Motion Planning Templates:
A Motion Planning Framework for Robots with Low-power CPUs

Jeffrey Ichnowski* and Ron Alterovitz*

***Abstract*— Motion Planning Templates (MPT) is a C++ template-based library that uses compile-time polymorphism to generate robot-specific motion planning code and is geared towards eking out as much performance as possible when running on the low-power CPU of a battery-powered small robot. To use MPT, developers of robot software write or leverage code specific to their robot platform and motion planning problem, and then have MPT generate a robot-specific motion planner and its associated data-structures. The resulting motion planner implementation is faster and uses less memory than general motion planning implementations based upon runtime polymorphism. While MPT loses runtime flexibility, it gains advantages associated with compile-time polymorphism—including the ability to change scalar precision, generate tightly-packed data structures, and store robot-specific data in the motion planning graph. MPT also uses compile-time algorithms to resolve the algorithm implementation, and select the best nearest neighbor algorithm to integrate into it. We demonstrate MPT's performance, lower memory footprint, and ability to adapt to varying robots in motion planning scenarios on a small humanoid robot and on 3D rigid-body motions.**

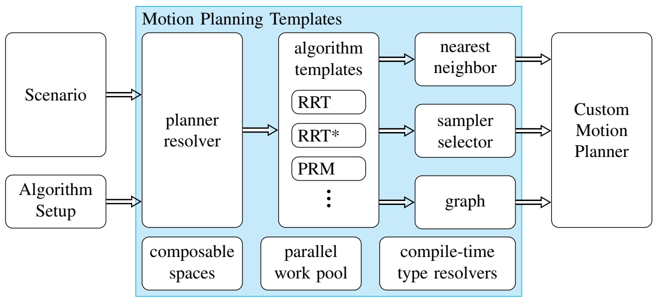


Fig. 1. The process flow of Motion Planning Templates (MPT) starts with a developer supplying a robot's motion planning problem scenario and selecting an algorithm setup. At compile time, the template system of MPT generates code for a robot-specific implementation of a motion planning algorithm. This system trades off runtime flexibility (algorithms and their data structures cannot be changed without recompiling) in favor of improved performance and reduced memory utilization, both of which are critical to battery-powered small robots that use their on-board low-power CPU to perform motion planning.

I. Introduction

Planning motions for battery-powered robots with many degrees of freedom using their on-board computers is often a difficult proposition. It is first made difficult by the computationally demanding nature of the general motion planning problem [1], which involves computing a sequence of robot actions that take the robot to a goal state while avoiding obstacles and satisfying task-specific constraints. The difficulty is then compounded when the robot's size is measured in the tens of centimeters, as its form factor and battery-life constraints only allow for low-power CPUs. While a wealth of planning algorithms aim to address the first problem [2], the latter problem is typically left as an implementation detail requiring developers to write fast robot-specific code. To address this issue, we introduce Motion Planning Templates (MPT)[1], a system that generates robot-specific code from set of motion planning algorithms.

The key philosophy behind MPT is that it *generates* robot-specific motion planning code. This means that a software developer writes code specific to the robot and the scenario, and then MPT generates the code and data structures for a custom implementation of a motion planning algorithm. The resulting implementation will have performance competitive with hand-written implementations of the same motion-planning algorithm that use robot-specific data structures. The system behind MPT's code generation is C++ templates which is a Turing-complete [3] compile-time polymorphic system—which is a fancy way of saying that C++ templates are programs that write code.

In order to eke out as much performance as possible from low-power embedded processors, MPT is also multi-core ready. Low-power processors have supported hardware-level concurrency for many years. This parallelism can be exploited in a complete robot system to allow robots to take on multiple computational tasks simultaneously (e.g., sensor processing, actuation, etc.) or to tackle computationally demanding tasks such as motion planning. As available parallelism and demands on computation can vary from robot to robot, MPT can be set to use as little or as much parallelism as desired. When parallelism is enabled, MPT's parallelized motion planning algorithms make use of concurrent data structures for nearest neighbors searching [4] and motion planning graphs. But concurrent data structures do not come for free—in order to ensure correct operation, they must use locks and ordered memory operations [5] that can result in decreased per-thread performance and increased memory usage. When parallelism is disabled, MPT generates code without locks or ordered memory operations, to maximize single-threaded performance.

This paper presents MPT, the design principles behind it, background on its compile-time polymorphic system, how to use it, and examples from applications in our own lab using low-powered processors that one finds, or might find, in small battery-powered robots.

*Jeffrey Ichnowski and Ron Alterovitz are with the Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599, U.S.A. {jeffi,ron}@cs.unc.edu

[1]MPT is available at https://robotics.cs.unc.edu/mpt

A. *Design Principles*

The design principles behind MPT help differentiate it from related and complementary libraries. This section describes those principles.

