Fast Nearest Neighbor Search in SE(3) for Sampling-Based Motion Planning

Jeffrey Ichnowski and Ron Alterovitz

University of North Carolina at Chapel Hill, USA
{jeffi,ron}@cs.unc.edu

Abstract. Nearest neighbor searching is a fundamental building block of most sampling-based motion planners. We present a novel method for fast exact nearest neighbor searching in $SE(3)$—the 6 dimensional space that represents rotations and translations in 3 dimensions. $SE(3)$ is commonly used when planning the motions of rigid body robots. Our approach starts by projecting a 4-dimensional cube onto the 3-sphere that is created by the unit quaternion representation of rotations in the rotational group $SO(3)$. We then use 4 kd-trees to efficiently partition the projected faces (and their negatives). We propose efficient methods to handle the recursion pruning checks that arise with this kd-tree splitting approach, discuss splitting strategies that support dynamic data sets, and extend this approach to $SE(3)$ by incorporating translations. We integrate our approach into RRT and RRT* and demonstrate the fast performance and efficient scaling of our nearest neighbor search as the tree size increases.

1 Introduction

Nearest neighbor searching is a critical component of commonly used motion planners. Sampling-based methods, such as probabilistic roadmaps (PRM), rapidly exploring random trees (RRT), and RRT* [1, 2], create a motion plan by building a graph in which vertices represent collision-free robot configurations and edges represent motions between configurations. To build the graph, these motion planners repeatedly sample robot configurations and search for nearest neighbor configurations already in the graph to identify promising collision-free motions.

Because nearest neighbor search is a fundamental building block of most sampling-based motion planners, speeding up nearest neighbor searching will accelerate many commonly used planners. This is especially true for asymptotically optimal motion planners, which typically require a large number of samples to compute high-quality plans. As the number of samples in the motion planning graph rises, nearest neighbor search time grows logarithmically (or at worst linearly). As the samples fill the space, the expected distance between samples shrinks, and correspondingly reduces the time required for collision detection. Collision detection typically dominates computation time in early iterations, but

as the number of iterations rises, nearest neighbor search will dominate the overall computation—increasing the importance of fast nearest neighbor searches.

We introduce a fast, scalable exact nearest neighbor search method for robots modeled as rigid bodies. Many motion planning problems involve rigid bodies, from the classic piano mover problem to planning for aerial vehicles. A planner can represent the configuration of a rigid body in 3D by its 6 degrees of freedom: three translational (e.g., x, y, z) and three rotational (e.g., yaw, pitch, roll). The group of all rotations in 3D Euclidean space is the special orthogonal group $SO(3)$. The combination of $SO(3)$ with Euclidean translation in space is the special Euclidean group $SE(3)$.

Our approach uses a set of kd-trees specialized for nearest neighbor searches in $SO(3)$ and $SE(3)$ for dynamic data sets. A kd-tree is a binary space partitioning tree data structure that successively splits space by axis-aligned planes. It is particularly well suited for nearest neighbor searches in Minkowski distance (e.g., Euclidean) real-vector spaces. However, standard axis-aligned partitioning approaches that apply to real-vector spaces do not directly apply to rotational spaces due to their curved and wrap-around nature.

In this paper, we describe a novel way of partitioning $SO(3)$ space to create a kd-tree search structure for $SO(3)$ and by extension $SE(3)$. Our approach can be viewed as projecting the surface of a 4-dimensional cube onto a 3-sphere (the surface of a 4-dimensional sphere), and subsequently partitioning the projected faces of the cube. The 3-sphere arises from representing rotations as 4-dimensional vectors of unit quaternions. The projection and partitioning we describe has two important benefits: (1) the dimensionality of the rotation space is reduced from its 4-dimensional quaternion representation to 3 (its actual degrees of freedom), and (2) the splitting hyperplanes efficiently partition space allowing the kd-tree search to check fewer kd-tree nodes. We propose efficient methods to handle the recursion pruning checks that arise with this kd-tree splitting approach, and also discuss splitting strategies that support dynamic data sets. Our approach for creating rotational splits enables our kd-tree implementation to achieve fast nearest neighbor search times for dynamic data sets.

We demonstrate the speed of our nearest neighbor search approach on scenarios in OMPL [3] and demonstrate a significant speedup compared to state-of-the-art nearest neighbor search methods for $SO(3)$ and $SE(3)$.

2 Related Work

Nearest neighbor searching is a critical component in sampling-based motion planners [1]. These planners use nearest neighbor search data structures to find and connect configurations in order to compute a motion plan.

Spatial partitioning trees such as the kd-tree [4–6], quadtrees and higher dimensional variants [7], and vp-trees [8] can efficiently handle exact nearest neighbor searching in lower dimensions. These structures generally perform well on data in a Euclidean metric space, but because of their partitioning mechanism (e.g., axis-aligned splits), they do not readily adapt to the rotational group

$SO(3)$. Kd-trees have a static construction that can guarantee a perfectly balanced tree for a fixed (non-dynamic) data set. Bentley showed how to do a static-to-dynamic conversion [9] that maintains the benefits of the balanced structure produced by static construction, while adding the ability to dynamically update the structure without significant loss of asymptotic performance.

Yershova and LaValle [10] showed how to extend kd-trees to handle $\mathbb{R}^1$, S^1, $SO(3)$, and the Cartesian product of any number of these spaces. Similar to kd-trees built for $\mathbb{R}^m$, they split $SO(3)$ using rectilinear axis-aligned planes created by a quaternion representation of the rotations [11]. Although performing well in many cases, rectilinear splits produce inefficient partitions of $SO(3)$ near the corners of the partitions. Our method eschews rectilinear splits in favor of splits along rotational axes, resulting in splits that more uniformly partition $SO(3)$.

Non-Euclidean spaces, including $SO(3)$, can be searched by general metric space nearest neighbor search data structures such as GNAT [12], cover-trees [13], and M-trees [14]. These data structures generally perform better than linear searching. However, except for rare pathological cases, these methods are usually outperformed by kd-trees in practice [10].

Nearest neighbor searching is often a performance bottleneck of sampling-based motion planning, particularly when the dimensionality of the space increases [15, 16]. It is sometimes desirable in such cases to sacrifice accuracy for speed by using approximate methods [15–19]. These methods can dramatically reduce computation time for nearest neighbor searches, but it is unclear if the proofs of optimality for asymptotically optimal motion planners hold when using approximate searches. Our focus is on exact searches, though we believe that some approximate kd-tree speedups can be applied to our method.

