Learning Model Preconditions for Planning with Multiple Models

Alex LaGrassa
Robotics Institute
Carnegie Mellon University
alagrass@andrew.cmu.edu

Oliver Kroemer
Robotics Institute
Carnegie Mellon University
okroemer@andrew.cmu.edu

Abstract: Different models can provide differing levels of fidelity when a robot is planning. Analytical models are often fast to evaluate but only work in limited ranges of conditions. Meanwhile, physics simulators are effective at modeling complex interactions between objects but are typically more computationally expensive. Learning when to switch between the various models can greatly improve the speed of planning and task success reliability. In this work, we learn model deviation estimators (MDEs) to predict the error between real-world states and the states outputted by transition models. MDEs can be used to define a model precondition that describes which transitions are accurately modeled. We then propose a planner that uses the learned model preconditions to switch between various models in order to use models in conditions where they are accurate, prioritizing faster models when possible. We evaluate our method on two real-world tasks: placing a rod into a box and placing a rod into a closed drawer.

Keywords: planning, manipulation

1 Introduction

Predictive models that are helpful for intelligent behaviors can take a variety of different forms, e.g. analytical models [1, 2], physics simulations [3, 4], and learned models [5, 6, 7], and the best choice of model is context-dependent. For example, a high fidelity simulator can model complex interactions between a large number of objects when dumping a pile of non-convex objects on a table [8]. However, for a simple tabletop pick and place task, a simulator model is often unnecessary when a simple kinematic model is available.

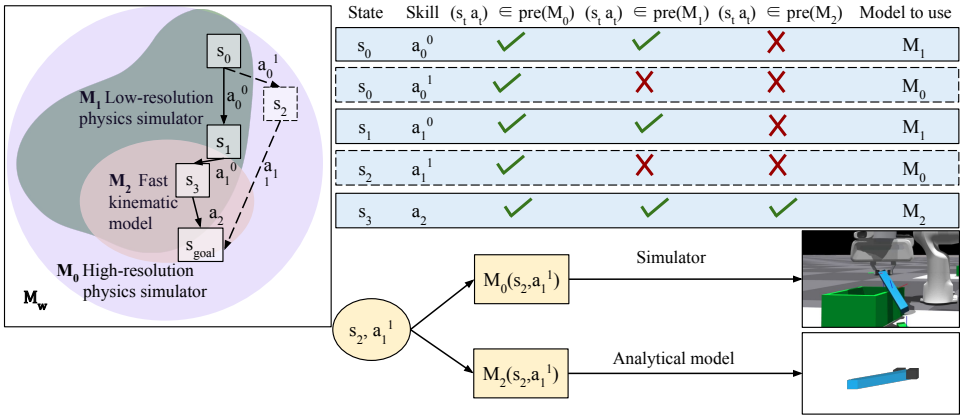


Figure 1: Each colored region on the left represents a model precondition, which we learn in this work for analytical and simulator models for manipulation skills. In this example, a kinematic model is faster than a low-fidelity simulator, and a low-fidelity simulator is faster than a high-fidelity simulator. On the right we show for each step which state-action pair (s_t, a_t^i) was in the precondition for each model, and the best model to use to evaluate that edge. The states and actions on the path chosen for the plan have a bold border. States not expanded because of high model cost have a dashed border. Multiple actions at a state s_t are denoted by superscripts: For example, two different actions from s_0 would be $[a_0^0, a_0^1]$.

5th Conference on Robot Learning (CoRL 2021), London, UK.

A planner may need to make many predictions to find a plan, so it is important to only use computationally expensive models when they are most needed. We define *model preconditions*, shown abstractly in Figure 1, that demarcate regions of states and actions where the model accurately represents real-world dynamics. We use those model preconditions during planning to leverage the complementary advantages and disadvantages of different models.

To capture the model preconditions, the robot learns to predict the total state deviation as a continuous value for each model. These deviation predictions are used to inform the planner to avoid transitions that are inaccurately modeled. The amount of acceptable model error is task-dependent, but by predicting state deviation as a continuous value rather than a binary one, the thresholds can be set based on the task accuracy requirements.

This work's contributions include a method for defining model preconditions using a learned scalar model deviation term and then using those preconditions for planning with multiple models to minimize model error at execution time. We first show how to fit Model Deviation Estimators (MDEs) from planning-relevant data to predict the error between a transition model and real-world dynamics. We then use MDEs to define the model preconditions for multiple skills for specific tasks. We evaluate our method for learning and planning with MDEs on two real-world tasks: placing a rod that can rotate in-hand into a box and placing a rod into a closed drawer. The robot has access to incomplete analytical models and a simulator model for each action class. Like in many real-world planning applications, none of the models match the real-world dynamics for all possible interactions. We compare two planning approaches with MDEs against other model selection baselines.

