Journal of Computational Physics 231 (2012) 2528–2536

Contents lists available at SciVerse ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

A local level-set method using a hash table data structure

Emmanuel Brun [a], Arthur Guittet [a,*], Frédéric Gibou [a,b]

[a] Department of Mechanical Engineering, University of California, Santa Barbara, CA 93106-5070, United States
[b] Department of Computer Science, University of California, Santa Barbara, CA 93106-5070, United States

ARTICLE INFO

Article history:
Received 28 May 2011
Received in revised form 7 November 2011
Accepted 1 December 2011
Available online 23 December 2011

Keywords:
Local level-set
Level-set
Irregular domains
Hash tables
Reinitialization

ABSTRACT

We present a local level-set method based on the hash table data structure, which allows the storage of only a band of grid points adjacent to the interface while providing an $O(1)$ access to the data. We discuss the details of the construction of the hash table data structure as well as the advection and reinitialization schemes used for our implementation of the level-set method. We propose two dimensional numerical examples and compare the results to those obtained with a quadtree data structure. Our study indicates that the method is straightforward to implement but suffers from limitations that make it less efficient than the quadtree data structure.

Published by Elsevier Inc.

1. Introduction

The level-set method was originally introduced by Osher and Sethian [10] and has proven to be efficient for tracking evolving interfaces in numerous cases [16,9]. It relies on the simple idea of embedding a problem in a higher dimensional space and considering the interface as the zero level-set of a function, called the level-set function, in this space. This modeling procedure allows sophisticated behaviors of the interface, such as cusps, sharp corners and topological changes. Applications of the level-set method can be found in various domains such as fluid mechanics, electrodynamics and solid mechanics. The well-known drawback of this method is the so-called mass loss due to numerical approximations. In practice, the level-set function is stored by sampling its value on a mesh. The values of interest are located close to the interface, where high accuracy is desirable in order to locate correctly the interface as well as its geometrical quantities. Therefore, computations needed in the level-set method are only needed locally to the interface and methods providing this capability are dubbed local level-set methods.

Various approaches exist: Adalsteinsson and Sethian [1] suggested to perform the calculations for the level-set function evolution solely in a tube located close to the interface. The tube is updated every given number of steps so that it stays located close to the interface. The mesh still covers the entire domain so there is no advantages in terms of memory, but only a fraction of the nodes are processed thus improving significantly the computational time. The same idea of building a tubular grid around the interface has been exploited by Nielsen and Museth [5]. They developed a powerful but intricate data structure to keep track of the points located in the tubular area only. Another approach is to use an octree data structure [13,14] as for example in the work of Strain [19], Popinet [11], Losasso et al. [4] and Min and Gibou [7]. This approach consists of meshing the domain using an octree data structure, which allows to refine the mesh more accurately where the interface lies. Octree grids are very efficient, especially in the case where the grids can be ungraded, but they still suffer from slow lookup

* Corresponding author.
E-mail address: arthur.guittet@engineering.ucsb.edu (A. Guittet).

0021-9991/$ - see front matter Published by Elsevier Inc.
doi:10.1016/j.jcp.2011.12.001

performance: data access scales as O(log (n)) in the case of a grid with n nodes. Octree data structures also consume some extra memory to store relevant information needed to represent the structure.

We present an alternative method for building adaptive meshes to track interfaces based on the hash table structure, which we will refer to as *local grid*. This data structure, widespread in computer science, allows a very fast access to elements, up to an O(1) access for some implementations, instead of O(log (n)) for octrees, and provides an efficient strategy for storing an implicit surface in term of memory usage. The literature abounds with examples of usage of hash table data structures [21,18]. In this paper, we present an implementation of the level-set method using a hash table data structure and discuss the benefits and the drawbacks of the method.

2. The Hash table structure

A hash table, or hash map, is a type of data structure often used in computer science. The goal of such structures is to provide efficient access to data, and they often outperforms other classical structures like search trees or lookup tables. It relies on three sets identified as the *keys*, the *buckets* in which the values associated to the keys are stored, and the *hash function*.

The keys can be any type of data, and in our case it will be a two dimensional set of indices (i,j) referencing the grid points in a band around the interface. A bucket is associated to each key, which in practice is an index in an array. The hash function is the crux of the method. Its role is to associate a value to each key in the best possible way, as illustrated in Fig. 1, thus providing an access to the values corresponding to a key in O(1).

There is no general hash function that would give an optimal solution to every problem, and finding an efficient hash function can be a challenging task. This is the bottleneck of the structure, and there is no general method for designing this function. Ideally, the hash function should match each key to a single bucket, in which case it is called a perfect hash function. If this ideal function exists, every element can be accessed in a single lookup. In practice, such a function may be impossible to design, and different keys may be associated to the same bucket, creating so-called hash collisions. There are various strategies for dealing with hash collisions, which can be grouped into two classes: closed hashing and open hashing.

The first strategy to deal with collisions is closed hashing (also called open addressing), which consists in finding another available bucket in the hash table. Consider the case where the key $(i,j-2)$ is associated with the bucket k, and we want to associate a bucket to the key $(i+1,j-1)$ but the hash function produces the already used bucket k. In this case, alternate buckets need to be probed, for example with a linear probing sequence, until a free bucket is found. This procedure is illustrated in Fig. 2. Various closed hashing methods can be found in the literature, a more detailed description can be found in [3].

