Computational and Applied Mathematics (2022) 41:19
https://doi.org/10.1007/s40314-021-01716-y

Efficient implementation of characteristic-based schemes on unstructured triangular grids

S. Cacace[1] · **R. Ferretti**[1]

Received: 14 July 2021 / Revised: 10 November 2021 / Accepted: 25 November 2021 /
Published online: 18 December 2021

Abstract
Using characteristics to treat advection terms in time-dependent PDEs leads to a class of schemes, e.g., semi-Lagrangian and Lagrange–Galerkin schemes, which preserve stability under large Courant numbers, and may therefore be appealing in many practical situations. Unfortunately, the need of locating the feet of characteristics may cause a serious drop of efficiency in the case of unstructured space grids, and thus prevent the use of large time-step schemes on complex geometries. In this paper, we perform an in-depth analysis of the main recipes available for characteristic location, and propose a technique to improve the efficiency of this phase, using additional information related to the advecting vector field. This results in a clear improvement of execution times in the unstructured case, thus extending the range of applicability of large time-step schemes.

Keywords Large time-step schemes · Unstructured grids · Point location · Computational complexity

Mathematics Subject Classification 65-04 · 65D18 · 65M06 · 65M25

1 Introduction

Born in the 50s in the framework of environmental fluid dynamics and Numerical Weather Prediction, large time-step, characteristic-based schemes have become in recent years a useful tool for various PDE models, mainly of hyperbolic type. While this class of schemes collects various techniques (for example semi-Lagrangian Falcone and Ferretti 2013, Lagrange–Galerkin Douglas and Russell 1982; Pironneau 1982, ELLAM Russell and Celia 2002) having in common the use of the method of characteristics to treat advection terms, to fix

Communicated by Raphaèle Herbin.

✉ R. Ferretti
ferretti@mat.uniroma3.it

S. Cacace
cacace@mat.uniroma3.it

[1] Dipartimento di Matematica e Fisica, Università Roma Tre, Largo S. Leonardo Murialdo, 1, Rome, Italy

ideas we will refer in what follows to the case of semi-Lagrangian (SL) schemes, which probably employ this strategy in its simplest form. We consider, as a model problem, the simple variable-coefficient advection equation with Dirichlet boundary conditions,

$$\begin{cases} u_t + f(x,t)\cdot\nabla u = 0 & (x,t)\in\Omega\times\mathbb{R}^+, \\ u(x,t) = \gamma(x,t) & (x,t)\in\Gamma_{in}(t)\times\mathbb{R}^+, \\ u(x,0) = u_0(x) & x\in\Omega, \end{cases} \tag{1}$$

in which $\Omega\subset\mathbb{R}^d$, $f:\Omega\times\mathbb{R}^+\to\mathbb{R}^d$, $\gamma:\partial\Omega\times\mathbb{R}^+\to\mathbb{R}$, $u_0:\Omega\to\mathbb{R}$, $\Gamma_{in}(t)$ is the portion of the boundary $\partial\Omega$ where $f(x,t)$ points inwards. The solution of (1) may be represented via the well-known formula of characteristics, which will be concisely recalled here. We start by defining $X(x,t;s)$ as the solution at time s of the ordinary differential equation

$$\begin{cases} \frac{d}{ds}X(x,t;s) = f(X(x,t;s),s) & s\in\mathbb{R}, \\ X(x,t;t) = x, \end{cases}$$

that is, the trajectory moving with velocity $f(X,s)$ and passing through the point x at time t. If, for some time s, the trajectory $X(x,t;s)$ falls outside Ω, then we define

$$\bar{s}(x,t) = \sup\{s\le t : X(x,t;s)\notin\Omega\},$$

that is, the last time at which $X(x,t;s)$ hits the boundary of Ω. Then, the solution u has the representation

$$u(x,t) = \begin{cases} u_0(X(x,t;0)) & \bar{s}(x,t) < 0 \\ \gamma(X(x,t;\bar{s}(x,t)),\bar{s}(x,t)) & \bar{s}(x,t)\ge 0. \end{cases} \tag{2}$$

We assume that f is C^1 with bounded derivatives on $\Omega\times\mathbb{R}^+$, so that:

$$\|f(x_1,t_1) - f(x_2,t_2)\| \le L_x\|x_1-x_2\| + L_t\|t_1-t_2\|,$$

with L_x, L_t denoting the two Lipschitz constants associated to, respectively, space and time increments, and $\|\cdot\|$ denoting the Euclidean norm. Clearly, such a framework is ultimately directed towards nonlinear equations in which the advection term has a smooth space and time dependence, at least in a large majority of the computational domain.

Once a time grid $t_n = n\Delta t$ has been set, a SL discretization of (1) uses the representation formula (2) written on a single time step, i.e.,

$$u(x,t_{n+1}) = \begin{cases} u(X(x,t_{n+1};t_n),t_n) & \bar{s}(x,t_{n+1}) < t_n \\ \gamma(X(x,t_{n+1};\bar{s}(x,t_{n+1})),\bar{s}(x,t_{n+1})) & \bar{s}(x,t_{n+1})\ge t_n. \end{cases}$$

To turn this relationship into a computable scheme, we first build a space grid with space scale Δx and with nodes in the set $\mathcal{V} = \{x_i\}_{i=1,\dots,N}$. Then, the foot of the characteristic $X(x_i,t_{n+1};t_n)$ is replaced by a numerical (e.g., one-step) approximation $X^{\Delta}(x_i,t_{n+1};t_n)$, and the value $u(\cdot,t_n)$ by an interpolation $I[V^n](\cdot)$, constructed using the vector $V^n = (v_1^n\cdots v_N^n)$ of the node values at time t_n, with v_i^n corresponding to the i-th node x_i and the n-th time step t_n. Here and in what follows, $N = |\mathcal{V}|$ will denote the total number of nodes.

Following Falcone and Ferretti (2013), we will also need an estimate of the time $\bar{s}$ for the discretized representation formula. For example, if Ω is defined by the inequality $g(x) < 0$

for a suitable function $g : \mathbb{R}^d \to \mathbb{R}$, this estimate might be obtained as the solution $\bar{s}_i^{n+1}$ of the equation

$$g\left(X^{\Delta}(x_i, t_{n+1}; s)\right) = 0, \tag{3}$$

to be solved every time $X^{\Delta}(x_i, t_{n+1}; t_n) \notin \Omega$ (if Δt is not too large, we expect that only one solution exists). The scheme is therefore in the form

$$v_i^{n+1} = \begin{cases} I[V^n]\left(X^{\Delta}(x_i, t_{n+1}; t_n)\right) & X^{\Delta}(x_i, t_{n+1}; t_n) \in \Omega \\ \gamma\left(X^{\Delta}(x_i, t_{n+1}; \bar{s}_i^{n+1})), \bar{s}_i^{n+1}\right) & X^{\Delta}(x_i, t_{n+1}; t_n) \notin \Omega. \end{cases} \tag{4}$$

In (4), the discrete approximation $X^{\Delta}(x_i, t_{n+1}; t_n)$ of $X(x_i, t_{n+1}; t_n)$ might be computed in the simplest case by applying, backward in time, the explicit Euler scheme:

$$X^{\Delta}(x_i, t_{n+1}; t_n) = x_i - \Delta t \; f(x_i, t_{n+1}). \tag{5}$$

Then, $\bar{s}_i^{n+1}$ is obtained by reducing Δt so as to bring $X^{\Delta}(x_i, t_{n+1}; \bar{s}_i^{n+1})$ on the boundary: in the Euler scheme (5) this amounts to solving, with respect to s, the equation

$$g\big(x_i - (t_{n+1} - s) f(x_i, t_{n+1})\big) = 0. \tag{6}$$

A different approach to boundary conditions is to assign a value to the point $X(x_i, t_{n+1}; t_n)$, whenever it falls outside Ω, by extrapolation (see Bonaventura et al. 2021). This approach requires to build an additional mesh to obtain extrapolated values, and will not be pursued here.

Concerning the interpolation, this step is typically accomplished in local form, using the values of the numerical solution at nodes close to $X^{\Delta}(x_i, t_{n+1}; t_n)$. Selecting the relevant values requires an $\mathcal{O}(1)$ cost on a structured array of nodes, and therefore is not a critical issue from the viewpoint of complexity. On the other hand, when working on unstructured (typically, but not necessarily, triangular) grids, the interpolation is usually computed via Lagrange finite elements: interpolating at a given point requires first select the element containing the point, and then use the Lagrange basis associated to this specific triangle. In comparison with the structured case, in the unstructured case the former phase (point location) represents a clear bottleneck, which either prevents the use of large time-step schemes, or causes a substantial drop in their efficiency. In fact, as we will show in the last section, the point location phase covers a significant part of the total CPU time.

Despite this difficulty, a certain amount of literature has been devoted to unstructured implementations of characteristic-based schemes; in most cases, however, we found that an in-depth discussion of the efficiency issues is eluded. In other cases, practical recipes are provided: the two typical techniques used are on one hand the quadtree search (see Giraldo 1998, 2000), on the other the tracking of characteristics via substepping, which requires in general to move from one element to its neighbour, thus making the search easier (see Restelli et al. (2006) for the case of a triangular mesh, Boscheri et al. (2013; 2020) for a Voronoi mesh). We will briefly review the ideas behind these techniques in the next section.

To the authors' knowledge, the optimal complexity of known general-purpose point location algorithms is $\mathcal{O}(\log N)$, where N is the number of grid nodes. In this paper, we will show that this complexity may be brought to $\mathcal{O}(1)$, by using the information related to the specific problem under consideration, that is, moving from a general-purpose algorithm to an algorithm tuned on the case of characteristics, at the price of introducing some additional data structures related to the mesh. A first motivation for this study is to apply efficient semi-Lagrangian techniques to Navier–Stokes equations on non-orthogonal geometries (Bonaventura et al. 2020).

The paper is structured as follows. In Sect. 2, we review the two main techniques to locate a point in an unstructured triangulation, and study their computational complexity. In Sect. 3, we study in detail some possibilities to improve the point location algorithms. Last, in Sects. 4 and 5 we present a numerical validation for the algorithm and draw some conclusions.

