Communications in Applied Mathematics and Computational Science

HIGH-ORDER METHODS FOR COMPUTING DISTANCES TO IMPLICITLY DEFINED SURFACES

ROBERT I. SAYE

vol. 9 no. 1 2014

msp

COMM. APP. MATH. AND COMP. SCI.
Vol. 9, No. 1, 2014
dx.doi.org/10.2140/camcos.2014.9.107

HIGH-ORDER METHODS FOR COMPUTING DISTANCES TO IMPLICITLY DEFINED SURFACES

ROBERT I. SAYE

Implicitly embedding a surface as a level set of a scalar function $\phi : \mathbb{R}^d \to \mathbb{R}$ is a powerful technique for computing and manipulating surface geometry. A variety of applications, e.g., level set methods for tracking evolving interfaces, require accurate approximations of minimum distances to or closest points on implicitly defined surfaces. In this paper, we present an efficient method for calculating high-order approximations of closest points on implicit surfaces, applicable to both structured and unstructured meshes in any number of spatial dimensions. In combination with a high-order approximation of ϕ, the algorithm uses a rapidly converging Newton's method initialised with a guess of the closest point determined by an automatically generated point cloud approximating the surface. In general, the order of accuracy of the algorithm increases with the approximation order of ϕ. We demonstrate orders of accuracy up to six for smooth problems, while nonsmooth problems reliably reduce to their expected order of accuracy. Accompanying this paper is C++ code that can be used to implement the algorithms in a variety of settings.

1. Introduction

A powerful technique for representing curves in two dimensions and surfaces in three dimensions is to define them implicitly, via a fixed level set of a continuous function $\phi : \mathbb{R}^d \to \mathbb{R}$. Implicit representations of surfaces lead to mathematical and computational advantages in a wide array of problems, such as in methods for computing diffusion and advection processes on embedded surfaces [2; 5; 15], in level set methods [20; 28; 19] for propagating interfaces coupled to computational fluid dynamics [34; 30], and in mesh generation for implicitly defined geometry [21; 24].

Given an implicitly defined surface embedded in $\mathbb{R}^d$, a common task is to calculate the closest point on the surface to a given query point $x \in \mathbb{R}^d$. For example, level set methods may require the construction of extension velocities [16; 3] or signed distance functions corresponding to a moving interface. In this context, the query

MSC2010: primary 65D99, 68U05, 35R37; secondary 65D17, 65D18, 35R01.
Keywords: implicit surfaces, closest point, level set methods, reinitialisation, redistancing, high-order.

points are the set of grid points of the computational domain, possibly in a narrow band [1]. As another example, embedding techniques for solving partial differential equations (PDEs) defined on curved surfaces work by: (i) embedding the (unknown) solution u in a higher-dimensional function u_{ext} defined on $\mathbb{R}^n$; and (ii) deriving a PDE for u_{ext} in such a way that the restriction of u_{ext} to the surface is the solution of the original surface PDE. Techniques using this idea generally use closest points on the surface to define the extension function u_{ext} and its corresponding PDE [2; 29; 5; 15].

In many of these applications, a high-order approximation of the closest point on the surface is required. This is because the resulting distance function or closest point function is used to infer the geometry of the surface, such as when calculating normal vector fields or curvature quantities like the mean curvature or Gaussian curvature. It is often the case that the level set function ϕ is known only at the grid points of a background grid/mesh. It follows that some form of interpolation must be used to define the interface throughout the domain. Even though the values of ϕ at those grid points may have an associated error (e.g., those arising from finite difference approximations or temporal errors in an evolving simulation), it remains necessary to accurately resolve the geometry of the interpolated interface.

In this paper, we develop a general purpose method for computing high-order approximations of closest points on implicitly defined surfaces. The algorithm is largely based on geometry alone and consists of two main stages. First, in an initialisation stage, a level set function ϕ defined on a grid is piecewise approximated by high-order polynomials. Assuming it is the zero level set which defines the surface/interface, these polynomials are then "sampled" by seeding points on their zero level set with sufficient density to form a scattered cloud of points approximating the interface of ϕ. In the second stage of the algorithm, given a query point x, the closest point in the cloud to x is found. This closest point forms an approximation of the actual closest point to x, and this approximation is then improved by using the original polynomial from which it was created together with Newton's method for solving the minimum-distance optimisation problem. As shown below, this combination of first finding the closest point in the cloud, and then "polishing" it with Newton's method, leads to highly accurate and robust closest point calculations. By making use of a k-d tree optimised for surfaces, the method is also inexpensive, as finding the closest point in the cloud is relatively cheap, and not many iterations of Newton's method are required for convergence. Except for the initial stage of forming high-order approximations of ϕ, the method does not rely on any computational grid and can be used to compute closest points at arbitrary locations.

The outline of the paper is as follows. In Section 2, we briefly review previous work on computing distance approximations. In Section 3, the high-order method is presented, starting with a discussion of high-order polynomial approximations, followed by the sampling procedure and Newton's method. We then discuss some implementation choices, before presenting convergence results and test cases in Section 4. In Section 5 some final remarks are given, including a short description of the C++ code that accompanies this paper. Lastly, in the Appendix, a k-d tree optimised for codimension-one surfaces is presented.

2. Motivation and previous work

Our interest in the minimum-distance/closest-point problem stems from work on level set methods for tracking the interface between two evolving regions, and on Voronoi Implicit Interface Methods [25; 26] for tracking interconnected interfaces with junctions in multiphase physics. Two common tasks in these applications are: (i) calculating extensions of some quantity F defined on the interface, e.g., extension velocities, such that $F_{\text{ext}}(x) = F(\text{cp}(x))$, where $\text{cp}(x)$ is the closest point on the interface to x, and (ii) replacing a function that implicitly defines the interface by the distance function to the interface. In the level set method literature, the latter procedure is a well-known task commonly referred to as *reinitialisation* or *redistancing* and is often performed frequently over the course of a simulation. For example, one reason for maintaining a distance function is related to the treatment of jumps in density and viscosity of a multiphase fluid, or singular forces such as surface tension on a liquid-gas interface, which may require smoothing of Heaviside and Dirac delta functions [8; 34; 30].

Methods for computing distances to implicitly defined surfaces differ in how the geometry of the surface is determined. Approaches include geometrically approximating the surface by explicitly reconstructing it, using root-finding to locate specific points on the surface, employing Eulerian grid-based techniques as in the level set method, or a combination of these methods.

Explicit approaches often use piecewise linear interpolation to find a faceted mesh representation of the interface, from which closest points can be computed by simple geometry [9; 16]. Strain [31] extended this idea to a fast quadtree-based reinitialisation algorithm which enables distances to be efficiently computed on the entire domain based on a mesh that locally adapts to the shape of the interface. Explicit representations also play an important role in the fields of computational geometry and graphics, in which different approximations are applicable; see for instance the review [14].

In the context of level set methods, a common technique for computing the distance function to the interface is to solve a PDE, which is typically done through one of two methods:

- Solve a static boundary-value problem: find ψ such that $\|\nabla\psi\| = 1$, with the requirement that the zero level set of ψ coincides with the zero level set of ϕ.

The solution to this equation is a signed distance function to the interface, i.e., $\psi(x) = \pm \min_{y,\,\phi(y)=0} \|x-y\|$. A common method for solving this special instance of an Eikonal equation is to apply the Fast Marching Method [27], which solves the general Eikonal equation $\|\nabla\psi\| = F$ where $F = F(x)$ is a general speed function, but apply it to the simpler equation with $F \equiv 1$.

- The second PDE-based method converts the static equation $\|\nabla\psi\| = 1$ into a time-dependent auxiliary PDE whose steady-state solution returns the signed distance function. Here it is generally assumed that ϕ is already close to a distance function, making this an iterative type method. First used in [34], this PDE takes the form
$$\frac{\partial\psi}{\partial\tau} + \operatorname{sign}(\phi)(\|\nabla\psi\| - 1) = 0, \quad \psi(\tau = 0) = \phi.$$
In theory, the zero level set of ψ remains fixed by the process of evolving ψ as $\tau \to \infty$; in practice, the sign(ϕ) function must be suitably smoothed for the discretised version. These methods rely on high-order ENO and WENO methods to approximate spatial derivatives and high-order Runge–Kutta methods in time. A variety of methods have been developed to a improve the accuracy of this approach, see, e.g., [23; 32; 12; 17].

High-order approaches typically compute accurate distances nearby the interface and then employ a PDE-based method to compute distances elsewhere. A notable example is in Chopp's method [10], which uses a piecewise bicubic (in 2D) or piecewise tricubic (in 3D) interpolant of the level set function that is globally C^1 smooth. For grid points adjacent to the interface, a quasi-Newton method is used to compute closest points on the zero level set of the bicubic/tricubic polynomials, which are then input to a second or third order fast marching method to build the distance function away from the interface. The resulting method is approximately third order in the distance function for smooth interfaces [10]. A similar approach can be used in gradient augmented level set methods [18], where both ϕ and its gradient are defined at each grid point, in which case a type of Hermite interpolation defines a high-order approximation of the interface. This again requires a nonlinear minimisation method to find closest points for query points adjacent to the interface — reinitialisation methods for gradient augmented level set methods include that of [4], which is based in part on Chopp's quasi-Newton method, and the method of [7], which follows the principles of the fast marching method by using Huygens' principle and Newton's method restricted to individual tetrahedrons. Other high order methods include the discontinuous spectral element method of [33], in which a root finding procedure is used to convert the zero level set of a polynomial into a height function, followed by Newton's method to find closest points on this height function. In this last work, distance functions were computed with up to sixth order accuracy. Rigorous analyses of errors in reinitialisation methods for finite-element based level set methods have also been performed [22; 13].

In comparison, the high-order method presented in this paper is essentially entirely geometric. The approach extends the ideas of Chopp's method to arbitrary-order polynomials and replaces the quasi-Newton method with a full Newton's method that converges much more rapidly when the query point is far away from a curved interface. The method does not rely on any PDE technique, and as such can be used to calculate closest points from arbitrary query points making it suitable for, for example, highly unstructured grids.

3. High-order calculation of closest points

Given a level set function ϕ defined on a computational grid, the high-order closest point algorithm essentially consists of two parts: initialisation, and closest point computation. In the initialisation, a high-order approximation of ϕ is defined on each mesh element containing the interface, followed by a "sampling" procedure which creates a cloud of points approximating the interface. Given a query point $x_q \in \mathbb{R}^d$, a closest point calculation proceeds by finding the closest point in the cloud to x_q, which is then improved by using Newton's method on the minimum-distance optimisation problem applied to the high-order approximation of the interface.

In order to present the essential ideas of the algorithm, motivated in part by common finite difference-based implementations of the level set method, we mainly consider the case that ϕ is defined on a regular Cartesian grid. The presented techniques can be adapted in a natural way to other cases, such as gradient-augmented level set methods, continuous and discontinuous finite element methods on unstructured grids, etc.; guidelines for doing so are also discussed.

3.1. ***Piecewise polynomial approximation.*** Given a level set function defined on a Cartesian grid, many possibilities exist for finding high-order approximations of ϕ between grid points. A natural choice is to find a piecewise polynomial interpolant, such as that used in Chopp [10], in which each grid cell is represented by a bicubic (in 2D) or tricubic (in 3D) polynomial in such a way that the global interpolant is C^1. However, finding high-order interpolants that are continuous with continuous derivatives can be expensive, since enforcing the continuity requirements requires many degrees of freedom that do not necessarily contribute to the approximation accuracy of the interpolant. For example, a C^1 piecewise tricubic interpolant requires 64 polynomial coefficients per grid cell, but is only third-order accurate; many of the degrees of freedom in the polynomial $\sum_{i,j,k=0}^{3} c_{ijk} x^i y^j z^k$ are lost through the enforcing of the C^1 continuity requirement. Compare this to the polynomial corresponding to a third-order accurate Taylor series in three dimensions, which has only 10 coefficients. For even higher order interpolants that are required to be continuous, possibly with continuous derivatives as well, the situation considerably worsens. Since these polynomials need to be constructed and evaluated many times, it is thus worthwhile to consider an alternative method of approximation.

In regards to the reinitialisation/closest point problem, it is not actually necessary to find a continuous interpolant. All we need is a high-order approximation of the zero level set of ϕ in each grid cell containing the interface. We can achieve this by using polynomials on each grid cell with the minimum number of degrees of freedom necessary for a certain accuracy (as determined by the canonical Taylor series expansion). Note, however, that in doing so, continuity of the zero level set between grid cells may be lost. When the interface is sufficiently smooth, the amount of discontinuity is of the same order as the truncation error of the approximating polynomial and thus will not affect the global approximation order. When the interface is not smooth, such as at the corner of a square, the amount of discontinuity is in general first order in the grid cell size; this cannot be avoided unless specific knowledge of nonsmooth features is incorporated. In either case, provided the polynomials on each grid cell are suitably defined, and the discontinuities of the interface are robustly handled by associated algorithms, the location of the interface defined by the set of polynomials carries the expected order of accuracy.