1) Performance over runtime flexibility: MPT started with the design decision that performance of robot-specific motion planners in small battery-powered robots is more important than runtime flexibility. For example, an articulated robot does not need the flexibility to compute motion plans for a wheeled robot or aerial drone. Thus MPT uses compile-time algorithms to generate robot-specific motion planners instead of using a flexible runtime system.

2) Floating-point precision selection: Robots with low-power CPUs may have performance and memory requirements that benefit from using single-precision (32-bit) floating-point arithmetic. Conversely, some robots must plan motions with accuracy and thus require double-precision (64-bit) arithmetic or better. MPT allows the selection of floating point precision at compile-time.

3) Custom state and trajectory data types: Motion planners must inter-operate with other robot software components, and thus MPT should generate and operate on graph structures with robot-specific data types that do not require runtime translation. For example, when using a robot's built-in software to send motion commands to the actuators, MPT can directly use the robot's built-in data types when computing its motion planning graphs and storing the plan, creating added efficiency.

4) (De-)Composable Metric Spaces: Some motion planners (e.g., KPIECE [6]) and nearest neighbor data structures (e.g., kd-trees [7], [8]) benefit from the ability to decompose the state space into its constituent components. Complex metric state spaces in MPT can be composed from simpler metric spaces and decomposed at compile-time to select and construct state-space specific implementations of motion planners and data structures.

5) Multi-core Ready: CPUs are trending towards increased multi-core parallelism. However, many low-power CPUs are still single-core, and robots with multi-core CPUs may only wish to use a single-core for motion planning. Since multi-core parallelism requires additional overhead and is not always necessary, MPT can switch between generating multi-core parallel and single-core planners.

6) C++ 17 Header-only Library: The latest C++ standard [9] provides a wealth of capabilities that eases development of template-based programs while remaining compatible with existing C and C++ software libraries. A header-only library means that none of the code is compiled until an application makes use of it, which can ease deployment.

B. *Related Work*

The Open Motion Planning Library (OMPL) [10] is an actively developed, well-maintained, and popular motion planning library. It implements a wide variety of motion planning algorithms using an architecture that allows for maximum flexibility at runtime. The architecture is based upon virtual classes and methods which are popular and well-studied, thus OMPL provides many with a familiar development environment and a relatively gentle learning curve. MPT does not use virtual classes and methods and is thus less flexible at runtime and instead uses templates to generate robot-specific motion planners. MPT's reliance on templates likely introduces a steeper learning curve since template-based programming is less thoroughly covered in many university courses. OMPL provides mostly single-core motion planners, with some notable multi-core ready exceptions (e.g., C-FOREST [11]). In contrast MPT supports selectable concurrency, and provides planners and frameworks for parallel multi-core motion planning. OMPL will likely be the first choice of anyone learning motion planning or exploring a specific motion problem, whereas MPT aims to replace hand-writing custom motion planners once the planning problem is understood and needs to eke out as much performance as possible on small battery-powered robots.

OpenRAVE [12] integrates motion planning, perception, and control algorithms into a runtime-configurable system. The architecture allows developers to add functionality using plugins and uses virtual classes for maximum runtime flexibility, but as a result may not perform motion planning as fast as a robot-specific planner. MPT could generate motion planners that run as OpenRAVE plugins, allowing robots to benefit from the best of both systems.

Robotics Library (RL) [13] provides a large collection of robot planning and control software in one coherent whole. This library includes a collection of sampling-based planners, including RRT [14] and PRM [15]. RL makes some use of templates but largely depends on virtual classes and methods to adapt different robot systems.

MoveIt! [16], [17] is an open-source tool for mobile manipulation built on top of ROS and OMPL. It aims to automate the setup of motion planning integrated with perception and control. MPT automates less of the motion planning setup process, but instead aims to provide greater efficiency for battery-powered small robots.

Robot Operating System (ROS) [18] is a popular software framework that aims to provide a complete system to operate a robot. It includes modules (e.g., OMPL and MoveIt!) for motion planning. MPT could similarly integrate with ROS, providing motion planners specific to the robot on which it runs and operating directly on ROS data types.

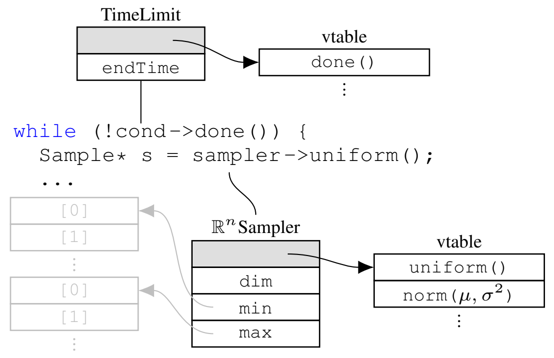
Murray et al. show that another route for low-power and fast motion plan computation is through the use of programmable circuitry [19]. But these methods require specialized hardware that is not always available on robot systems. The software-based approach of MPT aims to be compatible with readily available low-power CPUs.

