IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 20, 2018, accepted February 19, 2019, date of publication March 22, 2019, date of current version April 5, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2906782

Fast Methods for Eikonal Equations: An Experimental Survey

J. V. GÓMEZ, D. ÁLVAREZ, S. GARRIDO, (Member, IEEE), AND L. MORENO, (Member, IEEE)
Robotics Laboratory, University Carlos III of Madrid, 28911 Madrid, Spain

Corresponding author: D. Álvarez (dasanche@ing.uc3m.es)

This work is funded by the projects: "RoboCity2030-DIH-CM Madrid Robotics Digital Innovation Hub (Robtica aplicada a la mejora de la calidad de vida de los ciudadanos. Fase IV; S2018/NMT-4331), funded by Programas de Actividades I+D en la Comunidad de Madrid and cofunded by Structural Funds of the EU," and "Investigación para la mejora competitiva del ciclo de perforación y voladura en minerıa y obras subterráneas, mediante la concepción de nuevas técnicas de ingenierıa , explosivos, prototipos y herramientas avanzadas (TUÑEL)."

ABSTRACT Fast methods are very popular algorithms to compute time-of-arrival maps (distance maps measured in time units) solving the Eikonal equation. Since fast marching was proposed in 1995, it has been applied to many different applications, such as robotics, medical computer vision, fluid simulation, and so on. From then on, many alternatives to the original method have been proposed with two main objectives: reducing its computational time and improving its accuracy. In this paper, we collect the main single-threaded approaches, which improve the computational time of the standard fast marching method and study them within a common mathematical framework. Then, they are evaluated using isotropic environments, which are representative of their possible applications. The studied methods are the fast marching method with the binary heap, the fast marching method with Fibonacci heap, the simplified fast marching method, the untidy fast marching method, the fast iterative method, the group marching method, the fast sweeping method, the locking sweeping method, and the double dynamic queue method.

INDEX TERMS Eikonal equation, fast methods, fast marching method, fast sweeping method.

I. INTRODUCTION

The Fast Marching Method (FMM) has been extensively applied since it was first proposed in 1995 [1] as a solution to isotropic control problems using first-order semi-Langragian discretizations on Cartesian grids. The first approach was introduced by Tsitsiklis [1], but the most popular solution was given a few months later by Sethian [2] using first-order upwind finite differences in the context of isotropic front propagation. The differences and similarities between both works can be found in [3]. The Fast Sweeping Method (FSM) is a more modern iterative algorithm which uses Gauss-Seidel iterations with alternating sweeping ordering to also solve a discretized Eikonal equation on a rectangular grid [4]. As long as the same first-order upwind discretization is used in both methods, the computed solution with the Fast Sweeping Method is exactly the same as the one given by the Fast Marching Method.

These two different methods, commonly named as Fast Methods [5], [6], were originally suggested 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 Methods, we refer the interested readers to [7].

One of the reasons for the popularity of the Fast Methods is that they can be applied in many different fields, such us: path planning in robotics [8]–[12]; image segmentation [13]–[15], shape and surface recognition and segmentation [16], [17], volumetric data representation [18] or quantification of lesions in contrast-enhanced tomography [19] in computer vision; traveltime computation in geophysical applications such as tomography [20]–[22] or seismology [23], [24]. Despite the vast amount of literature on Fast Methods, there is a lack of in-depth comparison and benchmarking among the proposed methods.

In this paper, nine sequential (mono-thread), isotropic, grid-based Fast Methods are detailed in the following sections: 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), Locking Sweeping Method (LSM) and Double Dynamic Queue Method (DDQM). All these algorithms provide

The associate editor coordinating the review of this manuscript and approving it for publication was Bora Onat.

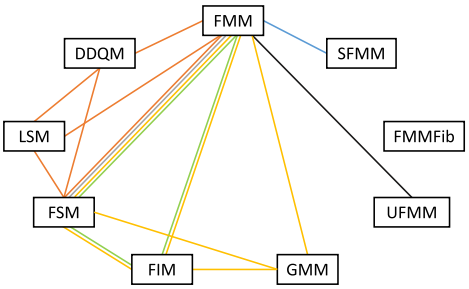

FIGURE 1. Comparisons among algorithms. Colors refer to different works: orange [27], gray [5], yellow [28], green [26], black [29], and blue [25].

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 because of the lack of comparison. For example, [5] compares only FMM and FSM in spite of the fact that GMM and UFMM had already been published. Survey [25] mentions most of the algorithms but only compares FMM and SFMM. A more recent work compares FMM, FSM and FIM in 2D [26]. However, FIM was parallelized and implemented in CUDA providing a biased comparison. Fig. 1 schematically shows the comparisons between algorithms carried out in the literature. As an example, UFMM has barely been compared to its counterparts whereas FMM and FSM are compared in many papers despite the fact that it is well known when each of them performs better: FSM is faster in simple environments with constant speed. In addition, results from one work cannot be directly extrapolated to other works since the performance of these methods highly depends on their implementation.

Statement of Contributions: Three main contributions are included in this paper: 1) Based on previous publications, a common formulation and notation are given for all the algorithms, presented in Section II. This way, it is possible to easily understand their working principles and mathematical formulation. 2) A survey of the research on designing n-dimensional sequential Fast Methods is explained along Sections III, IV and V. 3) Extensive and systematic comparison of the mentioned methods is carried out and explained in Section VI, followed by a discussion in Section VII. The experiments are designed to take into account their possible applications and the results previously reported.

OTHER ALGORITHMS NOT INCLUDED IN THE SURVEY

There are some variations of these approaches which focus on improving some of the characteristics of the solution given by the fast methods. For example, in order to enhance the computation time, parallel approaches have been proposed of both the Fast Marching [30] and Fast Sweeping [31] methods. In [32] the Heat Method, used for computing the geodesic distances in near-linear time, was introduced. Although it outperforms the presented methods in terms of computation time, it only works with constant speed functions, so it does not solve the problems analyzed in the experimental section. Additionally, it is obvious that the accuracy of the computed solution for these methods depends on the chosen grid size, however, higher order approaches [33], [34] are able to improve the accuracy using the same grid at the cost of more computation time. For applications in which a high accuracy solution of the Eikonal equation at the source point is needed, the factored Eikonal equation leads to much more accurate solutions by analytically handling the source singularity [35], [36].

Different two-scale methods are proposed in [7]: 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) the performance of HCM and FHCM depends on the discretization of the coarse grid, where the optimal parameter depends on the speed profile. Furthermore, FHCM includes additional error. 2) the FMSM error is not mathematically bounded. Thus, the comparison with other Fast Methods becomes more complex. 3) they suppose that the speed is almost constant on domains of arbitrary size [7], although this is not a restriction for the actual speed function, it is a strong assumption for some of the designed experiments.

Additionally, the single-pass methods suggested in [37] have not been included in this survey because, as the authors conclude, it is not always possible to know in advance which method, among those presented, should be used. This is an important drawback for practical applications such as robotics.

II. PROBLEM FORMULATION

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

$$\begin{aligned} |\nabla T(\mathbf{x})| F(\mathbf{x}) &= 1, \quad \Omega \subset \mathbb{R}^N \\ T(\mathbf{x}) &= 0, \quad \mathbf{x} \text{ in } \mathcal{X}_s \end{aligned} \tag{1}$$

Once solved, $T(\mathbf{x})$ represents a distances (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 speed on $F(\mathbf{x})$.

We assume, without loss of generality, that the domain is a unit hypercube of N dimensions: $\Omega = [0, 1]^N$. The domain is represented with a rectangular Cartesian 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. We refer to grid points $\mathbf{x}_{ij} = (x_i, y_j)$, $\mathbf{x}_{ij} \in \mathcal{X}$ as the point $\mathbf{x} = (x, y)$ in the space corresponding to a cell (i, j) of the grid (for the 2D case). For simplicity of

[1] This problem formulation closely follows [38]

notation, we will denote $T_{ij} = T(\mathbf{x}_{ij}) \approx T(\mathbf{x})$, $T_{ij} \in \mathcal{T}$, that is, T_{ij} represents an approximation to the real value of the function $T(\mathbf{x})$. Analogously, $F_{ij} = F(\mathbf{x}_{ij}) \approx F(\mathbf{x})$, $F_{ij} \in \mathcal{F}$. We also denote the set of von Neumann (4-connectivity in 2D) neighbors of grid point $\mathbf{x}_{ij}$ by $\mathcal{N}(\mathbf{x}_{ij})$. For a general grid of N dimensions, we will refer to cells by their index (or key) i as $\mathbf{x}_i$, since a flat representation is more efficient for such a data structure.

A. N-DIMENSIONAL DISCRETE EIKONAL EQUATION

In this section the most common first-order discretization of the Eikonal equation is detailed. It is first derived in 2D for better understanding and then an n-dimensional approach is explained. The most common first-order discretization is given in [39], which uses an upwind-difference scheme to approximate partial derivatives of $T(\mathbf{x})$ ($D_{ij}^{\pm x}$ represents the one-sided partial difference operator in direction $\pm x$):

$$T_x(\mathbf{x}) \approx D_{ij}^{\pm x} T = \frac{T_{i\pm1,j} - T_{ij}}{\pm\Delta_x}$$
$$T_y(\mathbf{x}) \approx D_{ij}^{\pm y} T = \frac{T_{i,j\pm1} - T_{ij}}{\pm\Delta_y} \tag{2}$$

$$\left\{ \begin{array}{l} \max(D_{ij}^{-x}T, 0)^2 + \min(D_{ij}^{+x}T, 0)^2 + \\ \max(D_{ij}^{-y}T, 0)^2 + \min(D_{ij}^{+y}T, 0)^2 \end{array} \right\} = \frac{1}{F_{ij}^2} \tag{3}$$

A simpler but less accurate solution to (3) is proposed in [40]:

$$\left\{ \begin{array}{l} \max(D_{ij}^{-x}T, -D_{ij}^{+x}T, 0)^2 + \\ \max(D_{ij}^{-y}T, -D_{ij}^{+y}T, 0)^2 \end{array} \right\} = \frac{1}{F_{ij}^2} \tag{4}$$

and Δx and Δy are the grid spacing in the x and y directions. Substituting (3) in (4) and letting

$$\begin{aligned} T &= T_{i,j} \\ T_x &= \min(T_{i-1,j}, T_{i+1,j}) \\ T_y &= \min(T_{i,j-1}, T_{i,j+1}) \end{aligned} \tag{5}$$

we can rewrite the Eikonal Equation, for a discrete 2D space as:

$$\max\left(\frac{T - T_x}{\Delta_x}, 0\right)^2 + \max\left(\frac{T - T_y}{\Delta_y}, 0\right)^2 = \frac{1}{F_{ij}^2} \tag{6}$$

Since we are assuming 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 passed over the coordinates (i, j). Therefore, (6) can be simplified as:

$$\left(\frac{T - T_x}{\Delta x}\right)^2 + \left(\frac{T - T_y}{\Delta y}\right)^2 = \frac{1}{F_{ij}^2} \tag{7}$$

Equation (7) is a regular quadratic equation of the form $aT^2 + bT + c = 0$, where:

$$\begin{aligned} a &= \Delta_x^2 + \Delta_y^2 \\ b &= -2(\Delta_y^2 T_x + \Delta_x^2 T_y) \\ c &= \Delta_y^2 T_x^2 + \Delta_x^2 T_y^2 - \frac{\Delta_x^2 \Delta_y^2}{F_{ij}^2} \end{aligned} \tag{8}$$

In order to simplify the notation for the n-dimensional case, we assume that the grid is composed of hypercubic cells, that is, $\Delta_x = \Delta_y = \Delta_z = \cdots = h$. Let us denote T_d as the generalization of T_x or T_y for dimension d, up to N dimensions. We also denote by F the propagation speed for the point with coordinates $(i, j, k, \dots)$. Operating and simplifying terms, the discretization of the Eikonal is a quadratic equation with parameters:

$$\begin{aligned} a &= N \\ b &= -2\sum_{d=1}^{N} T_d \\ c &= (\sum_{d=1}^{N} T_d^2) - \frac{h^2}{F^2} \end{aligned} \tag{9}$$

B. SOLVING THE N-D DISCRETE EIKONAL EQUATION

A solution to equation (7) is not straightforward, since there may be more unknowns than equations if the dimension is greater than 1. However, the entropy condition formulated in [41] for moving fronts yields a unique viscosity solution of the equation (7), which forces the wavefront to follow causality [40]. In other words, in order to reach a point with a higher time of arrival, it should have first traveled through neighbors of such a point with smaller values. The opposite would imply a jump in time continuity and the solutions would be erroneous.

The proposed Eikonal solution (9) does not guarantee the causality of the resulting map, since 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:

$$T = \frac{T_x + Ty}{2} + \frac{1}{2}\sqrt{\frac{2h^2}{F^2} - \left(T_x - T_y\right)^2} \tag{10}$$

called the *two-sided* update, as both parents T_x and T_y are taken into account. The solution is only accepted if $T \geq \max\left(T_x, T_y\right)$. The *upwind condition* [7] shows that:

$$T \geq \max\left(T_x, T_y\right) \Longleftrightarrow |T_x - T_y| \leq \frac{h}{F} \tag{11}$$

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

$$T = \min\left(T_x, T_y\right) + \frac{h}{F} \tag{12}$$

This is a top-down approach, in which the parents are iteratively discarded until a causal solution is found. However, generalizing (11) is complex, hence, we choose to use a bottom-up approach: (12) is solved and parents are iteratively included until the time of the next parent is higher than the current solution: $T_k > T$. This 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.). This approach has been found to be more robust when used in three or more dimensions with a negligible impact on the computational performance.

The next sections introduce the different algorithms, compared in this survey, used to solve the Eikonal equation.

Algorithm 1 Solve Eikonal Equation

```
1:  procedure SolveEikonal(x_i, T, F)
2:      a ← N
3:      for dim = 1 : N do
4:          min_T ← MinTDimdim
5:          if min_T ≠ ∞ and min_T < T_i then
6:              T_values.push(min_T)
7:          else
8:              a ← a − 1
9:          end if
10:     end for
11:     if a = 0 then                ▷ FSM can cause this situation.
12:         return ∞
13:     end if
14:     T_values ← SortT_values
15:     for dim = 1 : a do
16:         T̃_i ← SolveNDims x_i, dim, T_values, F
17:         if dim = a or T̃_i < T_values,dim+1 then
18:             break
19:         end if
20:     end for
21:     return T̃_i
22: end procedure
```

Algorithm 2 Solve Eikonal for N Dimensions

```
1:  procedure SolveNDims(x_i, dim, T_values, F)
2:      if dim = 1 then
3:          return T_values,1 + h/F_i
4:      end if
5:      sumT ← Σ_{i=1}^{dim} T_values,i
6:      sumT^2 ← Σ_{i=1}^{dim} T^2_values,i
7:      a ← dim
8:      b ← −2sumT
9:      c ← sumT^2 − h^2/F_i
10:     q ← b^2 − 4ac
11:     if q < 0 then                ▷ Noncausal solution
12:         return ∞
13:     else
14:         return (−b+sqrt(q))/2a
15:     end if
16: end procedure
```

III. FAST MARCHING METHODS

The Fast Marching Method (FMM) [2] is the most common Eikonal solver. It can be classified as a label-setting, Dijkstra-like algorithm [42]. It uses a first-order upwind finite difference scheme, which is described in detail in Section II, to simulate an isotropic front propagation computing the solution following Bellman's optimality principle [43]:

$$T_i = \min_{x_i \in \mathcal{N}(x_i)} (c_{ij} + T_j) \tag{13}$$

In other words, a node $\mathbf{x}_i$ is connected to the parent $\mathbf{x}_j$ in its neighborhood $\mathcal{N}(\mathbf{x}_i)$ which minimizes (or maximizes) the value of the function (in this case T_i) composed by the value of T_j plus the addition of the cost of *traveling* from $\mathbf{x}_j$ to $\mathbf{x}_i$, represented as c_{ij}. This discretization takes into account the spatial representation (i.e., a rectangular grid in two dimensions) and the values of all the causal upwind neighbors. This is the main difference with Dijsktra's algorithm, since Dijkstra is designed to work on graphs, assuming discrete traveling, and the value of a node $\mathbf{x}_i$ only depends on one parent $\mathbf{x}_j$.

