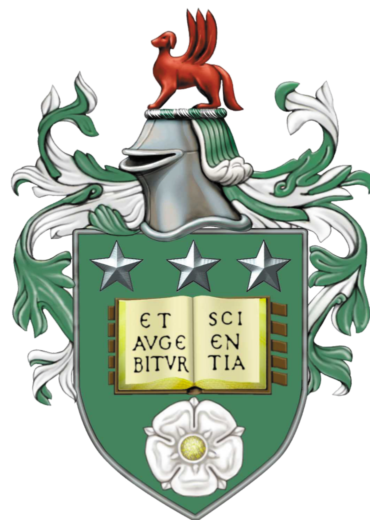
Robust Physics-Based Robotic Manipulation in Real-Time

Wisdom C. Agboh

Submitted in accordance with the requirements
for the degree of Doctor of Philosophy



The University of Leeds
School of Computing
March 2021

The candidate confirms that the work submitted is his own, except where work which has formed part of a jointly authored publication has been included. The contribution of the candidate and the other authors to this work has been explicitly indicated below. The candidate confirms that appropriate credit has been given within the thesis where reference has been made to the work of others.

Some parts of the work presented in this thesis have been published in the following articles.

Agboh W.C. and Dogar M.R. (2021) Robust Physics-Based Manipulation by Interleaving Open and Closed-loop Execution, arXiv (2021).

Agboh W.C., Grainger O., Ruprecht D., and Dogar M.R. (2020) Parareal with a Learned Coarse Model for Robotic Manipulation, Computing and Visualization in Science. 23, 8 (2020).

Agboh W.C. and Dogar M.R. (2020) Pushing Fast and Slow: Task-Adaptive Planning for Nonprehensile Manipulation Under Uncertainty. In: Morales M., Tapia L., Sánchez-Ante G., Hutchinson S. (eds) Algorithmic Foundations of Robotics XIII. WAFR 2018. Springer Proceedings in Advanced Robotics, vol 14. Springer, Cham.

Agboh W.C., Ruprecht D., and Dogar M.R. (2019) Combining Coarse and Fine Physics for Manipulation using Parallel-in-Time Integration, Proceedings of the International Symposium on Robotics Research (ISRR), 2019.

Agboh W.C. and Dogar M.R. (2018) Real-Time Online Re-Planning for Grasping Under Clutter and Uncertainty. IEEE-RAS 18th International Conference on Humanoid Robots.

The above publications are primarily the work of the candidate.

This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis maybe published without proper acknowledgement.

©2021 The University of Leeds and Wisdom C. Agboh

Acknowledgements

I would like to say a big thank you to my advisor Mehmet Dogar, for his friendship, guidance, and continuous support. My PhD journey was fun and I learnt a lot from you. I would like to thank my secondary advisor Tony Cohn for all the insightful discussions and guidance throughout my PhD.

To my examiners, Yanlong Huang and Subramanian Ramamoorthy, I want to say a big thank you. Your feedback and comments have helped me to significantly improve my thesis.

I would like to thank Gaynor Butterwick for promptly helping me out whenever I had official business.

I would like to thank Daniel Ruprecht, whom I collaborated the most with. Thanks for the fun and your support. I was fortunate to collaborate with many other great researchers, Matteo Leonetti, Hassan Mohammed, Wissam Bejjani, Oliver Grainger, and Leo Pauly. Thanks for the fun and hard work.

I enjoyed my time at the robotics lab. I would like to thank my fellow roboticists, Rafael Papallas, Wissam Bejjani, Leo Pauly, Alexia Toumpa, Francesco Foglino, Ricardo Luna, Luis Figueredo, Lipeng Chen, Hasan Mohammed, and Logan Dunbar, for their friendship, discussions, support, and the many coffee breaks that got me through my PhD.

Finally, I want to say a big thank you to my family, Helen, Matthew, Eliza, Obi, Emmanuel, Anwulie, Ugo, Favour, Ife, and Gift for always being there for me. This would not be possible without you.

Abstract

This thesis presents planners and controllers for robust physics-based manipulation in real-time. By physics-based manipulation, I refer to manipulation tasks where a physics model is required to predict the consequences of robot actions, for example, when a robot pushes obstacles aside in a fridge to retrieve an object behind them.

There are two major problems with physics-based planning using traditional techniques. First, uncertainty, both in physics predictions and in state estimation, can result in the failure of many physics-based plans when executed in the real-world. Second, the computational expense of making physics-based predictions can make planning slow and can be a major bottleneck for real-time control. I address both of these problems in this thesis.

To address uncertainty, first, I present an online re-planning algorithm based on trajectory optimization. It reacts, in real-time, to changes in physics predictions to successfully complete a manipulation task. Second, some open-loop physics-based plans succeed in the real-world under uncertainty. How can one generate such robust open-loop plans with guarantees? I provide conditions for robustness in the real-world based on contraction theory. I also present a robust planner and a controller. It autonomously switches between robust open-loop execution, and closed-loop control to complete a manipulation task. Third, a robot can be optimistic in the face of uncertainty. It can adapt its actions to the accuracy requirements of a task. I present such a task-adaptive planner that embraces uncertainty, pushing fast for easy tasks, and slow for more difficult tasks.

To address the problem of computationally expensive physics-based predictions, I present learned and analytical coarse physics models for single and multi-object manipulation. They are cheap to compute but can be inaccurate. On the other hand, fine physics models provide the best predictions but are computationally expensive. I present algorithms that combine coarse and fine physics models through parallel-in-time integration. The result is orders of magnitude reduction in physics-based planning and control time.

Contents

1 Introduction **1**
1.1 Main themes . . . 2
1.2 Contributions . . . 5
1.3 Roadmap . . . 6

2 Related Work **7**
2.1 Manipulation in Clutter . . . 8
2.2 Open-Loop Execution . . . 9
2.3 Robust Open-Loop Execution . . . 10
2.4 Closed-loop Control Policies . . . 11
2.5 Uncertainty-Aware Planning and Control . . . 13
2.6 Speeding-up Physics Simulators . . . 14
2.7 Combining Different Physics Models . . . 14
2.8 Manipulation with Visual Dynamics . . . 15
2.9 Summary . . . 16

3 Real-Time Online Re-Planning for Grasping Under Clutter and Uncertainty **19**
3.1 Physics-based Grasping in Clutter . . . 22
3.2 Physics-based Grasping through Online Re-planning . . . 23
3.3 Experiments and Results . . . 27
3.4 Discussion . . . 35

4 Robust Physics-Based Manipulation by Interleaving Open and Closed-Loop Execution **37**
4.1 The Robust Manipulation Problem . . . 39
4.2 Interleaving Open and Closed-Loop Execution . . . 40
4.3 Robustness to Uncertainty . . . 42
4.4 Robust Planning and Control . . . 48
4.5 Robot Experiments and Results . . . 53
4.6 Discussion . . . 60

5 Task-Adaptive Planning for Non-prehensile Manipulation Under Uncertainty **61**
5.1 Task-Adaptive Planning as an MDP 63
5.2 Approximate Online MDP Solution 65
5.3 Generating a Variety of Actions 68
5.4 Trajectory Optimization 69
5.5 Baseline Approach 71
5.6 Experiments 71
5.7 Discussion 76

6 Combining Coarse and Fine Physics for Manipulation using Parallel-in-Time Integration **77**
6.1 Combining Physics Models for Planning 79
6.2 Push Planning and Control 84
6.3 Experiments and Results 86
6.4 Discussion 91

7 A Learned Coarse Model for Robotic Manipulation using Parareal **93**
7.1 Robotic Manipulation with Parareal 93
7.2 Coarse models 95
7.3 Planning and control with hybrid models 97
7.4 Experiments and Results 98
7.5 Summary 105

8 Conclusion **107**
8.1 Limitations and Future Work 108
8.2 Summary 110

Bibliography **111**

List of Figures

3.1 Snapshots from execution with online re-planning. 20
3.2 Goal cost terms 23
3.3 Edge cost terms 23
3.4 Top row: The planned control sequence and state evolution. Middle row: Open-loop execution of the planned control sequence fails under medium uncertainty. Bottom row: Online re-planning, OR succeeds under medium uncertainty. 29
3.5 Simulation results for 100 random scenes. In b-d, I plot the average with 95% confidence interval of the mean. 31
3.6 Real robot results for 5 random scenes. In (b)-(d), I plot the average with 95% confidence interval of the mean 32
3.7 Top row: Naive re-planning (no added uncertainty) fails to grasp the target (in green). Bottom row: Online re-planning succeeds. 33

4.1 A combination of robust open-loop execution and closed-loop control to reach for the green target object. Top row: Planned trajectories and execution strategy. Bottom row: Real-robot open and closed-loop execution to reach for the green target object. The robot starts with robust open-loop loop execution due to motion in free-space. It falls back to closed-loop control during multi-contact interactions. Finally, it uses a robust funnelling motion to get the green target object in the gripper. 38
4.2 Search in a robustness graph to find a combination of robust and non-robust segments that maximize the number of time-points that are robust. In the robustness graph, the robust edges are shown in green, the non-robust edges are shown in blue. 42

4.3 Computing the real-world expected divergence metric $\hat{E}_e^r$. First, I compute the divergence metric using the nominal model f ($\hat{E}_e^f$). I draw N_c sample initial states from the observed initial state $\boldsymbol{x}_0$ and roll-out a trajectory from each state, to obtain final state samples $\boldsymbol{x}_N^i$. Since the size of the final state distribution is less than that of the initial state distribution, $\hat{E}_e^f < 1$, and the trajectory is convergent in f. Thereafter, we randomly create N_w real-world realizations from f and compute the divergence metric in each of these N_w worlds. These metrics can be very different from each other across the real-world realizations. We pick the metric with the maximum value as a worst-case approximation of the real-world divergence metric. 46
4.4 Experimental results comparing the interleaved open and closed-loop execution method (OCL) with other baselines. I randomly created 20 real-world scenes and ran all five methods on each scene. I recorded success, planning time, execution time, divergence metrics, and finally what percentage of trajectory segments are executed open-loop vs. closed-loop in a given scene. All error bars indicate a 95% confidence interval of the mean. 56
4.5 A sample robust segment from the interleaved open and closed-loop approach. The left image shows a part of the robustness metric computation. I see that the initial state uncertainty is reduced through a funnelling action that pushes the target object towards the shelf, and grasps it. On the right image I see the robust segment executed open-loop in the real world. 58
4.6 A sample robust segment from the interleaved open and closed-loop approach. The left image shows a part of the robustness metric computation. I see that the initial state uncertainty is reduced through a stable side push. On the right image I see the robust segment executed open-loop in the real world. 58
4.7 Examples of successful and failed manipulation plans from different planning and execution methods. 59

5.1 Task-adaptive pushing with 21 slow actions for a high accuracy task (top) and a single fast action for a low accuracy task (bottom). 62

5.2 First column: Initialization of the task-adaptive approach with control sequences including fast (top) and slow (bottom) actions. Second column: Stochastic trajectory optimization of the initial candidate control sequences. Last column: Action evaluation under uncertainty through sampling. 66
5.3 Success rate and total elapsed time versus uncertainty level for low and high accuracy tasks. 73
5.4 Push planning in a changing environment (top) using a single fast push initially and then slow pushes later on due to the narrow strip. For the L-shaped environment (bottom), the robot executes many actions to successfully navigate the edge. 74
5.5 Grasping in clutter: The robot uses fast actions initially but chooses slower actions as it gets closer to the goal object near the edge of the table. 75
5.6 MPC using a large number of actions to complete a low accuracy level task (top), and causing the pushed object to fall off for a high accuracy level task (bottom). 76

6.1 A spectrum of physics predictions from cheapest and least accurate (a) to expensive and most accurate (d). 78
6.2 Combining coarse and fine physics with the Parareal algorithm (a) Initial coarse physics predictions across time with a cheap model (C), (b) Fine physics predictions in *parallel* starting from coarse initial guesses with F. (c) A Parareal update at time $n = 1$ as a linear combination of coarse and fine approximations of the state (d) A Parareal update at time $n = 2$ using the updated state at time $n = 1$, (e) Final trajectory updates after $k = 1$ Parareal iteration, (f) Next Parareal iteration begins with fine physics predictions in *parallel*. 80
6.3 Coarse physics model 82
6.4 Push planning with a hybrid physics model to avoid an obstacle (in black) while pushing the cylindrical slider to a goal location (in red). 85
6.5 Root mean square error (in log scale) along the full trajectory for pushing a cylinder (a) and box (b) from the center and side respectively, for increasing Parareal iterations. The motions are illustrated lower-right in each plot. 88
6.6 Physics simulation time averaged over 100 runs for a box side push within 95 % confidence interval of the mean. 89

6.7 Total task completion time (within 95 percent confidence interval of the mean) for push planning with obstacle avoidance using different physics models for 100 randomly sampled initial states. 91
6.8 Pushing with a hybrid physics model. I complete the push planning task about **four times faster** than a physics engine. . . . 92

7.1 Example of a robotic manipulation planning and control task using physics predictions. The robot controls the motion of the green object solely through contact. The goal is to push the green object into the target region marked X. The robot must complete the task without pushing other objects off the table or into the goal region. 94
7.2 Root mean square error (in log scale) of Parareal along the full trajectory for single object pushing using both a learned and an analytical coarse model (top). These results are for a control sequence with 4 actions where the average object displacement is 0.043 ± 0.033 m. The error at iteration four is 0. The learned coarse model gives a better Parareal convergence rate. Sample motions for the learned coarse model (bottom, right) and the analytical coarse model (bottom, right). The learned coarse model's prediction is closer to the fine model prediction shown in green. 98
7.3 Root mean square error (in log scale) along the full trajectory per slider in a 4-slider pushing experiment (top) using *only* the learned model. Two sample motions are illustrated (bottom, left and right) for multi-object physics prediction. These results are for a control sequence with 4 actions where the average object displacement is 0.015 ± 0.029 m. The error at iteration four is 0 except for accumulation of round-off errors. I find that the learned model enables Parareal convergence for the multi-object case. 101
7.4 Root mean square error (in log scale) along the full trajectory per object for single object pushing (bottom) and multiple object pushing (top) using *only* the learned model. Here I consider a control sequence of 8 actions. The average object displacement for multi-object pushing is 0.034 ± 0.082 m and for single object pushing it is 0.046 ± 0.040 m. The error at iteration eight is 0. I find that the convergence of Parareal appears similar even with a longer control sequence. 103

7.5 Root mean square error along the full trajectory for all 4 sliders *measured with respect to the real-world pushing data*. The vertical bars indicate a 95% confidence interval of the mean. The learned coarse physics model at iteration 0 has the largest error and the fine model provides the best prediction w.r.t the real-world pushing physics. 104
7.6 The resulting sequence of states for applying a random control sequence starting from some random initial state in the real-world. Our goal is to assess the accuracy of the Parareal physics models with respect to real-world physics. I collect 50 such samples. These are some snapshots for 3 of such scenes - one per row with initial state on the left and final state on the right. 104
7.7 Robotic manipulation planning and control for 2 different scenes. The robot succeeds in all scenes using Parareal with a learned coarse model for physics predictions. The third planning and control scene is in Fig. 7.1. 106

8.1 An example of robust and non-robust object motions in a given trajectory segment. On the left image, each object takes several possible positions. This illustrates the initial state uncertainty. After a robot motion that pushes on both a cylinder to the left and a box to the right, the size of the state uncertainties change. It is smaller for the cylinder to the left, and larger for the box to the right. Hence the cylinder's motion is robust and the boxes' motion is not. 110

List of Tables

4.1 Experimental parameters 54

6.1 Open-loop pushing 90

Chapter 1

Introduction

Imagine a future society where robots instead of humans do most of the physical tasks; for example, house chores like cleaning or factory work like picking and packing objects into boxes. At the heart of this robotics revolution are robots that can reason about the result of their physical interactions with objects in the environment. These physical interactions include a wide variety of actions such as toppling, throwing, pushing, bending etc. Robots use these actions to manipulate objects, changing their states, to achieve a desired goal - like a cleared table-top. This is the physics-based robotic manipulation problem.

Today, I do not see these robots in our everyday lives due to two major challenges - uncertainty and multi-contact interactions.

Uncertainty: To manipulate an object, a robot needs to know where the object is - its state. However, this is uncertain, i.e. not exactly known. State uncertainty is one of the major reasons for failure in robotic manipulation. Another source of uncertainty is predicting states with physics models. Incomplete knowledge about object parameters such as friction, mass, and exact shape at high resolution contribute to this physics uncertainty. For example, the effects of a robot's push on an object can be uncertain. The object can rotate left or right and this can lead to failures if the correct prediction is not used, or if an uncertainty-aware strategy is not used.

Multi-contact interactions: Reaching into a shelf can require a robot to make contact with a single object and maybe push it aside. This pushed object can then go on to push and dislocate other objects in the shelf. Physics predictions for such a multi-contact interaction is computationally expensive. It leads to long manipulation planning times.

In this thesis, I develop planners, controllers, and associated algorithms to perform robotic manipulation under both sources of uncertainty - in state and physics model parameters. I also address long prediction times during multi-contact interactions. I propose planning and control algorithms where physics predictions play a central role, to achieve robust manipulation in real-time.

1.1 Main themes

My goal of robust physics-based robotic manipulation in real-time leads to several interesting opportunities and challenges. These are the main themes of this thesis:

1. *Closed-loop physics-based control significantly improves robotic manipulation success rates in the real-world.*

   Physics-based manipulation in clutter has been addressed with motion planners tailored for open-loop execution. However, such open-loop methods are likely to fail, since it is not possible to model the dynamics of the multi-body multi-contact physical system with enough accuracy, neither is it reasonable to expect robots to know the exact physical properties of objects, such as frictional, inertial, and geometrical.

   In this thesis, I propose closed-loop approaches to physics-based manipulation under uncertainty. I show significant improvements in real-world success rates in difficult cluttered environments, compared to traditional open-loop methods.

2. *Real-time re-planning cycles can be achieved through an appropriate underlying planning algorithm — one that accepts warm-starts.*

   The main challenge in closed-loop methods for physics-based manipulation is the long planning times. It makes fast re-planning and fluent execution difficult to realize.

   In order to address this, I propose an easily parallelizable stochastic trajectory optimization based algorithm that generates a sequence of optimal controls. During execution, warm-starting this optimizer with portions of a previous plan leads to convergence after a small number of iterations. This makes it possible to perform real time re-planning cycles and achieve reactive manipulation under clutter and uncertainty.

3. *Fully robust open-loop plans are desirable but may not exist for many physics-based manipulation problems. Interleaving robust open-loop execution and closed-loop control can improve success rates and lead to more fluent/real-time execution.*

   Fully robust trajectories are guaranteed to succeed when executed open-loop, one action after the other, without feedback from the environment.

They are desirable for a few reasons. First, physics-based re-planning time is still high for some problems, inducing noticeable delays during manipulation plan execution, while robust open-loop execution is fluent. More importantly, some dynamic tasks may require a much smaller re-planning time — otherwise a pushed object can quickly roll and fall-off the table before the robot takes its next action.

Fully robust trajectories can involve a robot's motion in free space or funnelling actions, for example, where an object is tightly caged between a robot's gripper and a shelf. These fully robust trajectories are extremely difficult to find or may not exist for many multi-contact manipulation problems. For example, a multi-object robot push that reduces uncertainty in all the objects in contact may not exist, perhaps because of the specific configuration of objects.

In this thesis, I separate a trajectory into robust and non-robust segments through a minimum cost path search on a robustness graph. Robust segments are executed open-loop and non-robust segments are executed with model-predictive control. This results in improved success rates while simultaneously achieving a more fluent/real-time execution.

4. *Robots can adapt their actions to the accuracy requirements of a task — pushing fast for low accuracy tasks and slow for high accuracy tasks.*

Humans use a wide variety of actions to complete everyday manipulation tasks. For example, reaching quickly into an almost empty shelf to retrieve an object versus carefully searching a shelf filled with glassware. An important question is how can robots embrace uncertainty and exhibit such task-adaptive behaviour.

In this thesis, I propose an algorithm for task-adaptive physics-based manipulation. The key feature of the algorithm is that it can adapt to the accuracy requirements of a task, by slowing down and generating "careful" motion when the task requires high accuracy, and by speeding up and moving fast when the task tolerates inaccuracy. I formulate the problem as a Markov Decision Process (MDP) with action-dependent stochasticity and propose an approximate online solution to it. I use a trajectory optimizer with a deterministic model to suggest promising actions to the MDP, to reduce computation time spent on evaluating different actions. My real-robot results show that with a task-adaptive planning and control

approach, a robot can choose fast or slow actions depending on the task accuracy and uncertainty level.

5. *Coarse and fine physics models can be combined to generate hybrid models. With such models, physics-based manipulation planning time can be significantly reduced, without sacrificing success rates.*

   Given an initial state and a sequence of controls, the problem of predicting the resulting sequence of states is a key component of a variety of model-based planning and control algorithms. However, this process is computationally expensive.

   In this thesis, I propose combining a coarse (i.e. computationally cheap but not very accurate) predictive physics model, with a fine (i.e. computationally expensive but accurate) predictive physics model, to generate a hybrid model that is at the required speed and accuracy for a given manipulation task. My approach is based on the Parareal algorithm, a parallel-in-time integration method used for computing numerical solutions for general systems of ordinary differential equations. I adapt Parareal to combine a coarse pushing model with an off-the-shelf physics engine to deliver physics-based predictions that are as accurate as the physics engine but run in substantially less wall-clock time, thanks to parallelization across time. With these hybrid physics models, I can achieve the same success rates as the planner that uses the off-the-shelf physics engine directly, but significantly faster.

6. *A learned coarse physics model can lead to faster Parareal convergence, and thus faster physics-based manipulation planning and control.*

   Parallel-in-time integration methods can help to leverage parallel computing to accelerate physics predictions and thus planning. The Parareal algorithm iterates between a coarse serial integrator and a fine parallel integrator. A key challenge is to devise a coarse model that is computationally cheap but accurate enough for Parareal to converge quickly.

   In this thesis, I investigate the use of a deep neural network (DNN) physics model as a coarse model for Parareal in the context of robotic manipulation. It handles the multi-contact cases as opposed to prior work. More importantly, I find that such a learned DNN model leads to faster Parareal convergence compared to an analytical coarse model. Faster Parareal convergence then leads to faster physics-based manipulation planning and control.

1.2 Contributions

Here, I provide a list of my contributions:

- An on-line re-planning algorithm to address uncertainty during grasping in clutter. It is based on a novel stochastic trajectory optimizer where I minimize a weighted combination of grasping in clutter costs, along a trajectory. This is the first work that shows real-time re-planning cycles in difficult and cluttered real-robot physics-based manipulation environments [3].

- A planning and control framework that autonomously switches between open-loop execution (for robust trajectory segments), and model-predictive control (for non-robust segments) to complete a physics-based manipulation task. I formulate the problem of separating a trajectory into robust and non-robust segments as search on a directed robustness graph [4].

- A derivation of divergence metrics through contraction theory, to quantify robustness to state uncertainty, in the presence of real-world model inaccuracies. It is a better predictor of real-world robustness compared to those proposed in prior work [4].

- A novel robust planner based on trajectory optimization. It uses robustness metrics as a cost to be minimized along a trajectory [4].

- A formulation of task-adaptive manipulation planning as a Markov Decision Process (MDP) with action-dependent stochasticity [2].

- An online solution to the stochastic MDP for task-adaptive manipulation. I sample the action space with a trajectory optimizer, to generate actions for evaluation under the MDP setting. The robot generates fast or slow pushes depending on the task [2].

- Proposing learned and analytical coarse physics models for multi-contact physics-based manipulation [5, 6].

- A combination of coarse and fine physics models to speed-up physics predictions during robotic manipulation through parallel-in-time integration [5, 6].

- An extension of the parallel-in-time integration algorithm, Parareal to handle infeasible contact state updates through projections to the feasible state space [5, 6].

- Implementation of the aforementioned algorithms on a real-robot platform, the UR5 with a 2-finger gripper, mounted on the ridgeback, an omni-directional base.

1.3 Roadmap

Chapter 3 introduces the online re-planning algorithm based on trajectory optimization for grasping in clutter. Chapter 4 introduces the robust planning framework that interleaves robust open-loop and closed-loop execution for manipulation. Chapter 5 introduces a task-adaptive planning algorithm that allows a robot to embrace uncertainty, adapt to tasks, pushing fast or slow. Chapter 6 combines coarse and fine physics models through parallel-in-time integration. Chapter 7 introduces a learned coarse model for robotic manipulation.

Chapter 2

Related Work

As robots generate plans that involve both prehensile and non-prehensile actions, the problem of uncertainty is inevitable. This is demonstrated for example by Yu et al. [97], where they collect a million controlled object pushes in the real-world, and show the associated real-world uncertainty. This thesis addresses two major sources of uncertainty - state uncertainty and model inaccuracies. Prior work has addressed manipulation under uncertainty through open-loop execution [52, 24, 102], robust open-loop execution [73, 57, 101], closed-loop control policies [40, 3, 17], and uncertainty-aware controllers [2, 48, 80].

Clutter is another challenge that is encountered during manipulation. Robots often need to make multi-contact interactions to reach for a target object [76, 3], retrieve a target object [15], or re-arrange objects on a table surface [39]. During these contact interactions, physics predictions play a central role, but can be inaccurate and most importantly computationally expensive.

In this thesis I propose a variety of planning and control methods to handle physics-based manipulation under uncertainty. I investigate robust open-loop methods, closed-loop methods, combinations of robust open-loop and closed-loop methods, and uncertainty-aware controllers. This thesis also includes novel work on combining different physics models to generate hybrid models. The goal is to speed-up physics predictions, and thus manipulation planning and control.

In Sec. 2.1 I discuss prior work on manipulation tasks with multiple objects and contact interactions. Sec. 2.2 explains prior work where open-loop execution is used to complete manipulation tasks. Sec. 2.3 details related work on methods that account for uncertainty at the planning stage to generate robust plans. Sec. 2.4 explains prior work on closed-loop policies for manipulation. Sec. 2.5 discusses related work on uncertainty-aware controllers. Sec. 2.6 discusses different methods used to speed-up physics simulators for planning. Sec. 2.7 explains prior work on combining physics models for planning. Sec. 2.8

discusses prior work on using visual dynamics models for robotic manipulation, and Sec. 2.9 summarizes the related works chapter.

2.1 Manipulation in Clutter

Stilman et al. [87], investigated the manipulation planning amongst movable obstacles problem. The task involves static and movable objects and the goal is to place a desired movable object in a target location. They propose an efficient algorithm that generates quick manipulation plans in a highly nonlinear search space of exponential dimension. Here, contact interactions between the robot and objects are not allowed.

Haustein et al. [39] considered the rearrangement planning problem. It involves re-arranging a scene, placing multiple objects in desired locations, through pushes. They propose a planner that solves the rearrangement planning problem through search for dynamic transitions between statically stable states in the joint configuration space, instead of the state space. Thus, they reduced the search space by a factor of two.

Dogar et al. [24] propose a method to solve the grasping through clutter problem. The task is for a robot to reach through multiple objects, and grasp a target object. In the proposed framework, a grasp approach trajectory is planned by considering offline computations of robot-object interactions. This lead to improved grasp success rates, compared to methods that avoided all contact interactions. However, more complex object-object interactions were avoided to make the problem computationally tractable.

Srivastava et al. [86] tackle a similar grasping through clutter problem through a task and motion planning approach. They use off-the-shelf task and motion planners with a new representational abstraction, and show results on a real PR2 robot. However, contact interactions were not considered. Laskey et al. [58] address the grasping in clutter problem. They use a hierarchy of supervisors for learning from demonstrations. They show the learned policy on a custom planar robot, where multi-contact interactions were allowed. More recently, Kitaev et al. [53] also tackle the grasping in clutter problem, through physics-based trajectory optimization. They define a cost for grasping in clutter and minimize it with the iterative linear quadratic regulator (iLQR). They considered robot-object and object-object interactions, but showed results only in simulation.

This thesis considers manipulation tasks that involve multiple objects. Most prior work [87, 39, 24, 86] either completely avoid contact or limit their nature.

Similar to a few prior works [58, 53], I focus on grasping in clutter and push-planning tasks with no restrictions on the nature of contact interactions to be used — all robot-object and object-object contact interactions are allowed.

2.2 Open-Loop Execution

Prior work has presented planners to address the physics-based manipulation problem, with open-loop execution on a real robot. These planners generate a manipulation plan without considering uncertainty, and execute the actions open-loop, without feedback from the environment.

King et al. [52] present a randomized kinodynamic planner to address the rearrangement planning problem. They carefully select the physics model to reduce the state and action space, and thus make the search tractable. Specifically, they use a quasi-static pushing model and show real-robot full arm manipulation results.

Dogar et al. [24] present a planning method for grasping in clutter where multiple robot-object interactions are possible. They pre-compute and cache these robot-object interactions such that physics-based planning can be tractable. The resulting manipulation plans are executed open-loop on a real-robot.

Zito et al. [102] present a two-level rapidly exploring random trees (RRT) planner one for the global path, and the other to plan local pushes. The manipulation problem is split into a planner that works in the object's configuration space, and a local planner in the robot's joint space that moves the object between a pair of RRT nodes.

Cruciani and Smith [21] proposed a three-stage method for in-hand manipulation, with a focus on pivoting. It uses a simplified physics model for pivoting an object in the robot's gripper. The simplified model uses inertial forces from the robot's arm and a controlled rotational friction at the gripper's tip for pivoting. They show real-world, open-loop pivoting motions on a Baxter robot.

Papallas and Dogar [76] include high-level human-operator input at the manipulation planning stage, to simplify planning. Human inputs are in the form of an ordered sequence of objects and their approximate goal locations. The underlying framework uses randomized kinodynamic planning and converts this high-level plan into a low-level manipulation plan. The resulting plans are then executed open-loop on a real-robot.

While these open-loop methods succeed in limited settings, their success rates decrease in more complex uncertain environments with increased multi-contact interactions [3]. In this thesis I conduct experiments that show how open-loop manipulation plans fail in the real-world.

2.3 Robust Open-Loop Execution

To increase success rates during open-loop execution, prior work has considered uncertainty at the manipulation planning stage.

Muhayyuddin et al. [73] propose p-KPIECE, a randomized kinodynamic, physics-based planner that generates robot actions that are robust to both object pose and contact dynamics uncertainty.

Anders, Kaelbling, and Lozano-Perez [9] perform belief-space planning to generate planar robust push actions, through a learned belief-state transition model. The problem involves contact interactions with multiple objects and the deterministic belief-state transition model is trained offline through supervised learning.

Luders, Kothari, and How [65] propose chance-constrained rapidly-exploring random trees. It handles environmental uncertainty by considering the trade-off between planner conservatism and the risk of infeasibility. It has mostly been applied to motion planning problems in the configuration space, without contact interactions.

Johnson, King, and Srinivasa [47] derive divergence metrics to quantify robustness to state uncertainty. It is based on contraction theory [63] which studies the evolution of the infinitesimal distance between any two neighboring trajectories and provides conclusions on the finite distance between them. They use these metrics in randomized kinodynamic planners to generate robust motion plans.

Koval et al. [57] formulate robust open-loop manipulation planning as a best-arm variant of the multi-armed bandit problem. They use a kinodynamic planner to generate manipulation plans and evaluate these plans through noisy rollouts to pick the "best arm". Given a rollout budget, they use the successive rejects algorithm to allocate rollouts between candidates. They tackle the rearrangement planning problem and show robust open-loop plans on a real-robot.

With these robust open-loop planners, success rates increase compared to traditional open-loop execution. However, robust open-loop plans are difficult to find as they may depend on a particular set of "funneling" actions. These

robust plans tend to be pessimistic/conservative and may not exist for many physics-based manipulation problems.

In this thesis, I also seek robust open-loop plans that complete the manipulation task without feedback. However, I propose methods that find and accept segments of a plan that are robust and can be executed open-loop. .

2.4 Closed-loop Control Policies

Feedback during execution can help improve success rates in physics-based manipulation. I discuss two lines of work that include feedback — closed-loop controllers and learned policies.

2.4.1 Closed-loop controllers

Hogan and Rodriguez [41] introduce the pusher-slider problem, where the motion of a slider object is controlled solely through contact from the pusher object. The work uses a mixture of model-predictive control (MPC) and integer programming to capture dynamics constraints and achieve real-time control. However, they rely on a family of contact mode sequences to choose from during planning. With MPC, they execute the first action in a planned trajectory, get feedback, update the internal state, generate a new plan, execute the first action of this new plan, and repeat until the task is complete. Extending the prior work in [41], Hogan, Grau, and Rodriguez [40] search for contact mode sequences offline, and optimal control inputs online in a convex hybrid MPC setting, for reactive planar contact-based manipulation. Such MPC methods have also been used in prior work to stabilize complex humanoid behaviours [90], and visually manipulate fabric [42].

Zhou et al. [101] propose a probabilistic algorithm to sequentially reduce state uncertainty during grasping until an object's pose is uniquely known, under stochastic dynamics. They create an offline planning tree through a combination of open-loop action sequence search and feedback state estimation with particle filtering.

Huang, Jia, and Mason [45] solve a tabletop rearrangement planning problem with policy roll-outs and an iterated local search approach to escape local minima. Controls are executed with MPC on a real-robot, in densely packed environments with up to 100 objects.

Papallas, Cohn, and Dogar [75] consider the reaching through clutter problem under uncertainty. They propose a framework with a human-in-the-loop

during closed-loop execution. In the framework, a robot plans and executes a trajectory autonomously, but can also seek high-level suggestions from a human operator if required at any point during execution.

While these prior methods have achieved improved success rates under uncertainty, a major drawback in online control here is computationally expensive physics predictions that lead to high re-planning times [6, 5].

2.4.2 Learned Policies

Robust learned robot policies have also been developed in prior work for physics-based manipulation.

Bejjani et al. [16] use a learned value function and look-ahead planning, in an online setting, to generate physics-based manipulation plans in clutter. Yuan et al. [98] use deep reinforcement learning to learn an object rearrangement policy which is then further adapted for the real-world through additional real-world manipulation data. Laskey et al. [59] learn grasping in clutter from demonstrations provided by a hierarchy of supervisors, to reduce the burden on human experts to provide demonstrations. Pauly et al. [77] learn to perform robot manipulation tasks from a single third-person demonstration video. Li, Lee, and Hsu [60] propose push-net, a deep recurrent neural network that uses only an image as input to push objects of unknown physical properties. Bejjani et al. [14] address the problem of occlusion in lateral access into shelves, with a hybrid planner based on a learned heuristic. Kiatos and Malassiotis [51] learn an optimal push policy to singulate objects in cutter with lateral pushing actions.

While these learned policies have shown impressive success rates with typically low policy-lags for real-world manipulation tasks, they require lots of training data/demonstrations for a given task, and do not generalize well to new tasks. There has been recent works aimed at making learning more efficient and improving generalization to new tasks. Davchev et al. [22] combine behavioural cloning based dynamic movement primitives (DMP), and a reinforcement learning based residual correction policy, to improve the generalization abilities of DMPs for insertion tasks. Angelov et al. [10] combine motion planning trajectories, dynamic motion primitives and neural network controllers automatically in a hybrid policy to efficiently complete temporally extended manipulation tasks. Zhan et al. [99] propose a framework for efficient robotic manipulation based on data augmentation and unsupervised learning. They demonstrate sample-efficient training of robotic manipulation policies with sparse rewards for tasks like reaching, picking, and moving. While these works have improved learning

efficiency and generalization, the challenge of generalization for learning-based systems across manipulation tasks in a data-efficient manner still remains.

2.5 Uncertainty-Aware Planning and Control

There are recent works that develop uncertainty-aware controllers in robotics, but the focus has mainly been on navigation and collision avoidance. With these systems the robot exhibits adaptive behaviour in the face of uncertainty.

