Robotics: Science and Systems 2024
Delft, Netherlands, July 15-July 19, 2024

Collision-Affording Point Trees: SIMD-Amenable Nearest Neighbors for Fast Collision Checking

Clayton W. Ramsey, Zachary Kingston*, Wil Thomason*, and Lydia E. Kavraki
Rice University
{clayton.w.ramsey,zak,wbthomason,kavraki}@rice.edu
*Equal Contribution.

***Abstract*—Motion planning against sensor data is often a critical bottleneck in real-time robot control. For sampling-based motion planners, which are effective for high-dimensional systems such as manipulators, the most time-intensive component is collision checking. We present a novel spatial data structure, the collision-affording point tree (CAPT): an exact representation of point clouds that accelerates collision-checking queries between robots and point clouds by an order of magnitude, with an average query time of less than 10 nanoseconds on 3D scenes comprising thousands of points. With the CAPT, sampling-based planners can generate valid, high-quality paths in under a millisecond, with total end-to-end computation time faster than 60 FPS, on a single thread of a consumer-grade CPU. We also present a point cloud filtering algorithm, based on space-filling curves, which reduces the number of points in a point cloud while preserving structure. Our approach enables robots to plan at real-time speeds in sensed environments, opening up potential uses of planning for high-dimensional systems in dynamic, changing, and unmodeled environments.**

I. Introduction

Motion planning underpins many applications of high-degree-of-freedom robots, allowing them to efficiently find collision-free trajectories between arbitrary poses. Modern motion planning methods capably solve problems with many obstacles for these high-dimensional robots, typically by either building and searching a graph or tree approximating the collision-free subset of the robot's state space (*i.e.*, sampling-based motion planning (SBMP) [1–3]) or by solving a numerical optimization problem (*i.e.*, trajectory optimization [4–6]). The most time-consuming component of most motion planners—and SBMPs in particular—is *state validation*, which ensures that a robot's state does not violate its constraints or collide with obstacles [7, 8]. State validators commonly assume knowledge of the precise geometries and positions of all obstacles in the environment—an assumption that does not hold for general real-world settings where only sensed representations of the world may be available. Earlier work that checks for collisions between robot geometries and point clouds [9–14] attempts to lift this assumption, but point cloud collision checking remains a relatively slow bottleneck for motion planning.

Recent work [15–18] has produced new approaches to hardware-accelerated motion planning that exploit parallelism in collision checking and other core planning operations to find complete trajectories in microseconds to milliseconds. This use of parallelism motivates the need for higher-throughput parallelism-friendly algorithms and data structures for efficiently planning collision-free motions in environments that are only perceived as point clouds, *e.g.*, from a common depth camera like the Intel RealSense. Sundaralingam et al. [16] use GPU parallelism to allow batched querying of an approximate Euclidean Signed Distance Field (ESDF) built from a series of sensor measurements [19]; similarly, Vasilopoulos et al. [17] perform a GPU-parallel brute-force SDF computation between a discretized set of points on the robot geometry and a dense point cloud. Although these GPU-based methods are promising for some applications, data synchronization costs between the GPU and CPU limit the direct applicability of these techniques for motion planning algorithms, many of which are CPU-based. Further, for applications of field robotics such as planetary rovers, agricultural robots, and others, the power requirements of an onboard GPU may be untenable.

In this work, we propose a data structure and associated construction and search algorithms for *exact* point cloud distance computation and collision checking. Our proposed data structure, the *collision-affording point tree* (CAPT), adapts and refines concepts from the classical k-d tree to support efficient parallel evaluation. The core insights guiding our design of the CAPT are that (1) exploiting the spatial correlation present in SBMP edge validation collision queries allows for aggressive early-termination of batched queries without sacrificing correctness, and that (2) many motion planning problems (*e.g.*, in manipulation) only need to represent a relatively small, *local* part of the environment for collision checking. This first insight extends ideas from Thomason et al. [15]; the second enables us to rethink traditional assumptions about minimizing memory overhead in point cloud representations: we in fact duplicate select subsets of points to create a parallelism-friendly data model. By combining these insights with a machine-sympathetic data structure and the use of data-level parallelism, CAPTs can return collision results against an observed 3D point cloud in a mean time of **under ten nanoseconds per query** on a single core of a consumer desktop CPU[1]. Although we

This work was supported by NSF CCF 2336612, NSF ITR 2127309 for the CRA CIFellows Project, and Rice University Funds.

[1]The point clouds used to generate these collision-checking throughput results contained up to 50000 points and had mean dispersions between 7mm and 2.2cm. For detailed analysis, refer to the results in Sec. VI-A1 and Appendix C; for examples of such point clouds, see Fig. 1c and Fig. 3b.

focus on CPU-based *single-instruction, multiple-data* (SIMD) parallelism in this paper, our approach also applies to and may benefit GPU-based planners using a *single-instruction, multiple-thread* (SIMT) parallelism model.

Concretely, we contribute:

1) the collision-affording point tree (CAPT), a novel data structure for storing sensed point clouds for collision checking.
2) efficient construction, branch-free parallel query, and collision check algorithms for CAPTs.
3) a method for efficient point cloud down-sampling by exploiting properties of space-filling curves.
4) proofs of correctness (*i.e.*, that using CAPTs and our filtering algorithm does not modify planning problem feasibility).
5) empirical evaluations for collision throughput and error against a set of competitive baselines.
6) integration with a vectorized motion planner [15], demonstrating the utility of CAPTs for high-performance SBMP on a number of difficult and cluttered problems.
7) a proof-of-concept demonstration with a depth camera and physical robot hardware.
8) an open source implementation of CAPTs[2].

II. Related Work

Despite the inherent computational complexity of motion planning, which is known to be PSPACE-complete [20, 21], SBMPs [2, 3] are nonetheless capable of solving many motion planning problems in tens to hundreds of milliseconds, given explicit representations of the problem environment. Thomason et al. [15] proposed an approach to accelerating SBMPs via SIMD parallelism, decreasing planning times to the order of microseconds—sufficient for real time operation. Much of this speedup is due to the use of fine-grained parallelism in collision checking; however, this planner still requires an explicit, known model of the environment's geometry. Particularly, in real-world planning problems, explicit geometry models are often unavailable—thus, planning from sensor data has been desirable since the beginning of the field [22].

A. Planning from sensor data

Planning-amenable representations of sensor data usually take the form of some space-partitioning data structure. k-d trees [23] are particularly relevant data structures for point cloud representation. They are often used for nearest-neighbor search in low-dimensional spaces [24–26], and allow for logarithmic-time collision-checking against points in a point cloud [9]. These trees can also be augmented with a sphere covering [27] for efficient rejection of non-colliding geometry. Many efforts have been made to accelerate k-d tree queries by low-level optimization, including through parallel subtree search [28] and dedicated instruction-set architectures [29]. FLANN [30, 31] and Nigh [32] are popular implementations of k-d trees; the former supports approximate nearest-neighbor search, while the latter is exact.

The most popular approaches to representing sensor data for SBMP collision checking often use *occupancy maps* [33–35]. OctoMaps [10] implement a probabilistic occupancy map using octrees [36] of voxels, where each voxel is considered "occupied" based on Bayesian updates computed upon each new point cloud inserted into the map. They allow for collision-checking based on bounding-volume hierarchies, which use a branch-and-prune search to limit the set of possible points for collision checking. OctoMaps are also integrated with collision checking libraries, *e.g.*, the Flexible Collision Library [11], and popular planning frameworks such as MoveIt [37]. They are a commonly used representation in practice, *e.g.*, for underwater vehicles [38], subterranean exploration [39], autonomous vehicles [40], and aerial vehicles [41].

Voxel-based approaches have also seen significant use recently due to many optimization-based planners [5, 17] using signed distance fields for collision-checking—the representation of the signed-distance field is backed by a voxel representation [42, 43]. CuRobo [16] is an optimization-based planner which uses GPU parallelism for high-efficiency collision-checking, including against point clouds. It relies on the NVBlox [19] GPU-accelerated signed-distance field library for its collision-checking and optimization. VoxBlox [44] and VoxGraph [45] are also CPU-based sensor-based mapping tools for constructing signed-distance fields for collision-checking.

Recently, there has also been interest in using implicit representations of the environment, such as neural radiance fields (NeRFs) [46], which have been used for motion planning [47, 48]. However, constructing NeRFs, while relatively fast [49], still take on the order of seconds to construct with GPU hardware, making them infeasible for online planning.

