# On the Efficiency of Nearest Neighbor Searching with Data Clustered in Lower Dimensions

Songrit Maneewongvatana
Department of Computer Science
University of Maryland
College Park, Maryland 20742
songrit@cs.umd.edu

David M. Mount
Institute For Advanced Computer Studies &
Department of Computer Science
University of Maryland
College Park, Maryland 20742
mount@cs.umd.edu

UMIACS-TR-2001-05
CS-TR-4209

**Abstract**

In nearest neighbor searching we are given a set of $n$ data points in real $d$-dimensional space, $\mathbf{R}^d$, and the problem is to preprocess these points into a data structure, so that given a query point, the nearest data point to the query point can be reported efficiently. Because data sets can be quite large, we are interested in data structures that use optimal $O(dn)$ storage. Given the limitation of linear storage, the best known data structures suffer from expected-case query times that grow exponentially in $d$. However, it is widely regarded in practice that data sets in high dimensional spaces tend to consist of clusters residing in much lower dimensional subspaces. This raises the question of whether data structures for nearest neighbor searching adapt to the presence of lower dimensional clustering, and further how performance varies when the clusters are aligned with the coordinate axes.

We analyze the popular kd-tree data structure in the form of two variants based on a modification of the splitting method, which produces cells satisfy the basic packing properties needed for efficiency without producing empty cells. We show that when data points are uniformly distributed on a $k$-dimensional hyperplane for $k \leq d$, then expected number of leaves visited in such a kd-tree grows exponentially in $k$, but not in $d$. We show that the growth rate is even smaller still if the hyperplane is aligned with the coordinate axes. We present empirical studies to support our theoretical results.

**Keywords:** Nearest neighbor searching, kd-trees, splitting methods, expected-case analysis, clustering.

## 1 Introduction

Nearest neighbor searching is an important and fundamental problem in the field of geometric data structures. Given a set $S$ of $n$ *data points* in real $d$-dimensional space, $\mathbf{R}^d$, we wish to preprocess these points so that, given any *query point* $q \in \mathbf{R}^d$, the data point nearest to $q$ can be reported quickly. We assume that distances are measured using any Minkowski distance metric, including the Euclidean, Manhattan, and max metrics (see, e.g. [AMN$^+$98] for definitions). Nearest neighbor searching has numerous applications in diverse areas of science.

In spite a recent theoretical progress on this problem, the most popular linear-space data structures for nearest neighbor searching are those based on hierarchical decompositions of space. Although these algorithms do not achieve the best asymptotic performance, they are easy to implement, and can achieve fairly good performance (especially for approximate nearest neighbor searching) in moderately high dimensions. Friedman, Bentley, and Finkel [FBF77] showed that kd-trees achieve $O(\log n)$ expected-case search time and $O(n)$ space, for fixed $d$, assuming data distributions of bounded density. As mentioned earlier, Arya, et al. [AMN$^+$98] shows that a somewhat more sophisticated tree can perform approximate nearest neighbor queries with guaranteed worst-case performance. There are a vast number of variations on this theme.

The unpleasant exponential factors of $d$ in the worst-case analyses of these simple data structures would lead one to believe that they would be unacceptably slow, even for moderate dimensional nearest neighbor searching (say in dimensions up to 20). Nonetheless, practical experience shows that, if carefully implemented, they can applied successfully to problems in these and higher dimensions [AMN$^+$98].

The purpose of this paper is not to propose new data structures, but to attempt to provide some theoretical explanation for a possible source for this unexpectedly good performance, and to comment on the limitations of this performance. Conventional wisdom holds that because of dependencies between the dimensions, high dimensional data sets often consist of many clusters, each of which resides in or near a much lower dimensional subspace. A great deal of work in multivariate data analysis deals with the problems of dimension reduction and determining the *intrinsic dimensionality* of a data set. For example, this may be done through the use of techniques such as the Karhunen-Loeve transform (also known as principal component analysis) [Fuk90].

This suggests the question of how well do nearest neighbor data structures take advantage of the presence of low-dimensional clustering in the data set to improve the speed of search? Traditional worst-case analysis does not model the behavior of data structures in the presence of simplifying structure in the data. In fact, quite to the contrary, it focuses on worst-case situations, which may be rare in practice. Even expected-case analyses based on the assumption of (full dimensional) uniformly distributed data [FBF77, Cle79] are not dealing with "easy" instances since the curse of dimensionality is felt in its full force.

We consider the following very simple scenario. Assuming that the data points and query points are sampled uniformly from a $k$-dimensional hyperplane (or $k$-flat), where $k < d$, what is the expected-case search time for kd-trees as a function of $n$, $k$ and $d$? In [FBF77] it is shown that when $k = d$ (the full dimensional case) and if boundary effects are ignored (see Section 4 for definitions), the expected number of leaf cells in the tree to be visited is at most $(G(d)^{1/d} + 1)^d$, where $G(d)$ is the ratio of the volumes of a $d$-dimensional hypercube and a maximal enclosed ball for the metric inside the hypercube. (Note that this does not involve the number of points $n$, which enters as a logarithmic factor in finding these leaves.) These results rely on the fact that when data points are uniformly distributed, the cells of the kd-tree can be approximated by $d$-dimensional hypercubes. However this is not the case when data points lie on a lower dimensional hyperplane.

It is natural to conjecture that if $k \ll d$, then search times grow exponentially in $k$ but not in $d$. Indeed, we show that this is the case, for a suitable variant of the kd-tree. We introduce a new splitting method, called the *canonical sliding-midpoint splitting method.* This is a variant of a simpler splitting method called *sliding-midpoint*, which is implemented in the ANN approximate nearest neighbor library [MA97]. (Definitions are given in the next section.)

Our main result is (assuming this splitting rule) kd-trees can indeed achieve query times depending exponentially on the intrinsic dimension of data, and not on the dimension of the space. We show that if the data points are uniformly distributed on a $k$-flat, then the expected number of leaf cells that intersect a nearest neighbor ball is $O(d^{k+2})$. Further, we show that if the points are clustered along a $k$-flat that is aligned with the coordinate axes, even better performance is possible. The expected number of leaf cells intersecting the nearest neighbor ball decreases to $O((d - k + 1)c^k)$, where $c$ is the quantity $(G(k)^{1/k} + 1)$.

The restrictions of using the sliding-midpoint splitting method and having points lie on a flat do not seem to be easy to eliminate. It is not hard to show that if points are perturbed away from the flat, or if some other splitting method is used, there exist point configurations for which $2^d$ cells will be visited.

We present empirical results that support our results. Furthermore, we consider its robustness to violations in our assumptions. We consider the cases where there is more than just a single cluster of points, but a number of clusters of points lying on different hyperplanes, and where the points do not lie exactly on the hyperplane, but are subject to small perturbations. These empirical results bear out the fact that the query times are much more strongly dependent on $k$ than on $d$.

The rest of the paper is organized as follows. Section 3 introduces kd-trees and the two splitting rules, sliding-midpoint and canonical sliding-midpoint. Section 4 presents the analysis for kd-trees when the points lie on an arbitrarily oriented $k$-flat. Section 5 presents the analysis for points lying on an axis-aligned $k$-flat.

Finally, Section 6 presents our empirical results.