3 Problem Definition

Let $\mathcal{C}$ be the configuration space of the robot. For a rigid-body robot, the configuration space is $\mathcal{C} = \mathbb{R}^m$ if the robot can translate in m dimensions, $\mathcal{C} = SO(3) = P^3$ if the robot can freely rotate in 3 dimensions, and $\mathcal{C} = SE(3) = \mathbb{R}^3 P^3$ if the robot can freely translate and rotate in 3 dimensions. Let $\mathbf{q} \in \mathcal{C}$ denote a configuration of the robot. When $\mathcal{C} = \mathbb{R}^m$, $\mathbf{q}$ is an m-dimensional real vector. When $\mathcal{C} = P^3$, we define $\mathbf{q}$ as a 4-dimensional real vector in the form (a, b, c, d) representing the components of a unit quaternion $\mathbf{q} = a + bi + cj + dk$. We use the notation $\mathbf{q}[x]$ to represent the x component of a configuration $\mathbf{q}$.

Computation of nearest neighbors depends on the chosen distance metric. Let $\texttt{DIST}(\mathbf{q}_1, \mathbf{q}_2)$ be the distance between two configurations. For brevity, we will focus on a few commonly used distance functions, which are included in OMPL [3]. We only consider exact functions, and approximate versions are left to future work. In $\mathbb{R}^m$ we use the Euclidean (L^2) distance:

$$\texttt{DIST}_{\mathbb{R}^m}(\mathbf{q}_1, \mathbf{q}_2) = \left(\sum_{i=1}^{m} (\mathbf{q}_1[i] - \mathbf{q}_2[i])^2 \right)^{1/2}$$

In P^3 we use a distance of the shorter of the two angles subtended along the great arc between the rotations [11, 3, 10]. This metric is akin to a straight-line

distance in Euclidean space mapped on a 3-sphere:

$$\texttt{DIST}_{P^3}(\mathbf{q}_1, \mathbf{q}_2) = \cos^{-1} |\mathbf{q}_1 \cdot \mathbf{q}_2| = \cos^{-1} \left| \sum_{i \in \{a,b,c,d\}} \mathbf{q}_1[i]\mathbf{q}_2[i] \right|.$$

In $\mathbb{R}^3 P^3$, we use the weighted sum of the $\mathbb{R}^3$ and P^3 distances [3]:

$$\texttt{DIST}_{\mathbb{R}^m P^3}(\mathbf{q}_1, \mathbf{q}_2) = \alpha \, \texttt{DIST}_{\mathbb{R}^m}(\mathbf{q}_1, \mathbf{q}_2) + \texttt{DIST}_{P^3}(\mathbf{q}_1, \mathbf{q}_2).$$

where $\alpha > 0$ is a user-specified weighting factor. We assume the distance function is symmetric, i.e., $\texttt{DIST}(\mathbf{q}_1, \mathbf{q}_2) = \texttt{DIST}(\mathbf{q}_2, \mathbf{q}_1)$, and define $\texttt{DIST}(\mathbf{q}, \varnothing) = \infty$.

We apply our approach to solve three variants of the nearest neighbor search problem commonly used in sampling-based motion planning. Let $\mathbf{Q}$ denote a set of n configurations $\{\mathbf{q}_1 \ldots \mathbf{q}_n\} \subset \mathcal{C}$. Given a configuration $\mathbf{q}_{\text{search}}$, the *nearest neighbor search* problem is to find the $\mathbf{q}_i \in \mathbf{Q}$ with the minimum $\texttt{DIST}(\mathbf{q}_{\text{search}}, \mathbf{q}_i)$. In the *k-nearest neighbors* variant, where k is a positive integer, the objective is to find a set of k configurations in $\mathbf{Q}$ nearest to $\mathbf{q}_{\text{search}}$. In the *nearest neighbors in radius r* search, where r is a positive real number, the objective is to find all configurations in $\mathbf{Q}$ with $\texttt{DIST}(\mathbf{q}_{\text{search}}, \mathbf{q}_i) \leq r$.

Sampling-based motion planners make many calls to the above functions when computing a motion plan. Depending on the planner, the set of nodes $\mathbf{Q}$ is either a static data set that is constant for each query or $\mathbf{Q}$ is a dynamic data set that changes between queries. Our objective is to achieve efficiency and scalability for all the above variants of the nearest neighbor search problem for static and dynamic data sets in $SO(3)$ and $SE(3)$.

4 Method

A kd-tree is a binary tree in which each branch node splits space by an axis-aligned hyperplane, and each child's subtree contains only configurations from one side of the hyperplane. In a real vector metric space, such as Euclidean space, it is common for each split to be defined by an axis-aligned hyperplane, though other formulations are possible [6]. For performance reasons it is often desirable for the splits to evenly partition the space, making median or mean splits good choices [20]. We will describe these methods and how to apply our $SO(3)$ partition scheme to them.

In our method, we eschew rectilinear axis-aligned splits in favor of partitions that curve with the manifold of $SO(3)$ space. The set of all unit quaternion representations of rotations in $SO(3)$ forms the surface of a 4-dimensional sphere (a 3-sphere). We partition this space by projecting the surface of a 4-dimensional cube onto the surface of the 3-sphere. Because of the double-coverage property in which a quaternion and its negative represent the same rotation [11], half of the projected surface volumes are redundant, and we build kd-trees by subdividing 4 of the projected surface volumes. Similar projections are used in [21] to generate deterministic samples in $SO(3)$, and in [22] to create a minimum spanning tree on a recursive octree subdivision of $SO(3)$. When subdividing the surface

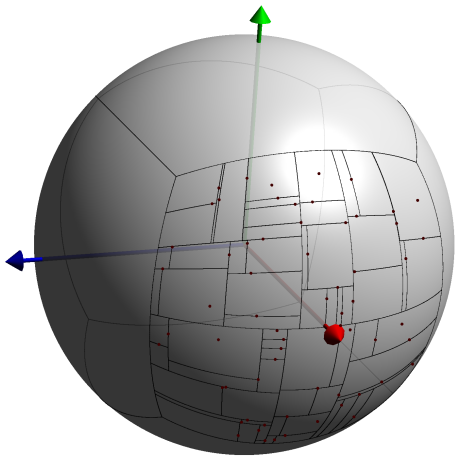


Fig. 1. A kd-tree projected onto the surface of a 2-sphere. An axis-orthogonal cube is projected into a sphere. Each face of the cube is a separately computed kd-tree; however, for illustrative purposes, we show the kd-tree of only one of the faces. In our method we extend the analogy to 4-dimensional space for use with quaternions.

volumes into kd-trees, we apply a novel approach in which the partitioning hyperplanes pass through the center of the 3-sphere, and thus radially divide space. These partitions are curved, and thus standard kd-tree approaches that apply to real-vector spaces must be adapted to maintain consistency with the great arc distance metric we use for $SO(3)$. In Fig. 1, we depict a lower dimensional analog consisting of a 3-dimensional cube projected onto a 2-sphere, with only one of the projected cube surfaces subdivided into a kd-tree.

