Motions in Microseconds via Vectorized Sampling-Based Planning

Wil Thomason†, Zachary Kingston†, and Lydia E. Kavraki

***Abstract*— Modern sampling-based motion planning algorithms typically take between hundreds of milliseconds to dozens of seconds to find collision-free motions for high degree-of-freedom problems. This paper presents performance improvements of more than 500x over the state-of-the-art, bringing planning times into the range of microseconds and solution rates into the range of kilohertz, without specialized hardware. Our key insight is how to exploit fine-grained parallelism within sampling-based planners, providing generality-preserving algorithmic improvements to any such planner and significantly accelerating critical subroutines, such as forward kinematics and collision checking. We demonstrate our approach over a diverse set of challenging, realistic problems for complex robots ranging from 7 to 14 degrees-of-freedom. Moreover, we show that our approach does not require high-power hardware by also evaluating on a low-power single-board computer. The planning speeds demonstrated are fast enough to reside in the range of control frequencies and open up new avenues of motion planning research.**

I. Introduction

High degree-of-freedom (DoF) robots rely on *motion planning* to move in complex workspaces, either using sampling-based approximations [1–5] or numerical optimization [6, 7]. These planners are general and can solve realistic, challenging problems in hundreds of milliseconds to dozens of seconds on consumer CPUs. However, this level of performance falls short—it is too slow for reactive operation in evolving environments and hampers algorithms for higher-level autonomy such as integrated task and motion planning.

A large literature accelerates motion planning with *coarse-grained* (*e.g.*, thread or process-level) parallelism [8–12], but these methods have seen relatively little uptake in practice, as their performance gains do not justify the added complexity. More recent work [13–15] uses GPU-based parallelism to improve performance, but at the cost of communication overhead, algorithmic limitations, and the additional expense and power consumption of GPU hardware. In general, the field has come to believe that sampling-based motion planner (SBMP) primitives (*e.g.*, checking if motion between two states is valid) are either inherently serial, cannot be accelerated without specialized hardware, or cannot be parallelized without paying a greater cost than the parallelism saves.

We refute this belief and show several orders of magnitude performance improvement over the state-of-the-art (more than 500x faster) by contributing insights into *fine-grained* parallelism and work-ordering in SBMPs. Our core insight is the use of *vector-oriented* state representations and planning primitives (*e.g.*, forward kinematics and collision checking), which enable fine interleaving of parallel and serial operation. Crucially, we use "Single Instruction/Multiple Data" (SIMD) instructions to execute these primitives with high throughput and low latency on ubiquitous consumer CPUs, accelerating *almost any* SBMP for "free". These insights let us plan high-quality paths at reactive speeds (*e.g.*, a median time of 40 µs for the 7 DoF Panda over the MotionBenchMaker [16] dataset, *i.e.*, 25 kHz—see Table I) on a single CPU core.

Our method significantly outperforms standard implementations [17] of state-of-the-art SBMPs on both desktop and low-power single board computers. Moreover, this work will enhance any work that uses motion planning, and our perspective on vector-oriented planning primitives extends beyond CPU SIMD instructions to other, similar parallelism models, *e.g.*, GPUs. The planning speeds demonstrated blur the line between planning and control, and give cause to re-evaluate assumptions about robot motion.

II. Related Work

Motion planning is PSPACE-complete [18, 19] in general, but sampling-based motion planners (SBMPs) empirically can efficiently solve challenging problems. SBMP requires computationally expensive subroutines, *e.g.*, forward kinematics (FK), collision checking (CC), and nearest neighbor (NN) search [3]. CC dominates computational cost in SBMP (noted by many, *e.g.*, [10]); however, NN search may dominate in high-dimensions or with large numbers of states [20]. FK is used by CC and contributes to its cost. For efficiency, SBMPs must use fast CC and avoid as much FK/CC as possible—as such, in this work, we target FK and CC performance.

Most CC algorithms use a *broadphase* to approximate the set of possible collisions, and a *narrowphase* to find exact collisions from the approximate set [21]. The broadphase is vital for performant CC and can be accelerated with GPU [22] or SIMD [23] parallelism. Efficient, general CC libraries (*e.g.*, [24, 25]) offer many options for these phases. Given its impact on SBMP performance, many works have reduced CC effort via *e.g.*, heuristic ordering [26], lazy checking [27, 28], checking only likely-valid edges [29, 30], etc. Recent work has learned a combined FK/CC "primitive" [31–33]; other work has learned distance-to-collision functions [34, 35].

A. Parallelism in Motion Planning

Parallelized planners have been sought since the advent of motion planning (*e.g.*, Barraquand and Latombe [36] and Henrich [37]). We broadly categorize parallelism in motion planning as either *coarse-grained* or *fine-grained*. In our use,

† Equal contribution. All authors are affiliated with the Department of Computer Science, Rice University, Houston TX, USA {wbthomason, zak, kavraki}@rice.edu. This work was supported in part by NSF RI 2008720, NSF ITR 2127309 for the Computing Research Association CIFellows Project, and Rice University Funds.

coarse-grained refers to parallelism at the level of subroutines or planner components, such as running many planners in parallel, or running CC in a separate thread. Fine-grained refers to parallelism at the level of primitive operations, such as checking several states for collisions simultaneously in the same thread, without architectural changes.

Parallelism in SBMP is typically coarse-grained, *e.g.*, simply running independent planners in parallel. This improves average-case performance [38]; the set of solutions can also be hybridized together to improve plan quality [39]. Early work (*e.g.*, Amato and Dale [9]) observed that roadmap-based planners (*e.g.*, PRM [5]) are amenable to coarse-grained parallelism. Parallel SBMP has also been achieved by constructing a forest of planning trees [40], potentially in distinct regions of the search space [11], and by parallelizing components of the RRT* ASAO planner [41, 42]. These methods typically offer sub-linear (in the degree of parallelism) performance improvement (with some exceptions [42]) due to the synchronization overhead and architectural complexity required. Still other work uses coarse-grained parallelism in graph search used in roadmap- or search-based planning, both on CPUs [43, 44] and GPUs [45–47]. In contrast to the work discussed above, this work investigates a novel approach to *fine-grained* parallelism in SBMPs, which is understudied in the field.

