An efficient data structure to solve front propagation problems

Olivier Bokanowski, Emiliano Cristiani, Hasnaa Zidani

▶ **To cite this version:**

Olivier Bokanowski, Emiliano Cristiani, Hasnaa Zidani. An efficient data structure to solve front propagation problems. Journal of Scientific Computing, 2009, 10.1007/s10915-009-9329-6 . inria-00273977v1

HAL Id: inria-00273977

https://inria.hal.science/inria-00273977v1

Submitted on 16 Apr 2008 (v1), last revised 22 Oct 2009 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An efficient data structure to solve front propagation problems

O. Bokanowski[*], E. Cristiani[†], H. Zidani[‡]

Abstract

In this paper we develop a general efficient sparse storage technique suitable to coding front evolutions in $d \geq 2$ space dimensions. This technique is mainly applied here to deal with deterministic target problems with constraints, and solve the associated minimal time problems. To this end we consider an Hamilton-Jacobi-Bellman equation and use an adapted anti-diffusive Ultra-Bee scheme. We obtain a general method which is faster than a full storage technique. We show that we can compute problems that are out of reach by full storage techniques (because of memory). Numerical experiments are provided in dimension $d = 2, 3, 4$.

Moreover, the application of the sparse storage technique to the implementation of the Fast Marching Method for the eikonal equation, in dimensions 2 and 3, is discussed.

AMS Classification: 65M06, 49L99.
Keywords: Ultra-Bee scheme, narrow band methods, Fast Marching method, sparse grid, Hamilton-Jacobi-Bellman equations, front propagations, time optimal problem

1 Introduction

This paper is devoted to the study of an efficient numerical technique for coding front propagation in high dimension, coming from the study of the reachable sets of a minimum time problem. More precisely, we consider a dynamical system described by the following differential equation:

$$\begin{aligned} &\dot{\xi}(t) = f(\xi(t), a(t)), \quad \text{for a.e. } t \geq 0, \\ &\xi(0) = x, \end{aligned} \tag{1}$$

where $a \in L^\infty([0, \infty[; A)$, $f : \mathbb{R}^d \times A \to \mathbb{R}^d$ is the dynamics, and $A \subset \mathbb{R}^m$ is a compact set of "admissible controls". Let $\mathcal{C} \subset \mathbb{R}^d$ be a closed set. We are interested in finding the set of all the initial conditions $x \in \mathbb{R}^d$ from which it is possible to find an admissible trajectory ξ_x (solution of (1)) reaching the target $\mathcal{C}$ before time $t \geq 0$:

$$\text{Capt}_t(\mathcal{C}) = \{x \in \mathbb{R}^d;\ \exists(\xi_x, a) \text{ satisfying (1) and } \exists 0 \leq \tau \leq t,\ \xi(\tau) \in \mathcal{C}\}.$$

It is already known that the reachable set (or Capture Basin) $\text{Capt}_t(\mathcal{C})$ can be interpreted as a "burnt region" for a flame front propagation problem, see [11] and references therein. To

[*]Lab. JLL, Universités Paris 6 & 7, 175 rue chevaleret, 75013 Paris. boka@math.jussieu.fr
[†]Projet Commands, Ensta & Inria Saclay, 32 Bd Victor, 75739 Paris. emiliano.cristiani@ensta.fr
[‡]Projet Commands, Ensta - Inria Saclay, 32 Bd Victor, 75379 Paris Cx 15, Hasnaa.Zidani@ensta.fr

describe the shape of flame front it is usual to use the level curves of the viscosity solution of an Hamilton-Jacobi equation. This approach allows to obtain results on the stability properties and the asymptotic behavior of the flame front. It has been also used for computing purposes. We refer to [16, 17, 19] or [11] for numerical methods based on finite differences scheme (ENO, WENO, ...) or on Semi-Lagrangian scheme. In all these contributions, the front is interpreted as the 0-level set of a continuous solution of an evolutive Hamilton-Jacobi (HJ) equation.

Here, we want to use a slightly different approach. We define a "reachability function" $\theta(t,x)$ that takes value 0 if there exists a trajectory ξ_x that can reach the target $\mathcal{C}$ before time t and starting from x, otherwise takes the value 1, i.e.:

$$\theta(t,x) := \begin{cases} 0 & \text{if } \exists \tau \in [0,t],\ \exists a \in L^\infty([0,\tau],A),\ \xi_x^a(\tau) \in \mathcal{C} \\ 1 & \text{otherwise} \end{cases} \tag{2}$$

where ξ_x^a is the solution of (1) associated to a and starting from x at time 0. The function θ is obviously discontinuous. From [2, 12], we can characterize θ as the unique lower semi-continuous (l.s.c.) solution of the HJB equation (see Section 2 for the assumptions on f):

$$\begin{cases} \theta_t(t,x) + \max\limits_{\substack{a\in A \\ \lambda\in\Lambda(x)}} \{-\lambda f(x,a)\cdot\nabla\theta(t,x)\} = 0 & t>0,\ x\in\mathbb{R}^d \\ \theta(0,x) = \varphi(x) & x\in\mathbb{R}^d \end{cases} \tag{3}$$

where φ is given by

$$\varphi(y) = \begin{cases} 0 & \text{if } y\in\mathcal{C} \\ 1 & \text{otherwise,} \end{cases} \qquad \text{and } \Lambda(x) := \begin{cases} [0,1] & \text{if } x\in\mathcal{C}, \\ \{1\} & \text{otherwise.} \end{cases}$$

At every time $t \geq 0$, the function $\theta(t,\cdot)$ takes only values 0 and 1. This is different from classical level set methods which approximate the front as the 0-level set of a more regular function $v(t,\cdot)$ (see Fig. 1).

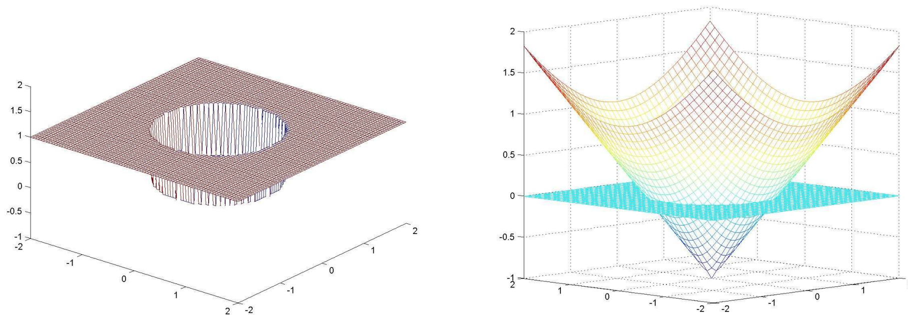

Figure 1: Discontinuous approach (left) vs. level set approach (right)

Then it is not necessary to compute this function everywhere: we should concentrate the numerical effort only around the front. For this, we need two ingredients, the first one is a scheme able to compute the front with a good accuracy without diffusion and the second one (as important as the first one) is an efficient way for stocking and handling the calculations on a sparse grids. Let us mention the paper [1] where structures for handling computations for propagating interfaces are studied.

