TOWARDS EFFICIENT AND ROBUST ROBOT PLANNING

by

C. Barrett Ames

Department of Computer Science
Duke University

Date: ______________________________

Approved:

____
George Konidaris, Supervisor

____
Ron Parr

____
Leila Bridgeman

____
Michael Zavlanos

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2022

ABSTRACT

TOWARDS EFFICIENT AND ROBUST ROBOT PLANNING

by

C. Barrett Ames

Department of Computer Science
Duke University

Date: ______________________________
Approved:

____
George Konidaris, Supervisor

____
Ron Parr

____
Leila Bridgeman

____
Michael Zavlanos

An abstract of a dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2022

Abstract

In this work, three contributions are made to state-of-the-art robot planning. The contributions expand robot planning to be more efficient and robust by first expanding the mapping between task space and joint space via improved inverse kinematics. This improved mapping allows planning to be more robust by increasing the size of the goal set. Second, an algorithm for the optimization of provably stable controllers is provided. This allows the controlled system to be stable and performant. This is accomplished by extending the LQR-Trees algorithm with inspiration from Reinforcement Learning and Motion planning. Finally, a new method for constructing symbolic representations with controllers that are parameterized expands the applicability of symbolic planning to a wider set of controllers.

Contents

List of Tables

List of Figures

Acknowledgements

First and foremost I must express my gratitude to my wife, Mackenzie, and my children. Without their support, none of this would have happened. In particular, Mackenzie's ability to understand that many nights after dinner I was going right back to work. She also always understood that paper deadlines are a magical time when I forget to do everything other than program and write. She kept me and the children fed and rested, as much as one can be, during those times.

To my children, you have provided a special kind of motivation, the wonder in your eyes when you see a robot renews my own enthusiasm. This is a motivation that only a father can enjoy and it, more than anything else has kept me going.

To my Father-in-Law, Bob Wadas, thank you for moving to be near us so that you could help with the children. You've always been interested in, and supportive of, my work.

To my parents, Chris and Sharon Ames, I owe much of my progress to the strong foundation that you provided for me. You went above and beyond in your efforts and I appreciate it. I appreciate that you slept on a concrete floor so I could go to a good elementary school, that you sent me to robot camp instead of letting me blow stuff up, and the time and energy spent mentoring me in robotics. Certainly, without all of that care and attention, I would have turned out differently.

To my Advisor, George Konidaris, thank you for your unending support, whether that was getting the Duke students an apartment in Providence or asking questions when I had my latest hair-brained idea. As in many graduate careers, there were several dark moments and you didn't just highlight the light at the end of the tunnel you were always ready to craft a plan to reach a better tomorrow. I have grown a lot and it was due to the strategies and fundamentals that you provided. The space you provided allowed me to explore and understand the entire landscape of robotics. My writing, while still that of an engineer, has improved significantly. My research will forever carry the indelible mark of NAFN.

To Ron Parr, I have become a more methodical, logical researcher because of the care and effort that you demonstrated in answering my questions. Your patience with my crazy questions was commendable, but what was exceptional was the amount of time you would take to explain, from first principles, the error in my thought process. The answers were useful, but the demonstration of the process was life-changing. I hope to one day demonstrate the same patience and understanding.

To Rob Platt, for taking a risk on a young engineer and welcoming me into the field of robotics. I'll never forget the speech you gave at Del Taco in Clear Lake, Texas.

To Steve Hart, for connecting me to George and thoroughly interrogating my desire to pursue a graduate degree. I felt more prepared to tackle the challenges because of your questions and guidance.

To Patrick Beeson, for being a great collaborator, and igniting a curiosity that still exists to this day.

To Rob Burridge, your early work on controller funnels was inspirational, and working together at TracLabs changed the way that I look at my career.

To Ephrahim Garcia, your honesty and advice helped shape the early trajectory of my career.

To Benjamin Burchfiel, thanks for all the discussions about research and life. My time at Duke would have gone by a lot more slowly without your witty repartee and travel companionship.

To Missy Cummings, you're intellectual lightning in a bottle and it's great fun to get zapped.

To Cam Allen, I have always appreciated our discussions about Reinforcement Learning, life, and food. I particularly appreciated your willingness to be honest, open, and frank with me.

To Ian Abraham, thanks for agreeing to meet up completely randomly, my time spent working with you set a high bar for hard work.

To the students of ECE 383, thanks for teaching me about the life of an instructor. You all altered the course of my life.

To Mark Donahue, for being a good neighbor who asks great questions. Your demonstrated focus has always been an inspiration.

To Brent Wadas and Colin Devine, without you two I wouldn't be applying all that I learned during this dissertation to the construction industry. Your support and understanding were crucial in finishing my degree.

To Marilyn Butler, you were the heart and soul of the department. Your guidance through the bureaucracy and paperwork ensured that even with my extreme allergy to all things administrative the gears kept turning.

To Allison Thackston, you gave me a place to expand my ideas and put them to the test on real hardware.

To Kate O'Hanlon, your patience and ability to explain all things algorithmic were critical. You also taught me just how critical candy is to solving hard problems.

To Mark Nemecek, our many discussions on research and life motivated me to always push the boundaries. Thank you for always being willing to say that you didn't understand, not many people have that courage and it always helped me clarify my thoughts.

A special thanks to Miles Aubert, Jane Li, Alessio Rocchi, Ben Abbattamateo, Matt Corsaro, Adam Konnenker, Sean Murray, Will Floyd-Jones, your interactions made graduate school more enjoyable. It's tough to imagine what my time would have been like without you.

Chapter 1

Introduction

Imagine that you have just been invited to dinner with a friend across town. It usually takes you 35 minutes to arrive there, and you have 45 minutes. You may open your phone and check the traffic or you may just hop in your car and begin the drive, confident that whatever traffic appears you will find the best way around it. On the drive, you hardly pay attention to the mechanics of driving but are instead consumed with the possible future delights of the meal. A car suddenly changes lanes into your lane, and you, being a seasoned driver, respond accordingly, only momentarily distracted from your dinner dreams. As you come to your preferred exit of choice you notice that it looks rather backed up, since while being the shortest route to your destination, it also has a nasty habit of fender-benders. You quickly mentally re-route your path and arrive at your destination with 5 minutes to spare. You parallel park your car on the street opposite the restaurant. A short walk later, you are in the restaurant beginning a wonderful evening of food and conversation.

This series of events illustrates a striking disparity between how humans and robots plan and operate. As physical agents, humans are almost completely unaware of *how* we accomplish physical tasks. Even more amazing is that we plan at a level that appears to assume that we can perform the task, and a great percentage of the time it works! Robot planning, on the other hand, is extremely detailed— every joint must have a known position for every loop of the controller. In the case of walking this typically requires footstep path planning, or when driving a car this requires planning the synchronized operation of the gas, brake, and steering wheel. This level of detail generates a computational complexity that makes it infeasible to plan for all but the shortest of horizons. In order for robots to achieve their promise they must plan a higher level of abstraction, instead of planning at the joint level for every single time step, robots must be able to plan at a higher level of abstraction. In the dinner scenario, many humans would plan with very abstract actions, for example, driving, and walking. A robot that can construct plans at such a high level can reason further into the future than one that can only begin to move once it has planned every single step.

Planning is difficult for robots because of the variety of tasks that they can perform. In order for robot planning to tackle this complexity, it needs many different layers of abstraction. First, the robot needs a way to transform from planning in one space to planning in another. For example, planning how to draw an h would be easy when defined with respect to a plane, whereas the plan would be brittle if defined with respect to the joint positions of the robot. Second, when the robot plans how to draw an h it must consider its capabilities, the constraints of the task, and the environment in which it is accomplished. Once it has created this plan it can then be validated by successfully drawing an h. This validation provides conditions on the start and end states for a plan that enables the next layer of abstraction. Another way to view this is that if the valid start states and end states of a plan are known, the intermediate states can be ignored. In order to achieve this compositional quality, the planner must identify the key features that impact execution, and expose those to the next layer. Lastly, it is important to know how to improve the available controllers and their abstractions. The work presented here expands and strengthens the state of the art in robot planning by providing three new tools at three different levels of abstraction.

The first contribution is an improvement to Inverse Kinematics for robots with seven or more joints. Inverse Kinematics is a fundamental operation in robotics that provides the mapping between task space and the joint space of the robot. Many tasks, like drawing an h, are easier to define in convenient task space. However, the robot must be controlled at the level of joints because it is only at the joint level that the full configuration of the robot is defined, and ultimately it is at this level that commands are issued. Inverse Kinematics is particularly difficult in systems with 7 joints or more because there can be an infinite number of solutions. Previous approaches were not capable of quickly providing an accurate representation of the solution space. This limitation makes path planning more difficult than necessary because it unnecessarily restricts the solution space. The extension presented in the first chapter, IKFlow, quickly provides a large number of solutions. This improved mapping between task space and joint space enables more robust motion planning because it provides a large sampling of possible goals that can be reached, which inherently makes the motion planning task easier by increasing the size of the goal set.

The second contribution is at a higher level of abstraction, an improvement to the construction of sequentially composable feedback controllers. A feedback controller is composable with other feedback controllers if it has a well-defined region of attraction. A controller's region of attraction is the set of start states from which a controller is guaranteed to converge. This well-defined region of attraction provides a useful interface for planning at an abstract level. Previous work [63] accomplished the sequencing of controllers by overlapping end states of one controller with the region of attraction of another. This approach enabled a controller to be constructed by concatenating many smaller controllers together. Chapter 3 provides a new algorithm that optimizes these composable feedback controllers. This optimization enables controllers to improve performance as more controllers are created.

The final contribution is an improvement to an even higher level of abstraction for robot planning, the construction of symbols. At this level, planning takes place in an abstract space. This abstract space reduces much of the complexity that occurs in the continuous low level representation of the environment. Instead, it allows the robot to make long-term task-level decisions without considering unnecessary details for the sequencing of tasks. In order for these abstract plans to map to actions that the robot can execute, there must be a strong correspondence between the symbols used at the planning level and the capabilities of the robot. Previous work has established the necessary and sufficient conditions for this mapping, but it assumed black box controllers which are insufficient for planning with robots that have parameterized controllers. Chapter 4 provides an extension to the creation of symbols that includes parameterized controllers, while still maintaining an abstract domain description. This parameterized controller extension is demonstrated in both virtual and physical environments. All together these contributions push the state of the art in robot planning to a more robust and efficient place.

Chapter 2

Improved Inverse Kinematics for 7+-DOF Systems

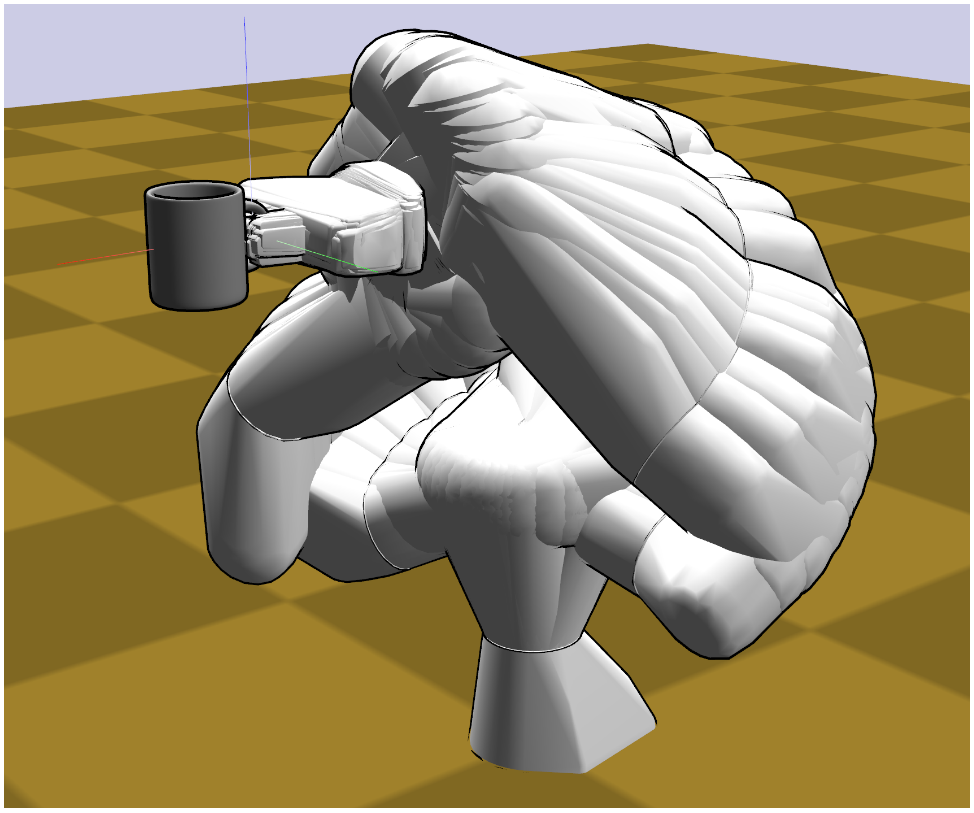


Figure 2.1: One hundred solutions generated by IKFlow, for a Panda arm reaching to a given end effector pose.

Inverse Kinematics (IK) maps a task-space Cartesian pose to a joint space configuration, which is a critically important operation for several reasons. For example, it is typically easier to define tasks in a specialized Cartesian coordinate frame that is easily interpretable by the robot's operator—for example, specifying a curve to draw on a whiteboard is more easily done in the frame of the whiteboard than in the joint space of the robot. Similarly, it is beneficial to express a grasp pose in Cartesian space because it leaves the robot free to choose among a multitude of valid joint space poses. Of course, each Cartesian-space goal must be translated to a joint pose for the robot to control to; therefore, these scenarios are only possible with a fast IK solver.

Although there are several open source IK packages [9, 67, 17], their functionality is still incomplete. Analytical solvers like IKFast [20] and IKBT [67] are fast and return all the solutions for an arm, but cannot be applied to arms with more than 6 degrees of freedom (DOF). Numerical solvers can be applied to arms with any number of joints, but only return a single solution, if one is found at all. A solver that can provide many solutions for an arm with 7-DOF or greater, and do so quickly, would add significant functionality that would support more robust robot planning and control. More specifically, it allows for multiple joint solutions to be evaluated for any particular end effector pose. This is important for grasping in cluttered environments because there may be a large number of solutions that are in collision with an object. A large number of solutions is also important for the pathwise-Inverse Kinematics problem [48]. A complete mapping between joint space and task space enables planning to take place in the task space. IK can then be used to validate the task space plan with high confidence. Planning in the task space has a smaller dimension than in the joint space. This smaller dimension is useful for learning and sampling-based approaches because fewer samples are required to cover the space with the same density.

An ideal 7+ DOF IK solver should return a set of solutions that covers the entire solution set. These samples should be near-exact solutions. Additionally, it should do so quickly. This allows the solver to serve as a primitive in the inner loop of higher-level decision-making algorithms. Of these three requirements, accuracy is the least important because verifying a sample using forward kinematics is very fast, and numerical IK solvers can be seeded with an approximate solution to rapidly refine to any desired accuracy.

We propose IKFlow, a new IK method that satisfies these requirements by training a neural network to output a diverse set of poses that approximately satisfy a given Cartesian

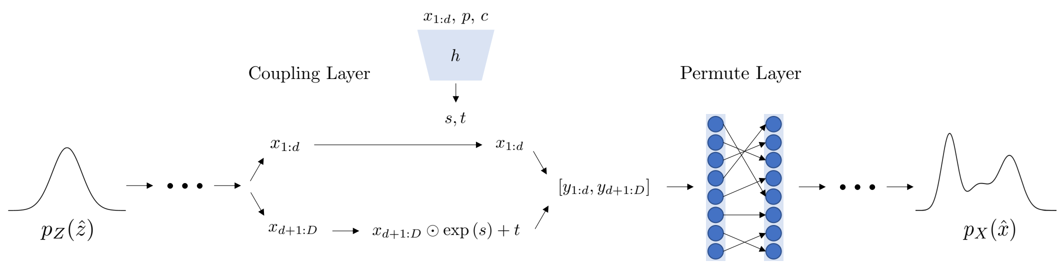


Figure 2.2: The basic architecture of a conditional normalizing flow network. The base distribution sample is fed into a coupling layer. The coupling layer contains a coefficient network that transforms a subset of the sample. The permutation changes the order of the samples so that the coefficient network of the next coupling layer affects a different subset of the sample. Conditional information is passed into every coupling layer.

goal pose. By viewing the problem as a generative modeling problem over the solution space, we are able to exploit recent advances in Normalizing Flows [51]. Normalizing Flows are a generative modeling approach capable of modeling multi-modal distributions with nonlinear interactions between variables. We show, using several different kinematic models, that IKFlow can be trained—once off, per robot —to rapidly generate hundreds to thousands of diverse approximate solutions; we also provide a practical open-source implementation of our approach.[1]

2.1 Background

2.1.1 Inverse Kinematics (IK)

Inverse Kinematics defines the mapping from a robot's operational space to its joint space:

$$f : P \to T^n,$$

where n is the number of joints. The topology of the joint space is the n-dimensional torus, $T^n = S^1 \times S^1 ... \times S^1$. The topology of P is determined by the workspace of the robot, but we focus on the case where the end effector pose is in $P \coloneqq SE(3)$. If the robot has joint

[1]`https://sites.google.com/view/ikflow`

limits then the topology of the mapping changes:

$$f : P \to R^n,$$

where R^n is the n-dimensional Euclidean space. This topological difference is important because it allows for many robots to be modeled by generic density estimators, instead of special purpose estimators built for tori [52].

