensional configuration space (two spatial dimensions and the vehicle orientation) [60]. When applying FM^2 in this configuration space, smooth paths are guaranteed. However, this approach did not take into account explicitly the kinematic constraints. However, in this Chapter the regular 2-dimensional version is being applied since the vehicle employed (detailed in section 6.2.2) does not have kinematic constraints. However, the vehicle's kinematics and dimensions require a similar study to that carried out in vehicles with such constraints.

During the maintenance operations of transportation, the pose (position and orientation) of each vehicle must be continuously evaluated using sensors data. Although the first studies of localization of the CTS in ITER have been accomplished, as described in [113], in this Chapter it is assumed that the pose of the vehicle is always known without any uncertainty.

The Chapter is organized as follows. Section 6.2 describes the ITER problem statement: the scenario, the vehicle, the goals and the optimization criteria. Next, Section 6.3 introduces the solution proposed for trajectory planning at ITER. Section 6.4 describes the optimization in terms of clearance and smoothness applied to the paths returned by the FM^2 (or CDT). Section 6.5 presents simulated results in the ITER scenarios, including a comparison between CDT and FM^2 initializations. Finally, conclusions and future work are pointed out in Section 6.6.

6.2 Problem Statement

The problem statement description is divided into four different issues: the environments, the vehicles, the missions (goals) and the optimization criteria.

6.2.1 Environments

The TB, shown in Figure 6.2, lodges the tokamak reactor with access by vacuum vessel port cells (from this point forward simply identified as ports). The HCB, depicted in Figure 6.3, will work mainly as a support area. A lift establishes the only interface between the different levels of TB and the HCB.

In ITER, the environments in all levels of TB and HCB are mostly composed by static and well structured scenarios. Each level of the buildings is modeled as 2D map representation, M, with a set of 2D points, p_i, on the global Cartesian referential of ITER and a set line segments, l_{jk}, where each line segment connects two different points p_j and p_k, i.e.,

$$M = \{p_i, l_{jk} | i, j, k = 1, \ldots, M_p\} \tag{6.1}$$

where M_P is the number of points, $p_i = (x, y)$ and $l_{jk} = \{(1-t) \cdot p_j + t \cdot p_k | t \in [0, 1]\}$.

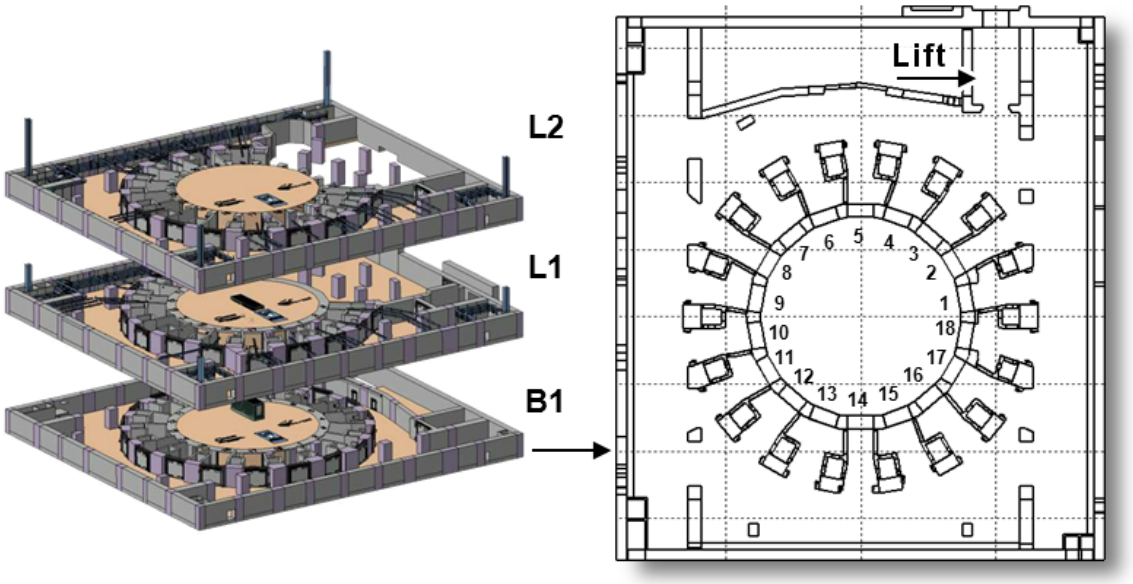


Figure 6.2: The three main level of Tokamak Building in ITER (left) and the 2D representation of the level B1 (right).

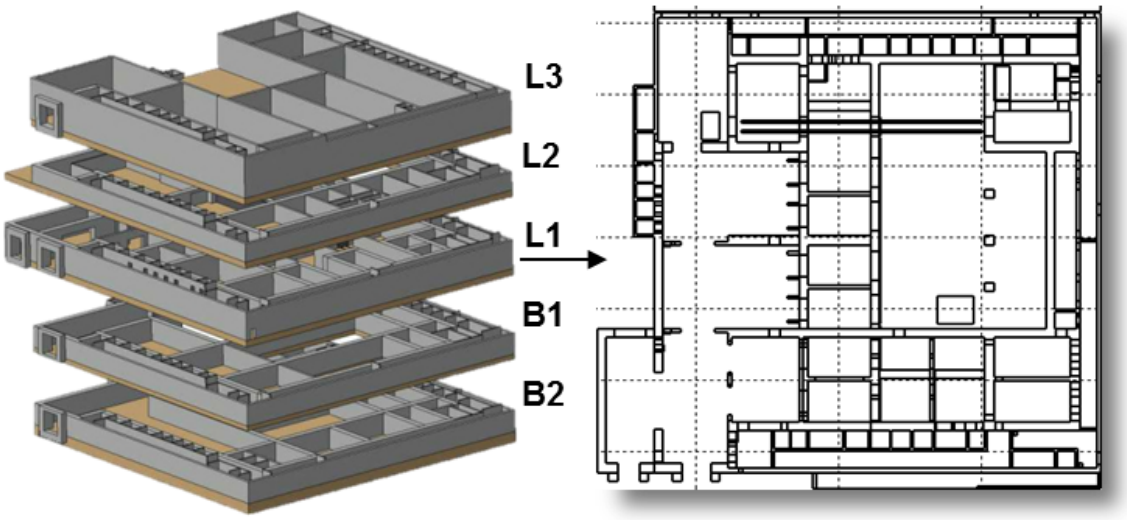


Figure 6.3: The five main levels of Hot Cell Building in ITER (left) and the 2D representation of the level L1 (right).

6.2.2 Transport cask

The vehicle, represented in Figure 6.1, is a large and complex unit to transport heavy and contaminated loads between the TB and the HCB. The geometry of the vehicle and its payload vary according to the cask and the components to be transported and hence, different vehicle typologies will operate. The vehicle is composed by three sub-systems: the cask envelope, the Cask Transfer System (CTS), and the pallet. The cask envelope is a container that enclosures the in-vessel components and the RH tools to be transported. The CTS acts as a mobile robot. The pallet is the interface between the cask and the CTS. It is equipped with a handling platform to support the cask load and help on docking procedures. When underneath the pallet the CTS transports the cask, but it can also move independently of the pallet and cask.

The CTS is equipped with two pairs of drivable and steerable wheels: one for nominal operation and the other for redundancy, operating in case of failure of the first pair, [114]. These locomotion wheels are installed along the longitudinal axis of the vehicle, providing the rhombic like capabilities. Since the locomotion wheels are installed along a straight line, there are free wheels in the vicinity of the boundaries of the vehicle's shape to assure the CTS's stability. For simplicity and from this point forward, the CTS is only represented with a single pair of drivable and steerable wheels, identified as 'F'ront and 'R'ear wheels, as illustrated in Figure 6.4. This configuration gives the vehicle a higher maneuverability in confined spaces than the

traditional cars with Ackerman or tricycle configurations [106].

As illustrated in Figure 6.4, consider the state vector $q = [x_c\, y_c\, \theta]$ as a representation of the vehicle pose in the frame {I}, with (x_c, y_c) the coordinates of the center of the vehicle and θ the orientation of the vehicle. Also, consider v as the longitudinal speed and β the controllable side-slip angle of the vehicle, both defined in {I}. A kinematic model for a rhombic like vehicle in {I}, that allows the simulation of the vehicle motion directly through the desired longitudinal speed v, instead of imposing an individual linear speed for each wheel, was introduced in [115] as:

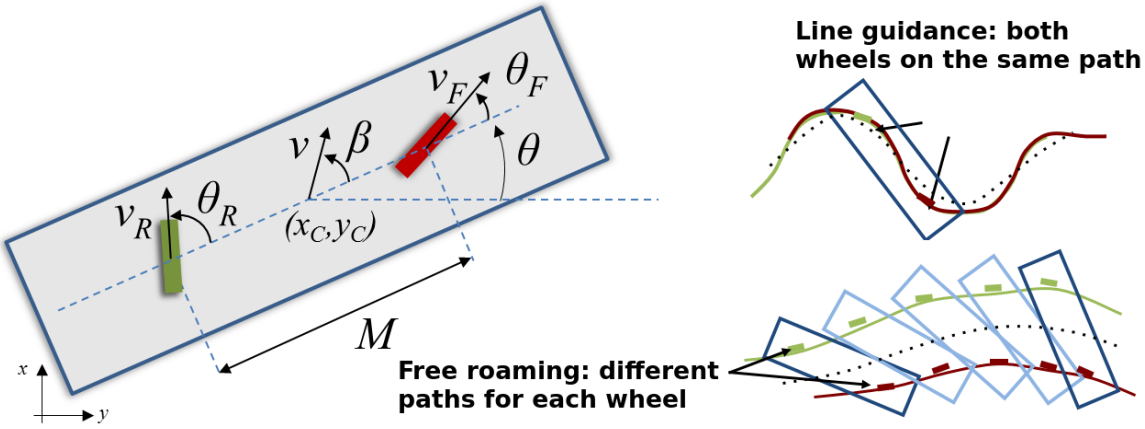


Figure 6.4: Rhombic vehicle model and the possible path following approaches.

$$\begin{bmatrix} \dot{x}_c \\ \dot{y}_c \\ \dot{\theta}_m \end{bmatrix} = \begin{bmatrix} \cos(\theta+\beta) \\ \sin(\theta+\beta) \\ \frac{\cos\beta\cdot[\tan\theta_F - \tan\theta_R]}{M} \end{bmatrix} \cdot v, \tag{6.2}$$

where

$$\beta = \arctan\left(\frac{v_F \cdot \sin\theta_F + v_R \cdot \sin\theta_R}{2 \cdot v_R \cdot \cos\theta_R}\right) \tag{6.3}$$

and

$$v = \frac{v_F \cdot \cos\theta_F + v_R \cdot \cos\theta_R}{2 \cdot \cos\beta}. \tag{6.4}$$

This modeling entails that the wheels of the vehicle roll without slipping, a constraint inherent to the nonholonomy of rhombic like vehicles, and also considers a rigid body constraint, common to this type of vehicles, as follows:

$$v_F \cos\theta_F = v_R \cos\theta_R. \tag{6.5}$$

The values v_F, v_R, θ_F and θ_R are the inputs to guide the vehicle, as detailed in [116].

6.2.3 Missions

The maintenance operations of transportation in ITER require the vehicle to move throughout the cluttered environments of the TB and the HCB, i.e. a mesh of paths between the target poses inside the buildings. For instance, a mission of transportation of a load for refurbishment requires a path between a port and the lift in TB and then between the lift and a docking port in HCB.

6.2.4 Optimization criteria

During a mission the vehicle describes a swept volume when follows its path. The volume is important given the free space available in the scenario, or given other parked vehicles. The speed of the path following is also relevant not only for the mission execution time, but in particularly given the dynamics of the vehicle, since it can reach up to 100 tons. As a result, each mission requires an optimized trajectory.

The trajectory optimization problem stated for the vehicle consists on evaluating a trajectory, i.e. a geometric path defined by a set of N points P_i, i.e., $S = \{P_0, P_1, \ldots, P_N\}$, combined with a speed profile. The geometric path must guarantee that the vehicle departs from the initial configuration q_S and achieves the specified goal q_F, without colliding with obstacles and keeping a safety margin. The trajectory optimization has three stages: the geometric path evaluation, the path optimization and the trajectory evaluation. The geometric path evaluation aims to find a path connecting the initial and goal configurations. This path acts as an initial condition for the path optimization stage. The geometric path evaluation is implemented using FM^2. The path optimization receives the preceding geometric solution as input and returns an optimized path. The optimization process, described in Section 6.4, first applies a spline interpolation to satisfy weaker differential constraints such as smoothness requirements. Afterward, a clearance based optimization is carried out to guarantee a collision free path that meets the safety requirements. In general, a minimum safety distance between the vehicle and the obstacles must be guaranteed. Finally, the trajectory evaluation defines the velocity function along the optimized path transforming it into a trajectory, which is the final output.

6.3 Current solution: a three-step approach

The vehicle is required to move along a path that simultaneously maximizes the clearance to obstacles and reducing the distance between the start and goal poses (position and orientation). The proposed motion planning methodology is based on a three step approach [7], as shown in Figure 6.5:

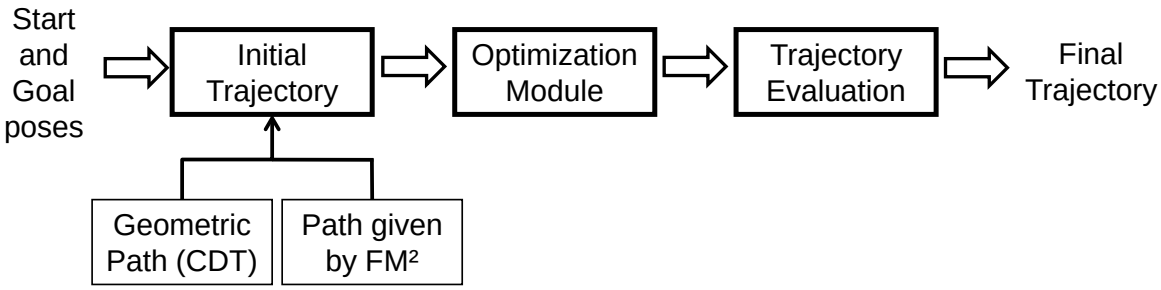


Figure 6.5: Workflow for trajectory optimization. This Thesis studies a new implementation of the Initial Trajectory block and how it affects remaining blocks.

Geometric path evaluation: given the environment model and initial and goal poses, an initial geometric path is found. At this point the aim is to find a path connecting the initial and goal objectives that can act as an initial condition for the next path optimization step. The contribution of this Thesis focuses on this stage, as FM^2 is proposed as planner instead of the current.

Path optimization: this module receives the preceding geometric solution as input and returns an optimized path. The results in this Thesis use the path optimization procedure detailed in previous ITER works.

Trajectory evaluation: in this final module, the speed profile is defined along the optimized path transforming it into a trajectory, which is the output of the proposed motion planning methodology. As for path optimization, this Thesis uses the same trajectory evaluation method than in previous ITER works.

6.3.1 Geometric Path Evaluation

From the 3D CAD models, a 2D representation is obtained by projecting at floor level all the relevant elements that might conflict with the CTS. The levels of TB and HCB are well structured scenarios that can be modeled as a set of planar walls, whose footprint is a line segment and thus the 2D map can be considered as a set of line segments. The 2D map is decomposed into a set of triangles, by using CDT, to account for all walls. Afterward the algorithm determines the set of sequence of triangles that contain and link the start and goal positions. Each sequence of triangles is then converted into a sequence of points (mid point of the common edge of two consecutive triangles) yielding a path. The shortest and feasible path is chosen as the geometric path, acting as the initial condition for the path optimization step.

The novel approach proposed consists on substituting CDT by FM^2, as shown in Figure 6.5 and analyze the performance in terms of robustness and how it is affected by the following path optimization procedure. Comparisons against the CDT-based path initialization are also included.

6.3.2 Path Optimization

The initial geometric path does not guarantee a collision free path for a rigid body, such as the CTS, as shown in the second image of Figure 6.6 and thus may be unfeasible. Moreover this path is not smooth. To obtain an optimized path, two criteria are included in the algorithm [7]: clearance from obstacles, by increasing the distance from the vehicle to the walls and path smoothness, entailing getting shorter and smoother paths without slacks. To address the referred issues, the optimization step uses the elastic band concept [117], where the path is modeled as an elastic band, similar to a series of connected springs, subjected to two types of forces: internal and external forces. The first are the internal contraction forces, whose magnitude is proportional to the amplitude of displacement and determine that the path becomes retracted and shorter. The repulsive forces are responsible for keeping the path, and consequently the vehicle, away from obstacles. As detailed in [7], the kinematics of the vehicle are taken into account in the optimization step. Even though unfeasible trajectories may occur only given the maximum angle achieved by the orientation of the wheels and the respective velocities. The proposed method is able to identify these cases. The path optimization procedure is detailed in Section 6.4.

6.3.3 Trajectory Evaluation

The final trajectory is obtained by defining the speed of the vehicle at each point of the optimized path. In order to reduce the risk of collision in the case of major malfunction, the speed is reduced once the distance to the nearest obstacle decreases below a threshold value. Dynamic constraints, such acceleration and speed limits, are also considered [7].

The entire process is summarized in Figures 6.6 for CDT-based initialization and 6.7 for FM^2-based initialization. In both cases, the output is an optimized path in terms of distance and smoothness, with a speed profile, assuming that the vehicle starts and ends with velocity equal to zero.

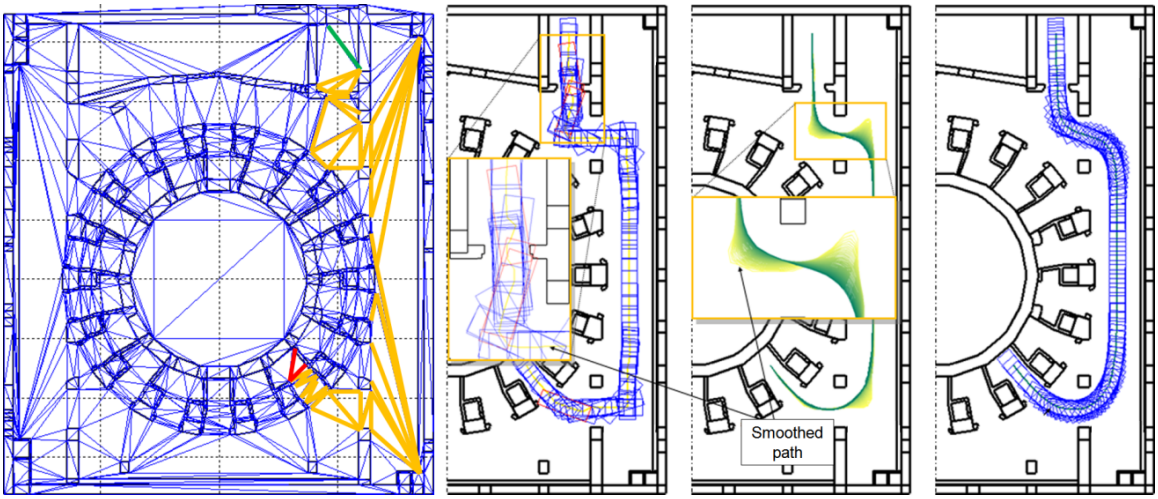


Figure 6.6: From left to right: the initial map with the generated Constrained Delaunay Triangulation and the computed sequence of triangles between start and goal points, initial geometric path, path optimization and final optimized trajectory [7, 8].

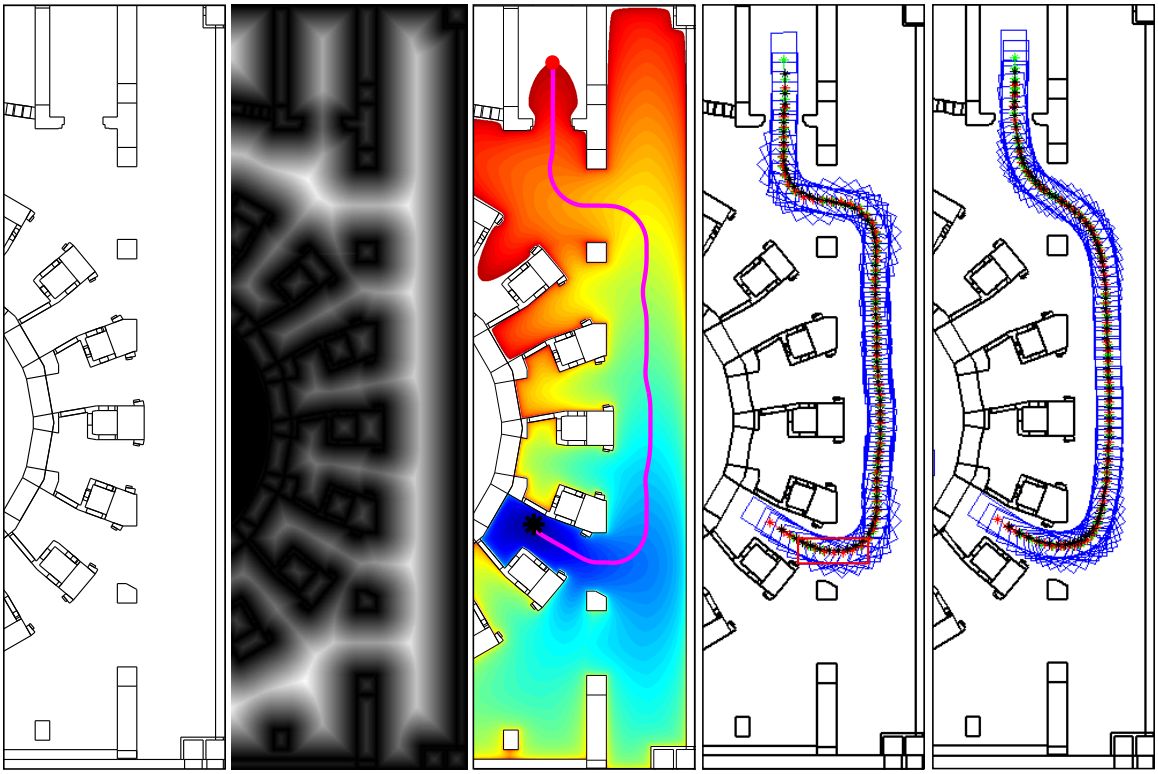


Figure 6.7: Steps of the proposed method on level TB/B1 (from left to right): initial map, velocity map, time-of-arrival map and FM^2 path, FM^2 path evaluated with the cask with a collision, and path after the the optimization.

6.3.4 Geometric Path Evaluation Issues

A geometric representation of the environment has some advantages since it provides a very accurate representation of the scenario and it does not depends on grid cell discretization. However, in complex scenarios, the geometric representation results in some issues. In particular with the CDT, the representation requires a huge number of triangles, yielding to a computational effort. In addition, the CDT may result on sharp initial paths still far from the optimal one, as shown in Figure 6.8. As a consequence, the optimization takes longer. To overcome these issues, in this Chapter a new initialization method is proposed based on FM^2.

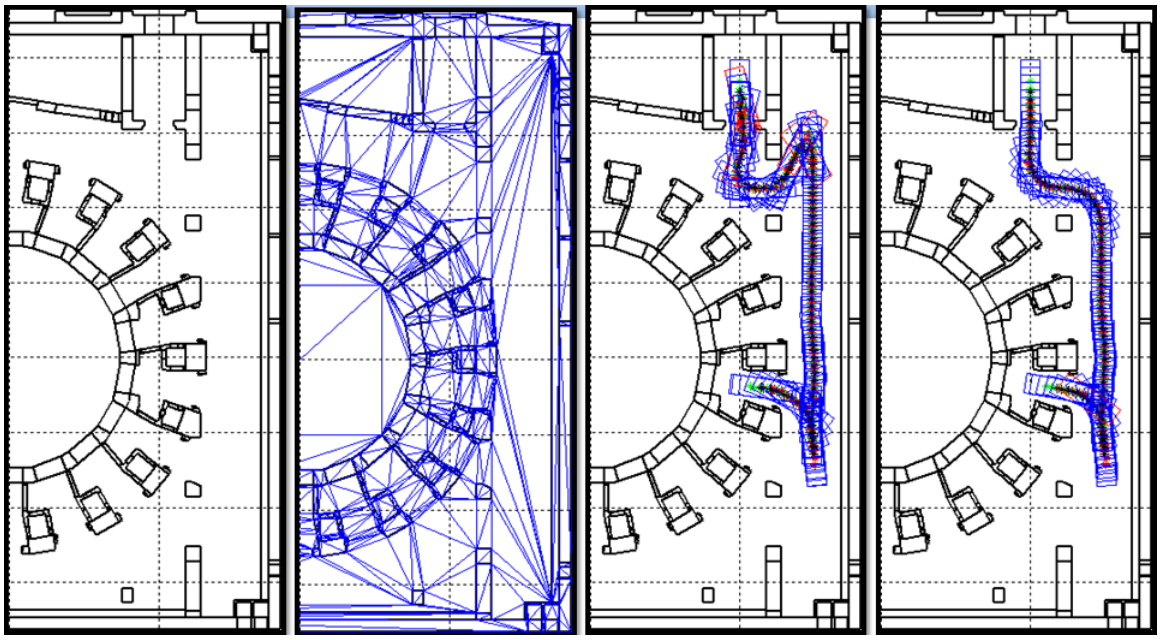


Figure 6.8: From left to right: map of level B1 in Tokamak Building, generated Constrained Delaunay Triangulation of the map, geometric path obtained from Constrained Delaunay Triangulation and path obtained with Fast Marching Square.

6.4 Path optimization and trajectory evaluation

An optimization methodology based on the elastic bands method [117] was designed [118]. The original concept associated with this approach appeared in the computer vision field, with the presentation of the so called "snakes" algorithm [119]. A snake is a deformable curve guided by artificial forces that pull it towards image features such as lines and edges. The solution herein proposed with the elastic bands methodology is similar to the snakes approach. Instead of retracting a curve to image features,

in the path planning problem, it repels the path out from obstacles. Following this approach, the path is modeled as an elastic band which can be compared to a series of connected springs subjected to two types of forces, as illustrated in Figure 6.9:

- Internal forces or elastic forces: the internal contraction force simulates the Hooke's elasticity concept [120, 121], i.e., the magnitude force is proportional to the amplitude of displacement. This modeling approach allows the simulation of the behavior of a stretched band. This is the reason why the paths become retracted and shorter.
- External forces or repulsive forces: the obstacle clearance is achieved using repulsive forces to keep the path, and consequently the vehicle, away from obstacles.

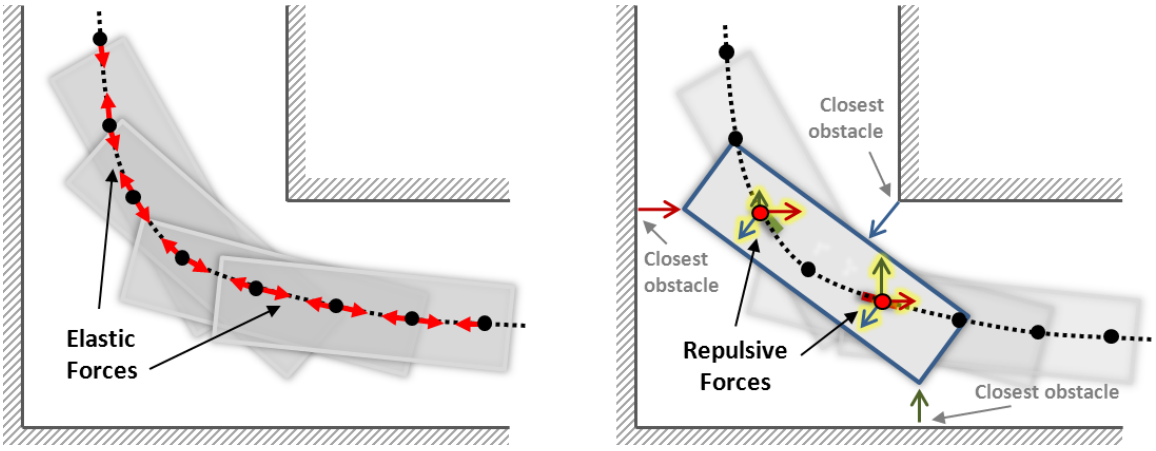


Figure 6.9: Elastic band concept: elastic forces to smooth the path (left) and repulsive forces generated by the closest obstacles (right).

When submitted to these artificial forces, the elastic band is deformed over time becoming a shorter and smoother path, increasing clearance from obstacles. Hooke's law evaluates the elastic force F_e applied to path point P_i as:

$$F_e(P_i) = k_e \cdot \left[(P_{i-1} - P_i) - (P_i - P_{i+1})\right] \tag{6.6}$$

where k_e is the elastic gain and P_{i-1} and P_{i+1} are the path points adjacent to P_i. The elastic band behavior can be controlled through k_e. The band stretches with high values of k_e while low values increase the band flexibility.

To determine the external forces, a collision detector algorithm is used to evaluate the nearest obstacle point (OP) to each vehicle pose. The use of a single OP to determine the repulsive forces may not be satisfactory to maintain clearance from

obstacles, and therefore, a larger set of obstacle points, such as the k-nearest (k-OPs), must be considered, as illustrated in Figure 6.9. This leads to a more balanced repulsive contribution ensuring effectiveness on most situations.

The repulsive force for each P_i is determined as a combination of different repulsive contributions

$$F_r(P_i) = k_r \cdot \sum_{l=\{F,R\}} \sum_{k=1}^{K} r_{l,k}(P_i) \tag{6.7}$$

where k_r is the repulsive gain, F and R the front and rear wheels and $r_{l,k}$ is the inverse of the distance between the k-OPs and the vehicle, considering the front and rear wheels, as detailed in [7].

Once the elastic and the repulsive forces are computed, an update equation procedure that defines the path evolution along each iteration is applied as

$$P_{i,new} = P_{i,old} + k \cdot F_{total}(P_{i,old}) \tag{6.8}$$

where k is a normalization factor adding up the total force contribution applied to all points $P_{i,old}$ and the total force contribution is given by

$$F_{total}(P_{i,old}) = F_e(P_{i,old}) + F_r(P_{i,old}) \tag{6.9}$$

Under the influence of these artificial forces, the elastic band is deformed over time becoming a shorter and smoother path. The stopping criteria is defined by detecting that the magnitude changes on F_{total} are smaller than a given threshold and by setting a maximum number of iterations. The path optimization is thus carried out by a path deformation approach where the computed paths are treated as flexible and deformable bands. Elastic interactions smooth the path by removing any existing slack, whereas repulsive forces improve clearance from obstacles by pushing away the points of the path.