II. Background

This section formally defines the motion planning problem, and provides background on tools MPT uses: compile-time polymorphism and C++ template metaprogramming.

A. *Motion Planning Problem*

Robot motion planning algorithms compute a sequence of states that takes a robot from an initial state to a goal state



```
while (!cond->done()) {
  Sample* s = sampler->uniform();
  ...
```

Fig. 2. In runtime polymorphism, calls to virtual method require a lookup into a virtual table (vtable). The vtable introduces a level of indirection that provides the flexibility to swap in different object types to get different behaviors. In this example, the time-limit termination condition can be changed by passing in a condition object with a different type. The sampler object is set at runtime to match the state space of the planner. In this example loop of a sampling-based motion planner, the termination condition and sampler, once set, rarely change. Thus the vtable lookup provides flexibility at runtime, but also introduces a repeated delay.

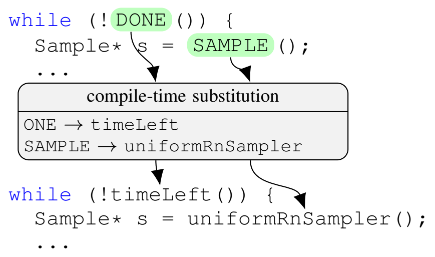
while avoiding obstacles and staying within task-specific constraints. The set of robot states is the state-space $\mathcal{X}$. Within the subset $\mathcal{X}_{\text{free}} \subseteq \mathcal{X}$, the robot does not collide with any obstacle and does not violate any constraint. Thus the input to the motion planning problem is: the initial state $\mathbf{x}_0 \in \mathcal{X}_{\text{free}}$, the set of goal states $\mathcal{X}_{\text{goal}} \subseteq \mathcal{X}_{\text{free}}$, and $\mathcal{X}_{\text{free}}$. The output is a path $\tau = (\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_n)$, where $\forall i : \mathbf{x}_i \in \mathcal{X}_{\text{free}}$, and $\mathbf{x}_n \in \mathcal{X}_{\text{goal}}$. When $\mathcal{X}$ is continuous, the output path τ must also satisfy the condition

$$\forall i \in \{1, 2, \dots, n\}, t \in [0,1] : L(t; \mathbf{x}_{i-1}, \mathbf{x}_i) \in \mathcal{X}_{\text{free}},$$

where $L(t; \mathbf{x}_\text{a}, \mathbf{x}_\text{b}) : [0,1] \to \mathcal{X}$ is a problem-specific local planner that continuously interpolates the robot's state as parameterized by two states, with $L(0; \mathbf{x}_\text{a}, \mathbf{x}_\text{b}) = \mathbf{x}_\text{a}$ and $L(1; \mathbf{x}_\text{a}, \mathbf{x}_\text{b}) = \mathbf{x}_\text{b}$. For sampling-based motion planners, it is sometimes sufficient to define $L_{\text{free}}(\mathbf{x}_\text{a}, \mathbf{x}_\text{b}) = \neg\exists t \in [0,1] : L(t; \mathbf{x}_\text{a}, \mathbf{x}_\text{b}) \notin \mathcal{X}_{\text{free}}$. Some motion planners require a distance function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ in order to operate efficiently and/or to minimize the resulting path length $\sum_{i=1}^{n} d(\mathbf{x}_{i-1}, \mathbf{x}_i)$. Some motion planners produce a motion graph $G = (V, E)$, with vertices $V \subseteq \mathcal{X}_{\text{free}}$, and each edge's vertex pair $(\mathbf{x}_i, \mathbf{x}_j) \in E$ satisfying $L_{\text{free}}(\mathbf{x}_i, \mathbf{x}_j)$.

B. Compile-time Polymorphism

Polymorphism, from the Greek meaning "many forms", refers to the ability of a single code interface to provide many different implementations [20]. In practice this means that the data and code behind a name can be changed without changing the code that refers to that name. When the executed code can be changed while the program is running, it uses *runtime polymorphism*, a concept that is likely familiar to people with experience with class-based object oriented programming in languages such as Java, Python, and C++. In runtime polymorphism, when code invokes a virtual method, it finds the the concrete implementation through a virtual table (vtable) lookup. Fig. 2 shows an example of a sampling-based motion planner's outer loop using runtime polymorphism to change its behavior. The loop continues until the `done()` method returns `true`—the exact meaning of `done()` is dependent on the `cond` object's concrete type. Similarly, the loop can work in any state space using `sampler` object of the appropriate concrete type.



```
while (!DONE()) {
  Sample* s = SAMPLE();
  ...
```

```
while (!timeLeft()) {
  Sample* s = uniformRnSampler();
  ...
```

Fig. 3. With template-based compile-time polymorphism, the compiler substitutes placeholders with direct function calls. In contrast to runtime polymorphism, flexibility to change the termination condition at runtime is lost, but execution time is sped up. The speedup comes from saving a level of indirection, and giving the compiler the ability to perform additional optimizations since it knows which code will be called—e.g., it can move simple code inline and remove the call altogether.

