Spelunking the Deep: Guaranteed Queries on General Neural Implicit Surfaces via Range Analysis

NICHOLAS SHARP, University of Toronto, Canada
ALEC JACOBSON, University of Toronto, Adobe Research, Canada

Neural implicit representations, which encode a surface as the level set of a neural network applied to spatial coordinates, have proven to be remarkably effective for optimizing, compressing, and generating 3D geometry. Although these representations are easy to fit, it is not clear how best to evaluate geometric queries on the shape, such as intersecting against a ray or finding a closest point. The predominant approach is to encourage the network to have a signed distance property. However, this property typically holds only approximately, leading to robustness issues, and holds only at the conclusion of training, inhibiting the use of queries in loss functions. Instead, this work presents a new approach to perform queries directly on general neural implicit functions for a wide range of existing architectures. Our key tool is the application of range analysis to neural networks, using automatic arithmetic rules to bound the output of a network over a region; we conduct a study of range analysis on neural networks, and identify variants of affine arithmetic which are highly effective. We use the resulting bounds to develop geometric queries including ray casting, intersection testing, constructing spatial hierarchies, fast mesh extraction, closest-point evaluation, evaluating bulk properties, and more. Our queries can be efficiently evaluated on GPUs, and offer concrete accuracy guarantees even on randomly-initialized networks, enabling their use in training objectives and beyond. We also show a preliminary application to inverse rendering.

CCS Concepts: • Computing methodologies → Shape analysis; Shape representations; • Mathematics of computing → Interval arithmetic.

Additional Key Words and Phrases: implicit surfaces, neural networks, range analysis, geometry processing

1 INTRODUCTION

Representing shapes presents a fundamental dilemma across visual and scientific computing: point clouds and voxel grids are easy to process efficiently, but lack explicit connectivity information; meshes offer a concise and precise description of a surface, but may require difficult unstructured computation, etc. Recently, neural implicit representations have emerged as a promising alternative for a variety of important tasks—the basic idea is to encode a surface as a level set of a neural network applied to spatial coordinates. These neural implicit surfaces inherit many of the strengths which have made neural networks ubiquitous across visual computing, including effective gradient-based optimization, integration with data-driven priors and objectives, and straightforward parallelization on modern hardware.

However, there is a price to pay in return for these strong properties: there is no clear strategy for evaluating even the most basic geometric queries against a neural implicit surface, such as intersecting a ray with the surface, or finding a closest point. It would seem that the only thing we can do with such a function is to sample it at a point. In a sense, the powerful generality of neural networks is exactly what makes them difficult to query—because they can approximate arbitrary functions with adaptive spatial resolution, it is very difficult to characterize the geometry of their level sets.

One popular recourse is to attempt to fit implicit functions which not only encode a surface via their zero level set, but furthermore have a signed-distance function (SDF) property away from the level set: the magnitude of the function gives the distance to the surface. Although exact SDFs are well-suited for many queries in geometry processing, approximate neural SDFs leave much to be desired. First, such networks are only approximately SDFs, and may overestimate the distance to the surface, causing queries to fail unpredictably. More importantly, the SDF property only applies after a network has been successfully fitted; thus we cannot make use of geometric queries in the early stages of training, e.g., to define geometric loss functions. Even more broadly, relaxing the expectation that a network fits an SDF opens up a broader class of neural network formulations and objectives, such as those based on occupancy (e.g., as in Section 5).

Authors' addresses: Nicholas Sharp, University of Toronto, Canada, nsharp@cs.toronto.edu; Alec Jacobson, University of Toronto, Adobe Research, Canada, jacobson@cs.toronto.edu.
This work develops and studies a technique for performing queries on general neural implicit surfaces—including not only SDFs, but also other functions which lack any special properties away from the zero level set. Importantly, we do not define any new architectures or loss functions, but rather show how to perform queries on a broad class of existing networks, making our method immediately compatible with a wide range of past and future work on neural implicit formulations. The result is a collection of subroutines for performing geometric queries on neural implicit surfaces, resolving a key weakness of the formulation and enabling promising new avenues of research.

To enable these queries, we leverage range analysis, a class of automatic arithmetic techniques for computing bounds on the range of a function over a specified input domain. These techniques were first popularized to bound the error incurred by floating point arithmetic, but can be applied more generally over any domain. However, there exists a wide variety of range analysis methods, and these have not been previously studied in the context of neural implicit shapes. In fact, we observe that many of these variants are entirely ineffective in this context, and a key component of this work is an investigation and empirical benchmark to identify range analysis schemes which are both efficient and effective for general neural implicit surface queries (Section 3).

With range analysis of neural networks in hand as our primary tool, we develop a suite of geometric queries including ray casting, empty sphere queries, fast hierarchical surface sampling and mesh extraction, closest point queries, intersection tests, and more (Section 4). Many of these queries were not previously possible on general neural implicit surfaces, or could be evaluated only by dense brute-force sampling. Our queries are guaranteed in the sense that they have bounded error with respect to the implicit surface regardless of the nature of the underlying neural network, i.e. they apply even on a randomly initialized networks. We demonstrate the potential of these queries applied to a wide variety of problems in computer graphics, vision, and simulation.

2 BACKGROUND AND RELATED WORK

The general literature on neural fields has rapidly exploded beyond the scope of this section, we point to the thorough survey of Xie et al. [2022] as a general introduction. In this work, we particularly consider queries on implicit solid surfaces, as opposed to volumetric, partial occupancy, or participating media queries. In particular, casting rays against solid surfaces has been very widely studied in computer graphics, and will serve as a proxy for many concerns that arise in the other queries we consider.

Neural Implicit Surfaces. Neural implicit surfaces define the boundary $S$ of a solid 3D shape as the zero level set of a Multi-Layer Perceptron (MLP) with internal parameters $\theta$. The function $f_\theta$ takes as input a 3D position $x \in \mathbb{R}^3$, and possibly additional latent inputs, and outputs a scalar value:

$$S \triangleq \{ x \in \mathbb{R}^3 \mid f_\theta(x) = 0 \}. \quad (1)$$

Neural implicit surfaces are immediately attractive because the sign of their forward evaluation classifies $x$ as inside or outside the solid bounded by $S$, typically assumed negative inside by convention. Networks which instead model an occupancy or density (e.g., [Chen and Zhang 2019; Mescheder et al. 2019]), such as the geometric component of recently-popular NeRF models [Mildenhall et al. 2020], can be viewed as an implicit surface by selecting an appropriate level set; this connection was explored Yariv et al. [2021]. More broadly, neural implicit surfaces are a special case of the larger family of implicit surfaces defined by any arbitrary function $f(x)$, which have a vast history in computer graphics (see, e.g., [Bloomenthal et al. 1997; Menon 1996]). Classic implicit surfaces are constructed via trees of constructive solid geometry operations with analytic functions at leaf nodes (planes, spheres, cones, etc.) [Ricci 1973], or by crafting smooth blending operations on radial basis functions (e.g., metaballs, blobs) [Blinn 1982; Wyvill et al. 1986].

Fig. 2. Neural implicit surface representations offer a variety of exciting properties, including efficient compression via gradient-based optimization (left), and interpolation through latent parameter spaces (right). Both examples are rendered in our ray casting framework.

Fig. 3. Neural implicit functions may fit a metric signed distance function (left), merely classify as an occupancy function (right), or use other formulations. The queries developed in this work apply in all cases. For occupancy, we plot the “logits” $y$ such that tanh($y$) gives occupancy.
Signed Distance Functions. An important special case of implicit surfaces are signed distance functions (SDFs), which add the requirement that the magnitude $|\text{SDF}(x)|$ is the distance to the closest point on the levelset $S$. The SDF property implies that no point on the surface is within a distance $|\text{SDF}(x)|$ in any direction. This observation forms the essence of the sphere tracing algorithm (Figure 4) for rendering implicit surfaces by casting rays [Bálint and Valasek 2018; Hart 1996; Keinert et al. 2013; Reimer et al. 2011].

Often, a function need not satisfy the SDF property exactly, but merely be a weak SDF, such that $|f(x)| < |\text{SDF}(x)|$. A weak SDF still guarantees that no point on the surface is within a distance $|f(x)|$, which is sufficient for algorithms like sphere tracing to be correct, though they must take smaller steps. Any smooth, bounded implicit function can be transformed to a weak SDF after scaling by its Lipschitz constant $L$, though this is not necessarily productive in practice (see Section 4.11). Representing arbitrary surfaces with exact SDFs is often difficult or unwieldy, but weak SDFs have been constructed to model fascinatingly complex surfaces (e.g. by Quilez [2008]). Space-warped SDFs may no longer maintain a tight SDF, but Seyb et al. [2019] show that the inverse-warped function can afford sphere-tracing along curved rays in the unwarped domain.