In Figure 6.10 it is shown a schematic of the evolution of the path during the optimization step. The path connects the fixed initial and final points and has 3 additional points. The index **n** identifies the iteration of the optimization step.

At each iteration it is computed the variation of the path in relation to the previous iteration. The variation is computed evaluating the distance between each point of the path at iteration **n** and the line segment defined by the two nearest points of the path from iteration **n-1**, as illustrated in Figure 6.11. The 20 highest distances are selected and their median is computed. The stopping criteria is achieved when the median value is below a threshold (defined as 0.02 m) or if the number of iterations reaches 70. These values have shown good performance in all cases so that they are fixed for all results along this Chapter.

The output of the path optimization is a collision free path suitable for execution. Then, the optimized paths are parametrized in terms of velocities, converting the

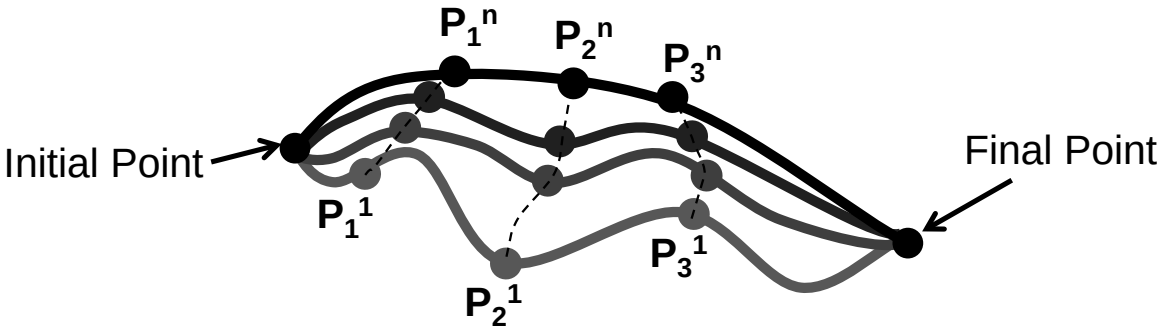


Figure 6.10: Schema of the path evolution in each iteration during the trajectory optimization.

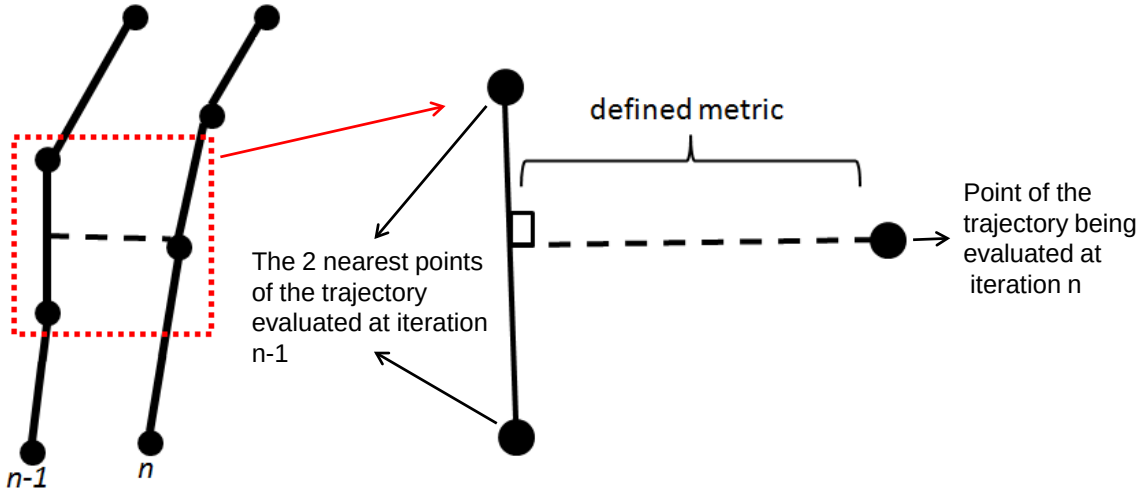


Figure 6.11: Definition of the variation of the path between consecutive iterations: distance evaluated to a single point.

paths into trajectories. Since the safety requirements are mandatory and the risk of collision shall be reduced in the cluttered environment, an initial approach defines the vehicle speed profile as a function of the distance to the obstacles. The velocity assumes low values when the vehicle is closer to the obstacles. Otherwise, the velocity is higher, under safety levels. To generate this initial speed profile, the minimum distance from the vehicle to the closest obstacle is identified for each point in the optimized path.

A maximum and minimum allowable speed are set to this profile, in order to integrate kinematic constraints. The safety margin identifies the threshold distance above which the maximum speed is considered. The speed profile thus obtained is saturated when the distance is above the threshold or below the safety margin and is referred as C-based speed profile [8]. However, the C-based speed profile is unable to handle vehicle dynamics constraints, meaning that the constraints on the admissible accelerations of the vehicle are ignored. To sidestep this issue, it has been developed a specific routine, which tests each one of the C-based speed profile transitions, checking whether the accelerations are feasible or not. Whenever a dynamic unfeasible transition is found (e.g., the calculated acceleration is out of the admissible bounds), the routine corrects the speed accordingly.

6.5 Simulated Results

FM^2 was implemented in MATLAB environment (using compiled MEX functions) and integrated in the specially designed software tool Trajectory Evaluator and Simulator (TES), developed at the Instituto de Plasmas e Fusão Nuclear (IPFN), Instituto Superior Técnico, University of Lisbon. The TES receives the models of the buildings, generates trajectories using line guidance and free roaming, evaluates the resulted 3D volume swept by the vehicle along the optimized paths and exports the optimized trajectories and the corresponding 3D swept volume directly to the CAD software. The TES provides also a GUI to preview the trajectory optimization, to manipulate the scenarios, to easily choose the vehicle typology to be used in the simulation and to generate results. The output of TES is a set of optimized trajectories which were validated by an independent software developed by ASTRIUM SAS [122]. The results achieved by the algorithms implemented in TES applied in the models of the real scenarios were important to proceed with the construction of the Tokamak Building.

The line guidance algorithm, using the FM^2 and the elastic bands optimization, were applied and tested in some levels of the TB and HCB to generate trajectories. Each optimized trajectory corresponds to a nominal operation of transportation between the lift and a port in the TB or a parking place in the HCB. Each mission only constrains the initial and final poses (positions and orientations) of the vehicle. A path is considered feasible when the minimum distance between the vehicle and the

closest elements of the scenario is above a safety margin. This minimum clearance is only allowed to be infringed when entering/exiting the lift and in the docking phase.

First, an individual result of the optimization procedure is included before proceeding with the full results. This example is a mission from the lift to the port 10 of the level L1 of the TB, since it is one of the most complicated cases. Figure 6.12 shows that the FM^2 initialization contains points in the path with collisions, since the standard FM^2 in 2 dimensions does not take into account the size of the vehicle when planning. However, the optimization provides a shorter, smoother path without collisions. In Figure 6.13 the spanned areas for both initialization and optimization are shown. It is possible to see that the optimization has reduced the spanned area, since most of the FM^2 small oscillations have been reduced. Finally, Figure 6.14 shows the evolution of the minimum clearances and the velocities profile. Clearances are improved in those places in which the initialization had collisions. In some points of the path clearances are decreased (always above the safety margin, depicted by a dashed line) in order to reduce path length. This causes the speed profile to decrease in those places as well.

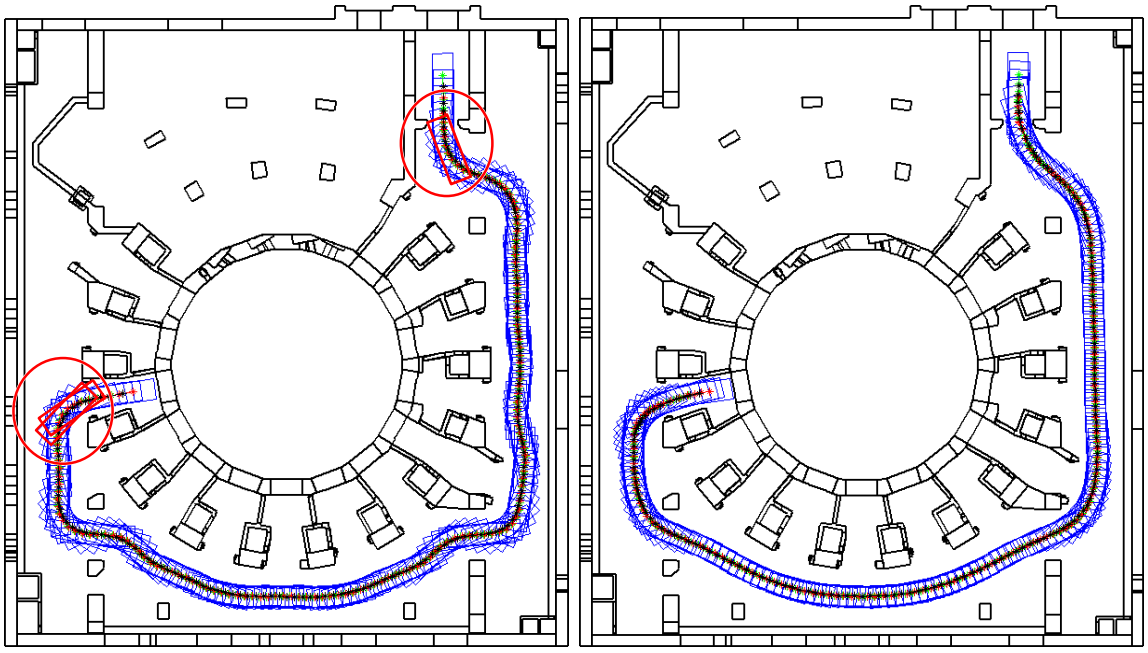


Figure 6.12: The path evaluation from the lift to the port 10 in level L1 of TB: the results from the initialization step (left) and from the optimization step (right).

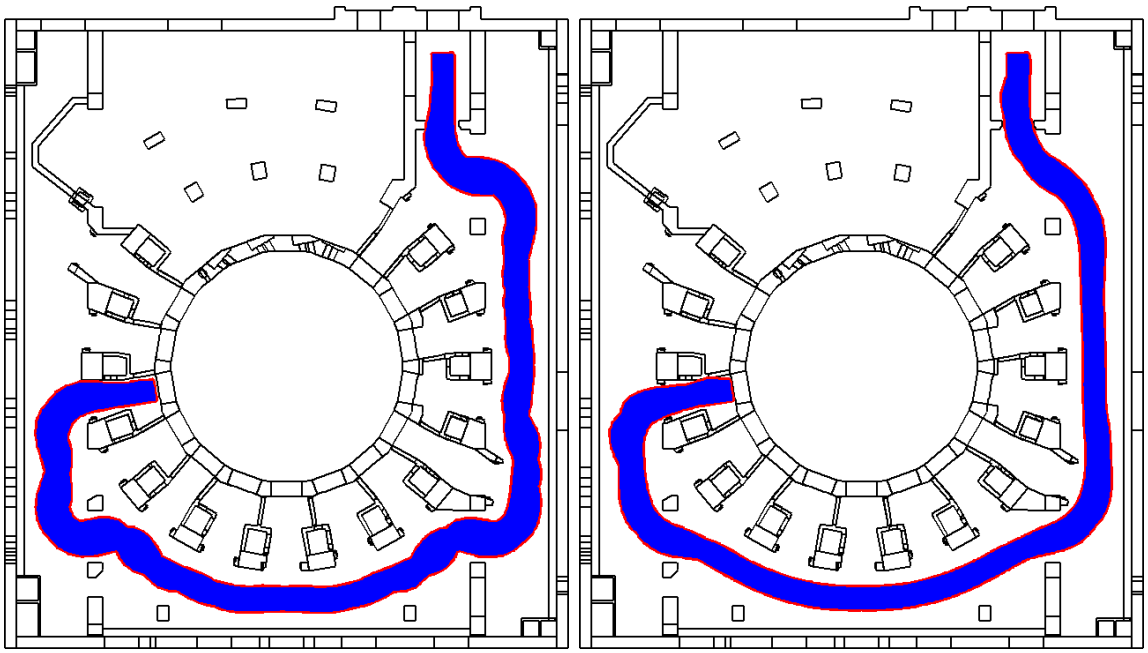


Figure 6.13: The spanned areas along the initialized path (left) and along the optimized path (right), from the lift to the port 10 in level L1 of TB.

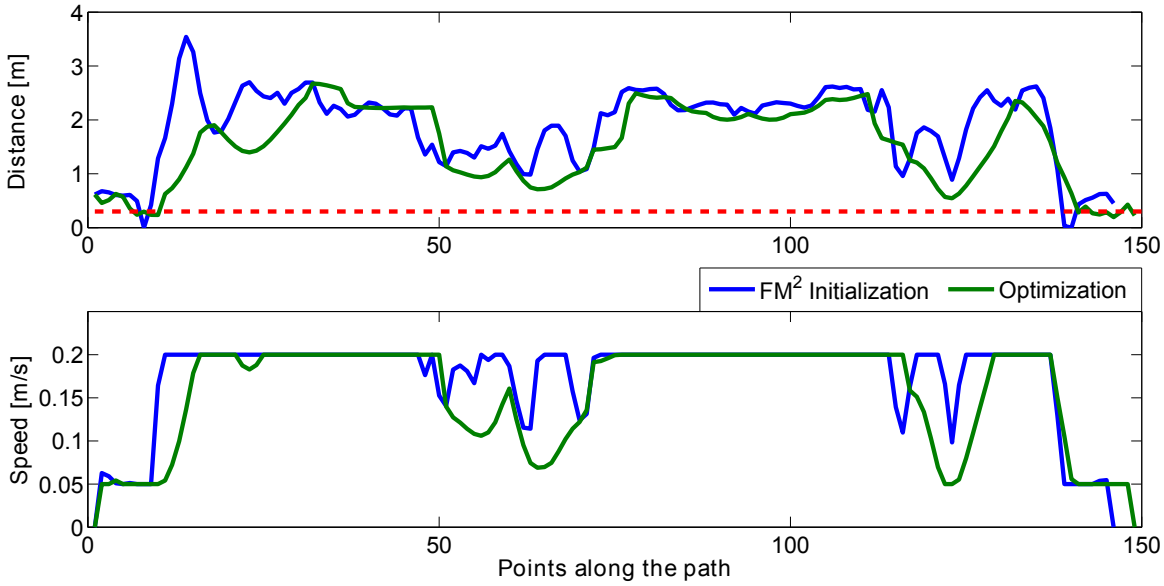


Figure 6.14: The minimum distance between the vehicle and the closest obstacles (top) and the speed of the vehicle (bottom) along the optimized path, from the lift to the port 10 in level L1 of TB.

A total of 47 missions have been analyzed between the 2 buildings: 35 missions in the TB (7 in level B1, 14 in level L1, and 14 in level L2) and 12 in the HCB (7 in level L1 and 5 in level B2). Different metrics are used for both initialization and optimization: path length, swept area, total vehicle rotation angle, estimated execution time and minimum clearance along the path. The results of these metrics, except clearance, are shown in Figure 6.15.

The optimization procedure has reduced the path length in all cases. Also the swept area has been optimized in all the missions. The total steered angle by the vehicle is also significantly reduced since most of the oscillations created by the FM^2 initialization have been eliminated. However, it is increased in 3 missions in the TB and in all missions in the HCB/L1. These cases corresponds to sharp curves in confined spaces in which the only option for the vehicle is to get a better angle by first turning to the opposite side. As a counterpart, the estimated path execution time has been increased in all cases.

Smoothness require a more careful analysis. The smoothness metric employed creates triangles formed by consecutive path segments and compute the angle between those segments using the Pythagoras' theorem. Then, its conjugate angle is normalized by the path segment. Finally, all the normalized angles along the path are added. The higher this value is, the less smooth the path is. Minimum value is 0 for a straight line and there is no maximum value. Therefore, it is important that both initial and optimized paths have the same number of points. Otherwise, the comparison would not be fair. This metric is formally defined as:

$$smoothness = \sum_{i=2}^{n-1} \left(\frac{2\left(\pi - \arccos\left(\frac{a_i^2+b_i^2-c_i^2}{2a_ib_i}\right)\right)}{a_i + b_i} \right)^2 \tag{6.10}$$

where $a_i = dist(s_{i-2}, s_{i-1})$, $b_i = dist(s_{i-1}, s_i)$, $c_i = dist(s_{i-2}, s_i)$, s_i is the i^{th} state along the path, and $dist(s_i, s_j)$ gives the distance (Euclidean in this case) between states i and j.

Smoothness results are shown in Figure 6.16. Smoothness is worsened in all cases. However, this is not an actual negative result. Note that the smoothness values for the initial FM^2 paths is low (highly smooth). However, these paths present collisions. Therefore, the optimization procedure decreases smoothness only as much as required in order to satisfy the clearance requirements. On the other hand, the total rotation angle has been decreased in most cases.

The optimization provokes a redistribution of the minimum clearance along the trajectories which have to be carefully analyzed. Table 6.1 shows that the percentage of points with clashes have been reduced to 0 in all TB levels. However, the amount

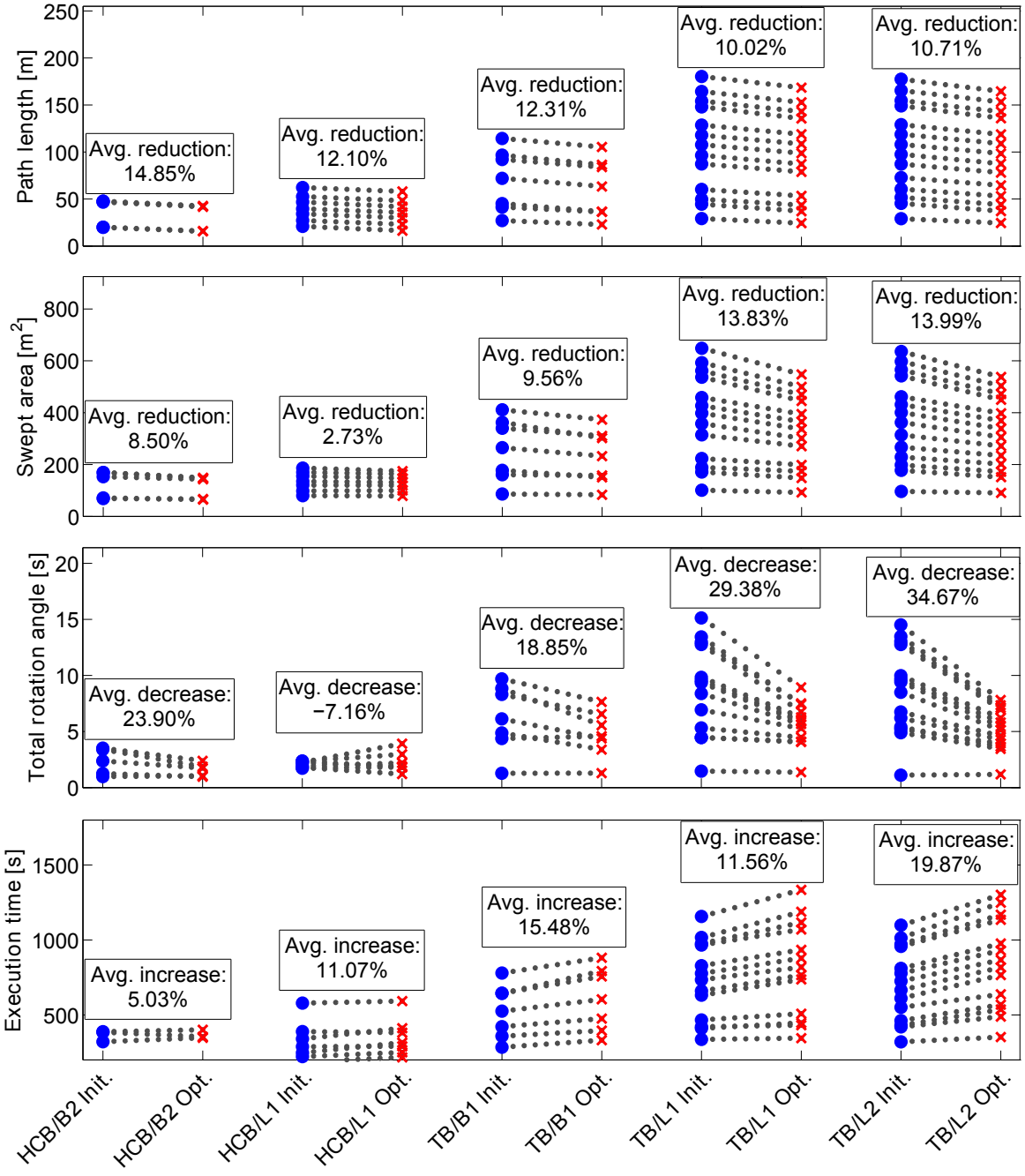


Figure 6.15: Metrics comparison for different levels in TB and HCB: initialization (Init.) versus optimization (Opt.).

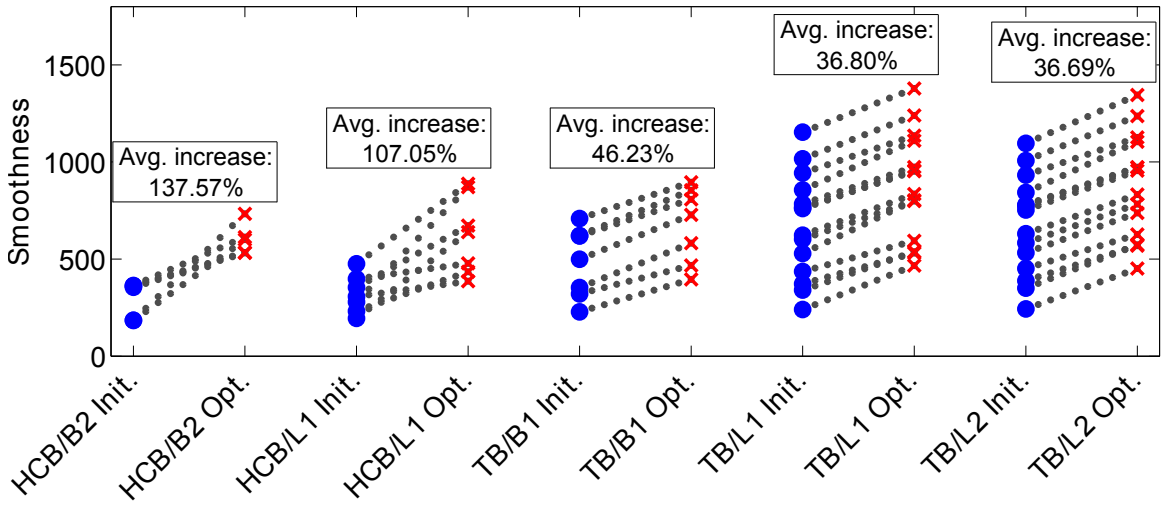


Figure 6.16: Smoothness metric comparison for different levels in TB and HCB: initialization (Init.) versus optimization (Opt.).

of points below the minimum clearance (0.3m) have increased. In TB there are two critical places in which it is not possible to accomplish this restriction: the lift exit and the docks gate. Therefore, the optimization *sacrifices* points in the surroundings of those critical places in order to avoid clashes by bringing them closer to the obstacles. This allows paths to have longer but smoother curves. The FM^2 method provides paths close to the optimal in terms of obstacle clearance. However, the optimization, in order to reduce oscillations and path length, decreases the clearance also in some of the points which are already far from obstacles. Thus, the percentage of points with clearance higher than a meter decreases while the group between 0.5-1m increases.

Figure 6.17 illustrates how close the path returned by the FM^2 is to the final solution and how fast the optimization is. As described in Section 6.1, a path is a set of 2D Cartesian points. During the optimization, each point *moves* as a result of the elastic and repulsive forces. Figure 6.17 presents, along the z-axis, the distance between each point and its final position, along the iterations represented in the x-axis. The y-axis represents the sequence of the points of the path. At iteration 30, the distance at all points are zero, since the optimized path was achieved and the points do not move anymore (or the oscillations are not perceptible, e.g., below a small threshold value). Figure 6.17 presents large distance values in the points that correspond to the areas of the scenario with more clearance. In the places where the scenario is very narrow, the points are stuck along all the iterations.

Figure 6.18 presents all the missions to the level L1 of TB. Figure 6.18 - left shows the paths resulted form the FM^2 algorithm applied to all ports. The paths are close to the optimized solutions, but with some clashes identified by circles. Figure 6.18 -

Table 6.1: Clearance distributions for the initialization (X_i) and optimized (X_o) trajectories.

Map / Level		Clash	$(0, 0.3)$m	$[0.3, 0.5)$m	$[0.5, 1)$m	$[1, \infty)$m
HCB	$B2_i$	0 %	5.48 %	2.74 %	44.52 %	47.26 %
	$B2_o$	0 %	2.91 %	5.81 %	42.44 %	48.84 %
	$L1_i$	0.82 %	2.45 %	3.67 %	47.35 %	45.7 %
	$L1_o$	0 %	3.69 %	4.43 %	43.17 %	50.21 %
TB	$B1_i$	1.14 %	3.42 %	3.42 %	23.29 %	68.72 %
	$B1_o$	0 %	4.26 %	9.36 %	36.17 %	50.21 %
	$L1_i$	1.49 %	1.8 %	2.82 %	14.58 %	79.31 %
	$L1_o$	0 %	6.6 %	4.83 %	24 %	64.57 %
	$L2_i$	0.66 %	0.88 %	2.86 %	19.03 %	76.57 %
	$L2_o$	0 %	2.75 %	4.48 %	36.3 %	56.47 %

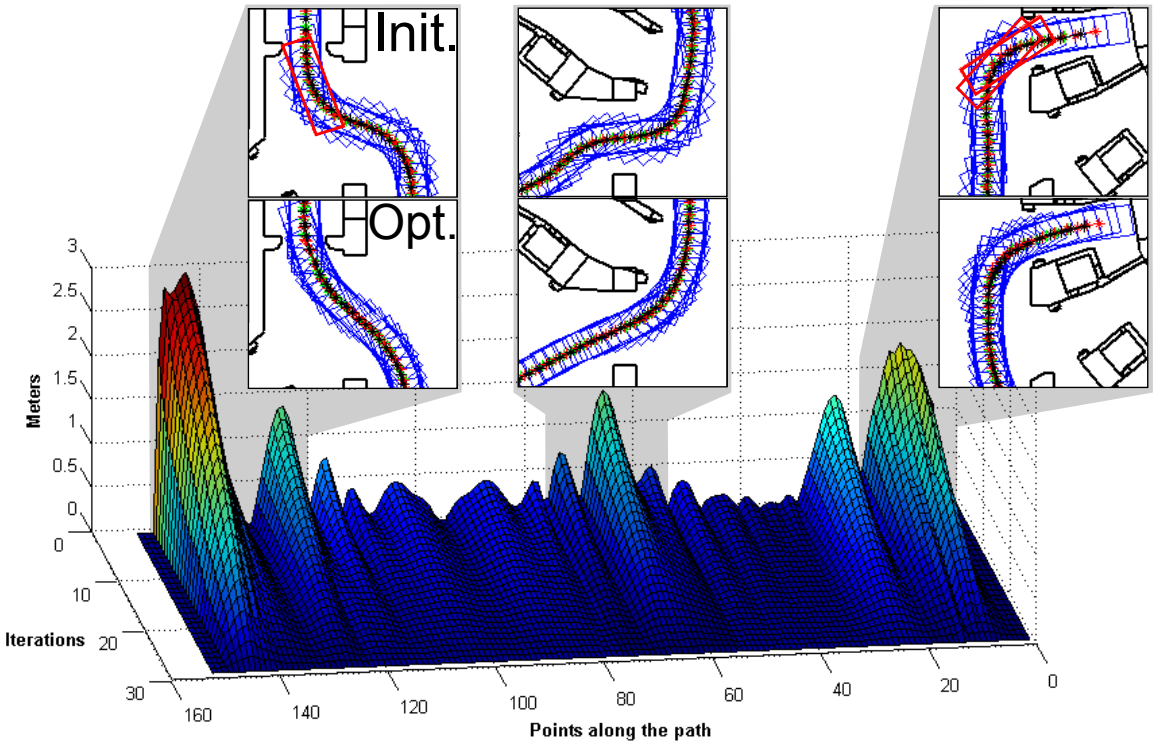


Figure 6.17: Evaluation of the distances between each point of the path along the iterations and its final value in the optimized path.