2 Related work

We review related work relevant to our two main contributions: predicting model error from data and planning using multiple imperfect models by reasoning about planning speed and reliability tradeoffs.

Improving planning reliability with past experience: Learning model preconditions for planning draws from the high-level idea of identifying similarities to states encountered in previous plans to apply behaviors that previously led to successful plans, which can significantly improve planning speed [9, 10]. Plan reliability can also be improved by learning skill preconditions, which describe the states from which skills achieve desired effects with high probability [11, 12, 13]. Instead of using skill preconditions, we use transition model preconditions.

Avoiding regions with model error: For predicting where the model is accurate using data, Chou et al. [14] and Knuth et al. [15] use hyperspheres to represent proximity to training data for a learned model as a proxy for the model precondition. CMAX and CMAX++ also use hyperspheres, but use points where the action effects differed significantly from those predicted by the internal model [16, 17]. However, proximity to training data and predicted collisions do not always capture model accuracy when executed on a real robot. Our work proposes a method to fit an estimator from plan execution data to predict whether model error will occur. Other works bias search away from states where the simple planning model is known to be inaccurate, but also do not use other models to compensate for transitions the simple model cannot model accurately, while our planner uses a slower model where appropriate [15, 14, 16, 18]. This distinction of not using alternative models when model error is predicted is especially significant when no one model can be used to compute a plan accurately, such as when each model only reasons about a subset of interactions necessary to achieve a task.

Predicting model error: The works most similar to ours use visual similarity to model the uncertainty of their simple model in Power and Berenson [19] and learning a classifier in the work of Mitrano et al. [20] and McConachie et al. [18]. The primary difference is that they do not use their model error estimates as criteria for switching models, whereas we use the planning data to learn model error estimates to decide between multiple models. McConachie and Berenson [21] does use multiple models, but uses task progress rather than error as selection criteria and does not do multi-step planning. Additionally, these works assume access to a simulator accurate enough to compute the ground-truth labels for accurate model predictions. We relax that assumption by treating the simulator as an additional model, which can also exhibit model error. Our labels then come from the real world. The methods above are also focused on a rope manipulation domain, where errors can usually be corrected with a recovery policy, as shown in [20]. Many actions in the

manipulation domain in which we evaluate our model error prediction lead to unrecoverable states, such as an object lying in an un-graspable pose in a constrained space.

Planning with multiple models: The work of Saleem and Likhachev [22] uses a rule where a simulator is only used when collision is detected using an internal model to speed up planning. However, there are many situations where there are transitions in which objects are in contact but an analytical model can still be used. Our planning algorithm is based on Multi-Graph Multi-Heuristic A* [23], which presents an algorithm that selects between models which operate on different parts of the state. The models we consider in this work operate on the same objects in the state, which makes choosing a model not straightforward. Furthermore, for the models we use, the best model to use depends on both the state and action rather than just state.

3 Problem Formulation

The overall problem is to first learn the MDEs to predict the total model deviation given a state, action, and transition model, then use MDEs for planning using multiple models, prioritizing planning using the faster models where possible while still minimizing cost.

The state space is denoted by $\mathcal{S}$ and is assumed to be fully observable at the end of an action. The action space $\mathcal{A}$ is a set of closed-loop skills $a \in \mathcal{A}$, parameterized by θ which are available to the robot. A user-defined skill-specific parameter generation function generates potentially useful parameters given a state. Skills follow the options formulation [24]. Skills are executed using a low-level controller until a termination condition is reached. The actions are intended to be run for a significant period of time such that feedback control at a lower level of abstraction can correct for low-level errors, leaving the MDEs to capture model error that is harder to correct. Skills have an associated precondition set, $\mathrm{pre}(a) \subseteq \mathcal{S}$ describing the set of states where the skill can be executed. The skill preconditions narrow the space where the skill has defined behavior, but do not comprehensively describe the set of states where the dynamics models are accurate. The skill precondition simply relates to the skill's applicability and are separate from MDEs and models of the skills' effects, which we describe next.

The robot is given an ordered list of models $[M_0, M_1, \ldots, M_K]$, used to compute the forward dynamics: $\hat{s}' \leftarrow M_i(s, a)$. The models are ordered by increasing computation speed, where M_i is slower to evaluate than M_j if $i < j$. For example, M_0 may be a high-fidelity simulator that is slow but very accurate, M_1 a simulator with a coarse timestep discretization, M_2 an analytical model requiring more computation, and M_3 a simple linear model. It is preferable to use the model with the highest i, corresponding to the fastest model, that is accurate for that (s, a). To measure model accuracy, the robot is given a distance function between states $d(s_i, s_j)$, such as Euclidean distance. The MDE predicts $\hat{d}$ from $\phi(s)$ and a, where $\phi(s)$ is a function that extracts relevant features such as object poses or distances between objects.