Richter and Roy [80] proposed safe visual navigation with deep learning. Given that current deep learning methods produce unsafe predictions when faced with out-of-distribution inputs, they detect these novel inputs using the reconstruction loss of an autoencoder and switch to a rule-based safe policy. Similarly, Choi et al. [20] switch between a learned policy and a rule-based policy for autonomous driving. However, they estimate uncertainty using a single mixture density network without Monte Carlo sampling. In this thesis, I focus on non-prehensile manipulation where designing a rule-based policy for each new task is not feasible.

Furthermore, Kahn et al. [48] proposed an uncertainty-aware controller for autonomous navigation with collision avoidance. First, they estimate uncertainty from multiple bootstrapped neural networks using dropout. Thereafter, they consider a very large number of fixed action sequences at a given current state. They evaluate these action sequences under an uncertainty-dependent cost function within an MPC framework. The resulting policy chooses low speed actions in unfamiliar environments and naturally chooses high speed actions in familiar environments.While this approach relieves the burden of designing a rule-based policy, it requires the evaluation of a large number of *a priori* fixed action sequences.

Focusing on manipulation, Arruda et al. [11] address the problem of pushing a single object to a desired goal location. They learn a forward robot pushing model together with the corresponding uncertainty in the predictions. They use the forward model in a model predictive path integral controller (MPPI) where uncertainty is part of the cost function to be minimized. MPPI is an MPC style online controller where the optimal control problem is solved mainly through forward sampling of trajectories. The resulting planner avoids high uncertainty regions as it pushes the object towards a desired target location. Abraham et al. [1] extend MPPI to generate robust actions. Specifically, they add uncertainty in physics parameters through an expanded free energy formulation of MPPI. The resulting controller is initially conservative but becomes more exploitative

with more input data. Thus, it is adaptive to model parameter uncertainty. They show robust robot actions in real-world manipulation problems.

In this thesis, I propose methods that embrace uncertainty in a task-adaptive sense, pushing fast or slow. I model the problem as a Markov Decision Process (MDP) and propose an online solution to it, using a trajectory optimizer to guide my action sampling process.

2.6 Speeding-up Physics Simulators

I can make physics engines faster by using larger simulation time steps. However, this decreases the accuracy and can result in unstable behavior where objects have unrealistically large accelerations. To generate stable behaviour at large time-step sizes, Pan et al. [74] propose an integrator for articulated body dynamics by using only position variables to formulate the dynamic equation. Moreover, Fan et al. [27] propose linear-time variational integrators of arbitrarily high order for robotic simulation and use them in trajectory optimization to complete robotics tasks. In this thesis I use physics simulators at the largest possible time-step - at the stability limit. Hence, they run as fast as possible. My goal is to provide much more speed-up beyond this.

2.7 Combining Different Physics Models

Physics predictions for multi-contact interactions are computationally expensive. In this thesis, I combine coarse and fine physics models to speed-up physics predictions. By coarse I mean computationally cheap but inaccurate and by fine I mean accurate but can be computationally expensive. I achieve this combination through parallel-in-time integration. I use a coarse physics model to obtain a rough initial guess of the state at each time point of a trajectory. Then, I evaluate the fine physics model in parallel across time starting from the initial guesses. Thereafter, I combine the coarse and fine predictions using the iterative Parareal algorithm [62, 33]. Coarse models can be quasi-static or even learned [6] and fine models can be a full physics engine like Mujoco [92].

Parareal has been used in many different areas. Trindade et al., for example, use it to simulate incompressible laminar flows [93]. Maday et al. have tested it to simulate dynamics in quantum chemistry [67]. The method was introduced by Lions et al. in 2001 [62].

Combinations of parallel-in-time integration and neural networks have not yet been studied widely. Very recently, Yalla and Enquist showed the promise

of using a machine learned model as coarse propagator [96] for test problems. Going the other way, Schroder [84] and Günther et al. [83] recently showed that parallel-in-time integration can be used to speed up the process of training neural networks.

The underlying equations of motion during multi-contact interactions are differential algebraic equations (DAEs). Results on how Parareal performs for DAEs are scarce. Guibert et al. [36] demonstrate that Parareal can solve DAEs, but can experience issues with stability for very stiff problems. Cadeau et al. [18] propose a combination of Parareal with waveform relaxation to introduce additional parallelism. For a DAE system of size 100,000, they demonstrate that adding Parareal does provide speedup beyond the saturation point of waveform relaxation alone.

Combining different physics models for robotic manipulation has been the topic of other recent research as well, even though the focus has *not* been improving prediction speed. Kloss, Schaal, and Bohg [54] focus on the question of accuracy and generalization in combined neural-analytical models. Ajay et al. [8] focus on modeling of the inherent stochastic nature of the real world physics, by combining an analytical, deterministic rigid-body simulator with a stochastic neural network.

Furthermore, physics predictions are essential in learning physics-based manipulation policies. For example, learning gentle object manipulation through curiosity [46], learning long-horizon robotic agent behaviours through latent imagination [37], learning visuo-motor policies by formulating exploration as a latent trajectory optimization problem [64], learning policies for manipulation in clutter[15], smoothing fabric with a da Vinci surgical robot through deep imitation learning [85], and learning human-like manipulation policies through virtual reality demonstrations [38]. The training time for these policies can potentially be reduced with a parallel-in-time approach to physics predictions.

2.8 Manipulation with Visual Dynamics

Visual dynamics involves using an image to represent the state of a system. Then given an initial image and a sequence of robot controls, the problem of (action-conditioned) visual dynamics is to compute the corresponding sequence of images. Visual dynamics has mainly been used for the manipulation of deformable objects such as ropes and cloth. One reason for that is the difficulty in representing the state of such objects with traditional methods. Finn, Goodfellow, and Levine [30] developed an action-conditioned video prediction model

that explicitly models pixel motion through predictions of distributions over the motion of pixels from prior frames. By explicitly modelling motion, the approach was partially invariant to object appearance in robotic pushing tasks. Babaeizadeh et al. [12] include stochasticity in video prediction and propose the stochastic variational video prediction (SV2P), for multi-step video prediction based on variational inference. They show how including sochasticity can significantly improve video prediction in a range of robotic pushing tasks. Finn and Levine [31] propose deep visual foresight for planning robot motion. They combine action-conditioned video prediction models with model-predictive control to complete robotic pushing tasks. More recently, Hoque et al. [42] take a similar visual foresight approach for sequential manipulation of fabric, but combine domain randomized RGB images and depth maps simultaneously in simulation. They showed how including depth sensing can significantly improve performance for sequential fabric manipulation.

While using visual dynamics for manipulation has seen significant progress in recent years, the challenge of predicting beyond a few future frames still remains.

2.9 Summary

Similar to prior work, this thesis considers manipulation tasks that involve multiple objects. Specifically, unlike some prior work [39, 86, 24], I focus on grasping in clutter and push-planning tasks with no restrictions on the nature of contact interactions to be used. I show that open-loop execution used in prior work [52, 102, 21] can lead to task failures due to uncertainty in state and physics predictions.

This thesis is also related to works in the literature that account for uncertainty at the planning stage to generate robust plans [73, 47, 57]. I seek fully robust open-loop plans that complete the task. However, I propose methods that find and accept partially robust open-loop plans wherever possible.

Prior work [41, 101, 45] has developed closed-loop policies for physics-based manipulation. I take a similar closed-loop approach in this work. A major drawback in online control here is computationally expensive physics predictions that lead to high re-planning times [6, 5]. I aim to find fully robust open-loop plans and fall back to closed-loop control only when needed.

Uncertainty-aware planners and controllers were developed in prior work [80, 48]. The focus has mainly been on navigation and collision avoidance. In this thesis, I propose methods that embrace uncertainty during manipulation in

a task-adaptive sense, pushing fast or slow. I model the problem as a Markov Decision Process (MDP) and propose an online solution to it, using a trajectory optimizer to guide my action sampling process.

Physics simulators can be sped-up by increasing the simulation time-step, but this can lead to unstable simulations. Prior work has proposed different methods to stabilize simulators at large time-steps [74]. In this thesis I use physics simulators at the largest possible time-step — at the stability limit. Hence, they run as fast as possible. My goal is to provide much more speed-up beyond this.

Combining physics models has also been a topic of recent works [54, 8]. Although the focus has not been on improving physics prediction speed. In this thesis, I combine coarse and fine physics models to speed-up physics predictions using parallel-in-time integration.

Chapter 3

Real-Time Online Re-Planning for Grasping Under Clutter and Uncertainty

This chapter considers the problem where a robot must reach through a cluttered environment to grasp a target object. This problem is typically seen in warehouses where robots are required to retrieve items from shelves to fulfil a customer's order, or in our homes where a robot must reach into the fridge to pick up an object. To do this, the robot needs to contact other objects in the environment and push them out of the way (Fig. 3.1). An object that is pushed by the robot may in turn push and dislocate other objects, including the target object. Undesired events can happen during the interaction, such as objects falling off the edge of the surface. The problem is the generation of robust and reactive robot actions that grasp the target object while preventing undesired events from taking place.

Existing work addresses this problem using motion planning followed by *open-loop* execution [53, 52, 24, 73] i.e. the robot executes a sequence of actions one after the other without getting any feedback from the environment. These approaches can be divided into two. The first approach uses motion planning algorithms, e.g. kino-dynamic sampling-based algorithms [52] or trajectory optimization methods [53], within a physics engine to generate the robot trajectory. Trajectories that are produced this way, however, are likely to fail in the face of uncertainty during real-world execution. Consider the scene in Fig. 3.1a, where the target object is near the center of the table. I can model this scene in a physics engine, plan a sequence of actions with a particular choice of physical parameters (e.g. friction coefficients, object masses, object shapes) that take the robot to the grasping goal state. However, if this plan is executed in an open-loop manner in the real world, it can easily fail as the objects will not move exactly as predicted during planning. This is due to the uncertainty

(a) Initial scene **(b)** Contact triggers re-planning

(c) Re-planning to ensure success **(d)** Target object grasped

Figure 3.1: Snapshots from execution with online re-planning.

in the physics model of the physics engine and the assumed physical parameters of the objects.

The second open-loop approach addresses this problem by accounting for the uncertainty during planning. This approach extends motion planning algorithms to generate actions that are robust to uncertainty [73, 24, 47]. Accounting for the uncertainty during planning, however, either requires the planners to limit themselves to a particular set of "funneling" actions, or results in highly conservative/pessimistic planners which return a solution only when the sequential execution of multiple actions are guaranteed to succeed under uncertain dynamics.

This chapter takes a different reactive approach and investigates the potential of closed-loop methods to address uncertainty during grasping in clutter.

One can use a planner to generate a plan to the goal state, execute a portion of this plan, observe the environment, and then re-plan from the resulting state to the goal, repeating this process until task completion. This online re-planning or model predictive control (MPC) approach has been implemented in many areas of robotics, including the problem of pushing a single object [41, 11], but it has not yet been explored for the problem of manipulation in clutter.

The major challenge with online-replanning is that, planning in this domain requires long times. The average planning time reported in the literature for the problem of grasping in clutter is in the order of minutes [73, 52, 53, 24]. Then, under the online-replanning approach, the robot would need to execute a small action, update its world model with feedback and then will need to wait for possibly minutes before it receives the next action from the planner. This long re-planning time makes it impractical for robots to use feedback from the environment in order to create new plans. Thus, this hampers real world applications and is highly undesirable.

I propose an online-replanning approach to address this challenge. First, I extend trajectory optimization methods that use parallel trajectory rollouts [94, 49] in search of a lower-cost trajectory. By performing each roll-out on a different core, I am able to reduce the time each iteration of our planner takes to be equivalent to a single roll-out. Second, I track the deviation of the actual state from the predicted state, and perform replanning only if the state deviation exceeds a threshold. This prevents us from planning at every time step and allows us to have an automatic system that can be adjusted between open-loop execution and standard model predictive control. Finally, I formulate the problem as optimizing a cost function where reaching the goal is not a hard constraint, and therefore even if a quick replanning cycle does not produce a trajectory that reaches the goal (i.e. grasps the target object) within the given time limit, I can still use it if it is a lower-cost trajectory. This is because we can rely on future re-planning cycles to grasp the target object.

Our specific contributions include an on-line re-planning (OR) algorithm to address uncertainty during grasping in clutter. I show that using our approach, one can achieve real time re-planning cycles with a robot in difficult and cluttered real environments. Real robot experimental results can be seen at `https://youtu.be/RcWHXL2vJPc`. Moreover, I compare OR to open-loop execution, particularly to *naive-replanning* (NR), which plans a trajectory, executes it open-loop until the end, checks if the goal is achieved, and repeats this process if not. I show that OR is more successful in grasping the target object in a time limit, produces lower cost execution trajectories, and is faster.

3.1 Physics-based Grasping in Clutter

As shown in Fig. 3.1, I consider the problem where a robot must plan a trajectory from a given initial pose to a final pre-grasping pose to retrieve an item from a cluttered environment. I consider a planar robot consisting of an arm and a gripper as shown in the figure. The robot's state is defined by a vector of joint values $\boldsymbol{q}^R = \{\theta_x, \theta_y, \theta_{rotation}, \theta_{gripper}\}$, where the θ values represent the x-axis prismatic joint, the y-axis prismatic joint, the rotational joint and the gripper's opening joint values, respectively. The scene includes $D+1$ movable dynamic objects. $\boldsymbol{q}^i$ refers to the six-dimensional pose (three translations and three rotations) of each object, for $i = 1, \ldots, D$. $\boldsymbol{q}^{Target}$ refers to the pose of the target object, i.e. the object to be grasped. I assume a flat surface with edges, such as the table in Fig. 3.1, and dropping any object off the edges is undesired.

I use $\boldsymbol{x}_t$ to represent the complete state of our system at time t, which includes the state of the robot and all objects; $\boldsymbol{x}_t = \{\boldsymbol{q}^R, \boldsymbol{q}^1, \ldots, \boldsymbol{q}^D, \boldsymbol{q}^{Target}\}$. I consider a control input $\boldsymbol{u}_t$ applied at time t for a fixed duration Δ_t. The controls in our case are velocities applied to the robot's degrees of freedom; $\boldsymbol{u}_t = \{\dot{\theta}_x, \dot{\theta}_y, \dot{\theta}_{rotation}, \dot{\theta}_{gripper}\}$. Then, the discrete time dynamics of the system is defined as:

$$\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t, \boldsymbol{u}_t) \tag{3.1}$$

where f is the state transition function.

I assume an initial state of the system, $\boldsymbol{x}_0$, and I define our goal as generating a sequence of control inputs, such that the gripper grasps the target object as quickly as possible, without dropping objects off the table. I use the notation $\boldsymbol{u}_{0:n-1}$ to represent a sequence of control signals through n time steps, each applied for a fixed duration. Similarly, I use $\boldsymbol{x}_{0:n}$ to represent a sequence of states.

I use a physics engine [92] simulating rigid-body dynamics to model f. Nevertheless, any physics engine is an inaccurate model of the real-world physics and uncertainties over the system dynamics are inevitable. Indeed even if I assumed perfect modelling, it is difficult for a robot to know the exact geometric, frictional, and inertial properties of objects in an environment. In addition, object tracking systems come with inaccuracies in the estimation of object poses in an environment. Therefore, our objective in this chapter is to find a sequence of controls that would move the system to a goal state even under an inaccurate model of the system and its dynamics.

Figure 3.2: Goal cost terms

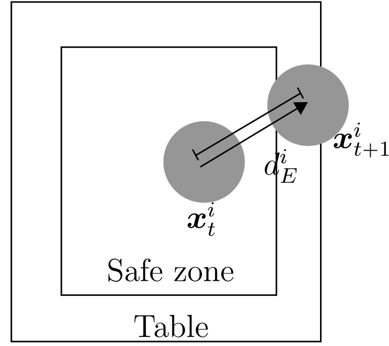


Figure 3.3: Edge cost terms

3.2 Physics-based Grasping through Online Re-planning

To address the inaccuracies mentioned above, I propose to use an online re-planning approach, where the robot makes a plan, executes a portion of it, observes the resulting state, and re-plans.

Below, in Sec. 3.2.1, I first present the planner that I use to generate a sequence of controls to the goal from a given state. In Sec. 3.2.2, I show how I use this planner within an online-replanning framework. In Sec. 3.2.3, I present the baseline approach I compare against in this chapter.

3.2.1 Physics-based trajectory optimization

Recent stochastic trajectory optimization methods such as STOMP [49] and model predictive control methods such as MPPI [94] show impressive speed by using parallel rollouts. Moreover, since these are optimization-based methods, even when they are used with a small time limit, they can still output an improved lower-cost trajectory, even if the trajectory is not necessarily reaching a goal state. In contrast, sampling-based planners such as RRTs and PRMs [52, 73] typically do not return a useful solution unless they are run until a path to the goal is found, which can take minutes. To the best of our knowledge, such parallelizable stochastic trajectory optimization methods have not yet been used to solve grasping in clutter problems. However, the properties I mention above make parallelizable stochastic trajectory optimization methods a promising approach for online re-planning to address problems in this domain.

I formulate the following problem:

$$\min_{\boldsymbol{u}_{0:n-1}} [w_g \cdot c_g(\boldsymbol{x}_n) + \sum_{t=0}^{n-1} (w_a \cdot c_a + w_d \cdot c_d + w_e \cdot c_e)] \tag{3.2}$$

$$s.t. \quad \boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t, \boldsymbol{u}_t),$$

$$\boldsymbol{x}_0 \ is \ fixed, \quad \boldsymbol{u}_t = 0 \ for \ t < 0.$$

where I search for an optimal sequence of controls $\boldsymbol{u}_{0:n-1}$ that minimizes the weighted combination of costs. I use four cost terms; c_g, c_d, c_e, c_a and corresponding weights w_g, w_d, w_e, w_a.

- $c_g(x_n) = d_T^2 + w_\phi \cdot \phi_T^2$. This is the terminal *goal cost* term, quantifying how far the robot hand is from grasping the target object at the final state. I illustrate how the distance d_T and the angle ϕ_T are computed in Fig. 3.2. I first draw a vector from a fixed point in the gripper to the target object. d_T is the length of this vector, i.e. the distance between the fixed point in the gripper and the target object. ϕ_T is the angle between the forward direction of the gripper and the vector. I use w_ϕ to weight angles relative to distances.

- $c_d(\boldsymbol{u}_{t-1:t}, \boldsymbol{x}_{t:t+1}) = \sum_i^D (\boldsymbol{x}_{t+1}^i - \boldsymbol{x}_t^i)^2$. This is the *disturbance cost* term, quantifying how much each object moved between two timesteps. This term encourages the robot to minimize the change in the configuration of the rest of the scene.

- $c_e(\boldsymbol{u}_{t-1:t}, \boldsymbol{x}_{t:t+1}) = \sum_i e^{\{k \cdot d_E^i\}}$ for all i out of the safe zone. This is the *edge cost* term, penalizing those objects that get too close to the boundary of the table or that get out of the boundary. As I illustrate in Fig. 3.3, I define a safe zone that is smaller than the boundary of the table. If at time $t+1$ an object i is out of this safe zone, I compute the distance it is pushed between t and $t+1$, which I define as d_E^i. k is a constant term. I do not add any edge costs for objects that are in the safe zone.

- $c_a(\boldsymbol{u}_{t-1:t}, \boldsymbol{x}_{t:t+1}) = ||\boldsymbol{u}_t - \boldsymbol{u}_{t-1}||^2$. This is the *acceleration cost* term, with which I penalize large changes in robot velocities between two time steps.

Note that, instead of imposing the terminal grasping state as a hard constraint, I declare it as a cost term, c_g. I am able to accept trajectories that

Algorithm 1: Physics-Based Stoch. Traj. Optim. (PBSTO)

Input : $\boldsymbol{x}_0$: Initial state
$\boldsymbol{u}_{0:n-1}$: Initial control sequence
I_{max}: Maximum number of iterations

Output : $\boldsymbol{u}_{0:n-1}$: Control sequence
$\boldsymbol{x}_{0:n}$: Predicted states

Parameters : K: Number of noisy trajectory rollouts
ν: Sampling variance
C_{thresh}: Cost threshold implying success
n_{min}: Minimum number of time steps

Subroutines: $Cost$: Computes total cost, i.e. the minimized value in Eq. (2).

```
1  x_{0:n} ← Roll out u_{0:n-1} over x_0 to get initial state sequence
2  while I_max not reached and Cost(u_{0:n-1}, x_{0:n}) > C_thresh do
3      for k ← 0 to K − 1 do
4          x_0^k ← x_0
5          u_{0,n-1}^k ← N(u_{0:n-1}, ν)
6          for t ← 0 to n − 1 do
7              x_{t+1}^k ← f(x_t^k, u_t^k)
8              if Cost(u_{0:t}^k, x_{0:t+1}^k) ⩽ C_tresh and t ⩾ n_min then
9                  return (u_{0:t}^k, x_{0:t+1}^k)
10     k* ← arg min_k (Cost(u_{0:n-1}^k, x_{0:n}^k))
11     if Cost(u_{0:n-1}^{k*}, x_{0:n}^{k*}) < Cost(u_{0:n-1}, x_{0:n}) then
12         u_{0:n-1} ← u_{0:n-1}^{k*}
13         x_{0:n} ← x_{0:n}^{k*}
14 return (u_{0:n-1}, x_{0:n})
```

do not reach the goal completely, because I use this planner in a re-planning framework, i.e. I can rely on future re-planning cycles to take us to the goal.

I solve this problem using Alg. 1, which adapts the STOMP algorithm [49] for physics-based grasping through clutter.

I start with an initial candidate control sequence $\boldsymbol{u}_{0:n-1}$. During each iteration between lines 2-13, I try to improve this control sequence, until the cost is lower than a threshold, or until a maximum number of iterations is reached (Line 2). During each iteration, I create K new control sequences, roll out these controls in parallel using our model of the system, and compute the cost for each (Lines 3-9), using the $Cost(.)$ subroutine which is based on cost terms in Eq. 3.2. Each new control sequence $\boldsymbol{u}^k_{0:n-1}$ is created by adding stochastic noise to the candidate control sequence $\boldsymbol{u}_{0:n-1}$ (Line 5). The control sequence with the minimum cost is then identified and set as the new candidate control sequence. The cost threshold C_{thresh} is chosen for the case where the robot gets the target object in its gripper at the final state and where no failure occurs

(e.g. no objects are toppled).

Most robot motion planners that use trajectory optimization formulate the problem as a fixed horizon problem, i.e. with a pre-determined number of timesteps/waypoints. In the problem of grasping in clutter however, the length of the required trajectory can change significantly: For example, the target object may be pushed and moved away from its initial position, and this may require a much longer trajectory than a case where the target is grasped at its original position. Therefore, I initialize the planner with a long enough control sequence, but also allow it to short-cut trajectories if the cost indicates success earlier (Lines 8-9). Moreover, physics-based trajectory roll outs are time consuming, hence truncating the roll out when success has been achieved leads to lower planning times.

3.2.2 Online Re-planning

If allowed to run for many iterations, i.e. with a large I_{max}, Alg. 1 can generate successful plans for the problem of grasping under clutter, as I show in our results in Sec. 3.3. However, when executed open-loop, these plans are likely to fail due to the uncertainties in the system dynamics, inaccuracies in the physical properties of the objects, and the state observations. To address this, I use Alg. 1 within an online re-planning (OR) algorithm, which I present in Alg. 2.

On line 2, I generate a locally optimal open-loop trajectory by calling the PBSTO planner with a large number of iterations. Then I start executing this trajectory. After execution of every control action (line 4), I observe the current state (line 7), and then re-plan from this current state (line 12). However, when I re-plan, I call the planner with only a few iterations, to receive fast, close to real-time, updates to the plan. I warm-start the trajectory optimizer by providing the previous plan. Furthermore, I re-plan only if it is necessary. To do this, I check if the final predicted state of the current plan grasps the target object (line 8), and I check if there are too few controls left in the plan (line 10). More importantly, if the real observed state is evolving according to the planner's predictions, and the other previously mentioned conditions are still satisfied, I do not re-plan. I check this on line 9, where I compute the deviation between the observed state and the first state of the planned trajectory, and verify if this deviation is less than a threshold. This threshold can be used to adjust how reactive the system is to unexpected events.

Algorithm 2: Online Re-planning (OR)

Input : $\boldsymbol{u}_{0:n-1}$: Initial controls, e.g. straight line motion
Params: SD_{thresh}: State deviation threshold
n_{min}: Minimum number of controls to optimize
$ManyIter$: Large number of iterations, e.g. 50
$FewIter$: Small number of iterations, e.g. 1

1 $\boldsymbol{x}_{current} \leftarrow$ Observe current state
2 $(\boldsymbol{u}_{0:n-1}, \boldsymbol{x}_{0:n}) \leftarrow$ PBSTO($\boldsymbol{x}_{current}$,$\boldsymbol{u}_{0:n-1}$,$ManyIter$)
3 **while** *target object not grasped* **do**
4 Execute $\boldsymbol{u}_0$
5 Remove $\boldsymbol{u}_0$ from sequence, i.e. $\boldsymbol{u}_{0:n-2} \leftarrow \boldsymbol{u}_{1:n-1}$
6 Remove $\boldsymbol{x}_0$ from sequence, i.e. $\boldsymbol{x}_{0:n-1} \leftarrow \boldsymbol{x}_{1:n}$
7 $\boldsymbol{x}_{current} \leftarrow$ Observe current state
8 **if** *target object not predicted to be grasped at* $\boldsymbol{x}_n$
9 ***or*** *large state deviation, i.e.* $||\boldsymbol{x}_0 - \boldsymbol{x}_{current}|| > SD_{thresh}$
10 ***or*** *too few controls left, i.e.* $n - 1 < n_{min}$ **then**
11 $\boldsymbol{u}_{0:n-1} \leftarrow \boldsymbol{u}_{0:n-2}$ + single straight step to target
12 $(\boldsymbol{u}_{0:n-1}, \boldsymbol{x}_{0:n}) \leftarrow$ PBSTO($\boldsymbol{x}_{current}$,$\boldsymbol{u}_{0:n-1}$,$FewIter$)
13 **else**
14 $n \leftarrow n - 1$ Decrement length of controls

Algorithm 3: Naive Re-planning (NR)

Params: $ManyIter$: Large number of iterations, e.g. 50

1 **while** *target object not grasped* **do**
2 $\boldsymbol{x}_{current} \leftarrow$ Observe current state
3 $\boldsymbol{u}_{0:n-1} \leftarrow$ initial controls, e.g. straight to target object
4 $(\boldsymbol{u}_{0:n-1}, \boldsymbol{x}_{0:n}) \leftarrow$ PBSTO($\boldsymbol{x}_{current}$,$\boldsymbol{u}_{0:n-1}$,$ManyIter$)
5 Execute $\boldsymbol{u}_{0:n-1}$

3.2.3 Naive Re-planning

Open-loop execution during grasping in clutter can be unsuccessful due to uncertainty. In this chapter I propose to address this problem through online feedback control. However, a naive approach to fixing this problem can be replanning if success is not achieved after the complete open-loop execution of a plan. I present this Naive Replanning (NR) approach in Alg. 3, and use it as a baseline in our experiments.

3.3 Experiments and Results

Through our experiments, I compare the online replanning (OR) approach with the naive replanning (NR) approach. I hypothesize that OR is more successful in grasping the target object, that OR results in an execution cost that is

lower-cost, and that OR is also faster. I investigate whether using the physics-based stochastic trajectory optimization (PBSTO) method, I can reactively re-plan close to real-time, or at least fast enough to avoid noticable delays during execution.

I implemented our algorithms using the Mujoco [92] physics engine. I perform experiments both in simulation and on a real robot. As shown in Fig. 3.1, I assume a world consisting of objects on a table, and a planar robot with a two finger gripper. I make a distinction between two different type of worlds I deal with.

Planning world: The planning world is a simulation environment where the robot generates its plans/controls.

Execution world: The execution world is the environment where the robot executes actions and observes the resulting actual state. The execution world is simulated for the simulation experiments and it is the physical world for real robot experiments.

Whether in simulation or on the real robot, I assume a mismatch between the physics of the execution world and the planning world, the physical object properties of the two worlds, and the state of the two worlds. I use the term *uncertainty level* to refer to the degree of this mismatch. For example, *no uncertainty* implies a perfect match between the Planning World and the Execution World, which is only possible in simulation experiments. *Low uncertainty* implies a low level of mismatch, and so on.

3.3.1 Simulation experiments

I perform experiments in simulation to evaluate the performance of our planners in scenes with varying degrees of clutter and uncertainty. I begin by creating *execution worlds*. Here, the *execution world* is created in Mujoco and it consists of 15 objects (boxes and cylinders), a $0.6m \times 0.6m$ table and our planar robot as shown in Fig. 3.4.

For each execution world:

- I randomly select a shape (box or cylinder) for each of the 15 objects.
- For each object, I randomly select[1] shape dimensions (extents for the boxes, radius and height for the cylinder), mass, and coefficient of friction.

[1]The uniform range used for each parameter is given here. Box x-y extents: $[0.03m, 0.05m]$; box height: $[0.036m, 0.04m]$; cylinder radius:$[0.035m, 0.04m]$; cylinder height:$[0.04m, 0.055m]$; mass:$[0.2kg, 0.8kg]$; coef. fric.:$[0.2, 0.6]$.

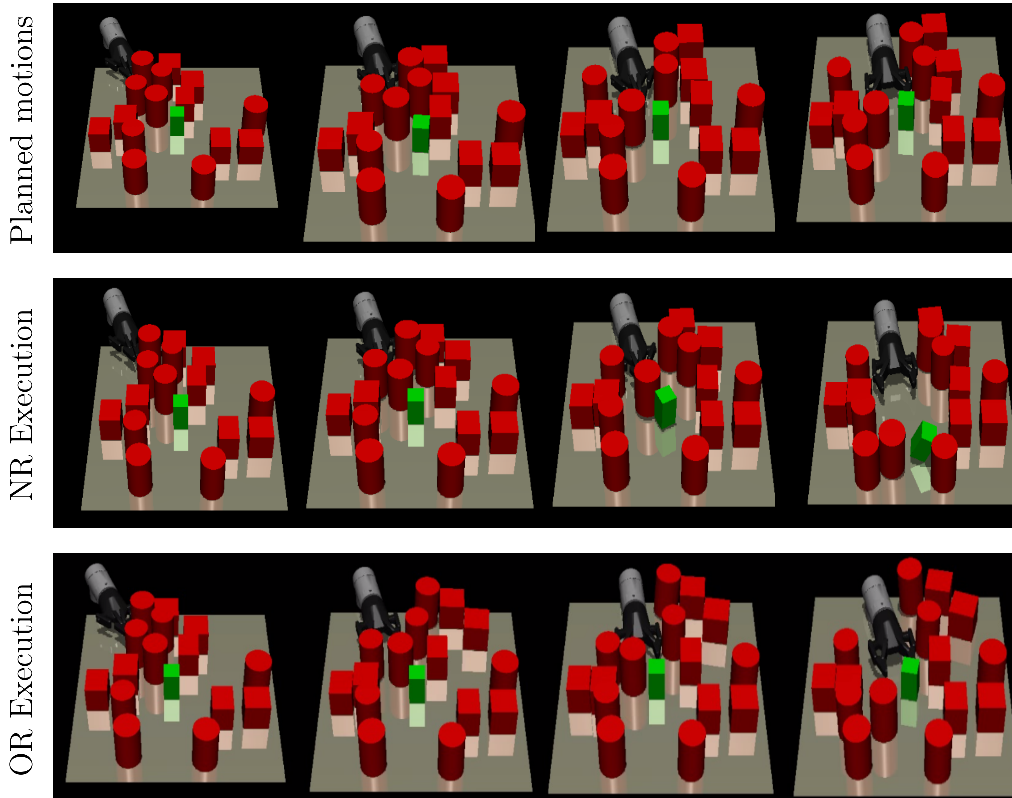


Figure 3.4: Top row: The planned control sequence and state evolution. Middle row: Open-loop execution of the planned control sequence fails under medium uncertainty. Bottom row: Online re-planning, OR succeeds under medium uncertainty.

- I select a pose for the target object from a Gaussian with a mean at the center of the table and a variance of $0.01m$.
- For the other 15 objects, I randomly select non-colliding object poses on the table.

I generate 100 such execution worlds. To generate a planning world from an execution world, I add Gaussian noise onto the physical parameters of the execution world[2]. For each execution world, I create four such planning worlds with increasing amounts of noise, corresponding to the four uncertainty levels: no uncertainty, low, medium, and high uncertainty. Given a pair of Planning world and Execution world, I then run and execute one of our planners. Moreover, I simulate physics stochasticity in the execution world by adding Gaussian

[2]The variance of the Gaussian noise for each parameter under low-uncertainty are given here. These values are multiplied by 2 for medium, and 3 for high uncertainty. Object pose translation: 0.005; Object pose rotation around vertical axis: 0.005; Box x-y extents, cylinder radius, and height: 0.005; mass:0.01; coef. fric.:0.005.

noise[3] on the velocities (linear and angular) $\boldsymbol{v}$ of the robot and dynamic objects at every simulation time step, using the Mujoco simulator.

$$\tilde{\boldsymbol{v}} = \boldsymbol{v} + \boldsymbol{\mu}, \quad \boldsymbol{\mu} \sim \mathcal{N}(0, \boldsymbol{\beta}) \tag{3.3}$$

where $\mathcal{N}$ is the Gaussian distribution and β is the vector of variances. I give each planner a timeout of 15 minutes, which includes all planning, re-planning, and execution times. A planner may return long before this timeout, if the robot manages to grasp the target object in the execution world. I run and compare the following planners:

- NR: The naive re-planning algorithm, with $ManyIter = 50$, $\nu = 0.008$, $K = 8$.
- OR: The online re-planning algorithm, with $ManyIter = 50$, $FewIter = 1$, $n_{min} = 2$, $\nu = 0.008$, $K = 8$, $SD_{thresh} = 0.5$.

For all planners, I initialize the control sequences to straight line trajectories towards the goal. Each initial control sequence includes six actions, with an average resultant velocity of $0.04m/s$. Each action is executed for $\Delta_t = 1s$. The weights and constants used in the cost terms are: $w_g = 10000$, $w_\phi = 1.0$, $w_e = 1.0$, $k = 1000$, $w_a = 0.1$, $w_d = 800$.

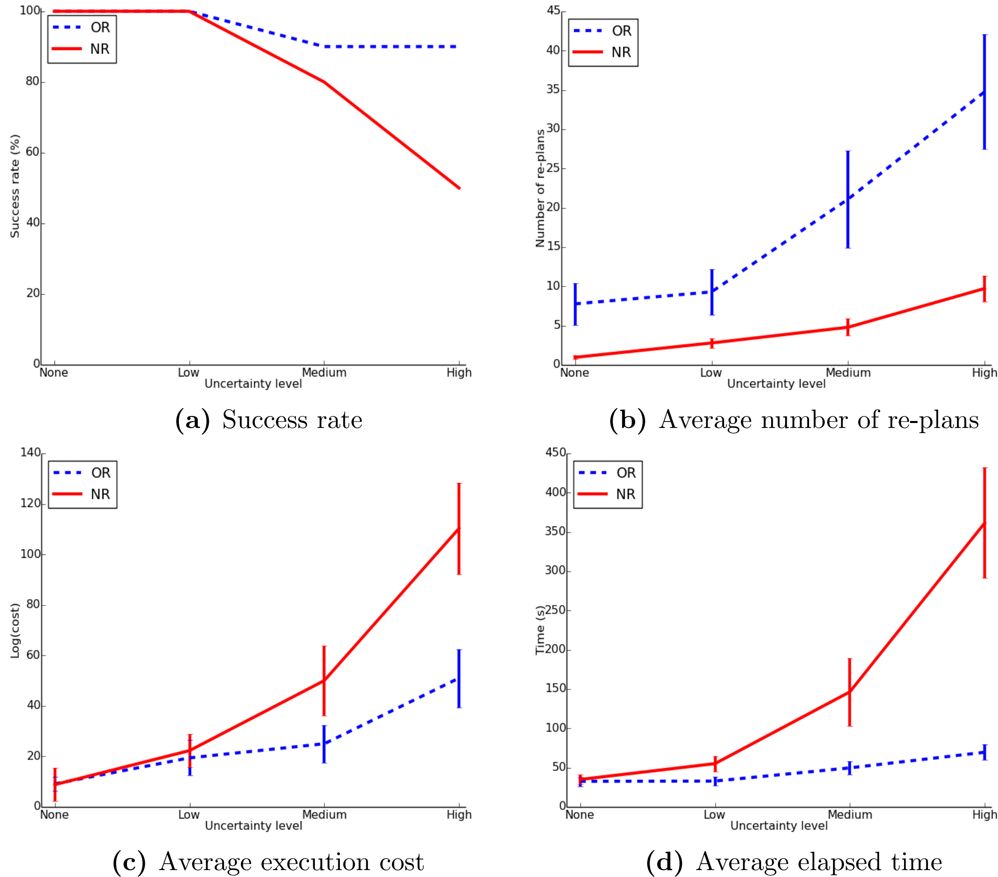
3.3.2 Simulation Results

I discuss and compare the performance of OR and NR.

