Forward inclusion functions for ray-tracing implicit surfaces

Melike Aydinlilar, Cédric Zanni

Forward Inclusion Functions for Ray-Tracing Implicit Surfaces

M. Aydinlilar[1] , C. Zanni[1]

[1]Université de Lorraine, CNRS, Inria, LORIA

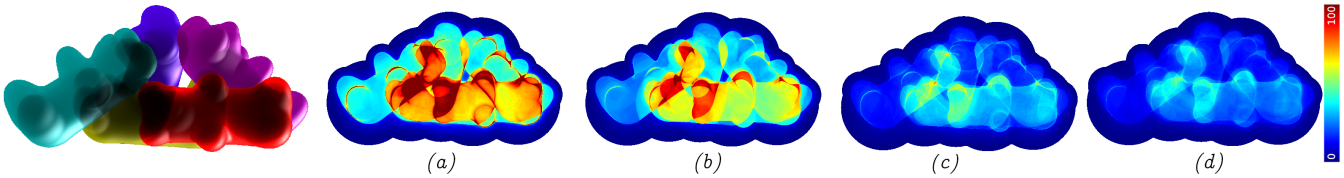

Figure 1: *Left: Transparent rendering of blobby objects combined with set-theoretic union operation defined as max() function. Right: Comparison of number of iteration steps for local directional Lipschitz bounds [GGPP20] (a), our linear mixed inclusion (b), our quadratic bottom-up inclusion (c) and our quadratic mixed inclusion (d).*

Abstract

Using Lipschitz bounds as linear inclusion functions, we show that both Lipschitz-based ray-tracing and bottom-up inclusion functions can be used together in the same framework. We propose asymmetrical forward inclusion functions that are exact at the query point and can better encode the function's behavior on a given interval; therefore well suited for iterative processing. We show how to derive the linear and quadratic versions of these inclusion functions either by bounding the derivatives or building bottom-up inclusion functions and combining these two. We show our results on density fields defined from point primitives with compactly supported kernels and Gaussian kernels as well as Hermite radial basis functions. We demonstrate that our method provides noticeable improvement for grazing rays and transparent rendering.

CCS Concepts

• ***Computing methodologies*** → ***Ray tracing; Volumetric models;***

1. Introduction

Implicitly defined surface representations, such as density fields [WMW86], functional representations [PASS95] and signed distance fields (SDFs) [Har96; TPFA21] are widely used for modeling surfaces. They support smooth blending of shapes by field compositions and provide direct inside/outside testing.

Extracting intermediate representations such as triangle meshes or voxel grids is commonly used for real-time rendering [WMW86; LC87; JLSW02; CZ21]. However, these intermediate representations are resolution-dependent and require additional memory for otherwise compact surface representations. On the other hand, ray tracing is a direct method for rendering iso-surfaces by finding the intersections between camera rays and the surface. As most field function definitions do not admit closed-form solutions, this root-finding problem is solved numerically by iteratively processing ray subintervals. The number of field evaluations performed along the rays is usually the driving complexity factor with typically challenging areas such as grazing rays where the ray is close to tangent to the surface, producing multiple roots for the root finding problem. In addition, transparent rendering amplifies numerical robustness and complexity issues.

Ray-tracing methods can be divided into two main families of approaches: the self-validated numerical methods, such as Interval Arithmetic [Mit90] and Affine Arithmetic [dFS04] and Lipschitz methods, such as Sphere Tracing [Har96] and Segment Tracing [GGPP20].

Self-validated numerical methods construct bottom-to-top inclusion functions by bounding each elementary arithmetic operation of the field function and are used with recursive ray bisection to iso-

late the roots. Inclusion functions are designed to limit the maximal error on a ray sub-interval, a property well adapted to the ray subdivision approach. However, the bounds suffer from overestimation for complex expressions due to error accumulations.

For Lipschitz methods, the evaluation position is marched iteratively on the ray with a guaranteed intersection-free step size computed from a bound on the absolute value of the field derivative (either globally [Har96] or on a ray-subinterval to improve the processing of grazing rays [GGPP20]). However, due to the symmetrical nature of these bounds, they do not capture the field functions' local increasing/decreasing behavior. This results in a large number of steps – and field evaluations – in blending areas as the variations of primitive fields cannot compensate for each other, and for transparent rendering as the rays leave the surface slowly.

We propose to bridge the gap between these two families of approaches by considering the Lipschitz property as a *forward* inclusion function: an inclusion function that is exact at the beginning of the interval and, therefore, well suited for iterative ray processing. Based on this observation, we first propose using *asymmetric* bounds; better suited for processing blending areas and transparent objects. Then, we extend our forward asymmetric inclusion functions to *quadratic* forward bounds that can be built using either *derivative analysis* or a *bottom-up constructive* approach. Finally, we show that both approaches can be combined in the same framework. We emphasize the efficiency of the proposed approach – including a significant reduction in the number of processing steps and lesser sensitivity to ray-tracing hidden parameters (minimal step size and the maximum number of steps) – on various examples consisting of blobby surfaces and Hermite radial basis functions, including both opaque and transparent rendering (Figure 1).

2. Preliminaries

Implicit surfaces are defined with a field function f as:

$$\left\{\mathbf{p} \in \mathbb{R}^3 \mid f(\mathbf{p}) = 0\right\}. \tag{1}$$

For density fields, the interior of the shape correspond to positive field values. These representations are compact and resolution independent. For modeling purposes, the field function is generally defined by hierarchical composition of base primitive field function – such as Blobby surfaces [Bli82; WMW86] – in a tree structure called a BlobTree [WGG99] using operations such as smooth blending or set- theoretic union operations (respectively summation and maximum function for density fields, see Figure 1). Primitives used for the evaluation of our method are presented in Section 4.4.

To ray-trace an implicit surface, the ray equation with origin $\mathbf{o}$ and direction $\mathbf{u}$:

$$\delta(t) = \mathbf{o} + t\mathbf{u} \tag{2}$$

is plugged into the surface equation (1), resulting in a one-dimensional root finding problem:

$$f \circ \delta(t) = 0. \tag{3}$$

Ray-tracing implicit surfaces comes down to solving this one-dimensional root-finding equation (3) for the ray parameter t. For most field functions, closed-form solutions are not available and numerical methods are used.

3. Related work

Numerical methods for ray-tracing implicit surfaces require iterative evaluation of the field function along the ray. The complexity is driven by the cost of the field evaluation itself and the number of field evaluations performed during the processing. In this work, we focus on the latter. Reducing the cost of the field evaluations for a large scene is an orthogonal problem and has been addressed by using acceleration techniques such as bounding volume hierarchies [GPP*10; GDW*16] and screen space bounding volumes [Bru19; AZ21].

Several numerical methods have been suggested for solving the ray-surface intersection equation. They differ in their robustness, performance, computational complexity, and universality. Early methods have suggested using *regula falsi* with Newton-Raphson root refinement [Bli82]. For polynomial fields, Laguerre's method [WT90] and Bézier clipping [NN94; BJ07; LZLW09] have been used. Bézier clipping uses the inclusion property of the Bézier control polygons to iteratively converge towards the first root and discard grazing rays more rapidly. Our work on the other hand, focuses on building polynomial bounds for a larger family of field functions. While using higher degree approximations is possible, root computations are more expensive and can easily get numerically unstable [LZLW09]. In this work, we chose to use linear and quadratic bounds that have more stable root computations. Polynomial approximation have been previously used for rendering convolution surfaces [AZ21; She99a] however, the approximations rely on specific knowledge on the field function behavior and do not provide robust visualization. Robust root isolation has been achieved by using Lipschitz constants on the first and the second derivatives to drive the bisection methods [KB89].

Lipschitz methods Lipschitz constant measures how fast can a given function change. For a function $f : \mathbb{R}^n \to \mathbb{R}$, a Lipschitz constant is a positive value L verifying:

$$|f(\mathbf{x}) - f(\mathbf{y})| < L\|\mathbf{x} - \mathbf{y}\|, \quad \mathbf{x}, \mathbf{y} \in \mathbb{R}^n.$$

In dimension three, if the function is differentiable, it can be defined as:

$$L = \max_{\mathbf{p} \in \mathbb{R}^3} \|\nabla f(\mathbf{p})\|.$$

Sphere tracing [Har96] has been introduced as a marching method with a guaranteed marching step size that would not cross the surface. Using the Lipschitz bound L, they show that it is possible to march along the ray with the following formula:

$$stepsize = \frac{f \circ \delta(t)}{L} \tag{4}$$

with t the current parameter along the ray. This formulation provides an easy-to-implement ray-marching algorithm, provided that the computational cost of calculating a practical Lipschitz bound is not high. SDFs present a useful special case of this approach, where $L = 1$. This way, the marching step size is simply equal to the field function evaluation value.

