Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013 (SNA + MC 2013)
La Cité des Sciences et de l'Industrie, Paris, France, October 27–31, 2013

Computing Numerically-Optimal Bounding Boxes for Constructive Solid Geometry (CSG) Components in Monte Carlo Particle Transport Calculations

David L. Millman[1*], David P. Griesheimer[1], Brian R. Nease[1], and Jack Snoeyink[2]

[1]*Bettis Laboratory, West Mifflin, Pennsylvania, USA*
[2]*Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, USA*
[*]Corresponding Author, E-mail: david.millman@unnpp.gov

For large, highly detailed models, Monte Carlo simulations may spend a large fraction of their run-time performing simple point location and distance to surface calculations for every geometric component in a model. In such cases, the use of bounding boxes (axis-aligned boxes that bound each geometric component) can improve particle tracking efficiency and decrease overall simulation run time significantly. In this paper we present a robust and efficient algorithm for generating the numerically-optimal bounding box (optimal to within a user-specified tolerance) for an arbitrary Constructive Solid Geometry (CSG) object defined by quadratic surfaces. The new algorithm uses an iterative refinement to tighten an initial, conservatively large, bounding box into the numerically-optimal bounding box. At each stage of refinement, the algorithm subdivides the candidate bounding box into smaller boxes, which are classified as inside, outside, or intersecting the boundary of the component. In cases where the algorithm cannot unambiguously classify a box, the box is refined further. This process continues until the refinement near the component's extremal points reach the user-selected tolerance level. This refinement/classification approach is more efficient and practical than methods that rely on computing actual boundary representations or sampling to determine the extent of an arbitrary CSG component. A complete description of the bounding box algorithm is presented, along with a proof that the algorithm is guaranteed to converge to within specified tolerance of the true optimal bounding box. The paper also provides a discussion of practical implementation details for the algorithm as well as numerical results highlighting performance and accuracy for several representative CSG components.

KEYWORDS: Monte Carlo, constructive solid geometry, octree, bounding box calculation, robust operations

I. Introduction

Particle tracking through complex models in Monte Carlo (MC) transport codes can account for a significant fraction of the total simulation cost, especially for MC codes that use Constructive Solid Geometry (CSG) model representations. In CSG, models are defined by collections of components, where each component is defined by the intersection, union, and/or set difference of half-spaces.

Most of the computational cost of particle tracking is due to two simple geometric operations. The first operation, `pointLocation`, determines the orientation of a point with respect to a component's surfaces. The second operation, `distanceToSurface`, calculates the distance of a particle to a surface intersection along a given ray. Individually, these operations are typically simple and can be performed quickly. However, tracking algorithms often require the operations to be performed on many (or all) components every time a particle moves. Thus, as the size and complexity of a model increases, the time spent tracking also increases. For large models, tracking time may dominate the cost of the simulation.

In other ray tracing applications, such as computer graphics, bounding boxes are used to dramatically accelerate similar point location and surface intersection operations.(1) A simple component (typically a box) is defined that completely encloses (bounds) a more complex and/or arbitrarily oriented component. Because an axis-aligned bounding box is defined by six axis-aligned planes, the operations `pointLocation` and `distanceToSurface` for an axis-aligned bounding box are significantly faster than performing the same operations on the enclosed component. Often, the bounding box can be used as a low-cost initial test for `pointLocation` and `distanceToSurface` operations. If a given point (or ray) does not lie within (or intersect) a bounding box then there is no need for further tests against the enclosed component. The efficiency of bounding box acceleration, however, depends on how tightly each box bounds its component.

Over the years, almost all MC codes have used techniques (such as multi-level hierarchy(2,3)) to accelerate particle tracking by reducing the number of components checked during each particle movement. In fact, several modern MC codes(2,4) are known to use simple forms of the bounding box acceleration technique. However, to the authors' knowledge, all implementations of the bounding box method have been limited to restricted sets of pre-defined CSG shapes (e.g., parallelepipeds, cylinders, spheres, cones). Methods for general CSG components have either produced loose bounding boxes, ran too slowly, or were non-robust.

The widely used ray tracing engine POV-Ray(5) suggests bounding complicated shapes by applying set operations on boxes for pre-defined shapes. However, the documentation notes that, "for difference and intersection operations this will hardly ever lead to an optimal bounding box."

In many ray-tracing applications, objects can be represented by a collection of polygons. In this representation, one can bound the input by bounding the point set defining the polygons. O'Rourke[(6)] gave the first (and to our knowledge, only) optimal time algorithm for computing a minimal volume arbitrarily oriented box for a point set. For n points in $\mathbb{R}^3$, O'Rourke's algorithm takes $O(n^3)$ time and $O(n)$ space. For an approximation value $\epsilon > 0$, Barequet and Har-Peled,[(7)] presented two numerically-optimal ($(1+\epsilon)$-approximation) algorithms taking time $O(n + 1/\epsilon^{4.5})$ and $O(n \log n + n/\epsilon^3)$. However, because Monte Carlo particle tracking codes do not usually represent geometric objects as polygons, these algorithms are of limited applicability. For example, codes such as MC21[(2)] and MCNP[(3)] allow users to build models with unbounded primitives (such as paraboloids or hyperboloids).

Another approach to computing bounding boxes for CSG models is to explicitly compute the model's boundary. In an older survey paper, Lin and Gottschalk,[(8)] note that in practice, doing the conversion is very difficult. Mainly, the difficulty is caused by floating point errors and degenerate inputs. In more recent work, Dupont *et al.*[(9)] investigated intersecting surfaces and Schömer and Wolpert[(10)] investigated spatial decompositions, both of which are useful operations for the conversion. Keyser*et al.*[(11–14)] created the ESOLID modeling system specifically for the conversion, but, even this system sometimes produces an incorrect boundary for extremely complicated models.[(11)]

Alternatively, point location or ray tracing algorithms offer a purely numerical approach for approximating bounding boxes of CSG components. In these algorithms, a conservatively-large bounding box is created around the component of interest. Test points (or rays) are then used to identify regions of the initial bounding box that do not include the CSG component. While conceptually simple, these algorithms are highly dependent on the sampling resolution of test points (or rays). If too few test points are used, these methods may miss small features of an object and produce an incorrect bounding box.