OR is more successful than NR. I call an experiment success, if the execution stopped with the target object inside the hand pre-grasp region and if no other object is dropped off the table. I show the success rates over the 100 random scenes under four different uncertainty levels in Fig. 3.5a. Both OR and NR succeed in all scenes for the no uncertainty and low uncertainty conditions. However, as the uncertainty increases, NR shows a dramatic drop to 50% success rate, while OR can maintain 90%.

I show example plans in Fig. 3.4. In the top row, I show the output of the planner and the state sequence as predicted by the planner in the Planning World. In the middle row, I show the NR execution of the same scene in the Execution World with noise added at the medium uncertainty level. As the hand pushes on a cylinder, it does not move out of the way as the planner

[3] In the case of no uncertainty, I did not add any extra noise to the system dynamics. However, the vector of variances of the added Gaussian noise for each object was $\boldsymbol{\beta} = \{0.003, 0.006, 0.009\}\mathbf{1}$ for low, medium and high uncertainty levels respectively.



(a) Success rate

(b) Average number of re-plans

(c) Average execution cost

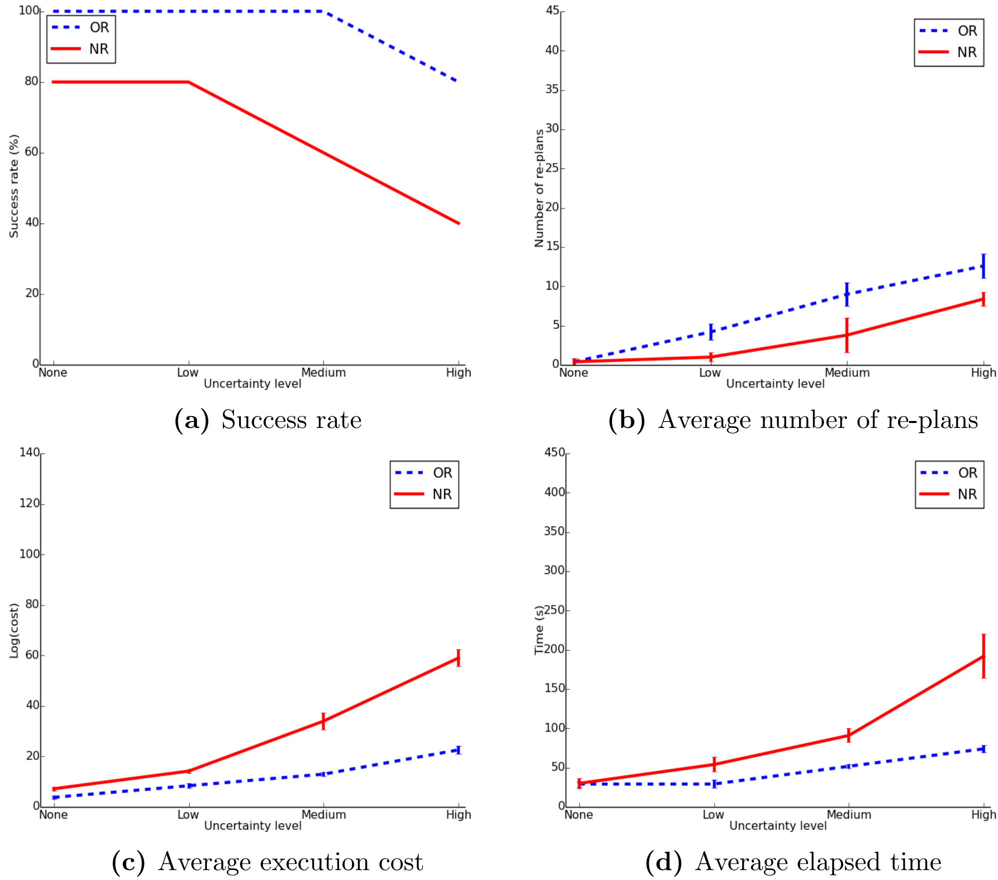
(d) Average elapsed time

Figure 3.5: Simulation results for 100 random scenes. In b-d, I plot the average with 95% confidence interval of the mean.

predicted. It pushes and topples the target object, resulting in a failure. In the bottom row, I show the OR execution in the same Execution World. Detecting that the cylinder does not move as predicted, OR re-plans and shifts the gripper to the side, so that the cylinder can be pushed out of the way.

It is important to note that, the success rates in Fig. 3.5a are not attained after one planning and execution cycle. In other words, the 100% success rate for NR under low uncertainty does **not** mean that open-loop executions of all plans were successful in this case. Instead, it is more often that the execution of an open-loop plan fails, but leaves the robot at a close enough point to the target that, the subsequent plans achieve success. I present Fig. 3.5b to explain this, which shows the average number of re-plans of each planner under varying uncertainty. As can be seen, both planners show increasing number of re-plans with increasing uncertainty. Although, each re-plan is much cheaper for online re-planning compared to naive re-planning.

OR generates lower execution cost than NR. After execution of a planner is completed successfully, I compute the total cost of the executed trajectory. Fig. 3.5c shows the average execution costs of the planners in log



(a) Success rate

(b) Average number of re-plans

(c) Average execution cost

(d) Average elapsed time

Figure 3.6: Real robot results for 5 random scenes. In (b)-(d), I plot the average with 95% confidence interval of the mean

scale versus uncertainty. I use only the successful plans for this plot since failure examples run until the arbitrary time limit I have set (15 minutes), and can accumulate arbitrarily large costs. Again, while OR and NR perform similar at low uncertainties, the execution cost of NR grows significantly with increasing uncertainty.

OR is faster than NR. I record the total re-planning and execution time a planner takes after the robot makes its first move. Again, I use the successful examples only, since failure examples run until the pre-set time limit of 15 minutes and therefore do not give an indication of speed. I plot this total elapsed time in Fig. 3.5d. Observe that the time NR takes grows rapidly with uncertainty, while OR is much faster in reaching the goal.

Note that, the above plot does not include the time spent to find the initial control sequence. I use the PBSTO planner to find this initial sequence as well for both OR and NR, with a limit of 50 iterations. Averaged over 400 runs (100 scenes, noisified four different ways), the PBSTO planner needed **28 seconds** with a standard deviation of 16 to find a plan.

The advantage of the PBSTO planner, however, is that it can also be used

successfully with a small number of iteration limit to quickly adapt plans under uncertainty. For online re-planning (OR), I ran the PBSTO planner with the iteration limit of 1 for these quick updates. During 400 executions, the OR planner performed 7816 such re-plans. On average, each such update took **0.4 seconds** with a standard deviation of 0.25 seconds. Therefore, I am able to perform online grasping through clutter in near real time. Moreover, in comparison with works in the literature [73, 52, 53, 24] about grasping in clutter where the average planning time is in the order of minutes, our approach shows significantly lower planning and re-planning times.

3.3.3 Real robot experiments

Figure 3.7: Top row: Naive re-planning (no added uncertainty) fails to grasp the target (in green). Bottom row: Online re-planning succeeds.

In the real robot experiments, I use a Robotiq two finger gripper attached to a UR5 arm which is then mounted on an omnidirectional robot (ridgeback). As shown in Fig. 3.1, I fix the orientation of the arm relative to the table such that is at a specified height, above the table and is parallel to it. This way the gripper moves with the omnidirectional base yielding a 4 degrees of freedom robot. The gripper velocities which is the output of our optimization is then transformed to the omnidirectional base through a fixed velocity transform. I place markers on objects (cylinders and boxes) and sense their full pose (position and orientation) in the environment using the OptiTrack motion capture system.

I create $N = 5$ execution worlds. I created a mix of difficult (where the target object is behind many closely packed objects) and easy (where the target object is easily accessible) scenes for the experiments. All these scenes can be seen

in our video at `https://youtu.be/RcWHXL2vJPc`. Then, I create a planning world by using estimated values of mass and shape of objects and then get the pose information from our motion capture system. In addition, I sample the coefficient of friction for the various objects from a multivariate Gaussian distribution with a mean of 0.5 and a variance of 0.01.

I am aware that motion capture systems provide a level of object tracking performance which cannot be achieved by using a standard vision system especially in clutter. Therefore, to see how our online replanning approach would cope in reality with vision systems, I perform experiments where I artificially insert different levels of pose (x,y positions) uncertainty. I do this by sampling from a Gaussian distribution where the mean is the measured position from our motion capture system. I select a variance of $\{0.005, 0.01, 0.015\}m$ for low, medium and high uncertainty levels respectively.

3.3.4 Real robot experimental results

I ran a total of 40 real robot experiments for 5 scenes and 4 uncertainty levels using both naive re-planning and online re-planning. Our results are shown in Fig. 3.6. In general they are similar to the simulation experiments. Moreover, in Fig. 3.6a, the naive re-planning approach is not always successful even when no artificial uncertainty is added. This is due to the inherent uncertainty in the real world dynamics.

In Fig. 3.7, I show an example scene from our real robot experiments. The naive re-planning approach (top row) was not successful in grasping the target object even under no additional uncertainty. The reason for this is the inherent uncertainty in the real world. More specifically, it is due to the mismatch between the planning environment in simulation and the real world especially in terms of object shape, mass, and friction coefficient. Moreover, the real objects are not fully rigid bodies. Hence predictions of physics in the real world becomes difficult especially for cases where the robot pushes on multiple objects in contact with each other (second snapshot, top row). Therefore, at the end of an open-loop execution in the real world, the robot can put the state of the system in a dead-end (fourth snapshot, top row) from which recovery and task completion becomes extremely difficult. On the other hand, our online re-planning approach shown in the bottom row succeeds in this scene. It is able to track changes between a planned trajectory and the actual state trajectory in the real world. I re-plan if the changes are large and continue this process until the robot successfully grasps the target object. Videos of sample executions can be found at `https://youtu.be/RcWHXL2vJPc`.

3.4 Discussion

To the best of our knowledge, this is the first work that shows how a robot can complete physics-based manipulation in clutter with online planning in real time. Compared to traditional MPC, the online re-planning approach I introduce in this chapter does not re-plan at every step. It re-plans when necessary as directed by a state deviation threshold parameter. Thus, it includes partially open-loop plans.

Our problem set-up includes many simplifications though. Most importantly, I do not consider static obstacles which may create jamming effects between the robot and the objects. These are considered in subsequent chapters.

Chapter 4

Robust Physics-Based Manipulation by Interleaving Open and Closed-Loop Execution

In this Chapter, I seek more efficient methods to achieve robotic manipulation success under uncertainty, and to realize more fluent/real-time manipulation plan execution. Let's consider the scene in Fig. 4.1, where static obstacle jamming effects make the grasping in clutter problem much more difficult. The goal is to reach for the green can without pushing other objects off the shelf.

This problem can be solved with a fully closed-loop approach like I introduced in Chapter 3. However, three problems remain. First, re-planning times are still non-zero and can induce noticeable delays during a robot's manipulation plan execution. This is aesthetically not desirable. Second, full state estimation is required with such closed-loop systems at every step during execution, this may not be possible for some manipulation tasks. Finally, if re-planning time is high during dynamic tasks, some objects can quickly roll and fall-off a table before re-planning is complete, leading to task failures.

Prior work has investigated the generation of robust open-loop manipulation plans to solve this problem. These robust open-plans are guaranteed to succeed in the face of uncertainty. They may include certain funneling actions that reduce uncertainty. For example, grasps where the robot pushes an object towards a shelf wall, caging the object and uniquely identifying its position, before grasping it. Such completely open-loop roust plans that complete the task are difficult to find, or may not exist for many physics-based manipulation problems.

In this chapter, I propose a planning and control framework that tightly integrates open and closed-loop execution for physics-based manipulation. It

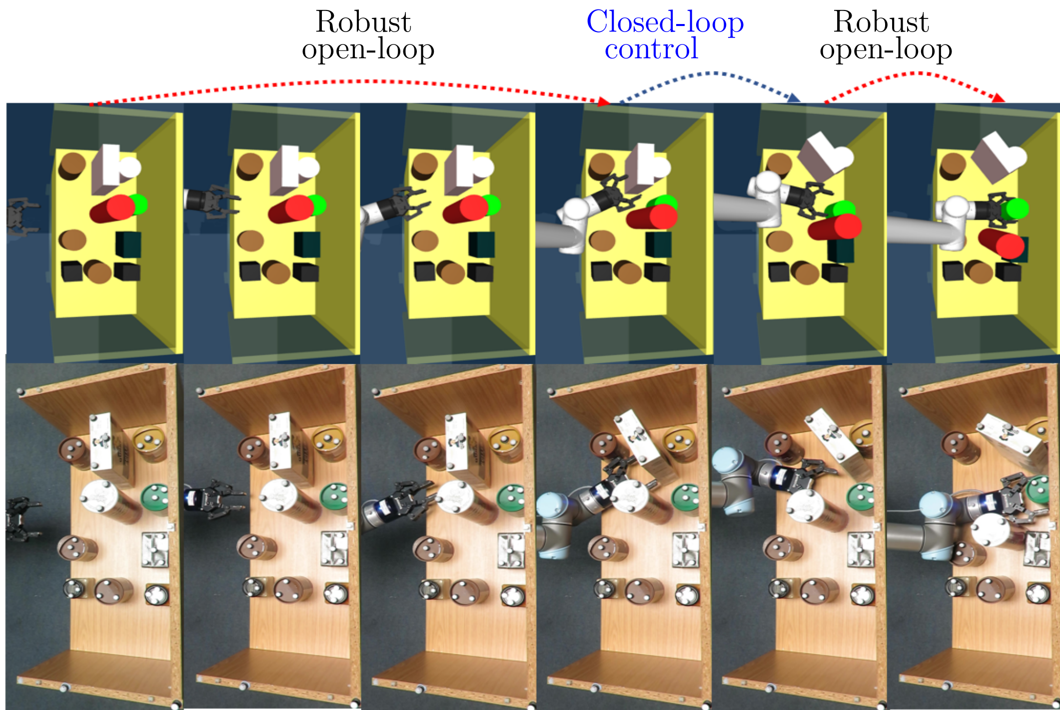


Figure 4.1: A combination of robust open-loop execution and closed-loop control to reach for the green target object. Top row: Planned trajectories and execution strategy. Bottom row: Real-robot open and closed-loop execution to reach for the green target object. The robot starts with robust open-loop loop execution due to motion in free-space. It falls back to closed-loop control during multi-contact interactions. Finally, it uses a robust funnelling motion to get the green target object in the gripper.

autonomously generates a combination of robust open-loop plans (wherever possible) and closed-loop controllers (wherever needed) to complete a manipulation task.

Consider the scene in Fig. 4.1. This is a scene that requires a large number of contact interactions. A robust open-loop plan that solves the complete task here is difficult to find due to uncertainty in state and more importantly in physics predictions. A closed-loop physics-based controller may be successful but will be slow due to computationally expensive physics predictions, re-planning at each step to reach the goal.

My approach generates a three-part plan for this scene. The first part is a robust open-loop plan. It is possible mainly due to the near free-space motion of the robot. The second part is executed closed-loop since there is a significant amount of contact interactions. The final part is also a robust open-loop plan. While there are still significant contact interactions, the final part involves a funnelling action that reduces uncertainty through object contact with the shelf.

How do I detect what parts are robust open-loop and what parts should be closed-loop? I need a metric that quantifies robustness to state uncertainty and model inaccuracies.

I propose such a robustness metric in this work, based on contraction theory [63]. Prior work by Johnson, King, and Srinivasa [47] and Kong and Johnson [55] have proposed robustness metrics for state uncertainty. I build on these works, and derive new metrics to quantify robustness to not only state uncertainty, but more importantly model inaccuracies. I use the metric in a robust planner based on trajectory optimization.

After robust planning for a given manipulation task, I analyze the plan to extract robust parts which are executed open-loop and non-robust parts which are executed with model-predictive control. I separate a plan into robust and non-robust segments through search on a directed graph, where nodes are time-points of a trajectory and edges are robust or non-robust connections between these time-points.

I found that the interleaved open and closed-loop execution approach leads to significant success under uncertainty with fluent/real-time execution, compared to baselines in the literature.

I make the following contributions:

- A planning and control framework that autonomously switches between robust open-loop execution, and model-predictive control to complete a physics-based manipulation task.
- A derivation of divergence metrics through contraction theory, to quantify robustness to state uncertainty, in the presence of real-world model inaccuracies.
- A novel robust planner based on trajectory optimization.

The rest of this chapter is organized as follows: Sec. 4.1 formulates the problem. Sec. 4.2 provides an overview of the interleaved open and closed-loop execution approach. Sec. 4.3 provides a background on contraction theory and my robustness metric derivation. Sec. 4.4 introduces the robust planning and control framework. Sec. 5.6 details robot experiments and results. Sec. 4.6 concludes the chapter.

4.1 The Robust Manipulation Problem

The robot's goal is to retrieve an item from a cluttered environment under state uncertainty and model inaccuracies. It starts from an observed initial state and

generates non-prehensile actions to reach a final pre-grasping state. A scene includes at most D movable dynamic objects. $\boldsymbol{q}^i,\ i = 1, \ldots, D$, refers to the full pose of each dynamic object. The robot's pose is defined by a vector of joint values $\boldsymbol{q}^R$. I represent the complete state of my system as $\boldsymbol{x}_t \in \mathbb{R}^n$ at time t. This includes the pose and velocity of the robot and all dynamic objects; $\boldsymbol{x}_t = \{\boldsymbol{q}^R, \boldsymbol{q}^1, \ldots, \boldsymbol{q}^D, \dot{\boldsymbol{q}}^R, \dot{\boldsymbol{q}}^1, \ldots, \dot{\boldsymbol{q}}^D\}$.

The control inputs, $\boldsymbol{u}_t \in \mathbb{R}^m$ are velocities applied to the robot's joints: $\boldsymbol{u}_t = \dot{\boldsymbol{q}}^R$ for a fixed control duration Δ_t. Then, the discrete time dynamics of the system is:

$$\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t, \boldsymbol{u}_t) \tag{4.1}$$

where $f : \mathbb{R}^n \times \mathbb{R}^m \longmapsto \mathbb{R}^n$ is the state transition function and I assume an observed initial state of the system, $\boldsymbol{x}_0$.

System dynamics in Eq. 4.1 is modeled with the physics engine Mujoco [92]. Nevertheless, any physics engine is an inaccurate model of the real-world physics and uncertainties over the system dynamics are inevitable. Indeed, even if I assumed perfect modelling, it is difficult for a robot to know the exact geometric, frictional, and inertial properties of objects in an environment. Thus, I represent the real-world dynamics as:

$$\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t, \boldsymbol{u}_t) + w(\boldsymbol{x}_t, \boldsymbol{u}_t) \tag{4.2}$$

where $w(\boldsymbol{x}_t, \boldsymbol{u}_t) : \mathbb{R}^n \times \mathbb{R}^m \longmapsto \mathbb{R}^n$ is an unknown disturbance term due to model inaccuracies.

My goal in this chapter is to build a robust planning and control framework for physics-based manipulation in clutter. The robot must generate control inputs, $\boldsymbol{u}_t$ at time t that drive the system in Eq. 4.2, from an observed initial state $\boldsymbol{x}_0$, to a desired goal state set, under state uncertainty and model inaccuracies.

In the foregoing paragraphs, $\boldsymbol{U} = [\boldsymbol{u}_{t_0}, \boldsymbol{u}_{t_1}, \ldots, \boldsymbol{u}_{t_{N-1}}]$ denotes a sequence of control inputs of length N. Similarly, a sequence of states is $\boldsymbol{X} = [\boldsymbol{x}_{t_0}, \boldsymbol{x}_{t_1}, \ldots, \boldsymbol{x}_{t_N}]$.

4.2 Interleaving Open and Closed-Loop Execution

In this section, I provide my overall framework called *OCL*, for *open and closed-loop execution* during physics-based robotic manipulation. My target is open-loop trajectory execution in the real-world wherever possible. I aim to find robust plans that are guaranteed to be *successful* under state uncertainty and

Algorithm 4: Open and Closed-loop (OCL)

```
Input      : x_0: Initial state
             U: Initial candidate control sequence
1 X*, U*, Ê_e^r ← RobustSTO (x_0, U)
2 RobustSegs,  NonRobustSegs ← GetSegments(Ê_e^r)
3 AllSegs ← RobustSegs ∪ NonRobustSegs
4 for seg in AllSegs do
5 |   if seg in RobustSegs then
6 |   |   Execute U*_seg open-loop
7 |   else
8 |   |   Execute U*_seg with MPC
```

model inaccuracies. However, this is not always possible. Therefore, when the open-loop trajectory is not guaranteed to be successful, I design and use a feedback controller.

OCL is presented in Alg. 4. It is the main algorithm that finds robust and non-robust trajectories and executes them on a real-robot.

It begins with robust planning by minimizing a robust objective function with trajectory optimization (line 1). By robust trajectories, I mean trajectories that will succeed in the real-world regardless of state and model parameter uncertainty. I derive real-world divergence metrics in Sec. 4.3. They quantify robustness to both state uncertainty and model inaccuracies. I use these metrics in the robust objective. Details of the robust planner can be found in Sec.4.4. The robust planner also returns the corresponding robustness metric ($\hat{\boldsymbol{E}}_e^r$) for the planned trajectory.

Robust planning does not always find a trajectory that is guaranteed to be open-loop robust from start to end. However, some parts of this trajectory may be robust. Hence I seek to divide a given trajectory into a combination of robust and non-robust segments with the GetSegments(.) subroutine (line 2). Thereafter, the robust segments are executed open-loop while the non-robust segments are executed with model-predictive control (lines 3-8).

The GetSegments(.) subroutine generates a directed graph to address the problem of dividing the trajectory ($\boldsymbol{X}^*$, $\boldsymbol{U}^*$) into robust and non-robust parts. The nodes of this robustness graph are all the time-points of the trajectory and edges are connections between these nodes, forward in time. An edge is robust or non-robust, indicating that the trajectory segment between those two time-points is robust or non-robust. Details of this segment robustness metric computation can be found in Sec. 4.3.5. To find a complete plan with as many robust segments as possible, I convert the problem into a graph search by assigning costs to each edge. A robust edge going from node i to j costs $\frac{c_{ro}}{j-i}$,

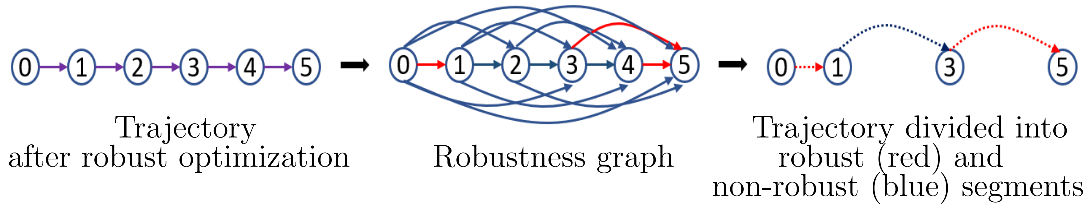


Figure 4.2: Search in a robustness graph to find a combination of robust and non-robust segments that maximize the number of time-points that are robust. In the robustness graph, the robust edges are shown in green, the non-robust edges are shown in blue.

and a non-robust edge costs $c_{nr} \cdot (j - i)$, where $c_{ro} << c_{nr}$, and are constants, and $i < j$. I search this robustness graph for the lowest cost path from the start point 0, to the end point of the trajectory, N. The output is a path consisting of robust and non-robust segments.

Consider the example in Fig. 4.2. On the left, I have a trajectory with 6 time-points, the output of robust planning. I build a robustness graph with robustness metrics for the trajectory (center), where robust segments are in green and non-robust segments are in blue. I search this graph for the lowest cost path from time-point 0 to time-point 5. It produces a robust segment (0-1), a non-robust segment (1-3), and a robust segment (3-5). The robust segments are executed open-loop and the non-robust segment is executed with model-predictive control.

Note that after computing the robustness metric for a trajectory, the cost of an edge in the robustness graph is only an algebraic computation (Sec. 4.3.4). Moreover, some edges can quickly be labelled as non-robust without the algebraic computation. For example, looking at Fig. 4.2, if in addition to segment (3-4), the segment from (4-5) was non-robust, then the segment from (3-5) would also be non-robust.

4.3 Robustness to Uncertainty

I define divergence metrics through contraction analysis to quantify robustness to state uncertainty for a nominal trajectory, under model inaccuracies. I provide a brief introduction to contraction analysis, a derivation of divergence metrics for the real-world case, and a corresponding numerical approximation.

4.3.1 Contraction analysis:

Contraction theory [63] studies the evolution of infinitesimal distance between any two neighboring trajectories and provides quantification on the finite distance between them. Note that Lyapunov Characteristic Exponents [89] similarly provide a measure of the rate of exponential divergence between neighboring trajectories.

I begin with the continuous time version of the autonomous system in Eq. 4.1:

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}(t), \boldsymbol{u}(t)) \tag{4.3}$$

Consider two neighbouring trajectories separated by a virtual displacement (infinitesimal variation) $\delta\boldsymbol{x}$. The squared distance between them is $\delta\boldsymbol{x}^T\delta\boldsymbol{x}$. The rate of change of this distance is given by:

$$\frac{d}{dt}(\delta\boldsymbol{x}^T\delta\boldsymbol{x}) = 2\delta\boldsymbol{x}^T\delta\dot{\boldsymbol{x}} \tag{4.4}$$

From Eq. 4.3, $\delta\dot{\boldsymbol{x}} = \frac{\partial f}{\partial \boldsymbol{x}}\delta\boldsymbol{x}$. Therefore Eq. 4.4 becomes:

$$\frac{d}{dt}(\delta\boldsymbol{x}^T\delta\boldsymbol{x}) = 2\delta\boldsymbol{x}^T\frac{\partial f}{\partial \boldsymbol{x}}\delta\boldsymbol{x} \tag{4.5}$$

The Jacobian of f, $\frac{\partial f}{\partial \boldsymbol{x}}$ can be written as a sum of symmetric and skew-symmetric parts:

$$\frac{\partial f}{\partial \boldsymbol{x}} = \frac{1}{2}(\frac{\partial f}{\partial \boldsymbol{x}} + \frac{\partial f^T}{\partial \boldsymbol{x}}) + \frac{1}{2}(\frac{\partial f}{\partial \boldsymbol{x}} - \frac{\partial f^T}{\partial \boldsymbol{x}}) \tag{4.6}$$

Let $\lambda^f_{max}(\boldsymbol{x}, \boldsymbol{u})$ be the largest eigenvalue of the symmetric part (first term of Eq. 4.6) of the Jacobian $\frac{\partial f}{\partial \boldsymbol{x}}$. Then, given that the eigenvalue of the skew-symmetric part is 0 or imaginary, it follows that:

$$\frac{d}{dt}(\delta\boldsymbol{x}^T\delta\boldsymbol{x}) \leqslant 2\lambda^f_{max}\delta\boldsymbol{x}^T\delta\boldsymbol{x} \tag{4.7}$$

If I define $D^f_m := \lambda^f_{max}$ as the maximal divergence metric, I find:

$$||\delta\boldsymbol{x}(t)|| \leqslant ||\delta\boldsymbol{x}(t_0)||e^{\int_{t_0}^{t} D^f_m(\boldsymbol{x},\boldsymbol{u},t)dt} \tag{4.8}$$

From Eq. 4.8, I find that if D^f_m is uniformly negative definite, any infinitesimal distance $||\delta_x(t)||$ converges exponentially to zero. Moreover, any finite path converges exponentially to zero (by path integration).

I reach an important result of contraction analysis: Given a nominal trajectory $\bar{\boldsymbol{x}}(t)$, a solution to Eq. 4.3 with nominal controls $\bar{\boldsymbol{u}}(t)$, any other trajectory

that starts in a region centered on $\bar{\boldsymbol{x}}(t)$ where D_m^f is uniformly negative definite everywhere, converges exponentially to the nominal $\bar{\boldsymbol{x}}(t)$ ([63], Theorem 1).

The system in Eq. 4.3 models the real-world but can be inaccurate. Thus, I ask the following questions: If the nominal controls $\bar{\boldsymbol{u}}(t)$ are executed in the real-world, will the real trajectory $\bar{\boldsymbol{x}}^r(t)$ be different? More importantly, will the convergence properties for the real trajectory change?

4.3.2 Contraction analysis for the real-world

Extending beyond contraction analysis used in prior work [47, 55], I consider the continuous-time version of Eq. 4.2, a representation of the real-world dynamics:

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}(t), \boldsymbol{u}(t)) + w(\boldsymbol{x}(t), \boldsymbol{u}(t)) \tag{4.9}$$

Indeed, the resulting real nominal trajectory $\bar{\boldsymbol{x}}^r(t)$ may be different due to the disturbance w. More importantly, I show that the resulting convergence properties may change.

The real maximal divergence metric D_m^r for the system in Eq. 4.9 can be written as:

$$D_m^r := D_m^{f+w}(\boldsymbol{x}, \boldsymbol{u}) \tag{4.10}$$

where $D_m^{f+w} := \lambda_{max}^{f+w}(\boldsymbol{x}, \boldsymbol{u})$ is the largest eigenvalue of the symmetric part of the Jacobian $\frac{\partial(f+w)}{\partial \boldsymbol{x}}$. We reach this conclusion directly by applying steps in Eq.[4.4-4.8] to the system in Eq. 4.9.

Another important divergence metric we consider in this work is the expected divergence metric. It was first introduced by Johnson, King, and Srinivasa [47]. It quantifies the evolution of the expected value of a virtual displacement. From Eq. 4.8, the expected divergence metric D_e^f for the system in Eq. 4.3 can be written as:

$$E[||\delta\boldsymbol{x}(t)||] = E[||\delta\boldsymbol{x}(t_0)||]e^{\int_{t_0}^{t} D_e^f(\boldsymbol{x}, \boldsymbol{u}, \tau)d\tau} \tag{4.11}$$

$$D_e^f := \frac{d}{dt} \ln E[||\delta\boldsymbol{x}(t)||] \tag{4.12}$$

For the real-world case, similar to the real maximal divergence metric, one can write the real expected divergence metric for the system in Eq. 4.9 as:

$$D_e^r := D_e^{f+w} \tag{4.13}$$

Consider now a state trajectory from an initial time t_0 to a final time t_N, and define divergence path metrics such that:

$$E_h^f = e^{\int_{t_0}^{t_N} D_h^f dt},\ h \in \{m, e\} \tag{4.14}$$

The nominal trajectory starting from t_0 and ending at t_N is convergent if $E_h^f < 1$. Recall that I am also interested in the real nominal trajectory $\bar{\boldsymbol{x}}^r$. Thus, one can write:

$$E_h^r = e^{\int_{t_0}^{t_N} D_h^{f+w} dt},\ h \in \{m, e\} \tag{4.15}$$

and the real nominal trajectory is convergent if $E_h^r < 1$, depending on the metric of choice - expected or maximal. This means that any other trajectory that starts in a region centered on the real nominal trajectory $\bar{\boldsymbol{x}}^r$ will converge exponentially to the real nominal trajectory.

4.3.3 Metric approximations

Johnson, King, and Srinivasa [47] approximate divergence metrics D_m^f and D_e^f by approximating the virtual displacement $\delta\boldsymbol{x}$ with finite samples. Let $\delta\boldsymbol{x}(t) = \boldsymbol{x}^i(t) - \bar{\boldsymbol{x}}(t)$, where $\boldsymbol{x}^i(t)$ is a solution for sample i to the same system (Eq. 4.3) for nominal controls $\bar{\boldsymbol{u}}(t)$, starting with a different initial condition: $\delta\boldsymbol{x}(t_0) = \boldsymbol{x}^i(t_0) - \bar{\boldsymbol{x}}(t_0)$. Then, from Eq. 4.8 I can write an approximation $\hat{D}_m^f$ to the maximal divergence metric for a small time step δt as:

$$\hat{D}_m^f := \frac{1}{\delta t} \ln \max_i \frac{||\boldsymbol{x}^i(t+\delta t) - \bar{\boldsymbol{x}}(t+\delta t)||}{||\boldsymbol{x}^i(t) - t\bar{\boldsymbol{x}}(t)||} \tag{4.16}$$

where $i = 1, \ldots, N_c$, and N_c is the number of finite samples.

Similarly, from Eq. 4.11, I can write an approximation $\hat{D}_e^f$ to the expected divergence metric as:

$$\hat{D}_e^f := \frac{1}{\delta t} \ln \frac{\frac{1}{N_c}\sum_{i=0}^{N_c} ||\boldsymbol{x}^i(t+\delta t) - \bar{\boldsymbol{x}}(t+\delta t)||}{\frac{1}{N_c}\sum_{i=0}^{N_c} ||\boldsymbol{x}^i(t) - \bar{\boldsymbol{x}}(t)||} \tag{4.17}$$

From Eq. 4.16 and Eq. 4.17, and considering the definition in Eq. 4.14, I define approximations $\hat{E}_e^f$ to the expected divergence path metric and $\hat{E}_m^f$ to the maximal divergence path metric valid for a finite time from t_0 to t_N:

$$\hat{E}_e^f := \frac{\frac{1}{N_c}\sum_{i=0}^{N_c} ||\boldsymbol{x}^i(t_N) - \bar{\boldsymbol{x}}(t_N)||}{\frac{1}{N_c}\sum_{i=0}^{N_c} ||\boldsymbol{x}^i(t_0) - \bar{\boldsymbol{x}}(t_0)||} \tag{4.18}$$

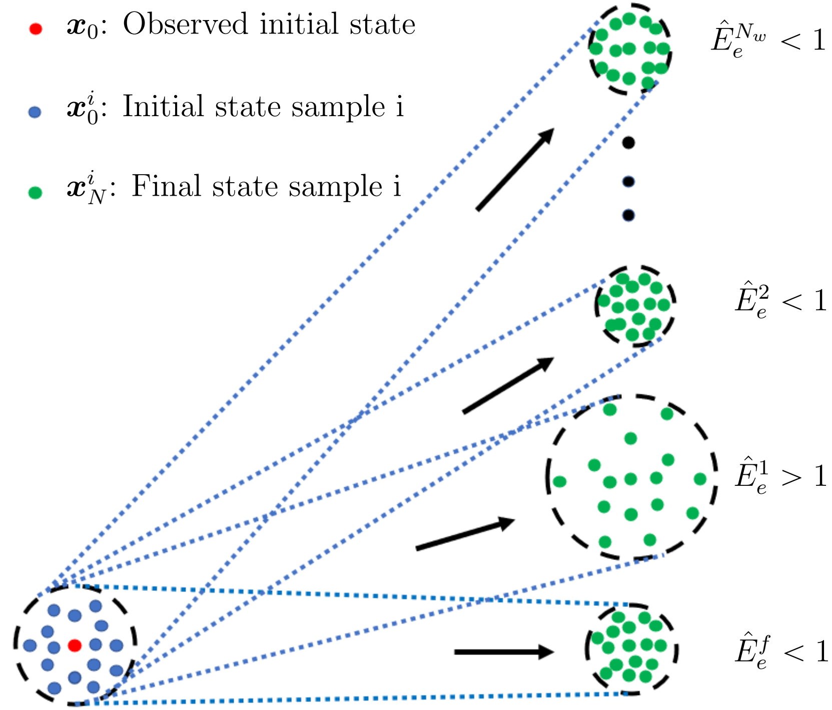


Figure 4.3: Computing the real-world expected divergence metric $\hat{E}_e^r$. First, I compute the divergence metric using the nominal model f ($\hat{E}_e^f$). I draw N_c sample initial states from the observed initial state $\boldsymbol{x}_0$ and roll-out a trajectory from each state, to obtain final state samples $\boldsymbol{x}_N^i$. Since the size of the final state distribution is less than that of the initial state distribution, $\hat{E}_e^f < 1$, and the trajectory is convergent in f. Thereafter, we randomly create N_w real-world realizations from f and compute the divergence metric in each of these N_w worlds. These metrics can be very different from each other across the real-world realizations. We pick the metric with the maximum value as a worst-case approximation of the real-world divergence metric.

$$\hat{E}_m^f := \max_i \frac{||\boldsymbol{x}^i(t_N) - \bar{\boldsymbol{x}}(t_N)||}{||\boldsymbol{x}^i(t_0) - \bar{\boldsymbol{x}}(t_0)||}, \; i = 0, 1, \ldots, N_c \tag{4.19}$$

If $\hat{E}_h^f < 1$, $h \in \{m, e\}$, the nominal trajectory $\bar{\boldsymbol{x}}(t)$ from t_0 to t_N is convergent, depending on one's choice of robustness metric - maximal (m) or expected (e). This means that any other trajectory that starts in a region centered on the nominal trajectory $\bar{\boldsymbol{x}}^f$ will converge exponentially to it.