With the second strategy, called open hashing or closed addressing, each index in the array is pointing to a data structure in which the values are stored. This external structure can be any organized structure, such as a tree or an array. We will use linked lists, in which case one talks of separate chaining or direct chaining. An illustration of the separate chaining strategy can be found in Fig. 3. If we want to store a new value associated to the key $(i-1,j+2)$ and the hashing function gives the already used bucket l for this key, we just need to add a new element to the linked list stored in bucket l. Note that accessing elements is, in general, no longer done with a single operation. Once the bucket associated to a key is given by the hash function, the linked list it contains needs to be browsed until the right member is found. The number of hash collisions, and hence of the lookup time, depends on the efficiency of the hash function. In the worst case scenario (i.e. using an ill-behaved hash

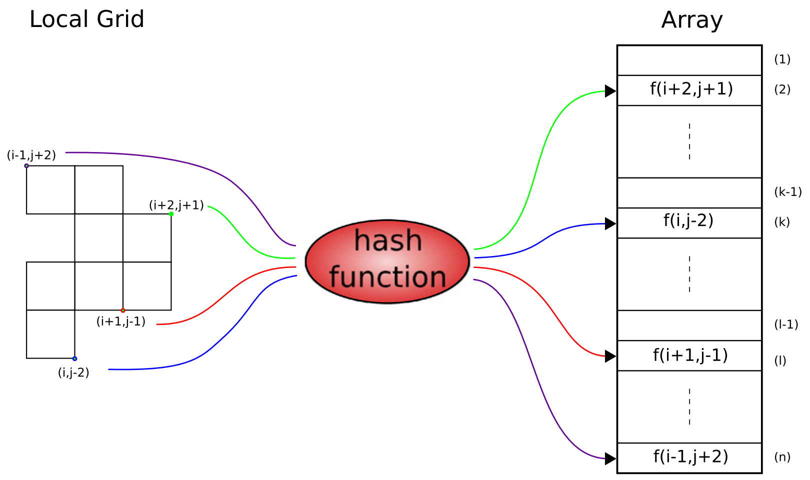

Fig. 1. Illustration of the hash table data structure. On the left is the set of keys to be associated with the set of values (on the right) using the hash function.

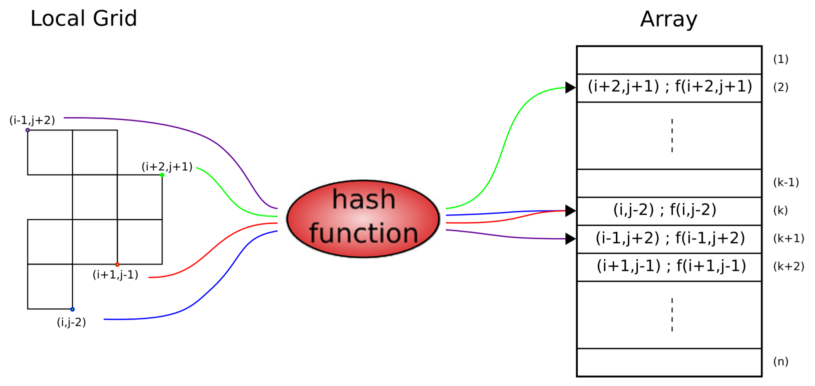

Fig. 2. Example of a hash collision treated with a closed hashing method using a linear probing sequence.

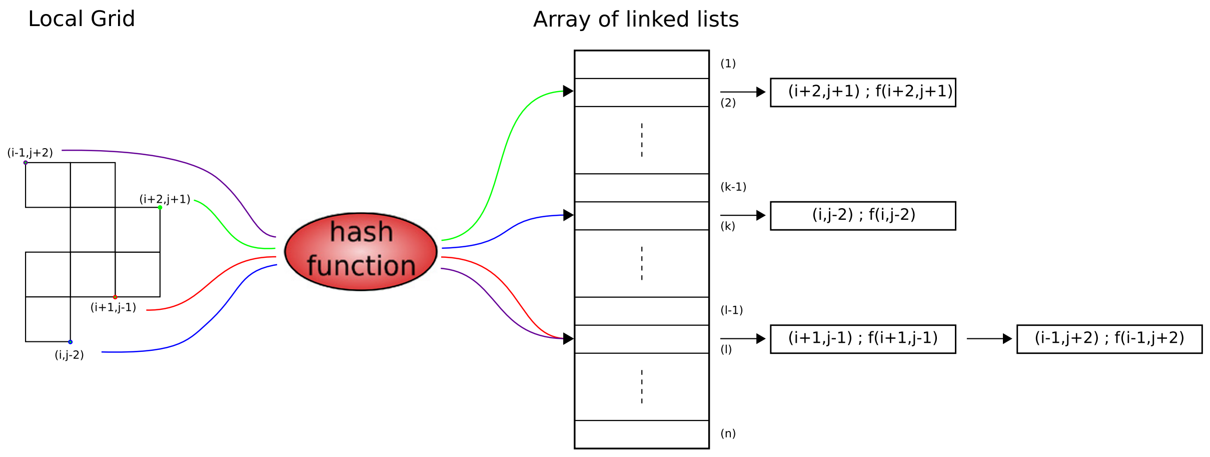

Fig. 3. Configuration of the hash table structure with an open hashing strategy. Each bucket is pointing to a linked list in which the desired information is stored.

function), a lookup might be done in O(n), but with a reasonably good function the average lookup time remains of the order of O(1).