Stated in the language of MLPs, if $f_0$ is an SDF, or some appropriate Lipschitz constant $L$ is known a priori, then a single forward evaluation would simultaneously reveal the implicit’s value at the current point along a ray and the safe stepping distance along a cast ray (see Figure 4). However, a generic neural implicit $f_0$ will not automatically encode an SDF, nor have a known or small Lipschitz constant $L$. The most widespread remedy in practice is to supervise training of the neural implicit function with precomputed SDF samples from known shapes. If a network is well-trained to fit an SDF [Davies et al. 2020; Park et al. 2019], then sphere-tracing and other queries may be applied, though still with some risk of overzealous step scaling resulting in missed ray hits.

Another approach is by changing an MLP architecture to have a deterministic global Lipschitz constant [Yariv et al. 2021]. However, this may degrade surface fidelity, and moreover the global constant may be very high despite being small in a region of interest (e.g. near $S$). Computing precise local Lipschitz constants for common MLP architectures is NP-Hard [Jordan and Dimakis 2020; Virmaux and Scaman 2018].

Yet another route is to incorporate loss functions encouraging Eikonality $\|\nabla \text{MLP}_\theta\| \approx 1$ [Atzmon and Lipman 2020a,b; Davies et al. 2020; Gropp et al. 2020] or bounded Lipschitz constant [Elsner et al. 2021]. This approach may also affect surface fidelity and—even if successful—may only be true when training has completed, precluding safe sphere-tracing type queries during training.

Querying General Implicit Surfaces. What if we do not want to change the architecture of our MLP, or its training loss, to accommodate queries? What if we are handed an MLP which is not SDF-like? For ray-casting, we could march with very small fixed steps [Perlin and Hoffert 1989], but small steps are excessively expensive while large steps will cause rays to erroneously miss the surface (Figure 6). We could contour the level set to a triangle mesh [Genova et al. 2020; Park et al. 2019] via a method such as marching cubes [Lorensen and Cline 1987], but this introduces discretization error and aliasing, and also complicates differentiability [Liao et al. 2018; Remelli et al. 2020]. For ReLU activations, the level-set will be polyhedral and could be theoretically triangulated exactly, at significant cost [Lei and Jia 2020]. Even beyond ray-casting, we would seek many geometric queries to support the burgeoning geometry processing of neural implicit surfaces [Yang et al. 2021; Yifan et al. 2022].

Instead, we look beyond the neural world toward past efforts of ray casting with a larger class of arbitrary implicit functions. A major theme of these works—dating at least to Duff [1992]—is to apply interval arithmetic or more generally range analysis with its numerous variants [Rump and Kashiyagi 2015]. Interval arithmetic is a code transformation technique to compute strict upper and lower bounds for a composition of simple functions (see, e.g., [Alefeld and Mayer 2000; Moore et al. 2009; Stolfi and De Figueiredo 1997] and our more detailed discussion in Section 3).

Applied to ray casting, range analysis can be employed to safely increase or decrease the step size of a ray or bundle of rays [De Cusatis et al. 1999; Fryazinov et al. 2010; Galin et al. 2020; Gamito and Madock 2007; Heidrich and Seidel 1998; Heidrich et al. 1998; Keeter 2020; Knoll et al. 2007, 2009; Mitchell 1990; Thonat et al. 2021], or for other queries, such as finding closest points [Chan 2008]. Despite this recurring interest, range analysis has not yet been studied in the context of neural implicit to the best of our knowledge. In this paper, we demonstrate that carefully-selected variants of range analysis offer a highly effective strategy for analyzing general neural implicit surfaces, and show how a wide variety of geometric queries can be built upon it.

3 RANGE ANALYSIS OF NEURAL IMPLICIT SHAPES

We propose to apply range analysis to neural implicit shape functions: these techniques take bounds on the input to a function, then apply automatic arithmetic rules to propagate bounds for each intermediate operation in a computation, and ultimately produce a bound on the function output [Stolfi and De Figueiredo 1997]. In our setting, range analysis can concretely bound the implicit function away from 0 in a region, classifying the region as strictly inside or outside the shape. This will be our foothold from which to build higher-level geometric queries (Section 4).

Range analysis has been widely studied within numerical computing, and particularly in the context of computer graphics (see Section 2). In the neural network literature, it has been leveraged...
for robustness and verification [Adam et al. 2016; Betancourt and Muhanna 2021; Dai et al. 2021; Gowal et al. 2018; Mirman et al. 2021; Sahoo et al. 2015], and even in the 3D setting by Proszewska et al. [2021], albeit in a voxel context. However, the combination of these lines of research has not previously been realized by applying range analysis to neural implicit surfaces—we find this application to be very fruitful, but not without challenges. For example, we observe that interval arithmetic, the most basic and common range analysis, is largely ineffective when applied to our neural implicit surfaces. For this reason, in this section we review core ideas in several forms of range analysis, and discuss particular issues which arise in application to MLPs, as well as performing an empirical study. Ultimately, in Section 3.4 we give concrete recommendations for variants of affine arithmetic which are highly effective for range analysis of neural implicit surfaces.

3.1 Interval Arithmetic

Interval arithmetic [Young 1931] is a technique for automatically computing bounds on the value of a function over a domain. The basic idea is to replace each scalar quantity \( x \) in a computation with a pair of bounds \([x_-, x_+]\). Given bounds on the input \( x \) to a function \( y = f(x) \), arithmetic rules are applied to propagate the bounds forward through each elementary operation, eventually yielding bounds on the output \([y_-, y_+]\) such that \( f(x) \in [y_-, y_+] \) \( \forall x \in [x_-, x_+] \). For example, elementary rules for addition, scalar multiplication, and the exponential are given by:

\[
\begin{align*}
[x_-, x_+] + [y_-, y_+] &= [x_- + y_-, x_+ + y_+] \\
ax &= [\min(ax_-, ax_+), \max(ax_-, ax_+)] \\
\exp([x_-, x_+]) &= [\exp(x_-), \exp(x_+)].
\end{align*}
\]

In general, these rules can derived as needed, or looked up in a standard reference (e.g. [Stolli and De Figueiredo 1997]). This technique extends directly to vector-valued quantities by tracking intervals for each component, and there are no restrictions on \( \epsilon \) the smoothness of \( f \), so long as interval bounds can be derived for each constitutive operation.

These interval arithmetic rules already allow us to compute bounds on the output of an MLP. However, it turns out that interval arithmetic alone is not a practical tool in our setting.

The Dependency Problem. The main downside of interval arithmetic is that the computed bounds may be extremely pessimistic. As an example, consider the simple operation \( y = 2x - x \), evaluated on the range \( x \in [-1, 1] \). Clearly the actual bound on \( y \) is \([-1, 1] \), but applying the rules above yields a looser bound of \([-1, 2]\) even in this simple example. This issue is that the same interval-bounded quantity appears multiple times in the expression, and should cancel out, but the rules naively treat all interval quantities as being distinct. In fact, the tightness of the bounds even depends on how the function is written algebraically—unfortunate property in contrast to other automatic transformations such as automatic differentiation. This effect is particularly problematic in MLPs, where linear layers \( y \leftarrow Ax \) involve a great deal of cancellation which is not captured by basic interval arithmetic, leading to extremely pessimistic bounds. For this reason, we turn to an extension of interval arithmetic which tracks additional data to compute tighter bounds.

3.2 Affine Arithmetic

Affine arithmetic [Comba and Stolfi 1993] generalizes interval arithmetic by tracking a collection of affine symbols, and offers the advantageous property that bounds are preserved exactly under affine operations. In affine arithmetic, each scalar value \( x \) is expanded to a base \( x_0 \) and a collection of affine coefficients \( \{x_1, \ldots, x_N\} \)

\[
\hat{x} = x_0 + \sum_{i=1}^{N} x_i \epsilon_i,
\]

where each \( \epsilon_i \) is a “noise symbol” representing some variation or uncertainty in the value of \( \hat{x} \). Crucially, this representation distinguishes distinct sources of variation: if \( \hat{x} \) and \( \hat{y} \) both vary due to some \( \epsilon_i \), we know that these are the same, correlated uncertainties, and can e.g. allow them to cancel when subtracting \( x - y \).

Notice that for any uncertain quantity represented in affine form, we can easily read off bounds on the value of that quantity as

\[
\text{range}(\hat{x}) = [x_0 - r, x_0 + r], \quad r = \sum_{i=1}^{N} |x_i|.
\]

Much like interval bounds, affine bounds can be propagated by automatic rules as a computation proceeds. For instance, addition amounts to simply summing the base and coefficients

\[
\hat{x} + \hat{y} = x_0 + \sum_{i=1}^{N} x_i \epsilon_i + y_0 + \sum_{i=1}^{N} y_i \epsilon_i = (x_0 + y_0) + \sum_{i=1}^{N} (x_i + y_i) \epsilon_i
\]

and likewise multiplication by a constant \( a \) is given by

\[
a\hat{x} = ax_0 + \sum_{i=1}^{N} ax_i \epsilon_i.
\]

Both operations are exact; they do not introduce new uncertainty.