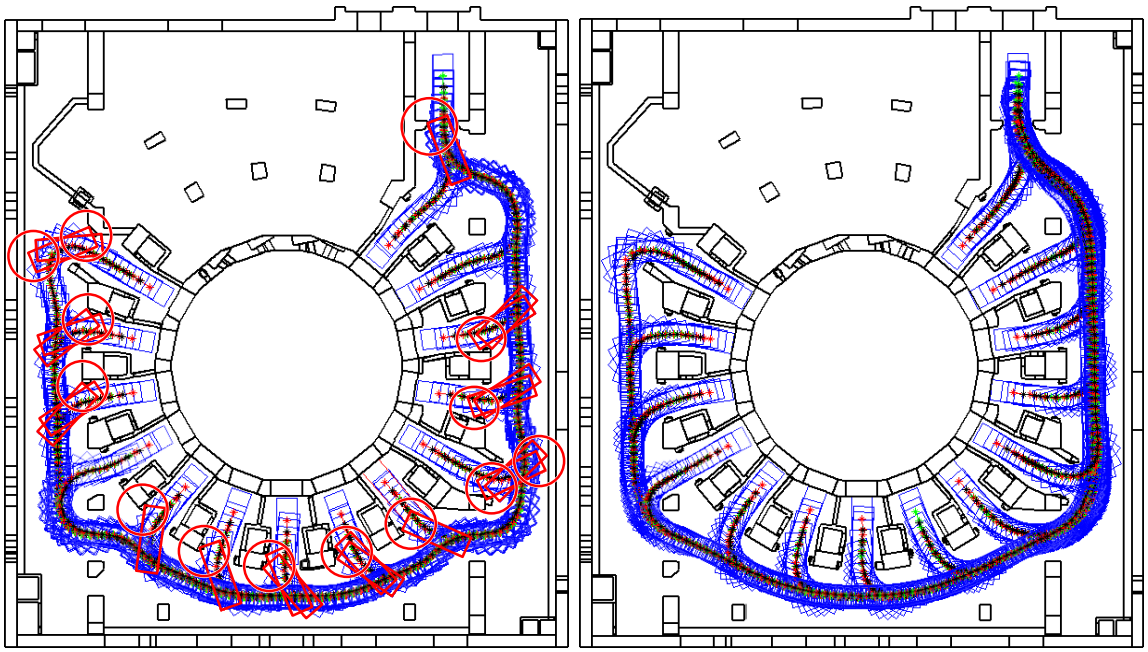


Figure 6.18: The path evaluation from the lift to all ports in level L1 of TB: the results from the initialization step (left) and from the optimization step (right).

right shows all the optimized paths. In some situations, as in the mission to port 11, the clearance of the path returned by the FM^2 in the vicinity of the pillars is greater when compared to the optimized path to the same port. However, the optimized path has no collision in the entrance to the same port while satisfying the clearance constraints. Figure 6.19 includes all the trajectories studied for the HCB. In this case the optimization is not essential as most of the paths are collision-free. However, optimization is applied in order to guarantee that the requirements are accomplished. Figure 6.19 - right shows the effect the optimization algorithm has in very cluttered scenarios: before entering the parking place, the cask has to get away from the wall in order to obtain a better angle to enter. As result, the total rotation angle is incremented.

Finally, an interesting point is raised. Different scenarios require different FM^2 velocity maps. However, for small modifications in the scenario it is not necessary to recompute the velocity map, since the optimization procedure will successfully adapt the path. For instance, Figure 6.20 - left shows an optimized path between the lift and a parking place where the initial map and the respective velocity map did not considered the parked vehicle. Running again the optimization algorithm, the new optimized path is still smooth, but without clashes, as illustrated in Figure 6.20 - right.

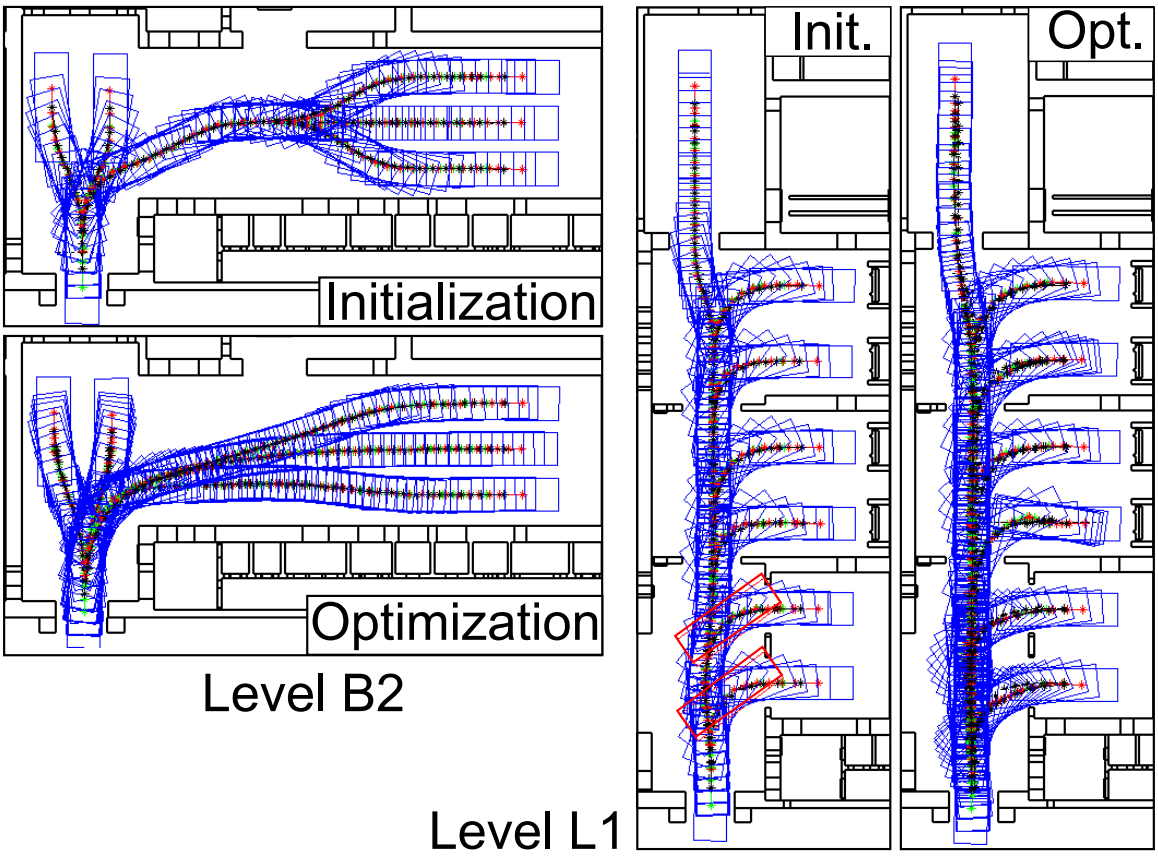


Figure 6.19: Path initialization and optimization from the lift to main parking places in levels L1 and B2 of HCB.

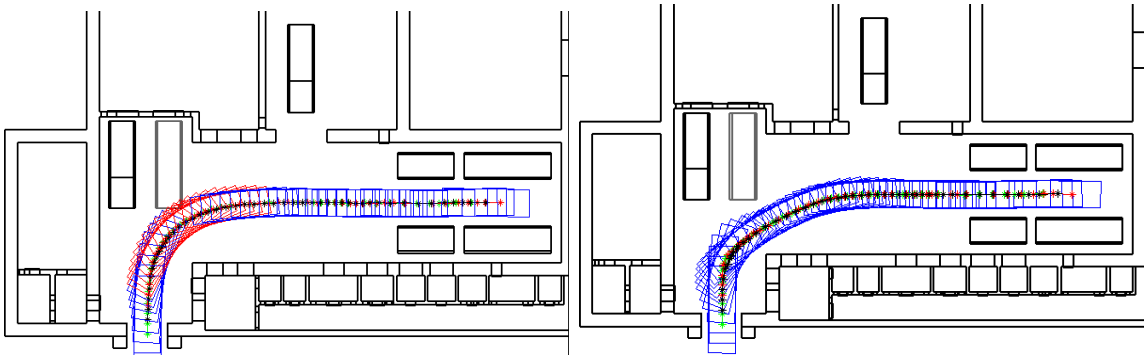


Figure 6.20: Path for a parking place in level B2 of HCB, in collision with a temporary vehicle (left), and the re-optimization of the path without the need of the initialization step (right).

The parameters k_e and k_r play an important rule for tuning the final trajectory in terms of shortness and smoothness. The k_e regulates the elastic path behavior. Higher values increase the path shortness approaching the path points connectivity. Lower values allow to increase path flexibility to obstacle-repulsive deformation. Outsized values either make the deformation process unstable or compromise path clearance. The k_r controls the repulsion behavior by determining the preponderance of the repulsive forces from obstacles. Gain increase allows to improve path clearance. Outsized values conflict with path smoothness and connectivity. These parameters were largely tested in several scenarios with similar dimensions and layout of ITER and the best results, as the ones depicted in previous figures, were achieved with values of k_e and k_r between $[0.3; 0.4]$ and $[0.05; 0.01]$, respectively.

Lastly, the proposed framework allows to include trajectories with maneuvers [110], as shown in Figure 6.21. So far, the maneuver poses have to be manually defined and they will not be modified by the optimization procedure. This allows to increase clearance, find feasible paths where it was not possible before, and to accomplish restrictions about the orientation of the final poses within the ports.

6.5.1 Comparison against previous ITER solution

The trajectory provided by FM^2 algorithm does not take into account the kinematics of the vehicle, neither it assures that it is feasible, since clashes might occur. However, the trajectory given by FM^2 algorithm is closer to a final solution, when compared to the initial geometric path obtained with CDT, as illustrated in the right image of Figure 6.22.

Comparisons will be made considering only line guidance, as most of the nominal operations are accomplished using line guidance. For simplicity, only results of the

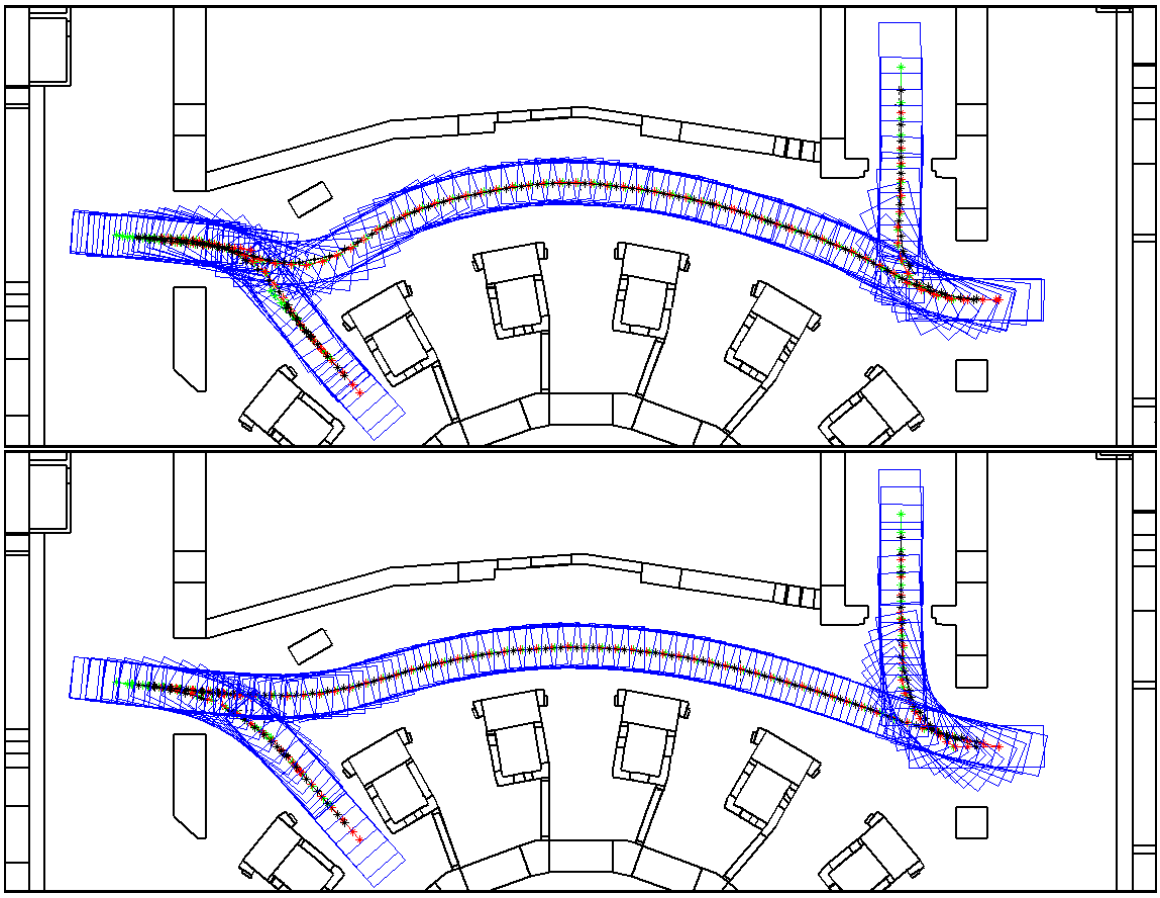


Figure 6.21: Example of a double maneuver in level B1 of TB, port 7: initialization (top) and optimization (bottom).

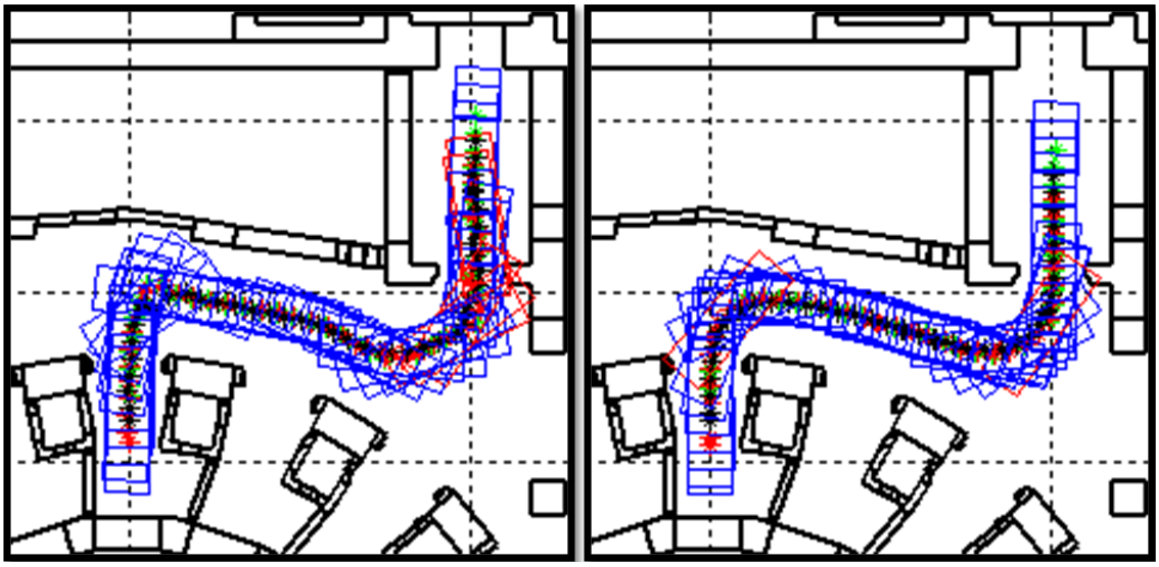


Figure 6.22: Left - initial geometric path obtained with Constrained Delaunay Triangulation; Right - trajectory obtained with FM^2.

level B1 of the TB are included. However, these results are close to those obtained for the rest of the ITER environments.

In order to assess the quality of the trajectories obtained using either CDT or FM^2 initializations a comparison criteria is defined. The trajectories are compared in terms of clearance and smoothness of the final trajectory, computation time and number of iterations required in the optimization step.

In Figure 6.23 it is presented a comparison between the initial and final trajectories using CDT and FM^2, for port 12. The map was slightly modified to reduce the effect of the triangulation in CDT in the vicinity of the lift, as illustrated in Figure 6.8. Initially both present clashes, but the initial trajectory using FM^2 is smoother. However, the final optimized trajectories using the CDT and FM^2 initialization are very similar as illustrated in Figure 6.23. The variation of the 20 highest distances and the respective median along the iterations are shown in Figure 6.24, where the methodology with the FM^2 initialization required less number of iterations to converge. The differences between the optimized trajectories are difficult to appreciate but they can be identified when comparing the minimum distances to the closest obstacle, as depicted in Figure 6.25. The trajectory using FM^2 initialization in general has slightly higher obstacles clearance.

In Figure 6.26 it is presented the initial and final results for a trajectory to port cell 7. Once again the FM^2 initialization proves to be a smoother path and takes less iterations to converge when comparing with CDT initialization, see Figure 6.27. The

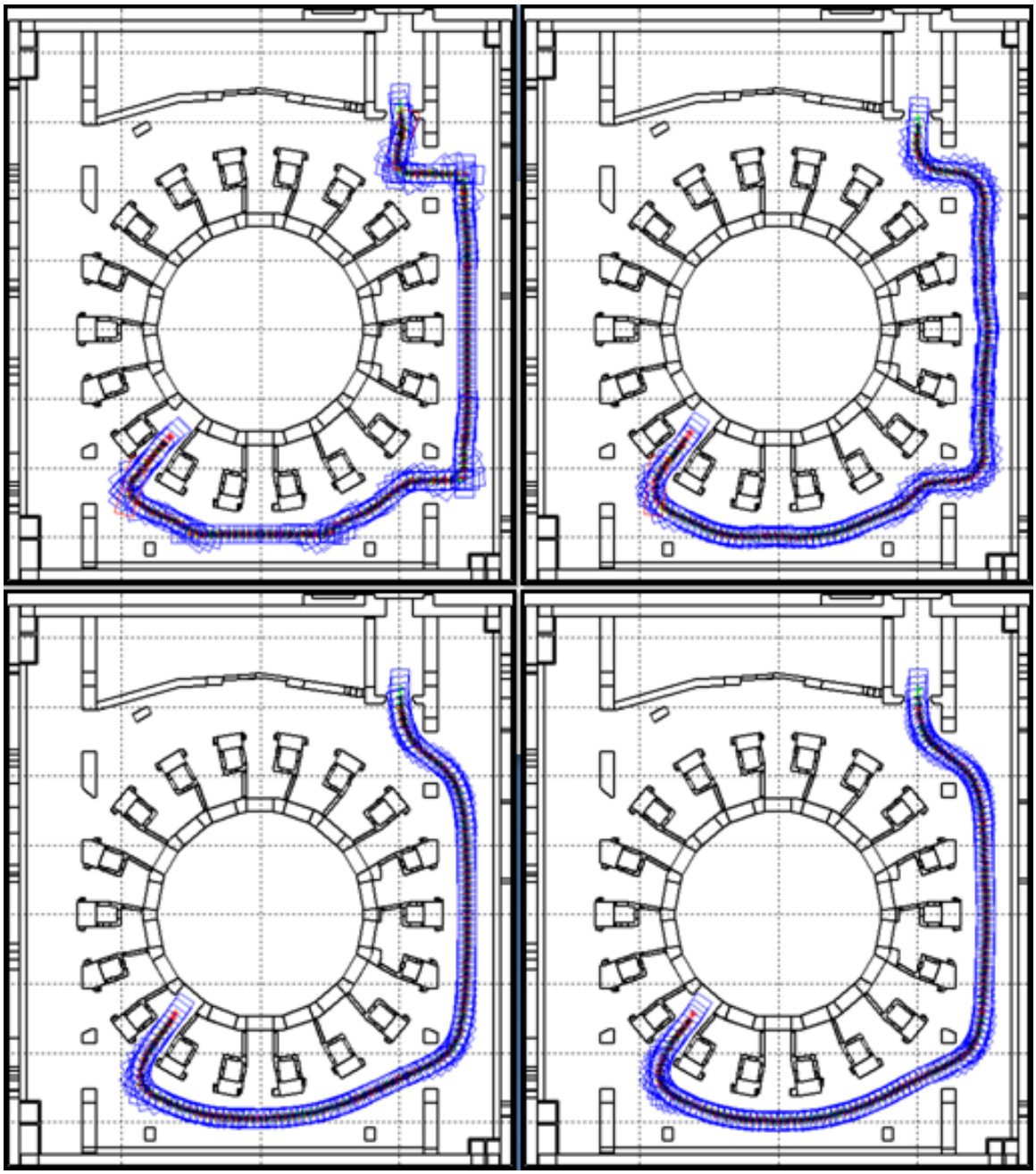


Figure 6.23: Top: the initial trajectory for port 12 using as initialization the Constrained Delaunay Triangulation (left) and the Fast Marching Square (right); bottom: the final optimized trajectory for port 12 using as initialization the Constrained Delaunay Triangulation (left) and Fast Marching Square (right).

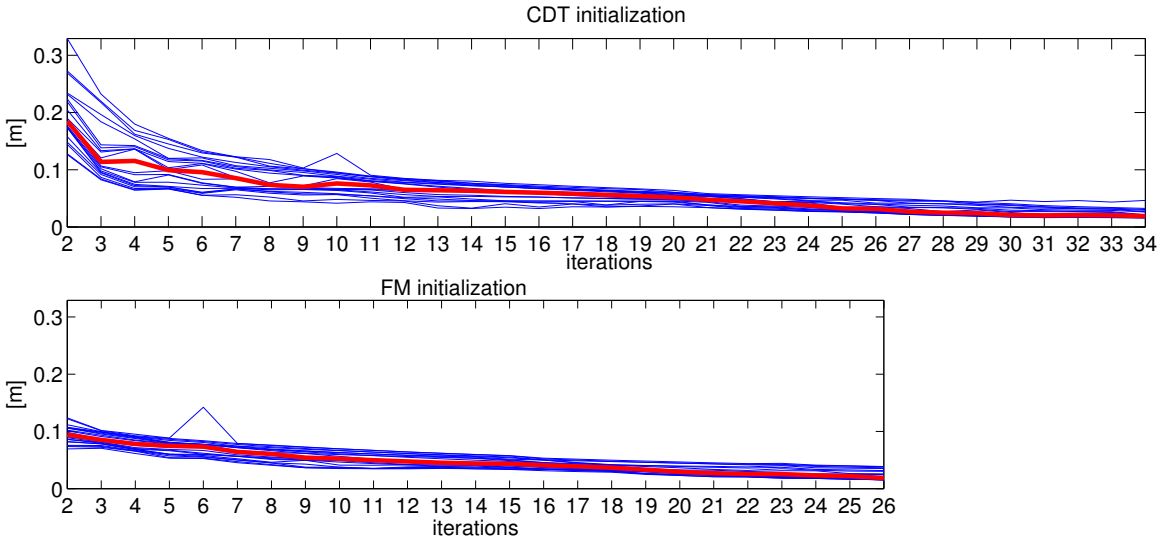


Figure 6.24: Variation of the median (in red) along iterations for port 12 in level B1 of Tokamak Building.

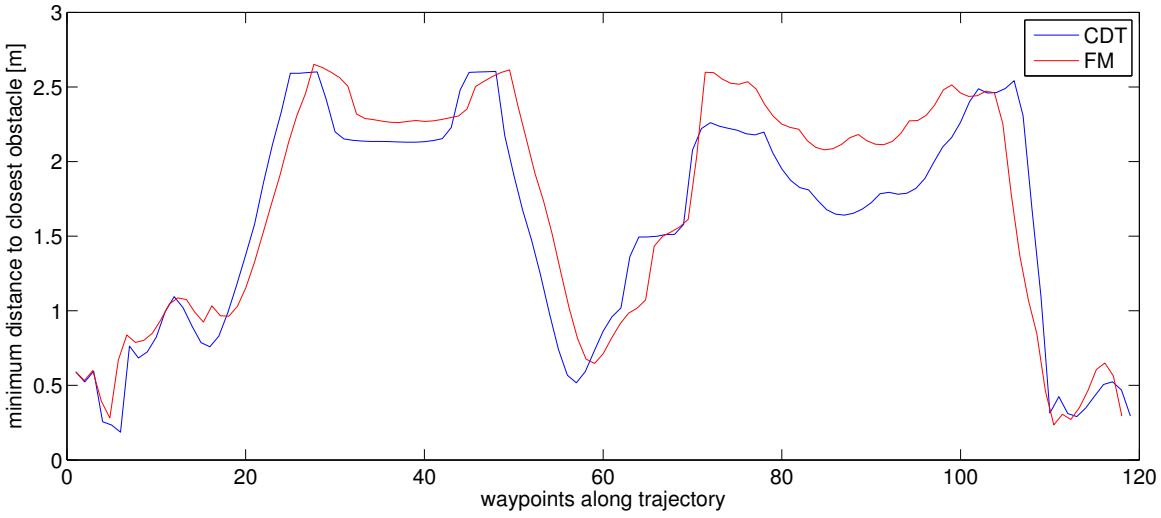


Figure 6.25: Comparison between the minimum distances along the optimized trajectories using the Constrained Delaunay Triangulation and Fast Marching Square initializations for port 12 in level B1 of Tokamak Building.

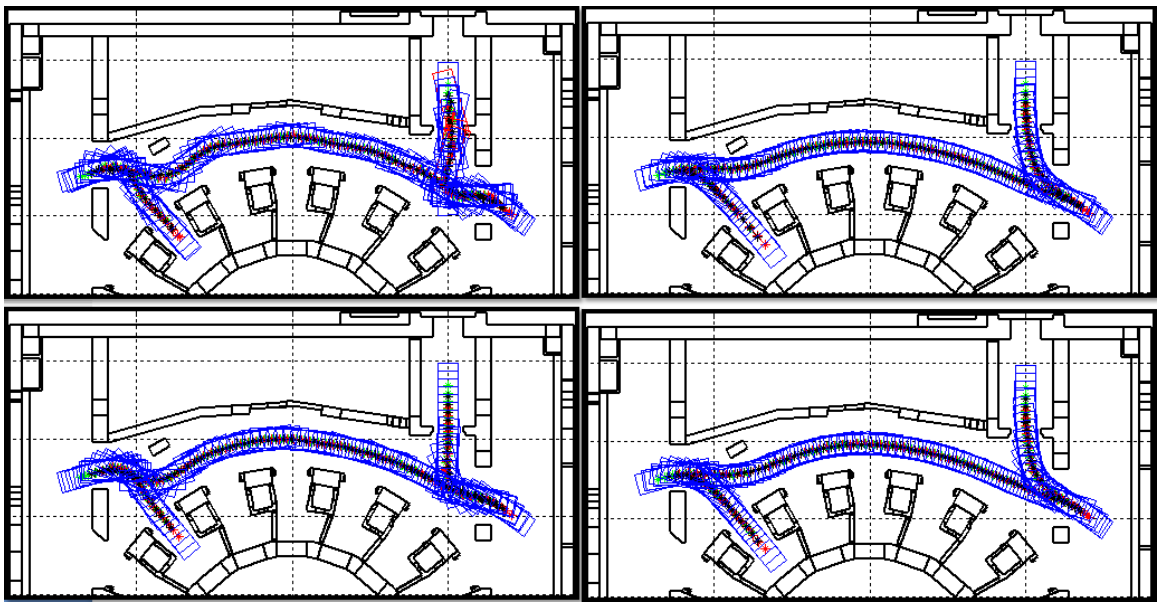


Figure 6.26: Top: the initial trajectory for port 7 using as initialization the Constrained Delaunay Triangulation (left) and the Fast Marching Square (right); bottom: the final optimized trajectory for port 7 using as initialization the Constrained Delaunay Triangulation (left) and Fast Marching Square (right).

minimum distance for the trajectory with FM^2 initialization also presents slightly higher values, as shown in Figure 6.28.

In Figure 6.29 it is presented a summary of the comparison of results between FM^2 and CDT initializations. In general the FM^2 initialization requires less computation time and number of iterations to converge, over the CDT initialization, which allows to decrease the computational effort that is needed for the optimization.

6.6 Conclusions

This Chapter presented a summary of an algorithm to optimize trajectories in terms of clearance, smoothness and execution time. The algorithm has three steps: the initialization based on the FM^2, the path optimization using rigid body dynamics and the trajectory evaluation, where a velocity profile is created attending the clearance and maximum/minimum velocities and accelerations. The inputs of the 2D path planning algorithm are: the vehicle dimensions, the map of the environment and the initial and final goals of each mission. The path initialization is fast, robust and close to the final solution. The FM^2-based initialization proposed in this Chapter also provides trajectories with no clashes whenever possible, but with better performance

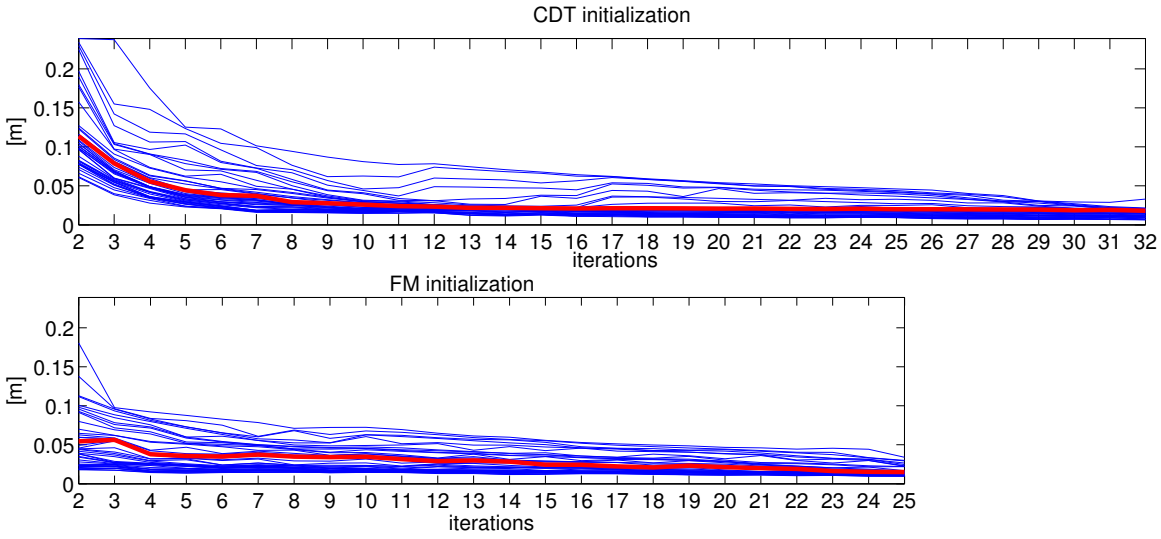


Figure 6.27: Variation of the median along iterations for port 7 in level B1 of Tokamak Building.