Lastly, the robot is given a task that includes a goal set, a cost function, and optionally additional task-specific action parameter generators. The state satisfies the goal if it is in the goal set $\mathcal{S}^g$. The cost function describes the cost of high-level state transitions and is denoted by $c(s, a, s')$.

To train the MDEs, the robot needs to first collect an offline dataset $\mathcal{D}$ of real-world transitions (s, a, s'). We assume the robot knows the initial state distribution but not all states visited during planning. Initial states are sampled from the initial state distribution, but the rest of the states are determined from planning. Then the robot trains the MDE $\hat{d}(s, a)$ which estimates $d(s', \hat{s}')$ using $\mathcal{D}$. The eventual goal is for the robot to search for a high-level plan of actions $[a_0, a_1, \ldots, a_{T-1}]$ to create a sequence of states $[s_0, s_1, \ldots, s_T]$ such that the final state is a goal state: $s_T \in \mathcal{S}^g$.

4 Approach

First we explain how to define and learn model preconditions using MDEs. We then show how MDEs can be used in planning to use multiple models while prioritizing states that can be reliably evaluated using simpler models.

4.1 Learning Model Preconditions Using Model Deviation Estimators

If a transition is in a model's precondition, then that model can be used to accurately model that transition. A model precondition is defined using MDEs as:

$$\mathrm{pre}(M_i) = \{(s, a) \,|\, \hat{d}(M_i(s, a), s') < d_{\max}\} \tag{1}$$

Training data for MDEs is collected by computing and executing plans using all models. If a path to the goal is found, it is executed. Each transition is saved as (s, a, s') tuples for every transition observed in the path. The list of transitions is expensive to collect as it requires real-world data, but several datasets for MDEs can be derived from it.

To train the MDE for a particular model M_i, $\hat{s}'$ is computed using M_i for each (s, a) in $\mathcal{D}$. The labels are $d(s', \hat{s}')$. Training data for transitions is used across all models even though that transition was computed using just one model during planning.

The inputs are state features, $\phi(s)$ concatenated with θ. $\phi(s)$ can be the identity function, but local features such as the distances between objects can enable easier generalization.

We train an MDE for every combination of skill and model, though MDEs can be shared if skill parameters have similar meaning. The reason why MDEs are regression models instead of classification models is to enable sharing of MDEs across tasks with different accuracy requirements. To model the MDEs we use a 3-layer MLP with 32 hidden units in each layer.

Next we describe the loss function for MDEs. An *underestimated* predicted deviation is worse for plan reliability than an *overestimated* predicted deviation. Overestimates cause the planner to be overly conservative, but an underestimate can cause plans to be executed unreliably on the real robot. As a result, we propose a loss function that penalizes underestimates more than overestimates: $L_g(d, \hat{d}) = c_1 \max(0, d - \hat{d})^2 + c_2 \max(0, \hat{d} - d)^2$. To get the desired behavior, we set $c_1 > c_2$.

4.2 Planning With Multiple Models Using MDEs

Model preconditions can be used in the same manner as skill preconditions by using the model preconditions as a constraint during planning, such as by using Equation 1 as an inequality constraint during optimization, where the M used at each t are additional variables. Model preconditions as predicate constraints can be applied to STRIPS-style planners using actions with a particular M to compute skill effects by including $(s, a) \in \text{pre}(M)$ to the existing action preconditions as an additional predicate.

Although MDEs can be used to select between models in a more general setting, we propose a planner that can more effectively use MDEs by using a user-defined model priority weighting to prioritize expanding subgraphs that are less expensive to evaluate. Our planner prioritizes simpler models by using a weighting factor inspired by the one used in Multi-Heuristic A*(MHA*) [25]. Because our planner uses successors from different models, we also build on Multi-Heuristic Multi-Representation A* [23].

Our proposed planning algorithm uses up to K models for each skill. Models can be shared across skills. The planner expands K implicit bidirectional graphs, which maintain their own open queues for nodes to expand and closed sets for nodes that have already been expanded, but all graphs share successors. All open queues are initialized with the start state and all closed sets are initialized to $\emptyset$.

An anchor search expands its graph by selecting between all models, and the $K - 1$ additional graphs expand using the $K - 1$ faster models. Priorities between models are adjustable using a list of model preference weights, $w = [w_0, w_1, \ldots, w_{K-1}]$ decreasing in value representing how much to penalize slower models. We now describe how the separate queues and graphs are used to switch between models.