2 Locating a point on a triangular grid: some basic facts

Consider a triangulation of the set Ω in the form $\mathcal{T} = \bigcup_k \mathcal{T}_k$, where each simplex $\mathcal{T}_k$ is called an *element*. We assume that this triangulation satisfy the usual requirements of the finite element setting, in particular, that it does not contain "hanging nodes", and that the interior sets of the elements do not intersect one another. In what follows, we will identify Ω and $\mathcal{T}$, that is, we will neglect the possible inaccuracies in the reproduction of the boundary $\partial\Omega$, whenever Ω is not a polygonal set itself. We start by stating formally the *point location* problem:
(PL) Given a point x, a set $\Omega \subset \mathbb{R}^d$ and a covering triangulation $\mathcal{T} = \bigcup_k \mathcal{T}_k$, then:

1. If $x \in \mathcal{T}$, find the index k such that $x \in \mathcal{T}_k$;
2. If $x \notin \mathcal{T}$, return a suitable flag.

Our interest here is to solve efficiently this problem for the whole set of points $X^{\Delta}(x_i, t_{n+1}; t_n)$. The case 1. allows one to find the element in which local interpolation should be performed, while case 2. indicates the need to enforce boundary conditions via (3).

In this section, we briefly review two major approaches to point location on triangulations, namely the quadtree search and the barycentric walk search, including an experimental analysis of their computational complexities. To fix ideas, we will mostly refer to the case $d = 2$ (i.e., the simplices $\mathcal{T}_k$ are triangles), and discuss, when necessary, the generalization to higher dimensions.

2.1 Quadtree algorithm

As far as the authors know, the first appearance of this algorithm dates back to the 70s (Finkel and Bentley 1974). The algorithm is based on an auxiliary data structure of quadtree type, i.e., a tree where all nodes but the leaves have precisely four children. Each node (also termed as a *quad*) corresponds to a rectangle, starting with the quad associated to the root and containing the whole triangulation, and each successive level divides the quad into four. Once fixed an integer $q \geq 2$, the subdivision is stopped as soon as one of the following conditions is satisfied (see Fig. 1):

(a) The quad intersects a number n_t of triangles such that $1 \leq n_t \leq q$, and contains no vertex;
(b) The quad contains exactly one vertex, regardless of the number of triangles n_v it joins;
(c) The quad does not intersect the triangulation.

A leaf of the tree is generated at the final level of the subdivision, and the list of triangles intersecting the final quad is associated with the leaf. A point location requires to visit the tree: once found the leaf containing the point, the location of the point in the triangulation is completed with a number $\mathcal{O}(\max(q, n_v))$ of operations. The tree is unbalanced in general; however, for a regular Delaunay triangulation we can reasonably assume that the average

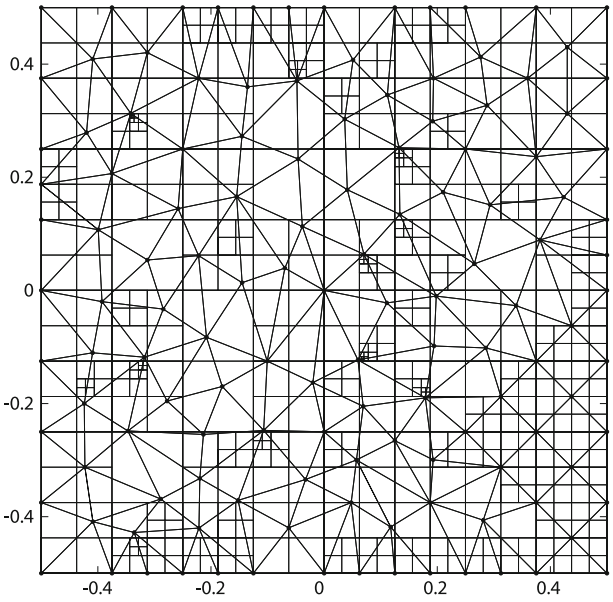


Fig. 1 Quadtree partition associated with an unstructured triangulation of $\Omega = \left[-\frac{1}{2}, \frac{1}{2}\right]^2$, with $q = 3$

complexity for the visit of the tree (and, therefore, for one point location) is $\mathcal{O}(\log N)$, where N is the number of grid nodes, while the complexity of the checks to be done at the leaves is constant. The complexity of a single point location takes then the form

$$\mathcal{O}\left(C_1^Q + C_2^Q \log N\right).$$

In principle, the complexity of the visit should depend on q; however, a decrease of q causes at the same time a higher depth of the quadtree and a shorter list of elements to be checked at a leaf, and vice versa for an increase of q (for example, in the grid of Fig. 1, the relatively low value of $q = 3$ causes a tree depth of ten levels with only 218 elements). Except for the lowest values of q, which may lead to an extremely deep tree, the two effects tend to compensate, as shown by the following numerical test.

Quadtree: numerical example. We show here an experimental assessment of the performance of quadtree search. In the first plots (Fig. 2), we consider meshes of size N ranging from about 10^5 to about $1.7 \cdot 10^6$. In the left plot, we compare the depth of the tree obtained for different values of q: the plot shows a clear saturation effect, and for $q \geq 7$ the depth becomes constant for all meshes. This effect might be explained by the fact that, in a regular Delaunay mesh, this is the typical maximum number of triangles joined at a node, this meaning that leaves of both types (a) and (b) contain typically a similar number of triangles (no more than q). Then, the tree is likely to be more balanced, and at the increase of q we don't expect any improvement in the depth of the tree.

In the right plot, we compare the search time for a set of N random query points (the same size of the mesh), obtained with different values of q; here, it is clear that this parameter has a small effect, if any at all, on the execution times. From now on, we will choose $q = 7$ in all the tests.

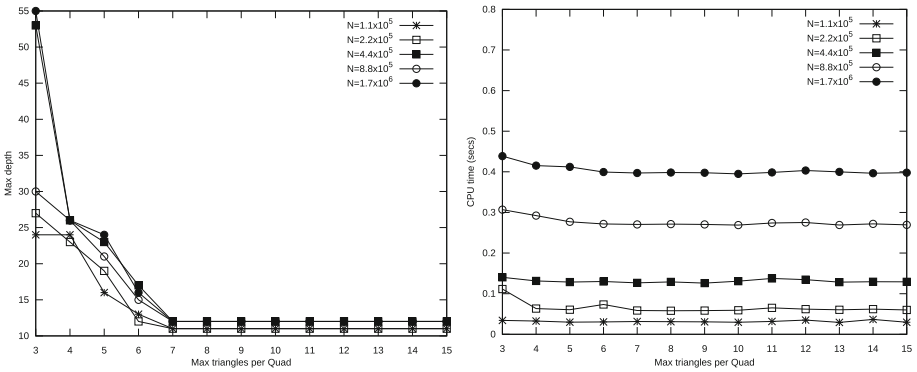


Fig. 2 Depth of the quadtree versus q (left) and search times versus q (right), for $10^5 \lesssim N \lesssim 1.7 \cdot 10^6$

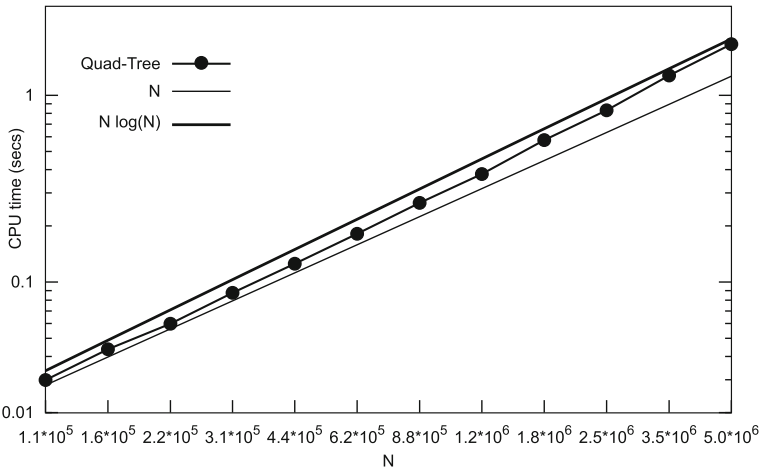


Fig. 3 Execution times for the quadtree search on the whole mesh, compared with N and $N \log N$ curves, for $10^5 \leq N \leq 5 \cdot 10^6$

Next, we report in Fig. 3 the search times for N random query points, with mesh size N ranging from about 10^5 to about $5 \cdot 10^6$, compared to both N and $N \log N$ orders. As we are performing N searches, each one of the expected complexity $\mathcal{O}(\log N)$, the expected order for the total CPU time is $\mathcal{O}(N \log N)$. In practice, while this asymptotic behaviour is confirmed, the intermediate scenario can be somewhat less predictable. For example, Fig. 3 shows an almost linear behavior even for a relatively large number of nodes (about $5 \cdot 10^5$). This occurs because, under the subdivision rules described above, the resulting, unbalanced quadtree structure reaches its maximum depth only in a few regions, compared to the whole mesh (see again Fig. 1).

Note that, even if the $\log N$ term might seem irrelevant in terms of order of complexity, in the case of a large number of elements it can increase multiple times the global execution interval, as we will show in the numerical tests. Note also that this complexity is retained by the three-dimensional version, based on a tree structure in which all internal nodes have eight children (the so-called *octree*).

2.2 Barycentric walk

Among the various algorithms which locate a point by stepping along the elements of the triangulation, we review here the so-called *barycentric walk*, which is probably the simplest one – complexity issues are in all cases similar for all the algorithms of this class (see Devillers et al. 2001 for an extensive review). In this algorithm, in order to locate the point x, we start from a given element of the triangulation and change element on the basis of the barycentric coordinates of x with respect to the current element, as shown in Fig. 4. Given the nodes x_1, x_2 and x_3 of the element T (with $x_i = (\xi_i, \eta_i)$), we write $x = (\xi, \eta)$ by means of the barycentric coordinates $\theta_1, \theta_2, \theta_3$ as

$$x = \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3,$$

with the θ_i given by

$$\begin{cases} \theta_1 = \frac{(\eta_2-\eta_3)(\xi-\xi_3)+(\xi_3-\xi_2)(\eta-\eta_3)}{(\eta_2-\eta_3)(\xi_1-\xi_3)+(\xi_3-\xi_2)(\eta_1-\eta_3)} \\ \theta_2 = \frac{(\eta_3-\eta_1)(\xi-\xi_3)+(\xi_1-\xi_3)(\eta-\eta_3)}{(\eta_2-\eta_3)(\xi_1-\xi_3)+(\xi_3-\xi_2)(\eta_1-\eta_3)} \\ \theta_3 = 1-\theta_1-\theta_2 \end{cases} \tag{7}$$

and we repeat the following steps:

(a) if all the barycentric coordinates are nonnegative, then $x \in T$ and the point location is complete;
(b) if there exists (at least) one negative coordinate, we look for the node associated with the negative coordinate of largest magnitude. Then:
   (i) If there exists a triangle adjacent to the opposite side, change element passing to this triangle, and repeat the computation of the barycentric coordinates on the new element;
   (ii) If no triangle is adjacent to the opposite side, mark the point x as "out of the triangulation".