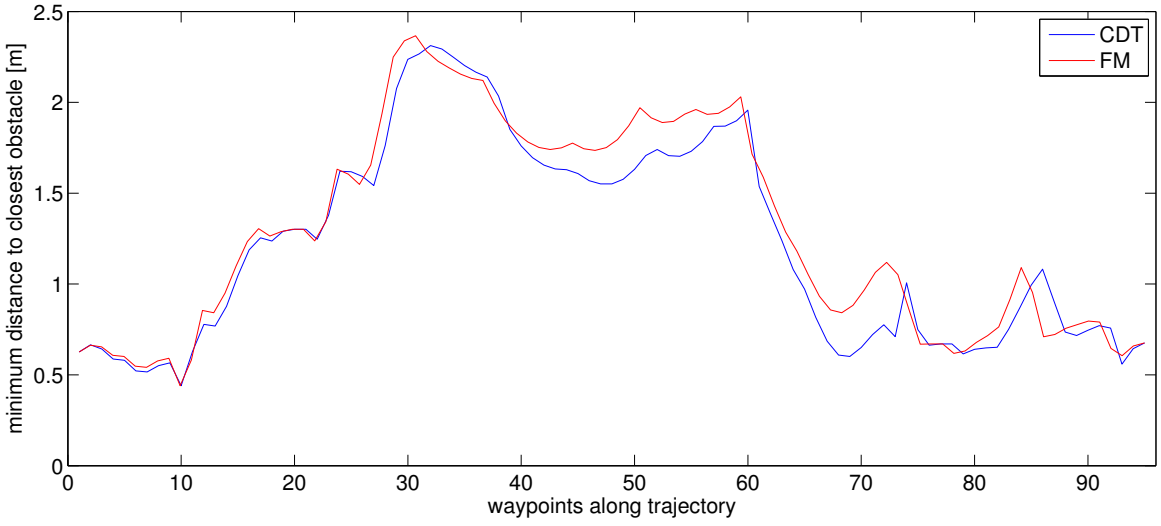


Figure 6.28: Comparison between the minimum distances along the optimized trajectories using the Constrained Delaunay Triangulation and Fast Marching Square initializations for port 7 in level B1 of the Tokamak Building.

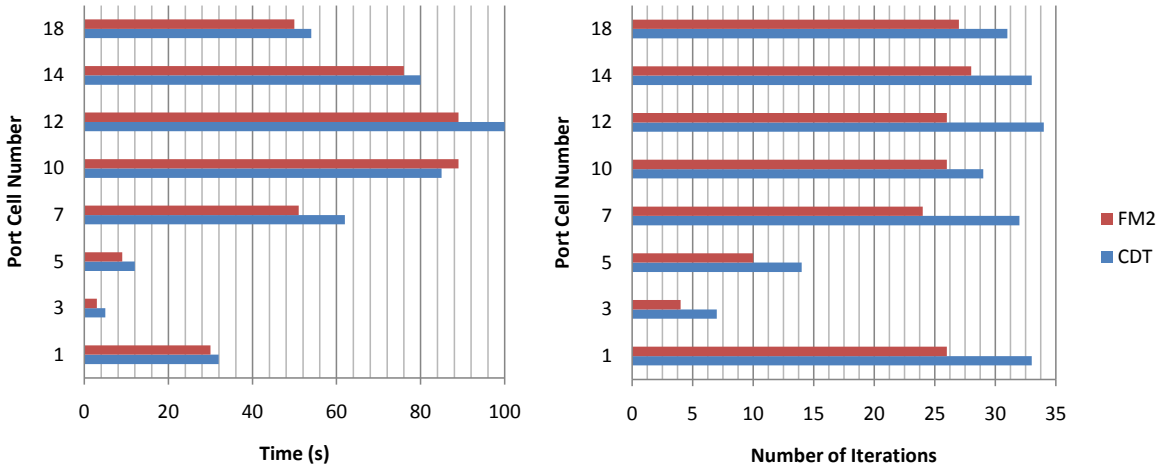


Figure 6.29: Comparisons of computational time (left) and number of iterations (right) for trajectory optimization using Constrained Delaunay Triangulation and Fast Marching Square.

in terms of computation time, smoothness, and safety than the previous CDT-based approach.

The algorithm was extensively tested using the maps of the ITER test facility: a structured, but complex and cluttered scenario. More experiments were done in the reactor building, since it is the core of the ITER. Results show how robust and flexible outputs are. The algorithm is also applicable to other environments, such as warehouses and using other vehicle kinematic configurations.

The future work focuses on extending the algorithm to a free roaming level (wheels do not follow the same path) avoiding the inclusion of maneuvers.

Further details about the ITER project and the proposed solutions for robotic navigation can be found in [123].

Chapter 7

Fast Marching-based motion learning

7.1 Introduction and motivation

The development of human-like robots has led to an increase in the number of degrees of freedom, which makes the planning and control tasks very difficult to be accomplished in real time. Nowadays, it is common to have redundant manipulators carrying out complex tasks in which conventional control over the inverse kinematics is not enough for successfully completing a given motion. The main trend to overcome this problem is the use of learning techniques. Learning algorithms try to identify and generalize the relevant features of a motion in order to be able to reproduce previously given experience, even if the environment has changed or the motion query is not the same as the one taught.

This Chapter focuses on programming by demonstration (PbD): the robot is given a set of demonstrations as an input for the learning algorithm. In PbD the demonstrations can be provided either by observing a demonstrator doing a task or by physical guiding of the robot during the task (kinesthetic teaching). While the first method requires the system to handle the re-targeting problem, the kinesthetic teaching method simplifies the problem using the same embodiment for both demonstration and reproduction. At the end, a set of motions (usually given as point-to-point trajectories) is given as input to the learning algorithm.

The motion learning objectives is commonly to execute a specific task as it was previously taught to the robot. Two different task types can be distinguished: 1) to execute a motion in which accuracy is not a critical point but the motion dynamics is, 2) to execute a motion which surely reaches a specific point of the space (the importance of dynamics depends on the problem).

The first task type focuses on teaching the robot how to complete a given task with no specific initial or final points, e.g. ball-in-a-cup game[124] or hitting a table tennis over the net[125]. In these cases, Dynamic Motion Primitives (DMP)[126, 127], a set of nonlinear differential equations which creates smooth control policies have become very popular[128]. For learning the primitives, reinforcement learning has played an important role during the last years[129]. A more recent approach based on the motion primitives idea is proposed in [130], where the primitives learning is carried out using incremental kinesthetic teaching by means of Hidden Markov Models (HMM).

The second task type consists of completing a given motion in which there exists a specific goal but the initial states can change. Concretely, it faces the problem of showing the robot how to perform a discrete motion (i.e. point-to-point trajectories). The Stable Estimator of Dynamical Systems (SEDS) approach[131] is able to generalize and reproduce the demonstrations even when spatial and temporal perturbation appear. Calinon goes a step further proposing a control strategy for a robotic manipulator operating in unstructured environments while interacting with human operators

[132]. Such situations are starting to be common in manufacturing applications. In fact, dynamical systems have shown to be a powerful alternative to model robot motions [133, 134] and different statistical approaches have been proposed: Gaussian Process Regression [135], Locally Weighted Projection Regression [136] and so on.

Other approaches have been proposed in order to leverage previous robot experience. For instance, obstacle rearrangement for faster replanning in a new environment [137] or the creation of a collision-free paths database so that paths can be reused in the future to speed up the planning process[138]. In these cases, the objective is to reduce the computational time when planning in a high-dimensional space by previously training the robot with environment-path information. The consequence is that the computed paths will be similar to those given during the training phase. Other example is trajectory prediction[139, 140] which is also able to adapt path initialisations to new environments for optimal planning.

Most of the previous approaches have shown a good performance[141], but their underlying mathematical model is usually based on probabilistic terms, causing the learning to be stochastic. Depending on the problem to solve, this stochasticity may not be a desirable property since with the same demonstrations different solutions are given each time. It could also not converge to a solution, becoming unstable under certain conditions since stochastic optimization algorithms are used to solve the learning. Besides, most of these approaches are based on learning motion control parameters. To include changes in the environment becomes challenging[142].

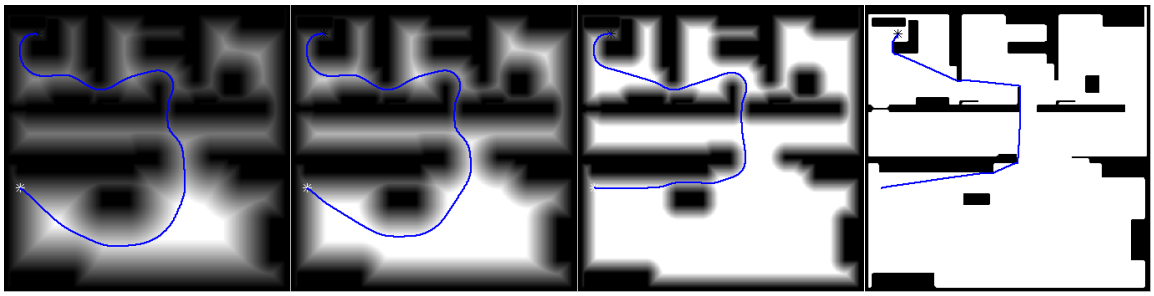
In this Chapter the Fast Marching Learning (FML) algorithm is detailed: a deterministic, asymptotically globally stable motion learning algorithm designed from a path planning point of view, using FM^2 algorithm as the underlying path planning method.

The rest of the Chapter is organized as follows. Section 7.2 details FML and some simulation results are shown. Section 7.3 includes an in-depth analysis of the proposed method, its characteristics and parameters. Experimental evaluation and comparison against SEDS method are shown in section 7.4. Finally, section 7.5 outlines the main conclusions of this Chapter and the future work in this area.

7.2 Path Planning Learning with Fast Marching Square

The FM^2 paths tend to go through those places in which the propagation velocity is higher, since it means that the total path can be covered in less time (as depicted in Figure 7.1). This fact can be exploited by *forcing* the path to take a specific direction if the map $\mathcal{F}$ is carefully modified. Therefore, the objective is to encode the experience given to the robot by an expert in the velocity map. The consequence is

that the final paths could be completely different to those given by the standard FM^2 method. However, the main characteristics of the FM^2 method will remain, such as smoothness and local-minima-free.



(a) Saturation: 0.75 (b) Saturation: 0.5 (c) Saturation: 0.25 (d) Saturation: 0

Figure 7.1: FM^2 saturated variation: modification of the path depending on the saturation value.

This Chapter is focused on point-to-point demonstrations: taught trajectories are codified as points in the workspace. Therefore, the principal objective of motion learning is to be able to successfully reproduce the taught motion when the robot is asked for a similar plan. The experience is expected to be generalized while improving the motion taught by making it faster, more efficient, smoother, etc.

7.2.1 Fast Marching Learning Method

The algorithm described next takes as input data gathered during a kinesthetic teaching process. Although it is applicable to any number of dimensions, end-effector's Cartesian coordinates will be used in order to help the reader to understand the methodology. In this case, the re-targeting problem is avoided, as the dataset (set of end-effector's positions) is recorded in Cartesian coordinates. Every taught path P will contain a set of N three-dimensional points $p(x, y, z)$ sampled with a constant cycle time T. If K paths are taught to the system, the experience E can be codified then as:

$$E = \langle P_1, P_2, ..., P_K \rangle \tag{7.1}$$

where

$$P_i = \langle p_{i,1}, p_{i,2}, ..., p_{i,N_i} \rangle \tag{7.2}$$

The environment representation is the same as for FM^2, an n-dimensional cell grid. An empty workspace is assumed. However, obstacles can be directly included in the algorithm as shown in sections 7.2.2 and 7.3.1. Hence, the steps of the Fast Marching Learning (FML) method are the following:

1. All the points contained in E are labeled as 1 (white) in an workspace with no data (represented in black, value 0). This workspace is denoted as $\mathcal{F}_p$.

2. Apply the dilation operation on $\mathcal{F}_p$ by the size *aoi*, measured in cells, which defines the *area of influence* of the taught data.

   These two steps *divide* the workspace into two different zones: those affected by the previous experience, and those not influenced (where the algorithm will behave as a regular path planner). The *aoi* parameter sets the size of these zones since it is responsible for dilating the initial demonstrations.

3. The FMM is applied as done in the first step of the FM^2 method, so that $\mathcal{F}_p$ is converted into a velocity map. All the zero-valued points of $\mathcal{F}_p$ are used as wave sources.

4. By linearly rescaling $\mathcal{F}_p$ in order to be within the bounds defined by $[sat, 1]$ the final velocity map $\mathcal{F}$ is obtained. This second parameter *sat* is a saturation which weights the importance of the new data against the rest of the environment.

   At this point, $\mathcal{F}_p$ contains a generalization of the demonstrations. Those areas with higher value (lighter) represent, in an intuitive manner, the center of the demonstrations.

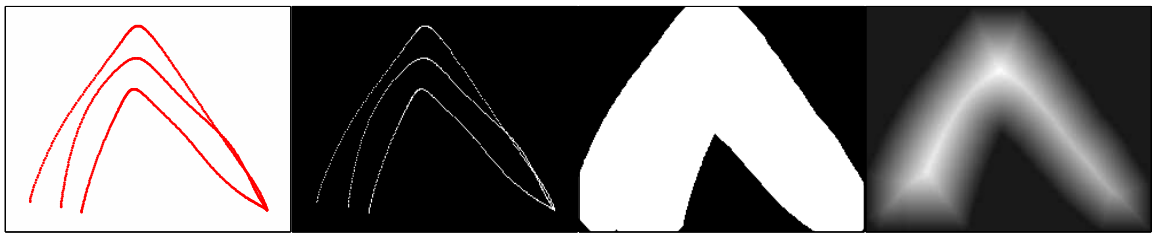
5. Apply FMM over the entire workspace using as unique wave source point the centroid of the final point of all the trajectories $P_i \in E$, and considering the velocity map $\mathcal{F}$.

This algorithm is formalized in algorithm 14. DILATE(map, s) applies the morphological dilation operation on *map* with a structuring element with size s, given in cells. Any shape of this structuring element is valid. For the examples of this Chapter the *ball* shape has been used. FastMarching$(\mathcal{F}, x_S)$ applies the FMM using the velocity map $\mathcal{F}$ and the point s as wave source (s can be an array with more than one wave source). RESCALE(map, min, max)) linearly rescales *map* between *min* and *max* values. Finally, CENTROID(E) takes as input a set of trajectories and output the centroid of the first point of all the demonstrations. Lines 1-9 create the velocity map according to the demonstrations, as shown in Figure 7.2. This way, the second FMM wave (lines 10-12) can travel through areas with no experience but with less priority to those affected by DILATE(). In case of new path queries, paths will be *attracted* towards areas with experience since the second FM^2 wave will expand faster as depicted in Figure 7.3.

Algorithm 14 The Fast Marching Learning algorithm

Input: Experience $E = \langle P_1, P_2, ..., P_K \rangle$.
Output: Modified velocity map $\mathcal{F}$, reproduction field $\mathcal{T}$.

$\mathcal{F}_p \leftarrow \{0\}$;
for $i = 1$ to K **do**
  for $j = 1$ to N_i **do**
    $x \leftarrow P_{i,j}$;
    $\mathcal{F}_p(x) := 1$;
$\mathcal{F}_p \leftarrow \textsc{Dilate}(\mathcal{F}_p, aoi)$;
$x_S \leftarrow \{\forall x \in \mathcal{F}_p | \mathcal{F}_p(x) = 0\}$;
$\mathcal{F}_p \leftarrow \textsc{FastMarching}(\mathcal{F}_p, x_S)$;
$\mathcal{F} \leftarrow \textsc{Rescale}(\mathcal{F}_p, sat, 1)$;
$x_S \leftarrow \textsc{Centroid}(E)$;
$\mathcal{T} \leftarrow \textsc{FastMarching}(\mathcal{F}, x_S)$;
return $\mathcal{F}, \mathcal{T}$;



(a) Demonstrations (b) Initial $\mathcal{F}_p$ (c) Dilated $\mathcal{F}_p$ (d) $\mathcal{F}$

Figure 7.2: Fast Marching Learning steps. $sat = 0.1$ and $aoi = 25$ cells.

7.2.2 Including Obstacles in the Workspace

In case that the workspace is not completely free, obstacles are required to be included. Obstacles can either be part of the workspace from the beginning or appear once after demonstrations were done. Thanks to the FM^2 underlying method, both methods can be easily addressed.

Let us assume that the robot has been given an experience E and a velocity map $\mathcal{F}$ has been already computed. Let also assume that the initial workspace $\mathcal{X}$ is not obstacle-free, or that it was free in the beginning but new obstacles appear. In this case, the updated velocity map of the workspace $\mathcal{F}_{up}$ has to be computed by saturating at level sat. Finally, in order to compute the final velocity map $\mathcal{F}$, the following update step is necessary for all those points in which $\mathcal{F}_{up}$ is not saturated:

$$\mathcal{F} := min(\mathcal{F}_{up}, \mathcal{F}) \quad \forall i \in \mathcal{F}_{up} | \mathcal{F}_{up}(i) < sat \tag{7.3}$$

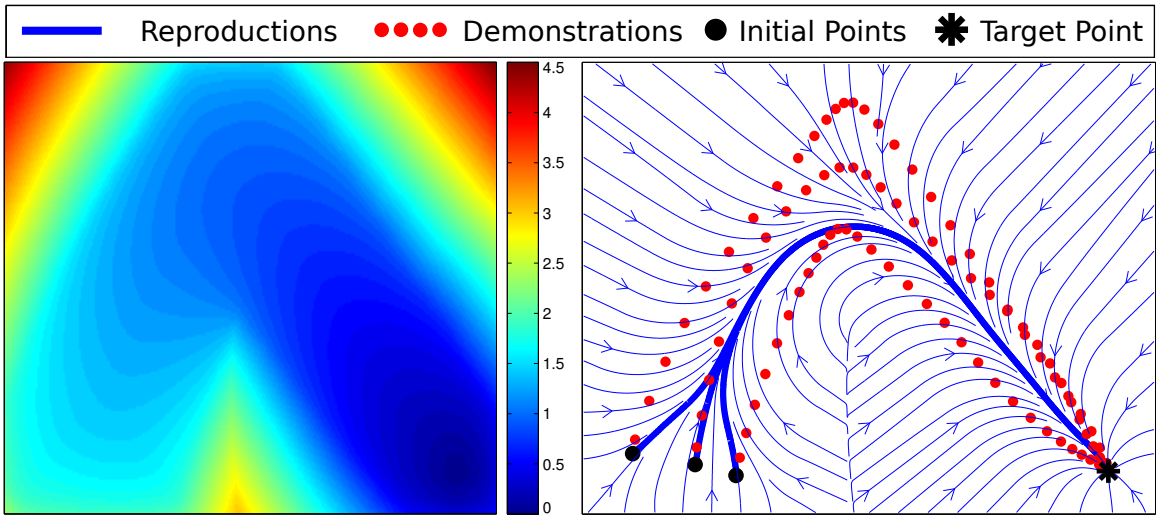
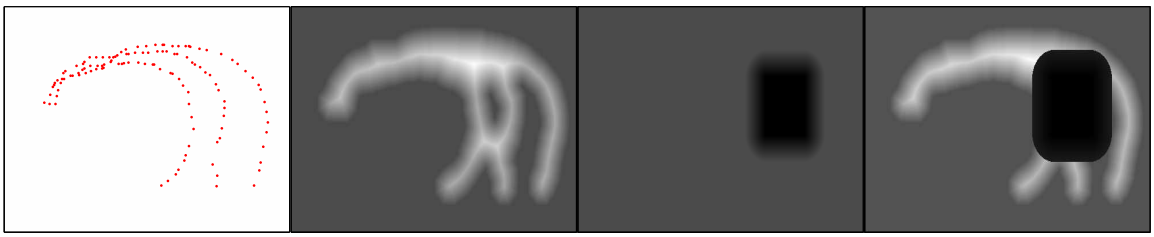


Figure 7.3: Left: FM^2 time-of-arrival map $\mathcal{T}$ using the modified $\mathcal{F}$. Right: streamlines (set of possible reproductions) of $\mathcal{T}$ with parameters $sat = 0.1$ and $aoi = 25$ cells.

This operation has to be repeated any time a new obstacle appears. The $\mathcal{T}$ needs also to be updated by propagating an FMM wave from the target point. It will take into account the demonstrations given to the robot as much as possible while avoiding the new obstacles. However, it would be worthy to recompute $\mathcal{F}$ from scratch with different, more restrictive parameters *sat* and *aoi* because the new obstacles will deviate the reproductions and the behaviour of the solution could change. The algorithm is detailed in Figure 7.4 and its results are shown in Figure 7.5.

7.3 Analysis of the Fast Marching Learning Method

In this Section an in-depth analysis of the FML and its characteristics is carried out.



(a) Demonstrations (b) $\mathcal{F}$ (c) $\mathcal{F}_{up}$ (d) Updated $\mathcal{F}$

Figure 7.4: Fast Marching Learning obstacles update steps. $sat = 0.1$ and $aoi = 25$ cells.

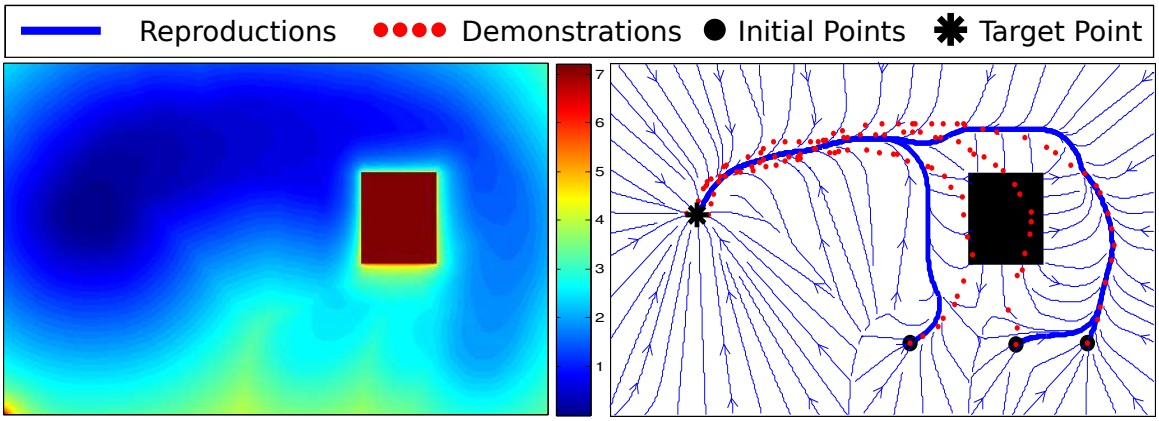


Figure 7.5: Left: map of times using the propagation velocities learned. Right: result of the learning method with parameters $sat = 0.1$ and $aoi = 25$ cells.

7.3.1 FML Main Characteristics

Duality

Let us suppose that the taught paths have a starting point close to a region A of the workspace and the final points are close to any region B of the same workspace. Usually, motion learning algorithms are designed supposing that the new queries (reproductions of the robot) will be in the same direction $A \rightarrow B$. In the case of a query which asks for a plan in the opposite direction, $B \rightarrow A$, the behaviour of other learning algorithms could be unexpected. However, in the case of FML the same motion as taught will be performed but in the opposite direction. Although this property could become a limitation under some circumstances, it allows to predict the behaviour of the robot which is a very desirable property regarding safety.

Determinism

The FMM is a deterministic method. This means that the output will remain always the same if the input does not change. Since FML is based on FMM, and the way $\mathcal{F}$ is computed is deterministic, FML is also deterministic. This is an important factor since the behaviour is easy to be predicted and no spurious behaviours will occur, which is common in probabilistic, optimization-based learning methods.

Behaviour with no experience

In other learning algorithms, if a motion query is done from a point far away from the given experience, the behaviour is often unpredictable. In those cases, the FML

method will provide the fastest path from the starting point to the goal point according to the metrics given by the velocity map $\mathcal{F}$. Since it has a constant value of *sat* in those places away from the experience, the fastest path also means the shortest path to the goal point in terms of distance. This is shown in Figure 7.6.

One-shot Learning

One-shot learning refers to the characteristic of a learning system to be able to successfully reproduce and generalize the experience when only one demonstration has been given [143]. This is a desirable characteristic from the point of view of the final user of the system.

When many demonstrations are given to the FML method, the dilation step of the algorithm brings all these demonstrations together into one area of influence of the learning (depending on the *aoi* parameter). Therefore, many demonstrations act as just one demonstration with a larger *aoi*. If the robot is going to be taught with only one demonstration, then the *aoi* parameter has to be set with a larger value than when many demonstrations are provided.

Figure 7.7 shows an example of FML one-shot learning in an environment with one obstacle. Notice that in both cases, it is possible that the generated reproduction fields are very similar.

7.3.2 Stability Analysis

Although the FML method is not based on dynamical systems, its stability is analyzed with an analogy to the the Lyapunov Stability theorem [144].This theorem expresses that a function $\dot{x} = f(x)$ is asymptotically stable at the point x_g if a continuous

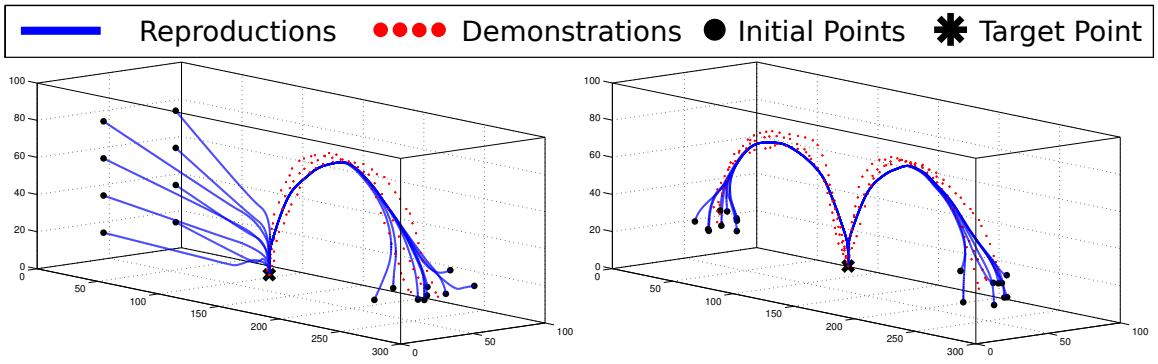


Figure 7.6: Behaviour of the Fast Marching Learning algorithm in zones with no experience, and its adaptation when new experience is included. $sat = 0.1$ and $aoi = 20$ cells.

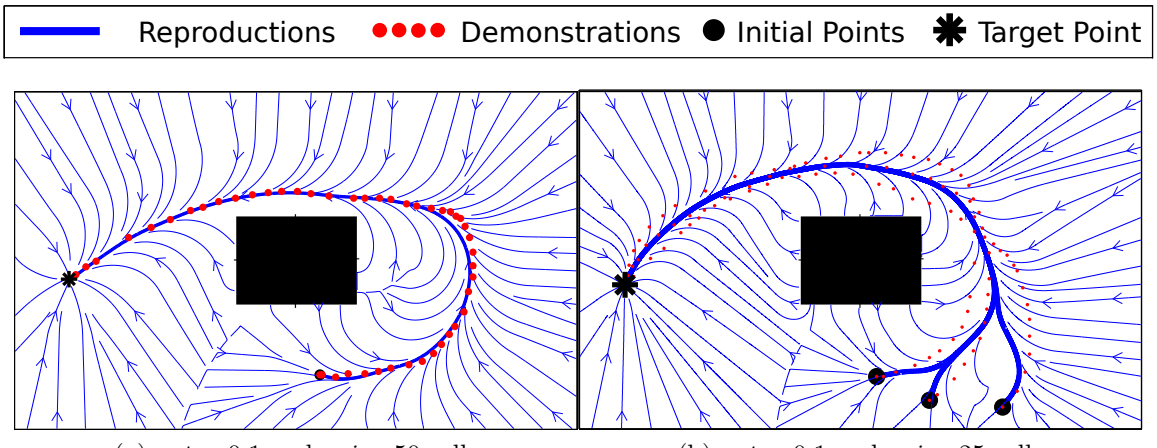
and continuously differentiable Lyapunov function $V(x)$ can be found such that it is always positive, its derivative is always negative and $V(x_g) = \dot{V}(x_g) = 0$.

Let us consider as Lyapunov function the one generated when expanding the second wave of FM^2, $T(x)$. This function starts at the goal point of the robot x_g, where the $T(x_g)$ value is 0. Given the fact that this wave expands always with non-negative velocities, the value of $T(x)$ will be higher (positive) as the wave gets farther from x_g. Finally, the derivative of the function is always the same sign since $T(x)$ is free of local minima. Actually, the derivative of equation $\mathcal{T}$ is always positive at any point, as the time is always increasing from the wave source point. However, when running gradient descent from a given point, the path generated will follow the direction in which the time decreases the most. Therefore, in this case the Lyapunov conditions are satisfied to ensure a globally asymptotically stable system. In other words, all the motion reproductions of the FML will converge to the same point as the $T(x)$ function has a unique minimum.

Conceptually, FML is really close to the work presented in [145], where vector fields are created using different Lyapunov function candidates. Qualitatively, their results are close to the SEDS algorithm [131], but prone to have local attractors and unexpected behaviours. However, FML guarantees a globally stable system by numerically computing the Lyapunov function with FMM. This Lyapunov function is modified by parameters *sat* and *aoi*.

7.3.3 Parameters Analysis

The proposed FML method counts with two parameters whose configuration changes the behaviour of the learning procedure. In this Section an intuitive explanation of



(a) $sat = 0.1$ and $aoi = 50$ cells (b) $sat = 0.1$ and $aoi = 25$ cells

Figure 7.7: One-shot against many demonstrations. Workspace: $300x500$ cells.

their influence is given. Also, Figure 7.8 includes the learning results when the same demonstrations but different set of parameters are given.

Saturation, *sat*

The possible values of the saturation are $sat \in (0, 1]$. This is the value of $\mathcal{F}$ in those places where no experience is given, which represents the propagation velocity of the FMM wave. According to the design of the FML algorithm, the places with experience have a higher propagation velocity. Therefore, the places with experience will be reached earlier by the wave.

The reproductions will only ignore the experience when it takes less time for the wave to reach the target points by propagating through the zones with no experience. Hence, the saturation parameter acts as a weighting factor between the importance of the experience given to the robot and the rest of the space. Since the objective is for the robot to reproduce the motions that have been taught, this factor is usually close to zero. When an excessively high value is given to this parameter, the reproductions will ignore the experience (top row of Figure 7.8). On the other hand, a extremely low value will result in a very greedy learning, where reproductions will always go to those zones with experience (bottom row of Figure 7.8). Along this Chapter, good results are commonly given when $sat \in [0.05, 0.1]$.

Area of influence, *aoi*

This is the size used in the dilation step of the FML, where all the recorded points are enlarged in order to give connectivity to the demonstrations and its spatial surroundings. It is measured in cells and directly depends on the size of the workspace. Experimentally, it has been found that a good value for this parameter is around 5% for multiple demonstrations and 10% for one-shot learning (percentage given over the smallest dimension of the workspace).