## 2 Prior Work

There is an extensive literature on methods for nearest neighbor searching in high dimensional spaces. Our primary interest is in data structures that can be stored in main memory (as opposed to database applications). Since such data sets can be quite large in practice (tens of thousands to tens of millions) we limit ourselves to consideration of data structures whose total space grows linearly with $d$ and $n$. Under these stringent assumptions, it is difficult to achieve very efficient search times in higher dimensions. Arya, et al. [AMN+98] showed that $(1+\epsilon)$ approximate nearest neighbor queries can be answered from such a data structure in $O((d/\epsilon)^d \log n)$ time, assuming $O(dn)$ storage. There have been a number of approaches to reduce the exponential dependence on $d$. The closest to achieving the linear storage bound are the data structures by Indyk and Motwani [IM98], which achieves $O(d \log^{O(1)} n)$ search time using $O(1/\epsilon)^d n \log^{O(1)} n$ storage. For the large data sets we are interested in, even polylogarithmic factors and polynomial factors in $(1/\epsilon)$ may exceed reasonable storage limits. If space is not an issue, then there are a number of alternatives. For example, Clarkson [Cla88] presents a data structure that has $O(d^{O(d)})$ search time and $O(n^{(1+\delta)\lceil d/2 \rceil})$ space.

There is a wealth of literature on methods for "dimension reduction" for high dimensional data sets. A good survey is presented by Carreira-Perpiñán [CP96]. However, our interest is on how high dimensional data structures adapt to low-dimensional structure.

The problem of how hierarchical decomposition methods perform when given data with low intrinsic dimensionality has been studied before. Faloutsos and Kamel [FK94] have shown that under certain assumptions, the query time of range queries in an R-tree depends on the fractal dimension of the data set. Their results do not apply to nearest neighbor queries, because their analysis holds in the limit for a fixed query range as the data size tends to infinity. However, with nearest neighbor queries, the analogy of the query range is the nearest neighbor ball, and its size varies with point density. Otherwise, we know of no theoretical results on our problem.

## 3 Background

First we recall the basic facts about kd-trees [Ben75]. Consider a set $S$ of $n$ data points in $\mathbf{R}^d$. A kd-tree is a binary tree that represents a hierarchical subdivision of space, using splitting planes that are orthogonal to the coordinate axes. Each node of the kd-tree is associated with a closed rectangular region of space, called a *cell*. Each is the product of $d$ closed intervals, along each of the coordinate axes. The root's cell is associated with a bounding hypercube that contains all the points of $S$. Each cell is associated with two pieces of information, a splitting dimension $i$ (from 1 to $d$) and splitting value $x$. These define an axis-orthogonal *splitting hyperplane*. The points of the cell are partitioned to one side or the other of this hyperplane (and points lying on the hyperplane can be placed on either side). The resulting subcells are the children of the original cell. This process continues until the number of points is at most one (or more generally a small constant value). There are a number of ways of selecting the splitting hyperplane, which we outline below.

**Standard split:** Friedman, Bentley and Finkel [FBF77] selected the splitting dimension to be the one for which the data points have the maximum *spread* (difference between the maximum and minimum values). The splitting value is chosen to be the median in that dimension. This is the most well-known and widely used splitting method.

**Midpoint split:** The splitting hyperplane passes through the center of the cell and bisects the longest side of the cell. If there are many sides of equal length, any may be chosen first, say, the one with the lowest coordinate index. This is just a binary version of the well-known quadtree and octree decompositions.

Observe that in the standard splitting rule, roughly half of the data points are associated with each child. This implies that the tree has $O(\log n)$ depth and $O(n)$ nodes. The midpoint tree has the feature that for all cells, the ratio of the longest to shortest side (the *aspect ratio*) is at most 2. (We will sometimes use the term *box* to mean a cell of bounded aspect ratio.) This is not necessarily true for the standard splitting method. As shown in [AMN+98], bounded aspect ratio is important to the efficiency of approximate nearest neighbor searching. Unfortunately, if the data are clustered, it is possible to have many empty cells that contain no data points. This is not uncommon in practice, and may result in trees that have many more than $O(n)$ nodes.

Note that the set of possible splitting planes in midpoint split is not determined by the data points, only by the position of the initial bounding hypercube. For example, suppose that the initial bounding box is affinely mapped to a unit hypercube $[0, 1]^d$. The splitting values are all of the form $k/2^i$, for some odd integer $k$, $1 \leq k < 2^i$. We call any cell which could result from the application of this method a *midpoint box*. The concept of such a *canonical set* of splitting planes and cells will be considered later.

Unfortunately, there does not seem to be a single simple splitting rule that provides us with all the properties one might wish for (linear size, logarithmic depth, bounded aspect ratio, convexity, constant cell complexity). In [AMN+98] the BBD-tree was introduced. This tree uses a combination of two operations, splitting and shrinking to provide for all of these properties (except for convexity). The BAR-tree [DGK99] provides all of these properties, by using nonorthogonal splitting planes, but the cells may have as many as $2^d$ bounding faces.

We discuss two other variants of kd-trees, both designed to provide the same simplicity that makes kd-trees so popular, while overcoming some of the shortcomings in the above splitting methods. To understand the problem, suppose that the data points are highly clustered along a few dimensions but vary greatly along some the others (see Fig. 1). The standard kd-tree splitting method will repeatedly split along the dimension in which the data points have the greatest spread, leading to many cells with high aspect ratio. In nearest neighbor processing it is necessary to visit all the leaf cells that overlap the nearest neighbor ball, since any one of them might contain the nearest neighbor. A nearest neighbor query near the center of the bounding square would visit a large number of these cells. On the other hand, midpoint split visits limited number of cells because of the bounded aspect ratios, but produces a large number of empty cells.

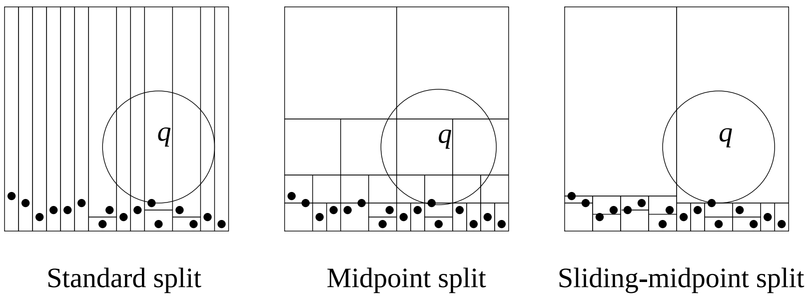



Figure 1: Splitting methods with clustered point sets.

**Sliding-midpoint:** It first attempts to perform a midpoint split, by considering a hyperplane passing through the center of the cell and bisecting the cell's longest side. If the data points lie on both sides of the splitting plane then the splitting plane remains here. However, if a *trivial split* were to result (in which all the data points lie to one side of the splitting plane), then it "slides" the splitting plane towards the data points until it encounters the first such point. One child is a leaf cell containing this single point, and the algorithm recurses on the remaining points.

This splitting method was first introduced in the ANN library for approximate nearest neighbor searching [MA97] and was subsequently analyzed empirically in [MM99a]. This method produces no empty nodes, and hence the tree has $O(n)$ nodes. Although cells may not have bounded aspect ratio, observe that every skinny cell that is produced by sliding is adjacent to a fat leaf cell. In [MM99b] we show that this is sufficient to satisfy the necessary packing constraint that fat subdivisions possess. This tree can be constructed in $O(dn \log n)$ time, as is true for the standard kd-tree [MM99a].

Because there is no guarantee that the point partition is balanced, the depth of the resulting tree may exceed $O(\log n)$. This theoretical deficiency could be remedied by introducing more complex splitting methods or auxiliary data structures. However this additional complexity does not seem to be warranted in practice. In our experience with real data sets in higher dimensions, we have observed that the depth of the search tree (which is almost always $O(\log n)$) seems to be less of a dominating factor in running time than the number leaves visited in the search (which is almost always grows exponentially with dimension).