A straightforward technique for determining these polynomials is to employ a simple least squares method: given a space of polynomials and a stencil of grid points, find the best polynomial in that space which minimises the pointwise interpolation errors in an L^2 norm. Provided the stencil has enough points, this polynomial is uniquely determined. To illustrate, consider a two-dimensional case in which we seek a degree 2 polynomial of the form

$$p(x, y) = c_0 + c_1 x + c_2 y + c_3 x^2 + c_4 xy + c_5 y^2$$

for determining ϕ in the grid cell $(x_i, x_{i+1}) \times (y_i, y_{i+1})$. We would like it to interpolate the values ϕ_{ij}, $\phi_{i+1,j}$, $\phi_{i,j+1}$ and $\phi_{i+1,j+1}$. However, these 4 conditions are not enough to uniquely determine the 6 coefficients of p, so more grid points are required. We could add exactly two more grid points and this would uniquely determine p, but the resulting stencil would be asymmetric. This may not be a problem when ϕ is smooth, but generally speaking, such asymmetries can lead to stability problems in an evolving interface. Instead, we opt for a symmetric 12-point stencil, as shown in Figure 1. Enumerating the points of the stencil as $\{(x_k, y_k)\}_{k=1}^{12}$, the least squares problem amounts to finding

$$\arg\min_p \sum_{k=1}^{12} |p(x_k, y_k) - \phi(x_k, y_k)|^2,$$

and this can be solved in the usual fashion: form the 12×6 Vandermonde matrix

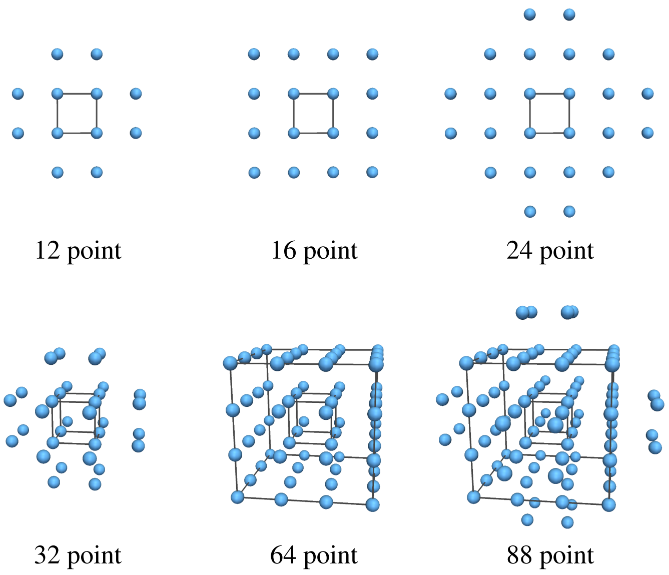

Figure 1. Stencils used to find the polynomials in Table 1. Top: two-dimensional stencils. Bottom: three-dimensional stencils.

A with i-th row $[1, x_i, y_i, x_i^2, x_i y_i, y_i^2]$ and calculate

$$[c_0, \dots, c_5]^T = (A^T A)^{-1} A^T \; [\phi(x_1, y_1), \dots, \phi(x_{12}, y_{12})]^T.$$

Provided A has full rank, the pseudoinverse $(A^T A)^{-1} A^T$ exists. In this particular example involving a 12-point stencil, this is indeed the case.

This technique easily generalises to other spaces of polynomials and in different dimensions. In each case, a stencil of grid points is designed to be as small as possible such that the corresponding Vandermonde matrix A has full rank. Table 1 and Figure 1 summarise the stencils and polynomials used in this work. Note that the 12-point stencil in the previous example can be used to find both degree 2 polynomials and degree 3 polynomials in 2D. Geometrically this states that the 12-point stencil contains enough information to determine a fourth-order accurate Taylor series approximation. Another point of interest is that the Taylor polynomial of degree 4 in two dimensions, having 15 coefficients, requires a wider stencil of extent 6, despite there being 16 degrees of information in a square 4×4 stencil.[1] This is necessary because the 4×4 stencil does not carry enough information to uniquely determine all of the higher-order terms such as x^4.

Note that by using a standard reference cell, e.g., $[0, 1]^d$, the pseudoinverses of the Vandermonde matrices can be precomputed. Thus, a polynomial with n

[1]The Vandermonde matrix corresponding to the Taylor polynomial of degree 4 in 2D and a square 4×4 stencil has rank 13.

d	Polynomial type	n_c	stencil	p
2	Bicubic, $\sum_{i,j=0}^{3} c_{ij} x^i y^j$	16	16	3
2	Taylor degree 2, $c_0 + c_1 x + c_2 y + c_3 x^2 + c_4 xy + c_5 y^2$	6	12	3
2	Taylor degree 3, $\sum_{\lvert\alpha\rvert\le 3} c_\alpha (x, y)^\alpha$	10	12	4
2	Taylor degree 4, $\sum_{\lvert\alpha\rvert\le 4} c_\alpha (x, y)^\alpha$	15	24	5
2	Taylor degree 5, $\sum_{\lvert\alpha\rvert\le 5} c_\alpha (x, y)^\alpha$	21	24	6
3	Tricubic, $\sum_{i,j,k=0}^{3} c_{ijk} x^i y^j z^k$	64	64	3
3	Taylor degree 2, $c_0 + c_1 x + c_2 y + c_3 z + \cdots + c_9 z^2$	10	32	3
3	Taylor degree 3, $\sum_{\lvert\alpha\rvert\le 3} c_\alpha (x, y, z)^\alpha$	20	32	4
3	Taylor degree 4, $\sum_{\lvert\alpha\rvert\le 4} c_\alpha (x, y, z)^\alpha$	35	88	5
3	Taylor degree 5, $\sum_{\lvert\alpha\rvert\le 5} c_\alpha (x, y, z)^\alpha$	56	88	6

Table 1. d-dimensional polynomials used in this work, indicating the form of the polynomial, number of coefficients n_c, number of points in the stencil (see Figure 1), and the expected order of accuracy p for sufficiently smooth problems.

coefficients can be determined by a stencil of m points by a single matrix-vector multiplication of size $n \times m$. We also note that while the stencils often involve more points than there are coefficients in the polynomials, ultimately it is only the polynomial and its derivatives that need to be evaluated many times.

3.2. ***Sampling the interface.*** Using the above piecewise polynomial approximation, we can find a high-order approximation of ϕ in each grid cell. For those grid cells containing the interface[2], we would like to sample the cell's polynomial by placing points on its zero level set. It will not be necessary to do this with a high degree of resolution, in fact only a few seed points per grid cell are required.[3] Thus, a very simple strategy can be adopted: subdivide each grid cell containing the interface into a 2×2 subgrid (in 2D) or $2 \times 2 \times 2$ subgrid (in 3D), and in each subcell, place a point in the centre. Then, "project" these points onto the zero level set of the polynomial p with a simple Newton-style procedure: given a point $x_0 \in \mathbb{R}^d$, we iterate

$$x_{i+1} = x_i - \frac{p(x_i)\nabla p(x_i)}{\|\nabla p(x_i)\|^2},$$

until a suitable convergence criterion is met. This iterative procedure can be viewed[4] as moving x_i to its closest point on the zero level set of the linear approximation of p at x_i, given by $p(x_i + \delta) \approx p(x_i) + \delta \cdot \nabla p(x_i)$. Generally, as in Newton's

[2]Methods to determine whether a grid cell contains the interface are discussed shortly.

[3]Generally speaking, the 2×2 subcell division described here is sufficient for most level set applications. If a grid cell contains a polynomial with very high curvature, more points may be required, depending on the application; adaptive approaches are discussed shortly.

[4]It is also the "δ_1" direction used in Chopp's method [10], as discussed later.

method, this iterative method exhibits second order convergence; in practice it is very quick and reliable. Some remarks:

- A point starts in the centre of its subcell and is projected onto the zero level set of p. As a result, it may move outside its subcell or indeed the parent cell. If the gap between the point and the parent cell is small (i.e., a fraction of Δx), we keep the point — this fits in with the strategy employed later for allowing polynomials from adjacent grid cells to slightly “overlap.” If the point is far away from its subcell, it is discarded.
- It is not necessary for the point to lie exactly on the zero level set of p, as these points only form an initial guess to a full Newton's method (see Section 3.3). In practice, a simple convergence criterion suffices, which is to stop iterating when $\|x_{i+1} - x_i\|$ is a small fraction of the subcell size, e.g., 1%; typically only one or two iterations of the above scheme are then necessary.

By doing this for each cell containing the interface, a collection of points is generated. The points are in no particular order and form a cloud of scattered points approximating the interface on the entire domain.

3.3. ***High-order closest point calculations via Newton's method.*** The output of the above sampling stage is a set of points $C = \{x_1, \dots, x_N\} \subset \mathbb{R}^d$ approximating the interface of ϕ. To each point we associate the polynomial p_i from which it was generated, coming from the high-order approximation of ϕ in each grid cell. For the general closest point problem, we are given an arbitrary query point $x_q \in \mathbb{R}^d$ and need to approximate the closest point on the zero level set of ϕ. This is accomplished in two steps:

(i) Find the closest point in C to x_q. Denote it by x_0, with associated polynomial p.

(ii) Return the closest point on the zero level set of p restricted to a small domain.

Step (i) is a well-known scattered-data closest-point query problem for which various efficient methods exist, including the use of k-d trees, quadtrees, octrees, etc. In this application, the points lie on a codimension-one surface, and this extra information can be exploited to gain greater efficiency. The Appendix presents a k-d tree optimised for surfaces that was developed as part of this work. Independent of the implementation details, however, step (i) can be considered to be a black box.

To solve step (ii), a simple Newton's method for the minimum distance optimisation problem works well. Consider the functional $f : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}$ given by

$$f(x, \lambda) = \tfrac{1}{2}\|x - x_q\|^2 + \lambda p(x),$$

whose gradient and Hessian are

$$\nabla f = \begin{pmatrix} \nabla_x f \\ \partial_\lambda f \end{pmatrix} = \begin{pmatrix} x - x_q + \lambda \nabla p(x) \\ p(x) \end{pmatrix} \quad \text{and} \quad D^2 f = \begin{pmatrix} I + \lambda D^2 p(x) & \nabla p(x) \\ \nabla p(x)^T & 0 \end{pmatrix}.$$

Minimising f amounts to minimising the squared distance from x_q to a point x, with the constraint that x be on the zero level set of p, implemented via a Lagrange multiplier. Generally speaking, this optimisation problem is well-conditioned provided that (a) the gradient of p_i does not vanish near x, and (b) the closest point is unique, i.e., x_q is not located at a shock of the distance function

$$d(x) = \min_{y,\, p(y)=0} \|x - y\|.$$

Part (a) is a natural regularity assumption in the context of level set methods, while (b) is guaranteed near smooth parts of the distance function d. Naturally, either one of these conditions might fail in practice, but with appropriate safeguards the optimisation problem can be made to be highly robust and efficient. Newton's method for minimising f is as follows: we start at the closest point in C (i.e., x_0) and initialise the Lagrange multiplier at step 0 to be[5] $\lambda_0 := (x_q - x_0) \cdot \nabla p(x_0) / \|\nabla p(x_0)\|^2$. Let $y = (x, \lambda) \in \mathbb{R}^d \times \mathbb{R}$, with initial value $y_0 := (x_0, \lambda_0)$. We thus iterate

$$y_{k+1} = y_k - \left(D^2 f(y_k)\right)^{-1} \nabla f(y_k) \tag{1}$$

until convergence to within a suitable tolerance, or else halt the iterations if x travels "too far" from the initial point x_0. Several remarks are in order:

- To evaluate the Hessian and gradient of f, the Hessian and gradient of the polynomial p are needed. These are straightforward to evaluate for any particular class of polynomial.
- The polynomials were generated from high-order approximations to ϕ on each grid cell. Thus, each polynomial is only valid in a small region surrounding its cell. We therefore only allow the iterates to travel a maximum distance (proportional to Δx) away from the initial starting point x_0. In addition to preventing iterates from travelling too far (which may occur when the interface is not smooth, e.g., at the corner of a square, as discussed later), this "bounding ball" also provides a straightforward mechanism to allow polynomials from adjacent grid cells to slightly overlap. This is mainly relevant to the case when the closest seed point (i.e., x_0) is close to the boundary of a grid cell, but the true closest point lies slightly in the neighbouring grid cell. It is important to note, however, that the order of accuracy is unaffected by using slightly overlapping polynomials (whether or not the interface is smooth).
- In (1), the Hessian of f, a small $(d+1) \times (d+1)$ square matrix, must be inverted. We can use a simple Gaussian elimination method with partial pivoting, which also indicates when the matrix is nearly singular.[6] Singularity indicates that

[5]The initial value for the Lagrange multiplier comes from the approximation that $\nabla f \approx 0$ at x_0.

