J Sci Comput (2010) 42: 251–273
DOI 10.1007/s10915-009-9329-6

An Efficient Data Structure and Accurate Scheme to Solve Front Propagation Problems

O. Bokanowski · E. Cristiani · H. Zidani

Received: 14 November 2008 / Revised: 28 August 2009 / Accepted: 22 September 2009 /
Published online: 6 October 2009

Abstract In this paper, we are interested in some front propagation problems coming from control problems in d-dimensional spaces, with $d \geq 2$. As opposed to the usual level set method, we localize the front as a discontinuity of a characteristic function. The evolution of the front is computed by solving an Hamilton-Jacobi-Bellman equation with discontinuous data, discretized by means of the antidissipative Ultra Bee scheme.

We develop an efficient dynamic storage technique suitable for handling front evolutions in large dimension. Then we propose a fast algorithm, showing its relevance on several challenging tests in dimension $d = 2, 3, 4$. We also compare our method with the techniques usually used in level set methods. Our approach leads to a computational cost as well as a memory allocation scaling as $O(N_{nb})$ in most situations, where N_{nb} is the number of grid nodes around the front. Moreover, we show on several examples the accuracy of our approach when compared with level set methods.

Keywords Ultra Bee scheme · Narrow band method · Sparse matrices · Data storage · Hamilton-Jacobi-Bellman equations · Front propagation · Level set methods

O. Bokanowski (✉)
Laboratoire Jacques-Louis Lions (UMR 7598), Université Paris 6, 75005 Paris, France
e-mail: boka@math.jussieu.fr

O. Bokanowski
UFR Mathématiques, Université Paris 7, 175 rue Chevaleret, 75013 Paris, France

E. Cristiani
CEMSAC, Università di Salerno, Salerno, Italy
e-mail: emiliano.cristiani@gmail.com

E. Cristiani
IAC-CNR, Rome, Italy

H. Zidani
Projet Commands, Ensta - Inria Saclay, 32 Bd Victor, 75379 Paris Cx 15, France
e-mail: Hasnaa.Zidani@ensta.fr

1 Introduction

This paper is devoted to the study of an efficient numerical technique for coding front propagation in high dimension coming from some optimal control problems.

The idea of level set formulation to propagate curves and surfaces has been introduced in [25] by Osher and Sethian. The advantages of this approach are well known by now. It treats self-intersections, topological changes, kinks, and it is easily extended to capture hypersurfaces in $\mathbb{R}^d$, $d \geq 1$.

Let us consider a closed moving front Γ_t, $t \geq 0$, and let Ω_t be the closed region that Γ_t encloses (i.e. here Ω_t is closed and such that $\Gamma_t = \partial\Omega_t$). The level set method aims to find a function $u(t,x)$ such that at each time t the front Γ_t can be provided by the 0-level set of the function $u(t,\cdot)$, and we have:

$$\begin{aligned} u(t,x) = 0 \quad &\Leftrightarrow \quad x \in \Gamma_t, \\ u(t,x) < 0 \quad &\Leftrightarrow \quad x \in \mathrm{Int}(\Omega_t), \\ u(t,x) > 0 \quad &\Leftrightarrow \quad x \in \Omega_t^c. \end{aligned}$$

Moreover, the function $x \mapsto u(t,x)$ is uniformly continuous and it increases as the distance between x and Γ_t increases. It is known [29] that u can be obtained by solving a time-dependent Hamilton-Jacobi (HJ) equation with continuous initial datum φ vanishing only on the initial front Γ_0, $\varphi < 0$ in $\mathrm{Int}(\Omega_0)$, $\varphi > 0$ in Ω_0^c, and φ is a strictly non-decreasing function of the distance to Γ_0. Many numerical studies have been carried out to construct stable, accurate and efficient methods. See [18, 25, 26, 30] for a description of such methods on regular grids, and [1, 2] on triangular meshes. It is known that in general, once initialized φ as the signed distance function to Γ_0, it is impossible to maintain the level set function as the signed distance function to the moving interface in the advection step. Flat and/or steep regions develop as the interface moves rendering the computation and the localization of the front inaccurate. For this reason, it is necessary to reset the level set function, at regular time intervals, to be the signed distance function to the interface. This step of reinitialization was investigated in [17, 27, 31].

As pointed out in [3], an important drawback of the level set approach "stems from the expense; by embedding the interface in $\mathbb{R}^d$ as the level set of a $d+1$-dimensional function, considerable computational labor is required per time step". To overcome this difficulty, Adalsteinsson and Sethian [3] suggest a localization of the level set method. This method allows to compute the evolution of the level set function only in a neighborhood around the front. Another fast local level set method has been also proposed by Peng et al. in [27]. However, in [3] as well in [27] a full d-dimensional matrix is stored, which limits the methods.

In our work, we use a different approach (see Fig. 1). We seek a discontinuous function $\vartheta(t,x)$ which takes only values in $\{-1, 1\}$

$$\begin{aligned} \vartheta(t,x) = -1 \quad &\Leftrightarrow \quad x \in \Omega_t, \\ \vartheta(t,x) = 1 \quad &\Leftrightarrow \quad x \in \Omega_t^c. \end{aligned}$$

The notion of lower semicontinuous (l.s.c.) viscosity solution [4, 16] leads also to a characterization of ϑ by means of a Hamilton-Jacobi-Bellman (HJB) equation with discontinuous initial datum φ given by

$$\varphi(y) = \begin{cases} -1 & \text{if } y \in \Omega_0, \\ 1 & \text{otherwise.} \end{cases}$$

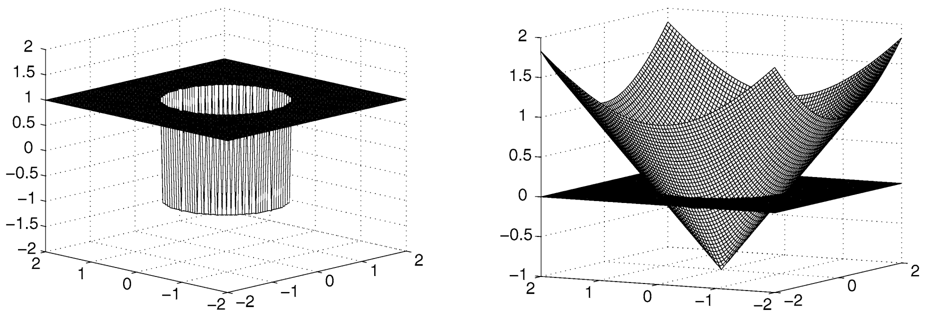


Fig. 1 Discontinuous approach (*left*) vs. level set approach (*right*)

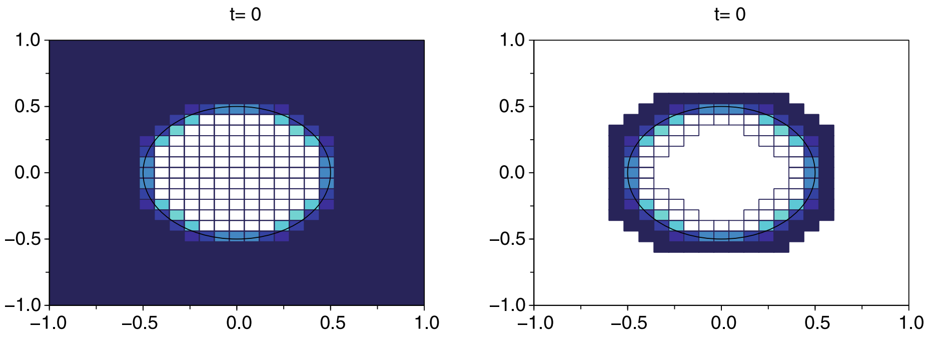


Fig. 2 Full (*left*) and sparse (*right*) matrix coding

Thus it is natural to concentrate the numerical effort only around the front separating the area of -1 values (Ω_t) from the region of 1 values. 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 to store and handle *only* the nodes around the front.

The scheme we consider here is an adaptation of the Ultra Bee scheme used by Desprès and Lagoutiè for linear advection and conservation laws [14, 15, 19, 20]. An extension in order to treat HJB equations was proposed in [12], numerically showing very good anti-diffusive properties in one and two dimensions (see also [9] for computation of the Capture Basin). Convergence results for 1-dimensional HJB equations have been obtained in [8, 11], 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.

In the 2-dimensional case, the Ultra Bee scheme computes values $V_{i,j}^n$ which approximate $\frac{1}{\Delta x \Delta y}\int_{I_{ij}} \vartheta(t_n, x, y)\, dx\, dy$, where $I_{i,j}$ is a cell associated to (i, j) and t_n is a discrete time. Since ϑ takes values in $\{-1, 1\}$, the discrete approximations $V_{i,j}^n$ will belong to the interval $[-1, 1]$.

The full matrix (V_{ij}^n) is represented in Fig. 2 (left), where values are drawn in white for -1, black for 1, and gray for intermediary values. However, instead of coding the matrix (V_{ij}^n) we can use a sparse matrix where only the values $V_{ij}^n \in]-1, 1[$ are coded, as well as their first neighboring values, leading to a sparse structure as illustrated in Fig. 2 (right).