We introduce a small modification of sliding-midpoint. It has been introduced primarily for technical reasons. The proof of the main theorem of Section 4 relies on the presence of having a canonical set of splitting planes, while retaining the property that no empty cells are produced. Although this method is somewhat artificial, our empirical studies indicate that its performance is very similar to sliding-midpoint. We conjecture that similar results hold for the sliding-midpoint method, but we have no proof of this. This method is based on using the same midpoint cuts that midpoint split uses, rather than using the true midpoint of the cell.

**Canonical sliding-midpoint:** Define the *enclosure* for a cell to be the smallest midpoint box that encloses the cell. During the construction phase, each node of the tree is associated both with its cell and the cell's enclosure. We first try to split the cell using a hyperplane that bisects the longest side of this enclosure (rather than the cell itself). By the minimality of the enclosure, this cut intersects the cell. If this results in a trivial split, then it slides the splitting plane towards the data points until it encounters the first such point. This point is stored in a leaf cell, and the algorithm recurses on the remaining points.

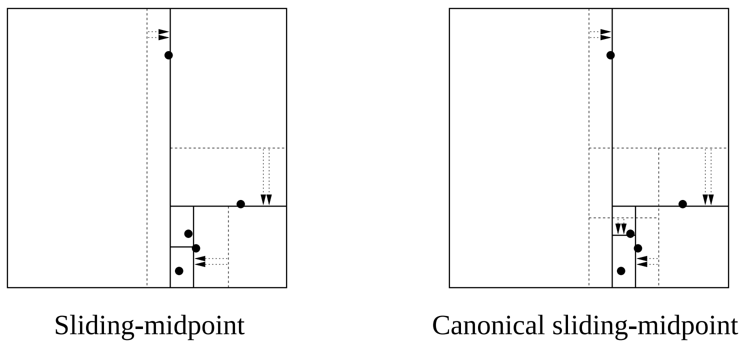



Figure 2: Sliding-midpoint and canonical sliding-midpoint.

The differences between these two splitting methods is illustrated in Fig. 2. Notice that in the sliding-midpoint method the slides originate from a line that bisects the cell (shown in dashed lines), whereas in the canonical sliding-midpoint method, the slides originate from the midpoint cuts of the enclosing midpoint cell (shown in dashed lines).

Because of prior sliding operations, the initial split used in the canonical sliding-midpoint method may not pass through the midpoint of the cell. After splitting, the enclosures for the two child cells must also be computed. This can be done in $O(d)$ time [BET93]. Thus, this tree can be constructed in $O(dn \log n)$ time, and has $O(n)$ nodes, just like the sliding-midpoint split kd-tree.

# 4 Points Clustered on Arbitrarily Oriented Flats

Let $F$ be an arbitrary $k$-dimensional hyperplane (or *$k$-flat*, for short) in $\mathbf{R}^d$. We assume that $F$ is in general position, and in particular that $F$ is not parallel to any of the coordinate axes. Let $S$ denote a set of data points sampled from a closed convex, *sampling region* of $F$ according to some probability distribution function. We assume that the distribution function satisfies the following *bounded density assumption*[BWY80]. There exist constants $0 < c_1 \leq c_2$, such that for any convex open subregion of the sampling region with $k$-dimensional volume $V$, the probability that a given sampled point lies within this region is in the interval $[c_1 V, c_2 V]$. (This is just a generalization of a uniform distribution but allows some variation in the probability density.)

To avoid having to deal with boundary effects, we will assume that there are sufficiently many data points sampled, and that the query points are chosen from a sufficiently central region, such that with high probability the nearest neighbor ball for any query point lies entirely within the sampling region. More formally, fix any compact convex region on $F$, called the *query region*, from which query points will be sampled. Let $w$ denote the diameter of this region. Now, take the data points to be sampled from a hypercube of side length $w' > w$ centered around this region, such that the local density of the distribution is independent of $w'$. Our results hold in the limit as $w'$ tends to infinity. The work of Arya, et al. [AMN96] shows that consideration of boundary effects for kd-trees with uniformly distributed points only tends to decrease the number of cells of the tree visited. Nonetheless these effects can be quite messy to deal with.

Let $B(r)$ denote a ball of radius $r$. Let $V_F(q, r)$ denote the $k$-dimensional volume of intersection of $F$ and ball $B(r)$ centered at point $q$. If we restrict $q$ to lying on $F$, then $V_F(q, r)$ is a constant for all $q$, which we denote as $V_F(r)$. Following the approach taken in [AMN96], let us first scale space so that the lower density bound becomes $c_1 = 1/V_k(1)$. After this scaling, a ball of unit radius is expected to contain at least one point of the sample. As observed in [AMN96], as $k$ increases, a ball of unit radius is a very good approximation to the expected nearest neighbor ball. The reason is that $V_F(r)$ is growing as $r^k$, and so for large $k$, the probability that a data point lies in $B((1-\delta)r)$ drops rapidly with $\delta$, and the probability that there is at least one point in $B((1+\delta)r)$ increases rapidly with $\delta$.

Consider a kd-tree built for such a distribution, assuming the canonical sliding-midpoint splitting method. Our analysis will focus on the number of leaf cells of the kd-tree that are visited in the search. The running time of nearest neighbor search (assuming priority search [AMN$^+$98]) is more aptly bounded by the product of the depth of the tree and the time to access these nodes, which can be assumed to be $O(\log n)$ (either because the tree is balanced, or auxiliary data structures are used). We focus just on the number of leaf cells primarily because in higher dimensions this seems to be the more important factor influencing the running time.

The main result of this section is that the expected number of cells of a canonical sliding-midpoint kd-tree that intersect a unit ball centered on $F$ is exponential in $k$, but not in $d$. To see that the proof is nontrivial, suppose that instead of a kd-tree we had stored the points in a regular grid instead. If the nearest neighbor ball contained even a single vertex of the grid, then it would overlap at least $2^d$ cells. A remarkable feature of the canonical midpoint-split tree is that it is not possible to generate a vertex that is incident to such a large number of cells when the points lie on a lower dimensional flat. This feature of kd-trees seems to be an important reason that these trees adapt well to the intrinsic dimensionality of the point set. Although it is not clear how to establish this property for other types of splitting methods in the worst case, we believe that something analogous to this holds in the expected case (over all possible placements of the query point).

Before we can prove the main result, we define some definitions and state some lemmas that will be used in the theorem.

Define a *splitting hyperplane* to be any axis orthogonal plane. Define a *cut* to be the intersection of a splitting hyperplane and a cell of the kd-tree (that is, a $d-1$ dimensional rectangle). Let $B$ denote the unit ball, and let $Bcut$ denote the set of cuts that intersect $B$. Since each successive element of $Bcut$ subdivides a cell that intersects $B$ into two subcells that intersect $B$, it follows that the total number of leaf cells that intersect $B$ is just $Bcut + 1$. Thus it suffices to bound the size of $Bcut$.

The choice of the initial splitting hyperplanes (prior to sliding) made at each node of the canonical midpoint-split kd-tree depends only on the indexing of the coordinate axes and the shape of the enclosure. The data points themselves only affect whether a split is made and whether sliding takes place. For each dimension, these splitting values naturally define an (infinite) tree structure. Order these values according to a breadth-first traversal of this tree. (For example, for a kd-tree built on the unit interval $[0, 1]$, this order would be $\langle 1/2,\ 1/4,\ 3/4,\ 1/8,\ 3/8,\ 5/8, \ldots \rangle$.) We call this the *canonical tree ordering* of splitting hyperplanes.