In this paper, we present a simple algorithm for efficiently computing a numerically-optimal ($(1+\epsilon)$-approximation) axis-aligned bounding box for an arbitrary CSG component. The algorithm uses an iterative refinement, called an octree, to tighten an initial, conservatively large, bounding box into the numerically-optimal bounding box for the component. At each stage of refinement, all newly-created octree cells are classified as inside, outside, or intersecting the boundary of the component by using a robust box classification test. In cases where the test cannot unambiguously classify an octree cell, an additional refinement and classification is performed on that cell. This process continues until the refinement near the extremal points of the component has reached the user-selected tolerance level. This refinement/classification approach is more efficient and practical than computing the actual boundary representation of an arbitrary CSG component, while guaranteeing that the component will be completely contained in the bounding box.

At the heart of our algorithm is our main operation, called `classify`, that given an axis aligned box and a component, returns `INSIDE`, `OUTSIDE`, `BOUNDARY` or `UNKNOWN`. The value `UNKNOWN`, is returned when the input to `classify` is too complicated to resolve (i.e., `classify` cannot unambiguously classify a given box with respect to the CSG component). In Section IV, we describe one possible implementation for `classify`, however, other implementations are possible. In fact, there is a strong relationship between the sophistication of the `classify` operation and the number of levels of refinement required to achieve bounding box convergence for a given tolerance. As the `classify` operation becomes stronger, and is able to uniquely classify a higher percentage of boxes, there is less need for additional refinement steps. This dependency provides some flexibility to tailor the implementation of the algorithm to a particular application. Provided that `classify` follows a simple set of invariants, our algorithm convergences, which is shown in Section V.

The main body of the paper is structured as follows: Section II provides a review of the CSG terminology and modeling framework, Section III gives an overview of the general refine/classify algorithm, Section IV describes the `classify` operation in detail, Section V provides a formal proof that the algorithm will converge to a numerically-optimal bounding box, Section VI gives information on practical implementations of the algorithm, and Section VII provides numerical results for the algorithm applied to three representative models. The reader can get a high level idea of our algorithm by reading the description of our input in Section II, the recursive subdivision algorithm in Section III, and the experiments in Section VII, while lower level details and convergence proofs are described in the remaining sections.

II. Input & Problem Statement

In previous work,[(15)] we presented a formalization for the multi-component CSG model commonly used in Monte Carlo transport codes such as MC21[(2)] and MCNP.[(3)] Next, we recall the formalization for the input.

We define a *primitive* as the set of points $\mathbb{R}^3$, with coordinates (x, y, z) satisfying the polynomial inequality

$$\begin{aligned} g(x,y,z) < A_1x^2 + A_2y^2 + A_3z^2 \\ + A_4xy + A_5xz + A_6yz \\ + A_7x + A_8y + A_9z + A_{10}. \end{aligned}$$

A primitive defines a half space whose boundary is a *quadric* (i.e., points where $g(x, y, z) = 0$). A subset of the quadrics define the boundaries of common CSG modeling primitives such as cones, ellipsoids, and planes.

A model is defined by a tree of nodes, called the *model tree*. Each node N stores one parent P, zero or more children and a formula F of regularized* unions and intersections of primitives. Each node defines three *components*, depicted in Figure 1:

The *basic component* $B(N)$ is the region of space defined by F.

* A regularized set operation[(16)] applies the set operation and then takes the closure of the interior. Regularized operations are used to remove lower dimensional features. For example, if two cubes share only a face their intersection is the face but the regularized intersection is empty.

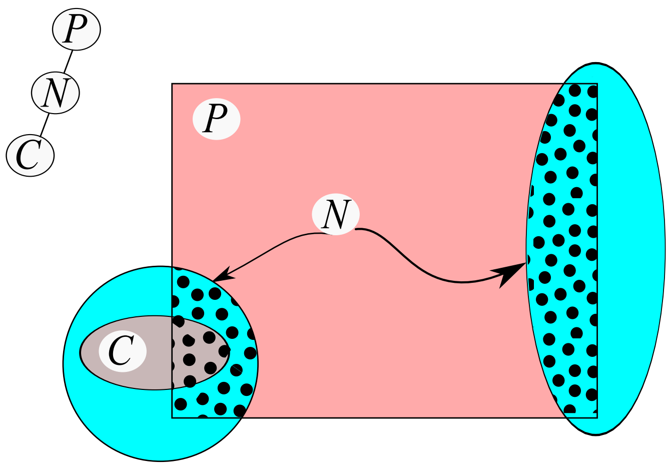

Figure 1: A 2D slice of a model and its model tree. Node N has parent P and child C. The basic component $B(N)$ is disconnected, it is the union two disjoint cyan cylinders. The restricted component $R(N)$ is the black polka dotted region with a cyan or grey background. The hierarchical component $H(N)$ is the black polka dotted region with the cyan background.

The *restricted component* $R(N)$ is the basic component of N intersected with the restricted component of P. That is, $R(N) = B(N) \cap R(P)$.

The *hierarchical component* $H(N)$ is the restricted component of N minus the restricted components of its children.

For the root node N_0, we define the restricted component of N_0 as $R(N_0) = B(N_0)$. The basic component $B(N_0)$ must be bounded, but the basic component for any other nodes may be unbounded. A restricted component $R(N)$ is *connected* if for any two points $p, q \in R(N)$ there exists a path $\rho(p, q)$ from p to q such that every point on $\rho(p, q) \in R(N)$[†].

Observe that the various component definitions gives us an inclusion that is useful for point location and particle tracking algorithms. For a node N with parent P, we have $H(N) \subseteq R(N) \subseteq R(P)$, i.e., each child is contained in its parent's restricted component. The inclusion allows tracking algorithms to avoid some unnecessary tests, a point outside a parent must be outside of all of its children. Since we are interested in computing bounding boxes to accelerate point location and particle tracking we would like our bounding boxes to have a similar inclusion property.

Consider bounding the three component types. The bounding boxes of the restricted components are the best choice because in a valid model, every restricted component is bounded and the bounding boxes of restricted components maintains the inclusion property. The other two types of components are not good choices. First, a restricted component may be unbounded. Second, a hierarchical components may have a bounding box outside of its parent. For example, in Figure 1, the bounding box of the $H(N)$ is outside the bounding box of $H(P)$.

We can compute the bounding boxes for every restricted component of the model tree by traversing the model tree starting from the root and descending:

[†]Typically this is the definition for a path connected set, which in $\mathbb{R}^3$ implies that the set is connected.