The algorithm labels the cells in three different sets: 1) `Frozen`: those cells whose value has already been computed and will not change during new iterations, 2) `Unknown`: cells with no value assigned, to be evaluated, and 3) `Narrow` band (or just `Narrow`): the frontier between `Frozen` and `Unknown` containing those cells with a value assigned that may still improve. 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, so a more detailed discussion will be carried out in Section III-A.

The procedure to compute FMM is detailed in Algorithm 3. Initially, all points[2] in the grid belong to the `Unknown` set and have an infinite arrival time. The initial points (wave sources) are assigned a value of 0 and inserted in `Frozen` (lines 2-7). Then, the main FMM loop starts by choosing the element with minimum arrival time from `Narrow` (line 10) and 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 improves the existing one. In case the evaluated cell is in `Unknown`, it is transferred to `Narrow` (lines 11-18). Finally, the previously chosen point from `Narrow` is transferred to `Frozen` (lines 21 and 22) 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.

A. 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, whose value has been improved, 3) Top: to retrieve the element with minimum value, and 4) Pop: to remove the element with minimum value. As stated before, this is the most critical aspect of the implementation of FMM. 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.

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

Algorithm 3 Fast Marching Method

procedure FMM($\mathcal{X}, \mathcal{T}, \mathcal{F}, \mathcal{X}_s$)
Initialization:
  $\texttt{Unknown} \leftarrow \mathcal{X}, \texttt{Narrow} \leftarrow \emptyset, \texttt{Frozen} \leftarrow \emptyset$
  $T_i \leftarrow \infty \; \forall \mathbf{x}_i \in \mathcal{X}$
  for $\mathbf{x}_i \in \mathcal{X}_s$ **do**
    $T_i \leftarrow 0$
    $\texttt{Unknown} \leftarrow \texttt{Unknown} \backslash \{\mathbf{x}_i\}$
    $\texttt{Narrow} \leftarrow \texttt{Narrow} \cup \{\mathbf{x}_i\}$
  end for

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

TABLE 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	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

Among all the existing heaps, FMM is usually implemented with a binary heap [44]. However, the Fibonacci Heap [45] 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 of few elements and the performance is still far from its asymptotic behavior, the binary heap performs better. Table 1 summarizes the time complexities for these heaps.[3] Note that n is the number of cells in the map, as the worst case is to have all the cells in the heap.

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, which is the worst case scenario. Furthermore, as pointed out in [7], the method has a bad cache locality, since adjacent cells on the FMM heap have no spatial relationship and this problem becomes worse as the number of dimensions increases.

[3] http://bigocheatsheet.com/

B. SIMPLIFIED FAST MARCHING METHOD

The Simplified Fast Marching Method (SFMM) [25] is a relatively unknown variation of the standard FMM which in some cases has an impressive performance. SFMM, detailed in Algorithm 4, is a reduced version of FMM where Narrow, implemented as a simple priority queue which 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 queue. Once it is popped and inserted in Frozen, the remaining instances in the queue are simply ignored.

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

C. UNTIDY FAST MARCHING METHOD

The Untidy Fast Marching Method (UFMM) [29], [46] follows exactly the same procedure as FMM. However, a special heap structure, which reduces the computational complexity of the method to $\mathcal{O}(n)$, is used: the untidy priority queue.

This untidy priority queue is closer to a look-up table than to a tree. It assumes that the values of $\mathcal{F}$ are bounded, hence the values of $\mathcal{T}$ are also bounded. The untidy queue, depicted in Fig. 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 values of T_i. The low and high threshold values of each bucket evolve with the iterations of the algorithm, trying to maintain a 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 and Pop. Besides, as the number of buckets is smaller than the number of cells, the Increase operation is, in average, $\mathcal{O}(1)$. Therefore, the total complexity of UFMM is $\mathcal{O}(n)$. However, since elements within a bucket are not sorted (a FIFO strategy is applied in each bucket), errors are introduced into the final result. Nevertheless, it has been shown that the accumulated additional error is bounded by $\mathcal{O}(h)$, with h being the cell size, which is the same order of magnitude as in the original FMM.

Algorithm 4 Simplified Fast Marching Method

1: **procedure** SFMM($\mathcal{X}, \mathcal{T}, \mathcal{F}, \mathcal{X}_s$)
Initialization as FMM (in Algorithm 3)

Propagation:
2: **while** Narrow $\neq \emptyset$ **do**
3: $\mathbf{x}_{min} \leftarrow \arg\min_{\mathbf{x}_i \in \text{Narrow}} \{T_i\}$ ▷ Narrow top operation.
4: **if** $\mathbf{x}_{min} \in$ Frozen **then**
5: Narrow $\leftarrow$ Narrow$\backslash\{\mathbf{x}_{min}\}$
6: **else**
7: **for** $\mathbf{x}_i \in (\mathcal{N}(\mathbf{x}_{min}) \cap \mathcal{X} \backslash$ Frozen$)$ **do** ▷ All neighbors not in Frozen.
8: $\widetilde{T}_i \leftarrow$ SolveEikonal $\mathbf{x}_i, \mathcal{T}, \mathcal{F}$
9: **if** $\widetilde{T}_i < T_i$ **then** ▷ Update arrival time.
10: $T_i \leftarrow \widetilde{T}_i$
11: Narrow $\leftarrow$ Narrow $\cup \{\mathbf{x}_i\}$ ▷ Narrow push operation.
12: **end if**
13: **if** $\mathbf{x}_i \in$ Unknown **then**
14: Unknown $\leftarrow$ Unknown$\backslash\{\mathbf{x}_i\}$
15: **end if**
16: **end for**
17: Narrow $\leftarrow$ Narrow$\backslash\{\mathbf{x}_{min}\}$ ▷ Narrow pop operation.
18: Frozen $\leftarrow$ Frozen $\cup \{\mathbf{x}_{min}\}$
19: **end if**
20: **end while**
21: **return** $\mathcal{T}$
22: **end procedure**

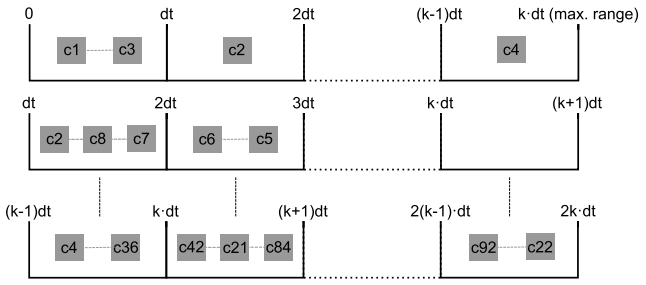

FIGURE 2. Untidy priority queue representation. Top: first iteration, the four neighbors of the initial point are pushed. Middle: the first bucket becomes empty, so the circular array advances one position. Cell c2 is first evaluated because it was the first pushed into the bucket. Bottom: after a few iterations, an entire loop on the queue is about to be completed.

IV. FAST SWEEPING METHODS

The Fast Sweeping Method (FSM) [4], [47] 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 four possible Gauss-Seidel iterations (the combinations of traversing x and y dimensions forwards and backwards): North-East, North-West, South-East and South-West, as shown in Fig. 3.

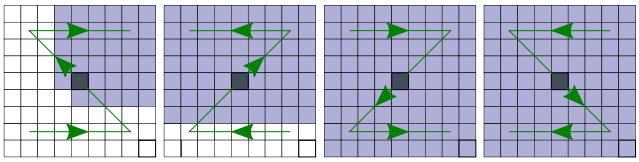

FIGURE 3. FSM sweep directions in 2D represented with arrows. The darkest cell is the initial point and the shaded cells are those analyzed by the current sweep (time improved or maintained).

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 and, up to our knowledge, there are only 2D and 3D solutions. In this survey we introduce a novel n-dimensional version, which is detailed in Algorithm 5. We will denote the sweeping directions 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 then a sweep is performed in the new direction (lines 9-10).

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:

$$\begin{aligned} &1 : [-1, -1, -1] \quad 5 : [-1, -1, 1] \\ &2 : [1, -1, -1] \quad 6 : [1, -1, 1] \\ &3 : [-1, 1, -1] \quad 7 : [-1, 1, 1] \\ &4.[1, 1, -1] \quad 8 : [1, 1, 1] \end{aligned} \tag{14}$$

Note that the literature describes at least three different sequences for the sweep pattern and shows that the optimal sequence depends on the environment [24], [47]. The sequence used in this work has been chosen to be calculated efficiently in an n-dimensional version. Besides, it is equally valid as the same directions are visited when the sweeps are done.

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. Note that the extent of dimension n is denoted by $\mathcal{X}_n$. Once the most inner loop is reached, the corresponding cell is evaluated and its value updated if necessary (lines 8-12).

The FSM carries out as many grid traversals as necessary until the value T_i for every cell has converged. Since no ordering is of the data is used, the evaluation of each cell is $\mathcal{O}(1)$. As there are n cells and t traversals, the total computational complexity of FSM is $\mathcal{O}(nt)$.

Algorithm 5 Fast Sweeping Method

```
1: procedure FSM(𝒳, 𝒯, ℱ, 𝒳_s)
   Initialization.
2:     SweepDirs ← [1, ..., 1]        ▷ Initialize sweeping directions.
3:     T_i ← ∞ ∀x_i ∈ 𝒳
4:     for x_i ∈ 𝒳_s do
5:         T_i ← 0
6:     end for

   Propagation:
7:     stop ← False
8:     while stop ≠ True do
9:         SweepDirs ← getSweepDirs 𝒳, SweepDirs
10:        stop ← Sweep 𝒳, 𝒯, ℱ, SweepDirs, N
11:    end while
12:    return 𝒯
13: end procedure
```

Algorithm 6 Sweep Directions Algorithm

```
1: procedure getSweepDirs(𝒳, SweepDirs)
2:     for i = 1 : N do
3:         SweepDirs_i ← SweepDirs_i + 2
4:         if SweepDirs_i ≤ 1 then
5:             break                  ▷ Finish For loop.
6:         else
7:             SweepDirs_i ← −1
8:         end if
9:     end for
10:    return SweepDirs
11: end procedure
```

However, the complexity constants depend greatly on the speed function $F(\mathbf{x})$. For instance, in the case of an 2D empty map with constant speed of propagation, four sweeps are enough to cover the entire map, therefore the complexity is $O(4n)$, which is the minimum possible constant (assuming the start point is not in a corner of the map). On the other hand, the more complex the speed function or the environment are, the more sweeps the algorithm will need to converge to the final solution, increasing the complexity of the method.

Note that, as long as the same first-order upwind discretization is used, the $\mathcal{T}$ returned by FSM is exactly the same as all the FMM-like algorithms (except UFMM).

A. LOCKING SWEEPING METHODS

The Locking Sweeping Method (LSM) [27] is a natural improvement over FSM. The FSM might spend time recomputing T_i of a cell even if none of its neighbors has changed their value since the last sweep, which means that the computed value T_i will be the same as in the last sweep. In order to avoid this, LSM labels each cell as *locked* or *unlocked*, and only the ones with the latter label are evaluated in each sweep.