Recently, there has been significant interest in applying GPUs more broadly to motion planning. The most successful approaches include parallelized sampling-based MPC [13], parallel particle-based optimization seeded by a partially parallelized RRT-like planner [14], and end-to-end learning of a neural local control policy from a dataset of motion plans [15]. Earlier work also investigated GPU-parallelized CC [10, 48]. Although these methods show promising performance, they require powerful GPUs for efficiency and impose the overhead of moving data between the GPU and CPU.

B. Hardware-Accelerated Motion Planning

Hardware acceleration is crucial for our proposed vector-based approach to SBMP. We use SIMD instructions, a feature ubiquitous on consumer CPUs[1]. Hardware acceleration has long been used for motion planning, with particular focus on accelerating CC [50]. Some work (*e.g.*, Ichnowski and Alterovitz [51]'s compile-time specialized SBMP, or Carpentier et al. [52]'s statically dispatched, precompiled dynamics algorithms) implicitly exploits hardware acceleration by creating "machine sympathetic" implementations—code that enables compilers, etc. to better exploit hardware capabilities.

Many modern robotics algorithms are accelerated by GPUs, *e.g.*, using CUDA. GPU acceleration has been applied to SBMPs [14, 53, 54], MPC [13, 55] and trajectory optimization [12, 14]. Recently, ASIC- or FPGA-based accelerators [56] have been proposed, *e.g.*, to validate an entire roadmap at once with an FPGA [57–59] or as an external CC accelerator [60]. Neuman et al. [61] investigated "robomorphic" computing with specialized accelerators for common robotics algorithms.

[1]Our primary implementation uses AVX2, which has been broadly available on Intel and AMD CPUs since 2013. Lower-width SIMD instruction sets such as SSE have been available since 1999 [49].

However, GPUs and other accelerators come at a cost: there is latency in communicating with the device, there are restrictions on the types of algorithms that can be applied on specialized hardware, and often there is a relatively high cost to send data back and forth [60, 62]. Our approach uses native SIMD instructions on the CPU, inflicting at worst a slight overhead penalty[2] to achieve large performance gains.

III. Method

Most SBMPs can be decomposed into a handful of "primitive" operations (see Ch. 7 in Choset et al. [2]). Algorithms typically use (approximate) nearest-neighbors (NN) to find nearby states, a state validity function (*e.g.*, checking for collisions), a local planner or steering function to grow edges between states, and an edge validity function to check these edges. Validity functions usually require forward kinematics (FK) to compute the poses of the robot's links in its workspace from a configuration.

We lift a selection of these primitives—FK and state/edge validity checking—to operate over *vectors* of states in parallel. This lifting immediately accelerates the primitive operations by multiplying their throughput. More importantly, shifting perspective to vector primitives (1) admits low-overhead parallelism that cooperates with sequential code, and (2) reveals beneficial algorithmic changes to the primitives based on insights about their specific uses in SBMP. This perspective allows us to exploit ubiquitous hardware parallelism via SIMD instructions, resulting in highly efficient implementations of our vector primitives. Finally, by focusing on primitives common across SBMPs, we improve the performance of *almost any* SBMP without requiring significant algorithmic changes.

A. Vectorized Motion Planning

"Vectorized" is an overloaded term; we use it in the SIMD sense, where a "vector" is a fixed-length set of values with the same scalar datatype (*e.g.*, floating-point numbers) and a "vectorized operation" is an operation that transforms all values in a vector independently, in parallel.

This parallelism model is similar to GPU computing (and our lifted vector operations may benefit GPU-based planners), but with a few key differences. We focus on CPU-based SIMD parallelism—our algorithms run on any modern computer, even those without a GPU. This increases applicability and decreases both the barrier to entry and the power consumption of our technique. CPU-based SIMD parallelism is also better-suited to the opportunities for parallelism in SBMP: it has significantly lower overhead than GPU-, thread-, or process-based parallelism[3] and is amenable both to fine-grained interleaving of parallel and sequential code and to efficient computation for relatively small workloads.

Exploiting SIMD instructions requires careful consideration of data structure and algorithm design, often involving

[2]CPUs may downclock when using SIMD instructions—there is also a cost to move data in and out of vector registers.

[3]While modern hardware is quite parallelism-performant, there is still non-negligible overhead (*e.g.*, GPU-CPU communication latency) in the tens of microseconds, which can easily add up into the milliseconds.

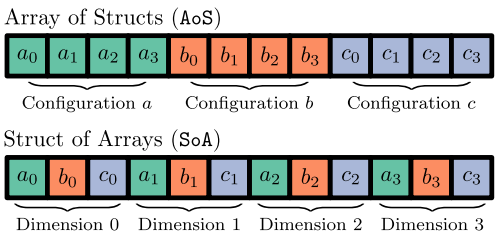

Fig. 1. Struct-of-Arrays (SoA) and Array-of-Structs (AoS) memory layout. Here, there are three configurations a, b, and c, each with four dimensions. AoS is the more "natural" layout of memory, but hard to exploit with SIMD.

unconventional memory layouts to ensure adequate *data parallelism*. Our approach addresses this challenge through a novel Struct-of-Arrays (SoA) memory layout for FK and CC. This choice enables seamless exploitation of data parallelism, allowing us to pose and check multiple configurations for collision in parallel. The SoA layout stands in contrast to the more common Array-of-Structs (AoS) layout (illustrated in Fig. 1), which is less favorable for SIMD approaches as it causes memory access patterns that slow access to values.

Many SBMP algorithms and subroutines make heavy use of conditional branching, inimical to parallel code. However, as CPU-based SIMD parallelism allows easy interleaving of parallel and sequential code, and as our subroutines have reduced branching, we are able to sidestep this problem more easily compared to other forms of parallelism[4]. These properties mean that our algorithm implementations, despite being parallelized, are close to "standard" algorithms—there is no explicit synchronization or communication code, etc.