4.3.4 Metric approximations for the real-world

I define approximations to the real-world maximal and expected path divergence metrics:

$$\hat{E}_h^r = \hat{E}_h^{f+w}, \; h \in \{m, e\} \tag{4.20}$$

The real-world nominal trajectory is convergent if $\hat{E}_h^r < 1$.

Given that the disturbance term $w(\boldsymbol{x}, \boldsymbol{u})$ is unknown, an important question is how to compute $\hat{E}_h^r$.

Consider a set of possible deterministic real-worlds:

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}(t), \boldsymbol{u}(t)) + w_j(\boldsymbol{x}(t), \boldsymbol{u}(t)), \;\; j = 0, \ldots, N_w \tag{4.21}$$

where $N_w > 0$ is a finite number.

I assume that model inaccuracies resulting in w_j arise only from inaccurate physics parameters. Specifically, I assume these parameters are the object's mass m, coefficient of friction μ, and size $\boldsymbol{p}$. These physics parameters are bounded such that:

$$m_l \leqslant m \leqslant m_u, \; \mu_l \leqslant \mu \leqslant \mu_u, \; \boldsymbol{p}_l \leqslant \boldsymbol{p} \leqslant \boldsymbol{p}_u \tag{4.22}$$

where m_l, m_u, μ_l, μ_u, are constants and $\boldsymbol{p}_l, \boldsymbol{p}_u$, are constant vectors. The size here is a vector of values describing the geometry of an object. For example, radius and height for a cylinder or length, breadth, and height for a box.

Then, a sample real-world realization corresponds to running a physics engine that models f with a uniformly sampled set of physics parameters. I consider a worst-case scenario and define $\hat{E}_h^r$ as the maximum divergence metric over all N_w real-world realizations:

$$\hat{E}_h^r := \max_j(\hat{E}_h^j), \; j = 0, \ldots, N_w, \; h \in \{m, e\} \tag{4.23}$$

I provide an illustration in Fig. 4.3, for the expected real-world divergence metric.

4.3.5 Segment metric computation

Consider the discrete time points of a nominal trajectory $\bar{\boldsymbol{x}}_t$, where $t = t_0, t_1, \ldots, t_N$. Then, the expected path metric for a one-step trajectory segment from t_p to t_{p+1} is:

$$\hat{E}_{e,t_p} = \frac{\frac{1}{N_c}\sum_{i=0}^{N_c} ||\boldsymbol{x}_{t_{p+1}}^i - \bar{\boldsymbol{x}}_{t_{p+1}}||}{\frac{1}{N_c}\sum_{i=0}^{N_c} ||\boldsymbol{x}_{t_p}^i - \bar{\boldsymbol{x}}_{t_p}||} \tag{4.24}$$

Thus, I can write the divergence metric vector as:

$$\hat{\boldsymbol{E}}_e = [\hat{E}_{e,t_0}, \hat{E}_{e,t_1}, \ldots, \hat{E}_{e,t_N}] \tag{4.25}$$

It is a list of one-step trajectory segment divergence metrics. Then, it follows from Eq. 4.24 that the divergence metric for any trajectory segment from time t_p to time t_q can be written as the following algebraic computation over $\hat{\boldsymbol{E}}_e$:

$$\hat{E}_{e,t_p,t_q} := \prod_{t=t_p}^{t=t_q} \hat{\boldsymbol{E}}_e[t] \tag{4.26}$$

Hence, a trajectory segment starting at t_p and ending at t_q is robust if $\hat{E}_{e,t_p,t_q} < 1$. Also, the divergence metric for the full trajectory can be written as:

$$\hat{E}_e = \prod_{t=0}^{t=t_N} \hat{\boldsymbol{E}}_e[t] \tag{4.27}$$

Recall that the GetSegments(.) subroutine in Alg. 4 returns robust and non-robust segments after search on a robustness graph. The robustness metric for each edge is given by the algebraic computation in Eq. 4.27.

4.4 Robust Planning and Control

I generate robust plans through trajectory optimization of a robust objective function J_R.

$$J_R(\boldsymbol{X}, \boldsymbol{U}) = v_e \hat{E}_e^{r2} + v_m \hat{E}_m^{r\,2} + v_j J \tag{4.28}$$

where v_e, v_m, and v_j are positive constant weights.

To achieve robustness to state uncertainty under real-world model inaccuracies, I minimize $\hat{E}_e^r$ and $\hat{E}_m^r$, the real expected and maximal divergence metrics respectively, for a given initial state and control sequence. They are computed from Eq. 4.23.

J is a deterministic objective for the task of reaching in clutter. Formally, the objective is a sum of running costs L_t and terminal cost L_{t_N} along a trajectory:

$$J(\boldsymbol{X}, \boldsymbol{U}) = v_f L_{t_N}(\boldsymbol{x}_{t_N}) + \sum_{t=t_0}^{t_{N-1}} L_t \tag{4.29}$$

where $v_f > 0$ is a constant weight. Details of L_t and L_{t_N} are given in Sec. 4.4.2.

I directly minimize the roboust objective J_R such that:

$$\boldsymbol{U}^* = \arg\min_{\boldsymbol{U}} J_R(\boldsymbol{X}, \boldsymbol{U}) \tag{4.30}$$

subject to the dynamics constraint, $\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t, \boldsymbol{u}_t)$, the terminal set constraint, $\boldsymbol{x}_{t_N} \in \mathbb{X}_f$, and the control sequence constraint, $\boldsymbol{U} \in \mathbb{U}$.

The set of feasible terminal states, $\mathbb{X}_f$ is an $\alpha \geqslant 0$ sub-level set of the terminal cost function. Specifically:

$$\mathbb{X}_f := \{\boldsymbol{x} \mid L_{t_N} \leqslant \alpha\} \tag{4.31}$$

and the set of control-limited sequences is:

$$\mathbb{U} := \{\boldsymbol{U} \mid \boldsymbol{b}_l \leqslant \boldsymbol{u}_t \leqslant \boldsymbol{b}_u\} \tag{4.32}$$

where $\boldsymbol{b}_l$ and $\boldsymbol{b}_u$ are constant vectors of lower and upper bound on the controls.

A trajectory is feasible (in the deterministic sense) if it satisfies the terminal state constraint, the control limits, and yields a total cost J less than a threshold $\beta > 0$. The cost threshold depends on the task. For reaching in clutter, it is defined such that no failures occur. Specifically, β is selected such that no objects are toppled/dropped from the working surface, and there are no robot collisions with static obstacles, at the end of plan execution.

4.4.1 Robust sampling-based trajectory optimization

Trajectory optimization methods such as STOMP [49] have shown impressive speed for motion planning with parallel rollouts on multiple cores of a PC. They also easily accept arbitrary cost functions that may not be differentiable.

In Alg. 5, I propose a robust sampling-based trajectory optimization algorithm (RobustSTO) for physics-based manipulation. It begins with an initial candidate control sequence $\boldsymbol{U}$ and seeks lower cost trajectories (lines 4-15) iteratively until a feasible and robust control sequence is found or the maximum number of iterations is reached (line 4). I add random control sequence variations $\boldsymbol{\delta U}^s$ on the candidate control sequence to generate S new control sequences at each iteration (line 7). Thereafter on line 8, I roll-out each sample control sequence and return the corresponding state sequence $\boldsymbol{X}$ and cost J. A roll-out starts from the initial state $\boldsymbol{x}_0$ and applies each control input in $\boldsymbol{U}$ sequentially, one after the other.

On line 9, I compute the maximal and expected divergence metric vectors along a given sample trajectory with the ComputeMetrics() subroutine. It involves N_c trajectory roll-outs for each of the N_w real-world realizations and also for the nominal model f. Then, I compute the robust cost of a sample trajectory using Eq. 4.28, on line 10.

Algorithm 5: Robust Stochastic Traj. Opt. (RobustSTO)

Input : $\boldsymbol{x}_0$: Initial state
$\boldsymbol{U}$: Initial candidate controls
Output : $\boldsymbol{U}^*$: Feasible and robust control sequence
$\boldsymbol{X}^*$: Feasible state sequence
$\hat{\boldsymbol{E}}_e^r$: Real-world expected divergence metric
Parameters : S: Number of noisy trajectory roll-outs
ν: Sampling variance vector
I_{max}: Maximum number of iterations

1. $\boldsymbol{X}, J \leftarrow$TrajRollout$(\boldsymbol{x}_0, \boldsymbol{U})$, $\quad I \leftarrow 0$
2. $\hat{\boldsymbol{E}}_e^r$, $\hat{\boldsymbol{E}}_m^r$ $\leftarrow$ComputeMetrics($\boldsymbol{X}$,$\boldsymbol{U}$)
3. $J_R \leftarrow$ Compute robust cost using Eq. 4.28
4. **while** $I \leqslant I_{max}$ ***and*** *(U not feasible or $\hat{E}_e^r > 1$)* **do**
5.   **for** $s \leftarrow 0$ **to** $S-1$ **do**
6. $\boldsymbol{\delta U}^s \leftarrow N(\boldsymbol{0}, \nu)$
7. $\boldsymbol{U}^s = \boldsymbol{U} + \boldsymbol{\delta U}^s$
8. $\boldsymbol{X}^s, J^s \leftarrow$TrajRollout$(\boldsymbol{x}_0, \boldsymbol{U}^s)$
9. $\hat{\boldsymbol{E}}_e^{r_s}$, $\hat{\boldsymbol{E}}_m^{r_s}$ $\leftarrow$ComputeMetrics($\boldsymbol{X}^s$, $\boldsymbol{U}^s$)
10. $J_R^s \leftarrow$ Compute robust cost using Eq. 4.28
11. $s^* \leftarrow$MinCostSample(J$_R^0$, J_R^1, ..., J_R^{S-1})
12.   **if** $J_R^{s*} < J_R$ **then**
13. $\boldsymbol{U} \leftarrow \boldsymbol{U}^{s*}$, $\boldsymbol{X} \leftarrow \boldsymbol{X}^{s*}$, $J_R \leftarrow J_R^{s*}$, $\hat{\boldsymbol{E}}_e^r \leftarrow \hat{\boldsymbol{E}}_e^{r_s*}$
14. $I \leftarrow I + 1$
15. **return** $\boldsymbol{U}$, $\boldsymbol{X}$, $\hat{\boldsymbol{E}}_e^r$

I take a greedy approach to trajectory updates. The minimum cost trajectory is selected as the update (line 11). However, the update is accepted only if it provides a lower cost (lines 12-13). Finally, the algorithm returns feasible and robust controls (U), the corresponding nominal state sequence, $\boldsymbol{X}$, and the corresponding real-world expected divergence metric (line 15).

4.4.2 Cost functions

For physics-based manipulation in clutter, I provide details of the objective. Let's begin with the running cost:

$$L_t(\boldsymbol{x}_t, \boldsymbol{x}_{t+1}, \boldsymbol{u}_{t-1}, \boldsymbol{u}_t) = \sum_i v_i c_i, \ i \in \{a, c, d, y\} \tag{4.33}$$

where $v_i \geqslant 0$ is a constant weight, the cost terms penalize robot acceleration (c_a), scene disturbance (c_d), collision (c_c), and toppling (c_y).

4.4.2.1 Acceleration cost (c_a):

I am interested in robot motion with minimal changes in robot velocity in between timesteps. I encode this desired behaviour in the robot acceleration cost:

$$c_a(\boldsymbol{u}_{t-1}, \boldsymbol{u}_t) = ||\boldsymbol{u}_t - \boldsymbol{u}_{t-1}||^2 \tag{4.34}$$

4.4.2.2 Disturbance cost (c_d):

The disturbance cost penalizes displacing each of the D dynamic objects from their initial positions and velocities:

$$c_d(\boldsymbol{x}_t, \boldsymbol{x}_{t+1}) = \sum_{i=1}^{D} ||\boldsymbol{x}_{t+1} - \boldsymbol{x}_t||^2 \tag{4.35}$$

4.4.2.3 Collision cost (c_c):

Here, I penalize collisions between the robot and all static objects in the environment through a discontinuous cost:

$$c_c(\boldsymbol{x}_{t+1}) = \begin{cases} 1 & \text{if robot collides with static objects} \\ 0 & \text{otherwise} \end{cases} \tag{4.36}$$

Here, I consider collisions between all robot links (including the end-effector) with static objects in the environment.

4.4.2.4 Toppling cost (c_y):

I penalize toppling of objects during contact interaction through a discontinuous cost term:

$$c_y(\boldsymbol{x}_{t+1}) = \sum_{i=1}^{D} C^i_{Topple} \tag{4.37}$$

$$C^i_{Topple} = \begin{cases} 1 & \text{if object i is toppled} \\ 0 & \text{otherwise} \end{cases} \tag{4.38}$$

In addition to the running cost, I also define the goal cost term for reaching in clutter as:

$$c_g(\boldsymbol{x}_N) = ||\boldsymbol{r}_{GO}||^2 + w_\phi \phi_N^2 \tag{4.39}$$

$$\phi_N = \arccos(\hat{\boldsymbol{v}}_G \cdot \hat{\boldsymbol{r}}_{GO}) \tag{4.40}$$

where $\boldsymbol{r}_{GO}$ is the position vector from a point G inside the gripper to a point O, the object's center of mass, $\hat{\boldsymbol{v}}_G$ is the gripper's forward unit direction, $w_\phi > 0$ is

a constant weight, and ϕ_N is the angular distance between the gripper's forward unit direction and the unit vector $\hat{\boldsymbol{r}}_{GO}$.

In some cases, a desired terminal state $\boldsymbol{x}^d$ can be provided to the optimizer. Then, I write the overall final cost function as:

$$L_{t_N}(\boldsymbol{x}) = \begin{cases} ||\boldsymbol{x}^d - \boldsymbol{x}|| & \text{if } \boldsymbol{x}^d \text{ is given} \\ c_g(\boldsymbol{x}) & \text{otherwise} \end{cases} \tag{4.41}$$

A sampling-based, derivative-free approach is well suited to handle these discontinuous cost functions.

4.4.3 Time complexity of robust planner

Physics simulation is the most computationally expensive operation for the RobustSTO algorithm. This is done both in the TrajRollout(.) and in the ComputeMetrics(.) subroutines.

Let T_u be the physics simulation time for a single control input $\boldsymbol{u}_t$ applied for control duration Δ_t. Then, the serial optimization time, T^s is a sum of $T_u N$(line 1), $T_u N N_c(N_w + 1)$ (line 2), $T_u NSI$ (line 8), and $T_u N N_c(N_w + 1)SI$ (line 9).

Thus, the serial computation time is:

$$T^s = T_u N(SI + 1)(N_c(N_w + 1) + 1) \tag{4.42}$$

Rollouts either in TrajRollout(.) or in ComputeMetrics(.) can be computed simultaneously in parallel. Thus, a parallel implementation gives an optimization time T^p, independent of S, N_w, and N_c:

$$T^p = T_u N(I + 1) \tag{4.43}$$

RobustSTO is a polynomial time algorithm with complexity $\mathcal{O}(N(I + 1))$. This means that the algorithm is fast, provided sufficient parallel cores are available (at least $N_c \times N_W \times S$).

4.4.4 Model-Predictive Control

I use a model-predictive controller (MPC) as the closed-loop controller in this work. It calls the trajectory optimizer online, during execution, to solve a finite

horizon optimal control problem. It executes the first action in the control sequence, updates the internal state with camera feedback, and then runs the trajectory optimizer again to generate a new control sequence, and repeats this process at every step. I use only the deterministic objective J during MPC, i.e, a version of Alg. 5 without robustness computations.

4.5 Robot Experiments and Results

Through my experiments, I ask two important questions:

- How does the open and closed-loop execution planning and control framework compare with only open-loop and only closed-loop approaches?
- How does the normal divergence metric proposed in prior work compare with the real-world divergence metric proposed in this chapter, for manipulation in the real-world?

I answer these questions through real-world robot experiments.

4.5.1 Setup

The robot is a 6 degrees of freedom (DOF) UR5 arm with a 1-DOF Robotiq 2-finger gripper. I use Mujoco[92] as the physics simulator to plan all robot actions. The planning environment consists of a shelf and 10 different objects, including some from the YCB dataset [19]. Object pose is captured with the OptiTrack motion capture system. In Table. 4.1, I detail all parameters used throughout the experiments.

4.5.2 Baselines

In this work, I compare against four baselines:

4.5.2.1 Open-loop execution (OL)

This is an approach where I minimize the deterministic objective J with trajectory optimization. It produces a sequence of controls that are then executed open-loop in the real-world.

Table 4.1: Experimental parameters

	Parameter value
Robustness graph	
$[C_{ro}, C_{nr}]$	[1, 1000]
Divergence metrics	
$[N_c, N_w]$	[4, 4]
$[m_l, m_u]$	$[0.5, 0.8]kg$
$[\mu_l, \mu_u]$	$[0.2, 0.4]$
Robust planning	
$[\alpha, \beta]$	[10, 50]
$[S, N, \Delta_t]$	[4, 5, 0.2s]
$[b_l, b_u]$	$\pm\ \pi\ rad/s$
I_{max}	10
$[v_e, v_m, v_j]$	[2, 0.5, 1]
$[v_a, v_c, v_d, v_y, v_\phi, v_f]$	[0.001, 200, 1000, 200, 0.019]
Environment	
D	10
m	7

4.5.2.2 Robust open-loop execution (ROL)

This baseline minimizes the robust objective with trajectory optimization. The trajectory is then executed open-loop in the real-world.

4.5.2.3 Convergent planning (CP)

This baseline uses divergence metrics in Johnson, King, and Srinivasa [47] to generate robust plans through trajectory optimization, and executes them open-loop.

4.5.2.4 Closed-loop control (CC)

The closed-loop control approach performs trajectory optimization at every step, during execution, using the deterministic objective J and then executes only the first planned control. It re-plans online until task completion.

4.5.3 Comparison of OCL with baselines

I randomly generate 20 real-world scenes by placing the target object behind other objects, such that reaching directly for it is almost impossible. I also pick different objects to surround the target in each scene. I fix the robot's initial configuration in all scenes to allow for easy scene reset. I run each baseline and *OCL* to complete the manipulation task. That's a total of 100 robot

manipulation runs - 5 methods per scene. I recorded success, planning time, execution time, divergence metrics, and finally what percentage of trajectory segments are executed open-loop vs. closed-loop in a given scene (for OCL).

Results are shown in Fig. 4.4.

4.5.4 Execution success

I find that the proposed method, OCL, is *more successful* in completing the reaching in clutter tasks, compared to all the other baselines. As shown in Fig. 4.4a, open-loop execution (OL) has the least success rate - a difference of about 35% compared to the *interleaved open and closed-loop* approach proposed in this work. An interesting observation is that the interleaved open and closed-loop approach outperforms closed-loop control. One explanation for this is that the starting trajectory for OCL is more robust compared to that of CC.

4.5.5 Planning time

The planning time is the total time spent by planners before the robot executes its first action.

I show the planning time in Fig. 4.4b.

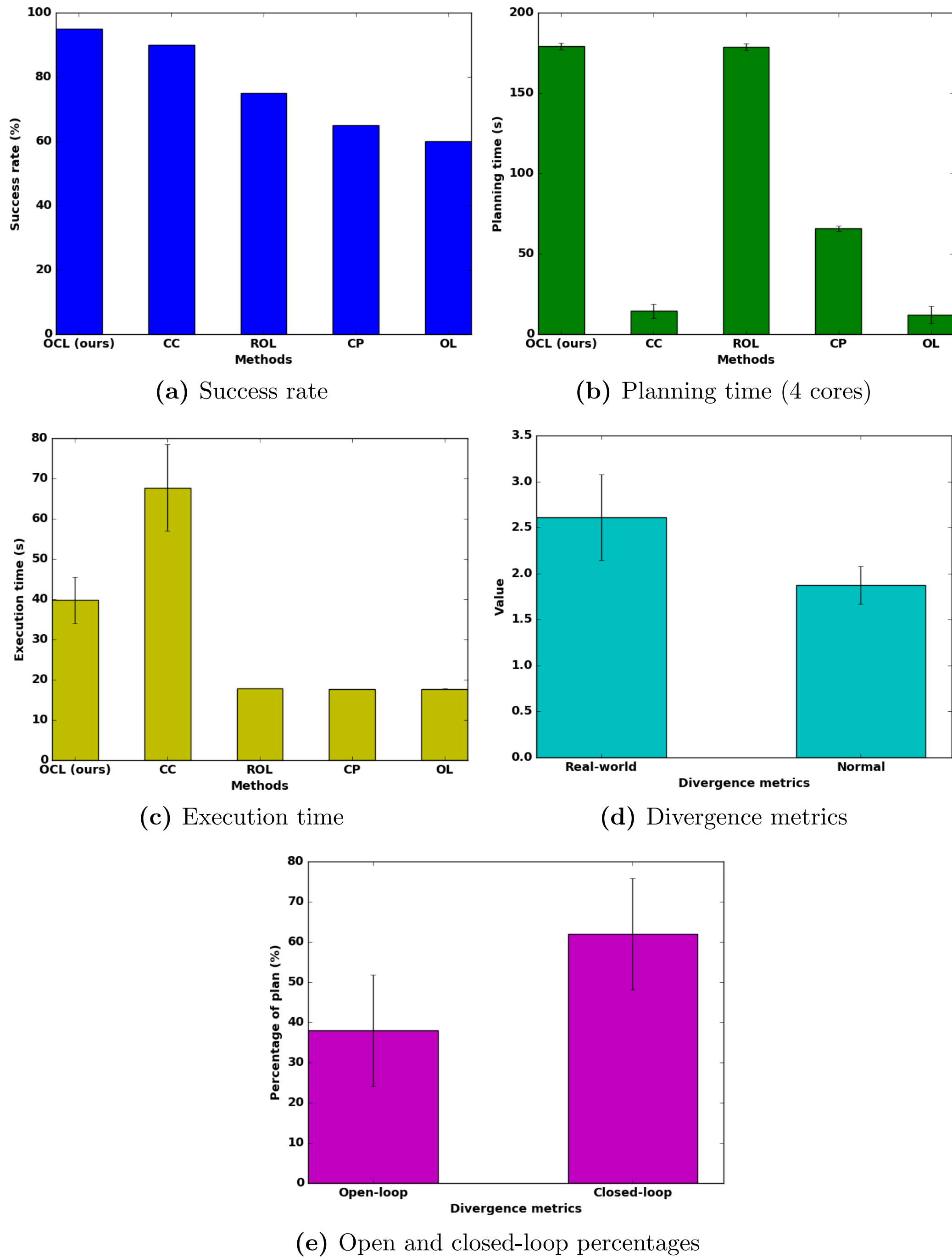
On my PC with 4 cores, we see that the robust planning time is higher than that of planning without robustness metrics (OL and CC). However, on a server with 64 cores ($S \cdot \mathrm{N}_c \cdot \mathrm{N}_w$), both standard planning (OL and CC) and robust planning would have *similar* planning times. This is thanks to potential parallelization of trajectory rollouts with time complexity detailed in Sec. 4.4.3.

4.5.6 Execution time

The execution time is the total time spent after the robot starts executing it's first action.

As shown in Fig. 4.4c, the execution time of OCL is close to that of traditional open-loop execution (i.e ROL, OL, and CP) but much lower than full closed-loop control.

This difference would be *much higher* for scenes with less number of objects, since completely open-loop plans could be found. However, even in such simple scenes, a fully closed-loop approach would stop at each step to re-plan, taking much more time to complete the manipulation task.



(a) Success rate

(b) Planning time (4 cores)

(c) Execution time

(d) Divergence metrics

(e) Open and closed-loop percentages

Figure 4.4: Experimental results comparing the interleaved open and closed-loop execution method (OCL) with other baselines. I randomly created 20 real-world scenes and ran all five methods on each scene. I recorded success, planning time, execution time, divergence metrics, and finally what percentage of trajectory segments are executed open-loop vs. closed-loop in a given scene. All error bars indicate a 95% confidence interval of the mean.

4.5.7 Real or normal divergence metrics

Normal divergence metrics proposed in prior work do not consider model inaccuracies. How does the real divergence metric compare with the normal divergence metric, for robust planning, and execution in the real-world? Based on data from the baseline experiments, I attempt to answer this question.

I found that the normal divergence metrics are much lower than the real divergence metrics, as shown in Fig. 4.4d. Implying that CP should be more successful than ROL. However, this is not the case. I've seen previously in Fig. 4.4a that ROL is more successful than CP by about 10%. Thus, the real-world divergence metric is a better estimate of the uncertainty for the reaching in clutter task.

4.5.8 Composition of OCL plans

Using the interleaved open and closed-loop approach, what percentage of a trajectory is executed open-loop versus closed-loop?

I recorded the percentages during OCL experiments. Results are shown in Fig. 4.4e. I found that about 60% of a plan was executed closed-loop, while about 40% was execute open-loop, on average.

Note that these percentages will change depending on task difficulty, especially the number of contact interactions. For example, one might see a higher percentage of open-loop execution for a scene with only a few number of objects.

In this way, the interleaved open and closed-loop approach (OCL) has the potential to adapt to varying task difficulty, executing fully open-loop or fully closed-loop or anything inbetween.

4.5.9 Examples of robust trajectory segments

What sort of robust trajectory segments were encountered during experiments?

One major robustness strategy is the funnelling action, where the robot pushes an object towards the shelf wall and grasps it. This is shown in Fig. 4.5. On the left image, I show part of the robustness metric computation where initial state uncertainty is reduced, through a robust robot motion. On the right, I see the real-world execution during one of my experiments.

Another robustness strategy is a stable side push on an object as shown in Fig. 4.6. The box is pushed towards the cylinder in a robust way, where the initial state uncertainties are reduced.

One final robustness strategy for the robot is to avoid any contact and move completely in free space. I've seen this strategy in Fig. 4.1.

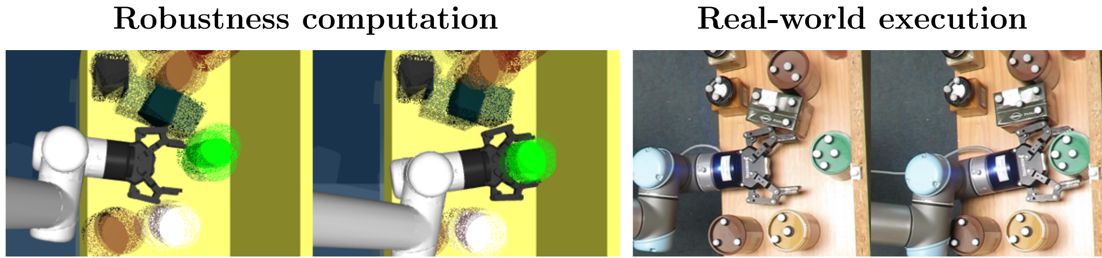


Figure 4.5: A sample robust segment from the interleaved open and closed-loop approach. The left image shows a part of the robustness metric computation. I see that the initial state uncertainty is reduced through a funnelling action that pushes the target object towards the shelf, and grasps it. On the right image I see the robust segment executed open-loop in the real world.

Figure 4.6: A sample robust segment from the interleaved open and closed-loop approach. The left image shows a part of the robustness metric computation. I see that the initial state uncertainty is reduced through a stable side push. On the right image I see the robust segment executed open-loop in the real world.

4.5.10 Sample plans from the different planners

In Fig. 4.7, I show several plans from all baselines tested in the real-world. I show these in four different scenes.

In the first row, I see an open-loop execution failure. The black box didn't move out of the way as planned. It ended up in the gripper at the final state. This is a failure.

In the second row I see a trajectory planned through the normal divergence metrics. The motion of the cylinder is different from originally planned in simulation. Thus, the cylinder ended up in the robot's gripper - a failure case.

In the third row, I show a successful run from the robust open-loop execution method where I use the real-world divergence metrics proposed in this work. The robot pushes the brown cylinder in a robust manner onto other supporting objects. It successfully reaches for the green target object.

Finally, in the last row, I show a closed-loop control run, where the robot re-plans at every step to successfully reach for the green target object.

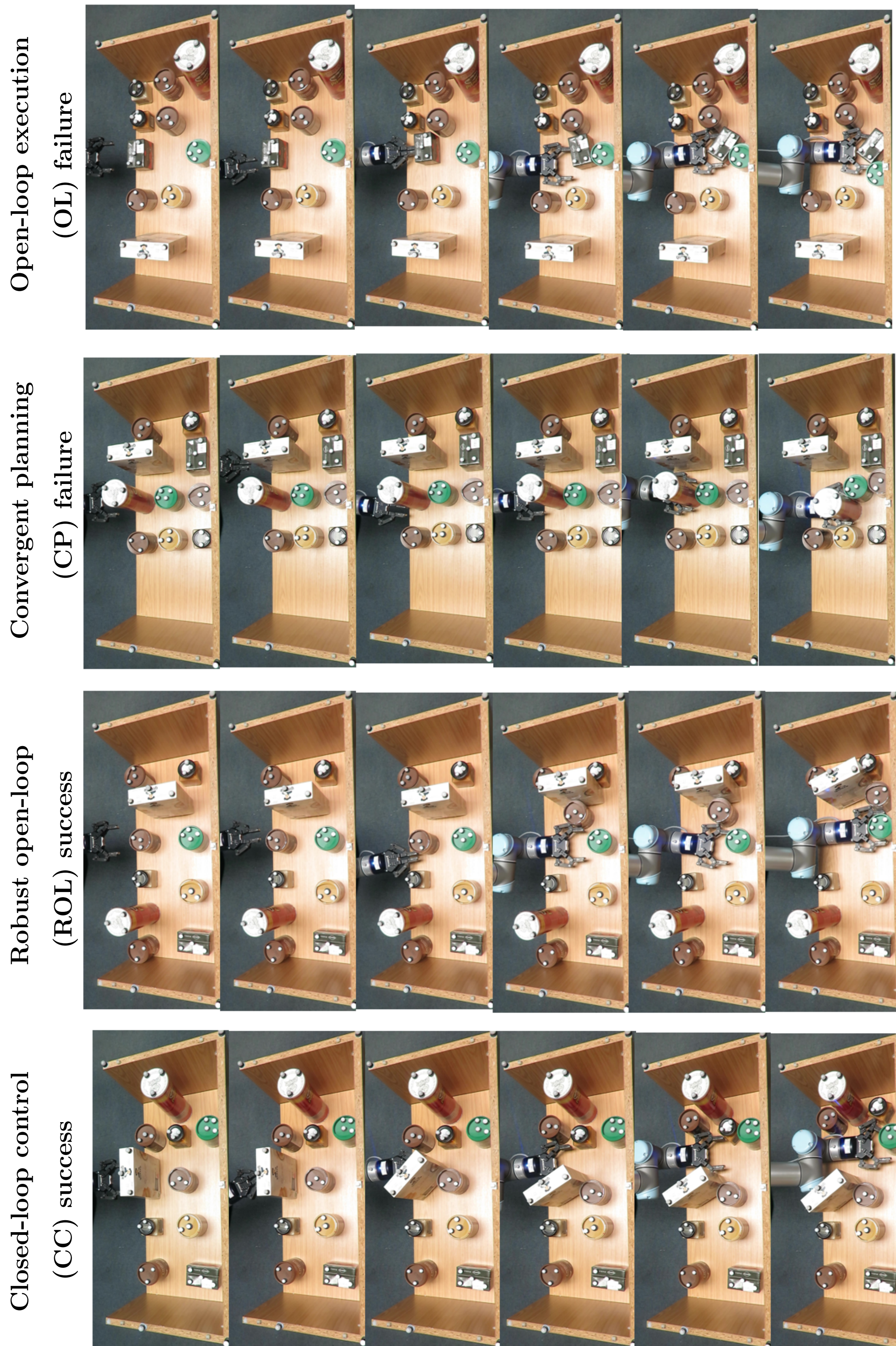


Figure 4.7: Examples of successful and failed manipulation plans from different planning and execution methods.

4.5.11 Summary of results

In summary, *OCL* achieves a good trade-off between success rates and execution time. It spends more time for closed-loop control where needed and resorts to open-loop execution wherever possible. It realizes a more successful and fluent execution compared to baselines.

4.6 Discussion

In this chapter, I present for the first time, an interleaved open and closed-loop control framework for physics-based manipulation under uncertainty. I derived robustness metrics through contraction theory, and used these metrics to plan robust robot motions. I separated a trajectory into robust and non-robust segments through a minimum cost search on a directed robustness graph. Robust segments are executed open-loop while non-robust segments are executed with model-predictive control. I show through experiments on a real robotic system, that the open and closed-loop approach is more successful for reaching in clutter, while achieving more fluent/real-time execution in comparison with the closest competing baseline, closed-loop control.

The method presented in this chapter is similar to the online re-planning approach proposed in Chapter 3 where the robot does not re-plan at every step. It uses deviation between the planned state and the current real-world state to decide whether or not to re-plan. The problem with such an approach is that the current real-world state may be a failure state or it may be close to such a failure state, whereas a robust action (if available) would prevent the system from even reaching such states.

Chapter 5

Task-Adaptive Planning for Non-prehensile Manipulation Under Uncertainty

Can a robot be optimistic in the face of uncertainty? Can it adapt its actions to the accuracy requirements of a task? Unlike methods introduced in prior chapters, I propose a planning and control algorithm for non-prehensile manipulation that embraces uncertainty. The key feature of my algorithm is *task-adaptivity*: the planner can adapt to the accuracy requirements of a task, performing fast or slow pushes.

For example in Fig. 5.1 (top), the robot is pushing an object on a narrow strip. The task requires high-accuracy during pushing — otherwise the object can fall down. The controller therefore generates slow pushing actions that make small but careful progress to the goal pose of the object. In Fig. 5.1 (bottom), however, the object is on a wide table and the goal region for the object is large (circle drawn on the table). In this case, the controller generates only a small number of fast pushing actions to reach the goal quickly — even if this creates more uncertainty about the object's pose after each action, the task can still be completed successfully. I present a controller that can adapt to tasks with different accuracy requirements, such as in these examples.