For example, in the case shown in Fig. 4, the only negative coordinate is θ_1, so that we change element from T to the triangle having in common with T the x_2x_3 side, if this triangle exists; otherwise, we consider x as an external point (see Sect. 3 for implementation details).

Note that, strictly speaking, case $(b.ii)$ is treated in inexact form. On a general (e.g, nonconvex) triangulation, even if $x \in \mathcal{T}$, the barycentric walk might eventually fall outside $\mathcal{T}$ at an intermediate step. However, on the Delaunay triangulation of a sufficiently regular set Ω, and if the point x is not "too far" from $\mathcal{T}$ (which is definitely the case for the feet of characteristics X^{Δ}), we can reasonably rule out this situation.

Remark 1 The point location ends as soon as we are in an element where all the barycentric coordinates are nonnegative. The finite element-type interpolation on this last element can be immediately computed in terms of these parameters, which are invariant with respect to affine transformations of the reference element. For example, in the $\mathbb{P}_1$ case, we have

$$I[V](x) = \theta_1 v_1 + \theta_2 v_2 + \theta_3 v_3.$$

Note also that, whichever algorithm is used for locating the feet of characteristics, the interpolation phase requires to compute the barycentric coordinates to interpolate. Therefore, in what follows, the comparison among the various recipes will always include this computation.

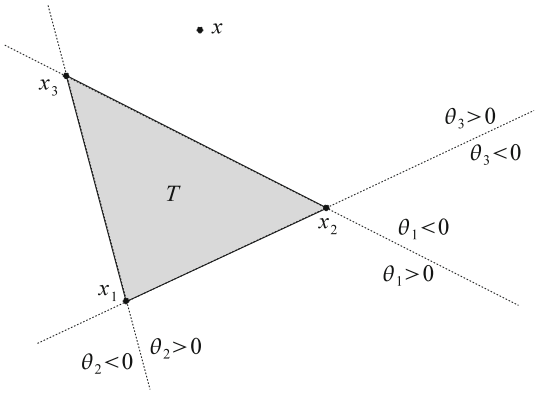


Fig. 4 Change of element on the basis of barycentric coordinates

Concerning the complexity, each change of element has a constant cost, and we can assume that, on a regular Delaunay mesh, the number of elements visited during a walk is asymptotically proportional to its length (this is not necessarily true on graded or anisotropic meshes). Therefore, if we want to locate a query point Q starting the search from a point P (i.e., from a triangle containing this point), the number of walk steps is

$$\mathcal{O}\left(\|Q-P\|\sqrt{N}\right), \tag{8}$$

where $\sqrt{N}$ is inversely proportional to the space scale of the triangulation (in higher dimensions, this term should be replaced by $N^{1/d}$). The location of Q has therefore a complexity of the order of

$$\mathcal{O}\left(C_1^B + C_2^B\|Q-P\|\sqrt{N}\right), \tag{9}$$

in which the constant term accounts for operations which cannot be avoided even in case of a very small distance $\|Q-P\|$: at least one computation of the barycentric coordinates, and, possibly, some change of element. In particular, we observe that if P is fixed, the complexity will be heavier than the quadtree search ($\sqrt{N}$ versus $\log N$). We validate our complexity analysis with the following numerical test.

Barycentric walk: numerical example. We first show, on a rough mesh of $N = 250$ nodes, the typical barycentric walk for the location of a query point starting from a mesh node, see Fig. 5. The initial triangle is randomly chosen from those containing the starting node. Note that the barycentric walk is forced, by construction, to perform a large number of steps around those nodes that lie on (or are close to) the line connecting the start and endpoints. This effect might locally increase the number of steps of the walk, although, as we will soon show, the average number retains a linear dependence on the distance.

Now, we provide an experimental assessment of the complexity of the algorithm in terms of the distance between the start point and the query point. We choose a fixed mesh of $N \sim 4{\cdot}10^5$ nodes, and we compute the total CPU time to locate, for each node, a corresponding query point at given distance. The results are reported in Fig. 6. For small distances we clearly observe a plateau in the search times, corresponding to the constant term C_1^B in (9), while the behaviour is linear, as expected, when the distance increases (compare with the reference line for the linear increase). To show that the conclusions of this work are applicable to higher space dimensions, this test has also been performed with a three-dimensional tetrahedral grid, obtaining the results shown in Fig. 7, which completely parallel the 2-d case.

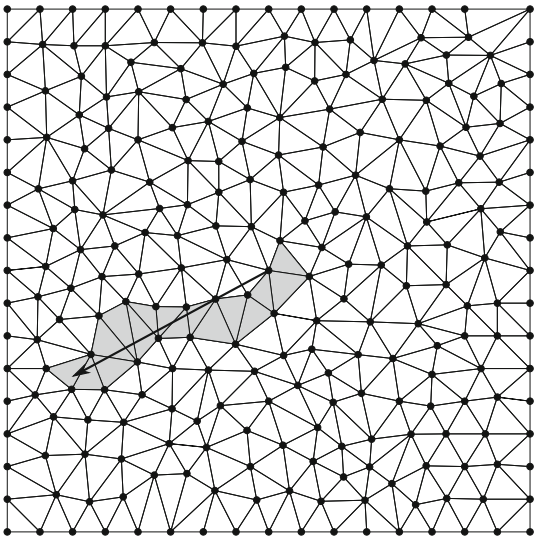


Fig. 5 Locating a point via barycentric walk (gray triangles) on a Delaunay triangulation (color figure online)

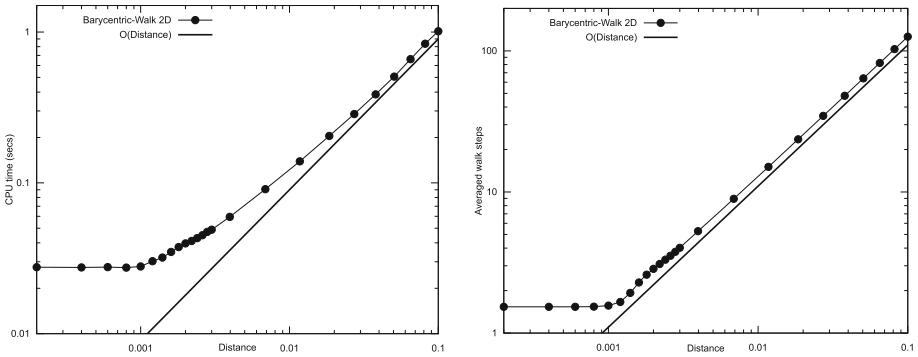


Fig. 6 Execution times (left) and averaged walk steps (right) for the barycentric walk search on the whole mesh ($d = 2$, $N \sim 4 \cdot 10^5$), versus distance between start and endpoints, compared with the line of linear increase

Remark 2 In this example, the space scale of the triangulation is estimated by $1/\sqrt{N} \sim 1.5 \cdot 10^{-3}$, while we observe that CPU time begins to grow already at a smaller distance. This reflects the fact that the actual intersection between the trajectory and each element amounts in general to a fraction of the space scale, as clearly shown by Fig. 5. The behavior of the averaged walk steps is similar, in particular, we observe a value of about 1.5 for the plateau. Here, the random choice of the initial triangle of the walk at each mesh node implies on average some change of element even at a very small distance.

2.3 Quadtree and barycentric walk complexity in space

In this section, we briefly compare the quadtree and the barycentric walk in terms of space complexity, i.e., of memory occupation. Recall that we have heuristically assumed that the number of elements is $\mathcal{O}(N)$, and that the average depth of the quadtree is $\mathcal{O}(\log N)$. Both

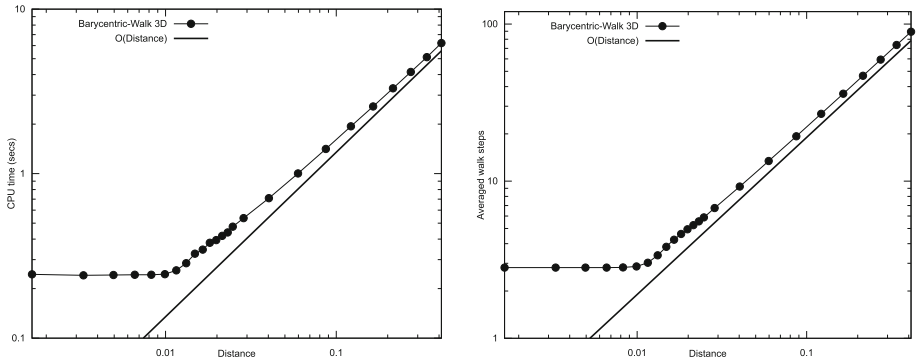


Fig. 7 Execution times (left) and averaged walk steps (right) for the barycentric walk search on the whole mesh ($d = 3$, $N \sim 4 \cdot 10^5$), versus distance between start and endpoints, compared with the line of linear increase

point location approaches use the mesh information, namely the list of point coordinates of each grid node and the connectivity, in the form of a list of triplets of vertex indices, ordered as they appear in the node list. In addition, the barycentric walk requires, for each triangle, the list of triangle neighbours to move across the elements. This list consists in triplets of triangle indices ordered as they appear in the connectivity list. On the other hand, the quadtree structure is more complicate. Starting from the root, each node of the tree must record the four coordinates (for the left/bottom and top/right vertices) of its quad, and four pointers to its children, while the leaves contain the indices of mesh nodes and triangles intersected by their quads. According to the rules discussed in Sect. 2.1, the construction stops if a quad contains at most one mesh node, regardless of the number of incident triangles, and at most q triangles if it contains no mesh nodes. This implies that the total number of vertex indices in the leaves is about N (some duplicates can be found if a mesh node stands on the side or is exactly one vertex of a quad), whereas the total number of triangle indices is much greater than the number of triangles, since a triangle typically overlaps with several quads. Note that, for point location, only the triangle indices are needed. Hence, in the following computation, we drop the list of vertex indices after the quadtree construction.