4.1 Projected Volume Partitioning of $SO(3)$

In the projection of the surface of a 4D cube onto the surface of a 3-sphere we label each of the projected 3D surface volumes by the axis on which they are aligned, thus a, b, c, and d. Any configuration whose quaternion is in a negative volume (e.g., $-a$, $-b$, $-c$, or $-d$) is inverted.

The advantage of using this projection is two-fold: (1) we reduce the dimensionality of the rotation representation from a 4-dimensional quaternion to a 3-dimensional position on the projected volume, and (2) it allows radially aligned splitting hyperplanes that more uniformly divide the curved manifold. There is, however, a small cost for these benefits. The projection leads to building 4 kd-trees, although asymptotically the cost is at worst a constant factor.

To determine in which projected volume a quaternion $\mathbf{q}$ lies, we find its component of greatest magnitude. Thus:

$$\texttt{proj\_volume}\,(\mathbf{q}) = \underset{i \in a,b,c,d}{\operatorname{argmax}} |\boldsymbol{q}[i]|$$

If θ is the angle between the unit quaternions $\mathbf{q}$ and $\mathbf{n}$, then $\mathbf{q} \cdot \mathbf{n} = \cos\theta$. We use this property and represent bounding and splitting hyperplanes by their normals

n. Determining on which side a quaternion **q** lies is a matter of evaluating the sign of the dot product—positive values are on one side, negative values are on the other, and a dot product of 0 lies on the hyperplane.

We will focus our discussion on the projected a-volume, with the other volumes (b, c, and d) being permutations on it. The normals bounding the 6 sides of the projected a-volume are the unit quaternions normalized from:

$(1, 1, 0, 0)$	$(-1, 1, 0, 0)$	b-axis bounds
$(1, 0, 1, 0)$	$(-1, 0, 1, 0)$	c-axis bounds
$(1, 0, 0, 1)$	$(-1, 0, 0, 1)$	d-axis bounds

We observe that within the projected a-volume, the a component of the hyperplane normals varies between $\sqrt{0.5}$ and $-\sqrt{0.5}$ (after normalizing), the axis component varies between $\sqrt{0.5}$ at the boundaries to 1 at $a = 0$, and the other components are always zero. The bounds for the b, c, and d projected volumes follow similarly.

Solving for $\mathbf{n}$ in $\mathbf{q} \cdot \mathbf{n} = 0$, we can determine the normal of the axis-aligned hyperplane that passes through the quaternion $\mathbf{q}$. We define $\texttt{axisnorm}_{\text{vol,axis}}(\mathbf{q})$ as the axis-aligned normal within a projected volume for quaternion $\mathbf{q}$. The a-volume definitions are:

$$\begin{aligned}
\texttt{axisnorm}_{a\text{-vol},b\text{-axis}}(\mathbf{q}) &= \texttt{normalize}(-\mathbf{q}[b], \mathbf{q}[a], 0, 0) \\
\texttt{axisnorm}_{a\text{-vol},c\text{-axis}}(\mathbf{q}) &= \texttt{normalize}(-\mathbf{q}[c], 0, \mathbf{q}[a], 0) \\
\texttt{axisnorm}_{a\text{-vol},d\text{-axis}}(\mathbf{q}) &= \texttt{normalize}(-\mathbf{q}[d], 0, 0, \mathbf{q}[a]),
\end{aligned}$$

where $\texttt{normalize}(\mathbf{q})$ normalizes its input vector to a unit quaternion. From the `axisnorm` we define an angle of rotation about the axis. The angle is computed as the arctangent of the normal's volume component over the normal's axis component, thus for example, $\mathbf{q}$'s angle about the b-axis in the a-volume is $\tan^{-1}(-\mathbf{q}[a]/\mathbf{q}[b])$. This angle forms the basis for a relative ordering around an axis, and can be shortcut by comparing the volume component alone, as $\mathbf{q}_1[a] < \mathbf{q}_2[a] \iff \tan^{-1}(-\mathbf{q}_1[a]/\mathbf{q}_1[b]) > \tan^{-1}(-\mathbf{q}_2[a]/\mathbf{q}_2[b])$.

4.2 Static KD-Tree

In a static nearest neighbor problem, in which $\mathbf{Q}$ does not change, we can use an efficient one-time kd-tree construction that allows for well balanced trees. Alg. 1 outlines a static construction method for kd-trees on real-vector spaces.

The algorithm works as follows. First it checks if there is only one configuration, and if so it returns a leaf node with the single configuration (lines 1–2). Otherwise the set of configurations is partitioned into two subsets to create a branch. `CHOOSE_PARTITION_AXIS` ($\mathbf{Q}$) in line 4 choses the axis of the partition. A number of policies for choosing the axis are possible, e.g., splitting along the axis of greatest extent. Then, `PARTITION` ($\mathbf{Q}$, `axis`) (line 5) splits $\mathbf{Q}$ along the `axis` into the partially ordered set $\mathbf{Q}'$ such that $\forall \mathbf{q}_\text{i} \in \mathbf{Q}'_{1..\text{m}-1} : \mathbf{q}_\text{i}[\texttt{axis}] \leq \textbf{split}$ and $\forall \mathbf{q}_\text{j} \in \mathbf{Q}'_{\text{m}..\text{n}} : \mathbf{q}_\text{j}[\texttt{axis}] \geq \textbf{split}$. Thus a median split chooses $m = n/2$.

Algorithm 1 `BuildKDTree` $(\mathbf{Q})$

Require: $\mathbf{Q}$ is a set of configurations of size $\mathbf{n} > 0$

if $\mathbf{Q}$ has 1 configuration **then**
  return leaf node with $\mathbf{Q}_1$
else
  `axis` $\leftarrow$ `CHOOSE_PARTITION_AXIS` $(\mathbf{Q})$
  $(\mathbf{Q}', \texttt{split}, \texttt{m}) \leftarrow$ `PARTITION` $(\mathbf{Q}, \texttt{axis})$
  `left` $\leftarrow$ `BuildKDTree` $\left(\mathbf{Q}'_{1..\mathrm{m}-1}\right)$
  `right` $\leftarrow$ `BuildKDTree` $\left(\mathbf{Q}'_{\mathrm{m}..\mathrm{n}}\right)$
  return branch node with split on (`axis`, `split`) and children (`left`, `right`)

The `PARTITION` function is implemented efficiently either by using a partial-sort algorithm, or sorting along each axis before building the tree. Assuming median splits, `BuildKDTree` builds a kd-tree in $O(n \log n)$ time using a partial-sort algorithm.