[6]In this work, the criterion for determining singularity is whether any pivot is less than 10^{-12} in magnitude. This tolerance is based on double precision arithmetic and the property that the length

there are many solutions to the closest point problem for the given x_q. In theory this may certainly occur, such as when finding the closest point to the centre of a circular or spherical interface. However, in practice, noninvertibility of the Hessian almost never arises. Nevertheless, if the Hessian is detected as singular, we employ a different method. This backup mechanism follows that of Chopp's method [10] — the update is replaced by $x_{k+1} = x_k + \delta_1 + \delta_2$, where δ_1 moves x_k to the zero level set of p, and δ_2 moves x_k tangentially to the level set to enforce the orthogonality condition of the closest point. These directions are given by

$$\delta_1 = -\frac{p(x_k)\nabla p(x_k)}{\|\nabla p(x_k)\|^2} \quad \text{and} \quad \delta_2 = \left(I - \frac{\nabla p(x_k)\nabla p(x_k)^T}{\|\nabla p(x_k)\|^2}\right)(x_q - x_k).$$

In fact, Chopp's method can be viewed as a type of gradient descent on f, i.e., $y_{k+1} = y_k - \alpha\nabla f(y_k)$, with a Lagrange multiplier that is suitably reevaluated at the beginning of each iteration.

- As an additional safeguard, no update in the iterative procedure is allowed to move x_k by a large amount (i.e., 50% of the bounding ball radius). This is effectively a simple type of line search common to many optimisation methods, and is generally only relevant when x_q is extremely close to a centre of curvature of p. Once again, this safeguard is rarely invoked in practice.
- To decide when the iterations have converged, we test if $\|x_{k+1} - x_k\| < \epsilon$, where ϵ is a small threshold relating to the accuracy of the polynomial. It is not necessary to converge to machine precision when the polynomial itself is only an approximation of ϕ. We take ϵ to be Δx^p where p is the order of accuracy of the class of polynomials being used — see Table 1.

Newton's method for finding the closest point is summarised in Algorithm 1. Since the initial guess x_0 for the closest point is almost always near the actual closest point, in practice the algorithm converges very quickly; in almost all cases it converges within 2–4 iterations to a sufficient accuracy. Nonconvergence occurs when either (a) the method failed to converge within a fixed number of iterations (20, say), or (b) the iterate left the bounding ball. Although both situations are rare, (a) typically occurs when x_q is very near a shock of the distance function generated by the zero level set of p, while (b) occurs when the closest point is near a nonsmooth part of the interface, e.g., the corner of a square, where the polynomial approximations of the level set functions lead to bumps in the interface (see, e.g., Figures 4 and 5). In the rare case that Newton's method does not successfully converge, but an approximate closest point is nevertheless required as the output of a black-box type algorithm,

scales considered in the test problems are $\mathbb{O}(1)$. Experiments indicated that the overall algorithm is not particularly sensitive to this choice.

```
1:  $\lambda_0 := (x_q - x_0) \cdot \nabla p(x_0) / \|\nabla p(x_0)\|$.
2:  **for** $k = 1$ **to** maximum number of iterations **do**
3:      $g := (x_k - x_q + \lambda_k \nabla p(x_k),\ p(x_k))$.
4:      $H := \begin{pmatrix} I + \lambda_k D^2 p(x_k) & \nabla p(x_k) \\ \nabla p(x_k)^T & 0 \end{pmatrix}$.
5:      Solve for $\delta = (\delta_x, \delta_\lambda)$ such that $H\delta = g$ via Gaussian elimination with partial pivoting.
6:      **if** succeeded **then**
7:          **if** $\|\delta_x\| > \frac{1}{2} r$ **then** $\delta \leftarrow \big(\frac{1}{2} r / \|\delta_x\|\big)\delta$.
8:          $(x_{k+1}, \lambda_{k+1}) := (x_k, \lambda_k) - \delta$.
9:      **else**
10:         $\delta_1 := -\big(p(x_k)/\|\nabla p(x_k)\|^2\big)\, \nabla p(x_k)$.
11:         $\lambda_{k+1} := (x_q - x_k) \cdot \nabla p(x_k) / \|\nabla p(x_k)\|^2$.
12:         $\delta_2 := x_q - x_k - \lambda_{k+1} \nabla p(x_k)$.
13:         **if** $\|\delta_2\| > \frac{1}{10} r$ **then** $\delta_2 \leftarrow (\frac{1}{10} r / \|\delta_2\|)\delta_2$.
14:         $x_{k+1} := x_k + \delta_1 + \delta_2$.
15:     **if** $\|x_k - x_0\| > r$ **then**
16:         **return** did not converge within ball $B(x_0, r)$.
17:     **else if** $\|x_{k+1} - x_k\| < \epsilon$ **then**
18:         **return** converged with solution $x_{k+1}$.
19: **return** did not converge within maximum number of iterations.
```

Algorithm 1. Newton's method for finding the closest point on the zero level set of p given an initial guess x_0 and a bounding ball of radius r.

an approach which suffices in most practical situations is to return the last iterate inside the bounding ball. This approximation carries the same order of accuracy that one many expect when either of the cases (a) or (b) occur, as demonstrated in our convergence tests.

3.4. ***General algorithm.*** Combining the above steps, we arrive at the following general approach for computing high-order approximations of closest points on implicit surfaces:

- *Initialisation.*

(1) For each grid cell/element detected to contain the interface, define or construct a high-order approximation of ϕ on that grid cell/element. For example, when ϕ is defined on a Cartesian grid we can use the least-squares determined polynomials outlined in Section 3.1. Other possibilities may naturally arise given the specific application, for example in a gradient augmented level set method, one can use the associated Hermite interpolants; in a discontinuous or continuous finite element method, ϕ is already naturally defined as a polynomial on the elements of an unstructured mesh.

(2) For each of these constructed polynomials, sample the zero level set of the polynomial on its domain. In the case that the domain is a rectangular element, e.g., a cell of a Cartesian grid, Section 3.2 described a method based on using a subcell decomposition together with a projection procedure. This approach can naturally be extended to other cases, such as polynomials defined on triangular or tetrahedral elements, by using a similar decomposition method to generate and project points. Guidelines regarding the sampling resolution are provided shortly.

(3) After sampling the interface on the whole domain, one obtains a cloud of points $C = \{x_1, \dots, x_N\} \subset \mathbb{R}^d$. In the final step of initialisation, a data structure for efficient closest point queries is then created; an example of a k-d tree optimised for surfaces is discussed in the Appendix.

• *Closest point evaluation.* Given an arbitrary point $x_q \in \mathbb{R}^d$, first find the closest point in C to x_q and use this as the initial guess to Newton's method for determining a high-order closest point, $\text{cp}(x_q)$; see Section 3.3.

Section 4.4 discusses the computational efficiency of the method. As an example application, for the reinitialisation problem in level set methods, we simply replace ϕ with the new signed distance function given by $x \mapsto \text{sign}(\phi(x))\|x - \text{cp}(x)\|$ evaluated at each grid node. We now consider some practical details:

• *Determining which cells contain the interface:* One of the simplest strategies for predicting when a grid cell contains the interface is to examine the sign of ϕ on the vertices of the grid cell — a grid cell is then declared to contain the interface if and only if the signs are not all the same. Clearly, this is not completely reliable. Two typical possibilities include: (a) a closed interface completely contained within a single cell; and (b) an interface which enters and exits the cell on one side/face without crossing any other side/face. In the case that the interface is well-resolved, (a) should not occur (unless subgrid details are to be expected as discussed shortly), but (b) may still occur. Nonetheless, the simple check of examining the signs on grid vertices can still be used to resolve situation (b), essentially because the polynomials from adjacent grid cells are allowed to overlap (as in Section 3.2 and Section 3.3). For example, a spherical interface defined on a high-resolution grid may partially cross the face of a grid cell without crossing any of its edges. On such a grid cell, the sign check of its vertices will not detect the interface, but the sign check on the neighbouring grid cell will identify the presence of the interface; the polynomial on this grid cell sufficiently approximates the interface in the original cell, due to the overlap allowed in sampling and in Newton's method.

Depending on the application, such as the subgrid capturing example in Section 4.5, it may be necessary to employ a more sophisticated strategy than to simply check signs of grid vertices. A simple approach is to suppose *every* grid cell contains the interface; in this case the sampling procedure, while being forced to sample more polynomials, would automatically avoid generating points for cells not containing the interface. Another possibility is to use properties of the polynomial p itself to evaluate bounds of the form

$$\max_{x \in B(x_c, r)} |p(x) - p(x_c)| < C,$$

where C provides a uniform bound on the values of $p(x)$ for x in a ball centred at x_c with a certain grid-dependent radius. If $|p(x_c)| > C$ we can thus prove the polynomial has no zero level set in the corresponding ball.

• *Sampling resolution.* Since the closest point in C to the query point x_q forms an initial approximation to the true closest point, it follows that the sampling resolution of the seed points in C should locally depend on the amount of curvature exhibited by the interface. In other words, on each individual grid cell/mesh element, the length scale characterising the typical separation distance between seed points should be on the same order as the smaller of Δx or the smallest radius of curvature of the interface on that mesh element. In almost all practical applications of the level set method, the interface (once approximated by polynomials) rarely exhibits curvature higher than $\mathbb{O}(1/\Delta x)$. Thus, in Section 3.2, the simple strategy of using a $m \times m$ $(\times m)$ subgrid to generate points with $m = 2$ or $m = 3$ typically suffices. In other cases, e.g., a triangular or tetrahedral mesh element, a similar decomposition can be used to sample with similar resolution. For very high-order level set methods, it is possible to capture subgrid effects, in which a single grid cell may contain, for example, an isolated spherical droplet. In these applications, it may be necessary to make m larger. On the other hand, for reasons of efficiency we do not want an excessive number of points in the cloud C since this affects the performance of closest point queries. Ideally, we would like a sampling algorithm that automatically adapts to the curvature exhibited by each polynomial on each mesh element. One possibility for achieving this is to analyse the polynomial and its coefficients to calculate bounds on second derivative information across the entire cell — using these bounds, m could be made automatically adaptive such that $m = 1$ or 2 in smooth parts of the domain, with m larger in regions of high curvature. Though feasible, we will not pursue this idea here or in the accompanying C++ code for the sake of overall simplicity.

• *Overlapping threshold.* The approximate distance between points in the cloud C is also a good measure of how much to allow adjacent grid cell polynomials to overlap. Generally, the polynomials overlap by about $\frac{1}{2}\Delta x$ or less; this is also used as the radius of the bounding ball in Newton's method.

• *Treatment of boundary conditions.* In the case of a Cartesian grid, we assumed that we could apply stencils at each relevant grid cell to obtain high-order polynomial approximations. However, this cannot be immediately applied at the boundary of the computational domain, since grid points outside the domain do not exist. Here it is necessary to either (i) find polynomials based on interior data only, and/or (ii) enforce a given boundary condition on ϕ or its interface, such as a zero Neumann boundary condition. One could use, for instance, "ghost layers" in which grid points are defined outside of the domain whose values are based on extrapolation. Since accurate treatment of boundary conditions is highly application-dependent, we will not consider this further here.

• *Narrow banding.* A common implementation of level set methods is to only define ϕ in a small narrow band surrounding the zero level set of ϕ, given by those grid points x for which $\|x - \mathrm{cp}(x)\| < r$, where r is a band radius equal to a fixed number of grid cells [1]. It is straightforward to modify a k-d tree search to consider only points for which the distance to the query point is less than r, returning null if no such points exist; search queries can use this extra information to very efficiently determine the location of the narrow band.

• *Parallelisation.* Parallelising closest point/distance algorithms such as this depends crucially on the intended application. In a level set method, it is often the case that the global domain is subdivided into subdomains, with individual subdomains assigned to individual processors. In this case, and when narrow banding, it is straightforward to parallelise the closest point algorithm: (i) each processor would examine the grid cells in its subdomain and sample the interface; (ii) points that are within a distance r from its subdomain boundary are communicated, together with their associated polynomials, to adjacent processors; (iii) each processor can then proceed completely independently from the rest by building a k-d tree for a slightly larger-sized subdomain. If the application cannot narrow-band, or if a different type of processor decomposition is used, another strategy is likely necessary, such as communicating between processors coarse-grained information about the geometry of the interface.

4. Results

4.1. ***Convergence tests.*** For a sufficiently smooth level set function ϕ, each of the methods in Table 1 for approximating ϕ in each grid cell is p-th order accurate. Combined with the closest point algorithm, this leads to an approximation of the closest point function $\mathrm{cp}_h(x)$ and distance function $d_h(x) := \|x - \mathrm{cp}_h(x)\|$. In general, we can expect the distance function approximation to be p-th order accurate, both near and far away from the interface. On the other hand, the closest point approximation may lose up to two orders of accuracy if x is near a "curvature singularity," e.g., near the centre of a circle. One way to see this is to note that the exact closest point and distance functions satisfy the relation $\mathrm{cp}(x) = x - d(x)\nabla d(x)$

almost everywhere.[7] Therefore, to recover cp_h from d_h we need to differentiate d_h, thereby incurring an error proportional to $D^2d(x)h^{p-1}$. The Hessian of a distance function is related to the curvature of its level sets — D^2d has a singularity behaving like $\|x-x_c\|^{-1}$ near a centre of curvature x_c. It follows that $\mathrm{cp}_h(x)$ may be $(p-2)$-th order accurate near such a singularity, and some of our results confirm this.