In this paper we use an adapted sparse matrix structure in order to store the discrete front. Note that the idea of using a sparse structure was tested for instance in [22] for $d = 2, 3$

(a quadtree structure was also tested in [10]). However, in these attempts the CPU time is not growing as $O(N_{nb})$, where N_{nb} is the number of nodes in the "narrow band" around the front.

In the literature, several techniques for storing sparse matrices have been studied [5, 28]. These techniques are usually designed to handle standard algebraic computations (for instance, product of matrices). For our purposes, a "good" storage technique should be fast for finding a nonempty value as well as its first neighboring values, and for adding (or removing) nonempty values in the sparse matrix.

Here we shall propose a new storage technique for sparse matrices, and compare it with known storage techniques strategies.

While the worst case complexity of the scheme is no better than dense implementations, in practice it often achieves significant savings $O(N_{nb})$, for both memory and CPU time. The advantage of the sparse storage technique we propose is that it can handle larger data sets than other known methods.

The combination of the Ultra Bee scheme and the sparse storage allows us to obtain an efficient algorithm that is able to treat "high-dimensional" problems. We also obtain on some specific examples better results than a traditional level set solver. Finally we mention that the algorithm is quite simple and does not need special features such as reinitialization.

The paper is organized as follows. In Sect. 2 we recall the HJB equation for ϑ and the Ultra Bee scheme. Section 3 is devoted to the presentation of the sparse dynamic data structure, and to extensive comparisons with other techniques. Section 4 is devoted to numerical examples for the HJB equation. Comparison between the discontinuous approach and the level set approach will be also given.

More applied examples will be treated in a forthcoming work.

2 Preliminaries

2.1 Motivations and Setting of the Problem

Let Ω_0 be a compact set of $\mathbb{R}^d$, and $f : \mathbb{R}^+ \times \mathbb{R}^d \times \mathbb{R}^m \to \mathbb{R}^d$ be a continuous function. We are interested by fast and efficient numerical methods for Hamilton-Jacobi equations of the form:

$$\begin{cases} \vartheta_t(t,x) + \max_{a\in A(x)}\{-f(t,x,a)\cdot\nabla\vartheta(t,x)\} = 0, \quad & t>0,\ x\in\mathbb{R}^d, \\ \vartheta(0,x) = \varphi(x), \quad x\in\mathbb{R}^d, \end{cases} \tag{1}$$

where φ is given by

$$\varphi(x) = \begin{cases} -1 & \text{if } x\in\Omega_0, \\ 1 & \text{otherwise,} \end{cases}$$

and $A(x)$ is a nonempty compact subset of $\mathbb{R}^m$, for $m \geq 1$. The function ϑ is obviously discontinuous. Nevertheless, the solution of (1) is well defined by the notion of lower semi-continuous solution introduced by Barron and Jensen [6] (see also [4, 16]).

At each time $t > 0$, the boundary of the set of -1 values of $\vartheta(t,\cdot)$ represents the front Γ_t which results from the propagation of the initial front $\Gamma_0 := \partial\Omega_0$.

Equation (1) models several problems. Let us see some examples.

2.1.1 Advection Equation

The simplest situation consists in taking

$$A(x) = \{a\} \quad \text{and} \quad f(t,x,a) = f(t,x).$$

Then (1) can simply be rewritten as:

$$\vartheta_t(t,x) + f(t,x)\cdot\nabla\vartheta(t,x) = 0, \quad t>0,\ x\in\mathbb{R}^d, \qquad \vartheta(0,x) = \varphi(x), \quad x\in\mathbb{R}^d.$$

Here the region Ω_0 is advected with the velocity f.

2.1.2 Eikonal Equation

Let us denote by $\mathcal{B}(0,1)$ the closed unit ball of $\mathbb{R}^d$. If we set

$$A(x) = \mathcal{B}(0,1) \quad \text{and} \quad f(t,x,a) = F(x)a,$$

where F is a positive function, then we get the Eikonal equation

$$\vartheta_t(t,x) + F(x)\|\nabla\vartheta(t,x)\| = 0, \quad t>0,\ x\in\mathbb{R}^d, \qquad \vartheta(0,x) = \varphi(x), \quad x\in\mathbb{R}^d.$$

The framework of (1) includes also the case of anisotropic propagation. However, it does not include propagation problems with motion by mean curvature.

2.1.3 Target Problem: Capture Basin (or Backward Reachable Set)

Now we assume that $\mathcal{C} := \Omega_0$ is a *target*. We consider a dynamical system described by the following differential equation:

$$\begin{aligned} &\dot{\xi}(t) = f(\xi(t),\alpha(t)), \quad \text{for a.e. } t\ge 0,\\ &\xi(0) = x, \end{aligned} \tag{2}$$

where $a \in L^\infty([0,\infty[;A)$ and A is a compact subset of $\mathbb{R}^m$ (with $m\ge 1$). The Capture Basin (or backward reachable set) associated to $\mathcal{C}$ is defined as 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 (2), reaching the target $\mathcal{C}$ before time $t\ge 0$:

$$\mathrm{Capt}_t(\mathcal{C}) := \{x\in\mathbb{R}^d : \exists(\xi_x,\alpha) \text{ satisfying (2) and } \exists 0\le\tau\le t \text{ s.t. } \xi_x(\tau)\in\mathcal{C}\}.$$

We define the *reachability function* $\vartheta(t,x)$ by

$$\vartheta(t,x) := \begin{cases} -1 & \text{if } x\in\mathrm{Capt}_t(\mathcal{C}),\\ 1 & \text{otherwise.}\end{cases} \tag{3}$$

Under usual assumptions[1] on f, we can characterize ϑ as the unique lower semicontinuous solution of the following HJB equation (by adapting the arguments of [23], see also [22])

$$\begin{cases} \vartheta_t(t,x) + \max(0,\ \max_{a\in A}\{-f(x,a)\cdot\nabla\vartheta(t,x)\}) = 0, & t>0,\ x\in\mathbb{R}^d, \\ \vartheta(0,x) = \varphi(x), \quad x\in\mathbb{R}^d, \end{cases} \tag{4}$$

where

$$\varphi(x) := \begin{cases} -1 & \text{if } x\in\mathcal{C}, \\ 1 & \text{otherwise.} \end{cases}$$

Moreover, from ϑ it is possible to recover the minimal time function $\mathcal{T}$ defined by

$$\mathcal{T}(x) := \min\{t\ge 0 : \exists a\in L^\infty([0,t];A) \text{ s.t. } \xi_x(t)\in\mathcal{C}\}$$

by means of the following expression

$$\mathcal{T}(x) = \min\{t\ge 0,\ \vartheta(t,x) = -1\},$$

with the convention that $\mathcal{T}(x) = +\infty$, whenever $\{t\ge 0,\ \vartheta(t,x)=-1\} = \emptyset$. This result is straightforward and was already stated in the continuous level set setting [24].

2.1.4 Rendez-Vous Problem

Suppose we want to find out if we can reach the target exactly at time t and not only "before time t". Then instead of (3), we can consider the following definition:

$$\vartheta(t,x) := \begin{cases} -1 & \text{if } \exists\alpha\in L^\infty([0,t],A) \text{ s.t. } \xi_x(t)\in\mathcal{C}, \\ 1 & \text{otherwise.} \end{cases} \tag{5}$$

It is proved in [16] (see also [6]) that ϑ is the solution of the following HJB equation:

$$\begin{cases} \vartheta_t(t,x) + \max_{a\in A}\{-f(x,a)\cdot\nabla\vartheta(t,x)\} = 0, & t>0,\ x\in\mathbb{R}^d, \\ \vartheta(0,x) = \varphi(x), \quad x\in\mathbb{R}^d, \end{cases} \tag{6}$$

which is still is a particular case of (1).

Remark 2.1 For time optimal control problems, in order to reconstruct the optimal trajectories (see [4, Appendix by Falcone]) it is necessary to know the minimal time function $\mathcal{T}$ on all over the domain, and not only around the front at the final time. In practice we can compute the solution ϑ together with the minimum time function $\mathcal{T}$: the values of $\mathcal{T}$ need only to be saved on the hard disk during computation, before advancing the narrow band. These values are not needed to compute ϑ, and their storage on the hard disk do not reduce

[1](i) The function $f:\mathbb{R}^d\times A\to\mathbb{R}^d$ is continuous; (ii) $F(x) := \{f(x,a),\ a\in A\}$ is convex compact for all x; (iii) There exists $c_o\ge 0$ s.t. $\sup_{a\in A}|f(\xi,a)|\le c_o(1+|\xi|)$; (iv) For every $R\ge 0$, there exists $L_R\ge 0$, such that

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

the maximal admissible size of the problem which is limited by the RAM's size. (On the contrary, an approach which needs to store the full matrix to make computation will be limited by the RAM memory.) On going work based on the present approach (see [7]) focuses on minimal time function computation and optimal trajectory reconstruction for an optimal control problem including state constraints.

