A kd-tree algorithm to discover the boundary of a black box hypervolume or how to peel potatoes by recursively cutting them in halves

Jean-Baptiste Rouquier, Isabelle Alvarez, Romain Reuillon, Pierre-Henri Wuillemin

A KD-TREE ALGORITHM TO DISCOVER THE BOUNDARY OF A BLACKBOX HYPERVOLUME OR HOW TO PEEL POTATOES BY RECURSIVELY CUTTING THEM IN HALVES

JEAN-BAPTISTE ROUQUIER*, ISABELLE ALVAREZ†, ROMAIN REUILLON‡, AND PIERRE-HENRI WUILLEMIN§

Abstract. Given a subset of $\mathbb{R}^n$ of non-zero measure, defined through a blackbox function (an *oracle*), and assuming some regularity properties on this set, we build an efficient data structure representing this set. The naive approach would consists in sampling every point on a regular grid. As compared to it, our data structure has a complexity close to gaining one dimension, both in terms of space and in number of calls to the oracle. This data structure produces a characteristic function (i.e. a function that can be used in lieu of the oracle), allows to measure the volume of the set, and allows to compute the distance to the boundary of the set for any point.

Key words. kd-tree, octree, quadtree, blackbox, oracle, hyper-surface, hyper-volume, numerical integration, boundary approximation

1. Introduction.

1.1. Motivation. Viability theory is a set of mathematical and algorithmic methods proposed for analyzing the evolutions of controlled dynamical systems inside a set of admissible states K, called the viability constraint set ([1]). The research described in this paper has been motivated by a problem that arose when numerically exploring the phase space of a complex (for instance biological) dynamic model in such viability analysis, as in [12]. The main concepts of the viability theory are the following:

- Viable state: A point in K is called *viable* if there exists at least one control function such that a trajectory starting from that point and following this control function remains in K until time T (or indefinitely). See Figure 1.1.
- Viability kernel: The set of all viable states is called the viability kernel and is denoted $Viab(K)$.

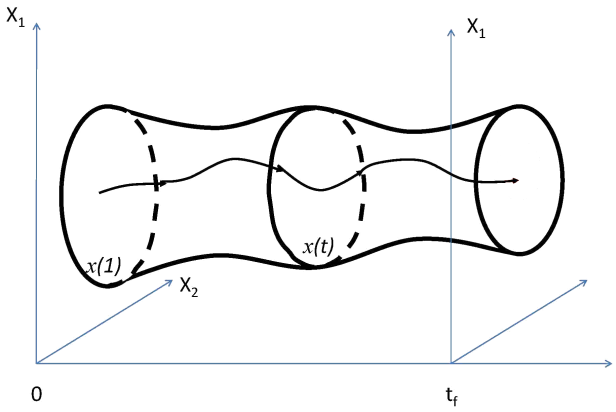


FIGURE 1.1. *A trajectory inside a viability kernel.*

A frequent problem that occurs during viability analysis is the computation of the viability kernel and its representation since the complexity of these tasks is exponential with space or time for complex dynamical systems ([10], [4]).

In this paper, we focus on a method to represent the viability kernel. In order to allow application to various models and ensure maximum generality, we consider the dynamical model

*Eonos, 53 rue La Boëtie, 75008 Paris, France. jean-baptiste.lastname@ens-lyon.fr
†LISC-Irstea, UPMC-LIP6, 4 place Jussieu, 75252 Paris cedex 05. isabelle.lastname@lip6.fr
‡Institut des Systèmes Complexes, 57-59 rue Lhomond, 75005 Paris, France. firstname.lastname@iscpif.fr
§UPMC-LIP6, 4 place Jussieu, 75252 Paris, France. pierrehenri.lastname@lip6.fr

as a blackbox that maps one point to the point at the next time step. Given the set at one time step, we can tell if a point is viable at the previous time step simply by testing if its image by the blackbox is within the current set.

We call f the function that tells if a point is viable (the oracle), and V the set of viable points at a given time step. We thus need a way to

- represent the sets encountered at each time step, starting with the final states at time horizon T,
- given one set and the model, compute the set V at the previous time step.

The naive way would be to sample the phase space along a (fine) grid, and store the coordinates of all viable points, but this is costly in terms of time and memory, especially because the number of variables, i.e. the dimension n, can be high. Indeed, each point of the phase space, or search space, is a possible configuration of those n variables.

Instead of this representation in extension of the set V, we develop compact representation (in intension) of the function f.

The field of machine learning gives another point of view on this problem. Indeed, our problem consists in learning a classification function that is compatible with an oracle f. More precisely, this problem can be seen as an active learning problem as in [7], where the computation of the oracle function f is considered very costly. The authors optimize the choice of examples to label with the oracle function. But in their settings, examples are to be chosen among a given finite set.