To assess the approximation errors in d_h and cp_h, we consider a variety of smooth and nonsmooth test problems and measure the error locally and globally, in both the $\|\cdot\|_1$ and $\|\cdot\|_\infty$ norms. These tests are performed on a uniform grid such that $\Delta x = \Delta y = \Delta z = h$. More precisely:

- Let S be the set of points in the domain Ω for which the exact closest point function is multivalued, e.g., the centre of a sphere or the inside diagonals of a square. In a numerical setting, it would be overly complicated to request the closest point algorithm to return all possible solutions when the query point is in S. Hence, to simplify the convergence analysis, we will ignore grid points that are in S or situated very close to S, as follows. Let S_h be the set of grid points whose minimum distance to S is less than δ; in our results, the threshold has been set to half a grid cell, $\delta = \frac{1}{2}h$. Grid points in S_h are ignored *only* when measuring errors in the closest point function; they are still considered in the case of the distance function.
- Local errors are measured in a narrow band of radius 8 grid cells: let N_h be the set of grid points x for which $d_h(x) < 8h$. Define

$$\|d-d_h\|_{1,N_h} = \frac{1}{|N_h|}\sum_{x\in N_h}|d(x)-d_h(x)|,$$

$$\|\mathrm{cp}-\mathrm{cp}_h\|_{1,N_h\backslash S_h} = \frac{1}{|N_h\backslash S_h|}\sum_{x\in N_h\backslash S_h}\|\mathrm{cp}(x)-\mathrm{cp}_h(x)\|,$$

  and

$$\|d-d_h\|_{\infty,N_h} = \max_{x\in N_h}|d(x)-d_h(x)|,$$

$$\|\mathrm{cp}-\mathrm{cp}_h\|_{\infty,N_h\backslash S_h} = \max_{x\in N_h\backslash S_h}\|\mathrm{cp}(x)-\mathrm{cp}_h(x)\|.$$

- Global errors are measured across the entire computational domain Ω, with the same definitions of the norm, except that N_h is replaced with the set of all grid points; grid points in S_h are ignored only when measuring errors in the closest point function.

A variety of test problems have been analysed, including a circle, sphere, ellipse, ellipsoid, square, cube, rectangle with rounded ends and a cylinder with rounded

[7]The relation is not defined at shocks where d is not differentiable — equivalently, where there is more than one closest point to x.

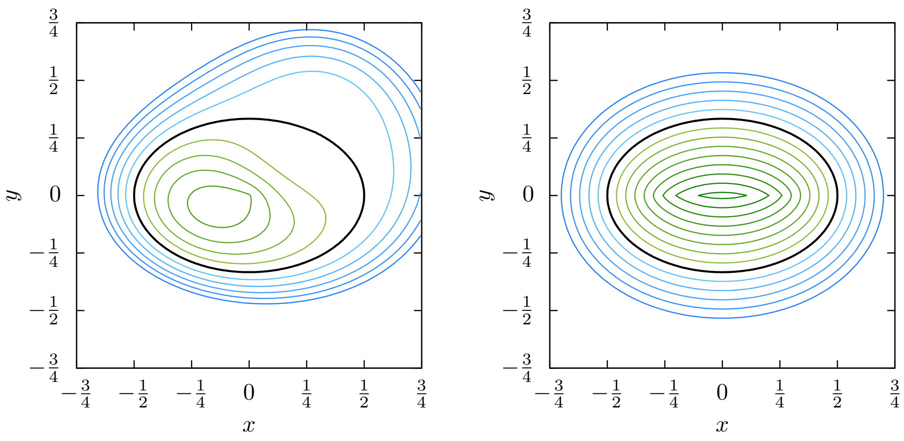

Figure 2. Two-dimensional test case corresponding to an ellipse with semimajor axis $\frac{1}{2}$ and semiminor axis $\frac{1}{3}$. Left: contours of the initial level set function ϕ given by (2), with the zero level set indicated by a thick line. Right: reinitialised signed distance function.

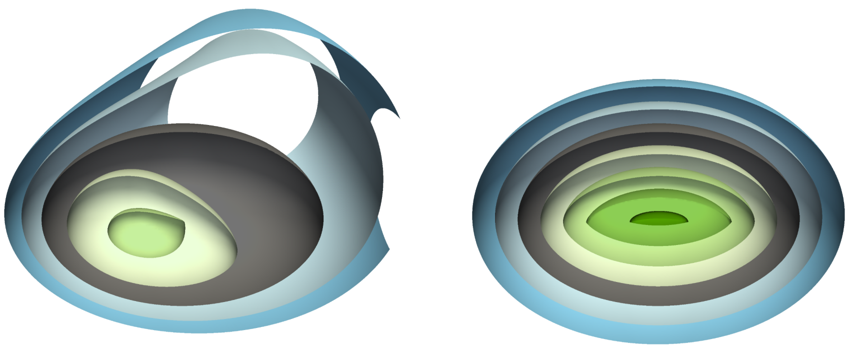

Figure 3. Three-dimensional test case corresponding to an ellipsoid with semiprincipal axes $\frac{1}{2}$, $\frac{1}{3}$ and $\frac{1}{2}$. Left: contours of the initial level set function ϕ given by (3), with the zero level set shown in dark grey. Right: reinitialised signed distance function. In both figures, the contours have been sliced by a plane in order to reveal the inner structure.

ends. Here we show results for the most instructive case, that of an ellipse in 2D and ellipsoid in 3D, followed by a summary of the results of the other tests. In all cases, the domain is $\Omega = \left[-\frac{3}{4}, \frac{3}{4}\right]^d$ and the level set function ϕ is defined on a uniform Cartesian $n \times n$ grid (in 2D) or $n \times n \times n$ grid (in 3D). For the ellipse, ϕ is evaluated at grid points via

$$\phi(x, y) = \left(1 - \exp(-(x - 0.3)^2 - (y - 0.3)^2)\right)\left(\sqrt{4x^2 + 9y^2} - 1\right), \tag{2}$$

and for the ellipsoid,

$$\phi(x, y, z) = \left(1 - \exp\left(-(x - 0.3)^2 - (y - 0.3)^2\right)\right)\left(\sqrt{4x^2 + 9y^2 + 4z^2} - 1\right). \tag{3}$$

These functions were designed to exhibit large changes in the norm of the gradient near the zero level set. Figures 2 and 3 show contours of ϕ as well as their reinitialised counterparts. We can see that the resulting distance function is not smooth: in the case of the ellipse, $d(x)$ is not smooth on the segment

$$\left\{(x,y) : |x| \le \tfrac{1}{2} - \tfrac{2}{9},\, y = 0\right\},$$

and has curvature singularities at

$$(x,y) = \left(\pm\left(\tfrac{1}{2} - \tfrac{2}{9}\right), 0\right).$$

A similar disc of nonsmoothness exists for the ellipsoid. Thus we expect to see differing rates of convergence depending on the local and global metrics. Tables 2 and 3 presents the convergence results[8] for all the polynomials of Table 1 in both 2D and 3D. For each type of polynomial, the convergence rate is estimated by taking ratios of errors between different grid sizes and are indicated by bold numbers in the two tables. The results can be summarised as follows:

- The bicubic and tricubic polynomials (which recall are designed to find a C^1 interpolant of the level set function) are locally third order accurate, for both the distance function and closest point function. Globally, the distance function is third order; however, the closest point function is approximately first order.
- For each of the Taylor polynomials of degree d_T, letting $p = d_T + 1$ (as in Table 1), both the distance and closest point functions are locally p-th order accurate. Globally, the distance function is also p-th order accurate. The closest point function is globally $(p-1)$-th order accurate in the L^1 norm, and is between $(p-2)$-th and $(p-1)$-th order accurate in the maximum norm.

Thus, we obtain the optimal convergence rate in both the distance function and closest point function, depending on proximity to the interface or curvature singularities. To be more precise, for a sufficiently smooth interface there are three zones of convergence: (i) if x is such that $d(x) = \mathbb{O}(h)$ (as in a narrow band), then the closest point approximation is p-th order accurate; (ii) if x is a fixed distance away from the interface (i.e., independent from h) and is not located at a curvature singularity, then the closest point approximation is $(p-1)$-th order accurate; and (iii) if x has distance $\mathbb{O}(h)$ from a curvature singularity, then the closest point approximation is $(p-2)$-th order accurate. The distance function approximation d_h is p-th order accurate in all three zones (for a sufficiently smooth interface).

[8]In order to measure the error, we need the exact closest point function for an ellipse and ellipsoid. This was implemented by using Newton's method, similar to that developed in Section 3.3, applied to the polynomials $4x^2 + 9y^2 - 1$ in 2D and $4x^2 + 9y^2 + 4z^2 - 1$ in 3D, with enough iterations to compute the closest point to machine precision accuracy.

			Distance error				Closest point error			
	d	n	$\lVert\cdot\rVert_1$		$\lVert\cdot\rVert_\infty$		$\lVert\cdot\rVert_1$		$\lVert\cdot\rVert_\infty$	
Bicubic	2	64	3.66×10^{-5}		1.14×10^{-3}		6.21×10^{-4}		2.81×10^{-2}	
		128	6.90×10^{-6}	**2.4**	8.95×10^{-4}	**0.3**	2.07×10^{-4}	**1.6**	2.10×10^{-2}	**0.4**
		256	5.75×10^{-7}	**3.6**	1.23×10^{-4}	**2.9**	5.47×10^{-5}	**1.9**	1.04×10^{-2}	**1.0**
		512	6.95×10^{-8}	**3.0**	2.25×10^{-5}	**2.5**	1.50×10^{-5}	**1.9**	4.04×10^{-3}	**1.4**
Tricubic	3	64	2.85×10^{-5}		7.85×10^{-3}		5.47×10^{-4}		4.38×10^{-2}	
		128	3.77×10^{-6}	**2.9**	1.56×10^{-3}	**2.3**	1.58×10^{-4}	**1.8**	2.58×10^{-2}	**0.8**
		256	3.54×10^{-7}	**3.4**	2.59×10^{-4}	**2.6**	4.10×10^{-5}	**1.9**	1.18×10^{-2}	**1.1**
		512	3.75×10^{-8}	**3.2**	3.55×10^{-5}	**2.9**	9.89×10^{-6}	**2.1**	5.37×10^{-3}	**1.1**
Taylor degree 2	2	64	5.03×10^{-4}		1.20×10^{-2}		2.11×10^{-3}		6.88×10^{-2}	
		128	5.05×10^{-5}	**3.3**	1.32×10^{-3}	**3.2**	3.65×10^{-4}	**2.5**	2.09×10^{-2}	**1.7**
		256	5.95×10^{-6}	**3.1**	2.14×10^{-4}	**2.6**	8.86×10^{-5}	**2.0**	6.74×10^{-3}	**1.6**
		512	6.74×10^{-7}	**3.1**	3.11×10^{-5}	**2.8**	2.19×10^{-5}	**2.0**	2.64×10^{-3}	**1.3**
	3	64	2.20×10^{-4}		1.23×10^{-2}		1.32×10^{-3}		6.77×10^{-2}	
		128	2.25×10^{-5}	**3.3**	1.55×10^{-3}	**3.0**	2.84×10^{-4}	**2.2**	2.25×10^{-2}	**1.6**
		256	2.56×10^{-6}	**3.1**	2.24×10^{-4}	**2.8**	7.14×10^{-5}	**2.0**	7.10×10^{-3}	**1.7**
		512	2.91×10^{-7}	**3.1**	2.99×10^{-5}	**2.9**	1.80×10^{-5}	**2.0**	3.05×10^{-3}	**1.2**
Taylor degree 3	2	64	7.24×10^{-6}		4.31×10^{-4}		1.05×10^{-4}		1.84×10^{-2}	
		128	4.19×10^{-7}	**4.1**	1.79×10^{-5}	**4.6**	1.46×10^{-5}	**2.8**	2.00×10^{-3}	**3.2**
		256	2.52×10^{-8}	**4.1**	9.30×10^{-7}	**4.3**	1.93×10^{-6}	**2.9**	2.96×10^{-4}	**2.8**
		512	1.61×10^{-9}	**4.0**	5.94×10^{-8}	**4.0**	2.34×10^{-7}	**3.0**	3.88×10^{-5}	**2.9**
	3	64	4.48×10^{-6}		2.54×10^{-4}		7.28×10^{-5}		1.23×10^{-2}	
		128	2.69×10^{-7}	**4.1**	1.82×10^{-5}	**3.8**	9.58×10^{-6}	**2.9**	2.08×10^{-3}	**2.6**
		256	1.66×10^{-8}	**4.0**	1.05×10^{-6}	**4.1**	1.22×10^{-6}	**3.0**	3.08×10^{-4}	**2.8**
		512	1.03×10^{-9}	**4.0**	6.73×10^{-8}	**4.0**	1.54×10^{-7}	**3.0**	4.23×10^{-5}	**2.9**
Taylor degree 4	2	64	1.93×10^{-6}		8.14×10^{-5}		1.75×10^{-5}		1.12×10^{-3}	
		128	5.68×10^{-8}	**5.1**	2.53×10^{-6}	**5.0**	1.03×10^{-6}	**4.1**	9.43×10^{-5}	**3.6**
		256	1.80×10^{-9}	**5.0**	8.64×10^{-8}	**4.9**	6.32×10^{-8}	**4.0**	6.57×10^{-6}	**3.8**
		512	5.65×10^{-11}	**5.0**	2.89×10^{-9}	**4.9**	3.94×10^{-9}	**4.0**	4.92×10^{-7}	**3.7**
	3	64	9.97×10^{-7}		1.23×10^{-4}		1.10×10^{-5}		1.32×10^{-3}	
		128	3.14×10^{-8}	**5.0**	2.79×10^{-6}	**5.5**	6.69×10^{-7}	**4.0**	1.00×10^{-4}	**3.7**
		256	9.88×10^{-10}	**5.0**	1.01×10^{-7}	**4.8**	4.15×10^{-8}	**4.0**	6.34×10^{-6}	**4.0**
		512	3.09×10^{-11}	**5.0**	3.32×10^{-9}	**4.9**	2.59×10^{-9}	**4.0**	4.72×10^{-7}	**3.7**
Taylor degree 5	2	64	5.22×10^{-8}		1.52×10^{-6}		6.41×10^{-7}		4.84×10^{-5}	
		128	7.39×10^{-10}	**6.1**	3.01×10^{-8}	**5.7**	1.79×10^{-8}	**5.2**	1.08×10^{-6}	**5.5**
		256	1.18×10^{-11}	**6.0**	4.67×10^{-10}	**6.0**	5.28×10^{-10}	**5.1**	5.64×10^{-8}	**4.3**
		512	1.95×10^{-13}	**5.9**	7.31×10^{-12}	**6.0**	1.67×10^{-11}	**5.0**	2.31×10^{-9}	**4.6**
	3	64	4.64×10^{-8}		2.78×10^{-6}		6.25×10^{-7}		4.15×10^{-5}	
		128	7.22×10^{-10}	**6.0**	5.33×10^{-8}	**5.7**	1.86×10^{-8}	**5.1**	1.65×10^{-6}	**4.6**
		256	1.12×10^{-11}	**6.0**	8.21×10^{-10}	**6.0**	5.75×10^{-10}	**5.0**	4.86×10^{-8}	**5.1**
		512	1.75×10^{-13}	**6.0**	1.27×10^{-11}	**6.0**	1.79×10^{-11}	**5.0**	1.94×10^{-9}	**4.6**