For nonlinear functions \( \hat{y} \leftarrow f(\hat{x}) \) such as \( \exp, \tan, \text{etc.} \), there is a straightforward recipe to propagate affine bounds by leveraging a linear approximation \( f(x) \approx \tilde{f}(x) := ax + b \) on range(\( \hat{x} \)). Letting \( y \) be the maximum error of this approximation \( y = \max_{x} \epsilon \text{range}(x) | f(x) - \tilde{f}(x)| \), then affine bounds can be propagated through \( f \) according to

\[
\tilde{y} = f(\hat{x}) = \tilde{f}(x) = ax + b + \sum_{i=1}^{N} ax_i \epsilon_i + y_{EN} \epsilon_{N+1},
\]

where \( \epsilon_{N+1} \) introduces \( y \) as a new, additional affine coefficient which models the nonlinear variation of \( f(x) \) and is carried forward in subsequent computation. For a particular nonlinear \( f(x) \), we must
then define $\alpha, \beta, \gamma$ for any input domain $\hat{x}$. This approximation can be manually derived as needed, or looked up in a standard reference (e.g. [Stolfi and De Figueiredo 1997]). Appendix A gives formulae for $\alpha, \beta, \gamma$ for common activation functions, and furthermore explicitly defines all affine arithmetic rules used in this work.

Here, we consider only MLP-like neural network computations, and thus sidestep difficult operations in full general affine arithmetic implementations, such as multiplication between two affine terms [Rump 1999].

### 3.3 Reduced Affine Arithmetic

In full affine arithmetic, each nonlinear operation introduces a new affine coefficient (Equation 4), gradually increasing computational cost. In our setting, each network layer of width $W$ would add $W$ new coefficients due to nonlinearities, which must then be propagated forward. In an 8-layer 32-width network, this means that before the final dense layer what is normally a $\mathbb{R}^{32}$ vector will be replaced with a collection of 224 $\mathbb{R}^{32}$ vectors encoding affine coefficients, each of which must be propagated via matrix multiplication, resulting in more than a 200× increase in computation. Although this cost may be worthwhile, it is pragmatic to consider alternatives.

Rather than retaining all affine coefficients, one can periodically reduce, or "condense" to some smaller set of coefficients, decreasing computational cost at the expense of potentially missing opportunities for cancellation [Stolfi and De Figueiredo 1997, §3.18.1], [Gamito and Maddock 2007]. Concretely, condensation replaces some set of affine coefficients at indices $D = \{i_0, \ldots, i_N\}$ with a single new coefficient holding the sum of their magnitudes

$$\text{condense}(\hat{x}, D) = x_0 + \sum_{i \in D} x_i \epsilon_i + \left( \sum_{i \in D} |x_i| \right) \epsilon_{N+1}. \quad (5)$$

One still must decide when to condense, and which coefficients to keep. We consider four policies:

- **affine-full**: no condensation, retain all affine terms
- **affine-fixed**: retain only affine terms from the original input domain; immediately condense all others
- **affine-truncate**: retain the $n_{\text{keep}}$ largest-magnitude terms
- **affine-append**: after each nonlinearity, append the $n_{\text{append}}$ largest-magnitude new affine terms and condense the rest

In all cases we must also retain one additional affine term to hold the condensed value. The tradeoffs between these strategies are not clear a priori, motivating an empirical approach (Section 3.4).

### 3.4 Selecting a Range Analysis Strategy

The extensive literature on automatic arithmetic for range analysis leads to many variants which could compute bounds on neural implicit functions. However, these approaches vary drastically in their computational cost, and the tightness of the resulting bounds.

**Performance.** Analyzing neural networks goes hand-in-hand with vectorized, GPU-based computation, and the performance trade-offs therein. For instance, vectorization implies maintaining the same set of affine terms for all quantities, precluding sparse representations with different terms for each. Additionally, from the outset we consider only methods for which range analysis matrix multiplication can be implemented as a sequence of ordinary fast matrix multiplication primitives, which are crucial for performance. Likewise, large amounts of dense arithmetic (e.g. in affine-full) may be surprisingly performant compared to sorting and irregular data access (e.g. in affine-truncate).

**Correctness.** We emphasize that all considered variants of range analysis always yield correct bounds, in the sense the output of the function on the interval is necessarily contained within the computed bounds. Even floating-point inaccuracy can be addressed via careful control of rounding modes [Stolfi and De Figueiredo 1997], although we do not find it necessary in this work. However, although these bounds are always correct, they are not necessarily tight, and some variants of range analysis discussed above yield dramatically tighter bounds than others. The tightness of the bounds in-turn affects the efficiency of downstream algorithms.

**Empirical Study.** We conduct an empirical study to analyze the tradeoffs of range analysis techniques for neural implicit surfaces, which is to our knowledge the first in the context of neural networks. We construct a dataset of neural implicit shapes fit via a variety of strategies, and for each we measure the tightness of the resulting bounds, as well as the added computational burden of range arithmetic—details are in Appendix B. We consider ordinary interval arithmetic (Section 3.1), several variants of affine arithmetic.
(Sections 3.2 and 3.3), and also a slope-interval form akin to the method of Ratz [1996], which combines interval arithmetic and automatic differentiation to bound derivatives. See Table 1 for results. Interval arithmetic is the least expensive, but yields extremely pessimistic bounds, and the slope-interval method offers little beyond affine arithmetic. The full, fixed, and truncate variants of affine arithmetic all have advantages, depending on the context.

**Recommendations.** We always recommend the use of affine arithmetic as opposed to interval or other arithmetics. With this approach, a single implementation can easily adjust the truncation policy based on the task at hand. For 1d ray casting queries, affine-fixed arithmetic offers the best performance, due to its low cost. For spatial 3d queries, we find that surprisingly affine-full arithmetic is generally the most effective—although it performs a great deal of the function truly is bounded away from ±107:6, may report the value of the neural function as ˆx = x.

3.5 Applying Range Analysis

We have now identified variants of affine arithmetic which are well-suited to computing range bounds on neural implicit functions, providing a foothold to design geometric queries. To be clear, we do not propose any new network architecture or training objectives, but instead enable these queries directly on a wide range of existing architectures. The class of networks to which our method applies includes MLPs, and is trivially extended to other common architecture components such as residual connections or latent inputs. In principle, it can be applied to any layer operation for which an affine bound can be derived.

We will abstract over the use of range analysis via a function RANGEBOUND, which takes as input an s-dimensional query box (which need not be axis-aligned), and classifies the value of the neural function as POSITIVE, NEGATIVE, or UNKNOWN within the box. For instance, ray casting requires bounds along a 1d box which is not axis-aligned, while spatial queries decompose space into axis-aligned 3d boxes. Because the bounds from range analysis are not necessarily tight, RANGEBOUND may report UNKNOWN over large queries, even if the function truly is bounded away from 0; this can be resolved by subdividing and repeating the query over multiple smaller ranges.

This formulation is not limited to spatial coordinates; it applies to any network input such as latent parameters of a network, and the queries derived herein still apply with only small modifications. Any network inputs for which we are not computing bounds are assigned the constant affine value ˆx = x.

### Procedure 1 RANGEBOUND( fθ, c, {ui})

**Input:** A function fθ : Rd → R and a query box B of dimension s ≤ d defined by its center c ∈ Rd, and s orthogonal box axis vectors {ui} ∈ Rd; not necessarily coordinate axis-aligned.

**Output:** A bound on the sign of fθ(x) ∀x ∈ B as one of POSITIVE, NEGATIVE, or UNKNOWN.

1. ˆx ← c + ∑i=1s ui vi 
   ▷ Construct affine bounds defining the box
2. ˆy ← fθ( ˆx) 
   ▷ Propagate affine bounds (Section 3.2)
3. [y−, y+] ← range( ˆy) 
   ▷ Bound the output (Equation 3)
4. if y− > 0 then return POSITIVE
5. if y+ < 0 then return NEGATIVE
6. else return UNKNOWN

3.6 Implementation

To facilitate integration in deep learning pipelines, we implement affine arithmetic for neural implicit surfaces, as well as our geometric queries, in the JAX framework [Bradbury et al. 2018]. We also investigated a prototype JAX range analysis implementation as a general transformation applied to arbitrary JAX programs—we ultimately found that specifying to MLPs resulted in a more efficient implementation, but this is an exciting avenue for future investigation. Range analysis on neural networks and the queries below are evaluated entirely on the GPU, where we leverage parallel traversals and dynamic batching to compute efficiently with fixed-size array kernels; an implementation is included as supplementary material. All timings are measured on an RTX 2070 GPU.