Algorithm 1. *First, compute a bounding box for $R(N_0)$. Second, for each remaining node N, take the bounding box of $R(P)$ and "tighten" it to be a bounding box for $R(N)$.*

The main operation of Algorithm 1 is called `tighten`. Since most of the remainder of this paper describes `tighten` and its primitives and `tighten` doesn't need to know anything about the model tree we will sometimes simplify notation and refer to a restricted component C directly. Before we can give the details of the `tighten` we need a few more definitions.

To compute the optimal axis-aligned bounding box for a restricted component C, it is sufficient to compute the extremal points of C in the x-, y-, and z-directions. In practice, it is better to compute a box that is a little looser in order to avoid numerical errors. For an axis-aligned box B, let ∂B be its boundary. Let B^* be the tightest axis-aligned bounding box for a restricted component. An axis-aligned box B is an *ϵ-box of a restricted component* if $B^* \subseteq B$ and for any point $p \in \partial B$ there is a point $q \in \partial B^*$ such that $\|p - q\|_{\inf} \leq \epsilon$.

Now, we can formally state the problem that `tighten` solves:

Problem. *Given a connected restricted component C and an initial bounding box of C, compute an ϵ-box of C.*

We pause to discuss why we focus on restricted components that are connected. Bounding regions are used to accelerate other calculations (in our case, point location, ray tracing, and volume calculation). Consider bounding regions for a point location query in the model depicted in Figure 1. Because the cyan polka dotted restricted component is disconnected, any convex bounding shape is mostly empty, which reduces the benefit of bounding boxes.

Our algorithm, however, will still produce a bounding box for disconnected components, although we lose the ϵ-box guarantee. In Section VI and Section VIII we revisit disconnected restricted components and suggest some ways of handling them in practice.

III. Algorithm

In this section, we describe an algorithm for computing a bounding box for a restricted operation. Before giving the algorithm, we describe one data structure and state our main operation.

An *octree*[(17)] is a tree data structure for representing a spatial subdivision. Each node of the tree represents a box, called an *octree cell*. An internal node N, representing box B, has eight children that represent a subdivision of B. In our algorithm, we will traverse an octree to compute a bounding box.

Our main operation, named `classify`, takes an axis aligned box B and a restricted component C and returns one of four values: `INSIDE`, `OUTSIDE`, `BOUNDARY`, or `UNKNOWN`. The full details of the operation are given in Section IV. We summarize the `classify` operation as follows:

Operation `classify`. *Given a restricted component C and an axis aligned box B, operation* `classify`(C, B) *returns:*

`INSIDE` $\Rightarrow B \subseteq C$

OUTSIDE $\Rightarrow B \cap C = \varnothing$

BOUNDARY $\Rightarrow \exists$ *points* $p, q \in B$ *with* $p \in C$ *and* $q \notin C$.

UNKNOWN $\Rightarrow$ *could not classify.*

Note that the return values of the `classify` operation are not double implications. For example, B may be inside of C, yet the operation returns UNKNOWN. In Figure 2 we see a collection of boxes labeled by `classify`. Note that some boxes are labeled with a U for UNKNOWN even though by visually inspecting the image we can see that the box intersects the triangle's boundary. We will see in Section IV why `classify` cannot resolve the intersection.

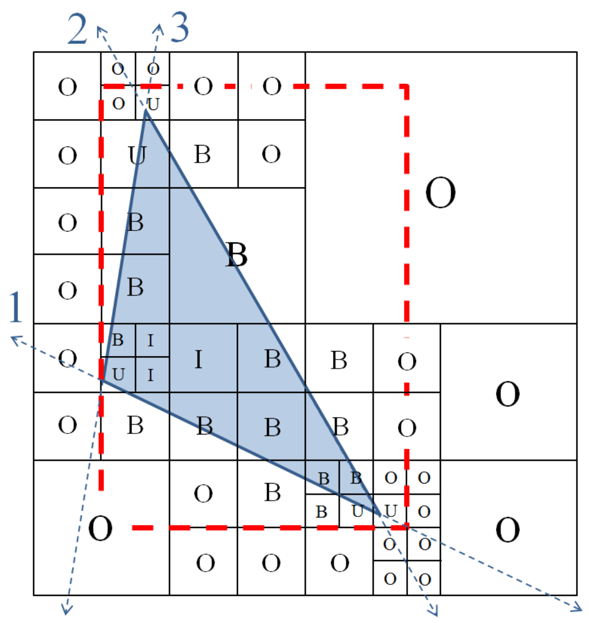

Figure 2: The blue triangle is defined by the intersections of three half-planes with boundaries labeled 1,2, and 3. Boxes labeled O are on the outside of a half plane. For example, the large box in the top right is outside of the half-plane bounded by line 2, so it does not require any subdivision. Boxes closer to multiple lines, such as the boxes near the intersection of lines 1 and 2, require further subdivision to resolve structure.

Next, we describe a simplified version of our our algorithm, called `simple_tighten` that takes an initial box W and a component C and computes a guaranteed bounding box of C. The algorithm does not give a guarantee of an ϵ-box, however, we suggest it in practice because it is simple to implement and for most applications, the box produced works well enough. In Section V, we describe the full details of the algorithm (which does produce an ϵ-box). The biggest difference between the two versions is in computing the stopping condition.

The pseudo-code for the `simple_tighten` is given in Algorithm 2. In Line 2, the initial box W is added to the queue Q. In Lines 4–40 the algorithm traverses the octree, tightening the bounding box. At each iteration, the algorithm pops an octree cell T from Q. The algorithm classifies the cell with respect to C and partitions the cell. Cells classified as OUTSIDE are discarded because they will not give us information about the bounding box. Cells classified as INSIDE are kept because the bounding box must at least enclose them. Cells classified as