The overhead of conventional mechanisms (*e.g.*, threads) limits naive parallelization of SBMP—reducing this overhead is especially challenging as most SBMP algorithms rapidly alternate between expensive, parallelization-friendly subroutines (*e.g.*, CC) and code that relies on these subroutines but is not itself easily parallelized (*e.g.*, graph search). Fortunately, the relatively low overhead of CPU SIMD-based parallelism—and the specific nature of this overhead, which modern compilers excel at reducing—means that our vector-oriented primitives (the expensive subroutines) can be efficiently interleaved within a SBMP. This property may suffice to accelerate SBMP by, *e.g.*, reducing the cost of a single collision check. However, by using SoA memory layouts for our vector-oriented operations, we can not only perform the computation required for SBMP in parallel, but also exploit motion-planning-specific independence patterns in this computation to *intelligently order* the requisite operations to improve overall performance.

B. Vectorized Forward Kinematics

Vectorized FK is necessary to pose batches of states in parallel for subsequent parallel CC; sequentially posing each element in a batch introduces a bottleneck that reduces overall throughput. Naive vectorization of FK attempts to compute the poses for a *single* configuration faster—we instead choose a less conventional use of vectorization: carrying out each operation in sequence, but on *multiple configurations* simultaneously.

FK implementations commonly use dynamic branching and joint-type polymorphism to compute transforms between links (*e.g.*,, KDL [63]). This structure is difficult for compilers to optimize and has spurious data dependencies between link transforms, which decreases throughput and causes slower operations across the entire vector of configurations. These dependencies arise as the compiler cannot determine if poses later in the kinematic tree depend on earlier poses (or, better still, on components of these poses). Even naively vectorizing FK for multiple configurations requires vector configuration and pose data structures, and use of vector operations, which are tedious to manually implement.

We overcome these challenges with a novel *tracing compiler* for robot kinematics. This compiler takes in standard Universal Robot Description Format (URDF) files and *traces* the operations of arbitrary functions of the robot's kinematics (*e.g.*, FK). It uses this trace to automatically generate (1) a vector configuration structure representing a batch of configurations and (2) the minimal set of operations required to compute the traced function. This latter output constitutes an "unrolled" FK loop that avoids branching and spurious data dependencies, allowing an optimizing compiler to generate faster machine code. Further, our tracing compiler applies optimizations to reduce the operations required, *e.g.*, constant folding, algebraic simplification, removing redundant negations, etc. This use of automatic code generation creates hyper-specialized vector-lifted FK without loss of generality, as the tracing compiler itself is general. We note other techniques for efficient FK by, *e.g.*, Carpentier et al. [52], which uses static polymorphism and the Curiously Recurring Template Pattern (CRTP) for compile-time optimized FK routines. In contrast, our tracing compiler, by merit of tracking the precise operations (*e.g.*, the multiplies, sines, etc. from input configuration to output pose), outputs "straightline" code that removes operations that are not necessarily detectable at compile-time with CRTP.

C. Vectorized Collision Checking

Existing approaches to CC (*e.g.*, broadphases, narrowphase triangle mesh collision algorithms [64]) are optimized for checking a single configuration. Fully vectorizing these approaches is challenging. We instead draw inspiration from classical work on simplified representations of robots and obstacles [65, 66] to automatically generate collision geometry from meshes using primitives (*i.e.*, spheres, cylinders, and cuboids). Sphere-based representations are common in the trajectory optimization literature [7, 15].

By representing the robot and environment as geometric primitives we can vectorize intersection tests between pairs of such primitives. We check batches of robot poses for self-collision and environment collision in parallel and reject the whole batch if any collide. Surprisingly, we see that this narrowphase-only approach (due to its reduced branching)

[4]We also benefit from advances in modern hardware, which have produced branch predictors and fused "test-and-branch" instructions that are highly performant on the limited set of branches we retain.

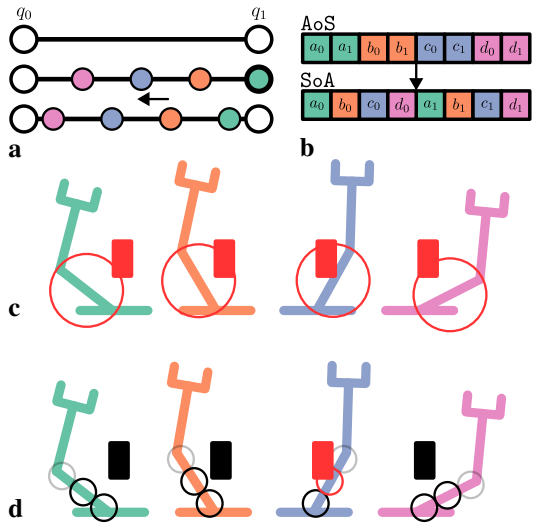

Fig. 2. Illustration of the "raked" motion validator for a two-link mechanism. **a)** The rake consists of evenly spaced configurations (here, $n = 4$), which are "raked" backwards to achieve sufficient resolution. **b)** These configurations are computed in SoA form from an initial AoS layout, then checked in parallel. **c)** Spherical approximations of collision geometry are checked in parallel. A hierarchy of spheres is used to avoid unnecessary checks. **d)** When any collision is discovered at the lowest refinement level (in red), the entire check terminates (the last sphere in grey is skipped).

can be highly efficient even in complex environments. Further, using spheres to represent the robot's geometry synergizes with our tracing compiler for FK—as we only compute the position of each sphere (rather than a full $SE(3)$ pose), the compiler skips a large number of irrelevant operations.

D. Vectorized Motion Validation

SBMPs validate not only individual states, but also the motions between them. Typically, motion validation requires discretizing a continuous motion and validating each state in the discretization. Thus, this is where SBMPs spend most of their time, and a significant body of work [27, 67, 68] has gone into reducing the number of edges validated during planning. However, here is where our perspective on vector-lifted primitives shines: by combining our previously developed insights into vectorized FK and individual state CC, we unearth further algorithmic insights to improve the performance of motion validation via vectorization.