To ensure correctness, we also validate our range analysis bounds by fuzz-testing Procedure 1 on 106 randomly sampled input regions with a variety of network architectures, ensuring that point-sampled function evaluations always lie within floating point tolerance of the computed bounds. These tests succeed in all cases.

4 GEOMETRIC QUERIES

Range analysis via affine arithmetic now provides the key missing tool for geometric operations on general neural implicit surfaces by efficiently computing bounds on the value of the implicit function over spatial regions. In this section, we develop a variety of useful geometric queries using these bounds—in many cases, these queries are possible on general neural implicit surfaces for the first time. Appendix B lists additional configuration details for experiments.

**Visualization.** All figures are rendered via direct ray casting of neural implicit surfaces using our ray cast query (Section 4.2), and are shaded via material capture (except in Figure 14). Ground shadows are evaluated via casting rays upward followed by a Gaussian blur, and an ambient occlusion term is approximated by sampling points on a hemisphere, again both using our ray casting operation.
4.1 Defining Convergence and Correctness

Selecting an appropriate convergence criterion is a subtle but important dilemma in all of the queries we would like to perform, both to avoid excess computation, and to ensure termination even in imperfect floating-point arithmetic. In other settings (e.g. implicit SDFs), convergence can be defined in terms of the magnitude of the implicit function, when some $|f(p)| < \epsilon$. However, such a convergence test is not appropriate for general implicit functions, where the magnitude of $f$ might vary wildly, and is not known a priori. Similarly, it is infeasible to provably capture all, pathologically small features which might exist in an implicit surface, given the bounded accuracy of numerical computation.

Instead, we argue for defining convergence and correctness in the sense that the output of a query must be correct for some surface which is a dilation or contraction of the true level set by at most some small $\delta$ (see inset). For example, in the case of ray casting below, this criterion implies detecting convergence whenever a point $p$ within $\delta$ along the ray has $f(p)$ with the opposite sign from the origin, and it also means that when not converged, a step of size $\delta$ is always safe. Similar convergence policies are used in all other queries as well—we guarantee that results are within $\delta$ of the true level set, and that a region of the surface of size at least $\delta$ is never “missed” by the query, no matter what the magnitude of $f$ might be. We use $\delta = 0.001$ for all examples, on shapes normalized to the unit sphere.

![Image of ray casting and convergence](image)

Fig. 6. Previously, casting rays against general neural implicit surfaces required small, fixed-sized steps, used here to render an image. Large steps can miss parts of the surface (top, left), but smaller steps are expensive (top, middle). Our approach avoids tuning a step size, and is much faster (top, right). We measure this effect via the image error at various render time budgets (bottom), varying the fixed step size or our convergence parameter $\delta$, respectively.

4.2 Ray Casting

Finding the intersection with a ray is the most common and widely-studied geometric query on implicit surfaces. Precisely, given a ray source $p \in \mathbb{R}^3$ and direction $r \in \mathbb{R}^3$, we seek the smallest $t > 0$ such that $f(p + tr) = 0$. Often these rays are primary pixel rays in some view of the scene, but ray casting also arises more broadly in rendering and beyond.

We adopt a simple strategy of marching along the ray, attempting steps of size $\sigma$: if range arithmetic can bound the implicit function away from 0 on the step interval, we take the step and increase $\sigma$ by a factor $\eta_+$, otherwise we decrease $\sigma$ by a factor of $\eta_-$ and retry the step. Similar approaches have a long history in computer graphics [De Cusatis et al. 1999; Gamito and Maddock 2007; Mitchell 1990]. Equipped with range bounds on MLPs from Section 3, we can now apply this strategy to general neural implicit surfaces for the first time. This scheme is described precisely in Procedure 2. It is used to render all visualizations in this work, and Figure 6 gives a comparison to alternative approaches.

Validating steps with range analysis guarantees correct output regardless of the choice of initial step size, and adapting the step size by $\eta_+\eta_-$ automatically adjusts to an appropriate scale for the problem. We suggest $\sigma_0 = \tau_{max}/10, \eta_+ = 0.5, \eta_- = 1.5$ as reasonable step size parameters, and use these values in all experiments. The miss threshold $\tau_{max}$ depends on the length scale and typical casting distance; we use $\tau_{max} = 10$ for shapes normalized to the unit sphere.

In principle, ray casting queries could be further accelerated using the spatial bounding hierarchy from Section 4.5.

![Procedure 2: CASTRAY($f_0, p, r$)](image)

**Input:** An implicit surface $f_0 : \mathbb{R}^d \rightarrow \mathbb{R}$, and a ray source and direction $p, r \in \mathbb{R}^3$.

**Output:** The distance $t$ to the ray-surface intersection, or no hit.

1. $\sigma \leftarrow \sigma_0$, $t \leftarrow 0$ $\quad$ \text{\textit{\#Initialize steps}}
2. $f_0 \leftarrow f_0(p)$
3. \textbf{while} $t < \tau_{max}$ \textbf{do} $\quad$ \text{\textit{\#March forward}}
4. $p_c \leftarrow p + (t + \delta)r$ $\quad$ \text{\textit{\#Convergence test}}
5. $f_c \leftarrow f_0(p_c)$
6. \textbf{if} DifferentSigns($f_0, f_c$) \textbf{then return} $t$ $\quad$ \text{\textit{Found hit}}
7. $c \leftarrow p + (t + \sigma/2)r$ $\quad$ \text{\textit{\#Construct 1d query box}}
8. $\nu \leftarrow (\sigma/2)r$
9. \textbf{if} RangeBound($f_0, c, \nu$) \textbf{\#Unknown then} $\quad$ \text{\textit{Section 3}}
10. $\sigma^* \leftarrow \sigma$, $\sigma \leftarrow \sigma \eta_+$ $\quad$ \text{\textit{Step is safe, increase step size}}
11. \textbf{else} $\quad$ \text{\textit{Step is not safe, decrease step size}}
12. $\sigma^* \leftarrow 0$, $\sigma \leftarrow \sigma \eta_-$
13. $t \leftarrow t + \max(\sigma^*, \delta)$ $\quad$ \text{\textit{Take step (tolerance $\delta$ is always safe}}$
14. \textbf{end while}$
15. \textbf{return} NO_HIT
4.3 Frustum Ray Casting

When casting primary rays to render an image, rays from adjacent pixels traverse similar regions of space, wastefully repeating similar computation—an effect which becomes more pronounced at higher resolutions. This observation has led to a variety of techniques in traditional ray tracing and even range analysis which process groups of nearby rays simultaneously e.g. by Flórez et al. [2006]; Reshetov et al. [2005]. Similar concerns arise in neural volumetric rendering (e.g. [Barron et al. 2021]), although there the objective is approximate anti-aliasing moreso than exact spatial acceleration. By applying 3D range analysis to a box which bounds a frustum of rays, we can accelerate ray casting against existing neural implicit surfaces while still guaranteeing precisely correct results.

Our strategy is to initialize a coarse set of frusta over the primary pixel rays in an image, and march each frustum forward with steps similar to Procedure 2 (see inset). The frusta are dynamically subdivided whenever the step size $\sigma$ becomes smaller than the width of the frustum, eventually reducing to individual pixel rays as they hit the surface. Although 3D range analysis over boxes is moderately more expensive than 1D analysis (see Table 1), the algorithmic advantage of amortizing locally-similar computation leads to significant performance improvements over casting individual rays (Figure 8). This gap widens as resolution increases, improving scaling for high-fidelity renderings.

4.4 Empty Spheres

An empty sphere query at a point $p$ reports the radius of a sphere at $p$ which does not intersect the surface. This radius is not required to be maximal, though larger radii are preferred. Much like interval ray marching, we can evaluate empty sphere queries by applying range analysis on a box centered at $p$: if range arithmetic determines the box is POSITIVE or NEGATIVE then we have answered the query, and if not we try a smaller box.

A natural application of empty sphere queries is traversing random walks in space, repeatedly moving to a random point on the empty sphere until eventually reaching the surface. These random walks enable a grid-free Monte-Carlo scheme for the Poisson-like PDEs which are widespread in graphics and geometry processing [Nabizadeh et al. 2021; Sawhney and Crane 2020]. However, the true value of bounding 3d spatial regions arises from constructing hierarchies over the domain.