Similar to standard Weighted A* (WA*), each planning node has a cost g, heuristic value h, and priority $f = g + \epsilon h$. The function MINKEY() returns the node in a queue with the lowest f value. To handle partial expansions, a node can optionally have a set of un-evaluated actions associated with it, $A_{inc}(s)$. The anchor search uses open queue OPEN_0 and closed set CLOSED_0. It uses a *full expansion*, which computes the successor with the fastest model that satisfies the precondition for a particular state and action. If no model satisfies the precondition, there is no successor for that transition. Nodes that are fully expanded in the anchor search are added to all closed sets. Each additional search maintains its own open queue OPEN_i and closed set CLOSED_i. It computes successors using a *partial expansion* only using M_i as long as $(s, a) \in \text{pre}(M_i)$, saving other actions for expansion by other graphs.

Graph and node selection: We expand from queue i using a rule inspired from MHA*:

$$i = \underset{i<K}{\arg\min}\ w_i \, \text{OPEN}_i.\text{MINKEY}() \tag{2}$$

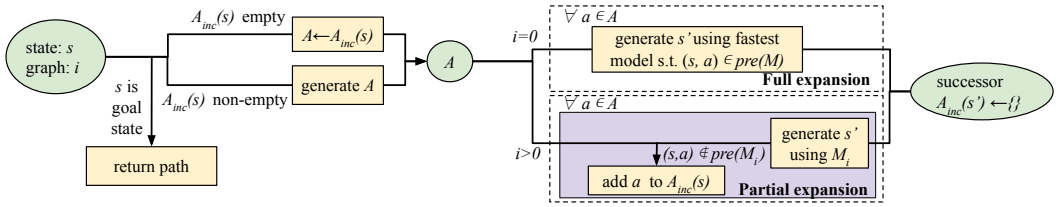


Figure 2: Graphical representation of the two types of expansions using model preconditions. Green ovals are data and yellow rectangles represent operations on data. The left side shows how actions are selected for both expansion types. The right side shows full expansion (top) and partial expansion (bottom).

If $i = 0$, then a node is expanded in the anchor search using the full expansion. If $i > 0$ then the node is expanded using a partial expansion for graph i. The node to be expanded is OPEN$_i$.MINKEY(), as with standard WA* search.

We begin by describing steps used for both types of expansions, and then describe what is different. All expansions begin by first checking the goal condition. As shown in Figure 2, the search is terminated when a goal state is expanded. Otherwise, expansion begins by selecting the set of actions to evaluate, A. A is A_{inc} if A_{inc} is non-empty. If not, a parameter generation function generates a candidate list of possible parameterized actions. The graph to expand is chosen using the rule in Equation 2. For each successor s', if the path found to s' is lower cost than any current path found to s' in any graph, the path is updated using s'. Then s' is added to to all OPEN$_i$ if it is not in CLOSED$_i$.

Full expansion: Full expansion is used as a fallback when the other searches with faster models run out of nodes or only contain highly suboptimal nodes. As shown in the top right box in Figure 2, full expansion evaluates all successors using $M_i(s, a)$ for the highest value of i that satisfies $(s, a) \in \text{pre}(M_i)$. Then, s is added to all closed sets.

Partial expansion: The intuition behind using partial expansion is to evaluate nodes using faster models sooner while delaying slower evaluations. For all $(s, a) \in \text{pre}(M_i)$, the successors s' are computed using M_i, and they are added to all open queues. For all states and actions not in the precondition, the successors are not evaluated. Instead, those actions are added to $A_{inc}(s)$ for later evaluation in other graphs, then the node is added to CLOSED$_i$. This step helps bias the search to nodes that are cheap to evaluate, while still allowing the planner to reason about when to use more expensive models for completeness or optimality.

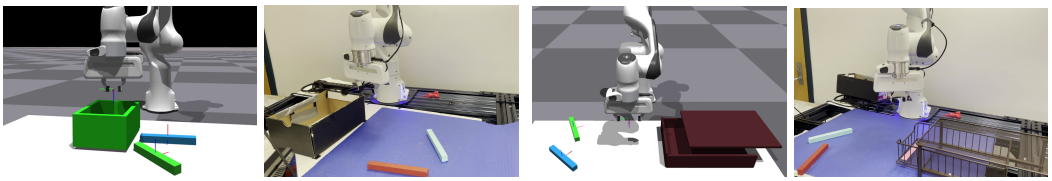
5 Experiments

First, we describe the metrics we use to evaluate our method, then describe the experimental setup we use for two real-world task domains shown in Figure 3: placing one of two steel rods in a box (`RodInBox`), and in a drawer (`RodInDrawer`). Then, we show the accuracy of MDEs for each skill. Finally, we evaluate two planners that use MDEs on two real-world tasks against baselines: one using a single queue with model preconditions as constraints and another using our multiple-queue planner described in Section 4.2.