For efficient motion validation, we want to stop checking invalid motions as quickly as possible. Assuming uniform probability of collision along a motion[5], $\frac{n}{2}$ CC attempts are wasted (in expectation) for an invalid motion discretized into n states. Due to our perspective on vector-lifted FK and CC, we can reduce the amount of wasted computation by testing a *spatially distributed* set of states in parallel. Without loss of generality, consider a vector of eight states, and a motion discretized into n states. We can *simultaneously* check states $[0, \frac{n}{8}, \ldots, \frac{7n}{8}]$ for the cost of a single check[6], and—if no collisions are found—comb through the remaining states by incrementing each index, for at most $\frac{n}{8}$ iterations. We refer to this spatially distributed collision check as the "rake" (Fig. 2). Beyond improving CC throughput by decreasing the total number of checks required by a factor of the width of the vector, by spatially distributing the states checked, we increase the probability of exiting CC early for invalid motions—the validity of close states is correlated, so we have a higher chance of finding an invalid state by testing along the entire motion at once, compared to, *e.g.*, the first eight states at once. Other work has investigated spatially distributed collision check scheduling in both hardware [60] and non-parallel software [27].

Although the pure narrowphase approach is highly effective, we augment it with a "mid-phase" check. Specifically: we employ a hierarchy of increasingly refined sphere collision models of the robot, and use coarse levels of this hierarchy to avoid expensive checks at the finer levels. We generate a sphere model for the robot with a single sphere per link, conservatively over-approximating the actual collision geometries. If this sphere does not collide with a given obstacle, we know that the actual collision geometry cannot collide with that obstacle, and can skip checking the spheres of the higher-fidelity model. Notably, this does not require the typical branching-heavy approaches to broadphase collision detection, *e.g.*, bounding volume hierarchies—our approach does not require or use the typical recursive tree structure or any update operations beyond FK.

For efficiency, it is preferable to not compute poses for any of a robot's links that come *after* a link in collision. We exploit a property of our tracing compiler, which can re-order instructions topologically, to *interleave* each sphere's collision check (environment and self-collision) within the generated FK code, placing checks immediately after the position of the sphere has been computed, wasting almost no effort on irrelevant FK computation and achieving a significant performance gain. This interleaving is compatible with the previously-described hierarchical sphere tree.

E. Bringing it Together: Design of the Planner

We also leverage SIMD instructions elsewhere to improve planner performance. Although not required in general, we assume the configuration space of the robot is Euclidean and thus linear interpolation between two AoS configurations becomes simply adding and multiplying their vectors together. Similarly, the ℓ_2-norm is computed efficiently as a horizontal summation. We use these improvements to quickly compute the intermediate configurations used in the rake as well as distances in our NN data-structure [51, 69].

We have implemented two SBMPs: RRT-Connect [1] and PRM [5], without algorithmic changes or additional complexity due to our focus on planner primitives.We have also implemented simplification algorithms: randomized shortcutting [70, 71] and B-spline smoothing [72].

IV. Experiments

We evaluate our approach against two baselines which use the Open Motion Planning Library (OMPL) [17]: MoveIt [73]

[5]In reality, for, *e.g.*, motion toward objects, this distribution is not uniform.

[6]This is a slight simplification; vector operations have low but nonzero overhead, and using them as we do may prevent auto-vectorization.

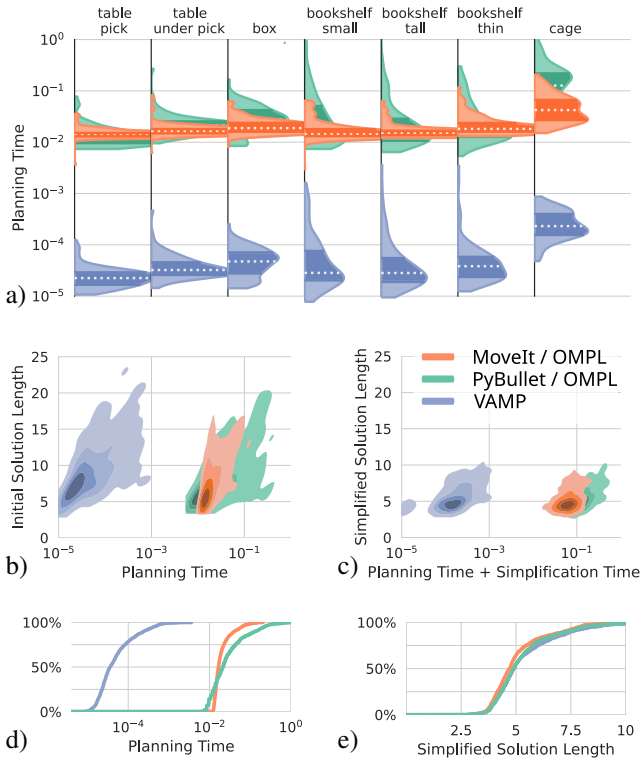

Fig. 3. Results for the 7 DoF Panda. **a)** Planning times for each problem class. **b)** Planning time vs. initial path length and **c)** planning and simplification time vs. simplified path length for entire dataset. **d)** Cumulative distribution of planning time and **e)** cumulative distribution of simplified path length for entire dataset. All times are on a **logarithmic** scale.