IK for 7+ DOF

When the degrees of freedom of the robot exceed the degrees of freedom of the operational space (i.e. the robot is kinematically redundant) there are infinitely many solutions. This occurs because the extra degree of freedom allows for a continuum of configurations that satisfy the desired Cartesian goal. Thus for a specific pose $p \in SE(3)$ an IK solver should return a subset of joint space:

$$f : p \to R^n_g \subseteq R^n.$$

There are two current approaches for describing the solution space of 7+ DoF arms. The first category uses only a single point to describe the solution space, but returns quickly, on the order of 1 millisecond. The second approach returns a more thorough set of solution points, by running the first approach with randomly sampled start states. In order to obtain a representative set of the solution space this approach requires extensive random sampling, and thus is significantly slower, on the order of seconds for thousands of solutions. The approach we detail in Section 2.3 uses generative modeling to generate upwards of a thousand solutions in under 10 milliseconds.

2.1.2 Deep Generative Modeling

Generative modeling represents arbitrary distributions in such a way that they can be sampled. This is achieved by transforming known base distributions into target distributions. For example:

$$g : N(\hat{0}, I) \to Q,$$

where Q is an arbitrary distribution and g is a neural network. We additionally define the *latent space* as the space in which the base distribution lies - in our case R^D, where D is the dimension of the network. We draw samples from the arbitrary distribution by drawing samples from the base distribution and passing them through g. There are several different generative modeling methods. In order to select an approach, we used the following criteria. First, the diversity of samples returned is important to ensure that the full breadth of the solution space is returned. Second, the approach must be able to handle multi-modal data and nonlinear dependencies between variables in order to cover the solution space. Third, the method must be capable of handling conditional information, because the solution space is conditioned on the Cartesian goal. Fourth, the method must sample solutions quickly because IK is often used as a primitive by other procedures. Finally, the sampling procedure must produce samples with sufficient accuracy. Given these desired properties we propose to use a normalizing flow approach.

Normalizing Flows

Normalizing Flows are a generative modeling approach that provides quick sampling, stable training, and arbitrary data distribution fitting [42]. Normalizing flows are based on two components. The first component is a series of functions that are easy to invert. These easily invertible functions enable the same network to quickly estimate densities and produce samples. In the literature, this series of functions is commonly referred to as the *coupling layers*. Sampling is performed by passing a sample from the base distribution through each of the coupling layers:

$$\hat{x} = f_1 \circ f_2 ... \circ f_n(\hat{z}), \hat{z} \sim N(\hat{0}, I), \tag{2.1}$$

where $\hat{x}$ is the sample in the data space and $\hat{z}$ is a sample from the base distribution. The second important component is the efficient calculation of the determinant of the Jacobian. This enables density estimation of a data point to be computed with the change of variables formula:

$$p_X(\hat{x}) = p_Z(\hat{z}) \left| \det \left(\frac{\partial g(\hat{z})}{\partial \hat{z}^T} \right) \right|^{-1}.$$

The change of variables formula allows a distribution over one set of variables to be described by another set of variables given the determinant of the Jacobian between the two variables. A normalizing flow uses this along with a simple prior distribution, $p(z)$—here a Normal distribution—to enable density estimation of the data distribution $p(x)$. To make the determinant of the Jacobian tractable, special coupling layers are used. The coupling layer used in IKFlow was developed by Kingma and Dhariwal [33]:

$$y_{1:d} = x_{1:d} \tag{2.2}$$

$$s, t = h(x_{1:d}) \tag{2.3}$$

$$y_{d+1:D} = x_{d+1:D} \odot \exp(s) + t, \tag{2.4}$$

where x is the input data to a layer, and y is the output. s affects the scaling of the layer, t shifts the input of the layer, and $d = \lfloor D/2 \rfloor$. This layer is then inverted by:

$$\begin{aligned} x_{1:d} &= y_{1:d} \\ s, t &= h(x_{1:d}) \\ x_{d+1:D} &= (y_{d+1:D} - t) \odot \exp(-s)\,. \end{aligned}$$

An important property of the layers is that they can be inverted without inverting the function h that produces s and t. h, also known as the coefficient network, can therefore be arbitrarily complex. Further, by holding $\hat{x}_{1:d}$ constant through the transformation, the Jacobian will have zeros in the upper diagonal, and thus the determinant will only be the product of the diagonals of the Jacobian.

Conditional Normalizing Flows

Conditional Normalizing Flows change the scaling and shifting of a coupling layer based on conditional information. The conditional information, κ, is passed as part of the input to the network which estimates s and t. Now h is a function of both $x_{1:d}$ and κ. The invertibility of the coupling layer remains unaffected because the conditioning information is passed into h which need not be inverted to invert the layer. Figure 2.2 provides a simple diagram of the architecture. While other methods have been proposed for conditioning normalizing flows (i.e. Ardizzone et al. [6]) this formulation reduces the number of hyperparameters that must be tuned, and has a simple Maximum Likelihood of training loss:

$$\mathcal{L} = -\log(p_Z(z)) - \log \left| \det \left(\frac{\partial g(z|\kappa)}{\partial z^T} \right) \right|^{-1}. \tag{2.5}$$

For a more thorough review of Normalizing Flows see Papamakarios et al. [42].

Maximum Mean Discrepancy (MMD)

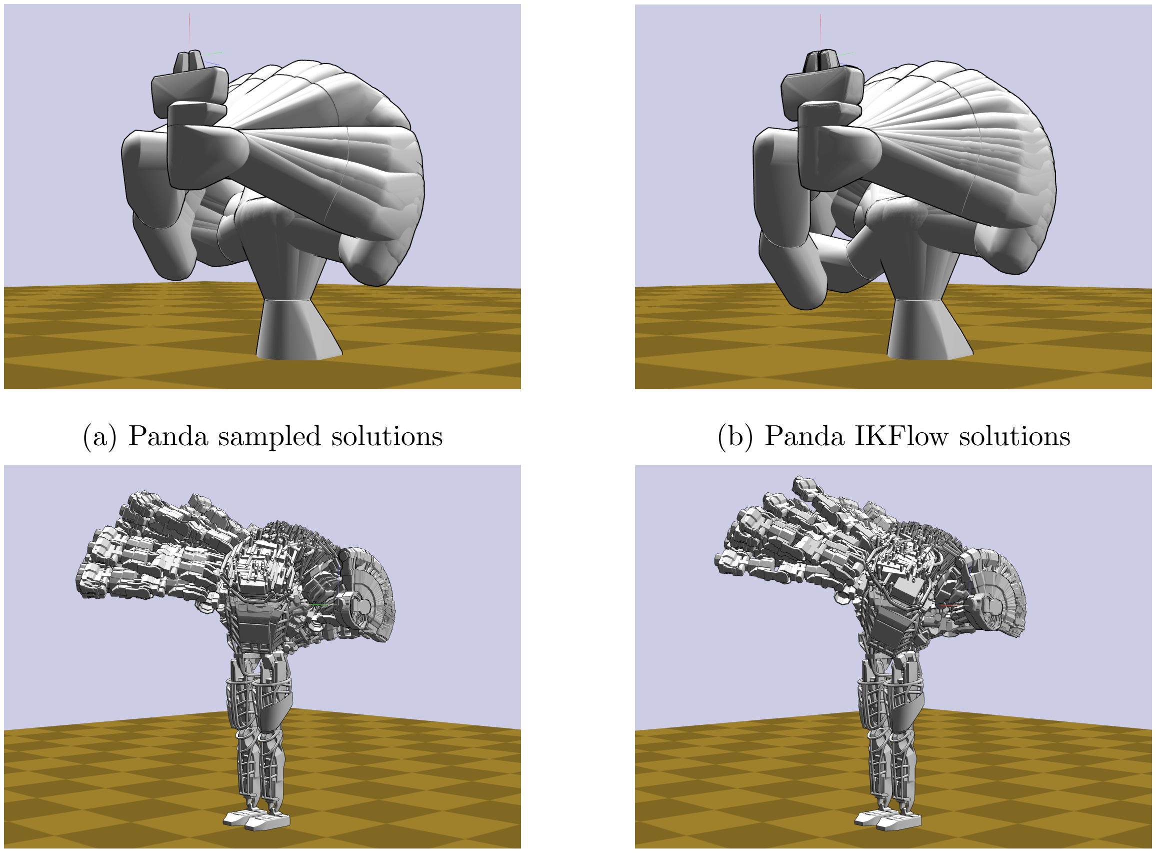
Finally, a method for comparing distributions is necessary to evaluate the performance of a generative model. We use Maximum Mean Discrepancy (MMD) because of its strong theoretical properties and ease of implementation. Given two distributions, $P(X)$ and $Q(Y)$, MMD computes the distance between the two distributions as the squared distance between the mean of the embeddings of the distributions.

$$\text{MMD}(P, Q) = ||E_{X\sim P}[\phi(X)] - E_{Y\sim .Q}[\phi(Y)]||.$$

Notably, the embeddings must be in a Reproducing Kernel Hilbert Space (RKHS). The capability of MMD to capture the distance between two distributions is dependent on the selection of a good embedding function ϕ. We selected the inverse multi-quadric kernel because of its long tails and previous use Ardizzone et al. [6]. A more detailed analysis of Maximum Mean Discrepancy is provided by Gretton et al. [25].

2.2 Related Work

There have been a number of attempts to use neural networks for inverse kinematics [1, 16, 19, 50]. Most of these approaches do not solely implement the inverse kinematics function. For example, Csiszar et al. [16] incorporate the problem of calibrating the physical robot with its internal model. Almusawi et al. [1] learn the kinematics model in addition to performing Cartesian control. Demby'S et al. [19] create a neural network approach that



(a) Panda sampled solutions

(b) Panda IKFlow solutions

(c) ATLAS sampled solutions

(d) ATLAS IKFlow solutions

Figure 2.3: A comparison of samples created by providing random seeds to TRAC-IK and samples from IKFlow for two robots, the Panda arm and ATLAS - Arm and Waist. Qualitatively, the solutions returned by the IKFlow models look to cover the same space of solutions found through rejection sampling. The MMD scores for the Panda and ATLAS solutions are respectively 0.0122 and 0.0081. A lower MMD score implies distributions that are more similar.

focuses exclusively on the IK problem but concludes that it is not a fruitful path because of large error.

The architectures used in the previous approaches are all fundamentally limited as they can only return a single solution for a particular input. However, IK is a one-to-many mapping, and access to additional solutions may prove useful for the control layer above IK. Ren and Ben-Tzvi [50] take a generative approach by using several different types of GANs, but the error of samples is worse than a fully connected network, with the best performing approach achieving 8cm of error. Ardizzone et al. [6] apply a similar deep generative approach to IKFlow but only to a small planar IK problem. We extend and refine their work in three key ways. First, we decrease the amount of error present in the solutions by adopting a conditional Invertible Neural Network. Second, we improve training stability by addressing a corner case related to the dimensionality of solution spaces. Third, we expand the coverage to include the entire physical workspace of the arm.

Kim and Perez [32] are the most similar point of comparison because they also use normalizing flows to solve inverse kinematics. IKFlow has several differences, the use of Conditional Normalizing Flows, the addition of training noise, and a sub-sampling procedure for the base distribution. These differences result in better performance for theoretical and practical reasons that are explained in the following section. Additionally, IKFlow lacks a network that can calculate the forward kinematics, because the forward kinematics of a system are easy to calculate analytically. IKFlow's single network can perform both inverse kinematics and density estimation, which are difficult to perform analytically but with reduced overall complexity. As a result of these differences, the accuracy achieved by IKFlow is $\sim$ 5x better in position, and $\sim$ 3x better in orientation, as measured with the Panda Arm and compared with Figure 3 in Kim and Perez [32].

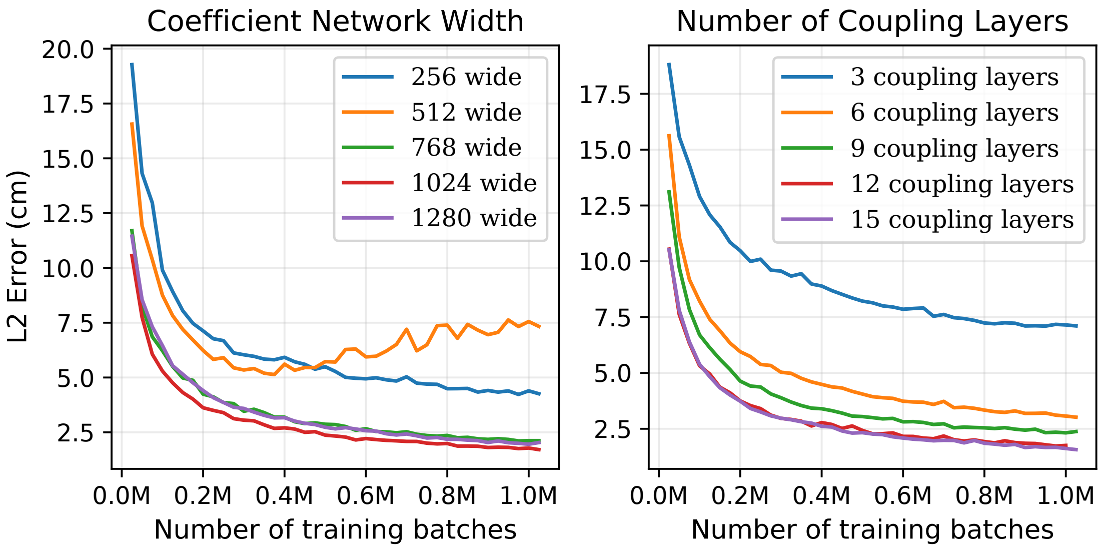


Figure 2.4: Learning curves for networks with increasing coefficient network widths (left) and the number of coupling layers (right) for the Panda Arm robot. For the coefficient network width comparison, the network has 12 coupling layers, where each coefficient network has 3 fully connected layers. A width of 1024 has the highest performance. For the coupling layer comparison, the coefficient function is a 3x1024-wide fully connected neural network with Leaky-Relu activation.

2.3 IKFlow: Learned Inverse Kinematics

The first step to learning inverse kinematics is to treat the solution space as a distribution that has a uniform probability, which may have multiple intervals, and is conditioned on the desired pose. We model the solution distribution with Conditional Normalizing Flows because they are quicker than alternatives to sample, are stable when training, and are capable of representing the full solution space [35]. Additionally, they don't face problems such as vanishing gradients or mode collapse which are common when training Generative Adversarial Networks (GANs), a different neural generative modeling approach [35].

There are three steps to applying Conditional Normalizing Flows to the inverse kinematics problem. First, a data set must be constructed. Second, we must select an architecture and parameters of the architecture. Third, we must design a loss function for minimizing the difference between samples from the generative model and the data distribution.

2.3.1 Data Generation

One advantage of the inverse kinematics problem is the relative ease with which data can be generated. All serial kinematic chains have known forward kinematics that can be used to generate training data. Data is generated by uniformly sampling from the interval defined by the joint limits. The joint samples are then fed through a forward kinematics function to obtain the corresponding Cartesian pose. The Cartesian pose becomes the conditional input to the network and the joint data is used as the target distribution. More sophisticated sampling methods could be used to account for effects at joint limits and for self-collisions, but we were able to obtain satisfactory performance without them and thus leave this investigation for future work.

2.3.2 Normalizing Flow Architecture

The fundamental architecture of the network is defined by the use of normalizing flows and was detailed in equations 2.1-2.4. The remaining design choices include the selection of a base distribution, selecting the number of coupling layers, the width of each coupling layer, and the specification of the coefficient network.

The base distribution affects the complexity of the density estimation and the sampling speed. We chose the Normal distribution because it simplifies the calculation of the Maximum Likelihood Loss function and is quick to sample. The remaining choices for the architecture affect the capability of the network to fit a target distribution. The width of each coupling layer must be at least as large as the degrees of freedom; if it is larger than the DOF it allows for multi-modal distributions to be more easily modeled. The number of coupling layers required depends on the interactions of joints with each other. The more complex the inter-joint relationship, the more coupling layers are required, as more layers allow for more interaction between joints because of the permutation layers. The number of coupling layers and the width of each can be found automatically by performing a hyperparameter sweep over a reasonable range (i.e. [1, 30] and $[n,\, n+10]$). Finally, the coefficient network must be expressive enough to capture dependencies between the conditional information (i.e. the goal pose) and a subset of the layer values. Table 2.1 details the variable parameters (selected by hyperparameter search) for each kinematic chain that we test. An analysis was performed to demonstrate the impact of the number of coupling layers and the width of the coefficient function networks on the average Positional error of a network Figure 2.4. This demonstrates that the error of a network decreases with increased size up to a point, at which point increasing the network size does not improve performance.

2.3.3 Loss Functions

The Maximum Likelihood loss detailed in equation 2.5 is theoretically the only loss function necessary for achieving high performance. However, for several of the kinematic chains tested, training diverged when trained with the Maximum Likelihood loss. The cause of this divergence was a mismatch between the dimension of the solution manifolds and the base distribution.

Solution Sub-manifolds

In Figure 2.3(a) the last joint of the arm appears in only one position. This implies that the solution space of that particular pose is not the same dimension as the joint space, because only 6 of the 7 joints in the arm can vary. This is not a unique situation but occurs throughout the configuration space as some Cartesian poses can only be reached by fixing one of the joints at a specific configuration, for example at one of its limits. This is problematic because the normalizing flow approach does not perform well on distributions that have a lower dimension than the base distribution. Specifically, the change of variables equation (2.1.2) does not hold if the base distribution and the target distribution are not the same dimension. One way to ensure the target distribution and the base distribution are the same dimension is to add noise that is the full dimension [31]. If the magnitude of the added noise is passed in as a conditional variable, its effect can be removed at test time by setting that piece of the conditional to 0:

$$c \sim U(0,1)$$
$$v \sim N(0,c)$$
$$\hat{x} = x + v,$$

where c and v are drawn every training iteration, and added to the training data. The resulting loss function is:

$$\mathcal{L} = -\log(p_Z(z)) - \log\left|\det\left(\frac{\partial g(z|p,c)}{\partial z^T}\right)\right|^{-1}, \tag{2.6}$$

where p is the desired Cartesian pose.