2.2 Ultra Bee Scheme

We recall here the Ultra Bee (UB) scheme for solving the HJB equation (1). This scheme was first studied for advection equations with constant velocity [15] (in this context, the scheme is exact). A generalization of the scheme to advection equations with changing-sign velocity is suggested in [12], where the properties and the convergence result are proved in dimension 1. The adaptation of the scheme to solve HJB equations is done in [9] (some convergence results are proved in [11]). 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 $i, j \in \mathbb{Z}$. Let us also define

$$x_{i+\frac{1}{2}} := \left(i + \frac{1}{2}\right)\Delta x, \qquad y_{j+\frac{1}{2}} := \left(j + \frac{1}{2}\right)\Delta x \quad \text{and}$$

$$I_{ij} := \left]x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}\right[\times \left]y_{j-\frac{1}{2}}, y_{j+\frac{1}{2}}\right[.$$

The Ultra Bee scheme aims at computing a numerical approximation of the averages $V^n_{i,j} := \frac{1}{\Delta x^2}\int_{I_{ij}} \vartheta(t_n, \xi)\, d\xi$, for $i, j \in \mathbb{Z}$. Since the function $\vartheta(t_n, \cdot)$ takes only values in $\{-1, 1\}$, their averages $V^n_{i,j}$ contain the information of the discontinuities localization. The UB-HJB scheme, for the discretization of (1), takes the following form (when there is no ambiguity, we shall omit the time dependence in f and denote $f(\xi_{ij}, a)$ instead of $f(t_n, \xi_{ij}, a)$) :

$$\frac{V^{n+1}_{i,j} - V^n_{i,j}}{\Delta t} + \max_{a\in A}\left(-f(\xi_{i,j}, a)\left[D^{\mathrm{UB}}_a V^n\right]_{ij}\right) = 0, \tag{7a}$$

with the initialization

$$V^0_{i,j} := \frac{1}{\Delta x^2}\int_{I_{ij}} \varphi(\xi)\, d\xi, \tag{7b}$$

where $[D^{\mathrm{UB}}_a V^n]$ will play the role of a consistent approximation of the term $\nabla\vartheta(t_n, \cdot)$. To define precisely this approximation, let us first introduce, for $j \in \mathbb{Z}$, the fluxes $W^R_j - \frac{1}{2}$ and $W^L_j + \frac{1}{2}$ for a one-dimensional vector $(W_j)_{j\in\mathbb{Z}}$ and for real numbers $(\mu_j)_{j\in\mathbb{Z}} \in [-1, 1]$, as follows:

- If $\mu_j \geq 0$, set

$$W^L_{j+\frac{1}{2}} := \begin{cases} \min(\max(W_{j+1}, b^+_j), B^+_{j,}) & \text{if } \mu_j > 0, \\ W_{j+1} & \text{if } \mu_j = 0 \text{ and } W_j \neq V^n_{j-1}, \\ W_j & \text{if } \mu_j = 0 \text{ and } W_j = V^n_{j-1}. \end{cases}$$

- 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 in (9a)–(9b).

- If $\mu_j \geq 0$ and $\mu_{j+1} > 0$, set $W^R_{j+\frac{1}{2}} := W^L_{j+\frac{1}{2}}$.
- If $\mu_{j+1} \leq 0$ and $\mu_j < 0$, set $W^L_{j+\frac{1}{2}} := W^R_{j+\frac{1}{2}}$.
- If $\mu_j < 0$ and $\mu_{j+1} > 0$, then set

$$W^R_{j+\frac{1}{2}} := \begin{cases} W_{j+1} & \text{if } W_{j+1} = W_{j+2}, \\ W_j & \text{otherwise} \end{cases} \quad \text{and} \quad W^L_{j+\frac{1}{2}} := \begin{cases} W_j & \text{if } W_j = W_{j-1}, \\ W_{j+1} & \text{otherwise.} \end{cases} \tag{8}$$

With

$$\text{if } \mu_j > 0, \quad \begin{cases} b_j^+ := \max(W_j, W_{j-1}) + \frac{1}{\mu_j}(W_j - \max(W_j, W_{j-1})), \\ B_j^+ := \min(W_j, W_{j-1}) + \frac{1}{\mu_j}(W_j - \min(W_j, W_{j-1})), \end{cases} \tag{9a}$$

$$\text{if } \mu_j < 0, \quad \begin{cases} b_j^- := \max(W_j, W_{j+1}) + \frac{1}{|\mu_j|}(W_j - \max(W_j, W_{j+1})), \\ B_j^- := \min(W_j, W_{j+1}) + \frac{1}{|\mu_j|}(W_j - \min(W_j, W_{j+1})). \end{cases} \tag{9b}$$

We also use in the sequel the notations F^R and F^L defined by:

$$F^L(W, \mu)_j + \frac{1}{2} := W^L_{j+\frac{1}{2}} \quad \text{and} \quad F^R(W, \mu)_{j-\frac{1}{2}} := W^R_{j-\frac{1}{2}}.$$

For every $(i, j) \in \mathbb{Z} \times \mathbb{Z}$ and every $\alpha \in A(\xi_{i,j})$, we define

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

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

$$\text{CFL} := \max\left(\frac{\Delta t}{\Delta x}|f_1(\xi_{i,j}, \alpha)|, \frac{\Delta t}{\Delta x}|f_2(\xi_{i,j}, \alpha)|\right) \leq 1. \tag{10}$$

Now, we define the UB-HJB scheme for (1) as follows (see [11, 12]).

Algorithm UB-HJB

Initialization: We compute the initial averages $(V^0_{ij})_{i,j\in\mathbb{Z}}$ as in (7b).
Loop: For $n \geq 0$,

For $\alpha \in A$, for $j \in \mathbb{Z}$, evolve in the x_1-direction

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

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

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

where $V^{n,1}_{i,\cdot} = (V^{n,1}_{i,j})_{j\in\mathbb{Z}}$.
Set $V^{n+1}_{i,j} := \min_{a\in A(\xi_{i,j})}(V^{n+1}_{i,j}(a))$.

Remark 2.2 A general version of the Ultra Bee scheme, for 1-dimensional problems, is given in [11], for any l.s.c. initial condition in $L^1_{\mathrm{loc}}(\mathbb{R})$. Here, the algorithm is specified to the case of an initial condition taking values only in $\{-1, 1\}$.

In [20], the author proved the very interesting property that the Ultra Bee scheme advects exactly a particular class of step functions, in the case of constant advection. We refer to [15, 20] for other properties.

3 Storage Data Structure

In this section we introduce the new data structure we will use with the Ultra Bee scheme to solve numerically (1) in high dimension. The goal is to construct a data structure which allows to save memory and to keep acceptable CPU times.

All the numerical results have been obtained on a 2×2GHz processor (AMD Turion(tm) 64 X2 Mobile Technology), 4GB RAM, on a laptop computer with Mandriva Linux release 2008.1. Codes are written in C++ (serial) with GNU gcc compiler.

3.1 Some Known Storage Techniques

First, we recall some classical storage methods. The word "full" will refer to the methods in which every element is stored (using a full matrix), while in the other methods sparse storage techniques are used.

Full Storage (FS): A full d-dimensional matrix is stored, it contains all the values of the function at each grid node. The computation is performed on all over the grid, at each time step. Searching for neighbors is very fast because we have a direct access to each element of the matrix. On the other hand the computation is slow, because at each time step we compute on the full matrix. This is the simplest method to be implemented because there is no need to track the front during its evolution.

Full & List Storage (FLS): this is the classical Narrow Band method (see [3]). A full matrix is stored as in *FS*, but it is also stored a dynamic linked list which contains the indexes of the nodes around the front, that is the zone where we want to compute the solution at each time step. The list must be updated in such a way it follows the front during its evolution. It is not needed to keep the list sorted in any way. Searching for neighbors is done using the full matrix.

List Storage (LS): A dynamic linked list containing the indexes and the values of the nodes around the front is stored, ordered for example in a row-wise fashion with respect to the full matrix (not stored here). Each element of the list has a pointer to the next and to the previous element, so the search for neighbors is done going forward and backward in the list. Moreover, in dimension 2, the indexes (i, j) of a node are "compressed" in only one index k to save memory defining $k = iN + j$ where N is the number of columns. This procedure to "compress" the indexes is easily generalizable to any dimension, provided N and d are not too large (otherwise the integer runs out of bits). This is the best method for memory allocation (see [28] for a detailed presentation and comparisons with other methods).

Compressed Row Storage (CRS): This is probably the most common method to store sparse matrices. We use here a slightly different version of the *CRS* method, substituting static vectors by linked lists to allow resize when needed. As in *LS*, the full matrix is not stored. See [21] for an explanation of the method in dimension $d \geq 2$ and an improvement of the basic method for matrix-matrix addition and multiplication.