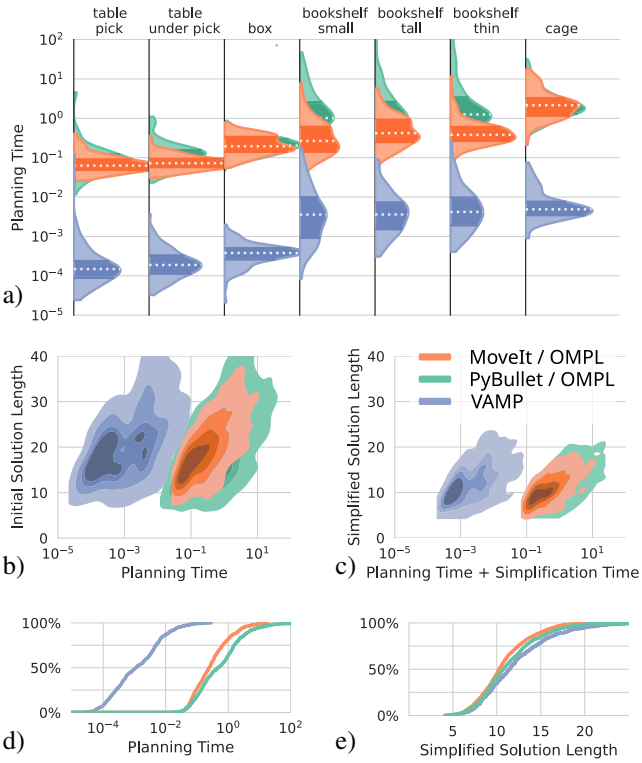

Fig. 4. Results for the 8 DoF Fetch. **a)** Planning times for each problem class. **b)** Planning time vs. initial path length and **c)** planning and simplification time vs. simplified path length for entire dataset. **d)** Cumulative distribution of planning time and **e)** cumulative distribution of simplified path length for entire dataset. All times are on a **logarithmic** scale.

	System	Mean	Q1	Median	Q3	95%	Mean Simpl.	Succ.
Panda	PyBullet/OMPL	58.47	12.76	21.21	43.35	224.41	121.38	99.6%
	MoveIt/OMPL	23.51	14.28	16.58	22.54	63.08	46.95	**100%**
	VAMP (ARM)	0.70	0.18	0.29	0.62	2.73	1.03	**100%**
	VAMP	**0.10**	**0.02**	**0.04**	**0.08**	**0.43**	**0.12**	**100%**
Fetch	PyBullet/OMPL	3035.83	135.88	461.48	1544.34	10358.44	225.17	99.8%
	MoveIt/OMPL	788.68	94.11	243.80	666.56	3107.80	137.14	**100%**
	VAMP (ARM)	28.24	1.72	5.76	23.25	116.74	3.93	99.5%
	VAMP	**6.25**	**0.25**	**1.12**	**4.92**	**25.16**	**0.52**	99.5%
Baxter	PyBullet/OMPL	4309.19	881.31	1500.44	2711.61	13100.54	938.62	99.3%
	MoveIt/OMPL	668.80	192.23	362.34	757.69	2267.35	96.81	**100%**
	VAMP (ARM)	32.54	8.02	13.93	25.89	104.57	9.77	**100%**
	VAMP	**6.68**	**1.22**	**2.32**	**4.68**	**22.97**	**1.22**	**100%**

Table I. Planning time for RRT-Connect from Figs. 3 to 5. The mean, first quantile, median, third quantile, and 95% quantile are shown, along with mean simplification time and success rate. All times are in **milliseconds**.

	System	Mean	Q1	Median	Q3	95%	Succ.
Panda	PyBullet/OMPL	4481.66	127.63	328.73	1328.37	14183.90	99.1%
	MoveIt/OMPL	615.58	415.14	416.73	418.23	1116.90	96.2%
	VAMP	**11.12**	**0.16**	**0.37**	**1.69**	**39.39**	**99.7%**
Fetch	PyBullet/OMPL	36422.87	2417.08	13060.40	38550.10	174517.60	71.3%
	MoveIt/OMPL	4514.73	468.71	1037.15	3000.64	23895.72	85.7%
	VAMP	**337.23**	**7.69**	**30.09**	**181.73**	**2014.40**	**94.7%**

Table II. Planning times for PRM over problem classes *table pick*, *table under pick*, and *box*. The mean, first quantile, median, third quantile, 95% quantile, and success rate are shown. All times are in **milliseconds**.

through Robowflex [74] (MoveIt/OMPL) and OMPL's Python bindings with PyBullet [25] (PyBullet/OMPL). These represent two common interfaces of motion planning in practice: the standard motion planner for ROS [75], and a Python implementation using a popular simulation framework. We evaluate our implementation, "Vector Accelerated Motion Planning" on an x86-based desktop computer (VAMP) as well as a small ARM-based single-board computer (VAMP (ARM))[7]. All hyperparameters are shared between each implementation: all planners use equivalent implementations of algorithms with identical validity checking resolution. Moreover, we determinize all planners by sampling from a multi-dimensional Halton sequence [76–78]. The same sequence is used between all systems. Thus, performance differences can be attributed to vector-acceleration[8].

All benchmarks for MoveIt/OMPL, PyBullet/OMPL, and VAMP were performed with a AMD Ryzen™ 9 7950X CPU clocked at 4.5GHz. For VAMP (ARM), benchmarks were run on an Orange Pi 5B with an ARM Cortex-A76 CPU clocked at 2.4GHz. Our approach is implemented in C++17 with Python bindings through `nanobind` [79]. All code (including OMPL and MoveIt) was compiled using `clang 15.0.7` with the `-Ofast` optimization level[9] and with all architecture

[7]For VAMP, AVX2 was used. The authors attempted using AVX-512, but found lower throughput than AVX2, possibly due to downclocking, lack of 512-bit registers, or other issues that will be investigated in future work. For VAMP (ARM), ARM's Neon SIMD instructions were used.

[8]MoveIt/OMPL uses the ℓ_1 metric for nearest neighbors rather than ℓ_2.

[9]Note that some of the issues with `-ffast-math`, *e.g.*, handling non-finite values, subnormals, etc., are not particularly relevant for the motion planning case, where configurations are from a compact, closed, and bounded space with relatively similar range in each dimension. However, we warn practitioners to still be wary of issues arising from reciprocal approximation.