2.3.4 Base Distribution Sub-sampling

One downside of using the Normal distribution for the base distribution is the tails of the distribution. At evaluation time a point sampled from the tail of the Normal distribution is likely to produce a solution that has a high error because the loss function encourages known solutions to be closer to the mean of the distribution. In order to reduce the impact of the tails the base distribution is sub-sampled at test time. This reduces the likelihood of a point from the tail of a distribution, however, this encourages less diverse solutions. This trade-off is demonstrated by Figure 2.7. Thus the scaling of the base distribution should be treated as a tuning parameter for the particular application of IKFlow.

2.4 Experiments

The models were built with the FrEIA framework [6] and trained with PyTorch [43]. Additional details about the network architectures and training parameters can be found in the appendix. Once trained the model was evaluated on the three desired quantities: speed, accuracy, and solution space coverage.

While evaluating a model, we use a scaling factor of 0.25, which provides an adequate trade-off between accuracy and diversity. We found that in general, this scaling factor lowers the average positional error by $\sim 30\%$, at the expense of an $\sim 150\%$ increase in MMD Score.

Speed was measured by the time required to sample 100 solutions for a given Cartesian pose, averaged over 50 randomly-sampled Cartesian poses. In addition, to understand how the model scales, the runtime was measured as the number of requested solutions increased.

Model accuracy was measured by sampling 1000 test Cartesian poses and obtaining 250 joint solutions for each Cartesian pose. The joint solutions were then passed into a forward kinematics routine to compute the realized Cartesian poses. The averaged difference between the 250000 desired and realized poses was recorded and reported in terms of position and geodesic distance.

Sample diversity and solution space coverage was measured by calculating the MMD score for ground truth samples and IKFlow samples. The MMD Score is found by taking the average of 2500 Maximum Mean Discrepancy values, each calculated between ground truth samples and the samples returned IKFlow. For each calculation, 50 joint space solutions are generated by each method for a randomly drawn pose. The minimum possible MMD value is 0. Values close to zero imply that the distributions are more similar; thus when the IKFlow achieves a low MMD score, it provides good coverage of the solution space. The ground truth samples were calculated by providing different seeds to TRAC-IK for the same Cartesian pose.

In addition, we compare IKFlow to an Invertible Neural Network (INN) [6] and to a Mixture Density Network (MDN) [10] on two benchmarks—railY_chain3 and Panda Arm. The railY_chain3 robot (first used by Ardizzone et al.), is a planar robot with 4 joints—the first is a prismatic actuator along the y axis with limits [-1, 1]. The final three are revolute joints with limits $[-\pi, \pi]$ and associated arm segments of length 1. For a fair comparison, the IKFlow and INN models have the same number of coupling layers and size of coefficient networks. The coefficient networks are 3x1024-wide fully connected networks with Leaky-Relu activation. A softflow noise scale of 1×10^{-3} was used across all of the models. The learning rate was set to 5×10^{-4} and decayed exponentially by a factor of 0.979 after

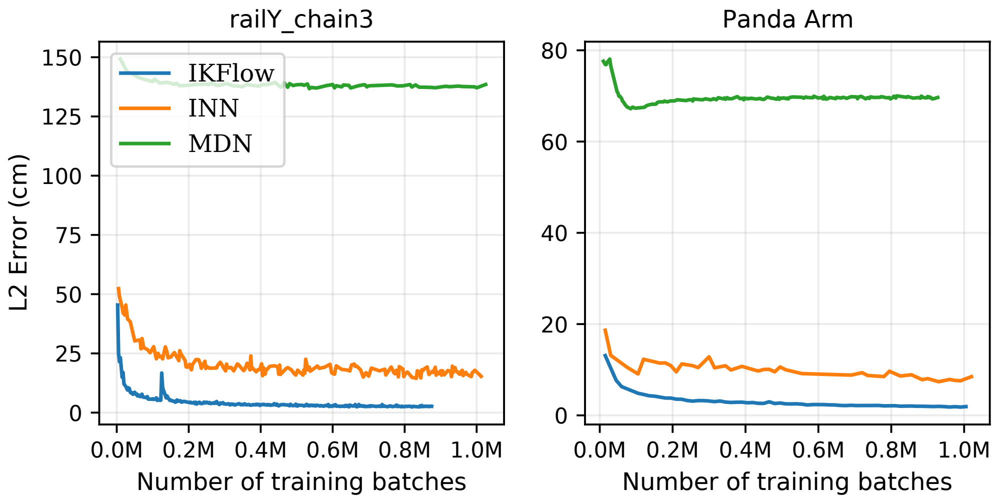


Figure 2.5: Learning curves for IKFlow, INN, and MDN for the railY_chain3 and Panda Arm robots.

every 39000 batches. Models were trained until convergence on 2.5 million points using the Ranger optimizer with batch size 128 using a NVIDIA GeForce RTX 2080 Ti graphics card. These experiments were carried out on 10 different kinematic chains across 6 different robots, to evaluate the generality of the approach across different kinematic structures.

2.4.1 Benchmarking Implementation

For the railY_chain3 test, the IKFlow and INN models both have 6 coupling layers with 3x1024-wide fully connected networks and a latent space dimension of 5. For Panda Arm, both models have 12 coupling layers with 3x1024-wide fully connected coefficient networks and a latent space dimension of 10. The Mixture Density Network (MDN) has 70 and 125 mixture components for the railY_chain3 and Panda arm respectively. MDN models were implemented using the *tonyduan/mdn* repository [21].

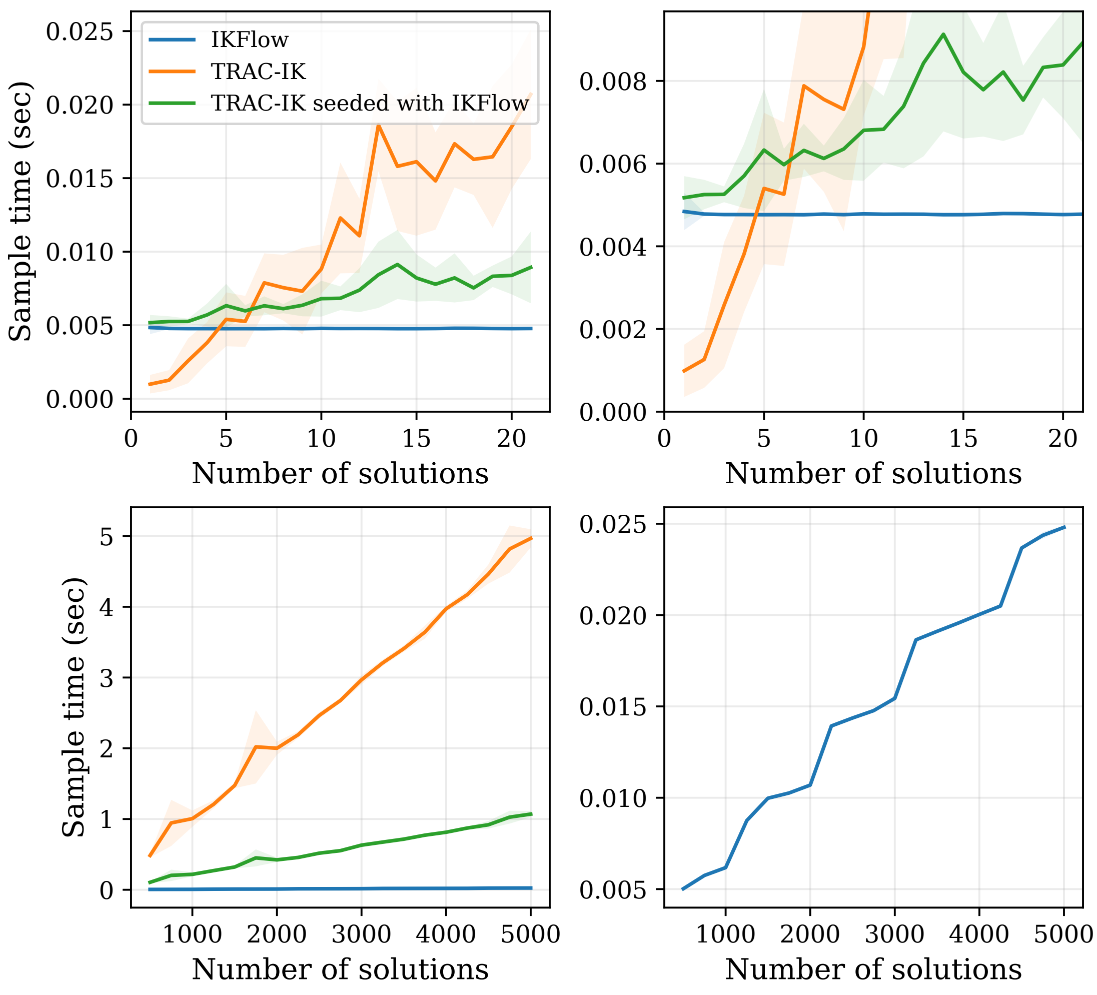


Figure 2.6: The runtime of IKFlow, TRAC-IK, and TRAC-IK when seeded with solutions returned by IKFlow as a function of the number of requested solutions for the ATLAS Arm. The relationship for the IKFlow model is roughly linear. The steps in the IKFlow graph correspond to the size of the batch that can be fit on the GPU. Seeding TRAC-IK with IKFlow provides the 1e-6 accuracy of TRAC-IK with one fifth the runtime of TRAC-IK.

2.5 Results

Our results demonstrate that IKFlow provides a representative set of solutions, quickly, with acceptable error. The results of all the experiments can be found in Table 2.2.

2.5.1 Comparative Evaluation

Benchmarking results are shown in Figure 2.5. On both test beds, the IKFlow model performs considerably better than the other two models. These results are consistent with previous findings [37], in which it is shown that Conditional INNs (cINNs) outperform INNs on the railY_chain3 testbed. The Panda Arm test bed demonstrates that IKFlow handles higher dimensional problems better than the INN and MDN. The higher the dimension of the problem, the more likely there is to be a lower dimension solution space that would introduce instabilities in training.

2.5.2 Accuracy

The accuracy of the system output ranges from 7.72mm to 0.36mm and from 2.81 degrees to 0.15 degrees. For a point of reference, the mechanical repeatability of many industrial arms is 0.1mm. This level of accuracy is sufficient for many tasks; additionally, these solutions can also be quickly refined with numerical optimization approaches to reach arbitrary levels of precision. For ATLAS Arm, refinement takes on average 0.20 ms, as presented in Figure 2.6.

2.5.3 Runtime

The runtime of the approach is fast enough to enable its use as a sub-routine in other algorithms. Nonlinear optimization approaches find a single solution in about 0.3 millisecond [9], whereas IKFlow can return 500 solutions in 5ms. Figure 2.6 also demonstrates that the approach scales linearly with the number of requested solution samples. The gradient of the increase is low with $4,000$ samples found in 20 milliseconds. This means that even complex solution sets can be approximated quickly. For a point of comparison, if TRAC-IK,

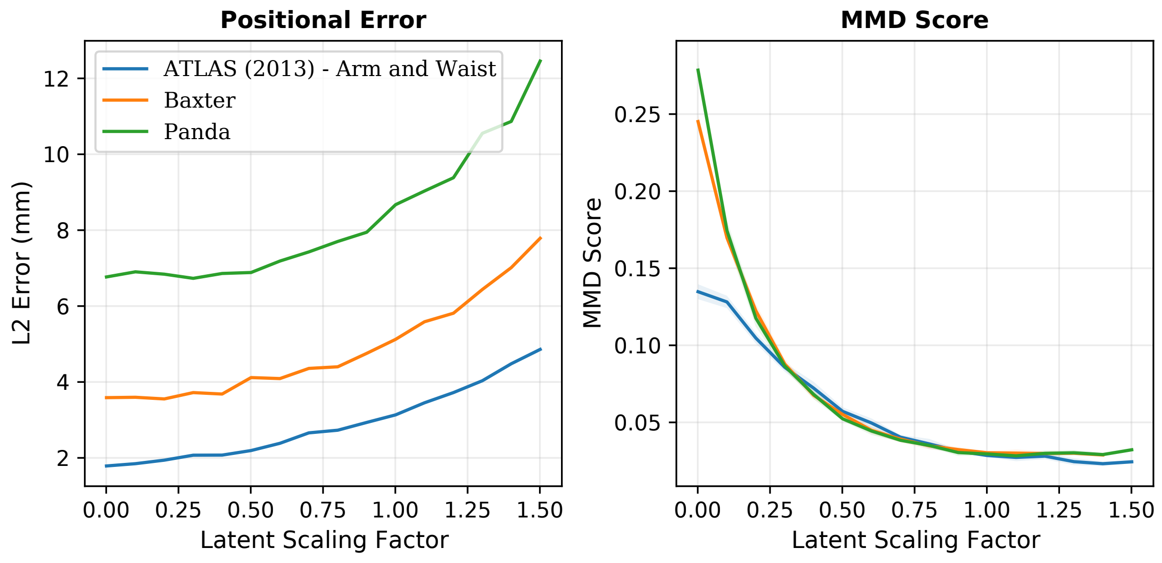


Figure 2.7: Positional Error and MMD Score of IKFlow as a function of the scaling factor of the latent noise used to sample from the model. Reducing the scaling value increases the model's accuracy at the cost of lowering the diversity of the returned solutions.

a common nonlinear optimization-based IK solver [9], is fed random seeds in hopes that the local minima it finds are different, it takes more than a second to return 1,000 samples.

Notably, however, TRAC-IK takes approximately 5x less time to run when seeded with an approximate solution returned by IKFlow. Empirically, it is faster to first run IKFlow to generate seeds before running TRAC-IK when requesting 7 or more solutions.

2.5.4 Solution Space Coverage

Figure 2.3 provides a qualitative comparison of ground truth samples with samples from IKFlow. The solution spaces look quite similar and provide reference points for the MMD score for other chains. All of the kinematic chains included have a MMD score under 0.05. This implies that the IKFlow solution spaces are very similar to the ground truth samples, with only a few erroneous solutions, or small gaps in coverage.

2.5.5 Limitations

While the proposed method is shown to accurately model kinematic chains with L2 Position error as low as 0.36 mm, it has been found that the training time required for models to reach a given position error grows with the complexity of the kinematic chain that is being modeled. While there exist geometrically derived formulations of kinematic complexity, we choose a simple formulation—the sum of the differences between the upper and lower limits for every joint in the robot:

$$\sum_{i=1}^{n} u_i - l_i, \tag{2.7}$$

where u_i and l_i are the upper and lower limits, respectively, for joint i. Here this is referred to as the sum of joint limit ranges. An experiment was performed by artificially expanding the joint limits for three robots and measuring the resulting number of training batches required for the respective IKFlow models to reach 1cm of error. The results indicate that the number of training batches required for IKFlow to reach a given L2 error grows exponentially with an increase in equation 2.7 of the kinematic system it is modeling. The results of this experiment are shown in Figure 2.8.

IKFlow is a novel IK solver capable of providing quick, accurate, and diverse solutions for kinematically redundant robots operating in $SE(3)$, based on modeling IK solutions as a distribution over joint poses and using deep generative modeling to model these distributions. Our experiments demonstrated that IKFlow can generate hundreds to thousands of solutions covering the solution space in milliseconds. The average pose error of the results showed that our approach is capable of finding solutions with millimeters of translation error, and less than 1.5 degrees of rotational error. These results demonstrate that IKFlow can serve as the basis for the expanded functionality of 7+ DOF kinematic chains. In the next chapter we step up a layer in the planning hierarchy to construct and optimize controllers.

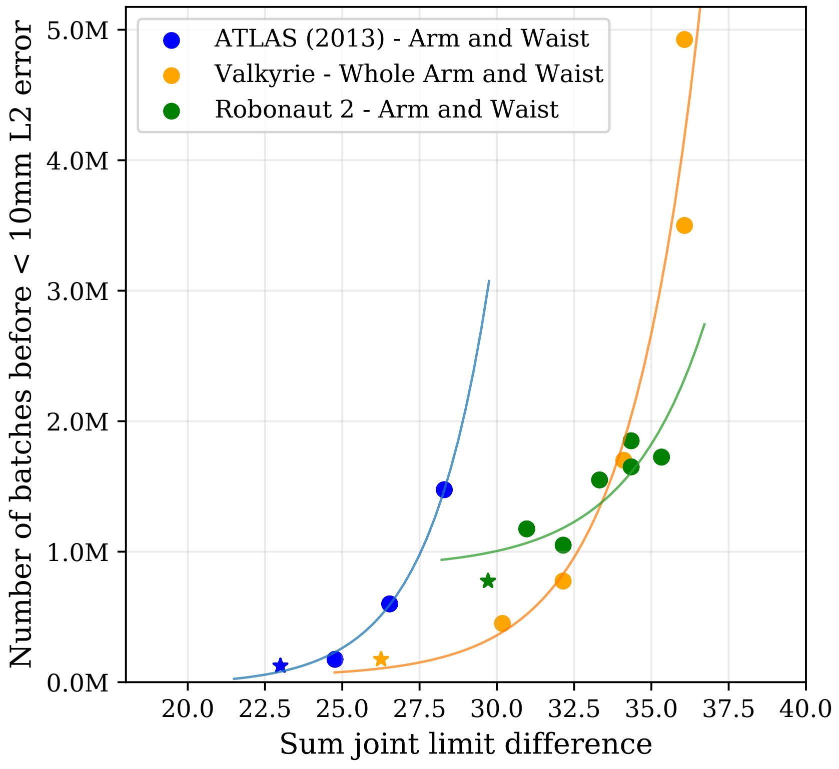


Figure 2.8: The number of training batches before the IKFlow model reaches an average of 1cm L2 position error as a function of the sum of the joint limit ranges of the kinematic chain. The actual robot is plotted as a star - the other kinematic chains have artificially expanded joint limits. The results indicate that the number of training batches to reach 1cm of error grows exponentially with the sum of the joint limit ranges.