Compile-time polymorphism, also called static polymorphism, operates on a similar principle, but instead resolves implementations when the code is compiled, so it does not need a virtual table. Fig. 3 shows a compile-type polymorphic equivalent of Fig. 2. In this case, the behavior cannot be changed at runtime, and as a result, can run faster than the vtable-based approach. Virtual calls are an important enough performance consideration that researchers have put effort into devirtualizing calls at runtime [21]. The loss of runtime flexibility in this example is likely to be acceptable for the performance gained by the robot-specific motion planner.

C. C++ Template Metaprogramming

MPT uses compile-time polymorphism based on C++ templates. Templates are like functions that run in the compiler that take data types and constants as parameters and generate code that will be executed. Template data type parameters can be arbitrarily complex structures, which allows seemingly simple template substitutions to transitively lead to complex results—e.g. robot-specific motion planners.

C++ templates can also be *specialized* to allow for specific substitutions based upon a template parameter matching a condition. As an example, specialization can select an appropriate nearest neighbor data structure depending on whether or not the distance function is symmetric.

Templates are defined using a `template` keyword, followed by parameter declaration within < angle > brackets, followed by the class or method template. Template substitution occurs when the compiler encounters the template name followed by parameters within angle brackets.

III. Approach

This section describes MPT's design from the users' perspective. All motion planners in MPT are available through a single `mpt::Planner` template, which takes two type parameters: the *Scenario* and the *Algorithm*. The user provides the Scenario and selects the algorithm, and MPT provides

```
1  template <typename Scalar = double>
2  struct ExampleScenario {
3    using Space = mpt::SE3Space<Scalar>;
4    using State = typename Space::State;
5    using Goal = mpt::GoalState<State>;
6    using Bounds = mpt::BoxBounds<Scalar, 3>;
7
8    Space space();
9    Bounds bounds();
10   Goal goal();
11
12   bool validState(State q);
13   bool validMotion(State a, State b);
14 };
```

Listing 1. Minimal definition of a scenario

the algorithm's implementations and the building blocks to make a scenario.

A. Scenario Specification

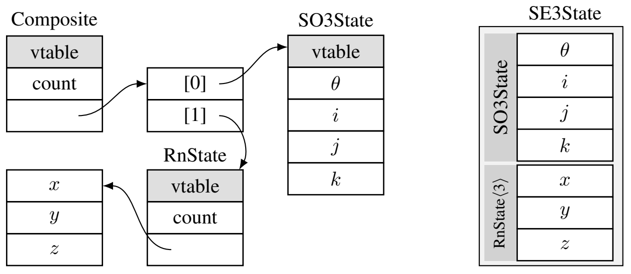
In MPT, a *Scenario* is a user-provided C++ class whose member types and methods define a robot-specific motion planning problem (i.e., $\mathcal{X}$, $\mathcal{X}_{\text{free}}$, L_{free}, etc.). To give a high-level overview of how this is done and to show some of the capabilities of MPT, we will walk through the example scenario shown in listing 1. For brevity, the listing does not include `const` and reference modifiers, nor does it include implementation code.

A scenario definition starts with the declaration of a (template) class, as shown in lines 1 and 2. There is *no* base class from which to inherit members, instead Scenario classes must conform to a few of MPT's requirements. The scenario defines the state space ($\mathcal{X}$) as a type alias called `Space` (line 3). In the example, it will plan for a robot that can translate and rotate in 3D space, and thus it uses the SE(3) state space. The `Scalar` type parameter allows the scenario to switch between single-precision and double-precision (the latter being the default). The `Space` defines both the metric and the C++ data type (more details in Sec. III-B). For SE(3), the state type is a class with a quaternion for rotation and a 3-element vector for translation (see Fig. 4 (b)). Line 4 creates an alias for the state data type used later. Since some spaces carry data members to implement their metric (e.g., a weighting components in a Cartesian space), MPT requires a `space()` method (line 8) to return a `Space` object.

Sampling-based motion planners require a mechanism to generate random states from $\mathcal{X}$. Were this class to define a `sample()` method, MPT would use it to generate samples. This scenario instead has MPT use uniform sampling by defining the sampling bounds (lines 6 and 9).

The scenario defines the problem's goal set ($\mathcal{X}_{\text{goal}}$) as a type (line 5) and method (line 10) pair. The goal type provides an indicator function that checks if a state is in $\mathcal{X}_{\text{goal}}$. Motion planners and goal types that support goal-biased sampling make use of template specialization to obtain biased samples from $\mathcal{X}_{\text{goal}}$.