As stated in the original work, this formulation is similar to Newton-Raphson iterations. It uses the steepest possible slope, which is guaranteed not to cross the first intersection. Therefore,

Sphere Tracing converges linearly, and quadratically if the function is steepest at the first root [Har96]. Using a global Lipschitz bound can cause slow marching along the grazing rays where the field gradient tends to be orthogonal to the ray direction. To overcome this problem, Segment Tracing [GGPP20] uses *local* and *directional* Lipschitz bounds: an upper bound of the absolute value of the field directional derivative is computed on the ray sub-intervals to be processed.

Several successful attempts have been made to reduce the number of steps required by Sphere Tracing using various over-relaxation strategies [KSK*14; BV18]. These methods provide improvements on the marching step size for the overly conservative Lipschitz bounds in specific cases. Non-linear sphere tracing [SJNJ19] have been developed for rendering deformed implicit surfaces. Sphere-tracing has also been adapted to provide differentiable rendering of signed distance field for learning frameworks [TLY*21; VSJ22].

When marching on the density fields, due to the shape of the smoothing kernels, Sphere Tracing becomes less efficient as the global Lipschitz bound produces too small step sizes (Figure 2). Two distinct strategies have been applied to solve this problem [GGPP20; Bru19]. On the one hand, the *local* derivative bounds used in Segment Tracing [GGPP20] allow taking into account the low kernel derivative close to the limit of kernel support. However, due to the usage of a bound on the *absolute value* of the derivative, it does not distinguish the local increasing/decreasing behavior of the field function. By considering the local monotonicity, we can further improve the processing of grazing rays and transparent rendering. On the other hand, Bruckner [Bru19] proposes to map the density fields to approximate signed distance fields to render large-scale molecular models. Then, it is possible to render the scene efficiently using sphere tracing on the resulting normalized field. However, this method is specific to spherical molecular primitives with a unique radius in the scene. Furthermore, field normalization also requires a limited number of local primitive interactions. This field inversion technique has been extended for rendering integral surfaces with line segment skeletons [AZ21].

Self validated numerical methods Interval arithmetic was introduced by Moore [Moo66] in the 1960s to overcome measurement errors in scientific computing. It has been used to render implicit surfaces in early methods [Mit90; CHMS00] and recent GPU implementations [Kee20]. With interval arithmetic, every arithmetic operation is replaced with its interval equivalent. Hence, given an input interval, an output interval including all the possible function values can be constructed by simply evaluating the expression.

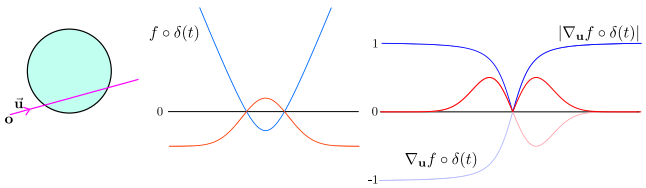

Figure 2: *Left: Ray/primitive configuration. Middle: SDF (blue) vs density field with Gaussian kernel (red) along the example ray. Right: The directional derivative of the SDF (blue) remains nearly constant on a large proportion of the ray (i.e. everywhere except in a small area around the argminimum of the distance).*

The main drawback of interval arithmetic is the overestimation of the bounds caused by the dependency problem. Higher-degree forms have been introduced to address this problem, such as Centered forms and Taylor forms [JKDW01; Neu02; MT06; Rat97].

Affine Arithmetic [dFS04] computes bottom-up inclusion functions – similar to interval arithmetic – while keeping track of dependent variables. Simplified versions have been suggested for ray-tracing implicit surfaces [FPC10; KHK*09; SJ22], which can be considered linear inclusion functions similar to first-order Taylor forms.

We build our linear and quadratic inclusion functions by drawing attention to the inclusion property of Lipschitz bounds. This way, we build a robust and efficient marching method using linear and quadratic inclusion functions.

4. Overview

Building on the previous methods, we start by emphasizing the inclusion property of Sphere Tracing [Har96]. The Lipschitz property provides us with a robust (up to numerical stability) inclusion function that can be visualized as a linear bound around the query point (Figure 3). Similarly, Segment Tracing [GGPP20] defines this inclusion function locally.

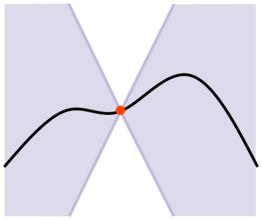

Figure 3: *Geometric interpretation of the Lipschitz bound*

We first extend this idea to local *asymmetric* linear bounds, and then, to quadratic bounds. We then show two different ways of calculating these bounds: bottom-to-top inclusion functions and, more directly, by bounding the derivatives.

These two methods can be used together, bounding the directional derivatives when it gives tighter bounds, and using bottom-up inclusion functions where it is not possible to work with derivatives (e.g., when the functions are piecewise-defined and locally non-differentiable).

Keeping the values exact at the interval starting point (Figure 4), it is possible to use these bounds in an iterative ray-marching algorithm for finding ray-surface intersections.

4.1. Inclusion Functions

For all t on a ray interval $[t_0, t_1]$, the *inclusion function* $[f]$ is defined as the set of points containing all possible values of $f(t)$. $[f]$ consists of two functions f_- and f_+, where $f_-(t) \leq f(t)$ and $f_+(t) \geq f(t)$. The *lower* and *upper bounds* of the inclusion function $[f]$ are $f_-(t)$ and $f_+(t)$. For a detailed reference on inclusion

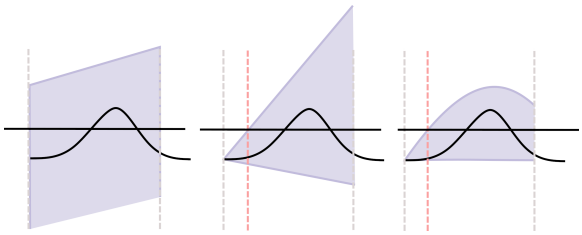

Figure 4: *Behaviour of Revised Affine Arithmetic [FPC10] (left), asymmetric linear (middle) and quadratic (right) inclusion functions. The latter two always allow to discard a subpart of the ray, and quadratic bounds are more likely to discard the entire interval.*

functions, related literature can be consulted [JKDW01]. We define our *forward* inclusion functions to be exact at the query point t_0, the starting point of the interval $[t_0, t_1]$:

$$f(t_0) = f_-(t_0) = f_+(t_0) \tag{5}$$

4.2. Validity Intervals

When combining inclusion functions, some inputs or intermediate results can cause invalid results, such as negative values in logarithms or square roots, and division by zero. In the previous self-validated numerical methods, these were declared *infinite* bounds and worked out by bisection; as the intervals get smaller, the degenerate cases would resolve. Since we march on the inclusion functions instead of bisecting them, we introduce validity intervals. Whenever an invalid result is encountered, the interval is shortened automatically during the evaluation of the inclusion function up to a point where the results are valid. In the subsequent operations of the inclusion function evaluation, this new shortened interval is used.

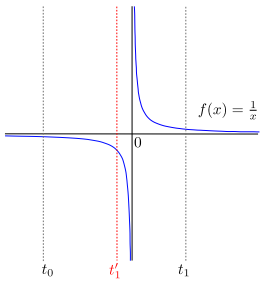

Figure 5: *Shortening the interval in case of division by zero. The subsequent operations in the field expression are going to be evaluated on the new interval* $[t_0, t_1']$.

4.3. Ray Surface Intersection

Similar to the previous safe marching methods, asymmetric linear and quadratic inclusion functions can be marched safely provided that the inclusion function has the same value as the field function at the starting point of the interval (i.e., it is a *forward* inclusion function). This is illustrated in Algorithm 1, which follows the same overall strategy as Segment Tracing [GGPP20]. The step size to advance along the ray is defined by the x-intercept of the upper or lower bound of the inclusion function, depending on whether the query point is inside or outside the surface. As inclusion functions must be constructed on a given input interval, an heuristic is used to compute the length of the next interval to process. In practice, we choose the next interval length to be twice the last safe step size (i.e. $k = 2$). In our case, the inclusion functions can also shorten their range of validity whenever an invalid operation is encountered during the calculations (Section 4.2). This algorithm, combined with asymmetric local bounds, provides a straight-forward and efficient marching for grazing rays and transparent rendering. The behaviour of different methods as forward inclusion functions can be seen in Figure 6.