To estimate the order of memory occupation for the quadtree, we note that starting from the root, each successive level has four times the number of quads of the previous. The total memory occupation is proportional to the total number of quads, i.e.,

$$\sum_{k=0}^{\log N} 4^k = \mathcal{O}(N),$$

as it can be easily seen, for example, via comparison with an integral. On the other hand, the barycentric walk requires to store the list of neighbouring elements for each triangle, which results again in a linear memory occupation.

The following numerical test validates the expected $\mathcal{O}(N)$ space complexity in terms of storage for the corresponding data structures.

Storage for quadtree and barycentric walk: numerical example. We report in Fig. 8, for meshes of size $10^5 \lesssim N \lesssim 2.5 \cdot 10^6$, the storage in Mbytes corresponding to the two data structures, including for both the load due to the mesh (vertices plus triangles). The experiment confirms the $\mathcal{O}(N)$ space complexity for both approaches. Nevertheless, we found that the number of tree nodes is about $2N$, while the number of triangle indices in the leaves is about

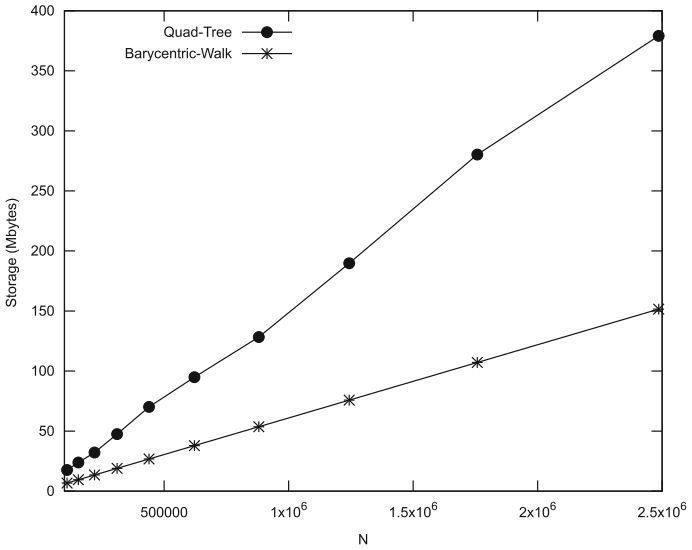


Fig. 8 Memory storage (Mbytes) required for quadtree (black circles) and barycentric walk (crosses), for $10^5 \lesssim N \lesssim 2.5 \cdot 10^6$ (color figure online)

$9N$, and this results in a gain factor about 2.5 for the barycentric walk. For completeness, we remark the code has run on a 64bit architecture, in which the storage for integers, doubles and pointers amounts, respectively, to 4, 8 and 8 bytes each.

3 Efficient initializations for the barycentric walk search

In this section, we present the main contributions of the paper. In particular, we propose two new strategies for the initialization of the barycentric walk search, adapted to the case of characteristics tracking for SL schemes. We show that these strategies are a key ingredient to obtain an $\mathcal{O}(1)$ complexity for the location of a point on the grid, as compared to the $\mathcal{O}(\sqrt{N})$ complexity of the standard barycentric walk and the $\mathcal{O}(\log N)$ complexity of the quadtree search. This makes it convenient to replace the quadtree algorithm, which is memory-consuming and complex to code, with an easier and smarter procedure, which only requires some additional memory to record the initial elements for the barycentric walks associated with the mesh nodes. Note that, in view of the results shown in Fig. 7, it will be reasonable to extend the conclusions of this analysis to higher dimensions, in particular $d = 3$. Finally, we provide some implementation details, and also pseudo-codes for both the proposed point location algorithm and the basic SL scheme for advection equations.

To fix ideas, we consider a regular Delaunay triangulation with N nodes and space scale $\Delta x \propto 1/\sqrt{N}$, and use Euler tracking of characteristics (5), that we recall here for the reader's convenience:

$$X^{\Delta}(x_i, t_{n+1}; t_n) = x_i - \Delta t \; f(x_i, t_{n+1}).$$

In this setting, the complexity (9) for the barycentric walk algorithm reads

$$\mathcal{O}\left(C_1^B + C_2^B \frac{\|X^\Delta(x_i, t_{n+1}; t_n) - \overline{X}_i\|}{\Delta x}\right),$$

where X^Δ is the query point and $\overline{X}_i$ is a starting point (or the corresponding starting element) related to the node x_i from which the characteristic originates. Now, we introduce different choices for $\overline{X}_i$, suitable to obtain a point location with a complexity independent of the grid size.

Strategy (BWa): follow the characteristic. We set $\overline{X}_i = x_i$, i.e., we start from an element neighbouring x_i. This is maybe the simplest and most natural choice, corresponding to track all the characteristic starting from a fixed element. As discussed in the introductory overview, this technique has already been applied to SL schemes, for example, in Restelli et al. (2006), Boscheri et al. (2013), Boscheri (2020), coupled with a substepping along the characteristic. With this choice,

$$\|X^\Delta(x_i, t_{n+1}; t_n) - x_i\| = \Delta t \|f(x_i, t_{n+1})\|,$$

so that the number of steps is of the order of the local Courant number, and a single element search has, therefore, a complexity of

$$\mathcal{O}\left(C_1^B + C_2^B \frac{\|f(x_i, t_{n+1})\| \Delta t}{\Delta x}\right),$$

that is, asymptotically constant under linear $\Delta t/\Delta x$ relationship. Note that, if one works at large Courant numbers in order to increase efficiency of the scheme, the element search becomes in turn more complex. Moreover, complexity is no longer asymptotically constant under nonlinear refinements in which $\Delta x = o(\Delta t)$.

Strategy (BWb): look at the previous time step. We set $\overline{X}_i = X^\Delta(x_i, t_n; t_{n-1})$, i.e., we start from the element containing the foot of the characteristic at the previous time step (it has already been computed by the scheme (4)). In this case,

$$\begin{aligned}\left\|X^\Delta(x_i, t_{n+1}; t_n) - X^\Delta(x_i, t_n; t_{n-1})\right\| &= \Delta t \|f(x_i, t_{n+1}) - f(x_i, t_n)\| \\ &\leq L_t \Delta t^2,\end{aligned}$$

and the location of the foot of characteristics has therefore a complexity of

$$\mathcal{O}\left(C_1^B + C_2^B L_t \frac{\Delta t^2}{\Delta x}\right),$$

in which, since L_t is a global Lipschitz constant, we are bounding the computational cost from above. In this case, the complexity is asymptotically constant provided $\Delta t = \mathcal{O}\left(\Delta x^{1/2}\right)$. On the other hand, under a linear refinement, it tends to coincide with the complexity of a single computation of the barycentric coordinates: in other terms, the event of a change of element becomes more and more unlikely. In particular, regions of the domain in which the advecting vector field has slow changes (or tends towards a regime state) require only minor adjustments from one time step to the next. In the limit case of an advection term constant in time, no change of element is necessary.

Note that this initialization is constructed independently for each node, and hence the location of the points $X^\Delta(x_i, t_{n+1}; t_n)$ can be performed in parallel w.r.t. i. Since it requires the same sequence of operations for each node (except for a possibly different length of the walk), the resulting algorithm might be particularly convenient on a SIMD architecture.

Strategy (BWc): look at the neighbour. We set $\overline{X}_i = X^\Delta(x_k, t_{n+1}; t_n)$, i.e., the walk algorithm is initialized with the element containing the foot of the characteristic at the same time step, but at a node x_k adjacent to x_i. With this choice,

$$\begin{aligned}\|X^\Delta(x_i, t_{n+1}; t_n) - X^\Delta(x_i, t_{n+1}; t_n)\| &= \|x_i - x_k - \Delta t(f(x_i, t_{n+1}) - f(x_k, t_{n+1}))\| \\ &\leq (1 + L_x \Delta t)\Delta x.\end{aligned}$$

Taking into account the fixed complexity terms, the point location has therefore a cost of

$$\mathcal{O}\left((C_1^B + C_2^B) + C_2^B L_x \Delta t\right).$$

Again, we obtain an asymptotically constant complexity, but it appears to have a less critical dependence (if any dependence at all) on the $\Delta t/\Delta x$ relationship, and in particular to be applicable when $\Delta x = o(\Delta t)$. However, opposite to what happens in the previous case (BWb), even with stationary advection terms, we expect that a change of element is needed in general, and this causes an increase of the constant term.

This strategy clearly requires that the nodes are put in a sequence where all nodes but the first one have a neighbour for which the final element has already been computed. In practice, this may be accomplished by constructing a spanning tree of the grid, once and for all after the grid construction. Note that, with respect to the previous strategy, this technique is more complicate to set in parallel form. Parallelization should be performed on successive levels of the spanning tree, by computing in parallel all the nodes having parents at the previous level, and its efficiency is clearly related to the depth of the spanning tree, and ultimately to the mesh size.

Note also that, in SL schemes (see, e.g., the discussion of this point in Ferretti and Mehrenberger 2020), it is usually required for stability reasons that characteristics passing through neighbouring nodes do not cross. In practice, $X^\Delta(x_i, t_{n+1}; t_n)$ and $X^\Delta(x_k, t_{n+1}; t_n)$ must always have a positive distance, so that, using (5) and the reverse triangular inequality,

$$\begin{aligned}\|X^\Delta(x_i, t_{n+1}; t_n) - X^\Delta(x_k, t_{n+1}; t_n)\| &\geq \|x_i - x_k\| - \Delta t \|f(x_i, t_{n+1}) - f(x_k, t_{n+1})\| \\ &\geq (1 - \Delta t\ L_x)\|x_i - x_k\| > 0.\end{aligned}$$

This leads to the well-known condition

$$L_x \Delta t < 1,$$

and, as a consequence,

$$\|X^\Delta(x_i, t_{n+1}; t_n) - X^\Delta(x_k, t_{n+1}; t_n)\| \leq 2\Delta x, \tag{10}$$

which also implies a uniform bound w.r.t. Δx on the complexity.