3. Implementation of the local level-set method

3.1. Presentation of the level-set method

Our goal is to store the local grid on which the level-set function is defined in a hash table data structure. The main idea behind the level-set method is to describe an interface $\Gamma \in \mathbb{R}^n$ as the zero contour of a higher dimensional function $\phi \in \mathbb{R}^n$. Thus, in two spatial dimensions, a curve is described as $\Gamma = \{(x,y) : \phi(x,y) = 0\}$. The interior region is defined as $\Omega^- = \{\mathbf{x} : \phi(\mathbf{x}) < 0\}$, and the exterior region as $\Omega^+ = \{\mathbf{x} : \phi(\mathbf{x}) > 0\}$. The interface Γ is evolved in time by evolving the level-set function according to the level-set equation:

$$\frac{\partial \phi}{\partial t} + \mathbf{V} \cdot \nabla \phi = 0, \tag{1}$$

where **V** is the velocity field.

3.2. The reinitialization equation

The level-set method has been proven to be more robust and of higher accuracy when using the signed distance function to the interface as the level-set function. In order to maintained ϕ as a signed distance function, the following reinitialization equation [20] can be solved for a few iterations:

$$\phi_\tau + \text{sgn}(\phi^0)(|\nabla \phi| - 1) = 0, \tag{2}$$

where τ represents a fictitious time and sgn(ϕ^0) denotes the signum of ϕ^0. This algorithm thus reinitializes an arbitrary level-set function ϕ^0 into a signed distance function. The solution of this Hamilton–Jacobi equation produces shocks and rarefactions that can be captured using a combination of a Godunov scheme in space and a Total Variation Diminishing second-order Runge Kutta (TVD-RK2) scheme in time (see [17,10,7]). In this paper, we use the following discretization:

$$\frac{d\phi}{d\tau} + \operatorname{sgn}(\phi^0)\left[H_G\left(D_x^+\phi, D_x^-\phi, D_y^+\phi, D_y^-\phi\right) - 1\right] = 0, \tag{3}$$

where H_G is the numerical Godunov Hamiltonian defined as:

$$H_G(a,b,c,d) = \begin{cases} \sqrt{\max(|a^+|^2, |b^-|^2) + \max(|c^+|^2, |d^-|^2)} & \text{if } \operatorname{sgn}(\phi^0) \leqslant 0, \\ \sqrt{\max(|a^-|^2, |b^+|^2) + \max(|c^-|^2, |d^+|^2)} & \text{if } \operatorname{sgn}(\phi^0) > 0 \end{cases}$$

with $a^+ = \max(a,0)$ and $a^- = \min(a,0)$. The one-sided derivatives $D_x^\pm\phi$ and $D_y^\pm\phi$ are discretized using second-order accurate one-sided finite differences:

$$D_x^+\phi_{i,j} = \frac{\phi_{i+1,j} - \phi_{i,j}}{\Delta x} - \frac{\Delta x}{2}\operatorname{minmod}(D_{xx}\phi_{i,j}, D_{xx}\phi_{i+1,j})$$

and

$$D_x^-\phi_{i,j} = \frac{\phi_{i,j} - \phi_{i-1,j}}{\Delta x} - \frac{\Delta x}{2}\operatorname{minmod}(D_{xx}\phi_{i,j}, D_{xx}\phi_{i-1,j}),$$

with $D_{xx}\phi$ the second-order derivative of ϕ in the x-direction, computed with a central-difference discretization. The semi-discrete Eq. (3) is discretized in time with the TVD-RK2 scheme of Shu and Osher [17]:

$$\frac{\tilde{\phi}^{n+1} - \phi^n}{\Delta\tau} + \operatorname{sgn}(\phi^0)\left[H_G\left(D_x^+\phi^n, D_x^-\phi^n, D_y^+\phi^n, D_y^-\phi^n\right) - 1\right] = 0,$$
$$\frac{\tilde{\phi}^{n+2} - \tilde{\phi}^{n+1}}{\Delta\tau} + \operatorname{sgn}(\phi^0)\left[H_G\left(D_x^+\tilde{\phi}^{n+1}, D_x^-\tilde{\phi}^{n+1}, D_y^+\tilde{\phi}^{n+1}, D_y^-\tilde{\phi}^{n+1}\right) - 1\right] = 0$$

and then we define ϕ^{n+1} by simple averaging: $\phi^{n+1} = (\phi^n + \tilde{\phi}^{n+2})/2$.

The reinitialization is required not to change the original location of the interface. This is enforced following the idea of Russo and Smereka [12] of including the interface location, given by ϕ_0, in the stencils of the one-sided derivatives, and its modifications from Min and Gibou [7].