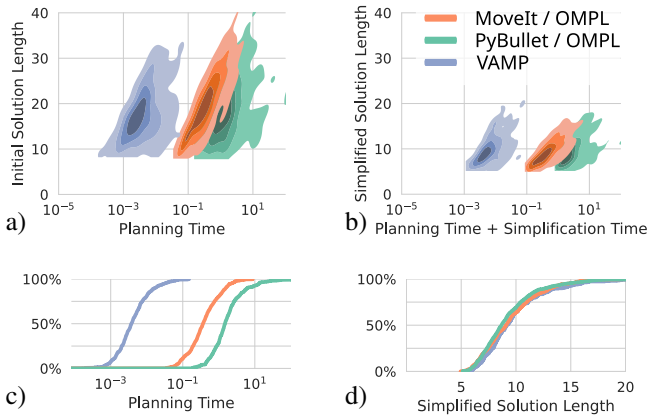

Fig. 5. Results for the 14 DoF Baxter over entire dataset. **a)** Planning time vs. initial path length. **b)** Planning time plus simplification time vs. simplified path length. **d)** Cumulative distribution of planning time and **e)** cumulative distribution of simplified path length. All times are on a **logarithmic** scale.

optimizations (`-march=native`, *i.e.*, `znver4`).

We evaluate on seven different environments from the MotionBenchMaker [16] dataset, a collection of realistic, difficult motion planning problems: 1) *table pick* and *table under pick* environments to evaluate tabletop manipulation, 2) *bookshelf small*, *tall*, and *thin* to demonstrate reaching, and 3) *box* and *cage* to demonstrate highly constrained reaching. We use the publicly available pre-generated 100 problems for each of the environments. We evaluate on the following systems: 1) the 7 DoF Franka Emika Panda[10], 2) the 8 DoF Fetch Robotics Fetch, including the prismatic torso joint, and 3) the 14 DoF bimanual Rethink Robotics Baxter. For the Baxter, we use the *bookshelf tall* {*easy, medium, hard*} datasets for bimanual manipulation. For Fetch and Baxter, we use the algorithm of Bradshaw and O'Sullivan [65] to automatically generate a spherized model (after ensuring manifold meshes [80, 81])—these approximations were also manually tuned. We evaluate each planner on each problem 5 times. For **MoveIt/OMPL** and **PyBullet/OMPL**, we give a timeout of 5 minutes. For **VAMP** and **VAMP (ARM)**, we give a limit of 1 million planner iterations.

Results for RRT-Connect on the Panda, Fetch, and Baxter are respectively shown in Figs. 3 to 5 and summarized in Table I. We note the following general features of these plots: 1) times are all reported on a logarithmic scale, 2) the distribution of planning time for **VAMP** is almost completely separated from both **PyBullet/OMPL** and **MoveIt/OMPL**, and 3) the distribution shapes of planning time versus path length are qualitatively similar and simplified path length distributions are equivalent for each planner, indicating planner similarity at the algorithmic level. Over all robots, **VAMP** is roughly 500x faster than **PyBullet/OMPL** and 100 to 200x faster than **MoveIt/OMPL**, while achieving similar path quality. **VAMP** provides high-quality plans at control frequencies, *e.g.*, 10 kHz mean, 25 kHz median, and 2.3 kHz 95% planning rates for the Panda arm for the entire dataset, which includes trivial problems such as tabletop manipulation and complex problems such as reaching into shelves. Note that even the slowest number in Table I, 25 ms for the Fetch's 95% quantile, achieves a 40 Hz planning rate. Morever, **VAMP (ARM)** also achieves similar speed-ups on a low-power single-board computer (the Orange Pi 5B uses up to 7 W), still 20–50x faster than baselines on a desktop CPU. We also report times for PRM for the Panda and Fetch over the *table pick*, *table under pick*, and *box* environments (Table II), and show the same caliber of performance improvements, indicating our approach generalizes across SBMPs.

[10]We use the approximation of the Panda from Fishman et al. [15]

V. Discussion

Efficient motion planning is critical for many applications of robotics. In this paper, we demonstrate a novel approach to accelerating motion planning, based on a new perspective on *vector-oriented* operations for simple, high-frequency interleaving of high-performance parallelized and serial sections of code present in most sampling-based motion planning algorithms. By applying this perspective to the most expensive and ubiquitous motion planning subroutines (*i.e.*, collision checking, forward kinematics, and distance computation), we achieve algorithmic improvements and create proof-of-concept planners that achieve more than 500x speedup over the state of the art on realistic, challenging planning problems for three different robots. Our approach solves planning problems at kilohertz rates on ordinary consumer CPUs and low-power single-board computers.

We also believe that our ideas will extend naturally to harder motion planning problems, such as kinodynamic and manifold-constrained planning. Further, because we can produce so many motion plans so fast, we may be able to efficiently provide empirical proofs of *solution nonexistence*, a feat that has long been challenging for SBMP. There may also be potential for using our vector-oriented planners as *local planners* inside higher-level motion planning algorithms.

The planning performance demonstrated in this work pushes SBMP for high-DoF manipulators to frequencies required for control—providing a complete, global, high-quality plan at each update. We believe that this is cause to re-examine old assumptions in robotics about the roles of planning and control, as well as about the "best" way to solve problems such as planning under uncertainty or integrated task and motion planning. In particular, we are excited to explore extensions of this work around, *e.g.*, rapid replanning, integrated task and motion planning, etc. In general, there is a rich discussion to be had about implications for algorithms that use motion planning as a subroutine, and that have traditionally needed to be designed around motion planning as an *expensive* subroutine, now that we can consistently provide high-quality motion plans at high frequencies.

Acknowledgements

The authors would like to thank Mark Moll for help evaluating MoveIt, Sofia Paola Medina-Chica for ARM development, and Stefan Bukorovic for collision primitive development.

REFERENCES

[1] 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.

[2] H. Choset, K. M. Lynch, S. Hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, and S. Thurn. *Principles of robot motion: theory, algorithms, and implementations*. MIT press, 2005.

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

[4] L. E. Kavraki and S. M. LaValle. "Motion planning". In: *Springer Handbook of Robotics*. Springer, 2016, pp. 139–162.

[5] 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.

[6] 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.

[7] 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.

[8] W. C. Agboh and M. R. Dogar. "Real-Time Online Re-Planning for Grasping Under Clutter and Uncertainty". In: *IEEE-RAS International Conference on Humanoid Robots*. 2018, pp. 1–8.