Remark 3 For an actual implementation, all the three initialization strategies (BWa), (BWb) and (BWc) presented above require some additional data structures with respect to the standard barycentric walk. More precisely, we need a list of N integers for storing the indices of all the triangles, one for each vertex of the mesh, from which to start the barycentric walks. Furthermore, strategy (BWc) also requires the spanning tree for ordering the grid nodes. This results in another list of N integers, storing for each node the index of its parent in the spanning tree. In the left plot of Fig. 9 we report, for meshes of size $10^5 \lesssim N \lesssim 2.5 \cdot 10^6$, the storage in Mbytes corresponding to the different strategies (clearly the same for strategies (BWa) and (BWb)), including the mesh data and the neighbour list for the standard barycentric walk. Finally, the right plot reports the storage improvement, showing that the initialization strategies require less than half the memory required by the quadtree location.

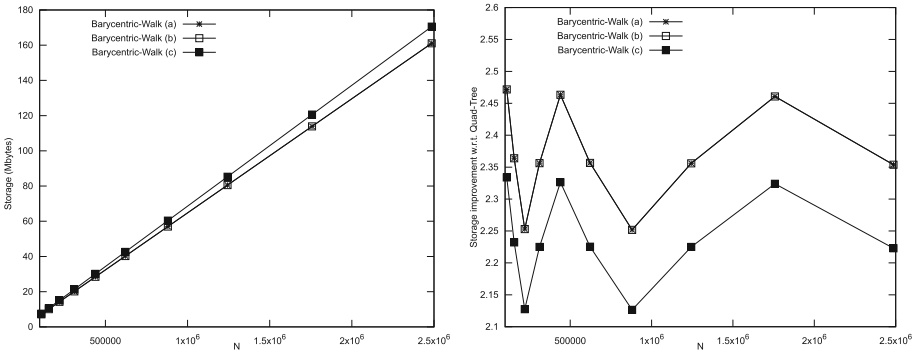


Fig. 9 Memory storage (Mbytes) required by the initialization strategies for the barycentric walk (left) and improvement w.r.t. storage for the quadtree, for $10^5 \lesssim N \lesssim 2.5 \cdot 10^6$. Strategy (BWa): crosses, strategy (BWb): white squares, strategy (BWc): black squares (color figure online)

We now provide some implementation details and a pseudo-code for our barycentric walk algorithm, according to the initialization strategy chosen. We consider, as follows, a suitable data structure containing all the relevant information for the triangulation. We recall that $\mathcal{V} = \{x_i\}$, for $i = 1, \ldots, N$, is the list of point coordinates of each node. We denote by $\mathcal{T}$ the list of triangles, namely the list of vertex indices $\mathcal{T}_j = (i_1, i_2, i_3) \in \{1, \ldots, N\}^3$ defining the triangle with vertices $x_{i_1}, x_{i_2}, x_{i_3}$, for $j = 1, \ldots, N_t$, where N_t is the total number of triangles. Moverover, we denote by $\mathcal{N}$ the list of triangle neighbours, namely the list of triangle indices $\mathcal{N}_j = (j_1, j_2, j_3) \in \{1, \ldots, N_t\}^3$ corresponding to the three neighbours $\mathcal{T}_{j_1}, \mathcal{T}_{j_2}, \mathcal{T}_{j_3}$ of $\mathcal{T}_j$, for $j = 1, \ldots, N_t$. We adopt the standard convention for which the index j_k corresponds to the neighbouring triangle of $\mathcal{T}_j$ sharing the edge opposite to the vertex with index i_k, for $k = 1, 2, 3$, and we set $j_k = 0$ if $\mathcal{T}_j$ is a boundary element and the corresponding k-th neighbour is missing. Furthermore, we denote by $\mathcal{T}^0$ a list of triangle indices associated to the nodes, so that $\mathcal{T}^0_i \in \{1, \ldots, N_t\}$, for $i = 1, \ldots, N$, identifies the initial triangle for the barycentric walk which tracks the characteristic originating from the node x_i. We always initialize $\mathcal{T}^0$ assigning to each node x_i a random triangle among those having x_i as a vertex. We remark that to implement the initialization strategy (BWc), we need a root node for the spanning tree of the grid. For simplicity, we assume that $\mathcal{V}$ is already ordered according to the spanning tree, so that x_1 is the root node, followed by its first neighbouring nodes, and so on recursively. Then, we denote by $\mathcal{P}$ the list of indices of parent nodes, so that $\mathcal{P}_i \in \{1, \ldots, N\}$, for $i = 1, \ldots, N$, identifies the parent node of x_i. In particular, the root node is the only one satisfying $\mathcal{P}_1 = 1$. Finally, given the dynamics f, we can build the list $\mathcal{Q} = \{q_i\}_{i=1,\ldots,N}$ of query points for the barycentric walks, tracking the characteristics with a suitable solver for ordinary differential equations (e.g., $q_i = x_i - \Delta t\, f(x_i, t_{n+1})$ for the Euler scheme at time t_{n+1}). The procedure is implemented as a pseudo-code in Algorithm 1, according to the initialization strategy chosen ((BWa), (BWb) or (BWc), corresponding to $\sigma = a, b, c$). The function `PointLocationBW` returns the indices of the final elements for the corresponding walks, the related barycentric coordinates of the query points, and the type of the final elements ('internal' if the element encloses the query point, 'boundary' if the walk stopped at the boundary of the domain).

We remark that the choice of the initialization strategy affects only a few lines of code. More precisely, for strategies (BWa) and (BWb), we just read in line 4 the initial element of node x_i from $\mathcal{T}^0_i$, while for strategy (BWc) we refer to the element $\mathcal{T}^0_{\mathcal{P}_i}$, according to

Algorithm 1 Function `PointLocationBW`

1: Given $\mathcal{V}, \mathcal{T}, \mathcal{N}, \mathcal{T}^0, \mathcal{P}, \mathcal{Q}$ and an itialization strategy $\sigma \in \{a, b, c\}$
2: **function** $(\mathcal{T}^f, \mathcal{B}, \mathcal{K}) =$ `PointLocationBW`$[\sigma](\mathcal{Q})$
3: **for** $i = 1 : N$ **do**
4: Set $j \leftarrow \mathcal{T}_i^0$ if $\sigma = a, b$ or Set $j \leftarrow \mathcal{T}_{\mathcal{P}_i}^0$ if $\sigma = c$
5: Set BW $\leftarrow$ true
6: **while** BW **do**
7: Set $(i_1, i_2, i_3) \leftarrow \mathcal{T}_j$
8: Compute $(\theta_1, \theta_2, \theta_3)$ for q_i^n w.r.t. $x_{i_1}, x_{i_2}, x_{i_3}$ using (7)
9: Set $\theta^* \leftarrow \min_{k=1,2,3} \theta_k$ and $k^* \leftarrow \operatorname{argmin}_{k=1,2,3} \theta_k$
10: **if** $\theta^* \geq 0$ **then**
11: Set $\mathcal{K}_i \leftarrow$ 'internal'
12: Set BW $\leftarrow$ false
13: **else**
14: Set $(j_1, j_2, j_3) \leftarrow \mathcal{N}_j$
15: **if** $j_{k^*} = 0$ **then**
16: Set $\mathcal{K}_i \leftarrow$ 'boundary'
17: Set BW $\leftarrow$ false
18: **else**
19: Set $j \leftarrow j_{k^*}$
20: **end if**
21: **end if**
22: **end while**
23: Set $\mathcal{T}_i^f \leftarrow j$
24: Set $\mathcal{B}_i \leftarrow (\theta_1, \theta_2, \theta_3)$
25: Set $\mathcal{T}_i^0 \leftarrow j$ if $\sigma = c$
26: **end for**
27: Set $\mathcal{T}^0 \leftarrow \mathcal{T}^f$ if $\sigma = b$
28: **return** $\mathcal{T}^f, \mathcal{B}$ and $\mathcal{K}$
29: **end function**

the spanning-tree embedded in the list $\mathcal{P}$ of parent nodes. Moreover, for strategy (BWb), we update all the list $\mathcal{T}^0$ at the end of the process in line 27, while for strategy (BWc) the update is performed point by point at line 25, to provide the children of the nodes in the spanning tree with the initial elements just updated by their parents. We also remark that, whenever a barycentric walk reaches the boundary and the neighbouring element is missing (see line 15), we stop the walk, marking the element as 'boundary', and we do not update the current element as for the 'internal' ones (see line 19). This ensures that the walks can restart correctly from elements on the boundary, if the corresponding query points change in time and re-enter the domain.

We finally report, in Algorithm 2, the basic SL pseudo-code for the advection equation (1), using $\mathbb{P}_1$ interpolation for the solution and the Euler scheme for tracking characteristics. This will be used later to show which fraction of the total computational time is due to the point location phase.

Remark 4 It could be observed that once assumed that Ω is defined by the inequality $g(x) < 0$, the internal/boundary test could be performed by checking the sign of g at the point $X^{\Delta}(x_i, t_{n+1}; t_n)$, thus avoiding to start the walk-in case of external points. In practice, however, such an explicit representation of Ω is seldom available, and the test is better accomplished during the walk algorithm, as we have described above. In lack of an explicit form of g, the exact location of the intersection of $X^{\Delta}(x_i, t_{n+1}; s)$ with the boundary requires some additional technicalities, which will be omitted here.