In the case of our local grid, nodes on the outer edge of the band are missing at least one immediate neighbor, which poses problems when approximating the different derivatives needed in the evolution and the reinitialization of the level-set. For such nodes, we choose to construct the missing neighbors using a linear extrapolation of the known values of ϕ using a fast marching method approach [15,22]. This could be improved upon by using a higher-order extrapolation.

3.3. Evolving the level-set function with a semi-Lagrangian scheme

If the velocity field is externally generated, i.e. it does not depend on the level-set, the level-set Eq. (1) is linear and semi-Lagrangian schemes (SLS) can be used. These schemes are unconditionally stable and thus avoid the standard CFL condition stating that the interface cannot move by more than one grid cell at every time step, in our case $\Delta x_{\text{smallest}}$, the smallest space step in the computational domain. The idea behind SLS is to reconstruct the trajectory of each individual particle of a system by starting from a point $\mathbf{x}$ and integrating numerically the equation governing its motion along its characteristic curves, thus tracing the particle back to its departure point $\mathbf{x_d}$.

In this article, we use a second-order accurate semi-Lagrangian method to solve the level-set Eq. (1) with the velocity field $\mathbf{V}$ externally generated. From the fact that solutions to hyperbolic problems are constant along characteristic curve, we have that for any grid point $\mathbf{x}^{n+1}$, $\phi^{n+1}(\mathbf{x}^{n+1}) = \phi^n(\mathbf{x_d})$, with ϕ^k the level-set function at time k. We use the second-order accurate mid-point method for locating the departure point, as explained in [23,7]:

$$\hat{\mathbf{x}} = \mathbf{x}^{n+1} - \frac{\Delta t}{2} \cdot V^n(\mathbf{x}^{n+1}),$$
$$\mathbf{x_d} = \mathbf{x}^{n+1} - \Delta t \cdot V^{n+\frac{1}{2}}(\hat{\mathbf{x}}).$$

We define the velocity $V^{n+\frac{1}{2}}$ at the mid-time step $t^{n+\frac{1}{2}}$ linearly from the previous velocities as $V^{n+\frac{1}{2}} = \frac{3}{2}V^n - \frac{1}{2}V^{n-1}$. Since the points $\mathbf{x_d}$ and $\hat{\mathbf{x}}$ are not grid points in general, the associated quantities $\phi^n(\mathbf{x_d})$ and $V^{n+\frac{1}{2}}(\hat{\mathbf{x}})$ are approximated using an interpolation procedure. In this work, we take the non-oscillatory interpolation procedure of [7].

3.4. Implementation of the hash table structure

We describe here the two key steps for the implementation of a cartesian grid on a hash table data structure, namely the construction of the local grid and its advection.

3.4.1. Building the adaptive grid

The adaptive grid stored in the hash table data structure is built in two steps: we first construct a full non-graded adaptive cartesian mesh before restraining it to the regions of interest and storing a refinement of those regions using the hash table data structure. We use a quadtree structure because we will compare the performance of the local grid method with a quadtree implementation [7], but in practice a standard uniform grid could serve the purpose of 'initializing' the grid.

The local mesh is then constructed using the most refined cells of the quadtree mesh. Those cells are further refined, the new nodes values being obtained by interpolation of the quadtree nodes values, and the corresponding nodes are stored in a hash table data structure. Since the initial number of nodes is given, we can build the hash table structure together with its hash function in an efficient way. We chose the size of the hash table s to be the smallest prime number larger than the number of nodes to be stored, and the hash function H to be

$$H(n) = i_n \cdot p_1 + j_n \cdot p_2 \pmod s,$$

where n is the node index, i_n and j_n are the grid coordinates of the node and p_1 and p_2 are two large prime numbers. We choose to define the size s of the hash table in the beginning of the algorithm and we do not modify it afterwards. Since the number of nodes can exceed s after the interface evolves, we handle collisions in the hash table with the direct chaining method presented in Section 2. Note that closed hashing would require the table to be resized when the number of nodes becomes larger than s.

3.4.2. Advecting the local grid

Constructing the local grid is computationally expensive because a reference mesh is needed, but the strength of the approach, in addition to the reduced number of points to handle, comes from the advection step that is a rather inexpensive and straightforward procedure. With structures like quadtrees, one has to rebuild the structure at each time step in such a way as to enforce that each node has known neighbors, a costly process. In the case of a local grid, there is no need to know the relation between neighboring nodes since the access to any node is in O(1). Therefore, grid nodes can be added or removed very simply and efficiently. The procedure for adapting the local grid to the changes undergone by the interface after advecting the level-set function ϕ is described in Algorithm 1.

Algorithm 1: Algorithm for the advection of the local grid

```
1: for all node $n$ in the local grid do
2:   if $|\phi(n)| > $ threshold then
3:     remove node $n$ from the local grid
4:   else
5:     for all neighbor $ngbd$ not in the local grid do
6:       compute $\phi(ngbd)$ by using the fast marching approach to solve $|\nabla\phi| = 1$
7:       if $|\phi(ngbd)| \leqslant$ threshold then
8:         add $ngbd$ to the local grid with the value $\phi(ngbd)$
9:       end if
10:    end for
11:  end if
12: end for
```

4. Validation

In order to validate our implementation of the local grid on a hash table data structure, we perform typical tests in two spatial dimensions. The local grid is located close to the interface forming a tube of width $5\Delta x$ on each side of the interface. The advection is done using the Semi-Lagrangian scheme in a band of $2\Delta x$ around the interface, and we use a time step $\Delta t = \Delta x$. The information is then propagated to the rest of the tube with a fast marching algorithm. The tube is required to be larger than $2\Delta x$ to allow second order accuracy.