B. Space-filling curves

Space-filling curves are continuous real bijections that map every point of a one-dimensional line to a higher-dimensional space, such as $\mathbb{R}^3$. Z-order curves [50], also known as Morton curves, are a class of space-filling curve often used for nearest-neighbor applications. A point in a high-dimensional space can be projected onto a Z-order curve by interleaving the bits of the binary representation of its coordinates. If two points' projections onto the curve are close together, they are likely to also be near in the higher-dimensional pre-image space. Ying et al. [51] used a Z-order curve to produce a low-discrepancy sub-sampling of a point cloud; however, their sub-sampling procedure does not guarantee that the overall point cloud structure is preserved. Likewise, Connor and Kumar [52] used a Z-order sorting to accelerate construction of k-nearest-neighbor graphs by limiting their search to a single range of the space-filling curve.

C. SIMD parallelism and planning

Parallel motion planning algorithms [7, 15, 16, 18, 53–57] require parallelizable collision-checking data structures. Further, to reap the benefits of early-termination in collision checking

[2]Available at https://github.com/kavrakilab/vamp.

(a) (b) (c)

Figure 1: 1a, 1b: A cluttered tabletop scene, captured as RGB-D with an Intel Realsense D455 sensor. 1c: The point cloud rendering of the same scene, filtered using our proposed space-filling curve method (Sec. IV-C).

that enable SIMD-accelerated planning to plan at the microsecond scale, a SIMD-amenable data structure is required. However, the major data structures used in planning for sensor data representation are not amenable to this flavor of parallelism—hierarchical space-subdividing data structures (*e.g.*, OctoMaps, k-d trees, other voxel grids) require conditional-branch-heavy searches through large subtrees, which result in a highly sub-optimal memory access pattern: collision checks in these data structures must access potentially many fragmented segments of memory. Additionally, the branching nature of these searches makes them impractical to adapt to the branchless computation framework that SIMD parallelism best matches. To overcome these issues, we present a new data structure, designed with branchless, cache-friendly access in mind and demonstrate its effectiveness for efficient collision checking.

III. Preliminaries

A k-d tree [58] is a class of space partitioning tree which represents and organizes a set of n points, $PC \subseteq \mathbb{R}^k$. Each leaf of the tree corresponds to a *cell*, which is an axis aligned bounding box in $\mathbb{R}^k$ containing exactly one *representative point*, $p_r \in PC$. The union of a k-d tree's cells spans the entirety of $\mathbb{R}^k$ [3] and the union of all representative points is equal to PC. Each branch of the tree partitions $\mathbb{R}^k$ about an axis-aligned hyperplane: a branch at depth d in the tree with test value t partitions the space such that, for any point $p \in PC$, if $p[d \mod k] \leq t$ (where $p[i]$ is the i-dimension component of p), it belongs to the left sub-tree of the branch; otherwise, it belongs to the right sub-tree.

We can construct a k-d tree from a point cloud PC in $O(kn \log n)$ time using a recursive partitioning algorithm. At each level, we split PC into two equally-sized subsets B_1, B_2 by a hyperplane intersecting the median point in PC along axis d. This splitting continues recursively for B_1 and B_2 until the point cloud contains exactly one point. Each splitting hyperplane forms one branch in the tree, and the walls of each leaf's cell are the splitting hyperplanes of its parent branches for each dimension.

k-d trees are most commonly used for nearest-neighbor search. Given some query point x, a recursive branch-and-bound search through the tree can find the closest element of PC to x in $O(\log n)$ operations [58].

We can use the nearest-neighbor facility of a k-d tree to check for collisions between a sphere and a point cloud. Given a sphere centered at x with radius r, the aforementioned branch-and-bound search procedure can find the nearest point $p \in PC$ to x. Then, if $\|x-p\| \leq r$, the sphere is in collision; otherwise, there must be no point in PC whose distance to x is less than r, and x therefore does not collide with PC.

IV. Method

CAPTs redesign aspects of the classic k-d tree to make it amenable to high-throughput parallel querying for robot collision checking. The major problems with directly using a k-d tree for this purpose are (1) cache-unfriendly random memory access patterns that arise from the tree's storage representation, and (2) the inherently conditional-branch-heavy backtracking recursive algorithm used for normal k-d tree nearest-neighbor queries. Accordingly, the defining features of a collision-affording point tree, as opposed to a k-d tree, are its use of a memory layout that improves cache coherency during tree traversal, and that each leaf of a collision-affording point tree contains an *affordance set*, a conservative approximation of the possible nearest-neighbors to any point in the cell. This deceptively simple change allows the implementation of a search to avoid the backtracking stage of a search through a k-d tree, enabling branch-free, parallelism-friendly exact collision checking in a fraction of the time.

To use a CAPT to check for robot collision, we first assume that the robot is made up of some set of spheres S (as in other motion planning work, *e.g.*, [15, 16, 59]). Non-spherical robots can be approximated conservatively by constructing a spherical bounding volume hierarchy [60] which contains all of their collision geometry. Let the smallest sphere of the robot's geometry have some known radius r_{min} and the largest sphere of the robot's geometry have some known radius r_{max}. Then, given a query sphere with center x and radius $r : r_{\text{min}} \leq r \leq r_{\text{max}}$, we can use the CAPT to check if the sphere is in collision. First, we search through the tree to find the leaf cell of the tree which contains x. If any point in the leaf's affordance set collides with the sphere, then the sphere is in collision; otherwise, the query sphere is not in collision.

[3] Slightly abusing the notion of an AABB, the cells on the boundary of the k-d tree are half-open.

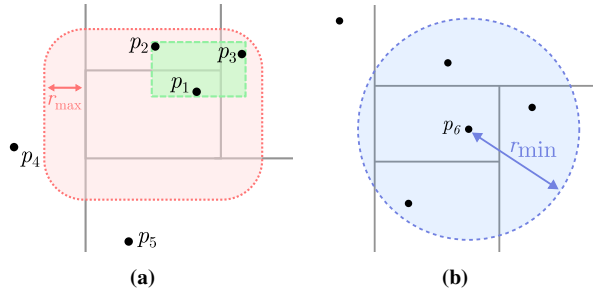


Figure 2: 2a: The cell containing p_1 affords p_2 and p_3 at radius $r_{\max}$, but not p_4 or p_5. The axis-aligned bounding box containing all afforded points is depicted in green. 2b: The sphere centered at p_6 of radius $r_{\min}$ contains the entire cell, so no other points need to be included in the cell's affordance set.

We begin by explaining the collision-affording point tree and its construction process in Secs. IV-A and IV-B. Next, we detail a point cloud filtering algorithm based upon space-filling curves in Sec. IV-C. Lastly, we describe our branch-free parallel collision-checking algorithm for CAPTs in Sec. IV-D.

A. The collision-affording point tree

In a CAPT, each leaf of the tree contains the affordance set for the leaf's corresponding cell, illustrated in Fig. 2. Given the cell c corresponding to a leaf, that leaf's affordance set is the set of all points in PC such that c *affords* collision with l at the radius $r_{\max}$. A cell c affords a point p at radius r if there exists some point $q \in c$ such that $\|p - q\| \leq r$, as depicted in Fig. 2a. Intuitively, a point p is afforded by a cell c if a sphere of radius r whose center is contained by c could collide with p. Finally, each leaf is associated with a *second* axis-aligned bounding box. This bounding box is *not* the same as the cell; instead, it is the minimal bounding box containing all points in the leaf's affordance set, as shown in green in Fig. 2a.

For simplicity, we also assume that PC contains n points, such that n is a power of two. If n is not a power of two, we pad PC with points at ∞ to the next greatest power of two.

A CAPT is then the tuple (T, A, P), such that the test sequence T is an array of $n-1$ test values in $\mathbb{R} \cup \{\infty\}$, A is an array of n axis-aligned bounding boxes over $\mathbb{R}^k$, and the affordance table P is a ragged two-dimensional array of n different affordance sets. This representation is *implicit*: the tree does not store any information about its branches other than in the test sequence T.