In our $SO(3)$ projection, we define an axis comparison that allows us to find the minimum and maximum along each projected axis, and to perform the partial sort required for a median partition. The axis comparison is the relative ordering of each quaternion's `axisnorm` angle for that volume and projection.

The minimum and maximum extent along each axis is the quaternion for which all others are not-less-than or not-greater-than, respectively, any other quaternion in the set. The angle of the arc subtending the minimum and maximum `axisnorm` values is the axis's extent. Thus, if we define $\mathbf{N}$ as the set of all `axisnorm` values for $\mathbf{Q}$ in the a-volume and along the b-axis therein: $\mathbf{N}_{\mathrm{a,b}} = \{\texttt{axisnorm}_{a\text{-vol},b\text{-axis}}(\mathbf{q}) : \mathbf{q} \in \mathbf{Q}\}$, then the minimum and maximum `axisnorm` along the b-axis is:

$$\mathbf{n}_{\min} = \underset{\mathbf{n}_{\mathrm{i}} \in \mathbf{N}_{\mathrm{a,b}}}{\operatorname{argmin}} \mathbf{n}_{\mathrm{i}}[a] \qquad \mathbf{n}_{\max} = \underset{\mathbf{n}_{\mathrm{j}} \in \mathbf{N}_{\mathrm{a,b}}}{\operatorname{argmax}} \mathbf{n}_{\mathrm{j}}[a]$$

and the angle of extent is $\cos^{-1} |\mathbf{n}_{\min} \cdot \mathbf{n}_{\max}|$. After computing the angle of extent for all axes in the volume, we select the greatest of them and that becomes our axis of greatest extent.

4.3 Dynamic KD-Tree

Sampling-based motion planners, such as RRT and RRT*, generate and potentially add a random configuration to the dataset at every iteration. For these algorithms, the nearest neighbor searching structure must be dynamic—that is, it must support fast insertions and removals interleaved with searches. In [9], Bentley and Saxe show that one approach is to perform a "static-to-dynamic conversion". Their method builds multiple static kd-trees of varying sizes in a manner in which the amortized insertion time is $O(\log^2 n)$ and the expected query time is $O(\log^2 n)$. In the text that follows, we describe our implementation for modifying the kd-tree to a dynamic structure, and we compare the approaches in Sec. 5.

Algorithm 2 `DynamicKDInsert` $(\mathbf{q})$

```
1: n ← &kdroot
2: (C_min, C_max) ← volume bounds
3: for depth = 0 → ∞ do
4:    (axis, split) ← KD_SPLIT (C_min, C_max, depth)
5:    if n = ∅ then
6:       *n ← new node with (axis, split, q)
7:       return
8:    if q[axis] < split then
9:       n ← & (*n_left)
10:      C_max[axis] ← split
11:   else
12:      n ← & (*n_right)
13:      C_min[axis] ← split
```

The kd-tree may also be easily modified into a dynamic structure by allowing children to be added to the leaves of the structure, and embedding a configuration in each tree node. When building such a dynamic kd-tree, the algorithm does not have the complete dataset, and thus cannot perform a balanced construction like the median partitioning in Sec. 4.2. Instead, it chooses splits based upon an estimate of what is likely to be the nature of the dataset. When values are inserted in random order into a binary tree, Knuth [23, p. 430–431] shows that well-balanced trees are common, with insertions requiring about $2 \ln n$ comparisons, and the worst-case $O(n)$ is rare. In our experiments, we observe results suggesting that the generated trees are indeed well-balanced across a variety of scenarios. In the results section, we split at the midpoint of the bounding box. A few possible choices that empirically work well with sampling-based motion planners are: (1) split at the midpoint of the bounding box implied by the configuration space and the prior splits, (2) split at the hyperplane defined by the point being added, or (3) an interpolated combination of the two.

`DynamicKDInsert` (Alg. 2) adds a configuration into a dynamic kd-tree. In this formulation, each node in the kd-tree contains a configuration, an axis and split value, and two (possibly empty) child nodes. Given the bounding box of the volume and a depth in the tree, the `KD_SPLIT` function (line 4) generates a splitting axis and value. In Euclidean space, `KD_SPLIT` can generate a midpoint split along the axis of greatest extent by choosing the `axis` that maximizes $\mathbf{C}_{\max}[\texttt{axis}] - \mathbf{C}_{\min}[\texttt{axis}]$, and the split value of $(\mathbf{C}_{\min}[\texttt{axis}] + \mathbf{C}_{\max}[\texttt{axis}])/2$.

In our $SO(3)$ projection, the axis of greatest extent is computed from the angle between $\mathbf{c}_{\min}$ and $\mathbf{c}_{\max}$, where $\mathbf{c}_{\min}$ and $\mathbf{c}_{\max}$ are an axis's bounding hyperplane normals from $\mathbf{C}_{\min}$ and $\mathbf{C}_{\max}$. An interpolated split is computed using a spherical linear interpolation [11] between the bounds:

$$\mathbf{c}_{\text{split}} = \mathbf{c}_{\min} \frac{\sin t\theta}{\sin \theta} + \mathbf{c}_{\max} \frac{\sin(1-t)\theta}{\sin \theta} \qquad \text{where} \qquad \theta = \cos^{-1} |\mathbf{c}_{\min} \cdot \mathbf{c}_{\max}|$$

A split at the midpoint ($t = 0.5$) simplifies to $\mathbf{c}_{\text{mid}} = (\mathbf{c}_{\min} + \mathbf{c}_{\max})/(2 \cos \frac{\theta}{2})$.