This parameter affects the generalization of the learning. When an excessively low value is given, the algorithm will not generalize and it will follow the taught trajectories strictly (left column of Figure 7.8). In the opposite case, the reproductions will generalize too much and the shape of the demonstrations will be lost (right columns of Figure 7.8).

Special attention is required by the *aoi* parameter when working in dimensions which are not in the same domain. This Chapter is focused on working in end-effector's Cartesian coordinates, so the three dimensions are in the same spatial domain. However, when using other dimensions, i. e. velocities or angles, the size of this parameter has to be coherent with the desired result.

Figure 7.9 shows examples of learning results with wrong parameters. In Figure 7.9 a) reproduction queries are demanded from points which are close to the target point.

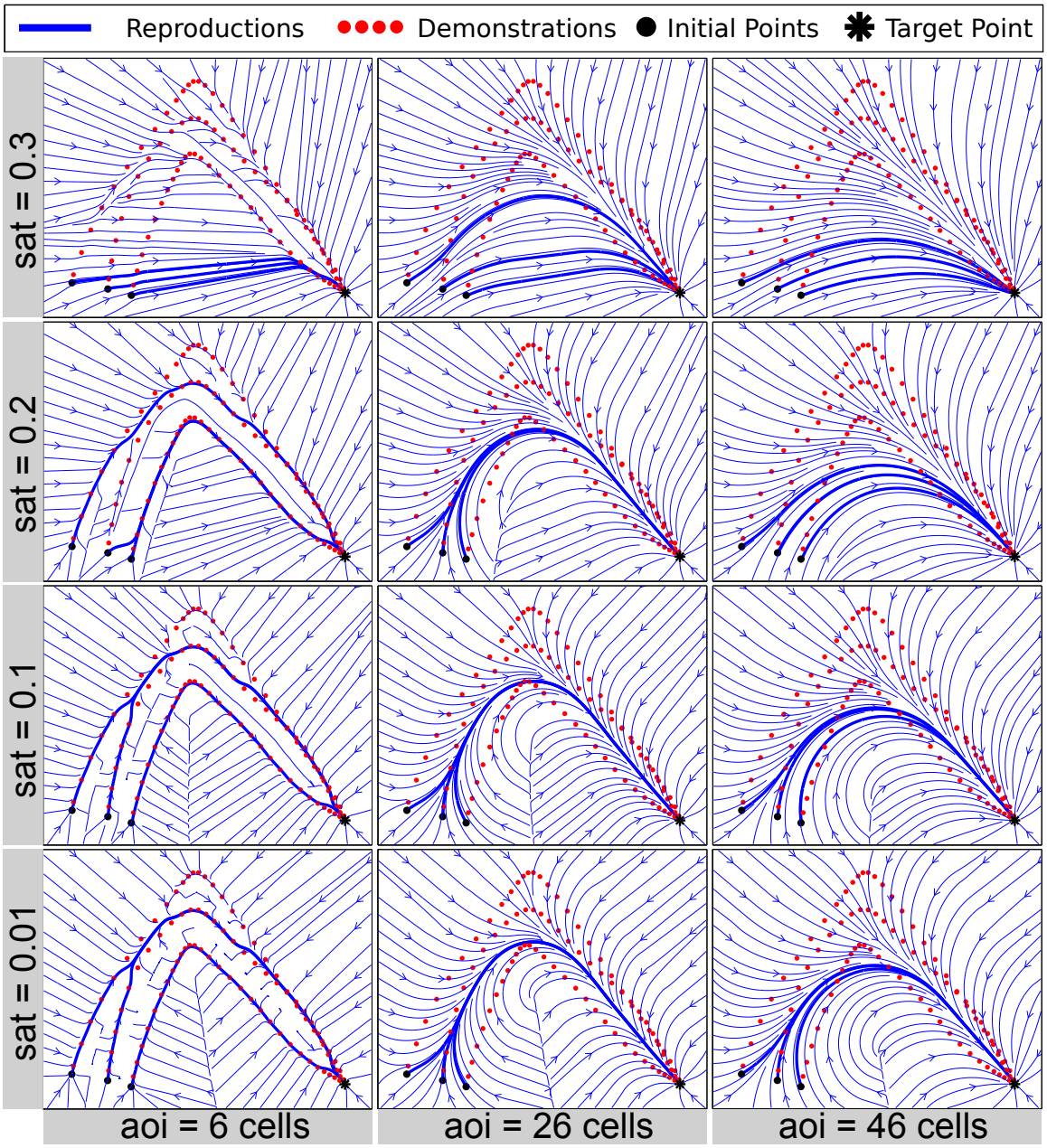
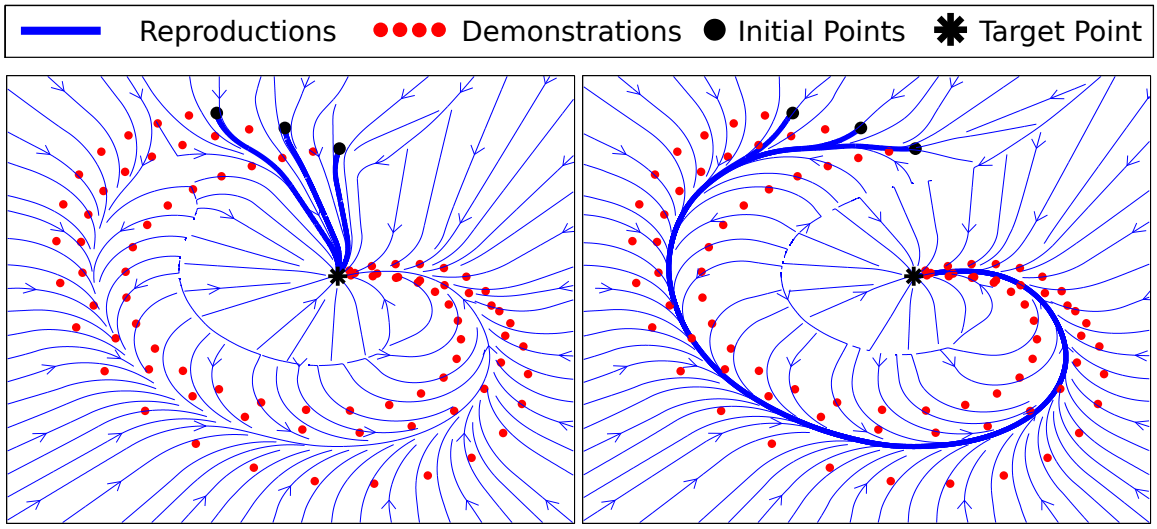


Figure 7.8: Analysis of the results using different parameter settings. Workspace: $250x185$ cells.

In this case the weight of the experience has to be very high in relation to the rest of the space, so that by decreasing the value of *sat* the learning becomes successful (Figure 7.9 b)). A different example is given in Figure 7.9 c). In this case the



(a) $sat = 0.1, aoi = 25$ cells (b) $sat = 0.05, aoi = 25$ cells

(c) $sat = 0.1, aoi = 25$ cells (d) $sat = 0.1, aoi = 10$ cells

Figure 7.9: The shape of the trajectories to learn could influence the parameters of the algorithm. Workspace: $250x185$ cells.

excessively high value of *aoi* converts the N-shaped path into a much smoother shape. By decreasing the size of the *aoi* it is possible to keep the shape of the demonstrations, as shown in Figure 7.9 d).

Table 7.1: Results of SEDS and FML in the handwriting motions dataset.

Times (s)			
FML		SEDS	
μ	σ	μ	σ
0.17	0.12	23.28	15.58

7.4 Experimental Evaluations

This Section includes several demonstrations of the FML performance using real and simulated data. Since the proposed method does not include dynamics, it is not possible to carry out an exhaustive comparison against those learning methods based on dynamical systems. However, a brief comparison against the SEDS method [131] is included. It has been carried out over the 2D handwriting motions described in the SEDS paper: 24 different handwriting motions collected with a Tablet-PC. Each motion is composed by three demonstrations with a uniform sampling time $T = 0.02s$.

Figure 7.10 shows the results of FML over 9 of the motions in the dataset. The last row of the Figure is worthy to mention since it is composed by multi-model motions: the motion to learn changes completely depending on the area of the workspace. FML is able to generalize the demonstrations.

Examples in a three-dimensional space are shown in Figure 7.11. The demonstrations given in this case are simulated, manually introduced.

The FML-SEDS comparison, shown in Figure 7.12, focuses on the reproduction field of these methods. Both are running in two dimensions, in a workspace of $250x185$ cells. The parameter set in SEDS is: 4 Gaussian distributions and a likelihood optimization method with a maximum of 500 iterations. In FML parameters are, $sat = 0.1$ and $aoi = 25$ cells. These results show that the performance of both algorithms is quite different. While SEDS looks for a complete generalization of the motion, following the same motion pattern from any point of the space, FML converges to the area with experience in a smooth way, creating motions always similar to the reproductions. However, the behaviour of SEDS in some areas is unexpected and may cause problems when operating in a real robot. This is likely to occur in those places close to the target point but in the opposite direction of the demonstrations or in places far away from the target. Table 7.1 includes the average and standard deviation of the computation times as an interesting result. However, this is very implementation-dependent.

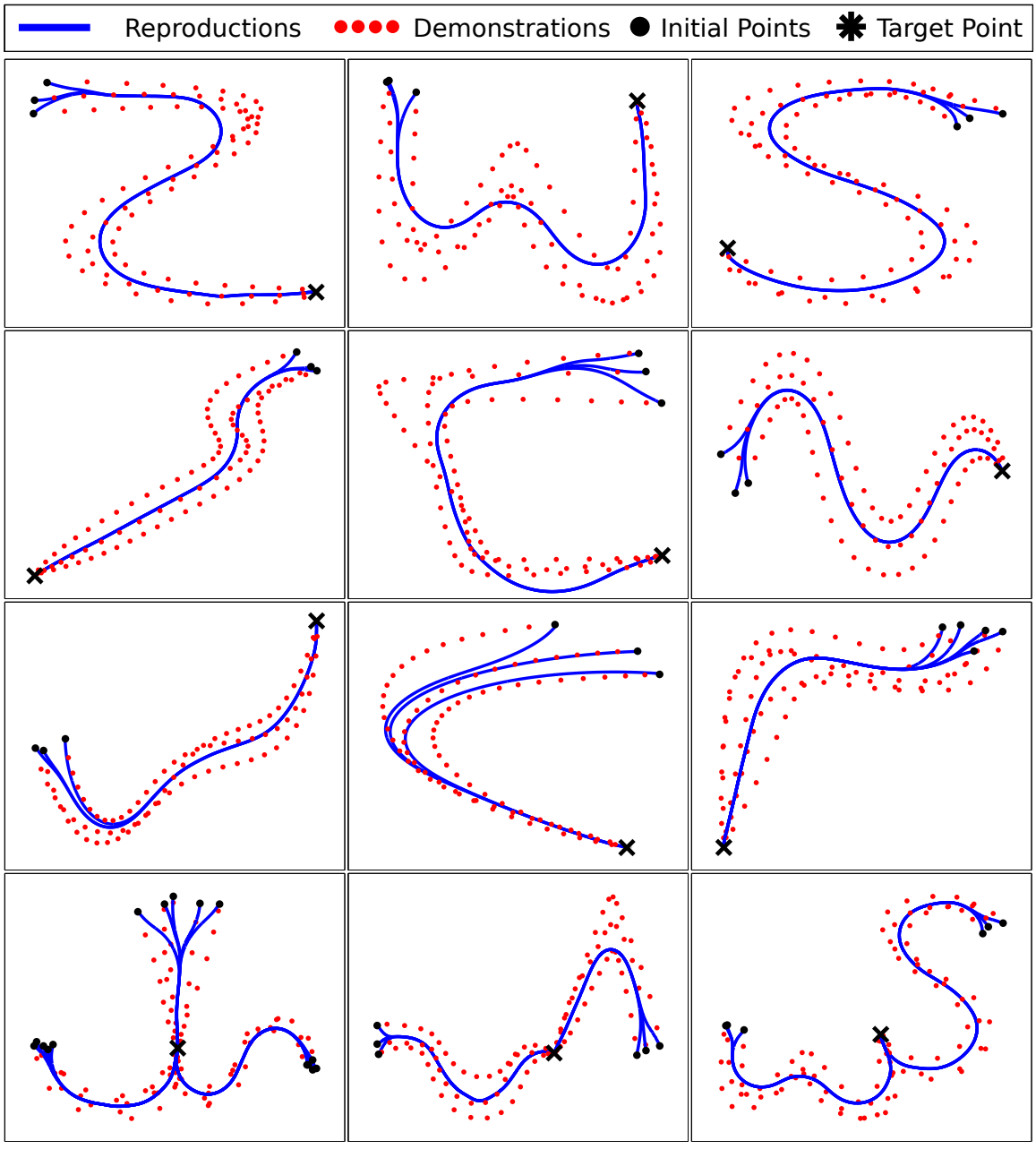


Figure 7.10: Results of the Fast Marching Learning algorithm applied to handwriting motions. $sat = 0.1$ and $aoi = 25$ cells.

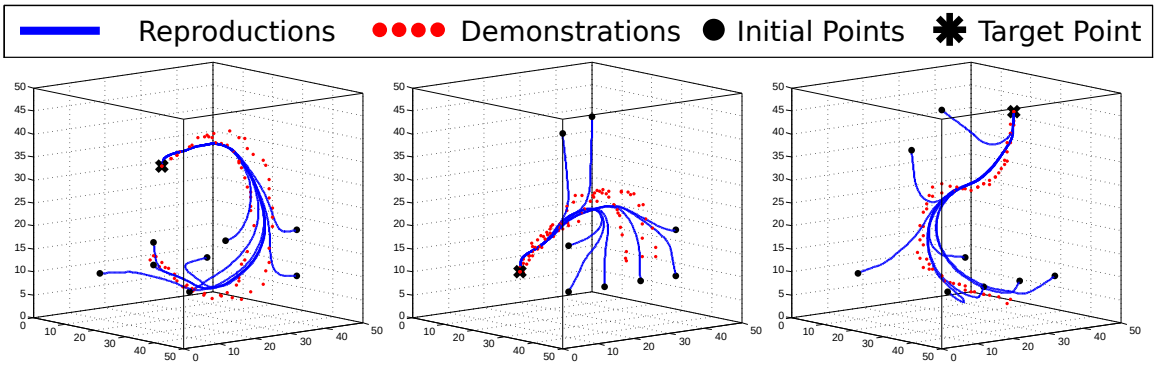


Figure 7.11: Results of the Fast Marching Learning algorithm in three dimensions, $sat = 0.05$ and $aoi = 10$ cells.

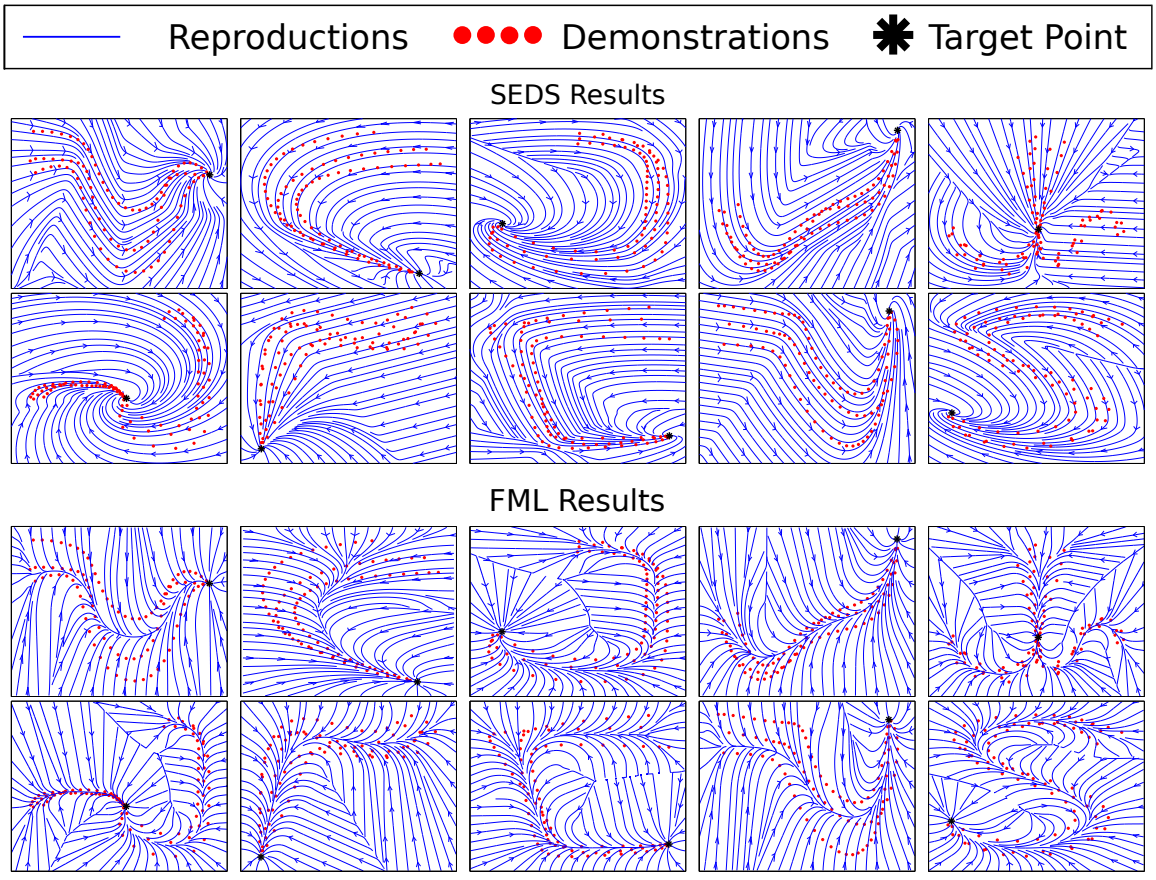


Figure 7.12: Qualitative comparison of the learning results of algorithms SEDS and FML in the handwriting motions dataset.

7.5 Conclusions

Along this Chapter FML, a novel learning algorithm for robot motions, has been detailed. It has been shown that it can work well with an empty workspace and also with obstacles. It is not based on an optimization procedure but on a well-studied path planning algorithm, FM^2. This means a completely different point of view of the previous work in motion learning.

A deep analysis on the performance of FML and the dependance on its parameters has been carried out. The main advantages of this method against others are: determinism, asymptotically globally stable, one-shot learning method and low computational time. Besides, experimental results show that FML is reliable and safe. No major problems have been identified.

A brief comparison with a state of the art method has been included. Other metrics could be employed, such as storage size, adaptation to online changes, accuracy reproducing demonstrations, etc. However, the nature of FML regarding the current learning algorithms is very different. Therefore a deeper comparison would not be meaningful. FML does not pretend to improve SEDS, DMP or other methods, but to propose an alternative in which simpler, robust solutions are required. For example, writing motions, door openings, boosting planning from experience (to be addressed in future work), and other problems in which the dynamics are not critical but the final trajectory is.

The main current drawback is that motion dynamics are not codified into the learning. Also, as FML is based on grid cell, the application to more than 3 dimensions can be very expensive in terms of computational time.

Therefore, future work will focus on solving both of these problems. The anisotropic Fast Marching Method, together with parallel implementations are promising research lines which can improve the proposed method significantly. It is also worthy to explore optimization methods in order to automatically set the parameters involved. Finally, to study the influence of the grid resolution in the reproduction field would be very valuable, as it would be possible to optimize the computation-time/reproduction-quality ratio.

Chapter 8

Bidirectional Fast Marching Trees: motion planning in high-dimensional spaces

DISCLAIMER: The BFMT* algorithm [146] was developed jointly between the author of this Thesis and members of the Stanford Autonomous Systems Laboratory (ASL). Specifically, the implementation code and design of the algorithm resampling strategy upon failure (presented as Algorithm 18) represent the author's collective contribution. The remaining algorithm design, theoretical work, and simulation results were performed by members of ASL.

8.1 Introduction

Arguably, *sampling-based algorithms* are among the most pervasive, widespread planners available in robotics, including the Probabilistic Roadmap algorithm (PRM)[25], the Expansive Space Trees algorithm (EST) [147, 148], and the Rapidly-Exploring Random Tree algorithm (RRT) [24]. Since their development, efforts to improve the "quality" of paths led to asymptotically-optimal (AO) variants of RRT and PRM, named RRT* and PRM*, respectively, whereby the cost of the returned solution converges almost surely to the optimum as the number of samples approaches infinity [26, 149]. Many other planners followed, including BIT* [27] and RRT# [28] to name a few. Recently, a conceptually different asymptotically-optimal, sampling-based motion planning algorithm, called the Fast Marching Tree (FMT*) algorithm, has been presented in [150, 12]. Numerical experiments suggested that FMT* converges to an optimal solution faster than PRM* or RRT*, especially in high-dimensional configuration spaces and in scenarios where collision-checking is expensive.

It is a well-known fact that *bi-directional* search can dramatically increase the convergence rate of planning algorithms, prompting some authors [151] to advocate its use for accelerating essentially any motion planning query. This was first rigorously studied in [152] and later investigated, for example, in [153, 154]. Collectively, the algorithms presented in [152, 153, 154, 151] belong to the family of *non*-sampling-based approaches and are more or less closely related to a bi-directional implementation of the Dijkstra Method. More recently, and not surprisingly in light of these performance gains, bi-directional search has been merged with the sampling-based approach, with RRT-Connect and SBL representing the most notable examples [155, 156].

Though such bi-directional versions of RRT and PRM are probabilistically complete, they do not enjoy optimality guarantees. The next logical step in the quest for fast planning algorithms is the design of *bi-directional*, sampling-based, asymptotically-optimal algorithms. To the best of author's knowledge, the only available results in

this context are [157] and the unpublished work [158], both of which discuss bi-directional implementations of RRT*. Neither work, however, provides a rigorous proof of asymptotic optimality starting from first principles.

This Chapter introduces the Bi-directional Fast Marching Tree (BFMT*) algorithm which, to the best of the author's knowledge, is the first asymptotically-optimal, bi-directional, tree-based, sampling-based planner. BFMT* extends FMT* to bi-directional search and essentially performs a "lazy," bi-directional dynamic programming recursion over a set of probabilistically-drawn samples in the free configuration space.

Firstly, the BFMT* algorithm is presented in Section 8.2. Then numerical experiments are performed in Section 8.3 across a number of planning spaces that suggest BFMT* converges to an *optimal* solution at least as fast as FMT*, PRM*, and RRT*, and sometimes significantly faster. Finally, conclusions for BFMT* are pointed out in Section 8.4. Mathematical proofs are omitted as this Thesis focuses on practical applications. Interested readers are referred to [12] for mathematical proofs.

8.2 The BFMT* Algorithm

In this Section, the Bi-Directional Fast Marching Tree algorithm, BFMT*, is represented in pseudocode as Algorithm 15. To begin, a high-level description of FMT* is provided in Section 8.2.1, on which BFMT* is based. Following in Section 8.2.2 with BFMT*'s own high-level description, and then additional details are discussed in Section 8.2.3.

As these algorithms rely on graph theory, the nomenclature of this Chapter is not consistent with the rest of this Thesis. This way, the nomenclature used along this Chapter is consistent with the rest of literature on FMT* and asymptotically-optimal, sampling-based methods (with the exception of Chapter 2, as this Chapter relies on the same problem formulation).

8.2.1 FMT* – High-level description

The FMT* algorithm, introduced in [150, 12], is a unidirectional algorithm that essentially performs a forward dynamic programming recursion over a set of sampled points and correspondingly generates a *tree of paths* that grow steadily outward in cost-to-come space. The recursion performed by FMT* is characterized by three key features: (1) It is tailored to disk-connected graphs, where two samples are considered *neighbors* (hence connectable) if their distance is below a given bound, referred to as the *connection radius*; (2) It performs graph construction and graph search *concurrently*; and (3) For the evaluation of the immediate cost in the dynamic programming recursion, one "lazily" ignores the presence of obstacles, and whenever

a locally-optimal (assuming no obstacles) connection to a new sample intersects an obstacle, that sample is simply skipped and left for later (as opposed to looking for other locally-optimal connections in the neighborhood).

In order for two points to be considered neighbors, FMT* establishes a lower-bound in the connection radius to use. Two points are considered neighbors if their Euclidean distance (or more generally, the cost) is smaller than:

$$r_n = 2(1+\eta)^{1/d} \cdot \left(\frac{1}{d}\right)^{1/d} \left(\frac{\mu(\mathcal{X}_{\text{free}})}{\zeta_d}\right)^{1/d} \left(\frac{\log(n)}{n}\right)^{1/d} \tag{8.1}$$

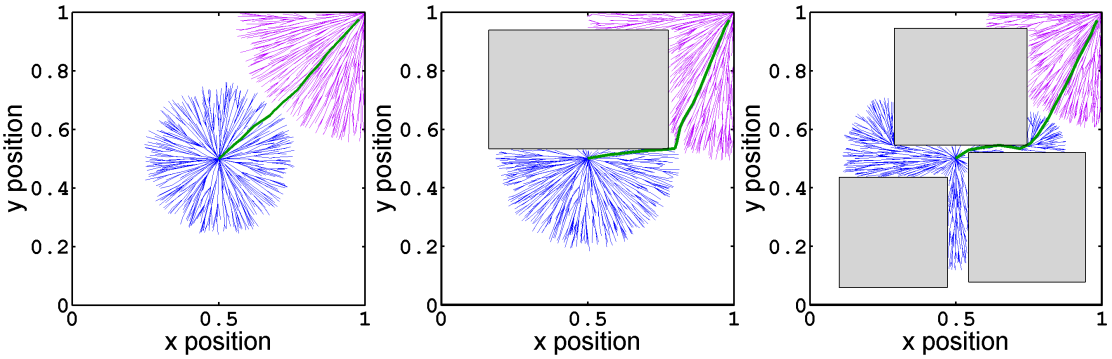
where n is the number of samples used, $\eta \geq 0$ is a tuning parameter, d is the number of dimensions, $\mu(\mathcal{X}_{\text{free}})$ is the Lebesgue measure of the free space (its hypervolume), a ζ_d is the Lebesgue measure of a ball of unit radius. This r_n, derived from the Poisson point processes theory, provides guarantees regarding the connectivity of the points randomly sampled in the space. Increasing η causes a point to have more neighbors, which will lead to more optimal paths, but slower computation times as the nearest neighbor search has to explore more space. Therefore, this parameter is experimentally tuned. In [12] a in-depth analysis of the influence of this parameter is included.

The last feature, which makes the algorithm "lazy," may cause *suboptimal* connections. A central property of FMT* is that the cases where a suboptimal connection is made become vanishingly rare as the number of samples goes to infinity, which helps maintain the algorithm's asymptotical optimality. This manifests itself into a key computational advantage—by restricting collision detection to only locally-optimal connections, FMT* (as opposed to, *e.g.*, PRM* [26]) avoids a large number of costly collision-check computations, at the price of a vanishingly small "degree" of suboptimality. Interested readers are referred to [150, 12] for a detailed description of the algorithm and its characteristics.

8.2.2 BFMT* – High-level description

At its core, BFMT* implements a *bi-directional* version of the FMT* algorithm by simultaneously propagating two wavefronts (henceforth, the leaves of an expanding tree will be referred to as the wavefront of the tree) through the free configuration space. BFMT*, therefore, performs a *two-source* dynamic programming recursion over a set of sampled points, and correspondingly generates a *pair* of search trees: one in cost-to-come space from the initial configuration and another in cost-to-go space from the goal configuration (see Figure 8.1). Throughout the remainder of the Chapter, these will be referred correspondingly as the *forward tree* and *backward tree*.

The dynamic programming recursion performed by BFMT* is characterized by the same lazy feature of FMT* (see Section 8.2.1). However, the time it takes to run



(a) 0% Obstacle coverage (b) 25% Obstacle coverage (c) 50% Obstacle coverage

Figure 8.1: The BFMT* algorithm generates a *pair* of search trees: one in cost-to-come space from the initial configuration (blue) and another in cost-to-go space from the goal configuration (purple). The path found by the algorithm is in green color.

BFMT* on a given number of samples can be substantially smaller than for FMT*. Indeed, for uncluttered configuration spaces, the search trees grow hyperspherically, and hence BFMT* only has to expand about half as far (in both trees) as FMT* in order to return a solution. This is made clear in Figure 8.1(a), in which FMT* would have to expand the forward tree twice as far to find a solution. Since runtime scales approximately with edge number, which scales as the linear distance covered by the tree raised to the dimension of the state space, it is expected in loosely cluttered configuration spaces an approximate speed-up of a factor 2^{d-1} over FMT* in d-dimensional space (the -1 in the exponent is because BFMT* has to expand 2 trees, so it loses one factor of 2 advantage). Note that the connection radius r_n remains the same as in FMT*.

8.2.3 BFMT* – Detailed description

To understand the BFMT* algorithm, some background notation must first be introduced. Let $\mathcal{S}$ be a set of points sampled independently and identically from the uniform distribution on $\mathcal{X}_{\text{free}}$, to which $\mathbf{x}_{\text{init}}$ and $\mathbf{x}_{\text{goal}}$ are added. Let tree $\mathcal{T}$ be the quadruple $(\mathcal{V}, \mathcal{E}, \mathcal{V}_{\text{unvisited}}, \mathcal{V}_{\text{open}})$, where $\mathcal{V}$ is the set of tree nodes, $\mathcal{E}$ is the set of tree edges, and $\mathcal{V}_{\text{unvisited}}$ and $\mathcal{V}_{\text{open}}$ are mutually exclusive sets containing the *unvisited* samples in $\mathcal{S}$ and the *wavefront* nodes in $\mathcal{V}$, correspondingly. To be precise, the unvisited set $\mathcal{V}_{\text{unvisited}}$ stores all samples in the sample set $\mathcal{S}$ that have not yet been considered for addition to the tree of paths. The wavefront set $\mathcal{V}_{\text{open}}$, on the other hand, tracks in sorted order (by cost from the root) only those nodes which have already been added to the tree that are near enough to tree leaves to actually form

better connections. These sets play the same role as their counterparts in FMT*, see [150, 12]. However, in this case BFMT* "grows" two such trees, referred to as $\mathcal{T} = (\mathcal{V}, \mathcal{E}, \mathcal{V}_{\text{unvisited}}, \mathcal{V}_{\text{open}})$ and $\mathcal{T}' = (\mathcal{V}', \mathcal{E}', \mathcal{V}'_{\text{unvisited}}, \mathcal{V}'_{\text{open}})$. Initially, $\mathcal{T}$ is the tree rooted at $\mathbf{x}_{\text{init}}$, while $\mathcal{T}'$ is the tree rooted at $\mathbf{x}_{\text{goal}}$. Note, however, that the trees are exchanged during the execution of BFMT*, so $\mathcal{T}$ in Algorithm 15 is not always the tree that contains $\mathbf{x}_{\text{init}}$.