The scenario defines $\mathcal{X}_{\text{free}}$ and L_{free} using the indicator functions `validState()` (line 12) and `validMotion()` (line 13) respectively. For some robots, testing $L_{\text{free}}(\mathbf{x}_a, \mathbf{x}_b)$ may require complex and time-consuming forward-kinematics



(a) Runtime construction (b) Compile-time

Fig. 4. An SE(3) state is constructed by combining and reusing state definitions for SO(3) and $\mathbb{R}^n$. Using run-time polymorphism (a) requires the composite state to carry an array of sub-states, each of which is dynamically allocated and addressed through pointers—this allows for maximum flexibility as composite states can vary in number of sub-states, and $\mathbb{R}^n$ state can vary in number of components. In contrast, compile-time polymorphism (b) defines a single composite state type at compile-time, reducing the amount of memory and objects required at runtime. In this example system, the runtime polymorphic system requires $2\times$ the memory and $5\times$ the objects of the compile-time polymorphic system.

computation of $L(\cdot; \mathbf{x}_a, \mathbf{x}_b)$. As such, it may be desirable to save the result of the computation to avoid regenerating it later. MPT detects when `validMotion()` returns something other than a boolean, and changes the motion graph definition to store the result for later retrieval. For example, given the method declaration

```
std::optional<Trajectory> validMotion(...);
```

MPT stores a `Trajectory` value in each graph edge. Similar graph-altering and planning behavior capabilities apply to changing the return type of `validState`.

B. (De-)Composable Metric Spaces

While MPT supports arbitrary data-types and metric combinations, it provides special handling for metric spaces commonly found in motion planning, including L^p (with $p \geq 1$), SO(2), SO(3), and weighted Cartesian products thereof. A mathematical metric space is an ordered combination of a set $\mathcal{X}$ and metric d. In MPT a metric space is expressed as an ordered pair of state data type (e.g., a vector of floats), and a metric tag type (e.g. L2). Using specialization, MPT provides support for a variety of common C++ data types available in the standard library and in the popular Eigen [22] linear algebra library. Using the built-in spaces allows MPT to inspect the space in order to make an informed selection of data structures and planning algorithm behaviors.

MPT allows easy setup of supported metric space to match the data types in the rest of the robot's system. For example, to use a Euclidean metric on $\mathbb{R}^3$ using a custom vector type `Vec3d`, the syntax is: `MetricSpace<Vec3d, LP<2>>`. It is also possible to create weighted Cartesian metric spaces. For example, to create an SE(3) space that combines translations in $\mathbb{R}^3$ with rotations in SO(3), the syntax is:

```
using R = MetricSpace<Quaternion, SO3>;
using T = MetricSpace<Vec3d, LP<2>>;
using SE3 = CartesianSpace<R, T>;
```

Assuming `Quaternion` and `Vec3d` are appropriately defined, the above code is equivalent to:

```
using SE3 = MetricSpace<
```

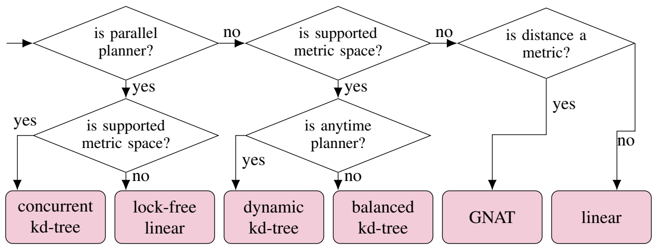


Fig. 5. Using a compile-time algorithm, MPT automatically selects and generates the planner's nearest neighbor data structure based upon the requirements of the scenario and planning algorithm.

Fig. 6. The Nao robot uses a low-power Intel Atom CPU to solve a 10 DOF motion planning problem (from [26]) that avoids obstacles in order to drop an effervescent tablet into a glass while not spilling in the process.

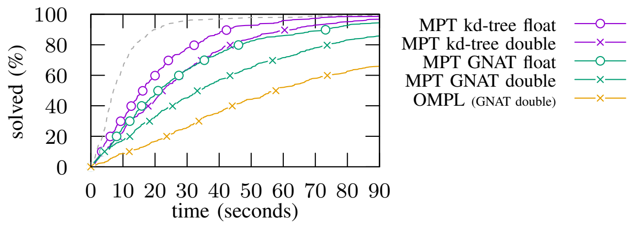


Fig. 7. **RRT probability vs. compute time**. An Intel Atom CPU computes a solution to a 10 DOF problem for the Nao robot using RRT. The compile-time polymorphism in MPT more efficiently computes samples which results in finding solutions sooner. The dashed gray line shows OMPL running on an Intel i7-7820HQ @ 2.9 GHz—showing the CPU performance difference that MPT aims to address.