Define the *initial splitting hyperplane* to be the splitting hyperplane chosen before any sliding. This is always a midpoint-split hyperplane. Consider the initial splitting hyperplanes in $T$. For each axis $i$, $1 \le i \le d$, define the *first split* for $i$th axis, denoted $Fsplit(i)$, to be the earliest (in the canonical tree order) hyperplane orthogonal this axis to intersect $B$. Note that if $B$ intersects two parallel hyperplanes at some level in this tree ordering, then it must intersect a parallel hyperplane between them at the next higher (closer to the root) level. Thus $Fsplit(i)$ is unique highest level splitting hyperplane intersecting $B$. Define $Fsplit$ to be $d$-element set consisting of $Fsplit(i)$ for $1 \leq i \leq d$. Note that $Bcut$ is a set of cuts (the intersection of a hyperplane and a cell), whereas $Fsplit$ is a set of hyperplanes.

Here is a high-level overview of the proof of the main result. We first bound the number of nonempty midpoint kd-tree cells that are bounded on some side by one of these hyperplanes. This bound is based on the fact that there are only $d$ such $Fsplit$ hyperplanes. We use combinatorial arguments, similar to bounds on the number of faces in an arrangement of $d$ hyperplanes in dimension $k$. We argue that each of the $Bcut$'s arise in one of three ways.

(1) It is supported an $Fsplit$ hyperplane, in which case the previous bound is applied. This is a worst-case bound (holding irrespective of the data distribution), and turns out to provide the dominant term in the overall bound.

(2) It arose by applying a slide to a midpoint cut that does not intersect the ball. We argue that each such slide results in a leaf cell on one side, which will not be split further, thus preventing this from happening again with a parallel split on the same side of the ball. We bound the number of cuts of this type by arguing that there can be at most $2d$ such cuts for each cut of types (1) and (3).

(3) Its initial cut is not a first split and intersects the ball. To bound this last type of cut, we observe that they arise from cells whose width is bounded (because this not the first cut in this direction to intersect the ball). We apply an argument based on the observation that because points lie on a $k$-flat, their density grows exponentially with $k$ and not with $d$.

We classify the cuts of $Bcut$ into three types. Consider a cut $c$ orthogonal to dimension $i$. If the initial splitting hyperplane (prior to any sliding) is $Fsplit(i)$, then we call this a *first cut*. If its initial splitting hyperplane does not intersect $B$, (and hence the cut intersects $B$ because of sliding) we call it a *sliding cut*. Otherwise, (its initial splitting hyperplane intersects $B$ but is not $Fsplit(i)$) we call it a *close cut*. For example, Fig. 3 shows the various types of cuts. The initial midpoint splitting cuts are shown as dashed lines, and the final cuts are shown as solid lines. Double arrows indicate where initial cuts were slid. The two $Fsplit$ hyperplanes are shown as dotted lines. Note that the horizontal first cut was slid away from its initial splitting hyperplane. The sliding cuts all arose from initial splitting hyperplanes that do not intersect $B$. The remaining cut is a close cut. There is one vertical cut that does not intersect $B$, and so is not a $Bcut$.

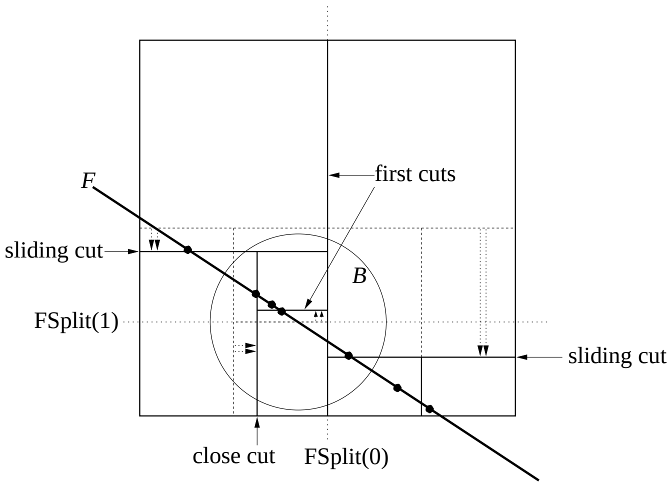



Figure 3: Types of cuts.

**Lemma 4.1.** *If $B$ is a Euclidean ball, the expected number of close cuts in Bcut is $O((2d)^{k/2})$.*

**Proof** Consider some close cut $c \in Bcut$, and let $i$ be its splitting axis. Let $h$ be the initial splitting hyperplane (before any sliding). Since $c$ is a close cut, $h \neq Fsplit(i)$. Let $c'$ be the initial midpoint cut from which $c$ resulted. Since both $h$ and $Fsplit(i)$ intersect $B$, and since $Fsplit(i)$ would be tried before $h$, it follows that the cell $C$ that is bisected by $c'$ lies entirely to one side of $Fsplit(i)$. Since initial cuts are midpoint cuts, it follows that the width of $C$ along dimension $i$ is at most 2, the diameter of $B$. Since $c'$ is a midpoint cut, the longest side $C$ is of length at most 4. There must be at least two points lying within $C$ that are separated by $c$. The Euclidean diameter of $C$ is at most $4\sqrt{d}$.

Thus, each close cut separates at least two points that are within a distance of $4\sqrt{d}$ of the center of $B$. If $m$ denotes the expected number of such points, the number of close cuts is at most $m-1$. A ball of radius $4\sqrt{d}$ intersects the $k$-flat as a $k$-dimensional ball whose $k$-volume is proportional to $(4\sqrt{d})^k$. This implies that the expected number of points lying within this ball is $O(4\sqrt{d})^k$. Therefore the total number of cuts is $O((2d)^{k/2})$. □

It is easy to generalize the above to any Minkowski ball. The corresponding bound for the $L_m$ Minkowski $m < \infty$ is $O((4d^{1/m})^k)$. For the $L_\infty$ ball (a hypercube) this reduces to $O(4^k)$.

Next, we consider first cuts. We say that a cell is *void* if does not intersect $F$, and *nonvoid* otherwise. Note that void cells cannot possible contain points of $S$, whereas nonvoid cells might, but need not. Void cells may be created as a result of the initial midpoint subdivision, but if so, sliding will be invoked so that the would-be void child has exactly one data point. We will make use of the well known bound $f_k(m)$ on the number of $k$-dimensional cells in an arrangement of $m$ hyperplanes in $k$-space [Ede87].

$$f_k(m) = \sum_{j=0}^{k} \binom{m}{j} \ \in \ \Theta(m^k).$$

**Lemma 4.2.** *The number of first cuts in Bcut is at most $f_{k+1}(d) - 1$, which is $O(d^{k+1})$.*

**Proof** It suffices to consider the tree resulting from the initial splitting hyperplanes, since these initial cuts all intersect $B$, and sliding can only decrease the number of cuts in *Bcut*.

Consider a minimal subtree of the kd-tree that includes all the first cuts. More formally, starting at the root, for each internal node whose splitting hyperplane does not intersect $B$, one child is entirely disjoint from $B$ and the other contains $B$. Discard the entire subtree of the first child, and replace this node with second child. After encountering $Fsplit(i)$ it is possible to see a descendent with a parallel cut in *Bcut*, but before encountering $Fsplit(j)$ for $j \neq i$. We may ignore these parallel cuts (taking either child) since their cuts are counted among the close cuts or sliding cuts. Take either child to be in the subtree. The resulting tree only has cuts supported by *Fsplit* hyperplanes. We will show that the number of cuts in the subtree is at most $f_{k+1}(d) - 1$,

Our proof is based on a recurrence, whose solution is the desired number of cuts. Because of the canonical ordering of cuts in the kd-tree, we can think of these cuts as being introduced in a series of rounds. All the cuts supported by the same *Fsplit* hyperplane are introduced in the same round. Let us assume that the axes have been sorted according to the order of introduction of the *Fsplit* hyperplanes (that is, if $Fsplit(i)$ is at a higher level than $Fsplit(j)$ then all the cuts supported by $Fsplit(i)$ will be inserted in one round before those supported by $Fsplit(j)$). There are $d$ rounds.