Algorithm 2 Basic Semi-Lagrangian advection scheme

Given $\Omega \subset \mathbb{R}^2$, $u_0 : \Omega \to \mathbb{R}$, $f : \Omega \times \mathbb{R}^+ \to \mathbb{R}^2$, $\gamma : \partial\Omega \times \mathbb{R}^+ \to \mathbb{R}$, Δx, Δt, $T > 0$
Choose an initialization strategy $\sigma \in \{a, b, c\}$
Build a triangulation $(\mathcal{V}, \mathcal{T}, \mathcal{N}, \mathcal{T}^0, \mathcal{P}) \leftarrow (\Omega, \Delta x)$
Set $V_i^0 \leftarrow u_0(x_i)$ for $i = 1, \dots, N$
Set $n \leftarrow 0$
while $n\Delta t \leq T$ **do**
  Set $q_i \leftarrow x_i - \Delta t\, f(x_i, t_{n+1})$ for $i = 1, \dots, N$ and $\mathcal{Q} \leftarrow \{q_i\}_{i=1,\dots,N}$
  Compute $(\mathcal{T}^f, \mathcal{B}, \mathcal{K}) \leftarrow$ `PointLocationBW`$[\sigma](\mathcal{Q})$ using Algorithm 1 according to σ
  for $i = 1 : N$ **do**
    if $\mathcal{K}_i =$ 'internal' **then**
      Set $(i_1, i_2, i_3) \leftarrow \mathcal{T}_i^f$
      Set $(\theta_1, \theta_2, \theta_3) \leftarrow \mathcal{B}_i$
      Set $V_i^{n+1} \leftarrow \theta_1 V_{i_1}^n + \theta_2 V_{i_2}^n + \theta_3 V_{i_3}^n$
    else if $\mathcal{K}_i =$ 'boundary' **then**
      Compute $\bar{s}_i^{n+1}$ by solving (6) for s
      Set $V_i^{n+1} \leftarrow \gamma\left(x_i - (t_{n+1} - \bar{s}_i^{n+1}) f(x_i, t_{n+1}), \bar{s}_i^{n+1}\right)$
    end if
  end for
  Set $n \leftarrow n + 1$
end while

4 Numerical examples

In this section, we present several numerical tests, showing the performance of the search strategies proposed, as compared with the standard quadtree search and with a direct search on a structured grid. Moreover, we provide a comparison with the built-in Matlab function `pointLocation`. Finally, we present the performance of the basic SL scheme, equipped with the different point location algorithms.

All the codes have been implemented from scratch in C++ language on the basis of Algorithms 1–2, compiled with GCC compiler 7.5.0, and run (in serial for the present work) on a PC Desktop equipped with an Intel i9-9900K CPU with 16 cores 3.60Ghz, 32Gb RAM, under the OS Ubuntu 18.04.3 LTS. We have also built a simple wrapper to easily employ the library `Triangle` for the generation of quality Delaunay meshes (TRI 2021). In particular, the library accepts an input constraint $A_{\max}$ for the maximal area of each triangle in the mesh. Then, we set the space scale $\Delta x = \sqrt{2A_{\max}}$, so that triangle areas are proportional to $\frac{1}{2}\Delta x^2$.

In all the tests, we consider the following advecting vector field:

$$f(x, t) = \begin{pmatrix} \cos\left(C_0\|x\| + C_1 t\right) \\ \sin\left(C_0\|x\| + C_1 t\right) \end{pmatrix}, \tag{11}$$

with $x \in \Omega$, $t \in [0, 1]$ and $C_0, C_1 > 0$, namely a rotating vector field with frequencies C_0 and C_1, respectively in space and time (see Fig. 10).

To compare the execution times with the structured case, we take the square domain $\Omega = [-1/2, 1/2]^2$, and, focusing on the point location for internal query points, we exclude from the computation all the grid nodes for which the corresponding characteristic, tracked by the Euler scheme (5), falls out the domain. Moreover, in the construction of the spanning tree for the initialization strategy (BWc), we take the root node as the closest to the center of the domain.

Note that, by the definition (11), we have $\|f\| = 1$ everywhere in Ω, while the Lipschitz constants of f are given by $L_x = C_0$ and $L_t = C_1$. This allows to better analyze the

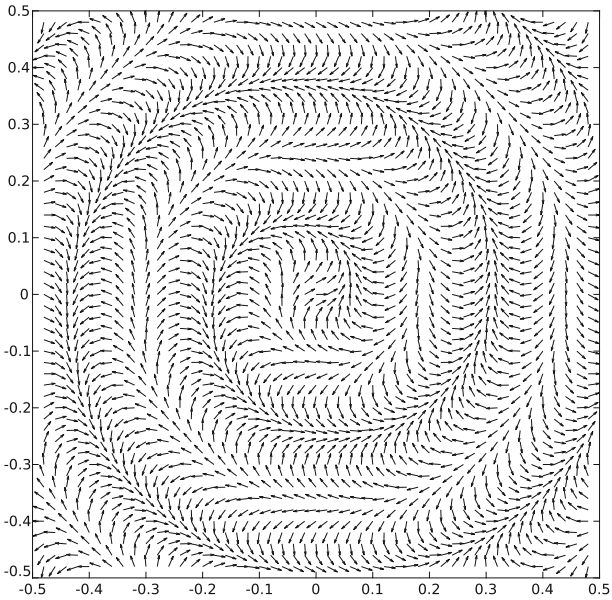


Fig. 10 Test dynamics with $C_0 = 8\pi$, $C_1 = 2\pi$ at time $t = 0$, sampled on a uniform grid with $\Delta t/\Delta x = 1$

complexity of the search strategies in terms of the Courant number $\|f\|\Delta t/\Delta x$, which is indeed the same on the whole domain: setting $\Delta t = \alpha \Delta x$, with $\alpha \geq 0$, the Courant number is simply given by α. Finally, we average in time, dividing by the number of time steps $\lceil 1/\Delta t \rceil$, both the computational times and the averaged walk steps.

Comparison of quadtree search versus initialization strategies for the barycentric walk. In this test, we compare the performance of the initialization strategies (BWa)–(BWc) for the barycentric walk with that of the quadtree. We fix the Lipschitz constants of the dynamics to $L_x = L_t = 2\pi$, the Courant number to $\alpha = 5$, and we consider finer and finer triangular meshes with $10^5 \lesssim N \lesssim 7 \cdot 10^6$. The results are reported in Fig. 11: in the left plot we show the average CPU time for a single point location (CPU time of the point location phase over the whole grid, divided by N), while in the right one we show the improvement factor with respect to the quadtree.

As N increases, all the proposed initialization strategies for the barycentric walk exhibit, as expected, an $\mathcal{O}(1)$ complexity, whereas CPU times for the quadtree algorithm grow due to its $\mathcal{O}(\log N)$ complexity. Nevertheless, the quadtree still performs better than the walking strategy (BWa), due to the relatively large Courant number, although this advantage tends to decrease at the increase of N. On the other hand, in this setting, the improvement provided by the initialization strategies (BWb) and (BWc) with respect to the quadtree is apparent, with an improvement factor ranging between 2 and 5, and increasing with N.

Dependence on the Courant number. The efficiency of the initialization strategies proposed for the barycentric walk depends on the ratio $\alpha = \Delta t/\Delta x$ (which coincides, in our examples, with the Courant number), as it can be seen by rewriting the complexity estimates of the previous section in terms of α as:

$$\begin{aligned} &(BWa) : \mathcal{O}(C_1^B + C_2^B \|f\|_\infty \alpha)\,, \\ &(BWb) : \mathcal{O}(C_1^B + C_2^B L_t \alpha^2 \Delta x)\,, \\ &(BWc) : \mathcal{O}(C_1^B + C_2^B + C_2^B L_x \alpha \Delta x)\,. \end{aligned} \tag{12}$$

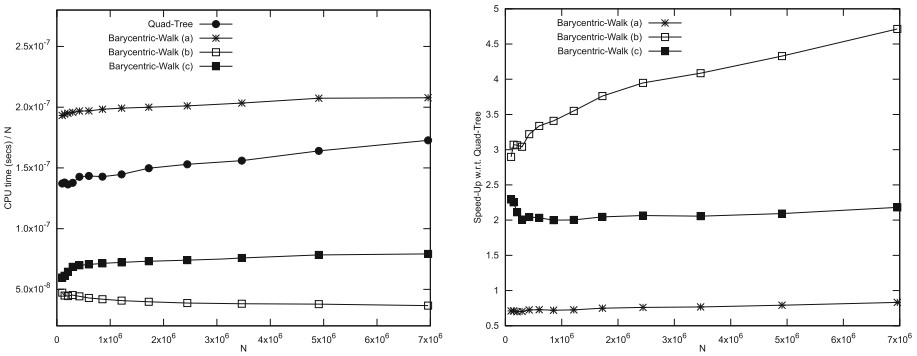


Fig. 11 Averaged search time of the quadtree and barycentric walk (left) and improvement factor versus the quadtree search (right), for $10^5 \lesssim N \lesssim 7 \cdot 10^6$. Strategy (BWa): crosses, strategy (BWb): white squares, strategy (BWc): black squares, quadtree: black circles (color figure online)

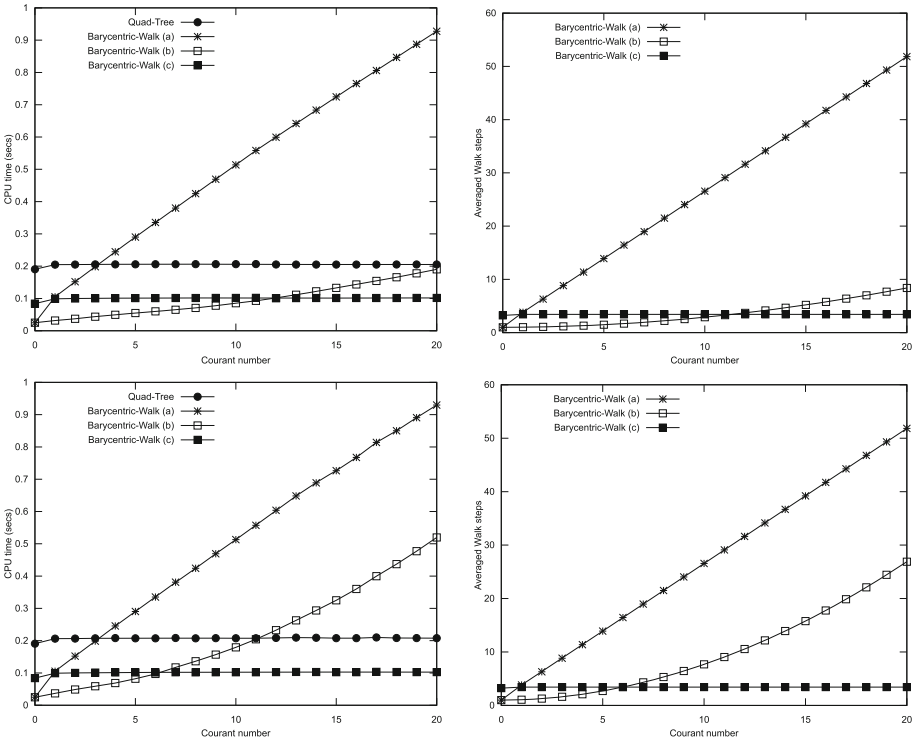


Fig. 12 Search times versus Courant number (left) and averaged barycentric walk steps versus Courant number (right), for $N = 1.5 \cdot 10^6$, $L_x = 2\pi$, and with $L_t = 2\pi$ (upper plots), $L_t = 8\pi$ (lower plots). Strategy (BWa): crosses, strategy (BWb): white squares, strategy (BWc): black squares, quadtree: black circles (color figure online)