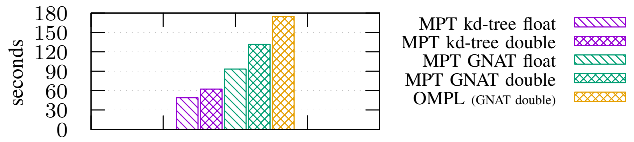


Fig. 8. **Nao computes 5 000 vertex RRT* graph** for a 10 DOF task using MPT and OMPL running on an Intel Atom processor.

```
std::tuple<Quaternion, Vec3d>,
Cartesian<SO3, L2>>;
```

The result of this construction is that the Cartesian state space is flexibly defined at compile-time and its state data type is compact at runtime. Fig. 4 (b) shows the resulting state type as it will be stored in memory. A similar flexibility is possible in a runtime-polymorphic system and is shown in Fig. 4 (a), but requires significantly more overhead since the states must be assembled as object graphs at runtime. While it is possible to avoid this overhead with a custom implementation, such an approach would lose the benefit of code reuse.

C. Nearest Neighbors

Nearest neighbor searching is a fundamental building block for many motion planning algorithms. The performance of nearest neighbor searching can dramatically affect the performance of a planning algorithm [7], [23]. MPT thus uses a *compile-time* algorithm to select and define a nearest neighbor data structure that best matches the needs of the scenario and planner. This algorithm is shown in Fig. 5.

The nearest neighbor searching data structures in MPT are: kd-tree that supports concurrent inserts and queries [4] (ideal for parallelized motion planners) and a non-concurrent variant of it, a (non-concurrent) kd-tree that maintains near optimal balance [24] at the expense of periodic rebalancing (ideal for non-parallel, long running motion planners), GNAT [25], and linear searching for custom metric and non-metric spaces. When the scenario uses an MPT-supported metric space, MPT can decompose it at compile-time to generate a custom implementation of a kd-tree.

D. Planner Algorithm Selection

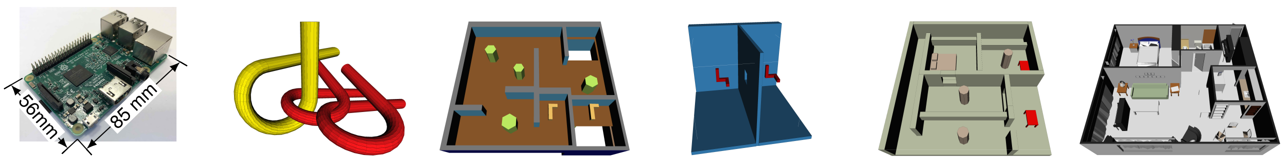
In a compile-time algorithm that is similar to, though more involved than Fig. 5, MPT uses a template argument to determine the motion planner implementation to generate. The process starts with the creation of a `mpt::Planner<Scenario, Algorithm>` object, where the `Scenario` is defined in a similar manner to Listing 1, and `Algorithm` is an MPT-provided algorithm selection tag, such as `mpt::RRT<>`. Under the hood, MPT uses a cascade of template specializations to resolve a final algorithm implementation. The planning algorithms included in MPT's initial release are parallel lock-free [26] versions of RRT [14], RRT* [27], PRM [15], PRM* [27], and IRS [28].

IV. Applications

In this section we demonstrate MPT's performance on an articulated robot and in OMPL's SE(3) rigid-body planning benchmarks. We compare to OMPL as it is an example of a well-designed flexible motion planning library that uses runtime polymorphism. To the extent possible, we set up corresponding motion planners from MPT and OMPL to run identical algorithms. The performance benefit of MPT over OMPL thus comes from the MPT's compile-time data-structure and algorithm selections, compact state representation, non-virtual methods, and affordances that allow the compiler to inline and vectorize code. This does however come at the cost of losing runtime flexibility and a potentially steeper learning curve. We run MPT with both single ("float") and double precision arithmetic. OMPL only supports double precision arithmetic. OMPL uses GNAT for nearest neighbor searching so we compare to MPT using GNAT. We also compare against MPT's automatic selection of kd-trees for nearest neighbor searching.

A. Small Humanoid Motion Planning using an Intel Atom

We use MPT to solve a 10 degree of freedom (DOF) task on a SoftBank Nao small humanoid robot shown in Fig. 6. This robot has a low-power (2 to 2.5 W) Intel Atom Z530 @ 1.6 GHz CPU. To avoid taxing our robot, we run hundreds of simulations on a more recent Atom N270 @ 1.6 GHz, noting that the CPUs perform similarly in benchmarks [30].



(a) Raspberry Pi 3 B (b) alpha-1.5 (c) cubicles (d) Twistycool/Easy (e) Home (f) Apartment

Fig. 9. The low-power CPU of a Raspberry Pi Model 3 B [29] (a) uses MPT to solve SE(3) rigid-body motion planning problems (b)–(f) from OMPL [10].