Algorithm 1 Ray Marching Algortihm

```
procedure INTERSECT(Ray, Interval [t0, t1])
    t = t0
    t_end = t1
    while t < t1 do
        eval = f(o + tu)
        if eval < eps then
            root ← t                          ▷ A root is registered
        end if
        Inclusion, t_end = CalculateInclusion(f, Ray, [t, t_end])
                ▷ Inclusion calculation can shorten the interval
        r = Inclusion.CalcNextRoot()
        if r ∈ [t, t_end] then
            stepsize = r − t
        else
            stepsize = t_end − t          ▷ No roots in the interval, skip
        end if
        t += stepsize;
        t_end = min(t + k * stepsize, t1);
                ▷ Interval size based on the previous step
    end while
end procedure
```

4.4. Primitives

We focus our study on implicit surfaces defined by distance to point primitives combined with fall-off functions. For fall-off kernels, Gaussian kernels have been suggested for molecule rendering [Bli82]. Later, they were extended to polynomial kernels with similar fall-off behavior and compactly supported kernels for keeping the influence of a primitive contained in a prescribed area [She99b].

Given the distance d to the skeleton, and a radius r, Gaussian kernels [Bli82] are given in the form:

$$k_s(d) = \frac{1}{N} e^{-s\left(\frac{d}{r}\right)^2} \tag{6}$$

and compactly supported polynomial kernels [WW89] as:

$$k_{n,\sigma}(d) = \frac{1}{N} \begin{cases} \left(1 - \left(\frac{d}{\sigma r}\right)^2\right)^{\frac{n}{2}} & \text{if } d < \sigma r \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

where s and σ denote the kernel scale and N is a constant normalizing factor for achieving the radius r. In this falloff filter, n is the degree of the kernel controlling the smoothness of the field. For instance, for $n = 6$, the field is of class C^2. With this definition, our query functions along the ray for base primitives are in the form :

$$f(t) = k \circ d \circ \delta(t). \tag{8}$$

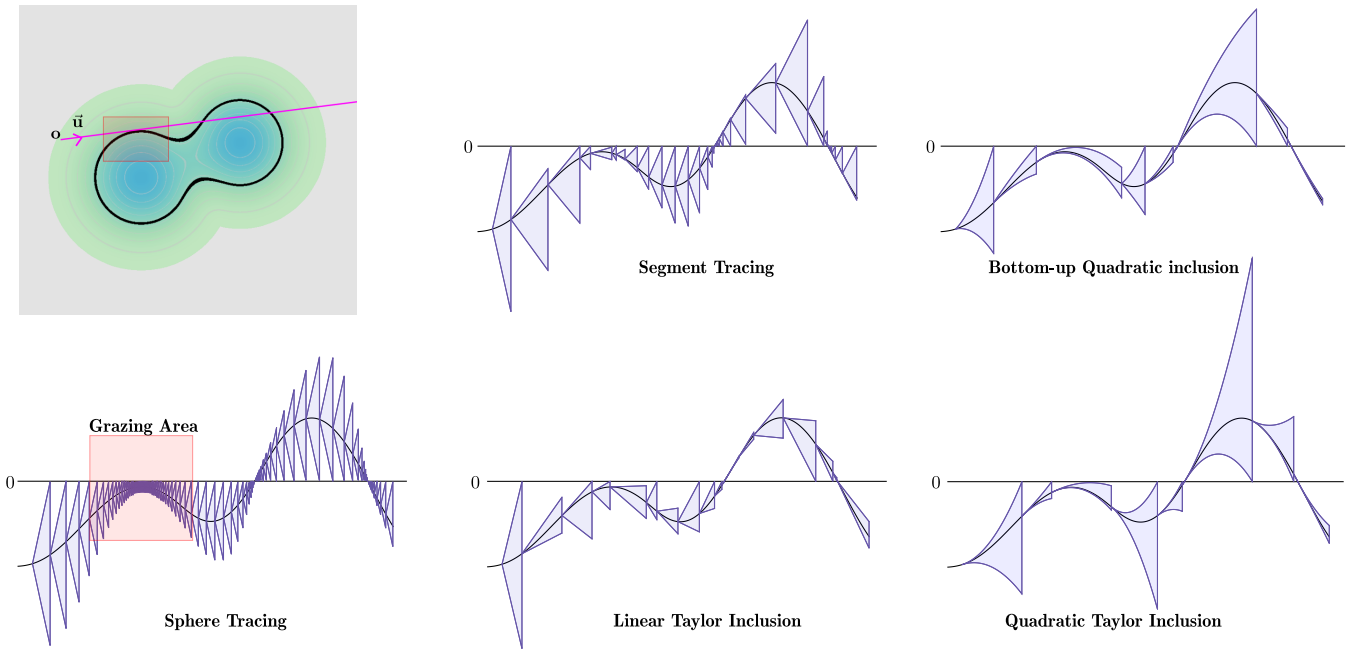

Figure 6: *Behaviour of Sphere Tracing [Har96], Segment Tracing [GGPP20], our asymmetrical linear tracing, and quadratic tracing with bottom-up quadratic inclusion function and quadratic Taylor inclusion function along a near grazing ray with transparency.*

Base primitive fields can then be combined using blending by summation for smooth blending, or maximum for sharp union.

We have also tested our method on Hermite radial basis functions [Wen05] which are defined as:

$$f(\mathbf{p}) = \sum_i \alpha_i \phi(\|\mathbf{p}-\mathbf{p}_i\|) + \beta_i^T \nabla\phi(\|\mathbf{p}-\mathbf{p}_i\|) \tag{9}$$

where ϕ is a smooth radial basis function and α_i and β_i are weights computed automatically to approximate a set of points p_i with prescribed normals.

Since our contribution focuses on 1-dimensional ray processing, for the sake of brevity, we denote $f \circ \delta(t)$ as $f(t)$ and $\nabla_{\mathbf{u}} f \circ \delta(t)$ as $f'(t)$ in the remainder of the paper.

5. Linear Tracing

We define *forward linear inclusion functions* as a pair of bounding lines that have the same value as the field function at the starting point of the queried interval.

Considered as a linear inclusion function (Figure 3), Sphere Tracing [Har96] bounds can be written as:

$$\begin{aligned} [f](t) &= f(t_0) \pm m(t-t_0), \quad \text{where} \\ m &= m_{global} = \max_{\mathbf{p}\in\mathbb{R}^3} (\|\nabla f(\mathbf{p})\|) = L. \end{aligned} \tag{10}$$

When the first point of a query interval is outside (or inside) of the implicit volume, computing the root of the upper (or lower) bound allows performing the same marching step as with the previous formulation (see Equation 4).

Similarly, we can formulate the local directional Lipschitz bounds of Segment Tracing [GGPP20] as a forward linear inclusion function with the following formula:

$$m = m_{local} = \max_{t\in[t_0,t_1]} \left(|f'(t)|\right) = \lambda, \quad t \in [t_0,t_1], \tag{11}$$

where λ is the local Lipschitz bound. In this form, we can see that such bounds can be easily extended to asymmetric bounds as follows:

$$\begin{aligned} f_-(t) &= f(t_0) + m_-(t-t_0) \\ f_+(t) &= f(t_0) + m_+(t-t_0), \qquad t \in [t_0,t_1]. \end{aligned} \tag{12}$$

We will study two alternative ways to compute $m_\pm$: first, by bounding the derivatives, and second, with bottom-up inclusions on the arithmetic operations.

5.1. Linear Taylor Inclusion Functions

The local Lipschitz constant of a differentiable function can be obtained by studying its first derivative:

$$\begin{aligned} m = \lambda &= f'(\xi), \\ \text{where } \xi &= \arg\max_{t\in[t_0,t_1]} \left(|f'(t)|\right) \end{aligned} \tag{13}$$

on an interval. Similarly, we can define our asymmetric local minimum and maximum bounds as:

$$\begin{aligned} m_\pm &= f'(\xi_\pm) \\ \text{where } \xi_- &= \arg\min_{t\in[t_0,t_1]} \left(f'(t)\right) \text{ for } f_- \\ \text{and } \xi_+ &= \arg\max_{t\in[t_0,t_1]} \left(f'(t)\right) \text{ for } f_+ \end{aligned} \tag{14}$$

or equivalently:

$$m_- = \min_{t\in[t_0,t_1]} \left(f'(t)\right)$$
$$m_+ = \max_{t\in[t_0,t_1]} \left(f'(t)\right). \tag{15}$$

Due to the close relation of this formulation to local Taylor expansion at $t = t_0$, we call this approach *Linear Taylor Inclusion* in the remainder of this paper. Depending on the field function, the exact values of $m_\pm$ can be derived analytically. In practice, we rely on the approach proposed in [GGPP20] except considering the sign of the directional derivative of the distance function in the bound derivation. Details are provided in A.