We compare timing metrics, demonstrate the model selection for each skill in the `RodInDrawer` task, and discuss how the model selection distribution in each algorithm impacts plan time and reliability. For plan performance, we test: *1)* the time to compute a plan (seconds), *2)* the model evaluation rate (model evaluations per second) to measure the planner's ability to explore efficiently, *3)* the success rate in computing a plan, and *4)* the success rate achieving the goal in the real-world tasks if a plan was found (to measure reliability).

Tasks: For both `RodInBox` and `RodInDrawer` tasks, a Franka Emika Panda manipulates one of two steel rods into a desired container. The rod to place in the container is chosen randomly. Due to its weight, the rod can drop or rotate in the gripper (Figure 6). Both rods are placed in arbitrary reachable poses not occluded by other objects on the table.

The first task is `RodInBox`. A state is in $\mathcal{S}^g$ if the target rod is in the green box shown in Figure 3. The purpose of this task is to evaluate what happens when our method is applied on a task where high precision is not necessary for task completion, but the effects of some actions are easier to model than others. The second task we evaluate on is `RodInDrawer`, shown in Figure 3 where the



(a) `RodInBox` (Sim) (b) `RodInBox` (Real) (c) `RodInDrawer` (Sim) (d) `RodInDrawer` (Real)

Figure 3: Robot setup for test tasks in simulation and on the real robot.

goal is to place a rod in the partially closed drawer. Goal states are those where the rod is within the bounds of the drawer. Heuristics for both tasks are in Appendix 1.

Actions are movements in Cartesian end-effector space, implemented in the real world using the library from Zhang et al. [26] and with end-effector attractors in simulation. In simulation, the world is fully observable. In the real world, the rod poses are estimated using an the overhead camera. When a grasped object is occluded by the robot, an in-hand camera is used to confirm that the object is in the gripper, and the previously recorded pose is used. This perception is sufficient for our use case because the skills with the most variability do not end with an object in the gripper.

Skills: The skills the robot uses are *Pick*, *LiftAndDrop*, and *OpenDrawer*. Because the effect models used are high-level and the end-effector moves horizontally at a height above all obstacles, collisions are only checked at the beginning and end of each motion. The state includes the pose of all objects in the scene, including the robot end-effector pose. *Pick* is parameterized with the goal pose, which is sampled around the target rod. *LiftAndDrop* brings the end-effector and rod to a target location, then opens the grippers. The precondition of *LiftAndDrop* is that an object is between the grippers and that the goal pose will not cause collisions. *OpenDrawer* moves the gripper to a fixed location relative to the front of the drawer, then opens the drawer moving to a final pose. Detailed descriptions of all the skills, including hyperparameters, are in Appendix 7.1.

Models: The robot has access to a simulator and two analytical models. One analytical model, *Analytical (Pick & Place)*, only computes the transition model for pick and place actions, assuming a rod close to the gripper is rigidly attached. When the rod is placed, it falls to a height determined by the object (if any) directly below it. Another model, *Analytical (Drawer)*, only reasons about a simplified articulation mechanism of the drawer, assuming the drawer edge is rigidly attached to the gripper if both surfaces are close, but does not model interactions between the rods and drawer. Details about each analytical model are in Appendix 7.2.2

The simulator we use for both tasks is IsaacGym [3]. For `RodInBox`, we use a setup with a box shown in Figure 3(a). The simulation parameters used for both experiments are shown in Appendix 7.2.1. For `RodInDrawer`, the drawer is modeled as a single drawer chest using cuboids, as shown in Figure 3(c). To evaluate the result of a skill from a state s, the simulator sets the state of the world to s, executes skill a and returns the resulting state as $\hat{s}'$

5.1 MDE Accuracy

In this section, we evaluate MDE accuracy on each skill used, which correlates with the variance of d. As shown in Table 1, deviation prediction error is highest for *LiftAndDrop* because there is a wider range of possible outcomes. The predicted deviation error for the simulator model is low but not zero because predicting the precise deviation requires an accurate real-world dynamics model. For example, a poorly-grasped rod may drop in simulation earlier than it would on the real-robot system by dragging on the ground differently. Since *Analytical (Drawer)* does not model movement of the rods, the exact deviation depends on the state and parameters, but the error is always predicted to be high for *LiftAndDrop*, which is sufficient information to guide planning. The deviation for *Pick* using *Analytical (Drawer)* is low because modeling *Pick* does not require any physical interactions between robots and rods, and the robot is not in contact with the drawer. Simulator deviation prediction error for *Pick* is high be-