Table 2.1: The primary parameter to select when fitting a network to a new kinematic chain is the number of coupling layers. The number of coupling layers increases the expressivity of the invertible portion of the network. The network can also be made more expressive by increasing the coupling layer width. When the width becomes larger than the DOF the system becomes more capable of representing multi-modal target distributions.

Robot	DOF	Coupling Layer Width	Coupling Layers	Number of Parameters
ATLAS (2013) - Arm and Waist	9	15	9	3.836×10^7
ATLAS (2013) - Arm	6	15	12	5.115×10^7
Baxter	7	15	16	6.820×10^7
Panda	7	9	12	5.093×10^7
PR2	8	15	8	3.410×10^7
Robonaut 2 - Arm and Waist	8	15	12	5.115×10^7
Robonaut 2 - Arm	7	15	12	5.115×10^7
Valkyrie - Whole Arm and Waist	10	15	16	6.820×10^7
Valkyrie - Lower Arm	4	15	9	3.836×10^7
Valkyrie - Whole Arm	7	15	12	5.115×10^7

Table 2.2: This table contains the results of the experimental runs. Since the IKFlow distribution is compared with a ground truth distribution the MMD score provides a measure of how well the solution distribution from the network covers the whole solution space. Time is the total time to return 100 solutions for one Cartesian pose measured in milliseconds. Average position error and Average angular error are the position and geodesic distance, respectively, between the desired goal pose and the pose achieved from joint solutions.

Robot	DOF	MMD Score	Time (msec)	Average L2 Position Error (mm)	Avg Ang. Error (degree)
ATLAS (2013) - Arm and Waist	9	0.03723	4.8	2.66	0.61
ATLAS (2013) - Arm	6	0.004828	6.4	1.19	0.28
Baxter	7	0.0331	8.29	4.5	1.19
Panda	7	0.0306	6.28	7.72	2.81
PR2	8	0.2128	4.31	3.3	1.56
Robonaut 2 - Arm and Waist	8	0.03691	6.44	3.63	0.77
Robonaut 2 - Arm	7	0.0327	6.45	1.91	0.46
Valkyrie - Whole Arm and Waist	10	0.0361	8.55	3.49	0.74
Valkyrie - Lower Arm	4	1.508e-06	4.87	0.36	0.15
Valkyrie - Whole Arm	7	0.0292	6.45	1.64	0.38

Chapter 3

Improved Feedback Motion Planning

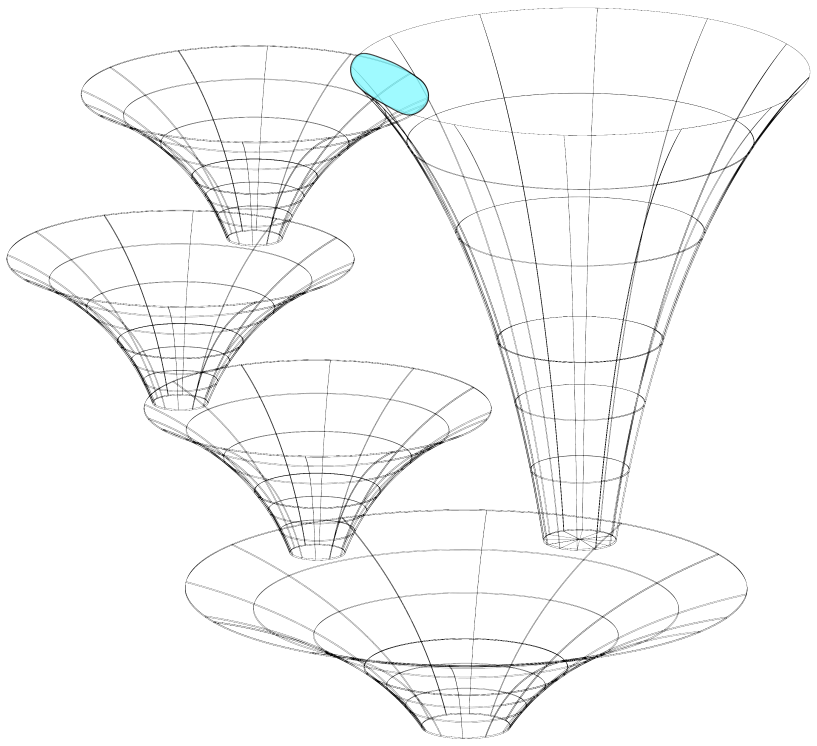


Figure 3.1: An illustration of controller overlap. The cyan region highlights the overlap between funnels, which can be used for estimating the consistency of their value estimates.

In the previous part, symbols were defined to have a precondition and an effect. The precondition and effect ensure that the symbol can be sequenced with other symbols. In the prior chapter, these were constructed by gathering data, clustering the data, and then constructing a classifier around the clusters. This may not always be a feasible, or desirable, approach because it requires a number of demonstrations, and gathering demonstrations on a robot is expensive. In the control community, there has been a considerable amount of work applied to understanding the region of attraction for a controller. The region of attraction for a controller is the subset of state space from which the controller can be operated and it is guaranteed that the controller will reach a particular end state. The region of attraction is the same definition as the precondition of the symbol. Operationally there is an important difference, the region of attraction can sometimes be built with the knowledge of system dynamics and the controller. This reduces the amount of data that must be collected in order to build a symbol.

Specifically in this chapter, we will look at LQR-Trees. LQR-Trees is an algorithm that sequences controllers by overlapping the end state of one controller with the region of attraction of another controller. We extend the work done in LQR-Trees by improving the optimality of the controllers that it constructs. Our solution takes inspiration from the Bellman Residual and demonstrates empirically that the controllers can be improved by reducing the discrepancy between value estimates of two different funnels.

3.1 Background

LQR-Trees [60] is a prominent feedback motion planning algorithm. It uses advances in automatic controller construction to form a set of controllers for underactuated nonlinear systems. LQR-Trees is a sampling-based planner [39] but the local planner is replaced with a method that creates provably stable controllers. It continues to add controllers, stably sequencing them, until they cover a bounded region of configuration space. The components of the algorithm follow.

3.1.1 Stable Controller Construction

The first major component of the algorithm constructs a stable controller that connects two states. First, a reference trajectory between the two states must be found. Trajectory optimization finds a sequence of states and actions that will take the robot from the start point to the goal while obeying constraints enforced by the physics of the robot and attempting to minimize the cost function. However, it can only be guaranteed to find a local minimum. The output of trajectory optimization is a single trajectory and if the robot ever deviates from it, there is no defined control behavior. Therefore, the LQR-Tree algorithm wraps each trajectory in a Time-Varying Linear-Quadratic Regulator (TV-LQR) [5]. Before we review TV-LQR, we will briefly touch on the Linear-Quadratic Regulator.

Linear-Quadratic Regulator (LQR)

The Linear-Quadratic Regulator (LQR) [5] assumes the state dynamics are linear in the state and action variables, and that the cost function is quadratic in the state and action variables. When these assumptions are satisfied a continuous problem of high dimension can be solved directly. The transition function of the LQR is defined as:

$$\dot{s}(s,a) = As + Ba. \tag{3.1}$$

A describes how an agent's state would evolve passively over time and B describes how an action changes the state of the agent. s is the state of the system and a is the action taken. The cost function of LQR is defined as:

$$c(s,a) = s^{\intercal}Qs + a^{\intercal}Ra, \tag{3.2}$$

where Q weights the how much the state affects the cost incurred and R is the cost associated with taking an action. LQR is heavily used in control theory because an optimal solution is known for the continuous case. The value function of an LQR problem is found via the Discrete Algebraic Riccati equation:

$$P = A^{\intercal}PA + (A^{\intercal}PB)(R + B^{\intercal}PB)^{-1}(B^{\intercal}PA) + Q. \tag{3.3}$$

The value function for this control problem is:

$$V(s) = s^{\intercal}Ps. \tag{3.4}$$

The ease of computing the solution is balanced by the rather stringent requirements that the state dynamics are linear, and the cost is quadratic. While this might seem to limit the applicability of LQR, locally linearizing has proved to be an effective solution for many systems [59].

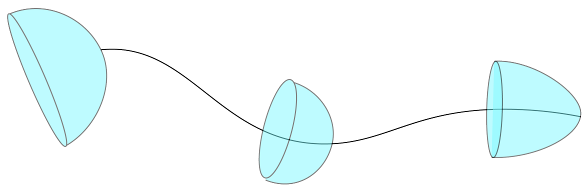


Figure 3.2: Time Varying LQR is an extension of LQR to handle trajectory following. This figure depicts three different value functions for three different time points along the trajectory. Since the LQR problem is solved in relation to the trajectory the value function solutions will change in time. Thus some portions may have larger regions of stability than others.

Time-Varying LQR (TV-LQR)

Time-Varying LQR [5] is an extension of LQR which handles a time-varying goal point, i.e., a trajectory. TV-LQR approximates the region around a trajectory as an LQR problem and provides feedback commands for states that are off of the trajectory.

The state dynamics for the time-varying problem are linearized as:

$$\dot{s}(s, a) = A(t)s + B(t)a. \tag{3.5}$$

The value function and control are then determined in relation to the goal trajectory, $\tilde{s}$, given:

$$V(s, a, t) = \int_t^T \big(a^\intercal R a + (s - \tilde{s}(t))^\intercal Q (s - \tilde{s}(t))\big) dt.$$

The feedback control is then a linear term, K, summed with the command selected by the trajectory optimizer, a_{traj}:

$$a_{TV} = K^\intercal s + a_{traj}. \tag{3.6}$$

Combining TV-LQR with trajectory optimization allows for a small region of acceptable states around a trajectory to be stabilized. Figure 3.2 depicts how the value function, and the corresponding region of attraction, change along a trajectory at three different time points.

While the TV-LQR feedback controller is defined for all states off of the trajectory, it is not stable for all of those points. In order to determine the region of stability around the trajectory, the LQR-Tree algorithm uses a convex optimization approach which constructs an ellipse that is an inner approximation of the region of stability. Any point inside the region is stable in the sense of Lyapunov.

Lyapunov Stability Analysis

Lyapunov stability analysis is a method for ensuring that a dynamical system reaches a region around an equilibrium point. If a controller can be constructed whose value function has a derivative that is negative everywhere but at the goal, then that controller is guaranteed to converge to the goal region:

$$\dot{V} = \nabla V f(s(t), a(t)) \leq 0. \tag{3.7}$$

Intuitively, the negative derivative of the value function for a controller ensures that every step of every trajectory decreases in cost. The region around the goal point for which the value function has a negative derivative is the region of attraction. Any point in the region of attraction is guaranteed, asymptotically, to converge to the goal.

There are several ways to construct Lyapunov functions and their associated regions of attraction. The Sum-of-Squares (SOS) [61] approach uses semi-definite programming and a polynomial description of the dynamical system to construct a provably conservative estimate of the Lyapunov function. However, it does not scale well to high dimensions. This is in part due to the explosion of polynomial terms as the dimension of the system increases. There are also simulation-based approaches [49] which rely on probabilistic arguments for defining their regions of attraction. Most recently, the simulation-based approaches were paired with function approximators [53] to learn oddly-shaped regions of attraction.

3.1.2 Space Covering

Each of the controllers constructed in a manner consistent with the previous section covers a portion of the state space. The LQR-Trees algorithm continues to construct these stability region bounded controllers until a bounded region of the state space has been covered in controllers. The addition of controllers happens in a method similar to RRT [56]. A random point, not contained in any current controller's stability region, is selected. If a stable controller can be constructed from that point to any point already on the tree then that controller is added. If a controller cannot be constructed then a new point is selected. Figure 3.1 provides an idea of how the tree might grow to cover the configuration space.

While the LQR-Trees algorithm provides a way to construct provably stable policies for underactuated nonlinear systems it is not capable of reducing the cost of the composite policy because it terminates whenever the space is covered and has no way of removing and replacing sub-optimal controllers.

3.2 Optimizing LQR-Trees

We use a combination of optimal control methods and reinforcement learning to construct a more optimal policy for nonlinear underactuated systems. This combination produces a composite policy that is more optimal than the original LQR-Trees policy. Importantly, the resulting policy retains its stability bounds which provide assurance that the system will perform as expected, which is becoming increasingly important as robotics begins to be applied in close proximity to humans and in safety-critical applications.

The method described here combines several existing methods in order to construct a set of controllers that cover the configuration space with verifiable and approximately optimal controllers. Trajectory optimization provides trajectories that are locally optimal and respect constraints. TV-LQR and Lyapunov analysis provide approximate methods for determining the value of states around the trajectory and the boundaries of those estimates. Sampling-based motion planning provides a method for covering the space, as well as guidance on which neighbors to connect to. Finally, reinforcement learning provides a way to remove and replace controllers so that the policy improves over time. Figure 3.3 provides an overview of the whole process.

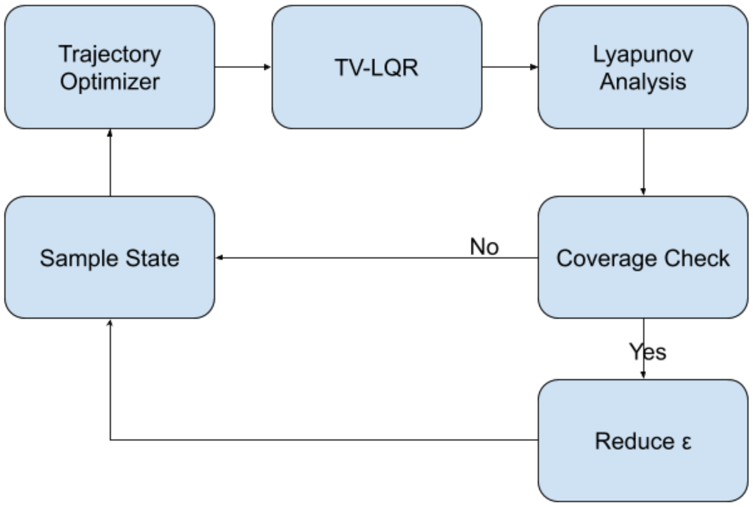


Figure 3.3: A flow diagram of the system detailed in Section 3.2. The start point for the diagram is Sample State. The process is an iterative process that continues as long as there is empty space that needs to be covered, or there exists a value function discrepancy between two controllers that is too large.

3.2.1 Dynamic Programming and Trajectory Optimization

Trajectory optimization has significant benefits over dynamic programming as the dimensionality of the problem grows. However, trajectory optimization only finds locally optimal trajectories. We combine these two approaches to construct bounded cost controllers. Atkeson [7] notes that a Bellman Residual can be calculated at any point where two trajectories overlap. The Bellman Residual is a measure of the inconsistency in value function estimates. Each trajectory contains an estimate of the value function at that point, thus the difference of the value functions at that point is a sample of the Bellman Residual. This is accomplished by constructing a value function for every trajectory returned by the trajectory optimizer. The extent of the value function is limited by Lyapunov analysis. Lastly, this cycle is repeated until all controllers overlap with at least one other controller, and all samples returned are within the threshold on the Bellman residual.

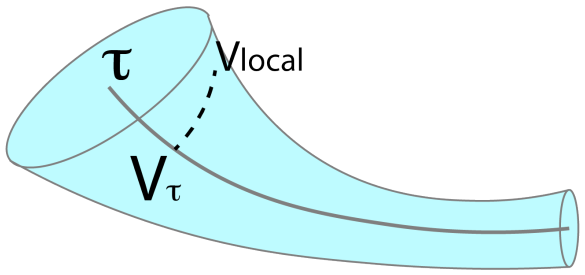


Figure 3.4: The trajectory τ and its associated TV-LQR funnel are used to approximate the value at V_{local}. The end state of the controller is at the right of the trajectory and can be reached by any state that would be contained within the cyan section of the funnel. The value function can be estimated at any point within the cyan region. This value estimate is a combination of value backup along the trajectory and LQR approximations at each point along the trajectory.

Trajectory-Based Approximate Value Function

The first step is to approximate the value function of a task using a set of trajectories returned by the trajectory optimizer. The trajectory τ returned by the trajectory optimizer is the reference for a TV-LQR policy, which defines a value function centered on the trajectory. The region of stability for the TV-LQR defines the extent to which the value function is considered valid. To be clear, the TV-LQR could approximate the value of any point, however here the approximation is restricted to the stable region around the trajectory. Only using the values from inside the stability region defined by the Lyapunov analysis ensures that only stable controllers are constructed.

In order to estimate the value of a particular state, first the value of the closest point, s_0, on a trajectory is found by backing up along the trajectory:

$$V_\tau(s_0) = \sum_{i=0}^{T} \gamma^i r(s_i, \tau(s_i)) + \gamma^{T+1} V(s_{T+1}), \tag{3.8}$$

where $V(s_{T+1})$ is the value of the endpoint the trajectory optimizer used, and γ is the discount factor, which weights how a reward received now should be traded with a reward in the future. The process of selecting the endpoint will be discussed in Section 3.2.1. Next, the value function constructed by TV-LQR for that point on the trajectory is used to estimate the local value of the sample point:

$$V_{local}(s) = s^{\intercal} P_{local} s. \tag{3.9}$$

This local value is then combined with the backup from equation 3.8 to provide a global action-value estimate:

$$V(s) = V_{local}(s) + V_{\tau}(s_0). \tag{3.10}$$

This approximation is visualized in Figure 3.4. The local value estimate, V_{local}, performs two duties here. First, it provides an approximate value for the value function from a particular point on the trajectory. Note that, the value function, V_{local} returned by TV-LQR for a trajectory is not the same as the value function for the task, but rather it is the value function for stabilizing the system around the trajectory. Freeman and Primbs [23] demonstrate that the Lyapunov value function, in this case V_{local} is an upper bound on the optimal value function for some meaningful reward function. Primbs et al. [46] relate V_{local} and the optimal value function of a known reward function by a scalar multiple:

$$V_{Task}(s) \approx \lambda(s) V_{local}(s). \tag{3.11}$$

Knowledge of λ allows an approximation of the optimal value function to be calculated from the TV-LQR value function. Primbs et al. [46] demonstrate that λ can be derived from Sontag's formula [23], which results in the following:

$$\lambda(s) = 2\left(\frac{V_s A + \sqrt{(V_s A)^2 + Q(s)(V_s B B^t V_s^t)}}{V_s B B^t V_s^t}\right), \tag{3.12}$$

where V_s is the derivative of the local value function, V_{local}, with respect to the state space, A is the passive state dynamics, B is the dynamics resulting from control, and Q is the cost due to state only. This approximation works well in areas where the shape of the local value function and the optimal value function are similar. The Lyapunov analysis ensures that the local value function and the optimal value function are of similar shape in the stability region.