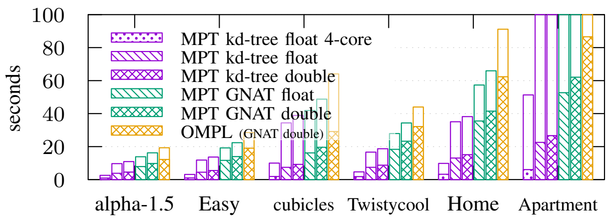


Fig. 10. **The Raspberry Pi 3 computes a 10 000 vertex RRT* graph** for SE(3) rigid body motion planning problems from OMPL [10]. To the extent possible, MPT and OMPL are set up to run identically. Collision detection, which is *not* provided by MPT or OMPL, is plotted in the unfilled blocks.

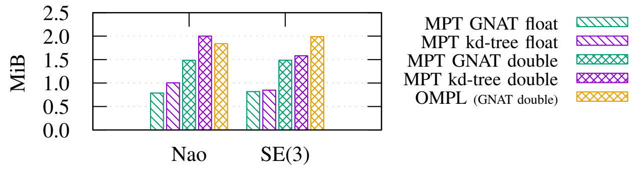


Fig. 11. **Memory usage with 10 000 RRT* graph vertices** as generated by MPT and OMPL for the Nao and SE(3) problems on 32-bit CPUs.

The Nao simulation uses an RRT [14] motion planner that terminates as soon as it finds a feasible plan. We plot the observed solution probability given the wall-clock time spent computing. As the graph in Fig. 7 shows, MPT's custom generated motion planner solves the planning problem in less than half the time of a runtime polymorphic system.

We also run the asymptotically-optimal RRT* [27] motion planner until it creates a 5 000 vertex motion graph. Over 50 runs all implementations of the planner require approximately the same number of iterations and generate paths of similar cost distribution, confirming the planners implement nearly identical algorithms. Fig. 8 shows the wall-clock time to compute the graph, showing the performance impact of having a custom generated planner, using single-precision floats, and using kd-trees for nearest neighbor.

B. Rigid Body Motion Planning using a Raspberry Pi

We use a Raspberry Pi 3 Model B v 1.2 (Fig 9 (a)) to compute RRT* solutions to SE(3) rigid-body planning problems from OMPL (Fig. 9 (b)–(f)). The Pi is a low-power (2 to 3 W) 4-core ARM-architecture processor which would make a suitable processor for a battery-powered small robot due to its low power consumption and small form factor. Fig. 10 shows the wall-clock time elapsed when computing a 10 000 vertex graph. In this setup, we also show the benefit of the parallelism included in MPT by running PRRT* [26], a parallelized version of RRT*, running on all 4-cores.

C. Reduced Memory Usage

We measure and compare the mean memory usage of RRT* runs from the Nao and SE(3) scenarios. The results in Fig. 11 show the impact of the compact memory representation and the choice of nearest neighbor structures. The difference between MPT's GNAT and kd-tree data structures shows that GNAT is more memory efficient. This implies some MPT users will have to choose between the speed of a kd-tree vs. the lower memory usage of GNAT. The comparison between MPT's GNAT using double-precision and OMPL shows the impact of the compact data-structures that MPT is able to generate. The difference in improvements between the Nao and SE(3) highlights the impact of compile-time state composition since the complex object graph for SE(3) states (Fig. 4) incurs more overhead than the Nao scenario's relatively simple $\mathbb{R}^{10}$ state. Finally, the figure shows the significant impact of changing floating-point precision—when the loss of precision is acceptable, MPT may enable planners to run on systems where memory usage comes at a premium.

V. Conclusions and Future Work

We presented Motion Planning Templates, a framework based upon the compile-time polymorphic system of C++ templates for building motion planners for robots with low-power CPUs. MPT's template system generates custom planning code specific to the robot and a set of tasks encompassed by a concept of a scenario.

In benchmarks on a small humanoid robot and synthetic benchmarks on rigid body motions, MPT's generated planners demonstrate better performance and lower memory usage than planners based upon runtime polymorphism. While this approach loses the flexibility of runtime polymorphism and introduces a potential learning curve to developers more familiar with runtime polymorphic systems, the trade off may be worth the cost, especially in small low-powered robots where every CPU cycle counts.

MPT is an evolving project under active development. Current plans involve adding more planners, supporting more robots with low-power CPUs, and integrating with other libraries. Compiling on low-power CPUs is slow, so we plan to build tools allow standard desktop setups to cross-compile for the target robot systems.

Acknowledgments

This research was supported in part by the U.S. National Science Foundation (NSF) under Awards IIS-1149965 and CCF-1533844.

References

[1] J. H. Reif, "Complexity of the mover's problem and generalizations," in *20th Annual IEEE Symp. on Foundations of Computer Science*, Oct. 1979, pp. 421–427.