Our case is special in the following respect:

1. The oracle f is complete: in other words, a hypothesized learning database would contain all points of the search space. However, the computation of f may be resource intensive, thus we need to minimize the number of calls to the oracle. In that sense, our problem is close to the domain of active learning [11], i.e. the learning process implies a sub-task of choosing the samples to be labeled by the oracle. We can choose any point of $\mathbb{R}^n$, we are not restricted to a given set of vectors.
2. We need to provide a guarantee on generalization. Precisely, we bound a distance between the oracle and our learned function: if a point P is classified as viable, there is a viable point at most at a constant (independent from P) distance. Nearest neighbour classification also gives this type of bound, but the bound is not constant (it depends on the point). Statistical methods classically provide a bound *on average*, while we need a bound for each point.
3. As opposed to methods that are tolerant on outliers, we look for an exact method. Indeed, as we will chain the learned functions (i.e. we will iterate our algorithm), one learned function being the oracle for the next one, we want to propagate the guarantees, transitively.

In the framework of viability theory, we can take into account the properties of V induced by the properties of the set of constraints K. In particular, when the dynamic is sufficiently regular (for example such that the boundary ∂V is differentiable) and when K is a simply connected closed set, then V is also a simply connected closed set. The exact properties we assume are formalized in Section 4, which allows us to provide some guarantees on generalization.

The same problem has been addressed in [5], using SVM to encodes the boundary of $V(t)$. Unfortunately, it is not possible with SVM to have any guarantee on the distance between the boundary of V and the boundary of the SVM. To fulfill the need for a guarantee, we suggest to use a well-know data structure for the representation of a set of points in high dimension: the *kd*-tree.

1.2. Representing f with *kd*-trees. *Kd*-trees, proposed by [2], are a data structure designed to store a set of points in a n-dimensional space. It is a binary search tree where each node represents a region of the space. Each node is either a leaf or has its region partitioned in two, each sub-region being associated to a child. This partition of the node region is always done along one of the n dimensions. To find which leaf region contains a given point P, one simply descends the tree, choosing at each node the very child representing the region containing P.

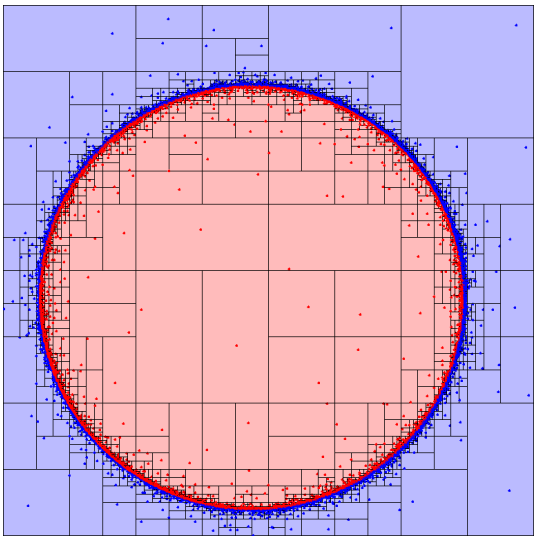
Kd-trees have been used to store a dynamic set of points and tell if a given point belongs

to the set, find the nearest neighbor of a given point, find all points in a given portion of the space, etc. See [8] for a more detailed introduction.

There are numerous applications to *kd*-trees, including databases [3], clustering [6], or geometry: computer graphics (especially ray-tracing) [14] and collision detection. While the former applications are about a finite set of points, our approach is closer to the geometry application: here, the tree is used to represent a continuous set of points, i.e. a 3D volume, or in our case a hyper-volume. To do so, the tree represents the boundary of this set, i.e. a hyper-surface.

Hence, our approach is to subdivide the search space with a *kd*-tree that is fine on the boundary of V and coarse elsewhere. This *kd*-tree yields a guaranteed approximation of the boundary: we provide a set containing V and a set contained in V. Figure 1.2 illustrates the use of a *kd*-tree to approximate V, focusing on its boundary.

The main difference between this problem and the application of *kd*-trees to computer graphics is that in the latter the boundary is known, whereas in our case we want to discover this boundary by sampling the n-dimensional space. A close problem is addressed in [9] where the aim is to discover an unknown iso-surface from a set of measurements, each measurement consisting in 3D coordinates and an intensity. But those 3D coordinates are imposed, while in our setting we can choose the sampling points.



- Viable points are red,
- non viable points are blue,
- the ground truth is in the background (light red or blue),
- divisions made by the algorithm are black,
- points used to label the zones are in bright red or blue.

FIGURE 1.2. *Example run of our algorithm for a viable 2D-disk (color online).*

2. Definitions and notations.

2.1. Problem description.

Formally, the inputs of our problem are

1. a *search space*, namely a hypercube of dimension n and side length c,
2. a blackbox function $f : [0, c]^n \to \{0, 1\}$.