4.5 Spatial Hierarchies

Hierarchical spatial acceleration structures are a cornerstone tool in high-performance visual computing, from bounding volume hierarchies to mipmaps. Recent work has likewise shown dramatic performance benefits in the neural implicit context (e.g. [Barron et al. 2021; Takikawa et al. 2021]). However, these methods target fast forward evaluation and training rather than evaluating geometric queries as investigated here, and introduce new customized architectures. Instead, we will show that range analysis can be used to build guaranteed hierarchies over arbitrary existing neural implicit surfaces, even ordinary MLPs, enabling a variety of fast guaranteed spatial queries.

First, in Procedure 3 we describe a general branch-and-bound procedure for constructing a bounding $k$-D tree of a neural implicit surface using range analysis. The queries below in Section 4.6 through Section 4.10 all make use of variants of this strategy, adapting either the refinement criterion for the tree or the manner in which it is traversed.

On convergence the resulting tree has guaranteed accuracy, in the sense that nodes classified as POSITIVE or NEGATIVE are necessarily strictly outside or inside of the shape, respectively, and UNKNOWN nodes are within $\delta$ of the level set. This property in-turn enables spatial queries with guaranteed accuracy by evaluation over the set of tree nodes. An exception to the convergence policy is Sections 4.6 and 4.7, where the query demands refinement to a predefined depth. In practice, BUILDSPATIALTREE is not implemented recursively, but iteratively in parallel rounds of exploring and expanding batches of nodes. A similar strategy could be used to build other acceleration structures such an octree or bounding volume hierarchy, but here we use a $k$-d tree because it is simple and extends trivially to higher dimension (e.g. for bounding with respect to latent parameters).
In principle a similar tree could be constructed without range analysis, via a sufficiently dense grid of samples to classify nodes in the sense of Section 4.1. However, such an approach would be dramatically more expensive. As an example, constructing such a tree for the shape in Figure 11 would require 5.3B function samples, which already require 193 seconds to evaluate; building our tree takes just 0.610 seconds.

**Procedure 3**  
**BuildSpatialTree**(\(f_0, x_l, x_u\))

**Input:** An implicit surface \(f_0: \mathbb{R}^d \rightarrow \mathbb{R}\), and domain bounds \(x_l, x_u \in \mathbb{R}^d\) as lower and upper corners of bounding box.

**Output:** A \(k\)-d tree bounding the level set of \(f_0\).

1. \(\triangleright\) Classify the value of \(f_0\) in the node (Section 3)
2. \(x_c \leftarrow (x_l + x_u)/2\)
3. \(v \leftarrow \text{Diag}(x_u - x_l)\)
4. \(t \leftarrow \text{RangeBound}(f_0, x_c, v)\)
5. \(\triangleright\) Test convergence for nodes away from the level set
6. \(\triangleright\) Test convergence for small nodes near surface (Section 4.1)
7. \(\triangleright\) (alternately, recurse to some fixed depth instead)
8. if \(t \in \{\text{NEGATIVE}, \text{POSITIVE}\}\) then return \((x_l, x_u, t)\)
9. \(\triangleright\) Compute a split point
10. \(d_l \leftarrow \text{argmax}(x_u - x_l)\)
11. \(x_l \leftarrow 0^d\), \(x_{l,t} \leftarrow (x_u - x_l)_t\)
12. \(\triangleright\) Recurse on both subtrees and return the union of all nodes
13. \(T_a \leftarrow \text{BuildSpatialTree}(f_0, x_l, x_{l,t})\)
14. \(T_b \leftarrow \text{BuildSpatialTree}(f_0, x_l + x_{l,t}, x_u)\)
15. return \(T_a \cup T_b\)

**4.6 Surface Sampling**

Sampling a set of points on a surface is a common task in geometric learning. Our spatial hierarchy can be used to sample efficiently, outperforming naive rejection sampling by an order of magnitude for large sample sets.

![Fig. 10. Sampling a points on a surface is a common task in geometric learning. Our spatial hierarchy can be used to sample efficiently, outperforming naive rejection sampling by an order of magnitude for large sample sets.](image)

**Figure 10** shows the result of this process, and Figure 9 plots the corresponding runtime for generating a specified number of samples with our method and with naive rejection sampling. After an initial cost of building the hierarchy, our method becomes an order of magnitude faster—this gap widens as smaller sampling bands are used or the number of samples increases.

For applications requiring points lying more precisely on the surface, our sampling may benefit from “polishing” using the method of Wang et al. [2021], but this is left as future work.

**4.7 Hierarchical Mesh Extraction**

Implicit representations are appealing for their mesh-free nature, but nonetheless it is common to extract an explicit triangle mesh of the level set, either as a format conversion or to enable operations defined only on a mesh [Park et al. 2019]. Our spatial hierarchy can be leveraged to accelerate marching cubes mesh extraction [Lorensen and Cline 1987] by only extracting from cells near the surface, while necessarily producing the same output as brute-force extraction.

In this case, given an extraction resolution \(2^m\) along each dimension, we build a spatial hierarchy via Procedure 3 to a fixed depth of \(3m\) while retaining all nodes classified as \text{UNKNOWN}. The factor of 3 arises from splitting the k-D along each dimension. Applying marching cubes extraction individually in each node then generates the resulting mesh—the nodes excluded by the hierarchy necessarily would not contribute. In practice, we apply dense extraction at the bottom \(l = 3\) levels of our hierarchical for performance.

The hierarchical nature of our mesh extraction bears resemblance to the method of Mescheder et al. [2019]. When the network is smooth, we suspect their method could be faster, though surface features smaller than their initial low-resolution grid could be missed. In contrast, our method is guaranteed to split cells that contain the surface, with error bounded by the finest grid resolution. Moreover, our method will work for arbitrarily misbehaving implicit (e.g., randomly initialized networks, for use in loss functions).
Figure 11 shows the result of this procedure, where a 1.23 million face mesh is extracted in 1.46 seconds, including the time to build the hierarchy. Merely evaluating \( f \) at an equivalent grid of dense points would take 5.03 seconds. In fact, this procedure scales remarkably well to even higher resolution meshes: extracting a 4.9 million face mesh of the same implicit surface takes just 4.13 seconds, while dense evaluation would require 39 seconds. As an added bonus, the resulting mesh lies in the leaf nodes of a k-D tree by construction—it already has a spatial acceleration structure ready for subsequent processing if desired.

Here we treat only classic marching cubes, other approaches such as dual contouring [Ju et al. 2002] could be applied using a similar strategy. Additionally, our spatial acceleration is also compatible with differentiable variants of mesh extraction [Liao et al. 2018; Shen et al. 2021], though we do not yet pursue an implementation.

4.8 Bulk Properties
Physical simulation on implicit shapes requires evaluating bulk properties such as mass and moments of inertia via integrals over the interior of the shape. The spatial hierarchy (Procedure 3) makes it straightforward to evaluate these integrals by accumulating contributions from all interior nodes. We estimate the contribution from any \( \text{UNKNOWN} \) nodes straddling the boundary via random sampling, akin to Section 4.6. If the tree is fully refined, the resulting values already satisfy the convergence guarantee in Section 4.1, although faster runtimes can be obtained by refining to some fixed depth. If desired, we can further bound the possible error in an integral via the largest and smallest possible contribution from any \( \text{UNKNOWN} \) nodes in the hierarchy. The mass of the inset shape is computed to a relative accuracy of \( 5 \times 10^{-5} \) via this strategy in 0.77 seconds, vs. 6.72 seconds for integration with uniform random samples.

4.9 Intersection and Collisions
Given two neural implicit shapes, how can we test whether they intersect one another? This basic operation will be increasingly necessary for tasks like path planning and simulation if neural implicit surfaces are to be incorporated in realistic virtual environments. Our spatial hierarchy (Procedure 3) can be used to test for intersections by simultaneously subdividing the tree with respect to two implicit functions. If either function is bounded \textsc{positive} in a node, then that node necessarily does not contain an intersection. Repeatedly subdividing the tree either yields a node in which the surfaces intersect, or verifies that there is no such intersection.

Figure 12 shows this procedure testing for intersections between two neural implicit surfaces, one of which is encoded as an SDF and the other as an occupancy function. The runtime for that example is 80ms per query; computing the query to the same accuracy guarantee via a densely sampled grid would require 5.3B function samples and several minutes of processing. We note that, coupled with the bulk properties in Section 4.8, we have now developed the core computational ingredients for rigid body simulation of neural implicit surfaces, an exciting avenue for ongoing applications.

4.10 Closest Points
Given a point in space, a closest point query seeks the nearest location on the neural implicit level set. On an exact signed distance function the nearest point on the surface can be computed as \( p_{\text{nearest}} = p - f(p) \nabla f(p) \), but for approximate neural SDFs and more general neural implicit surfaces, there is no such clear strategy. Fortunately, our hierarchical bounding k-D tree is also a natural data structure to perform closest point queries. Indeed, finding nearest-neighbor points is a classic application of k-D trees; the only nuance in this case is that the target set of points is a continuum encoded by the implicit function.