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:         end for
7:     else
8:         for i ∈ 𝒳_1 following SweepDirs_1 do
9:             T̃_i ← SolveEikonal x_i, 𝒯, ℱ      ▷ x_i is the corresponding cell.
10:            if T̃_i < T_i then
11:                T_i ← T̃_i
12:                stop ← False
13:            end if
14:        end for
15:    end if
16:    return stop
17: end procedure
```

The LSM procedure is detailed in Algorithm 8. During the initialization, all the cells are labeled as `Frozen` (the meaning of *locked* and `Frozen` is the same). Then, the starting cells $\mathcal{X}_s$ are assigned a 0 value and all their neighbors are labeled as `Narrow` (the meaning of *unlocked* and `Narrow` is the same). Then, the wave propagation is computed performing as many grid traversals as necessary until no cell improves its time-of-arrival value. As in FSM, the GetSweepDirs() procedure is in charge of generating the appropriate Gauss-Seidel iteration directions.

For every iteration, the recursive locking sweeping algorithm, detailed in Algorithm 9, is performed. Essentially, it is the same procedure as in FSM. However, there are two main differences: 1) the Eikonal equation is computed only for those cells labeled as `Narrow`, otherwise they are skipped (see line 9), and 2) after every evaluation, if the time-of-arrival value (T_i) of cell $\mathbf{x}_i$ is improved, all neighbors of cell $\mathbf{x}_i$ which have a higher value than T_i are labeled as `Narrow` so that they are evaluated in the next iteration (lines 14-19).

Note that the asymptotic computational complexity of FSM is kept as $\mathcal{O}(n)$ and 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 time saved during the computation is important.

V. OTHER FAST METHODS

This section includes different algorithms which cannot be categorized as Fast Marching-like or Fast Sweeping methods.

A. GROUP MARCHING METHOD

The Group Marching Method (GMM) [48] is an FMM-based Eikonal solver which solves for a group of grid points in `Narrow` at once, instead of sorting them in a heap structure.

Algorithm 8 Locking Sweeping Method

1: **procedure** LSM($\mathcal{X}, \mathcal{T}, \mathcal{F}, \mathcal{X}_s$)
Initialization.
2: `Frozen` $\leftarrow \mathcal{X}$, `Narrow` $\leftarrow \emptyset$
3: $T_i \leftarrow \infty \; \forall \mathbf{x}_i \in \mathcal{X}$
4: `SweepDirs` $\leftarrow [1, \ldots, 1]$ ▷ Initialize sweeping directions.
5: **for** $\mathbf{x}_i \in \mathcal{X}_s$ **do**
6: $T_i \leftarrow 0$
7: **for** $\mathbf{x}_j \in \mathcal{N}(\mathbf{x}_i)$ **do** ▷ Unlock neighbors of starting cells.
8: `Frozen` $\leftarrow$ `Frozen`$\backslash\{\mathbf{x}_j\}$
9: `Narrow` $\leftarrow$ `Narrow` $\cup \{\mathbf{x}_j\}$
10: **end for**
11: **end for**

Propagation:
12: `stop` $\leftarrow$ `False`
13: **while** `stop` $\neq$ `True` **do**
14: `SweepDirs` $\leftarrow$ getSweepDirs $\mathcal{X}$, `SweepDirs`
15: `stop` $\leftarrow$ LockSweep $\mathcal{X}, \mathcal{T}, \mathcal{F}$, `SweepDirs`, N
16: **end while**
17: **return** $\mathcal{T}$
18: **end procedure**

Consider a front propagating, at a given time, the `Narrow` band 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. In order to select those values, GMM follows:

$$G = \{\mathbf{x}_i \in \texttt{Narrow} : T_i \leq \min(T_{\texttt{Narrow}}) + \delta_\tau\} \quad (15)$$

where

$$\delta_\tau = \frac{1}{\max(\mathcal{F})} \quad (16)$$

Although in [48], in which the GMM was presented, $\delta_\tau = \frac{h}{\max(\mathcal{F})\sqrt{N}}$ was used, we have chosen (16) as detailed in [28], since the results for the original δ_τ are much worse than FMM in most 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 done in the same way as in FMM. Then, a reverse traversal through the selected points is performed, computing and updating their value (lines 20-24). Next, in lines 28-40 a forward traversal is carried out. The operations used are the same as in the reverse traversal but updating the `Narrow` and `Frozen` sets in the same way as it is done in FMM. Then, the main loop updates the threshold T_m every iteration. The simple design of the method allows a straightforward implementation of a generalized n-dimensional solution.

Algorithm 9 Recursive Locking Sweeping Algorithm

1: **procedure** LockSweep($\mathcal{X}, \mathcal{T}, \mathcal{F}$, `SweepDirs`, n)
2: `stop` $\leftarrow$ `True`
3: **if** $n > 1$ **then**
4: **for** $i \in \mathcal{X}_n$ following `SweepDirs`$_n$ **do**
5: `stop` $\leftarrow$ LockSweep $\mathcal{X}, \mathcal{T}, \mathcal{F}$, `SweepDirs`, $n-1$
6: **end for**
7: **else**
8: **for** $i \in \mathcal{X}_1$ following `SweepDirs`$_1$ **do**
9: **if** $\mathbf{x}_i \in$ `Narrow` **then**
10: $\widetilde{T}_i \leftarrow$ SolveEikonal $\mathbf{x}_i, \mathcal{T}, \mathcal{F}$ ▷ $\mathbf{x}_i$ is the corresponding cell.
11: **if** $\widetilde{T}_i < T_i$ **then**
12: $T_i \leftarrow \widetilde{T}_i$
13: `stop` $\leftarrow$ `False`
14: **for** $\mathbf{x}_j \in \mathcal{N}(\mathbf{x}_i)$ **do**
15: **if** $T_i < T_j$ **then** ▷ Add improvable neighbors to Narrow.
16: `Frozen` $\leftarrow$ `Frozen`$\backslash\{\mathbf{x}_j\}$
17: `Narrow` $\leftarrow$ `Narrow` $\cup \{\mathbf{x}_j\}$
18: **end if**
19: **end for**
20: **end if**
21: `Narrow` $\leftarrow$ `Narrow`$\backslash\{\mathbf{x}_i\}$ ▷ Add $\mathbf{x}_i$ to Frozen.
22: `Frozen` $\leftarrow$ `Frozen` $\cup \{\mathbf{x}_i\}$
23: **end if**
24: **end for**
25: **end if**
26: **return** `stop`
27: **end procedure**

It is important to point out that that GMM returns the same solution as FMM. In GMM, every node is evaluated twice before inserting it into `Frozen`, whereas in FMM it is done only once. However, GMM does not require any sorting of the `Narrow` set, therefore, it is an $\mathcal{O}(n)$ iterative algorithm that converges in only two iterations (traversals). The value of δ_τ could be modified by the user keeping in mind that a higher δ_τ would require more iterations to converge. However, a smaller δ_τ would require also two traversals, but the group G will be composed by fewer cells. Therefore, as the authors of GMM mention, GMM can be interpreted as an intermediary point between FMM ($\delta_\tau = 0$) and a purely iterative method [49] ($\delta_\tau = \infty$).

B. DYNAMIC DOUBLE QUEUE METHOD

The Dynamic Double Queue Method (DDQM) [27] is inspired by LSM but resembles GMM. DDQM is

Algorithm 10 Group Marching Method

1: **procedure** GMM($\mathcal{X}, \mathcal{T}, \mathcal{F}, \mathcal{X}_s$)
Initialization:
2: $\texttt{Unknown} \leftarrow \mathcal{X}, \texttt{Narrow} \leftarrow \emptyset, \texttt{Frozen} \leftarrow \emptyset$
3: $T_i \leftarrow \infty \; \forall \mathbf{x}_i \in \mathcal{X}$
4: $\delta_\tau \leftarrow \frac{1}{\max(\mathcal{F})}$
5: **for** $\mathbf{x}_i \in \mathcal{X}_s$ **do**
6: $T_i \leftarrow 0$
7: $\texttt{Unknown} \leftarrow \texttt{Unknown} \backslash \{\mathbf{x}_i\}$
8: $\texttt{Frozen} \leftarrow \texttt{Frozen} \cup \{\mathbf{x}_i\}$
9: **for** $\mathbf{x}_j \in \mathcal{N}(\mathbf{x}_i)$ **do** ▷ Adding neighbors of starting points to Narrow.
10: $T_i \leftarrow \textsc{SolveEikonal}\ \mathbf{x}_j, \mathcal{T}, \mathcal{F}$
11: **if** $T_i < T_m$ **then**
12: $T_m \leftarrow T_i$
13: **end if**
14: $\texttt{Unknown} \leftarrow \texttt{Unknown} \backslash \{\mathbf{x}_i\}$
15: $\texttt{Narrow} \leftarrow \texttt{Narrow} \cup \{\mathbf{x}_i\}$
16: **end for**
17: **end for**

Propagation:
18: **while** $\texttt{Narrow} \neq \emptyset$ **do**
19: $T_m \leftarrow T_m + \delta_\tau$
20: **for** $\mathbf{x}_i \in (\texttt{Narrow} \leq T_m)$ REVERSE **do** ▷ Reverse traversal.
21: **for** $\mathbf{x}_j \in (\mathcal{N}(\mathbf{x}_i) \cap \mathcal{X} \backslash \texttt{Frozen})$ **do**
22: $\widetilde{T}_i \leftarrow \textsc{SolveEikonal}\ \mathbf{x}_j, \mathcal{T}, \mathcal{F}$
23: **if** $\widetilde{T}_i < T_i$ **then**
24: $T_i \leftarrow \widetilde{T}_i$
25: **end if**
26: **end for**
27: **end for**
28: **for** $\mathbf{x}_i \in (\texttt{Narrow} \leq T_m)$ FORWARD **do** ▷ Forward tranversal.
29: **for** $\mathbf{x}_j \in (\mathcal{N}(\mathbf{x}_i) \cap \mathcal{X} \backslash \texttt{Frozen})$ **do**
30: $\widetilde{T}_i \leftarrow \textsc{SolveEikonal}\ \mathbf{x}_j, \mathcal{T}, \mathcal{F}$
31: **if** $\widetilde{T}_i < T_i$ **then**
32: $T_i \leftarrow \widetilde{T}_i$
33: **end if**
34: **if** $\mathbf{x}_i \in \texttt{Unknown}$ **then**
35: $\texttt{Unknown} \leftarrow \texttt{Unknown} \backslash \{\mathbf{x}_i\}$
36: $\texttt{Narrow} \leftarrow \texttt{Narrow} \cup \{\mathbf{x}_i\}$
37: **end if**
38: **end for**
39: $\texttt{Narrow} \leftarrow \texttt{Narrow} \backslash \{\mathbf{x}_i\}$
40: $\texttt{Frozen} \leftarrow \texttt{Frozen} \cup \{\mathbf{x}_i\}$
41: **end for**
42: **end while**
43: **return** $\mathcal{T}$
44: **end procedure**

conceptually simple: the Narrow set 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, an element from the first queue is evaluated. If its arrival 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 updated cell. Once the first queue is empty, the 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 the queues are not sorted, the arrival time of the same cell could require being solved many times until its value converges. DDQM dynamically computes the threshold value, which sets the division of the two queues, depending on the number of points of each queue, trying to reach an equilibrium. Reference [27] includes an in depth analysis of the update of the threshold in each iteration. In this work, the initial value of the step of the threshold is increased every iteration according to:

$$step = \frac{1.5hn}{\sum_i F_i} \tag{17}$$

where n is the total number of cells in the grid. Originally, it was suggested to compute this step as $step = \frac{1.5n}{h\sum_i \frac{1}{F_i}}$. However, the step value should have time units whereas this expression has $[t^{-1}]$ units (probably an error due to the ambiguity of using speed F or slowness $f = \frac{1}{F}$). Therefore, (17) is proposed as an alternative in this work.

While the algorithm evolves, every time the first queue is emptied UpdateStep() (see Algorithm 11) is called, using the value of the current *step*, the number of cells inserted in the first queue c_1, and the total number of cells inserted c_{total} as inputs. Then, *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 provoked by percentages lower than 50% is much more significant than for higher percentages. Note that in Algorithm 11 the step is increased by a factor of 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, the next iteration will finish faster and a better *step* value can be computed.

The method of DDQM is detailed in Algorithm 12. As in LSM, points are divided into the locked (Frozen) or unlocked (Narrow) sets. The initialization labels all the points as frozen except for the neighbors of the start points, which are added to the first queue (lines 2-14). While the first queue is not empty, its front element is extracted and evaluated (lines 18-20). If its time value is improved, all its locked neighbors with higher value are unlocked and added to their corresponding queue.

In [27], three methods were proposed: 1) single-queue (SQ), and therefore a simpler algorithm, 2) two-queue static (TQS), where the *step* is not updated, and 3) two-queue dynamic (which we call DDQM). SQ and TQS slightly improve on DDQM in some experiments, but when

Algorithm 11 DDQM Threshold Increase

procedure UpdateStep($step$, c_1, c_{total})
  $\mathtt{m} \leftarrow 0.65$