Here, for the scheme, we consider an adaptation of the Ultra-Bee scheme used by Desprès and Lagoutière for linear advection and conservation laws [9, 10, 13, 14]. An extension in order to treat HJB equations was proposed in [7], numerically showing very good anti-diffusive properties in one and two dimensions (see also [3] for computation of Capture Basin sets). Convergence results for some 1-dimensional HJB equations have been obtained in [6, 4], where an error bound of order Δx (the spatial mesh step) in L^1 norm for general bounded l.s.c. initial data is obtained, also proving anti-diffusive properties in particular cases.
The anti-diffusive behavior of the Ultra-Bee scheme, in the case of $\theta \in \{0, 1\}$, leads to numerical values which belong to $[0, 1]$ and where the front is located in a small narrow band. In the 2-dimensional case, the computed values $\theta^n_{i,j}$ approximate $\frac{1}{\Delta x \Delta y} \int_{I_{ij}} \theta(t_n, x, y)\, dxdy$. A classical way to code the matrix (θ^n_{ij}) leads to Fig. 2, where values are stored in white for 0, black for 1, and gray for intermediary values. However, instead of coding the full matrix (θ^n_{ij}) we can use a sparse matrix where only the values $\theta^n_{ij} \in]0, 1[$ are coded, as well as first neighboring values (0 and 1), leading to a sparse structure as illustrated in Fig. 2.

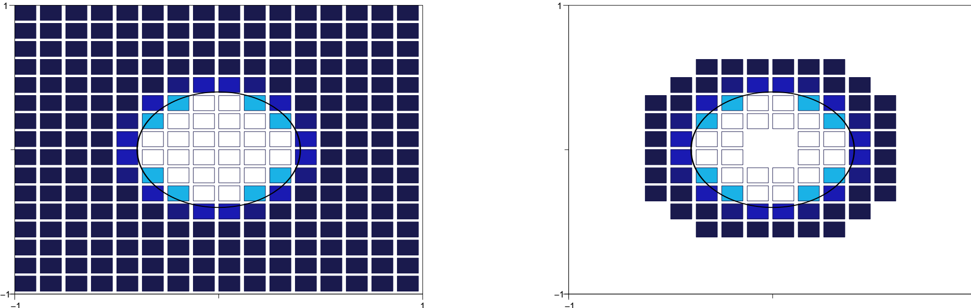

Figure 2: Full (left) and sparse (right) matrix coding

The central idea of the present paper is to use an adapted sparse matrix structure in order to code the discrete front of θ^n. Note that the idea of using a sparse structure was tested for instance in [15] in 2d and 3d (a quadtree structure was also tested in [5]). However these attempts where computationally not growing in $O(N)$ where N is the size of the narrow band. Here we shall reach the optimal $O(N)$ growth by using an adapted sparse matrix structure.
Note that coding only the narrow band in a list structure makes more difficult the search of neighboring values of a given cell. In the usual level-set approach [18] (even when computation is restricted in a narrow band around the front) a full matrix is used for storage purposes, and this makes access of a given cell value (reading/writing) cost $O(1)$. However the use of full grid limits the size of the computation since the access to memory on hard disk is too costly, in particular when looking for neighboring values. Here our method enables to access larger sizes by making evolve only the narrow band itself. Storage of results on a full matrix on the hard disk can be allowed, for instance when computing the minimal time function as explained in Section 4 (this will not be costly in CPU time).

The paper is organized as follows. In Section 2 we recall the Ultra-Bee scheme we use. Section 3 is devoted to the presentation of the sparse structure. Two methods will be proposed and compared. Section 4 is devoted to some numerical examples. Finnally, in Section 5, we discuss the application of the storage technique to the Fast marching method.

2 Preliminary results

Let $\mathcal{C}$ be a convex closed subset of $\mathbb{R}^d$, and A be a non empty compact convex subset of $\mathbb{R}^m$. Let $f : \mathbb{R}^d \times A \longrightarrow \mathbb{R}^d$ satisfies:

(A1) f is a continuous function. There exists two functions $g(\xi)$ and $h(\xi)$ such that $f(\xi, u) = g(\xi) + h(\xi) \cdot u$. There exists $c_o \geq 0$ s.t. $\sup_{a\in A} |f(\xi, a)| \leq c_o(1 + |\xi|)$.

Moreover, for every $R > 0$, there exists $L_R > 0$, such that:

$$\forall \xi, z \in B(0, R), \quad \sup_{a\in A} |f(\xi, a) - f(z, a)| \leq L_R|\xi - z|.$$

Under assumption (A1), the function θ defined in (2) is the l.s.c. bilateral solution of the HJB equation (3), see [12, 2]. Moreover, from θ it is possible to recover the minimal time function $\mathcal{T}$ defined by

$$\mathcal{T}(x) := \min\{t \geq 0 \mid \exists a \in L^\infty(0, t; A), \ \text{ s.t. the trajectory } \xi^a_x \text{ associated to } a \text{ satisfies } \xi^a_x(t) \in \mathcal{C}\},$$

and we have the following lemma.

Lemma 2.1 *Under assumption (A1), for every $x \in \mathbb{R}^d$, we have:*

$$\mathcal{T}(x) = \min\{t \geq 0, \ \theta(x, t) = 0\},$$

with the convention that $\mathcal{T}(x) = +\infty$, whenever $\{t \geq 0, \ \theta(x, t) = 0\} = \emptyset$.

2.1 Ultra-Bee scheme

We recall here the Ultra Bee scheme for solving an HJB equation (3). This scheme was first studied for advection equations with constant velocity [10] (in this context, the scheme is exact). A generalization of the scheme to advection equations with changing sign velocity is suggested in [7], where the properties and the convergence result are proved in dimension 1. The adaptation of the scheme to solve HJB equations is done in [3]. Now we present directly the algorithm in dimension 2.