In this test, we compare the three initialization strategies and the quadtree search at the increase of the Courant number and of the Lipschitz constants of the advecting dynamics. To this end, we consider a grid with a fixed number of nodes $N = 1.5 \cdot 10^5$ (corresponding to $A_{\max} = 5 \cdot 10^{-5}$ and $\Delta x = 10^{-2}$), and choose a variable Courant number in the range

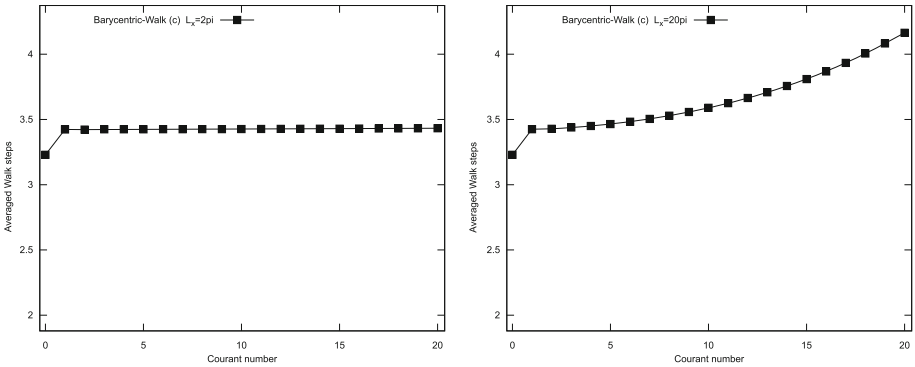


Fig. 13 Averaged barycentric walk steps versus Courant number for strategy (BWc), with $N = 1.5 \cdot 10^6$ and $L_x = 2\pi$ (left), $L_x = 20\pi$ (right)

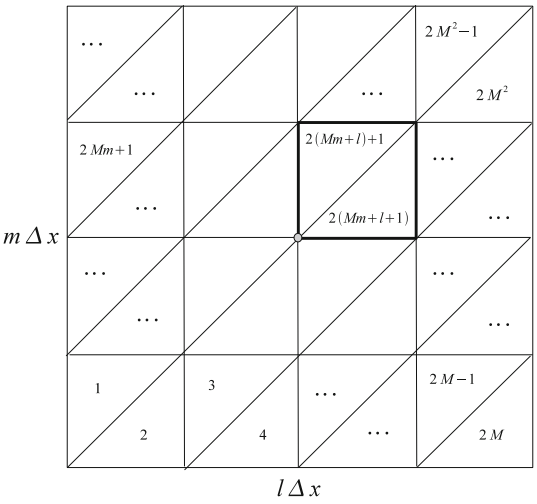
$0 \leq \alpha \leq 20$. In the upper plots of Fig. 12, we report the results for the case $L_x = L_t = 2\pi$, while in the lower plot we report the same data for $L_x = 2\pi$ and $L_t = 8\pi$.

As expected, the complexity of the quadtree search does not depend at all on the Courant number. Moreover, for $\alpha \to 0$, we recover the constant terms in the estimates (12). In particular, strategies (BWa) and (BWb) have the same constant C_1^B, while strategy (BWc) shows an additional cost due to the constant C_2^B. On the other hand, as α increases, we clearly observe the linear increase of complexity for strategy (BWa), which eventually performs worse than the quadtree search. Strategies (BWb) and (BWc) are the most effective, due to the terms Δx in (12). Moreover, we recognize a quadratic behavior in α for strategy (BWb), with a loss of performance at the increase of L_t, while the linear behavior for strategy (BWc), in the chosen range for α, is somewhat hidden by both its slope Δx and the constant term. This confirms the uniform bound in (10), since the crossing condition $L_x \Delta t \geq 1$ for the characteristics reads, in the present case, as $2\pi\alpha\Delta x \geq 1$, namely $\alpha \gtrsim 16$. The effect of the Lipschitz constant L_x on the number of steps for the strategy (BWc) is analyzed more in detail in Fig. 13. In the left plot, we use $L_x = 2\pi$, and obtain an averaged number of walk steps of about 3.5, which is in agreement with (10) in view of Remark 2. The increase in the number of steps becomes apparent when choosing a larger Lipschitz constant $L_x = 20\pi$, which forces the crossing of characteristics around $\alpha \gtrsim 1.6$, so that the uniform bound on the walking steps fails, as shown in the right plot of Fig. 13. We point out, however, that this makes the scheme work in unstable (and unphysical) conditions.

Comparision with direct location on a structured grid. In this test, we compare the barycentric walk search (initialized with the strategy (BWb)) with a direct search on a mesh which is still triangular, but structured. More precisely, we consider in $[-1/2, 1/2]^2$ the Courant triangulation shown in Fig. 14, with a uniform number of nodes in each dimension and a natural labelling of the corresponding triangles.

In this setting, a given target point $X^\Delta(x_i, t_{n+1}; t_n) = (\xi, \eta)$ can be directly located, with constant complexity, by the couple

$$l = \left\lfloor \frac{\xi + 1/2}{\Delta x} \right\rfloor, \quad m = \left\lfloor \frac{\eta + 1/2}{\Delta x} \right\rfloor$$
$$X^\Delta(x_i, t_{n+1}; t_n) \in \begin{cases} T_{2(Mm+l)+1} & \text{if } \xi + 1/2 - l\Delta x < \eta + 1/2 - m\Delta x \\ T_{2(Mm+l+1)} & \text{otherwise,} \end{cases} \tag{13}$$



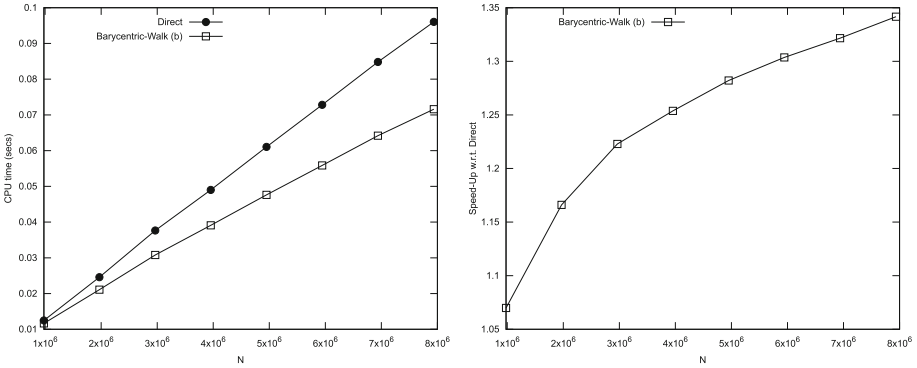


Fig. 14 Courant triangular grid for structured/unstructured search comparison

Fig. 15 Search time (left) and improvement factor (right) of the barycentric walk with initialization strategy (BWb) versus the direct search on a structured grid, for $10^6 \lesssim N \lesssim 8 \cdot 10^6$. Direct access: black circles, strategy (BWb): white squares (color figure online)

with M denoting the number of elements for each side of the square. We consider the same advecting dynamics of the previous tests, with $L_x = L_t = 2\pi$ and Courant number $\alpha = 5$, and we choose the initialization strategy (BWb) for the barycentric walk, which achieves the best performance in this case (see again Fig. 12). For a fair comparison, we include in the direct search both the location of the triangle and the computation of the corresponding barycentric coordinates for the target point (the minimal requirement for computing the interpolation). The results are reported in Fig. 15.

In the left plot we show the CPU times for the search on finer and finer meshes with a total number of nodes $10^6 \lesssim N \lesssim 8 \cdot 10^6$. As expected, both algorithms show a linear behavior in N, but, surprisingly, the initialization strategy (BWb) for the barycentric walk outperforms the direct search, with a factor between 1.05 and 1.35 (see the right plot in Fig. 15). We found out that the most expensive task of the direct search consists in the computation of the two *lower integer parts* in (13), and it is deeply related to the assembly code generated by the GCC compiler. Among the various possible implementations of this operation, we have used a division followed by casting to integer, while the use of the built-in floor function would

result in worse performances. A more careful test should be performed by running the code against different compilers and architectures, but this goes beyond the scope of the present paper.

We can conclude anyway that, even in less favourable conditions, the initialization strategy (BWb) (and the strategy (BWc), which performs slightly worse in these conditions), when implemented on unstructured meshes, has comparable performances with respect to the fully structured case, for interpolations of a finite element type.

Comparison with Matlab `pointLocation`**.** In this test, we compare the barycentric walk search, initialized with the strategy (BWb), with the built-in `pointLocation` Matlab function. It is known that the Matlab environment provides several facilities for practitioners, including toolboxes for generating unstructured meshes and for solving PDEs. Unfortunately, many Matlab functions (as the general-purpose `pointLocation`) are closed-source, pre-compiled, and cannot be modified for specific tasks. Here, we provide some hints to implement the barycentric walk algorithm with a few lines of code in Matlab, then evaluate its performances versus the `pointLocation` function.

To this end, we employ the Matlab command `mex` to build a MEX function from our C++ implementation, namely a binary file that can be called, as any Matlab built-in function, by a Matlab script. Starting from the Matlab `triangulation` data structure, containing point coordinates of the nodes and vertex indices of the triangles in the mesh, we add three additional fields:

- a list containing the indices of the initial triangles for the barycentric walk (one index per node);
- a list containing the index triplets of neighbouring triangles (one triplet per triangle)
- a list containing the indices of the parent nodes for the initialization strategy (BWc) (one index per node).