As mentioned earlier, it is not necessarily the case that an nonvoid cell will be split with the introduction of a new *Fsplit*. However, to produce an upper bound, we may assume that this always happens. Let $n_i$ denote the number of nonvoid cells that are created in round $i$, by splitting all nonvoid cells at round $i-1$. Let $v_i$ denote the number newly created void cells, which result whenever a nonvoid cell has been split but only one of its child cells is nonvoid. Our goal is to determine the total number of void and nonvoid at the end of the process, namely

$$n_d + \sum_{i=0}^{d} v_i.$$

Once a void cell is created, it cannot be split further. However, for the sake of analysis, imagine that each void cell is split with each subsequent round. Let $s_i$ denote the number of void cells under the assumption that they continue to be split. Call these *pseudo cells.*

For the basis, we have one nonvoid cell and no void cells, and hence $n_0 = 1$ and $v_0 = s_0 = 0$. As we proceed from round $i-1$ to round $i$, observe that all nonvoid cells are split, and all pseudo cells are split. Since these sets are disjoint, it follows that $n_i + s_i = 2^i$. At round $i$, the first $i$ *Fsplit* hyperplanes introduced so far subdivide the flat $F$ into an arrangement of $i$ hyperplanes. Each nonvoid cell at round $i$ intersects $F$ as a $k$-dimensional cell in this arrangement. Hence, in the worst case we have $n_i = f_k(i)$, and thus $s_i = 2^i - f_k(i)$.

Pseudo cells arise in two ways. First off, they can arise from newly created void cells at round $i$, or they can be pseudo cells from the previous round that were split. Thus, $s_i = v_i + 2s_{i-1}$. Combining this, we have

$$\begin{aligned} v_i &= s_i - 2s_{i-1} \;=\; 2^i - f_k(i) - 2(2^{i-1} - f_k(i-1)) \\ &= 2f_k(i-1) - f_k(i) \;=\; 2\sum_{j=0}^{k}\binom{i-1}{j} - \sum_{j=0}^{k}\binom{i}{j}. \end{aligned}$$

To simplify this, we break the first summation into two copies and break off the last term. For the second term, we use the identity $\binom{i}{j} = \binom{i-1}{j} + \binom{i-1}{j-1}$.

$$\begin{aligned} v_i &= \left(\sum_{j=0}^{k}\binom{i-1}{j} + \sum_{j=0}^{k-1}\binom{i-1}{j} + \binom{i-1}{k}\right) - \sum_{j=0}^{k}\left(\binom{i-1}{j} + \binom{i-1}{j-1}\right) \\ &= \sum_{j=0}^{k}\binom{i-1}{j} + \sum_{j=0}^{k}\binom{i-1}{j-1} + \binom{i-1}{k} - \sum_{j=0}^{k}\binom{i-1}{j} - \sum_{j=0}^{k}\binom{i-1}{j-1} \\ &= \binom{i-1}{k}. \end{aligned}$$

Using the fact that $\sum_{i=0}^{n}\binom{i}{k} = \binom{n+1}{k+1}$, we conclude that the total number of cells is

$$n_d + \sum_{i=0}^{d} v_i \;=\; f_k(d) + \sum_{i=0}^{d}\binom{i-1}{k} \;=\; f_k(d) + \binom{d}{k+1} \;=\; f_{k+1}(d),$$

which is $O(d^{k+1})$. The total number of cuts is one less that this. □

**Lemma 4.3.** *The number of sliding cuts in Bcut is at most $2d$ times the number of first cut and close cuts combined.*

**Proof** By definition, a sliding cut arises from a initial splitting hyperplane that lies outside of $B$ and was slid until it intersects $B$. Whenever a slide occurs, the cell is partitioned into two children, one contains a single point and cannot be further subdivided. Thus, it is impossible to slide twice from the same direction, without an intervening split. Hence, if we label all the internal nodes of the kd-tree according to whether they are sliding cuts or not, the sliding cut nodes will form linear chains (a node with at least one leaf child) of length at most $2d$. If we were to compress this tree by removing all these chains, the result would be a tree that is smaller by a factor of at most $2d$, but in which the *Bcut*'s consist only of first cuts and close cuts. This establishes the bound. □

**Theorem 4.1.** *Let $S$ be a set of points from $\mathbf{R}^d$ sampled independently from a $k$-flat $F$ by a distribution satisfying the bounded density assumptions and scaled as described above. Let $T$ be a kd-tree built for $S$ using the canonical sliding-midpoint splitting method. Then, the expected number of leaf cells of $T$ that intersect a unit ball centered on $F$ is $O(d^{k+2})$.*

**Proof** Combining Lemmas 4.1, 4.2, and 4.3, it follows that the total number of cuts in *Bcut* is dominated by the bound on the sliding cuts (in conjunction with the first cuts) to get $O(2d(d^{k+1})) = O(d^{k+2})$. This completes the proof of Theorem 4.1. □

Using Theorem 4.1 and the observation made earlier that a ball of unit radius is good approximation to (or larger than) the nearest neighbor ball, we have the following bound on the number of leaf cells visited in nearest neighbor searching.

**Corollary 4.1.** *The expected number of leaf cells of $T$ encountered in nearest neighbor searching is $O(d^{k+2})$.*

# 5 Points Clustered on Axis-Aligned Flats

We consider the case where the set $S$ of data points in $\mathbf{R}^d$ sampled independently from a distribution of bounded density along an axis-aligned $k$-flat. This is a particularly good case for the standard kd-tree splitting method, since the point spread is always widest along axes spanned by the flat, and hence all splits will be orthogonal to the flat. The behavior of the algorithm will be isomorphic to its behavior on points that are uniformly distributed in $\mathbf{R}^k$. However, we will analyze the sliding-midpoint methods, for the sake of comparison with the results of the previous section.

If we split orthogonal to any of the $d-k$ coordinate axes that are orthogonal to the flat, the points will all lie to one side of this splitting hyperplane (barring the degenerate case where the flat lies on the splitting hyperplane). The splitting hyperplane will slide until it lies on the flat. After any sequence of $2(d-k)$ such slides, the flat will be tightly enclosed within a cell. Splits along other axes will be orthogonal to the flat, and so will behave essentially the same a sliding-midpoint decomposition in $k$-space. The main complication is that the algorithm does not know the location of the flat, and hence these two types of splits may occur in an unpredictable order.

Let $G(k)$ denote the dimension dependent ratio of the volumes of a $k$-dimensional hypercube and a maximal enclosed $k$-ball for the metric inside the hypercube. Let $c(k) = (G(k)^{1/k} + 1)$. For example, for the $L_\infty$ (max) metric the metric ball is a hypercube, and $c(k) = 2$. For the $L_2$ (Euclidean) metric $G(k) = k\Gamma(k/2)/(2^{k+1}\pi^{k/2})$.