Given a query point \( p \), we descend the tree from Procedure 3: if test points sampled on the faces of a node include both signs of the implicit function, then the node necessarily spans the surface, and we update the closest-point distance as the farthest distance from \( p \) to any point in the node, using the node center as the corresponding...
much worse as resolution increases. We provide several experimental comparisons to brute-force sampling approaches, showing that our guaranteed range-based queries offer significant benefits (Figure 6, Figure 9, Section 4.7, Section 4.9, and Section 4.10).

Another possibility is to extract a mesh of the surface, and apply mesh-based techniques. This too may be an expensive option, and runs the risk of aliasing fine-scale features—typical mesh extraction resolutions are much coarser than the $\delta = 0.001$ convergence tolerance used in our experiments. Nonetheless, when mesh-based computation is used, our fast mesh extraction (Section 4.7) can be used to accelerate the process.

A more principled approach is to leverage a global Lipschitz bound $|\nabla f| < L$. Intuitively, Lipschitz bounds are a global counterpart to our local range analysis. However, whereas our range analysis bounds the function locally with respect to each individual evaluation, the Lipschitz constant is computed once for the entire domain, and hence typically gives much less tight bounds. One popular technique is to estimate $L$ as the product of the maximum eigenvalues of dense layer matrices computed via a power method, which has found applications in deep network regularization and robustness [Arjovsky et al. 2017; Miyato et al. 2018; Tsuzuku et al. 2018]. In our context, replacing $f \rightarrow f/L$ would transform any implicit function in to a weak signed distance function. However, global Lipschitz bounds computed in this manner are extremely pessimistic, on the order of $10^5$ for networks even when they fit high-quality SDFs, making Lipschitz bounds ineffective for guaranteed geometric queries on existing networks.

5 AN APPLICATION TO INVERSE RENDERING

Inverse rendering directly optimizes scene data to match target images, encompassing many tasks in computer graphics and vision. A full review is beyond the scope of this section; we refer to Tewari et al. [2021], Li et al. [2018], and Nicolet et al. [2021].

Neural implicit surfaces are an appealing representation for surface geometry in inverse rendering, but their use requires somehow intersecting primary rays from a camera against the implicit surface. Existing work resorts to either raycasting with small, fixed timesteps (e.g., [Niemeyer et al. 2020]), or extracting a mesh and then rasterizing (e.g., [Cole et al. 2021]). Our ray casting queries Section 4.2 are a valuable new primitive operation in this context, enabling fast and accurate rendering of a neural implicit surface, even when randomly initialized.

In Figure 14 we demonstrate a simple inverse rendering application as a proof of concept. Here, we fit a neural implicit surface to synthetic target camera views, and render using our ray casting queries and Blinn-Phong shading [Blinn 1977]. Only two loss terms are used, an $L_1$ image difference loss and a ray occupancy loss which prevents collapse by encouraging the shape to match the foreground mask of the target images (a similar loss appears in Niemeyer et al. [2020, Eqn. 14 & 15]). We note that this simple and effective occupancy loss is only available when fitting occupancy networks, as opposed to SDF networks. Enabling the use of such more general networks is a key goal of this work. Appendix B gives training details.
Though intentionally simple, Figure 14 demonstrates the promise of our approach. In the future, our ray casting primitive could be leveraged in a variety of neural inverse rendering formulations, from deep priors [Mescheder et al. 2019] to global illumination differentiable renderers [Nimier-David et al. 2019].

6 CONCLUSION

This work studies the application of automatic range analysis to general neural implicit surfaces, enabling a wide variety of useful geometric queries to be applied to existing architectures.

Limitations. Our queries offer guaranteed accuracy in sense of Section 4.1. However, we do not address inaccuracies which might arise due to evaluation in inexact floating point arithmetic, although we do not notice any such instabilities in practice. Existing work in range analysis can bound floating point error via careful manipulation of rounding modes [Stolfi and De Figueiredo 1997], but exact queries would likely demand exact arithmetic, a pursuit which has yielded powerful albeit complex and expensive algorithms elsewhere in computational geometry [Fabri and Pion 2009].

There remains a gap between the runtime performance of neural SDF-based queries such as fast ray casting (demonstrated at real-time rates by Takikawa et al. [2021] and elsewhere), and the more general queries presented here. Indeed, if one has a high-quality SDF on which sphere tracing is acceptably accurate, it should certainly be used for that purpose. Rather, the goal of this work is to efficiently extend ray casting—and many other queries—to a much broader set of architectures and applications where a high-quality SDF is unavailable, such as the inverse rendering task in Section 5. It is then unsurprising that our more general approach comes at additional computational cost. In this work we have focused mainly on introducing new algorithms, but further investigation of high-performance kernels could likely accelerate our method significantly.

Future Work. For now, we primarily study small MLP-like architectures for neural implicit surfaces, but our techniques should also apply more broadly to other architectures with little modification. Interestingly, architectures which store data in an underlying spatial structure [Müller et al. 2022; Takikawa et al. 2021], could be handled by deriving range bounds for sampling from the structure. With generalization in mind, we investigated a prototype implementation of range analysis as a fully-automatic transformation which can be applied to arbitrary JAX programs, and we are eager to explore such an approach as a general tool for more exotic architectures.

More broadly, the ability to perform range analysis and build spatial hierarchies likely has value to many other applications of neural fields [Xie et al. 2022]. One particular area of interest is neural volumetric rendering [Mildenhall et al. 2020] where we could apply range analysis to hierarchical volumetric integration with guaranteed accuracy. In general, we hope that providing tools which apply broadly across many architectures and applications will be a key that unlocks many promising research directions.

ACKNOWLEDGMENTS

The authors are grateful to Aravind Ramakrishnan, Otman Benchechroun, and Rohan Sawhney for assistance preparing experiments and reviewing early drafts, as well as Towaki Takikawa, David Levin, and Derek Nowrouzezahrai for insightful discussions.

This research is funded in part by the Fields Institute for Mathematical Sciences, the Vector Institute for AI, the Sloan Research Foundation, NSERC Discovery Grants (RGPIN2017-05235, RGPAS–2017-507938), New Frontiers of Research Fund (NFRFE–201), the Ontario Early Research Award program, the Canada Research Chairs Program, and gifts by Adobe Systems.

REFERENCES

Stavros P Adam, George D Magoulas, Dimitrios A Karras, and Michael N Vrahatis. 2016. Bounding the search space for global optimization of neural networks learning error: an interval analysis approach. Journal of Machine Learning Research 17 (2016), 1–40.

Götz Alefeld and Günter Mayer. 2000. Interval analysis: theory and applications. Journal of Computational and Applied Mathematics 121, 1-2 (2000), 421–464.

Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein generative adversarial networks. In International conference on machine learning. PMLR, 214–223.

Matan Atzmon and Yaron Lipman. 2020a. Sal: Sign agnostic learning of shapes from raw data. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2565–2574.

Matan Atzmon and Yaron Lipman. 2020b. SalD: Sign Agnostic Learning with Derivatives. In International Conference on Learning Representations.

Csaba Bálint and Gábor Valasek. 2018. Accelerating Sphere Tracing. Proceedings of Eurographics Short Papers (2018), 4 pages.

Jonathan T Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Braila, and Pratul P Srinivasan. 2021. Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields. In Proceedings of the IEEE/CVF International Conference on Computer Vision. 5855–5864.

David Betancourt and Rafi Mahanna. 2021. Interval Deep Learning for Uncertainty Quantification in Safety Applications. arXiv preprint arXiv:2105.06438 (2021).

James F Blinn. 1977. Models of light reflection for computer synthesized pictures. In Proceedings of the 4th annual conference on Computer graphics and interactive techniques. 192–198.

James F Blinn. 1982. A Generalization of Algebraic Surface Drawing. ACM Trans. Graph. 1, 3 (1982), 235–256.

Jules Bloomenthal, Chandrani Bajaj, Jim Blinn, Marie-Paule Cani, Alyn Rockwood, Brian Wyvill, and Geoff Wyvill. 1997. Introduction to Implicit Surfaces. Chapman and Hall, New York.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Neuca, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Quan Zhang. 2018. JAX: Composable transformations of Python+NumPy programs. https://github.com/google/jax.

Bryan Chen. 2008. Static Analysis for Efficient Affine Arithmetic on GPUs. Master’s thesis. University of Waterloo.