We arrange T according to an Eytzinger layout, a class of array-backed implicit tree layout originally used for heaps [61]. This layout reduces memory fragmentation by storing all data in a single contiguous block, and also improves performance by allowing for branch-free traversal. In such a layout, T_0 corresponds to the root branch of the tree: all points whose x-value is less than T_0 belong to the left sub-tree, while all others belong to the right sub-tree. Next, T_1 and T_2 correspond to the first branches in the left and right sub-trees about the y-value of each point. Recursively, if T_i corresponds to a partition of the tree about the dimension d, then T_{2i+1} corresponds to the next branch in the left sub-tree, while T_{2i+2} corresponds to the next branch in the right sub-tree, both of which split on dimension $d+1 \mod k$. The value of T_i at depth d is the median value of $p[d \mod k]$ across all representative points p in its sub-tree. Intuitively, each T_i partitions the space by a new axis-aligned hyperplane, splitting the points in its subtree in half. For simplicity, we choose to partition on a repeating sequence of axes every time (first along the x-axis, then the y-axis, and so on), but could substitute other methods, such as randomly selecting an axis.

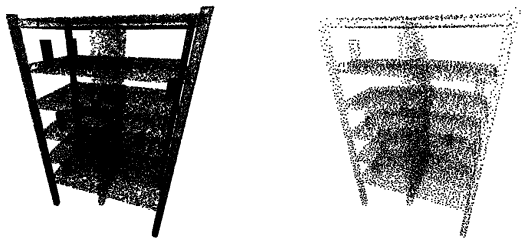
Algorithm 1: Construct

Input: Point cloud PC containing n points of dimension k, minimum query radius $r_{\min}$, maximum query radius $r_{\max}$, axis-aligned bounding box c, affordance set z, uninitialized CAPT (T, A, P), index i, dimension index d

```
if |PC| = 1 then
    x ← only element of PC;
    if ∃q ∈ c : ‖q − x‖ > r_min then
        PC ← PC ∪ z;
    a ← bounding box containing all points in PC;
    append PC onto P;
    append a onto A;
else
    T_i ← median value of p_d for all p ∈ PC;
    B_1 ← {p ∈ PC : p_d ≤ T_i};
    B_2 ← {p ∈ PC : p_d > T_i};
    c_1, c_2 ← c;
    shrink the upper bound of c_1 on dimension d to T_i;
    raise the lower bound of c_2 on dimension d to T_i;
    z_1 ← {p ∈ z ∪ B_2 : c_1 affords p at r_max};
    z_2 ← {p ∈ z ∪ B_1 : c_2 affords p at r_max};
    Construct(B_1, r_min, r_max, c_1, z_1, (T, A, P), 2i + 1, (d + 1) mod k);
    Construct(B_2, r_min, r_max, c_2, z_2, (T, A, P), 2i + 2, (d + 1) mod k);
```

B. Collision-affording point tree construction

To construct a CAPT, we apply the same recursive partitioning approach as in k-d tree construction, using quick-select [62] for an expected linear-time selection of the median value for each partition. The exact construction algorithm is specified in Alg. 1. At each step of the construction procedure, we retain two additional sets of information: the current cell c and the current afforded set z. c is initialized to the cell containing all of $\mathbb{R}^k$, while z is initialized to the empty set. Every time we split the point cloud along a median plane, we split c about the same median plane into two adjacent cells, c_1 and c_2. Next, we duplicate z to produce two new affordance sets, z_1 and z_2. We expand z_1 to include all points in the cloud contained by c_2, and vice versa for z_2. Finally, we filter out all points from z_1 which are not afforded by c_1, and likewise with z_2. This process can be thought of as maintaining the set of all points outside of a cell c that are still close enough to c that they may collide with a query sphere centered in c. Once each cell



(a) (b)

Figure 3: 3a: A point cloud with 126000 points, created by randomly sampling the surface of a shelf. 3b: The same point cloud, filtered to 8889 points using $r_{\text{filter}} = 2\text{cm}$.

contains exactly one point, we determine the final affordance set of that cell's leaf as the union of the set containing the representative point and all points afforded by the leaf cell.

We can use knowledge of r_{min} to prune the affordance set slightly more than the conservative approximation created by the construction procedure. If all points $x_c \in c$ are so close to the representative point p that $\|x_c - p\| \leq r_{\text{min}}$, then all spheres with center $x \in c$ and radius $r \geq r_{\text{min}}$ will collide with c's representative point p, as shown in Fig. 2b. If this is the case, then there is no need to include any points outside of c in the affordance set, since we know that any query sphere centered in c is already in collision. Therefore, for these sufficiently small cells, we can simply store $\{p\}$ as the affordance set, without including any other points.

C. Filtering

Not all points in an input point cloud are required for collision-checking, especially in extremely dense clouds, as there are many redundant points when conservatively approximating the colliding volume of the cloud. As the construction time of collision-affording point trees grows with the size of the point cloud, we introduce an efficient filtering procedure to significantly reduce the density of the cloud that also guarantees that it will not remove critical points.

As the cloud is used for collision checking, it cannot remove a point p unless there is another point p^* in the cloud sufficiently close to p, such that $\|p - p^*\|$ is less than some threshold radius r_{filter}. This guarantees that the robot cannot penetrate further than r_{filter} into the point cloud; alternately, all spheres of the robot may be padded by r_{filter} to produce a conservative approximation of the cloud. Lastly, this filtering procedure must be computationally cheap: it must be significantly faster than the construction time of the CAPT to achieve any useful speedup.

In order to verify that any removed point p has a sufficiently close neighbor p^*, we must compute the distance between p and p^*. A naive algorithm would compare every two points in PC, but this approach yields $O(n^2)$ runtime, which is unacceptable for large point clouds. Instead, we reduce the candidate set of pairs by exclusively comparing points which are relatively near to one another, according to an arbitrary measure of locality.

Space-filling curves (specifically in our implementation, Z-order curves [50]) provide such a measure of locality: we use one to map all points in PC to a one-dimensional space-filling curve. Once sorted in order of their position along the curve, points which are adjacent in the curve are also likely to be adjacent in their higher-dimensional space. Therefore, by exclusively checking the distance between neighboring points in the space-filling curve, we can dramatically reduce the point cloud size in $O(n \log n)$ time (Alg. 2).

However, nearby points in a higher-dimensional space are not guaranteed to be adjacent in a fixed space-filling curve. We mitigate this by checking for neighbors on multiple space-filling curves, one for each permutation of dimensions. If two points are near to each other, then it is likely that at least one such curve will place them adjacent to one another. Repeating the filtering process on each permutation of dimensions means that the filtering procedure scales with $O(k!n \log n)$, but for $k = 3$, there are only six such permutations, so the cost of extra filter checks is minimal compared to the savings in point cloud size.

Algorithm 2: Filter

Input: list of points PC, filter radius r_{filter}
Output: list of filtered points $PC' \subseteq PC$

```
PC' ← PC;
foreach permutation X of dimensions do
    sort PC' along a Z-order curve by dimension order X;
    i ← 0;
    foreach j ← 1, 2, ··· |PC'| − 1 do
        if ‖PC'_i − PC'_j‖ > r_filter then
            i ← i + 1;
            PC'_i ← PC'_j;
    PC' ← PC'_{0···i};
return PC';
```

Additionally, we filter out any points which we can prove will never be in collision with the robot. This process is simple for fixed-base arm robots: a point can only collide with a robot if its distance to the base link of the robot is less than the maximum extension length of the arm.

All together, this filter can achieve dramatic reductions in point cloud size even for small values of r_{filter}, filtering clouds with over a hundred thousand points to less than ten thousand in a few milliseconds. As shown in Fig. 3, relatively conservative values of r_{filter} cull the point cloud by a dramatic amount; additionally, the filtering process significantly reduces point cloud density, reducing the expected colliding-set size and therefore improving tree construction and query times.

D. Collision querying

When collision-checking for a robot, a robot's configuration is valid only if all of the robot's physical geometry is not in collision with the environment. If any sphere of the robot's geometry is in collision with the environment, then the entire configuration is invalid. This provides us with an early-termination condition: we need only find a single colliding

sphere to invalidate an entire configuration. By parallelizing collision checks across multiple query spheres, the entire search can terminate as soon as one collision is found.

The first step of a collision check is a search through the tree, as outlined in lines 1-7 of Alg. 3. Given some query sphere with center x and radius r, the search begins with a test index $i = 0$ and dimension $d = 0$. Then, at each step of the search, if $x_d < T_i$, i is updated to $2i+1$, or $2i+2$ otherwise, while d is updated to $d+1 \mod k$. This is the same update rule for the Eytzinger layout as used during construction: index $2i+1$ corresponds to the left sub-tree, while index $2i+2$ corresponds to the right subtree. When the search completes, the final value of i is an integer in the range $[n-1, 2n-1)$, with each value corresponding to a unique leaf of the tree. $i-n+1$ is an integer in the range $[0, n)$ corresponding to each point stored at a leaf of the tree. This traversal can be performed branchlessly by converting the boolean value $x_d \geq T_i$ comparison into an integer l, assigning $i \leftarrow 2i+1+l$. Since n is a power of two, all traversals of the tree terminate in exactly the same number of iterations, and we do not need to test for reaching the end of the tree. Likewise, the memory access pattern of the traversal is extremely predictable: all accesses at a given iteration are restricted to a small set of possible values. The branchless traversal sequence allows for efficient SIMD parallelism: each lane of a single register contains a different test index, and each load, comparison, and index update is parallel, providing a large improvement in performance.