  $\mathtt{M} \leftarrow 0.75$
  $Perc \leftarrow 1$
  if $c_1 > 0$ **then**
    $Perc \leftarrow \frac{c_1}{c_{\text{total}}}$
  end if
  if $Perc \leq m$ **then**
    $step \leftarrow step * 1.5$
  else if $Perc \geq M$ **then**
    $step \leftarrow \frac{step}{2}$
  end if
  return $step$
end procedure

DDQM improves on SQ and TQS (for instance, in environments with noticeable speed changes) the difference can reach one order of magnitude. Therefore, we decided to include DDQM instead of SQ and TQS since it has shown a more adaptive behavior. In any case, any of these methods returns the same solution as FMM.

Regarding its complexity, 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), and therefore there is no need for a real swap operation.

C. FAST ITERATIVE METHOD

The Fast Iterative Method (FIM) [28] is based on the iterative method proposed by [49] but inspired by FMM. It also resembles DDQM (concretely, 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 as converged. As a result of FIM, if a small enough ϵ (depending on the environment) is chosen, the same solution as FMM is returned. However, it can be sped up allowing small errors bounded by ϵ. FIM is designed to be efficient for parallel computing, since all the elements in `Narrow` can be evaluated simultaneously. However, we are focusing on its sequential implementation in order to have a fair comparison with the other methods.

Algorithm 13 details FIM steps. Its initialization is the same as FMM. Then, for each element in `Narrow`, its value is updated (lines 13-14). If the value difference is less than ϵ, the neighbors are evaluated and, in case their value is improved, they are added to `Narrow` (lines 15-22). Since `Narrow` is a list, the new elements should be inserted

Algorithm 12 Double Dynamic Queue Method

1: **procedure** DDQM($\mathcal{X}$, $\mathcal{T}$, $\mathcal{F}$, $\mathcal{X}_s$)
   Initialization:
2: $\mathtt{Frozen} \leftarrow \mathcal{X}$, $\mathtt{Narrow} \leftarrow \emptyset$
3: $\mathcal{Q}_1 \leftarrow \emptyset$, $\mathcal{Q}_2 \leftarrow \emptyset$
4: $c_1 \leftarrow 0$ ▷ Counters
5: $c_{\text{total}} \leftarrow 0$
6: $step = \frac{1.5hn}{\sum_i F_i}$ ▷ n is the total number of cells.
7: $th \leftarrow step$
8: $T_i \leftarrow \infty \; \forall \mathbf{x}_i \in \mathcal{X}$
9: **for** $\mathbf{x}_i \in \mathcal{X}_s$ **do**
10: $T_i \leftarrow 0$
11: **for** $\mathbf{x}_j \in \mathcal{N}(\mathbf{x}_i)$ **do**
12: $\mathcal{Q}_1 \leftarrow \mathcal{Q}_1 \cup \{\mathbf{x}_j\}$
13: $\mathtt{Unknown} \leftarrow \mathtt{Unknown} \backslash \{\mathbf{x}_i\}$
14: $\mathtt{Narrow} \leftarrow \mathtt{Narrow} \cup \{\mathbf{x}_i\}$
15: **end for**
16: **end for**

   Propagation:
17: **while** $\mathcal{Q}_1 \neq \emptyset$ *or* $\mathcal{Q}_2 \neq \emptyset$ **do**
18: **while** $\mathcal{Q}_1 \neq \emptyset$ **do**
19: $\mathbf{x}_i \leftarrow \mathcal{Q}_1$ .FRONT ▷ Extracts the front element.
20: $\widetilde{T}_i \leftarrow$ SOLVEEIKONAL $\mathbf{x}_i$, $\mathcal{T}$, $\mathcal{F}$
21: **if** $\widetilde{T}_i < T_i$ **then**
22: $T_i \leftarrow \widetilde{T}_i$
23: **for** $\mathbf{x}_j \in (\mathcal{N}(\mathbf{x}_i) \cap \mathtt{Frozen})$ **do**
24: ▷ Add improvable neighbors to their queue.
25: **if** $T_i < T_j$ **then**
26: $\mathtt{Frozen} \leftarrow \mathtt{Frozen} \backslash \{\mathbf{x}_j\}$
27: $\mathtt{Narrow} \leftarrow \mathtt{Narrow} \cup \{\mathbf{x}_j\}$
28: $c_{\text{total}} \leftarrow c_{\text{total}} + 1$
29: **if** $T_i \leq th$ **then**
30: $\mathcal{Q}_1 \leftarrow \mathcal{Q}_1 \cup \{\mathbf{x}_j\}$
31: $c_1 \leftarrow c_1 + 1$
32: **else**
33: $\mathcal{Q}_2 \leftarrow (\mathcal{Q}_2 \cup \{\mathbf{x}_j\})$
34: **end if**
35: **end if**
36: **end for**
37: **end if**
38: $\mathtt{Narrow} \leftarrow \mathtt{Narrow} \backslash \{\mathbf{x}_i\}$
39: $\mathtt{Frozen} \leftarrow \mathtt{Frozen} \cup \{\mathbf{x}_i\}$
40: **end while**
41: $step \leftarrow$ UPDATESTEP $step$, c_1, c_{total}
42: $SWAP \mathcal{Q}_1$, $\mathcal{Q}_2$
43: $c_1 \leftarrow 0$
44: $c_{\text{total}} \leftarrow 0$
45: $th \leftarrow th + step$
46: **end while**
47: **return** $\mathcal{T}$
48: **end procedure**

just before the point being currently evaluated, $\mathbf{x}_i$. Finally, this point is removed from `Narrow` and labeled as `Frozen` (lines 25 and 26).

During the different iterations of the algorithm, a node can be added several times to the `Narrow` set, 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 $\mathcal{O}(n)$.

Algorithm 13 Fast Iterative Method

1: **procedure** FIM($\mathcal{X}, \mathcal{T}, \mathcal{F}, \mathcal{X}_s, \epsilon$)
Initialization:
2: `Frozen` $\leftarrow \mathcal{X}$, `Narrow` $\leftarrow \emptyset$
3: $T_i \leftarrow \infty \; \forall \mathbf{x}_i \in \mathcal{X}$
4: **for** $\mathbf{x}_i \in \mathcal{X}_s$ **do**
5: $T_i \leftarrow 0$
6: **for** $\mathbf{x}_j \in (\mathcal{N}(\mathbf{x}_i) \cap$ `Unknown`$)$ **do**
7: `Frozen` $\leftarrow$ `Frozen`$\backslash\{\mathbf{x}_i\}$
8: `Narrow` $\leftarrow$ `Narrow` $\cup \{\mathbf{x}_i\}$
9: **end for**
10: **end for**

Propagation:
11: **while** `Narrow` $\neq \emptyset$ **do**
12: **for** $\mathbf{x}_i \in$ `Narrow` **do**
13: $\widetilde{T}_i \leftarrow T_i$
14: $T_i \leftarrow$ SolveEikonal $\mathbf{x}_i, \mathcal{T}, \mathcal{F}$
15: **if** $|T_i - \widetilde{T}_i| < \epsilon$ **then**
16: **for** $\mathbf{x}_j \in (\mathcal{N}(\mathbf{x}_i) \cap$ `Frozen`$)$ **do**
17: $\widetilde{T}_j \leftarrow$ SolveEikonal $\mathbf{x}_j, \mathcal{T}, \mathcal{F}$
18: **if** $\widetilde{T}_j < T_j$ **then**
19: $T_j \leftarrow \widetilde{T}_j$
20: `Frozen` $\leftarrow$ `Frozen`$\backslash\{\mathbf{x}_j\}$
21: ▷ Insert in the Narrow band just before $\mathbf{x}_i$
22: `Narrow` $\leftarrow$ `Narrow` $\cup \{\mathbf{x}_j\}$
23: **end if**
24: **end for**
25: `Narrow` $\leftarrow$ `Narrow`$\backslash\{\mathbf{x}_i\}$
26: `Frozen` $\leftarrow$ `Frozen` $\cup \{\mathbf{x}_i\}$
27: **end if**
28: **end for**
29: **end while**
30: **return** $\mathcal{T}$
31: **end procedure**

VI. EXPERIMENTAL COMPARISON

A. EXPERIMENTAL SETUP

In order to make an impartial and meaningful comparison, all the algorithms have been implemented from scratch, in C++11 using the Boost Heap library. An automatic benchmark application has been created so that the experiments can be carried out and evaluated in the most systematic possible way. This implementation is focused on time performance, and was compiled using G++ 4.8.4 with optimization flag -Ofast. However, no special optimizations have been included. All algorithms use the same primitive functions for the grid and cell computations. The times reported correspond to an Ubuntu 14.04 64 bits on a virtual machine using 6 cores of 4GHz with 8GB of RAM. However, all experiments were carried out in one core. In order to calculate the time used by each algorithm, 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 is only a small 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 the runs. However, the results shown are the mean of ten runs for every algorithm, and the deviation of the results is practically zero.

For UFMM, the default parameters are a maximum range of $\mathcal{T}$ of 2 units and 1000 buckets (the checkerboard experiment required different ones, see Section VI-A.4). The ϵ parameter for FIM is set to 0 (actually 10^{-47} to provide robust 64 bit double comparison).

Although error analysis is not within the scope of this paper, it can be compared using the results in the existing literature since it is implementation-independent. UFMM errors are reported in those experiments with non-constant speed. Usually, the L_1 and L_∞ norms of the error are reported. Most of the literature computes norm L_1 as:

$$|\mathcal{T}|_1 = \sum_{\mathbf{x}_i \in \mathcal{X}} |T_i| \tag{18}$$

where $\mathcal{X}$ is treated as a regular vector. However, following [7], we treat the numerical solutions as elements of L_p spaces (a generalization of the p-norm to vector spaces), in which the 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 and grids):

$$|\mathcal{T}|_1 = \sum_{\mathbf{x}_i \in \mathcal{X}} |T_i h^N| = h^N \sum_{\mathbf{x}_i \in \mathcal{X}} |T_i| \tag{19}$$

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 can be remarked. These experiments were chosen attending to the most common situations tested in the literature. By combining the characteristics of these problems, it is possible to obtain similar features to those found in real applications. Furthermore, two more experiments have been included to test the application of the Fast Methods to real applications such as path planning or medical imaging.

1) 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 speed 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 cells increases. Composed of empty 2D, 3D and 4D hypercubical environments of size $[0, 1]^N$, with $N = 2, 3, 4$. The speed is a constant $F_i = 1$ on $\mathcal{X}$. The wavefront starts at the center of the grid. This experimental setup can be found in previous publications such as [27], [28], and [50].

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 different numbers of dimensions. Thus, we have chosen the following number of cells for each dimension for 2D grid:

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

Consequently, the 3D and 4D cells are:

$$3D : \{14, 22, 34, 54, 86, 100, 131, 159, 184, 208, 252\}$$
$$4D : \{7, 10, 14, 20, 28, 32, 39, 45, 50, 55, 63\}$$

2) ALTERNATING BARRIERS

In this case, we want to analyze how the algorithms behave with obstacles ($F_i = 0$) in a constant speed environment ($F_i = 1$). 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 Fig. 4. Analogously, in 3D, a $[0, 1]$x$[0, 1]$x$[0, 2]$ environment represented by a 100x100x200 grid is chosen, with equally distributed alternating barriers (from 0 to 9) along the z-axis. In all cases, the wavefront starts close to a corner of the map. Similar experimental setups can be found in the literature [7], [26], [28].

3) RANDOM SPEED FUNCTION

This experiment aims to test the performance of the algorithms with random speed function (similar to noisy images, as in the case of medical computer vision). It creates 2D, 3D and 4D environments 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 make a direct comparison with the empty map problem for the corresponding grid sizes. The wavefront starts in the center of the grid. This setup is inspired by the experiments carried out in [27], [28], and [50].

Additionally, the maximum speed is increased from 1 to 100 (in steps of 10 units) to analyze how the algorithms behave with increasing speed changes. 2D examples are shown in Fig. 5.

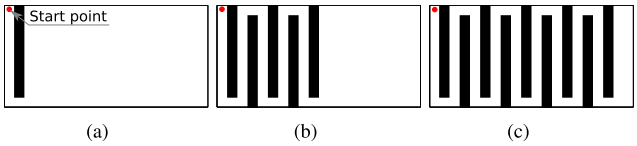

FIGURE 4. 2D alternating barriers environments. (a) 1 barrier. (b) 5 barriers. (c) 9 barriers.

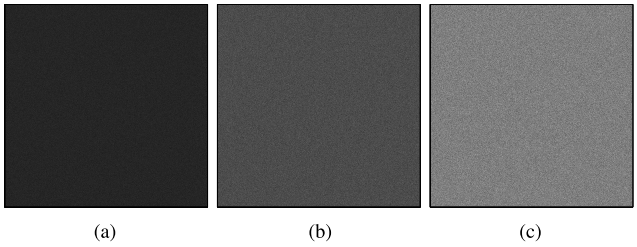

FIGURE 5. 2D random speed function environments. Lighter color means faster wave propagation. (a) Max. speed = 30. (b) Max. speed = 60. (c) Max. speed = 100.

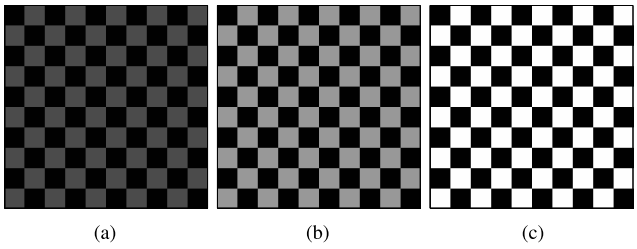

FIGURE 6. 2D checkerboard environments. Lighter color means faster wave propagation. (a) Max. speed = 30. (b) Max. speed = 60. (c) Max. speed = 100.

4) CHECKERBOARD

The random speed function experiment already tested changes in the speed. However, those are high-frequency changes because it is unlikely to have two adjacent cells with the same speed. In this experiment low-frequency changes are studied. The same environment and discretizations as with random speed function are now divided like a checkerboard, alternating minimum and maximum speed. Analogously, the maximum speed is increased from 1 to 100, while the minimum speed is always 1. There are 10 checkerboard divisions for each dimension. The wavefront starts in the center of the grid. 2D examples are shown in Fig. 6. This experimental setup is inspired by the experiments carried out in [7].