5.2. Bottom-up Linear Inclusion Functions

In cases where the bounds on the derivative are challenging to calculate – or not defined in the case of locally non-differentiable piecewise-defined functions (e.g., max operation) – we build the linear inclusion functions using the same strategy as previous self-validated numerical methods; bounds are computed bottom-up by using properties of the arithmetic operations constituting the field function defined along the ray.

Rules for summation, subtraction, and scalar multiplication of forward linear inclusion functions are trivially defined. For instance, for the summation, we have:

$$[f+g](t) = (f(t_0)+g(t_0)) + (m_{f,\pm}+m_{g,\pm})(t-t_0). \tag{16}$$

For non-linear operations such as multiplication, division and power operations, we introduce linear inclusion functions exploring the properties of each operation. A first common case is the squared distance to primitives (see Equation 6-8), which is a quadratic function for point primitives.

Convex/Concave functions All convex/concave functions of the ray parameter – including quadratic functions – can be handled in the same way. This special case can then be used as a base building block for operations such as multiplications.

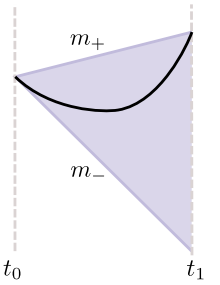

Figure 7: *Linear inclusion functions for a convex function.*

For a convex function, we define the forward linear inclusion functions using the derivative at the interval starting point for the lower slope m_- and the line interpolating the values at interval endpoints for the upper slope m_+ (Figure 7). This leads to the following formulas

$$m_-(t) = f'(t_0)$$
$$m_+(t) = \frac{f(t_1)-f(t_0)}{t_1-t_0}. \tag{17}$$

For the forward inclusion functions, i.e., when the value at the interval starting point is equal to the function value, this provides the tightest possible linear inclusion function for the interval. As the interval size gets smaller, the bounds get closer to the function, reducing the error. Inclusion functions for concave functions are obtained by simply swapping the definition of the upper and lower bound in Equation 17.

Multiplication Given two forward linear inclusion functions, all pairwise multiplications of linear functions defining the inclusion produce four quadratics: f_+g_+, f_+g_-, f_-g_+ and, f_-g_-. We can then find linear bounds on each of them separately. By construction, all quadratics share the same value at the beginning of the range. Then, the values of m_- and m_+ are simply defined as the highest and lowest slopes of all four bounds.

Maximum Building the maximum operation on two forward linear inclusion functions is similar. For the upper bound, the value at the beginning of the range t_0 is simply the maximum of the two input inclusion functions in t_0, and for the interval end-point, it is possible to use the maximum values of the two upper bounds. For the lower bound, the lower linear bound with the maximum initial value is chosen.

Once we define the inclusion functions for the basic operations, we can combine them to build the field functions. As in the previous inclusion methods, after combining the operations, the bounds may lose their tightest possible bound property, still providing the guarantee to contain the function reliably. To reduce the overall error in the inclusion functions, we introduce quadratic inclusion functions.

6. Quadratic Tracing

Quadratic inclusion functions can provide better approximation of the field functions as they have more degrees of freedom than linear ones. Therefore they can get closer to the field function along the ray and reduce the error; and converge faster to the root during ray marching. Moreover, due to the shape of the bounds, empty areas are skipped more quickly, which is especially advantageous for grazing rays.

Similar to the local asymmetric linear inclusion functions, we can define quadratic inclusion functions that are exact at the interval starting point as two quadratics bounds given in Horner notation:

$$[f](t) = f(t_0) + (t-t_0)\left(a_\pm + b_\pm(t-t_0)\right),\ t\in[t_0,t_1],\ a_\pm, b_\pm \in \mathbb{R}. \tag{18}$$

As for our linear inclusion functions, we propose two strategies to derive parameters $a_\pm$ and $b_\pm$: directly, by bounding the derivatives, and bottom-up construction.

6.1. Quadratic Taylor Inclusion Functions

Similar to linear inclusion functions, we can bound the higher degree derivatives to get an inclusion function.

If the function is twice differentiable, and the bounds on the second derivative are easily found, using the Taylor expansion at $t = t_0$, we can define the quadratic inclusion function as:

$$[f](t) = f(t_0) + (t-t_0)\left(f'(t_0) + \frac{1}{2}f''(\xi_\pm)(t-t_0)\right),$$
$$\text{where } \xi_- = \arg\min_{t\in[t_0,t_1]} \left(f''(t)\right) \text{ for } f_-$$
$$\text{and } \xi_+ = \arg\max_{t\in[t_0,t_1]} \left(f''(t)\right) \text{ for } f_+ \tag{19}$$

keeping both the function and derivative values fixed at t_0. Derivation for the point primitives are provided in A.

6.2. Bottom-up Quadratic Inclusion Functions

Similarly to the bottom-up linear inclusion functions, we define the quadratic inclusion functions for each arithmetic operation we need for the field function calculations along a given ray. Basic arithmetic operations are again trivial for summation, subtraction and scalar multiplication.

For non-linear operations, we rely heavily on local convexity/concavity. We present here the case of the multiplication. The general case for convex/concave functions is discussed in B and the maximum operation for two quadratic inclusion functions is discussed in C.

Quadratic Multiplication A quadratic forward inclusion function can be formulated as the integral of linear bounds of a function derivative around the query point:

$$[f](t) = f(t_0) + \int_{t_0}^{t} L_{\pm}(t).$$

We use this approach to bound the multiplication between two quadratic inclusion functions $[f]$ and $[g]$.

Multiplication between bounds $f_{\pm}$ and $g_{\pm}$ defines four quartic polynomials f_+g_+, f_+g_-, f_-g_+, and f_-g_-; and their derivatives are cubic polynomials, which are simpler to analyze since they can be split into convex and concave parts. For a given cubic $(f_{\pm}g_{\pm})'$, the inflection point defining the two parts can be calculated directly by solving a linear equation. Once we identify the convex and concave parts, we can linearly bound them as described in Section 5.2 and Figure 8 (left).

We can then generate a safe linear inclusion function using the highest and lowest possible points of each linear bound at the interval end-points – see Figure 8 (right) – which can then be integrated to obtain the quadratic inclusion function of the multiplication operator.

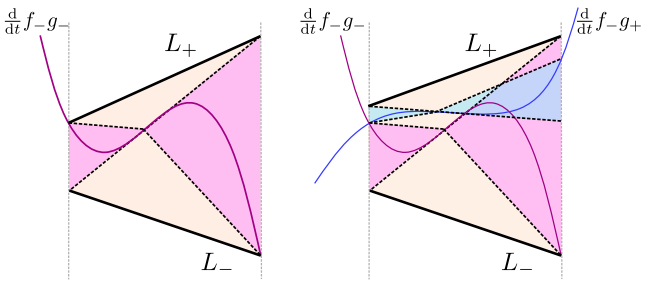

Figure 8: *Multiplication of two quadratics produces a quartic. We can bound its derivative using convex/concave regions (left). When multiplying two quadratic inclusion functions, four quartics are produced. They can be linearly bounded the same way and these bounds can be combined to get the overall bound for the multiplication operation. The illustration for two curves is shown on the right.*

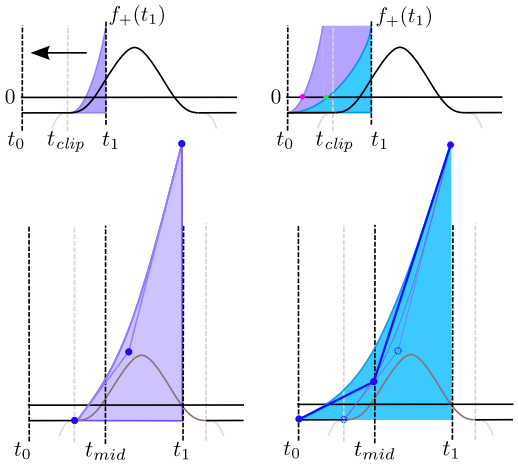

Figure 9: *Per-primitive bound extension for compact kernels: computing the bound on the intersection of the kernel support and ray interval, then extending it to the whole interval improves the step size (top). For this, we manipulate the control points defining the quadratic and move the initial query point to the interval start point. Carrying the middle control point as well guarantees that the resulting bounds are still valid after the extension.*

6.2.1. Per-Primitive Bounds

When using compactly supported kernels (Equation 7), the smoothing function is piecewise-defined. When the query interval includes the clipped part, we calculate the inclusion functions only on the unclipped part to get the appropriate bounds. Then, we can either extend this over the entire interval or, as illustrated in Figure 9 (top), modify this quadratic bound without recalculating any additional field function values, and achieve a larger iteration step size.