Let $\Delta t > 0$ be a constant time step, and $t_n := n\Delta t$ for $n \geq 0$. Let $\Delta x > 0$ be a step size of a spatial grid, and let $\xi_{i,j} := (x_i, y_j) := (i\ \Delta x, j\ \Delta x)$ denote a uniform mesh, with $i, j \in \mathbb{Z}$. Consider also

$$x_{i+\frac{1}{2}} = (i + \frac{1}{2})\,\Delta x, \quad y_{j+\frac{1}{2}} = (j + \frac{1}{2})\,\Delta x \quad \text{ and } \quad I_{ij} :=]x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}[\times]y_{j-\frac{1}{2}}, y_{j+\frac{1}{2}}[.$$

The Ultra-Bee scheme aims at computing a numerical approximation of the averages $\overline{\vartheta}^n_{i,j} := \frac{1}{\Delta x^2}\int_{I_{ij}} \vartheta(t_n, \xi)\, d\xi$, for $i, j \in \mathbb{Z}$. Since the function $\vartheta(t_n, \cdot)$ takes only values in $\{0, 1\}$, their averages $\overline{\vartheta}^n_{i,j}$ contain the information of the discontinuities localization. The UB-HJB scheme takes the following form.

$$\frac{V^{n+1}_{i,j} - V^n_{i,j}}{\Delta t} + \max_{\substack{a\in A\\ \lambda\in\Lambda(\xi_{i,j})}} \Big(\lambda f(\xi_{i,j}, a)[D^{\mathrm{UB}}_a V^n]_{ij}\Big) = 0, \tag{4a}$$

with the initialization:

$$V^0_{i,j} := \frac{1}{\Delta x^2}\int_{I_{ij}} \varphi(\xi) d\xi, \tag{4b}$$

and where $[D_a^{\rm UB}V^n]$ will play the role of a consistent approximation of the term "$D_x\vartheta(t_n,\cdot)$". To define precisely this approximation, let us first introduce, for $j\in\mathbb{Z}$, the fluxes $W^R_{j-\frac12} := [F^R(W,\mu)]_{j-\frac12}$ and $W^L_{j+\frac12} := [F^L(W,\mu)]_{j+\frac12}$ for a one-dimensional vector $(W_j)_j$ and for real numbers $(\mu_j)_{j\in\mathbb{Z}}\subset[-1,1]$, as follows:

- If $\mu_j\geq 0$, set

$$W^L_{j+\frac12} := \begin{cases} \min(\max(W_{j+1},b_j^+),B^+_{j,}) & \text{if } \mu_j>0\\ W_{j+1} & \text{if } \mu_j=0 \text{ and } W_j\neq V^n_{j-1}\\ W_j & \text{if } \mu_j=0 \text{ and } W_j=V^n_{j-1},\end{cases}$$

- If $\mu_j\leq 0$, set

$$W^R_{j-1/2} := \begin{cases} \min(\max(W_{j-1},b_j^-),B^-_{j}) & \text{if } \mu_j<0\\ W_{j-1} & \text{if } \mu_j=0 \text{ and } W_j\neq V^n_{j+1}\\ W_j & \text{if } \mu_j=0 \text{ and } W_j=V^n_{j+1},\end{cases}$$

  where b_j^+, b_j^-, B_j^+ and B_j^- are defined by (6a)-(6b)).

- If $\mu_j\geq 0$ and $\mu_{j+1}>0$, set $W^R_{j+\frac12} := W^L_{j+\frac12}$.
- If $\mu_{j+1}\leq 0$ and $\mu_j<0$, set $W^L_{j+\frac12} := W^R_{j+\frac12}$.
- If $\mu_j<0$ and $\mu_{j+1}>0$, then set

$$W^R_{j+\frac12} := \begin{cases} W_{j+1} & \text{if } W_{j+1}=W_{j+2}\\ W_j & \text{otherwise}\end{cases} \quad\text{and } W^L_{j+\frac12} := \begin{cases} W_j & \text{if } W_j=W_{j-1}\\ W_{j+1} & \text{otherwise.}\end{cases} \tag{5}$$

With

$$\text{if } \mu_j>0,\quad \begin{cases} b_j^+ := \max(W_j,W_{j-1})+\dfrac{1}{\mu_j}(W_j-\max(W_j,W_{j-1})),\\ B_j^+ := \min(W_j,W_{j-1})+\dfrac{1}{\mu_j}(W_j-\min(W_j,W_{j-1})),\end{cases} \tag{6a}$$

$$\text{if } \mu_j<0,\quad \begin{cases} b_j^- := \max(W_j,W_{j+1})+\dfrac{1}{|\mu_j|}(W_j-\max(W_j,W_{j+1})),\\ B_j^- := \min(W_j,W_{j+1})+\dfrac{1}{|\mu_j|}(W_j-\min(W_j,W_{j+1})).\end{cases} \tag{6b}$$

Now, for every $j\in\mathbb{Z}$ and every $a\in A$, we set

$$\nu^1_{i,j}(a) := \frac{\Delta t}{\Delta x}f_1(\xi_{i,j},a),\quad \nu^2_{i,j}(a) := \frac{\Delta t}{\Delta x}f_2(\xi_{i,j},a),$$

the "local CFL" number. In the following we assume that the mesh sizes satisfy the following condition:

$$\max(\frac{\Delta t}{\Delta x}f_1(\xi_{i,j},a),\frac{\Delta t}{\Delta x}f_2(\xi_{i,j},a))\leq 1. \tag{7}$$

Now, we define UB-HJB scheme as follows (see [7, 6]).

Algorithm UB-HJB :

Initialization: We compute the initial averages $(V^0_{ij})_{i,j\in\mathbb{Z}}$ as in (4b).

Loop: For $n \geq 0$,

For $a \in A$, for $j \in \mathbb{Z}$ set

$$V^{n,1}_{i,j} = V^n_{i,j} - \frac{\Delta t}{\Delta x} f_1(\xi_{i,j}, a) \left(F^L(V^n_{\cdot,j}, \nu^1_{\cdot,j}(a))_{i+\frac{1}{2}} - F^R(V^n_{\cdot,j}, \nu^1_{\cdot,j}(a))_{i-\frac{1}{2}} \right), \quad \forall i \in \mathbb{Z},$$

where $V^n_{\cdot,j} = (V^n_{i,j})_{i\in\mathbb{Z}}$. Then we evolve in the x_2-direction using, for $i \in \mathbb{Z}$,

$$V^{n+1}_{i,j}(a) = V^{n,1}_{i,j} - \frac{\Delta t}{\Delta x} f_2(\xi_{i,j}, a) \left(F^L(V^{n,1}_{i,\cdot}, \nu^2_{i,\cdot}(a))_{j+\frac{1}{2}} - F^R(V^{n,1}_{i,\cdot}, \nu^2_{i,\cdot}(a))_{j-\frac{1}{2}} \right), \quad \forall j \in \mathbb{Z},$$

where $V^{n,1}_{i,\cdot} = (V^{n,1}_{i,j})_{j\in\mathbb{Z}}$.

Set $V^{n+1}_{i,j} := \min_{a\in A} \left(V^{n+1}_{i,j}(a) \right)$.