Table 2. Convergence results (*global error*) for the ellipse (dimension $d = 2$) and ellipsoid ($d = 3$) for several polynomial classes: the bicubic (in 2D), tricubic (in 3D) and the Taylor polynomials in Table 1. The left pair measures the error in the distance function and the second pair the error in the closest point function. For each polynomial type, the error is indicated for a grid of size $n \times n$ in 2D and $n \times n \times n$ in 3D. Ratios between errors on successive grids yield the convergence rates in bold.

	d	n	Distance error				Closest point error			
			$\|\cdot\|_1$		$\|\cdot\|_\infty$		$\|\cdot\|_1$		$\|\cdot\|_\infty$	
Bicubic	2	64	2.98×10^{-5}		1.14×10^{-3}		3.47×10^{-4}		1.76×10^{-2}	
		128	4.31×10^{-6}	**2.8**	6.81×10^{-4}	**0.7**	5.95×10^{-5}	**2.5**	8.25×10^{-3}	**1.1**
		256	4.97×10^{-7}	**3.1**	7.22×10^{-5}	**3.2**	7.47×10^{-6}	**3.0**	1.69×10^{-3}	**2.3**
		512	5.89×10^{-8}	**3.1**	7.62×10^{-6}	**3.2**	8.09×10^{-7}	**3.2**	1.54×10^{-4}	**3.5**
Tricubic	3	64	2.25×10^{-5}		7.85×10^{-3}		2.67×10^{-4}		3.88×10^{-2}	
		128	2.26×10^{-6}	**3.3**	1.50×10^{-3}	**2.4**	3.62×10^{-5}	**2.9**	1.30×10^{-2}	**1.6**
		256	2.42×10^{-7}	**3.2**	1.35×10^{-4}	**3.5**	4.29×10^{-6}	**3.1**	4.27×10^{-3}	**1.6**
		512	2.97×10^{-8}	**3.0**	1.37×10^{-5}	**3.3**	5.27×10^{-7}	**3.0**	2.73×10^{-4}	**4.0**
Taylor degree 2	2	64	3.24×10^{-4}		1.19×10^{-2}		9.08×10^{-4}		4.04×10^{-2}	
		128	3.64×10^{-5}	**3.2**	1.31×10^{-3}	**3.2**	1.00×10^{-4}	**3.2**	5.56×10^{-3}	**2.9**
		256	4.29×10^{-6}	**3.1**	2.13×10^{-4}	**2.6**	1.15×10^{-5}	**3.1**	1.08×10^{-3}	**2.4**
		512	5.18×10^{-7}	**3.0**	3.09×10^{-5}	**2.8**	1.30×10^{-6}	**3.1**	1.09×10^{-4}	**3.3**
	3	64	1.58×10^{-4}		1.23×10^{-2}		6.41×10^{-4}		3.97×10^{-2}	
		128	1.71×10^{-5}	**3.2**	1.53×10^{-3}	**3.0**	7.91×10^{-5}	**3.0**	6.96×10^{-3}	**2.5**
		256	1.96×10^{-6}	**3.1**	2.18×10^{-4}	**2.8**	9.67×10^{-6}	**3.0**	1.08×10^{-3}	**2.7**
		512	2.37×10^{-7}	**3.0**	2.84×10^{-5}	**2.9**	1.18×10^{-6}	**3.0**	1.30×10^{-4}	**3.1**
Taylor degree 3	2	64	5.84×10^{-6}		2.04×10^{-4}		5.56×10^{-5}		9.14×10^{-3}	
		128	3.48×10^{-7}	**4.1**	1.42×10^{-5}	**3.8**	3.68×10^{-6}	**3.9**	3.62×10^{-4}	**4.7**
		256	2.20×10^{-8}	**4.0**	9.00×10^{-7}	**4.0**	2.24×10^{-7}	**4.0**	2.58×10^{-5}	**3.8**
		512	1.40×10^{-9}	**4.0**	5.94×10^{-8}	**3.9**	1.34×10^{-8}	**4.1**	1.34×10^{-6}	**4.3**
	3	64	3.83×10^{-6}		2.25×10^{-4}		3.55×10^{-5}		9.04×10^{-3}	
		128	2.36×10^{-7}	**4.0**	1.54×10^{-5}	**3.9**	2.28×10^{-6}	**4.0**	3.62×10^{-4}	**4.6**
		256	1.48×10^{-8}	**4.0**	9.59×10^{-7}	**4.0**	1.44×10^{-7}	**4.0**	2.57×10^{-5}	**3.8**
		512	9.25×10^{-10}	**4.0**	6.33×10^{-8}	**3.9**	8.99×10^{-9}	**4.0**	1.48×10^{-6}	**4.1**
Taylor degree 4	2	64	1.67×10^{-6}		8.14×10^{-5}		8.95×10^{-6}		6.96×10^{-4}	
		128	4.95×10^{-8}	**5.1**	2.48×10^{-6}	**5.0**	2.61×10^{-7}	**5.1**	1.75×10^{-5}	**5.3**
		256	1.58×10^{-9}	**5.0**	8.61×10^{-8}	**4.9**	8.00×10^{-9}	**5.0**	4.54×10^{-7}	**5.3**
		512	4.94×10^{-11}	**5.0**	2.80×10^{-9}	**4.9**	2.47×10^{-10}	**5.0**	1.77×10^{-8}	**4.7**
	3	64	9.32×10^{-7}		1.21×10^{-4}		5.56×10^{-6}		1.32×10^{-3}	
		128	2.92×10^{-8}	**5.0**	2.68×10^{-6}	**5.5**	1.72×10^{-7}	**5.0**	1.53×10^{-5}	**6.4**
		256	9.20×10^{-10}	**5.0**	1.01×10^{-7}	**4.7**	5.40×10^{-9}	**5.0**	5.07×10^{-7}	**4.9**
		512	2.88×10^{-11}	**5.0**	3.31×10^{-9}	**4.9**	1.69×10^{-10}	**5.0**	1.71×10^{-8}	**4.9**
Taylor degree 5	2	64	4.50×10^{-8}		1.35×10^{-6}		3.24×10^{-7}		2.71×10^{-5}	
		128	6.57×10^{-10}	**6.1**	3.01×10^{-8}	**5.5**	4.84×10^{-9}	**6.1**	2.11×10^{-7}	**7.0**
		256	1.03×10^{-11}	**6.0**	4.61×10^{-10}	**6.0**	7.06×10^{-11}	**6.1**	3.39×10^{-9}	**6.0**
		512	1.73×10^{-13}	**5.9**	7.15×10^{-12}	**6.0**	1.13×10^{-12}	**6.0**	5.65×10^{-11}	**5.9**
	3	64	4.08×10^{-8}		2.48×10^{-6}		2.87×10^{-7}		2.27×10^{-5}	
		128	6.44×10^{-10}	**6.0**	5.05×10^{-8}	**5.6**	4.50×10^{-9}	**6.0**	2.65×10^{-7}	**6.4**
		256	1.00×10^{-11}	**6.0**	7.70×10^{-10}	**6.0**	7.04×10^{-11}	**6.0**	4.27×10^{-9}	**6.0**
		512	1.57×10^{-13}	**6.0**	1.21×10^{-11}	**6.0**	1.10×10^{-12}	**6.0**	7.22×10^{-11}	**5.9**

Table 3. Convergence results (*local error in a narrow band of radius* 8 *grid cells*) for the ellipse (dimension $d = 2$) and ellipsoid ($d = 3$) for several polynomial classes: the bicubic (in 2D), tricubic (in 3D) and the Taylor polynomials in Table 1. The left pair measures the error in the distance function and the second pair the error in the closest point function. For each polynomial type, the error is indicated for a grid of size $n \times n$ in 2D and $n \times n \times n$ in 3D. Ratios between errors on successive grids yield the convergence rates in bold.

Three more test problems were considered, each with a different degree of smoothness. Here we summarise the convergence results[9]:

- *Circle in 2D and sphere in 3D.* In this case, the distance function is infinitely smooth except for a single isolated point at the origin of the circle/sphere. The results confirm that the distance and closest point functions are p-th order accurate everywhere, except near the singularity where only the closest point function loses two orders of accuracy.
- *Square in 2D and cube in 3D.* This test problem is more subtle. Locally, both the distance and closest point function are second order accurate in the L^1 norm, and first order accurate in the maximum norm, which is to be expected. Globally, the distance function is first order accurate in both norms. However, the closest point function is approximately half-order accurate globally in the maximum norm. The reason for achieving only half-order accuracy is as follows: the interpolation/approximation of the corner of a square inevitably leads to small bumps (see, e.g., Figure 5 on page 129). These small bumps are $\mathcal{O}(h)$ perturbations of the flat edge of the square, and are "seen" far away from the interface. The locus of points for which the distance to the bump equals the distance to the edge of the square approximately forms a parabola; see Figure 4. For a fixed distance away from the edge, the parabola's width is $\mathcal{O}(\sqrt{h})$. All grid points within the parabola see the bump, leading to a half-order error in the maximum norm of the closest point function. This loss of accuracy applies only to the outside of the square/cube. On the inside, the closest point function is multivalued along the diagonals (either two values in 2D or up to three in 3D). Along these shock lines of the distance function, the algorithm returns the exact distance to the interface (since the flat sides of the square/cube are exactly recovered).
- *Rounded rectangle and cylinder with rounded ends.* This example serves as a somewhat smooth but not infinitely smooth surface. In two dimensions, the interface is a square of width $\frac{1}{2}$ with two semicircles on the left and right sides; see Figure 4. In three dimensions, the interface is a cylinder with two hemispheres on either side. In both cases, the interface has a continuous normal vector field, but its curvature is discontinuous. Results show that both the distance function and closest point function are locally second order accurate, the distance function is globally second order accurate, and the closest point function is approximately second order accurate except near curvature singularities (all in the maximum norm).

[9]These convergence results may be reproduced by the reader with the C++ code accompanying this paper.

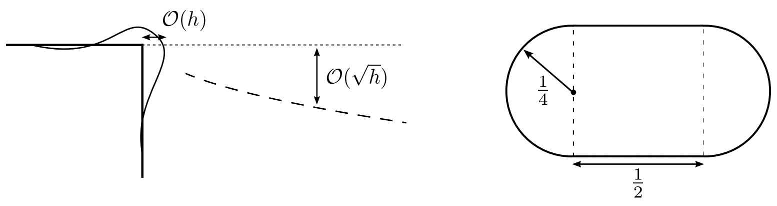

Figure 4. Left: polynomial approximation of a square leads to bumps on the corners which are $\mathbb{O}(h)$ in width and are seen far away from the interface by the closest point function. The set of points for which the distance to the bump equals the distance to the original square approximately forms a parabola (indicated by the dashed line). Right: a rounded rectangle used in one of the convergence tests, consisting of a square of width $\frac{1}{2}$ with two semicircles of diameter $\frac{1}{2}$ on either end.