The second way the local value function is used is to determine the boundary around the trajectory:

$$\{s|V_{local}(s) < \epsilon\}. \tag{3.13}$$

Here the level set of the local value function is used to determine the region of stability for the controller. The combination of these two aspects permits a value function approximation to be constructed any time a point is contained within a funnel.

Neighbor Selection

In addition to refining the trajectories that a particular controller takes, the neighbor that a particular start state is connected to will also affect the optimality of the solution. Following Hauser and Zhou [27], we only consider neighbors with a decreasing cost as iterations increase. Thus, in the limit, the trajectory optimization will be using the approximately optimal value function, $\hat{V}^*$, to connect to the goal.

Termination Criteria

Convergence is determined by evaluating the Bellman Residual at sample points drawn from a uniform distribution over the state space. Since a point may belong to more than two funnels, only the two lowest cost estimates are used. If the lowest two approximations are within ϵ of each other then we assume that the funnels aren't far from optimal. This assumption is based on a technique from Williams and Baird [65]:

$$||V^* - V|| = \frac{\epsilon}{1-\gamma}. \tag{3.14}$$

One difficulty with this approach is that the Williams and Baird bound is based on an ϵ that is defined over the entire value function of a task, and here it is being used to determine the convergence of a small number of funnels. Another difficulty is that the Bellman Residual is defined over the whole function as opposed to samples. We leave the proof of the validity of this assumption to future work. We continue adding trajectories until all sampled points have two funnels with approximate values within ϵ of each other.

3.2.2 Optimizing LQR-Trees Algorithm

Algorithm 1 Optimizing LQR-Trees

```
1: procedure ConstructLQRTree($c, g, \alpha, \epsilon_{max}$)
2: ▷ The cost function, the goal, the contraction coefficient, and the maximum amount of error
3:     $T \leftarrow \emptyset$
4:     $T \leftarrow LQR(g)$                    ▷ Add LQR solution for goal to tree.
5:     $coverThresh$          ▷ A number used to determine probability of coverage.
6:     $coverSum \leftarrow 0$
7:     $\epsilon = \frac{R_{max}}{2(1-\gamma)}$
8:     while $\epsilon > \epsilon_{max}$ do
9:         $\epsilon = \alpha * \epsilon$
10:        while $coverSum < coverThresh$ do
11:            $s \leftarrow RandomConfiguration$
12:            if $ValidValue(s, \epsilon)$ then
13:                $coverSum \leftarrow coverSum{+}1$
14:            else
15:                $coverSum \leftarrow 0$
16:            if $\neg InFunnel(s)$ then
17:                $n \leftarrow SelectNeighbor(s, T)$
18:                $\tau \leftarrow ConstructFunnel(s, n, c)$
19:                $T \leftarrow AddFunnel(\tau, T)$
20:    return $T$
```

The algorithm combines the two previous sections into an alternating algorithm which first constructs an LQR-Tree that probabilistically covers the configuration space. Lines 6-12 implement the same probabilistic coverage mechanism that was used in the original LQR-Trees paper [60]. The *ValidValue* function implements the bound from equation 3.14. *InFunnel* checks whether a point is contained in any of the regions of attraction for nodes of the LQR-Tree. The *selectNeighbor* function implements the method described in the Neighbor Selection subsection. *ConstructFunnel* uses a trajectory optimizer to connect s to s_n, where s_n is the zero point defined by the Lyapunov function at node n, such that the cost according to c is minimized. It then passes that trajectory to a TV-LQR method which constructs a candidate Lyapunov function that is then scaled by a Sum-of-Squares Lyapunov analysis. *AddFunnel* takes the previously constructed funnel and adds representative Lyapunov functions to the tree.

3.2.3 Experiments

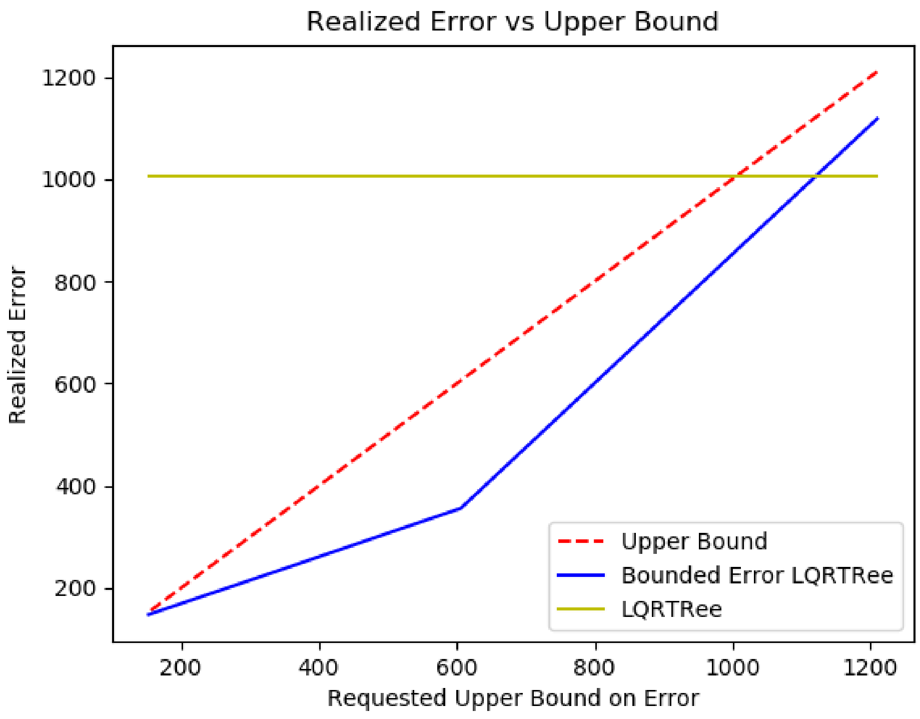
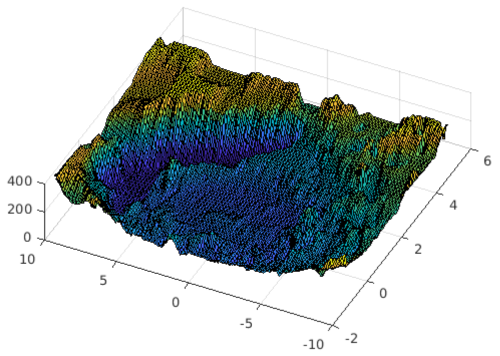


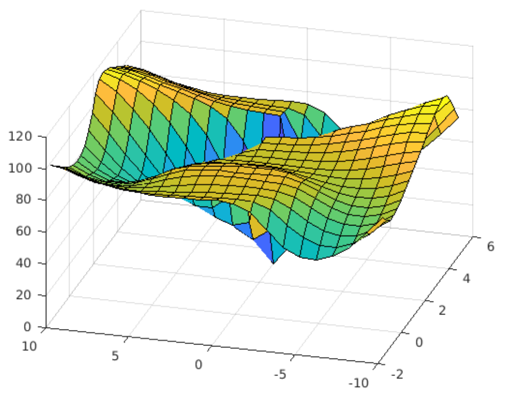
Figure 3.5: A comparison of Bounded Error LQR-Trees and LQR-Trees. The red line is the error upper bound. The yellow line is the maximum observed error of the LQR-Trees method. The blue line is the observed maximum error for the Bounded-Error LQR-Trees method described in Section 3.2. All the bounded error trees constructed stay under the requested error bound.

In order to validate the model we evaluate its performance on the constrained inverted pendulum problem. The constrained inverted pendulum problem was chosen because it is an underactuated nonlinear control problem, which makes it a reasonable first test for robotics control frameworks. The approach detailed in the previous section is compared to two baselines: a Value Iteration dynamic programming solution that uses state space discretization, and LQR-Trees.

Comparisons between the baselines and the proposed approach were performed by randomly selecting a start state from the configuration space of the pendulum. The goal in all experiments was to reach the inverted position, 3.14, with velocity 0. Trajectories from



(a) The approximate value function constructed by sampling from the Lyapunov stability regions.



(b) The value function constructed by Value Iteration for the same problem.

Figure 3.6: The LQR-Trees value function is similar in structure to the optimal value function from Value Iteration.

all the different baselines were re-sampled so that they used the same time step. Then a trapezoidal integration was used on the re-sampled trajectories to calculate the cost. This was repeated for 100 samples to gather sufficient statistics on max cost and average cost. All experiments were carried out in MATLAB using Drake [62].

3.2.4 Results

In order to compare bounded error LQR-Trees to the two baselines the value function created by the value iteration baseline was considered to be V^*. In Figure 3.5 several different bounded error LQR-Trees are compared to the upper bound that they were given. Every tree respects the upper bound it was provided. The horizontal yellow line is the maximum observed error from the baseline LQR-Tree. In summary, Figure 3.5 provides evidence that the bound from equation 3.14 holds for composite policies. It also suggests that the approximation of the value function via the Primbs' multiplier (eqn. 3.12) and a quadratic Lyapunov function are an upper bound on the optimal value function.

Next, we compare the value functions produced by Value Iteration and the first iteration of the bounded LQR-Trees approach. Figure 3.6(a) is the approximate value function constructed by the proposed method after one iteration. Figure 3.6(b) is the value function created by Value Iteration. The overall structures of the two value functions are similar. Both contain a deep valley near the solution, with large ridges near the corners $(4.71, 10)$ and $(4.71, -10)$. In addition, they both contain a small bump centered on $(0, 0)$. The major difference between the two is the sharpness of the creases. This is due to the application of Lyapunov stability analysis to the approximation of values. In regions where the dynamics might change rapidly, equation 3.7 may no longer hold true, and thus the region of attraction will have boundaries on or near rapid changes in the dynamics. This prevents the value approximation from smoothing out creases as is seen in Figure 3.6(b). For example, the regions of attraction found in the valley around the solution, $(3.14, 0)$, are aligned with the long axis of the valley, because the dynamics of the system rapidly change once a certain velocity threshold is surpassed. This threshold corresponds to the top of the ridge that defines the solution valley. In addition, since the trajectory optimizer is discretizing in time as opposed to space, there is no lower bound on how fine the representation can be in regions of rapidly changing dynamics. Lastly, the value function constructed according to the proposed method estimates the cost-to-go of states to be much higher; this is to be expected as the trajectory optimizer is a nonlinear optimization. This will be corrected as more iterations are performed and poorly performing funnels are replaced.

Trajectory Centric Reinforcement Learning [7] and LQR-Trees [49, 60] are the two pieces of work closest to our method. While Trajectory Centric Reinforcement Learning (TCRL) uses the Bellman Residual to construct optimal value functions from sampled trajectories, it does not provide a principled way for sampling new trajectory start points. It also assumes that all trajectories end at the goal. While this is a reasonable assumption for the inverted pendulum, trajectory optimizers will have an easier time reaching subgoals in high dimension nonlinear dynamics. Further, TCRL uses the iLQR [59] to approximate the value function of the trajectory. While this is reasonable for approximation, it says nothing about the stability of points off the trajectory. LQR-Trees is the direct predecessor to this work, sharing the most in common with the approach outlined here. However, it is missing any notion of optimization: the policies returned by it are stable and will reach the goal, but they have no way to improve performance. The proposed approach tackles this by combining Primbs' multiplier with the Bellman Residual approach of TCRL to optimize the performance of the resulting composite policy.

In this chapter, we introduced a new algorithm that constructs optimized stable policies. The performance of the resulting policies was improved by minimizing the maximum sampled value discrepancy. This enables symbols to be constructed that leverage a more optimal controller. In the next chapter, we demonstrate how to extend symbolic representations to parameterized controllers.

Chapter 4

Synthesizing Symbolic Representations for Planning with Parameterized Skills

A defining characteristic of intelligent robots is the ability to generate goal-oriented behavior for a wide variety of tasks. One approach to generating such behavior is task-level planning, whereby a robot reasons about sequences of predefined motor skills to find a sequence that reaches a goal with high probability. Task level planning is fast and capable of considering relatively long planning horizons. However, it depends on the availability of a composable set of motor skills, and an abstract representation that supports efficiently reasoning about composing the motor skills.

Previous work [36] has addressed the question of how to construct abstract representations suitable for planning with a given collection of motor skills, providing a principled method for constructing representations that support the operations necessary for probabilistic planning. However, that work assumed that the motor skills are simple black-box controllers. Due to the continuous nature of certain tasks, such controllers are too inflexible to support the range of dexterous behaviors required of intelligent robots that must interact with a complex world.

Existing approaches for generating more flexible behavior find either detailed low-level motion plans [38], which is generally only feasible for short-horizon tasks or use higher-level actions but still reason in detail [26], which performs well but still requires complex planning to find medium-horizon plans. We propose an alternative approach that plans using more flexible *parameterized motor skills* [13] [30] [34] [40] where the behavior of each high-level action is modified by a real-valued parameter vector. Here, the robot must select both the motor skill to execute at a particular time and additionally select appropriate parameters for it.

We extend existing work on constructing abstract representations [36] to support parameterized motor skills and specify conditions which separate the selection of motor skills from the parameterization of those skills. This extension results in a relatively simple discrete abstract representation for planning, which is followed by a fixed parameter selection process.

We demonstrate the effectiveness of our framework by first learning a symbolic representation for a virtual domain based on Angry Birds and then create a symbolic representation for a robot manipulation task.

4.1 Background

4.1.1 Parameterized Skills

A parameterized skill is a motor controller which takes as input a parameter vector and generates behavior that varies based on the parameter's value. We describe the robot's task as a parameterized action Markov decision process (PAMDP) [40], described by a tuple:

$$(S, A, R, T, \gamma)\,,$$

where $S \subseteq \mathbb{R}^n$ is a set of states; A is a set of parameterized skills (described next); R is a reward function describing the reward received for executing a parameterized action in a particular state; T is a model of the transition dynamics; and γ is a discount factor.

Following the *options framework* [58], each parameterized skill is described by a tuple:

$$(\pi_o, \Theta_o, I_o, \beta_o)\,,$$

where $\pi_o(a|s, \theta)$ is a policy that returns the probability of the agent executing action a in state s, given input parameter θ; $\Theta_o \subseteq \mathbb{R}^m$ is a range of acceptable policy parameter vectors; I_o is the set of states from which the motor skill can be executed;[1] and $\beta_o(s)$ is a

[1]Note that we assume that any parameter value $\theta \in \Theta_o$ can be selected for any state in I_o; we leave removing this assumption to future work.

termination condition, which determines the probability with which the motor skill should terminate in a given state s.

In a MDP, the agent aims to finds a policy, π, which maps from states to a distribution over actions. In a PAMDP, the agent must find a policy which maps from states to a distribution over, (a, θ), where a is a parameterized action and θ is its parameterization.

Masson et al. [40] introduced the use of parametrized action Markov Decision Processes to include parameterized skills in the reinforcement learning setting. They present a reinforcement learning algorithm that learns separate policies for selecting skills and skill parameters, given a state. This method was successfully applied to a simulated robot soccer domain but is unsuitable for use on physical robots as it is model-free, and consequently requires a great deal of experience to learn a good policy.

4.1.2 Planning with Motor Skills

If we are to construct a representation that enables a robot to plan using a set of motor skills, it must support the operations required to compute the probability with which a plan succeeds, and the expected reward received should execution be successful. Prior work [36] has shown that the agent must represent two important pieces of information for each motor controller: the *precondition*, $\text{Pre}(o, s) = P(s \in I_o)$, which estimates the probability that a motor skill o can be run from state s, and the *image* $\text{Im}(o, Z)$, which returns a distribution over states, expressing the probability of the agent entering state s' after executing motor skill o from a start state drawn from the distribution Z. Given these, the robot can compute the probability of a plan succeeding by multiplying the probability of successfully executing each action:

$$p_i = \int \text{Pre}(o_i, s) Z_i(s) ds,$$

where Z_0 is the start distribution and each $Z_i = \text{Im}(o_{i-1}, Z_{i-1})$. Similarly, we can compute the expected reward of executing action o_i from distribution Z_i as $r_i = \int R(s, o_i) Z_i(s) ds$.

A symbol is an abstraction applied to preconditions and images. Here we differentiate between two types of symbols. A *distributional symbol*, σ_z, names a distribution, Z, over states. Distributional symbols are used to estimate the state distributions of images. A *conditional symbol*, σ_E, is the name associated with a function $P(C(s) = True)$ which returns the probability that condition C holds true at state s. Conditional symbols are probabilistic classifiers used to approximate the probability that a state belongs to the precondition of a skill.

A symbolic representation for motor skill planning can be constructed by first creating a pair of symbols for each skill. There is a minimum of at least one conditional symbol to represent the skill's precondition and one distributional symbol to represent the skill's image. These pairs of symbols are used to compute every reachable p_i and r_i value as defined above, and finally used to construct a forward model for obtaining p_i and r_i using only the symbols themselves (and not the classifiers and distributions that they name). However, the ability to do so requires that there be only finitely many such p_i and r_i values, which in turn means that there can be only finitely many reachable image distributions.