Algorithm 2 `simple_tighten`(W, C, ϵ): Given an initial bounding box W, a connected restricted component C and a tolerance ϵ compute a bounding box that is "close" to an ϵ-box for C.

```
1:  // Initialize the queue
2:  $Q.push(W)$
3:
4:  while not $Q.empty$ do
5:      // Get the next cell to process
6:      $T = Q$.pop
7:
8:      // classify $T$ for component $C$ and
9:      // partition the boxes into sets or further traverse octree
10:     switch classify$(C, T)$
11:         case OUTSIDE : discard $T$ end case
12:         case INSIDE : Add $T$ to $I$ end case
13:         case BOUNDARY or UNKNOWN :
14:             if $T.largest\_side > \epsilon$ then
15:                 Subdivide $T$ into eight eight boxes $T_1, \ldots T_8$
16:                 $Q$.push$(T_i)$ for $i = 1, \ldots, 8$
17:             else
18:                 if BOUNDARY then
19:                     Add $T$ to $B$
20:                 else
21:                     Add $T$ to $U$
22:                 end if
23:             end if
24:         end case
25:     end switch
26:
27:     // check for stopping condition
28:     if $Q.empty$ then
29:         $B_{low}$ = optimal bounding box of $I \cup B$
30:         $B_{high}$ = optimal bounding box of $I \cup B \cup U$
31:         $dist$ = distance between $B_{low}$ and $B_{high}$
32:         if $dist < \epsilon$ then
33:             return $B_{high}$
34:         else
35:             $\widehat{U}$ = subdivision of each box of $U$
36:             $Q.push(\widehat{U})$
37:             $U = \varnothing$
38:         end if
39:     end if
40: end while
```

`BOUNDARY` or `UNKNOWN` give us information about the bounding box, but if they are bigger than ϵ they are too big to give useful information. For these cells we subdivide and traverse further down the octree. When cells have all edge lengths smaller than ϵ they are no longer added to Q and Q becomes empty.

Once Q is empty the algorithm checks for convergence in lines Lines 27–39. Let B_{low} be the optimal bounding box of the cells classified as `INSIDE` and `BOUNDARY` and let B_{high} be the optimal box of the cells classified as `INSIDE`, `BOUNDARY`, and `UNKNOWN`. If the distance between B_{low} and B_{high} is less than ϵ, return B_{high}, else, traverse further down the octree by: subdividing the cells classified as `UNKNOWN`, pushing them onto Q, clearing U, and continuing.

Note that this algorithm may not produce an ϵ-box, however the box will still bound C. This could happen if, for example, we identified all cells containing the extremal points for C. Each cell would be inside of B_{low}, which means the optimal bounding box would be inside of B_{low}. If the distance between B_{low} and B_{high} were ϵ, the distance between the B_{high} and the optimal bounding box would be greater than ϵ. In Section V, we will see how a bit more complicated version of this algorithm can be used to guarantee an ϵ-box. In Section VI, we will discuss some other practical decisions that must be made when implementing this algorithm.

IV. Geometric Operations

In this section we describe the `classify` operation, which is used by the algorithms described in Section III and V. The operation is a generalization of `pointLocation`, in which all points in an axis-aligned box are classified simultaneously.

1. The `classify` Operation

In previous work,[15] we defined the `box_classification` operation. In related work, Millman[18] improved the operation and analyzed its precision requirement. The operation takes an axis aligned box B and a primitive s, and returns if B is inside, outside, intersects, or bounds s. We used the `box_classification` operation in the `restriction` construction. The construction takes an axis aligned box B and a boolean formula F, represented a component, and simplifies F inside B. The construction can be summarized in two steps. First, for any surface s_i in F, if B is inside or outside of s_i replace s_i with a true or false, respectively. Second, simplify the formula using a simple set of rewriting rules. The result is a formula describing the component inside the box of only surfaces that intersect B.

Here, we strengthen the `restriction` construction to create the `classify` operation, which in some cases, returns if the box intersects the boundary of a component. Recall the statement of the `classify` operation from Section III:

Operation `classify`. *Given a restricted component C and an axis aligned box B, operation* `classify`(C, B) *returns:*

`INSIDE` $\Rightarrow B \subseteq C$

`OUTSIDE` $\Rightarrow B \cap C = \varnothing$

`BOUNDARY` $\Rightarrow \exists$ *points* $p, q \in B$ *with* $p \in C$ *and* $q \notin C$.

`UNKNOWN` $\Rightarrow$ *could not classify.*

We implement the operation as follows. Given a box B and a formula F, when the `restriction` construction can simplify the boolean formula to an evaluation, we can determine if B is inside or outside the component. In such a case, return `INSIDE` or `OUTSIDE`. If the formula is not simplified to an evaluation, count the number of primitives in the simplified formula. If it is one, B intersects the boundary, return `BOUNDARY`, otherwise, return `UNKNOWN`.

In Figure 2 we see examples of the `classify` operation on a collection of octree cells. Observe that the cells labeled as U for `UNKNOWN` have multiple lines passing though them, while the cells labeled as B for `BOUNDARY` have only one line passing through them.

2. A Stronger `classify` Operation

We could also check if a collection of surfaces that intersect B are the same surface. For most cases, one could check if surfaces s_1 and s_2, defined by quadrics $q_1(x, y, z)$ and $q_2(x, y, z)$, respectively are a scaler multiple of one another. That is, there exists an $\lambda \in \mathbb{R}$ such that $q_1(x, y, z) = \lambda q_2(x, y, z)$. Note, however, that this will not work if s_1 or s_2 are pairs of planes.

For a pair of planes, we suggest constructing two individual planes and treating them separately. This construction factors the quadric representing the pair of planes into into two degree 1 polynomials. Details of this factorization, however, are outside of the scope of this document.

The `classify` operation could be further strengthen by cherry picking special cases that are commonly modeled. For example, if users often model polyhedra, the implementer could write a stronger `classify` that also handles intersections of two and three planes. The stronger operation could then identify boxes containing edges or a vertices of a polyhedra.

In our prototype implementation, however, we preferred a `classify` that is purely combinatorial. By using just the number of surfaces that pass though a box we can encapsulated the numerical calculations (i.e., intersecting a box and surface) in the `box_classify` operation. This maintains a clean separation between the numeric and combinatorial, which is helpful for debugging and avoiding error propagation.

V. Constructing an ϵ-box

In this section, we describe how to extend the algorithm from Section III to compute an ϵ-box for a restricted component.

1. Stopping conditions

Given a restricted component C, initial bounding box W, and a tolerance ϵ the operation `tighten`(C, W, ϵ) iteratively subdivides W until the three conditions, described below, are met. Once the conditions are met `tighten` outputs an ϵ-box of C.

The conditions follow from the observations:

Observation 1. *Given a restricted component C, when computing:*

1. *the optimal axis-aligned bounding box, it is sufficient to compute the extremal points of C in the cardinal directions.*
2. *an ϵ-box, it is sufficient to compute six boxes $B_{-x}, B_{+x}, \ldots, B_{+z}$ (one for each cardinal direction), such that for direction w: the width of box B_w in direction w is less than ϵ, and B_w contains an extremal point of C in direction w.*

In the remainder of this section we describe the `tighten` operation, and then prove that the conditions are sufficient to produce an ϵ-box.