4.2. ***Convergence of Newton's method.*** Recall that the threshold for deciding convergence in Newton's method was whether $\|x_{k+1} - x_k\| < \epsilon$. In our test cases, we set[10] $\epsilon = h^p$, where p is the expected order of accuracy of the class of polynomials being used. Across all test problems it was found that in the vast majority of cases, Newton's method converged within 2–4 iterations. Table 4 illustrates the typical convergence behaviour with a histogram counting the number of steps taken by Newton's method accumulated across the entire computational grid. Generally

Test case	Polynomial	Number of iterations in Newton's method								
		1	2	3	4	5	6	7–20	F	E
2D ellipse	Bicubic	0.02	**33.6**	**64.6**	1.2	0.2	0.1	0.01	0.2	
2D ellipse	Taylor degree 2	0.02	**34.4**	**65.3**	0.2					
2D ellipse	Taylor degree 4		0.3	**78.9**	**20.8**					
2D square	Bicubic	0.3	**87.8**	5.4	4.1	1.1	0.2	0.7	0.01	0.4
2D square	Taylor degree 2	0.3	**87.1**	8.6	4.0	<0.01				
2D square	Taylor degree 4	0.3	**87.2**	0.4	7.5	4.6				
3D ellipsoid	Tricubic	<0.01	9.7	**89.4**	0.6	0.2	0.04	<0.01	0.07	<0.01
3D ellipsoid	Taylor degree 2	<0.01	9.3	**90.7**	0.05	<0.01				<0.01
3D ellipsoid	Taylor degree 4		<0.01	**62.2**	**37.8**					
3D cube	Tricubic	<0.01	**72.4**	**10.9**	**10.3**	3.7	0.5	1.5	0.1	0.6
3D cube	Taylor degree 2	<0.01	**71.2**	**19.1**	9.7	<0.01				
3D cube	Taylor degree 4	<0.01	**70.7**	0.08	**16.2**	**13.0**	<0.01	<0.01		

Table 4. Convergence of Newton's method. In each case, executed on either a 256×256 grid (in 2D) or $256 \times 256 \times 256$ grid (in 3D), the percentage of grid points which needed the indicated number of steps for convergence is shown; entries greater than 10% are in boldface. A blank cell indicates exactly 0%, "F" means Newton's method did not converge within 20 iterations, and "E" means the iterate left the bounding ball that determines the amount of overlap between adjacent grid cells.

[10]In fact, we set $\epsilon = \max(10^{-14}, h^p)$ to ensure that convergence is declared when using a highly resolved grid for which errors are limited to double precision arithmetic.

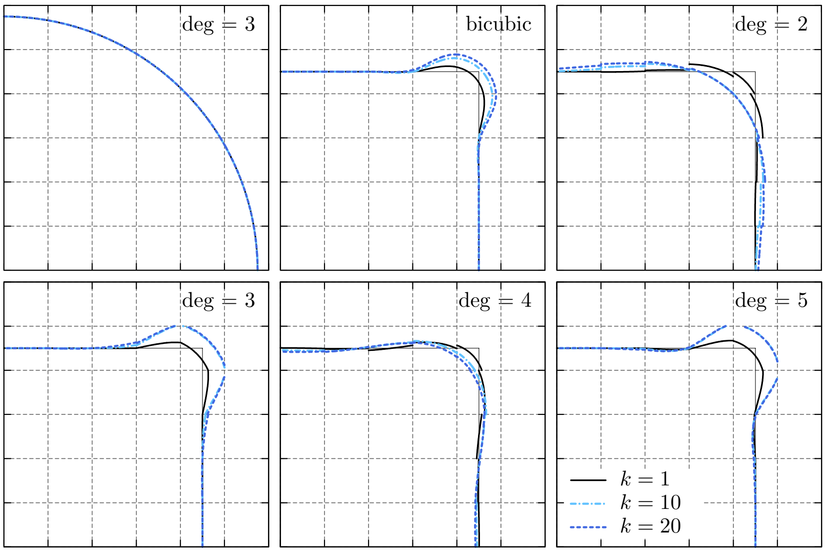

Figure 5. Effect of multiple reinitialisations on an interface, zooming in on a 6×6 patch of grid cells to observe cell-sized effects. Each patch shows the interface after one reinitialisation (solid line), after 10 reinitialisations (dashed-dot line), and after 20 reinitialisations (dashed). Top left: reinitialisation applied to a circle of radius approximately 5.5 grid cells, computed using a Taylor polynomial of degree 3. Remaining grids: reinitialisation applied to the corner of a square where, except for the bicubic, a Taylor polynomial of the indicated degree is used.

speaking, it is very rare that more than 6 iterations are needed. Note also that in the case of the square and cube, the majority of grid points requires just two iterations: when the closest point is on a flat face of the square/cube, only one iteration is needed for convergence, but two iterations are necessary to detect this.

4.3. ***Repeated reinitialisation.*** In level set methods, it is often necessary to periodically reinitialise the level set function as a distance function and a common practice for doing this is to reinitialise ϕ every fixed number of steps. Reinitialising as frequently as this may even be necessary to converge to the correct solution, such as in the Voronoi Implicit Interface Method [25; 26] which evolves an unsigned distance function. It follows that an important requirement of a reinitialisation method is that any perturbation in the location of the interface should be made as small as possible. As an example, if the time step for an evolving simulation is $\Delta t = \mathcal{O}(h^2)$, then the level set function will be reinitialised approximately $\mathcal{O}(h^{-2})$ many times over the course of the simulation. The accuracy of the reinitialisation procedure must then necessarily be at least second order accurate — in fact, it often needs to be much higher to ensure that accumulated errors from reinitialisation do not dominate the overall error.

In addition to requirements on the order of accuracy, it should also be confirmed that the reinitialisation method is stable when invoked on the same problem multiple times [10]. Let $R(\phi)$ be the operator which takes a grid-defined level set function ϕ and returns an approximation to the signed distance function evaluated on the same grid. Of interest is the error in the interface of $R \circ \cdots \circ R(\phi) = R^k(\phi)$ after k reinitialisations, compared to the original interface of ϕ. Figure 5 illustrates the behaviour for a smooth interface (a circle) and nonsmooth interface (the corner of a square), zooming in on a 6×6 patch of grid cells. In Figure 5, the interface, defined by the zero level set of the relevant polynomial of each grid cell, is shown for $k = 1$, 10, and 20. For smooth interfaces, and provided the reinitialisation method is at least third-order accurate, the effect of reinitialisation is essentially unobservable, except on extremely coarse grids. For nonsmooth interfaces, we expect to see $\mathcal{O}(h)$ perturbations; however, we do not wish the amount of perturbation to rapidly grow as k is increased.

To analyse this more carefully, a metric measuring the amount of perturbation is required. Here we use a metric that measures the maximum deviation in the interface, defined by the Hausdorff distance d_H: given two interfaces Γ_1 and Γ_2 (each a surface of codimension-one), define

$$d_H(\Gamma_1, \Gamma_2) = \max\bigl(\sup_{x\in\Gamma_1} d(x, \Gamma_2), \sup_{x\in\Gamma_2} d(x, \Gamma_1)\bigr), \tag{4}$$

where $d(x, \Gamma_i)$ is the minimum distance from x to interface Γ_i. Figure 6 plots the error[11] for a circle in 2D, a sphere in 3D, a square in 2D, and a cube in 3D, for k between 1 and 20 iterations, for all polynomial types considered in this paper. We observe that in all cases, the error after multiple reinitialisation steps is stable and remains on the same order as the original approximation error. Another metric for measuring accuracy of reinitialisation methods is the ability to conserve area/volume — since the Hausdorff metric bounds the error in area/volume conservation, it follows that the closest point algorithm also preserves volume (both locally and globally) with at least the same order of accuracy.[12]

4.4. ***Computational efficiency.*** Let N be the number of grid cells containing the interface. Then the basic computational complexity of the algorithm is: (i) $\mathcal{O}(N)$ to

[11] To actually compute the Hausdorff distance, we supersample each interface by using a subgrid of 10^d subcells per grid cell, such that the exact solution has a cloud of points $\{x_{i,\text{exact}}\} \subset \Gamma_{\text{exact}}$, and the approximate interface has a cloud of points $\{x_{j,h}\} \subset \Gamma_h$. With these source points, we compute $d_H \approx \max(\max_i d(x_{i,\text{exact}}, \Gamma_h), \max_j d(x_{j,h}, \Gamma_{\text{exact}}))$, where $d(\cdot, \Gamma_{\text{exact}})$ is evaluated using knowledge of the exact solution, while $d(\cdot, \Gamma_h)$ is evaluated using Newton's method, similar to that described in Section 3.3, but with convergence to machine precision. Altogether, this is a sufficiently accurate approximation of the true Hausdorff distance.

[12] In the case of the square, which has first order approximation errors at the corners of the square, the area of the reinitialised square is in fact second order accurate.

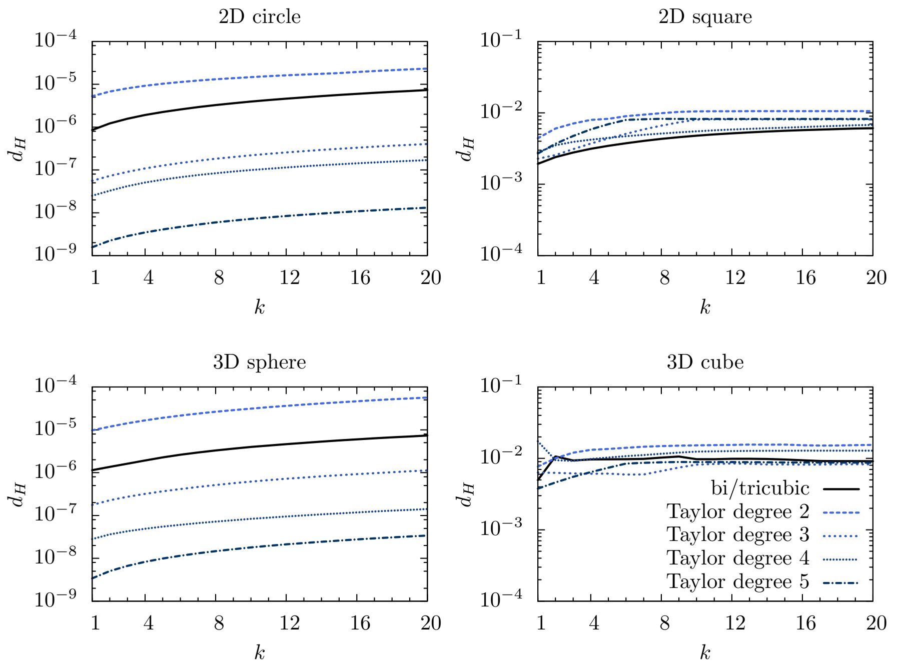

Figure 6. Error in the position of the interface after k repeated reinitialisations, where the error is measured by the Hausdorff distance (4) between the exact interface and the interface defined by the relevant polynomials of each grid cell. The legend in the bottom-right figure applies to all figures. In each case, a domain $[-1, 1]^d$ is subdivided into a grid of 128×128 cells (in 2D) or $128 \times 128 \times 128$ cells (in 3D). Top left: error for a circle of radius $\frac{1}{2}$. Top right: error for a square of width 1. Bottom left: error for a sphere of radius $\frac{1}{2}$. Bottom right: error for a cube of width 1.

construct grid cell polynomials and sample their zero level set; (ii) $\mathcal{O}(N \log N)$, on average, to construct the k-d tree; (iii) $\mathcal{O}(\log N)$, on average, per closest point query in searching the k-d tree for the closest point in C (in the best case it is $\mathcal{O}(1)$; in the worst case it can be $\mathcal{O}(N)$, as for example when x_q is at the centre of a sphere); and (iv) $\mathcal{O}(1)$ cost per query point in applying Newton's method in all cases. Roughly speaking, timing of individual components of the algorithm shows that:

- When computing the closest point function in a narrow band with radius a fixed number of grid cells (e.g., 5–15): constructing the polynomials takes 15–20% of the time, sampling up to 5%, constructing the k-d tree up to 5%, searching the tree between 40–70%, and running Newton's method between 15–30%.

- When computing the closest point in the entire domain, the size of N relative to the total number of grid points is more relevant. For medium to highly resolved grids, the majority of the overall computation time is spent solely in searching the k-d tree and running Newton's method. Depending on the dimension of the problem

Test problem	FMM	High-order
2048×2048 grid, narrow band	0.08 s	0.1 s
2048×2048 grid, entire domain	1.6 s	1.9 s
$256 \times 256 \times 256$ grid, narrow band	1.6 s	1.8 s
$256 \times 256 \times 256$ grid, entire domain	20 s	36 s

Table 5. Timing tests for a circular/spherical interface, performed on an Intel i7 3.1 GHz desktop machine (single core), comparing a second-order fast marching method [27] to the high-order closest point algorithm.

and accuracy of the polynomials, this ranges from 20% of the time in Newton's method and 80% of the time in searching to an even split between the two.

It follows that no single component of the algorithm clearly dominates the overall cost. To provide a general idea of the practical performance of the method, Table 5 compares its speed to a fast marching method which has been optimised for computing distances. (It is important to note that the two methods are intended for different classes of problems, so the comparison in speed should only be used as a guideline.) Further improvements in efficiency could be made by taking into account specific computing architecture, e.g., using more advanced optimisation techniques such as SSE instructions in k-d tree searches, but in the interest of simplicity and code portability these were not considered here.