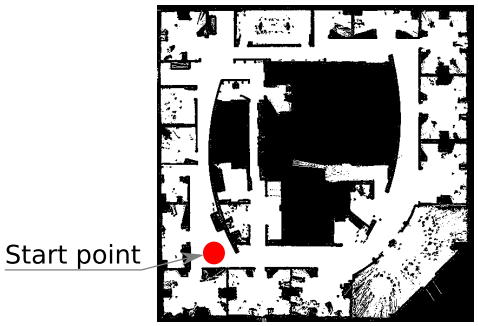

FIGURE 7. 2D gridmap: Intel Research Lab building in Seattle, 2500x2500px.

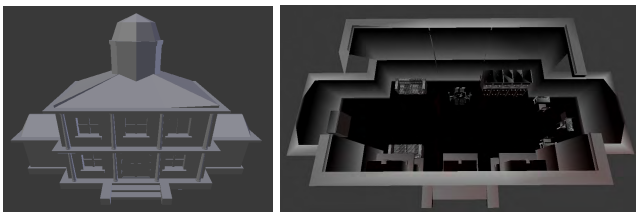

FIGURE 8. 3D mesh of the building used to generate 3D gridmaps. (a) Exterior view. (b) Interior view (internal walls are omitted).

In this case, UFMM in 3D and 4D performed very poorly with the default parameters. Additional tests not included in the present paper 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 a maximum range of 0.025 are used.

5) PATH PLANNING

Fast Methods are commonly used for low-dimensional path planning problems, given that they produce deterministic results, are complete (will find a solution if there is any), and it is possible to easily influence some properties of the paths, such as their smoothness or obstacle clearance [51]. Fast Methods are applied from a given start point until a goal point is reached and labeled as frozen, then gradient descent is applied from the goal point in order to obtain a path to the global minimum (the start point). The gradient descent step is omitted in these experiments as it is out of the scope of this paper.

Experiments in 2D and 3D maps are included. It is important to remark that since the Fast Methods are based on grids, they scale exponentially with the number of dimensions. To the best of the authors' knowledge, in path planning, there are no practical applications of Fast Methods further than 3D.

For 2D, the chosen map shown in Fig. 7 belongs to the Intel Research Laboratory building in Seattle. This map is commonly used in path planning and robot navigation research [52], [53]. The map is square-shaped, and contains binary values (1 for free cells, 0 for occupied cells). Similarly to the empty map experiment, different resolutions have been chosen for the sides of the square:

$$\{400, 800, 1000, 1500, 2000, 2500, 3000, 4000\}$$

For the 3D case, binary 3D gridmaps with different resolutions are created from a 3D model of a building used for video games, shown in Fig. 8, which has been taken from the Internet[4] with slight modifications (such as adding more pieces of furniture or more internal rooms). This model is composed of two floors. The first floor is divided into three rooms and all of them contain some furniture. The second floor has one large room. The Fast Methods are applied to the whole environment with the center of the first floor as starting point. The maximum size of the gridmap created is 355x190x237. In this case, the following resolutions are chosen for the first dimension (the other dimensions are linearly scaled):

$$\{19, 30, 48, 76, 122, 141, 185, 224, 260, 293, 355\}$$

The initial point for the propagation has been arbitrarily chosen in both cases. However, as the algorithms are run without a goal point, all the reachable cells in the map are evaluated. Therefore, although the choice of the initial point can modify the results, the experimental setup guarantees that the impact is negligible.

6) VESSEL SEGMENTATION

Another of the main uses of Fast Methods is computer vision for medical applications, as intermediary steps in more complex algorithms. For example, in [54], FMM is used for 2D vessel segmentation in retina images based on the assumption that vessels usually follow a smooth path. Flórez-Valencia *et al.* [55] perform a center line extraction in arteries using FMM, after defining an appropriate speed function and stopping criterion. It is then used to extract 2D contours in cross-sectional planes. The contours are finally used to progressively reconstruct a regularized continuous 3D surface.

This section aims to show the performance of the Fast Methods in such applications. More concretely, a vessel segmentation algorithm similar to the one used in [54] is implemented. When using a Fast Method for segmentation, the main goal is to define a speed function, $F(\mathbf{x})$, which provokes the wave expansion to happen faster in those areas which have to be segmented. In this case, the image is first processed with a high pass filter in order to subtract the smoothly varying background. Then, a speed function, in which the pixels corresponding to vessels have a higher value, is computed. Using this function, the Fast Method is performed using a central point of a vessel as starting point. Finally, gradient descent is used to extract the geodesics, which correspond to center lines of the vessels.

This experiment involves the use of the Fast Methods in grids with slightly structured propagation speed, which can

[4]https://www.cgtrader.com/free-3d-models/exterior/exterior-public/low-polygon-townhall-for-games

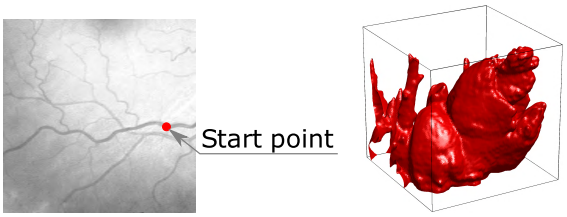

FIGURE 9. Grids used as inputs for the vessels segmentation algorithm. (a) 2D vessels initial grid. (b) Binarized 3D vessels grid after segmentation.

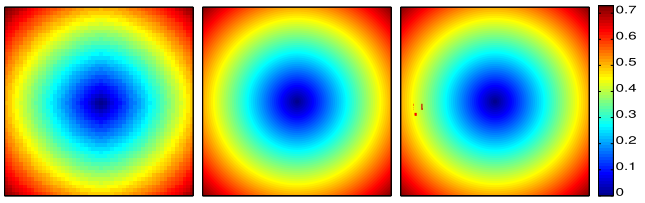

FIGURE 10. Example of the resulting time-of-arrival maps applying FMM to the empty environment in 2D. (a) 50x50. (b) 800x800. (c) 4000x4000.

be considered as a mix between the random speed function and checkerboard experiments. In this case, two examples are covered: 2D and 3D vessel segmentation, using as input for the algorithms the images shown in Fig. 9, taken from the DRIVE database of retinal vessels [56]. The 2D image has a resolution of 2560x2560 pixels with a range of speed [1.7234, 100] (of the preprocessed image according the aforementioned segmentation algorithm), whereas the 3D case is composed of a grid of 128x128x128 voxels with a range of speed [1, 100]. As in previous cases, a total of ten runs per algorithm are executed for each case. The starting point in the 3D case is close to one of the bottom corners.

In this experiment, as it focuses on a real application, the range of speed values used is smaller than those presented in the experiments above, since these values are due to the real image values. Besides, after the application of the Fast Methods, the necessary steps to complete the segmentation algorithm are not included in the experiment, because they are out of the scope of this study. It is also important to remark that in this kind of application, the Fast Methods employed are usually those based on high-accuracy solutions [57], which are generally slower than the methods included in this paper.

B. RESULTS

The next sections present the results of the experiments explained in Section VI-A. Their analysis is grouped attending to the classification given in Sections III,IV, and V.

1) EMPTY MAP

An example of the time-of-arrival field computed by FMM is shown in Fig. 10. 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 Fig. 11 for 2D, Fig. 12 for 3D, and Fig. 13 for 4D. In all cases two plots are included: raw computation times for each algorithm, and time ratios computed as:

$$ratio = \frac{\text{FMM Time}}{\text{Alg. Time}} \tag{20}$$

so that larger ratios represent better performances.

FMM is, as expected, the slowest algorithm in almost all cases, because the rest of the algorithms were proposed as improvements of FMM. Besides, an empty map environment is the most favorable case for any of the algorithms. As the number of cells increases, FMMFib quickly outperforms FMM since the number of elements in the narrow band increases exponentially with the number of dimensions, and therefore the better amortized times of the Fibonacci heap become useful.

SFMM, the other $\mathcal{O}(n \log n)$ Fast Method, is always faster than FMM and most of the times also faster then FMMFib, due to its simpler and faster heap management. However, as the number of cells increases, the tendencies of FMM-Fib and SFMM are very similar to that of FMM (the ratio remains constant as the number of cells increase). When the number of cells is large enough, it is hard to say whether SFMM or FMMFib are faster.

The sweeping-based methods show a similar behavior to the FMM-based methods. FSM is only slower than FMM-based methods for environments with a small number of cells. However, its linear complexity quickly makes it faster than FMM, FMMFib, and SFMM if the environment becomes larger. In spite of this, when the number of dimensions increases, the number of required sweeps also increases (duplicates) and therefore this penalizes the algorithm, as it will evaluate each cell more times. LSM and DDQM are methods that improve on FSM by avoiding the recomputation of the cells. Therefore, they become the fastest algorithms as they do not deal with heap operations and they minimize cell recomputation. In this case, LSM is faster in most cases (in 4D it is slower than DDQM but presents a better tendency than DDQM). This happens because DDQM maintains two queues. Whereas the operations of these queues are efficient, they still represent some overhead over LSM, which does not update any internal container.

The iterative algorithms, such as GMM and FIM, present moderate results with similar behavior. As they keep simple data structures, they usually are faster than FMM-like methods. However, they do not leverage the brute-force approach followed by sweeping-based methods, adding some additional overhead to the iterations and thus, being usually slower than them. UFMM also provides average computation times for a similar reason: it maintains a heap, something which is more efficient than FMM-like algorithms, but still requires performing additional operations in comparison to sweeping-based methods. As the speed is constant all over the grid, UFMM provides the same solution as the other methods.

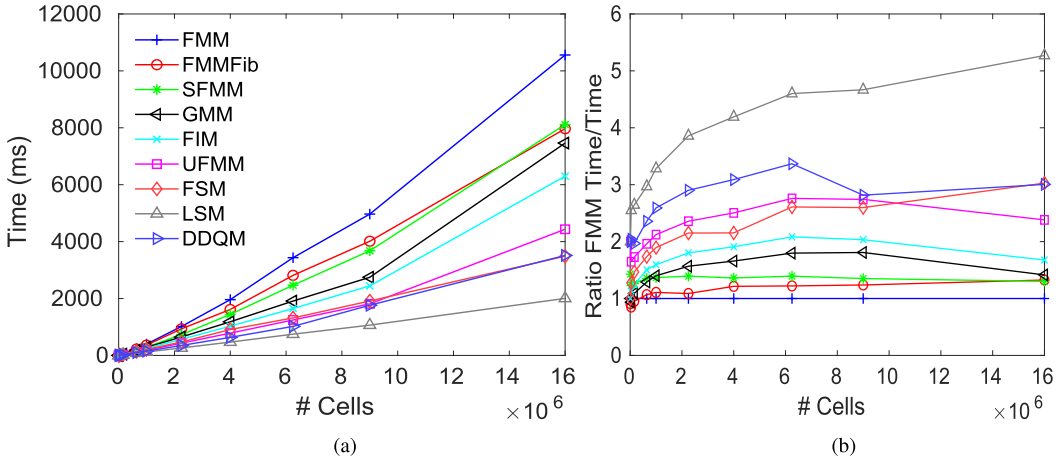

FIGURE 11. Computation times and ratios for the empty map environment in 2D. (a) Computation times. (b) Time ratios against FMM.

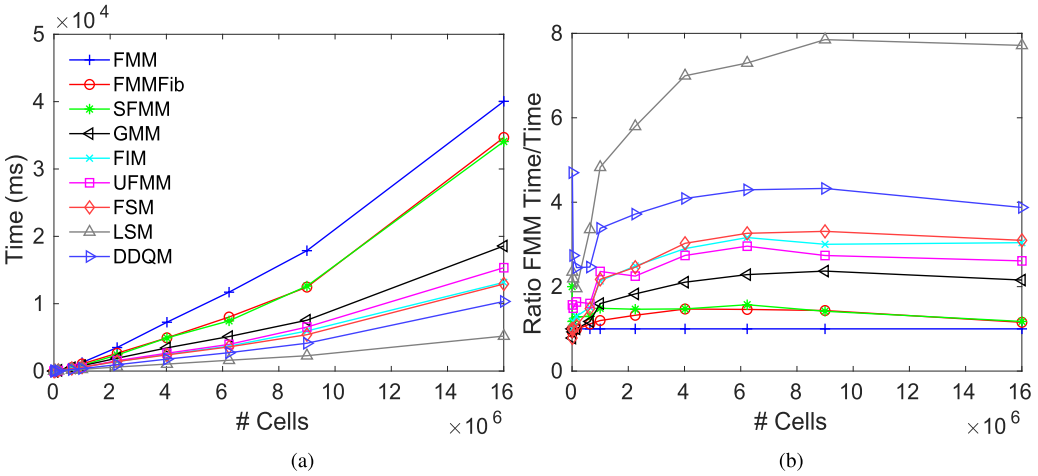

FIGURE 12. Computation times and ratios for the empty map experiment in 3D. (a) Computation times. (b) Time ratios against FMM.

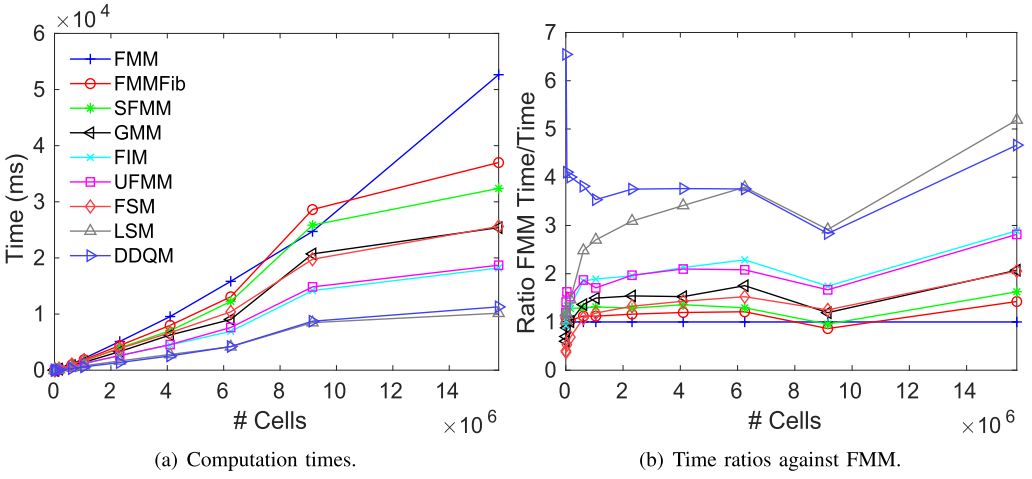

FIGURE 13. Computation times and ratios for the empty map experiment in 4D. (a) Computation times. (b) Time ratios against FMM.

In a previous comparison between GMM and FMM [28], GMM was about 50% faster than FMM in all cases. In the results presented here, GMM is at most 40% better. We attribute this difference to the implementation, as the heaps for FMM and FMMFib are highly optimized. Therefore, it is worth mentioning that the results shown here are

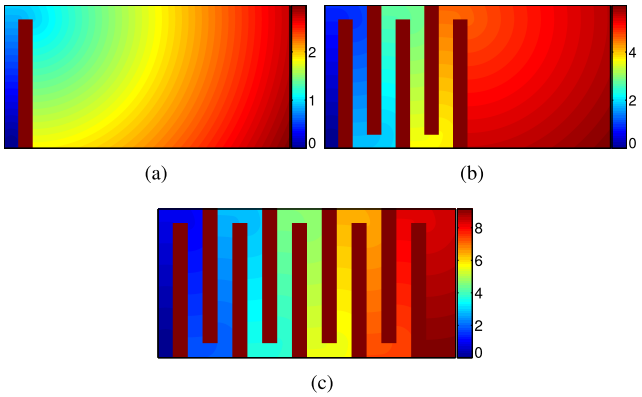

FIGURE 14. Example of the resulting time-of-arrival maps applying FMM to some of the alternating barriers environment in 2D. (a) 1 barrier. (b) 5 barriers. (c) 9 barriers.

also slightly subject to implementation details and, sometimes, details that are out of the reach of regular users, such as internal cache memory management, prefetchers, and other low-level details of the hardware used.

The conclusions from these experiments are that, in the absence of obstacles and propagation speed modifications, sweeping-like methods perform the best. Although this setup is unlikely to be present in a practical scenario, the results allow understanding the behavior of the algorithms in ideal situations and their major advantages.

2) ALTERNATING BARRIERS