The output is a characteristic function g that approximates f. Building g is done by the user as a preprocessing step; we try to minimize the number of calls to f during this step. Then, the user can call g on many $P \in [0, c]^n$ (to label P as viable or not), while f is not needed any more. Indeed, evaluating $g(P)$ is much faster than $f(P)$, which accelerate viability algorithms that were limited by the evaluation time of f.

A naive approach to build g would be to sample f on every point of a regular grid of step ε. But in practical applications, this has proven to be too costly in terms of space and number of calls to f. Instead, we propose to build g as a *kd*-tree. Each node in the *kd*-tree requires exactly one call to f to be built. Thus, the complexity, both in terms of space an in number of model calls, is the number of nodes in the tree.

Recall that $V = \{P \in [0, c]^n, f(P) = 1\}$; the points of this set are called *viable*. Our algorithm requires that V is sufficiently regular. Precisely, we assume[1] that V is a bounded

[1]Section 4 gives a more formal specifications of these assumptions.

simply connected set, and has no "too thin tentacles" since our approximation algorithm could miss the regions connected by these tentacles.

2.2. Notations. In this paper, we use these notations:

- n the dimension of the search space,
- c the length of one side of the hypercube bounding the search space,
- $f : [0, c]^n \to \{0, 1\}$ the oracle.
- $g : [0, c]^n \to \{0, 1\}$ the approximation of f that we build,
- $V = \{P \mid f(P) = 1\}$ the input set to approximate, $V^{\complement}$ the set of non-viable points,
- $W = \{P \mid g(P) = 1\}$ the set of points labeled as viable by our algorithm, i.e. the set of points belonging to a leaf labeled as viable,
- ∂V the boundary of V, ∂W the one of W,
- ε the parameter for the stopping criterion, explained in the next section. The user should choose ε to ensure the regularity properties of V, see Section 4.

2.3. Sketch of the method. The *kd*-tree is a way to divide the search space into *zones*. Each node of the binary tree is associated with a zone, and the root node is associated with the whole search space. If a node is an internal node (i.e. not a leaf), then its associated zone is divided in two halves along one dimension. The two resulting zones are associated with the children of the node.

Note that, as opposed to the usual *kd*-trees, a zone is always divided in the middle.

In each zone, a point is drawn at random, labeled as viable or not thanks to f, and this label is used to label the whole zone. Then, computing $g(P)$ is simply finding the leaf zone containing P and returning its label.

To build the tree, the algorithm consists in refining the zones that need to, until the desired precision ε. Precisely, we divide the pairs of zones that are adjacent, undivided, and have opposite label. By adjacent, we mean "distinct and sharing a common border of dimension $n - 1$". We call those pairs *critical pairs*.

We call MDN (for "Maximally Divided Node") a node that will not be divided anymore, because it meets the stopping criterion: all its sides are of length at most ε. (An MDN is thus necessarily a leaf.) At the end of the execution, all critical pairs are composed of two MDN.

3. Algorithm. As sketched in the previous section, the algorithm to build the *kd*-tree is simply:

1. Start with a tree of only one node. We require a viable point as input, this point is used to label the root node.
2. Find the critical pairs (see below).
3. Divide once each zone belonging to a critical pair, unless it is an MDN. A zone is always divided according to its longest dimension.
4. If no zone has been divided, stop, else iterate from point 2.

Note that even if a zone belongs to several critical pairs, it is divided only once per iteration.

To break ties in Step 3, one can either follow a cyclic order among directions (as in usual *kd*-trees) and gain some storage space (since there is then no need to store the division direction), or choose the direction of adjacency (see Section 3.1.1), which experimentally yields a small complexity gain (roughly 4% less nodes, according to first experiments).

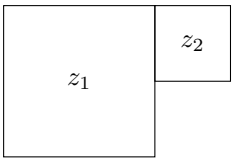
The hard part is step 2, which we now explain.

3.1. Find critical pairs. The algorithm to find critical pairs is composed of three functions, described in Sections 3.1.2 to 3.1.4. Those functions are recursive and walk down the tree from the root to the leaves.

But first, we need to describe how to determine if two zones are adjacent.

3.1.1. How to determine if two zones are adjacent.

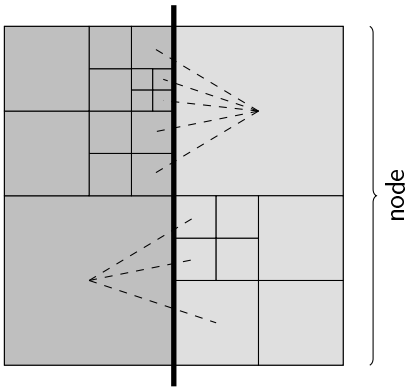
A zone is represented as a vector of n half-open intervals. Those intervals are computed by successive divisions: one starts with $[0; c)$ and divides in the middle recursively. In particular, two of those intervals can



never overlap: either one is included (or equal) in the other, or they are disjoint.