Consider subdividing W into cells of size δ such that $\delta \leq \epsilon$. Label every cell c_i with the `classify`(C, c_i) operation. Partition the cells such that $\mathcal{I}$, $\mathcal{O}$, $\mathcal{B}$, and $\mathcal{U}$ are the set of cells labeled `INSIDE`, `OUTSIDE`, `BOUNDARY`, and `UNKNOWN` respectively. Assume that $\mathcal{I} \cup \mathcal{B}$ is not empty and let β be the optimal bounding box of $\mathcal{I} \cup \mathcal{B}$. Since $\delta \leq \epsilon$ if β bounds C then β is an ϵ-box for C. Next, we will check if β is a bounding box for C.

Partition $\mathcal{U}$ into two sets. Let $\mathcal{U}_{in}$ and $\mathcal{U}_{out}$ be the cells of $\mathcal{U}$ whose interiors are inside and outside of β respectively. Since the cells of $\mathcal{U}_{in}$ are inside of β, they are already bounded, thus, they can be discarded.

Next, we consider the cells of $\mathcal{U}_{out}$. Find each connected region R_i of the cells in $\mathcal{U}_{out}$. We call each R_i a *finger*. Since C is connected, a finger that does not intersect β must be disjoint from C and can be discarded. If all fingers were discarded, $\mathcal{U}_{out}$ is empty, and therefore β is a bounding box for C.

In the fingers, we cannot resolve the structure of C for a cell size of δ. In particular, we cannot determine if a finger contains a point of C. One solution is to grow β to a box $\widehat{\beta}$ containing the fingers. We must be careful, however, if none of the fingers contained a point of C, we may lose the ϵ bound. If we let $k = \lfloor \epsilon/\delta \rfloor - 1$ and grow β by $k * \delta$ in each direction we get a box, named $\widehat{\beta}$. If all fingers are contained in $\widehat{\beta}$ then, as we will show in the next section, $\widehat{\beta}$ is an ϵ-box for C. If $\widehat{\beta}$ is not an ϵ-box, repeat with $\delta = \delta/2$.

We summarize the discussion above by stating the three conditions in which the `tighten` operation produces an ϵ-box for C:

c1: $\delta \leq \epsilon$

c2: $\mathcal{I} \cup \mathcal{B} \neq \varnothing$

c3: all fingers are discarded by β or $\widehat{\beta}$.

Theorem 2. *Given a connected restricted component C, in an arbitrary axis-aligned bounding box W,* `tighten`(C, W, ϵ) *returns an ϵ-box for C.*

Proof. Let δ be the size of the longest edge of a cell. We begin by noting that as we let $\delta \to 0$ each cell converges to a point. Since any point, can be classified as inside, outside or laying on the boundary of the restricted component, as $\delta \to 0$ the box $\beta \to B^*$. Thus, in the limit, `tighten` terminates and produces an optimal bounding box.

Next we show that when `tighten` terminates, it produces an ϵ-box for C. There are two cases for the algorithm to terminate. For all cases, assume (c1) $\delta \leq \epsilon$ and (c2) $\mathcal{I} \cup \mathcal{B} \neq \varnothing$.

First, assume (c1), (c2), and (c3) all fingers are discarded by β. Then, `tighten` reports β as the bounding box. Next, we show that β is an ϵ-box. Without loss of generality, assume p is an extremal point with maximal x-coordinate. Let A be the cell containing p. We will show that β is an ϵ-box by showing that A is on the x-max boundary of β. This implies that the distance from p to the x-max boundary of β is at most ϵ.

The set $\mathcal{I} \cup \mathcal{B} \cup \mathcal{U}$ covers the restricted component and $\mathcal{B} \cup \mathcal{U}$ covers the boundary of the restricted component. As p must be on the boundary of the component, $A \in \mathcal{B}$ or $A \in \mathcal{U}$. If $A \in \mathcal{B}$, by construction, A is on the boundary of β. If $A \in \mathcal{U}$, since all fingers were discarded by β, A must be in $\mathcal{U}_{in}$, which implies $A \subset \beta$. Moreover, A must be on the boundary of β, for if not, there was a box in $\mathcal{B}$ with a point more extreme than p, which is a contradiction. Since the distance from p to its orthogonal projection onto any side of A is at most δ and $\delta \leq \epsilon$, the box β is an ϵ-box for C.

Second, assume (c1), (c2), and (c3) all fingers are discarded by $\widehat{\beta}$. Since the boundaries of $\widehat{\beta}$ are expanded by $(\lfloor \epsilon/\delta \rfloor - 1) * \delta$, the box $\widehat{\beta}$ is an ϵ-box for β. Thus, if any fingers poking out of β contains a point of C then $\widehat{\beta}$ is an ϵ-box of C. If all fingers do not contain any points of C, by the arguments in the previous paragraph, a maximal point p must be contained in a box A on the boundary of β. The distance d, between p and its orthogonal projection on to the x-maximum face is upper bounded by

$$d \leq \delta + (\lfloor \epsilon/\delta \rfloor - 1) * \delta \leq \epsilon.$$

Therefore, $\widehat{\beta}$ is an ϵ-box for C. □

VI. Practial Implementation

While the conditions in the previous section will converge to an ϵ-box there are a few implementation details worth mentioning.

First, one does not actually want to subdivide to a collection of cells of size δ. As in, `simple_tighten`, in Section III, it is better to do a depth first traversal of an octree. In our implementation we do a depth-first traversal to the level in which the longest edge of the octree cell is less than ϵ. Once we have reached this level we can build representations for $\mathcal{I}$, $\mathcal{O}$, $\mathcal{B}$ and $\mathcal{U}$. In fact, one doesn't even need to explicitly store $\mathcal{I}$ or $\mathcal{B}$. Since we only care about β, the bounding box for $\mathcal{I} \cup \mathcal{B}$, we compute β while traversing the octree. Each cell c of $\mathcal{I}$ can be discarded once β is expanded to contain c. Each cell of $\mathcal{B}$ can be discarded once their largest side is smaller than ϵ.