An example of the results of FMM in some of the alternating barriers environments is shown in Fig. 14, whereas its performance results (times and ratios) for 2D and 3D are shown in Fig. 15 and Fig. 16 correspondingly.

Overall, the results are very similar to the empty map experiment. FMM and FMMFib are again the slowest algorithms when the number of barriers is low, since this setup is very similar to the empty map experiment. SFMM outperforms FMM and FMMFib in all cases, and it again performs better that FIM and GMM. However, in the case of 3D, GMM performs noticeably worse than in 2D due to its overhead while managing the narrow band. The results of FIM and UFMM are also similar to those in the previous experiment.

The most important results are those of the sweep-based methods. This type of map requires more sweeps to cover the full environment as it becomes more complex (i.e., as the number of barriers increases), which negatively affects FSM and LSM. FSM becomes the slowest algorithm as soon as the environment is slightly complex. LSM is still faster than many of the other algorithms since it avoids reevaluating the cells over the sweeps, although its increasing trends show that it is still heavily affected by the environment. In 3D the trend is not as uniform as in 2D. The reason is that in 2D a cell can only be evaluated from four directions. But these directions increase exponentially with the number of dimensions, therefore there are more chances that the sweep directions are *aligned* with the environment making the sweeps more efficient since each sweep evaluates more unexplored cells.

As the propagation speed of the environment remains constant, the effect on DDQM of the complexity of the map is almost negligible. Also, UFMM provides again the same solution as the other methods.

Finally, the fact that most of the algorithms show a decreasing trend as the number of barriers increase is simply because with more barriers there are fewer cells to be evaluated.

3) RANDOM SPEED FUNCTION

: The output of the FMM for the random speed map (Fig. 17) is apparently close to the one of the empty map, but with the wavefronts slightly distorted because of the speed changes. However, the performance of the algorithms is greatly modified. The 2D, 3D and 4D results are respectively shown in Fig. 18, Fig. 19, and Fig. 20. 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 the methods become slower with non-constant speed functions. This happens because the narrow band tends to contain more elements in these cases, leading to slower operations. But also the cells have to be reevaluated more times than usual because the values of the neighbors change more frequently.

FMM and FMMFib have the predictable behavior, FMM being faster (just because of the size of the chosen map). As expected, SFMM has a better performance than FMM *and similar to FMMFib*. These algorithms were designed so that they do not make any assumption on the environment and therefore they are not affected by complexity or propagation speed.

Sweep-based methods become unstable (in terms of asymptotic computational time) with non-uniform speed function. The main reason is that high contrasts in the speed can act as *barriers* and therefore more sweeps can be required. DDQM is able to maintain the fastest time for slight speed changes, but when the changes are sharper the double-queue threshold becomes unstable. However, for a large number of dimensions, this effect vanishes (its computation time is barely modified with the number of dimensions) and DDQM becomes one of the fastest algorithms. An explanation for this is that the overhead caused by the required sweeps is lower (even if more sweeps are required) than the overhead the other algorithms suffer when increasing the dimensions.

In this case, GMM and FIM are the fastest algorithms. Although FIM requires multiple iterations over the same cells in order to converge to the real value, it only iterates over the narrow band, being faster than sweep-based methods (although these iterations makes it slower in lower dimensions). On the other hand, GMM presents a constant computation time because it will always converge in two iterations, regardless of the speed, thus becoming the most efficient algorithm.

Finally, UFMM requires special attention as it does not provide the same solution as the other Fast Methods.

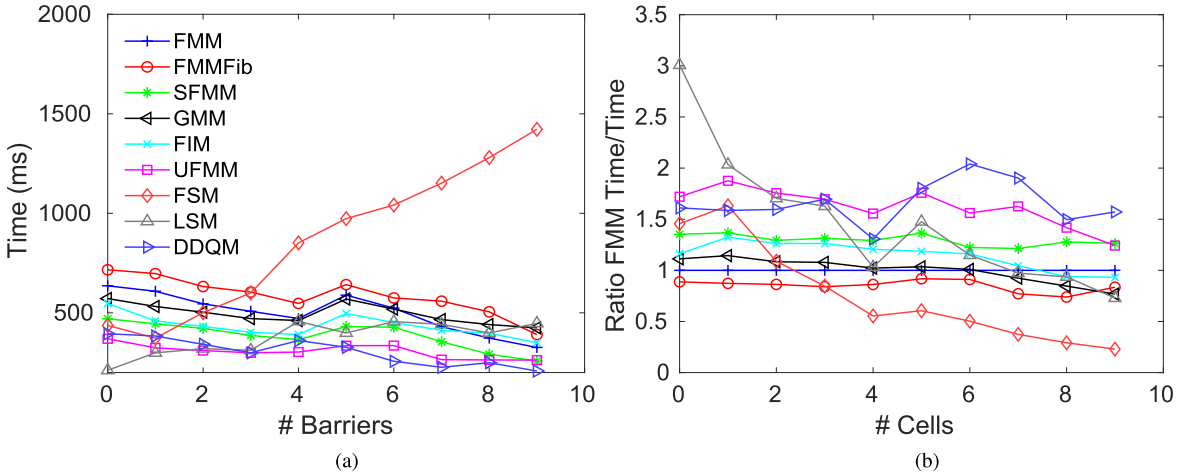

FIGURE 15. Computation times and ratios for the alternating barriers experiment in 2D. (a) Computation times. (b) Time ratios against FMM.

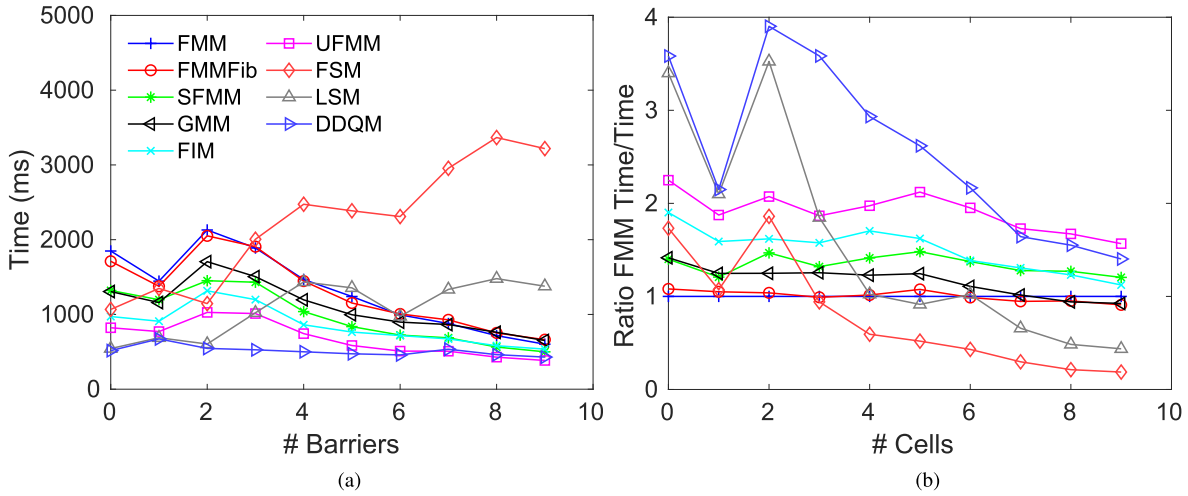

FIGURE 16. Computation times and ratios for the alternating barriers experiment in 3D. (a) Computation times. (b) Time ratios against FMM.

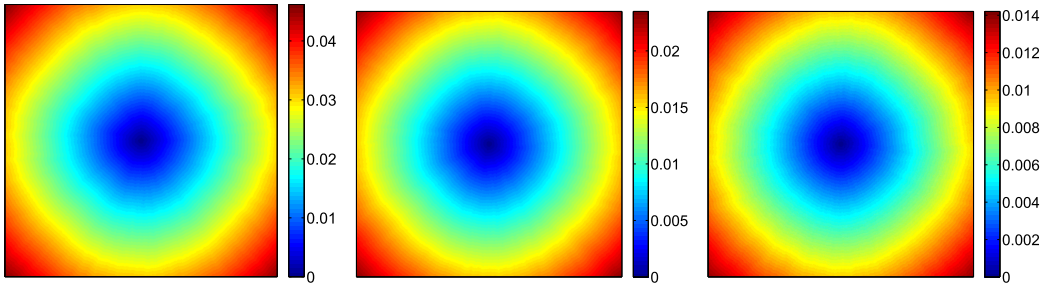

FIGURE 17. Example of the resulting time-of-arrival maps applying FMM to the random speed function environment in 2D. (a) Max. speed = 30. (b) Max. speed = 60. (c) Max. speed = 100.

Its performance is very sensitive to the number of dimensions. The main reason is the election of the parameters: they were experimentally chosen to optimize the 2D performance. However, these parameters are no longer useful for different number of dimensions. We consider UFMM parameter tuning to be a complex task.

Table 2 summarizes the largest errors for this experiment. As the number of dimensions increases, the error decreases

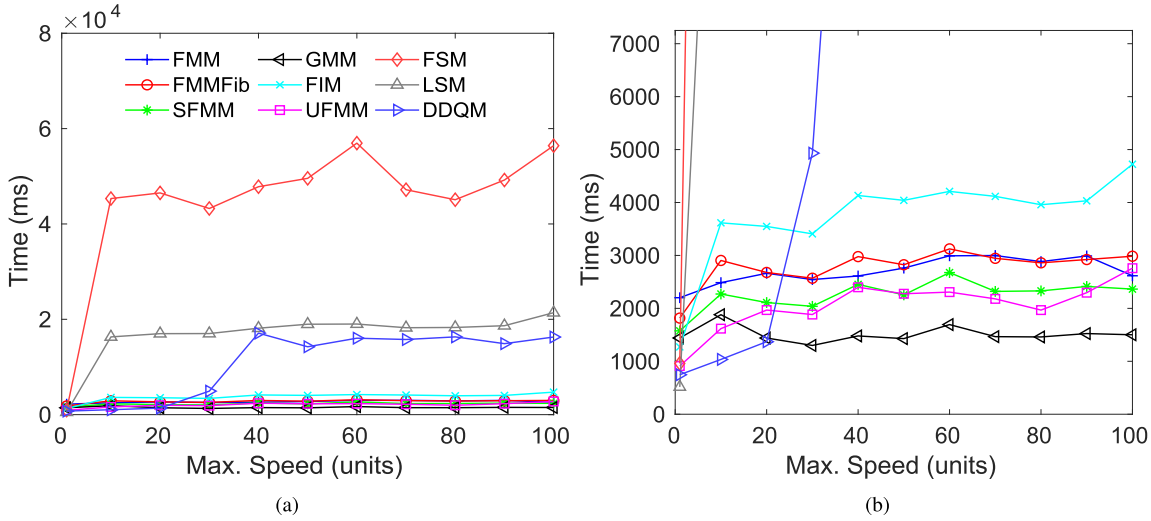

FIGURE 18. Computation times for the random speed function experiment in 2D. (a) Computation times. (b) Zoomed in view of the lower part of the chart in a).

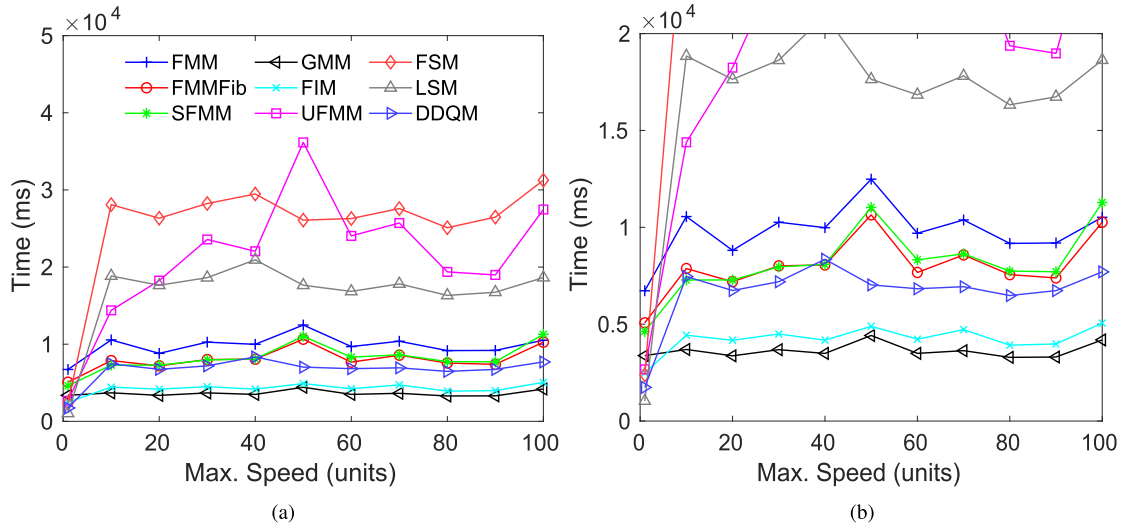

FIGURE 19. Computation times for the random speed function experiment in 3D. (a) Computation times. (b) Zoomed in view of the lower part of the chart in a).

TABLE 2. Largest L_1 and L_∞ errors for UFMM in the random speed function 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}

while the computation time increases exponentially. Therefore, by properly tuning the parameters for 3D and 4D better times could be achieved while keeping a negligible error in most practical cases.

4) CHECKERBOARD

Different time-of-arrival maps computed by FMM based on the checkerboard grid are shown in Fig. 21. The numerical results of the computation times are included in Fig. 22 for 2D, Fig. 23 for 3D, and Fig. 24 for 4D.

The results of FMM, FMMFib and SFMM remain the same as in previous experiments. The main difference between these experiments and the one with random speed function is that the environment presents a well-defined structure and, locally, acts as a constant speed environment as in the empty map experiment.

For sweeping-based methods, the results are relatively close to those in the random speed function experiment. However, the differences between the algorithms are much smaller. DDQM presents a poor performance in 2D, but in 3D and 4D it becomes the fastest algorithm for higher speed modifications. The reason for this behavior is that DDQM sweeps over a queue and, therefore, there is an overhead for

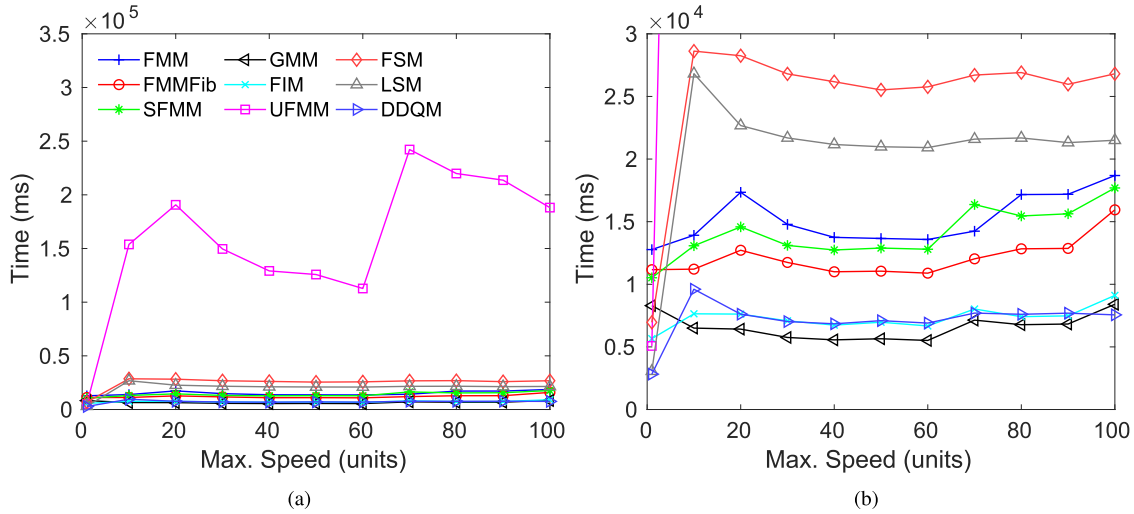

FIGURE 20. Computation times for the random speed function experiment in 4D. (a) Computation times. (b) Zoomed in view of the lower part of the chart in a).

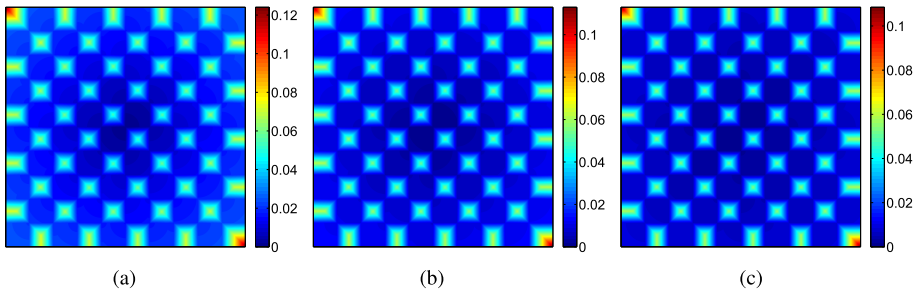

FIGURE 21. Example of the resulting time-of-arrival maps applying FMM to the checkerboard environment in 2D. (a) Max. speed = 30. (b) Max. speed = 60. (c) Max. speed = 100.

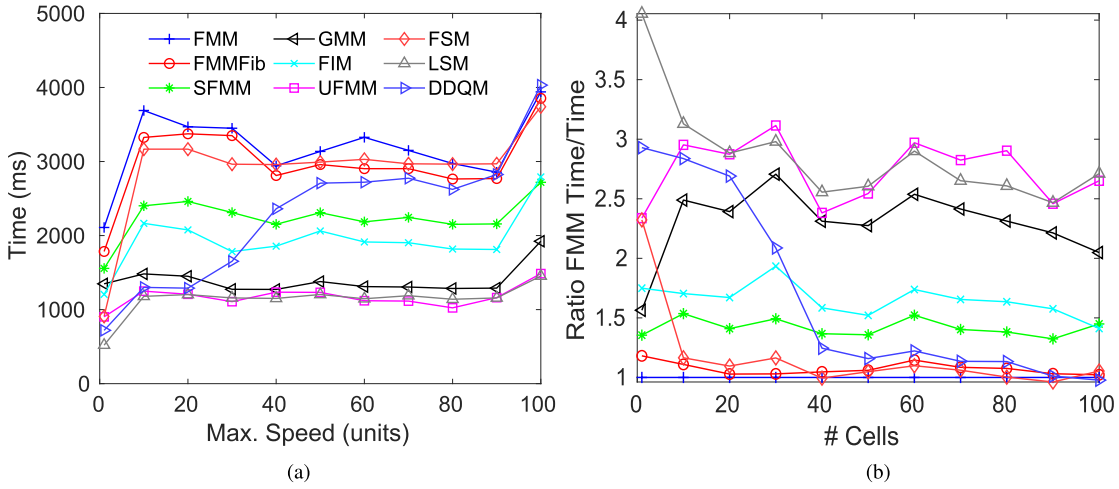

FIGURE 22. Computation times and ratios for the checkerboard experiment in 2D. (a) Computation times. (b) Time ratios against FMM.

maintaining this queue that becomes negligible once the number of elements in the queue is large enough. Besides, if the number of cells of the environment is increased, the same behavior is expected. Also, the gap between FSM/LSM and the remaining algorithms is much smaller than in the random speed function experiment. This is because in this case the environment presents some structure, allowing for more cells to be evaluated at every sweep and thus becoming more efficient, counteracting the overhead of performing more sweeps because of the different speed values.

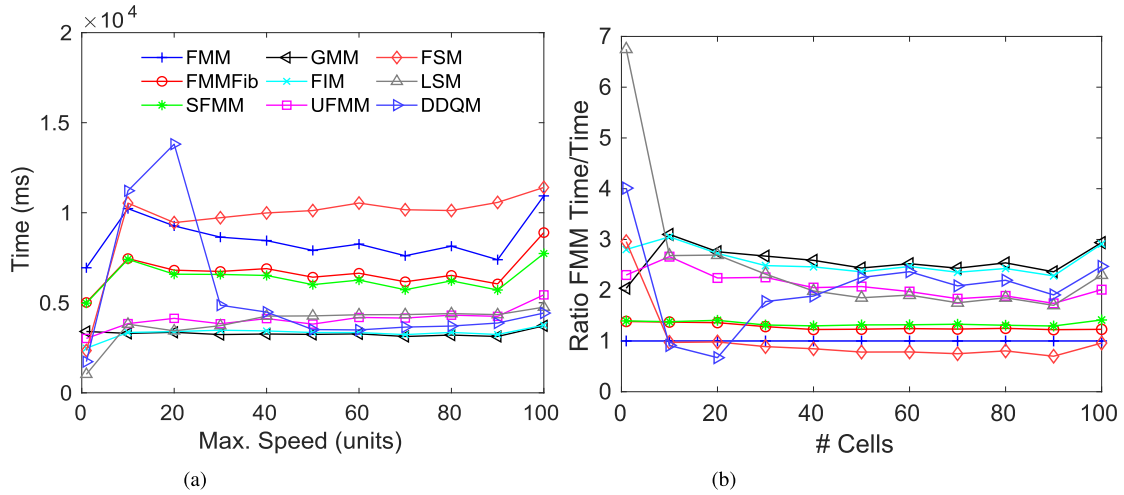

FIGURE 23. Computation times and ratios for the checkerboard experiment in 3D. (a) Computation times. (b) Time ratios against FMM.

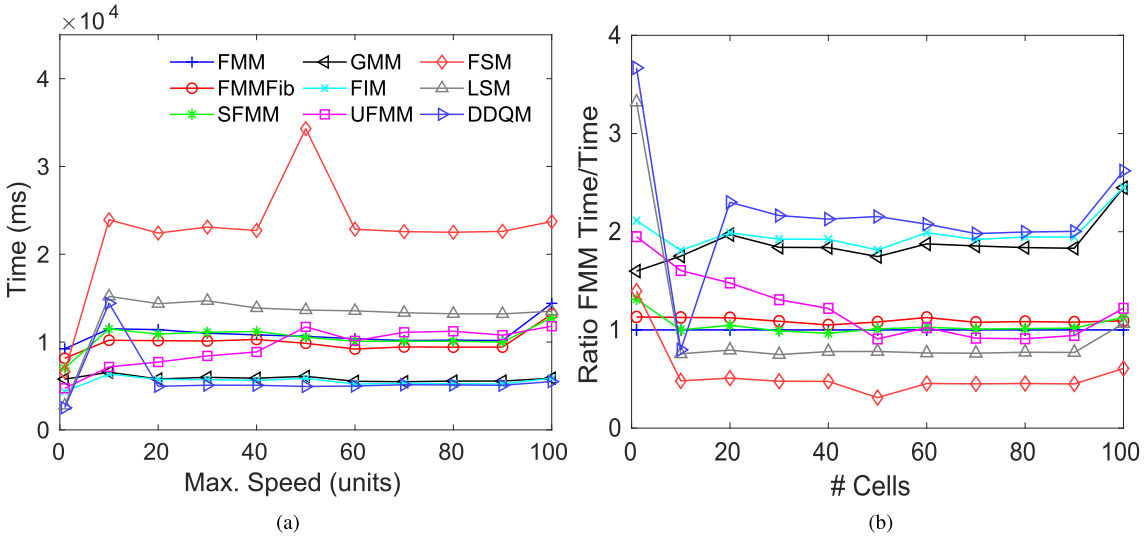

FIGURE 24. Computation times and ratios for the checkerboard experiment in 4D. (a) Computation times. (b) Time ratios against FMM.

GMM and FIM perform very similarly to the experiment with random speed function, but in this case their results are closer to each other. FIM requires more evaluations only on those cells whose neighbors have different propagation speed than the evaluated one. Because of the structure of the environment, FIM does not require so many iterations to converge to the cell value. In fact, it can solve most of the cells in only two iterations, as it is very likely that the propagation speed is constant in the neighborhood of the cell to be evaluated. These additional evaluations with respect to GMM are only noticeable in 2D, when the results are more susceptible to internal data structure overheads.

UFMM becomes worse with the number of dimensions but it is among the fastest algorithms in lower dimensions. Again, it is worth noting that these results greatly depend on the parameters chosen, which in this case seem to be close to optimal. UFMM errors are shown in Table 3.

TABLE 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}

5) PATH PLANNING

The FMM time-of-arrival field is shown in Fig. 25 for the 2D path planning problem. As the map is composed of binary values, all algorithms provide the exact same solution. Visualization of the results of the 3D path planning problem is not included as they cannot be clearly depicted with a single figure.

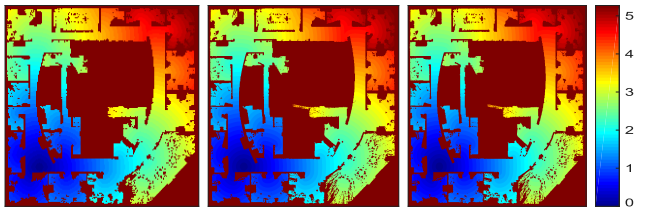

FIGURE 25. Example of the resulting time-of-arrival maps applying FMM to 2D path planning map. (a) 200x200. (b) 1500x1500. (c) 4000x4000.

TABLE 4. Computation times (seconds) for the vessels segmentation experiment.

	2D	3D
FMM	3.36	3.22
FMMFib	3.36	2.9
SFMM	2.38	2.81
GMM	2.06	5.87
FIM	25.15	5.51
UFMM	1.58	**1.84**
FSM	30.92	18.43
LSM	8.7	15.42
DDQM	**1.01**	3.32

The results for the path planning experiments are shown in Fig. 26 for 2D, and Fig. 27 for 3D. In both cases two plots are included: the raw computation times for each algorithm and the time ratios (analogously to the results of the empty map experiment). The results for such a binary and complex map greatly depend on its topology. In this case, the Intel Research Laboratories map has both large clear spaces and cluttered, irregular areas, therefore, the results can be interpreted as a mix between those of the empty map experiment and those from the alternating barriers experiment.

For both the 2D and 3D experiments, the results are very similar to the alternating barriers experiments. FMM and FMMFib perform similarly, with SFMM being faster in all cases.

FSM and LSM suffer from the complexity of the environments, being the slowest in many of the cases. As the number of cells in the map increases, the ratio of these methods with respect to FMM decreases because the map becomes relatively simpler. In other words, there is a higher density of free cells (especially in open areas) and, therefore, a larger number of cells can be computed in every sweep even if the proportion to the total map remains the same. This is a manifestation of the fact that the total number of cells increases exponentially, but the complexity of such methods increases linearly. However, DDQM shows the fastest computation times since it is rather immune to the environment complexity.