3.2 The Sparse Semi-Dynamic Data Structure

Now, we introduce what we will call the *Sparse Semi-Dynamic* (*SSD*) structure, beginning from the case $d = 2$. Let us consider without loss of generality a square domain in which the computation is performed. Let M be the $N \times N$ matrix which corresponds to the domain. We store a (static) vector $p = (p_1, \ldots, p_N)$ of pointers such that every pointer p_i is the beginning of a linked list which corresponds to the i-th line of the matrix (not stored), see Fig. 3. Every list is made by elements which contain only the index j and the value of the nodes we want to store in the structure (that is the nodes around the front). 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 store 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 of a given node. For $d = 2$ this data structure corresponds to a dynamic version of the *CRS* method.

In the d-dimensional version of the algorithm the full matrix (not stored) has N^d elements. Then we consider a N^{d-1}-dimensional matrix of pointers. As before, every pointer is the beginning of a list which runs parallel to the last dimension of the matrix, containing all the nodes around the front (see Fig. 4). Note that in the *SSD* method every element of the lists has only the fields *lastindex* and *value*. The first $d - 1$ indexes can be recovered by the pointer we are currently using to run throughout the structure.

For $d \geq 3$ our method *differs* from *CRS*. To understand the difference, let us focus on the case $d = 3$. The proposed method, for any couple of indexes (i, j), stores a list corresponding to the indexes k. On the contrary, the *CRS* method for any index i stores a list corresponding to the couple of indexes (j, k).

Remark 3.1 We can also consider a slightly different storage method, which is a variation of the SSD and that allows to save more memory allocation. The basic idea is that, in the 2D structure of Fig. 3, it is not necessary to store pointers which point to *NULL*. In this case the vector containing the pointers is substituted by a linked list similar to those containing the values of the nodes. This modification yields to a more complex numerical code. We have verified experimentally that the CPU time is smaller for $d = 2$, but not always for $d > 2$.

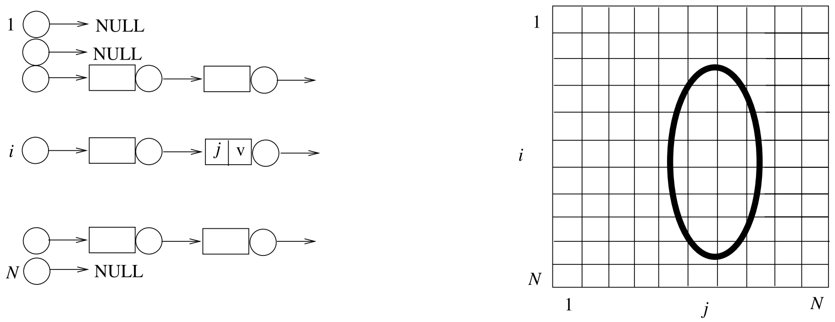


Fig. 3 A sparse semi-dynamic linked structure storing the nodes

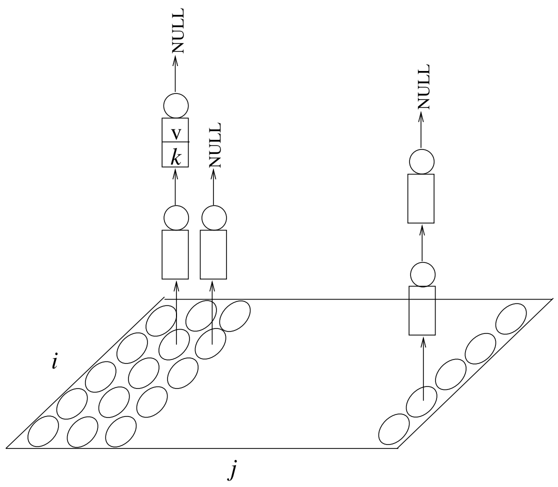


Fig. 4 *SSD* structure in three dimensions

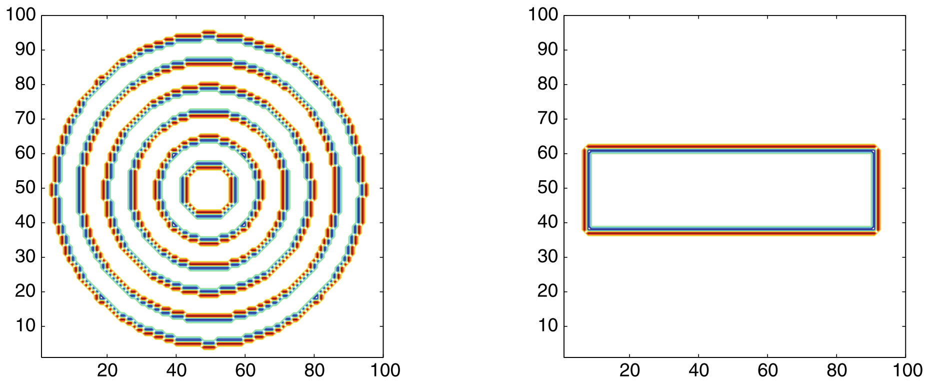


Fig. 5 Fronts for example 1 (*left*) and example 2 (*right*)

3.3 Comparisons

We now compare the previous storing data structures. We focus on searching neighbors, and on updating the data structures (inserting/removing elements).

Let Γ_t be the front at a given time t. We define $F_+(t)$ (resp. $F_-(t)$) as the set of nodes which are external (resp. internal) to Γ_t with at least one neighbor internal (resp. external) to Γ_t. Here by means of "neighbors" we intend the adjacent nodes in the coordinated directions (i.e. we do not consider the adjacent nodes along the diagonal directions). In all cases but *FS* and *FLS* the data structure contains only the nodes in $F_+ \cup F_-$. We set the value of the node in $F_\pm$ equal to ± 1. All other values are set to ND (Not Declared).

We will consider two different configurations for the initial front Γ_0 (see Fig. 5):

- *Example 1*: Γ_0 is composed by 6 circles (spheres for $d = 3$, hyperspheres for $d = 4$) one inside the other. This case is interesting because of the large number of nodes in $F_+ \cup F_-$.
- *Example 2*: Γ_0 is a rectangle (cylinder for $d = 3$, hypercylinder for $d = 4$) with the longest dimension aligned to the d-th one.

Table 1 CPU times for neighbor searching, Example 1

d	N^d	N_{nb}	*FS*	*FLS*	Order	*LS*	Order	*CRS*	Order	*SSD*	Order
2	400^2	7104	1.2e−3	1.1e−4		2.6e−3		2.8e−3		2.8e−3	
2	800^2	14 232	5.0e−3	3.4e−4	3.1	5.3e−3	2.0	5.7e−3	2.0	5.7e−3	2.0
2	1600^2	28 488	2.0e−2	1.3e−3	3.8	1.1e−2	2.1	1.2e−2	2.1	1.2e−2	2.1
3	100^3	104 576	1.3e−2	3.2e−3		1.6e0		2.4e0		2.0e−2	
3	200^3	423 704	1.1e−1	2.0e−2	6.2	1.3e1	8.1	1.9e1	7.9	8.3e−2	4.0
3	400^3	1 702 192	9.3e−1	9.0e−2	4.5	1.0e2	7.7	1.5e2	7.9	3.4e−1	4.2
3	800^3	6 826 816	Out of mem.	Out of mem.	–	>1e3	–	>1e3	–	1.8e0	4.1
4	25^4	78 704	7.2e−3	2.7e−3		4.2e0		8.1e0		2.1e−2	
4	50^4	674 064	1.2e−2	2.6e−2	9.6	1.5e2	35.7	3.1e2	38.2	1.9e−1	9.0
4	100^4	5 549 888	2.1e0	3.1e−1	11.9	>1e3	–	>1e3	–	1.4e0	7.1
4	200^4	45 098 160	Out of mem.	Out of mem.	–	>1e3	–	>1e3	–	1.2e1	8.4

Table 2 CPU times for neighbor searching, Example 2

d	N^d	N_{nb}	*FLS*	Order	*SSD*	Order
3	100^3	13 560	4.3e−4		6.6e−3	
3	200^3	55 808	2.2e−3	5.1	4.5e−2	6.8
3	400^3	222 480	6.6e−3	3.0	3.3e−1	7.3
3	800^3	894 056	Out of mem.	–	3.0e0	9.0
4	25^4	4 392	1.4e−4		2.8e−3	
4	50^4	35 776	1.0e−3	7.1	3.4e−2	12.1
4	100^4	297 200	1.0e−2	10.0	4.8e−1	14.1
4	200^4	2 450 144	Out of mem.	–	5.9e0	12.3

All the tests are performed on uniform Cartesian grids on the box $[-2, 2]^d$. We consider the same number N of points in each axis (N^d is the total number of grid nodes). We denote by N_{nb} the number of nodes in $F_+ \cup F_-$ (nodes in the narrow band).

3.3.1 Comparisons for Searching Neighbors

In this test we measure the CPU time (in seconds) needed to search the value of the $2d$ neighbors (with $d = 2, 3, 4$) of all the nodes in $F_+ \cup F_-$. The position of the front is fixed (so the data structure is not updated). Results are reported in Tables 1 and 2.

We recall that the time for the FS method include all neighbors of all nodes, not just $F_+ \cup F_-$.