Algorithm 3 $\texttt{DynamicKDSearch}\,(\mathbf{q}, \mathtt{n}, \mathtt{depth}, \mathbf{C}_{\min}, \mathbf{C}_{\max}, \mathbf{q}_{\text{nearest}}, \mathbf{s}, \mathbf{a})$

```
 1: if n = ∅ then
 2:     return q_nearest
 3: if DIST(q, q_n) < DIST(q, q_nearest) then
 4:     q_nearest ← q_n                          // q_n is the configuration associated with n
 5: (axis, split) ← KD_SPLIT(C_min, C_max, depth)
 6: (C'_min, C'_max) ← (C_min, C_max)
 7: C'_min[axis] ← C'_max[axis] ← split
 8: if q[axis] < split then
 9:     q_nearest ← DynamicKDSearch(q, n_left, depth + 1, C_min, C'_max, q_nearest, s, a)
10: else
11:     q_nearest ← DynamicKDSearch(q, n_right, depth + 1, C'_min, C_max, q_nearest, s, a)
12: s[axis] ← split
13: a[axis] ← 1
14: if PARTIAL_DIST(q, s, a) ≤ DIST(q, q_nearest) then
15:     if q[axis] < split then
16:         q_nearest ← DynamicKDSearch(q, n_right, depth + 1, C'_min, C_max, q_nearest, s, a)
17:     else
18:         q_nearest ← DynamicKDSearch(q, n_left, depth + 1, C_min, C'_max, q_nearest, s, a)
19: return q_nearest
```

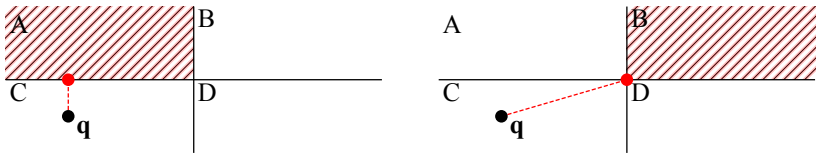
If instead we wish to split at the hyperplane that intersects the point being inserted, we use the `axisnorm` to define the hyperplane's normal. Furthermore, we may combine variations by interpolating between several options.

4.4 Kd-Tree Search

In Alg. 3, we present an algorithm of searching for a nearest neighbor configuration $\mathbf{q}$ in the dynamic kd-tree defined in Sec. 4.3. The search begins with $\mathtt{n}$ as the root of the kd-tree, a $\mathtt{depth}$ of 0, $\mathbf{C}_{\min}$ and $\mathbf{C}_{\max}$ as the root volume bounds, an empty $\mathbf{q}_{\text{nearest}}$, and the split vectors $\mathbf{s} = \mathbf{a} = \mathbf{0}$.

The search proceeds recursively, following the child node on the side of the splitting hyperplane on which $\mathbf{q}$ resides (lines 8–11). Upon return from recursion, the search algorithm checks if it is possible that the other child tree could contain a configuration closer to $\mathbf{q}$ than the nearest one. This check is performed against the bounding box created by the splitting hyperplanes of the ancestor nodes traversed to reach the current one. It is essentially the bounding box defined by $\mathbf{C'}_{\min}$ and $\mathbf{C'}_{\max}$. However, a full bounding box distance check is unnecessary—only the distance between the point and the bounds closest to the point are necessary. This distance is computed by the `PARTIAL_DIST` function, and is depicted in Fig. 2.

$\texttt{PARTIAL\_DIST}\,(\mathbf{q}, \mathbf{s}, \mathbf{a})$ (line 14) computes the distance between a configuration $\mathbf{q}$ and the corner of a volume defined by $\mathbf{s}$ and $\mathbf{a}$. The components of $\mathbf{s}$ are the split axis values between the current region and the region in which $\mathbf{q}$ resides. The components of $\mathbf{a}$ are 1 for each axis which is defined in $\mathbf{s}$ and 0 otherwise. This results in the `PARTIAL_DIST` definition for the L^2 distance metric: $\texttt{PARTIAL\_DIST}_{\mathrm{L}^2}\,(\mathbf{q}, \mathbf{s}, \mathbf{a}) = \left(\sum_{i=1}^{d} (\mathbf{q}_i - \mathbf{s}_i)^2 \mathbf{a}_i\right)^{1/2}$.



(a) single axis (b) double axis

Fig. 2. A kd-tree search for $\mathbf{q}$ determining if it should traverse the second node. The search checks if it is possible for any configuration in the region contained within the node to have a point closer than the one already found. In (a), the search computes the distance between $\mathbf{q}$ and region A—this is a 1-dimensional L^2 distance between $\mathbf{q}$ and the hyperplane that splits regions A and C. In (b), the search computes the distance between $\mathbf{q}$ and region B—and it computes a 2-dimensional L^2 distance. Our method extends this computation to the curved projection on a 3-sphere.

The partial distance metric must return a distance less than or equal to the closest possible configuration in the node's region. A poorly bound partial distance (e.g. $\texttt{PARTIAL\_DIST} = 0$) is valid, however search performance will suffer, dropping to $O(n)$ in the worst case. Thus a tightly bound `PARTIAL_DIST` is critical to performance.

The `PARTIAL_DIST` function in our projected volume mapping of $SO(3)$ is the distance between a configuration $\mathbf{q}$ and a volume defined by hyperplanes partitioning a unit 3-sphere, and is complicated by the curvature of the space. For this function to be tightly bounded, it must take into account that the volume defined by the bounds on our projected manifold are curved (see Fig. 1). When only 1 hyperplane is defined (i.e. the first split in $SO(3)$), the distance is the angle between a configuration and a great circle defined by a splitting hyperplane's normal $\mathbf{n}_{\text{split}}$ and its intersection with the unit 3-sphere. This distance is:

$$\texttt{PARTIAL\_DIST}_{P^3|\mathbf{n}_{\text{split}}} = \sin^{-1}(\mathbf{q} \cdot \mathbf{n}_{\text{split}})$$

When 2 of the 3 axes are split, the distance is the angle between the configuration and an ellipse. The ellipse results from projecting the line defined by the two splitting hyperplanes onto a unit 3-sphere. If the split axis values are the normals $\mathbf{n}_{\text{b}}$ and $\mathbf{n}_{\text{c}}$ in the projected a volume, and thus the d-axis is not yet split, the partial distance is:

$$\texttt{PARTIAL\_DIST}_{P^3|\mathbf{n}_{\text{b}},\mathbf{n}_{\text{c}}} = \min_{\omega} \cos^{-1} |\mathbf{q} \cdot \texttt{ell}(\mathbf{n}_{\text{b}}, \mathbf{n}_{\text{c}}, \omega)|$$

where `ell` is an ellipsoid parameterized by the normals $\mathbf{n}_{\text{b}}$ and $\mathbf{n}_{\text{c}}$, and varied over ω:

$$\texttt{ell}(\mathbf{n}_{\text{b}}, \mathbf{n}_{\text{c}}, \omega) = \left(\omega, -\omega \frac{\mathbf{n}_{\text{b}}[a]}{\mathbf{n}_{\text{b}}[b]}, -\omega \frac{\mathbf{n}_{\text{c}}[a]}{\mathbf{n}_{\text{c}}[c]}, \pm \sqrt{1 - \omega^2 - \left(\omega \frac{\mathbf{n}_{\text{b}}[a]}{\mathbf{n}_{\text{b}}[b]}\right)^2 - \left(\omega \frac{\mathbf{n}_{\text{c}}[a]}{\mathbf{n}_{\text{c}}[c]}\right)^2} \right)$$