[9] 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.

[10] 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.

[11] 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.

[12] C. Park, J. Pan, and D. Manocha. "Real-Time Optimization-Based Planning in Dynamic Environments Using GPUs". In: *IEEE International Conference on Robotics and Automation*. 2013, pp. 4090–4097.

[13] 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.

[14] 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.

[15] A. Fishman, A. Murali, C. Eppner, B. Peele, B. Boots, and D. Fox. "Motion Policy Networks". In: *Proceedings of The 6th Conference on Robot Learning*. Ed. by K. Liu, D. Kulic, and J. Ichnowski. Vol. 205. Proceedings of Machine Learning Research. PMLR, Dec. 2023, pp. 967–977.

[16] 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.

[17] 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.

[18] 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.

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

[20] M. Kleinbort, O. Salzman, and D. Halperin. "Collision Detection or Nearest-Neighbor Search? On the Computational Bottleneck in Sampling-based Motion Planning". In: *Algorithmic Foundations of Robotics*. Ed. by K. Goldberg, P. Abbeel, K. Bekris, and L. Miller. Cham: Springer International Publishing, 2020, pp. 624–639.

[21] C. Ericson. *Real-time collision detection*. CRC Press, 2004.

[22] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. "Fast BVH construction on GPUs". In: *Computer Graphics Forum*. Vol. 28. 2. Wiley Online Library. 2009, pp. 375–384.

[23] T. Tan, R. Weller, and G. Zachmann. "SIMDop: SIMD Optimized Bounding Volume Hierarchies for Collision Detection". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2019, pp. 7256–7263.

[24] 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.

[25] E. Coumans and Y. Bai. *PyBullet, a Python module for physics simulation for games, robotics and machine learning*. http://pybullet.org. 2016–2021.

[26] G. Sánchez and J.-C. Latombe. "A single-query bi-directional probabilistic roadmap planner with lazy collision checking". In: *International Symposium on Robotics Research*. Springer. 2003, pp. 403–417.

[27] R. Bohlin and L. E. Kavraki. "Path planning using lazy PRM". In: *IEEE International Conference on Robotics and Automation*. Vol. 1. IEEE. 2000, pp. 521–528.

[28] N. Haghtalab, S. Mackenzie, A. Procaccia, O. Salzman, and S. Srinivasa. "The provable virtue of laziness in motion planning". In: *International Conference on Automated Planning and Scheduling*. Vol. 28. 2018, pp. 106–113.

[29] C. L. Nielsen and L. E. Kavraki. "A two level fuzzy PRM for manipulation planning". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Vol. 3. IEEE. 2000, pp. 1716–1721.

[30] S. Choudhury, S. Srinivasa, and S. Scherer. "Bayesian active edge evaluation on expensive graphs". 2017. arXiv: 1711.07329 [cs.RO].

[31] N. Das and M. Yip. "Learning-based proxy collision detection for robot motion planning applications". In: *IEEE Transactions on Robotics* 36.4 (2020), pp. 1096–1114.

[32] 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.

[33] 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.

[34] D. Rakita, B. Mutlu, and M. Gleicher. "RelaxedIK: Real-time Synthesis of Accurate and Feasible Robot Arm Motion". In: *Robotics: Science and Systems*. Vol. 14. Pittsburgh, PA. 2018, pp. 26–30.

[35] M. Koptev, N. Figueroa, and A. Billard. "Neural Joint Space Implicit Signed Distance Functions for Reactive Robot Manipulator Control". In: *IEEE Robotics and Automation Letters* 8.2 (Feb. 2023), pp. 480–487.

[36] J. Barraquand and J.-C. Latombe. "A Monte-Carlo algorithm for path planning with many degrees of freedom". In: *IEEE International Conference on Robotics and Automation*. IEEE. 1990, pp. 1712–1717.

[37] D. Henrich. "Fast motion planning by parallel processing—a review". In: *Journal of Intelligent and Robotic Systems* 20 (1997), pp. 45–69.

[38] N. A. Wedge and M. S. Branicky. "On heavy-tailed runtimes and restarts in rapidly-exploring random trees". In: *AAAI Conference on Artificial Intelligence*. Citeseer. 2008, pp. 127–133.

[39] B. Raveh, A. Enosh, and D. Halperin. "A little more, a lot better: Improving path quality by a path-merging algorithm". In: *IEEE Transactions on Robotics* 27.2 (2011), pp. 365–371.

[40] 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.

[41] S. Xiao, N. Bergmann, and A. Postula. "Parallel RRT* architecture design for motion planning". In: *International Conference on Field Programmable Logic and Applications*. IEEE. 2017, pp. 1–4.

[42] 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.

[43] S. Mukherjee, S. Aine, and M. Likhachev. "ePA*SE: Edge-based Parallel A* for Slow Evaluations". In: *International Symposium on Combinatorial Search*. Vol. 15. 1. 2022, pp. 136–144.

[44] S. Mukherjee, S. Aine, and M. Likhachev. "MPLP: Massively Parallelized Lazy Planning". In: *IEEE Robotics and Automation Letters* 7.3 (2022), pp. 6067–6074.

[45] Y. Zhou and J. Zeng. "Massively Parallel A* Search on a GPU". In: 29.1 (), p. 7.

[46] A. Fukunaga, A. Botea, Y. Jinnai, and A. Kishimoto. "A Survey of Parallel A*". 2017. arXiv: 1708.05296 [cs.AI].

[47] A. Fukunaga, A. Botea, Y. Jinnai, and A. Kishimoto. "Parallel A* for State-Space Search". In: *Handbook of Parallel Constraint Reasoning*. Ed. by Y. Hamadi and L. Sais. Springer International Publishing, 2018, pp. 419–455.

[48] 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.

[49] Intel®. *Intel® 64 and IA-32 architectures software developer's manual*. Tech. rep. Intel®, 2023.

[50] M. Kameyama, T. Amada, and T. Higuchi. "Highly parallel collision detection processor for intelligent robots". In: *IEEE Journal of Solid-State Circuits* 27.4 (1992), pp. 500–506.