Both GMM and FIM are among the slowest methods. This behavior has been appreciated already in other experiments with constant propagation speed (empty map and alternating barriers). However, UFMM is among the fastest algorithms because of its efficient heap structure.

In the 3D results there is a bump on the times and ratios for a given resolution. The reason for this irregularity is that in the chosen map, for that given resolution, there are very specific voxels that are difficult to reach, requiring many iterations of the algorithm. This is the reason why it affects only the purely iterative algorithms (FSM, LSM, and FIM). For UFMM, however, these specific voxels cause the untidy heap to be less uniform and therefore also requires more operations.

6) VESSELS SEGMENTATION

The time-of-arrival map returned by FMM applied to the 2D vessels grid is shown in Fig. 28. The numerical results of the computation times for 2D and 3D are included in Table 4. The results for 2D and 3D are not directly comparable among them as the environments are noticeable different. Nevertheless, some of the behavior can be appreciated and correlated with the generic experiments explained previously. In 2D, the results are in line with those for random speed function for a range of speed values smaller than 20 units. Although to human perception the vessel images seem to be slightly structured, the irregular shapes and the image quality causes the algorithms to behave very similarly to the case of a completely random speed map. In 3D, the results are closer to those observed in the checkerboard experiment.

As in all the other experiments, FMM and FMMFib provide almost identical results, with SFMM outperforming them in both cases.

Since the propagation speed is non-constant, sweep-based methods require many sweeps in order to converge to the final solution. Therefore, FSM and LSM are slower than all the other methods. However, DDQM is again the fastest algorithm in 2D, and among the fastest ones in 3D. The reason for this is that there are numerous changes in the speed of the environment, but not as frequent as in the random speed function experiment, because there is some structure along the lines of the vessels and the surrounding tissues.

As in the 2D random speed function experiment, FIM is among the slowest algorithms because many iterations are required to converge to the final solution due to the frequent changes of the propagation speed. This effect vanishes in 3D as the iterations become more efficient since the narrow band contains more elements. Also, the difference in the environments play an important role in these results.

GMM is one of the fastest methods in 2D, similarly to the other variable speed experiments. However, in 3D its results are similar to those shown in the alternating barriers, showing once again the sensitivity of GMM to the environment.

Finally, UFMM is among the fastest algorithms, mostly due to a lucky selection of parameters for the environments chosen, because in both random speed function and checkerboard experiments UFMM does not show good results.

Considering that the speed in the environments is not constant, the errors for UFMM are included in Table 5.

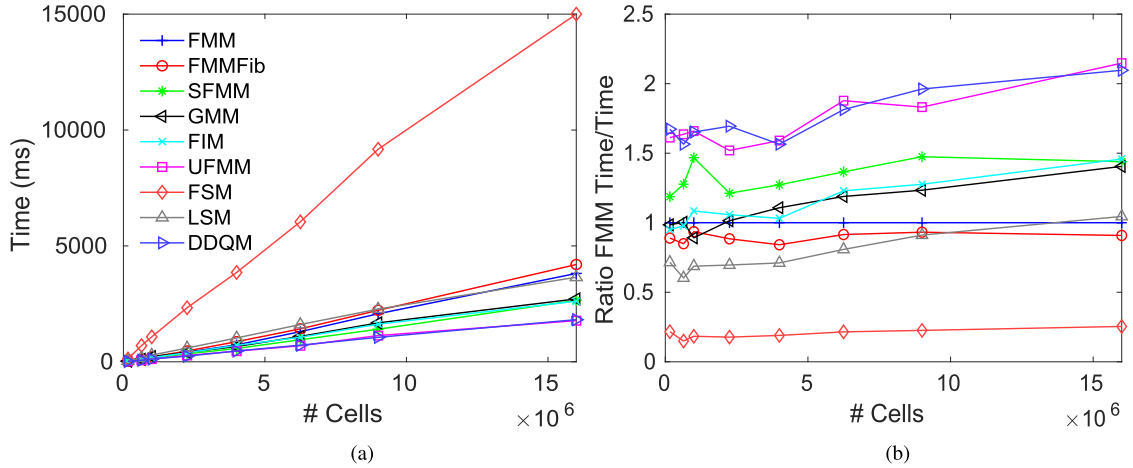

FIGURE 26. Computation times and ratios for the 2D path planning experiment. (a) Computation times. (b) Time ratios against FMM.

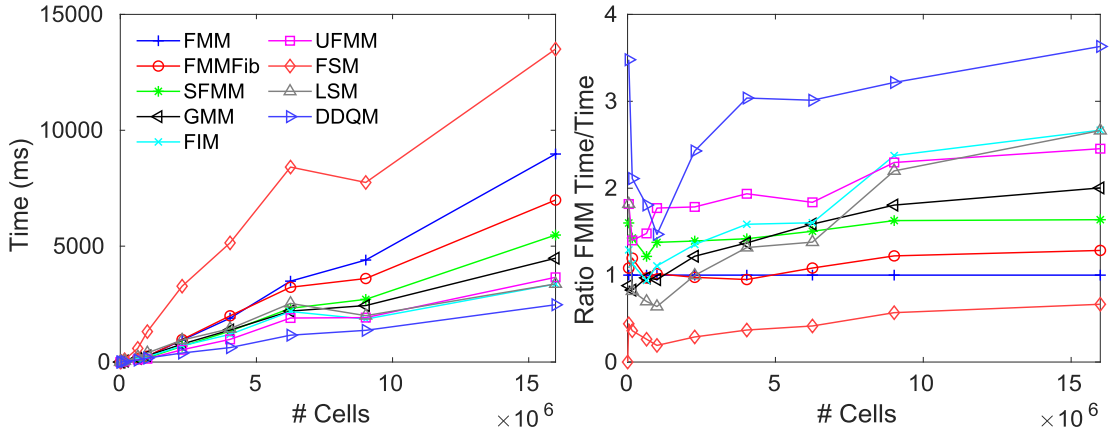

FIGURE 27. Computation times and ratios for the 3D path planning experiment. (a) Computation times. (b) Time ratios against FMM.

TABLE 5. Largest L_1 and L_∞ errors for UFMM in the vessels segmentation experiment.

	2D	3D
L_1	$3.6 \cdot 10^{-6}$	$5.6 \cdot 10^{-4}$
L_∞	$1.4 \cdot 10^{-4}$	$2 \cdot 10^{-4}$

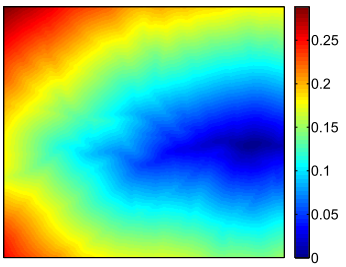

FIGURE 28. Example of the resulting time-of-arrival maps applying FMM to the vessels grid in 2D.

VII. DISCUSSION

Two different sets of experiments have been carried out. The first set contains canonical problems previously included in the literature that cover the best and worst case scenarios for all the methods. This set consists of: empty map, alternating barriers, random speed function and checkerboard experiment. The main hypothesis considered while designing these experiments is that any other environment can be thought of as a combination of free space with obstacles and high-frequency or low-frequency speed changes of different magnitudes. Consequently, the second set aims to test this hypothesis while applying the Fast Methods to problems which represent real world applications. The second set is composed of the path planning experiment and the vessels experiment, in which the environments are more unpredictable and can contain many of the characteristics of the canonical problems. Note that the different grid sizes used in the experiments vary from extremely small to extraordinarily big grids.

The path planning and vessels results can be correlated with those in the canonical problems, therefore supporting the

choice of the latter. However, it is important to remark that these results are very sensitive to the environments chosen. Some algorithms, such as FMM, FMMFib, and SFMM are completely independent of the environment, as their internal data structures behave in the same manner regardless of shape or speed, and therefore show consistent behavior for all the experiments, but they are never among the fastest methods. FIM results can be sped up for non-constant speed problems if larger errors are allowed. UFMM can probably be improved as well. However, our experience is that the configuration of its parameters is complex and requires a deep analysis of the environment to which it is to be applied on.

The results are also subject to low-level factors out of the scope of this paper. These factors are, for example, internal cache levels and memory management, data prefetching, etc. For example, the performance of algorithms that maintain a heap for the narrow band greatly depends on whether the narrow band is small enough that can be stored completely in microprocessor cache memories. Another example is that some algorithms such as DDQM or FIM make it complex for the prefetchers to prefetch the appropriate indices of the narrow band that are going to be evaluated on the following iterations since, commonly, grid neighbors are not contiguously stored in memory.

Taking into account the results from the conducted experiments, several conclusions for the use of the different types of Fast Methods will now be summarized:

- There is no practical reason to use FMM or FMMFib as SFMM is faster in every case. It shows the same behavior as its counterparts regardless of the characteristics of the environment, since the internal narrow band implementation is more efficient.
- If a sweep-based method is to be used, LSM should be always chosen, as it greatly outperforms FSM, since it recomputes cells only if there is a chance of improving their value. In fact, it was not possible to find any case in which FSM performs better than LSM. Therefore, FSM methods are recommended when applied to simple scenarios with constant propagation speed, because they require lower number of sweeps.
- In problems with constant speed and negligible dependence on the environmental complexity, DDQM should be chosen, as it has shown the best performance for the empty map and the alternating barriers environments, given that it combines the advantages of sweep-based and wavefront-based methods. However, for problems with variable speed values, its performance is highly influenced by the distribution of speed changes throughout the environment, as it might require many evaluations of the same cells in order to converge to the final solution.
- For variable speed functions, but simple scenarios, GMM is the algorithm to choose, as it guarantees that only two cell evaluations are required. However, complex environments distort the narrow band, requiring more iterations of its main loop.
- UFMM is hard to tune and its results include errors. Also, it has been outperformed in most of the cases by DDQM in constant speed scenarios, or by SFMM or FIM in experiments with variable speed. In order to evaluate whether this method should be considered, an in-depth study for the specific problem is required.
- There is no clear winner for complex scenarios with variable speed. UFMM can perform well in all cases if tuned properly. Otherwise, SFMM is a safe choice, especially in cases where there is not much information about the environment.

If a goal point is selected, cost-to-go heuristics can be applied [58], which would greatly affect the results. Heuristics for FMM, FMMFib and SFMM are straightforward and they can be similarly applied to UFMM. They would improve the results in most of the cases. However, it is not clear if they can be applied to other Fast Methods. Anisotropic solutions given to anisotropic problems based on some of the presented methods are also interesting [9], [47], [59], [60].

VIII. CONCLUSIONS

In this paper we have introduced the main Fast Methods in a common mathematical framework, adopting a practical point of view. Besides, an exhaustive comparison of the methods has been performed, which allows the users to choose among them depending on the application.

The code is publicly available,[5] as are the automatic benchmark programs. This code has been thoroughly tested and it can serve as a basis for future algorithm design, as it provides all the tools required to easily implement and compare novel Fast Methods.

Future research will focus on three different aspects: developing an analogous review for parallelized Fast Methods [31], studying the application of these methods to anisotropic problems as well as to the new Fast Marching-based solutions focused on path planning applications [61], [62]. Finally, the combination of UFMM and SFMM seems straightforward and it would presumably outperform both algorithms.

ACKNOWLEDGMENT

The authors would like to thank the contribution of Pablo Gely Muñoz to the GMM, FIM and UFMM implementations, and to Adam Chacon for the interesting discussions and suggestions towards improving the work.

REFERENCES

[1] J. N. Tsitsiklis, "Efficient algorithms for globally optimal trajectories," *IEEE Trans. Autom. Control*, vol. 40, no. 9, pp. 1528–1538, Sep. 1995.
[2] J. A. Sethian, "A fast marching level set method for monotonically advancing fronts," *Proc. Nat. Acad. Sci. USA*, vol. 93, no. 4, pp. 1591–1595, 1996.
[3] J. A. Sethian and A. Vladimirsky, "Ordered upwind methods for static Hamilton–Jacobi equations: Theory and algorithms," *SIAM J. Numer. Anal.*, vol. 41, no. 1, pp. 325–363, 2003.
[4] H. Zhao, "A fast sweeping method for eikonal equations," *Math. Comput.*, vol. 74, no. 250, pp. 603–627, 2005.

[5] https://github.com/jvgomez/fast_methods

[5] P. A. Gremaud and C. M. Kuster, "Computational study of fast methods for the Eikonal equation," *SIAM J. Sci. Comput.*, vol. 27, no. 6, pp. 1803–1816, 2006.
[6] J. A. Sethian and A. Vladimirsky, "Fast methods for the Eikonal and related Hamilton–Jacobi equations on unstructured meshes," *Proc. Nat. Acad. Sci. USA*, vol. 97, no. 11, pp. 5699–5703, 2000.
[7] A. Chacon, "Eikonal equations: New two-scale algorithms and error analysis," Ph.D. dissertation, Dept. Math., Cornell Univ., Jan. 2014.