Remark 2.2 *A general version of the Ultra-Bee scheme, for 1-dimensional problem, is given in [6], for any l.s.c. initial condition in* $L^1_{loc}(\mathbb{R})$*. Here, the algorithm is specified to the case of an initial condition taking values only in* $\{0, 1\}$*.*

Lagoutière [13] proved the very interesting property that the Ultra-Bee scheme advects exactly a particular class of step functions, in the case of constant advection. For instance, for 2-dimensional problems, let u_0 be such that $V^0_{i,j}$ initialized as in (4b) belongs to the following space $\mathcal{S}$:

$$\mathcal{S} := \{(V_{i,j}),\ \forall (a, b) \in \{0, 1, 2\},\ V_{3i+a,3j+b} = V_{3i,3j}\}.$$

Consider the Ultra-Bee scheme for $v_t + f \cdot \nabla v = 0$ where $f = (f_1, f_2) = const$ is a constant advection vector of $\mathbb{R}^2$. Then, assuming the CFL condition $\max(|f_1|\frac{\Delta t}{\Delta x}, |f_2|\frac{\Delta t}{\Delta x}) \leq 1$ holds, we have $\forall i, j$ and $n \geq 0$:

$$V^n_{i,j} = \frac{1}{\Delta x^2} \int_{I_{ij}} v(t_n, \xi)\ d\xi$$

where $v(t, \xi) = v_0(\xi_1 - f_1 t, \xi_2 - f_2 t)$ is the exact solution (see also [13] for more general functions that are exactly advected). This exact advection property, which corresponds to an "anti-dissipative" behavior of the Ultra-Bee scheme, motivates us for using it in front propagation problems such as (3). We refer to Després and Lagoutière[10] for other interesting properties of the Ultra-Bee scheme.

3 The dynamic data structure

Several algorithms for the numerical solution of front propagation problems are based on the "narrow band method" (see, among others, [18, 17]) which consists in focusing the numerical effort only in a narrow band around the front.
The implementation of narrow band methods is usually very easy because it relies on the fact that you can stock both a dynamic vector which contains the nodes they are to be computed and a (n-dimensional) matrix where the computation really takes place. Storing the whole n-dimensional matrix is needed in order to have access to the neighbors of a node and to store the evolution of the front at any time. Although this technique is fast, it is not always suitable for problems in high dimension (≥ 4) or involving a large number of nodes due to the huge space

required to stock the matrix. This is the case, for example, of image processing on large images or optimal control problems with many state variables.

In this section we propose a new implementation which only requires to stock the nodes around the front. We aim to compute the solution on the nodes stocked in a dynamic structure without using an additional matrix which is usually used to find the respective position of the nodes. The main difficulty here is to find neighbors of the node which is actually being computed.
For example, in 2D we can stock all the node around the front in a unique linked list (see Fig. 3) in which every element has the fields i, j and *value*. Being at the node (i, j), to find the

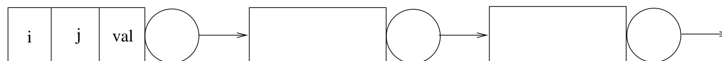

Figure 3: a unique linked list containing the nodes

node $(i \pm 1, j \pm 1)$ we can search through the list starting from the beginning until we find it. However this method can be very slow, especially in high dimension in case the data structure has a huge number of nodes.
Our goal is to propose efficient storage techniques allowing to find rapidly neighbor values of a given node.

3.1 First method (sparse semi-dynamic)

Let us consider a rectangular domain in which the computation is performed. Let M be the $N_1 \times N_2$ matrix which corresponds to the domain. We store a vector $p = (p_1, \dots p_{N_1})$ of pointers such that every pointer p_i is the beginning of a list which corresponds to the i-th line (see Fig. 4). Every list is made by elements which contain only the index j and the value of the node we

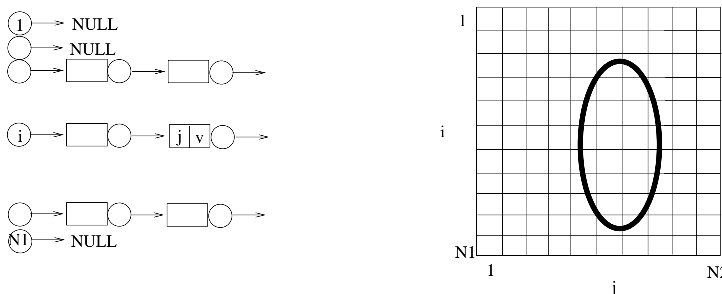

Figure 4: a semi-dynamic linked structure storing the nodes

want to stock in the structure. The elements are ordered by increasing j's. Note that if a line does not contain any node around the front the corresponding pointer points to NULL.
In this way we stock only nodes we are interested in and, at the same time, we have a direct access to every line of the matrix, so that we can quickly and easily search for neighbors.
In the 3D version of the algorithm the full matrix (not stored) has $N_1 \times N_2 \times N_3$ elements and the vector of pointers becomes a $N_1 \times N_2$ matrix. As before, every pointer is the beginning of a list. Note that the required memory is not greater than the case when we store all the nodes in a unique list because in our method every element has only the fields *lastindex* and *value*. The first $n - 1$ indexes can be recovered by the name of the pointer we are currently using to run throughout the structure.

3.2 Second method (sparse-dynamic)

The second method we propose is a modification of the first one and it is optimized for memory requirements. The basic idea is that it is not necessary to store pointers which point to NULL, so that in 2D the structure is made by a structure like that in Fig. 5. In this case the vector

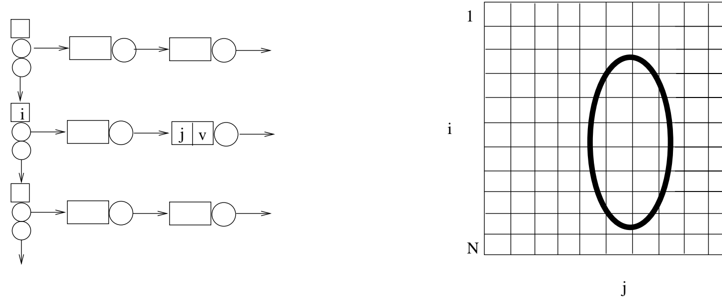

Figure 5: a sparse (completly) dynamic linked structure storing the nodes

containing the pointers is substituted by a linked list similar to those containing the values of the nodes. Of course this modification yields to a more complicated code. In order to preserve the efficiency, every time we change line to visiting, we store a pointer at the beginning of the former line so we do not lose the direct access to the line we will be interested for during computation.

3.3 Managing the data structure

When using a data structure which does not contain all the nodes of the grid we have to pay attention to not loose any important informations. In particular we have to be sure that either the data structure contains all the nodes used for computation or we can recover the value of the node even if it is not stored in the data structure. It seems that solving equation (3) by means of Ultra-Bee scheme is a very good choice to take the maximum advantage of the sparse structures introduced above.