4.1. Rotation of a disk

The first test is the rotation of a disk. We consider a domain $\Omega = [-1.5, 1.5]^2$ and a disk of radius $R = 0.3$ centered initially at $(0, 0.5)$. We rotate this disk under the following rigid-body velocity field:

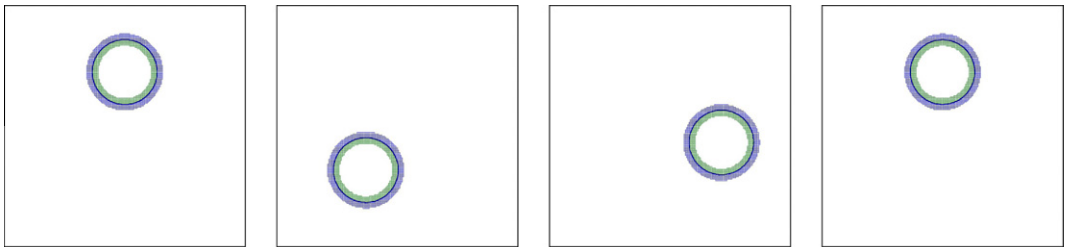

Fig. 4. Snapshots illustrating a full revolution of a disk using a local grid of equivalent uniform resolution 256×256 stored in a hash table data structure. The disk is initially centered at $(0, 0.5)$ and has a radius $R = 0.3$. The snapshots are taken, from left to right, at time $t = 0$, $t = 2\pi/4$, $t = 2\pi/7$ and $t = 2\pi$. The level-set is evolved using the semi-Lagrangian approach.

Table 1
Accuracy of the local grid stored in a hash table structure for the revolution of a disk. The disk is initially centered on $(0, 0.5)$ and the computation domain is $[-1.5, 1.5]^2$. The disk is evolved with the velocity field $(u, v) = (-y, x)$ until the time $t = 2\pi$. In this article, a 'resolution' of r^2 defines the grid size equivalent to a $r \times r$ discretization on uniform grid.

Resolution	Time (s)	L_∞ error of ϕ	Rate	L_1 error of ϕ	Rate	Mass loss (%)	Rate
64^2	5	3.21×10^{-2}		2.70×10^{-2}		17.15	
128^2	13	8.20×10^{-3}	1.97	6.65×10^{-3}	2.02	4.40	1.96
256^2	38	2.36×10^{-3}	1.80	1.74×10^{-3}	1.93	1.16	1.92
512^2	139	7.91×10^{-4}	1.58	4.65×10^{-4}	1.90	0.31	1.90
1024^2	530	3.46×10^{-4}	1.19	1.38×10^{-4}	1.75	0.092	1.75

Table 2
Resources used by the present local grid algorithm in comparison with the quadtree data structure for the disk revolution test. The disk is initially centered on $(0, 0.5)$ and the computation domain is $[-1.5, 1.5]^2$. The disk is evolved with the velocity field $(u, v) = (-y, x)$ until the time $t = 2\pi$. The number of nodes used is of the same order for both methods, and so is the memory required. Note that around 35% slots are empty for the local grid scheme, and a better hash function would improve those results. But according to [3] a total load of around 65% is close to the optimal case of 80%.

Resolution	Nb of nodes	Nb of slots	Nb of empty slots	Average occupied slots load	Memory (Kio)
Local grid					
64^2	404	409	194	1.88	18.94
128^2	806	823	229	1.35	37.78
256^2	1606	1609	588	1.57	75.28
512^2	3226	3301	1239	1.56	151.22
1024^2	6432	6719	2223	1.43	301.50
Quadtree					
64^2	309	NA	NA	NA	17.57
128^2	609	NA	NA	NA	35.24
256^2	1237	NA	NA	NA	72.19
512^2	2509	NA	NA	NA	146.91
1024^2	5001	NA	NA	NA	293.15

$$u(x,y) = -y,$$
$$v(x,y) = x$$

and rotate the disk until the final time $t = 2\pi$ is reached, i.e. we perform one complete revolution. The procedure is illustrated in Fig. 4. The accuracy of the procedure, given in Table 1, is monitored using the error close to the interface as well as the mass loss, which is a good measure of accuracy for the level-set method. The comparison with the quadtree data structure is developed in Table 2.

The error is computed close to the interface only, in a band of 1.2 Δx with Δx the resolution of the grid, since those points define the interface location, which we are interested in. The loss of mass is given by $|V_{initial} - V_{final}|/V_{initial}$, with V_t the area inside the interface (i.e. for $\phi < 0$) at time t. In practice, V_t is calculated by extending the local grid to the whole negative ϕ region, i.e. to the interior of the domain, and summing the area of the negative region contained in each cell. The area of the negative region contained in grid cells adjacent to the interface is computed as described by Min and Gibou in [6] The grid resolution corresponds to the number of points on a uniform grid with the same Δx.