After determining which cell contains x, the search algorithm first performs an efficient collision check between the query sphere and the axis-aligned bounding box A_{i-n+1}, which contains all points afforded by the cell. This step occurs in lines 8-9 of Alg. 3. This check does not change the final output of the search algorithm; instead, it simply filters out spheres which can be trivially proven not to collide with any points to reduce the number of expensive traversals through the affordance set. This step is once again parallelizable across multiple queries: for each SIMD lane of the processor, we can compute the distance from each query sphere to its leaf's bounding box in parallel instead of sequentially. During a parallel query, if spheres are not in collision with the bounding box, then they can be masked out from all remaining collision checks, reducing the overall search time. If no spheres in the query set collide with the axis aligned bounding box, then the query set is provably not in collision, and the search can terminate immediately.

Finally, the collision-checking procedure exhaustively checks for collision between the query sphere and all points $p \in P_{i-n+1}$. If x is nearer to any p than r, then it is in collision. This step of collision-checking occurs in lines 10-13 of Alg. 3. We parallelize this step differently from the previous two: instead of parallelizing across the set of query spheres, we parallelize across the set of test points in each affordance set. This is all for cache locality: parallelizing across the test points means that all memory accesses are in the same contiguous region, instead of requiring inefficient gather instructions across multiple different affordance sets. Such parallelism would not be possible with a conventional k-d tree, as the set of possibly-colliding points is not known to the search until it explores each sub-tree.

Algorithm 3: CollisionCheck

Input: CAPT (T, A, P) containing n points in $\mathbb{R}^k$, sphere s with center x and radius r
Output: Whether s collides with any point in the tree

```
1  d, i ← 0;
2  while i < n − 1 do
3  |  if x_d ≤ T_i then
4  |  |  i ← 2i + 1;
5  |  else
6  |  |  i ← 2i + 2;
7  |  d ← d + 1 mod k;
8  if A_{i−n+1} does not intersect s then
9  |  return false;
10 for p ∈ P_{i−n+1} do
11 |  if ‖x − p‖ ≤ r then
12 |  |  return true;
13 return false;
```

V. Analysis

A. Runtime

The runtime of the construction procedure is dictated by two sub-procedures: the partitioning of the space, which is the same as a k-d tree at $O(kn \log n)$ for a point cloud with n points; and the construction of the final affordance set, which requires $O(ka)$ time for each point, where a is the maximum size of an affordance set. Therefore the total runtime of construction is $O(kn \log n + kna)$. When the dispersion of the point cloud is high, a tends to be small, so the construction runtime is $O(kn \log n)$. However, for extremely low-dispersion point clouds, all points in the point cloud are afforded by each cell, so $a = O(n)$, yielding a much larger construction runtime of $O(kn^2)$. In total, a CAPT consumes $O(kna)$ memory, which may be as much as $O(kn^2)$ for extremely low-dispersion clouds.

In total, each collision query against the collision-affording point tree performs $O(\log n)$ comparisons while traversing the tree, then $O(a)$ checks evaluating the distance to each point $p \in P_i$, where a is the maximum size of any affordance set P_i. Therefore the total search procedure requires $O(\log n + ka)$ steps.

B. Correctness

Lemma V.1. *If P_i is the set of points corresponding to a cell c of the tree, then any sphere s with center x contained by c and with radius $r \in [r_{min}, r_{max}]$ collides with a point in the point cloud PC if and only if s collides with a point in P_i.*

Proof: Since $P_i \subseteq PC$, it is trivial to show that if s collides with a point in P_i, then it collides with a point in PC. Now we must prove the converse; that is, $\exists p \in PC : \|x - p\| \leq r \rightarrow \exists q \in P_i : \|x - q\| \leq r$. Let t be the single point in PC contained by c.

Case 1: $\exists q \in c : \|q - t\| > r_{\text{min}}$. Then, by construction, $P_i = \{p \in PC : \min_{x_c \in c} \|x_c - p\| \leq r_{\text{max}}\}$. If $\exists p \in PC : \|x - p\| \leq r$, then $\min_{x_c \in c} \|x_c - p\| \leq r \leq r_{\text{max}}$. Therefore if the sphere centered at x collides with p, then x collides with a point in P_i.

Case 2: $\nexists q \in c : \|q - t\| > r_{\text{min}}$. By construction, $P_i = \{t\}$. Since $x \in c$, $\|x - t\| \leq r_{\text{min}} \leq r$, so if s is in collision with PC, then s is in collision with P_i. ∎

Lemma V.2. *There exists a filter radius r_{filter} such that the filtering process does not insert a gap in the point cloud PC larger than the minimum collision-check sphere diameter $2r_{min}$.*

Proof:

Let $O \subseteq \mathbb{R}^k$ be the ground truth obstacle volume for a robot environment, and let $PC \subseteq O$ be its surface approximation as a point cloud. Then the L_2 dispersion $\delta(O, PC)$, defined following LaValle [8], is:

$$\delta(O, PC) = \sup_{x \in O} \min_{p \in PC} \|x - p\|$$

That is, any point x contained in O must be no further than $\delta(O, PC)$ from some point p contained in PC.

Now consider some sphere s with diameter $2r_{\text{min}}$. To fit in a gap of width $2r_{\text{min}}$ in PC, s must have center $x \in O$. Then, by the definition of dispersion, $\exists p \in PC : \|x - p\| \leq \delta(O, PC)$.

The filter algorithm presented in Alg. 2 only removes a point $p \in PC$ if $\exists p^* \in PC' : \|p - p^*\| \leq r_{\text{filter}}$; that is, if there is already a retained point closer than r_{filter} to the point in question. Thus, the largest gap that the filter can introduce is bounded by $2\left(r_{\text{filter}} + \delta(O, PC)\right)$. So, it follows that $\exists p^* \in PC' : \|x - p^*\| \leq r_{\text{filter}} + \delta(O, PC)$. To prevent introducing gaps larger than $2r_{\text{min}}$, we then require that:

$$\|x - p^*\| \leq r_{\text{filter}} + \delta(O, PC) \leq r_{\text{min}}$$

Rearranging, we reach $r_{\text{filter}} \leq r_{\text{min}} - \delta(O, PC)$. ∎

The choice of r_{filter} as outlined by Lemma V.2 ensures that the center of any sphere of the robot does not intersect the obstacle O, preventing the robot from travelling directly through an obstacle's surface. However, like any filtering scheme, this may somewhat reduce the envelope of the point cloud. The simplest way to overcome this issue is to instead substitute all query radii r_q with $r'_q = r_q + r_{\text{filter}}$. An alternate interpretation of this is that all points in the filtered point cloud are padded by r_{filter} to become solid, volumetric spheres. With this padding, any value of r_{filter} may be used.

In practice, however, we find empirically that aggressively padding queries or selecting a small r_{filter} is overly conservative (*i.e.*, unnecessary for maintaining plan validity) and detrimental to CAPT construction performance, so more aggressive filtering is possible.

The lemmas above ensure that collision-checking using a collision-affording point tree does not change problem feasibility compared to brute-force collision-checking; *i.e.*, a planner using a collision-affording point tree will not erroneously report that a problem is solvable or unsolvable due to the collision-checking backend. Lemma V.1 implies that a collision-affording point tree does not alter the collision status of any query sphere, so any trajectory which is valid through brute force collision checking is also valid when using a collision-affording point tree. Likewise, Lemma V.2 implies that the filtering procedure is sufficiently conservative to avoid creating spurious gaps, meaning that the filtering process does not make invalid plans feasible, subject to the correct radius padding.