[2] H. Choset, K. M. Lynch, S. A. Hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, 2005.

[3] T. L. Veldhuizen, "C++ templates are turing complete," Indiana University, Tech. Rep., 2003.

[4] J. Ichnowski and R. Alterovitz, "Concurrent nearest-neighbor searching for parallel sampling-based motion planning in SO(3), SE(3), and Euclidean topologies," in *Algorthmic Foundations of Robotics (Proc. WAFR 2018)*. Springer, 2018 (to appear).

[5] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2011.

[6] I. A. Şucan and L. E. Kavraki, "Kinodynamic motion planning by interior-exterior cell exploration," in *Algorithmic Foundation of Robotics VIII*. Springer, 2009, pp. 449–464.

[7] A. Yershova and S. M. LaValle, "Improving motion-planning algorithms by efficient nearest-neighbor searching," *IEEE Trans. Robotics*, vol. 23, no. 1, pp. 151–157, 2007.

[8] J. Ichnowski and R. Alterovitz, "Fast nearest neighbor search in for sampling-based motion planning," in *Algorithmic Foundations of Robotics XI*. Springer, 2015, pp. 197–214.

[9] ISO/IEC, "ISO international standard ISO/IEC 14882:2017(E)—programming languages—C++," International Organization for Standards (ISO), Geneva, Switzerland, Standard, Dec 2017.

[10] I. A. Şucan, M. Moll, and L. E. Kavraki, "The open motion planning library," *IEEE Robotics and Automation Magazine*, vol. 19, no. 4, pp. 72–82, Dec. 2012. [Online]. Available: http://ompl.kavrakilab.org

[11] M. Otte and N. Correll, "C-FOREST: Parallel shortest path planning with superlinear speedup," *IEEE Trans. Robotics*, vol. 29, no. 3, pp. 798–806, 2013.

[12] R. Diankov and J. Kuffner, "OpenRAVE: A planning architecture for autonomous robotics," *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34*, vol. 79, 2008.

[13] M. Rickert and A. Gaschler, "Robotics library: An object-oriented approach to robot applications," in *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*. IEEE, 2017, pp. 733–740.

[14] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *Int. J. Robotics Research*, vol. 20, no. 5, pp. 378–400, May 2001.

[15] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high dimensional configuration spaces," *IEEE Trans. Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[16] I. A. Şucan and S. Chitta, "Moveit!" http://moveit.ros.org, 2013.

[17] D. Coleman, I. Sucan, S. Chitta, and N. Correll, "Reducing the barrier to entry of complex robotic software: a moveit! case study," *Journal of Software Engineering for Robotics*, 2014.

[18] ROS.org, "Robot Operating System (ROS)," http://ros.org, 2012.

[19] S. Murray, W. Floyd-Jones, Y. Qi, D. J. Sorin, and G. Konidaris, "Robot motion planning on a chip." in *Robotics: Science and Systems*, 2016.

[20] B. Stroustrup, *The C++ programming language*. Pearson Education, 2013.

[21] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, "A study of devirtualization techniques for a Java just-in-time compiler," in *ACM SIGPLAN Notices*, vol. 35, no. 10. ACM, 2000, pp. 294–310.

[22] B. Jacob, G. Guennebaud, *et al.* (2018) Eigen. [Online]. Available: http://eigen.tuxfamily.org

[23] M. Kleinbort, O. Salzman, and D. Halperin, "Collision detection or nearest-neighbor search? on the computational bottleneck in sampling-based motion planning," in *Algorthmic Foundations of Robotics (Proc. WAFR 2016)*. Springer, 2016.

[24] J. L. Bentley and J. B. Saxe, "Decomposable searching problems I. static-to-dynamic transformation," *J. Algorithms*, vol. 1, no. 4, pp. 301–358, 1980.

[25] S. Brin, "Near neighbor search in large metric spaces," in *Proc. 21st Conf. on Very Large Databases (VLDB)*, Zurich, Switzerland, 1995, pp. 574–584.

[26] J. Ichnowski and R. Alterovitz, "Scalable multicore motion planning using lock-free concurrency," *IEEE Transactions on Robotics*, vol. 30, no. 5, pp. 1123–1136, 2014.

[27] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *Int. J. Robotics Research*, vol. 30, no. 7, pp. 846–894, June 2011.

[28] J. D. Marble and K. E. Bekris, "Asymptotically near optimal planning with probabilistic roadmap spanners," *IEEE Transactions on Robotics*, vol. 29, no. 2, pp. 432–444, 2013.

[29] Raspberry Pi Foundation. (2018) Raspberry Pi 3 model B. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[30] Snapsort, Inc. (2018) Intel Atom Z530 vs N270. [Online]. Available: http://cpuboss.com/cpus/Intel-Atom-Z530-vs-Intel-Atom-N270