We can see that *FLS* is the fastest method because computation is restricted to a narrow band around the front and the research of neighbors is done by using the full matrix. However *FLS* cannot be used when the number of grid nodes is too large.

In dimension 2, the *SSD* method coincides with *CRS*, and their performances are very similar to that of *LS*. However, as the dimension increases, the difference between these methods becomes larger and larger and the *SSD* seems to be the only sparse method able to find neighbors in reasonable time.

Table 3 CPU times for updating the data structure, 3-dimensional examples

Ex.	N^3	N_{nb}	*FLS*	Order	*LS*	Order	*CRS*	Order	*SSD*	Order
1	50^3	25 672	5.3e−3		1.8e−1		2.3e−1		1.1e−2	
1	100^3	104 576	2.3e−2	4.3	1.3e0	7.2	2.0e0	8.7	4.9e−2	4.4
1	200^3	423 704	1.0e−1	4.3	1.0e1	7.7	1.7e1	8.5	2.2e−1	4.5
1	400^3	1 702 192	4.5e−1	4.5	8.1e1	8.1	1.3e2	7.6	8.9e−1	4.0
1	800^3	6 826 816	Out of mem.	–	6.6e2	8.2	>1e3	–	3.8e 0	4.3
2	50^3	3 272	5.0e−4		1.3e−3		3.3e−3		1.3e−3	
2	100^3	13 560	2.0e−3	4.0	9.6e−2	7.3	3.7e−2	11.2	6.6e−3	6.2
2	200^3	55 808	8.9e−3	4.4	1.3e−1	13.5	4.9e−1	12.9	3.9e−2	7.1
2	400^3	222 480	3.7e−2	4.1	1.8e0	13.6	5.8e0	12.0	3.0e−1	7.4
2	800^3	894 056	Out of mem.	–	4.2e1	24.1	7.3e1	12.5	2.9e 0	9.8

In Example 1 (see Table 1), we remark that the cost for searching neighbors with the *SSD* method is of the expected order $O(N_{nb}) = O(N^{d-1})$. The scaling of LS or CRS is not as good as SSD, because it uses a usual sparse matrix structure that have slower neighbor searching of order $O(N^d)$. For Example 2, we have numerically observed that the scaling factor of the *SSD* method remains better then the one of *LS* or *CRS*. However, this is the worst situation for *SSD* in the sense that it does not behave as $O(N^{d-1})$.

On the other hand, if it is known a priori that the front has a dimension predominant over the others, by swapping two coordinates we could easily avoid the worse case for *SSD*.

3.3.2 Comparison for Updating the Data Structure

Now we compare the CPU time needed to update the data structure. This step consists in inserting and removing nodes, without searching for neighbors. More precisely, we insert all the neighbors of the nodes in F_+ having value ND and remove all the nodes in F_-. This simulates an update step for an expanding front.

FS is not considered here because it does not need an update procedure. In *FLS* we do not keep the list sorted, we just add the new nodes at the beginning of the list. The full matrix is used to check if a node to be inserted is already in the list. We perform the tests on the two examples as before, in the case $d = 3$. Results are summarized in Table 3.

The *FLS* is clearly the fastest method but it needs much more memory than the sparse methods. *SSD* greatly overcomes the *LS* and *CRS* methods in any test, being only two times slower than *FLS* in normal situations (Example 1). We shall see in Sect. 3.5 a concrete example where *SSD* compares to *FLS*.

3.4 Comments on CPU Time and Memory Allocation

In the *SSD* structure, although the search for a neighboring value can have a cost up to $O(N)$ (in the worst case), as well as for updating (adding or removing a node at a given location), we observe in general an order $O(1)$ for a “standard” front propagation problem. Therefore for a narrow band of size $N_{nb} = O(N^{d-1})$, we obtain the CPU time of order $O(N^{d-1})$ in most cases. This will be also exemplified in the following tests (see also next section).

Regarding the size of the front, *FS* and *FLS* require a memory allocation of order $O(N^d)$, while *LS*, *CRS* and *SSD* are all of order $O(N_{nb})$, which is in general $O(N^{d-1})$.

More precisely, *FS* and *FLS* require to store, during all the computation process, a N^d-matrix for the values (V_i^n) which are real variables (often double precision), while *SSD*

Table 4 Memory used in the case of Example 1 and with $d = 3$ (in Megabytes)

d	N^d	*FS*	Order	*FLS*	Order	*LS*	Order	*CRS*	Order	*SSD*	Order
3	100^3	9.0		9.0		0.9		0.9		1.8	
3	200^3	63.3	7.0	64.2	7.1	1.8	2.0	1.8	2.0	3.6	2.0
3	400^3	500.7	7.9	502.5	7.8	6.3	3.5	6.3	3.5	10.8	3.0
3	800^3	Out of mem.		Out of mem.		21.7	3.4	21.7	3.4	38.9	3.6

requires only to store a N^{d-1}-matrix of pointers plus N_{nb} pointers, N_{nb} integers and N_{nb} reals in the dynamic part of the structure.

In Table 4, we give the memory used for the various methods in the case of Example 1 and for dimension $d = 3$. We observe the expected scalings of the methods (which should be 4 for the sparse methods). We see that *SSD* is a little bit more costly than *LS* and *CRS* methods (because it has to store furthermore a matrix of N^{d-1} pointers) but also that it is far less costly than the full methods.

We also see the memory limitation of the full methods: for $d = 3$, we could not go further $N = 500$ approximately (resp. $N = 110$ for $d = 4$).

3.5 The Sparse Semi-Dynamic Storage for HJB Equations

When using a data structure which does not contain all the nodes of the grid we have to be sure that either the data structure contains all the nodes used for computation or we can recover the values of the nodes even if they are not stored in the data structure.

We know that the function ϑ takes value in the set $\{-1, 1\}$, so the numerical solution V takes value in the set $[-1, 1]$ because it represents an average of the function ϑ. So, at every time step n, the front is localized in the set of nodes $\{I : V_I^n \in (-1, 1)\}$ and around this region there is a 1-node thick band where $V^n = -1$ (internal to the front) and a 1-node thick band where $V^n = 1$ (external to the front), see Fig. 2. *Only these nodes are stored in the data structure*, but they are not enough to perform computation with the UB-HJB 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 -1 (resp., 1) and it is not stored in the data structure, then its value is -1 (resp., 1).

Let D^n be the set of nodes stored in the data structure at time step t_n. Let I be a generic node of the grid and $\mathrm{Neigh}(I)$ be the set of the $3^d - 1$ neighbors of the node I (from now on we have to include also the diagonal directions as required in the UB-HJB scheme). At each time step, D^n must first be updated in order to follow the evolution of the front. Then we set

$$D^{n+\frac{1}{2}} := D^n \cup \mathrm{Neigh}(D^n).$$

In order to maintain the data structure as slim as possible, after each computation (at every time step) we remove from the data structure the set of nodes whose value is either -1 or 1 and it is equal to the values of the neighbors:

$$\mathcal{R}^n := \big\{ I : V_I^n = -1 \text{ or } 1, \text{ and } V_I^n = V_J^n, \ \forall J \in \mathrm{Neigh}(I) \big\}.$$

Thus, $D^{n+1} := D^{n+\frac{1}{2}} \setminus \mathcal{R}^n$.

Now we apply the new data structure to a test problem, solving a simple HJB equation in the form of (4). We set $d = 3$ and $f(x, y, z, a) = (a, 1, 1)^{\mathsf{T}}$. For testing, we consider $N_a = 11$

Table 5 HJB equation, CPU time for one time step (seconds)

N^3	$\|D^0\|$	*FLS*	Order	*SSD*	Order
50^3	8016	0.07		0.08	
100^3	31 416	0.23	3.2	0.37	4.6
200^3	125 968	0.96	4.2	1.45	3.9
400^3	502 368	5.75	6.0	6.52	4.5
800^3	2 011 032	Out of mem.		27.2	4.1
1600^3	8 043 808	Out of mem.		113.4	4.1

discrete control variables $a_i \in [-1, 1]$ defined by $a_i = -1 + 2i/(N_a - 1)$, $i = 0, \dots, N_a - 1$. The initial front (target) is the unit ball. Computation is performed in the box $[-2, 2]^3$. The time step Δt is computed so that the CFL number is 0.9.

The results are summarized in Table 5. We report the CPU time for one time step, without considering the initialization of the data structures nor saving results on disk. $|D^0|$ is the number of nodes used to locate the initial front. (We have excluded here *FS* because it is clearly not efficient, as well as *CRS* and *LS* because previous tests showed that they are not better than the *SSD* method.)

In this test, the *SSD* method shows an excellent behavior, with correct scaling, being able to save memory without loosing too much rapidity. The *FLS* is the fastest method, but it is out of memory as soon as the number of nodes is greater than 500^3.

Also we have numerically observed that as the number of controls N_a increases, *SSD* and *FLS* have a tendency to have similar CPU times. This is because the cost of managing the sparse structure becomes neglectable with respect to the computational cost for the scheme and which is similar for both approaches.

4 Numerical Examples

In this section we test the UB-HJB scheme with the *SSD* structure on some numerical examples.