This condition is not true in general, but it does hold for some common types of motor skills. The most important one is the *subgoal skill*, where a feedback motor controller drives the state toward some subgoal set of states, and the feedback process eliminates any dependency on the state from which execution started. In that case we can replace any instance of the image operator with an *effect distribution* which does not depend on start state:

$$\text{Im}(o, Z) = \text{Eff}(o).$$

As a result, there are only finitely many reachable image distributions—in fact, exactly as many as there are motor controllers—this allows the construction of a finite representation.

Of course, in many real-life scenarios, the subgoal property does not hold. In practice, we can often *partition* a motor skill's initiation set such that the subgoal property approximately holds when execution only occurs from each of the resulting partitions. In this case, we can build a finite model by considering each partition a distinct motor skill.

4.2 Building Symbolic Models for Probabilistic Planning with Parameterized Skills

Due to the uncertain nature of robot sensing and action we are interested in *probabilistic planning*. Probabilistic planning is the process of creating a plan and evaluating its probability of success. Formally, a plan is defined as a sequence of actions intended to accomplish a goal from a particular starting configuration. In order to perform probabilistic planning with parameterized skills, we must extend the previous definition to handle motor skill parameters and thus define a plan as follows:

Definition 1. A plan $p = \{o_1(\theta_1)....o_n(\theta_n)\}$ from a start state distribution Z is a sequence of parameterized skills and their parameters $o \in O, \theta \in \Theta_o$, to be executed from a state drawn from Z.

This plan definition maintains the important aspects of the previous definition. The plan is still a sequence of actions and dependent on the start state distribution, however each plan also includes the parameter for each action. It is important to note that each motor skill has a bounded range of parameters from which it can draw. For example, o_1 may accept a parameter value between 0 and 5, while o_2 may accept a parameter value between 25 and 100. The start state distribution, Z, is required because without a start state the probability of plan completion cannot be evaluated. As before, the probability of plan completion is obtained by computing the probability of successfully completing each successive motor skill execution in the plan, and multiplying these probabilities together.

Konidaris et al. [36] prove that the precondition and image are necessary and sufficient for symbolic planning of unparameterized motor skills. Here we assume that the preconditions of parameterized motor skills have no dependency on the parameter, and thus are the same as previously defined. However, the image of a parameterized motor skill is dependent on the parameter and must be redefined. We therefore define the *parameterized probabilistic image operator*.

Definition 2.

$$\begin{aligned}\text{Image}(Z,o,\theta) &= \frac{\int_S P(s'|s,o,\theta)Z(s)P(s\in I_o)ds}{\int_S Z(s)P(s\in I_o)ds} \\ &= P(s'|o,\theta).\end{aligned}$$

The image is important for calculating the probability of skill j being executable after skill i has completed. The probability of the image of skill i overlapping with the initiation set of skill j is the probability of *skill intersection*:

Definition 3.

$$\text{Inter}(o_i,\theta_i,o_j,Z) = \int_S \text{Image}(Z,o_i,\theta_i)P(s\in I_{o_j})ds. \tag{4.1}$$

These definitions allow us to now prove that the probability of plan completion can be constructed from these parts.

Theorem 1. Given a PAMDP, a model of the probabilistic precondition of each parameterized skill and the parameterized probabilistic image, the probability of completing a plan, (Z,p), can be determined.

Proof

Consider an arbitrary plan tuple (Z_0,p), with plan length n. The probability of successfully executing plan p from starting distribution, Z_0, is

$$\prod_{j=0}^{n-1}\text{Inter}(o_j,\theta_j,o_{j+1},Z_j)\int_S Z_0P(s\in I_{o_0})ds.$$

Where Z_{j+1} can be found via $Z_{j+1}=\text{Image}(Z_j,o_j,\theta_j)$, for $j\in\{1...n\}$. $\square$

Starting with a distribution of states, the agent can repeatedly apply the image operator to obtain the distribution of states after taking an action. Using these images the agent can determine the probability of execution of each parameterized skill. It can then multiply all of these probabilities together to obtain the probability of success of a plan. However, we are interested in finding the best plan given the motor skills available to the robot, not only the probability of success of a particular plan. Thus we define the *Skill Optimal Plan*:

Definition 4.

$$\underset{\theta_0 \ldots \theta_n}{\operatorname{argmax}} \prod_{j=0}^{n-1} \operatorname{Inter}(o_j, \theta_j, o_{j+1}, Z_j) \int_S Z_0 P(s \in I_{o_0}) ds. \tag{4.2}$$

This is the set of motor skill parameters which maximize a given plan's probability of success. In order to maximize the probability of a plan's success, the planner must select a sequence of skills and for each skill the parameter which maximizes the probability of that skill. In Figure 4.1 the initiation classifier is represented on the left and the parameterized image of the skill is the yellow ribbon on the right. The planner must select a θ which results in the maximum probability that the image of the skill is in the initiation classifier of the next skill. By selecting the maximum probability parameter setting the representation for a skill optimal plan remains discrete, as we prove next.

Theorem 2. Assuming that only the probability of a successful plan execution is of interest and given n parameterized motor skills, the number of symbols required to represent a skill optimal plan is $\mathcal{O}(n^2)$.

Proof

For each skill intersection the only parameter setting which needs to be represented by a symbol is one which maximizes the probability of skill intersection. All other parameter settings will result in a lower probability and thus can be ignored without impacting the planner's ability to find a skill optimal plan. This is because a skill optimal plan must be constructed of skills with parameters tuned to maximize the probability of the following skill. This is a consequence of the multiplicative nature of our plan definition. Lastly, there are n^2 skill intersections, each with one parameter setting which maximize the probability thus a discrete representation with $\mathcal{O}(n^2)$ symbols can represent all necessary information.

□

Now that the number of symbols has been bounded, a parameter must be selected. In practice we must search for the θ which maximizes the probability of $\text{Inter}(o_j, \theta_j, o_i, Z)$ for all $o \in O$. This is achieved by uniformly sampling over the range of θ and estimating the intersection probability for each sample. This approach will always incur some error between the estimated max probability and the true max probability, $|\tilde{m} - m|$. Next we bound this approximation error when the intersection is assumed to be Lipschitz continuous. In order for this assumption to be reasonable the perception and dynamics of the robot need to be Lipschitz. Many mobile manipulation platforms have dynamics which are Lipschitz continuous. As for perception, in many cases the environment being observed can be treated as Lipschitz by linearly interpolating between any discontinuities.

Lemma 1. Given a Lipschitz continuous skill intersection, with Lipschitz constant K, and the distance between parameter sample values, Δ, and d the dimensionality of the parameter space, the approximation error of the maximum is $\frac{\sqrt{d}K\Delta}{2}$.

Proof

First, uniformly sample the parameter space of θ by Δ. For all of these points find the probability of intersection. Next, define $\tilde{m}$ as the max taken over all of the skill intersection samples and let m be the true max. The approximation error is defined as $|\tilde{m} - m|$. Given an $\tilde{m}$, an m can be constructed that does not change the selection of $\tilde{m}$ and maximizes the approximation error. Starting at $\tilde{m}$ increase at a rate of K for $\frac{\sqrt{d}\Delta}{2}$ while heading in the direction of a point that is $\sqrt{d}\Delta$ away. At $\frac{\sqrt{d}\Delta}{2}$ change the rate to $-K$. At $\sqrt{d}\Delta$ from its start point the function will be equal to $\tilde{m}$. The point $\sqrt{d}\Delta$away is the furthest point on the hypercube that is defined by Δ spaced uniform sampling. Thus the furthest m can be from $\tilde{m}$ is $\frac{\sqrt{d}\Delta}{2}$. Finally, since the error can only grow by K per unit of distance, the maximum error is $\frac{\sqrt{d}K\Delta}{2}$.

□

Using this approximate max operator we now bound the error of an approximate finite symbolic representation for achieving the skill optimal plan.

Theorem 3. Given a finite number of parameterized motor skills with Lipschitz continuous skill intersections, a finite number of bounded parameters for each motor skill, and the approximate max operator, the error of an approximate finite symbolic representation is upper bounded by $\frac{\sqrt{d}K\Delta n}{2}$, where n is the plan length.

Proof

The probability of success of the optimal plan is defined as $\prod_{i=0}^{n} P_i$, while the probability of success of the approximate plan is at most $\prod_{i=0}^{n}\big(P_i - \frac{K\Delta}{2}\big)$. For clarity, we define $\delta = \frac{\sqrt{d}K\Delta}{2}$. Next we show by induction that

$$\prod_{i=0}^{n}(P_i - \delta) \geq \prod_{i=0}^{n} P_i - \delta n.$$

The base case is when $n = 1$

$$(P_1 - \delta) \geq P_1 - (\delta)(1).$$

We assume it holds for $n = j$ and show that it holds for $n = j + 1$

$$\begin{aligned}
\prod_{i=0}^{j+1}(P_i - \delta) = & \quad (P_{j+1} - \delta)\prod_{i=0}^{j}(P_i - \delta) \\
\geq & \quad (P_{j+1} - \delta)((\prod_{i=0}^{j} P_i) - \delta j) \\
= & \quad P_{j+1}\prod_{i=0}^{j}(P_i) - P_{j+1}\delta j - \delta\prod_{i=0}^{j}(P_i) + \delta^2 j \\
\geq & \quad P_{j+1}\prod_{i=0}^{j}(P_i) - P_{j+1}\delta j - \delta\prod_{i=0}^{j}(P_i)
\end{aligned}$$

$$\begin{aligned} &\geq P_{j+1}\prod_{i=0}^{j}(P_i) - \delta j - \delta \\ &= \prod_{i=0}^{j+1}(P_i) - \delta(j+1) \\ &= \prod_{i=0}^{j+1}(P_i) - \tfrac{\sqrt{d}K\Delta(j+1)}{2}. \end{aligned}$$

Thus the difference between an approximate plan and the true plan is upper bounded by $\frac{\sqrt{d}K\Delta n}{2}$

$$\frac{\sqrt{d}K\Delta n}{2} \geq \prod_{i=0}^{n}(P_i) - \prod_{i=0}^{n}(P_i - \frac{\sqrt{d}K\Delta}{2}).$$

□

The error due to the approximation grows as the plan gets longer, and as the intersection becomes more sensitive to the change in motor skill parameters. In addition, the bound explains how Δ is related to these two quantities. This relationship shows that we can sample less (i.e. have a large value for Δ) for cases where plans are short, or parameter sensitivity is low.

Unfortunately, even in cases where Δ can be large, calculating the skill intersection in general is computationally expensive. Konidaris et al. [36] use the subgoal property to approximate the intersection. Here we generalize the subgoal property to include parameter values, we propose the *strong parameterized subgoal property*:

Definition 5.

$$\begin{aligned} \text{Image}(Z, o, \theta) = \text{Effect}_{\theta_l,\theta_u}(o) \\ \forall\theta, \theta_l \leq \theta \leq \theta_u. \end{aligned}$$

This property holds when the image and starting distribution are conditionally independent given the parameters of the controller. If this property holds over the entire range of θ, then the motor skill is not parameterized. In general, this property is expected to only

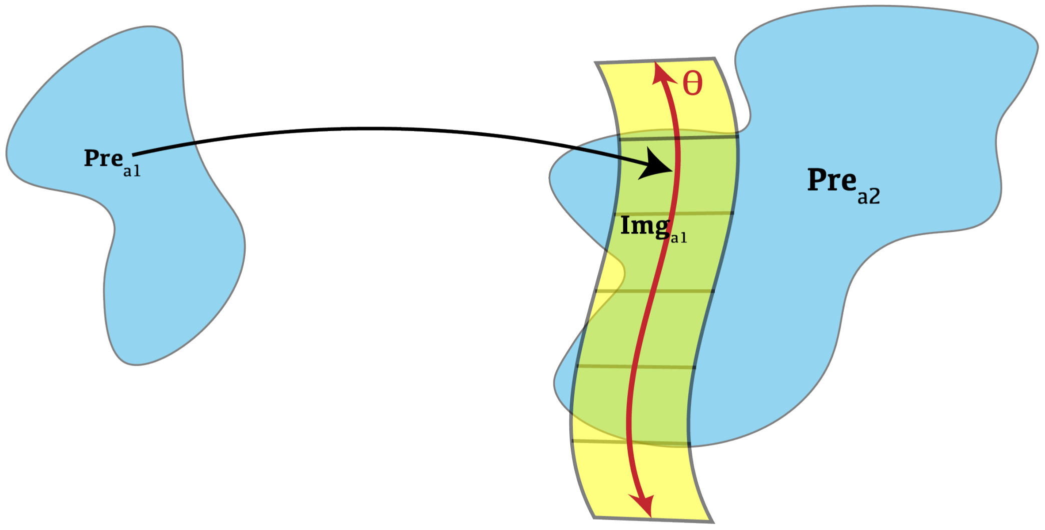


Figure 4.1: The yellow ribbon is the set of all image distributions. The image varies with the value of θ along the red curve. The goal is to select a range of θ such that the effect distribution lies inside the precondition of the next action with highest probability. Visually, this would be regions where the yellow ribbon is contained by the blue precondition.

hold over segments of the parameter space. When this property does hold, an effect defined over a parameter range can approximate the image in (4.1). In practice, this property may be overly restrictive, as it requires the image and effect to be the same distribution. Thus we introduce the *weak parameterized subgoal property:*

Definition 6.

$$\int_s \text{Image}(Z_i, o_i, \theta_i)(s) P(s \in I_{o_j}) ds = \int_s \text{Effect}_{\theta_l,\theta_u}(o_i)(s) P(s \in I_{o_j}) ds \quad \forall \theta_i, \theta_l \leq \theta_i \leq \theta_u, \forall o_j \in O.$$

This ensures that the probability of executing any following action is necessarily the same, but does not require that the two distributions are the same. Since the effect is a set of images the number of intersections that must be calculated is reduced. In addition, computationally friendly distributions can be chosen as long as they satisfy the subgoal property. The downside of using the effect is that it may not represent the set of images well, unless the weak parameterized subgoal property holds. Thus the combination of Theorem 3 and the weak parameterized subgoal property results in a discrete representation which supports efficient planning and has bounded error from the optimal.

4.3 Experiments

We applied our framework to two different domains: a virtual domain, which demonstrates the compression and expressiveness of the representation, and a robot manipulation task, which demonstrates its performance under sparse and noisy data.

4.3.1 Catapult Domain

The catapult domain is a model of the popular game, Angry Birds (see Figure 4.2). In this instantiation, three walls can be knocked down, and two obstacles that only fall when their corresponding wall falls. The agent is provided with one parameterized behavior, $shootAtAngle(\theta)$. The parameter, θ, defines the firing angle of the catapult and the goal of the agent is to knock down all the walls.

Data Collection

An agent uniformly at random selects θ from 0 to 1.57 until 10,000 parameterized skill executions have been collected. One consequence of the random agent is that in some cases a wall can be perturbed without knocking it over. These samples provide additional information about how the agent can knock down a wall that is not in the canonical start pose. The state space consists of the action parameter used, θ, and the x,y, and ϕ of each wall before and after the action. ϕ is the angle, from vertical, of a wall.

Symbol Creation

In order to prepare the data for symbol creation, the data was clustered using DBSCAN [55, 22, 44] with an $\epsilon = 0.32$ and the minimum number of samples set to 100. Using randomized logistic regression feature selection was performed on each cluster. A feature was kept if it appeared in more than 55% of 500 random re-samplings of the training data. The precondition classifiers were trained on the clustered data using only the selected features. The classifiers used were Probabilistic KSVMs [14].[2] The effect distributions were modeled with conditional kernel density estimators [41].

Each effect distribution is checked against the other effect distributions using a 2-sample Kolmogorov–Smirnov-test to ensure that the distributions are significantly different. If they are not, their grounding sets are merged into a single symbol.

Operator Creation

The clustered data was used to create a list of observed cluster transitions. Clusters which occurred before an action require a conditional symbol. Clusters which occurred after an action require a distributional symbol. In this case because there is only one option, all of the operators come from partitioning the initiation set of the option. The probability of the operator follows the methods outlined in Section 4.2.

Planning

The learned operators are transformed into the Probabilistic Planning Domain Description Language (PPDDL) [66] so that off-the-shelf planners can be used. During this transformation any operator with a probability less than 0.25 was downgraded to impossible and removed. The start state and goal are defined in PPDDL and the problem is posed to the mini-gpt [11] planner, an off-the-shelf MDP planner, with two heuristics applied, zero, and min-min-lrtdp.

[2]The radial basis kernel was used.

Results

The results of the experiments demonstrate that the symbolic representation for the virtual domain compresses the ambient state space of the domain to an abstraction that contains the salient features necessary for accomplishing both the original task and other novel tasks. One way to measure the compressed nature is to inspect the size of the symbolic representation versus a discretization. Only 16 symbols were created for the virtual domain. This is a significant reduction in representation size over even a coarse discretization of the combined action and state space.[3] This symbolic space allows very efficient planning, a plan for knocking down all of the walls can be found in 4.5ms.

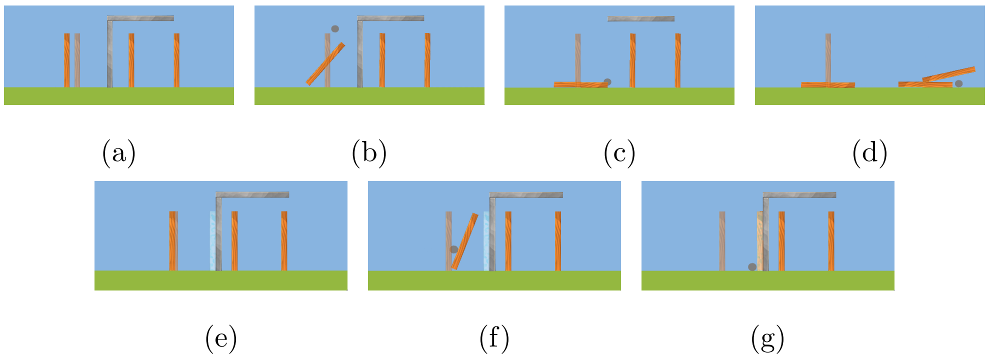


Figure 4.2: Two different goals from the same set of symbols. The first goal is to knock down all the walls (a-d). a.) The transparent wood wall is the canonical start pose. The first wall's position is now sampled from a start state distribution. b.) Just after impact. c.) After impact. d.) All walls down. The second goal is to move the first wall without knocking it down (e-g). e.) The translucent blue wall is the target location. f.) Moving the wall g.) The wall has been successfully moved from its starting position to its new goal.