Second, computing and culling fingers is rarely necessary. Often, subdividing a few extra levels reveals enough structure to achieve the requested tolerance. To compute the fingers, one has two options, either maintains neighbor relationships between cells labeled as `UNKNOWN` during the octree traversal or computes the fingers without a $O(1)$ time routine that returns the neighbors of a cell. The first option adds complexity to the implementation, while the second option would require substantial additional computational time.

Third, our convergence proof assumed that we can refine infinitely. In practice, a user may wish to stop subdivision at some level or after some amount of time. We are interested in computing bounding boxes to accelerate particle tracking algorithms. Thus, in our implementation, if we do not compute

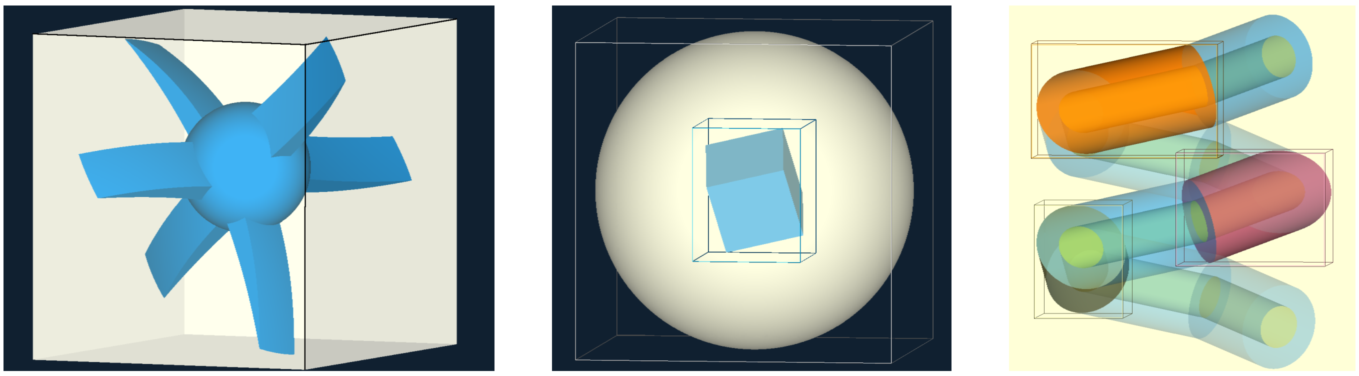
Figure 3: Rendering of three models: *SpikeyBall*, *RotatedCube*, *HelicalPipe20*. *Left: SpikeyBall* **is the union of a ball with six spikes. The spikes are formed by intersections of paraboloids and planes. The bounding box of the ball is displayed.** *Center: RotatedCube* **is a cube with a random rotation inside of a sphere. The vertices and edges of the bounding boxes for the cube and sphere are displayed.** *Right: HelicalPipe20* **is a subsection of helical piping defined by 21 components inside a sphere. The bounding box of three of the model's components are displayed.**

an ϵ-box, we reduce down to a minimum cell size of γ, where γ is determined by the resolution of our tracking algorithm. We return an optimal box $\bar{\beta}$ for $\mathcal{I} \cup \mathcal{B} \cup \mathcal{U}$, so that the box is guaranteed to bound the component, and return the distance between β and $\bar{\beta}$, which bounds the looseness of $\bar{\beta}$.

Fourth, while `tighten` and `simple_tighten` are described to take a connected restricted component as input, `simple_tighten` can also take a disconnected component as input. In addition, while `simple_tighten` may not produce an ϵ-box, with slight modification, it can produce a bounding box, a region containing the optimal bounding box, and an upper bound on the distance between the optimal and reported box.

VII. Experiment

In this section we consider computing bounding boxes for the three models depicted in Figure 3. In all cases, the optimal bounding box of the object was located inside the box with minimal coordinate $(-15, -15, -15)$ and maximal coordinate $(15, 15, 15)$.

SpikeyBall is a ball with six spikes. It has one level in its model tree. Each spike is formed by the intersection of three planes and two paraboloids. The model is formed by taking the union of the six spikes with an ellipsoid.

RotatedCube is a rotated cube inside of a sphere. It has two levels in its model tree. The first level is a sphere defined by one surface. The second level is a randomly rotated cube defined defined by the intersection of six surfaces.

HelicalPipe20 is a section of helical piping inside of a sphere. It has three levels in its model tree. The first level is a sphere. The second level has 10 components. Each component is a capped cylinder defined by the intersection of three surfaces. Each of the 10 components in the second level has a child. The third level, which defines the inner volume of the pipe, has 10 components (one child for each component of the second level). Each component in the third level is defined by the intersection of three surfaces.

All experiments were run in serial on a single core of a 2.6GHz Intel Xeon processor with 48GB RAM.

1. Experiment 1: Comparing Three Models

In the first experiment, we look at times for computing the bounding box of all components in each model at a two different tolerances, 0.5 and 0.05. We start with an initial bounding box with minimal coordinate $(-1000, -1000, -1000)$ and maximal coordinate $(1000, 1000, 1000)$ and reduce down to the bounding box within the specified tolerance. In all cases the bounding boxes were computed to within the specified tolerances. The times for computing bounding boxes are displayed in Table 1.

The first observation is that in all experiments most time is spent reducing from the initial guess of the initial bounding box to the ϵ-box for C0. A second observation is that reducing epsilon seems to increase the amount to time to compute bounding boxes. In the second and third experiments we look further into how the running time is effected by the initial guess of the bounding box and the specified tolerance, respectively.

2. Experiment 2: Varying Initial Bounding Box Size

In the second experiment we investigate how varying the initial bounding box effects the running time. We say that an initial bounding box has a size b if it has minimal coordinate $(-b, -b, -b)$ and maximal coordinate (b, b, b). In this experiment, we fix the tolerance at $\epsilon = 0.1$, and vary the size of the initial bounding box. We consider ranges from 50-100 in increments of 10, 100-1000 in increments of 100, and 1000-50000 in increments of 1000. For each initial bounding box, we compute the bounding box for the spikey ball (C0) in the *SpikeyBall* model, the rotated cube (C1) in the *RotatedCube* model and the bounding sphere (C0) in the *HelicalPipe20* model. The timings are plotted in Figure 4.