A common feature of existing work is the reliance on the quasi-static model of pushing [69, 43]. While one reason of the popularity of the quasi-static model may be the simpler analytic equations of motion it enables one to derive, another reason is the slow nature of quasi-static interactions, which keeps the uncertainty during pushing tightly bounded and therefore easier to control accurately.

However, accuracy is not the main criterion for every task, as I illustrate in Fig. 5.1. Fast motions, even if inaccurate, may be desired during some tasks. We humans also adapt our actions to the task (Fitts's law [32]). Imagine reaching

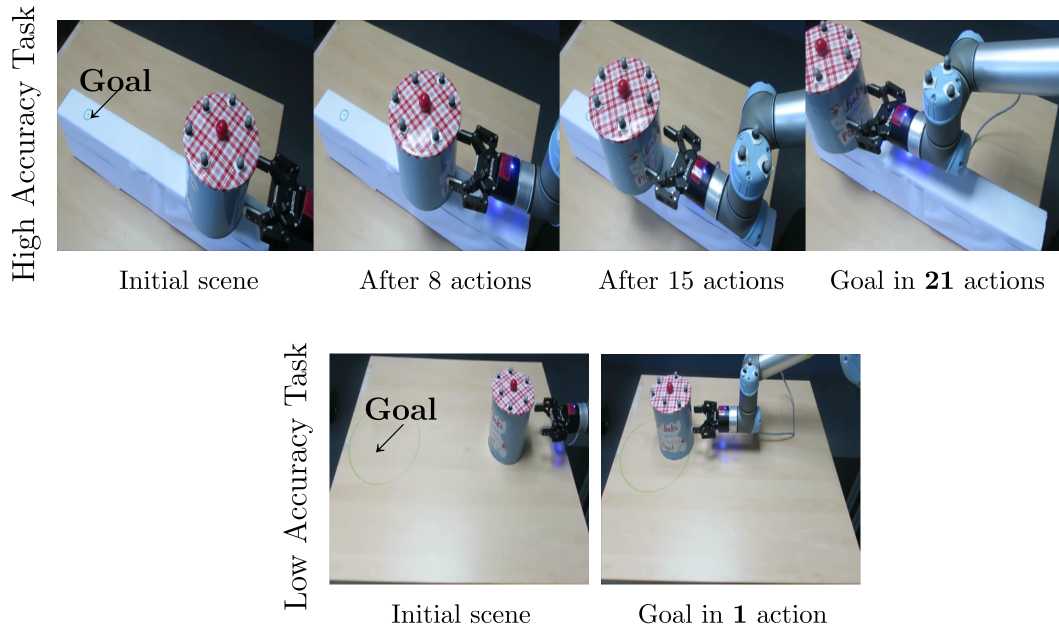


Figure 5.1: Task-adaptive pushing with 21 slow actions for a high accuracy task (top) and a single fast action for a low accuracy task (bottom).

into a fridge shelf that is crowded with fragile objects, such as glass jars and containers. You move slowly and carefully. However, if the fridge shelf is almost empty, only with a few plastic containers that are difficult-to-break, you move faster with less care.

The major requirements to build a task-adaptive planner/controller are:

1. The planner must **consider a variety of actions (different pushing speeds)**: The robot should not be limited to moving at quasi-static speeds. It must consider dynamic actions wherever possible to complete a given task as fast as possible.

2. The planner must **consider action-dependent uncertainty**: Different actions can induce different amounts of uncertainty into the system. For example, pushing an object for a longer distance (or equivalently pushing faster for a fixed amount of time) would induce more uncertainty than pushing a short distance (or equivalently pushing slower for a fixed amount of time) [97].

One way to build such a controller is to model the problem as a Markov Decision Process (MDP) with stochastic dynamics, where the stochasticity is action-dependent. Then, if this MDP is solved for an optimal policy under a cost that includes time to reach the goal, the resulting policy will use fast actions when it can, and fall back to slow actions for tasks that require higher accuracy.

In this chapter, I model the problem as an MDP. However, I do not search for a globally optimal policy as this would be prohibitively computationally expensive. Instead, I solve the MDP online [78, 50] with an approximate solution. Even in this online setting, evaluating the value of all possible actions (including actions of a wide variety of speeds), proves computationally expensive, since the cost of physics-based predictions is high. Therefore, instead of evaluating all possible actions, at any given state, I first use a fast trajectory optimizer to suggest a reduced set of promising actions, i.e. actions that are known to drive the system to the goal under the deterministic setting. I then evaluate these actions under uncertainty to pick the best one.

My specific contributions include a task-adaptive online solution to the MDP for pushing-based manipulation and a trajectory optimizer to generate actions for evaluation under the MDP setting. Additionally, I compare my task-adaptive planner with a standard model predictive control approach where the initial candidate control sequence for trajectory optimization is composed of only slow actions, for high and low accuracy tasks under different levels of uncertainty. I show that my approach achieves higher success rates in significantly smaller amounts of time, especially for tasks that do not require high accuracy. Finally, I implement my approach on a real robotic system for tasks requiring different accuracy levels and compare it with standard MPC. Results can be found in the video at `https://youtu.be/8rz_f_V0WJA`.

5.1 Task-Adaptive Planning as an MDP

I consider the problem where a robot must plan a sequence of non-prehensile actions to take an environment from an initial configuration to a desired goal configuration. I consider two task categories: In the *pushing* task the goal is to push a target object into a goal region; and in the *grasping in clutter* task the goal is to bring a target object, among other objects, into the robot's hand. My scenes contain D dynamic objects. $\boldsymbol{q}^i$ refers to the full pose of each dynamic object, for $i = 1, \ldots, D$. I assume a flat working surface and the robot is not allowed to drop objects off the edges.

The robot is planar with a 1-DOF gripper. The robot's configuration is defined by a vector of joint values $\boldsymbol{q}^R = \{\theta_x, \theta_y, \theta_{rotation}, \theta_{gripper}\}$. I represent the complete state of my system as $\boldsymbol{x}_t$ at time t. This includes the pose and velocity of the robot and all dynamic objects; $\boldsymbol{x}_t = \{\boldsymbol{q}^R, \boldsymbol{q}^1, \ldots, \boldsymbol{q}^D, \dot{\boldsymbol{q}}^R, \dot{\boldsymbol{q}}^1, \ldots, \dot{\boldsymbol{q}}^D\}$. My control inputs are velocities: $\boldsymbol{u}_t = \dot{\boldsymbol{q}}^R$ applied to the robot's joints. I then define

the stochastic discrete time dynamics of my system as:

$$\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t, \boldsymbol{u}_t) + \zeta(\boldsymbol{u}_t) \tag{5.1}$$

where f is a deterministic function that describes the evolution of state $\boldsymbol{x}_t$ given the action $\boldsymbol{u}_t$. I induce stochasticity in the system dynamics through $\zeta(\boldsymbol{u}_t) \propto ||\boldsymbol{u}_t||$, which is proportional to the magnitude of action $\boldsymbol{u}_t$. When I push an object over a long distance, there are a large number of interactions/contacts especially in cluttered environments. This implies that the uncertainty in the resulting state at the end of a long push should be larger than that for a shorter push [97].

I assume an initial state of the system $\boldsymbol{x}_0$. My goal is to generate a sequence of actions for the robot such that the desired final goal configuration of the environment is reached as quickly as possible without dropping objects off the edge of my working surface. The final goal configuration of the environment is defined in terms of the terminal cost of the system. It is a sub-level set of the terminal cost, which I will detail later in this Chapter. In the foregoing paragraphs, $\boldsymbol{U} = \{\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n-1}\}$ denotes a sequence of control signals of fixed duration Δ_t applied in n time steps and I use brackets to refer to the control at a certain time step, i.e. $\boldsymbol{U}[t] = \boldsymbol{u}_t$. Similarly, $\boldsymbol{X}$ is a sequence of states.

To build a task-adaptive controller, I formulate the problem as an MDP, and I provide an approximate solution to it. An MDP is defined by a tuple $< S, A, P, L' >$, where S is the set of states, A is the set of actions, P is the probabilistic transition function, and L' defines the costs. In the problem S is given by all possible values of $\boldsymbol{x}_t$. Similarly, A is given by all possible values of $\boldsymbol{u}_t$, and P can be computed using the stochastic transition function in Eq. 5.1. The optimal policy for an MDP is given by:

$$\pi^*(\boldsymbol{x}_t) = \underset{\boldsymbol{u}_t \in A}{\arg\min} \left[L'(\boldsymbol{x}_t, \boldsymbol{u}_t) + \gamma \cdot \int_S P(\boldsymbol{x}_{t+1} | \boldsymbol{x}_t, \boldsymbol{u}_t) \cdot V^*(\boldsymbol{x}_{t+1}) \cdot dx_{t+1} \right] \tag{5.2}$$

where $0 < \gamma < 1$ is the discount factor, and V^* is the optimal value function. An online one-step lookahead approximate solution to the MDP problem can be found by sampling and evaluating the average value over samples as in [78, 50]:

$$\widetilde{\pi}(\boldsymbol{x}_t) = \underset{\boldsymbol{u}_t \in A}{\arg\min} \left[L'(\boldsymbol{x}_t, \boldsymbol{u}_t) + \frac{1}{Q} \cdot \sum_{\boldsymbol{x}_{t+1} \in S(\boldsymbol{x}_t, \boldsymbol{u}_t, Q)} \widetilde{V}(\boldsymbol{x}_{t+1}) \right] \tag{5.3}$$

where $S(\boldsymbol{x}_t, \boldsymbol{u}_t, Q)$ is the set of Q samples found by stochastically propagating $(\boldsymbol{x}_t, \boldsymbol{u}_t)$, and $\widetilde{V}$ is an approximation of the value function. For the problem, to compute the cost $L'(\boldsymbol{x}_t, \boldsymbol{u}_t)$, I use a cost function L which also takes into account the next state $\boldsymbol{x}_{t+1}$:

$$L(\boldsymbol{x}_t, \boldsymbol{x}_{t+1}, \boldsymbol{u}_t) = \sum_{i}^{D}\{w_e \cdot e^{k \cdot d_E^i} + w_s \cdot (\boldsymbol{x}_{t+1}^i - \boldsymbol{x}_t^i)^2\} + k_{act} \tag{5.4}$$

The first term in the cost which I call the *edge cost* penalizes pushing an object close to the table's boundaries or static obstacles. I have shown the edge cost in Fig. 3.3 where I defined a safe zone smaller than the table's boundaries. If an object is pushed out of this safe region as a result of an action between t and $t+1$, I compute the pushed distance d_E. Also note that k is a constant term and no edge costs are computed for objects in the safe zone. The second term is the *environment disturbance cost* which penalizes moving dynamic objects away from their current states. The third term, k_{act} is a constant cost incurred for each action taken by the robot. I use w_e and w_s to represent weights for the edge and environment disturbance costs respectively. Then, I compute $L'(\boldsymbol{x}_t, \boldsymbol{u}_t)$ using the same set of Q samples:

$$L'(\boldsymbol{x}_t, \boldsymbol{u}_t) = \frac{1}{Q} \cdot \sum_{\boldsymbol{x}_{t+1} \in S(\boldsymbol{x}_t, \boldsymbol{u}_t, Q)} L(\boldsymbol{x}_t, \boldsymbol{x}_{t+1}, \boldsymbol{u}_t)$$

This solution requires propagating Q samples for every possible action $\boldsymbol{u_t}$, to find the one with the minimum total cost. Performing this for all actions $\boldsymbol{u}_t \in A$ is not feasible for my purposes for a variety of reasons: First, I am interested in actions that span a wide range of speed profiles (i.e. fast and slow), which make my potential action set large; second, each propagation in my domain is a physics simulation which is computationally expensive; and third, my goal is closed-loop pushing behaviour close to real-time speeds.

5.2 Approximate Online MDP Solution

Instead of considering a large action set in the online MDP solution (Eq. 5.3), I propose to use a small set of promising actions including both fast and slow actions. I identify such a set of actions using a trajectory optimizer based on the deterministic dynamics function f of the system.

Note that my approach does not discretize the action space or the state space *a priori*. I adaptively sample the action space using a trajectory optimizer to

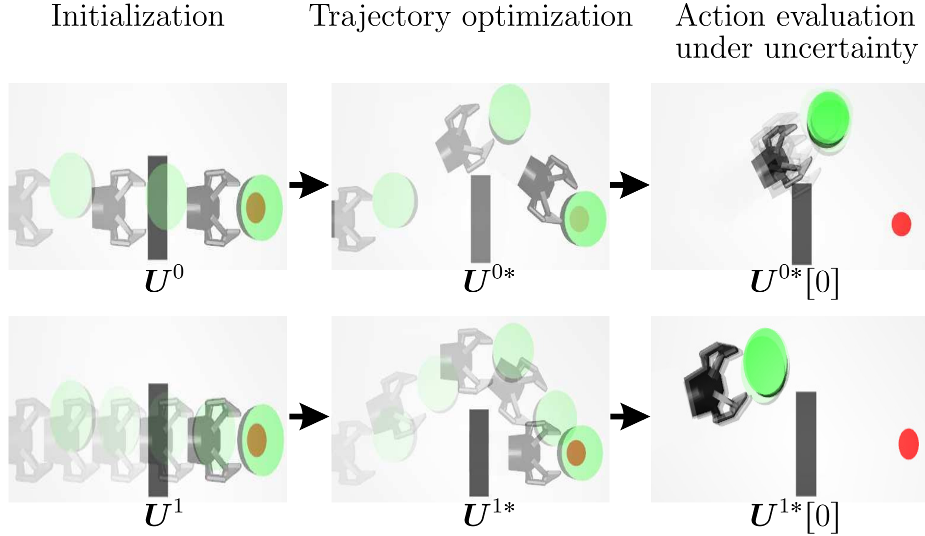


Figure 5.2: First column: Initialization of the task-adaptive approach with control sequences including fast (top) and slow (bottom) actions. Second column: Stochastic trajectory optimization of the initial candidate control sequences. Last column: Action evaluation under uncertainty through sampling.

find high value actions to consider at a given current state. I also get stochastic next state samples by applying these actions through a physics simulator.

In Alg. 6, I present my online approximate solution to the MDP.

Consider the scene in Fig. 5.2, where I have a planar gripper and an object. My task is to push the object to a desired goal location (the red spot) while avoiding the rectangular black obstacle. I begin by generating N candidate action sequences $\{\boldsymbol{U}^0, \ldots, \boldsymbol{U}^{N-1}\}$ to the goal by using the *GetActionSequences* procedure (line 1). The number of actions in each sequence varies between n_{min} and n_{max}, and each action is of fixed duration Δ_t. In the example task (Fig. 5.2), I show the candidate action sequences in the first column where $N = 2$, $n_{min} = 2$, and $n_{max} = 4$. Since each action is of fixed duration, the set of action sequences contain both fast (top) and slow (bottom) actions. Details of how I generate these candidate control sequences are explained in Sec 5.3.

Using a trajectory optimizer (Sec. 5.4), the procedure GetOptActionSequences returns N optimized control sequences $\{\boldsymbol{U}^{0*}, \cdots, \boldsymbol{U}^{N-1*}\}$, and an approximation of the value function $\{\widetilde{\boldsymbol{V}}^0, \cdots, \widetilde{\boldsymbol{V}}^{N-1}\}$ along the optimal trajectories. I visualize the optimized trajectories for the example task in the second column.

I seek a one-step lookahead solution to the *MDP*, hence, I get the first actions $\boldsymbol{U}^{i*}[0]$ from each of the optimized control sequences. My task is now to select the best amongst N actions even under uncertainty. I apply each action Q times to the stochastic state transition function in Eq. 7.1 to yield Q next state samples (line 7). More specifically, using my system dynamics model, I apply

Algorithm 6: Online MDP solver

Input : $\boldsymbol{x}_0$: Initial state

Parameters : Q: Number of stochastic samples
N: Number of initial candidate control sequences
n_{min}: Min. num. of actions in a control sequence
n_{max}: Max. num. of actions in a control sequence

1 $\{\boldsymbol{U}^0, \ldots, \boldsymbol{U}^{N-1}\} \leftarrow$GetActionSequences$(\boldsymbol{x}_0, n_{min}, n_{max}, N)$
2 **while** *task not complete* **do**
3 $\quad \{\boldsymbol{U}^{0*}, \cdots, \boldsymbol{U}^{N-1*}\}$, $\{\widetilde{\boldsymbol{V}}^0, \cdots, \widetilde{\boldsymbol{V}}^{N-1}\} \leftarrow$
4 $\quad\quad$ GetOptActionSequences $(\boldsymbol{x}_0, \{\boldsymbol{U}^0, \ldots, \boldsymbol{U}^{N-1}\})$
5 $\quad$ **for** $i \leftarrow 0$ **to** $N-1$, **do**
6 $\quad\quad V^i = 0$
7 $\quad\quad$ **for** *each sample in* Q, **do**
8 $\quad\quad\quad \boldsymbol{x}_1 =$StochasticExecution$(\boldsymbol{x}_0, \boldsymbol{U}^{i*}[0])$
9 $\quad\quad\quad V^i = V^i + L(\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{U}^{i*}[0]) + \widetilde{\boldsymbol{V}}^i[0]$
10 $\quad\quad V^i = V^i/Q$
11 $\quad i_{min} = \arg\min_{i \in N} V^i$
12 $\quad \boldsymbol{x}_1 \leftarrow$ execute $\boldsymbol{U}^{i_{min}}[0]$
13 $\quad$ check task completion
14 $\quad \boldsymbol{x}_0 \leftarrow \boldsymbol{x}_1$
15 $\quad \{\boldsymbol{U}^0, \ldots, \boldsymbol{U}^{N-1}\} \leftarrow$
$\quad\quad \{$GetActionSequences$(n_{min}, n_{max}, N-1)$, $\boldsymbol{U}^{i_{min}}[1:n-1]\}$

the controls (velocities) to the robot for the control duration Δ_t and thereafter I wait for a fixed extra time t_{rest} for objects to come to rest before returning the next state and computing the cost. I can see the Q samples in the third column (Fig. 5.2) for the example pushing task. Thereafter, on line 9, for each sample I add the immediate cost L to the approximate value $\widetilde{\boldsymbol{V}}^i[0]$ for the resulting state. I use the same approximate value for all Q samples. I then compute an average value for each action (line 10), select the best one and execute it (line 12). If the task is not yet complete, I repeat the whole process but also re-use the remaining portion of the best control sequence $\boldsymbol{U}^{i_{min}}[1:n-1]$ from the current iteration, thus generating only $N-1$ new action sequences (line 15). This algorithm chooses slow, low velocity actions for tasks that require high accuracy but faster actions for tasks that allow inaccuracies.

In the example pushing task, Fig. 5.2 (third column), I see a wider distribution in the resulting state for the fast action (top) compared to the slower action (bottom) as a result of my uncertainty model. Such a wide distribution in the resulting state increases the probability of undesired events happening especially in high accuracy tasks. For example, I see that some samples for the fast action result in collisions between the robot and the obstacle. This implies high costs with respect to the slower action, hence my planner chooses

Algorithm 7: GetActionSequences $(\boldsymbol{x}_0, n_{min}, n_{max}, N)$

Output : $\{\boldsymbol{U}^0, \ldots, \boldsymbol{U}^{N-1}\}$: A set of candidate action sequences
Parameters : Δ_t: Control duration for each action in an action sequence

1 **for** $k \leftarrow 0$ **to** $N-1$ **do**
2   $n^k = \lceil \frac{n_{max} - n_{min}}{N-1} k \rceil + n_{min}$
3   $\boldsymbol{U}^k[0 : n^k - 1] = \{\frac{Distance\ to\ goal}{n^k \Delta_t}\}\mathbf{1}$
4 **return** $\{\boldsymbol{U}^0, \ldots, \boldsymbol{U}^{N-1}\}$

Algorithm 8: GetOptActionSequences$(\boldsymbol{x}_0, \{\boldsymbol{U}^0, \ldots, \boldsymbol{U}^{N-1}\})$

Output : $\{\boldsymbol{U}^{0*}, \cdots, \boldsymbol{U}^{N-1*}\}$: Optimized set of action sequences
$\{\widetilde{\boldsymbol{V}}^0, \ldots, \widetilde{\boldsymbol{V}}^{N-1}\}$ Approx. value function along N optimal trajectories

1 **for** $i \leftarrow 0$ **to** $N-1$ **do**
2   $\boldsymbol{U}^{i*}, \widetilde{\boldsymbol{V}}^i \leftarrow$TrajectoryOptimization$(\boldsymbol{x}_0, \boldsymbol{U}^i)$
3 **return** $\{\boldsymbol{U}^{0*}, \cdots, \boldsymbol{U}^{N-1*}\}$, $\{\widetilde{\boldsymbol{V}}^0, \cdots, \widetilde{\boldsymbol{V}}^{N-1}\}$

the slow action in this case, executes it and starts the whole process again from the resulting state.

5.3 Generating a Variety of Actions

At each iteration of my online MDP solver, I provide N actions that are evaluated under uncertainty. First, using the *GetActionSequences* procedure in Alg. 7, I generate N candidate action sequences where the number of actions in an action sequence increases linearly from n_{min} to n_{max} (line 2).

On line 3, each of the action sequences is set to a straight line constant velocity profile to the goal. This is a simple approach, other more complicated velocity profiles could also be used here. Furthermore, in the GetOptActionSequences procedure, Alg. 8, I use these candidate action sequences to initialize a stochastic trajectory optimization algorithm (Sec. 5.4). The algorithm quickly finds and returns a locally optimal solution for each of the candidate control sequences. It also returns an approximation of the value function along N optimal trajectories. I use this approximate value while evaluating actions in my online MDP solver.

Algorithm 9: Stochastic Trajectory Optimization

Input : $\boldsymbol{x}_0$: Initial state
$\boldsymbol{U}$: Candidate control sequence containing n actions
Output : $\boldsymbol{U}^*$: Optimal control sequence
Parameters : K: Number of noisy trajectory rollouts
$\boldsymbol{\nu}$: Sampling variance vector
C_{thresh}: Success definition in terms of cost
I_{max}: Maximum number of iterations

1 $\boldsymbol{X}, \boldsymbol{C} \leftarrow$TrajectoryRollout$(\boldsymbol{x}_0, \boldsymbol{U})$
2 **while** I_{max} *not reached* ***and*** *Sum*$(\boldsymbol{C}) > C_{thresh}$ **do**
3   **for** $k \leftarrow 0$ **to** $K-1$ **do**
4     $\boldsymbol{\delta U}^k \leftarrow N(\boldsymbol{0}, \boldsymbol{\nu})$ Random control sequence variation
5     $\boldsymbol{U}^k = \boldsymbol{U} + \boldsymbol{\delta U}^k$
6     $\boldsymbol{X}^k, \boldsymbol{C}^k \leftarrow$TrajectoryRollout$(\boldsymbol{x}_0, \boldsymbol{U}^k)$
7   $\boldsymbol{U}^* \leftarrow$UpdateTrajectory$(\boldsymbol{U}, \{\boldsymbol{\delta U}^0, \ldots, \boldsymbol{\delta U}^{K-1}\}, \{\boldsymbol{C}^0, \ldots \boldsymbol{C}^{K-1}\})$
8   $\boldsymbol{X}^*, \boldsymbol{C}^* \leftarrow$TrajectoryRollout$(\boldsymbol{x}_0, \boldsymbol{U}^*)$
9   **if** *Sum*$(\boldsymbol{C}^*) <$ *Sum*$(\boldsymbol{C})$ **then**
10     $\boldsymbol{U} \leftarrow \boldsymbol{U}^*, \quad \boldsymbol{X} \leftarrow \boldsymbol{X}^*, \quad \boldsymbol{C} \leftarrow \boldsymbol{C}^*$
11 **for** $j \leftarrow 0$ **to** $n-1$ **do**
12   $\widetilde{\boldsymbol{V}}[j] = \sum_{h=j}^{n-1} \boldsymbol{C}^*[h]$ Approx. value function for state $\boldsymbol{x}_j^*$ along the optimal trajectory
13 **return** $\boldsymbol{U}^*, \widetilde{\boldsymbol{V}}$

5.4 Trajectory Optimization

Trajectory optimization involves finding an optimal control sequence $\boldsymbol{U}^*$ for a planning horizon n, given an initial state $\boldsymbol{x}_0$, an initial candidate control sequence $\boldsymbol{U}$, and an objective J which can be written as:

$$J(\boldsymbol{X}, \boldsymbol{U}) = \sum_{t=0}^{n-1} L(\boldsymbol{x}_t, \boldsymbol{x}_{t+1}, \boldsymbol{u}_t) + w_f L_f(\boldsymbol{x}_n) \tag{5.5}$$

J is obtained by applying the control sequence $\boldsymbol{U}$ starting from a given initial state and includes the sum of running costs L and a final cost L_f. I use the constant w_f to weight the terminal cost with respect to the running cost. I consider a deterministic environment defined by the state transition function $\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t, \boldsymbol{u}_t)$. This is a constraint that must be satisfied at all times. Then the output of trajectory optimization is the minimizing control sequence:

$$\boldsymbol{U}^* = arg \min_{\boldsymbol{U}} J(\boldsymbol{X}, \boldsymbol{U}) \tag{5.6}$$

In this chapter, I propose Alg. 9, which adapts the STOMP algorithm [49] for non-prehensile object manipulation.

I begin with an initial candidate control sequence $\boldsymbol{U}$ and iteratively seek

Algorithm 10: TrajectoryRollout

Input : $\boldsymbol{x}_0$: Initial state
$\boldsymbol{U}$: Control sequence with n actions

Output : $\boldsymbol{X}$:State sequence
$\boldsymbol{C}$:Costs along the trajectory

1 **for** $t \leftarrow 0$ **to** $n-1$ **do**
2 $\quad \boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t, \boldsymbol{u}_t)$
3 $\quad \boldsymbol{C}[t] = L(\boldsymbol{x}_t, \boldsymbol{x}_{t+1}, \boldsymbol{u}_t)$ Calculate cost using Eq. 5.4
4 $\quad$ **if** $t == n-1$ **then**
5 $\quad\quad \boldsymbol{C}[t] = \boldsymbol{C}[t] + L_f(\boldsymbol{x}_{t+1})$ Add final cost
6 **return** $\boldsymbol{X}, \boldsymbol{C}$

lower cost trajectories (lines 2-10) until the cost reaches a threshold or until the maximum number of iterations is reached (line 2). I add random control sequence variations $\boldsymbol{\delta U}^k$ on the candidate control sequence to generate K new control sequences at each iteration (line 5). Thereafter on line 6, I do a trajectory rollout for each sample control sequence using the TrajectoryRollout procedure in Alg. 10. It returns the corresponding state sequence $\boldsymbol{X}$ and costs $\boldsymbol{C}$ calculated for each state along the resulting trajectory. I.e $\boldsymbol{C}[t]$ is the cost of applying action $\boldsymbol{u}_t$ in state $\boldsymbol{x}_t$. After generating K sample control sequences and their corresponding costs, the next step is to update the candidate control sequence using the UpdateTrajectory procedure. One way to do this is a straightforward greedy approach where the minimum cost trajectory is selected as the update:

$$k* = arg\min_k \sum_{t=0}^{n-1} \boldsymbol{C}^k[t] \quad , \qquad \boldsymbol{U}^* = \boldsymbol{U} + \boldsymbol{\delta U}^{k*} \tag{5.7}$$

Another approach is a cost-weighted convex combination similar to [94]:

$$\boldsymbol{U}^*[t] = \boldsymbol{U}[t] + \frac{\sum_{k=0}^{K-1}[\exp(-(\frac{1}{\lambda})\boldsymbol{C}^k[t])]\boldsymbol{\delta U}^k[t]}{\sum_{k=0}^{K-1}\exp(-(\frac{1}{\lambda})\boldsymbol{C}^k[t])} \tag{5.8}$$

Where λ is a parameter to regulate the exponentiated cost's sensitivity. In my experiments, for a small number of noisy trajectory rollouts K (e.g. $K = 8$), a greedy update performs better. Hence, I use the greedy update in all my experiments.

Once the trajectory update step is complete, I update the candidate control sequence only if the new sequence has a lower cost (line 9, Alg. 9). The trajectory optimization algorithm then returns the locally optimal control sequence and an approximation of the value function for each state along the trajectory, where the value function is approximated using the sum of costs starting from

that state.

The cost terms for the state-action sequences in this algorithm are equal to the running costs in Eq. 5.4, with the addition of a terminal cost on the final state depending on the task. The terminal cost for the pushing task is given by:

$$L_f = \begin{cases} 0 & \text{if } R_o - R_g < 0 \\ (R_o - R_g)^2 & \text{if } R_o - R_g > 0 \end{cases}$$

where R_o is the distance between the pushed object and the center of a circular goal region of radius R_g. The terminal cost term for the task of grasping in clutter is given by: $L_f = d_T^2 + w_\phi \cdot \phi_T^2$. I have shown how the distance d_T and the angle ϕ_T are computed in Fig. 3.2. I use w_ϕ to weight angles relative to distances.

5.5 Baseline Approach

I implement a standard model predictive control algorithm (MPC) as a baseline approach. It involves repeatedly solving a finite horizon optimal control problem using the stochastic trajectory optimizer presented in Alg. 9 and proceeds as follows: optimize a finite horizon control sequence, execute the first action, get the resulting state of the environment and then re-optimize to find a new control sequence. When re-optimizing, I warm-start the optimization with the remaining portion of the control sequence from the previous iteration such that optimization now becomes faster. I initialize the trajectory optimizer for standard MPC with actions at the quasi-static speed. In addition, I propose another baseline approach to compare against in this work: uncertainty aware model predictive control (UAMPC). This is a version of my online MDP solver where only low speed actions are considered. Specifically, all the candidate control sequences are generated with the maximum number of actions i.e. $n_{min} \leftarrow n_{max}$.

5.6 Experiments

I call my planning approach task-adaptive model predictive control (TAMPC). I investigate how well my approach is able to handle uncertainty and adapt to varying tasks. First, I compare the performance of TAMPC with a standard model predictive control (MPC) approach. Here I hypothesize that TAMPC will complete a given pushing task within a significantly shorter period of time

and will be able to adapt to different tasks, maintaining a high success rate under varying levels of uncertainty.

Next, I compare the performance of my approach with uncertainty aware MPC (UAMPC). Here I hypothesize that: UAMPC will have a similar success rate and will take a longer amount of time to complete the task in comparison with TAMPC.

I conduct experiments in simulation and on a real robotic system. I consider the tasks of pushing an object to a goal region and grasping an object in clutter. Given an environment for planning, I create two instantiations:

- **Planning environment:** The robot generates plans in the simulated planning environment. The trajectory optimizer (Alg. 9) uses deterministic physics during planning and my online MDP solver uses stochastic physics to evaluate actions.

- **Execution environment:** Here, the robot executes actions and observes the state evolution. It is the physical world for real robot experiments but it is simulated for simulation experiments. The execution environment is stochastic.

For the planning environment and the execution environment when it is simulated, I use a physics engine, Mujoco[92], to model the deterministic state transition function f in Eq. 7.1. I model stochasticity in the physics engine by adding Gaussian noise on the velocities $\{\dot{\boldsymbol{q}}^R, \dot{\boldsymbol{q}}^1, \dots, \dot{\boldsymbol{q}}^D\}$ of the robot and objects at every simulation time step:

$$\{\dot{\tilde{\boldsymbol{q}}}^R, \dot{\tilde{\boldsymbol{q}}}^1, \dots, \dot{\tilde{\boldsymbol{q}}}^D\} = \{\dot{\boldsymbol{q}}^R, \dot{\boldsymbol{q}}^1, \dots, \dot{\boldsymbol{q}}^D\} + \boldsymbol{\mu}, \quad \boldsymbol{\mu} \sim \mathcal{N}(0, \beta(\boldsymbol{u_t})) \tag{5.9}$$

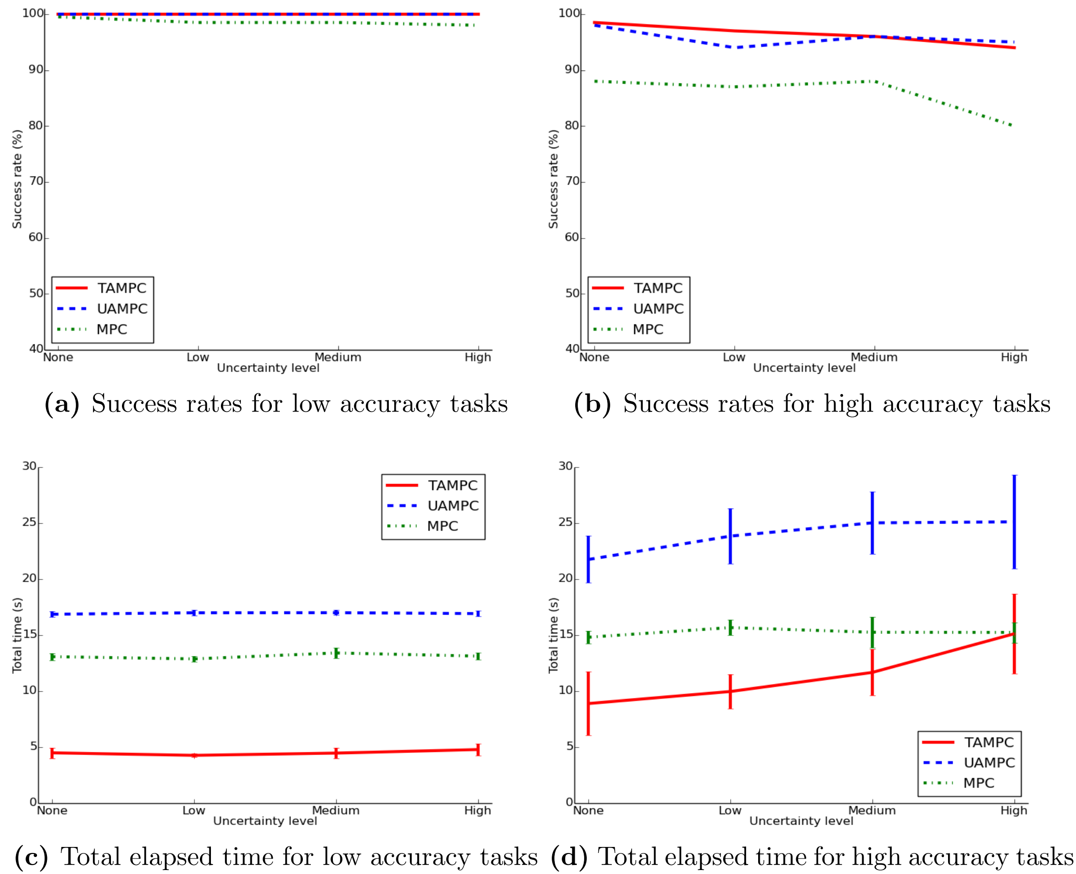
where $\mathcal{N}$ is the Gaussian distribution and β is an action dependent variance function. I create a linear model for the variance function as:

$$\beta(\boldsymbol{u}_t) = b||\boldsymbol{u}_t|| \tag{5.10}$$

Where b is a constant. In my simulation experiments, *uncertainty level* refers to the degree of stochasticity dictated by the slope b of the variance function β used to generate the Gaussian noise μ injected at every simulation time step in Eq. 5.9.