Skill	Model	MAE	d
Pick	*Analytical (Pick & Place)*	0.2	0.4 (0.4)
Pick	*Analytical (Drawer)*	0.2	0.4 (0.4)
Pick	*Simulator*	0.7	4.7 (9.5)
LiftAndDrop	*Analytical (Pick & Place)*	1.9	8.3 (7.41)
LiftAndDrop	*Analytical (Drawer)*	7.6	11.6 (8.0)
LiftAndDrop	*Simulator*	2.5	11.0 (10.7)
OpenDrawer	*Analytical (Pick & Place)*	0.4	16.0 (0.4)
OpenDrawer	*Analytical (Drawer)*	0.6	2.1 (0.3)
OpenDrawer	*Simulator*	1.2	7.0 (0.5)

Table 1: Mean absolute error (MAE) and d (mean and standard deviation) in centimeters for each model/skill combination.

Method	Plan time	Model eval. per second
Ours - multiple queues	0.40	25.2
Ours - one queue	8.98	1.13
Random	40.00	0.25
Analytical (Pick & Place) only	0.19	113.58
Simulator only	31.23	0.5

(a) `RodInBox`

Method	Plan time	Model eval. per second
Ours - multiple queues	3.69	19.0
Ours - one queue	5.27	6.0
Random	44.36	0.6
Simulator only	103.34	0.5

(b) `RodInDrawer`

Table 2: Planning times for methods that can find plans. We show the average time to compute a plan (not including timeouts).

cause when a drawer is in the environment, the joints sometimes interact with the drawer, causing interactions that do not occur in the real world.

5.2 Planning Speed

For planning, we evaluate the performance from ten different initial start states per task. `RodInBox` uses *Pick* and *LiftAndDrop*, sampling five parameter vectors per skill. `RodInDrawer` uses *Pick*, *LiftAndDrop*, and *OpenDrawer*, with three parameters generated per skill. $\epsilon = 5$ for `RodInBox` and $\epsilon = 10$ for `RodInDrawer`. The model preference weights are [10,1] corresponding to [*Simulator*, *Analytical (Pick & Place)*] for `RodInDrawer` and [10,1.1,1] corresponding to [*Simulator*, *Analytical (Drawer)* , *Analytical (Pick & Place)*] for `RodInBox`. The cost function is given by the total end effector distance covered in the trajectory. For timing experiments shown in Table 2a, we allow the planner to run for at most 300 seconds (5 minutes).

Baselines: We compare planning using MDEs to planning with each individual model, and randomly selecting which model to use when computing each successor. Additionally, we perform an ablation test by removing the additional queues and only performing full expansions using the anchor search, though still evaluating the fastest model that satisfies the preconditions.

For `RodInBox`, (Table 2a) the planning time and model evaluation rate are fastest using only the analytical model. Random model selection and simulator-only planning are the slowest. Because our method uses the fastest reliable model wherever possible, and for this task a short plan to the goal can be computed using only the fastest model, the model evaluation rate for our method is still high. The one-queue planner that uses MDEs is slower because the model precondition for the simulator model for *LiftAndDrop* is usually satisfied, so some actions that are unnecessary to find a plan are simulated.

The planning speeds for both methods using MDEs in `RodInDrawer` (Table 2b) are faster than the simulator-only baseline because it can always use an analytical model for *Pick* and *OpenDrawer*, and sometimes use the analytical model for *LiftAndDrop*. The improvement of using multiple queues in this domain is smaller than for `RodInBox`, which we discuss further in the next paragraph. The random baseline is the slowest because it uses the largest number of simulator calls.

Both MDEs and the planning method can affect the distribution of models chosen. For example, Figure 4 (left) shows planning with one queue causes the planner to compute successors using transitions in the model precondition of more expensive models at the same priority of sets in the faster models, which causes some actions to be simulated that would not be if they were using our multiple-queue planner, causing the longer planning times for `RodInBox`. For `RodInDrawer`, the one-queue ablation test selects very similar models to the multiple-queue planner, so using a single queue does not significantly affect planning speed. The reason is that the simulator model is more inaccurate in `RodInDrawer` than in `RodInBox`. The MDE can detect the task indirectly since the parameter distributions for the two tasks are different. As a result, the one-queue planner rarely uses the simulator, which can also be seen in Figure 4 (right).

5.3 Reliability of Computed Plans

The experiments shown in Figure 5, evaluate the ability of our method to improve plan reliability without significantly increasing planning time by balancing model accuracy and computational cost.