7. Results

We compare our methods with the methods described in Segment Tracing [GGPP20], Revised Affine Arithmetic [FPC10] and molecular rendering [Bru19] both in terms of the number of steps and processing time. We provide runtimes for both CPU and GPU computations. Our CPU runtimes are obtained with a C++ implementation; this configuration is used for all figures and tables if not specified otherwise. GPU runtimes for molecular rendering [Bru19] are obtained using a modified version of the implementation available at `https://github.com/sbruckner/dynamol`. GPU runtimes for transparency (see Figure 16) on an animated scene are obtained using a GLSL fragment shader. All CPU runtimes are obtained on a Intel® Core i7-10750H, and GPU runtimes are obtained on an NVIDIA GeForce® GTX 1650. We discuss opaque rendering in section 7.1 (including molecular rendering) and transparent rendering in section 7.2. We show our results on point skeletons with the two fall-off kernels described in Section 4.4, the Gaussian and the compact kernel.

For compact kernels (see Equation 7) with $n = 6$, linear and quadratic inclusion functions for the field function along the ray for a single blob are calculated using two subsequent multiplica-

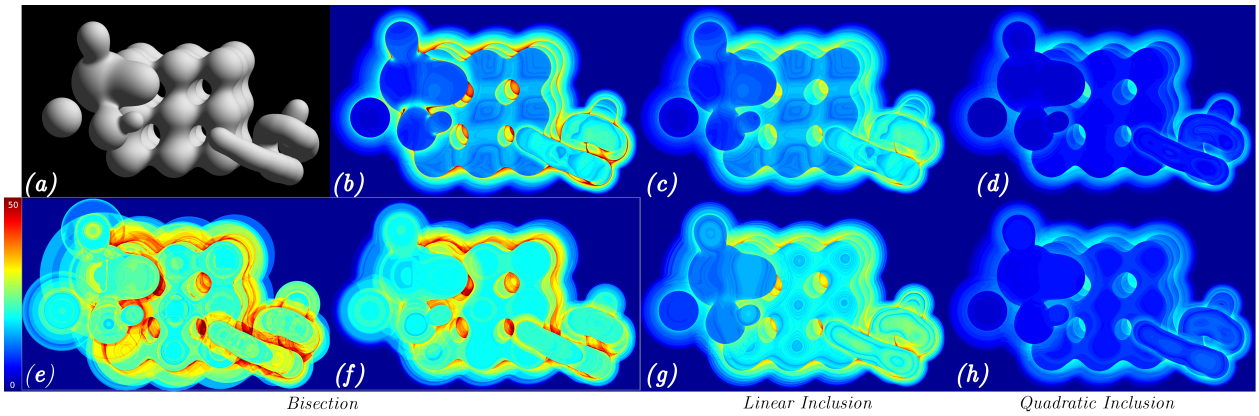

Figure 10: *An example scene (a) and comparison between the number of steps for (b) Segment Tracing [GGPP20], (c) linear Taylor inclusion function, (d) quadratic Taylor inclusion function, (e) Revised Affine Arithmetic [FPC10], (f) bottom-up linear inclusion function with bisection, (g) iterative bottom-up linear inclusion function and (h) bottom-up quadratic inclusion function. Improvement is seen in grazing rays and blending areas.*

Table 1: *Comparison of average calculation time and the number of steps per ray between different methods for the scene in Figure 10 for 11498388 rays around the bounding box.*

∼ 11M Bounding Box Rays	Avg. time per ray (μs)	Avg. # of steps	Gain (time)
Segment Tracing [GGPP20]	19.86	8.90	-
Revised Affine Arithmetic [FPC10]	41.56	15.40	-109.3%
Bottom-up Linear Bisection	27.18	12.55	-36.9%
Linear Taylor	19.98	8.11	-0.6%
Bottom-up Linear	20.91	9.62	-5.3%
Bottom-up Quadratic	14.22	5.56	28.4%
Quadratic Taylor	11.48	4.79	42.2%

tions (see Figure 11). The resulting bounds are then summed for blending. For the Gaussian kernels, we use the convexity property of the exponential function to construct the inclusion functions. The details are provided in B.

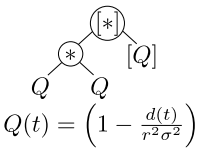

Figure 11: *Building the inclusion for one polynomial blob.*

We have defined the surfaces in Figures 1 and 17 with respectively a sharp union (max operation) and a summation blend to combine the base shapes.

$$f(t) = \max_n \left(\sum_m Blob_{mn}(t) \right). \quad (20)$$

For HRBF, we use $\phi(x) = x^3$, which results in a field that is neither polynomial nor a distance field (see Equation 9).

7.1. Opaque rendering

In Figure 10, we show how our proposed methods compare with the state-of-art methods and with each other. In (b) and (c), we show that using the asymmetrical version of the same kind of linear bounds (Section 5.1) provides noticeable improvement for blended areas and grazing rays (see Figure 12). Our linear bottom-up inclusion provides improvement compared to Revised Affine Arithmetic [FPC10] when roots are found using bisection with interval pruning (in (e) and (f)). Further improvement is shown when rays are processed iteratively using the same bottom-up linear inclusion functions (in (g)). Quadratic Taylor and bottom-up inclusion functions show the most improvement (in (d) and (h)).

Overall we demonstrate that Taylor/Lipschitz methods require fewer steps than bottom-up inclusion function calculations. The average number of field function evaluations and average time per ray is given in Table 1. Despite the increased number of computations per step, quadratic bottom-up inclusions for ray processing improve runtimes over linear inclusion functions. The greatest runtime improvements are observed for the Quadratic Taylor inclusion functions, which is expected as they require less computation per steps compared to bottom-up inclusion functions. Finally, our quadratic bounds provides the best benefits in challenging areas as shown in Figure 12.

HRBF Comparison with Revised Affine Arithmetic [FPC10] is given in Figure 13 and show a significant reduction in number of steps.

Molecules We have compared our method with the recent molecule rendering method [Bru19], where molecules are represented with density fields using Gaussian kernels (see Figure 15). Their method uses a highly specialized field inversion to transform the density field into a weak SDF. This allows efficient use of the sphere tracing algorithm to visualize scenes with a large number of molecules. The field inversion removes the inflection points that

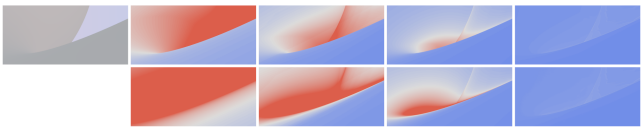

Figure 12: *Close-up in a challenging area. Top row use a minimal step size of 0.1 (used in all examples) and bottom use 0.01. From left to right: sphere, segment, linear and quadratic Taylor tracing - number of step is clamped to 75 for display. Quadratic tracing is less sensible to minimal step size.*

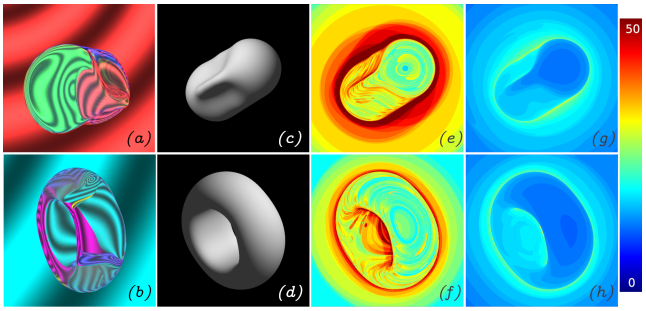

Figure 13: *Transparent (a, b) and opaque rendering (c, d) of HRBF for 17 (top) and 8 control points (down). Opaque rendering: comparison between number of steps between revised affine arithmetic (e, f) and bottom-up quadratic tracing (g, h) shows a large reduction.*