5.6.1 Push planning simulation experiments

I present a high accuracy task in Fig. 5.1 (top). It is made up of a thin strip and a small goal region. I also define a low accuracy task in Fig. 5.1 (bottom)



(a) Success rates for low accuracy tasks **(b)** Success rates for high accuracy tasks

(c) Total elapsed time for low accuracy tasks **(d)** Total elapsed time for high accuracy tasks

Figure 5.3: Success rate and total elapsed time versus uncertainty level for low and high accuracy tasks.

which is a much larger table with a wider goal region. I create 200 such planning environments for each of the high and low accuracy tasks. For each environment:

1. I randomly select the shape (box or cylinder) of the pushed object.
2. For each object, I randomly[1] select shape dimensions (radius and height for the cylinder, extents for the boxes), mass, and coefficient of friction.
3. I randomly[2] select a position on the working surface for the pushed object.

I create four uncertainty levels: no uncertainty, low uncertainty, medium uncertainty and high uncertainty. For the no uncertainty case, no extra noise was added to the physics engine. For low, medium and high levels of uncertainty, $b = \{0.05, 0.075, 0.1\}$ respectively. I test the different planning and control approaches and specify a timeout of 3 minutes including all planning, re-planning and execution.

[1]The uniform range used for each parameter is given here. Box x-y extents: $[0.05m, 0.075m]$; box height: $[0.036m, 0.05m]$; cylinder radius:$[0.04m, 0.07m]$; cylinder height:$[0.04m, 0.05m]$; mass:$[0.2kg, 0.8kg]$; coef. fric.:$[0.2, 0.6]$.

[2]The random position for the pushed object is sampled from a Gaussian with a mean at the lower end of the table ($0.1m$ from the edge of a $0.6m$ long table along the center axis and a variance of $0.01m$.)

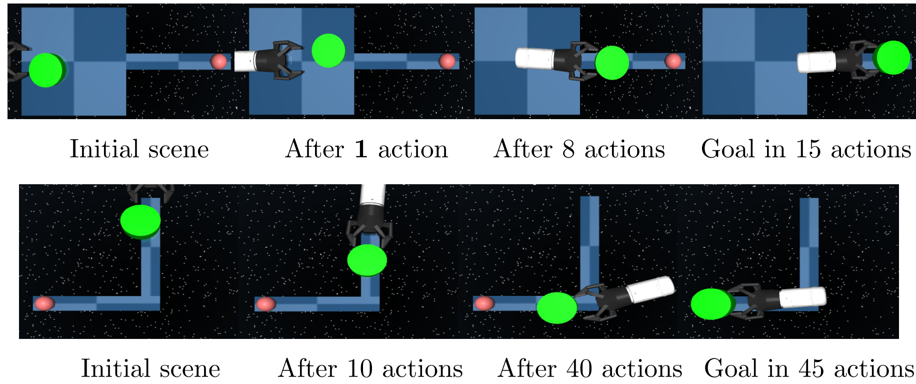
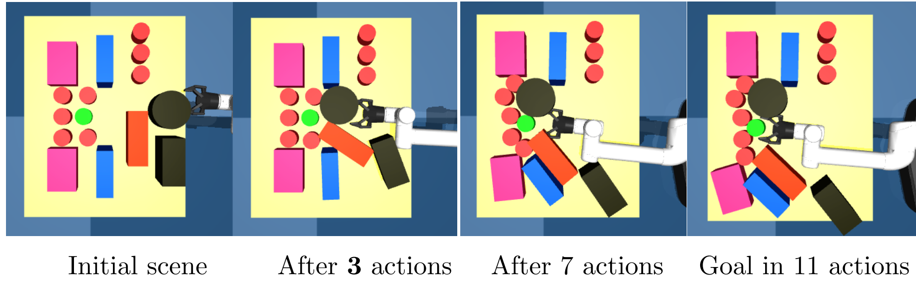


Figure 5.4: Push planning in a changing environment (top) using a single fast push initially and then slow pushes later on due to the narrow strip. For the L-shaped environment (bottom), the robot executes many actions to successfully navigate the edge.

Success rates: I declare success when the robot is able to push an object to the target region without dropping it off the edge of the table within the specified time limit. I plot the results in Fig. 5.3a and Fig. 5.3b. For the low accuracy level push planning task (Fig. 5.3a), TAMPC and UAMPC were able to maintain a 100 % success rate while MPC showed a slight decrease in success rates as uncertainty grew. For the high accuracy pushing task (Fig. 5.3b), TAMPC and UAMPC were also able to maintain a good average success rate. MPC on the other hand maintains a poor success rate. The major reason for this is uncertainty.

Total time: The total time in my experiments includes all planning and execution time. Fig 5.3c and Fig 5.3d show the average of 200 scenes with 95 % confidence interval of the mean. For the low accuracy level task, my TAMPC planner is able to achieve the goal in under 5s (Fig 5.3c), while UAMPC and MPC took significantly more time to complete the task. This clearly shows that my method is able to generate successful fast actions while maintaining a high success rate. For the high accuracy level task (Fig 5.3d), my planner is able to generate as many small actions as needed as the uncertainty grew. Hence it was able to maintain a high success rate and still complete the task within a very small amount of time in comparison with the baseline approach. Furthermore, I also test the adaptive behavior of my approach for the environments in Fig. 5.4. In the changing environment (Fig. 5.4, top), the robot begins with a fast push due to a large initial area. Thereafter, it naturally switches to slow pushes on the thin strip to complete the task. For pushing in the L-shaped environment (Fig. 5.4, bottom), the robot generally pushes slow. However, it spends a lot of time to navigate the corner.



Initial scene | After **3** actions | After 7 actions | Goal in 11 actions

Figure 5.5: Grasping in clutter: The robot uses fast actions initially but chooses slower actions as it gets closer to the goal object near the edge of the table.

5.6.2 Grasping in clutter simulation experiments

I conducted simulation experiments for grasping in clutter in scenes similar to Fig. 5.5. My scenes are randomly generated containing boxes and cylinders. In addition, my robot now has four control inputs (including the gripper). I tested the task adaptive planner in clutter to observe how the planner adapts given different environment configurations. I see that the robot manipulates clutter and is able to grasp the target object. An example scene is shown in Fig. 5.5 where the aim is to grasp the target object in green without pushing any other objects off the edge of the table. The robot initially begins with fast actions to push obstacles out of the way. However, as the robot gets closer to the target object, it chooses slower actions due to a higher probability of task failure in that region.

5.6.3 Real robot experiments

In the real robot experiments I use a UR5 robot with a Robotiq 2 finger gripper. I restrict the motion of the gripper to a plane parallel to the working surface such that I have a planar robot. I use OpenRave[23] to find kinematics solutions at every time step. For the push planning experiments, the gripper is completely open such that the robot receives three control inputs $\boldsymbol{u}_t = (\dot{\theta}_x, \dot{\theta}_y, \dot{\theta}_{rotation})$ at every time step. I use a medium uncertainty level to model the real world stochasticity. I place markers on the pushed object and track its full pose with a motion capture system (OptiTrack). I manually replicated three execution worlds for each task accuracy level from the randomly generated environments I created during push planning simulation experiments. I tested my planners in these environments. I show snapshots from my real robot experiments. In Fig. 5.6 (top), I have a low task accuracy environment where the standard MPC approach is successful after 20 actions. However, by using a single dynamic push

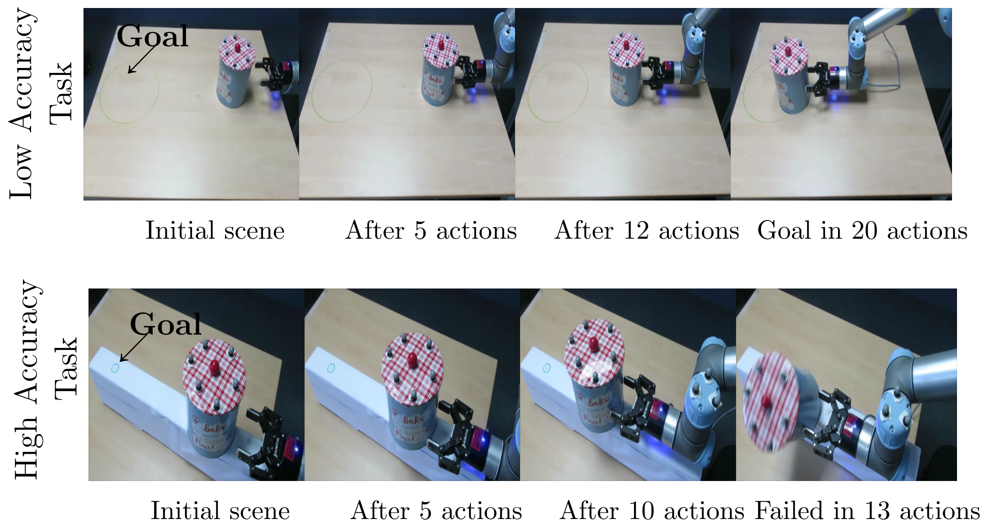


Figure 5.6: *MPC* using a large number of actions to complete a low accuracy level task (top), and causing the pushed object to fall off for a high accuracy level task (bottom).

in Fig. 5.1 (bottom), my task-adaptive control approach is able to complete the push planning task in under 2 seconds.

Moreover, for the high task accuracy problem, *MPC* was unable to push the target object to the desired goal location (Fig. 5.6 (bottom)). It executes actions without reasoning about uncertainty and pushed the goal object off the edge. The task-adaptive controller was able to consider uncertainty while generating small pushes (Fig. 5.1 (top)) to complete the task. These results can be found in the accompanying video at `https://youtu.be/8rz_f_V0WJA`.

5.7 Discussion

I presented a closed-loop planning and control algorithm capable of adapting to the accuracy requirements of a task by generating both fast and slow actions. This is an exciting first step toward realizing task-adaptive manipulation planners. In this work, I use a stochastic trajectory optimizer that outputs locally optimal control sequences. Thus, the resulting policy can get stuck in a local optima. Moreover, the trajectory optimizer may not return a good control sequence that reaches the goal for a given task if some design parameters (e.g. n_{max}) are chosen poorly.

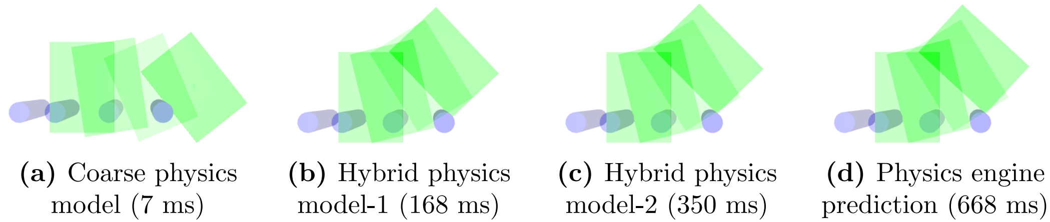
Chapter 6

Combining Coarse and Fine Physics for Manipulation using Parallel-in-Time Integration

Physics predictions are used during multi-contact interactions for methods throughout Chapters 3, 4, and 5. These predictions are computationally expensive. How can we make physics predictions for robotic manipulation faster?

In this chapter, I present a method for fast and accurate physics predictions during non-prehensile manipulation planning and control. Take the case study in Fig.6.1, where a cylindrical object moves towards the right, pushing a box. I am interested in predicting the motion of the pushed box, in a fast and accurate way. To achieve this, I combine *coarse* physics models with *fine* physics models. By coarse models, I mean computationally cheap but relatively inaccurate predictive physical models. For example in Fig.6.1a, I use a coarse model to compute the motion of the box. The motion is not completely realistic, but I can compute it extremely fast (7 ms wall-clock time to compute a simulated 8 s push). By fine models, I mean computationally expensive but accurate predictive physical models. In Fig.6.1d, I use a fine model (in this case, the Mujoco simulator [92]) to compute the motion of the same box. The motion is more realistic, but it also requires much more time to compute (668 ms).

I combine these two models to deliver a prediction that is as accurate as the fine model but runs in substantially less wall-clock time. The motion predicted in Fig.6.1b is similar to the fine model prediction, but is four times faster to compute. The motion predicted in Fig.6.1c is indistinguishable from the fine model prediction for real world manipulation purposes, and is two times faster to compute.



(a) Coarse physics model (7 ms) **(b)** Hybrid physics model-1 (168 ms) **(c)** Hybrid physics model-2 (350 ms) **(d)** Physics engine prediction (668 ms)

Figure 6.1: A spectrum of physics predictions from cheapest and least accurate (a) to expensive and most accurate (d).

Given an initial state and a sequence of controls, the problem of predicting the resulting sequence of states is a key component of a variety of model-based planning and control algorithms [41, 49, 47, 94, 11, 39, 3, 90, 61]. Mathematically, such a prediction requires solving an *initial value problem*. Typically, those are solved through numerical integration over time-steps (e.g. Euler's method or Runge-Kutta methods) using an underlying physics model. However, the speed with which these accurate physics-based predictions can be performed is still slow [26] and faster physics-based predictions can contribute significantly to contact-based/non-prehensile manipulation planning and control.

There are several ways that could be used to construct coarse models for manipulation planning. Quasi-static physics, which ignores accelerations, is widely used in non-prehensile manipulation planning and control [66, 69, 34], and can be seen as a coarse model. Learning is another method which can be used to generate approximate but fast predictions [71, 56, 81, 100]. Recently, with the advance of deep-learning, there has been multiple attempts at learning approximate "intuitive" physics models which are then used for manipulation planning [7, 28, 29, 79, 70, 25, 8]. Especially when these networks are faced with novel objects that are not in their training data (e.g. consider a network trained with boxes and cylinders, but used to predict the motion of an ellipse) they can generate approximate predictions of motion, and therefore are good candidates as coarse models.

A key question I investigate is whether I can combine such cheap but approximate models, with expensive but more accurate and general models (such as physics engines) to generate a hybrid model that is at the required speed and accuracy for a given manipulation task.

I do this by using a coarse physics model to obtain a rough initial guess of the state at each time point of a trajectory. Then, I evaluate the fine physics model in parallel across time starting from the initial guesses. Thereafter, I combine the coarse and fine predictions using the iterative Parareal algorithm [68, 62, 82].

In this chapter, I use this approach to perform physics-based predictions

within a planner for robotic manipulation. Specifically, I consider the task of pushing an object to a goal location while avoiding obstacles. I provide a cheap coarse model and combine it with the Mujoco physics engine as the fine model.[1] The planner performs trajectory optimization to generate a control sequence, executes the first control in the sequence, and then re-runs the trajectory optimizer, in a model-predictive-control fashion. I present this planner in Sec. 6.2. As a baseline, I use the same planner, with the fine model, Mujoco, as the predictive model. I conduct experiments in simulation and on a real setup and show that the planner with hybrid physics models achieves the same success rates but faster.

To the best of our knowledge, the use of Parareal for contact dynamics (and in general for robotic planning and control) has not been investigated before. When used for contact dynamics, the original Parareal formulation can produce infeasible states where rigid bodies penetrate. I extend Parareal to handle these infeasible state updates through projections to the feasible state space.

6.1 Combining Physics Models for Planning

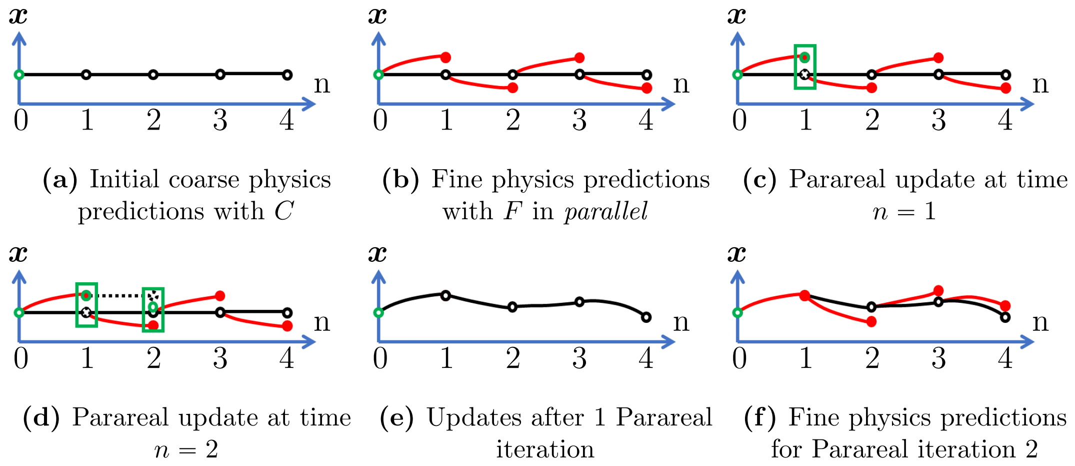
Given an initial state $\boldsymbol{x}_0$ and a sequence of N controls $\{\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{N-1}\}$, I am interested in predicting the resulting sequence of states $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$ of a physical system. As an example, I consider the problem of pushing an object to a goal location with a cylindrical pusher. The system's state consists of the pose $\boldsymbol{q}$ and velocity $\dot{\boldsymbol{q}}$ of the pusher P and slider S: $\boldsymbol{x}_n = [\boldsymbol{q}_n^P, \boldsymbol{q}_n^S, \dot{\boldsymbol{q}}_n^P, \dot{\boldsymbol{q}}_n^S]$. The slider's pose consists of the translation and rotation of the object on the plane $\boldsymbol{q}^S = [q^{S_x}, q^{S_y}, q^{S_\theta}]^T$. The pusher's pose is: $\boldsymbol{q}^P = [q^{P_x}, q^{P_y}]^T$ and control inputs are velocities $\boldsymbol{u}_n = [u_n^x, u_n^y]^T$ applied on the pusher for a control duration of Δt.

To predict the next state of the system given an initial state and a control input, I need a physics model F. I use a general physics engine [92] to model the system dynamics. It solves Newton's equations of motion for the complex multi-contact dynamics problem:

$$\boldsymbol{x}_{n+1} = F(\boldsymbol{x}_n, \boldsymbol{u}_n, \Delta t). \tag{6.1}$$

Normally, computing all states $\boldsymbol{x}_n$ happens in a serial fashion, by evaluating (6.1) first for $n = 0$, then for $n = 1$, etc. Instead, I replace this inherently serial procedure by a parallel-in-time integration process. Specifically, I adapt the Parareal algorithm for the contact-based manipulation problem.

[1] I use Mujoco since it is recently the most widely used physics-engine for model-based planning [79, 70, 25, 2].



(a) Initial coarse physics predictions with C

(b) Fine physics predictions with F in *parallel*

(c) Parareal update at time $n = 1$

(d) Parareal update at time $n = 2$

(e) Updates after 1 Parareal iteration

(f) Fine physics predictions for Parareal iteration 2

Figure 6.2: Combining coarse and fine physics with the Parareal algorithm (a) Initial coarse physics predictions across time with a cheap model (C), (b) Fine physics predictions in *parallel* starting from coarse initial guesses with F. (c) A Parareal update at time $n = 1$ as a linear combination of coarse and fine approximations of the state (d) A Parareal update at time $n = 2$ using the updated state at time $n = 1$, (e) Final trajectory updates after $k = 1$ Parareal iteration, (f) Next Parareal iteration begins with fine physics predictions in *parallel*.

Parareal begins with a rough initial guess of the state at each time point n of the trajectory as shown in Fig. 6.2a. To get an initial guess, I define a second, coarse physics model:

$$\boldsymbol{x}_{n+1} = C(\boldsymbol{x}_n, \boldsymbol{u}_n, \Delta t) \tag{6.2}$$

It needs to be computationally cheap relative to the fine model but does not need to be very accurate.

The next step is to evaluate the fine physics model in *parallel* starting from N initial guesses as shown in Fig. 6.2b. Thereafter, I do a coarse sweep across the time points. I start from the initial state $\boldsymbol{x}_0$ and make a coarse prediction for the next state $\boldsymbol{x}_1$ (dotted lines in Fig. 6.2c). Now, I have 3 approximations of the state at time $n = 1$. I linearly combine these approximations to get an update for $\boldsymbol{x}_1$ (in green). Then starting from this new update for $\boldsymbol{x}_1$, I make a coarse prediction for the next state $\boldsymbol{x}_2$ (dotted lines in Fig. 6.2d at $n = 2$). I combine the three approximations to get an update at $n = 2$. I continue this coarse sweep for all time points to get the updated trajectory in Fig. 6.2e. This is the end of the first iteration. I then repeat the whole process iteratively using new updates as initial guesses as shown in Fig. 6.2f.

In summary, Parareal starts by computing rough initial guesses $\boldsymbol{x}_n^{k=0}$ of the system states using the coarse model. The newly introduced superscript k

counts the number of Parareal iterations. In each Parareal iteration, the guess is then refined via

$$\boldsymbol{x}_{n+1}^{k+1} = C(\boldsymbol{x}_n^{k+1}, \boldsymbol{u}_n, \Delta t) + F(\boldsymbol{x}_n^k, \boldsymbol{u}_n, \Delta t) - C(\boldsymbol{x}_n^k, \boldsymbol{u}_n, \Delta t), \tag{6.3}$$

for all timesteps $n = 0, \ldots, N-1$. The key point in iteration (7.4) is that evaluating the fine physics model can be done in parallel for all $n = 0, \ldots, N-1$, while only the fast coarse model has to be computed serially.

Parareal iterations converge exactly to the fine physics solution after $k = N$ iterations. After one iteration, $\boldsymbol{x}_1^1$ is exactly the fine solution. I can see this in Fig. 6.2c where the two coarse physics predictions in Eq. 7.4 are same and cancel out. After two iterations, $\boldsymbol{x}_1^1$ and $\boldsymbol{x}_2^2$ are exactly the fine solutions and so on.

The idea is to stop Parareal at much earlier iterations such that it requires significantly less wall-clock time than running F serially step-by-step. To do this, C must be computationally much cheaper than F.

Parareal can be thought of as producing a spectrum of solutions increasing in accuracy and computational cost, from the cheap coarse physics model to the expensive fine physics model — i.e. the N different approximations after each iteration. An important question is which of these models to choose; i.e. how many iterations of Parareal to use? To decide on the required prediction accuracy, I rely on recent work which analyzes the stochasticity in real-world pushing [97, 13]. I propose to stop Parareal when the approximation error with respect to the fine model is below the real-world pushing stochasticity.

Note that, for the sake of simplicity, I assume here that the number of controls N and the number of processors used to parallelize in time are identical.

6.1.1 Expected speedup performance of Parareal

I can describe the expected performance of Parareal by a simple theoretical model [72]. Let c_c and c_f be the time needed to compute the coarse physics model C and the fine physics model F respectively, for a duration of Δt. The speedup of Parareal s_p over the serial fine model is approximately:

$$s_p = \frac{N \cdot c_f}{(1+K) \cdot N \cdot c_c + K \cdot c_f} = \frac{1}{(1+K)\frac{c_c}{c_f} + \frac{K}{N}} \tag{6.4}$$

This illustrates the importance of finding a cheap coarse model that minimizes the ratio c_c/c_f. In that case, speedup will be determined mainly by the number of iterations K. For example, for a coarse model with negligible cost, after

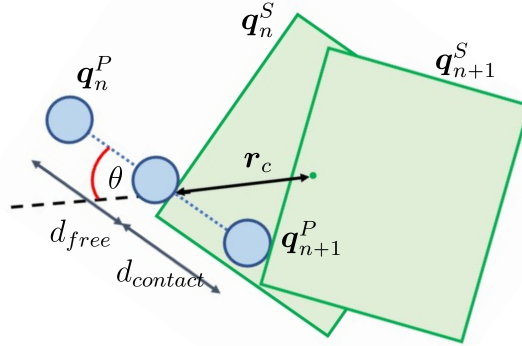


Figure 6.3: Coarse physics model

$K = 1$ Parareal iteration with $N = 4$ sub-intervals, then the theoretical speedup would be $s_p \sim N/K = 4$. That is, I can expect to make physics predictions about *four times faster* than using only the fine physics model in serial.

6.1.2 Coarse physics model

As a case study, I consider the challenging problem of pushing an object. I seek a general coarse physics model with the following requirements:

- It must be significantly cheaper to compute with respect to the fine model.
- It must provide a physics prediction for all possible pusher motions but can be inaccurate.
- It must provide a prediction for sliders of any shape and inertial parameters.

Instead of solving Newton's equation of motion for the multi-contact dynamics problem, I propose a simple kinematic pushing model $C(\boldsymbol{x}_n, \boldsymbol{u}_n, \Delta t)$. It moves the slider with the same linear velocity as the pusher, as long as there is contact between the two. I also apply a rotation to the slider, based on the position and direction of the contact, with respect to the center of the object. Formally, given the linear velocity of the pusher as the controls $\boldsymbol{u}_n = [u_n^x, u_n^y]^T$, the next state of the system is given by;

$$\boldsymbol{q}_{n+1}^S = \boldsymbol{q}_n^S + [u_n^x, u_n^y, \omega]^T \cdot p_c \cdot \Delta t \quad (6.5)$$

$$p_c = \frac{d_{contact}}{d_{contact} + d_{free}}, \quad \omega = K_\omega \cdot \frac{||\boldsymbol{u}_n|| \cdot \sin\theta}{||\boldsymbol{r}_c||} \tag{6.6}$$

$$\dot{\boldsymbol{q}}^S_{n+1} = \{[u^x_n, u^y_n, \omega]^T \ \ if \ \ p_c > 0, \ \ \dot{\boldsymbol{q}}^S_n \ \ otherwise\} \tag{6.7}$$

$$\boldsymbol{q}^P_{n+1} = \boldsymbol{q}^P_n + \boldsymbol{u}_n \cdot \Delta t, \quad \dot{\boldsymbol{q}}^P_{n+1} = \boldsymbol{u}_n. \tag{6.8}$$

In Eq. 6.5 the slider's pose is updated as described above. Here, p_c is the ratio of the distance $d_{contact}$ travelled by the pusher when in contact with the slider and the total pushing distance as shown in Fig. 6.3. $\boldsymbol{r}_c$ is a vector from the contact point to the object's center (green dot) at the current state $\boldsymbol{q}^S_n$, θ is the angle between the pushing direction and the vector $\boldsymbol{r}_c$. Moreover, ω is the coarse angular velocity induced by the pusher on the slider, where K_ω is a positive constant parameter.

Also note that, even though Fig. 6.3 shows the pusher and slider in contact at the next time step, this does not have to be so; i.e. the coarse model can leave the two in separation.

In Eq. 6.7 the velocity of the slider is updated to be the same as the current pusher velocity if there is any contact. In Eq. 6.8 the pusher position and velocity are updated.

6.1.3 Infeasible states

The new iterate $\boldsymbol{x}^{k+1}_{n+1}$ given by the Parareal iteration (Eq. 7.4) can be an infeasible state where the pusher and slider penetrate each other. Contact dynamics is not well-defined for such states. It can lead to infinitely large object accelerations and an unstable fine physics model. I have not encountered such a problem of infeasible (or unallowed) states in other dynamics domains that Parareal has been applied to.

To handle these cases, I project the infeasible state to the nearest feasible state. I write the following optimization problem:

$$\boldsymbol{q}^{S*}_{n+1} = \arg\min_{\boldsymbol{q}^S_{n+1}} ||\boldsymbol{q}^S_{n+1} - \boldsymbol{q}^{S_{infeasible}}_{n+1}||, \quad s.t. \ \ d_p \leqslant 0 \tag{6.9}$$

where $\boldsymbol{q}^{S_{infeasible}}_{n+1}$ is the infeasible slider's pose, and d_p is the penetration depth. The goal is to find the nearest slider pose $\boldsymbol{q}^{S*}_{n+1}$ that satisfies the no-penetration constraint $d_p \leqslant 0$.

I can use an off-the-shelf optimizer to find a solution rather efficiently. However, for simple systems I can analytically find the penetration depth and move the slider along the contact normal to resolve penetration.

In Sec. 6.3.2, I evaluate the open-loop pushing performance of our hybrid physics models. Our goal is to use these hybrid models for planning and control.

6.2 Push Planning and Control

I use the predictive models described above in a planning and control framework for pushing an object on a table to a goal location, avoiding a static obstacle. This task retains many challenges of general robotic control through contact such as impulsive contact forces, under-actuation and hybrid dynamics (separation, sticking, sliding, e.t.c.). I present an example scene and execution in Fig. 6.4.

To solve this problem, I take an optimization approach. Given the obstacle and goal position and geometry, the current state of the pusher and slider $\boldsymbol{x}_0$, and an initial candidate sequence of controls $\{\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{N-1}\}$, the optimization procedure outputs an optimal sequence $\{\boldsymbol{u}_0^*, \boldsymbol{u}_1^*, \ldots, \boldsymbol{u}_{N-1}^*\}$ according to some defined cost. I explain this optimization process, and the cost formulation that is optimized, below (Sec. 6.2.1).

The predictive models that I have developed earlier in the paper are used within this optimizer to *roll-out* a sequence of controls, to predict the states $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, which are then used to compute the cost associated with those controls.

Once the optimization produces a sequence of controls, I use it in a model-predictive-control (MPC) fashion, by executing only the first control in the sequence. Afterwards, I update $\boldsymbol{x}_0$ with the observed state of the system, and repeat the optimization to generate a new control sequence. This is repeated until task completion. I consider the task completed if the slider reaches the goal region (success), if it hits the obstacle (failure), if it falls off the edge of the table (failure) or if a maximum number of controls are executed before failure occurs.

When I repeat the optimization within MPC, I warm-start it by using the previously optimized control sequence as the initial candidate sequence. For the very first optimization, the initial candidate sequence is generated as a straight line push towards the goal (which collides with the obstacle in all our scenes).

Such an optimization-based MPC approach to pushing manipulation is frequently used [11, 41, 54, 3]. Here, our focus is to evaluate the performance of

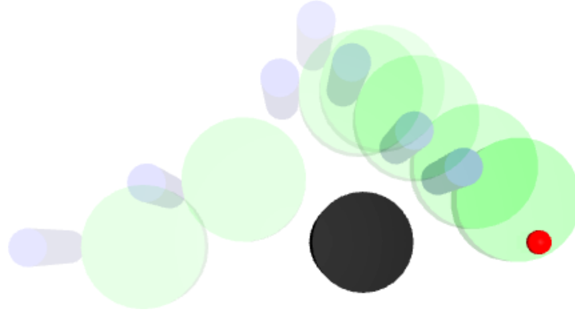


Figure 6.4: Push planning with a hybrid physics model to avoid an obstacle (in black) while pushing the cylindrical slider to a goal location (in red).

different predictive physics models described before in the paper within such a framework.

6.2.1 Trajectory Optimization

In this section I use the shorthand $\boldsymbol{u}_{0:N-1}$ to refer to the control sequence $\{\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{N-1}\}$. Similarly for states I use $\boldsymbol{x}_{0:N}$.

Our goal is to find an optimal control sequence $\boldsymbol{u}^*_{0:N-1}$ for a planning horizon N, given an initial state $\boldsymbol{x}_0$, and an initial candidate control sequence $\boldsymbol{u}_{0:N-1}$.

I define the cost function J, for a given control sequence and the corresponding state sequence:

$$J(\boldsymbol{x}_{0:N}, \boldsymbol{u}_{0:N-1}) = \sum_{n=1}^{N-1} J_n(\boldsymbol{x}_n, \boldsymbol{u}_{n-1}, \boldsymbol{u}_n) + w \cdot J_N(\boldsymbol{x}_N)), \tag{6.10}$$

where J_n is the running cost at each step, w is a positive weighting constant, J_N is the terminal (final) cost function.

The output of optimization is the minimizing control sequence:

$$\boldsymbol{u}^*_{0:N-1} = \underset{\boldsymbol{u}_{0:N-1}}{\arg\min}\, J(\boldsymbol{x}_{0:N}, \boldsymbol{u}_{0:N-1}), \quad s.t. \;\; \boldsymbol{x}_{n+1} = f(\boldsymbol{x}_n, \boldsymbol{u}_n, \Delta t), \;\; \boldsymbol{x}_0 \;\; fixed. \tag{6.11}$$

Here, f is the system dynamics constraint that must be satisfied at all times.

I define the running cost for our pushing around an obstacle problem as:

$$J_n(\boldsymbol{x}_n, \boldsymbol{u}_{n-1}, \boldsymbol{u}_n) = w_s \cdot (1/||[q^{s_x}, q^{s_y}]^T - \boldsymbol{p}_{obs}||^2) + w_p \cdot (1/||\boldsymbol{q}^p - \boldsymbol{p}_{obs}||^2) + w_u \cdot ||\boldsymbol{u}_t - \boldsymbol{u}_{t-1}||^2 + W_E$$

where w_s, w_p, w_u are positive constant weights. $\boldsymbol{p}_{obs}$ is the position vector of the static obstacle to be avoided, and $[q^{s_x}, q^{s_y}]$ are the x,y positions of the

slider respectively. The above formula associates high cost for the slider or the pusher to approach the obstacle. It has a smoothness cost, to prevent high accelerations. Additionally, it has a constant edge cost W_E for the slider falling off the table.

I define the final cost J_N as: $J_N(\boldsymbol{x}_n) = ||[q^{s_x}, q^{s_y}]^T - \boldsymbol{p}_{goal}||^2$, where $\boldsymbol{p}_{goal}$ is the position vector of the target/goal location.

There exists different optimization methods to solve this problem [61, 49, 94, 41, 3]. The main difference lies in the way the cost gradient is computed for a given sequence of controls. For ease of implementation, here I use derivative-free stochastic sampling-based methods [49, 94, 3]. Particularly, I use the algorithm Chapter 5. In each optimization iteration, to find the cost gradient at the current control sequence, these stochastic sampling methods generate multiple noisified versions of the current control sequence, they roll-out these noisy controls to find the cost associated with each one, and use these costs to compute a numerical gradient, which is then used to update the control sequence to minimize the cost. The roll-out of these noisy control sequences to compute the resulting states and the cost is where I use the physics models.

6.2.2 Parareal and MPC

The Parareal framework for generating hybrid physics models yields itself well to model-predictive control. Recall from Fig. 6.2e that after 1 Parareal iteration, physics prediction for the first state $\boldsymbol{x}_1$ is exactly the same as the fine model. This means planning is accurate at least for the first action for all our hybrid models. This aligns well with our MPC framework since I execute only the first action and then re-plan.

6.3 Experiments and Results

In our experiments, I address three key issues. First, I investigate how fast Parareal converges to the fine physics solution for pushing tasks. Second, I investigate the open-loop pushing performance of different physics models and compare it with real-world data. Finally, I investigate how the different physics models generated by Parareal (at different iterations) can be used within a planning and control framework to complete non-prehensile manipulation tasks.

The goal of Parareal for push planning is to simulate physics faster than any given fine physics model. Therefore, I must stop Parareal at much earlier iterations to achieve meaningful speedup. However, I must also understand how different Parareal's predictions are with respect to the fine physics predictions

at different iterations. I investigate Parareal's convergence for specific pushing examples in Sec. 6.3.1.1. In addition, in Sec. 6.3.1.2, I measure the empirical speedup I get from Parareal, and compare it with the theoretical speedup that was presented in Sec. 6.1.1.

During push planning and control, I must decide on a Parareal iteration that gives us an acceptable approximation to the fine solution. To this end, in Sec 6.3.2, I conduct open-loop pushing experiments. I statistically investigate Parareal's approximation error with respect to the fine solution, for a particular number of Parareal iterations. To do that, I start from a large number of random initial states and apply different control sequences open-loop. I then analyze how these statistical approximation errors compare with the standard deviation of the real-world uncertainty during similar pushing tasks [97].