Two zones (z_1, z_2) are adjacent if and only if, on each coordinate, one interval is included (or equal) in the other, except for one coordinate where both intervals are adjacent (their closures have exactly one common point). The unit vector normal to the hyperplane containing the common border (i.e. the intersection of their closures) is called the *adjacency direction*. It is oriented from z_1 to z_2. It is the base vector associated to this coordinate, or its opposite, depending on which interval is to the left of the other.

3.1.2. Find critical pairs in a subtree.



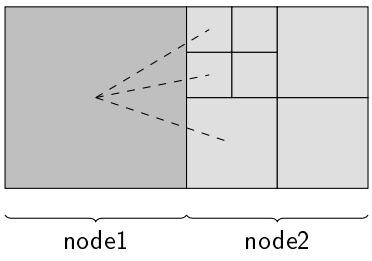
The function pairs_in_node(node) returns the critical pairs in the subtree rooted at node. This is the main function: the user will simply call it on the root of the *kd*-tree.
As an example, on the opposite figure where viable zones are gray and non viable ones are light gray (i.e. the thick line represents ∂W), this function returns the critical pairs represented by dashed lines.
See Algorithm 1 for implementation. One auxiliary function is needed, namely pairs_between_nodes(node1, node2), which we now describe.

Algorithm 1: pairs_in_node(node)

```
if node is a leaf then
    return ∅
else
    (node1, node2) := node.children
    result :=∅
    result ∪= pairs_in_node(node1)
    result ∪= pairs_in_node(node2)
    result ∪= pairs_between_nodes(node1, node2)
    return result
```

Inspired by the notation $x \mathrel{+}= y$ of the language C, the notation $X \cup= Y$ means $X := X \cup Y$.

3.1.3. Find critical pairs between two given nodes.

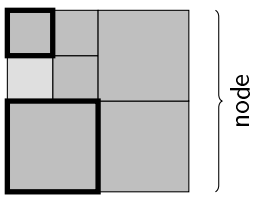


The function pairs_between_nodes(node1, node2) returns the critical pairs (z_1, z_2) such that z_1 is a descendant of node1 (or node1 itself), and z_2 a descendant of node2 (or node2 itself). It thus assumes that node1 and node2 are adjacent.
The opposite figure has the same conventions as the previous one.
See Algorithm 2 for implementation. One auxiliary function is needed, namely borders_of_node(node, direction, label), which we now describe.

3.1.4. Find nodes next to a border. The function borders_of_node(node, direction, label) returns the list of descendants of node, with label label, which furthermore are on the border of node in direction direction.

For instance, on the opposite figure, borders_of_node(node, west, 1) returns the zones drawn with thick borders (but in this example, not the light gray zone, since its label is not 1).

See Algorithm 3 for implementation.



Algorithm 2: pairs_between_nodes(node1, node2)

```
result := ∅
if node1 and node2 are leaves then
    if node1.label ≠ node2.label then
        result := {(node1,node2)}
    else result := ∅
else if node1 and node2 are internal nodes then
    foreach pair of adjacent nodes (node1', node2') where node1' is a child of node1 and
    node2' a child of node2 do
        result ∪= pairs_between_nodes(node1', node2')
else if node1 is a leaf and node2 an internal node then
    label := 1 − node1.label
    direction := adjacency direction between node1 and node2
    foreach node' in borders_of_node(node2, direction, label) do
        if node1 and node' are adjacent then result ∪= {(node1, node')}
else if node2 is a leaf and node1 an internal node then
    /* this is symmetrical to the previous case                                    */
    /* (in particular, direction is the opposite)                                  */
    label := 1 − node2.label
    direction := adjacency direction between node2 and node1
    foreach node' in borders_of_node(node1, direction, label) do
        if node2 and node' are adjacent then result ∪= {(node2, node')}

return result
```

Algorithm 3: borders_of_node(node, direction, label)

```
if node is a leaf then
    if node.label = label then
        return {node}
    else
        return ∅
else
    foreach child node' of node touching the boundary of node in direction direction (if
    node is divided in direction direction there is only one such child, else both children of
    node are such nodes) do
        result ∪= borders_of_node(node', direction, label)
    return result
```

3.1.5. Usage. The entry point of this algorithm is the main function pairs_in_node. The user will thus simply call pairs_in_node on the root node.