We know that the function θ takes value in the set $\{0, 1\}$, so the numerical solution V takes value in the set $[0, 1]$ because it represents an average of the function θ. So, at every time step, the front is localized in the region $\{\xi_I \ : \ V(\xi_I) \in (0, 1)\}$ and around this region there is a 1-node thick band where $V = 0$ (internal to the front) and a 1-node thick band where $V = 1$ (external to the front). *Only these nodes are stored in the data structure*, but they are not enough to perform computation with the Ultra-Bee scheme. The not stored values can be recovered by the existing ones by the following strategy: if a node is both a neighbor of a node with value 0 (resp., 1) and it is not stored in the data structure, then its value is 0 (resp., 1).

Let S^n be the set of nodes in the data structure at time step n. Let ξ_I be a generic node of the grid and $Neigh(\xi_I)$ be the set of the $3^d - 1$ first neighbors of the node ξ_I (d being the dimension of the problem). At the beginning of every time step n, S^n must be updated in order to follow the evolution of the front. Then we set

$$S^{n+1} = S^n \cup \{\xi_I \ : \ Neigh(S^n) \backslash S^n\}$$

Note that if it is known that the evolution of the front is monotone (*i.e.* the front is always increasing or decreasing, as it is the case for the minimum time problem) the insertion procedure

can be limited to the nodes in the neighborhood of nodes with values 1 (resp., 0), depending if the front is increasing (resp., decreasing).
By the numerical experiments, it seems that this operation is the slowest part of the algorithm. In order to maintain the data structure as slim as possible, after computation (at every time step) we remove from the data structure the set of nodes

$$\{\xi_I \ : \ V(\xi_I) = 0 \text{ or } 1 \text{ and } V(\xi_I) = V(\xi_J) \text{ for every } \xi_J \in Neigh(\xi_I)\}.$$

In order to show how easy can be the use of the proposed dynamic structure , we report the function which finds the value of a node (in any dimension) that is either present in the structure or not. Let us suppose that we need to compute the value at the node ξ_I and we want to know the value of its neighbor $\xi_J \in Neigh(\xi_I)$.

```
node *bus=pointer[J_1][J_2]...[J_{d-1}];
while (bus && bus->index<J_d) bus=bus->next;
if (!bus) return(V(ξ_I));
if (bus->index==J_d) return(bus->value);
return(V(ξ_I));
```

4 Numerical Experiments

In this section we provide some numerical tests to validate the proposed methods and to give a precise measure of the gain in memory usage and CPU time needed for computation. In particular we test the classical "full grid" method (the numerical scheme is applied at every node of the grid at every time step), the sparse semi-dynamic and the sparse-dynamic methods. The algorithms are written in C++ and they run on a PC equipped with a PENTIUM IV processor at 2.60 GHz, 256MB RAM and 512KB cache. As it can be seen in Table 3, starting from a 200^3 grid this machine uses the hard disk (in addition to RAM memory) to store data and make computation so the CPU times increases dramatically. In order to overcome this difficulty we perform computation also on another machine equipped with a AMD Opteron processor at 2.40 GHz, 8GB RAM and 1MB cache.

We suppose that the target is the ball $\mathcal{C} = B(0, 0.5)$ centred in 0 and with radius 0.5 (in $\mathbb{R}^d$). The considered HJB equation is:

$$\begin{cases} \theta_t(t,x) + \max_{\lambda\in\Lambda(x)}(-\lambda \mathbf{1}_d . \nabla\theta(t,x)) = 0 & t \in (0,T], x \in \mathbb{R}^d, \\ \theta(0,x) = \begin{cases} 0 & \text{if } x \in \mathcal{C} \\ 1 & \text{otherwise.} \end{cases} \end{cases}$$

where $\mathbf{1}_d := (1,\ldots,1)^T \in \mathbb{R}^d$. This equation is solved in the box$=[-2,2]^d$, with $d = 2,3,4$. The CFL condition is fixed to 0.9 and the final time is $T = 0.5$. The CPU time reported in the following tables regards just the main computation and not the initialization of the structure and the save of results because they are example-dependent and they will be included in the tests in the next section. In Tables 1-3 we summarized the results.
We can see that in these tests the proposed methods are faster than the full method in any case. It is important to remark from the above tables that the computations with the sparse semi-dynamic structure are performed in $O(N)$ where N is the number of the cells in S^0 (nodes of the initial front). This is not the case for the dynamic structure neither for the full method.
As expected, the first advantage of proposed algorithms is for speed, and essantially for memory requirement. They allows to run codes which are completely out of reach for existing algorithms.

nodes	full 256MB	sparse semi-dyn	sparse-dyn	time steps	#nodes in S^0
50^2	0	0	0	7	132
100^2	0.04	0.02	0.01	14	272
200^2	0.35	0.06	0.05	28	532
400^2	2.86	0.24	0.21	56	1068
800^2	23.94	0.94	0.87	112	2124

Table 1: CPU times for 2D tests

nodes	full 256MB	sparse semi-dyn	sparse-dyn	time steps	# nodes S^0
4x	8.4x	3.9x	4.1x	2x	1.9x

Table 2: rate of CPU time's growth for 2D tests

nodes	full256MB	full8GB	semi-dyn	sparse-dyn	ts	nodes in S^0
25^3	0.02	0.01	0.02	0.02	4	432
50^3	0.38	0.24	0.18	0.18	7	2016
100^3	6.10	4.39	1.51	1.85	14	8016
200^3	x	64.29	14.41	22.55	28	31416
400^3	x	1175	146.78	343.25	56	125968

Table 3: CPU times for 3D tests

nodes	full256MB	full8GB	semi-dyn	sparse-dyn	ts	nodes in S^0
8x	∞x	18.3x	10.2x	15.2x	2x	4x

Table 4: rate of CPU time's growth for 3D tests

nodes	semi-dyn	time steps	nodes in S^0
25^4	0.49	4	3024
50^4	*6.39	7	23552
100^4	123.97	14	179472

Table 5: CPU times for 4D tests

nodes	semi-dyn	time steps	nodes in S^0
16x	19.4x	2x	7.6x

Table 6: rate of CPU time's growth for 4D tests

5 Some applications

In this section we tested the Ultra-Bee scheme and the semi-dynamic structure on some real problems. We use a last-generation PC with an Intel dual core processor at 2.13 GHz and 2GB

RAM. Note that the code is not parallelized.
We use a variable time step Δt which is computed at each time iteration to fit the CFL condition with respect to the nodes currently involved in computation.