VI. Experiments

We benchmarked collision-checking throughput on an AMD Ryzen™ 9 7950X CPU clocked at 4.5GHz against six different collision-checking and nearest-neighbor implementations. We compared against OctoMaps [10], a voxel-based method, backed by FCL [11]; Nigh [32] and NanoFLANN [31], k-d tree implementations; GNAT [64], a hyperplane partitioning tree (as implemented in the Open Motion Planning Library [65]); and FLANN [30], an approximate nearest-neighbor library. Our approach was implemented in C++ and integrated with an existing vector-accelerated motion planning framework [15]. This planner represents the robot as a hierarchy of spheres, so all robot-environment collision-checking was performed using the CAPT. We also implemented a collision-checking backend using sequential queries against the tree. Here, sequential refers to collision checking each of the n spheres packed into a SIMD vector sequentially, rather than using SIMD intrinsics to evaluate, thus demonstrating the benefits of SIMD parallelism. Sequential queries are used as well for NanoFLANN and OctoMap collision checking.

We benchmark planning performance on the challenging MotionBenchMaker [63] dataset, with 3 robots (the 6-DoF UR5, the 7-DoF Panda, and the 8-DoF Fetch) in 7 scenes (*table pick*, *table under pick*, *box*, *cage*, *bookshelf small*, *bookshelf tall*, and *bookshelf thin*) each, performing 100 different planning problems per scene.

All motion planning was performed on a single thread, using an implementation of a dynamic-domain [66] balanced [67] RRT-Connect [68] limited to 1 million iterations. All plans used the same sequence of randomly sampled configurations, so any difference in performance is from collision-checking speed, not from sampling order. All code was compiled with `clang 16.0.6` using the `-O3` compiler optimization level along with native CPU optimizations.

1) Collision query throughput: We began by evaluating collision-checking throughput on all of the possible backends. First, we recorded the set of all collision-checking queries made by a motion planner using ground-truth primitive geometry on each scene from the MotionBenchMaker [63] dataset. We then generated point clouds for each scene by uniform random sampling of the geometry's surface, then filtered each point cloud with $r_{\text{filter}} \in [1\text{mm}, 10\text{cm}]$ to reach a desired size. All collision-checking throughput experiments were performed on the same set of point clouds. Finally, we executed the exact same queries on each collision-checking method, recording the total timing for collision-checking and avoiding any other timing overhead from other steps in the motion planning process. CAPTs were constructed with $r_{\text{min}} = 1\text{cm}$ and

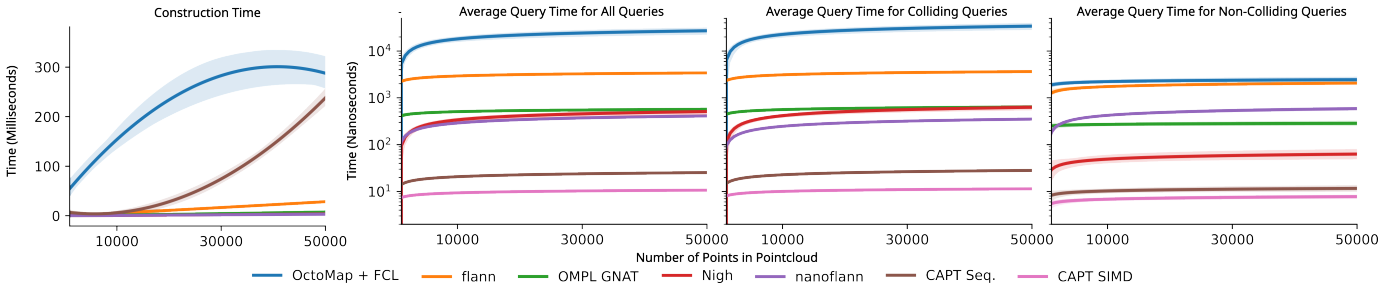


Figure 4: Construction times and average query throughput for pointcloud collision-checking approaches. Methods were evaluated on exemplary pointclouds from each of the 7 benchmark datasets from MotionBenchMaker [63], and evaluated against the set of all collision queries attempted by the motion planner. On the left, construction times are presented in milliseconds. On the right, average query time in nanoseconds are shown on a **logarithmic** scale for all queries, only colliding queries, and queries that are not in collision. 99% confidence intervals are shown over 2nd-order polynomial fitted curves for build time and linear log fit curves for query time.

	Backend	Mean Filter	Mean Build	Med. Build	95% Build	Mean Plan	Med. Plan	95% Plan	Mean Simpl.	Mean Total	Med. Total	95% Total	Succ.
UR5	OctoMap	2.897	120.935	104.892	218.075	78.894	24.736	375.782	23.290	220.906	179.170	521.604	96.5%
	nanoflann		0.346	0.339	0.596	15.526	4.797	78.979	7.810	26.730	15.806	92.221	100.0%
	CAPT Seq.		5.859	5.608	10.467	0.833	0.273	4.226	0.368	9.935	8.829	16.279	100.0%
	CAPT SIMD					**0.490**	**0.146**	**2.542**	0.225	**9.499**	**8.630**	**15.541**	100.0%
	Primitives	-	-	-	-	0.204	0.051	1.039	0.067	0.272	0.117	1.145	100.0%
Panda	OctoMap	2.712	66.127	49.503	122.043	25.579	9.363	109.210	24.424	118.844	103.436	223.829	99.6%
	nanoflann		0.315	0.233	0.611	5.958	3.241	20.352	7.423	16.607	13.726	39.647	100.0%
	CAPT Seq.		4.269	3.227	8.758	0.342	0.182	1.166	0.477	7.781	6.430	15.078	100.0%
	CAPT SIMD					**0.198**	**0.102**	**0.728**	0.261	**7.445**	**6.055**	**14.704**	100.0%
	Primitives	-	-	-	-	0.078	0.034	0.363	0.070	0.148	0.100	0.468	100.0%
Fetch	OctoMap	3.345	160.976	124.376	335.959	309.302	212.953	894.198	121.892	429.362	375.223	900.221	96.6%
	nanoflann		0.469	0.391	0.863	177.912	70.510	661.252	33.131	204.500	102.742	680.011	99.9%
	CAPT Seq.		6.073	5.145	10.508	18.331	3.863	76.251	1.831	29.624	16.015	88.011	99.7%
	CAPT SIMD					**10.736**	**2.159**	**42.974**	0.983	**21.213**	**12.648**	**58.239**	99.7%
	Primitives	-	-	-	-	3.873	0.779	16.290	0.262	4.136	0.966	16.744	99.3%

Table I: Statistics over the MotionBenchMaker [63] dataset. We compare parallel (CAPT SIMD) and sequential (CAPT Seq.) collision-affording point tree collision checking against other collision checking backends. All collision-checking backends except for the primitive geometry used the same filtered point clouds for planning. We report the mean, median, and 95th percentile times spent constructing each collision-checking data structure, planning, simplifying the path, and the total time spent from observation to completed plan for each robot and collision-checking backend. All times are in **milliseconds**.

$r_{\max} = 8\text{cm}$. OctoMaps were constructed with a resolution of 1cm. FLANN indices were created with 4 k-d trees. All experiments were performed on a single CPU thread.

2) Motion planning performance: We implemented full motion-planning backends using OctoMaps and NanoFLANN (as it had the the next-highest throughput, after CAPTs, of any collision-checking method). Each backend was used as part of the same motion planning system, so all speedups are due to collision-checking speed, not sampling order or other aspects of planner efficiency. We compared the relative performance of CAPTs, using both sequential and parallelized SIMD queries, with these two backends. All point clouds were filtered with $r_{\text{filter}} = 2\text{cm}$; although this filter radius is greater than that suggested by Lemma V.2, it was empirically tested to strike an appropriate balance between performance and fidelity, *i.e.*, not allowing invalid plans. See Appendix A for further experiments on the effect of r_{filter}. CAPTs were constructed with $r_{\min}$ and $r_{\max}$ derived from robot geometry; $(r_{\min}, r_{\max})$ was equal to $(1.5\text{cm}, 8\text{cm})$, $(1.2\text{cm}, 6\text{cm})$, and $(1.2\text{cm}, 5.5\text{cm})$ for the UR5, Panda, and Fetch respectively. OctoMaps were constructed with a resolution of 1cm. Once again, these tests were performed exclusively on a single CPU thread to isolate per-thread performance; we leave thread-level parallelization to future work.

A. Empirical Results

1) Collision query throughput: Fig. 4 presents timing results for CAPT construction times and average query throughput times for three different classes of tests: all-colliding queries are a set of queries where each sphere in the query set collides; non-colliding queries have no sphere in collision, and mixed queries are a mix of all-colliding, non-colliding, and partially-colliding queries. Queries against the point cloud were created by recording the set of all queries made by a motion planner in the scene, then recording the runtime of checking the same sequence of queries against each collision-checking system. We observe that CAPT construction is significantly slower than other tree-based nearest neighbor data structures, but is still faster than an OctoMap for construction on point cloud data. The construction procedure exhibits significantly superlinear scaling, showing that point cloud filtering is necessary to use a CAPT effectively.