[51] J. Ichnowski and R. Alterovitz. "Motion planning templates: A motion planning framework for robots with low-power CPUs". In: *IEEE International Conference on Robotics and Automation*. IEEE. 2019, pp. 612–618.

[52] J. Carpentier, G. Saurel, G. Buondonno, J. Mirabel, F. Lamiraux, O. Stasse, and N. Mansard. "The Pinocchio C++ library: A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives". In: *IEEE/SICE International Symposium on System Integration*. IEEE. 2019, pp. 614–619.

[53] B. Ichter, E. Schmerling, and M. Pavone. "Group Marching Tree: Sampling-Based Approximately Optimal Motion Planning on GPUs". In: *IEEE International Conference on Robotic Computing*. IEEE. 2017, pp. 219–226.

[54] R. C. Lawson, L. Wills, and P. Tsiotras. "GPU Parallelization of Policy Iteration RRT#". 2020. arXiv: 2003.04920 [cs.RO].

[55] P. Hyatt, C. S. Williams, and M. D. Killpack. "Parameterized and GPU-Parallelized Real-Time Model Predictive Control for High Degree-of-Freedom Robots". 2020. arXiv: 2001.04931 [eess.SY].

[56] Z. Wan, B. Yu, T. Y. Li, J. Tang, Y. Zhu, Y. Wang, A. Raychowdhury, and S. Liu. "A survey of FPGA-based robotic computing". In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 48–74.

[57] S. Murray, W. Floyd-Jones, Y. Qi, D. Sorin, and G. Konidaris. "Robot Motion Planning on a Chip". In: *Robotics: Science and Systems*. AnnArbor, Michigan, June 2016.

[58] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin. "The microarchitecture of a real-time robot motion planning accelerator". In: *IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2016, pp. 1–12.

[59] S. Murray. "Accelerated Motion Planning Through Hardware/Software Co-Design". PhD thesis. Duke University, 2019.

[60] D. Shah, N. Yang, and T. M. Aamodt. "Energy-Efficient Realtime Motion Planning". In: *International Symposium on Computer Architecture*. New York, NY, USA: ACM, June 17, 2023, pp. 1–17.

[61] S. M. Neuman, B. Plancher, T. Bourgeat, T. Tambe, S. Devadas, and V. J. Reddi. "Robomorphic Computing: A Design Methodology for Domain-Specific Accelerators Parameterized by Robot Morphology". In: *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2021, pp. 674–686.

[62] B. v. Werkhoven, J. Maassen, F. Seinstra, and H. Bal. "Performance Models for CPU-GPU Data Transfers". In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2014, pp. 11–20.

[63] H. Bruyninckx. "Open robot control software: the OROCOS project". In: *IEEE International Conference on Robotics and Automation*. Vol. 3. IEEE. 2001, pp. 2523–2528.

[64] E. Gilbert, D. Johnson, and S. Keerthi. "A Fast Procedure for Computing the Distance between Complex Objects in Three Space". In: *IEEE International Conference on Robotics and Automation*. Vol. 4. Mar. 1987, pp. 1883–1889.

[65] 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.

[66] C. O'Sullivan and J. Dingliana. "Real-Time Collision Detection and Response Using Sphere-Trees". In: *Spring Conference on Computer Graphics* 1 (Apr. 1999).

[67] A. Mandalika, S. Choudhury, O. Salzman, and S. Srinivasa. "Generalized lazy search for robot motion planning: Interleaving search and edge evaluation via event-based toggles". In: *International Conference on Automated Planning and Scheduling*. Vol. 29. 2019, pp. 745–753.

[68] M. P. Strub and J. D. Gammell. "Adaptively informed trees (AIT*) and effort informed trees (EIT*): Asymmetric bidirectional sampling-based path planning". In: *The International Journal of Robotics Research* 41.4 (2022), pp. 390–417.

[69] J. Ichnowski and R. Alterovitz. "Concurrent nearest-neighbor searching for parallel sampling-based motion planning in SO(3), SE(3), and Euclidean spaces". In: *Algorithmic Foundations of Robotics*. Springer. 2020, pp. 69–85.

[70] R. Geraerts and M. H. Overmars. "Creating high-quality paths for motion planning". In: *The International Journal of Robotics Research* 26.8 (2007), pp. 845–863.

[71] K. Hauser and V. Ng-Thow-Hing. "Fast smoothing of manipulator trajectories using optimal bounded-acceleration shortcuts". In: *IEEE International Conference on Robotics and Automation*. IEEE. 2010, pp. 2493–2498.

[72] J. Pan, L. Zhang, and D. Manocha. "Collision-free and smooth trajectory computation in cluttered environments". In: *The International Journal of Robotics Research* 31.10 (2012), pp. 1155–1175.

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

[74] Z. Kingston and L. E. Kavraki. "Robowflex: Robot motion planning with MoveIt made easy". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2022, pp. 3108–3114.

[75] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[76] J. H. Halton. "On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals". In: *Numerische Mathematik* 2 (1960), pp. 84–90.

[77] S. M. LaValle, M. S. Branicky, and S. R. Lindemann. "On the relationship between classical grid search and probabilistic roadmaps". In: *The International Journal of Robotics Research* 23.7-8 (2004), pp. 673–692.

[78] D. Hsu, J.-C. Latombe, and H. Kurniawati. "On the probabilistic foundations of probabilistic roadmap planning". In: *International Symposium on Robotics Research*. Springer. 2007, pp. 83–97.

[79] W. Jakob. *nanobind: tiny and efficient C++/Python bindings*. https://github.com/wjakob/nanobind. 2022.

[80] J. Huang, H. Su, and L. Guibas. "Robust watertight manifold surface generation method for shapenet models". 2018. arXiv: 1802.01698 [cs.CG].

[81] J. Huang, Y. Zhou, and L. Guibas. "ManifoldPlus: A Robust and Scalable Watertight Manifold Surface Generation Method for Triangle Soups". 2020. arXiv: 2005.11621 [cs.GR].