The distance is minimized at $\omega = \gamma / \sqrt{\eta(\gamma^2 - \eta \mathbf{q}[a])}$ where

$$\gamma = \mathbf{q}[a] - \mathbf{q}[b] \frac{\mathbf{n}_{\text{b}}[a]}{\mathbf{n}_{\text{b}}[b]} - \mathbf{q}[c] \frac{\mathbf{n}_{\text{c}}[a]}{\mathbf{n}_{\text{c}}[c]}, \qquad \eta = 1 + \left(\frac{\mathbf{n}_{\text{b}}[a]}{\mathbf{n}_{\text{b}}[b]}\right)^2 + \left(\frac{\mathbf{n}_{\text{b}}[a]}{\mathbf{n}_{\text{c}}[c]}\right)^2.$$

When all three axes are split (e.g., the b, c, and d axes in the a projected volume), the distance is the angle between the configuration and the corner of the hyperplane bounded volume defined by the 3 axes. If the split axis values are the normals $\mathbf{n}_\mathrm{b}$, $\mathbf{n}_\mathrm{c}$, and $\mathbf{n}_\mathrm{d}$ (in the projected a volume), the partial distance is:

$$\texttt{PARTIAL\_DIST}_{P^3|\mathbf{n}_\mathrm{b},\mathbf{n}_\mathrm{c},\mathbf{n}_\mathrm{d}} = \cos^{-1} |\mathbf{q} \cdot \mathbf{q}_\mathrm{corner}|$$

$$\text{where:} \qquad \mathbf{q}_\mathrm{corner} = \texttt{normalize}\left(1, -\frac{\mathbf{n}_\mathrm{b}[a]}{\mathbf{n}_\mathrm{b}[b]}, -\frac{\mathbf{n}_\mathrm{c}[a]}{\mathbf{n}_\mathrm{c}[c]}, -\frac{\mathbf{n}_\mathrm{d}[a]}{\mathbf{n}_\mathrm{d}[d]}\right)$$

Each of these `PARTIAL_DIST` functions for P^3 successively provide a tighter bound, and thus prunes recursion better.

Each query in the $SO(3)$ subspace must search up to 4 kd-trees of the projected volumes on the 3-sphere. The projected volume in which the query configuration lies we call the *primary* volume, and the remaining 3 volumes are the *secondary* volumes. The search begins by finding the nearest configuration in the kd-tree in the primary volume. The search continues in each of the remaining secondary volumes only if it is possible for a point within its associated volume to be closer than the nearest point found so far. For this check, the partial distance is computed between the query configuration and the two hyperplanes that separate the primary and each of the secondary volumes. There are two hyperplanes due to the curved nature of the manifold and the double-coverage property of quaternions. Since a closer point could lie near either boundary between the volumes, we must compare to the minimum of the two partial distances, thus:

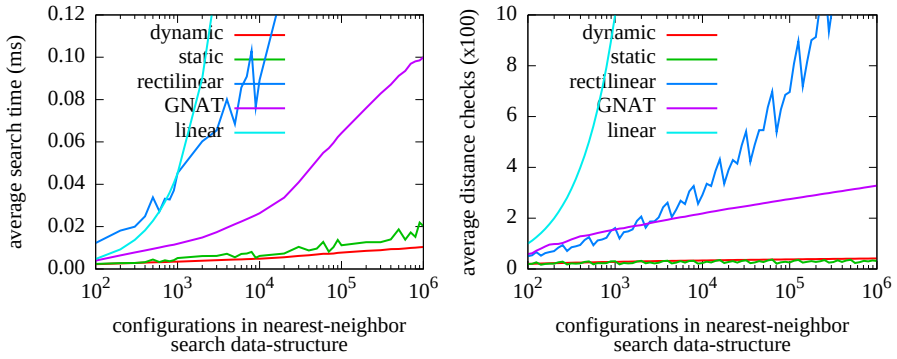
$$\min\left(\texttt{PARTIAL\_DIST}_{P^3|\mathbf{n}_\mathrm{ab}}(\mathbf{q}), \quad \texttt{PARTIAL\_DIST}_{P^3|\mathbf{n}_\mathrm{ba}}(\mathbf{q})\right)$$

where $\mathbf{n}_\mathrm{ab}$ and $\mathbf{n}_\mathrm{ba}$ are the normals of the two hyperplanes separating the volumes a and b.

4.5 Nearest, k-Nearest, and Nearest in Radius r Searches

Alg. 3 implements the nearest neighbor search. We extend it to k-nearest neighbor search by replacing $\mathbf{q}_\mathrm{nearest}$ with a priority queue. The priority queue contains up to k configurations and is ordered based upon distance from $\mathbf{q}$, with the top being the farthest of the contained configurations from $\mathbf{q}$. The queue starts empty, and until the queue contains k configurations, the algorithm adds all visited configurations to the queue. From then on, $\texttt{DIST}(\mathbf{q}, \mathbf{q}_\mathrm{nearest})$ (lines 3 and 14) is the distance between $\mathbf{q}$ and the top of the priority queue. When the search finds a configuration closer than the top of the queue, it removes the top and adds the closer configuration to the queue (line 4). Thus the priority queue always contains the k nearest configurations visited.

To search for nearest neighbors in radius r, $\mathbf{q}_\mathrm{nearest}$ in Alg. 3 is a result set. Distance comparisons on lines 3 and 14 treat $\texttt{DIST}(\mathbf{q}, \mathbf{q}_\mathrm{nearest}) = r$. When the algorithm finds a configuration closer than r, it adds it to the result set in line 4.