TEST 1 (Van der Pol oscillator)
The Van der Pol oscillator is a classical problem in electronics and dynamical systems. The controlled dynamics is

$$\begin{cases} \dot{x}_1(t) = x_2 \\ \dot{x}_2(t) = -x_1 + x_2(\mu - x_1^2) + u(t) \end{cases}$$

where $\mu = 1$ and $u \in [-1, 1]$. The goal is to drive the system to the target $\mathcal{C} = \{x_1^2 + x_2^2 \leq 0.2^2\}$ in minimal time choosing the optimal control variable on $(0, t)$. We solved the equation (3) with $T = 3.1$ in the box $[-1.8, 1.8]^2$. In Fig. 6-left we show the level sets of the value function

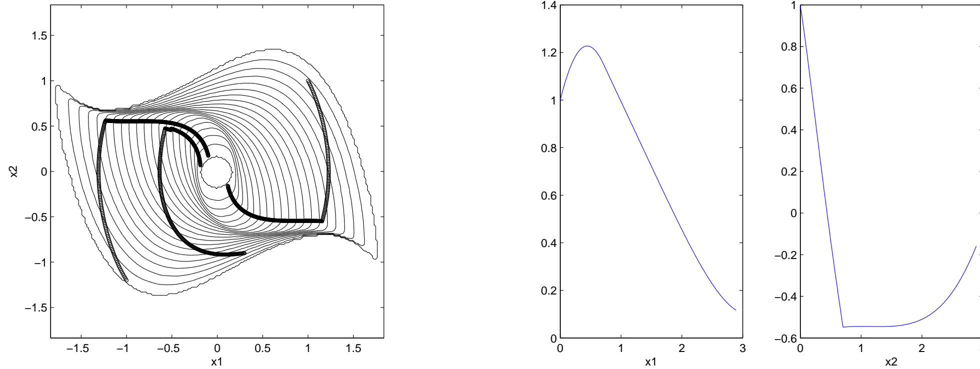

Figure 6: Test 1. Level sets of the value function and three optimal trajectories

and three optimal trajectories to reach target starting from the points $(1, 1)$, $(-1, -1.2)$ and $(0.3, -0.9)$ (for the reconstruction of optimal trajectories, see [2, Appendix by Falcone]). We can see that at final time T the front did not cover the domain completely, this means that it is not possible to reach the target in time T from all points of the considered box. The number of iterations was 393. The CPU time for the whole computation and saving the results on a 200^2 grid was 0.95 seconds. The CPU time for the reconstruction of the optimal trajectory is almost 0 seconds.
In Fig. 6-right we show the two components of the optimal trajectory starting from $(1, 1)$, the optimal control for this trajectory is in the form of bang-bang control:

$$u(t) = \begin{cases} -1, & t \in [0, \hat{t}] \\ 1, & t \in [\hat{t}, t_{fin}] \end{cases}$$

where $\hat{t} = 0.71$ and $t_{fin} = 2.87$ (these values are the same to those computed by shooting method)

TEST 2. In this test the dynamics is

$$\begin{cases} \dot{x}_1(t) = x_2 \\ \dot{x}_2(t) = x_3 \\ \dot{x}_3(t) = u \end{cases}$$

where the control u can vary in the set $\{-1,0,1\}$. This dynamic corresponds to the one-dimensional problem $x''' = u$. We solved equation (3) in $[-3,3]^3$ until a final time $T = 10$. The computational domain is divided in 50^3 nodes. The target is

$$\mathcal{C} = \{(x_1,x_2,x_3) \in \mathbb{R}^3 \ : \ x_1^2 + x_2^2 + x_3^2 \leq 0.4^2\}$$

Results are shown in Fig. 7 and Fig. 8. The number of iterations was 306 and the CPU time was 44 seconds for the whole computation and saving the results. As in the TEST 1, we can see that at the final time the front has touched the boundary of the domain of computation but it did not cover it completely, this means that it is not possible to reach the target in time T from any point of the considered box. By Fig. 8 we can see that the optimal trajectory to the target expects two switches for the control ($u = -1$ at the beginning, then $u = 1$ and finally $u = -1$ again).

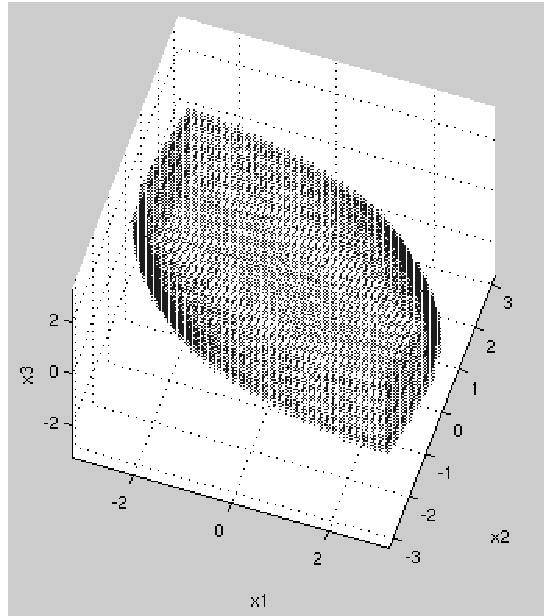

Figure 7: Test 2. The front at the final time T

TEST 3. In this test the dynamics is

$$\begin{cases} \dot{x}(t) = v_x \\ \dot{y}(t) = v_y \\ \dot{v}_x(t) = \frac{2}{5}R\cos(u) + \frac{C(M_x - x)}{((M_x-x)^2+(M_y-y)^2)^{\frac{3}{2}}+\varepsilon} \\ \dot{v}_y(t) = \frac{2}{5}R\sin(u) + \frac{C(M_y - y)}{((M_x-x)^2+(M_y-y)^2)^{\frac{3}{2}}+\varepsilon} \end{cases}$$

where the control u can vary in the set $[0,2\pi)$ (the set is discretized in 16 points) and the control R can vary in $\{0,1\}$. This dynamics corresponds to a motion in a plane of a particle subjected to two forces. The first one is the engine, its modulus is $\frac{2}{5}$ and it can be directed in any direction, the second one is the gravitational attraction of a mass located in (M_x, M_y). We chose $C = 0.5$,

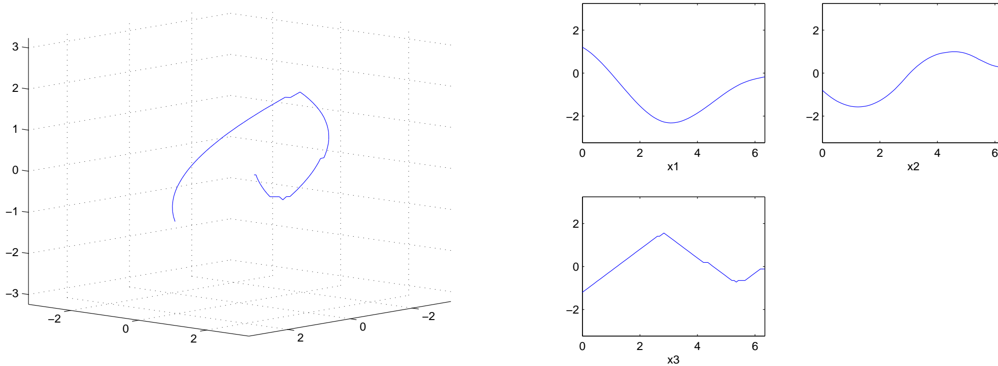

Figure 8: Test 2. The optimal trajectory starting from the point (1.2,-0.8,-1.2)