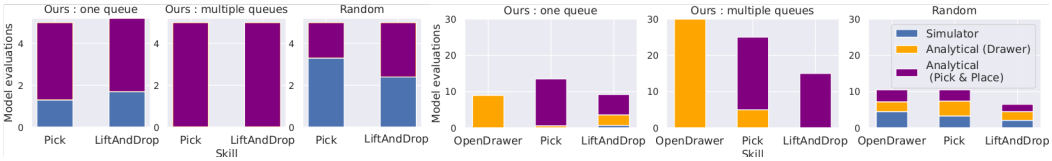
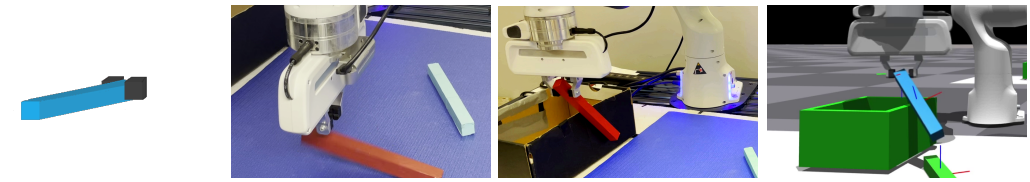


Figure 4: Average number of models evaluation of each type for all three skills for `RodInBox` (left) and `RodInDrawer`(right) across methods that use multiple models

Figure 5: Success rate in finding plans (left) and if found, executing them until the goal (right)



(a) No pivot (Analytical) (b) Drop during transport (Real) (c) Hitting target container (Real) (d) Unrealistic physics (Sim)

Figure 6: Situations where planning models are inaccurate. (a) *Analytical (Pick & Place)* assumes rigid attachment, although this (b) causes the rod to either drop during transport or (c) rotate and hit the target container. (d) The simulator sometimes exhibits unrealistic physics such as the rod sliding over the box walls.

All methods that use a model that reasons about interactions between the rod and robot are able to compute a plan in the allotted time for `RodInBox`. In the more challenging `RodInDrawer` task, the simulator-only baseline and one-queue version of our method sometimes fail to find a plan within the time limit. Plan success is highest for both methods using model preconditions and the simulator-only baseline. MDEs cause the planner to only use transitions where the model is predicted to be accurate, such as using *LiftAndDrop* when the rod is grasped in the center. The simulator is more accurate than the analytical models so plans using it often succeed. In contrast, plans found using the analytical-model-only and random-model baselines often fail because the lowest cost paths often include grasping the end of a rod, which is predicted to be rigidly attached by the analytical model (Figure 6a). In the real world, grasping the end of the rod causes it to pivot, which usually causes the rod to drop during transport (Figure 6b) or after hitting the target container (Figure 6c). Although the simulator can model pivoting in the gripper, unrealistic dynamics sometimes occur (Figure 6d).

6 Conclusion

We present a method for defining model preconditions using MDEs that predict model deviation given a state and a parameterized skill, which can be used to inform tradeoffs between models when each has complementary advantages and disadvantages in evaluation time and accuracy. Although MDEs can be used as constraints, we show a planner that uses multiple queues corresponding to different models that prioritizes planning using the faster models. Experimental evaluations show a speed up in planning by choosing between multiple models while keeping high plan reliability during execution. In both evaluation tasks, our results show that a robot can learn and reason about which transitions need to use more expensive models, which transitions none of the models can evaluate accurately, and how to inform planning with that information. For placing a rod in a drawer, we show that the robot can combine two low-fidelity models intended for representing different interactions, while avoiding using an expensive simulator that models all interactions. For future work, we will modify our framework to share more data across skills by using a shared action representation, which will also allow changing a low-level model multiple times during a skill execution. Furthermore, we will evaluate our method on tasks where model selection is more difficult, such as liquid, paper, and other deformable object manipulation.

Acknowledgements

This work was supported by the Office of Naval Research Grant No. N00014-18-1-2775, Army Research Laboratory grant W911NF-18-2-0218 as part of the A2I2 program, and National Science Foundation Grant No. CMMI-1925130 and IIS-1956163. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors. The authors also thank Kevin Zhang for assistance with robot experiments, as well as Shivam Vats, Jacky Liang, Mohit Sharma, and Saumya Saxena for their support and discussions.

References

[1] S. Goyal, A. Ruina, and J. Papadopoulos. Planar sliding with dry friction part 1. limit surface and moment function. *Wear (Amsterdam, Netherlands)*, 143(2):307–330, 1991.

[2] Z. Pan and K. Hauser. Decision making in joint push-grasp action space for large-scale object sorting. *arXiv preprint arXiv:2010.10064*, 2020.