Collision queries against the tree are overwhelmingly faster than any other data structure, running nearly ten times faster than the nearest data structures for collision checking, for an average performance of 9.89 nanoseconds per query—in comparison, the next best performing approach, NanoFLANN, takes on average 309 nanoseconds per query. Remarkably, collision checks against an OctoMap are over three orders of magnitude slower than against a CAPT (averaging 0.01 milliseconds per query). This demonstrates a need for the

field to reevaluate methods for planning with sensor data: it is possible to achieve extremely high performance gains with a different data structure.

2) Motion planning performance: Table I shows some critical statistics for filtering, collision checking data structure construction, planning, simplification, and total time for the 6-DoF UR5, 7-DoF Panda, and 8-DoF Fetch over the MotionBenchMaker dataset, as described above. These benchmarks show a dramatic improvement in performance over baseline approaches, with SIMD collision checking with a CAPT demonstrating planning times on par with the ground-truth primitive-based planner. For both the UR5 and Panda arms, the 95% quantile of total time end-to-end (filtering, building the CAPT, planning, and simplification) takes less than 16 milliseconds, faster than a 60FPS camera can refresh and provide a new pointcloud. We also highlight that CAPTs provide such an enormous speedup that **motion generation is no longer the most expensive step**. Instead, other steps in the planning pipeline dominate planning times: point cloud filtering and CAPT construction account for the lion's share of planning time.

B. Planning from real sensor data

Finally, we applied our planning system to point clouds observed from a real-world scene with an Intel RealSense D455 RGB-D camera. Fig. 1 shows an example of these data.

1) Static scene: We first created a planning problem in a static snapshot of this scene, requiring a UR5 robot to move from its initial pose to a "reach" point across the table. The original point cloud contained 166587 points; applying our space-filling curve filter (Sec. IV-C) with $r_{\text{filter}} = 2\text{cm}$ reduced the cloud down to 2732 points. In this experiment, we used a minimum radius $r_{\text{min}} = 1.5\text{cm}$ to match the geometry of our model of the UR5. As in Sec. VI-A2, we chose to use a larger r_{filter} than suggested by Lemma V.2 because it empirically did not reduce plan quality; for timings with different values of r_{filter}, see Appendix A. Running on the same machine as our other experiments, we observe a median planning time of 215 microseconds, with a simplified path returned in a total of 575 microseconds. The total duration from the start of point cloud filtering through CAPT construction, planning, and simplification was a median of 7.166 milliseconds—corresponding to a complete planning rate of roughly 140Hz.

2) Dynamic scene: We additionally evaluated our planning system in a live control loop on the UR5, tasking it with moving between a sequence of preset goal waypoints while dodging unmodeled dynamic obstacles (*i.e.*, pool noodles moved by humans to obstruct the robot). Planned trajectories are passed to a simple velocity interpolation controller for time parameterization and execution, and are replaced and updated on every new point cloud. We observe that the system is able to plan at or above the 60FPS camera frame rate; qualitatively, this speed enables the robot to reactively dodge obstacles and effectively maneuver in the scene, despite a lack of motion forecasting or obstacle modeling. We report additional statistics on planning performance and point cloud properties for this experiment in Appendix C; please also see the supplementary material for a video showing the robot in action.

VII. Conclusion

Planning from sensor data is a crucial component of autonomous robotics. In this paper, we present a novel data structure for motion planning with observed point clouds, demonstrating an order-of-magnitude speedup compared to state-of-the-art techniques. We also present a unique filtering algorithm to reduce the density of a point cloud while still providing safety guarantees on collision detection. Combined, these two contributions enable a robot to plan from sensor data in milliseconds on a single CPU core, allowing the robot to plan faster than standard 60FPS camera refresh rates. This means that *robots can now use sampling-based motion planning in real time on purely sensed environments, using only general-purpose low-power hardware.*

The primary limitation of a CAPT is that it is an immutable data structure. After construction, no points in the tree can be inserted or deleted. Since depth-camera images are streamed on a frame-by-frame basis, this is not a problem for collision-checking in dynamic environments, since we can reconstruct the CAPT from scratch for each frame. However, the CAPT's immutability precludes use of a CAPT for the state-space nearest-neighbor search required by most sampling-based planning algorithms. Future extensions to the CAPT structure could enable incremental updating, which would allow it to be used for nearest-neighbor search in the state space during sampling-based planning. Unlike OctoMaps [10], the CAPT does not distinguish between free and unobserved space. This may be problematic for cluttered environments due to occlusions, and in future work we are interested in extending the CAPT to more directly model visibility and occlusion, as well as to better handle perceptual uncertainty by *e.g.*, modeling points as probabilistic particles.

As well, although our choice to duplicate potentially colliding points to construct the affordance sets enables CAPTs to avoid branches and effectively exploit parallelism, it also limits their capacity for scaling to massive point clouds, such as those constructed by autonomous vehicles. In future work, we would be interested in exploring techniques for compressing or otherwise de-duplicating affordance sets. Perhaps more promising is the potential for using CAPTs as a secondary collision data structure paired with another form of spatial subdivision, such as a spatial hash or voxel grid [19, 44]. This hierarchical fused data structure would allow a set of CAPTs to each be "responsible" for only a local neighborhood of a large point cloud while maintaining efficient and parallelizable queries over the entire cloud.

Finally, our performance results challenge conventional assumptions about the nature of planning. We have demonstrated that judicious application of parallelism and insights into the core problems of collision checking against sensor data enables extraordinary improvements in planning time, so much so that motion planning from sensor data could now be seen as a cheap primitive operation, instead of a time-consuming bottleneck.

REFERENCES

[1] A. Orthey, C. Chamzas, and L. E. Kavraki. "Sampling-based motion planning: A comparative review". In: *Annual Review of Control, Robotics, and Autonomous Systems* 7 (2023).

[2] S. M. LaValle, J. J. Kuffner, B. R. Donald, et al. "Rapidly-exploring random trees: Progress and prospects". In: *Algorithmic and computational robotics: new directions* 5 (2001), pp. 293–308.

[3] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". In: *IEEE Transations on Robotics and Automation* 12.4 (1996), pp. 566–580.

[4] J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel. "Motion planning with sequential convex optimization and convex collision checking". In: *The International Journal of Robotics Research* 33.9 (2014), pp. 1251–1270.

[5] M. Zucker, N. Ratliff, A. D. Dragan, M. Pivtoraiko, M. Klingensmith, C. M. Dellin, J. A. Bagnell, and S. S. Srinivasa. "CHOMP: Covariant Hamiltonian optimization for motion planning". In: *The International Journal of Robotics Research* 32.9–10 (2013), pp. 1164–1193.

[6] M. Bhardwaj, B. Sundaralingam, A. Mousavian, N. D. Ratliff, D. Fox, F. Ramos, and B. Boots. "STORM: An Integrated Framework for Fast Joint-Space Model-Predictive Control for Reactive Manipulation". In: *Conference on Robot Learning*. 2021.

[7] J. Bialkowski, S. Karaman, and E. Frazzoli. "Massively Parallelizing the RRT and the RRT*". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Sept. 2011, pp. 3513–3518.

[8] S. M. LaValle. *Planning algorithms*. Cambridge university press, 2006.

[9] J. Schauer and A. Nüchter. "Collision detection between point clouds using an efficient kd tree implementation". In: *Advanced Engineering Informatics* 29.3 (2015), pp. 440–458.

[10] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. "OctoMap: An efficient probabilistic 3D mapping framework based on octrees". In: *Autonomous robots* 34 (2013), pp. 189–206.

[11] J. Pan, S. Chitta, and D. Manocha. "FCL: A general purpose library for collision and proximity queries". In: *IEEE International Conference on Robotics and Automation*. IEEE. 2012, pp. 3859–3866.

[12] J. Pan, I. A. Şucan, S. Chitta, and D. Manocha. "Real-time collision detection and distance computation on point cloud sensor data". In: *2013 IEEE International Conference on Robotics and Automation*. IEEE. 2013, pp. 3593–3599.

[13] M. Danielczuk, A. Mousavian, C. Eppner, and D. Fox. "Object rearrangement using learned implicit collision functions". In: *IEEE International Conference on Robotics and Automation*. IEEE. 2021, pp. 6010–6017.