A common trade-off for increasing performance is to decrease the expressiveness of a representation, however for the learned symbols there is enough expressiveness to accomplish multiple tasks. The symbols allow the planner to handle not just narrowly defined start states, but any state which is represented by the symbols. This is demonstrated by Figure 4.2(a) where the initial start state has been sampled from one of the symbols.

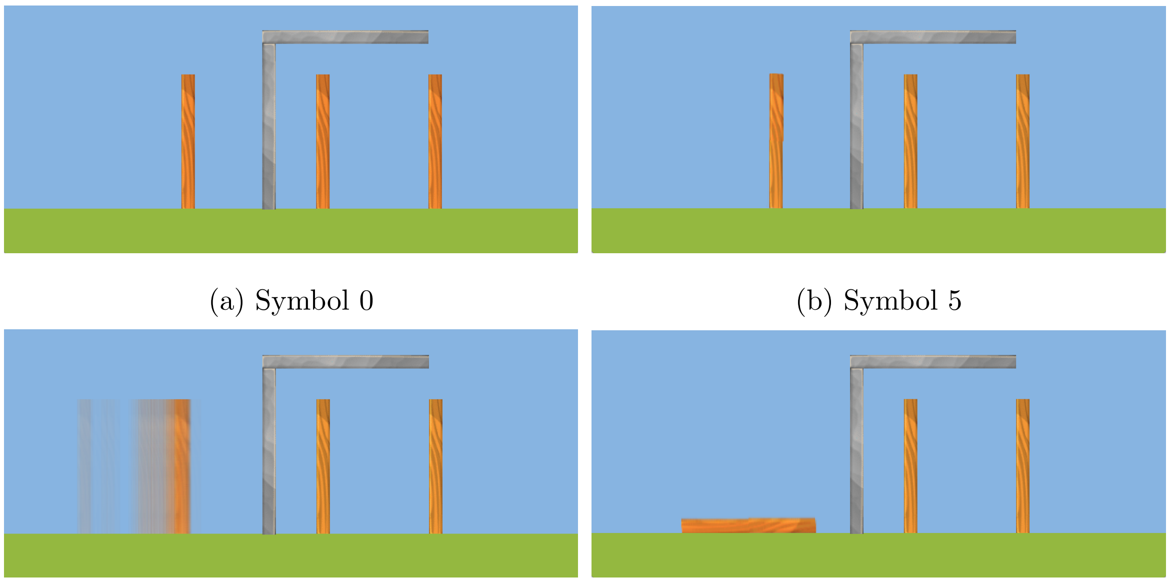
[3]For example, if each dimension had be discretized into 10 buckets, the resulting matrix would have 10^{10} elements.

Figures 4.2(b)-(d) demonstrate that the planner is capable of planning from different start states. Similarly, new goal states can be specified without requiring a change in representation. In Figure 4.2(e) the goal has been changed from knocking down the walls to moving the first wall against the first obstacle. The planner is able to select a θ which accomplishes this goal even under initial state uncertainty.

```
(:action KnockDownFirstWall
 :precondition (and (symbol_0) (symbol_5))
 :effect (probabilistic 0.96 (and
 (symbol_1)(symbol_6)
 (symbol_9)(symbol_13)
 (not(symbol_2))(not(symbol_8))
 (not(symbol_11))(not(symbol_0))
 (not(symbol_10))(not(symbol_4))
 (not(symbol_7))(not(symbol_15))
 (not(symbol_12))(not(symbol_14))
 (not(symbol_5))(not(symbol_3)))))
```

Figure 4.3: The PPDDL representation of the symbolic operator for knocking down wall 1. The operator changes the logical state of symbols from true to false. These changes in logical state mirror the change in the state space.

The text representation of a symbolic operator is shown in Figure 4.3. This is one of the operators which knocks down the first wall. Its preconditions require that the wall is in its canonical start pose. Symbol 0 is a distribution defined over the X-position of the first wall and is visible in Figure 4.4(a). Symbol 5 is in Figure 4.4(b) and is a distribution defined over the y and ϕ of the wall. Thus the precondition symbols are interpreted as the first wall being upright and in a specific X-position. The effects of the operator are listed after ":effect". The form "probabilistic 0.96" says that with probability p=0.96 the following effects will be set. There are four symbols set to true by this operator, two of which are over the state of the second wall, and two of which are over the first wall. For the sake of clarity we will ignore Symbols 13 and 9, and focus on symbols 1 and 6 which describe the new state of wall 1. Symbol 1 is Figure 4.4(c), thus the wall will have an X-position before the first obstacle. Symbol 6 is Figure 4.4(d), which shows the wall will be laying down. The rest of the symbols become false because they conflict with one of the positive symbols. For example, it is impossible for the wall to be in the state described by (symbol 0, symbol 5) and the state described by (symbol 1, symbol 6).



(a) Symbol 0

(b) Symbol 5

(c) Symbol 1

(d) Symbol 6

Figure 4.4: Symbols used for constructing the operator in Figure 4.3. In a) there is a symbol defined over the X position of the first wall. Here it is a very tight distribution. In b) a symbol defined over the Y position and ϕ of the first wall, c) a symbol defined over the X position of the first wall and has a different mean than (a) and a higher variance. In d) a symbol is defined over the Y position and ϕ of the first wall, which has a different mean than (b).

These results demonstrate that the proposed process is able to capture the salient features of continuous variation of state and action space in a discrete representation. This discrete representation allows for the creation of a probabilistic plan with a high chance of success, and the reduction of representation size, and can be used in novel situations. The combination of these factors demonstrates that the symbolic representation created in this fashion is a powerful and useful representation.

4.3.2 Robot Manipulation Task

In this experiment our mobile manipulator, Anathema, was used to perform an idealized form of the "Make Coffee" task (Figure 4.5). In this instance of the task, the coffee cup is in a locked cabinet. The cabinet can be opened by rotating a handle to a specific angle and pulling up on the handle. In addition, the coffee cup must be placed in the coffee machine within a specific angular range. If the cup is placed in the coffee machine with handle inwards the cup completely obscures the button (Figure 4.5(d)), preventing the robot from pressing the button. Lastly, the goal is to have the cup in the coffee machine while the button is pressed.[4]

Anathema has four behaviors that can be used for this task $PickPlaceCup(\theta)$, $UnlockCabinet(\phi)$, $OpenCabinet()$ and $PressButton()$. $PickPlaceCup(\theta)$, approaches the cup, picks it up, and then places it in the coffee machine. It accepts one parameter θ, which is the approach angle of the hand relative to the cup. $UnlockCabinet(\phi)$ grasps the cabinet handle, rotates it, and then lets go of the handle. It also accepts one parameter, ϕ, which is the angle of the handle relative to its starting position. $OpenCabinet()$ grasps the handle and attempts to pull it up. $PressButton()$ approaches the coffee machine from above and attempts to press the coffee machine button.

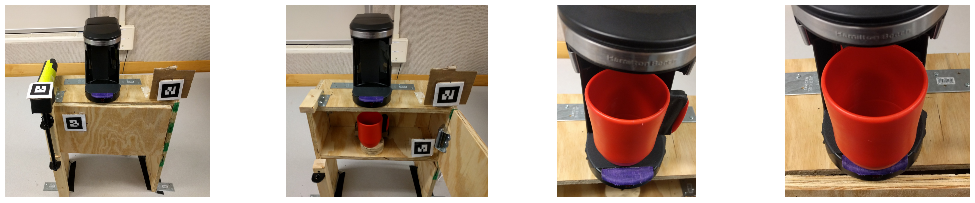

(a) (b) (c) (d)

Figure 4.5: The experimental setup: a.) the cupboard is closed and unlocked, b.) the cupboard is open c.) the cup is in the correct position, d.) the cup is blocking the button.

[4]In order to avoid mixing liquids and robots, no coffee is actually made in this task.

Methods

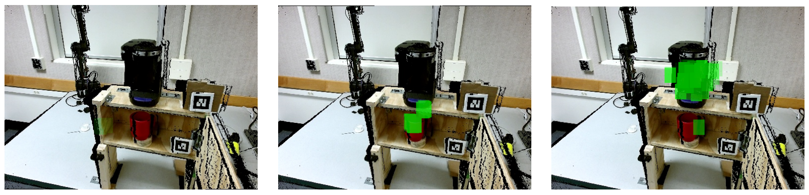
The behaviors in this experiment use MoveIt! [57], AprilTags [64], and the cmvision [12] blob detector. The state space is the position of the cup in image space, the area of the cup, the position of the button in image space, the visible area of the button, the locations of 4 AprilTags, and whether the coffee button is pressed. In order to determine whether the button was pressed, the coffee machine was outfit with a Raspberry Pi and a momentary switch which turned on when the button was depressed. ROS [47] was used to provide communication between all of the various components.

Symbol Creation

For primitive skills, the state space was clustered using DBSCAN with $\epsilon = 0.9$ and a minimum of 5 samples. For parameterized skills regression clustering was used. Feature selection was performed by univariate feature selection, the highest scoring four features, according to ANOVA F-value, were kept. Conditional symbols were modeled by SVMs with radial basis functions fit using grid search. Distributional symbols were modeled by conditional kernel density estimators. θ probability clustering was performed using xgboost [15] with 100 trees and $\alpha = 1$. Data was collected by performing 34 trials of $ButtonPress()$, 44 trials of $OpenCabinet()$, 40 trials of $PickPlaceCup(\theta)$, 54 trials of $UnlockCabinet(\phi)$, and 30 no-ops. Lastly, planning with the symbols was performed using mini-gpt, with two heuristics applied, zero, and min-min-lrtdp.

Results

An example of symbols created for this domain are illustrated in Figure 4.6. Symbol 8 is a distribution over X and Y positions of the cup which were viable locations for pickup. This distribution is tight because the hand went to the same position, regardless of actual cup position. Thus if the cup was outside of this distribution the gripper would completely miss it. Symbol 9 is the cup position distribution from which $PressButton()$ could be executed and the goal state reached. The highest density of positions is away from the button, which



(a) Symbol 7 (b) Symbol 8 (c) Symbol 9

Figure 4.6: Symbols learned from robot execution for cup position. a.) Symbol 7 has a low probability of cup visibility. b.) Symbol 8 is a distribution over cup pickup positions. c.) Symbol 9 is the $PressButton()$ precondition distribution.

makes sense because the button must be uncovered in order for it to be pressed. Overall the symbols capture where the cup needs to be for certain operators to take place.

Figure 4.7 and the included video demonstrate that the robot is capable of using the constructed symbols to create a plan which performs the desired task. The plan was constructed in 0.22s and consisted of four symbolic operators. They were, in order $UnlockCabinet(66 - 72)$, $OpenCabinet()$, $PickPlaceCup(-15 - 12)$, $ButtonPress()$ with their parameters uniformly at random selected from the provided ranges. In order to better understand how the symbols are representing the state space, we will follow how the cup position distribution changes throughout the plan. First, $UnlockCabinet(66 - 72)$ requires that the environment be in a configuration that could be sampled from symbol 7, i.e. that no cup is visible. This action has no impact on cup position. Thus the second action, $OpenCabinet()$, also expects the environment to be sampled from symbol 7. The result of $OpenCabinet()$ is expected to be a cup position that could be sampled from the effect distribution that is contained within symbol 8. Next, $PickPlaceCup(-15 - 12)$ expects the cup to be sampled from symbol 8, and places the cup in a position that could be sampled from symbol 9. This enables the last action, $ButtonPress()$, which presses the button and completes the task. This symbolic plan demonstrates that a continuous task can be accomplished using parameterized behaviors that were parameterized ahead of time via a discrete planner.

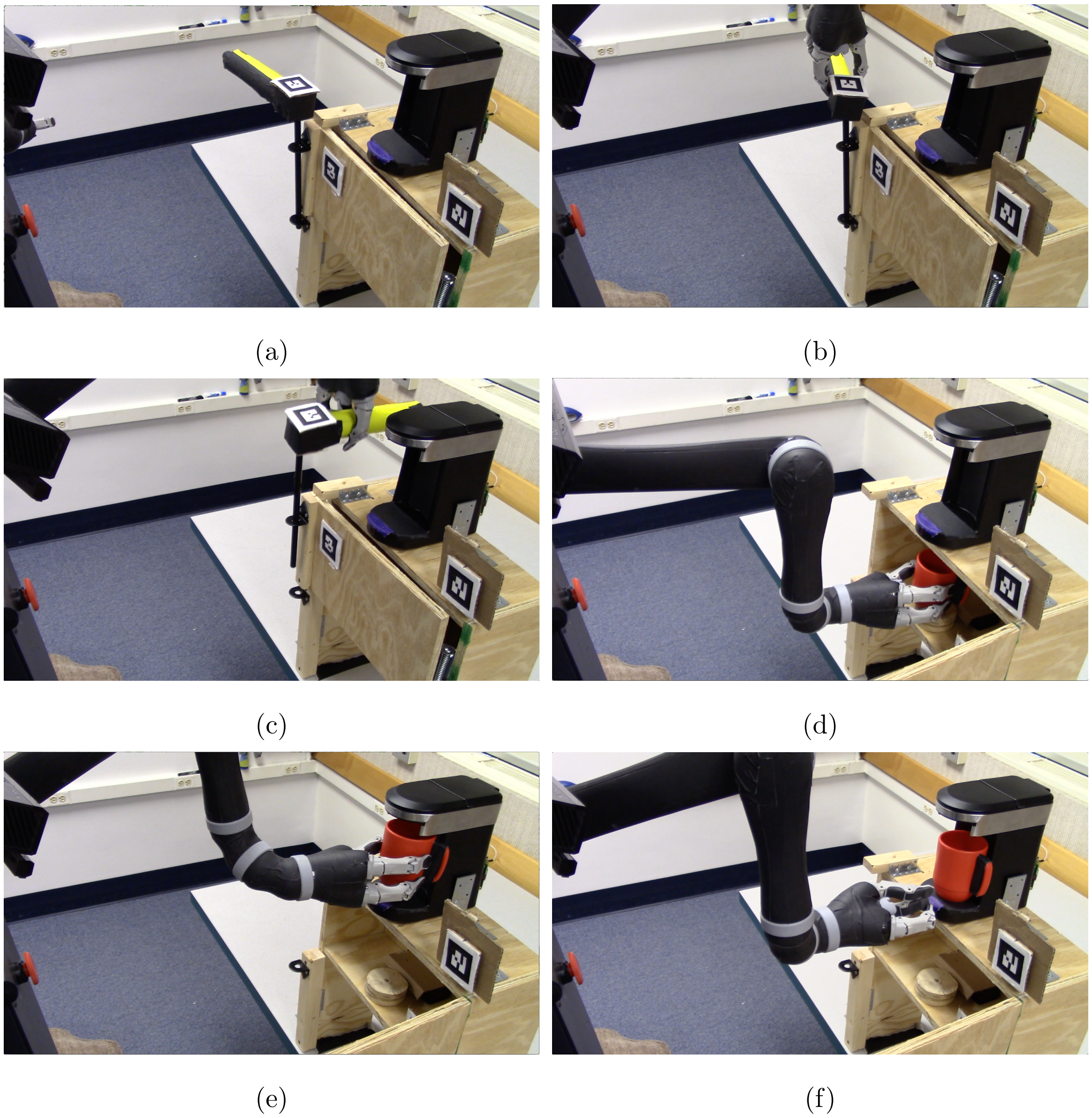


Figure 4.7: Execution of the symbolic plan: a.) the environment at start, b.) unlocking the cabinet door, c.) opening the cabinet door, d.) picking up the coffee cup, e.) placing the coffee cup, and f.) pressing the coffee machine button.

4.4 Related Work

Others have addressed the issue of combining low level actions and high level planning. Previous approaches can be split into two main categories. The first category is approaches which stitch together several configuration space representations of tasks and attempt to find a motion plan in this combined space [26][8]. Unfortunately, this may create portions of the combined space which have volume zero and thus are difficult to sample from but necessary to pass through in order to complete the task. In addition, these methods are not able to leverage advances in symbolic planning.

The other category uses abstractions of low level actions to search for a solution by leveraging symbolic planners. Within this category there are methods which impose a symbolic representation [18][29][45] and those that construct a symbolic representation from data. Our work is most closely related to the symbol creation methods. Gaudioso et al. [24] use Answer Set Programming to create abstractions but their approach is limited to discrete state and action spaces. The closest to our work is Jetchev et al. [28] they build a symbolic abstraction that maximizes reward for a relational reinforcement learning problem. However, they construct a predicate for each object, whereas our predicate is applied to all of the feature data that is available, nor do they address the complexity of parameterized motor skills.

In this chapter, we explored a symbolic representation for parameterized skill planning. First, we proved that the probability of a plan can be calculated if it is constructed of parameterized skills that obey the parameterized subgoal property. Next, we proved that a discrete representation can always be used to achieve the skill optimal plan. Then the compressive and expressive qualities of the representation were explored in the Catapult domain. We showed that with only 16 symbols the planner was able to solve the original task and other related tasks. Finally, the performance of the representation was evaluated on a robot manipulation task. The planner was able to perform the task and approximate the probability of the plan successfully. The presented representation is a step toward flexible goal-directed abstract planning for robots. This work also highlights the importance of having the proper amount of detail to construct the skill. For example, if the parameterized skills used in this paper had instead been treated as discrete skills the probability of plan success would have been much lower.