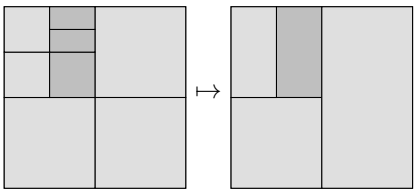
We assume that points outside the search space are not viable. Thus, also call borders_of_node(root,direction,1) for each direction, to find viable zones on the border of the search space, that also need to be divided.

Alternatively, one can decide to extend the search space whenever there is a viable zone on the border of the search space. Note that even if there is no such zone at one point in the execution, dividing a zone can make such a zone appear. To test if the search space must be extended, note that there is such a zone in direction direction if and only if borders_of_node(root,direction,1) returns a non empty list. To extend the search space, replace the root node by a new node having two children: the original root and a new leaf node, associated to a zone as big as the zone of the original root node, and adjacent to it.

If one chooses to extend the search space, one must ensure that the viable set is bounded, to ensure termination of the algorithm.

3.2. Clean the tree and produce a characteristic function. Now, to compute $g(P)$, we use the *kd*-tree as a decision tree: one simply follows a branch of the tree, at each node choosing the children that contains P. Once a leaf is reached, the label of the leaf is returned.

The tree produced by the algorithm can have a few divisions that are not necessary to compute g. It is a small optimization to clean the tree, and it is necessary for section 6.2. Cleaning the tree simply consists in, while a node has two leaf children of the same label, replacing this node with a leaf of that label. The cleaned tree clearly yields the same characteristic function.



4. Proof of correctness. We now prove that g is indeed an approximation of f: g converges to f in the sense of Theorem 4.1.

Let $B(P,\varepsilon)$ be the ball of center P and radius ε. We assume the following properties:

H1: A point $M \in V$ is known.
H2: $\Theta := \{P \mid B(P,\varepsilon\sqrt{n}) \subseteq V\}$ is path connected.
H3: Each point of V is at a distance at most εn of a point of Θ.

The same conditions are assumed for $V^{\complement}$. Additionally we assume that V is simply connected.

H1 is used to initialize the algorithm. **H2** and **H3** ensure that V has no thin tentacles (possibly leading to a large region).

The gap from $\varepsilon\sqrt{n}$ to εn between **H2** and **H3** allows V to be a rectangular parallelotope (e.g. a rectangle if $n = 2$, a rectangular parallelepiped if $n = 3$). Using $\varepsilon\sqrt{n}$ instead of εn in **H3** would impose rounded corners with radius $\varepsilon\sqrt{n}$.

We now prove that the boundary as seen by the algorithm, ∂W, and the real boundary ∂V are not far from each other:

THEOREM 4.1. *At the end of the execution,*

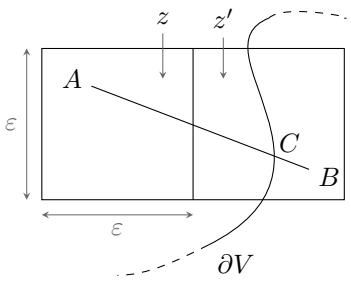
1. *any point of ∂W is at distance at most $\varepsilon\sqrt{n}$ of ∂V;*
2. *any point of ∂V is at distance at most εn of ∂W.*

Section 4.1 proves the first claim, Section 4.2 the second one.

4.1. Final critical zones are not far from the boundary.

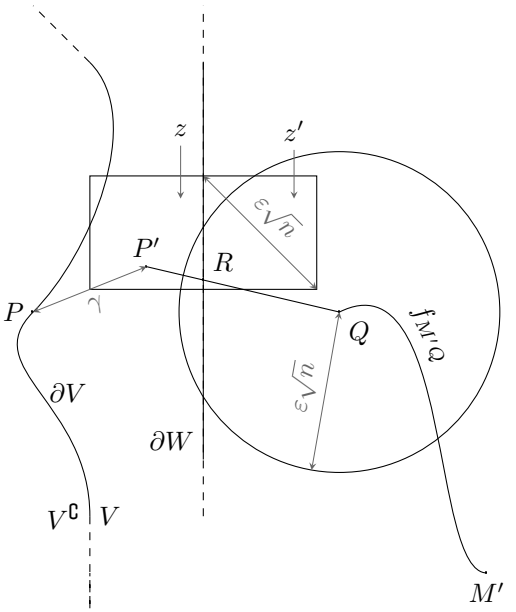
Let us consider a critical pair (z, z'). z and z' have have all their sides of length ε. Let A and B be the points from which z and z' are labeled. Since (z, z') is a critical pair, $g(A) \neq g(B)$, so there exists a point $C \in [A, B]$ that belongs to ∂V. Since $C \in z \cup z'$, any point of the boundary between z and z' is at distance at most $\varepsilon\sqrt{n}$ of C.