$M_x = 1.5$, $M_y = 0.5$ and $\varepsilon = 0.01$. We solved equation (3) in $[-2,2]^2$ until a final time $T = 7$. The computational domain is divided in 20^4 nodes. The target is

$$\mathcal{C} = \{(x, y, v_x, v_y) \in \mathbb{R}^4 \ : \ -0.8 \le x \le 0.8,\ 1.2 \le y \le 1.8,\ v_x^2 + v_y^2 \le 0.25\}$$

In addition, there is an obstacle O in the domain given by $O = O_1 \cup O_2 \cup O_3$, where

$$\begin{aligned} O_1 &= [-0.6, 0.6] \times [-0.6, -0.3], \\ O_2 &= [-0.6, -0.4] \times [-1.3, -0.6], \\ O_3 &= [0.4, 0.6] \times [-1.3, -0.6]. \end{aligned}$$

Results are shown in Fig. 9 and Fig. 10. The number of iterations was 202 and the CPU time was 11 minutes and 51 seconds for the whole computation and saving the results.
We can see how the optimal trajectory tries to get an advantage by the attractive mass in (M_x, M_y) in two ways. It is used both to speed up the journey and to slow down the particle near the target (note that the 4-dimensional target requires a small velocity on the rectangle).

6 Application to the Fast Marching method for the Eikonal equation

In this section we adapt the data structures introduced above to the numerical solution of the Eikonal equation

$$\begin{cases} c(x)|\nabla \mathcal{T}(x)| = 1 & \mathbb{R}^d \backslash \Omega \\ \mathcal{T}(x) = 0 & x \in \partial\Omega \end{cases} \tag{8}$$

where $c(x)$ is a given Lipschitz continuous and strictly positive function and Ω a closed subset of $\mathbb{R}^d$. Equation (8) can be seen as a particular minimum time problem, choosing the dynamics f as $f(x, a) = c(x)a$ and $A = B(0, 1)$. This choice means that it is possible to steer the system in any direction of the space with velocity $c(x)$, the target being Ω. The function $\mathcal{T}$ represents here the minimum time to reach the target starting from the point x. Equation (8) is also related to front propagation problems whenever the front evolves with speed $c(x)$ in the normal direction to the front itself (see [18]). The front Γ_t at any time $t \ge 0$ can be recovered by $\Gamma_0 = \Omega$ and $\Gamma_t = \{x \in \mathbb{R}^p : \mathcal{T}(x) = t\}$ for $t > 0$.

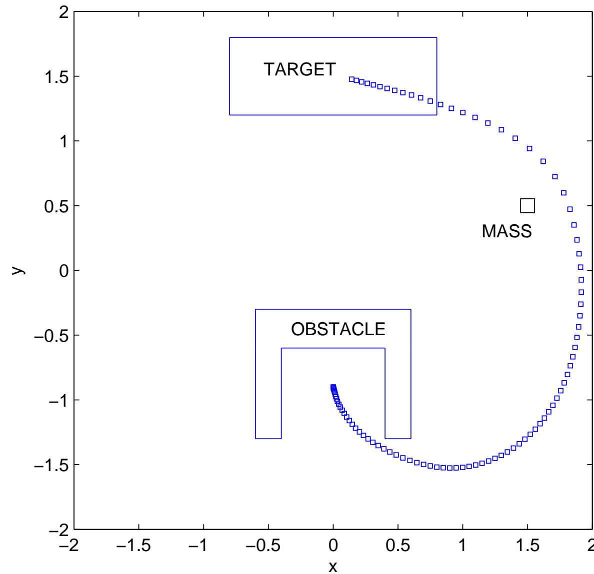

Figure 9: Test 3. An optimal trajectory starting from the point $(0, -1, 0, 0)$

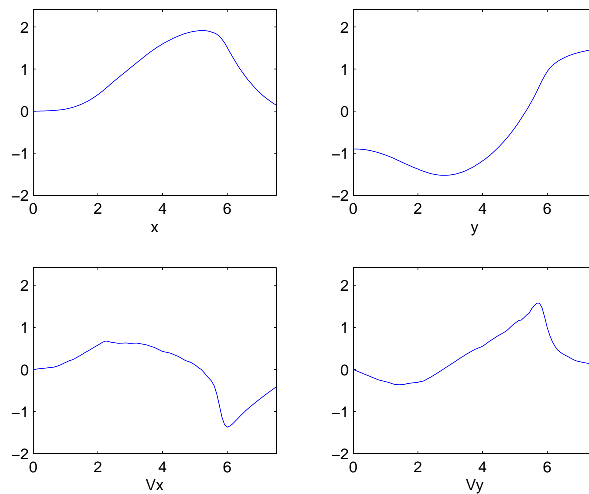

Figure 10: Test 3. The four components of the optimal trajectory starting from the point $(0, -1, 0, 0)$

The Fast Marching (FM) method was introduced by J. A. Sethian in [17] (see also [18, 20]) to solve numerically equation (8). Due to its great efficiency, it was used in different fields like, for example, optical physics, image processing, path planning and Shape-from-Shading.
It appears natural to extend our approach to this kind of numerical method which can be considered as the best for the Eikonal equation, comparing also the solution of the same physical problem via equation (3) (using Ultra Bee scheme) and via equation (8) (using finite difference

scheme).

6.1 Sparse Fast Marching method

The FM method is based on a first-order up-wind finite difference scheme. It was introduced to speed up the convergence of the classical iterative algorithm (in which the computation is performed on all over the grid at each iteration).
The FM technique consists in dividing the grid in three sets of nodes, *Accepted*, *Narrow Band* and *Far*. The Accepted zone is the region where the front already passed through, and their nodes are no more computed because their values are correct (within the discretization error). The Far zone is the region not yet reached by the front, their nodes are not yet computed and their values are set to $\mathcal{T} = +\infty$. The Narrow Band zone is the region where the computation takes place and it is updated at each step to follow the front during its evolution. At each step the node in the Narrow Band with the minimal value exits the Narrow Band and becomes Accepted. The algorithm ends when all the node of the grid are Accepted. (See also [8] for recent developpements.)
The computational cost of the FM method is of order $N \log N$ where N is the total number of grid nodes. The term $\log N$ comes from the fact that the list containing the Narrow Band must be ordered in some way to pick easily the node with the minimum value at each iteration. It is obtained, for example, using an heap structure to store the Narrow Band.
The easiest way to implement the classical FM method is to store a full matrix where the computation takes place and an ordered dynamic structure (list or heap) to stock the nodes in the Narrow Band. This procedure can results in a lack of memory whenever the number of nodes is very large. (see [18] for applications).
The dynamic structures we presented in the previous chapters can be adapted to deal with FM method. Every node can be inserted and removed in a fast way as described above whenever it is requested by the FM technique. Again, searching for the values of the neighbors is an easy task as for the evolutive case. The only difference here is that the data structures described in Sections 3.1 and 3.2 are not ordered by increasing or decreasing $\mathcal{T}$ and then they are not suitable for searching the minimum value in short time. To overcome this difficulty we have considered the following strategy (although others are possible): we double the stocked data by defining a static vector containing the nodes in the Narrow Band, ordered as an heap.