Chapter 5

Conclusion

The three contributions made in this work have improved the state of the art in robot planning. The first tool, IKFlow, enables more robust motion planning by extending the speed and quantity of Inverse Kinematics (IK) solutions for 7+ DoF kinematic chains. This improvement was achieved by representing the IK problem as a generative modeling problem. A normalizing flow generative modeling paradigm was selected because normalizing flows are stable to train and quick to sample. The performance of IKFlow was demonstrated on a number of different kinematic chains and the bounds of its performance were characterized so that it can be utilized where appropriate. One place that IKFlow can be extended is to better leverage the kinematic information that is available, or to in general include more structure from the problem.

The second tool, Optimizing LQR-Trees, provides a method for the generation, concatenation, and optimization of controllers. Inspired by the value iteration approach taken in Reinforcement Learning the performance of the assembled controller was improved. This tool was demonstrated on a nonlinear virtual domain and compared against traditional reinforcement learning approaches. A visual inspection of the value functions and an analysis of the performance demonstrated that the proposed algorithm is optimizing. A proof that establishes this algorithm will work in all cases is left to future work. The work is also dependent on polynomial models for dynamics and the calculation of Lyapunov functions. There has been significant improvement in the sample-based construction of Lyapunov functions that should be explored to extend the generality of the approaches provided here.

Finally, the ability to construct symbols for parameterized controls creates an abstract representation of controllers that is useful for task planning. This work demonstrates that even controllers with a continuous parameter set can be well represented symbolically. The performance of these symbols was analyzed both in virtual and physical domains. These domains demonstrate the generality of the learned symbols. In addition, the physical domain demonstrates that the method is capable of handling complex feature sets. The automatic construction of symbols from controllers and the inverse are avenues of future work that are of particular interest.

Someday a robot will take my car to get me dinner and along the way it will plan abstractly and robustly to handle the situations that arise and do so in an efficient manner. The contributions made here are concrete steps toward more efficient and robust robot planning.

Bibliography

[1] Ahmed R.J. Almusawi, L. Canan Dülger, and Sadettin Kapucu. A New Artificial Neural Network Approach in Solving Inverse Kinematics of Robotic Arm (Denso VP6242). *Computational Intelligence and Neuroscience*, 2016, 2016. ISSN 16875273. doi: 10.1155/2016/5720163. URL `/pmc/articles/PMC5005769//pmc/articles/PMC5005769/?report=abstracthttps://www.ncbi.nlm.nih.gov/pmc/articles/PMC5005769/`.

[2] Barrett Ames and George Konidaris. Bounded-Error LQR-Trees. *IEEE International Conference on Intelligent Robots and Systems*, pages 144–150, 11 2019. ISSN 21530866. doi: 10.1109/IROS40897.2019.8967750.

[3] Barrett Ames, Allison Thackston, and George Konidaris. Learning Symbolic Representations for Planning with Parameterized Skills. *IEEE International Conference on Intelligent Robots and Systems*, pages 526–533, 12 2018. ISSN 21530866. doi: 10.1109/IROS.2018.8594313.

[4] Barrett Ames, Jeremy Morgan, and George Konidaris. IKFlow: Generating Diverse Inverse Kinematics Solutions. *IEEE Robotics and Automation Letters*, 7(3):7177–7184, 6 2022. doi: 10.1109/LRA.2022.3181374.

[5] Brian D. O. Anderson and John B. (John Barratt) Moore. *Optimal control : linear quadratic methods*. Prentice Hall, 1990. ISBN 0136385605. URL `https://dl.acm.org/citation.cfm?id=79089`.

[6] Lynton Ardizzone, Jakob Kruse, Sebastian Wirkert, Daniel Rahner, Eric W. Pellegrini, Ralf S. Klessen, Lena Maier-Hein, Carsten Rother, and Ullrich Köthe. Analyzing inverse problems with invertible neural networks. In *International Conference on Representation Learning*. arXiv, 8 2019. URL `https://arxiv.org/abs/1808.04730v3`.

[7] Chris Atkeson and Benjamin Stephens. Random Sampling of States in Dynamic Programming. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 38(4):924–929, 8 2008. doi: 10.1109/TSMCB.2008.926610. URL `http://ieeexplore.ieee.org/document/4559368/`.

[8] J. Barry, L. Kaelbling, and T. Lozano-Pérez. A Hierarchical Approach to Manipulation with Diverse Actions. *International Conference on Robotics and Automation*, 2013.

[9] Patrick Beeson and Barrett Ames. TRAC-IK: An open-source library for improved solving of generic inverse kinematics. In *IEEE-RAS International Conference on Humanoid Robots*, volume 2015-December, pages 928–935. IEEE Computer Society, 12 2015. ISBN 9781479968855.

[10] Christopher M Bishop. Mixture Density Networks. Technical report, Aston University, 1994. URL `http://www.ncrg.aston.ac.uk/`.

[11] B. Bonet and H. Geffner. MGPT: A probabilistic planner based on heuristic search. *Journal of Artificial Intelligence Research*, 2005.

[12] J. Bruce, T. Balch, and M. Veloso. Fast and Inexpensive Color Image Segmentation for Interactive Robots. *International Conference on Intelligent Robots and Systems*, 2000.

[13] B. Castro, D. Silva, G. Konidaris, and A. Barto. Active Learning of Parameterized Skills. In *International Conference on Machine Learning*, pages 1737–1745, 2014.

[14] C. Chang and C. Lin. LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology*, 2011.

[15] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[16] Akos Csiszar, Jan Eilers, and Alexander Verl. On solving the inverse kinematics problem using neural networks. In *2017 24th International Conference on Mechatronics and Machine Vision in Practice, M2VIP 2017*, volume 2017-December, pages 1–6. Institute of Electrical and Electronics Engineers Inc., 12 2017. ISBN 9781509065462. doi: 10.1109/M2VIP.2017.8211457.

[17] Hongkai Dai, Gregory Izatt, and Russ Tedrake. Global inverse kinematics via mixed-integer convex optimization. *International Journal of Robotics Research*, 38(12-13): 1420–1441, 10 2019. ISSN 17413176. doi: 10.1177/0278364919846512.

[18] N. Dantam, Z. Kingston, S. Chaudhuri, and L. Kavraki. An Incremental Constraint-Based Framework for Task and Motion Planning. *The International Journal of Robotics Research*, 2018.

[19] Jacket Demby'S, Yixiang Gao, and G. N. Desouza. A Study on Solving the Inverse Kinematics of Serial Robots using Artificial Neural Network and Fuzzy Neural Network. In *IEEE International Conference on Fuzzy Systems*, volume 2019-June. Institute of Electrical and Electronics Engineers Inc., 6 2019. ISBN 9781538617281. doi: 10.1109/FUZZ-IEEE.2019.8858872.

[20] Rosen Diankov. *Automated Construction of Robotic Manipulation Programs.* PhD thesis, Carnegie Mellon University, Pittsburgh, 2010.

[21] Tony Duan. GitHub - tonyduan/mdn: Mixture density network implemented in PyTorch., 2019. URL `https://github.com/tonyduan/mdn`.

[22] Martin Ester, Hans-Peter Kriegel, Jiirg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. Association for the Advancement of Artificial Intelligence, 1996. URL `www.aaai.org`.

[23] Randy A Freeman and James A Primbs. Control Lyapunov Functions: New Ideas From an Old Source. In *Proceedings of 35th IEEE Conference on Decision and Control*, 1996. URL `https://pdfs.semanticscholar.org/a187/ad387bc70f00a2a02f5383a1a56f03ef0471.pdf`.

[24] G. Gaudioso, M. Leonetti, and P. Stone. State Aggregation through Reasoning in Answer Set Programming. *IJCAI Workshop on Autonomous Mobile Service Robots*, 2016.

[25] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Alexander Smola, and Bernhard Schölkopf. A Kernel Two-Sample Test. *Journal of Machine Learning Research*, 13:723–773, 2012. URL `www.gatsby.ucl.ac.uk/`.

[26] K. Hauser and J. Latombe. Integrating task and PRM motion planning: Dealing with many infeasible motion planning queries. *International Conference on Automated Planning and Scheduling*, 2009.

[27] Kris Hauser and Yilun Zhou. Asymptotically Optimal Planning by Feasible Kinodynamic Planning in State-Cost Space. *IEEE Transactions on Robotics*, 2016. URL `https://arxiv.org/pdf/1505.04098.pdf`.

[28] N. Jetchev, T. Lang, and M. Toussaint. Learning Grounded Relational Symbols from Continuous Data for Abstract Reasoning. *International Conference on Robotics and Automation*, 2013.

[29] L Kaelbling and T Lozano-Pérez. Hierarchical task and motion planning in the now. In *IEEE International Conference on Robotics and Automation*, pages 1470–1477, 2011.

[30] L. Kaelbling and T. Lozano-Pérez. Learning composable models of parameterized skills. In *International Conference on Robotics and Automation*, Singapore, 2017.

[31] Hyeongju Kim, Hyeonseung Lee, Woo Hyun Kang, Joun Yeop Lee, and Nam Soo Kim. SoftFlow: Probabilistic Framework for Normalizing Flow on Manifolds. Technical report, 2020. URL `https://github.com/ANLGBOY/SoftFlow`.

[32] Seungsu Kim and Julien Perez. Learning Reachable Manifold and Inverse Mapping for a Redundant Robot manipulator. In *International Conference on Robotics and Automation*, pages 4731–4737. Institute of Electrical and Electronics Engineers (IEEE), 10 2021. ISBN 9781728190778. doi: 10.1109/ICRA48506.2021.9561589.

[33] Diederik P Kingma and Prafulla Dhariwal. Glow: Generative Flow with Invertible 1×1 Convolutions. In *32nd Conference on Neural Information Processing Systems*, 2018. URL `https://github.com/openai/glow.`

[34] J. Kober and J. Peters. Policy Search for Motor Primitives in Robotics. *Machine Learning*, 2011.

[35] Ivan Kobyzev, Simon J.D. Prince, and Marcus A. Brubaker. Normalizing Flows: An Introduction and Review of Current Methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(11):3964–3979, 11 2021. ISSN 19393539. doi: 10.1109/TPAMI.2020.2992934.

[36] G. Konidaris, L. Kaelbling, and T. Lozano-Perez. From Skills to Symbols: Learning Symbolic Representations for Abstract High-Level Planning. *Journal of Artificial Intelligence Research*, 61:215–289, 2018.

[37] Jakob Kruse, Lynton Ardizzone, Carsten Rother, and Ullrich Köthe. Benchmarking Invertible Architectures on Inverse Problems. In *Workshop on Invertible Neural Networks and Normalizing Flows*, 2019.

[38] J. Kuffner and S. LaValle. RRT-connect: An efficient approach to single-query path planning. In *International Conference on Robotics and Automation.*, 2000.

[39] Steven Michael LaValle. *Planning algorithms.* Cambridge University Press, 2006. ISBN 0521862051.

[40] Warrick Masson, Pravesh Ranchod, and George Konidaris. Reinforcement Learning with Parameterized Actions. *Association for the Advancement of Artificial Intelligence*, 2016.

[41] T. O'Brien, K. Kashinath, N. Cavanaugh, W. Collins, and J. O'Brien. A fast and objective multidimensional kernel density estimation method: FastKDE. *Computational Statistics and Data Analysis*, 2016.

[42] George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. Normalizing Flows for Probabilistic Modeling and Inference. *Journal of Machine Learning Research*, 12 2019. URL `http://arxiv.org/abs/1912.02762`.

[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf Xamla, Edward Yang, Zach Devito, Martin Raison Nabla, Alykhan Tejani, Sasank Chilamkurthy, Qure Ai, Benoit Steiner, Fang Lu, Bai Junjie, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. Technical report, 2019.

[44] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, G. Louppe, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. 2012.

[45] E. Plaku and G. Hager. Sampling-based Motion and Symbolic Action Planning with Geometric and Differential Constraints. *International Conference on Robotics and Automation*, 2010.

[46] James A. Primbs, Vesna Nevistić, and John C. Doyle. Nonlinear Optimal Control: A Control Lyapunov Function and Receding Horizon Perspective. *Asian Journal of Control*, 1(1):14–24, 10 2008. doi: 10.1111/j.1934-6093.1999.tb00002.x. URL `http://doi.wiley.com/10.1111/j.1934-6093.1999.tb00002.x`.

[47] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. *International Conference on Robotics and Automation*, 2009.

[48] Daniel Rakita, Bilge Mutlu, and Michael Gleicher. STAMPEDE: A Discrete-Optimization Method for Solving Pathwise-Inverse Kinematics. In *International Conference on Robotics and Automation*, 2019.

[49] Philipp Reist, Pascal Preiswerk, and Russ Tedrake. Feedback-motion-planning with simulation-based LQR-trees. *The International Journal of Robotics Research*, 35(11): 1393–1416, 9 2016. ISSN 0278-3649. doi: 10.1177/0278364916647192. URL `http://journals.sagepub.com/doi/10.1177/0278364916647192`.

[50] Hailin Ren and Pinhas Ben-Tzvi. Learning inverse kinematics and dynamics of a robotic manipulator using generative adversarial networks. *Robotics and Autonomous Systems*, 124:103386, 2 2020. ISSN 09218890. doi: 10.1016/j.robot.2019.103386.

[51] Danilo Jimenez Rezende and Shakir Mohamed. Variational Inference with Normalizing Flows. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

[52] Danilo Jimenez Rezende, George Papamakarios, Sébastien Racanière, Michael S. Albergo, Gurtej Kanwar, Phiala E. Shanahan, and Kyle Cranmer. Normalizing Flows on Tori and Spheres. In *International Conference on Machine Learning*, 2 2020. URL `http://arxiv.org/abs/2002.02428`.

[53] Spencer M Richards, Felix Berkenkamp, and Andreas Krause. The Lyapunov Neural Network: Adaptive Stability Certification for Safe Learning of Dynamical Systems. In *Conference on Robot Learning*, 2018. URL `https://arxiv.org/pdf/1808.00924.pdf`.

[54] Alessio Rocchi, Barrett Ames, Zhi Li, and Kris Hauser. Stable simulation of underactuated compliant hands. *Proceedings - IEEE International Conference on Robotics and Automation*, 2016-June:4938–4944, 6 2016. ISSN 10504729. doi: 10.1109/ICRA.2016.7487699.

[55] Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel, Xiaowei Xu, and H.-P Kriegel. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. ACM Trans. Database Syst*, 42(3), 2017. doi: 10.1145/3068335. URL `https://doi.org/10.1145/3068335`.

[56] Steve M. LaValle. Rapidly Exploring Random Trees: A New Tool for Path Planning. Technical report, Iowa State University, 1998.

[57] I. Sucan and S. Chitta. MoveIt! URL http://moveit.ros.org.

[58] R. Sutton, D. Precup, and S Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

[59] Yuval Tassa, Nicolas Mansard, and Emo Todorov. Control-Limited Differential Dynamic Programming. In *IEEE International Conference on Robotics and Automation*, 2014. URL https://homes.cs.washington.edu/~todorov/papers/TassaICRA14.pdf.

[60] Russ Tedrake. LQR-Trees: Feedback Motion Planning on Sparse Randomized Trees. In *Robotics: Science and Systems*, 2009. URL https://groups.csail.mit.edu/robotics-center/public_papers/Tedrake09a.pdf.

[61] Russ Tedrake. *Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation (Course Notes for MIT 6.832)*. 2018. URL http://underactuated.csail.mit.edu/.

[62] Russ Tedrake and Drake Development Team. Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems, 2016. URL https://drake.mit.edu.

[63] Russ Tedrake, Ian R Manchester, Mark Tobenkin, and John W Roberts. LQR-Trees: Feedback Motion Planning via Sums-of-Squares Verification *. Technical report, 2010. URL https://groups.csail.mit.edu/robotics-center/public_papers/Tedrake10.pdf.

[64] J. Wang and E. Olson. AprilTag 2: Efficient and robust fiducial detection. In *International Conference on Intelligent Robots and Systems*, 2016.

[65] Ronald J Williams and Leemon C Baird. Tight Performance Bounds on Greedy Policies Based on Imperfect Value Functions. Technical report, Northeastern University, Boston, 1993. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.35.3281&rep=rep1&type=pdf.

[66] H. Younes and M. Littman. PPDDL1.0: An Extension to PDDL for Expressing Planning Domains with Probabilistic Effects. *Techn. Rep. CMU-CS-04-162*, 2004.

[67] Dianmu Zhang and Blake Hannaford. IKBT: solving closed-form Inverse Kinematics with Behavior Tree. *Journal of Artificial Intelligence Research*, 65, 11 2017.

Biography

Barrett Ames was born in Phoenix, Arizona where he enjoyed his childhood, and where the desert taught him important lessons about life. He obtained his B.S. from Cornell University in Ithaca, NY, where he studied robotics, after which he spent a couple of years as a contractor at NASA-Johnson Space Center where he worked on Valkyrie, ATLAS, and Robonaut 2. After his time at NASA he began his Ph.D. program in Computer Science at Duke University, during which had the privilege of interning at Toyota Research Institute, Millenium Management, and Realtime Robotics, and was awarded the NSF WISeNET and NDSGEG Fellowships. He created and taught the ECE 383: Introduction to Robotics at Duke in the Fall of 2019. Most recently he founded BotBuilt, a company dedicated to improving the housing supply through the application of robotics. BotBuilt was accepted to YCombinator and has delivered robot-built framing for multiple homes.

During his Ph.D. Barrett published the following papers: *Stable Simulation of Underactuated Compliant Hands [54], Learning Symbolic Representations for Planning with Parameterized Skills [3], Bounded-Error LQR-Trees [2], and IKFlow: Generating Diverse Inverse Kinematics Solutions [4].*