4.5. ***Subgrid features.*** As our final example, we demonstrate that the algorithm for finding closest points on implicitly defined surfaces can accurately capture subgrid features in the interface, such as "droplets" completely contained within one grid cell. The problem setup is as follows. We begin with a level set function ϕ that is defined only at grid points. In the context of subgrid resolution, we then make the natural assumption that subgrid details are successfully captured by high-order approximations of ϕ on each grid cell.

Given this assumption, the pertinent issue in the closest point algorithm is whether the sampling procedure described in Section 3.2 can successfully detect and sample these subgrid details. In Section 3.4 it was discussed how this could be achieved by using polynomial bounds to automatically and adaptively determine where to place points on the zero level set of each grid cell's polynomial. Once sampled sufficiently well, Newton's method will successfully find the closest point on the interface. Figure 7 illustrates a pair of two-dimensional examples in which the values of ϕ were defined on the grid points via a scaled version of the function

$$f(x, y) = x^2 - \tfrac{1}{27}y^3 + \tfrac{2}{3}y - \alpha.$$

Here α is a parameter determining which level set is considered the interface. Two cases with different values of α are shown in Figure 7, resulting in two topologically

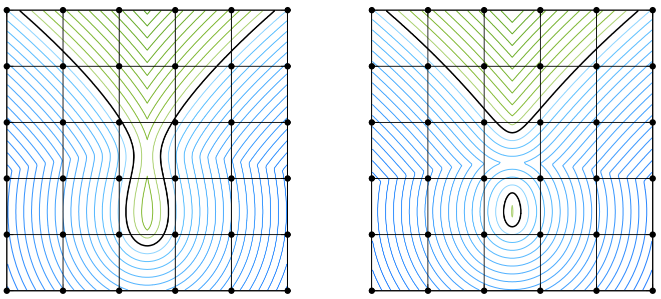

Figure 7. Capturing subgrid details of an interface defined by a high-order approximation on a 5×5 patch of grid cells. Here the level set function values are defined only at grid points. Shown are the contours of the computed signed distance function, evaluated throughout the 5×5 patch of grid cells. Left: a situation where three cells containing the interface have the same sign of the level set function at all their vertices. Right: an interface with two connected components with a droplet completely contained within one grid cell.

different interfaces. In either case, the cells in the middle cannot detect the presence of the interface by examining only the signs of ϕ at their vertices. High-order polynomial approximations can nevertheless recover these subgrid details, and these are accounted for in the closest point algorithm[13] throughout the 5×5 patch. An analogous problem in three dimensions is shown in Figure 8 using the same-sized grid cells, where again isolated droplets and long thin interfaces are correctly accounted for.

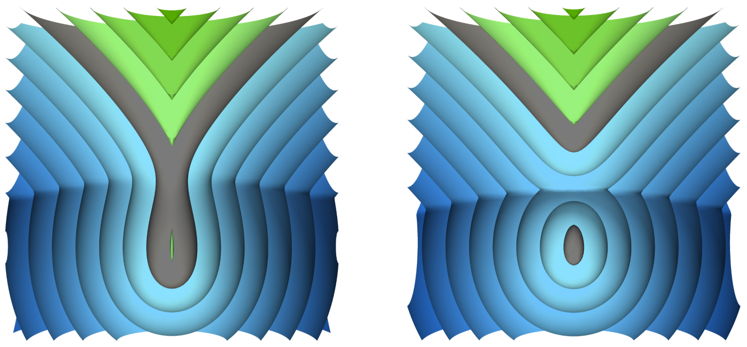

Figure 8. Analogy of Figure 7 in three dimensions corresponding to a $5 \times 5 \times 5$ patch of grid cells for which the interface (shown in dark grey) exhibits subgrid details. Though not shown in the figure, the cell sizes are identical to those in Figure 7. Here contours of the computed signed distance function have been cut by a plane in order to see inner details. Left: a situation in which the interface passes through several grid cells that have the same sign of the level set function on all their vertices. Right: a droplet completely contained within one grid cell.

[13]While automatic sampling is possible, in this particular example we used a simple method that used a subgrid of 10×10 to sample with (see Section 3.2) based on a Taylor polynomial of degree 3.

5. Concluding remarks

The presented method for computing high-order approximations of closest points on implicitly defined surfaces is straightforward — given a level set function defined by a high-order polynomial on each element of the computational domain, the zero level set is sampled to produce a sufficiently dense cloud of points approximating the interface. A closest point calculation proceeds by first finding the closest point in the cloud, and then improving this guess by using Newton's method. The results show that the algorithm is both robust and efficient — typically only 2–3 iterations of Newton's method are required to achieve convergence. In comparison to marching-based or PDE-based methods, for which implementation on unstructured meshes can be subtle, the presented approach can be used on highly unstructured meshes, or indeed at arbitrary query points. In the case of level set functions that are defined on a rectangular Cartesian grid, high-order polynomial approximations based on least-squares interpolation were presented. In other applications, such as gradient augmented level set methods or high-order discontinuous Galerkin finite element methods, the polynomials defining the interface are naturally specified. Convergence tests were performed and showed orders of accuracy of up to six in both the computed distance function and closest point function. For smooth problems, one obtains the optimal order of accuracy in both the computed distance and closest point functions. Near curvature singularities, the distance function remains high-order accurate, but the closest point function may lose up to two orders of accuracy.

The algorithm can be used to accurately reinitialise level set functions in level set methods. Though no time evolving simulations were presented in this paper, it has been successfully applied to a variety of moving interface problems which require very frequent reinitialisation, including in the Voronoi Implicit Interface Method [25; 26] for tracking interconnected interfaces with junctions. Some additional applications include:

- *Nonconstant extensions.* A common method for extending a function f defined on the interface is to make it constant along characteristics of the distance function, i.e., $f_{\text{ext}}(x) = f(\text{cp}(x))$. Some applications require this process to be bootstrapped in such a way that the extension is a linear or quadratic polynomial along characteristics; see, e.g., [29]. This can be achieved by a one-pass algorithm that calculates

$$f_{\text{ext}}(x) = f_0(\text{cp}(x)) + \|x - \text{cp}(x)\| f_1(\text{cp}(x)) + \tfrac{1}{2}\|x - \text{cp}(x)\|^2 f_2(\text{cp}(x)) + \cdots,$$

  where the f_i are the coefficients of the polynomial restricted to a particular characteristic.

- *High-order evaluation of curvature of the signed distance function.* Many applications of the level set method require accurate calculation of the mean curvature κ of the interface or other level sets of the signed distance function. Let $\boldsymbol{n}$ be the normal vector field of the signed distance function determined by the interface of a given level set function ϕ (which is not necessarily itself a distance function). Then it can be shown that derivatives of $\boldsymbol{n}$ at a point x are related to the derivatives of $\boldsymbol{n}$ evaluated at the closest point $\mathrm{cp}(x)$ via

  $$\nabla \boldsymbol{n}(x) = \big(I + \|x - \mathrm{cp}(x)\| \nabla \boldsymbol{n}(\mathrm{cp}(x))\big)^{-1} \nabla \boldsymbol{n}(\mathrm{cp}(x)),$$

  while derivatives of $\boldsymbol{n}$ at a point y on the interface can be evaluated with

  $$\nabla \boldsymbol{n}\big(y\big) = \frac{1}{\|\nabla\phi\|}(I - \boldsymbol{n}\boldsymbol{n}^T) D^2\phi (I - \boldsymbol{n}\boldsymbol{n}^T)\big|_y, \quad \boldsymbol{n}(y) = \frac{\nabla\phi(y)}{\|\nabla\phi(y)\|}, \quad y \in \{\phi = 0\}.$$

  These relations can be used to calculate curvature information of the signed distance function, such as the mean curvature $\kappa = \mathrm{tr}(\nabla \boldsymbol{n})$, via derivatives of ϕ evaluated only at the interface. This fits into the presented framework as we can then use the high-order polynomials approximating the interface itself, rather than relying on a finite difference scheme (say) applied to a precomputed grid-defined signed distance function.

We conclude by briefly describing the C++ code that accompanies this article (available on the author's web site). The code implements all the methods presented in this paper and can be used to verify the convergence results. In particular:

- Much of the code is templated on both the dimension d and the class of polynomials being used. To assist with part of this functionality, the code makes use of *blitz++* [6], an open-source implementation of d-dimensional arrays and fixed-length vectors in C++ with convenient expression template techniques.

- Ten different types of polynomials are provided: bicubic, tricubic, and each of the Taylor polynomials, with their corresponding pseudoinverses precomputed, as well as routines to evaluate the polynomial, its gradient and Hessian, using Horner's method.

- A k-d tree optimised for codimension-one surfaces (as described in the Appendix) is also supplied.

- A basic method for reinitialising a level set function as a signed distance function is also provided — it can be used to adapt the methods to different polynomial types and other applications.

Appendix: A k-d tree optimised for codimension-one manifolds

Given a fixed query point $x_q \in \mathbb{R}^d$, we would like to determine which point in $C = \{x_1, \dots, x_N\} \subset \mathbb{R}^d$ is closest to x_q. One of the most efficient data structures for closest point queries such as this is a *k-d tree*. A k-d tree organises the set of points into a hierarchy based on geometric considerations and allows for closest point queries in time approximately $\mathbb{O}(\log N)$. Each nonleaf node of the tree has two children: one child contains all the points on the "left" and the other child contains all the points on the "right." When searching a node in the tree for the closest point, the child which is more likely to contain the closest point is searched first; the other child is searched only if it could potentially contain a closer point than the current candidate. In a conventional k-d tree, "left" and "right" are determined by a hyperplane dividing the node's set of points into two, with normal direction equal to the x-axis, y-axis, z-axis, etc., cycled down the tree — the k in k-d tree refers to there being k dimensions to cycle through.

In the case that the points come from smooth surfaces, we can use the geometry of the surface itself to improve the efficiency of a k-d tree. The main idea for the tree developed in this work is to apply coordinate transformations in order to create "tight" bounding boxes. By using tighter bounding boxes, larger portions of the tree can be avoided when searching the tree. The essential ideas are as follows; for further details the reader is referred to the C++ code.

A node of the tree is either a leaf node, or else has exactly two children. A leaf node contains `leafsize` many points together with a bounding box of those points; typically `leafsize` is between 10–50 points, tunable according to computer hardware characteristics.[14] Each nonleaf node has four parameters: a pointer to the "left" and "right" children, a bounding box, and a pointer to a rotation matrix. The bounding box is of all the points represented by the node, i.e., the union of the bounding boxes of all its leaf nodes. The rotation matrix pointer, if not null, determines the coordinate transform which has been applied to all points represented by the node.

Delaying the description of constructing the tree for a moment, consider searching the tree to find the closest point to x_q. The basic routine for searching a node recursively is shown in Algorithm 2 and is initiated by calling `search` on the root node with $x = x_q$. The output is the index i of the closest point in C, where d^2 is the squared distance[15] from x_q to point i. Except for line 6, the search procedure is essentially identical to a normal k-d tree. The difference is that some nonleaf nodes may have a rotation matrix R, whose purpose is to apply a coordinate transform to

[14]It is often much more efficient to perform a linear search on a handful of points in the leaf nodes, compared to searching a tree whose leaf nodes contain a single point.

[15]It is much faster to compute and store squared distances, rather than the distance itself, as the former avoids expensive `sqrt` calls.

```
1:  **if** node is the root **then** initialise $i := -1$ and $d^2 := \infty$.
2:  **if** node is a leaf **then**
3:      **for** each of the points $x_j$ in the leaf **do**
4:          **if** $\|x_j - x\|^2 < d^2$ **then** update $i := j$ and $d^2 := \|x_j - x\|^2$.
5:  **else**
6:      **if** node has a rotation matrix $R$ **then** set $x \leftarrow Rx$.
7:      Calculate the squared distances $d_L^2$ and $d_R^2$ from $x$ to the bounding boxes of the left
8:        and right child nodes. (If inside a bounding box, the distance is zero.)
9:      **if** $d_L^2 < d_R^2$ **and** $d_L^2 < d^2$ **then**
10:         search(left child, $x$).
11:         **if** $d_R^2 < d^2$ **then** search(right child, $x$).
12:     **else if** $d_R^2 < d_L^2$ **and** $d_R^2 < d^2$ **then**
13:         search(right child, $x$).
14:         **if** $d_L^2 < d^2$ **then** search(left child, $x$).
```

Algorithm 2. `search`(node, x).

the query point x (the points contained in the children of such a node have already had their coordinates transformed). Ignoring the role of R for the moment, consider what makes k-d trees efficient: the ability to avoid searching entire parts of the tree. In particular, if the minimum distance from the query point to a child node's bounding box is larger than the distance to the current candidate closest point, then there is no point in searching that child node. It follows that a k-d tree can be made more efficient by attempting to make these bounding boxes as tight as possible. This is where we can utilise the fact that the cloud of points originate from a surface: after enough subdivision, groups of points are situated close to the same tangent plane of the surface, and so a bounding box rotated to align with the plane will be "thin." One possibility therefore is to store at each node of the tree a rotated bounding box. Instead, a more efficient approach is to transform the points themselves (once only, upon construction of the tree) by applying rotation matrices.