In terms of memory, the hash table data structure is expected to be more efficient than the quadtree structure. However, for the local grid to provide accurate results, the band around the interface needs to be wide enough as to minimize the error incurred by linearly extrapolated nodes on the edge of the band. Also a smaller band limits the time step we can take in a semi-Lagrangian framework since the departure point could be outside the band. We found that at least $5\Delta x$ on both sides of

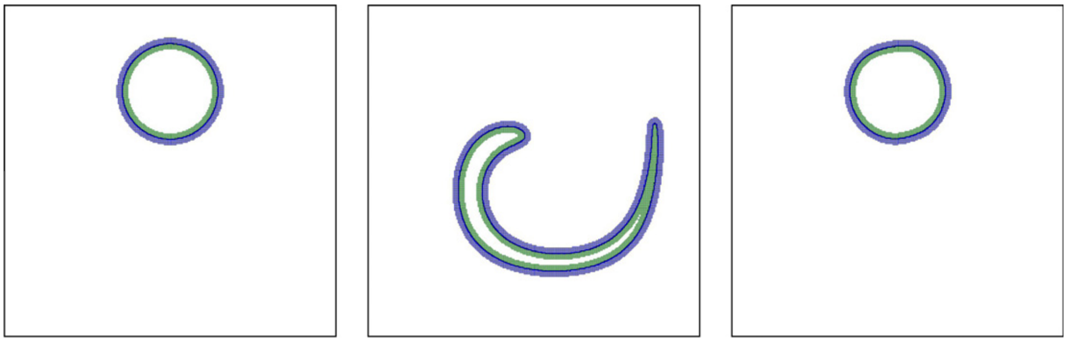
Fig. 5. Illustration of the deformation of a disk under a vortex velocity field using a local grid of equivalent uniform resolution 256×256 stored in a hash table data structure. The disk is evolved forward until time $t = 1$ and then backward to its initial position. The snapshots have been taken from left to right at respective times $t = 0,\ t = 1 = \frac{1}{2}t_{final}$ and $t = t_{final}$. The scheme used to evolved the level-set is based on the Semi-Lagrangian approach.

the interface are needed, while a band of only $3\Delta x$ are needed in the case of the quadtree. Overall, this restriction can lead to structures that have a size equivalent to or higher than a quadtree data structure. The hash function we use provides an average access to the nodes in O(1), since the average load of the hash table occupied slots is close to one. The heaviest load observed is six nodes for a single slot.

4.2. Motion under a vortex velocity field

The second test, based on a proposition by [2], is more challenging as the interface thins out under the velocity field: We consider a domain $\Omega = [0,1]^2$ and a disk of radius $R = 0.15$ centered initially at (0.5, 0.75). We deform the level-set under the divergence free velocity field:

Table 3
Accuracy of the local grid stored in a hash table structure for the evolution of a disk in a vortex velocity field $(u,v) = (-\sin^2(\pi x)\sin(2\pi y), \sin^2(\pi y)\sin(2\pi x))$. The disk is initially centered on (0.5, 0.75) and the computation domain is $[0,1]^2$. The disk is evolved until the time $t = 1$ and is then evolved back to its initial state with the inverse velocity field. In this article, "resolution" means the number of grid points for an uniform grid of the same accuracy. The L_∞ and L_1 errors of ϕ are computed on the nodes adjacent to the interface. As one can observe, the method is of order close to two.

Resolution	Time (s)	L_∞ error of ϕ	Rate	L_1 error of ϕ	Rate	Mass loss (%)	Rate
64^2	8	3.74×10^{-2}		1.43×10^{-2}		16.34	
128^2	22	1.81×10^{-2}	1.05	4.74×10^{-3}	1.59	5.58	1.55
256^2	76	8.53×10^{-3}	1.09	1.49×10^{-3}	1.67	1.84	1.60
512^2	591	3.98×10^{-3}	1.10	4.72×10^{-4}	1.66	0.61	1.59
1024^2	2509	1.80×10^{-3}	1.14	1.61×10^{-4}	1.55	0.20	1.61

Table 4
Resources used by the algorithm in comparison with the quadtree data structure for the evolution of a disk in a vortex velocity field. The disk is initially centered on (0.5, 0.75) and the computation domain is $[0,1]^2$. The disk is evolved until the time $t = 1$ and is then evolved back to its initial state. The minimum and maximum average loads of the occupied slots over the whole procedure are monitored, together with the number of nodes and the memory usage. The number of nodes used by the local grid is approximately twice the number of nodes used by the quadtree, leading to a structure that requires twice more memory.