The first observation is that as the size of the initial bounding box increases, the amount of time for computing the bounding box increases. However even when reducing the box by four orders of magnitude, the total time to compute a bounding box is under six seconds for the spikey ball, under two seconds for

Table 1: Times for computing bounding boxes for each component in the *SpikeyBall*, *RotatedCube*, and *HelicalPipe20* models. In *SpikeyBall*, the ball is C0. In *RotatedCube*, the bounding sphere is C0 and the rotated cube is C1. In *HelicalPipe20*, the bounding sphere is C0, the 10 pipes are C1-C10, and the inside of the pipes are C11-C12. Hierarchy is depicted by spacing. For example, in *RotatedCube*, C1 is a child of C0, and in *HelicalPipe20*, C1-C10 are children of C0, C11 is a child of C1, and C12 is a child of C2.

Component ID			Time (s) for	
			$\epsilon = 0.5$	$\epsilon = .05$
SpikeyBall				
C0			0.60	1.67
RotatedCube				
C0			0.02	0.10
	C1		<.01	<.01
Total			*0.02*	*0.10*
HelicalPipe20				
C0			0.13	1.62
	C1		0.02	0.25
		C11	0.03	0.19
	C2		0.02	0.39
		C12	0.05	0.36
	C3		0.02	0.63
		C13	0.03	0.22
	C4		0.02	0.30
		C14	0.04	0.28
	C5		0.02	0.45
		C15	0.05	0.46
	C6		0.02	0.23
		C16	0.03	0.20
	C7		0.03	0.41
		C17	0.05	0.35
	C8		0.02	0.59
		C18	0.03	0.21
	C9		0.02	0.26
		C19	0.04	0.27
	C10		0.03	0.44
		C20	0.05	0.42
Total			*0.75*	*8.53*

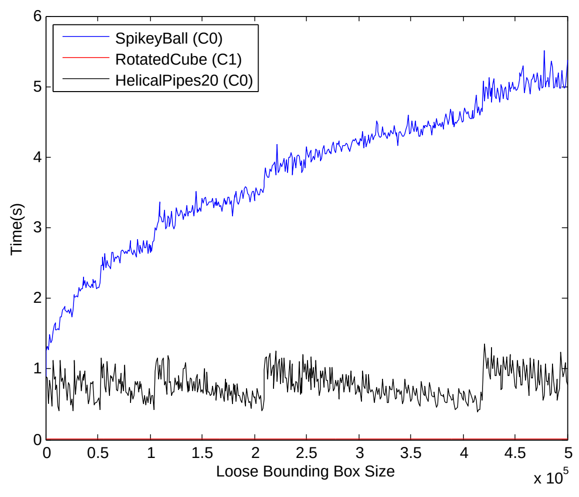

Figure 4: A plot comparing the time to compute a bounding box to a tolerance of $\epsilon = 0.1$ from varying initial bounding boxes for three examples.

the sphere bounding the helical pipes and less that 0.1 seconds for the rotated cube.

The second observation is that the time for computing models is dependent on the model. A curve fit of the times for *SpikeyBall* seems to indicate that the time for computing the bounding box grows at $O(\sqrt{n})$ where n is the size of the initial bounding box. The curve for *RotatedCube* is bounded between 0.40-1.36 seconds over the tested bounding box sizes.

The jumps in the function, for example *HelicalPipe20* between 4.5×10^5 and 5×10^5, are caused by the algorithm recursing one level deeper into the octree to achieve the tolerance of $\epsilon = 0.1$. Recall that a bounding box cannot be computed until the size of the boxes are at least the tolerance. Consider, for example, the jump between 4.5×10^5 and 5×10^5. For *HelicalPipe20* it takes 0.57 seconds for a initial bounding box of 4.19×10^5 and 1.04 seconds for an initial bounding box of 4.20×10^5. When started with an initial bounding box of 4.19×10^5 the algorithm must go down at least 23 levels in the octree where as 4.20×10^5 must go at least 24 levels.

3. Experiment 3: Varying Tolerance

In the third experiment we look at how varying the tolerance effects the running time. Starting from an initial bounding box with minimal coordinate $(-1000, -1000, -1000)$ and maximal coordinate $(1000, 1000, 1000)$ we compute bounding boxes to a tolerance between 0.001 and 1.0 using an increment of 0.001. As in the second experiment, we look at computing the bounding box for the spikey ball (C0) of the *SpikeyBall* model, the rotated cube (C1) of the *RotatedCube* model and the bounding sphere (C0) of the *HelicalPipe20* model. The timings are plotted in Figure 5.

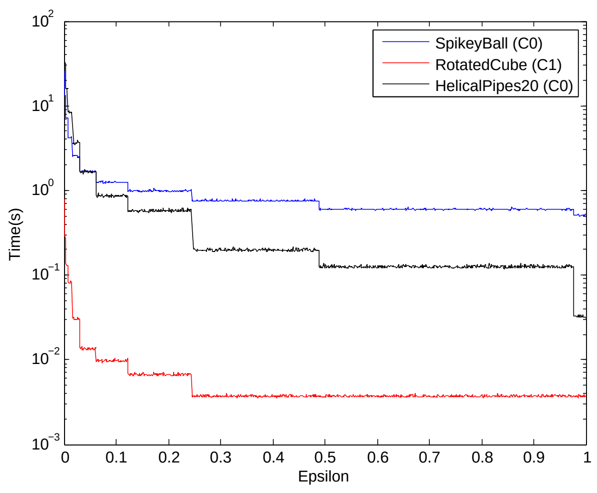

Figure 5: A plot comparing the time to compute a bounding box, from an initial bounding box with minimal coordinate $(-1000, -1000, -1000)$ and maximal coordinate $(1000, 1000, 1000)$ for varying tolerances for three examples.

The first observation is that as we compute ϵ to smaller values, the time increases. In all three examples, a curve of $O(1/\epsilon)$ seems to have good fit. We also observed, however, that there is a step in the three examples, e.g., when ϵ is 0.125 and 0.25. This is expected. For many adjacent values of ϵ, e_1 and e_2, the difference is only 0.001 the same bounding box that satisfies e_1 often satisfies e_2. The steps occur when a looser bounding box will suffice. Thus, the steps actually occur at values of ϵ where $\epsilon = 1000/2^i$. Indeed, the step around 0.25 is caused by the octree traversal only descending 11 levels ($1000/2^{11} \approxeq 0.488$) as opposed to 12 levels ($1000/2^{12} \approxeq 0.244$).