To explain the calculation of these rotation matrices, we turn now to the construction of the tree. Like `search`, this is also done recursively, and the basic method is shown in Algorithm 3. The tree construction is initiated by calling `buildtree` on the root node with the entire range of points ($j_\ell = 1$, $j_u = N$). Except for lines 5 to 13, the algorithm is essentially the same as constructing a normal k-d tree. These lines are responsible for deciding whether to perform a coordinate transformation and the specifics of that transform. Generally speaking, it is more efficient to delay the transforms until the k-d tree has subdivided sufficiently many times — on line 6, we can thus use metrics based on the current depth of the tree and how many levels there are to a leaf node; in addition, once a set of points is rotated, it is essentially unnecessary to consider them again for rotation. If the node is being considered for coordinate transformations, we must estimate the normal n of its set of points. One possibility for doing this is to use principal component analysis to estimate the principal direction for which the points change position the least, leading to an eigenvalue problem on the covariance matrix of the points' positions. A simpler and more efficient method for estimating n is as follows:

```
Compute the mean $\mu := (j_u - j_\ell + 1)^{-1} \sum_{j=j_\ell}^{j_u} x_j$.
Initialise $n := (1, 0, \dots, 0) \in \mathbb{R}^d$.
**for** $j = j_\ell$ **to** $j_u$ **do**
    $n \leftarrow n - \frac{(x_j - \mu) \cdot n}{\|x_j - \mu\|^2}(x_j - \mu)$
**if** $n \neq 0$ **then return** $n/\|n\|$ **else** do not consider transform.
```

Geometrically, this procedure removes components from n that are estimated to be in the tangent space. Although the calculation of n depends on the ordering of the points, it is unnecessary to estimate n with a high amount of accuracy. Returning to Algorithm 3, on line 10 the degree to which the new bounding box is "thin" is measured by comparing the new coordinates (in the normal direction) with 10% of the longest length of the untransformed bounding box. The factor of 10% was determined empirically to lead to the best overall efficiency. The remaining details of implementing this k-d tree are left to the C++ code.

In short, the above k-d tree, which has been optimised for point clouds arising from smooth surfaces, is about 4 to 10 times faster than a conventional k-d tree.

The reader may wonder if other methods of characterising the bounding regions may lead to even more efficient tree traversal. For example, if the points are known to come from a sphere (say) or a surface that locally looks like a sphere, one might construct a bounding "shell" which is curved to match the curvature of the sphere. While such an approach is possible, it turns out that computing the distance to a bounding shell is so computationally expensive that the overall cost of traversing the tree is greater, despite there being fewer nodes to search.

One final possibility for very efficient closest point queries is to use a combination of data structures. It is possible to construct a data structure with search operation costing $\mathbb{O}(1)$ (instead of $\mathbb{O}(\log N)$ on average as it is with k-d trees), specifically for points arising from smooth surfaces, provided the surfaces are very finely sampled. The idea is to use a k-d tree (or possibly a quadtree/octree/etc.) for the initial hierarchical subdivision of points, but only use a fixed number of levels so that the depth of the tree is bounded. The leaf nodes of the tree would then represent many thousands of points, and supposing that they essentially form a flat surface, these points could be binned into a conventional $(d-1)$-dimensional array, rotated to be tangent with the surface. Finding closest points in an array

```
1:  Compute the bounding box of the node.
2:  if $j_u - j_\ell + 1 <$ leafsize then
3:      Mark node as a leaf node and record range $j_\ell$, $j_u$.
4:      return
5:  Initialise node's $R$ matrix as null.
6:  if node is being considered for coordinate transformation then
7:      Estimate the normal $n$ of the surface approximated by the points $\{x_{j_\ell}, \dots, x_{j_u}\}$.
8:      Compute the coordinates of the points as though $n$ was an axis:
            $\alpha_{\min} = \min_{j_\ell \le j \le j_u} x_j \cdot n, \qquad \alpha_{\max} = \max_{j_\ell \le j \le j_u} x_j \cdot n.$
9:      Determine the longest length $L$ of the bounding box computed on line 1.
10:     if $\alpha_{\max} - \alpha_{\min} < 0.1L$ then
11:         Calculate an orthonormal basis $\{r_1, \dots, r_d\}$ using the normal $n$ as the first axis.
12:         Set the node's $R$ matrix as $R = [r_1, \dots, r_d]$.
13:         Transform all points: for $j_\ell \le j \le j_u$ do $x_j \leftarrow Rx_j$.
14: Determine the axis $k$ along which the (possibly new) bounding box of this node has
    greatest extent.
15: Calculate the median $m = \left\lfloor \frac{1}{2}(j_\ell + j_u) \right\rfloor$.
16: Rearrange the points $\{x_j\}$ such that $x_{j,k} \le x_{m,k}$ for all $j < m$ and $x_{j,k} \ge x_{m,k}$ for all
    $j > m$.
17: Split the points into two halves and build the left and right child nodes:
        node.left = new node; buildtree(node.left, $j_\ell$, $m$)
        node.right = new node; buildtree(node.right, $m$ + 1, $j_u$)
18: return
```

Algorithm 3. buildtree(node, j_ℓ, j_u).

such as this can be made to have $\mathcal{O}(1)$ cost, provided the points are essentially uniformly scattered throughout array. This idea was tested and compared with the performance of the above k-d tree. Despite being a $\mathcal{O}(1)$ search algorithm, the constant is sufficiently large that no benefits are obtained for standard-sized reinitialisation problems in level set methods — in other words, the interface is rarely "flat enough" compared to the resolution of the grid. The approach may be beneficial for very large 3D problems (such as those arising from $256 \times 256 \times 256$ grids or higher). For medium sized problems, the calculations required to search the k-d tree, and then transform the problem into searching a rotated array of binned particles, is too expensive compared to the above k-d tree with slightly larger depth. (On a related note, different techniques are possible for adapting k-d trees and other space partitioning algorithms to situations where the point data arises from a possibly unknown low-dimensional manifold embedded in a high number of dimensions; see, e.g., [11].)

Acknowledgements

The author thanks Ben Preskill and Ethan Van Andel at UC Berkeley for their helpful feedback in testing the C++ code as part of their work on advanced level set methods for inextensible and elastic surfaces. This research was supported in part by a Luis W. Alvarez Postdoctoral Fellowship at Lawrence Berkeley National Laboratory, the Laboratory Directed Research and Development Program of LBNL, and by the Applied Mathematics Program of the U.S. DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231.

References

[1] D. Adalsteinsson and J. A. Sethian, *A fast level set method for propagating interfaces*, J. Comput. Phys. **118** (1995), no. 2, 269–277.

[2] D. Adalsteinsson and J. A. Sethian, *Transport and diffusion of material quantities on propagating interfaces via level set methods*, J. Comput. Phys. **185** (2003), no. 1, 271–288.

[3] D. Adalsteinsson and J. A. Sethian, *The fast construction of extension velocities in level set methods*, J. Comput. Phys. **148** (1999), no. 1, 2–22.

[4] L. Anumolu and M. F. Trujillo, *Gradient augmented reinitialization scheme for the level set method*, Int. J. Numer. Methods Fluids **73** (2013), no. 12, 1011–1041.

[5] M. Bertalmio, L.-T. Cheng, S. Osher, and G. Sapiro, *Variational problems and partial differential equations on implicit surfaces*, J. Comput. Phys. **174** (2001), no. 2, 759–780.

[6] *Blitz++*, 2013, http://sourceforge.net/projects/blitz/.

[7] A. Bøckmann and M. Vartdal, *A gradient augmented level set method for unstructured grids*, J. Comput. Phys. **258** (2014), 47–72.

[8] J. U. Brackbill, D. B. Kothe, and C. Zemach, *A continuum method for modeling surface tension*, J. Comput. Phys. **100** (1992), no. 2, 335–354.

[9] D. L. Chopp, *Computing minimal surfaces via level set curvature flow*, J. Comput. Phys. **106** (1993), 77–91.

[10] ______, *Some improvements of the fast marching method*, SIAM J. Sci. Comput. **23** (2001), no. 1, 230–244.

[11] S. Dasgupta and Y. Freund, *Random projection trees and low dimensional manifolds*, Proceedings of the 40th Annual ACM Symposium on Theory of Computing, STOC '08, 2008, pp. 537–546.

[12] A. du Chéné, C. Min, and F. Gibou, *Second-order accurate computation of curvatures in a level set framework using novel high-order reinitialization schemes*, J. Sci. Comput. **35** (2008), 114–131.

[13] P. Esser and J. Grande, *An accurate and robust finite element level set redistancing method*, Tech. Report 379, Institut für Geometrie und Praktische Mathematik, 2013.

[14] M. W. Jones, J. A. Baerentzen, and M. Sramek, *3d distance fields: A survey of techniques and applications*, IEEE Trans. Vis. Comput. Graphics **12** (2006), no. 4, 581–599.

[15] C. B. Macdonald and S. J. Ruuth, *Level set equations on surfaces via the closest point method*, J. of Sci. Comput. **35** (2008), 219–240.

[16] R. Malladi, J. A. Sethian, and B. C. Vemuri, *Shape modeling with front propagation: A level set approach*, IEEE Trans. Pattern Analysis and Machine Intelligence **17** (1995), no. 2, 158–175.

[17] C. Min, *On reinitializing level set functions*, J. Comput. Phys. **229** (2010), no. 8, 2764–2772.

[18] J.-C. Nave, R. R. Rosales, and B. Seibold, *A gradient-augmented level set method with an optimally local, coherent advection scheme*, J. Comput. Phys. **229** (2010), no. 10, 3802–3827.

[19] S. Osher and R. Fedkiw, *Level set methods and dynamic implicit surfaces*, Applied Mathematical Sciences, Springer, 2003.

[20] S. Osher and J. A. Sethian, *Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations*, J. Comput. Phys. **79** (1988), no. 1, 12–49.

[21] P.-O. Persson and G. Strang, *A simple mesh generator in Matlab*, SIAM Review **46** (2004), no. 2, 329–345.

[22] A. Reusken, *A finite element level set redistancing method based on gradient recovery*, SIAM J. Numer. Anal. **51** (2013), no. 5, 2723–2745.

[23] G. Russo and P. Smereka, *A remark on computing distance functions*, J. Comput. Phys. **163** (2000), no. 1, 51–67.

[24] R. I. Saye, *An algorithm to mesh interconnected surfaces via the Voronoi interface*, Engin. Comput. (2013), 1–17.

[25] R. I. Saye and J. A. Sethian, *The Voronoi Implicit Interface Method for computing multiphase physics*, Proc. Nat. Acad. Sci. **108** (2011), no. 49, 19498–19503.

[26] ———, *Analysis and applications of the Voronoi Implicit Interface Method*, J. Comput. Phys. **231** (2012), no. 18, 6051–6085.

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

[28] ———, *Level set methods and fast marching methods: Evolving interfaces in geometry, fluid mechanics, computer vision, and materials sciences*, Cambridge University Press, 1999.

[29] J. A. Sethian and Y. Shan, *Solving partial differential equations on irregular domains with moving interfaces, with applications to superconformal electrodeposition in semiconductor manufacturing*, J. Comput. Phys. **227** (2008), no. 13, 6411–6447.

[30] J. A. Sethian and P. Smereka, *Level set methods for fluid interfaces*, Annual Review of Fluid Mechanics **35** (2003), 341–372.

[31] J. Strain, *Fast tree-based redistancing for level set computations*, J. Comput. Phys. **152** (1999), no. 2, 664–686.

[32] M. Sussman and E. Fatemi, *An efficient, interface-preserving level set redistancing algorithm and its application to interfacial incompressible fluid flow*, SIAM J. Sci. Comput. **20** (1999), no. 4, 1165–1191.

[33] M. Sussman and M. Y. Hussaini, *A discontinuous spectral element method for the level set equation*, J. Sci. Comput. **19** (2003), no. 1–3, 479–500.

[34] M. Sussman, P. Smereka, and S. Osher, *A level set approach for computing solutions to incompressible two-phase flow*, J. Comput. Phys. **114** (1994), no. 1, 146–159.

Received January 23, 2014. Revised April 30, 2014.

ROBERT I. SAYE: rsaye@lbl.gov
Lawrence Berkeley National Laboratory and Department of Mathematics, One Cyclotron Road, MS: 50A-1148, Berkeley, CA 94720, United States
http://math.lbl.gov/~saye/

Communications in Applied Mathematics and Computational Science

msp.org/camcos

See inside back cover or msp.org/camcos for submission instructions.

The subscription price for 2014 is US $75/year for the electronic version, and $105/year (+$15, if shipping outside the US) for print and electronic. Subscriptions, requests for back issues from the last three years and changes of subscribers address should be sent to MSP.

Communications in Applied Mathematics and Computational Science (ISSN 2157-5452 electronic, 1559-3940 printed) at Mathematical Sciences Publishers, 798 Evans Hall #3840, c/o University of California, Berkeley, CA 94720-3840, is published continuously online. Periodical rate postage paid at Berkeley, CA 94704, and additional mailing offices.

CAMCoS peer review and production are managed by EditFLOW® from Mathematical Sciences Publishers.

PUBLISHED BY

mathematical sciences publishers

nonprofit scientific publishing

http://msp.org/

© 2014 Mathematical Sciences Publishers

Communications in Applied Mathematics and Computational Science

vol. 9 no. 1 2014