[14] A. Murali, A. Mousavian, C. Eppner, A. Fishman, and D. Fox. "CabiNet: Scaling Neural Collision Detection for Object Rearrangement with Procedural Scene Generation". In: *IEEE International Conference on Robotics and Automation*. 2023, pp. 1866–1874.

[15] W. Thomason, Z. Kingston, and L. E. Kavraki. "Motions in Microseconds via Vectorized Sampling-Based Planning". In: *IEEE International Conference on Robotics and Automation*. 2024.

[16] B. Sundaralingam, S. K. S. Hari, A. Fishman, C. Garrett, K. Van Wyk, V. Blukis, A. Millane, H. Oleynikova, A. Handa, F. Ramos, N. Ratliff, and D. Fox. "CuRobo: Parallelized Collision-Free Robot Motion Generation". In: *IEEE International Conference on Robotics and Automation*. 2023, pp. 8112–8119.

[17] V. Vasilopoulos, S. Garg, P. Piacenza, J. Huh, and V. Isler. "RAMP: Hierarchical Reactive Motion Planning for Manipulation Tasks Using Implicit Signed Distance Functions". In: *arXiv preprint arXiv:2305.10534* (2023).

[18] A. T. Le, G. Chalvatzaki, A. Biess, and J. R. Peters. "Accelerating Motion Planning via Optimal Transport". In: *Advances in Neural Information Processing Systems* 36 (2024).

[19] A. Millane, H. Oleynikova, E. Wirbel, R. Steiner, V. Ramasamy, D. Tingdahl, and R. Siegwart. "nvblox: GPU-Accelerated Incremental Signed Distance Field Mapping". In: *arXiv preprint arXiv:2311.00626* (2023).

[20] J. H. Reif. "Complexity of the mover's problem and generalizations". In: *Annual Symposium on Foundations of Computer Science*. IEEE Computer Society. 1979, pp. 421–427.

[21] J. Canny. *The complexity of robot motion planning*. MIT press, 1988.

[22] S. Thrun, W. Burgard, and D. Fox. "Probabilistic robotics". In: *Communications of the ACM* 45.3 (2002), pp. 52–57.

[23] J. L. Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[24] P. Ram and K. Sinha. "Revisiting kd-tree for nearest neighbor search". In: *Proceedings of the 25th acm sigkdd international conference on knowledge discovery & data mining*. 2019, pp. 1378–1388.

[25] Y. Chen, L. Zhou, Y. Tang, J. P. Singh, N. Bouguila, C. Wang, H. Wang, and J. Du. "Fast neighbor search by using revised kd tree". In: *Information Sciences* 472 (2019), pp. 145–162.

[26] R. Pinkham, S. Zeng, and Z. Zhang. "Quicknn: Memory and performance optimization of kd tree based nearest neighbor search for 3d point clouds". In: *2020 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE. 2020, pp. 180–192.

[27] J. Klein and G. Zachmann. "Point cloud collision detection". In: *Computer Graphics Forum*. Vol. 23. 3. Wiley Online Library. 2004, pp. 567–576.

[28] F. Chen, R. Ying, J. Xue, F. Wen, and P. Liu. "ParallelNN: A Parallel Octree-based Nearest Neighbor Search Accelerator for 3D Point Clouds". In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 403–414.

[29] P. H. E. Becker, J.-M. Arnau, and A. González. "KD Bonsai: ISA-Extensions to Compress KD Trees for Autonomous Driving Tasks". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 2023, pp. 1–13.

[30] M. Muja and D. Lowe. "Flann-fast library for approximate nearest neighbors user manual". In: *Computer Science Department, University of British Columbia, Vancouver, BC, Canada* 5 (2009), p. 6.

[31] J. L. Blanco and P. K. Rai. *nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees*. https://github.com/jlblancoc/nanoflann. 2014.

[32] J. Ichnowski and A. Kuntz. *nigh: Concurrent exact nearest neighbor searching in robotics-relevant spaces, including Euclidean, SO(3), SE(3) and weighted combinations thereof*. https://github.com/UNC-Robotics/nigh. 2018.

[33] H. Moravec and A. Elfes. "High resolution maps from wide angle sonar". In: *Proceedings. 1985 IEEE International Conference on Robotics and Automation*. Vol. 2. 1985, pp. 116–121.

[34] S. Thrun and A. Bücken. "Integrating grid-based and topological maps for mobile robot navigation". In: *Proceedings of the national conference on artificial intelligence*. 1996, pp. 944–951.

[35] R. B. Rusu, I. A. Şucan, B. Gerkey, S. Chitta, M. Beetz, and L. E. Kavraki. "Real-time perception-guided motion planning for a personal robot". In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2009, pp. 4245–4252.

[36] D. Meagher. "Geometric modeling using octree encoding". In: *Computer graphics and image processing* 19.2 (1982), pp. 129–147.

[37] S. Chitta, I. Sucan, and S. Cousins. "Moveit!" In: *IEEE Robotics & Automation Magazine* 19.1 (2012), pp. 18–19.

[38] E. Vidal Garcia, M. Moll, N. Palomeras, J. D. Hernández, M. Carreras, and L. E. Kavraki. "Online Multilayered Motion Planning with Dynamic Constraints for Autonomous Underwater Vehicles". In: *IEEE International Conference on Robotics and Automation*. May 2019, pp. 8936–8942.

[39] T. Dang, M. Tranzatto, S. Khattak, F. Mascarich, K. Alexis, and M. Hutter. "Graph-based subterranean exploration path planning using aerial and legged robots". In: *Journal of Field Robotics* 37.8 (2020), pp. 1363–1388.

[40] C. Badue, R. Guidolini, R. V. Carneiro, P. Azevedo, V. B. Cardoso, A. Forechi, L. Jesus, R. Berriel, T. M. Paixao, F. Mutz, et al. "Self-driving cars: A survey". In: *Expert Systems with Applications* 165 (2021), p. 113816.

[41] Y. Lu, Z. Xue, G.-S. Xia, and L. Zhang. "A survey on vision-based UAV navigation". In: *Geo-spatial information science* 21.1 (2018), pp. 21–32.

[42] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. "Kinectfusion: Real-time dense surface mapping and tracking". In: *IEEE international symposium on mixed and augmented reality*. Ieee. 2011, pp. 127–136.

[43] T. Whelan, S. Leutenegger, R. Salas-Moreno, B. Glocker, and A. Davison. "ElasticFusion: Dense SLAM without a pose graph". In: Robotics: Science and Systems. 2015.

[44] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto. "Voxblox: Incremental 3D Euclidean signed distance fields for on-board MAV

planning". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2017, pp. 1366–1373.

[45] V. Reijgwart, A. Millane, H. Oleynikova, R. Siegwart, C. Cadena, and J. Nieto. "Voxgraph: Globally Consistent, Volumetric Mapping Using Signed Distance Function Submaps". In: *IEEE Robotics and Automation Letters* (2020).

[46] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis". In: *Commun. ACM* 65.1 (Dec. 2021), pp. 99–106.

[47] M. Adamkiewicz, T. Chen, A. Caccavale, R. Gardner, P. Culbertson, J. Bohg, and M. Schwager. "Vision-only robot navigation in a neural radiance world". In: *IEEE Robotics and Automation Letters* 7.2 (2022), pp. 4606–4613.

[48] T. Chen, P. Culbertson, and M. Schwager. "CATNIPS: Collision Avoidance Through Neural Implicit Probabilistic Scenes". In: *arXiv preprint arXiv:2302.12931* (2023).

[49] T. Müller, A. Evans, C. Schied, and A. Keller. "Instant neural graphics primitives with a multiresolution hash encoding". In: *ACM Transactions on Graphics (ToG)* 41.4 (2022), pp. 1–15.

[50] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. Tech. rep. International Business Machines Company New York, 1966.

[51] Z. Ying, S. Bhuyan, Y. Kang, Y. Zhang, M. T. Kandemir, and C. R. Das. "EdgePC: Efficient Deep Learning Analytics for Point Clouds on Edge Devices". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 2023, pp. 1–14.

[52] M. Connor and P. Kumar. "Fast construction of k-nearest neighbor graphs for point clouds". In: *IEEE transactions on visualization and computer graphics* 16.4 (2010), pp. 599–608.

[53] J. Ichnowski and R. Alterovitz. "Parallel sampling-based motion planning with superlinear speedup". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 1206–1212.

[54] E. Plaku, K. E. Bekris, B. Y. Chen, A. M. Ladd, and L. E. Kavraki. "Sampling-Based Roadmap of Trees for Parallel Motion Planning". In: *IEEE Transactions on Robotics* 21.4 (2005), pp. 597–608.