**Theorem 5.1.** *Let $S$ be a set of points from $\mathbf{R}^d$ sampled independently from an axis-aligned $k$-flat $F$ by a distribution satisfying the bounded density assumptions described in Section 4. Let $T$ be a kd-tree built for $S$ using the canonical sliding-midpoint splitting method. Then, the expected number of leaf cells of $T$ that intersect a unit ball centered on $F$ is $O((d-k+1)c(k)^k)$.*

**Proof** Let assume that space has been scaled so that the bounding box for the point set is a unit cube. Define the $i$th length of a cell to be its length along the $i$th coordinate axis. Let $D_F$ denote the set of $k$ axes that $F$ spans on and let $E_F$ be the remaining set of $d-k$ axes. We will assume the worst-case, that when a split is made along an axis in $D_F$ it results in a nontrivial partition of the points, and hence no sliding takes place. In contrast, when a split is made along an axis in $E_F$, the points will all lie on one side or the other of this split, and hence the split will slide until it contacts the flat. Thus, all splits along axes in $D_F$ behave essentially the same and all splits along $E_F$ behave essentially the same.

Observe that the cuts along the axes in $D_F$ behave with respect to $F$ exactly as they would if we were constructing a kd-tree for a set of points in $\mathbf{R}^k$. So, from the analysis given in [FBF77], it follows that the expected number of leaf cells visited is $O(c(k)^k)$. For each such cell, as many as $2(d-k)$ slides may have taken place surrounding this cell, and each of the resulting leaf cells might be visited. Thus the total number of expected cells visited is $O((2(d-k)+1)c(k)^k) = O((d-k+1)c(k)^k)$. □

# 6 Empirical Results

We conducted experiments on the query performance of the kd-tree for data sets lying on a lower dimensional flat. We used the ANN library [MA97] to implement the kd-tree. The experiments were run on a PC, running

Linux. The program was compiled by the g++ compiler. We measured a number of statistics for the tree, including its size, depth, and the average aspect ratio of its cells.

We used priority search to answer queries. We gathered a number of statistics including CPU time, the number of internal and leaf nodes visited in the tree, and the number of floating-point operations. We present the total number of nodes, and the number of leaf nodes in our grades, because these parameters are machine-independent, and they are closely correlated with CPU time.

## 6.1 Distributions tested

Before discussing what we did in the experiments, we briefly describe the distributions used.

**Uniform:** Each coordinate of each point was chosen uniformly from the interval $[-1, 1]$.

**Gauss:** Standard deviation, $\sigma$, is provided to the distribution. The points are generated using zero mean Gaussian distribution. All coordinates have the same value of $\sigma$.

**Clustered-orthogonal-ellipsoids:** This distribution is designed to model point sets that are clustered on lower dimensional flats, where all the flats are aligned with the coordinate axes. The distribution is given a number of clusters $c$. Then $c$ cluster centers are generated from a uniform distribution over $[-1, 1]$. The distribution is also given two standard deviation values, $\sigma_{thin}, \sigma_{fat}$. The dimensions are divided into fat and thin dimensions. Another parameter, $d_{max}$, indicates the maximum number of the fat dimensions. For each color class, a number, $k$, between 1 and $d_{max}$ is randomly chosen to be the number of fat dimensions (the dimension of the flat). Then $k$ dimensions are chosen at random to be fat dimensions. Each point is generated from a Gaussian distribution centered at a randomly chosen cluster center. For the fat (resp., thin) dimensions, the standard deviation is set to $\sigma_{fat}$ (resp., $\sigma_{thin}$).

**Uniform-on-orthogonal-flat:** The dimension of the flat, $k$, is provided, and $k$ dimensions are chosen at random. Among these dimensions, the points are distributed uniformly over $[-1, 1]$. For, the other $(d-k)$ dimensions, we generate a uniform random coordinate that is common to all the points. For our experiments, we considered both points exactly on the flat, and points perturbed off the flat by a Gaussian error with standard deviation 0.005.

**Uniform-on-rotated-flat:** This distribution is the result of applying $r$ random rotation transformations to the points in uniform-on-orthogonal-flat distribution. The flat is therefore rotated in a random direction. Each rotation is through a uniformly distributed angle in the range $[-\pi/2, \pi/2]$ with respect to two randomly chosen dimensions.

## 6.2 Sliding-midpoint and canonical sliding-midpoint

Our theoretical results for arbitrary flats apply only to the canonical sliding-midpoint method. This was largely for technical reasons. A natural question is how much this method differs from the more natural sliding-midpoint method. We discovered that the two trees are quite similar. If the number of points is fairly small (less than $2^k$) then the trees are identical. Because it bisects the cell, we expect sliding-midpoint to perform somewhat better on average. We ran a set of experiments to compare query times for both sliding-midpoint and canonical sliding-midpoint methods.

Distributions tested included Gauss and Clustered-orthogonal-ellipsoids. We set $\sigma$ to 0.4 for the Gauss distribution. For Clustered-orthogonal-ellipsoids, $\sigma_{fat} and \sigma_{thin}$ were set to 0.4 and 0.005, respectively, and $d_{max}$ was set to $d/2$, where $d$ is the number of dimensions of the space. The number of points, $n$, ranged from 40 to 163,840 and $d$ ranged from 2 to 16.

The queries come from the same distribution as the set of points and the number of queries is $n$. Table 4 shows the average numbers of nodes visited over all queries when the tree was built by two splitting methods. Observe that the number of nodes visited are similar, with differences the performance of canonical sliding-midpoint being worst with larger data sets and lower dimensions. Thus it would reasonable to conjecture that our upper bounds on the performance of canonical sliding-midpoint may hold for standard sliding-midpoint as well.

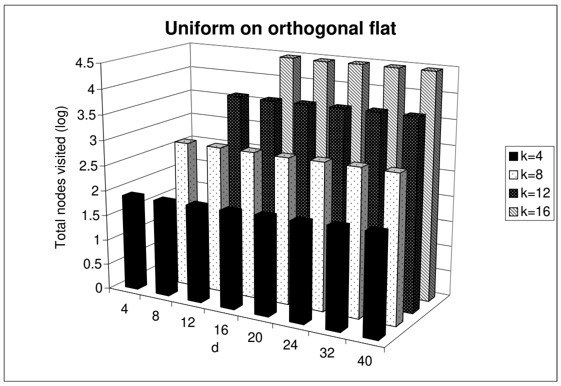
| Gaussian distribution | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $n$ | 40 | | 640 | | 10,240 | | 163,840 | |
| | c-sl | sl | c-sl | sl | c-sl | sl | c-sl | sl |
| $d = 4$ | 27.55 | 27.55 | 60.09 | 59.9 | 88.43 | 80.91 | 91.5 | 89.58 |
| $d = 8$ | | | 354.9 | 355 | 999.8 | 972.2 | 1447 | 1439 |
| Cluster-orthogonal-ellipsoids distribution | | | | | | | | |
| $n$ | 40 | | 640 | | 10,240 | | 163,840 | |
| | c-sl | sl | c-sl | sl | c-sl | sl | c-sl | sl |
| $d = 4$ | 9.675 | 7.925 | 28.05 | 26.04 | 67.18 | 60.43 | 102.2 | 82.55 |
| $d = 8$ | | | 44.58 | 43.46 | 262.3 | 256.2 | 833 | 782.4 |

Figure 4: Total number of nodes visited by search algorithm (c-sl: canonical sliding-midpoint, sl: sliding-midpoint)

### 6.3 Points on a $k$-flat

To support our theoretical bounds on number of leaf nodes visited when the point set is on a $k$-flat, we set up an experiment with both $k$ and $d$ varying, while fixing the other parameters. This allows us to observe the dependency of the query performance (in terms of the number of nodes visited) relative to $d$ and $k$. The Uniform-on-orthogonal-flat and Uniform-on-rotated-flat distributions were used in the experiments. We fixed $d$ at 4, 8, 12, 16, 20, 24, 32, 40 (note that the scale is nonlinear), and $k$ ranged from 4 to $\min(d, 16)$. Again, the number of points, $n$ ranged from 40 to 163,840. For Uniform-on-rotated-flat, the number of random rotations is $r = d^2/2$. The case of perturbed data is presented, but the results were very similar to the case of data points lying exactly on the flat. The queries were sampled from the same distribution. Number of query points was set to $\min(n, 2560)$.