		64^2	128^2	256^2	512^2	1024^2
Local grid	Number of slots	1103	2713	6427	13,577	28,111
	Min average occupied slots load	1.00	1.00	1.00	1.05	1.02
	Max average occupied slots load	1.00	1.03	1.05	1.20	1.19
	Min number of empty slots	354	798	2086	4654	1305
	Max number of empty slots	657	1722	4341	9333	6575
	Min number of nodes	504	1141	2367	4793	9630
	Max number of nodes	948	2690	6376	13,478	27,563
	Min memory usage	18.36	53.48	110.95	224.67	451.25
	Max memory usage	44.44	126.09	298.88	631.78	1292.02
Quadtree grid	Min number of nodes	283	588	1206	2426	4844
	Max number of nodes	484	1143	2744	6177	13,147
	Min memory usage	16.34	33.84	70.20	141.52	282.78
	Max memory usage	28.19	68.26	164.14	366.36	776.65

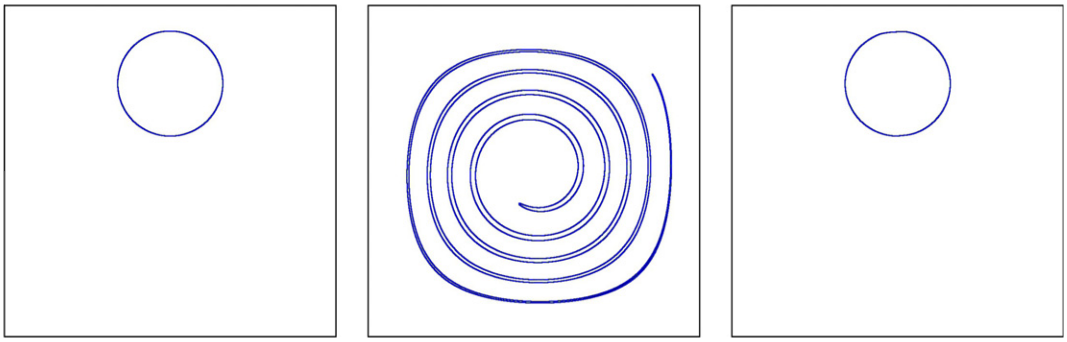

Fig. 6. Deformation of a disk under the vortex velocity field using a local grid of equivalent uniform resolution 8192 × 8192. The disk (on the left) is deformed until the time $t = 6$ (second picture), then the velocity field is inverted and the disk is evolved to its initial shape (right picture). The mass loss is 0.20%, and the maximum memory usage is 51 Mio.

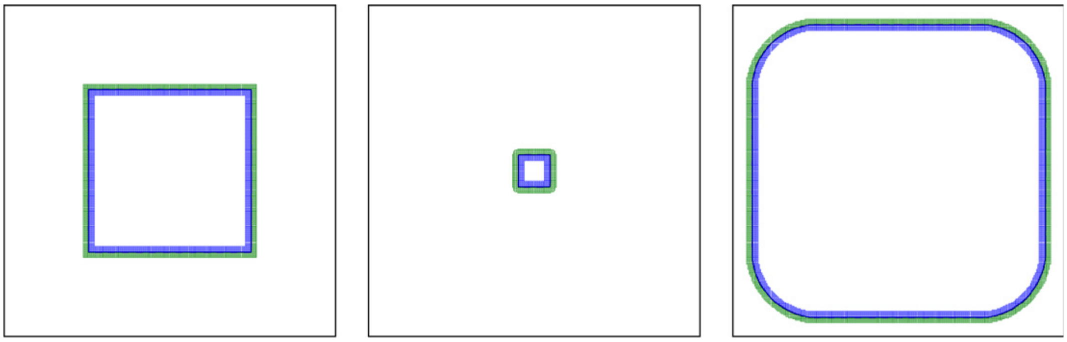

Fig. 7. Deformation of a square under a normal velocity to the interface. The original square, on the left, is contracted in the middle, and the same original square is expanded on the right. This illustrates the ability of our framework to capture shock and rarefaction solutions.

$$u(x,y) = -\sin^2(\pi x)\sin(2\pi y),$$
$$v(x,y) = \sin^2(\pi y)\sin(2\pi x).$$

This deformation is illustrated in Fig. 5, and the numerical results are collected in Tables 3 and 4. As can be observed, and as expected from the results obtained in [7], the order of convergence is not at good as for the case of the rotation for coarse grids. The method is of order close to two for the mass loss and the L_1 error, and of order slightly more than 1 for the L_∞ error. As explained by Min and Gibou, and reported in [8], this is due to the fact that part of the geometry is under resolved as it deforms. In particular, part of the tail of the interface will always be under-resolved, no matter how high the resolution.

The disk can be deformed further under the velocity field, and the largest the deformation is the hardest it is to recover the initial disk shape. This is precisely a situation where high resolution implementations can provide accurate results. Fig. 6 illustrates the deformation of the disk until the time $t = 6$, before being rewinded back to its initial state. As can be observed, the final result is close to the initial disk. We find a mass loss of 0.2%, which is quite small for such an extreme case of deformation. Therefore, the local grid we implemented succeeds in capturing the general features of this important deformation.

4.3. Case of shock and rarefaction solutions

In the case where the velocity **V** in Eq. (1) depends on the level-set function ϕ, Eq. (1) admits nonlinear solutions related to shock and rarefaction waves in conservation laws. In order to illustrate the ability of our framework to capture such solutions, we consider the case of a unit square moving under a normal velocity $\mathbf{V}_n = \pm\mathbf{n}$, where $\mathbf{n}$ is the outward normal to the interface. The level-set Eq. (1) is solved using a Godunov scheme similar to the one presented for the reinitialization procedure in Section 3.2. Fig. 7 depicts the correct shock and rarefaction solutions.