The BFMT* algorithm is represented in Algorithm 15. Before describing BFMT* in detail, the basic planning functions employed by the algorithm are listed. Let SAMPLEFREE(n) be a function that returns a set of $n \in \mathbb{N}$ points sampled independently and identically from the uniform distribution on $\mathcal{X}_{\text{free}}$. Let COST($\overline{\mathbf{x}'\mathbf{x}}$) be the cost of the straight-line path between configurations $\mathbf{x}'$ and $\mathbf{x}$. Let PATH($\mathbf{z}, \mathcal{T}$) return the unique path in tree $\mathcal{T}$ from its root to node $\mathbf{z}$. Also, with a slight abuse of notation, let COST($\mathbf{x}$, $\mathcal{T}$) return the cost of the unique path in tree $\mathcal{T}$ from its root to node $\mathbf{x}$, and let COLLISIONFREE($\mathbf{x}$, $\mathbf{y}$) be a boolean function returning true if the straight-line path between configurations $\mathbf{x}$ and $\mathbf{y}$ is collision free. Given a set of samples $\mathcal{A}$, let NEAR($\mathcal{A}$, $\mathbf{z}$, r) return the subset of $\mathcal{A}$ within a ball of radius r centered at sample $\mathbf{z}$ (*i.e.*, the set $\left\{\mathbf{x} \in \mathcal{A} \,\middle|\, \|\mathbf{x} - \mathbf{z}\| < r\right\}$). Let the TERMINATE function represent an external termination criterion (*i.e.*, timeout, maximum number of samples, etc.) which can be used to force early termination (or prevent infinite runtime for infeasible problems). Finally, regarding tree expansion, let SWAP($\mathcal{T}$, $\mathcal{T}'$) be a function that swaps the two trees $\mathcal{T}$ and $\mathcal{T}'$. and let COMPANION($\mathcal{T}$) return the companion tree $\mathcal{T}'$ to $\mathcal{T}$ (or vice versa).

Now the BFMT* algorithm is described. First, a set of n configurations in $\mathcal{X}_{\text{free}}$ is determined by drawing samples uniformly. Two trees are then initialized using INITIALIZE as shown in Algorithm 16, with a forward tree rooted at $\mathbf{x}_{\text{init}}$ and a reverse tree rooted at $\mathbf{x}_{\text{goal}}$. Once complete, tree expansion begins starting with tree $\mathcal{T}$ rooted at $\mathbf{x}_{\text{init}}$ using the EXPAND procedure in Algorithm 17. In the following, the node selected for expansion will be consistently denoted by $\mathbf{z}$, while $\mathbf{x}_{\text{meet}}$ will denote the lowest-cost candidate node for tree connection (*i.e.*, for joining the two trees). The EXPAND procedure requires the specification of a *connection radius* parameter, r_n.

EXPAND implements the "lazy" dynamic programming recursion described (at a high level) in Section 8.2.2, making locally-optimal collision-free connections *from* nodes $\mathbf{x}$ near $\mathbf{z}$ unvisited by tree $\mathcal{T}$ (those in set $\mathcal{V}_{\text{unvisited}}$ within search radius r_n of $\mathbf{z}$) *to* wavefront nodes $\mathbf{x}'$ near each $\mathbf{x}$ (those in set $\mathcal{V}_{\text{open}}$ within search radius r_n of $\mathbf{x}$). Any collision-free edges and newly-connected nodes found are then added to $\mathcal{T}$, the connection candidate node $\mathbf{x}_{\text{meet}}$ is updated, and $\mathbf{z}$ is dropped from the list of wavefront nodes. The key feature of the EXPAND function is that in the execution of the dynamic programming recursion it "lazily" ignores the presence of obstacles (see line 6) – this comes at no loss of (asymptotic) optimality (see also [150, 12]). Note

the EXPAND function is identical to that of unidirectional FMT*, with the exception here of additional lines for tracking the connection candidate $\mathbf{x}_{\text{meet}}$.

After expansion, the algorithm checks whether a feasible path is found on line 8. If unsuccessful so far, TERMINATE (which reports failure upon early termination) is checked before proceeding. If the algorithm has not terminated, it checks whether the wavefront of the companion tree is empty (line 14). If this is the case, the INSERT function shown in Algorithm 18 samples a new configuration $\mathbf{s}$ uniformly from $\mathcal{X}_{\text{free}}$ and tries to connect it to a nearest neighbor in the companion tree within radius r_n. This way, the expanding tree is ensured to have at least one configuration in its wavefront available for expansion on subsequent iterations (the alternative would be to report failure). This mimics anytime behavior, and by forcing samples to lie close to tree nodes they algorithm effectively "reopens" closed nodes for expansion again. Uniform resampling may require many attempts before finding a configuration $\mathbf{s}$ which can be successfully connected to $\mathcal{V}'_{\text{open}}$, though this appeared to have a negligible impact on running time in the experiments. On the other hand, a more effective strategy might bias resampling towards areas requiring expansion (*e.g.*, bottlenecks, traps) rather than uniformly within tree coverage.

The algorithm then proceeds on lines 16-17 with the selection of the next node (and corresponding tree) for expansion. As shown, BFMT* "swaps" the forward and backward trees on each iteration, each being expanded in turns. As INSERT ensures the companion tree $\mathcal{T}'$ always has at least one node in its frontier $\mathcal{V}'_{\text{open}}$, a node is always available for subsequent expansion as the next $\mathbf{z}$.

After selection, the entire process is iterated.

BFMT* – Variations

As for any bi-directional planner, the correctness and computational efficiency of BFMT* hinge upon two key aspects: (i) how computation is interleaved among the two trees (in other words, which wavefront at each step should be chosen for expansion), and (ii) when the algorithm should terminate. For instance, as an alternative tree expansion strategy (*i.e.*, item (i)), one could replace lines 16-17 with the "balanced trees" condition which enforces more of a balanced search, maintaining equal costs from the root within each wavefront such that the two wavefronts propagate and meet roughly equidistantly in cost-to-go from their roots:

16: $\mathbf{z}_1 \leftarrow \underset{\mathbf{x}\in\mathcal{V}_{\text{open}}}{\arg\min}\{\textsc{Cost}(\mathbf{x}, \mathcal{T})\}$
17: $\mathbf{z}_2 \leftarrow \underset{\mathbf{x}'\in\mathcal{V}'_{\text{open}}}{\arg\min}\{\textsc{Cost}(\mathbf{x}', \mathcal{T}')\}$
18: $(\mathbf{z}, \mathcal{T}) \leftarrow \underset{(\mathbf{z}_1,\mathcal{T}),(\mathbf{z}_2,\mathcal{T}')}{\arg\min} \left\{\textsc{Cost}(\mathbf{z}_i, \mathcal{T}_i)\right\}$
19: $\mathcal{T}' = \textsc{Companion}(\mathcal{T})$

Algorithm 15 The Bi-directional Fast Marching Tree Algorithm (BFMT*)

Require: Motion query $\left(\mathbf{x}_{\text{init}}, \mathbf{x}_{\text{goal}}\right)$, Connection radius r_n

1: $\mathcal{S} \leftarrow \{\mathbf{x}_{\text{init}}, \mathbf{x}_{\text{goal}}\} \cup \textsc{SampleFree}(n)$
2: $\mathcal{T} \leftarrow \textsc{Initialize}(\mathcal{S}, \mathbf{x}_{\text{init}})$
3: $\mathcal{T}' \leftarrow \textsc{Initialize}(\mathcal{S}, \mathbf{x}_{\text{goal}})$

4: $\mathbf{z} \leftarrow \mathbf{x}_{\text{init}}$, $\mathbf{x}_{\text{meet}} \leftarrow \varnothing$
5: success $\leftarrow$ false
6: **while** success = false **do**
7: $\quad\textsc{Expand}(\mathcal{T}, \mathbf{z}, \mathbf{x}_{\text{meet}})$
8: $\quad$**if** $\mathbf{x}_{\text{meet}} \neq \varnothing$
9: $\quad\quad\sigma^* \leftarrow \textsc{Path}(\mathbf{x}_{\text{meet}}, \mathcal{T}) \cup \textsc{Path}(\mathbf{x}_{\text{meet}}, \mathcal{T}')$
10: $\quad\quad$success $\leftarrow$ true
11: $\quad$**else**
12: $\quad\quad$**if** $\textsc{Terminate}()$
13: $\quad\quad\quad$**return** Failure
14: $\quad\quad$**else if** $\mathcal{V}'_{\text{open}} = \varnothing$
15: $\quad\quad\quad\textsc{Insert}(\mathcal{T}')$
16: $\quad\mathbf{z} \leftarrow \underset{\mathbf{x}' \in \mathcal{V}'_{\text{open}}}{\arg\min}\{\textsc{Cost}(\mathbf{x}', \mathcal{T}')\}$
17: $\quad\textsc{Swap}(\mathcal{T}, \mathcal{T}')$
18: **return** σ^*

Similarly, as an alternative termination condition (*i.e.*, item (ii)), one might replace line 8 with the "best path" criterion:

$$8:\ \mathbf{z} \in \left(\mathcal{V}' \setminus \mathcal{V}'_{\text{open}}\right)$$

Currently line 8 returns the first available path discovered, at the moment that the two wavefronts touch at $\mathbf{x}_{\text{meet}}$ (which is not, in general, the lowest cost path). This alternative condition, on the other hand, returns the *exact optimal path* from $\mathbf{x}_{\text{init}}$ to $\mathbf{x}_{\text{goal}}$ through the given set $\mathcal{S}$ of n samples. This change terminates BFMT* when the two wavefronts have propagated sufficiently far through each other that no better solution can be discovered. Intuitively-speaking, this occurs at the first moment where the two trees have both selected, at the current iteration or previously, the same node as the minimum cost node $\mathbf{z}$ from their respective roots.

Though seemingly promising ideas, no appreciable differences in performance were found using the above criteria in combination or otherwise; hence the simplest version of BFMT* is reported in Algorithm 15.

Algorithm 16 Initializes a Fast Marching Tree

1: **function** INITIALIZE($\mathcal{S}$, $\mathbf{x}_0$)
2: $\mathcal{V} \leftarrow \{\mathbf{x}_0\}$
3: $\mathcal{E} \leftarrow \varnothing$
4: $\mathcal{V}_{\text{unvisited}} \leftarrow \mathcal{S} \setminus \{\mathbf{x}_0\}$
5: $\mathcal{V}_{\text{open}} \leftarrow \{\mathbf{x}_0\}$
6: **return** $\mathcal{T} \leftarrow (\mathcal{V}, \mathcal{E}, \mathcal{V}_{\text{unvisited}}, \mathcal{V}_{\text{open}})$
7: **end function**

8.3 Experimental results

In this Section, numerical path-planning experiments are provided. The objective is to compare the performance of BFMT* with other sampling-based, asymptotically-optimal planning algorithms (namely, FMT*, RRT*, and PRM*)[1]. Given a planning workspace and query, the quality of the solution returned as a function of the execution time allotted to the algorithm is observed. Here dynamic constraints are neglected and arc-length is used as path cost. As a basis for quality comparison between incremental or "anytime" planners (such as RRT*) and non-incremental planners (such as BFMT*, which generate solutions via sample batches), the number of samples drawn by the planners during the planning process is varied (which in essence serves as a proxy to execution time). Note *sample count* has a different connotation depending on the planner that will not necessarily be the number of nodes stored in the constructed solution graph – for RRT* (with one sample drawn per iteration), this is the number of iterations, while for FMT*, PRM*, and BFMT*, this is the number of free space samples taken during initialization.

8.3.1 Simulation Setup

To generate simulation data for a given experiment, the planning algorithms were queried once each for a series of sample counts, recorded the cost of the solution returned, the planner execution time[2], and whether the planner succeeded or not, then repeated this process over 50 trials. To ensure a fair comparison, each planning algorithm was tested using the Open Motion Planning Library (OMPL) v1.0.0 [159], which provides high-quality implementations of many state-of-the-art planners and a common framework for executing motion plans. In this way, it is ensured that

[1]Existing state-of-the-art sampling-based, bi-directional algorithms (namely, RRT-Connect and SBL) were initially also included. However, average costs for RRT-Connect and SBL were roughly 2-4x greater, which occluded the details of other curves; they were thus omitted for clarity

[2]Code for all experiments was written in C++. Corresponding programs were compiled and run on a Linux-operated PC, clocked at 2.4 GHz and equipped with 7.5 GB of RAM.

Algorithm 17 Fast Marching Tree Expansion Step

Require: Connection radius r_n

function EXPAND($\mathcal{T} = (\mathcal{V}, \mathcal{E}, \mathcal{V}_{\text{unvisited}}, \mathcal{V}_{\text{open}})$, $\mathbf{z}$, $\mathbf{x}_{\text{meet}}$)
  $\mathcal{V}_{\text{open,new}} \leftarrow \varnothing$
  $Z_{\text{near}} \leftarrow \text{NEAR}(\mathcal{V}_{\text{unvisited}}, \mathbf{z}, r_n)$
  for $\mathbf{x} \in Z_{\text{near}}$ **do**
    $X_{\text{near}} \leftarrow \text{NEAR}(\mathcal{V}_{\text{open}}, \mathbf{x}, r_n)$
    $\mathbf{x}_{\min} \leftarrow \underset{\mathbf{x}' \in X_{\text{near}}}{\arg\min} \{\text{COST}(\mathbf{x}', \mathcal{T}) + \text{COST}(\overline{\mathbf{x}'\mathbf{x}})\}$
    if COLLISIONFREE($\mathbf{x}_{\min}, \mathbf{x}$) **then**
      $\mathcal{V} \leftarrow \mathcal{V} \cup \{\mathbf{x}\}$ ▷ Add $\mathbf{x}$ to tree
      $\mathcal{E} \leftarrow \mathcal{E} \cup \{(\mathbf{x}_{\min}, \mathbf{x})\}$ ▷ Add edge to tree
      $\mathcal{V}_{\text{unvisited}} \leftarrow \mathcal{V}_{\text{unvisited}} \setminus \{\mathbf{x}\}$ ▷ Mark $\mathbf{x}$ visited
      $\mathcal{V}_{\text{open,new}} \leftarrow \mathcal{V}_{\text{open,new}} \cup \{\mathbf{x}\}$ ▷ Save $\mathbf{x}$
      if $\{\mathbf{x} \in \mathcal{V}'$ and $\text{COST}(\mathbf{x}, \mathcal{T}) + \text{COST}(\mathbf{x}, \mathcal{T}') < \text{COST}(\mathbf{x}_{\text{meet}}, \mathcal{T}) +$ **then** $\text{COST}(\mathbf{x}_{\text{meet}}, \mathcal{T}')\}$
        $\mathbf{x}_{\text{meet}} \leftarrow \mathbf{x}$ ▷ Save $\mathbf{x}$ as best connection
  $\mathcal{V}_{\text{open}} \leftarrow (\mathcal{V}_{\text{open}} \cup \mathcal{V}_{\text{open,new}}) \setminus \{\mathbf{z}\}$ ▷ Add new nodes to wavefront; drop $\mathbf{z}$ from the wavefront
  return $\mathcal{T} \leftarrow (\mathcal{V}, \mathcal{E}, \mathcal{V}_{\text{unvisited}}, \mathcal{V}_{\text{open}})$
end function

all algorithms employed the *exact same* primitive routines (*e.g.*, nearest-neighbor search, collision-checking, data handling, *etc.*), and their performances can be fairly measured. Regarding implementation, BFMT*, FMT*, and PRM* used $\eta = 0$ for the nearest-neighbor radius r_n in order to satisfy the theoretical bounds provided in [26]. For RRT*, the default OMPL settings were used; namely, a 5% goal bias and a steering parameter equal to 20% of the maximum extent of the configuration space. For FMT*, the same INSERT routine as BFMT* for configuration resampling upon failure is included. For all algorithms, early termination (*e.g.*, using TERMINATE for BFMT*) was suppressed by defining a 1000 second time limit, well above each planner's worst-case execution time.

Three benchmarking test scenarios were considered: (1) a 2D "bug trap" and (2) a 2D "maze" problem for a convex polyhedral robot in the $\mathbb{SE}(2)$ configuration space, as well as (3) a challenging 3D problem called the "α-puzzle" in which two loops of metal (non-convex) are untangled in the $\mathbb{SE}(3)$ configuration space.

All problems were drawn directly from OMPL's bank of tests, and are illustrated in Figure 8.2. In each case, collision-checks relied on OMPL's built-in collision-checking

Algorithm 18 Insertion of New Samples

Require: Connection radius r_n

function INSERT($\mathcal{T} = (\mathcal{V}, \mathcal{E}, \mathcal{V}_{\text{unvisited}}, \mathcal{V}_{\text{open}})$)
  while $\mathcal{V}_{\text{open}} = \varnothing$ **and not** TERMINATE() **do**
    $\mathbf{s} \leftarrow$ SAMPLEFREE(1)
    $V_{\text{near}} \leftarrow$ NEAR($\mathcal{V}, \mathbf{s}, r_n$)
    while $V_{\text{near}} \neq \varnothing$ **do**
      $\mathbf{x}_{\min} \leftarrow \arg\min_{\mathbf{x} \in V_{\text{near}}} \{\text{COST}(\mathbf{x}, \mathcal{T}) + \text{COST}(\overline{\mathbf{xs}})\}$
      if COLLISIONFREE($\mathbf{x}_{\min}$, $\mathbf{s}$) **then**
        $\mathcal{V} \leftarrow \mathcal{V} \cup \{\mathbf{s}\}$ ▷ Add $\mathbf{s}$ to tree
        $\mathcal{E} \leftarrow \mathcal{E} \cup \{(\mathbf{x}_{\min}, \mathbf{s})\}$ ▷ Add edge to tree
        $\mathcal{V}_{\text{open}} \leftarrow \mathcal{V}_{\text{open}} \cup \{\mathbf{s}\}$ ▷ Add to wavefront
        break
      else
        $V_{\text{near}} \leftarrow V_{\text{near}} \setminus \{\mathbf{x}_{\min}\}$
  return $\mathcal{T} \leftarrow (\mathcal{V}, \mathcal{E}, \mathcal{V}_{\text{unvisited}}, \mathcal{V}_{\text{open}})$
end function

library, FCL. Additionally, to tease out the performance of BFMT* relative to FMT* in high-dimensional environments, a point mass robot moving in cluttered unit hypercubes of 5 and 10 dimensions is studied.[3]

(a) $\mathbb{SE}(2)$ bug trap (b) $\mathbb{SE}(2)$ maze (c) $\mathbb{SE}(3)$ α-puzzle

Figure 8.2: Depictions of the three OMPL rigid-body planning problems

[3]The space is populated up to 50% obstacle coverage with randomly-sized, axis-oriented hyper-rectangles. $\mathbf{x}_{\text{init}}$ was set to the center at $[0.5, \ldots, 0.5]$, with the goal $\mathbf{x}_{\text{goal}}$ at the ones-vector (*i.e.*, $[1, \ldots, 1]$).

Before proceeding, note that each marker shown on the plots throughout this Section represents a single simulation at a fixed sample count. The points on the curves, however, represent the mean cost/time of *successful* algorithm runs *only* for a particular sample count, with error bars corresponding to one standard deviation of the 50 run sample mean.[4] Sample counts varied from the order of 200 to 2000 points for 2D problems, from 1000 to 30000 points for 3D problems, and 500 to 4000 points for the hypercube examples.

8.3.2 Results and Discussion

Here benchmarking results are presented(average solution cost versus average execution times and success rates) comparing BFMT* to other state-of-the-art sampling-based planners.

Figure 8.3 shows the results for each BFMT*, FMT*, RRT*, and PRM*. Performance here is measured by execution time on the x-axis and solution cost on the y-axis—high quality data points are therefore located in the lower-left corner (low-cost solutions obtained quickly). The plots reveal that both FMT* and BFMT* for the most part outperform RRT* as well as PRM*. In particular, BFMT* and FMT* achieve higher success rates (always a flat 100% for the cases studied) in shorter time. To extract further information, each test is examined in detail.

[4]Standard deviation of the mean indicates where it is expected with one-σ confidence the distribution mean to lie based on the 50-run sample mean, and is related to the standard deviation of the distribution by $\sigma_\mu = \sigma/\sqrt{50}$.

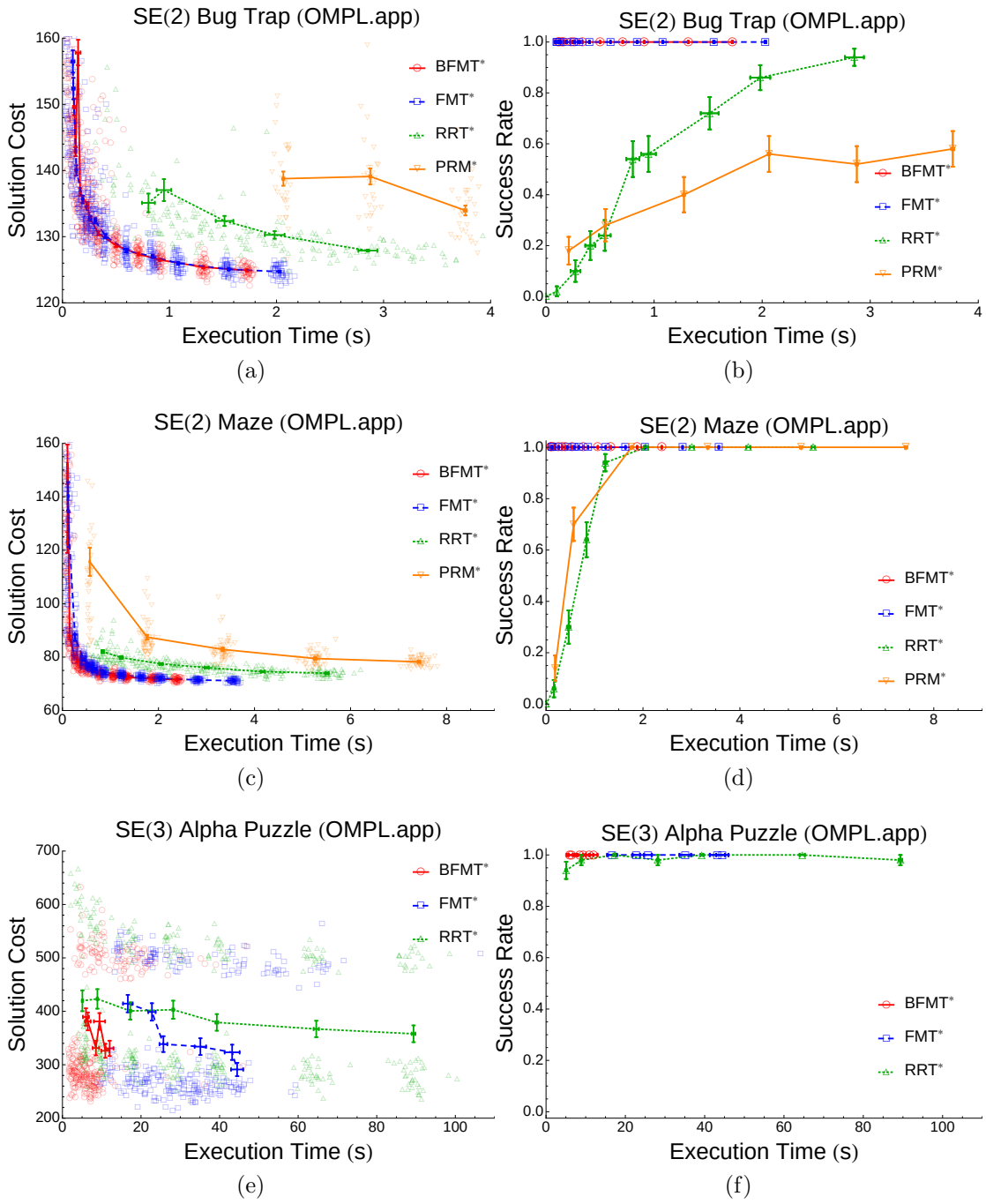


Figure 8.3: Simulation results for the three OMPL scenarios.

In the Bug Trap and Maze problems, BFMT* notably generates the same cost-time curve as FMT* (meaning they return solutions of very similar cost for a given

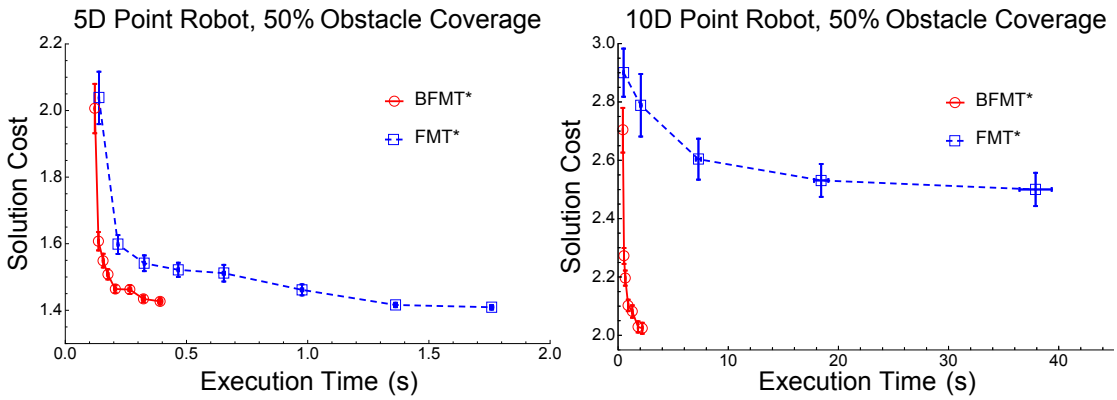
sample count), but with data points shifted to the left (indicating they were obtained in shorter execution time). Though not shown due to slow running times for PRM* (whose results had to be truncated to clarify detail), all planners appear to tend towards similar low-cost solutions as more execution time was allocated. However BFMT* and FMT* seem to converge to an optimum much faster, particularly for the Maze problem (on the order of 1.5 and 2.0 seconds respectively, compared to 3-4 seconds for RRT* and 5-7 seconds for PRM*). This contrast becomes even more evident for the α-puzzle. Here an unusual spread of solutions is appreciated– one in a band at around 500 cost and another at around 275. These indicate the presence of two solution types, or *homotopy classes*: one corresponding to the true α-puzzle solution, and another less-efficient path. This appears to have yielded a "bump" in the BFMT* cost-curve, where increasing the sample count momentarily gives an increased average cost. It is believed this is a result of how BFMT* trees interconnect; at this count, by unlucky circumstance, the longer homotopy seems to be found first more often than usual. But the behavior disappears as $n \to \infty$. Note RRT* seems to avoid this issue through goal biasing. Despite the difficult problem structure, BFMT* finds the cheaper homotopy faster than other planners, with many more of its data points clustered in the lower-left corner, generally at lower costs and times than RRT* and of equal quality but faster times than FMT*.

These results suggest that BFMT* tends to an optimal cost at least as fast as the other planners, and sometimes much faster. To shed light on the relative performance of FMT* and BFMT* further, they are compared in higher dimensions. Results for the 5D and 10D hypercube are shown in Figure 8.4 (success rates were again at 100%, and were thus omitted). Here BFMT* substantially outperforms FMT*, particularly as dimension increases, with convergence in roughly 0.5 and 1.4 seconds (5D), and 5 and 20 seconds (10D) on average.

This suggests that *reachable volumes* play a significant role in their execution time. The relatively small volume of reachable configurations around the goal at the corner implies that the reverse tree of BFMT* expands its wavefront through many fewer states than the forward tree of FMT* (which in fact needlessly expands towards the zero-vector); tree interconnection in the bi-directional case prevents its forward tree from growing too large compared to unidirectional search. This is pronounced exponentially as the dimension increases. In trap or maze-like scenarios, however, bi-directionality does not seem to change significantly the number of states explored by the marching trees, leading to comparable performance for the $\mathbb{SE}(2)$ bug-trap and maze.

Note it is expected a greater contrast in execution times in favor of BFMT* as the cost of collision-checking increases, such as with many non-convex obstacles or in time-varying environments.

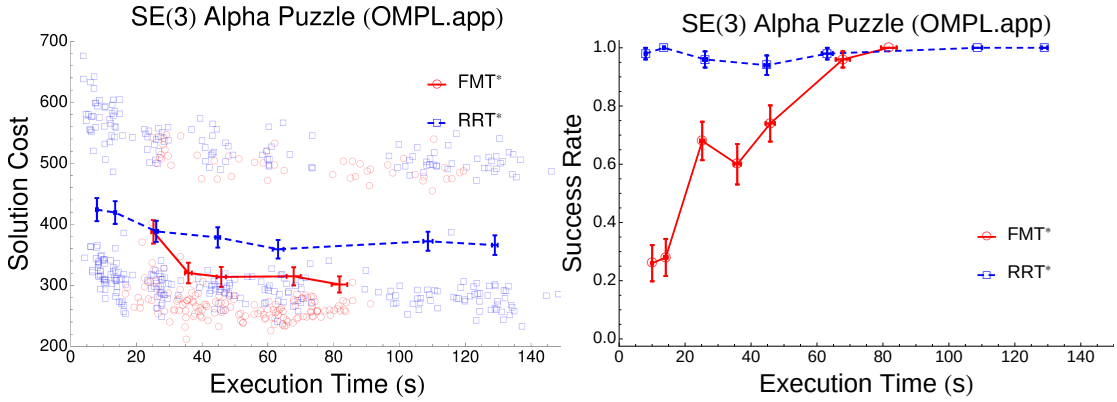
Finally, in order to analyze the impact of the INSERT function introduced, the



(a) (b)

Figure 8.4: FMT* and BFMT* results for 5D and 10D cluttered hypercubes, 50% obstacle coverage; all success rates were 100%.

results of the previous FMT* version for the α-puzzle are taken directly from the literature [12], shown in Figure 8.5. Although computation times are not directly comparable (since these results correspond to a different hardware setup), it is possible to compare its evolution with respect to RRT*. The results of this Chapter present a better convergence rate for FMT* than for RRT*, as the *gap* among them is larger. This is related with many implementation improvements not mentioned in this Thesis, such as better data structures and collision-check caching.