Figure 5 shows the average number of nodes visited, counting both internal and leaf nodes (total) and just leaf nodes visited by the priority search when the tree was built by the canonical sliding-midpoint method. We shows the results only for $n = 163,840$ since the relative performance of other cases is similar. Also, the performance of the splitting method is identical to one of the sliding-midpoint method in this distribution throughout the range of parameters we tested.



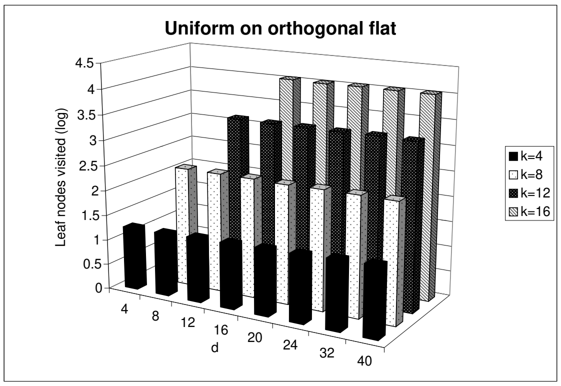
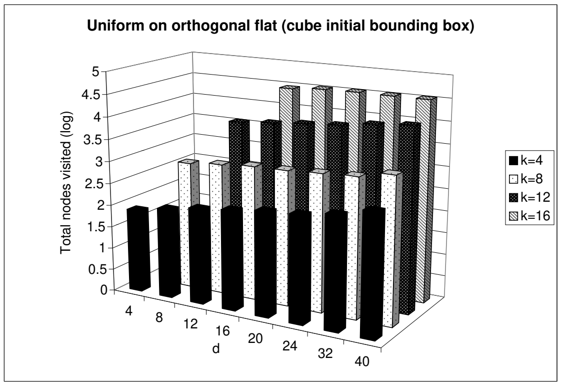




Figure 5: Number of total and leaf nodes visited, $n = 163,840$, Uniform-on-orthogonal-flat distribution

It is easy to see that the number of nodes visited depends only on $k$ for a fixed $n$. The performance does not respond to the change of $d$. This is because of the way the tree is built. In ANN library [MA97], the kd-tree is constructed recursively. Initially, there is one node, the root, that contains all points. Also it computes the tight bounding box of the point set in order to determine the splitting value and splitting dimension. Since the points in Uniform-on-orthogonal-flat lie on $k$-subspace, so does the initial bounding box. Therefore, the splitting methods never choose one of $d - k$ dimensions that the flat does not lie on as the splitting dimension.



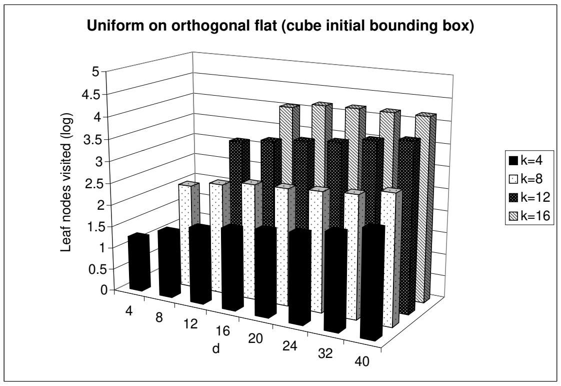




Figure 6: Number of total and leaf nodes visited, $n = 163,840$, Uniform-on-orthogonal-flat distribution with cube initial bounding box

In order to observe the behavior of the scenario considered in Theorem 5.1, we modified the library so that the initial bounding box is the hypercube $[-1, 1]^d$. The results of this modification are showed in Fig. 6. Note that we plotted the logarithm base 10 of the number of nodes visited. As predicted, the running shows a strong dependence on $k$, and very little dependence on $d$. However, it does not grow as fast as what Theorem 5.1 predicts. This suggests that the average case is much better than our theoretical bounds.

The Uniform-on-rotated-flat distribution is also used in the experiment to see the effect assuming that data is uniform on an arbitrarily oriented flat. Fig. 7 shows the results of this distribution, again showing the logarithm of the number of nodes visited and for $n = 163,840$. In this distribution, canonical sliding-midpoint is a little slower (typically, the difference is less than 5%) than sliding-midpoint in few cases. The plot shows the result of trees built by the canonical sliding-midpoint method. Notice that the number of nodes visited still shows a greater dependence on $k$ than on $d$, but the dependence on $d$ has increased, again as predicted by Theorem 4.1. Yet, the growth rate is still less than what the theorem predicts.

We also tested the sensitivity of our result to the presence of multiple clusters. Each data set consists of four clusters generated on four different hyperplanes for the uniform-on-rotated-flat distribution. Fig. 8 shows the results of this experiment.

## 6.4 Standard kd-tree

We were also interested in test the performance of trees built by the standard kd-tree method. We tested with Uniform-on-rotated-flat distribution with the same parameters as above. The results, showed in Fig. 9, are quite similar to the other splitting methods. Our experience has shown that standard kd-tree tends to perform very well when data and query points are taken from the same dimension, even though it seems to be much harder to establish theoretical bounds on its performance.

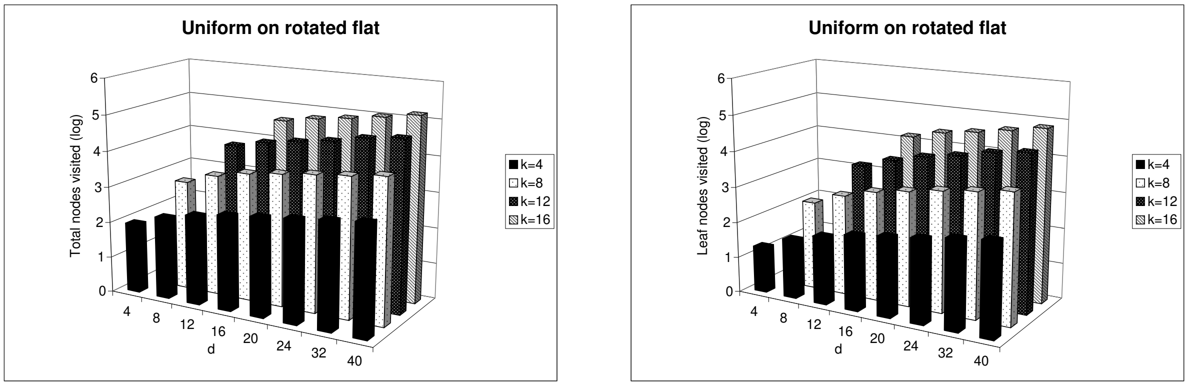



Figure 7: Number of total and leaf nodes visited, $n = 163,840$, Uniform-on-rotated-flat distribution