VIII. Conclusion

In this paper, we presented a robust and efficient algorithm for generating the numerically-optimal bounding box (optimal to within a user-specified tolerance) for an arbitrary CSG object defined by quadratic surfaces. The new algorithm uses a refinement/classification approach to tighten an initial, conservatively large, bounding box into the numerically-optimal bounding box. The proposed refinement/classification approach is more efficient and practical than methods that rely on computing actual boundary representations of an arbitrary CSG component, while guaranteeing that the component will be completely contained in the bounding box. To our knowledge, the proposed algorithm is the first practical method that is robust enough for routine use for calculating bounding boxes of CSG components in Monte Carlo particle transport codes. This ability to produce guaranteed bounding boxes for CSG components will enable improvements in ray tracing performance for Monte Carlo particle transport simulations. Moreover, similar to Gottschalk *et al.*,[1] one can then use the bounding boxes to produce trees or directed acyclic graphs, from the boxes to avoid processing components that are far away from a region of interest.

At the heart of the new algorithm is the box classification test, which determines if a given box is inside, outside, or intersects the boundary of a CSG component. If the test is not able to unambiguously classify a box it returns an unknown status, which signals the algorithm to perform additional refinement. In general, there is a trade-off between refinement and sophistication of the box classification that allows the implementer to tailor the box classification as needed. In this paper, we described a simple box classification that is purely combinatorial. The classification allows us to encapsulate the numerical calculation of intersecting surfaces with boxes to avoid error propagation.

We also proved that for a connected restricted component, our complete algorithm produces a numerically-optimal bounding box. In practice, however, we recommend implementing the simpler version of the algorithm described in Section III because it is easy to implement and it always produces a valid bounding box as well as a bound on the distance δ from the optimal box. While in theory, δ may sometimes be larger than the specified tolerance, we have not seen this in practice.

The proposed algorithm for generating bounding boxes was tested on several representative CSG components of varying complexity. In each test, the algorithm was able to compute a valid bounding box to within the specified tolerance of the optimal bounding box. As expected, the cost to generate a bounding box is dependent on the complexity of the component being bounded. During testing, the algorithm was able to compute a numerically-optimal (for $\epsilon = 0.5$) bounding box for a simple component in 0.02 seconds, a pathologically complex component in 0.65 seconds, and a collection of 20 hierarchically-arranged components in 0.75 seconds (0.0325 s/component). Furthermore, the results assumed an initial bounding box guess that was over four orders of magnitude larger than the optimal bounding box for each component. Reducing the size of this initial bounding box guess will increase the speed of the algorithm accordingly. These timing results suggest that the algorithm could be used as a routine pre-processing calculation to generate bounding boxes for all CSG components in Monte Carlo models, which, in turn, could provide a significant increase in particle tracking speed.

IX. Acknowledgements

The authors would like to thank Stephen C. Wilson of the Bettis Laboratory for his help creating the *HelicalPipe20* model. Dr. Snoeyink acknowledges support from a contract with the Bettis Laboratory and from a NSF research grant.

References

1) S. Gottschalk, M. C. Lin, and D. Manocha, "OBBTree: A Hierarchical Structure for Rapid Interference Detection," *Proc. SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, 1996.
2) D. P. Griesheimer et al., "MC21 v.6.0 – A Continuous-Energy Monte Carlo Particle Transport Code with Integrated Reactor Feedback Capabilities," *Proc. Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo (SNA + MC)*, 2013.
3) X-5 Monte Carlo Team, "MCNP – A General Monte Carlo *n*-Particle Transport Code Version 5," LA-UR-03-1987, Los Alamos National Laboratory (2003).
4) N. Candelore, R. Gast, and L. Ondis II, "RCP01 - A Monte Carlo Program for Solving Neutron and Photon Transport Problems in Three-Dimensional Geometry with Detailed Energy Description," WAPD-TM-1267, Bettis Laboratory (1978).
5) "POV-Ray, Persistence of Vision Raytracer (Version 3.6)," http://www.povray.org.
6) J. O'Rourke, "Finding Minimal Enclosing Boxes," *International Journal Computing and Information Sciences*, **14**, 183–199 (1985).
7) G. Barequet and S. Har-Peled, "Efficiently Approximating the Minimum-Volume Bounding Box of a Point Set in Three Dimensions," *Journal of Algorithms*, **38**, 91–109 (2001).
8) M. C. Lin and S. Gottschalk, "Collision Detection Between Geometric Models: A Survey," (1998).
9) L. Dupont, D. Lazard, S. Lazard, and S. Petitjean, "Near-optimal parameterization of the intersection of quadrics: I. The generic algorithm," *Journal of Symbolic Computation*, **43**, *3*, 168–191 (2008).
10) E. Schömer and N. Wolpert, "An exact and efficient approach for computing a cell in an arrangement of quadrics," *Computation Geometry Theory and Applications*, **33**, *1-2*, 65–97 (2006).
11) J. Keyser, *Exact Boundary Evaluation for Curved Solids*, PhD thesis, University of North Carolina–Chapel Hill, 2000.
12) J. Keyser, T. Culver, M. Foskey, S. Krishnan, and D. Manocha, "ESOLID—A System for Exact Boundary Evaluation," *Proc. Proceedings of the Seventh ACM Symposium on Solid Modeling and Applications*, p. 23–34, ACM Press, 2002.
13) J. Keyser, S. Krishnan, and D. Manocha, "Efficient and Accurate B-Rep Generation of Low Degree Sculptured Solids Using Ex-

act Arithmetic: I-Representations," *Computer Aided Geometric Design*, **16**, *9*, 841–859 (1999).

14) J. Keyser, S. Krishnan, and D. Manocha, "Efficient and Accurate B-Rep Generation of Low Degree Sculptured Solids Using Exact Arithmetic: II-Computation," *Computer Aided Geometric Design*, **16**, *9*, 861–882 (1999).
15) D. L. Millman, D. P. Griesheimer, B. R. Nease, and J. Snoeyink, "Robust Volume Calculations for Constructive Solid Geometry (CSG) Components in Monte Carlo Transport Calculations," *Proc. PHYSOR: Advances in Reactor Physics*, 2012.
16) A. G. Requicha, "Representations for Rigid Solids: Theory, Methods, and Systems," *ACM Computing Surveys*, **12**, *4* (1980).
17) H. Samet, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys*, **16**, *2*, 187–260 (1984).
18) D. L. Millman, *Degree-Driven Design of Geometric Algorithms for Point Location, Proximity, and Volume Calculation*, PhD thesis, University of North Carolina–Chapel Hill, 2012.