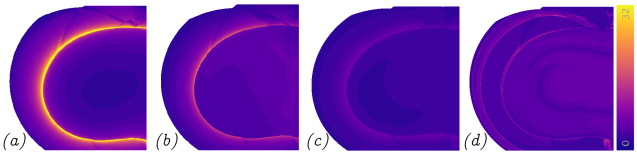

Figure 14: *Comparison between the number of steps for molecule rendering [Bru19] (a), Segment Tracing [GGPP20] (b), quadratic tracing with Taylor quadratic inclusion function (c) and bottom-up quadratic inclusion function with validity intervals (d). Visible discontinuities are due to Gaussian kernel clipping.*

Table 2: *Runtime comparison (frame per second) for molecule example. Increasing the maximum number of marching steps does not change the performance of our methods.*

Resolution 750x750	FPS		Gain
16791 atoms	Max # of steps = 32	Max # of steps = 1000	(time)
Molecule Rendering [Bru19]	22	19.7	-
Segment Tracing [GGPP20]	15	15	-46.7%
Quadratic bottom-up relaxed at t_0	25	25	12%
Quadratic bottom-up relaxed at maxima	23	23	4.4%
Quadratic bottom-up with validity	25	25	12%
Quadratic Taylor	28	28	21.4%

are due to the kernel shape (see Figure 2). This is especially useful for rays that are directed directly toward primitives center as visible in the close-up of Figure 14. In those areas, field inversion [Bru19] provides the smallest number of steps, however grazing rays are problematic. As our methods and Segment Tracing work directly with the density field, they provide a different trade-off: improving grazing rays at the cost of direct rays. Segment Tracing results in increased runtime (see comparisons in Table 2) but is less sensitive to the limit on number of steps used in the algorithm. Our experiments show that our quadratic inclusion functions are on par with the original technique – or faster depending the variant used – and do not suffer from increased limit on number of steps.

Our base strategy for quadratic inclusion function on exponential function uses validity intervals as explained in B. This cause many seemingly arbitrary jumps in the number of steps in neighboring regions as visible in Figure 14 (bottom right).

This would require further inspection regarding the effects of nearby molecules, which may have a very small influence on the final field value but impose unnecessary interval subdivisions. To overcome this, we also experimented with calculating bounds by relaxing some constraints (see Figure 21 in appendix) but did not observe major changes in terms of runtimes.

7.2. Transparency

Transparent rendering results are shown for the scenes in Figures 1 and 16 for blobby surfaces blended with summation, and max operation for Figure 1 (top). In Figure 16, the plane primitive is represented with the distance to the infinite plane combined with the same smoothing kernel as the blobs. Runtimes are provided in Tables 3, 4, 5.

Transparent rendering is challenging for marching algorithms like Sphere Tracing [Har96] which are optimized for converging to the first root along the ray with a symmetric constraint, therefore special care must be taken to avoid getting stuck at the first entry root.

Transparent rendering benefits greatly from asymmetric bounds. Small step sizes can thus be avoided while inside the object and still close to the recently recorded entry point. We saw a substantial reduction in the number of steps required to reach the surface as well as the time performance in our examples.

8. Conclusion

We have introduced an extension to the two families of ray-tracing methods, namely Lipschitz and interval methods, based on the new bounds tailored for iterative ray processing. We show that using asymmetric and quadratic bounds that are exact at the start of the query interval provides further significant improvement for ray tracing, especially when rendering transparent objects. Our bottom-up bound calculations show that we can achieve these improvements without directly calculating the bounds on the higher-order

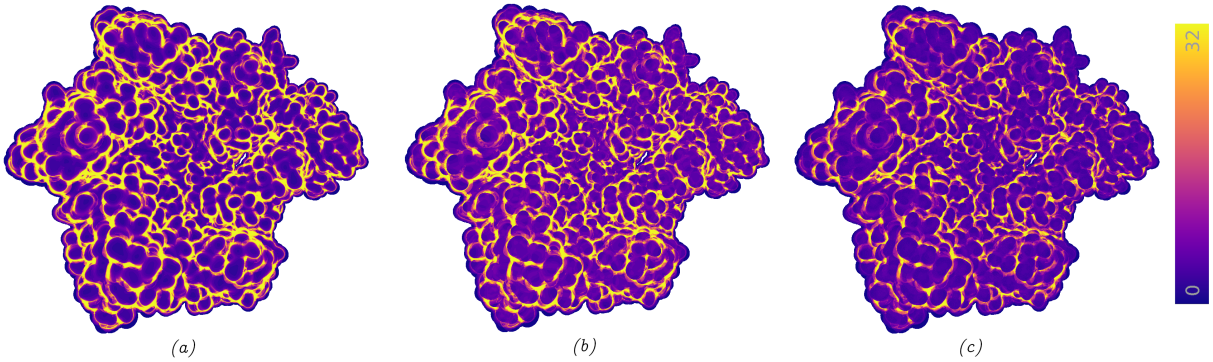

Figure 15: *Comparison between the number of steps for (a) molecule rendering [Bru19], (b) Segment Tracing [GGPP20] and (c) quadratic tracing with local Taylor quadratic inclusion function.*

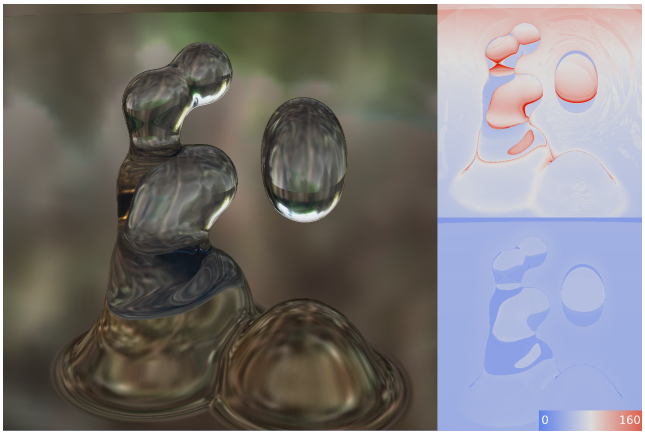

Figure 16: *Comparison between the number of steps for Segment Tracing [GGPP20] (top) and our quadratic bottom-up inclusion function for transparent rendering with reflection and refraction.*

Table 3: *Comparison of average calculation time and the number of steps per ray between different methods for the transparent scene in Figure 1 for 3305668 random rays around the bounding box. For this example, linear Taylor and quadratic Taylor inclusion functions are used as building blocks for the bottom-up inclusion function with the max operation.*

∼ 3M Bounding Box Rays	Avg. time per ray(μs)	Avg. # of steps	Gain (time)
Segment Tracing [GGPP20]	124.26	39.36	-
Linear Mixed	123.07	29.03	0.1 %
Bottom-up Quadratic	62.491	15.90	49.7 %
Quadratic Mixed	51.22	13.28	58.8 %

Table 4: *Comparison of average calculation time and the number of steps per ray between different methods for the transparent scene in Figure 17 for 3305668 random rays around the bounding box.*

∼ 3M Bounding Box Rays	Avg. time per ray(μs)	Avg. # of steps	Gain (time)
Segment Tracing [GGPP20]	136.06	40.97	-
Bottom-up Linear	125.33	28.95	7.89 %
Linear Taylor	107.03	28.95	21.34 %
Bottom-up Quadratic	59.39	15.01	56.35 %
Quadratic Taylor	44.26	12.64	67.47 %

Table 5: *GPU runtime comparison for the animated scene with refraction in Figure 16 average frame per second for 100 frames with bottom-up quadratic inclusion function.*

Resolution 1312 x 1017	FPS	ms	Gain (time)
Segment Tracing [GGPP20]	11	89	-
Bottom-up Quadratic	13.8	72.7	19 %
Quadratic Taylor	14.8	67.5	24 %

derivatives. While bottom-up bounds simplify the derivation of the new bounds by defining re-usable base building blocks, Taylor-based bounds are more efficient when available. We also show that the same framework can combine bottom-up and derivative-based bound computations.

Our study was focused on point-based primitives and HRBF, with summation and the set theoretic union operation defined as a maximum function. Extending the study to a more extensive set of primitives and operators is a natural follow-up. Similarly, providing tighter bounds or bounds that are less expensive to evaluate for the bottom-up approach would also be an interesting research direction.