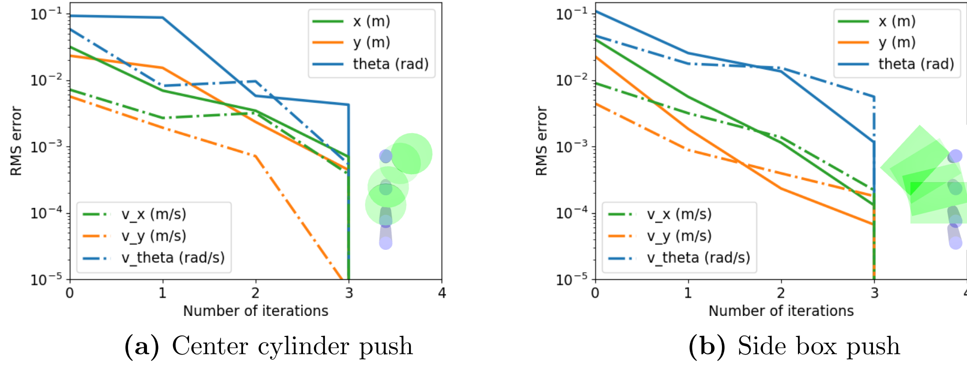
In Sec 6.3.3 I investigate the performance of different physics models produced by Parareal, when used within the MPC framework described in Sec. 6.2 to push an object to a goal region, avoiding an obstacle. I compare the success rates and total task completion times for the different physics models (the coarse model, Parareal at different iterations, and the fine model). I perform these experiments on a real robot setup Sec 6.3.3.

6.3.1 Parareal convergence for pushing

6.3.1.1 Parareal convergence for specific pushing examples:

I consider simulating the results of applying a control sequence starting from an initial state for a box and a cylinder as shown in Fig. 6.5. I consider four cases: pushing a cylinder from the center (Fig. 6.5a), pushing a cylinder from the side, pushing a box from the center, and finally pushing a box from the side (Fig. 6.5b). The control sequence used here is $\boldsymbol{u}_{0:3} = \{[25,0],[25,0],[25,0],[25,0]\}mm/s$ where each control input is applied for a control duration $\Delta t = 1s$ such that the total pushing distance is $100mm$.

I use the Mujoco [92] physics engine as the fine physics model. To make the fine model as fast as possible, I run it at the largest possible simulation time-step ($1ms$) for our model. Beyond this time-step, the physics engine becomes unstable and breaks down. In addition, all experiments are run on a desktop computer (Intel(R) Xeon(R) CPU E3-1225 v3) with $N = 4$ cores. At each iteration of Parareal, I calculate the root mean square (RMS) error between Parareal's predictions and the physics engine's predictions of the corresponding sequence of states. These RMS errors can be seen in Fig. 6.5 for two different cases. The results are similar for others. Note that the errors are given in log



(a) Center cylinder push

(b) Side box push

Figure 6.5: Root mean square error (in log scale) along the full trajectory for pushing a cylinder (a) and box (b) from the center and side respectively, for increasing Parareal iterations. The motions are illustrated lower-right in each plot.

scale for the full state of the slider (pose and velocities). In general, I see a quick decrease in the error along the full trajectory starting from the large error of the coarse model at iteration 0. In addition, at the final iteration, I verify that the Parareal solution is exactly the same as using the fine model since the errors go to 0.

6.3.1.2 Parareal speedup for specific pushing examples:

Using each hybrid physics model, I repeatedly predict the sequence of states (which is deterministic for a given physics model) and record the total time it takes (which varies slightly depending on computer load). In Fig. 6.6, I see the average prediction time over 100 runs for each physics model for a box side push. These actual prediction times are close to the expected prediction time (Eq. 6.4) for the different physics models. For example, at 1 Parareal iteration I spent 28% of the time spent by the full physics engine, i.e. about *four times faster*. The results for the cylinder side push, cylinder center push and box center push are similar to those in Fig. 6.6.

6.3.2 Open-loop pushing experiments

I compare the predictions of different physics models (Parareal iterations) for open-loop pushing. I start from 100 randomly sampled initial states[2]. At each initial state, I used three different control sequences, giving 300 different slider trajectories. The three control sequences $\boldsymbol{u}^{\{1,2,3\}}$ that I used at each initial state

[2]I change the pusher's position along the direction perpendicular to the direction of motion. I sample uniformly within the edges of the slider (rectangle).

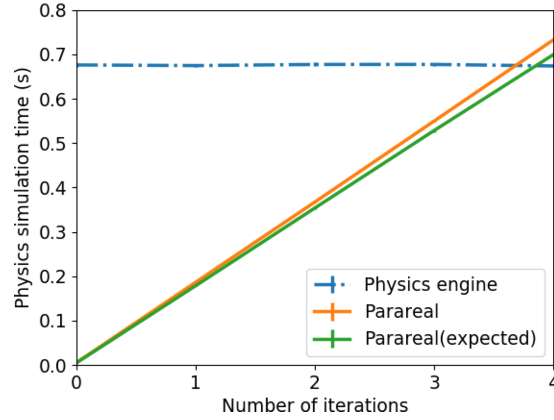


Figure 6.6: Physics simulation time averaged over 100 runs for a box side push within 95 % confidence interval of the mean.

were fixed, and are given by:

$$\boldsymbol{u}^i_{0:3} = v_c \cdot \{v, v, v, v\}, \quad v = [\cos(\alpha_i), \sin(\alpha_i)], \quad \alpha_i = \{0^{\circ}, 15^{\circ}, -15^{\circ}\}, i \in 1, 2, 3 \tag{6.12}$$

v_c is a constant pushing speed:$v_c = 25mm/s$.

I applied each control input for a duration of $\Delta t = 1.5s$ such that the total distance travelled by the pusher is $150mm$ in all cases. I calculated for each physics model (Parareal iteration), the RMS difference of the final state (in comparison with the final state prediction of the fine physics model) for the 300 trajectories.

Our results are shown in Table. 6.1. I see that on the average the coarse physics model is quite inaccurate but with increasing Parareal iterations, the mean difference from the fine physics model goes to zero. However, to decide on how much error with respect to the physics engine is appropriate for pushing tasks, I look at the real-world's uncertainty for pushing dynamics.

Yu et al. [97] provide real-world pushing data for a similar pusher-slider system. Starting at the same initial state, they push a box repeatedly in the real-world with a cylindrical pusher and record the resulting final positions. The pushing distance is $150mm$ with a quasi-static pushing speed of $20mm/s$. As shown in Table 6.1, they record a translation standard deviation of $8.10mm$ and a rotation standard deviation of 4.20° on a plywood surface. Notice that for Parareal after 2 iterations, I see a mean translation difference of $6.39mm$ and a mean rotation difference of 3.82° when compared to the fine model (physics engine) predictions.

I conclude that, for real-world purposes, it should not be necessary to run Parareal for more than 2 iterations, as approximating the physics engine more

Table 6.1: Open-loop pushing

	Mean trans. diff. (mm)	Mean rot. diff. (deg)
Coarse Physics	62.67	17.63
Parareal-1 iter.	28.43	6.30
Parareal-2 iter.	6.39	3.82
Parareal-3 iter.	2.47	0.79
Parareal-4 iter.	0.00	0.00
	Trans. std. (mm)	Rot. std. (deg)
Push dataset [97]	8.10	4.20

accurately than the inherent uncertainty in real-world pushing should not contribute to real-world performance. Note that, 2 Parareal iterations here corresponds to a) exactly the fine physics predictions for your first two actions and b) a model that is two times faster than the physics engine.

6.3.3 Push planning and control with hybrid physics models

I measure the performance of the different physics models when used within the optimization and MPC framework, described in Sec. 6.2, to push an object to a goal region, while avoiding an obstacle.

Our real robot setup is shown in Fig. 6.8 where I have a Robotiq two-finger gripper holding the cylindrical pusher. I place markers on the pusher and slider to sense their full pose in the environment with the OptiTrack motion capture system. I consider 5 randomly generated scenes (one shown in Fig. 6.8) where the pusher must avoid the obstacle at the center of the table before bringing the slider to the goal location. For each of the 5 scenes, I used the coarse physics model, Parareal iterations (1,2, and 3), and the full physics engine for push planning and control. That is a total of 25 planning and control runs with the real robot. I plan using our various physics models and execute actions in the real world in an MPC fashion. For the stochastic trajectory optimizer, our control sequences contain 4 control inputs each applied for a control duration of $\Delta t = 1s$. In addition, I use 20 noisy control sequence samples (as explained in Sec. 6.2.1) per optimization iteration for the trajectory optimizer with an exploration variance of 10^{-4}. Normally, each noisy control sequence of the optimizer is rolled out independently. However, since I use a standard quad-core desktop PC, the *parallelization is across time only.*

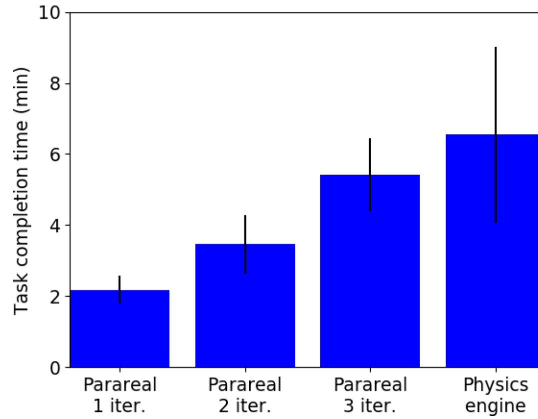


Figure 6.7: Total task completion time (within 95 percent confidence interval of the mean) for push planning with obstacle avoidance using different physics models for 100 randomly sampled initial states.

As the pusher attempts to bring the slider to a desired goal location, there are three possible failure modes. First, I declare failure when the slider collides with the static obstacle. Second, I declare failure when the pusher is unable to bring the slider to the goal location after executing 20 actions (5 times the number of actions in a given control sequence). Third, I declare failure when the slider falls off the edge of the table.

For each hybrid physics model, I achieved 100% success rate on the real robot. This is same as using a full physics engine, only significantly faster Fig. 6.7. For instance, using 1 Parareal iteration, I can complete the push planning task about *four times faster* than the physics engine. Note that the planning times here can be reduced by parallelizing the stochastic trajectory optimizer on a PC with more cores. Currently, the 4 cores of our PC is used to *parallelize across time*. Furthermore, in all the 5 scenes considered, the robot was unable to complete the push planning task by using only the coarse model. I present snapshots from the experiments in Fig. 6.8.

6.4 Discussion

This chapter introduces a method to combine coarse and fine physics models for manipulation planning and control, using parallel-in-time integration. It introduces a coarse physics model and combines it with a physics engine as fine model to speed-up physics predictions. I showed faster physics-based robotic manipulation planning and control with hybrid models where task success rates weren't sacrificed. I considered a simplified manipulation task of single object pushing, which however still retains many of the challenges of contact-based manipulation. Multi-object contact scenarios are addressed in the next chapter.

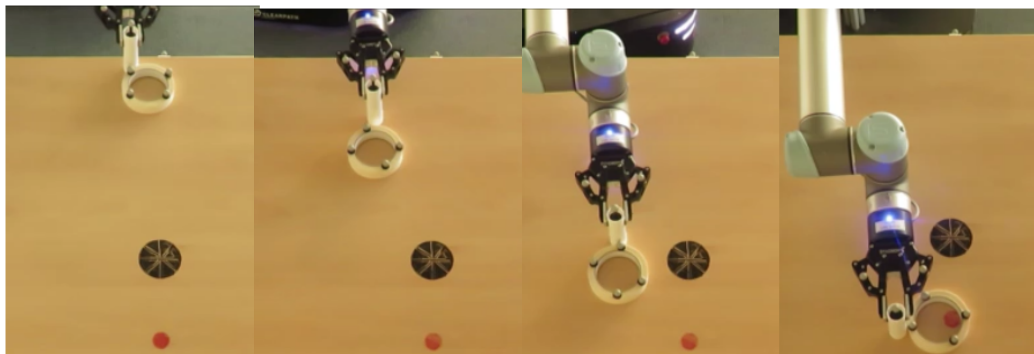


Figure 6.8: Pushing with a hybrid physics model. I complete the push planning task about **four times faster** than a physics engine.

Another important point is the convergence of Parareal. Faster convergence can result in faster physics predictions. Parareal's convergence can be improved with a better coarse physics model as we demonstrate in the next Chapter.

Chapter 7

A Learned Coarse Model for Robotic Manipulation using Parareal

In Chapter 6, I demonstrated that physics predictions for a robot pushing a single object can be made faster by combining a fine physics-based model with a simple, coarse physics-based model using the parallel-in-time method Parareal. Using 4 cores, Parareal was about a factor two faster than the fine physics engine alone while providing comparable accuracy and the same success rate for a push planning problem with obstacle avoidance.

In this chapter, I extend these results by investigating a deep neural network as coarse model and show that it leads to faster Parareal convergence. I also demonstrate that Parareal can be used to speed up physics prediction in scenarios where the robot pushes multiple objects, such as in Fig. 7.1.

7.1 Robotic Manipulation with Parareal

7.1.1 Robotic manipulation

Consider the scene shown in Figure 7.1. The robot's manipulation task is to control the motion of the green goal object through pushing contact from the cylindrical pusher in the robot's gripper. The robot needs to push the goal object into a goal region marked with an X. It is allowed to make contact with other sliders but not to push them off the table or into the goal region.

The system's state at time point n consists of the pose $\boldsymbol{q}$ and velocities, $\dot{\boldsymbol{q}}$ of the pusher P and N_s sliders, $S^i \ldots S^{N_s}$:

$$\boldsymbol{x}_n = [\boldsymbol{q}_n^P, \boldsymbol{q}_n^{S^i}, \ldots, \boldsymbol{q}_n^{S^{N_s}}, \dot{\boldsymbol{q}}_n^P, \dot{\boldsymbol{q}}_n^{S^i}, \ldots, \dot{\boldsymbol{q}}_n^{S^{N_s}}].$$

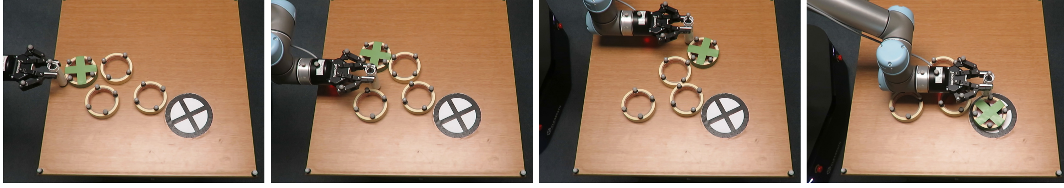


Figure 7.1: Example of a robotic manipulation planning and control task using physics predictions. The robot controls the motion of the green object solely through contact. The goal is to push the green object into the target region marked X. The robot must complete the task without pushing other objects off the table or into the goal region.

The pose of slider i consists of its position and orientation on the plane: $\boldsymbol{q}^{S^i} = [q^{S^i_x}, q^{S^i_y}, q^{S^i_\theta}]^T$. The pusher's pose is $\boldsymbol{q}^P = [q^{P_x}, q^{P_y}]^T$ and control inputs are velocities $\boldsymbol{u}_n = [u^x_n, u^y_n]^T$ applied on the pusher at time n for a control duration of Δt.

A robotics planning and control algorithm takes in an initial state of the system $\boldsymbol{x}_0$, and outputs an optimal sequence of controls $\{\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{N-1}\}$. However, to generate this optimal sequence, the planner needs to simulate many different control sequences and predict many resulting sequences of states $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$.

The planner makes these simulations through a physics model F of the real-world that predicts the next state $\boldsymbol{x}_{n+1}$ given the current state $\boldsymbol{x}_n$ and a control input $\boldsymbol{u}_n$

$$\boldsymbol{x}_{n+1} = F(\boldsymbol{x}_n, \boldsymbol{u}_n, \Delta t). \tag{7.1}$$

I use the general physics engine Mujoco [92] to model F. It solves differential algebraic equations of motion for the complex multi-contact dynamics problem

$$\begin{aligned} M(\boldsymbol{q})\, d\boldsymbol{v} = (\boldsymbol{b}(\boldsymbol{q}, \boldsymbol{v}) + \tau)\, dt + J_E(\boldsymbol{q})^T \boldsymbol{f}_E(\boldsymbol{q}, \boldsymbol{v}, \tau) \\ + J_C(\boldsymbol{q})^T \boldsymbol{f}_C(\boldsymbol{q}, \boldsymbol{v}, \tau) \end{aligned} \tag{7.2}$$

where $\boldsymbol{q}$, $\boldsymbol{v}$, and M are position vector, velocity vector, and inertia matrix respectively in generalized coordinates. $\boldsymbol{b}$ contains bias forces (Coriolis, gravity, centrifugal, springs), $\boldsymbol{f}_E$ and $\boldsymbol{f}_C$ are impulses caused by equality constraints and contacts respectively and J_E and J_C are the corresponding Jacobians and τ are external/applied forces. The equations are then solved numerically. Mujoco obtains a discrete-time system with two options for integrators — semi-implicit Euler or 4th order explicit Runge-Kutta.

7.1.2 Parareal

Normally, computing all states $\boldsymbol{x}_n$ happens in a serial fashion, by evaluating (7.1) first for $n = 0$, then for $n = 1$, etc. Parareal replaces this inherently serial procedure by a parallel-in-time integration process where some of the work can be done in parallel.

It requires a coarse physics model:

$$\boldsymbol{x}_{n+1} = C(\boldsymbol{x}_n, \boldsymbol{u}_n, \Delta t). \tag{7.3}$$

and it predicts states using the following equation:

$$\boldsymbol{x}_{n+1}^{k+1} = C(\boldsymbol{x}_n^{k+1}, \boldsymbol{u}_n, \Delta t) + F(\boldsymbol{x}_n^{k}, \boldsymbol{u}_n, \Delta t) - C(\boldsymbol{x}_n^{k}, \boldsymbol{u}_n, \Delta t), \tag{7.4}$$

for all timesteps $n = 0, \ldots, N-1$. The newly introduced superscript k counts the number of Parareal iterations. The key point in Eq. (7.4) is that evaluating the fine physics model can be done in parallel for all $n = 0, \ldots, N-1$, while only the fast coarse model has to be computed serially. Please see Chapter 6 for the complete detail on Parareal.

7.2 Coarse models

In this section, I introduce a learned coarse model and briefly summarize the analytical coarse model from Chapter 6.

7.2.1 Learned coarse model

As an alternative to the coarse physics model, I train a deep neural network as a coarse model for Parareal for robotic pushing.

7.2.1.1 Network architecture

The input to our neural network model is a state $\boldsymbol{x}_n$ and a single action $\boldsymbol{u}_n$. The output is the change in state $\Delta\boldsymbol{x}$ which is added to the input state to obtain the next state $\boldsymbol{x}_{n+1}$. The dimensions of the input is $2(3N_s{+}2)$ $+2$, and the dimension of the output is $2(3N_s{+}2)$. I use a feed-forward deep neural network (DNN) with 5 fully connected layers. The first 4 contain 512, 256, 128 and 64 neurons, respectively, with ReLU activation function. The output layer contains 24 neurons with linear activation functions.

7.2.1.2 Dataset

I collect training data using the physics engine Mujoco [92]. Each training sample is a tuple $(\boldsymbol{x}_n, \boldsymbol{u}_n, \boldsymbol{x}_{n+1})$. It contains a randomly[1] sampled initial state, action, and next state. I collect over 2 million such samples from the physics simulator.

During robotic pushing, a physics model may need to predict the resulting state even for cases when there is no contact between pusher and slider. I include both contact and no-contact cases in the training data.

I train a single neural network to handle one pusher with at least one and at most N_s objects being pushed (also called sliders). While collecting data for a particular number of sliders, I placed the unused sliders in distinct fixed positions outside the pushing workspace. These exact positions must be passed to the neural network at test time if fewer than N_s sliders are active. For example, if $N_s = 4$, to make a prediction for a 3 slider scene, I place the last slider at the same fixed position used during training.

7.2.1.3 Loss function

The standard loss function for training is the mean squared error between the network's prediction and the training data. On its own, this leads to infeasible state predictions where there is pusher-slider or slider-slider penetration. I resolve this by adding a no penetration loss term such that the final loss function reads:

$$
\begin{aligned}
f_l = W_F \cdot \sum_{i=1}^{N_s} \sum_{j=i+1}^{N_s} \min(||\boldsymbol{p}_i^{NN} - \boldsymbol{p}_j^{NN}|| - (r_i + r_j), 0)^2 \\
+W_F \cdot \sum_{i=1}^{N_s} \min(||\boldsymbol{p}_P - \boldsymbol{p}_i^{NN}|| - (r_p + r_i), 0)^2 \\
+||\boldsymbol{x}^f - \boldsymbol{x}^{NN}||^2.
\end{aligned}
\tag{7.5}
$$

Here, W_F is a constant weight, $\boldsymbol{x}^f$ is the next state predicted by the fine model, $\boldsymbol{x}^{NN}$ is the next state predicted by the DNN model. $\boldsymbol{p}_i^{NN}$ and $\boldsymbol{p}_j^{NN}$ are the new positions of sliders i and j predicted by the DNN model, respectively, and $\boldsymbol{p}_P$ is the position of the pusher. r_p is the radius of the pusher, and r_i, r_j represent the radius of sliders i and j, respectively. The first line of Equation 7.5 penalizes slider-slider penetration, the second line penalizes pusher-slider penetration, and the third line is the standard mean squared error.

[1] I use rejection sampling to ensure that sampled states do not have objects in penetration, i.e. fulfill the algebraic constraints of Eq. 7.2.

Finally, the network makes a single step prediction. However, robotic manipulation typically needs a multi-step prediction as a result of a control sequence. To do this, I start from the initial state and apply the first action in the sequence to get a resulting next state. Then, I use this next state as a new input to the network together with the second action in the sequence and so on. This way, I repeatedly query the network with its previous predictions as the current state input.

7.2.2 Analytical coarse model

In Chapter 6, I proposed a simple, kinematic coarse physics model for pushing a single object. The model moves the slider with the same linear velocity as the pusher as long as there is contact between the two. Further detail can be found in the Chapter 6.

7.3 Planning and control with hybrid models

I use the predictive model based on Parareal described above in a planning and control framework for pushing an object on a table to a target location. I take an optimization approach to solve this problem. Given the table geometry, goal position, the current state of the pusher and all sliders $\boldsymbol{x}_0$, and an initial candidate sequence of controls $\{\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{N-1}\}$, the optimization procedure outputs an optimal sequence $\{\boldsymbol{u}_0^*, \boldsymbol{u}_1^*, \ldots, \boldsymbol{u}_{N-1}^*\}$ according to some defined cost.

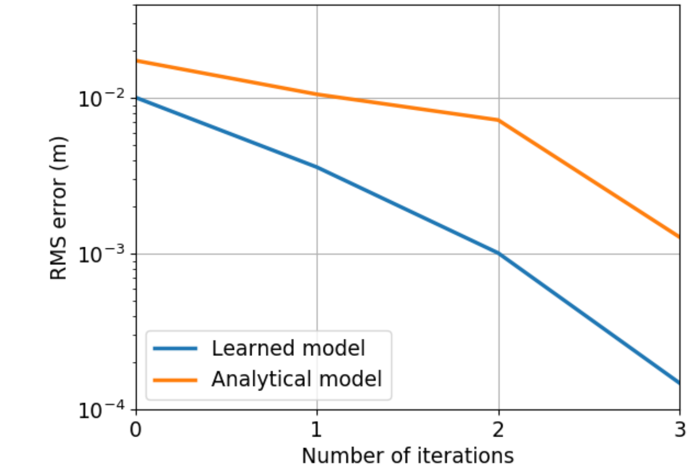
The predictive model is used within this optimizer to *roll-out* a sequence of controls to predict the states $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$. These are then used to compute the cost associated with those controls. The details of the exact trajectory optimizer are in Chapter 5. The cost function I use penalizes moving obstacle sliders and dropping objects from the table but encourages getting the goal object into the goal location.

I use the trajectory optimizer in a model-predictive control (MPC) framework. Once I get an output control sequence from the optimizer, I *do not* execute the whole sequence on the real-robot serially one after the other. Instead, I execute only the first action, update $\boldsymbol{x}_0$ with the observed state of the system, and repeat the optimization to generate a new control sequence. I repeat this process until the task is complete.

Such an optimization-based MPC approach to pushing manipulation is frequently used to handle uncertainty and improve success in the real-world [3, 11,

41, 54]. Here, our focus is to evaluate the performance of Parareal with learned coarse model for planning and control.

7.4 Experiments and Results



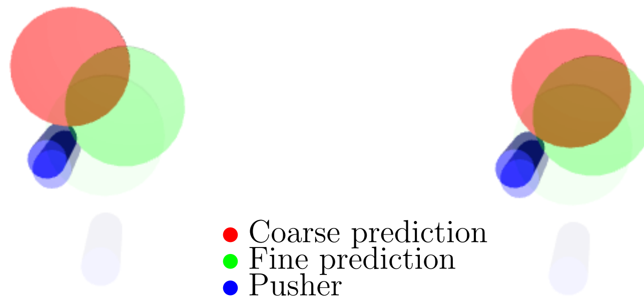


Figure 7.2: Root mean square error (in log scale) of Parareal along the full trajectory for single object pushing using both a learned and an analytical coarse model (top). These results are for a control sequence with 4 actions where the average object displacement is 0.043 ± 0.033 m. The error at iteration four is 0. The learned coarse model gives a better Parareal convergence rate. Sample motions for the learned coarse model (bottom, right) and the analytical coarse model (bottom, right). The learned coarse model's prediction is closer to the fine model prediction shown in green.

In our experiments, I investigate three key issues. First, I investigate how fast Parareal converges to the fine solution for robotic pushing tasks with different coarse models. Second, I investigate the physics prediction accuracy of

Parareal with respect to real-world pushing data. Finally, I demonstrate that the Parareal physics model can be used to complete real-robot manipulation tasks.

In Subsection 7.4.1 I provide preliminary information used throughout the experiments. Subsection 7.4.2 investigates convergence of Parareal for two different coarse models – the analytical coarse model for single object pushing and a learned coarse model for both single and multiple object pushing. In Subsection 7.4.3 I present results from real-robot experiments. First, I compare the accuracy of Parareal predictions against real-world pushing physics. Then, I show several real-robot plan executions using Parareal with a learned coarse physics model as predictive model.

7.4.1 Preliminaries

To generate physics-based robotic manipulation plans as fast as possible, I run Mujoco at the largest possible time-step (1ms) in all our experiments. Beyond this time-step the simulator becomes unstable, leading to unrealistically large object accelerations and breakdown of the simulator. I use the 4th order Runge-Kutta integrator for Mujoco. All computations run on a standard Laptop PC with an Intel(R) Core (TM) i7-4712HQ CPU @2.3GHz with $N = 4$ cores. Our control sequences consist of four or eight actions, each applied for a control duration $\Delta_t = 1s$.

The software version used to create training data and run experiments was Mujoco 2.00 with DeepMind DM Control bindings to Python 3.5 [91]. To develop, train and test the coarse model the Keras API was used, which is built in to TensorFlow 2.0. I used a learning rate of 5e-4 with 100 epochs and a batch size of 1024 to train the neural network model.

Our real robot setup is shown in Figure 7.1. I have a Robotiq two-finger gripper holding the cylindrical pusher of radius 1.45 cm. I place markers on the pusher and sliders to sense their full pose in the environment with an OptiTrack motion capture system. Sec. 7.1.1 states were defined to include orientation of objects but, to keep experiments simple, I use cylindrical objects such that only positions play a major role. The slider radius used in all experiments is 5.12 cm.

7.4.2 Parareal convergence

Parareal produces the exact fine physics solution when the number of iterations is equal to the number of timeslices regardless of the coarse physics model [33,

62]. The convergence rate for scalar ordinary differential equations was theoretically shown to be superlinear on bounded intervals [33]. However, for the differential algebraic equations in Eq. 7.2 that describe the multi-contact dynamics problem, no such theoretical result exists and I study the convergence rate numerically.

I investigate through experiments how fast Parareal converges using two coarse models - the analytic model for single object pushing and the learned model for both single object and multi-object pushing. At each iteration, I compute a root mean square (RMS) error between Parareal's predictions and the fine model's predictions of the corresponding sequence of states. I compute the RMS error over only positions since I used cylindrical objects in all experiments.

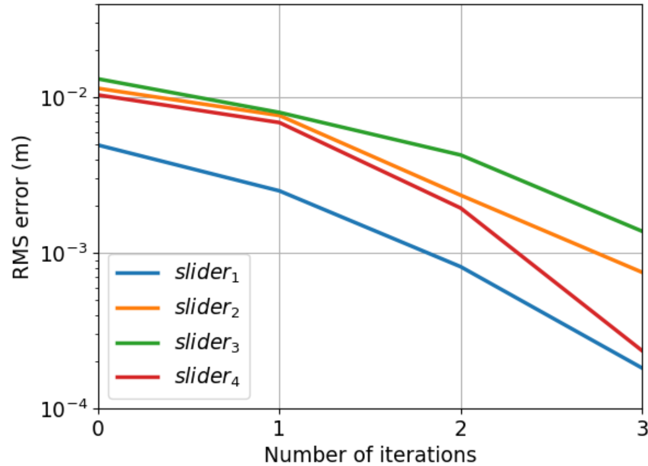
7.4.2.1 Single object pushing

I randomly sample an initial state for the pusher and slider. I also randomly sample a control sequence where the pusher contacts the slider at least once during execution. Thereafter, I execute the control sequence starting from the initial state using Parareal. For the sample state and control sequence, I perform two runs, one using the learned model and the other using the analytical model as coarse propagator in Parareal.

I collect 100 state and control sequence samples. The analytical model makes a single step prediction 227.1 times faster than the fine model on average, while the learned model is 228.4 times faster on average. For example, to predict a 4s long trajectory, the fine model requires 1.22s while one iteration of Parareal requires only 0.31s (for both models) on average. I see that both coarse models are so fast that our actual speedup in using Parareal is almost completely governed by the number of iterations.

Furthermore, for these samples, I also compute the RMS error between Parareal and the fine model run in serial. The results are shown in Fig. 7.2 (left) for a control sequence with 4 actions where the average object displacement is $0.043 \pm 0.033\ m$.

I see that the learned model leads to faster convergence of Parareal than the analytical model for single object pushing. One reason for this could be that, in general, more accurate coarse models lead to better convergence. The single-step prediction of the learned model, shown in read in Fig. 7.2 (right), is much closer to the fine prediction shown in green than the analytical model shown in Fig. 7.2 (center).



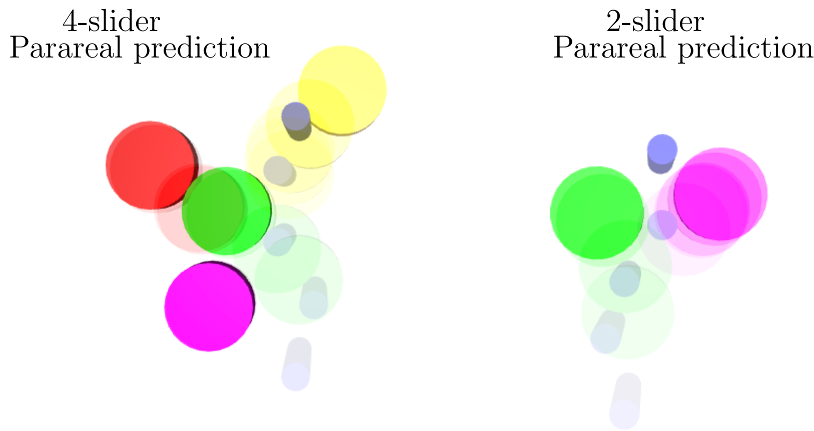


Figure 7.3: Root mean square error (in log scale) along the full trajectory per slider in a 4-slider pushing experiment (top) using *only* the learned model. Two sample motions are illustrated (bottom, left and right) for multi-object physics prediction. These results are for a control sequence with 4 actions where the average object displacement is $0.015 \pm 0.029\ m$. The error at iteration four is 0 except for accumulation of round-off errors. I find that the learned model enables Parareal convergence for the multi-object case.

7.4.2.2 Multi-object pushing

I randomly sample a valid initial state for the pusher and multiple sliders. Then, similar to the single object pushing case, I also sample a random control sequence that makes contact with at least one slider. I then predict the corresponding sequence of states using Parareal. However, for multi-object pushing I use only the learned model as the coarse physics model within Parareal. The analytical model for single-object pushing would need significant modifications to work for the multi-object case. Again, I collect 100 state and control sequence samples and run Parareal for each of them. Our results are shown in

Fig. 7.3.

Fig. 7.3 (left) shows the RMS error per slider for each Parareal iteration. While there are differences in the accuracy of the predictions for different slides, all errors decrease and Parareal converges at a reasonable pace.

These results are for a control sequence with 4 actions and where average object displacement is 0.015 ± 0.029 m. Some sample predictions are shown for a 4 slider environment in Fig. 7.3 (center), and for a 2-slider environment in Fig. 7.3 (right). In both scenes, the pusher moves forward making contact with multiple sliders and Parareal is able to predict how the state evolves.

I also investigate Parareal convergence for a longer control sequence of 8 actions. I do this for single object and multi-object pushing where all other conditions are the same as for the 4-action control sequence. Results can be found in Fig. 7.4 (left) for multi-object pushing and Fig. 7.4 (right) for single object pushing. The average object displacement for multi-object pushing is 0.034 ± 0.082 m and for single object pushing it is 0.046 ± 0.040 m. In general I find a similar convergence trend for both learned and analytical models for single and multi-object pushing.

Note that the shapes and sizes of the objects used are known and in fixed order. Therefore the learned model naturally does not generalize to new objects. However, it can still be used to make rather coarse predictions for similar objects.

7.4.3 Real robot experiments

In this section I investigate the physics prediction accuracy of Parareal with respect to real-world pushing physics. I do this for the multi-object case. In addition, I show real-world demonstrations for robotic manipulation where I use Parareal for physics prediction.

7.4.3.1 Parareal prediction vs. real-world physics

Our coarse model neural network was trained using simulated data. Here, I demonstrate that Parareal using the trained coarse model is also able to predict real-world states. I randomly set an initial state in a real-world example by selecting positions for the pusher and sliders. This state is recorded using our motion capture system. Next, I sample a control sequence and let the real robot execute it. Again, I record the corresponding sequence of states using motion capture. Then, for the recorded initial state and control sequence pair,

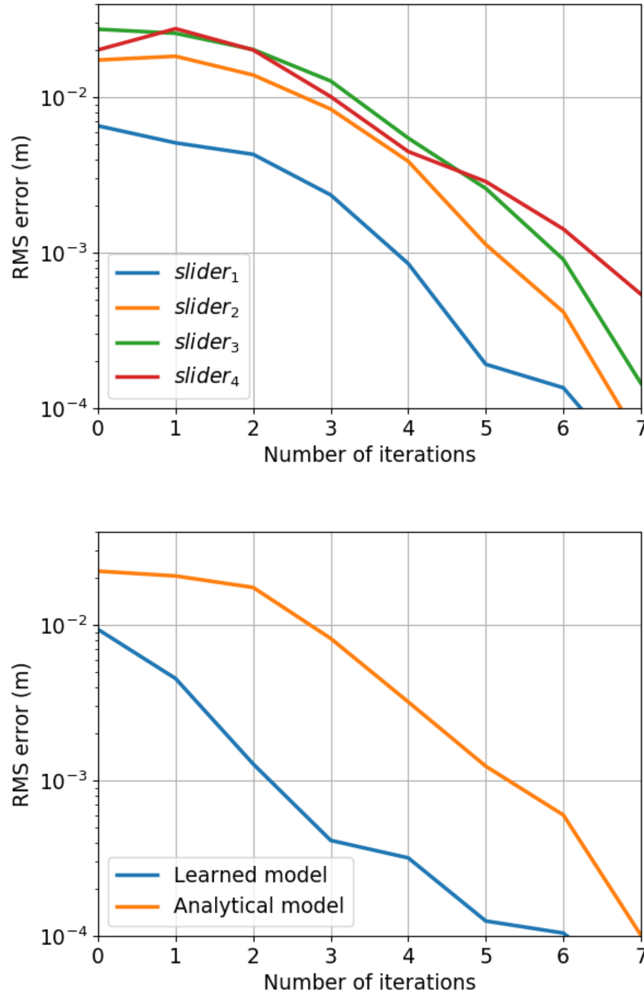


Figure 7.4: Root mean square error (in log scale) along the full trajectory per object for single object pushing (bottom) and multiple object pushing (top) using *only* the learned model. Here I consider a control sequence of 8 actions. The average object displacement for multi-object pushing is 0.034 ± 0.082 m and for single object pushing it is 0.046 ± 0.040 m. The error at iteration eight is 0. I find that the convergence of Parareal appears similar even with a longer control sequence.