Zhiqin Chen and Hao Zhang. 2019. Learning Implicit Fields for Generative Shape Modeling. In IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2019, Long Beach, CA, USA, June 16-20, 2019. Computer Vision Foundation / IEEE, 5939–5948. http://openaccess.thecvf.com/content_CVPR_2019/html/Chen_Learning_...
Implicit Fields for Generative Shape Modeling. CVPR 2019 paper.html
Forrest Cole, Kyle Genova, Avneesh Sud, Daniel Vlasic, and Zhongteng Zhang. 2021. Differentiable surface rendering via non-differentiable sampling. In Proceedings of the IEEE/CVF International Conference on Computer Vision. 6088–6097.
JLD Comba and J Stollh. 1993. A Flexible architecture and its applications to computer graphics. Anais do SIBGRAPI, 9–18.
Hongkai Dai, Benoit Landry, Lujie Yang, Marco Pavone, and Russ Tedrake. 2021. Lyapunov-Stable neuronal network control. Robotics: Science and Systems (2021).
Thomas Davies, Derek Nowrouzezahrai, and Alec Jacobson. 2020. On the Effectiveness of Weight-Encoded Neural Implicit 3D Shapes. (2020). https://arxiv.org/abs/2009.09980
A De Cuatis, Luiz Henrique De Figueiredo, and Marcelo Gattass. 1999. Interval methods for ray casting implicit surfaces with affine arithmetic. In XI Brazilian Symposium on Computer Graphics and Image Processing (Cat. No. PR0064J1). IEEE, 65–71.
Tom Duff. 1992. Interval Arithmetic Recursive Subdivision for Implicit Functions and Constructive Solid Geometry. In Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH ’92). Association for Computing Machinery, New York, NY, USA, 131–138. https://doi.org/10.1145/133994.134027
Tim Elsner, Moritz Bünz, Victor Czech, Julius Nehrung-Wirzel, and Leif Kobbelt. 2021. Intuitive Shape Editing in Latent Space. (2021). https://arxiv.org/abs/2111.12488
Andreas Fabi and Sylvain Pion. 2009. CGAL. The computational geometry algorithms library. In Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems. 538–539.
Jorge Flores, Mateu Sbert, Miguel A. Sáiz, and Josep Vehí. 2006. Improving the interval ray tracing of implicit surfaces. In Computers Graphics International Conference. Springer, 655–664.
Oleg Fryazinov, Alexander A. Pasko, and Peter Ciminoin. 2010. Fast reliable interrogation of procedurally defined implicit surfaces using extended revised affine arithmetic. Comput. Graph. 34, 6 (2010), 708–718. https://doi.org/10.1016/j.cag.2010.07.003
Eric Galin, Eric Guérin, Axel Pariz, and Adrien Peytavie. 2020. Segment Tracing Using Local Lipschitz Bounds. Comput. Graph. 39, 2 (2020), 545–554. https://doi.org/10.1016/j.cag.2019.09.004
Manuel N. Gamito and Steve C. Maddock. 2007. Ray casting implicit fractal surfaces with reduced affine arithmetic. The Visual Computer 23, 3 (2007), 155–165.
Kyle Genova, Forrest Cole, Avneesh Sud, Aaron Sarna, and Thomas A. Funkhouser. 2020. Local Deep Implicit Functions for 3D Shape. In 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13–19, 2020. Computer Vision Foundation / IEEE, 4856–4865.
Sven Gowal, Kirshnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy Mann, and Pushmeet Kohli. 2018. On the effectiveness of interval bound propagation for training verifiably robust models. arXiv preprint arXiv:1810.12715 (2018).
Amos Grop, Lior Yariv, Niv Haim, Matan Atzmon, and Yaron Lipman. 2020. Implicit Geometric Regularization for Learning Shapes. In Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13–18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119). PMLR, 3789–3799.
John C. Hart. 1996. Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces. In Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques. The Visual Computer 12, 10 (Dec. 1996), 527–545. https://doi.org/10.1007/bf02667065
Wolfgang Heidrich and Hans-Peter Seidel. 1998. Ray-tracing Procedural Displacement Shaders. In Proceedings of the Graphics Interface Conference 1998, June 18-20, 1998, Vancouver, BC, Canada. A. Kossowski, W. E. L. Stuerzlinger, and J. LeNours (Eds.). Canadian Human-Computer Communications Society, 8–16.
Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. 1998. Sampling Procedural Shaders Using Affine Arithmetic. ACM Trans. Graph. 17, 3 (1998), 158–176. https://doi.org/10.1145/285857.285859
Matt Jordan and Alexandros G. Dimakis. 2020. Exactly Computing the Local Lipschitz Constant of ReLU Networks. In Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual, Hugo Larochelle, Marc-Aurelio Ranzato, Rüdiger Urbanke, and Karsten Li, Editors. PMLR, 3504–3515.
Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2020. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. In ECCV 2020, Vol. 12346. Springer, 405–421.
Matthew Mirman, Maximilian Baader, and Martin Vechev. 2021. The Fundamental Limits of Interval Arithmetic for Neural Networks. arXiv preprint arXiv:2112.06235 (2021).
Don P Mitchell. 1990. Robust ray intersection with interval arithmetic. In Proceedings of the Graphics Interface, Vol. 90. 68–74.
Takuru Miyato, Toshihiko Kataoka, Masanori Koyama, and Yaichi Yoshida. 2018. Spectral normalization for generative adversarial networks. arXiv preprint arXiv:1802.05957 (2018).
Ramon E. Moore, R Baker Kearfott, and Michael J. Cloud. 2009. Introduction to interval analysis. SIAM.
Thomas Müller, Alex Evans, Christoph Schierd, and Alexander Keller. 2022. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding. arXiv:2201.05899 Jan. (2022).
Mohammad Sina Nabizadeh, Ravi Ramamoorthi, and Albert Chern. 2021. Kelvin transformations for simulations on infinite domains. ACM Transactions on Graphics (TOG) 40, 6 (2021), 1–15.
Baptiste Nicolet, Alec Jacobson, and Wenzel Jakob. 2021. Large steps in inverse rendering of geometry. ACM Transactions on Graphics (TOG) 40, 6 (2021), 1–13.
Michael Niemeyer, Lars Mescheder, Michael Oechslie, and Andreas Geiger. 2020. Differentiable volumetric rendering: Learning implicit 3d representations without 3d supervision. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 3044–3055.
Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. 2019. Mitsuba 2: A retargetable forward and inverse renderer. ACM Transactions on Graphics (TOG) 38, 6 (2019), 1–17.
Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. 2019. Deepsd: Learning continuous signed distance functions for shape representation. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 165–174.
Ken Perlin and Eric M. Hoffert. 1989. Hypertexture. In Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1989, Boston, MA, USA, July 31-Aug 4, 1989, James J. Thomas (Ed.). ACM, 253–262.
Magdalena Praszczenkowa, Marcin Mazur, Tomasz Traziński, and Przemyslaw Spurek. 2021. HyperCube: Implicit Field Representations of Voxeiled 3D Models. arXiv preprint arXiv:2110.05770 (2021).
Ingo Quilez. 2008. 3D SDF functions. https://www.iquilezles.org/www/articles/diffusions/diffusions.htm
Dietmar Ratz. 1996. An optimized interval slope arithmetic and its application. Inst. für Angewandte Mathematik, Tim Reiner, Gregor Mückl, and Karsten Dachsbacher. 2011. Interactive Modeling of Complex Surfaces Using a Direct Visualization Approach with Signed Distance Functions. Computers and Graphics 35, 3 (June 2011), 596–603.
Eduardo Remel, Artem Lukoianov, Stephan R. Richter, Benoit Guillard, Timur M. Bagautdinov, Pierre Baqué, and Pascal Fua. 2020. MeshSDF: Differentiable Isosurface Extraction. In Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual, Hugo Larochelle, Marc-Aurelio Ranzato, Rüdiger Urbanke, and Karsten Li, Editors. PMLR, 3504–3515.
Alexandra Reshetov, Alexei Soupikov, and Jim Hurley. 2005. Multi-level ray tracing with affine arithmetic. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2916–2925.
William E. Lorenzen and Harvey E. Cline. 1987. Marching cubes: A high resolution 3D surface construction algorithm. ACM siggraph computer graphics 21, 4 (1987), 163–169.
Ji Maien. 1996. An Introduction to Implicit Techniques. Lars Mescheder, Michael Oechslie, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. 2019. Occupancy networks: 3d reconstruction in function space. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 4460–4470.
Yi Yi Liao, Simon Donne, and Andreas Geiger. 2018. Deep marching cubes: Learning explicit surface representations. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2916–2925.
Siegfried M Rump and Masahide Kashirwagi. 2015. Implementation and improvements of affine arithmetic. Nonlinear Theory and Its Applications, IETC 6, 3 (2015), 341–359.

Deepthi Moyi Sahoo, Abhishek Das, and Sneshashak Chakraverty. 2015. Interval data-based system identification of multitory shear buildings by artificial neural network modelling. Architectural Science Review 58, 3 (2015), 244–254.