(a) (b)

Figure 8.5: Results for the previous version of FMT* in α-puzzle, without Insert procedure.

The success rate, however, can be directly compared as in FMT* is only depends on the sample set (given that the maximum computation time, 1000 seconds, is by far

enough to solve the α-puzzle). In fact, previous FMT* version required a huge number of samples in order to reach 100% success rate. However, in the results presented in this Chapter both FMT* and BFMT* present a constant success rate of 100%.

8.4 Conclusion

In this Chapter, a bi-directional, sampling-based, asymptotically-optimal motion planning algorithm named BFMT* has been presented. Numerical experiments in $\mathbb{R}^d$, $\mathbb{SE}(2)$, and $\mathbb{SE}(3)$ revealed that BFMT* tends to an optimal solution at least as fast as its state-of-the-art counterparts, and in some cases significantly faster. Convergence rates are expected to improve with parallelization, in which each tree is grown using a separate CPU.

Future research will examine BFMT*'s interaction with more advanced techniques, such as adaptive sampling near narrow passages or sample biasing in Insert (Algorithm 18) towards failed wavefronts. It is also planned to extend BFMT* to dynamic environments through lazy re-evaluation (leveraging its tree-like forward and reverse path structures) in a way that reuses previous results as much as possible. Maintaining bounds on run-time performance and solution quality in this new context will be the greatest challenges. Ultimately, it is expected that BFMT* will enable fast, easy-to-implement planning and re-planning in a wide range of time-varying scenarions, much as shown here for the static case.

Chapter 9

General Conclusions

This Thesis has focused on a practical approach to path and motion planning. The Fast Marching Method has been used along the document as a basis for the algorithms built. Despite the fact that this method is considered a classic approach, it has been shown that there is still room for improvement. Each Chapter contains specific conclusions about its content and possible future work. However, general conclusions are discussed here.

The key factors of FMM is that it is well formulated, easy to understand and to implement, while providing robust and deterministic results. Combined with additional methods, it is able to solve complex tasks such as motion learning or navigation of large robots in cluttered environments.

On the other hand, its main disadvantage is that is suffers from the curse of dimensionality. Although it can be very efficiently implemented using $\mathcal{O}(n)$ approaches, where n is the number of cells in the environment, its execution time escalates exponentially with the number of dimensions, as n is a power function of the number of dimensions. However, recent work shows that FMM-like algorithms can be extrapolated to sampling-based algorithms. This enormously increases the possibilities of FMM, as sampling-based algorithms evolve almost linearly with the number of dimensions. Therefore, most of the work done with classic, grid-based FMM (and derivatives) can be replicated using FMT* and BFMT*.

9.1 Future Work

This Thesis has explored many different aspects of FMM-like algorithms and their application to path and motion planning. Therefore, new ideas have arose given the flexibility of FMM-based methods. Concretely, three different main future works are devised:

Firstly, the proposal of new FMM-based methods with the objective of improving even further the computational time. For this, it is proposed to create a hybrid method between UFMM and SFMM. Although not deeply discussed in this Thesis, FMM and SFMM can be extended with cost-to-go heuristics. However, up to the author's knowledge, there is no previous work including these heuristics to other methods. Apparently, it is straight forward for UFMM and GMM. The addition of heuristics to DDQM and FIM do not seem that easy but could highly improve the computational time of these methods. However, given the nature of FSM and LSM, it seems complicated to include any kind of heuristics to these methods.

Regarding motion learning, the main drawback of the FML algorithm is that it is not able to properly generalize velocities of the motion for places without experience. Also, complex motions such loops or spirals cannot be properly learned: FML with take shortcut the motion in the intersection of the loop. This could be easily solved by increasing the number of dimensions, and using time derivatives as new dimensions,

but at the cost of exponentially increased computational time. A very promising idea is to use the anisotropic variations of Fast Marching. This way, the velocity map is a vector field, instead of scalar. Therefore, velocities could be also encoded without increasing number of dimensions. Together with optimization algorithms, it would return a policy with motion velocities and accelerations properly generalized.

Lastly, the combination of Fast Marching with sampling-based methods has proven to be a powerful idea, improving state-of-the-art methods. More basic research in this area is required in order to optimize the algorithms proposed. However, obvious improvements can be done, such as paralellize the BFMT*, nearest-neighbor precomputation, heuristics addition, etc. Furthermore, motion learning and other high-level task could be solved in higher dimensions and lower computational time using FMT* and BFMT*.

Appendix A

n-Dimensional Grid Maps

Appendix A.1: Introduction

Mathematically, grid maps are data structures that divide (discretize) the space in cubes (hypercubes) of N dimensions. They are commonly used in artificial intelligence algorithms, such path planning. Although their mathematical definition is clear and simple, it is not that easy to work with them when the number of dimensions is variable. Therefore, this Appendix details the mathematical generalization of common operations with grid maps and how to implement them. Another useful tutorial on grid maps (but only for 2D) can be found in `http://www-cs-students.stanford.edu/~amitp/game-programming/grids/` which includes triangular and hexagonal grids.

The main reason of this Appendix is that, when trying to implement such structures, it becomes difficult to generalize. For example, boost::multi_array library provides tools to create n-dimensional arrays in which the number of dimensions has to be known in compilation time, which is an important limitation. There is a lack in the available software of n-dimensional grid maps in which the size can be dynamic, even in the number of dimensions, in run time. Therefore, all the operations in this Appendix are parametrized by the number of dimensions and their size. This is probably not a novel work, but it is not easy to find a similar document in the literature.

Appendix A.2: Definitions

The definitions of this Appendix apply for any parallelogram-based grid map. Hence, define an n-dimensional grid map as the set of cells correctly ordered whose dimensions are consistent in terms of size. In other words, if the first row has 5 columns, the second row will also have 5 columns.

An n-dimensional grid map G is composed by n_{dims} dimensions. The size of each dimension is stored in a vector $d = [d_0, d_1, \ldots, d_{n-1}]$ and the size of the total grid map, given in number of cells, is:

$$size(G) = \prod_{i=0}^{n-1} d_i = d_0 \cdot d_1 \cdot \cdots \cdot d_{n-1} \tag{A.1}$$

Each cell within the grid map can be accessed in a double manner:

1. By its **index**. Each cell has a specific index within the grid map which completely depends on the ordering convention chosen.

2. By its **coordinates**, giving a set of coordinates $c = [c_0, c_1, \ldots, c_{n-1}]$.

The conversion from index to coordinates and vice-versa is well-known for a given number of dimensions. However, to parametrize is more complex. These operations are detailed in Section A.4

In Figure A.1 examples of 2D and 3D grid maps are shown. Note that the index ordering is not unique. In this case this specific ordering has been chosen ordering since it is easier to match with the physical dimensions of the grid (dimension 0 is x, dimension 1 is y, and so on). For instance, in computer vision it is almost an standard to place the first cell (pixel) in the top-left of the grid map (image) with the dimension 0 (rows) going downwards and dimension 1 (columns) leftwards. In any case, the formulation should be valid in any case (with minor adjustements).

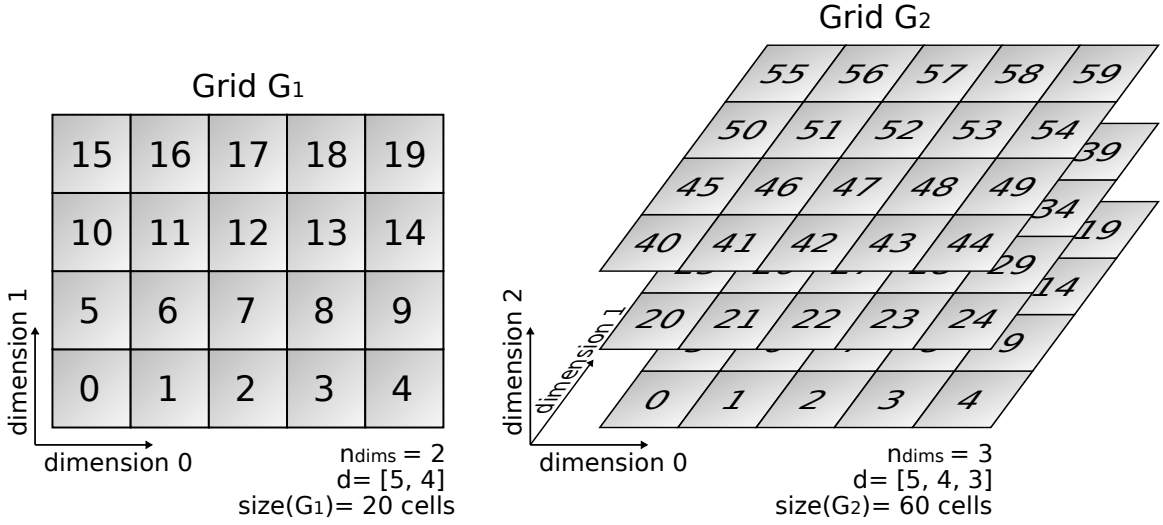


Figure A.1: Example of a 2D and a 3D grid map. Usually, 3D grid maps are represented with cubes. The numbers within the cells represent the indices of those cells.

Appendix A.3: General Neighbor Extraction

In this Section the generalization of the neighbor extraction in a **4-connectivity** scheme is detailed.

In order to help the reader, the 2D and 3D cases will be exposed, and the n-dimensional formulation will be derived. Along the Appendix cells are always referred by their indices. When the dimensions of the grid are known it is easy to build a vector of size n_{dims} and check the neighbors by doing ± 1 in each coordinate. However, in this case the dimensions of the grid are not known until execution. Therefore, to generalize it is much easier and efficient to work indices, as shown in the next paragraphs.

A.3.1 2-dimensional Neighbor Extraction

In a 2-dimensional map, the neighbor extraction is almost direct. In this case, there are 4 neighbors, as shown in Figure A.2.

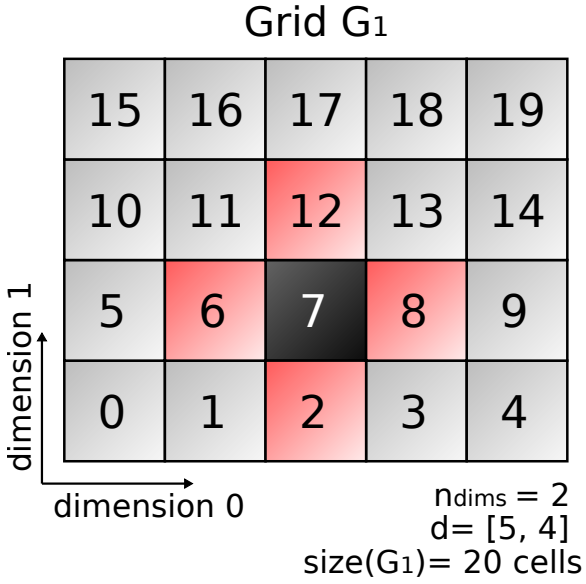


Figure A.2: 4 neighbors highlighted in red of cell with index 7 (shaded) in a 2D grid map.

Now, focus on the case shown in Figure A.3: a general 2D grid map where $d = [d_0, d_1]$ is not known in advance. The neighbors of a cell with index i, $\mathcal{N}(i)$ are given by the following expression:

$$\mathcal{N}(i) = \begin{cases} \begin{cases} i-1 \\ i+1 \end{cases} & \text{for dimension 0} \\ \begin{cases} i-d_0 \\ i+d_0 \end{cases} & \text{for dimension 1} \end{cases} \tag{A.2}$$

In the case shown in Figure A.3, $i = 2d_0 + 2$. Hence, its neighbors will be:

$$\mathcal{N}(i) = \mathcal{N}(2d_0+2) = \begin{cases} \begin{cases} i-1 = 2d_0+3 \\ i+1 = 2d_0+1 \end{cases} & \text{for dimension 0} \\ \begin{cases} i-d_0 = d_0+2 \\ i+d_0 = 3d_0+2 \end{cases} & \text{for dimension 1} \end{cases} \tag{A.3}$$

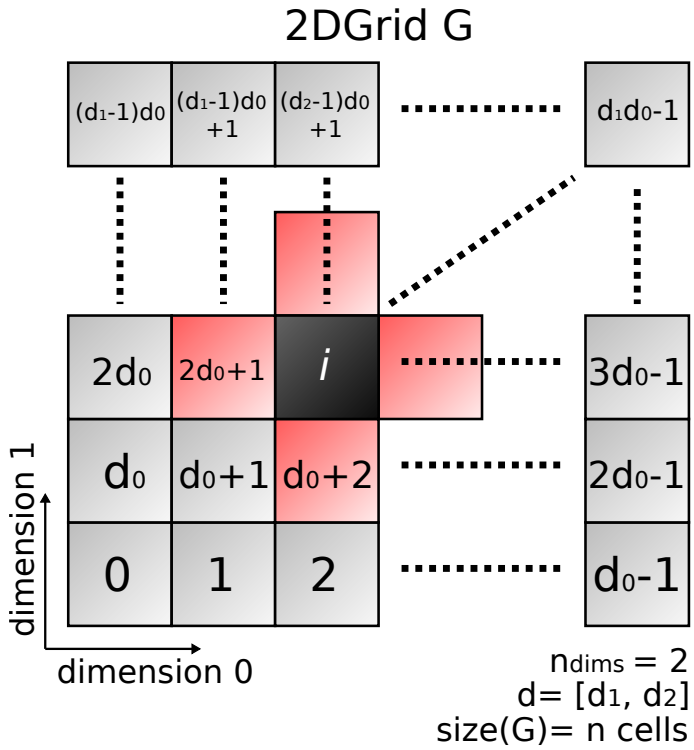


Figure A.3: 4 neighbors highlighted in red of cell with index i (shaded) in a generic 2D grid map.

Checking neighbors validity

If the queried index i is in one of the borders of the grid map, it will happen that the neighbors are not valid. Recalling Figure A.2, suppose that the cell $i = 19$ is queried for neighbors. According equation (A.2), the set of neighbors would be $\mathcal{N}(19) = 18, 20, 14, 24$. However, the cells with index greater or equal to 20 does not exist. The returned neighbor is out of bounds of the given grid map. Also, $\mathcal{N}(14) = 13, 15, 9, 19$ gives index 15 as neighbor. In this case it is supposed to be a neighbor in dimension 0, but its value for dimension 1 (coordinate 1) is $c_1 = 3$ while this value for cell index 14 is $c_1 = 2$. Therefore, it is not a neighbor, as their value for dimension 0 is not the same.

This checking is easy if when working with coordinates. Coordinates of cell index 14 are $c(14) = [4, 2]$. Neighbors in each dimension can be obtained by doing ± 1 in each dimension. This means $\mathcal{N}([4, 2]) = [3, 2], [5, 2], [4, 1], [4, 3]$, where $[5, 2]$ is out of bounds of the grid. When working with indices, the following has to be checked:

- **Dimension 0:** Are the 2 neighbors in the same row (c_1) that the queried cell?

- **Dimension 1:** Are the 2 neighbors within grid bounds?

The mathematical expression to check if the given indices are neighbors of i is as follows:

1. Neighbors of i in dimension 0 are valid if (operator $[\cdot]$ means the integer part, this is, the integer number immediately below):

$$\left[(i \pm 1)/d_0\right] = \left[i/d_0\right] \tag{A.4}$$

2. Neighbors of i in dimension 1 are valid if:

$$\begin{aligned} &i - d_0 \geq 0 \\ &i + d_0 < size(g) = d_0 \cdot d_1 \end{aligned} \tag{A.5}$$

A.3.2 3-dimensional Neighbor Extraction

Following the same procedure as for 2D grid maps, the neighbors extraction in a 3D grid map whose dimensions are not known until run time, as shown in Figure A.4, is detailed in the following lines. In this case, there are a maximum of 6 neighbors. The schema of a 3D grid map with undefined dimensions size is omitted as is too complex to interpret. The following expression is valid to get the neighbors of such grid map:

$$\mathcal{N}(i) = \begin{cases} \begin{cases} i-1 \\ i+1 \end{cases} & \text{for dimension 0} \\ \begin{cases} i-d_0 \\ i+d_0 \end{cases} & \text{for dimension 1} \\ \begin{cases} i-d_0 \cdot d_1 \\ i+d_0 \cdot d_1 \end{cases} & \text{for dimension 2} \end{cases} \tag{A.6}$$

Checking neighbors validity

The validity of the indices returned by the neighbors extraction function has to be checked as done previously for 2D grid maps. Analogously, the procedure is as follows:

- **Dimension 0:** Are the 2 neighbors in the same row (c_1) that the queried cell?
- **Dimension 1:** Are the 2 neighbors within the same 2D grid slice?
- **Dimension 2:** Are the 2 neighbors within grid bounds?

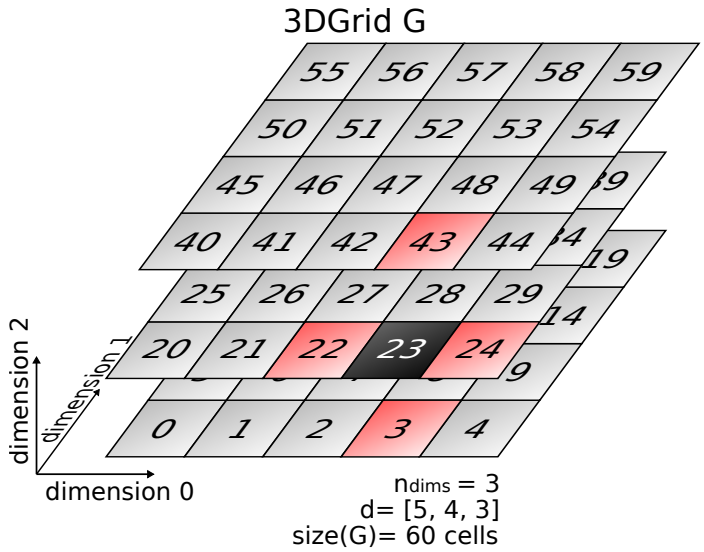


Figure A.4: 6 neighbors highlighted in red of cell with index i (shaded) in a 3D grid map.

The mathematical expression to check if the given indices are neighbors of i is as follows:

1. Neighbors of i in dimension 0 are valid if (operator $[\cdot]$ means the integer part, this is, the integer number immediately below):

$$\left[(i \pm 1)/d_0\right] = \left[i/d_0\right] \tag{A.7}$$

2. Neighbors of i in dimension 1 are valid if:

$$\left[(i \pm d_0)/(d_0 \cdot d_1)\right] = \left[i/(d_0 \cdot d_1)\right] \tag{A.8}$$

3. Neighbors of i in dimension 2 are valid if:

$$\left[(i \pm d_0 \cdot d_1)/(d_0 \cdot d_1 \cdot d_2)\right] = \left[i/(d_0 \cdot d_1 \cdot d_2)\right] \tag{A.9}$$

A.3.3 n-dimensional Neighbor Extraction

In light of the step from 2D to 3D neighbor extraction, it is possible to generalize the formulation for n-dimensions according to the next expressions:

$$\mathcal{N}(i) = \begin{cases} \begin{cases} i-1 \\ i+1 \end{cases} & \text{for dimension 0} \\ \begin{cases} i-d_0 \\ i+d_0 \end{cases} & \text{for dimension 1} \\ \begin{cases} i-d_0 \cdot d_1 \\ i+d_0 \cdot d_1 \end{cases} & \text{for dimension 2} \\ \vdots & \\ \begin{cases} i-\prod_{k=0}^{n-2} d_k = i-d_0 \cdot d_1 \cdot d_2 \cdot \cdots \cdot d_{n-2} \\ i+\prod_{k=0}^{n-2} d_k = i+d_0 \cdot d_1 \cdot d_2 \cdot \cdots \cdot d_{n-2} \end{cases} & \text{for dimension n-1} \end{cases} \tag{A.10}$$

Checking neighbors validity

- **Dimension 0:** Are the 2 neighbors in the same row (c_1) that the queried cell?
- **Dimension 1:** Are the 2 neighbors within the same 2D grid slice?
- **Dimension 2:** Are the 2 neighbors within the same 3D grid slice?
  ⋮
- **Dimension n-1:** Are the 2 neighbors within the same nD grid slice (in bounds)?

More formally:

1. Neighbors of i in dimension 0 are valid if (operator $[\cdot]$ means the integer part, this is, the integer number immediately below):

$$\left[(i \pm 1)/d_0\right] == \left[i/d_0\right] \tag{A.11}$$

2. Neighbors of i in dimension 1 are valid if:

$$\left[(i \pm d_0)/(d_0 \cdot d_1)\right] = \left[i/(d_0 \cdot d_1)\right] \tag{A.12}$$

$\vdots$

n. Neighbors of $n-1$ in dimension 2 are valid if:

$$\left[(i \pm \prod_{k=0}^{n-2} d_k)/\prod_{k=0}^{n-1} d_k\right] = \left[i/\prod_{k=0}^{n-1}\right] \tag{A.13}$$

that means:

$$\left[(i \pm d_0 \cdot d_1 \cdot \cdots \cdot d_{n-2})/(d_0 \cdot d_1 \cdot \cdots \cdot d_{n-1})\right] = \left[i/(d_0 \cdot d_1 \cdot \cdots \cdot d_{n-1})\right] \tag{A.14}$$

Appendix A.4: Helper Functions

This Section describes helpful functions that to handle such n-dimensional grid maps.

A.4.1 Index to coordinates

It is useful to transform cell indices into sets of coordinates for debug or printing purposes, or just to give a better interface to the user. Given an index i of an grid map with n dimensions with dimension sizes d, the set of coordinates c can be computed as follows (it is easier to start from the last dimension):

$$\begin{aligned}
c_{n-1} &= \left[i/\textstyle\prod_{k=0}^{n-2} d_k\right] \\
c_{n-2} &= \left[(i - c_{n-1} \cdot \textstyle\prod_{k=0}^{n-2} d_k)/\prod_{k=0}^{n-3} d_k\right] \\
c_{n-3} &= \left[(i - c_{n-1} \cdot \textstyle\prod_{k=0}^{n-2} d_k - c_{n-2} \cdot \prod_{k=0}^{n-3} d_k)/\prod_{k=0}^{n-4} d_k\right] \\
&\vdots \\
c_0 &= \left[(i - c_{n-1} \cdot \textstyle\prod_{k=0}^{n-2} d_k - c_{n-2} \cdot \prod_{k=0}^{n-3} d_k - \cdots - c_1 \cdot d_0)/1\right]
\end{aligned} \tag{A.15}$$

Note Special care has to be taken with parenthesis and operators preference when implementing this functions.

A.4.2 Coordinates to index

This operation can be also very useful when dealing with n-dimensional grid maps. Given a set of coordinates c of a cell within a grid map with n dimensions and

dimension sizes d, the cell index can be computed as shown in the next equation:

$$i = c_{n-1} \cdot \prod_{k=0}^{n-2} d_k + c_{n-2} \cdot \prod_{k=0}^{n-3} d_k + \cdots + c_1 \cdot d_0 + c_0 \tag{A.16}$$

Appendix A.5: Implementation

An open-source implementation of the previous formulation is available online: `https://github.com/jvgomez/fastmarching/tree/master/ndgridmap` The software is distributed under the free software license GNU/GPL v3.0.

This implementation is based on the STL vector and array templated classes. Previous versions allowed run time modifications in the number of dimensions and their size (no other code was found on the internet with this feature). However, in new versions it was decided to set the number of dimensions in compilation time (since this is easy to predict). This gives a plus of efficiency to the class since most of the for loops can be optimized by the compiler. To turn on this feature of the latest version is not difficult and does not require too much time.

The class is templated so that the cell element can be whatever the user wants (as long as a minimal required interface is accomplished) following a policies design (also called C++ traits). Its declaration is simple, mainly:

```
template <class T, size_t ndims> class nDGridMap {
              public:

       nDGridMap<T,ndims>() {leafsize_ = 1.0f;}
       nDGridMap<T,ndims> (const std::array<int, ndims> & dimsize, const float
           leafsize = 1.0f);
       virtual ~nDGridMap<T,ndims>();

       void resize (const std::array<int, ndims> & dimsize);
       int size () const;

       T & operator[](const int idx);
       T & getCell (const int idx);

       int getNeighbors (const int idx, std::array<int, 2*ndims> & neighs);
       void getNeighborsInDim (const int idx, std::array<int, 2*ndims>& neighs,
           const int dim);

       int idx2coord (const int idx, std::array<int, ndims> & coords);
       int coord2idx (const std::array<int, ndims> & coords, int & idx);

   private:
       std::vector<T> cells_; // The main container for the class.
```

```
        std::array<int, ndims> dimsize_; // Contains the size of each dimension.
        int ncells_; // Number of cells in the grid (size).

        // Auxiliar arrays to improve performance.
        std::array<int, ndims> d_;
        std::array<int, 2> n; // Internal use in getMinValueInDim().
                              int n_neighs;
};
```

An important performance trick is the array $d\_$. This array **DOES NOT** correspond to the d vector explained in previous sections (this one is $dimsize\_$). The array $d\_$ is computed in a way that $d\_[0] = dimsize\_[0]$, $d\_[1] = dimsize\_[0] \cdot dimsize\_[1]$, $d\_[2] = dimsize\_[0] \cdot dimsize\_[1] \cdot dimsize\_[2]$, and so on. It is precomputed in the constructor (and resize() method) as follows:

```
    for (int i = 0; i < ndims; ++i) {
        ncells_ *= dimsize_[i];
        d_[i] = ncells_;
    }
```

This array is used in many different functions in order to not compute every time the iterative product operation which, in the previous Section, it was used many times.

A.5.1 nDGridCell::getNeigbors()

This function implements the formulation give in equations A.10 and A.13 for n dimensions. The code is as follows:

```
int getNeighbors (const int idx, std::array<int, 2*ndims> & neighs) {
        n_neighs = 0;
          for (int i = 0; i < ndims; ++i)
                   getNeighborsInDim(idx,neighs,i);

          return n_neighs;
}
```

Explanation: In order to get the 4-connectivity neighbors, every dimension is analyzed separately, putting all the found neighbors in the neighs aerray. When this function is called, the private attribute n_neighs is set to 0 and it will count how many neighbors are found, up to a maximum up 2*ndims. Therefore, the getNeighborsInDim() function is called for each dimension:

```
void getNeighborsInDim(const int idx, std::array<int, 2*ndims>& neighs, const int
    dim) {
    int c1,c2;
    if (dim == 0) {
        c1 = idx-1;
        c2 = idx+1;
        // Checking neighbor 1.
    if ((c1 >= 0) && (c1/d_[0] == idx/d_[0]))
        neighs[n_neighs++] = c1;
        // Checking neighbor 2.
    if (c2/d_[0] == idx/d_[0])
        neighs[n_neighs++] = c2;
    }
    else {
        // neighbors proposed.
        c1 = idx-d_[dim-1];
        c2 = idx+d_[dim-1];
        // Checking neighbor 1.
        if ((c1 >= 0) && (c1/d_[dim] == idx/d_[dim]))
            neighs[n_neighs++] = c1;
        // Checking neighbor 2.
        if (c2/d_[dim] == idx/d_[dim])
            neighs[n_neighs++] = c2;
    }
}
```

Equations A.10 and A.13 are applied but with the small modification of using $d\_$ which already contains the iterative product results. Dimension 0 is done apart for code simplicity.

Note the bound checking ($c1 >= 0$) when checking the neighbor computed with $-$. For instance, neighbors of index 0 in a uni-dimensional grid of size 5 would be indices -1 and 1. In C/C++, (int)-1/5 will be 0, while the integer part is -1. Because of this, this additional checking is required. This behavior may differ in different programming languages. In this case, this implementation was chosen as it is more efficient than actually computing the integer part.

A.5.2 nDGridCell::idx2coord()

This function implements the equation A.15 with the same modification as before leveraging $d\_$. This function takes as input the index of the cell and the array of coordinates where the output will be stored. The code is:

```
int idx2coord (const int idx, std::array<int, ndims> & coords) {
    if (coords.size() != ndims)
        return -1;
    else {
```

```
        coords[ndims-1] = idx/d_[ndims-2]; // First step done apart.
        int aux = idx - coords[ndims-1]*d_[ndims-2];
                                for (int i = ndims - 2; i > 0; --i) {
                                                coords[i] = aux/d_[i-1];
                                                aux -= coords[i]*d_[i-1];
                                }
                                coords[0] = aux; //Last step done apart.
                }
                return 1;
}
```

Explanation: First, a dimensional check is carried out to avoid incorrect parameters. The coordinate of the last dimension is done first outside the for loop to initialize the aux variable, which accumulates the subtraction of the values to the index before the division. Lastly, the first coordinate is done as the rest of the subtraction.

A.5.3 nDGridCell::coord2idx()

In this case, the implementation of equation A.16 is much straight forward:

```
int coord2idx (const std::array<int, ndims> & coords, int & idx) {
                if (coords.size() != ndims)
                                return -1;
                else {
                                idx = coords[0];
                                for(int i = 1; i < ndims; ++i)
                                                idx += coords[i]*d_[i-1];
                }
                return 1;
}
```

Explanation: The function gets as parameters the array of indices to convert and the index to be returned. After a checking the dimensions, the idx variable is incremented for every dimension according to its coordinate in that dimension.

IMPORTANT NOTE Note that the indexing is not valid for certain languages. In Matlab, for instance, all the indices of an array (or matrix) will be from 1 to n (instead from 0 to $n-1$ as in C++). This applies to all the code shown in this Appendix. Take into account also the language-dependent behavior of certain functions, such as integer division.

Bibliography

[1] S. Bak, J. McLaughlin, and D. Renzi, "Some improvements for the fast sweeping method," *SIAM Journal on Scientific Computing*, vol. 32, no. 5, pp. 2853–2874, 2010.

[2] P. Gremaud and C. Kuster, "Computational study of fast methods for the eikonal equation," *SIAM Journal on Scientific Computing*, vol. 27, no. 6, pp. 1803–1816, 2006.