In all tests, we use a variable time step Δt which is computed at each iteration to fit the CFL condition (10) with respect to the nodes currently involved in the narrow band. We have chosen CFL $= 0.9$ in all cases.

We shall also compare the results with the level set method based on a very simple RK2-ENO2 solver (without reinitialization), following Osher and Shu [26], and which is a priori second order in space and time.[2] Even if this is certainly not the best level set solver we think it is sufficient to illustrate the advantages (or drawbacks) of our UB-HJB discontinuous approach with respect to a continuous level set approach, in particular when the front is not smooth everywhere.

[2]In the one-dimensional case, we consider the approximation

$$v(x_i)\partial_x u(x_i) \simeq \max(v(x_i), 0)Du_i^- + \min(v(x_i), 0)Du_i^+$$

where Du_i^-, Du_i^+ are second order ENO approximations of the first derivative $\partial_x u$. This is then coupled with a Heun scheme in time (a second order Runge-Kutta scheme).

Table 6 (Example 1) $f = (-2, -1)$: error and total CPU time, $T = 1$. The L^1 error is 0 for the UB-HJB scheme because of an exact-advection property in this particular example

N^2	UB-HJB					Level set		
	CPU	Order	L^1-error	Haus.	Haus./Δx	L^1 error	Haus.	Haus./Δx
50^2	0.01		0.000	0.044	0.55	0.185	0.158	1.97
100^2	0.03	3.0	0.000	0.020	0.50	0.177	0.101	2.52
200^2	0.13	4.3	0.000	0.014	0.70	0.020	0.064	3.20
400^2	0.61	4.7	0.000	0.007	0.70	0.008	0.041	4.10

Table 7 (Example 1) $f(x) = (-x_2, x_1)$: error and total CPU time, $T = 2\pi$

N^2	UB-HJB					Level set		
	CPU	Order	L^1 error	Haus.	Haus./Δx	L^1 error	Haus.	Haus./Δx
50^2	0.04		0.192	0.116	1.45	0.236	0.249	3.11
100^2	0.16	4.0	0.051	0.056	1.40	0.134	0.151	3.77
200^2	0.70	4.4	0.049	0.042	2.10	0.038	0.094	4.70
400^2	2.65	3.8	0.025	0.029	2.90	0.015	0.060	6.00

Example 1 (Advection of a square) In this first example we solve the following two-dimensional advection equation

$$\vartheta_t(t,x) + f(x) \cdot \nabla\vartheta(t,x) = 0, \quad t \in [0,T],\ x \in \mathbb{R}^2, \tag{11}$$

with a square box initial datum

$$\varphi^{\mathrm{UB}}(x) := \begin{cases} -1 & \text{if } x \in [0.5, 1.5] \times [-0.5, 0.5], \\ 1 & \text{otherwise.} \end{cases}$$

We consider two possible dynamics: either $f(x_1, x_2) = (-2, -1)^{\mathrm{T}}$ with $T = 1$ (constant advection), or $f(x_1, x_2) = (-x_2, x_1)^T$ with $T = 2\pi$ (rotation). Equation (11) is discretized on the domain $[-2, 2]^2$ (with boundary conditions $\vartheta(t, x) = 1$ when needed).

We show in Tables 6 and 7 the UB-HJB scheme results, with different discretization numbers (the same number of nodes N is used in each direction). CPU times are given in seconds. The error is computed in L^1 norm between the numerical and exact solution (see below), and the Hausdorff distance[3] is also estimated, between the exact and approximate front.

In the case of the level set method, we choose a regular (Lipschitz continuous) φ^{LS} which admits the square border as 0-level set,

$$\varphi^{\mathrm{LS}}(x_1, x_2) := \min(r - 0.5, 0.5), \quad \text{with } r = \max(|x_1 - 1|, |x_2|). \tag{12}$$

[3]The Hausdorff distance between to sets A, B is defined as $d_H(A, B) := \max(\delta(A, B), \delta(B, A))$ where $\delta(A, B) := \max_{a\in A} d(a, B)$.

In order to use a similar way for computing the L^1 error for both UB-HJB and for the level set approach, we define:

$$\varepsilon^{\mathrm{UB}} := \Delta x_1 \Delta x_2 \sum_{i,j} 1_{\{\mathrm{sign}(V_{ij}^{N_T,\mathrm{UB}}) \neq \mathrm{sign}(\varphi(\xi_{ij}))\}},$$

with $\mathrm{sign}(x) := -1$ if $x \le 0$, $\mathrm{sign}(x) := 1$ otherwise, $V_{i,j}^{n,\mathrm{UB}}$ is the numerical value of the UB-HJB scheme, N_T is the time step corresponding to the final time T and ξ_{ij} denotes the center of the cell (i, j); we define also

$$\varepsilon^{\mathrm{LS}} := \Delta x_1 \Delta x_2 \sum_{i,j} 1_{\{\mathrm{sign}(V_{i,j}^{N_T,\mathrm{LS}}) \neq \mathrm{sign}(\varphi^{\mathrm{LS}}(\xi_{ij}))\}}$$

where $V_{i,j}^{n,\mathrm{LS}}$ denotes the numerical values computed by the level set method.

We remark that the L^1 error estimate for UB-HJB may vanish, in the constant advection case, which means here that the computed points with value -1 are all in the expected correct cells. This comes from a known exact-advection property of the Ultra Bee scheme as mentioned in Sect. 2.2.

We also note that the UB scheme, which is only a first order scheme at best, gives here results comparable to the second order RK2-ENO2 level set method. This is because the level set method is second order only in the regions where the front is regular, and is not expected to be better than first order otherwise (the theoretical bound is $O(\Delta x^{1/2})$ for monotone schemes, following [13]).

Example 2 (Two-dimensional Rendez-Vous problem) In this example we consider an HJB equation of type (1) with $f(x,a) := (1,a)^{\mathsf{T}}$ and $a \in \{-1, 1\}$, that is

$$\vartheta_t(t,x) + \vartheta_{x_1}(t,x) + |\vartheta_{x_2}(t,x)| = 0, \quad t \in [0,T],\ x = (x_1,x_2) \in \mathbb{R}^2, \tag{13a}$$

$$\vartheta(0,x) = \varphi_r(x), \tag{13b}$$

where the initial data is given by

$$\varphi_r(x) := \begin{cases} -1 & \text{if } \|x\|_\infty \le r, \\ 1 & \text{otherwise} \end{cases} \tag{13c}$$

(here we use the notation $\|(x_1,x_2)\|_\infty := \max(|x_1|, |x_2|)$). We consider two types of target:

- $r = 0.1$: large target case.
- $r = 0$: thin target case. For numerical purpose, we set $r = \Delta x$ (so that there is only one single node that contains a negative value).

Numerically, the equation is discretized on $[-1,2]^2$. Results are given in Tables 8 and 9 and Fig. 6. We notice that, in the large target case, the level set method does not work for $N = 51$ (i.e. N small), because the set of points such that $V_{ij}^{n,\mathrm{LS}} < 0$ vanishes after a few time steps. The problem is coming from the initial data discretization, which has only one negative value (this is the same when r is only of the order of a few Δx). This problem would always occurs for thin target problems.

Table 8 (Example 2) Rendez-Vous problem, $T = 1$. Large target ($r = 0.1$) The level set scheme may loose the front (in this case no error is given)

N^2	UB-HJB		Level set	
	L^1 error	Haus.	L^1 error	Haus.
51^2	0.178	0.052	–	–
101^2	0.105	0.022	0.101	0.094
201^2	0.044	0.011	0.008	0.047
401^2	0.022	0.006	0.006	0.027

Table 9 (Example 2) Rendez-Vous problem, $T = 1$. Thin target ($r = \Delta x$). In this case, the 0-level set is lost after a few time steps

N^2	UB-HJB		Level set	
	L^1 error	Haus.	L^1 error	Haus.
51^2	0.166	0.043	–	–
101^2	0.080	0.031	–	–
201^2	0.040	0.016	–	–
401^2	0.020	0.008	–	–

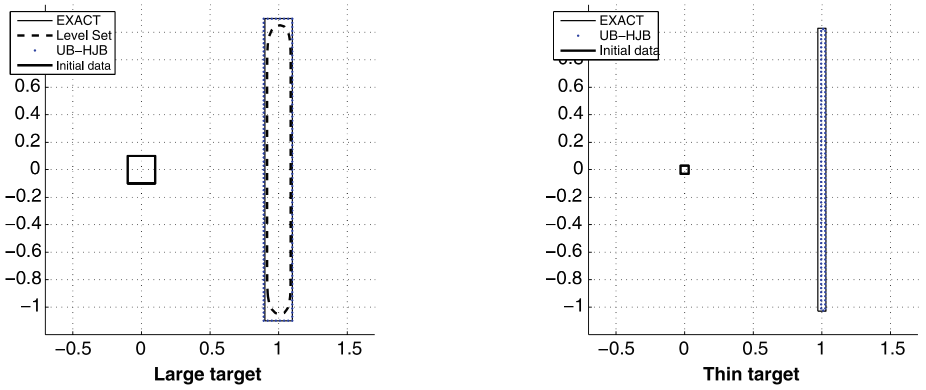


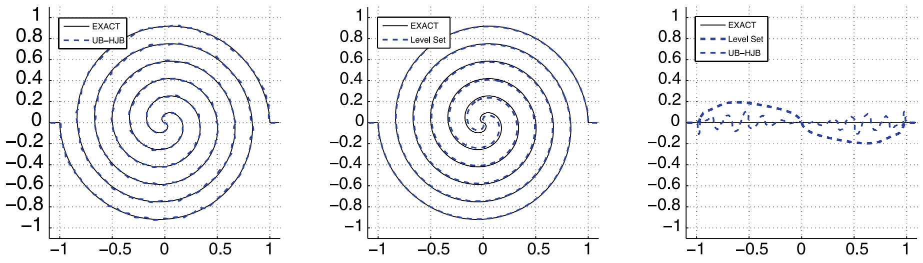
Fig. 6 (Example 2) Rendez vous problem with $N = 101$