Note further that any point of a critical zone is at distance at most $\sqrt{(n-1)\varepsilon^2 + (2\varepsilon)^2}$ of C, i.e. at distance at most $\varepsilon\sqrt{n+3}$ of ∂V.



4.2. Points of the boundary are not far from a final critical zone.

Let P be a point of ∂V. If $P \in \partial W$ then the claim is established.
Else, there is $\gamma > 0$ such that $\gamma < d(P, \partial W)$. Since $P \in \partial V$, there is a point $P' \in V$ such that $d(P, P') < \gamma < d(P, \partial W)$, i.e. such that $g(P) = g(P')$.
By symmetry, we can assume without loss of generality that $P \notin W$. By hypothesis **H3**, there is a point $Q \in \Theta$ such that $d(P', Q) \leqslant \varepsilon n$. Thanks to **H1**, there is a point M such that $g(M) \neq g(P')$. Thanks to **H3**, there is a point $M' \in \Theta$ such that $g(M') = g(M) \neq g(P)$.
Thanks to **H2**, Θ is path-connected, so there is a path $f_{M'Q} \subseteq \Theta$ from M' to Q. Concatenating it with the segment $[QP']$ yields a path from M' to P'. Since $g(P') \neq g(M')$ there is a point $R \in \partial W$ on this path.
Let (z, z') be the critical pair containing R, i.e. $g(z) \neq g(z')$, z and z' are zones with all sides of length ε, and R belongs to the boundary of both z and z'. Since z and z' have all their sides of length ε, $z \cup z' \subseteq B(R, \varepsilon\sqrt{n})$.



Let us assume for a moment that $R \in f_{M'Q}$. This implies $R \in \Theta$ i.e. $B(R, \varepsilon\sqrt{n}) \subseteq V$. Which means z and z' are both included in V and must have the same label, which is a contradiction. So, $R \in [P'Q]$. It follows that $d(P', R) \leqslant d(P', Q)$.
Since $R \in \partial W$, $d(P, \partial W) \leqslant d(P, R) \leqslant d(P, P') + d(P', R) \leqslant \gamma + \varepsilon n$.
Since γ can be arbitrarily small, $d(P, \partial W) \leqslant \varepsilon n$.

5. Complexity. Let S_ε be the set of points that are at distance at most $\varepsilon\sqrt{n+3}$ of ∂V, and $\mu(S_\varepsilon)$ the Lebesgue measure of its volume.

PROPOSITION 5.1. *The complexity of the algorithm is*

$$\Theta\left(\frac{\mu(S_\varepsilon)}{\varepsilon^n}\ n \log \frac{c}{\varepsilon}\right) \tag{5.1}$$

Proof. Let us consider a point of a critical zone. Section 4.1 proved that it is included in S_ε.
Because of the measure of their volume, the number of MDNs is at most $m = O(\mu(S_\varepsilon)/\varepsilon^n)$.

When a node A is divided by the algorithm, it means this node is a member of a critical pair (A, B): the border currently seen by the algorithm touches both A and B. After dividing those zones, the border seen by the algorithm touches at least one of A and B. Recursively, the border seen by the algorithm at the end of the execution touches at least one of A and B, which then contains a member of a critical pair (there may be several such critical pairs, we arbitrarily choose one). We associate A and B with this critical pair.
Let h be the height of the tree at the end of execution. Since the measure of the volume of an MDN is $\Theta(\varepsilon^n)$, and the measure of the volume of a zone at depth p is $c^n/2^p$, the height is $h = \Theta(n \log \frac{c}{\varepsilon})$. Each critical pair has been associated with at most $2h$ nodes: all its ancestors, plus one node per ancestor. Thus, there are at most $2mh$ divided nodes (or *inner nodes*).
Now, like in any binary tree, the number of leaves is one more than the number of inner nodes. The number of nodes, which is the number of calls to the oracle, is thus $O(mh)$, or $\Theta\left(\mu(S_\varepsilon)/\varepsilon^n\ n \log \frac{c}{\varepsilon}\right)$.

Furthermore, this bound is optimal. It is indeed reached when V is contained in one MDN: $\mu(S_\varepsilon) = O(\varepsilon^n)$, there is only one zone labeled as viable and the tree is degenerated, containing only $O(1)$ MDN. Formula (5.1) reduces to $\Theta(n \log(c/\varepsilon)) = \Theta(h)$, and precisely the tree contains $\Theta(h)$ nodes. ∎

5.1. Discussion. To get an intuition of what this complexity represents, let us consider a few cases of regular ∂V. Let $\mathcal{H}$ be the $n-1$ dimensional Hausdorff measure (e.g. the area if $n=3$), as in [13]. (The n dimensional Hausdorff measure coincides with the Lebesgue measure in n dimensional space). If ∂V is for instance the border of the search space, or the maximal sphere included in it, then $\mu(S_\varepsilon) = O(\varepsilon \mathcal{H}(\partial V)) = O(\varepsilon c^{n-1})$. The final complexity is then

$$O\left(\left(\frac{c}{\varepsilon}\right)^{n-1} n \log \frac{c}{\varepsilon}\right)$$

This is close to gaining one dimension as compared to the naive algorithm doing one model call for each point of the regular grid: the complexity of the naive algorithm is $O\left((\frac{c}{\varepsilon})^n\right)$.