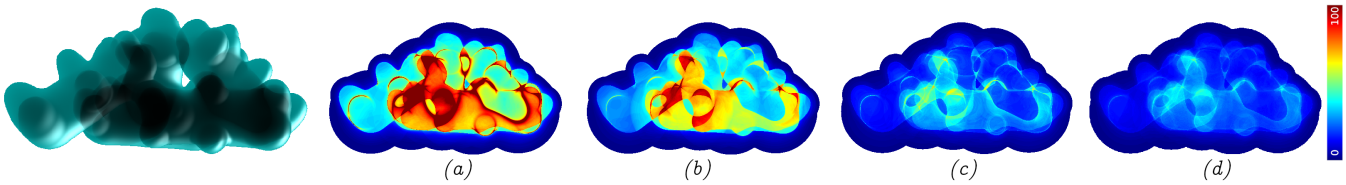

Figure 17: *Comparison between the number of steps for the transparent scene (left) combined with summation operation. (a) Segment Tracing [GGPP20], (b) Linear Taylor inclusion function, (c) Quadratic bottom-up inclusion function, (d) Quadratic Taylor inclusion function.*

Acknowledgements

We thank the authors of the papers [GGPP20; Bru19] for sharing their source codes. This work was supported by the ANR IMPRIMA(ANR-18-CE46-0004).

References

[AZ21] AYDINLILAR M., ZANNI C.: Fast ray tracing of scale-invariant integral surfaces. *Computer Graphics Forum 40*, 6 (2021), 117–134. doi:10.1111/cgf.14208. 2, 3

[BJ07] BARTOŇ M., JÜTTLER B.: Computing roots of polynomials by quadratic clipping. *Computer Aided Geometric Design 24*, 3 (2007), 125–141. doi:10.1016/j.cagd.2007.01.003. 2

[Bli82] BLINN J. F.: A generalization of algebraic surface drawing. *ACM Trans. Graph. 1*, 3 (jul 1982), 235–256. doi:10.1145/357306.357310. 2, 4

[Bru19] BRUCKNER S.: Dynamic visibility-driven molecular surfaces. *Computer Graphics Forum 38*, 2 (2019), 317–329. doi:10.1111/cgf.13640. 2, 3, 7, 8, 9, 10, 11

[BV18] BÁLINT C., VALASEK G.: Accelerating sphere tracing. In *Eurographics (Short Papers)* (2018), pp. 29–32. doi:10.2312/egs.20181037. 3

[CHMS00] CAPRANI O., HVIDEGAARD L., MORTENSEN M., SCHNEIDER T.: Robust and efficient ray intersection of implicit surfaces. *Reliable Computing 6*, 1 (Feb 2000), 9–21. doi:10.1023/A:1009921806032. 3

[CZ21] CHEN Z., ZHANG H.: Neural marching cubes. *ACM Trans. Graph. 40*, 6 (dec 2021). doi:10.1145/3478513.3480518. 1

[dFS04] DE FIGUEIREDO L. H., STOLFI J.: Affine arithmetic: Concepts and applications. *Numerical Algorithms 37*, 1-4 (dec 2004), 147–158. doi:10.1023/b:numa.0000049462.70970.b6. 1, 3

[FPC10] FRYAZINOV O., PASKO A., COMNINOS P.: Fast reliable interrogation of procedurally defined implicit surfaces using extended revised affine arithmetic. *Computers & Graphics 34*, 6 (2010), 708–718. doi:10.1016/j.cag.2010.07.003. 3, 4, 7, 8

[GDW*16] GRASBERGER H., DUPRAT J.-L., WYVILL B., LALONDE P., ROSSIGNAC J.: Efficient data-parallel tree-traversal for blobtrees. *Computer-Aided Design 70* (2016), 171–181. SPM 2015. doi:10.1016/j.cad.2015.06.013. 2

[GGPP20] GALIN E., GUÉRIN E., PARIS A., PEYTAVIE A.: Segment tracing using local lipschitz bounds. *Computer Graphics Forum 39*, 2 (2020), 545–554. doi:10.1111/cgf.13951. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

[GPP*10] GOURMEL O., PAJOT A., PAULIN M., BARTHE L., POULIN P.: Fitted bvh for fast raytracing of metaballs. *Computer Graphics Forum 29*, 2 (2010), 281–288. doi:10.1111/j.1467-8659.2009.01597.x. 2

[Har96] HART J. C.: Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 10 (Dec 1996), 527–545. doi:10.1007/s003710050084. 1, 2, 3, 5, 9

[JKDW01] JAULIN L., KIEFFER M., DIDRIT O., WALTER E.: *Applied Interval Analysis with Examples in Parameter and State Estimation, Robust Control and Robotics*. 08 2001. 3, 4

[JLSW02] JU T., LOSASSO F., SCHAEFER S., WARREN J.: Dual contouring of hermite data. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2002), SIGGRAPH '02, ACM, p. 339–346. doi:10.1145/566570.566586. 1

[KB89] KALRA D., BARR A. H.: Guaranteed ray intersections with implicit surfaces. *SIGGRAPH Comput. Graph. 23*, 3 (July 1989), 297–306. doi:10.1145/74334.74364. 2

[Kee20] KEETER M. J.: Massively parallel rendering of complex closed-form implicit surfaces. *ACM Trans. Graph. 39*, 4 (jul 2020). doi:10.1145/3386569.3392429. 3

[KHK*09] KNOLL A., HIJAZI Y., KENSLER A., SCHOTT M., HANSEN C., HAGEN H.: Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic. *Computer Graphics Forum* (2009). doi:10.1111/j.1467-8659.2008.01189.x. 3

[KSK*14] KEINERT B., SCHÄFER H., KORNDÖRFER J., GANSE U., STAMMINGER M.: Enhanced Sphere Tracing. In *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference* (2014). doi:10.2312/stag.20141233. 3

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1987), SIGGRAPH '87, ACM, p. 163–169. doi:10.1145/37401.37422. 1

[LZLW09] LIU L., ZHANG L., LIN B., WANG G.: Fast approach for computing roots of polynomials using cubic clipping. *Computer Aided Geometric Design 26*, 5 (2009), 547–559. doi:10.1016/j.cagd.2009.02.003. 2

[Mit90] MITCHELL D. P.: Robust ray intersection with interval arithmetic. In *Proceedings on Graphics Interface '90* (CAN, 1990), Canadian Information Processing Society, p. 68–74. doi:10.20380/GI1990.08. 1, 3

[Moo66] MOORE R.: *Interval Analysis*. Prentice-Hall series in automatic computation. Prentice-Hall, 1966. 3

[MT06] MESSINE F., TOUHAMI A.: A general reliable quadratic form: An extension of affine arithmetic. *Reliable Computing 12* (06 2006), 171–192. doi:10.1007/s11155-006-7217-4. 3

[Neu02] NEUMAIER A.: Taylor forms - use and limits. *Reliable Computing 2003* (2002), 9–43. doi:10.1023/A:1023061927787. 3

[NN94] NISHITA T., NAKAMAE E.: A method for displaying metaballs by using bezier clipping. *Comput. Graph. Forum 13* (08 1994), 271–280. doi:10.1111/1467-8659.1330271. 2

[PASS95] PASKO A., ADZHIEV V., SOURIN A., SAVCHENKO V.: Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer 11*, 8 (1995), 429–446. doi:10.1007/BF02464333. 1

[Rat97] RATZ D.: *An Optimized Interval Slope Arithmetic and its Application*. Institut für Angewandte Mathematik, 1997. 3

[She99a] SHERSTYUK A.: Fast ray tracing of implicit surfaces. *Comput. Graph. Forum 18* (1999), 139–147. doi:10.1111/1467-8659.00364. 2

[She99b] SHERSTYUK A.: Kernel functions in convolution surfaces: A comparative analysis. *The Visual Computer 15* (02 1999). doi:10.1007/s003710050170. 4

[SJ22] SHARP N., JACOBSON A.: Spelunking the deep: Guaranteed queries on general neural implicit surfaces via range analysis. *ACM Trans. Graph. 41*, 4 (jul 2022). doi:10.1145/3528223.3530155. 3

[SJNJ19] SEYB D., JACOBSON A., NOWROUZEZAHRAI D., JAROSZ W.: Non-linear sphere tracing for rendering deformed signed distance fields. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia) 38*, 6 (Nov. 2019). doi:10.1145/3355089.3356502. 3

[TLY*21] TAKIKAWA T., LITALIEN J., YIN K., KREIS K., LOOP C., NOWROUZEZAHRAI D., JACOBSON A., MCGUIRE M., FIDLER S.: Neural geometric level of detail: Real-time rendering with implicit 3d shapes. In *CVPR* (jun 2021), IEEE Computer Society, pp. 11353–11362. doi:10.1109/CVPR46437.2021.01120. 3

[TPFA21] TERESHIN A., PASKO A., FRYAZINOV O., ADZHIEV V.: Hybrid function representation for heterogeneous objects. *Graphical Models 114* (2021), 101098. doi:10.1016/j.gmod.2021.101098. 1