[55] N. M. Amato and L. K. Dale. "Probabilistic Roadmap Methods Are Embarrassingly Parallel". In: *IEEE International Conference on Robotics and Automation*. Vol. 1. May 1999, 688–694 vol.1.

[56] S. A. Jacobs, K. Manavi, J. Burgos, J. Denny, S. Thomas, and N. M. Amato. "A Scalable Method for Parallelizing Sampling-Based Motion Planning Algorithms". In: *IEEE International Conference on Robotics and Automation*. 2012, pp. 2529–2536.

[57] J. Pan and D. Manocha. "GPU-based parallel collision detection for fast motion planning". In: *The International Journal of Robotics Research* 31.2 (2012), pp. 187–200.

[58] J. H. Friedman, J. L. Bentley, and R. A. Finkel. "An algorithm for finding best matches in logarithmic expected time". In: *ACM Transactions on Mathematical Software (TOMS)* 3.3 (1977), pp. 209–226.

[59] M. Mukadam, J. Dong, X. Yan, F. Dellaert, and B. Boots. "Continuous-Time Gaussian Process Motion Planning via Probabilistic Inference". In: *The International Journal of Robotics Research* 37.11 (Sept. 2018), pp. 1319–1340.

[60] G. Bradshaw and C. O'Sullivan. "Adaptive medial-axis approximation for sphere-tree construction". In: *ACM Transactions on Graphics (TOG)* 23.1 (2004), pp. 1–26.

[61] J. W. J. Williams. "Algorithm 232: Heapsort". In: *Communications of the ACM* 7.6 (June 1964), pp. 347–348.

[62] C. A. Hoare. "Algorithm 65: find". In: *Communications of the ACM* 4.7 (1961), pp. 321–322.

[63] C. Chamzas, C. Quintero-Pena, Z. Kingston, A. Orthey, D. Rakita, M. Gleicher, M. Toussaint, and L. E. Kavraki. "MotionBenchMaker: A tool to generate and benchmark motion planning datasets". In: *IEEE Robotics and Automation Letters* 7.2 (2021), pp. 882–889.

[64] S. Brin. "Near neighbor search in large metric spaces". In: *VLDB*. Vol. 95. 58. Citeseer. 1995, pp. 574–584.

[65] I. A. Sucan, M. Moll, and L. E. Kavraki. "The open motion planning library". In: *IEEE Robotics & Automation Magazine* 19.4 (2012), pp. 72–82.

[66] L. Jaillet, A. Yershova, S. M. La Valle, and T. Siméon. "Adaptive tuning of the sampling domain for dynamic-domain RRTs". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2005, pp. 2851–2856.

[67] J. J. Kuffner and S. M. LaValle. *An efficient approach to path planning using balanced bidirectional RRT search*. Tech. rep. Robotics Institute, Carnegie Mellon University, 2005.

[68] J. J. Kuffner and S. M. LaValle. "RRT-connect: An efficient approach to single-query path planning". In: *IEEE International Conference on Robotics and Automation*. Vol. 2. IEEE. 2000, pp. 995–1001.

APPENDIX

A. Empirical impact of filter radii

We ran the simulated planning experiments from Sec. VI-A2 for the Panda robot with different values for the r_{filter} parameter to Alg. 2 to investigate its effect on CAPT construction times, planning performance, and problem feasibility. These results are shown in Table IV. We note that, although the CAPT is sensitive to the value of r_{filter} in its construction time in particular, even for conservative values of r_{filter} (*e.g.*, matching or less than the bound suggested by Lemma V.2), our planning times are always faster than any baseline, and our total times are competitive or fastest. Note also that the baselines are only evaluated with the most aggressive value of r_{filter}, and would also be slowed in planning and total time for more conservative values. Finally, Table IV also validates that, for all values of r_{filter}, the paths we find are valid **with respect to the exact, primitive geometric obstacles**, evaluated post hoc.

B. Empirical dispersion of point clouds

We recorded the values of dispersion $\delta(O, PC)$ as discussed in Lemma V.2 across the point clouds used for our simulation collision query throughput and planning performance experiments. We compute dispersion by using a standard nearest-neighbors data structure to query the distance to the closest neighbor of each point in each point cloud for each scene, and recording basic summary statistics, as shown in Table II. We note that $\delta(O, PC)$ is often quite small for well-observed obstacles; further, as argued in the proof sketch for Lemma V.2, the increase in dispersion resulting from applying Alg. 2 is bounded by $r_{\text{filter}} + \delta(O, PC)$. Therefore, selecting an r_{filter} in the same order of magnitude as r_{min} is a reasonable choice.

C. Real-robot planning experiments

We evaluate the impact of the value of r_{filter} via the real-robot demonstration of planning with CAPTs discussed in Sec. VI-B. We collected 300 observed point clouds from sequential frames of RGB-D video generated by an Intel Realsense D455 sensor, and computed the mean post-filter point cloud sizes and essential timing statistics (*i.e.*, timing for applying the filter, building a CAPT on the filtered point cloud, solving a motion planning problem with the CAPT, and simplifying the solution found—which requires further collision-checking) for a range of values of r_{filter}, keeping the values of r_{min} and r_{max} constant at 1.5cm and 8cm respectively. Table III shows these quantitative results, demonstrating that although CAPTs remain fast at conservatively small values of r_{filter}, increasing r_{filter} dramatically decreases both point cloud size and CAPT construction time.

Crucially, we also qualitatively find that, for any $r_{\text{filter}} \leq 2\text{cm}$, the generated trajectories are valid and do not intersect or contact any obstacles. As such, broadly speaking, a user can vary the value of r_{filter} to trade fidelity of representation for performance, and reasonable balances of the two are easy to find.

	$\delta(O, PC)$			$\delta(O, PC')$		
r_{filter}	Mean	Median	95%	Mean	Median	95%
0.5	0.00336	0.00198	0.01075	0.00763	0.00675	0.01314
1				0.01253	0.01182	0.01774
1.5				0.01758	0.01696	0.02314
1.8				0.02055	0.01995	0.02663
1.9				0.02158	0.02097	0.02790
2				0.02261	0.02199	0.02919

Table II: Empirical measurements of point cloud dispersion before (left grouping) and after (right grouping) applying the filter proposed in Alg. 2 with a range of r_{filter} values (leftmost column, in cm). Pre-filter values are identical and accumulated over all MotionBenchMaker problems used for evaluation with the Panda robot; all dispersion values are given in cm.

r_{filter}	$\lvert PC' \rvert$	Filter	Build	Plan	Simpl.	Total
1	16225	6.944	30.373	0.168	0.275	37.761
1.5	7872	5.682	7.669	0.079	0.085	13.517
2	4614	4.990	3.964	0.087	0.089	9.131

Table III: Impact of r_{filter} (in centimeters) on planning times for the real robot experiment with the UR5. The initial pointcloud size $|PC|$ is always 307200 (constructed from a 640x480 pixel depth image). We report mean point cloud sizes, mean point cloud filtering times, mean CAPT construction times, mean motion planning times, and mean path simplification times, as well as the mean total end-to-end planning time. r_{filter} is in **centimeters**, and all times are in **milliseconds**.

r_{filter}	Mean Filter	Mean Build	Med. Build	95% Build	Mean Plan	Med. Plan	95% Plan	Mean Simpl.	Mean Total	Med. Total	95% Total	Succ.
0.5	4.544	62.560	56.294	122.966	0.427	0.210	1.581	0.480	68.012	61.546	132.708	100.0
1	3.307	19.339	13.286	41.888	0.288	0.136	1.136	0.344	23.280	16.514	49.667	100.0
1.5	2.742	8.069	5.954	16.595	0.228	0.112	0.923	0.288	11.329	8.753	22.851	100.0
1.8	2.514	5.378	4.278	10.895	0.212	0.103	0.923	0.265	8.370	6.905	16.565	100.0
1.9	2.454	4.701	3.613	9.760	0.207	0.102	0.734	0.265	7.628	6.281	15.149	100.0
2	2.400	4.098	3.125	8.467	0.192	0.099	0.702	0.253	6.945	5.658	13.812	100.0

Table IV: Effect of r_{filter} on MotionBenchMaker planning performance for the Panda robot for CAPT SIMD. All planning results are valid with respect to the underlying primitive scene representation sampled to generate simulated pointclouds. r_{filter} is given in **centimeters**, and all times are given in **milliseconds**.