But in the worst case, S_ε can fill the entire search space, for instance if ∂V is a thin cylinder bent along a Hilbert curve of some finite order. In this case, $S_\varepsilon = \Theta(c^n)$ and the overhead of the algorithm, which is $n \log \frac{c}{\varepsilon}$, makes it worse than the naive approach. By making the cylinder arbitrarily thin, we can even have $\mathcal{H}(\partial V)$ arbitrarily small.

5.2. Simulation. As an illustration of this complexity, Table 5.1 shows the results of a few runs of the algorithm.

				complexity			volume		
n	c	t	$\varepsilon = 2^{-c}$	theoretical	observed	ratio	theoretical	observed	ratio
2	3	6	0.125	32	57	0.6	0.50265482	0.5	1.005
2	6	12	0.01563	519	715	0.7	"	0.50561523	0.994
2	10	20	0.00098	13 858	12 087	1.1	"	0.50264835	1.00001
2	15	30	0.00003	665 220	388 671	1.7	"	0.50265487	0.99999991
3	3	9	0.125	341	354	1.0	0.2680825	0.265625	1.009
3	6	18	0.01563	43 722	28 056	1.6	"	0.2678566	1.001
3	10	30	0.00098	18 655 049	7 141 724	2.6	"	0.2680832	0.999998
4	3	12	0.125	2 472	1 900	1.3	0.1263309	0.1228027	1.029
4	4	16	0.0625	26 375	15 715	1.7	"	0.1258392	1.004
4	5	20	0.03125	263 759	125 472	2.1	"	0.1264514	0.9990
4	6	24	0.01563	2 532 093	995 029	2.5	"	0.1263195	1.00009
5	3	15	0.125	14 098	8 387	1.7	0.053901	0.052734	1.02
5	4	20	0.0625	300 769	126 597	2.4	"	0.053787	1.002
5	5	25	0.03125	6 015 380	1 969 206	3.0	"	0.053904	0.99994
6	3	18	0.125	67 649	33 717	2.0	0.02117	0.02123	0.997
6	4	24	0.0625	2 886 368	948 750	3.0	"	0.02119	0.9988
7	3	21	0.125	283 966	122 570	2.3	0.007741	0.007734	1.001
7	4	28	0.0625	24 231 817	6 387 377	3.8	"	0.007738	1.0003
8	3	24	0.125	1 069 314	420 511	2.5	0.0026599	0.0026444	1.006
9	3	27	0.125	3 676 073	1 336 702	2.8	0.000864	0.000859	1.006
10	3	30	0.125	11 686 396	3 988 135	2.9	0.000267	0.000264	1.01

TABLE 5.1

Simulation of the algorithm. c is the number of cuts along each of the n dimensions: the search space is sampled up to 2^c times along each dimension. $t = nc$ is the kd-tree height. According to Equation (5.1), the theoretical complexity is proportional to $\frac{2\pi^{\frac{n}{2}}}{\Gamma(n/2)}\sqrt{n+3}\ 0.4^n \frac{1}{\varepsilon^{n-1}} n \log \frac{1}{\varepsilon}$ *while the theoretical volume is* $\frac{2\pi^{\frac{n}{2}}}{\Gamma(\frac{n}{2})}\ \frac{0.4^n}{n}$.

The toy model (the set of viable points) used is a hyper-sphere of radius .4, slightly offset from the center of the search space.

The observed complexity is the number of calls to the oracle; the theoretical formula only give a number proportional to the number of calls in the worst case. Thus, the fact that the ratio is close to one is a coincidence. The fact that it is slightly increasing with depth and dimension (i.e. simulation time gets better than predicted) suggests that the average case is better than the worst case.

About the volume, we do expect the ratio to converge to one as $\varepsilon \to 0$.

6. Other applications of the data-structure. Once the kd-tree is built, one can use it to compute other properties of V.

6.1. Measure of the volume. Computing the Lebesgue volume $\mu(W)$ of W is easy: it is simply $\sum_{z \text{ viable}} \mu(z)$.

Now, let us show that this is a good approximation of $\mu(V)$. Similarly to the definition of S_ε in Section 5, let T be the set of points at distance at most εn of ∂W.