The first list can be constructed choosing a random triangle from the output of the Matlab function `vertexAttachments`, the second list is simply the output of the Matlab function `neighbours`, while the third list can be obtained, starting from a root node, using `vertexAttachments` to find recursively the first neighbours of nodes already inserted in the list (some care must be taken to avoid duplicates). Then, we design our MEX function `pointLocationBW` with a syntax similar to `pointLocation`:

$$[\texttt{I}, \texttt{B}, \texttt{K}] = \texttt{pointLocationBW}(\texttt{TBW}, \texttt{Q}, \texttt{S}),$$

where `TBW` is the extended triangulation data structure, `Q` the list of query points, `S` the initialization strategy chosen (`'a'`,`'b'` or `'c'`), while `I` is the output list of the triangles enclosing the query points, `B` the corresponding list of barycentric coordinates, and `K` the list of element types (`'internal'` or `'boundary'`). This function is still in beta version, but available for the interested readers on reasonable request.

Now, we set the same parameters of the previous test, namely $L_x = L_t = 2\pi$, $\alpha = 5$, then we choose the initialization strategy (BWb) for the barycentric walk, and run the code on Matlab version `R2021a`. The results are reported in Fig. 16.

In the left plot, we show the CPU times for the search on finer and finer meshes with a total number of nodes $10^6 \lesssim N \lesssim 10^7$. We observe that also the Matlab `pointLocation` seems to have a linear complexity in N, and this suggests that its black-box algorithm might not be based on a quadtree structure. The function `pointLocationBW` with the initialization (BWb) improves the `pointLocation` CPU times by a factor ranging from 11 to 18, as shown in the right plot.

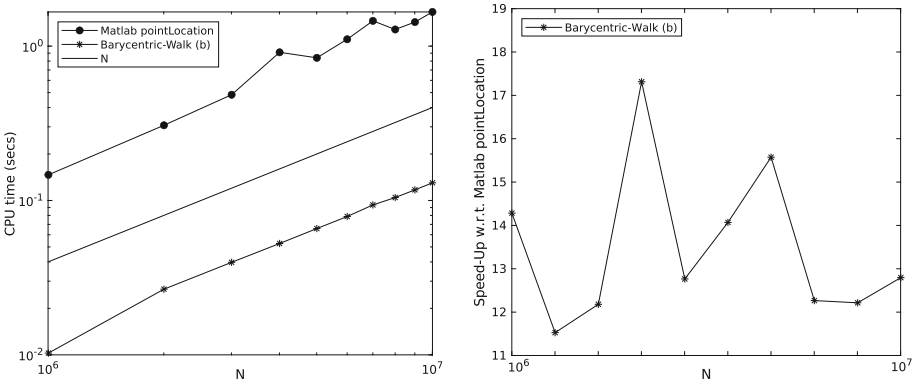


Fig. 16 Search time (left) and improvement factor (right) of the barycentric walk with initialization strategy (BWb) versus the Matlab `pointLocation`, for $10^6 \lesssim N \lesssim 10^7$. Matlab `pointLocation`: black circles, strategy (BWb): crosses (color figure online)

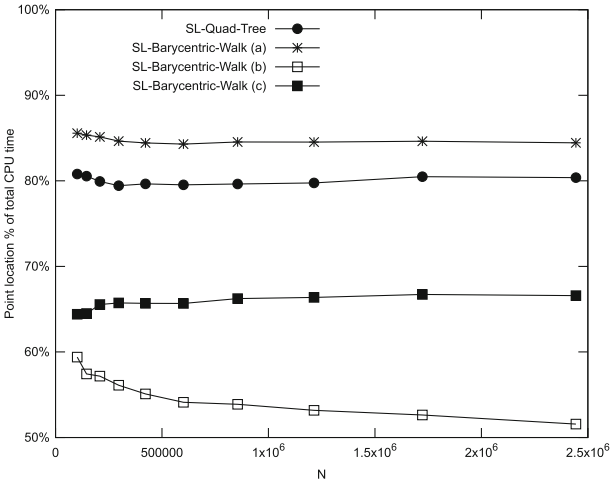


Fig. 17 Fraction of CPU time used for point location, for different algorithms in a SL scheme, versus number of grid nodes $10^5 \lesssim N \lesssim 2.5 \cdot 10^6$. Strategy (BWa): crosses, strategy (BWb): white squares, strategy (BWc): black squares, quadtree: black circles (color figure online)

Fraction of CPU time used for point location. In this last test, we combine the point location provided by the quadtree search and by the initialization strategies proposed above for the barycentric walk, with the SL scheme ($\mathbb{P}_1$ interpolation + Euler tracking of characteristics) illustrated in Algorithm 2. The aim is to measure, for the different algorithms, which percentage of the total computational load is due to the point location. To this end, we consider the same parameters for the advecting dynamics of the previous tests ($L_x = L_t = 2\pi$, $\alpha = 5$), we choose a Gaussian-like initial datum u_0, and homogeneous boundary conditions $\gamma \equiv 0$. The results for different meshes of size $10^5 \lesssim N \lesssim 2.5 \cdot 10^6$ are reported in Fig. 17. We remark that here the total CPU time for each run includes, on the whole mesh and for all the time steps, the computation of the query points $\mathcal{Q}$, the point location and the interpolation of the solution. Then we show the ratio between the point location time and the total CPU time. We observe that the quadtree point location requires about 80% of the total CPU time,

against the 85% of the initialization strategy (BWa). On the other hand, we get about 65% for strategy (BWc), while for strategy (BWb) the percentage drops between 50% and 60% (in particular it decreases as N increases before saturating, whereas the CPU time fraction is actually constant for the other algorithms). This is not surprising, since strategy (BWb) is designed to take advantage from small variations of the dynamics with respect to time. Note that in this test we have a moderate value for L_t, while the Courant number and the final time are kept fixed, so that the number of time steps increases with N. This implies that most characteristics eventually fall in the same triangle for more and more time steps.

5 Conclusions

In this paper, we have analyzed in detail the complexity issues related to characteristics location in SL-type schemes, for time-dependent PDEs with advection terms on 2-D unstructured triangular space grids. Moreover, we have proposed two new and clever initialization strategies for the barycentric walk point location, corresponding to the strategies (BWb) and (BWc) of Sect. 3, which may bring this algorithm to a higher degree of efficiency, in terms of both complexity and memory occupation, with respect to the recipes typically used so far. These strategies take advantage of the regularity of the advection term with respect to the space and time variables; on the other hand, fast space and/or time variations of the advecting field may represent a limitation to their efficiency, e.g., in turbulent flows.

We have reported implementation details and pseudo-codes for the proposed algorithms, and provided numerical experiments to validate the new techniques in a variety of scenarios.

Although the analysis has been carried out in a specific setting (triangular grids, two-dimensional problems), it is not difficult to extend the technique to more general situations, in particular to Voronoi meshes, as well as to three-dimensional problems. The choice of a walk algorithm different from the barycentric walk is also possible (see Devillers et al. 2001), especially to treat the case of less regular space grids than the ones we have used here.

Finally, a further direction of investigation would be to conceive a hybrid point location algorithm, by selecting the appropriate initialization strategy for the barycentric walk on the basis of local information provided by the dynamics at a specific point in the domain. This direction of research is still under investigation, and we plan to address it in a forthcoming work.

Acknowledgements This work has been partially supported by the PRIN 2017 project *"Innovative Numerical Methods for Evolutionary Partial Differential Equations and Applications"*, by the INdAM–GNCS project *"Approssimazione numerica di problemi di natura iperbolica ed applicazioni"* and by Roma Tre University. We thank Dr. Beatrice Beco and Dr. Lorenzo Della Cioppa for taking part in the first steps of this work.

References

Bonaventura L, Calzola E, Carlini E, Ferretti R (2020) A fully semi-Lagrangian method for the Navier–Stokes equations in primitive variables. In: van Brummelen H, Corsini A, Perotto S, Rozza G (eds) Numerical Methods for Flows. Lecture Notes in Computational Science and Engineering, vol 132. Springer, Cham

Bonaventura L, Calzola E, Carlini E, Ferretti R (2021) Second order fully semi-Lagrangian discretizations of advection–diffusion–reaction systems. J Sci Comp 88:1–29

Boscheri W (2020) A space-time semi-Lagrangian advection scheme on staggered Voronoi meshes applied to free surface flows. Comput Fluids 202

Boscheri W, Dumbser M, Righetti M (2013) A semi-implicit scheme for 3D free surface flows with high-order velocity reconstruction on unstructured Voronoi meshes. Int J Num Meth Fluids 72:607–631

Devillers O, Pion S, Teillaud M (2001) *Walking in a triangulation*. In: Proceedings of the seventeenth annual symposium on Computational geometry, 106–114

Douglas J, Russell TF (1982) Numerical methods for convection-dominated diffusion problems based on combining the method of characteristics with finite element or finite difference procedures. SIAM J Num Anal 19:871–885

Falcone M, Ferretti R (2013) Semi-Lagrangian approximation schemes for linear and Hamilton-Jacobi equations. SIAM, Philadelphia

Ferretti R, Mehrenberger M (2020) Stability of Semi-Lagrangian schemes of arbitrary odd degree under constant and variable advection speed. Math Comp 89:1783–1805

Finkel RA, Bentley JL (1974) Quad trees a data structure for retrieval on composite keys. Acta Informatica 4:1–9

Giraldo FX (1998) The Lagrange–Galerkin spectral element method on unstructured quadrilateral grids. J Comp Phys 147:114–146

Giraldo FX (2000) The Lagrange–Galerkin method for the two-dimensional shallow water equations on adaptive grids. Int J Num Meth Fluids 33:789–832

Pironneau O (1982) On the transport-diffusion algorithm and its application to the Navier–Stokes equations. Num Math 38:309–332

Restelli M, Bonaventura L, Sacco R (2006) A semi-Lagrangian discontinuous Galerkin method for scalar advection by incompressible flows. J Comp Phys 216:195–215

Russell TF, Celia MA (2002) An overview of research on Eulerian–Lagrangian localized adjoint methods (ELLAM). Adv Water Res 25:1215–1231

Triangle, A two-dimensional quality mesh generator and delaunay triangulator. https://www.cs.cmu.edu/~quake/triangle