[VSJ22] VICINI D., SPEIERER S., JAKOB W.: Differentiable signed distance function rendering. *Transactions on Graphics (Proceedings of SIGGRAPH) 41*, 4 (July 2022), 125:1–125:18. doi:10.1145/3528223.3530139. 3

[Wen05] WENDLAND H.: *Scattered data approximation*. Cambridge University Press, 2005. 5

[WGG99] WYVILL B., GUY A., GALIN E.: Extending the csg tree. warping, blending and boolean operations in an implicit surface modeling system. *Comput. Graph. Forum 18* (06 1999), 149–158. doi:10.1111/1467-8659.00365. 2

[WMW86] WYVILL G., MCPHEETERS C., WYVILL B.: Data structure for soft objects. *The Visual Computer - VC 2* (08 1986), 227–234. doi:10.1007/BF01900346. 1, 2

[WT90] WYVILL G., TROTMAN A.: Ray-tracing soft objects. In *CG International '90* (1990), Springer Japan, pp. 469–476. doi:10.1007/978-4-431-68123-6_27. 2

[WW89] WYVILL B., WYVILL G.: Field functions for implicit surfaces. *The Visual Computer 5* (01 1989), 75–82. doi:10.1007/BF01901483. 4

Appendix A: Linear and Quadratic Taylor Bounds

To calculate the local Taylor inclusion functions for point primitives, we analyze the first and second derivatives for Equation 8. For linear Taylor inclusion functions, as described for segment tracing [GGPP20], the linear bounds are calculated by bounding the gradient of the distance and derivative of the kernel along the ray separately.

$$\begin{aligned} f(t) &= k \circ d \circ \delta(t) \\ f'(t) &= \left(k' \circ d \circ \delta(t)\right)\left(\nabla_{\mathbf{u}} d \circ \delta(t)\right) \end{aligned} \tag{21}$$

Since the gradient of the distance function along the ray is monotonous (Figure 18 left), the maximum and minimum values occur at the interval end points. For the derivative of the kernel as a function of the distance d (Figure 18 right), the maximum and the minimum are reached either at the interval end points or at the extrema. For calculating the bounds on $f'(t)$, we analyze the product of these extremal points. This way, these bounds can be extended into more general primitives only by changing the distance function.

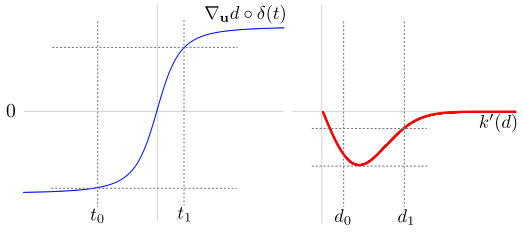

Figure 18: *Derivative analysis for the point primitive (21). By analyzing the maximum and minimum values of the multiplication of the two values in a given interval, the minimum and maximum bounds on the derivative of the field equation can be calculated.*

The bounds for second derivatives are calculated directly by analyzing the roots of the third derivatives to localize the extrema in a given interval. Examining the existence of extrema in a given interval, it is possible to locate the minimum and maximum values of the second derivatives of the field function at either interval endpoints or at the extrema. (Figure 19). For the Gaussian case, for a general quadratic given in the form:

$$ax^2+bx+c \quad a,b,c \in \mathbb{R}, \tag{22}$$

its third derivative can be found as:

$$\frac{d^3}{dx^3}\left(e^{ax^2+bx+c}\right) = (2ax+b)(4a^2x^2+4abx+b^2+6a)e^{ax^2+bx+c} \tag{23}$$

and the three roots that correspond to the maximum and minimum values of the second derivative can be calculated directly.

Appendix B: Quadratic inclusion function for monotonous convex/concave functions

We present here the computation of quadratic inclusion function for a monotonously decreasing function f and a convex input

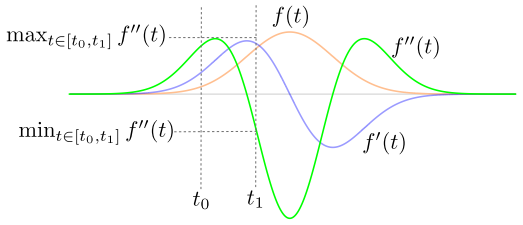

Figure 19: *Equation 8 with Gaussian kernel (orange), first derivative (blue), second derivative (green). The behaviour of extrema is the same for the compact kernel (7) within the support.*

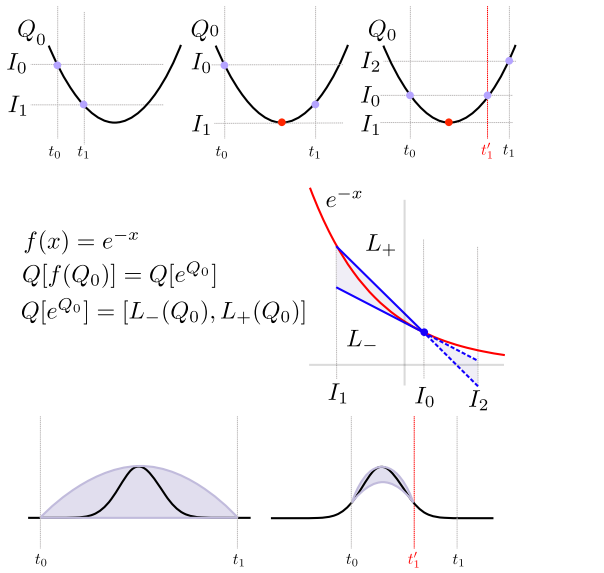

Figure 20: *Quadratic inclusion function for a monotonous convex function with a validity interval. The terms I_0 and I_1 denote the range of values the quadratic can reach in the given interval. If this prevents keeping the value at the interval start fixed, the interval is shortened.*

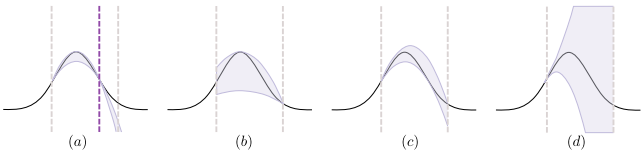

Figure 21: *Different approaches we have used for calculating the bound on the Gaussian kernel. (a) Validity intervals, (b) relaxing at the query point, (c) relaxing the maxima, (d) quadratic Taylor inclusion function.*

quadratic Q as illustrated in Figure 20 with an exponential function ($f(x) = e^{-x}$). We rely on the fact that the composition of a quadratic and linear function is a quadratic function. We therefore build a linear inclusion function for the function f on a range of interest defined by the input quadratic Q (see Figure 20). Let's first assume that the value of the quadratic Q at the interval start point is an extrema on the interval $[t_0, t_1]$. Given the extrema I_0 and I_1 of the quadratic, since the exponential f is convex, linear bounds can be constructed for f on the interval $[I_0, I_1]$ as described in Section 5.2. These linear bounds L_- and L_+ decrease monotonously as the exponential e^{-x} they are bounding. The composition of those bounds with the quadratic Q gives an upper and lower bounds on $[t_0, t_1]$.

This strategy cannot be applied when the start point is not an extrema of Q as the linear bound of f would not be exact at the interval starting point. In this case, the interval is shortened up to t'_1 the symmetric point where $Q(t_0) = Q(t'_1)$ as shown in Figure 20 (top right). This strategy can cause smaller steps as seen in molecule rendering (see Section 7.1).

We explored alternative strategies as shown in Figure 21: we either relax the value at the global extrema of Q or at the interval starting point. In our experiments, those strategies provide similar runtimes (Table 2).

When we start with a quadratic range, instead of a single quadratic, we simply perform the above mentioned steps separately for each quadratic and use their upper and lower bounds respectively.

Appendix C: Quadratic inclusion function for maximum

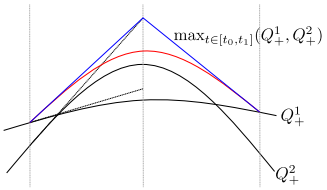

Figure 22: *Maximum of two upper quadratic bounds*

Maximum of two quadratic inclusion functions is calculated by finding the maximum of two upper and lower quadratic bounds separately. A sketch is given for a maximum bound in Figure 22. In this case, we build the quadratic with a control polygon consisting of three points: an initial fixed point matching the maximal value of the two input bounds at the begining of the interval, a middle point computed from the highest extent of the two derivatives calculated at the start point, and the maximum value at the interval end point. For the lower bounds, we simply pick the lower bound with the highest value at the interval starting point.