Rohan Sawhney and Keenan Crane. 2020. Monte Carlo Geometry Processing: A Grid-Free Approach to PDE-Based Methods on Volumetric Domains. ACM Trans. Graph. 39, 4 (2020).

Dario Seyb, Alec Jacobson, Derek Nowrouzezahrai, and Wojciech Jarosz. 2019. Non-linear sphere tracing for rendering deformed signed distance fields. ACM Trans. Graph. 38, 6 (2019), 229:1–229:12.

Tianchang Shen, Jun Gao, Kangxue Yin, Ming-Yu Liu, and Sanja Fidler. 2021. Deep Marching Tetrahedra: a Hybrid Representation for High-Resolution 3D Shape Synthesis. Advances in Neural Information Processing Systems 34 (2021).

Jorge Stolfi and Luiz Henrique De Figueiredo. 1997. Self-validated numerical methods and applications. In Monograph for 21st Brazilian Mathematics Colloquium, IMPA, Rio de Janeiro. Citeseer. Vol. 5. Citeseer.

Towaki Takikawa, Joe Litalien, Kangxue Yin, Karsten Kreis, Charles Loop, Derek Nowrouzezahrai, Alec Jacobson, Morgan McGuire, and Sanja Fidler. 2021. Neural geometric level of detail: Real-time rendering with implicit 3D shapes. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 11358–11367.

Ayush Tewari, Justus Thies, Ben Mildenhall, Pratul Srinivasan, Edgar Tretschk, Yifan Wang, Christoph Lassner, Vincent Sitzmann, Ricardo Martin-Triguera, Stephen Lombardi, et al. 2021. Advances in neural rendering. arXiv preprint arXiv:2111.05849 (2021).

Theo Thonat, Francois Beaune, Xin Sun, Nathaniel Carr, and Tamy Boubekeur. 2022. Tesselation-Free Displacement Mapping for Ray Tracing. 40, 6, Article 282 (December 2021), 16 pages. https://doi.org/10.1145/3478513.3480535

Yusuke Tsuzuku, Issei Sato, and Masashi Sugiyama. 2018. Lipschitz-margin training: To propagate these bounds forward through a network, we require rules to update bounds after addition (by a constant as well as other with affine quantities), multiplication by a constant, matrix multiplication by a constant, and activation functions. Table 2 lists the rules, while Table 3 gives expressions for computing affine approximation parameters for common activation functions.

B ADDITIONAL DETAILS

Here, we give miscellaneous configuration details for the algorithms and experiments above.

Range Analysis Empirical Study. To construct a small benchmark dataset of neural implicit surfaces, we gather a collection of 10 shapes including characters, mechanical models, and 3D scans, and fit several MLPs to each, using all combinations of ReLU vs. ELU nonlinearities, as well as fitting SDFs under an L1 penalty vs. occupancy under cross-entropy loss. Each MLP has 7 hidden layers of width 32 for a total of 7553 parameters. Training points are sampled as in [Davies et al. 2020], and we train for 100 epochs with the ADAM optimizer, using a batch size of 512 and learning rate of $10^{-2}$ decreased by a factor 10 after 50 epochs. Timings are measured in the experimental configuration described in Section 3.6, and all time statistics are normalized to the fastest variant. To quantify the tightness of the bounds from each range analysis variant, we sample many random regions of varying sizes, and test whether the implicit function can be bounded away from zero over the region. Precisely, for any region size $s$, we compute $f$, the fraction of regions such that Procedure 1 outputs POSITIVE or NEGATIVE. We then report the largest $s$ such that $f \geq 50\%$. Bigger $s$ means tighter bounds; this size is an indication of e.g. how large of steps must be taken for ray casting, or how large the cells in the spatial hierarchy can be. Values are averaged over 10,000 random regions per implicit surface, and for timings we furthermore take the fastest of 5 runs to account for warm-up and other variance. We additionally measure the time to cast rays corresponding to pixels of a 256 x 256 camera view centered on the surface.
Table 2. The update rules to propagate affine arithmetic bounds on MLPs. All quantities are in-general vector or matrix-valued.

| Nonlinearity | Parameters \( \alpha, \beta, \gamma \) | Notes | Diagram |
|--------------|------------------------------------------|-------|--------|
| \( \text{ReLU}(\hat{x}) \) | \[
\begin{align*}
[x_-, x_+] & \leftarrow \text{range}(\hat{x}) \\
\alpha & \leftarrow \frac{\text{ReLU}(x_+) - \text{ReLU}(x_-)}{x_+ - x_-} \\
\beta & \leftarrow \frac{(\text{ReLU}(x_-) - ax_-)/2}{\alpha} \\
\delta & \leftarrow \beta
\end{align*}
\] | Chebyshev approximation | ![Diagram](image1.png) |

| Nonlinearity | Parameters \( \alpha, \beta, \gamma \) | Notes | Diagram |
|--------------|------------------------------------------|-------|--------|
| \( \text{ELU}(\hat{x}) \) | \[
\begin{align*}
[x_-, x_+] & \leftarrow \text{range}(\hat{x}) \\
\alpha & \leftarrow \frac{\text{ELU}(x_+) - \text{ELU}(x_-)}{x_+ - x_-} \\
r_u & \leftarrow \text{ELU}(x_-) - ax_- \\
r_l & \leftarrow (\alpha - 1) - \alpha(\ln \alpha - ax_-) \\
\beta & \leftarrow \frac{(r_u + r_l)/2}{\alpha} \\
\delta & \leftarrow r_u - \beta
\end{align*}
\] | Chebyshev approximation | ![Diagram](image2.png) |

| Nonlinearity | Parameters \( \alpha, \beta, \gamma \) | Notes | Diagram |
|--------------|------------------------------------------|-------|--------|
| \( \sin(\hat{x}) \) | \[
\begin{align*}
[x_-, x_+] & \leftarrow \text{range}(\hat{x}) \\
[s_-, s_+] & \leftarrow \text{cos}([x_-, x_+]) \\
\alpha & \leftarrow (s_- + s_+)/2 \\
\epsilon_p & \leftarrow \arccos \alpha, \quad \epsilon_n \leftarrow -\arccos \alpha \\
E & \leftarrow [x_-, x_+], \\
& \quad \text{clamp}(2\pi \text{ceil}(x_- + \epsilon_p)/(2\pi) - \epsilon_p, x_-, x_+), \\
& \quad \text{clamp}(2\pi \text{ceil}(x_- + \epsilon_n)/(2\pi) - \epsilon_n, x_-, x_+), \\
& \quad \text{clamp}(2\pi \text{floor}(x_- - \epsilon_p)/(2\pi) + \epsilon_p, x_-, x_+), \\
& \quad \text{clamp}(2\pi \text{floor}(x_- - \epsilon_n)/(2\pi) + \epsilon_n, x_-, x_+) \\
r_u & \leftarrow \max_{e \in \mathcal{E}}(\sin(e) - \alpha e) \\
r_l & \leftarrow \max_{e \in \mathcal{E}}(\sin(e) - \alpha e) \\
\beta & \leftarrow (r_u + r_l)/2 \\
\delta & \leftarrow r_u - \beta
\end{align*}
\] | ![Diagram](image3.png) |
Ray Casting. We march with a small safety tolerance, taking steps of size \(0.98\sigma\) to mitigate floating point inaccuracy, although we do not observe failures due to floating point in any case. For frustum ray casting, we initialize a \(16 \times 16\) grid of coarse frusta, and subdivide frusta in half along the largest dimension whenever the width of that dimension along the forward face is greater than twice the current step size \(\sigma\). Additionally, we note that the length of the bounding box used for range analysis needs to be extended slightly beyond the front face of the frustum defined by the rays in its corners, because the contained rays actually sweep out a spherical region. We compute the correct extent by forming a ray along the center of the frustum with length \(t + \sigma\), and measuring the distance of its endpoint from the frustum base.

Mesh Extraction. We build the tree with affine-all range analysis, and use dense evaluation for the lowest \(l = 3\) levels.

Inverse Rendering. The inverse rendering example in Figure 14 fits a 5-layer, 128-width network with ELU activations to 20 camera views, each at \(512 \times 512\) resolution, equally spaced in a sphere around the subject. The ground truth is rendered from a triangle mesh with Blinn-Phong shading by 3 fixed point lights in the scene. The loss is \(10\times\) the \(L_1\) image difference on the rendered image, plus a cross-entropy occupancy loss on the minimum value of the implicit function as sampled at 100 points along each ray. Rays from each pixel in all views are combined and batched with size 512. We train for 4 epochs using the ADAM optimizer, with a learning rate of \(1e^{-3}\) decayed by a factor of 0.5 on each epoch. The rendered initial view in Figure 14 is shown after a small number (100) of training steps, because the randomly-initialized network on the first iteration is not visually coherent.