Example 3 (Two-dimensional deformation of a half plane) We consider a front propagation problem, where the initial front Γ_0 is given by: $\Gamma_0 := \{x = (x_1, x_2) \in \mathbb{R}^2 \mid x_2 = 0\}$. The velocity of the front evolution is given by

$$f(t, x_1, x_2) = -\operatorname{sign}\left(\frac{T}{2} - t\right) \max(1 - \|x\|_2, 0) \begin{pmatrix} -2\pi x_2 \\ 2\pi x_1 \end{pmatrix}.$$

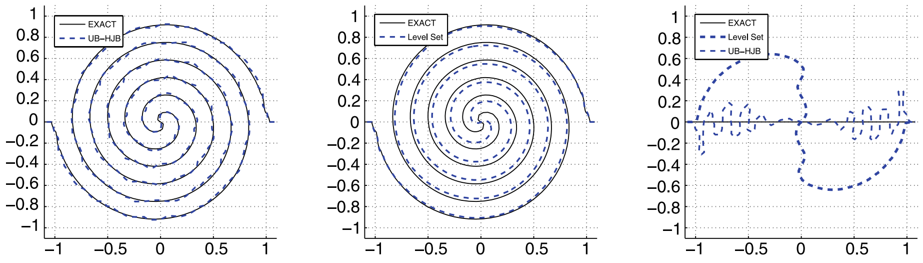
The front propagation leads to the following advection equation

$$\begin{cases} \vartheta_t(t, x) + f(t, x) \cdot \nabla\vartheta(t, x) = 0, & x \in \mathbb{R}^2,\ t \in [0, T], \\ \vartheta(0, x) = \varphi(x), \end{cases} \tag{14}$$



UB-HJB, $t = 3$ Level set, $t = 3$ $t = 6$

Fig. 7 (Example 3) UB-HJB and level set methods at times $t = 3$ (after three turns) and $t = 6$ (return to initial data), CFL= 0.9, with $N = 100$



UB-HJB, $t = 3$ Level set, $t = 3$ $t = 6$

Fig. 8 (Example 3) Same computation as in Fig. 7, with $N = 50$

where the function φ is continuous in the level set approach and discontinuous in our $\{-1, 1\}$ approach

$$\begin{aligned} \varphi^{\mathrm{LS}}(x_1, x_2) &:= \min(\max(x_2, -1), 1) \quad \text{in the level set approach,} \\ \varphi^{\mathrm{UB}}(x_1, x_2) &:= \begin{cases} -1 & x_2 \le 0, \\ 1 & \text{otherwise} \end{cases} \quad \text{in the } \{-1, 1\} \text{ approach,} \end{aligned} \tag{15}$$

It is not difficult to prove that the exact solution is given by: $\vartheta(t, x) = \varphi(R_{-2\pi a(x) t_1} x)$ where $R_t = \begin{pmatrix} \cos(t) & -\sin(t) \\ \sin(t) & \cos(t) \end{pmatrix}$, $a(x) = \max(1 - \|x\|_2, 0)$, and $t_1 = \min(t, T - t)$.

Results are given in Figs. 7 and 8 and Table 10, for $T = 6$, at two different times: $t = 3$ (three turns), and $t = 6$ (return to initial condition after three turns). Exact solution can be obtained by using the method of characteristics. In Figs. 7 and 8 we show the results for the UB-HJB and level set method, using $N = 100$ and $N = 50$ nodes per direction. Anew, the UB-HJB scheme gives good results even on a coarse grid.

Example 4 (A three-dimensional rotation problem) Now we consider a three-dimensional advection example:

$$\begin{cases} \vartheta_t(t, x) + f(x) \cdot \nabla \vartheta(t, x) = 0, & t \in [0, T], \ x \in [-2, 2]^3, \\ \vartheta(0, x) = \varphi(x) \end{cases} \tag{16}$$

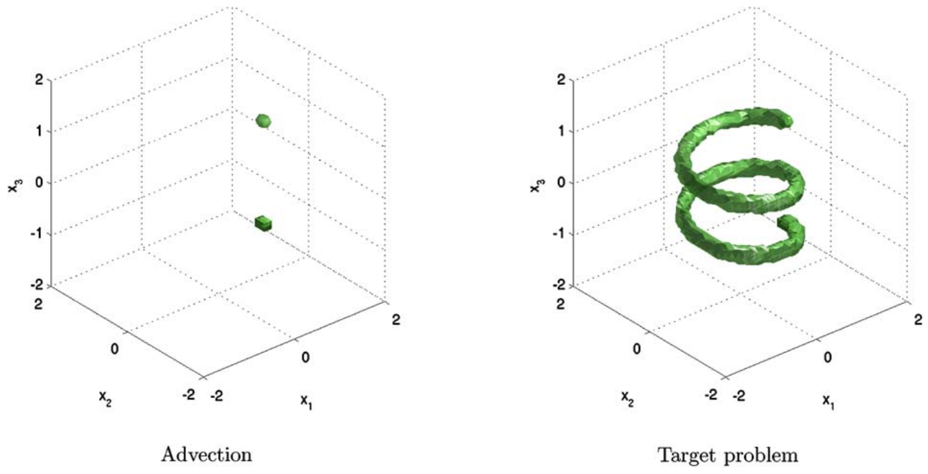


Fig. 9 (Example 4) advection problem (*left*), target problem (*right*). $T=2$, $N=50$

Table 10 (Example 3) deformation of a half plane at $t=3$ (three turns) and $t=6$ (return to initial data)

	$t=3$				$t=6$			
	UB-HJB		Level set		UB-HJB		Level set	
N^2	L^1 error	Haus.	L^1 error	Haus.	L^1 error	Haus.	L^1 error	Haus.
50^2	0.170	0.035	0.584	0.086	0.193	0.308	0.995	0.639
100^2	0.092	0.019	0.136	0.028	0.073	0.107	0.282	0.195
200^2	0.057	0.013	0.047	0.008	0.041	0.064	0.079	0.053

and the corresponding target problem (4), with $f(x_1,x_2,x_3)=(-2\pi x_2, 2\pi x_1, -1)^{\mathsf{T}}$ where the initial data is a sphere centered at $(-1,0,1)$ with radius $r=0.1$:

$$\varphi(x) := \begin{cases} -1 & \text{if } \|x-(1,0,1)\|_2 \le r, \\ 1 & \text{otherwise,} \end{cases}$$

where $\|x\|_2 := \sqrt{x_1^2+x_2^2+x_3^2}$.

For both problems (advection and target), we use only the UB-HJB approximation scheme, see Fig. 9. The computational domain is $[-2,2]^3$. CPU times and errors are summarized in Table 11 (with $T=2$ for the advection problem, and $T=0.5$ for the target problem). The scaling of the CPU time approximately follows the theoretical scaling of 8 as the mesh size doubles in each direction (a factor 4 comes from the number of cells localizing the surface, and a factor of 2 comes from the time step Δt, that is divided by 2 because of the CFL condition). The Hausdorff distance decreases well in $O(\Delta x)$ too.

Here we would have been limited to $N=500$ if we had used the *FLS* approach.