Managing the data structure. The data structure we use, at a given computational step, contains all the nodes in the Narrow Band and some Far and Accepted nodes (hence it does not coincide with the Narrow Band). In particular, a Far node enters the structure when its value must be computed for the very first time. On the other hand, it does not exit the structure when it becomes Accepted because its value can be necessary to future computations. It is important to note that in the case of FM method, we cannot recover the value of a node if it is no more present in the dynamic structure as we did in section 3.3. An accepted node is removed from the structure only when all its neighbors are Accepted too, because only at this stage we are sure it is no more useful.
We want to point out that it is not efficient to erase nodes from the dynamic structure at each step because only few nodes will be really removed. On the other hand, keeping the no-more-useful nodes in the dynamic structure is not efficient (since a larger dynamic structure implies a slower neighbor research).
To speed up the computation, we call the procedure which removes nodes from the Narrow Band every $N^{\frac{d-1}{d}}$ steps where N is the total number of nodes and d is the dimension of the problem.

The number $N^{\frac{d-1}{d}}$ is expected to be in average the size of the Narrow Band and this choice seems to optimize the CPU time and memory requirements.

Numerical experiments. We solved the equation (8) with $c(x) \equiv 1$ and $d = 2$ and 3 in the square $[-1, 1]^2$. The initial front is $B(0, 0.2)$. We compare the *full* method (where a full matrix is stored) and the *semi-dynamic sparse* method (which store nodes only in the semi-dynamic structure).
Note that in the computations for the full method is also only performed in the Narrow Band at every iteration. As a consequence we cannot expect the sparse method to be faster than the classical approach.
The results are summarized in Table 7. (computations done on a last-generation PC with an Intel dual core processor at 2.1 GHz and 2GB RAM.)
As we can see, the sparse approach gives results which are 3 times longer than the classical approach for the 2D test (resp. about 5 times longer for the 3D test). Still the scaling with increasing N is correct.
Moreover, it allows to complete the computation in reasonable time in cases where the full method would fails because of memory.

nodes	2D full	2D sparse
500^2	0.1	0.3
1000^2	0.4	1.4
2000^2	1.8	5.9
4000^2	8.2	25.9
8000^2	41.6	139.3
16000^2	x	802.1

nodes	3D full	3D sparse
50^3	0.08	0.3
100^3	0.9	3.8
200^3	10.3	53.4
400^3	104.8	541.0
600^3	x	2032.7

Table 7: FM method, CPU times for 2D and 3D tests

References

[1] D. Adalsteinsson and J. A. Sethian. A fast level set method for propagating interfaces. *J. Comput. Phys.*, 118(2):269–277, 1995.

[2] M. Bardi and I. Capuzzo-Dolcetta. *Optimal control and viscosity solutions of Hamilton-Jacobi-Bellman equations*. Systems and Control: Foundations and Applications. Birkhäuser, Boston, 1997.

[3] 0. Bokanowski, S. Martin, R. Munos, and H. Zidani. An anti-diffusive scheme for viability problems. *Applied Numerical Mathematics*, 56(Issue 9, in Numerical Methods for Viscosity Solutions and Applications):1135–1254, 2006.

[4] O. Bokanowski, N. Forcadel, and H. Zidani. Convergence of a non-monotone scheme for Hamilton-Jacobi-Bellman equations with discontinuous initial data. Preprint Inria (http://hal.inria.fr/inria-00267644/fr/), 2007.

[5] O. Bokanowski, N. Megdich, and H. Zidani. An adaptative antidissipative method for optimal control problems. *Arima*, 5:256–271, 2006.

[6] O. Bokanowski, N. Megdich, and H. Zidani. Convergence of a non-monotone scheme for Hamilton-Jacobi-Bellman equations with discontinuous initial data. Preprint Inria (http://hal.inria.fr/inria-00193157/fr/), 2007.

[7] O. Bokanowski and H. Zidani. Anti-diffusive schemes for linear advection and application to Hamilton-Jacobi-Bellman equations. *J. Sci. Computing*, 30(1):1–33, 2007.

[8] E. Cristiani and M. Falcone. Fast semi-Lagrangian schemes for the Eikonal equation and applications. *SIAM J. Numer. Anal.*, 45(5):1979–2011 (electronic), 2007.

[9] B. Després and F. Lagoutière. A non-linear anti-diffusive scheme for the linear advection equation. *C. R. Acad. Sci. Paris, Série I, Analyse numérique*, (328):939–944, 1999.

[10] B. Després and F. Lagoutière. Contact discontinuity capturing schemes for linear advection and compressible gas dynamics. *J. Sci. Comput.*, 16:479–524, 2001.

[11] M. Falcone, T. Giorgi, and P. Loreti. Level sets of viscosity solutions : some applications to fronts and rendez-vous problems. *SIAM J. Applied Mathematics*, 54(5):1335–1354, 1994.

[12] Hélène Frankowska. Lower semicontinuous solutions of Hamilton-Jacobi-Bellman equations. *SIAM J. Control Optim.*, 31(1):257–272, 1993.

[13] F. Lagoutière. *Modélisation mathématique et résolution numérique de problèmes de fluides compressibles à plusieurs constituants.* PhD thesis, University of Paris VI, Paris, 2000.

[14] F. Lagoutière. A non-dissipative entropic scheme for convex scalar equations via discontinuous cell reconstruction. *C. R. Acad. Sci.*, 338(7):549–554, 2004.

[15] N. Megdich. *Méthodes Anti-dissipatives pour les Equations de Hamilton Jacobi Bellman.* PhD thesis, University of Paris VI, Paris, 2008.

[16] S. Osher and J. A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. *J. Comput. Phys.*, 79(1):12–49, 1988.

[17] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proc. Nat. Acad. Sci. U.S.A.*, 93(4):1591–1595, 1996.

[18] J. A. Sethian. *Level set methods and fast marching methods*, volume 3 of *Cambridge Monographs on Applied and Computational Mathematics.* Cambridge University Press, Cambridge, second edition, 1999. Evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science.

[19] C.-W. Shu. High order ENO and WENO schemes for computational fluid dynamics. In *High-order methods for computational physics*, volume 9 of *Lect. Notes Comput. Sci. Eng.*, pages 439–582. Springer, Berlin, 1999.

[20] J.N. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Tran. Automatic. Control*, 40:1528–1538, 1995.