5. Concluding remarks

We have presented a successful implementation of the level-set method on a hash table data structure. It is important to mention that the development of the code for the hash table based level-set method, with its linear organization, was easier and more straightforward than the implementation of the quadtree data structure, which is recursive and intricate. We note that the hash function we provide produces satisfactory results and enables a meaningful comparison of the method's performances with the quadtree data structure. We also note that the extrapolation procedure to define missing neighbors is only first-order accurate, which impacts the overall accuracy. The analysis of the numerical tests shows that even if only the nodes close to the interface are stored, the method is less efficient than the quadtree data structure for three main reasons: (1) to obtain accurate results, a rather large band is required close to the interface, which counterbalances the absence of grid nodes far from the interface; (2) the performances are deteriorated by extrapolation procedures on the outer edges of the local grid and (3) the width of the band restricts the time step and slows down the method. These issues may be resolved by careful development of different algorithms. In addition, the hash table data structure is more suitable for parallelization than the quadtree data structure, but as it is, we find that a quadtree data structure seems more adapted than the hash table data structure for level-set algorithms.

Acknowledgement

This research was supported in part by ONR under grant agreement N00014-11-1-0027, by the National Science Foundation under grant agreement CHE 1027817, by the Department of Energy under grant agreement DE-FG02-08ER15991, by the Institute for Collaborative Biotechnologies through contract No. W911NF-09-D-0001 from the US Army Research Office and by the W.M. Keck Foundation.

References

[1] D. Adalsteinsson, J. Sethian, A fast level set method for propagating interfaces, J. Comput. Phys. 118 (1995) 269–277.
[2] J.B. Bell, P. Colella, H.M. Glaz, A second order projection method for the incompressible Navier-Stokes equations, J. Comput. Phys 85 (1989) 257–283.
[3] T.H. Cormen, Introduction to Algorithms, The MIT press, 2001.
[4] F. Losasso, F. Gibou, R. Fedkiw, Simulating water and smoke with an octree data structure, ACM Trans. Graph. (SIGGRAPH Proc.) (2004) 457–462.
[5] Ken Museth Michael B. Nielsen, Dynamic tubular grid: an efficient data structure and algorithms for high resolution level sets, J. Sci. Comput. 26 (3) (2006), doi:10.1007/s10915-005-9062-8.
[6] C. Min, F. Gibou, Geometric integration over irregular domains with application to level set methods, J. Comput. Phys. 226 (2007) 1432–1443.
[7] C. Min, F. Gibou, A second order accurate level set method on non-graded adaptive Cartesian grids, J. Comput. Phys. 225 (2007) 300–321.
[8] E. Olsson, G. Kreiss, A conservative level set method for two phase flow, J. Comput. Phys. 210 (2005) 225–246.
[9] S. Osher, R. Fedkiw, Level Set Methods and Dynamic Implicit Surfaces, Springer-Verlag, New York, NY, 2002.
[10] S. Osher, J. Sethian, Fronts propagating with curvature-dependent speed: algorithms based on Hamilton–Jacobi formulations, J. Comput. Phys. 79 (1988) 12–49.
[11] S. Popinet, Gerris: a tree-based adaptive solver for the incompressible euler equations in complex geometries, J. Comput. Phys. 190 (2003) 572–600.
[12] G. Russo, P. Smereka, A remark on computing distance functions, J. Comput. Phys. 163 (2000) 51–67.
[13] H. Samet, The Design and Analysis of Spatial Data Structures, Addison-Wesley, New York, 1989.
[14] H. Samet, Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS, Addison-Wesley, New York, 1990.
[15] J. Sethian, A fast marching level set method for monotonically advancing fronts, Proc. Natl. Acad. Sci. 93 (1996) 1591–1595.
[16] J.A. Sethian, Level Set Methods and Fast Marching Methods, Cambridge University Press, Cambridge, 1999.
[17] C.-W. Shu, S. Osher, Efficient implementation of essentially non-oscillatory shock capturing schemes, J. Comput. Phys. 77 (1988) 439–471.
[18] K. Steele, D. Cline, P.K. Egbert, J. Dinerstein, Modeling and rendering viscous liquids, Comput. Anim. Virtual Worlds 15 (3–4) (2004) 183–192.
[19] J. Strain, Tree methods for moving interfaces, J. Comput. Phys. 151 (1999) 616–648.
[20] M. Sussman, E. Fatemi, P. Smereka, S. Osher, An improved level set method for incompressible two-phase flows, Comput. Fluids 27 (1998) 663–680.
[21] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, et al, Collision Detection for Deformable Objects, Computer Graphics Forum, vol. 24, Wiley Online Library, 2005, pp. 61–81.
[22] J. Tsitsiklis, Efficient algorithms for globally optimal trajectories, IEEE Trans. Autom. Control 40 (1995) 1528–1538.
[23] D. Xiu, G. Karniadakis, A semi-Lagrangian high-order method for Navier–Stokes equations, J. Comput. Phys. 172 (2001) 658–684.