Table 11 (Example 4) CPU times and errors for the 3d advection problem (with $T = 2$) and the corresponding target problem (with $T = 0.5$), UB-HJB scheme

Advection

N^3	CPU		L^1 error	Hausdorff
50^3	0.22		4.1e−3	2.6e−1
100^3	1.00	4.6	2.2e−3	8.0e−2
200^3	7.19	7.2	6.2e−4	4.4e−2
400^3	64.5	8.9	4.8e−4	2.4e−2
800^3	546.0	8.4	2.6e−4	1.2e−2

Target problem

N^3	CPU		L^1 error	Hausdorff
50^3	0.24		2.3e−1	1.8e−1
100^3	1.15	4.8	1.3e−2	8.0e−2
200^3	8.25	7.2	6.7e−2	4.0e−2
400^3	76.0	9.2	3.5e−2	2.3e−2
800^3	763.0	10.0	1.8e−2	1.3e−2

Table 12 (Example 5) four-dimensional non constant advection of a square box, $T = 1$, UB-HJB scheme

N^4	CPU	Order	L^1 error	Haus.	Haus./Δx_1
25^4	0.89		0.0016	0.140	1.7
50^4	9.29	10.4	0.0021	0.080	2.0
100^4	129.1	13.9	0.0013	0.050	2.5
200^4	2316.2	18.1	0.0007	0.025	2.5

Example 5 (4-dimensional non-constant advection) We consider a 4-dimensional advection equation, with dynamics defined by coupling two rotations as follows:

$$f(x_1, x_2, x_3, x_4) = (-2\pi x_2, 2\pi x_1, -2\pi x_4, 2\pi x_3)^{\mathsf{T}}.$$

The initial data is given by

$$\varphi(x) := \begin{cases} -1 & \text{if } x \in (0.5, 0, 0.5, 0)^{\mathsf{T}} + [-0.1, 0.1]^4, \\ 1 & \text{otherwise} \end{cases} \tag{17}$$

(which corresponds to a small square box centered at $(0.5, 0, 0.5, 0)^{\mathsf{T}}$). For $T = 1$ the exact solution corresponds to the initial data. Computations are done in the unit box $[-1, 1]^4$.

Here we would have been limited to $N = 110$ (approximately) if we had used the *FLS* approach.

We summarize the numerical results in Table 12. In this example, for $N = 200$ there is an average of 230 000 cells in the narrow band (instead of $N^4 = 1\,600\,000\,000$), 384 time iterations, with an average CPU time of 6.0 s per time iteration.

Example 6 We consider the HJB equation

$$\begin{cases} \vartheta_t(t, x) + \max(0, \max_{\alpha=1,\ldots,4}(f(x, \alpha) \cdot \nabla\vartheta(t, x))) = 0, & t \in [0, T],\ x \in \mathbb{R}^4, \\ \vartheta(0, x) = \varphi(x), \end{cases} \tag{18}$$

where the dynamics $f(x, \alpha)$ can have 4 different values: $(1, 0, 0, 0)^{\mathsf{T}}$, $(1, 1, 0, 0)^{\mathsf{T}}$, $(1, 0, 1, 0)^{\mathsf{T}}$, or $(1, 0, 0, 1)^{\mathsf{T}}$. The initial data is given by

$$\varphi(x) := \begin{cases} -1 & \text{if } x = (0, 0, 0, 0), \\ 1 & \text{otherwise.} \end{cases}$$

Table 13 (Example 6) Results for $T = 1$, UB-HJB scheme

N^4	CPU	Order	Haus.	Haus.$/\Delta x_1$
25^4	0.28		0.120	0.75
51^4	1.66	5.9	0.039	0.49
101^4	11.97	7.2	0.029	0.73
201^4	175.90	14.7	0.015	0.75

The exact solution is given by $\vartheta(t,x) = -1$ on Ω_t and $\vartheta(t,x) = 1$ otherwise, where

$$\Omega_t := \{x \in \mathbb{R}^4 \mid x_1, x_2, x_3, x_4 \geq 0,\ x_2 + x_3 + x_4 \leq x_1 \leq t\}.$$

The computation is done in $[-2, 2]^4$. Results are given in Table 13.

In this example, the estimated CPU time for the computation for one control (one evaluation of the UB scheme for a given control a and for one time step), is about 10% of the CPU time needed to manage the sparse structure for the same time step.

Acknowledgement The authors thank the anonymous referee for his detailed and insightful comments, suggestions that greatly improved the presentation of the paper.

References

1. Abgrall, R.: Numerical discretization of first-order Hamilton-Jacobi equation on triangular meshes. Commun. Pure Appl. Math. **49**, 1339–1373 (1996)
2. Abgrall, R., Augoula, S.: High order numerical discretization for Hamilton-Jacobi equations on triangular meshes. J. Sci. Comput. **15**, 197–229 (2000)
3. Adalsteinsson, D., Sethian, J.A.: A fast level set method for propagating interfaces. J. Comput. Phys. **118**, 269–277 (1995)
4. Bardi, M., Capuzzo Dolcetta, I.: Optimal Control and Viscosity Solutions of Hamilton-Jacobi-Bellman Equations. Birkhäuser, Boston (1997)
5. Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., Van der Vorst, H.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd edn. SIAM, Philadelphia (1994)
6. Barron, E.N., Jensen, R.: Semicontinuous viscosity solutions for Hamilton-Jacobi equations with convex Hamiltonians. Commun. Partial Differ. Equ. **15**, 1713–1742 (1990)
7. Bokanowski, O., Cristiani, E., Laurent-Varin, J., Zidani, H.: Hamilton-Jacobi-Bellman approach for the climbing problem for heavy launchers. Preprint (2009)
8. Bokanowski, O., Forcadel, N., Zidani, H.: Convergence of a non-monotone scheme for Hamilton-Jacobi-Bellman equations with discontinuous initial data. To appear in Math. Comput.
9. Bokanowski, O., Martin, S., Munos, R., Zidani, H.: An anti-diffusive scheme for viability problems. Appl. Numer. Math. **56**, 1135–1254 (2006)
10. Bokanowski, O., Megdich, N., Zidani, H.: An adaptative antidissipative method for optimal control problems. Arima **5**, 256–271 (2006)
11. Bokanowski, O., Megdich, N., Zidani, H.: Convergence of a non-monotone scheme for Hamilton-Jacobi-Bellman equations with discontinuous initial data. Numer. Math. (2009). doi:10.1007/s00211-009-0271-1
12. Bokanowski, O., Zidani, H.: Anti-diffusive schemes for linear advection and application to Hamilton-Jacobi-Bellman equations. J. Sci. Comput. **30**, 1–33 (2007)
13. Crandall, M.G., Lions, P.-L.: Two approximations of solutions of Hamilton-Jacobi equations. Math. Comput. **43**, 1–19 (1984)
14. Desprès, B., Lagoutière, F.: Un schéma non linéaire anti-dissipatif pour l'équation d'advection linéaire. A non-linear anti-diffusive scheme for the linear advection equation. C. R. Acad. Sci. Paris, Sér. I, Math. **328**, 939–944 (1999)

15. Després, B., Lagoutière, F.: Contact discontinuity capturing schemes for linear advection and compressible gas dynamics. J. Sci. Comput. **16**, 479–524 (2001)
16. Frankowska, H.: Lower semicontinuous solutions of Hamilton-Jacobi-Bellman equations. SIAM J. Control Optim. **31**, 257–272 (1993)
17. Hartmann, D., Meinke, M., Schroeder, W.: Differential equation based constrained reinitialization for level set methods. J. Comput. Phys. **227**, 6821–6845 (2008)
18. Jiang, G.-S., Peng, D.: Weighted ENO schemes for Hamilton-Jacobi equations. SIAM J. Sci. Comput. **21**, 2126–2143 (2000)
19. Lagoutière, F.: A non-dissipative entropic scheme for convex scalar equations via discontinuous cell-reconstruction. C. R. Math. Acad. Sci. Paris **338**, 549–554 (2004)
20. Lagoutière, F.: Modélisation mathématique et résolution numérique de problèmes de fluides compressibles à plusieurs constituants. Ph.D. thesis, University of Paris VI, Paris, France (2000)
21. Lin, C.-Y., Chung, Y.-C.: Efficient data compression methods for multidimensional sparse array operations based on the EKMR scheme. IEEE Trans. Comput. **52**, 1640–1646 (2003)
22. Megdich, N.: Méthodes anti-dissipatives pour les equations de Hamilton-Jacobi-Bellman. Ph.D. thesis, University of Paris VI, Paris, France (2008)
23. Mitchell, I., Bayen, A., Tomlin, C.: A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games. IEEE Trans. Automat. Contr. **50**, 947–957 (2005)
24. Osher, S.: A level set formulation for the solution of the Hamilton-Jacobi equations. SIAM J. Math. Anal. **24**, 1145–1152 (1993)
25. Osher, S., Sethian, J.A.: Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. J. Comput. Phys. **79**, 12–49 (1988)
26. Osher, S., Shu, C.-W.: High-order essentially nonoscillatory schemes for Hamilton-Jacobi equations. SIAM J. Numer. Anal. **28**, 907–922 (1991)
27. Peng, D.P., Merriman, B., Osher, S., Zhao, H.K., Kang, M.J.: A PDE-based fast local level set method. J. Comput. Phys. **155**, 410–438 (1999)
28. Robins, G.: Robs algorithm. Appl. Math. Comput. **189**, 314–325 (2007)
29. Sethian, J.A.: Level Set Methods and Fast Marching Methods. Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science. Cambridge University Press, Cambridge (1999)
30. Shu, C.-W.: High order ENO and WENO schemes for computational fluid dynamics. In: High-order Methods for Computational Physics. Lect. Notes Comput. Sci. Eng., vol. 9, pp. 439–582. Springer, Berlin (1999)
31. Sussman, M., Smereka, P., Osher, S.: A level set approach for computing solutions to incompressible 2-phase flow. J. Comput. Phys. **114**, 146–159 (1994)