[8] J. V. Gómez, "Advanced applications of the fast marching square planning method," M.S. thesis, Syst. Eng. Automat. Dept., Carlos III Univ., 2012.
[9] C. Petres, Y. Pailhas, P. Patron, Y. Petillot, J. Evans, and D. Lane, "Path planning for autonomous underwater vehicles," *IEEE Trans. Robot.*, vol. 23, no. 2, pp. 331–341, Apr. 2007.
[10] Q. H. Do, S. Mita, and K. Yoneda, "Narrow passage path planning using fast marching method and support vector machine," in *Proc. IEEE Intell. Vehicles Symp.*, Jun. 2014, pp. 630–635.
[11] Y. Liu and R. Bucknall, "Path planning algorithm for unmanned surface vehicle formations in a practical maritime environment," *Ocean Eng.*, vol. 97, pp. 126–144, Mar. 2015.
[12] F. Gao, W. Wu, Y. Lin, and S. Shen, "Online safe trajectory generation for quadrotors using fast marching method and bernstein basis polynomial," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2018, pp. 344–351.
[13] N. Forcadel, C. Le Guyader, and C. Gout, "Generalized fast marching method: Applications to image segmentation," *Numer. Algorithms*, vol. 48, nos. 1–3, pp. 189–211, 2008.
[14] S. Basu and D. Racoceanu, "Reconstructing neuronal morphology from microscopy stacks using fast marching," in *Proc. IEEE Int. Conf. Image Process.*, Oct. 2014, pp. 3597–3601.
[15] N. Al Zaben, N. Madusanka, A. Al Shdefat, and H.-K. Choi, "Comparison of active contour and fast marching methods of hippocampus segmentation," in *Proc. 6th Int. Conf. Inf. Commun. Syst.*, Apr. 2015, pp. 106–110.
[16] A. Capar and M. Gokmen, "Concurrent segmentation and recognition with shape-driven fast marching methods," in *Proc. Int. Conf. Pattern Recognit.*, vol. 1, Aug. 2006, pp. 155–158.
[17] M. Frenkel and R. Basri, "Curve matching using the fast marching method," in *Energy Minimization Methods in Computer Vision and Pattern Recognition*. Berlin, Germany: Springer, 2003, pp. 35–51.
[18] K. Museth, "VDB: High-resolution sparse volumes with dynamic topology," *ACM Trans. Graph.*, vol. 32, no. 3, pp. 1–22, 2013.
[19] M. G. Linguraru *et al.*, "Computer-aided renal cancer quantification and classification from contrast-enhanced ct via histograms of curvature-related features," in *Proc. Annu. Int. Conf. IEEE Eng. Med. Biol. Soc.*, Sep. 2009, pp. 6679–6682.
[20] X.-X. Qu, S.-X. Liu, and F. Wang, "A new ray tracing technique for cross-hole radar traveltime tomography based on multistencils fast marching method and the steepest descend method," in *Proc. 15th Int. Conf. Ground Penetrating Radar*, Jun. 2014, pp. 503–508.
[21] S. Li, A. Vladimirsky, and S. Fomel, "First-break traveltime tomography with the double-square-root eikonal equation," *Geophysics*, vol. 78, no. 6, pp. U89–U101, 2013.
[22] E. Treister and E. Haber, "Full waveform inversion guided by travel time tomography," *SIAM J. Sci. Comput.*, vol. 39, no. 5, pp. S587–S609, 2017.
[23] X. Zhang and R. Bording, "Fast marching method seismic traveltimes with reconfigurable field programmable gate arrays," *Can. J. Explor. Geophys.*, vol. 36, no. 1, pp. 60–68, 2011.
[24] U. B. Waheed and T. Alkhalifah, "A fast sweeping algorithm for accurate solution of the tilted transversely isotropic Eikonal equation using factorization," *Geophysics*, vol. 82, no. 6, pp. WB1–WB8, 2017.
[25] M. W. Jones, J. A. Baerentzen, and M. Sramek, "3D distance fields: A survey of techniques and applications," *IEEE Trans. Vis. Comput. Graphics*, vol. 12, no. 4, pp. 581–599, Jul. 2006.
[26] 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 *Proc. 21st Telecommun. Forum*, Nov. 2013, pp. 685–688.
[27] S. Bak, J. McLaughlin, and D. Renzi, "Some improvements for the fast sweeping method," *SIAM J. Sci. Comput.*, vol. 32, no. 5, pp. 2853–2874, 2010.
[28] W. Jeong and R. Whitaker, "A fast iterative method for Eikonal equations," *SIAM J. Sci. Comput.*, vol. 30, no. 5, pp. 2512–2534, 2008.
[29] L. Yatziv, A. Bartesaghi, and G. Sapiro, "O(N) implementation of the fast marching algorithm," *J. Comput. Phys.*, vol. 212, no. 2, pp. 393–399, 2005.
[30] P. Kotas, R. Croce, V. Poletti, V. Vondrak, and R. Krause, "A massive parallel fast marching method," in *Domain Decomposition Methods in Science and Engineering XXII*, T. Dickopf, M. J. Gander, L. Halpern, R. Krause, and L. F. Pavarino, Eds. Cham, Switzerland: Springer, 2016, pp. 311–318.
[31] M. Detrixhe, F. Gibou, and C. Min, "A parallel fast sweeping method for the Eikonal equation," *J. Comput. Phys.*, vol. 237, pp. 46–55, Mar. 2013.
[32] K. Crane, C. Weischedel, and M. Wardetzky, "Geodesics in heat: A new approach to computing distance based on heat flow," *ACM Trans. Graph.*, vol. 32, no. 5, pp. 152:1–152:11, Oct. 2013.
[33] Y.-T. Zhang, H.-K. Zhao, and J. Qian, "High order fast sweeping methods for static Hamilton–Jacobi equations," *J. Sci. Comput.*, vol. 29, no. 1, pp. 25–56, Oct. 2006.
[34] A. M. Popovici and J. A. Sethian, "3-D imaging using higher order fast marching traveltimes," *Geophysics*, vol. 67, no. 2, pp. 604–609, 2002.
[35] S. Fomel, S. Luo, and H. Zhao, "Fast sweeping method for the factored Eikonal equation," *J. Comput. Phys.*, vol. 228, no. 17, pp. 6440–6455, Sep. 2009.
[36] E. Treister and E. Haber, "A fast marching algorithm for the factored Eikonal equation," *J. Comput. Phys.*, vol. 324, pp. 210–225, Nov. 2016.
[37] S. Cacace, E. Cristiani, and M. Falcone, "Can local single-pass methods solve any stationary Hamilton–Jacobi–Bellman equation?" *SIAM J. Sci. Comput.*, vol. 36, no. 2, pp. 570–587, 2014.
[38] J. A. Sethian, "Fast marching methods," *SIAM Rev.*, vol. 41, no. 2, pp. 199–235, 1999.
[39] S. Osher and J. A. Sethian, "Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton–Jacobi formulations," *J. Comput. Phys.*, vol. 79, no. 1, pp. 12–49, 1988.
[40] J. A. Sethian, *Level Set Methods and Fast Marching Methods*. Cambridge, U.K.: Cambridge Univ. Press, 1999.
[41] J. A. Sethian, "An analysis of flame propagation," Ph.D. dissertation, Dept. Math., Univ. California, Berkeley, CA, USA, Jun. 1982.
[42] E. Dijkstra, "A note on two problems in connection with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.
[43] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton Univ. Press, 1957.
[44] A. K. Jain, L. Hong, and S. Pankanti, "Random insertion into a priority queue structure," Stanford Univ., Stanford, CA, USA, Tech. Rep. Stan-Cs-74-460, 1974.
[45] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, 1987.
[46] C. Rasch and T. Satzger, "Remarks on the O(N) implementation of the fast marching method," *IMA J. Numer. Anal.*, vol. 29, no. 3, pp. 806–813, Nov. 2008.
[47] Y. Tsai, L. Cheng, S. Osher, and H. K. Zhao, "Fast sweeping algorithms for a class of Hamilton–Jacobi equations," *SIAM J. Numer. Anal.*, vol. 41, no. 2, pp. 659–672, 2003.
[48] S. Kim, "An O(N) level set method for Eikonal equations," *SIAM J. Sci. Comput.*, vol. 22, no. 6, pp. 2178–2193, 2001.
[49] E. Rouy and A. Tourin, "A viscosity solutions approach to shape-from-shading," *SIAM J. Numer. Anal.*, vol. 29, no. 3, pp. 867–884, 1992.
[50] W.-K. Jeong and R. Whitaker, "A fast iterative method for a class of Hamilton–Jacobi equations on parallel systems," School Comput., Univ. Utah, Salt Lake City, UT, USA, Tech. Rep. UUCS-07-010, 2007.
[51] J. V. Gómez, D. Álvarez, S. Garrido, and L. Moreno, "Fast marching-based globally stable motion learning," *Soft Comput.*, vol. 21, no. 10, pp. 2785–2798, 2017.
[52] R. Valencia, M. Morta, J. Andrade-Cetto, and J. M. Porta, "Planning reliable paths with pose SLAM," *IEEE Trans. Robot.*, vol. 29, no. 4, pp. 1050–1059, Aug. 2013.
[53] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard, "A tutorial on graph-based SLAM," *IEEE Intell. Transp. Syst. Mag.*, vol. 2, no. 4, pp. 31–43, Feb. 2010.
[54] W. Liao, K. Rohr, and S. Wörz, "Globally optimal curvature-regularized fast marching for vessel segmentation," in *Proc. Int. Conf. Med. Image Comput. Comput.-Assist. Intervent.*, 2013, pp. 550–557.
[55] L. Flórez-Valencia *et al.*, "Coronary artery segmentation and stenosis quantification in CT images with use of a right generalized cylinder model," in *Proc. MICCAI Workshop 3D Cardiovascular Imag., MICCAI Segmentation Challenge*, 2012, pp. 1–8.
[56] M. Niemeijer, J. Staal, B. van Ginneken, M. Loog, and M. Abramoff, "Comparative study of retinal vessel segmentation methods on a new publicly available database," *Proc. SPIE*, vol. 5370, pp. 648–657, May 2004.

[57] S. Ahmed, S. Bak, J. McLaughlin, and D. Renzi, "A third order accurate fast marching method for the Eikonal equation in two dimensions," *SIAM J. Sci. Comput.*, vol. 33, no. 5, pp. 2402–2420, 2011.
[58] A. Valero-Gómez, J. V. Gómez, S. Garrido, and L. Moreno, "The path to efficiency: Fast marching method for safer, more efficient mobile robot trajectories," *IEEE Robot. Autom. Mag.*, vol. 20, no. 4, pp. 111–120, Dec. 2013.
[59] S. Bougleux, G. Peyré, and L. D. Cohen, "Image compression with anisotropic triangulations," in *Proc. IEEE 12th Int. Conf. Comput. Vis.*, Sep. 2009, pp. 2343–2348.
[60] C.-Y. Kao, S. Osher, and Y.-H. Tsai, "Fast sweeping methods for static Hamilton–Jacobi equations," *SIAM J. Numer. Anal.*, vol. 42, no. 6, pp. 2612–2632, 2005.
[61] L. Janson and M. Pavone, "Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions," in *Robotics Research*. Cham, Switzerland: Springer, 2016, pp. 667–684.
[62] D. S. Yershov and E. Frazzoli, "Asymptotically optimal feedback planning using a numerical Hamilton–Jacobi–Bellman solver and an adaptive mesh refinement," *Int. J. Robot. Res.*, vol. 35, no. 5, pp. 565–584, 2016.

J. V. GÓMEZ received the degree in industrial engineering from the Technical University of Madrid, in 2011, and the master's degree in robotics and automation and the Ph.D. degree in robotics from the Universidad Carlos III de Madrid, Spain, in 2012 and 2015, respectively. His main research interests include robot navigation and environment modeling.

D. ÁLVAREZ received the degree in communications electronics engineering from the Technical University of Madrid, in 2009, and the master's degree in robotics and automation and the Ph.D. degree in robotics from the Universidad Carlos III de Madrid, Madrid, Spain, in 2011 and 2016, respectively. In 2010, he joined the Department of Systems Engineering and Automation, Universidad Carlos III de Madrid, where he has been involved in several robotics projects. His research interests include path planning, mobile manipulation, and environment modeling.

S. GARRIDO received the degree in mathematics from the Complutense University of Madrid, in 1979, and the degree in physics and the Ph.D. degree from the Universidad Carlos III de Madrid, Madrid, Spain, in 1955 and 2000, respectively. In 1997, he joined the Department of Systems Engineering and Automation, Universidad Carlos III de Madrid, where he has been involved in several mobile robotics projects. His research interests include mobile robotics, mobile manipulators, environment modeling, path planning, and mobile robot global localization problems.

L. MORENO received the degree in automation and electronics engineering and the Ph.D. degree from the Universidad Politécnica de Madrid, Madrid, Spain, in 1984 and 1988, respectively, where he was an Associate Professor, from 1988 to 1994. In 1994, he joined the Department of Systems Engineering and Automation, Universidad Carlos III de Madrid, Madrid, where he has been involved in several mobile robotics projects. His research interests include mobile robotics, mobile manipulators, environment modeling, path planning, and mobile robot global localization problems.