(a) nearest neighbor search time

(b) distance checks

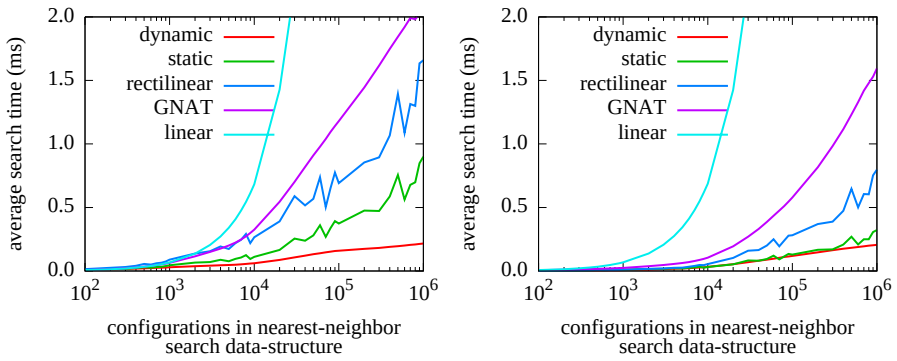
Fig. 3. Comparison of nearest neighbor search time and distance checks plotted with increasing configuration count in the searched dataset. In (a) we plot the average time to compute a single nearest neighbor for a random point. In (b) we track the average number of distance computations performed by a search.

5 Results

We evaluate our method for nearest neighbor searches in four scenarios: (1) uniform random rotations in $SO(3)$, (2) uniform random rotations and translations in $SE(3)$, (3) configurations generated by RRT [24] solving the "Twistycool" motion planning scenario in OMPL [3], and (4) configurations generated by RRT* [2] solving the "Home" motion planning scenario in OMPL [3]. We compare four methods for nearest neighbor searching: (1) "dynamic" is a dynamic kd-tree using our method and midpoint splits, (2) "static" is a static-to-dynamic conversion [9] of a median-split kd-tree using our method, (3) "rectilinear" is a static-to-dynamic conversion of a median-split kd-tree using rectangular splits [10] on $SO(3)$, and (4) "GNAT" is a Geometric Near-neighbor Access Tree [12]. All runs are computed on a computer with two Intel X5670 2.93 GHz 6-core Westmere processors, though multi-core capabilities are not used.

5.1 Random $SO(3)$ Scenario

In the Random $SO(3)$ scenario, we generated uniformly distributed random configurations in $SO(3)$ and compute nearest neighbors for random configurations. We compute the average search time and the average number of distance computations performed to search a nearest neighbor data structure of size n. We vary n from 100 to 1 000 000 configurations, and plot the result in Fig. 3. The average nearest neighbor search time in Fig. 3(a) shows an order of magnitude performance benefit when using our method. The number of distance computations in Fig. 3(b) is a rough metric for how much of the data structure each method is able to prune from the search. The performance gain in Fig. 3(b) gives insight into the reasons for the performance gains shown in Fig. 3(a).



(a) search times ($\alpha = 1$) (b) search times ($\alpha = 10$)

Fig. 4. Comparison of nearest neighbor search time for random configurations in $SE(3)$. In (a) and (b) the translation space is bounded to a unit cube, and the translation distance is weighted 1 and 10 respectively. In (a) the SO(3) component of a configuration is given more weigh, and thus has more impact on each search.

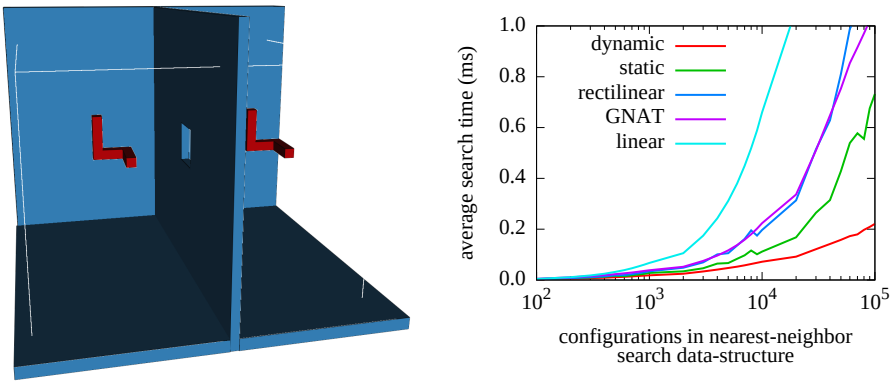
5.2 Random $SE(3)$ Scenario

In this scenario, we build nearest neighbor search structures with random configurations generated in $SE(3)$. Using $\texttt{DIST}_{\mathbb{R}^m P^3}$, we evaluate performance for $\alpha = 1$ and 10 in Fig. 4. For small α, the $SO(3)$ component of a configuration is given more weight, and thus provides for greater differentiation of our method. In Fig. 4 (a), we observe a 2 to 5× improvement in performance between our method and the rectilinear method, and an order of magnitude performance improvement over GNAT. As α increases, more weight is given to the translation component, so our $SO(3)$ splits have less impact on performance. Hence, our improvement drops, but is still 2 to 3× faster than rectilinear, and 8× faster than GNAT.

5.3 RRT on the Twistycool Scenario

We evaluate the impact of our method in the "Twistycool" motion planning scenario, using OMPL for both the scenario and the RRT planner. The Twistycool puzzle, shown in Fig. 5(a), is a motion planning problem in which a rigid-body object (the robot) must move through a narrow passage in a wall that separates the start and goal configurations. At each iteration, the RRT motion planner computes a nearest neighbor for a random sample against all samples it has already added to its motion planning tree. We have adjusted the relative weighting α for translation and rotation from its default, such that each component has approximately the same impact on the weighted distance metric.

As we see in Fig. 5(b), the performance of our method with the dynamic kd-tree is more than 5× faster than GNAT and rectilinear split kd-trees. This matches our expectations formed by the uniform random scenario results, and



(a) Twistycool scenario (b) RRT nearest neighbor search times

Fig. 5. Twistycool scenario and RRT nearest neighbor search times. The scenario in (a) requires the red robot to move from its starting configuration on the left, through a narrow passage in the wall, to its goal configuration on the right. The average time per nearest neighbor search is plotted in (b).

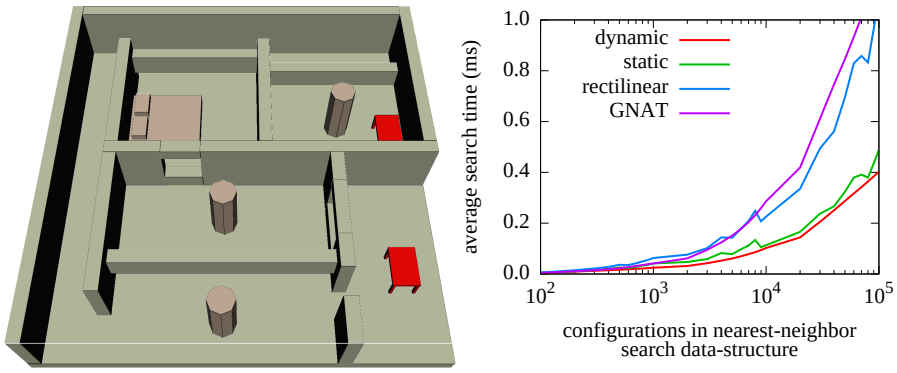
shows little degradation with the non-uniform dataset created by this motion planning problem.