PROPOSITION 6.1.

$$\mu(W \setminus T) \leqslant \mu(V) \leqslant \mu(W \cup T)$$

Proof. Thanks to Proposition 4.1 Item 2, $W \cup T$ contains ∂V. This means that $W \cup T$ contains either V or $V^{\complement}$, but thanks to algorithm initialization, the latter is not possible. Thus, $\mu(V) \leqslant \mu(W \cup T)$.

Likewise, $W^{\complement} \cup T$ contains ∂V, V contains $W \setminus T$ and $\mu(W \setminus T) \leqslant \mu(V)$. ∎

6.2. Distance to ∂V. Given a point $P \in V$, the *kd*-tree can be used to efficiently compute $d(P, \partial W)$. This is an approximation of $d(P, \partial V)$:

PROPOSITION 6.2. *Let $P \in V$,*

$$d(P, \partial W) - \varepsilon n \leqslant d(P, \partial V) \leqslant d(P, \partial W) + \varepsilon\sqrt{n}$$

Proof. Direct application of Theorem 4.1. ∎

To compute $d(P, \partial W)$, let us assume without loss of generality that $P \in W$, so we are looking for a point $Q \notin W$ minimizing $d(P, Q)$. This can be done with a classic branch and bound algorithm: branching consists in descending to the children of a node; bounding on zone z consists in computing the minimum and maximum of $d(P, z)$.

7. Conclusion and perspectives. We have detailed an algorithm to store the boundary of an n-dimensional hyper-volume, and to work with it: test if a point belongs to it, compute its volume, compute the distance of a point to the boundary.

This algorithm works by iteratively refining zones containing the boundary, as if to peel a potato one would cut it in half, keep the pieces containing some skin, and iterate until the kept pieces are small enough.

We have proved the complexity, and the convergence of the approximated boundary to the actual boundary.

Future work may include:

- finding an algorithm to compute the distance to ∂V for all nodes at once,
- finding a point $P \in V$ that maximizes $d(P, \partial V)$,
- benchmarking on real viability problems,
- computing not only an approximation of $\mu(V)$, but also the bounds of Proposition 6.1,
- computing the *viability kernel*, i.e. the set of points that are viable for any number of iterations of f.

Figure 1.2 illustrate the execution of the algorithm on a simple oracle. Additional material (illustrations, source code) can be found at http://www.rouquier.org/jb/research/papers/2011_kdtrees/

Acknowledgments. The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2009-2013) under grant agreement DREAM n. 222654-2.

We would like to thank Taras Kowaliw for interesting discussions.

REFERENCES

[1] Jean-Pierre Aubin, Alexandre Bayen, and Patrick Saint-Pierre, *Viability Theory: New Directions*, Springer, 2011.

[2] Jon Louis Bentley, *Multidimensional binary search trees used for associative searching*, Communications of the ACM, 18 (1975), pp. 509–517.

[3] ———, *Multidimensional binary search trees in database applications*, Software Engineering, IEEE Transactions on, SE-5 (1979), pp. 333–340.

[4] P.-A. Coquelin, S. Martin, and R. Munos, *A dynamic programming approach to viability problems*, in Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on, 2007, pp. 178 –184.

[5] Guillaume Deffuant, Laetitia Chapel, and Sophie Martin, *Approximating Viability Kernels With Support Vector Machines*, Automatic Control, IEEE Transactions on, 52 (2007), pp. 933 – 937.

[6] Anil K. Jain, *Data clustering: 50 years beyond k-means*, Pattern Recognition Letters, 31 (2010), pp. 651 – 666.

[7] M. Lindenbaum, S. Markovitch, and D. Rusakov, *Selective sampling for nearest neighbor classifiers*, Machine Learning, 54 (2004), pp. 125–152.

[8] Andrew W. Moore, *An intoductory tutorial on kd-trees*, 1991.

[9] Paul Rosenthal and Lars Linsen, *Direct isosurface extraction from scattered volume data*, in EUROVIS - Eurographics /IEEE VGTC Symposium on Visualization, Thomas Ertl, Ken Joy, and Beatriz Santos, eds., Eurographics Association, 2006, pp. 99–106.

[10] P. Saint-Pierre, *Approximation of the viability kernel*, Applied Mathematics & Optimisation, 29 (1994), pp. 187–209.

[11] Burr Settles, *Active learning literature survey*, Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[12] M. Sicard, N. Perrot, R. Reuillon, S. Mesmoudi, I. Alvarez, and S. Martin, *A viability approach to control food processes: Application to a camembert cheese ripening process*, Food Control, 23 (2012), pp. 312 – 319.

[13] C. Villani, *Mesures de hausdorff*, 2004.

[14] Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley, *State of the art in ray tracing animated scenes*, in Computer Graphics Forum, vol. 28, Blackwell Publishing Ltd, 2009, pp. 1691–1722.