[3] J. Liang, V. Makoviychuk, A. Handa, N. Chentanez, M. Macklin, and D. Fox. GPU-accelerated robotic simulation for distributed reinforcement learning. In *Conference on Robot Learning*, pages 270–282. PMLR, 2018.

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

[5] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson. Learning latent dynamics for planning from pixels. In *International Conference on Machine Learning*, pages 2555–2565. PMLR, 2019.

[6] B. Sundaralingam and T. Hermans. In-hand object-dynamics inference using tactile fingertips. *IEEE Transactions on Robotics*, 2021.

[7] A. Nagabandi, K. Konolige, S. Levine, and V. Kumar. Deep dynamics models for learning dexterous manipulation. In *Conference on Robot Learning*, pages 1101–1112. PMLR, 2020.

[8] M. Macklin, M. Müller, N. Chentanez, and T.-Y. Kim. Unified particle physics for real-time applications. *ACM Transactions on Graphics (TOG)*, 33(4):1–12, 2014.

[9] B. Kim, Z. Wang, L. P. Kaelbling, and T. Lozano-Pérez. Learning to guide task and motion planning using score-space representation. *The International Journal of Robotics Research*, 38 (7):793–812, 2019.

[10] M. Phillips, B. J. Cohen, S. Chitta, and M. Likhachev. E-graphs: Bootstrapping planning with experience graphs. In *Robotics: Science and Systems*, volume 5, page 110, 2012.

[11] G. Konidaris, L. P. 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.

[12] M. Sharma and O. Kroemer. Relational learning for skill preconditions. In *Proceedings of (CoRL) Conference on Robot Learning*, November 2020.

[13] Z. Wang, C. R. Garrett, L. P. Kaelbling, and T. Lozano-Pérez. Learning compositional models of robot skills for task and motion planning. *The International Journal of Robotics Research*, 40(6-7):866–894, 2021.

[14] G. Chou, N. Ozay, and D. Berenson. Model error propagation via learned contraction metrics for safe feedback motion planning of unknown systems. *ArXiv*, abs/2104.08695, 2021.

[15] C. Knuth, G. Chou, N. Ozay, and D. Berenson. Planning with learned dynamics: Probabilistic guarantees on safety and reachability via Lipschitz constants. *IEEE Robotics and Automation Letters*, 6(3):5129–5136, 2021.

[16] A. Vemula, Y. Oza, J. A. Bagnell, and M. Likhachev. Planning and execution using inaccurate models with provable guarantees. In *Proceedings of Robotics: Science and Systems (RSS '20)*, July 2020.

[17] A. Vemula, J. A. Bagnell, and M. Likhachev. CMAX++: Leveraging experience for planning and execution using inaccurate models. *arXiv preprint arXiv:2009.09942*, 2020.

[18] D. McConachie, T. Power, P. Mitrano, and D. Berenson. Learning when to trust a dynamics model for planning in reduced state spaces. *IEEE Robotics and Automation Letters*, 5(2): 3540–3547, 2020.

[19] T. Power and D. Berenson. Keep it simple: Data-efficient learning for controlling complex systems with simple models. *IEEE Robotics and Automation Letters*, 6(2):1184–1191, 2021.

[20] P. Mitrano, D. McConachie, and D. Berenson. Learning where to trust unreliable models in an unstructured world for deformable object manipulation. *Science Robotics*, 6(54), 2021. doi:10.1126/scirobotics.abd8170. URL https://robotics.sciencemag.org/content/6/54/eabd8170.

[21] D. McConachie and D. Berenson. *Bandit-Based Model Selection for Deformable Object Manipulation*, pages 704–719. Springer International Publishing, Cham, 2020. ISBN 978-3-030-43089-4. doi:10.1007/978-3-030-43089-4_45. URL https://doi.org/10.1007/978-3-030-43089-4_45.

[22] M. S. Saleem and M. Likhachev. Planning with selective physics-based simulation for manipulation among movable objects. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6752–6758. IEEE, 2020.

[23] D. Youakim, A. Dornbush, M. Likhachev, and P. Ridao. Motion planning for an underwater mobile manipulator by exploiting loose coupling. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7164–7171. IEEE, 2018.

[24] R. S. Sutton, D. Precup, and S. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[25] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev. Multi-heuristic A*. *The International Journal of Robotics Research*, 35(1-3):224–243, 2016.

[26] K. Zhang, M. Sharma, J. Liang, and O. Kroemer. A modular robotic arm control stack for research: Franka-interface and frankapy. *arXiv preprint arXiv:2011.02398*, 2020.

[27] C. R. Garrett. Pybullet planning, 2020. URL https://pypi.org/project/pybullet-planning.

[28] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.