I use Parareal to produce the corresponding sequence of states and compare the result against the states measured for the real robot with optical tracking.

Figure 7.5 shows the RMS error between Parareal's prediction at different iteration numbers and the real-world pushing data. Vertical red bars indicate 95% confidence intervals.

Parareal's real-world error decreases with increasing iteration numbers and it is eventually twice as accurate as the coarse model. These results indicate that Parareal's predictions with a learned coarse model are indeed close to the real-world physics predictions. Figure 7.7 shows snapshots of the experiments.

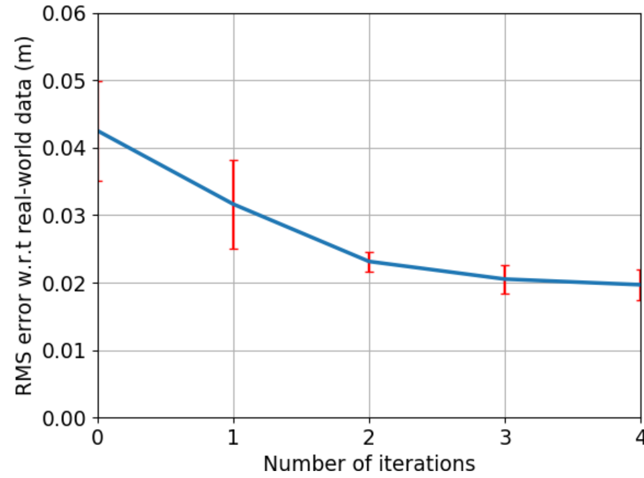


Figure 7.5: Root mean square error along the full trajectory for all 4 sliders *measured with respect to the real-world pushing data*. The vertical bars indicate a 95% confidence interval of the mean. The learned coarse physics model at iteration 0 has the largest error and the fine model provides the best prediction w.r.t the real-world pushing physics.

7.4.3.2 Planning and control

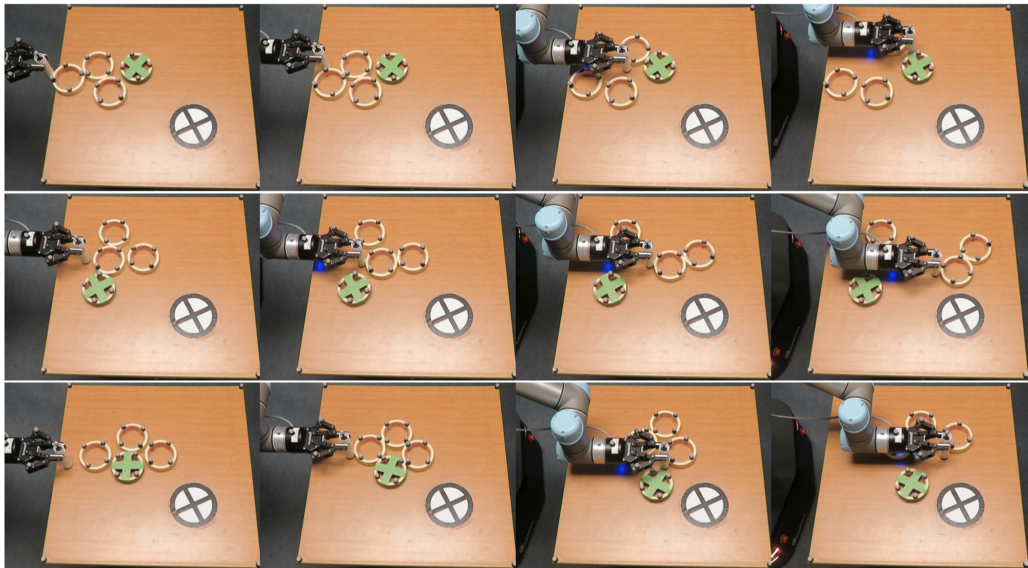


Figure 7.6: The resulting sequence of states for applying a random control sequence starting from some random initial state in the real-world. Our goal is to assess the accuracy of the Parareal physics models with respect to real-world physics. I collect 50 such samples. These are some snapshots for 3 of such scenes - one per row with initial state on the left and final state on the right.

I use the Parareal predictive model for robotic manipulation to generate plans faster than using the fine model directly. In this section, I complete 3 real

robot executions with Parareal at 1 iteration. I use the learned model as the coarse model in all cases.

As can be seen in Figure 7.7, the robot's task is to push the green slider into the target region marked with X. The robot is allowed to make contact with other sliders. An execution fails when a non-goal object is pushed into the goal region or over the edge of the table.

The robot was successful for all 3 sample scenes. Some sample plans for two scenes are shown in Figure 7.7. The third scene is shown in Figure 7.1. I find that using Parareal with a learned coarse model for physics predictions, a robot can successfully complete complex real-world pushing manipulation tasks involving multiple objects. At 1 Parareal iteration, I complete the tasks about 4 times faster than directly using the fine model.

In general, I trade-off physics prediction accuracy with respect to time. An important question then is how many iterations of Parareal to use for physics-based robotic manipulation i.e. how accurate should the physics predictions be? This depends on the manipulation task. For example, physics prediction accuracy should be higher when a robot is tasked with pushing an object on a narrow strip versus a large table where the chances of failure are lower.

Fig. 7.5 shows coarse physics errors (iteration 0) w.r.t. the real-world data of up to 5cm which is about the radius of a slider. Therefore, I conclude that the coarse model alone is not sufficient to complete the robotic manipulation task considered here — an object can easily fall-off the table due to an inaccurately planned action.

Furthermore, there is uncertainty during robotic pushing in the real-world [97].Agboh, Ruprecht, and Dogar [5] showed that physics predictions with errors below real-world stochasticity (e.g. position standard deviation at the end of a real-world push) have similar planning success rates. Hence it is usually pointless to have physics predictions as accurate as the fine model.

7.5 Summary

I demonstrate the use of Parareal to parallelize the predictive model in a robot manipulation task involving multiple objects. As coarse model, I propose a neural network, trained with a physics simulator. I show that for single object pushing, Parareal converges faster with the learned model than with a coarse physics-based model I introduced in Chapter 6. Furthermore, I show that Parareal with the learned model as coarse propagator can successfully complete

Figure 7.7: Robotic manipulation planning and control for 2 different scenes. The robot succeeds in all scenes using Parareal with a learned coarse model for physics predictions. The third planning and control scene is in Fig. 7.1.

tasks that involve pushing multiple objects. I also show that although a simulator is used to provide training data, Parareal with a learned coarse model can accurately predict experiments that involve pushing with a real robot.

An important question when combining coarse and fine models with Parareal is the choice of a coarse model. Should one use a learned or analytical coarse model? It depends on the task. If an intuitive, fast-to-compute model exists for a task then one should use it, otherwise one can rely on a learned coarse physics model for the task.

Chapter 8

Conclusion

I presented planners and controllers for physics-based manipulation. Our focus was on methods that improve real-world manipulation success under uncertainty and in multi-contact environments, while achieving fluent/real-time execution.

I learned the following lessons:

- An online re-planning/closed-loop approach to physics-based manipulation can significantly improve success rates under clutter and uncertainty.
- Given that physics simulations are computationally expensive, physics-based planning and re-planning is typically slow. I demonstrated that real-time re-planning cycles can be achieved through an appropriate underlying planning algorithm - one that accepts warm-starts. Thus, I showed for the first time, real-time reactive physics-based manipulation in clutter.
- I showed that fully robust open-loop plans are difficult to find/may not exist for many multi-contact manipulation environments. Our approach of interleaving robust open-loop execution and closed-loop control improved success rates and lead to more fluent/real-time execution.
- Robots can adapt their actions to the accuracy requirements of a task - pushing fast for low accuracy tasks and slow for high accuracy tasks. In this thesis, I proposed a task-adaptive planning algorithm that makes this possible. A robot can embrace uncertainty, and adapt to the accuracy requirements of a task, by slowing down and generating "careful" motion when the task requires high accuracy, and by speeding up and moving fast when the task tolerates inaccuracy.
- I combined coarse and fine physics models to generate hybrid models. The combination is possible thanks to the parallel-in-time integration algorithm, Parareal. With these hybrid models, I showed a significant reduction in physics-based manipulation planning time, without sacrificing success rates.

- I showed faster Parareal convergence with a learned coarse physics model. This translates to faster physics-based manipulation planning and control.

8.1 Limitations and Future Work

8.1.1 State estimation

Throughout this thesis, we rely on a motion capture system, Optitrack, to estimate the state of objects. This requires that markers are placed on objects. While these markers are not aesthetically desirable, they allow for one to conduct more controlled experiments, where uncertainty can be induced during state estimation. The focus of this thesis has been on the underlying planners and controllers, not on scene perception with cameras. In future work, I plan to investigate scene perception methods for robotic manipulation [95], and tightly couple these methods with my planners and controllers.

8.1.2 Full observability

In this thesis, I assume that the state space is fully observable. I.e. we have complete information about the position and velocities of all objects and the robot. However, for some problems this assumption would not hold. For example, imagine opening the shelf to retrieve a salt shaker, but it is not immediately visible. It is probably hidden behind other objects in the fridge. In such settings, it is not possible to know the state of all objects at all times. In future work, I plan to integrate my planners and controllers within a partial observability framework [44].

8.1.3 Deformable objects

Deformable objects such as ropes were not considered in this thesis. We used a rigid body physics model assumption. An important question is how to represent and estimate the state of such deformable objects. Also, deformable object physics predictions are currently much more computationally expensive in comparison with rigid-bodies. In the future, I plan to build coarse models of physics for deformable object manipulation [88, 35], and significantly speed-up their predictions through parallel-in-time integration.

8.1.4 Parallelization

Parallelization has been a central theme throughout this thesis. From parallelization in stochastic trajectory optimization to parallelization across time, to speed up physics predictions. Compute has become cheaper and more accessible, with more parallel cores becoming available. These advances will make the algorithms presented in this thesis much more relevant with time, significantly increasing their impact for robotic manipulation.

8.1.5 Robust or non-robust object motions

In this thesis, I proposed methods to generate robust robot manipulation plans. However, they focused on the overall system state. However, individual object motions can provide useful information for manipulation planning and control.

Robot actions can result in object motions that are robust or non-robust. In Fig. 8.1, I show an example of object motions that result in an overall robust or non-robust trajectory. I generate several sample initial states as shown on the left image, varying initial object positions, around their current position. Then I apply the forward robot motion shown. In the right image I see the resulting state distribution for several objects, at the end of a single action. The object position distribution is smaller at the final state for the cylinder compared with the initial state. Hence, this is a robust action for the cylinder. However, the opposite is the case for the box. The final object position distribution is larger for the box. This is due to the fact that sometimes the gripper makes contact with it, and rotates it, and other times it does not. The action is non-robust for the box.

The overall robustness of an action in these scenes is then judged through the aggregate of these robust and non-robust object motions. An important direction for future work is to identify 'important' objects of interest for a given task. Then one can focus planning on making these object motions robust.

8.1.6 Uncertainty model

In this work, we used simplified uncertainty models that are mainly based on randomly changing physics parameters in a simulator. This is only an approximation of the real world stochastic phenomena. I will work toward finding uncertainty models that better describe the real world physics stochasticity especially for manipulation in clutter.

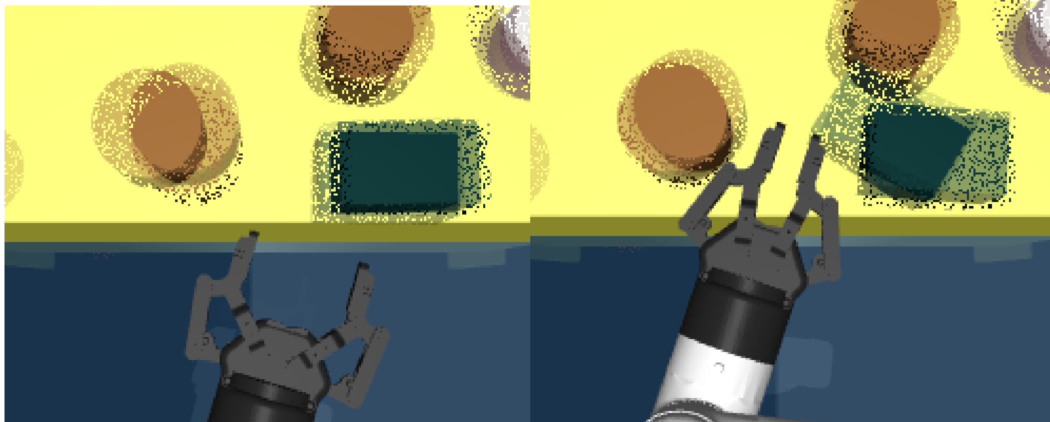


Figure 8.1: An example of robust and non-robust object motions in a given trajectory segment. On the left image, each object takes several possible positions. This illustrates the initial state uncertainty. After a robot motion that pushes on both a cylinder to the left and a box to the right, the size of the state uncertainties change. It is smaller for the cylinder to the left, and larger for the box to the right. Hence the cylinder's motion is robust and the boxes' motion is not.

8.1.7 Task adaptivity with physics models

Different tasks require different degrees of accuracy [2]. For example, think of searching for a sock in the sock drawer, versus searching for a wine glass in the glass cabinet. It is okay for a robot's physics predictions to be coarse in the former example, which is not the case in the latter. Parareal can be used to explore this spectrum, and generate coarse predictions when it is sufficient for the task, and more accurate predictions as the task requires.

8.2 Summary

In this thesis I addressed the problems of uncertainty and multi-contact interactions during physics-based manipulation. To handle uncertainty, I proposed robust open-loop methods, closed-loop methods, and a combination of both. To achieve real-time planning/re-planning, I proposed two approaches. First, a new stochastic trajectory optimization algorithm that accepts warm-starts, such that re-planning only takes a few iterations to converge. Second, I introduced hybrid physics models through a combination of coarse and fine physics models. These hybrid models are as accurate as the fine model but are significantly faster to compute, thanks to parallel computing.

I believe that through the physics-based manipulation planning and control algorithms presented in this thesis, we as a society can get closer to seeing robots in our everyday lives.

Bibliography

[1] Abraham, I., Handa, A., Ratliff, N., Lowrey, K., Murphey, T. D., and Fox, D. "Model-Based Generalization Under Parameter Uncertainty Using Path Integral Control". In: *IEEE Robotics and Automation Letters* 5.2 (2020).

[2] Agboh, W. C. and Dogar, M. R. "Pushing Fast and Slow: Task-Adaptive Planning for Non-prehensile Manipulation Under Uncertainty". In: *Algorithmic Foundations of Robotics XIII*. 2020.

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

[4] Agboh, W. C. and Dogar, M. R. "Robust Physics-Based Manipulation by Interleaving Open and Closed-Loop Execution". In: *arXiv*. 2021.

[5] Agboh, W. C., Ruprecht, D., and Dogar, M. R. "Combining Coarse and Fine Physics for Manipulation using Parallel-in-Time Integration". In: *International Symposium on Robotics Research (ISRR)* (2019).

[6] Agboh, W. C., Ruprecht, D., and Dogar, M. R. "Parareal with a learned coarse model for robotic manipulation". In: *Comput. Visual Sci.* 23.8 (2020).

[7] Agrawal, P., Nair, A. V., Abbeel, P., Malik, J., and Levine, S. "Learning to poke by poking: Experiential learning of intuitive physics". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2016.

[8] Ajay, A., Wu, J., Fazeli, N., Bauza, M., Kaelbling, L. P., Tenenbaum, J. B., and Rodriguez, A. "Augmenting Physical Simulators with Stochastic Neural Networks: Case Study of Planar Pushing and Bouncing". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018.

[9] Anders, A. S., Kaelbling, L. P., and Lozano-Perez, T. "Reliably Arranging Objects in Uncertain Domains". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018.

[10] Angelov, D., Hristov, Y., Burke, M., and Ramamoorthy, S. “Composing Diverse Policies for Temporally Extended Tasks”. In: *IEEE Robotics and Automation Letters* 5.2 (2020), pp. 2658–2665. DOI: 10.1109/LRA.2020.2972794.

[11] Arruda, E., Mathew, M. J., Kopicki, M., Mistry, M., Azad, M., and Wyatt, J. L. “Uncertainty averse pushing with model predictive path integral control”. In: *IEEE-RAS International Conference on Humanoid Robots (Humanoids)*. 2017.

[12] Babaeizadeh, M., Finn, C., Erhan, D., Campbell, R. H., and Levine, S. “Stochastic Variational Video Prediction”. In: *International Conference on Learning Representations (ICLR)*. 2018.

[13] Bauza, M. and Rodriguez, A. “A probabilistic data-driven model for planar pushing”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2017.

[14] Bejjani, W., Agboh, W. C., Dogar, M. R., and Leonetti, M. “Occlusion-Aware Search for Object Retrieval in Clutter”. In: *arXiv*. 2020. eprint: 2011.03334 (cs.RO).

[15] Bejjani, W., Dogar, M. R., and Leonetti, M. “Learning Physics-Based Manipulation in Clutter: Combining Image-Based Generalization and Look-Ahead Planning”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019.

[16] Bejjani, W., Papallas, R., Leonetti, M., and Dogar, M. R. “Planning with a Receding Horizon for Manipulation in Clutter Using a Learned Value Function”. In: *IEEE-RAS International Conference on Humanoid Robots (Humanoids)*. 2018.

[17] Bejjani, W., Leonetti, M., and Dogar, M. R. “Learning image-based Receding Horizon Planning for manipulation in clutter”. In: *Robotics and Autonomous Systems* 138 (2021), p. 103730. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2021.103730. URL: https://www.sciencedirect.com/science/article/pii/S0921889021000154.

[18] Cadeau, T. and Magoules, F. “Coupling the Parareal Algorithm with the Waveform Relaxation Method for the Solution of Differential Algebraic Equations”. In: *International Symposium on Distributed Computing and Applications to Business, Engineering and Science*. 2011, pp. 15–19.

[19] Calli, B., Walsman, A., Singh, A., Srinivasa, S., Abbeel, P., and Dollar, A. M. "Benchmarking in Manipulation Research: Using the Yale-CMU-Berkeley Object and Model Set". In: *IEEE Robotics Automation Magazine* 22.3 (2015), pp. 36–52.

[20] Choi, S., Lee, K., Lim, S., and Oh, S. "Uncertainty-Aware Learning from Demonstration Using Mixture Density Networks with Sampling-Free Variance Modeling". In: *IEEE International Conference of Robotics and Automation*. 2018.

[21] Cruciani, S. and Smith, C. "In-hand manipulation using three-stages open loop pivoting". In: *IEEE International Conference on Intelligent Robots and Systems (IROS)*. 2017.

[22] Davchev, T., Luck, K. S., Burke, M., Meier, F., Schaal, S., and Ramamoorthy, S. "Residual Learning from Demonstration: Adapting Dynamic Movement Primitives for Contact-rich Insertion Tasks". In: (2020). arXiv: `2008.07682 [cs.RO]`.

[23] Diankov, R., Srinivasa, S. S., Ferguson, D., and Kuffner, J. "Manipulation planning with caging grasps". In: *IEEE-RAS International Conference on Humanoid Robots (Humanoids)*. 2008.

[24] Dogar, M., Hsiao, K., Ciocarlie, M., and Srinivasa, S. "Physics-Based Grasp Planning Through Clutter". In: *Proceedings of Robotics: Science and Systems*. 2012.

[25] Ebert, F., Dasari, S., Lee, A. X., Levine, S., and Finn, C. "Robustness via Retrying: Closed-Loop Robotic Manipulation with Self-Supervised Learning". In: *Conference on Robot Learning (CoRL)*. 2018.

[26] Erez, T., Tassa, Y., and Todorov, E. "Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX". In: *IEEE International Conference on Robotics and Automation*. 2015.

[27] Fan, T., Schultz, J., and Murphey, T. "Efficient Computation of Higher-Order Variational Integrators in Robotic Simulation and Trajectory Optimization". In: *Algorithmic Foundations of Robotics XIII*. 2020, pp. 689–706.

[28] Finn, C., Goodfellow, I., and Levine, S. "Unsupervised learning for physical interaction through video prediction". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2016.

[29] Finn, C. and Levine, S. "Deep visual foresight for planning robot motion". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2017.

[30] Finn, C., Goodfellow, I., and Levine, S. "Unsupervised Learning for Physical Interaction through Video Prediction". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2016.

[31] Finn, C. and Levine, S. "Deep visual foresight for planning robot motion". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 2786–2793. DOI: 10.1109/ICRA.2017.7989324.

[32] Fitts, P. M. "The information capacity of the human motor system in controlling the amplitude of movement". In: *Experimental Psychology* (1954).

[33] Gander, M. and Vandewalle, S. "Analysis of the Parareal Time-Parallel Time-Integration Method". In: *SIAM Journal on Scientific Computing* 29.2 (2007), pp. 556–578.

[34] Goyal, S., Ruina, A., and Papadopoulos, J. "Planar sliding with dry friction Part 1. Limit surface and moment function". In: *Wear* 143.2 (1991), pp. 307 –330.

[35] Grannen, J., Sundaresan, P., Thananjeyan, B., Ichnowski, J., Balakrishna, A., Hwang, M., Viswanath, V., Laskey, M., Gonzalez, J. E., and Goldberg, K. "Untangling Dense Knots by Learning Task-Relevant Keypoints". In: *Conference on Robot Learning (CORL)*. 2020.

[36] Guibert, D. and Tromeur-Dervout, D. "Adaptive Parareal for Systems of ODEs". In: *Domain Decomposition Methods in Science and Engineering XVI*. Ed. by Widlund, O. and Keyes, D. Vol. 55. Lecture Notes in Computational Science and Engineering. Springer Berlin Heidelberg, 2007, pp. 587–594.

[37] Hafner, D., Lillicrap, T., Ba, J., and Norouzi, M. "Dream to Control: Learning Behaviors by Latent Imagination". In: *International Conference on Learning Representations (ICLR)*. 2020.

[38] Hasan, M., Warburton, M., Agboh, W. C., Dogar, M. R., Leonetti, M., Wang, H., Mushtaq, F., Mon-Williams, M., and Cohn, A. G. "Human-like Planning for Reaching in Cluttered Environments". In: *IEEE International Conference on Robotics and Automation*. 2020.

[39] Haustein, J. A., King, J., Srinivasa, S. S., and Asfour, T. "Kinodynamic randomized rearrangement planning via dynamic transitions between statically stable states". In: *IEEE International Conference on Robotics and Automation*. 2015.

[40] Hogan, F. R., Grau, E. R., and Rodriguez, A. "Reactive Planar Manipulation with Convex Hybrid MPC". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018.

[41] Hogan, F. R. and Rodriguez, A. "Feedback Control of the Pusher-Slider System: A Story of Hybrid and Underactuated Contact Dynamics". In: *Algorithmic Foundations of Robotics XII* (2020), pp. 800–815.

[42] Hoque, R., Seita, D., Balakrishna, A., Ganapathi, A., Tanwani, A. K., Jamali, N., Yamane, K., Iba, S., and Goldberg, K. "VisuoSpatial Foresight for Multi-Step, Multi-Task Fabric Manipulation". In: *Robotics: Science and Systems (RSS)*. 2020.

[43] Howe, R. D. and Cutkosky, M. R. "Practical force-motion models for sliding manipulation". In: *International Journal of Robotics Research (IJRR)* 15.6 (1996), pp. 557–572.

[44] Hsiao, K., Kaelbling, L. P., and Lozano-Perez, T. "Grasping POMDPs". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2007.

[45] Huang, E., Jia, Z., and Mason, M. T. "Large-Scale Multi-Object Rearrangement". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2019.

[46] Huang, S. H., Zambelli, M., Kay, J., Martins, M. F., Tassa, Y., Pilarski, P. M., and Hadsell, R. *Learning Gentle Object Manipulation with Curiosity-Driven Deep Reinforcement Learning*. 2019. arXiv: `1903.08542 [cs.RO]`.

[47] Johnson, A. M., King, J., and Srinivasa, S. "Convergent Planning". In: *IEEE Robotics and Automation Letters* 1.2 (2016), pp. 1044–1051. ISSN: 2377-3766. DOI: `10.1109/LRA.2016.2530864`.

[48] Kahn, G., Villaflor, A., Pong, V., Abbeel, P., and Levine, S. "Uncertainty-Aware Reinforcement Learning for Collision Avoidance". In: *CoRR* (2017).

[49] Kalakrishnan, M., Chitta, S., Theodorou, E., Pastor, P., and Schaal, S. "STOMP: Stochastic trajectory optimization for motion planning". In: *International Conference on Robotics and Automation*. 2011.

[50] Kearns, M. J., Mansour, Y., and Ng, A. Y. "A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes". In: *International Joint Conferences on Artificial Intelligence (IJCAI)*. 1999.

[51] Kiatos, M. and Malassiotis, S. "Robust object grasping in clutter via singulation". In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019.

[52] King, J. E., Haustein, J. A., Srinivasa, S. S., and Asfour, T. "Nonprehensile whole arm rearrangement planning on physics manifolds". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2015.

[53] Kitaev, N., Mordatch, I., Patil, S., and Abbeel, P. "Physics-based trajectory optimization for grasping in cluttered environments". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2015.

[54] Kloss, A., Schaal, S., and Bohg, J. "Combining learned and analytical models for predicting action effects". In: *CoRR* abs/1710.04102 (2017).

[55] Kong, N. J. and Johnson, A. M. "Optimally Convergent Trajectories for Navigation". In: *International Symposium on Robotics Research*. 2019.

[56] Kopicki, M., Zurek, S., Stolkin, R., Moerwald, T., and Wyatt, J. L. "Learning modular and transferable forward models of the motions of push manipulated objects". In: *Autonomous Robots* (2017).

[57] Koval, M. C., King, J. E., Pollard, N. S., and Srinivasa, S. S. "Robust trajectory selection for rearrangement planning as a multi-armed bandit problem". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2015.

[58] Laskey, M., Lee, J., Chuck, C., Gealy, D., Hsieh, W., Pokorny, F. T., Dragan, A. D., and Goldberg, K. "Robot grasping in clutter: Using a hierarchy of supervisors for learning from demonstrations". In: *IEEE Internatioanl Conference on Automation Science and Engineering (CASE)*. 2016.

[59] Laskey, M., Lee, J., Chuck, C., Gealy, D., Hsieh, W., Pokorny, F. T., Dragan, A. D., and Goldberg, K. "Robot grasping in clutter: Using a hierarchy of supervisors for learning from demonstrations". In: *IEEE International Conference on Automation Science and Engineering (CASE)*. 2016.

[60] Li, J., Lee, W. S., and Hsu, D. "Push-Net: Deep Planar Pushing for Objects with Unknown Physical Properties". In: *Robotics: Science and Systems*. 2018.

[61] Li, W. and Todorov, E. "Iterative Linear Quadratic Regulator Design for Nonlinear Biological Movement Systems". In: *International Conference on Informatics in Control, Automation and Robotics*. 2004.

[62] Lions, J.-L., Maday, Y., and Turinici, G. "A "parareal" in time discretization of PDE's". In: *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics* 332 (2001), pp. 661–668.

[63] Lohmiller, W. and Slotine, J. E. "On Contraction Analysis for Non-linear Systems". In: *Automatica* 34.6 (1998), pp. 683–696. ISSN: 0005-1098.

[64] Luck, K. S., Vecerik, M., Stepputtis, S., Amor, H. B., and Scholz, J. "Improved Exploration through Latent Trajectory Optimization in Deep Deterministic Policy Gradient". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019.

[65] Luders, B., Kothari, M., and How, J. "Chance constrained RRT for probabilistic robustness to environmental uncertainty". In: *AIAA Guidance, Navigation, and Control Conference*. 2010. DOI: 10.2514/6.2010-8160.

[66] Lynch, K. M., Maekawa, H., and Tanie, K. "Manipulation And Active Sensing By Pushing Using Tactile Feedback". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 1992.

[67] Maday, Y. and Turinici, G. "Parallel in time algorithms for quantum control: Parareal time discretization scheme". In: *International Journal of Quantum Chemistry* 93.3 (2003), pp. 223–228.

[68] Martin, K. "Parallel multiple shooting for the solution of initial value problems". In: *Parallel Computing* 20.3 (1994), pp. 275–295.

[69] Mason, M. "Mechanics and Planning of Manipulator Pushing Operations". In: *International Journal of Robotics Research* 5.3 (1986), pp. 53–71.

[70] Matas, J., James, S., and Davison, A. J. "Sim-to-Real Reinforcement Learning for Deformable Object Manipulation". In: *Conference on Robot Learning (CoRL)*. 2018.

[71] Meriçli T., V. M. and Akın, H. "Push-manipulation of complex passive mobile objects using experimentally acquired motion models". In: *Autonomous Robots* 38 (2015).

[72] Minion, M. "A hybrid parareal spectral deferred corrections method". In: *Commun. Appl. Math. Comput. Sci.* 5.2 (2010), pp. 265–301.

[73] Muhayyuddin, Moll, M., Kavraki, L., and Rosell, J. "Randomized Physics-Based Motion Planning for Grasping in Cluttered and Uncertain Environments". In: *IEEE Robotics and Automation Letters* 3.2 (2018), pp. 712–719.

[74] Pan, Z. and Manocha, D. "Time Integrating Articulated Body Dynamics Using Position-Based Collocation Method". In: *Algorithmic Foundations of Robotics XIII.* 2020, pp. 673–688.

[75] Papallas, R., Cohn, A. G., and Dogar, M. R. "Online Replanning With Human-in-the-Loop for Non-Prehensile Manipulation in Clutter — A Trajectory Optimization Based Approach". In: *IEEE Robotics and Automation Letters* 5.4 (2020), pp. 5377–5384. DOI: 10.1109/LRA.2020.3006826.

[76] Papallas, R. and Dogar, M. R. "Non-Prehensile Manipulation in Clutter with Human-In-The-Loop". In: *2020 IEEE International Conference on Robotics and Automation (ICRA).* 2020.

[77] Pauly, L., Agboh, W. C., Hogg, D. C., and Fuentes, R. "O2A: One-shot Observational learning with Action vectors". In: *arXiv.* 2020. eprint: 1810.07483 (cs.RO).

[78] Péret, L. and Garcia, F. "On-line search for solving Markov decision processes via heuristic sampling". In: *European Conference on Artificial Intelligence (ECAI).* IOS Press. 2004.

[79] Rajeswaran, A., Kumar, V., Gupta, A., Vezzani, G., Schulman, J., Todorov, E., and Levine, S. "Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations". In: *Robotics Science and Systems (RSS).* 2018.

[80] Richter, C. and Roy, N. "Safe Visual Navigation via Deep Learning and Novelty Detection". In: *Robotics Science and Systems (RSS).* 2017.

[81] Ruiz-Ugalde, F., Cheng, G., and Beetz, M. "Fast adaptation for effect-aware pushing". In: *IEEE/RSJ International Conference on Humanoid Robots (Humanoids).* 2011.

[82] Ruprecht, D. "Implementing Parareal - OpenMP or MPI?" In: *CoRR* (2015).

[83] S. Günther L. Ruthotto, J. S. E. C. N. G. "Layer-Parallel Training of Deep Residual Neural Networks". arXiv:1812.04352 [math.OC]. 2019. URL: https://arxiv.org/abs/1812.04352.

[84] Schroder, J. "Parallelizing Over Artificial Neural Network Training Runs with Multigrid". arXiv:1708.02276 [cs.NA]. 2017. URL: https://arxiv.org/abs/1708.02276.

[85] Seita, D., Ganapathi, A., Hoque, R., Hwang, M., Cen, E., Tanwani, A. K., Balakrishna, A., Thananjeyan, B., Ichnowski, J., Jamali, N., Yamane, K., Iba, S., Canny, J., and Goldberg, K. "Deep Imitation Learning of Sequential Fabric Smoothing Policies". In: *International Symposium on Robotics Research (ISRR)*. 2019.

[86] Srivastava, S., Fang, E., Riano, L., Chitnis, R., Russell, S., and Abbeel, P. "Combined task and motion planning through an extensible planner-independent interface layer". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2014.

[87] Stilman, M., Schamburek, J. U., Kuffner, J., and Asfour, T. "Manipulation Planning Among Movable Obstacles". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2007.

[88] Sundaresan, P., Grannen, J., Thananjeyan, B., Balakrishna, A., Laskey, M., Stone, K., Gonzalez, J. E., and Goldberg, K. "Learning Rope Manipulation Policies Using Dense Object Descriptors Trained on Synthetic Depth Data". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2020.

[89] Tancredi, G., Sánchez, A., and Roig, F. "A Comparison Between Methods to Compute Lyapunov Exponents". In: *The Astronomical Journal* 121 (2001), pp. 1171–1179.

[90] Tassa, Y., Erez, T., and Todorov, E. "Synthesis and stabilization of complex behaviors through online trajectory optimization". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012.

[91] Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Las Casas, D. de, Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A., Lillicrap, T., and Riedmiller, M. *DeepMind Control Suite*. Tech. rep. DeepMind, Jan. 2018. URL: https://arxiv.org/abs/1801.00690.

[92] Todorov, E., Erez, T., and Tassa, Y. "MuJoCo: A physics engine for model-based control". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012.

[93] Trindade, J. M. F. and Pereira, J. C. F. "Parallel-in-Time Simulation of Two-Dimensional, Unsteady, Incompressible Laminar Flows". In: *Numerical Heat Transfer, Part B: Fundamentals* 50.1 (2006), pp. 25–40.

[94] Williams, G., Aldrich, A., and Theodorou, E. "Model Predictive Path Integral Control using Covariance Variable Importance Sampling". In: *CoRR* (2015).

[95] Xiang, Y., Schmidt, T., Narayanan, V., and Fox, D. "PoseCNN: A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes". In: *Robotics: Science and Systems (RSS)*. 2018.

[96] Yalla, G. R. and Engquist, B. "Parallel in Time Algorithms for Multiscale Dynamical Systems Using Interpolation and Neural Networks". In: *Proceedings of the High Performance Computing Symposium*. 2018, 9:1–9:12.

[97] Yu, K. T., Bauza, M., Fazeli, N., and Rodriguez, A. "More than a million ways to be pushed. A high-fidelity experimental dataset of planar pushing". In: *IEEE International Conference on Intelligent Robots and Systems (IROS)*. 2016.

[98] Yuan, W., Hang, K., Kragic, D., Wang, M. Y., and Stork, J. A. "End-to-end nonprehensile rearrangement with deep reinforcement learning and simulation-to-reality transfer". In: *Robotics and Autonomous Systems* 119 (2019), pp. 119–134.

[99] Zhan, A., Zhao, P., Pinto, L., Abbeel, P., and Laskin, M. "A Framework for Efficient Robotic Manipulation". In: (2020). arXiv: `2012.07975 [cs.RO]`.

[100] Zhou, J., Mason, M. T., Paolini, R., and Bagnell, D. "A convex polynomial model for planar sliding mechanics: theory, application, and experimental validation". In: *International Journal of Robotics Research* 37.2 (2018), 249 – 265.

[101] Zhou, J., Paolini, R., Johnson, A. M., Bagnell, J. A., and Mason, M. T. "A Probabilistic Planning Framework for Planar Grasping Under Uncertainty". In: *IEEE Robotics and Automation Letters* 2.4 (2017).

[102] Zito, C., Stolkin, R., Kopicki, M., and Wyatt, J. L. "Two-level RRT planning for robotic push manipulation". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2012.