[3] W. Jeong and R. Whitaker, "A fast iterative method for eikonal equations," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2512–2534, 2008.

[4] A. Capozzoli, C. Curcio, A. Liseno, and S. Savarese, "A comparison of fast marching, fast sweeping and fast iterative methods for the solution of the eikonal equation," in *21st Telecommunications Forum*, pp. 685–688, 2013.

[5] L. Yatziv, A. Bartesaghi, and G. Sapiro, "O(n) implementation of the fast marching algorithm," *Journal of Computational Physics*, vol. 212, pp. 393–399, 2005.

[6] M. Jones, J. Baerentzen, and M. Sramek, "3D Sistance Fields: a Survey of Techniques and Applications," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 7, pp. 581–599, 2006.

[7] D. Fonte, F. Valente, A. Vale, and I. Ribeiro, "A Motion Planning Methodology for Rhombic-like Vehicles for ITER Remote Handling Operations," in *IFAC Symposium on Intelligent Autonomous Vehicles*, pp. 443–448, 2010.

[8] F. Valente, A. Vale, D. Fonte, and I. Ribeiro, "Optimized Trajectories of the Transfer Cask System in ITER," *Fusion Engineering and Design*, vol. 86, pp. 1967–1970, 2011.

[9] D. Álvarez, J. V. Gómez, S. Garrido, and L. Moreno, "3D Robot Formations Planning With Fast Marching Square," *Journal of Intelligent and Robotic Systems*, pp. 1–17, 2015.

[10] D. Álvarez, A. Lumbier, J. V. Gómez, S. Garrido, and L. Moreno, "Precision Grasp Planning with Gifu Hand III based on Fast Marching Square," in *IEEE/RSJ International Conference on Intelligent Robots & Systems*, 2013.

[11] J. V. Gómez, N. Mavridis, and S. Garrido, "Fast Marching Solution for the Social Path Planning Problem," in *Proceedings of the IEEE Conference on Robotics and Automation*, 2014.

[12] L. Janson, E. Schmerling, A. Clark, and M. Pavone, "Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions," *International Journal of Robotics Research*, 2015. In Press.

[13] S. Karaman and E. Frazzoli, "Sampling-based Algorithms for Optimal Motion Planning," *International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[14] T. Asano, T. Asano, L. J. Guibas, J. Hershberger, and H. Imai, "Visibility of Disjoint Polygons," *Algorithmica*, vol. 1, no. 1, pp. 49–63, 1986.

[15] S. K. Ghosh and D. M. Mount, "An Output-sensitive Algorithm for Computing Visibility Graphs," *SIAM Journal on Computing*, vol. 20, no. 5, pp. 888–910, 1991.

[16] B. Delaunay, "Sur la Sphère Vide," *Bulletin of Academy of Sciences of the USSR*, vol. 7, pp. 793–800, 1934.

[17] D. Attali, J.-D. Boissonnat, and A. Lieutier, "Complexity of the Delaunay Triangulation of Points on Surfaces: the Smooth Case," in *Symposium on Computational Geometry*, pp. 201–210, 2003.

[18] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[19] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, pp. 100–107, july 1968.

[20] A. Stentz, "The Focussed D* Algorithm for Real-time Replanning," in *International Joint Conference on Artificial Intelligence*, 1995.

[21] S. Koenig and M. Likhachev, "D* Lite," in *AAAI Conference of Artificial Intelligence*, 2002.

[22] J. A. Sethian, "A Fast Marching Level Set Method for Monotonically Advancing Fronts," *Proceedings of the National Academy of Sciences*, vol. 93, no. 4, pp. 1591–1595, 1996.

[23] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[24] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001.

[25] L. E. Kavraki, P. Švestka, J. C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[26] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[27] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Batch informed trees (bit*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs," Feb. 2015. Available at `http://arxiv.org/abs/1405.5848`.

[28] O. Arslan and P. Tsiotras, "Use of relaxation methods in sampling-based algorithms for optimal motion planning," in *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 2421–2428, 2013.

[29] J. Barranquand, B. Langlois, and J. C. Latombe, "Numerical Potential Field Techniques for Robot Path Planning," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 22, no. 2, pp. 224–241, 1992.

[30] DARPA, "Urban Challenge, last visit: November, 2012," 2011.

[31] S. Tadokoro, M. Hayashi, Y. Manabe, Y. Nakami, and T. Takamori, "On Motion lanning of Mobile Robots which Coexist and Cooperate with Humans ," in *IEEE/RSJ International Conference on Intelligent Robots & Systems*, vol. 2, pp. 518–523, 1995.

[32] N. Roy, G. J. Gordon, and S. Thrun, "Planning under Uncertainty for Reliable Health Care Robotics," in *International Conference on Field and Service Robots*, vol. 24, pp. 417–426, 2003.

[33] P. Fiorini and Z. Shiller, "Motion Planning in Dynamic Environments Using Velocity Obstacles ," *International Journal of Robotics Research*, vol. 17, no. 7, pp. 760–772, 1998.

[34] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion*. MIT Press, 2007.

[35] B. Xu, D. J. Stilwell, and A. Kurdila, "A Receding Horizon Controller for Motion Planning in the Presence of Moving Obstacles," in *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 974–980, 2010.

[36] N. E. Du Toit and J. W. Burdick, "Robot Motion Planning in Dynamic, Uncertain Environments," *IEEE Transactions on Robotics*, vol. 28, no. 1, pp. 101–115, 2012.

[37] A. Ettlin and H. Bleuler, "Randomised Rough-Terrain Robot Motion Planning," in *IEEE/RSJ International Conference on Intelligent Robots & Systems*, pp. 5798–5803, IEEE, 2006.

[38] L. Jaillet, J. Cortés, and T. Siméon, "Sampling-based Path Planning on Configuration-Space Costmaps," *IEEE Transactions on Robotics*, vol. 26, no. 4, pp. 635–646, 2010.

[39] J. Cortes, L. Jaillet, and T. Siméon, "Disassembly Path Planning for Complex Articulated Objects," *IEEE Transactions on Robotics*, vol. 24, no. 2, pp. 475–481, 2008.

[40] R. Iehl, J. Cortés, and T. Siméon, "Costmap Planning in High Dimensional Configuration Spaces ," in *IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pp. 166–172, 2012.

[41] A. C. Shkolnik and R. Tedrake, "Path Planning in 1000+ Dimensions Using a Task-space Voronoi Bias," in *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 2061–2067, 2009.

[42] I. S. Godage, D. Branson, E. Guglielmino, and D. G. Caldwell, "Path Planning for Multisection Continuum Arms ," in *IEEE International Conference on Mechatronics and Automation*, pp. 1208–1213, 2012.

[43] D. Michel and K. McIsaac, "New Path Planning Scheme for Complete Coverage of Mapped Areas by Single and Multiple Robots ," in *IEEE International Conference on Mechatronics and Automation*, pp. 1233–1240, 2012.

[44] J. D. Gammell, S. S. S., and T. D. Barfoot, "Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic," Nov. 2014. Available at `http://arxiv.org/abs/1404.2334`.

[45] M. W. Otte and N. Correll, "C-FOREST: Parallel Shortest Path Planning with Superlinear Speedup," *IEEE Transactions on Robotics*, vol. 29, no. 3, pp. 798–806, 2013.

[46] I. A. Sucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics and Automation Magazine*, vol. 19, no. 4, pp. 72–82, 2012.

[47] M. Moll, I. A. Sucan, and L. E. Kavraki, "An Extensible Benchmarking Infrastructure for Motion Planning Algorithms," *IEEE Robotics and Automation Magazine, submitted*, 2015.

[48] D. J. Webb and J. van den Berg, "Kinodynamic RRT*: Asymptotically Optimal Motion Planning for Robots with Linear Dynamics.," in *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 5054–5061, 2013.

[49] A. Perez, R. Platt, G. Konidaris, L. Kaelbling, and T. Lozano-Perez, "LQR-RRT*: Optimal Sampling-based Motion Planning with Automatically Derived Extension Heuristics," in *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 2537–2542, IEEE, 2012.

[50] O. Salzman and D. Halperin, "Asymptotically-Optimal Motion Planning using Lower Bounds on Cost," Sept. 2014.

[51] E. Schmerling, L. Janson, and M. Pavone, "Optimal Sampling-Based Motion Planning under Differential Constraints: the Driftless Case," Mar. 2014.

[52] E. Schmerling, L. Janson, and M. Pavone, "Optimal Sampling-Based Motion Planning under Differential Constraints: the Drift Case with Linear Affine Dynamics," May 2014.

[53] D. Yershov, M. Otte, and E. Frazzolli, "Planning for optimal feedback control in the volume of free space," Apr. 2015. Available at `http://arxiv.org/abs/1504.07940`.

[54] M. Elbanhawi and M. Simic, "Sampling-based robot motion plannin: a review," *IEEE Access*, vol. 2, pp. 56–77, 2014.

[55] S. Garrido, L. Moreno, M. Abderrahim, and F. Martin, "Path Planning for Mobile Robot Navigation Using Voronoi Diagram and Fast Marching," in *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 2376–2381, 2006.

[56] S. Garrido, L. Moreno, D. Blanco, and M. L. Muñoz, "Sensor-based Global Planning for Mobile Robot Navigation," *Robotica*, vol. 25, no. 2, pp. 189–199, 2007.

[57] S. Garrido, L. Moreno, and D. Blanco, "Exploration of a Cluttered Environment using Voronoi Transform and Fast Marching Method," *Robotics and Autonomous Systems*, vol. 56, no. 12, pp. 1069–1081, 2008.

[58] S. Garrido, L. Moreno, and P. U. Lima, "Robot Formations Motion Planning using Fast Marching," *Robotics and Autonomous Systems*, vol. 59, no. 9, pp. 675–683, 2011.

[59] S. Garrido, L. Moreno, M. Abderrahim, and D. Blanco, "FM2: A Real-time Sensor-based Feedback Controller for Mobile Robots," *IEEE Transactions on Automatic Control*, vol. 24, no. 1, pp. 3169–3192, 2009.

[60] S. Garrido, L. Moreno, D. Blanco, and F. Martin, "Smooth Path Planning for Non-holonomic Robots using Fast Marching," in *Proceedings of the 2009 IEEE International Conference on Mechatronics*, pp. 1–6, 2009.

[61] C. Arismendi, D. Álvarez, S. Garrido, and L. Moreno, "Nonholonomic Motion Planning using Fast Marching Squared Method," *International Journal of Advanced Robotics*, vol. 12, no. 56, pp. 1–10, 2015.

[62] S. Garrido, L. Moreno, and D. Blanco, "Exploration and Mapping Using the VFM Motion Planner," *IEEE Transactions on Instrumentation and Measurement*, vol. 58, no. 8, pp. 2880–2892, 2009.

[63] S. Garrido, M. Malfaz, and D. Blanco, "Application of the Fast Marching Method for Outdoor Motion Planning in Robotics," *Robotics and Autonomous Systems*, vol. 61, no. 2, pp. 106–114, 2012.

[64] L. D. Sanctis, "Determinación de Trayectorias Óptimas Sobre Superficies Tridimensionales Utilizando Fast Marching," Master's thesis, Carlos III University of Madrid, Nov. 2010.

[65] P. Jurewicz, "Smooth Path-planning for a Robot Manipulator Using Probabilistic Methods," Master's thesis, Carlos III University of Madrid, Nov. 2010.

[66] S. Garrido, "Robot Planning and Exploration Using Fast Marching," Master's thesis, Carlos III University of Madrid, Dec. 2008.

[67] C. Petres, Y. Pailhas, P. Patron, Y. Petillot, J. Evans, and D. Lane, "Planning for Autonomous Underwater Vehicles," *IEEE Transactions on Robotics*, vol. 32, no. 2, pp. 331–341, 2007.

[68] Y. Liu, W. Liu, R. Song, and R. Bucknall, "Predictive Navigation of Unmanned Surface Vehicles in a Dynamic Maritime Environment when using the Fast Marching Method," *International Journal of Adaptive Control and Signal Processing*, pp. 1099–1115, 2015.

[69] J. N. Tsitsiklis, "Efficient Algorithms for Globally Optimal Trajectories," *IEEE Transactions on Automatic Control*, vol. 40, no. 9, pp. 1528–1538, 1995.

[70] J. V. Gómez, "Advanced Applications of the Fast Marching Square Planning Method," Master's thesis, Carlos III University, 2012.

[71] Q. Do, S. Mita, and K. Yoneda, "Narrow passage path planning using fast marching method and support vector machine," in *IEEE Intelligent Vehicles Symposium Proceedings*, pp. 630–635, 2014.

[72] N. Forcadel, C. Le Guyader, and C. Gout, "Generalized fast marching method: Applications to image segmentation," *Numerical Algorithms*, vol. 48, no. 1–3, pp. 189–211, 2008.

[73] S. Basu and D. Racoceanu, "Reconstructing neuronal morphology from microscopy stacks using fast marching," in *IEEE International Conference on Image Processing*, pp. 3597–3601, 2014.

[74] N. Al Zaben, N. Madusanka, A. Al Shdefat, and H. Choi, "Comparison of active contour and fast marching methods of hippocampus segmentation," in *6th International Conference on Information and Communication Systems*, pp. 106–110, 2015.

[75] X. Qu, S. Liu, and F. Wang, "A new ray tracing technique for crosshole radar traveltime tomography based on multistencils fast marching method and the steepest descend method," in *15th International Conference on Ground Penetrating Radar*, pp. 503–508, 2014.

[76] X. Zhang and R. Bording, "Fast marching method seismic traveltimes with reconfigurable field programmable gate arrays," *Canadian Journal of Exploration Geophysics*, vol. 36, no. 1, pp. 60–68, 2011.

[77] J. A. Sethian and A. Vladimirsky, "Ordered upwind methods for static Hamilton-Jacobi equations: theory and algorithms," *SIAM Journal on Numerical Analysis*, vol. 41, no. 1, pp. 325–363, 2003.

[78] A. Chacon, *Eikonal Equations: New Two-scale Algorithms and Error Analysis*. PhD thesis, Cornell Univeristy, 1 2014.

[79] S. Osher and J. A. Sethian, "Fronts Propagating with Curvature Dependent Speed: Algorithms based on Hamilton-Jacobi Formulations," *Journal of Computational Physics*, vol. 79, no. 1, pp. 12–49, 1988.

[80] J. A. Sethian, "Fast Marching Methods," *SIAM Review*, vol. 41, no. 2, pp. 199–235, 1999.

[81] R. Kimmel and J. A. Sethian, "Computing Geodesic Paths on Manifolds," *Proceedings of the National Academy of Sciences*, vol. 95, no. 15, pp. 8431–8435, 1998.

[82] J. A. Sethian and A. Vladimirsky, "Fast Methods for the Eikonal and related Hamilton-Jacobi equations on unstructured meshes," *Proceedings of the National Academy of Sciences*, vol. 97, no. 11, pp. 5699–5703, 2000.

[83] S. Ahmed, S. Bak, J. McLaughlin, and D. Renzi, "A Third Order Accurate Fast Marching Method for the Eikonal Equation in Two Dimensions," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2402–2420, 2011.

[84] S. Luo, "High-order Factorizations and High-order Schemes for Point-source Eikonal Equations," *SIAM Journal on Numerical Analysis*, vol. 52, no. 1, pp. 23–44, 2014.

[85] J. Sethian, *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. No. 3, Cambridge University Press, 1999.

[86] R. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.

[87] I. Arvanitakis, A. Tzes, and M. Thanou, "Geodesic Motion Planning on 3D-terrains Satisfying the Robots's Kinodynamic Constraints," in *Annual Conference of the IEEE Industrial Electronics Society*, pp. 4144–4149, 2013.

[88] D. Ogay and E. Kim, "Kinodynamic motion planning with artificial wavefront propagation," *Journal of Information and Communications Convergence Engineering*, vol. 11, no. 4, pp. 274–281, 2013.

[89] S. Cacace, E. Cristiani, and M. Falcone, "Can local single-pass methods solve any stationary hamilton-jacobi-bellman equation?," *SIAM Journal on Scientific Computing*, vol. 36, no. 2, pp. 570–587, 2014.

[90] A. K. Jain, L. Hong, and S. Pankanti, "Random insertion into a priority queue structure," tech. rep., Stanford Univeristy Reports, 1974.

[91] M. Fredman and R. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *Journal of the Association for Computing Machinery*, vol. 34, no. 3, pp. 596–615, 1987.

[92] C. Rasch and T. Satzger, "Remarks on the o(n) implementation of the fast marching method," 2008.

[93] Y. Tsai, L. Cheng, S. Osher, and H. Zhao, "Fast sweeping algorithms for a class of hamilton-jacobi equations," *SIAM Journal on Numerical Analysis*, vol. 41, no. 2, pp. 659–672, 2003.

[94] H. Zhao, "Fast sweeping method for eikonal equations," *Mathematics of Computation*, vol. 74, pp. 603–627, 2005.

[95] S. Kim, "An o(n) level set method for eikonal equations," *SIAM Journal on Scientific Computing*, vol. 22, no. 6, pp. 2178–2193, 2001.

[96] E. Rouy and A. Tourin, "A viscosity solutions approach to shape-from shading," *SIAM Journal on Numerical Analysis*, vol. 29, pp. 867–884, 1992.

[97] Z. Fu, W. Jeong, Y. Pan, and R. Whitaker, "A fast iterative method for solving the eikonal equation on triangulated surfaces," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2468–2488, 2011.

[98] A. Valero-Gomez, J. Gomez, S. Garrido, and L. Moreno, "The Path to Efficiency: Fast Marching Method for Safer, More Efficient Mobile Robot Trajectories," *IEEE Robotics and Automation Magazine*, vol. 20, no. 4, pp. 111–120, 2013.

[99] P. Clement, Y. Pailhas, P. Patron, Y. Petillot, J. Evans, and D. Lane, "Path planning for autonomous underwater vehicles," *IEEE Transactions on Robotics*, vol. 23, no. 2, pp. 331–341, 2007.

[100] M. Detrixhe, F. Gibou, and C. Min, "A parallel fast sweeping method for the eikonal equation," *Journal of Computational Physics*, vol. 237, pp. 46–55, 2013.

[101] R. Siegwart and I. R. Nourbakhsh, *Introduction to Autonomous Mobile Robots*. Scituate, MA, USA: Bradford Company, 2004.

[102] J. Pardeiro, "Algoritmos de Planificación de Trayectorias basados en Fast Marching Square," Master's thesis, Carlos III University, 2015.

[103] A. Tesini and J. Palmer, "The ITER Remote Maintenance System," *Fusion Engineering and Design*, vol. 83, pp. 7–9, 2008.

[104] I. Ribeiro, P. U. Lima, P. Aparício, and R. Ferreira, "Conceptual Study on Flexible Guidance and Navigation for ITER Remote Handling Transport Casks," in *IEEE/NPSS Symposium on Fusion Engineering*, pp. 969–972, 1997.

[105] I. Ribeiro, P. Lima, P. Aparício, and R. Ferreira, "Active Docking of a Transport Cask Vehicle Subject to 6 Degrees of Freedom Misalignments," in *Proceedings of the 17th IEEE/NPSS Symposium on Fusion Engineering*, pp. 973–976, Oct. 1997.

[106] G. Dudek and M. Jenkin, "Computational Principles of Mobile Robotics," *Cambridge University Press (2 edition)*, 2010.

[107] B. R. Sarker and S. S. Gurav, "Route Planning for Automated Guided Vehicles in a Manufacturing Facility," *International Journal of Production Research*, vol. 43, no. 21, pp. 4659–4683, 2005.

[108] H. Martínez-Barberá and D. Herrero-Pérez, "Autonomous Navigation of an Automated Guided Vehicle in Industrial Environments," *Robotics and Computer-Integrated Manufacturing*, vol. 26, no. 4, pp. 296–311, 2010.

[109] B. Trebilcock, "Automatic Guided Vehicle Basics," *Modern Materials Handling*, vol. 62, no. 12, pp. 46–50, 2007.

[110] J. V. Gómez, A. Vale, F. Valente, J. Ferreira, S. Garrido, and L. Moreno, "Fast Marching in Motion Planning for Rhombic like Vehicles Operating in ITER," in *Proceedings of the IEEE Conference on Robotics and Automation*, 2013.

[111] Q. H. Do, S. Mita, H. T. Niknejad, and L. Han, "Dynamic and safe path planning based on support vector machine among multi moving obstacles for autonomous vehicles," *IEICE Transactions on Information Systems*, vol. 96-D, no. 2, pp. 314–328, 2013.

[112] R. Takei and R. Tsai, "Optimal trajectories of curvature constrained motion in the hamilton-jacobi formulation," *Journal on Scientific Computing*, vol. 54, no. 2-3, pp. 622–644, 2013.

[113] J. Ferreira, A. Vale, and I. Ribeiro, "Localization of Cask and Plug Remote Handling System in ITER using multiple Video Cameras," *Fusion Engineering and Design*, vol. 88, pp. 1992–1996, 2013.

[114] D. Locke, C.-G. Gutiéerrez, C. Damiani, J.-P. Friconneau, and J.-P. Martins, "Progress in the Conceptual Design of the ITER Cask and Plug Remote Handling System," *Fusion Engineering and Design*, vol. 89, pp. 2419–2424, 2014.

[115] D. Wang and F. Qi, "Trajectory Planning for a Four-wheel-steering Vehicle," in *Proceedings of the IEEE Conference on Robotics and Automation*, vol. 4, pp. 3320–3325, 2001.

[116] A. Vale, D. Fonte, F. Valente, and I. Ribeiro, "Trajectory Optimization for Autonomous Mobile Robots in ITER," *Robotics and Autonomous Systems*, vol. 62, pp. 871–888, 2014.

[117] S. Quinlan and O. Khatib, "Elastic Bands: Connecting Path Planning and Control," in *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 2, pp. 802–807, May 1993.

[118] A. Vale, D. Fonte, F. Valente, J. Ferreira, I. Ribeiro, and C. Gonzalez, "Flexible Path Optimization for the Cask and Plug Remote Handling System in ITER," *Fusion Engineering and Design*, vol. 88, pp. 1900–1903, 2013.

[119] M. Kass, A. Witkin, and D. Terzopoulos, "Snakes: Active Contour Models," *International Journal of Computer Vision*, vol. 1, no. 4, pp. 321–331, 1988.

[120] F. Kang and S. Zhong-Ci, *Mathematical Theory of Elastic Structures*. Springer New York, 1981.

[121] F. P. Beer, E. R. Johnston, and J. T. DeWolf, *Mechanics of Materials*. McGraw Hill, 2002.

[122] P. Ruibanys, C. Reig, E. Gazeau, J. Marmie, and N. Etchegoin, "Definition, Development and Operation of a Comprehensive Virtual Model of the ITER Buildings, ATS and TCS," in *26th Symposium on Fusion Technology*, 2010.

[123] A. Vale and I. Ribeiro, *Motion Planning of Large Scale Vehicles for Remote Material Transportation*, ch. 12, pp. 249–292. Springer Verlag, 2015.

[124] B. Nemec, M. Zorko, and L. Zlajpah, "Learning of a Ball-in-a-cup Playing Robot," in *International Workshop on Robotics in Alpe-Adria-Danube Region*, pp. 297–301, June 2010.

[125] K. Mülling, J. Kober, O. Kroemer, and J. Peters, "Learning to Select and Generalize Striking Movements in Robot Table Tennis," *International Journal of Robotics Research*, vol. 32, no. 31, pp. 263–279, 2013.

[126] S. Schaal, J. Peters, J. Nakanishi, and A. Ijspeert, "Learning Movement Primitives," in *International Symposium on Robotics Research*, vol. 15 of *Springer Tracts in Advanced Robotics*, pp. 561–572, 2003.

[127] P. Pastor, H. Hoffmann, T. Asfour, and S. Schaal, "Learning and Generalization of Motor Skills by Learning from Demonstration," in *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 1293–1298, 2009.

[128] K. Ning, M. Tamosiunaite, and F. Worgotter, "A Novel Trajectory Generation Method for Robot Control," *Journal of Intelligent and Robotics Systems*, vol. 68, no. 2, pp. 165–184, 2012.

[129] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement Learning in Robotics: A Survey," *International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[130] D. Lee and C. Ott, "Incremental Motion Primitive Learning by Physical Coaching using Impedance Control," in *IEEE/RSJ International Conference on Intelligent Robots & Systems*, pp. 4133–4140, 2010.

[131] S. M. Khansari-Zadeh and A. Billard, "Learning Stable Nonlinear Dynamical Systems with Gaussian Mixture Models," *IEEE Transactions on Robotics*, vol. 27, no. 5, pp. 943–957, 2011.

[132] S. Calinon, I. Sardellitti, and D. G. Caldwell, "Learning-based Control Strategy for Safe Human-robot Interaction Exploiting Task and Robot Redundancies," in *IEEE/RSJ International Conference on Intelligent Robots & Systems*, pp. 249–254, 2010.

[133] M. Hersch, F. Guenter, S. Calinon, and A. Billard, "Dynamical System Modulation for Robot Learning via Kinesthetic Demonstrations," *IEEE Transactions on Robotics*, vol. 24, no. 6, pp. 1463–1467, 2008.

[134] S. M. Khansari-Zadeh and A. Billard, "Imitation Learning of Globally Stable Non-linear Point-to-point Robot Motions using Nonlinear Programming," in *IEEE/RSJ International Conference on Intelligent Robots & Systems*, pp. 2676–2683, 2010.

[135] C. Rasmussen and C. Williams, *Gaussian Processes for Machine Learning*. New York, Springer-Verlag, 3rd ed., 2006.

[136] S. Schaal, C. Atkenson, and S. Vijayakumar, "Scalable Locally Weighted Statistical Techniques for Real-time Robot Learning," *Applied Intelligence*, vol. 17, no. 1, pp. 49–60, 2002.

[137] S. Martin, S. Wright, and J. Sheppard, "Offline and Online Evolutionary Bi-directional RRT Algorithms for Efficient Replanning in Dynamic Environments," in *IEEE International Conference on Automation Science and Engineering*, pp. 1131–1136, 2008.

[138] M. Branicky, R. Knepper, and J. Kuffner, "Path and Trajectory Diversity: Theory and Algorithms," in *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 1359–1364, 2008.

[139] N. Jetchev and M. Toussant, "Trajectory Prediction in Cluttered Voxel Environments," in *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 2523–2528, 2010.

[140] N. Jetchev and M. Toussant, "Fast Motion Planning from Experience: Trajectory Prediction for Speeding up Movement Generation," *Autonomous Robots*, vol. 34, no. 1-2, pp. 111–127, 2013.

[141] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A Survey of Robot Learning from Demonstration," *Robotics and Autonomous Systems*, vol. 57, no. 31, pp. 469–483, 2009.

[142] S. M. Khansari-Zadeh and A. Billard, "A Dynamical System Approach to Realtime Obstacle Avoidance," *Autonomous Robots*, vol. 32, no. 4, pp. 433–454, 2012.

[143] F.-F. Li, R. Fergus, and P. Perona, "One-Shot Learning of Object Categories," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, no. 4, pp. 594–611, 2006.

[144] J. Slotine and W. Li, *Applied Nonlinear Control*. Prentice Hall, 1991.

[145] A. Lemme, K. Neumann, R. Reinhart, and J. Steil, "Neural Learning of Vector Fields for Encoding Stable Dynamical Systems," *Neurocomputing*, vol. 141, pp. 3–14, 2014.

[146] J. A. Starek, J. V. Gómez, E. Schmerling, L. Janson, L. Moreno, and M. Pavone, "An Asymptotically-Optimal Sampling-Based Algorithm for Bi-directional Motion Planning," in *IEEE/RSJ International Conference on Intelligent Robots & Systems*, 2015.

[147] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," *International Journal of Computational Geometry & Applications*, vol. 9, no. 4, pp. 495–512, 1999.

[148] J. M. Phillips, N. Bedrossian, and L. E. Kavraki, "Guided expansive spaces trees: A search strategy for motion- and cost-constrained state spaces," in *Proceedings of the IEEE Conference on Robotics and Automation*, vol. 4, pp. 3968–3973, 2004.

[149] J. Luo and K. Hauser, "An empirical study of optimal motion planning," in *IEEE/RSJ International Conference on Intelligent Robots & Systems*, pp. 1761–1768, Sept. 2014.

[150] L. Janson and M. Pavone, "Fast Marching Trees: A fast marching sampling-based method for optimal motion planning in many dimensions," in *International Symposium on Robotics Research*, 2013.

[151] Y. K. K. Hwang and N. Ahuja, "Gross Motion Planning – a Survey," *ACM Compututing Surveys*, vol. 24, pp. 219–291, Sept. 1992.

[152] I. Pohl, *Bi-directional and Heuristic Search in Path Problems*. No. 104, Department of Computer Science, Stanford University., 1969.

[153] M. Luby and P. Ragde, "A bidirectional shortest-path algorithm with good average-case behavior," *Algorithmica*, vol. 4, no. 1-4, pp. 551–567, 1989.

[154] A. Goldberg, H. Kaplan, and R. Werneck, *Reach for A*: Efficient Point-to-Point Shortest Path Algorithms*, ch. 12, pp. 129–143. Springer, 2006.

[155] J. J. Kuffner and S. M. LaValle, "RRT-Connect: An Efficient Approach to Single-Query Path Planning.," in *Proceedings of the IEEE Conference on Robotics and Automation*, vol. 2, pp. 995–1001, 2000.

[156] G. Sánchez and J.-C. Latombe, "A single-query bi-directional probabilistic roadmap planner with lazy collision checking," in *Robotics Research*, vol. 6 of *Springer Tracts in Advanced Robotics*, pp. 403–417, Springer Berlin Heidelberg, 2003.

[157] B. Akgun and M. Stilman, "Sampling heuristics for optimal motion planning in high dimensions," in *IEEE/RSJ International Conference on Intelligent Robots & Systems*, pp. 2640–2645, IEEE, 2011.

[158] M. Jordan and A. Perez, "Optimal Bidirectional Rapidly-Exploring Random Trees." `http://people.csail.mit.edu/aperez/obirrt/csailtech.pdf`, Aug. 2013.

[159] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics and Automation Magazine*, vol. 19, pp. 72–82, Dec. 2012.