5.4 RRT* on the Home Scenario

We ran the "Home" scenario using the RRT* motion planner included in OMPL. As shown in Fig. 6(a), the motion planner computes a plan that moves a table from one room to another while avoiding obstacles. The RRT* planner incrementally expands a motion planning tree, while "rewiring" it towards optimality as it goes. In each iteration RRT* finds an extension point using a nearest neighbor search, and then rewires a small neighborhood after a k-nearest neighbor search. Unlike RRT, we can allow RRT* to continue for as many iterations as desired, and get incrementally better results. As with the RRT scenario, we proportionally scale α so that the $SO(3)$ and translation components have approximately equivalent impact on the distance metric. As shown in Fig. 6 (b), our method in both variants outperforms GNAT and rectilinear splits by roughly a factor of 3. In these results we observe also that the median split of "static" and the midpoint split of "dynamic" perform equally well, and the main differentiating factor between the kd-tree methods is thus the $SO(3)$ partitioning.

6 Conclusion

We presented a method for efficient nearest neighbor searching in $SO(3)$ space and by extension $SE(3)$, using a kd-tree with a novel approach to creating hyperplanes that divide rotational space. Our partitioning approach provides two key benefits: (1) it reduces the dimensionality of the rotation representation from 4-dimensional quaternion vector to match its 3 degrees of freedom, and (2)



(a) Home scenario (b) RRT* nearest neighbor search times

Fig. 6. Home scenario and RRT* nearest neighbor search times. In the scenario in (a), the motion planner must find a path that moves the red table "robot" from its starting configuration in the lower right room to the goal configuration in the upper right. The average time for nearest neighbor search is plotted in (b).

creates an efficient partitioning of the curved manifold of the rotational group. We integrated our approach into RRT and RRT* and demonstrated the fast performance and efficient scaling of our nearest neighbor search as the tree size increases.

In future and ongoing work, we view our approach as something that should augment or work well in tandem with existing nearest neighbor search algorithms and implementations. We are looking to adapt our approach to include the approximate nearest neighbor kd-trees of the Fast Library for Approximate Nearest Neighbors (FLANN) [19] and contribute an implementation to OMPL.

Acknowledgments. This research was supported in part by the National Science Foundation (NSF) through awards IIS-1117127 and IIS-1149965.

References

1. Choset, H., Lynch, K.M., Hutchinson, S.A., Kantor, G.A., Burgard, W., Kavraki, L.E., Thrun, S.: Principles of Robot Motion: Theory, Algorithms, and Implementations. MIT Press (2005)
2. Karaman, S., Frazzoli, E.: Sampling-based algorithms for optimal motion planning. Int. J. Robotics Research **30**(7) (June 2011) 846–894
3. Şucan, I.A., Moll, M., Kavraki, L.E.: The Open Motion Planning Library. IEEE Robotics and Automation Magazine **19**(4) (December 2012) 72–82
4. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9) (September 1975) 509–517
5. Friedman, J.H., Bentley, J.L., Finkel, R.A.: An algorithm for finding best matches in logarithmic expected time. ACM Transactions on Mathematical Software (TOMS) **3**(3) (1977) 209–226

6. Sproull, R.F.: Refinements to nearest-neighbor searching in k-dimensional trees. Algorithmica **6**(1-6) (1991) 579–589
7. Finkel, R.A., Bentley, J.L.: Quad trees a data structure for retrieval on composite keys. Acta informatica **4**(1) (1974) 1–9
8. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proc. ACM-SIAM Symp. Discrete Algorithms. (1993)
9. Bentley, J.L., Saxe, J.B.: Decomposable searching problems I. static-to-dynamic transformation. J. Algorithms **1**(4) (1980) 301–358
10. Yershova, A., LaValle, S.M.: Improving motion-planning algorithms by efficient nearest-neighbor searching. IEEE Trans. Robotics **23**(1) (2007) 151–157
11. Shoemake, K.: Animating rotation with quaternion curves. Proc. ACM SIGGRAPH **19**(3) (1985) 245–254
12. Brin, S.: Near neighbor search in large metric spaces. In: Proc. Int. Conf. Very Large Databases. (1995)
13. Beygelzimer, A., Kakade, S., Langford, J.: Cover trees for nearest neighbor. In: Proc. Int. Conf. Machine Learning, ACM (2006) 97–104
14. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: Proc. Int. Conf. Very Large Databases. (1997) 426
15. Indyk, P.: Nearest neighbors in high-dimensional spaces. In: Handbook of Discrete and Computational Geometry. 2 edn. Chapmann & Hall/CRC (2004)
16. Plaku, E., Kavraki, L.E.: Quantitative analysis of nearest-neighbors search in high-dimensional sampling-based motion planning. In: Algorithmic Foundation of Robotics VII. Springer (2008) 3–18
17. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. J. ACM **45**(6) (1998) 891–923
18. Kushilevitz, E., Ostrovsky, R., Rabani, Y.: Efficient search for approximate nearest neighbor in high dimensional spaces. SIAM J. Computing **30**(2) (2000) 457–474
19. Muja, M., Lowe, D.G.: Fast approximate nearest neighbors with automatic algorithm configuration. In: Int. Conf. Computer Vision Theory and Application (VISSAPP), INSTICC Press (2009) 331–340
20. Mount, D.M.: ANN programming manual. Technical report, Dept. of Computer Science, U. of Maryland (1998)
21. Yershova, A., LaValle, S.M.: Deterministic sampling methods for spheres and SO(3). In: Proc. IEEE Int. Conf. Robotics and Automation. (2004) 3974–3980
22. Nowakiewicz, M.: Mst-based method for 6dof rigid body motion planning in narrow passages. In: Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS), IEEE (2010) 5380–5385
23. Knuth, D.E.: The art of computer programming, volume 3: (2nd ed.) sorting and searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1998)
24. LaValle, S.M., Kuffner, J.J.: Rapidly-exploring random trees: Progress and prospects. In Donald, B.R., Others, eds.: Algorithmic and Computational Robotics: New Directions. AK Peters, Natick, MA (2001) 293–308