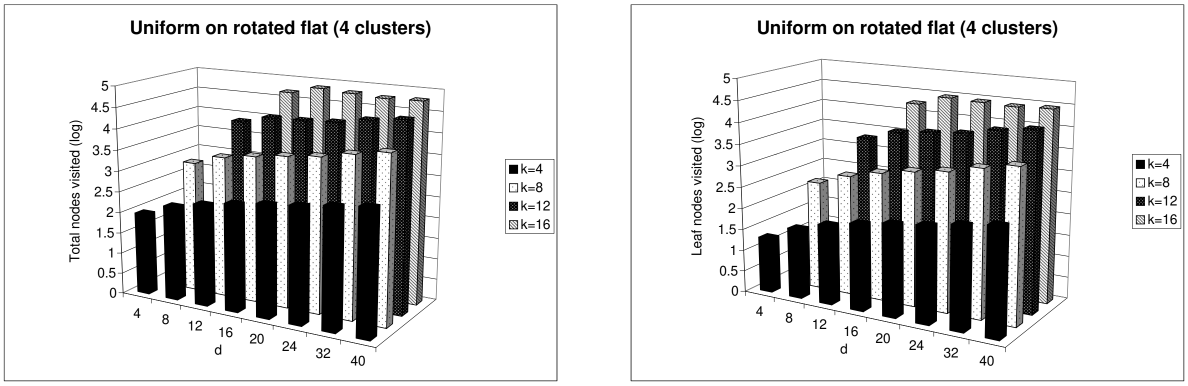



Figure 8: Number of total and leaf nodes visited, $n = 163,840$, four clusters Uniform-on-rotated-flat distribution

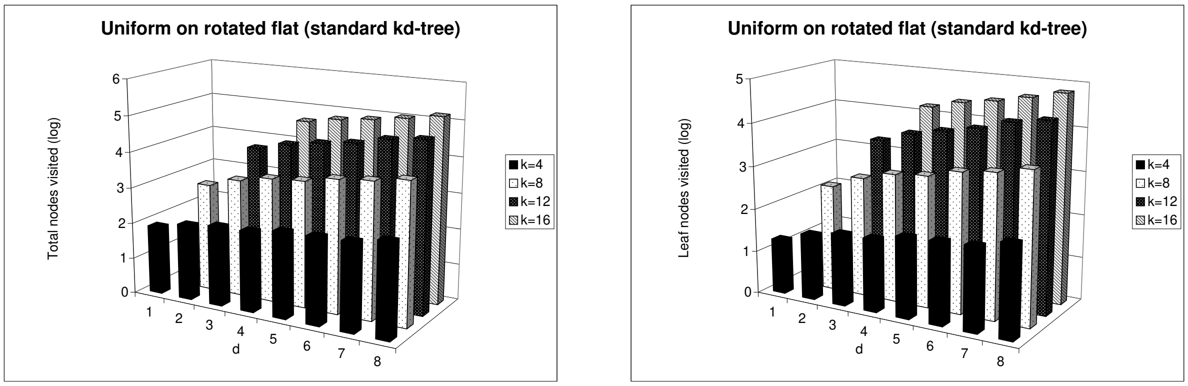



Figure 9: Number of total and leaf nodes visited, $n = 163,840$, Uniform-on-rotated-flat distribution, (standard kd-tree method is used)

## 6.5 Comparison with theoretical results

In this section, we take a closer look on whether or not our theoretical bounds can predict the actual query performance in terms of the number of leaf nodes visited. From Corollary 4.1, the expected number of leaf nodes of a kd-tree encountered in the search is $O(d^{k+2})$. We model this bound as $L = c_1(c_2 d)^{c_3 k}$, where $L$ is the number of leaf nodes visited and $c_1, c_2, c_3$ are constants. We set up the experiment such that the data and query distributions are uniform-on-rotated-flat. The parameters are slightly different from the previous experiments. The number of random rotations is $d^2$, and there is no gaussian noise. The number of data points, $n$, remains at 163,840 [illegible] $20, 40, 80$. The results are plotted in Fig 10.

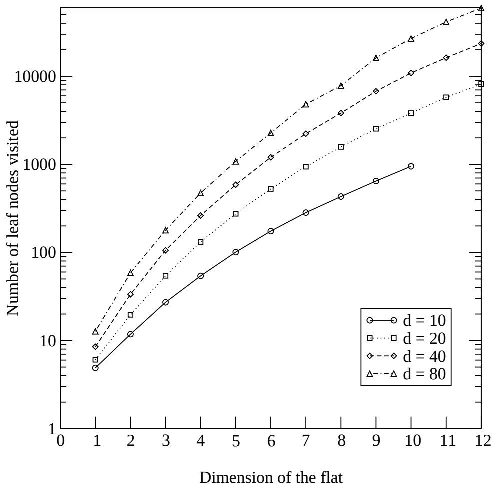



Figure 10: Number of leaf nodes visited, $n = 163,840$, Uniform-on-rotated-flat distribution

The model suggests that the curves (each for a fixed value of $d$) in Fig 10 should be linear. However, the empirical results show that it is not the case. Our conjecture is that this is due to boundary effects, which would presumably diminish as $n$ increases. These boundary effects are more pronounced for larger values of $k$ [AMN96]. Because of memory limitation, we cannot scale $n$ exponentially with the value of $k$.

We observed that for smaller values of $k$ (e.g. $k = 1, 2, 3$), the number of leaf nodes visited, $L$, is almost unchanged when $n$ is increased. It indicates the boundary effects are minimum within this range of $k$. Therefore we use the results from $k = 1, 2$ to find values of $c_1, c_2, c_3$ of our model equation. This yields the following equation,

$$L = 2.054(1.674 \cdot d)^{(0.312 \cdot k)}.$$

# 7 Acknowledgements

We would like to thank Sunil Arya for many helpful conversations and suggestions.

# References

[AMN96] S. Arya, D. M. Mount, and O. Narayan. Accounting for boundary effects in nearest neighbor searching. *Discrete Comput. Geom.*, 16(2):155–176, 1996.

[AMN+98] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45:891–923, 1998.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BET93] M. Bern, D. Eppstein, and S.-H. Teng. Parallel construction of quadtrees and quality triangulations. In *Proc. 3rd Workshop Algorithms Data Struct.*, volume 709 of *Lecture Notes in Computer Science*, pages 188–199. Springer-Verlag, 1993.

[BWY80] J. L. Bentley, B. W. Weide, and A. C. Yao. Optimal expected-time algorithms for closest-point problems. *ACM Trans. Math. Software*, 6(4):563–580, 1980.

[Cla88] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM J. Comput.*, 17(4):830–847, 1988.

[Cle79] J. G. Cleary. Analysis of an algorithm for finding nearest neighbors in euclidean space. *ACM Trans. Math. Software*, 5(2):183–192, 1979.

[CP96] M. Carreira-Perpiñán. A review of dimension reduction techniques. Technical Report CS–96–09, Dept. of Computer Science, University of Sheffield, UK, 1996.

[DGK99] C. Duncan, M. Goodrich, and S. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 300–309, 1999.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.

[FBF77] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Software*, 3(3):209–226, 1977.

[FK94] Christos Faloutsos and Ibrahim Kamel. Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension. In *Proc. Annu. ACM Sympos. Principles Database Syst.*, pages 4–13, 1994.

[Fuk90] K. Fukunaga. *Introduction to Statistical Pattern Recognition.* Academic Press, 2nd edition, 1990.

[IM98] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. 30th Annu. ACM Sympos. Theory Comput.*, pages 604–613, 1998.

[MA97] D. M. Mount and S. Arya. ANN: A library for approximate nearest neighbor searching. Center for Geometric Computing 2nd Annual Workshop on Computational Geometry, 1997.

[MM99a] S. Maneewongvatana and D. Mount. Analysis of approximate nearest neighbor searching with clustered point sets. In *ALENEX*, 1999.

[MM99b] S. Maneewongvatana and D. Mount. It's okay to be skinny, if your friends are fat. Center for Geometric Computing 4th Annual Workshop on Computational Geometry, 1999.