

HIGH-SPEED MOTION GENERATION FOR MOBILE MANIPULATORS

by

Ke Dong

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
University of Toronto Institute for Aerospace Studies
University of Toronto

© Copyright 2020 by Ke Dong

Abstract

High-speed Motion Generation for Mobile Manipulators

Ke Dong

Master of Applied Science

University of Toronto Institute for Aerospace Studies

University of Toronto

2020

Mobile manipulators consist of a mobile platform equipped with one or more robot arms and are of interest for a wide array of challenging tasks because of their extended workspace and dexterity. Typically, mobile manipulators are deployed in slow-motion collaborative robot scenarios, but they are capable of more challenging tasks. In this thesis, we consider scenarios where accurate high-speed motions are required. We proposed two different frameworks for this regime of tasks: *(i)* a modular framework that includes a bi-level motion planner and a learning-based controller; and *(ii)* an end-to-end framework that is based on deep reinforcement learning and simulation-to-reality transfer techniques. We evaluate our frameworks with a mobile manipulator platform through numerous high-speed ball catching experiments, where we show a success rate of **85.33%** for real robots and **85.60%** for simulated robots.

Acknowledgements

The past two year at UTIAS is really a challenging but fruitful journey. There are challenges, successes, and, most importantly, people who support me along the way. I want to thank all people who helped me, especially those mentioned hereafter.

I would like to start with my supervisors, Professor Angela Schoellig and Professor Florian Shkurti. During the course of this thesis, they helped me sort things out and pushed me back into the correct path. I want to thank Angela for accepting me as her student and supporting me in exploring my own research ideas. The biggest lesson that Angela told me is to start with simple things and build complex theory after, which is my current guideline for research and life. I want to thank Florian for our in-depth discussions of research ideas and his detailed proofreading of my paper. We have explored so many exciting fields together which largely expands my knowledge boundary.

As an international student, the Dynamic System Lab is a warm family to me. This is a group of talented Robotics researchers from all over the world, with whom I can always find insightful ideas. Besides academic discussions, we have experienced a lot of interesting things, like BBQ party, paint balls, and our daily lunchtime talk. Thank you all: Karime, Jacopo, Chris, Siqi, Melissa, Adam Heins, Wenda, Keenan, Carlos, Adam Hall, Michael, Sep, Jeremy and Abi.

Leaving the academic environment, many special thanks go to my family and my friends. Thank you Mom and Dad, for always being there with me, for your life advice, and your secret Chinese recipes. Thank you Wenwen, for your encouragement and friendship in all downs I experienced since when we first met. Thanks to all people who helped me along the way.

Contents

1	Introduction	1
1.1	Background & Motivation	1
1.2	Related Work	3
1.3	Contributions	5
1.4	Thesis Overview	6
2	Preliminaries	7
2.1	Mobile manipulators	7
2.1.1	Forward Kinematics	7
2.1.2	Inverse Kinematics	8
2.1.3	Dynamics Model	9
2.2	Bayesian Optimization	10
2.2.1	Gaussian Process	10
2.2.2	Acquisition Function	11
2.3	Neural Networks	12
2.3.1	Multi-Layer Perceptron	12
2.3.2	Activation Functions	13
2.4	Reinforcement Learning	14
2.4.1	Fundamentals	14
2.4.2	Representative Algorithms	15
2.4.3	Sim2real Transfer	16
3	Modular Framework	19
3.1	Problem Statement	19
3.1.1	System Model	19
3.1.2	Problem Statement	20
3.2	Methodology	20
3.2.1	Ball Estimation and Prediction	21

3.2.2	Motion Planning	23
3.2.3	Low-level Joint Control	27
3.3	Framework Implementation and Evaluation	28
3.3.1	Ball Estimation and Trajectory Prediction	28
3.3.2	Motion Planning	29
3.3.3	Low-level Control	31
3.4	Experiments	32
3.4.1	Experimental Setup	32
3.4.2	Ball Catching Experiment	33
3.4.3	Failure analysis	36
3.5	Conclusion	36
4	End-to-end Framework	37
4.1	Problem Statement	38
4.2	Methodology	38
4.2.1	Ball Catching MDP	38
4.2.2	Task Policy and Policy Optimizer	40
4.2.3	Simulation Optimizer	40
4.2.4	Acquisition Policy	41
4.3	Simulation Experiment	42
4.3.1	Environment Setup and Framework Implementation	42
4.3.2	Experiment Result	44
4.4	Conclusion	48
5	Conclusions and Future Work	49
5.1	Conclusions	49
5.2	Future Work	50
	Bibliography	51

List of Tables

3.1	Joint trajectory tracking error comparison	32
4.1	The simulation environment parameters.	45

List of Figures

1.1	A mobile manipulator consisting of a UR10 arm and a Ridgeback mobile platform.	2
2.1	A generic domain randomization algorithm framework.	18
3.1	The mobile manipulator with the world frame $_wF$ (or $o_0x_0y_0z_0$), base frame $_bF$ (or $o_2x_2y_2z_2$), arm frame $_aF$ (or $o_3x_3y_3z_3$) and end effector frame $_{ee}F$ (or $o_8x_8y_8z_8$). Rotation axes of the six arm joints are also presented. Adapted from [1].	20
3.2	Diagram of the overall framework for accurate high-speed ball catching. Note that the estimation module shown here is for ball state estimation, with separate, but not shown estimation modules for arm and base controllers.	21
3.3	The geometric model of the collision-free space. This is a semi-cylinder with its bottom centered in the arm frame's origin.	24
3.4	A diagram of the inverse dynamics learning framework and the training phase. The DNN module modifies the reference signal to the baseline system for trajectory tracking performance improvement. Adapted from [2].	28
3.5	The ball trajectory prediction error of the 75 ball catching experiments in Experiment A, described in Sec. 3.4.2. The dashed area represents the standard deviation.	29
3.6	Tracking performance comparison in the x position of the mobile base. .	32
3.7	The 61.76 mm diameter ball and 94.90 mm diameter cup used in this paper. The position accuracy requirement in the x, y, z axes is approximately $(94.90 - 61.76)/2 = 16.57$ mm. The typical speed at the catch time is around 5 m/s, leading to the $16.57/5 \approx 3$ ms time precision requirement.	33

3.8	Ball trajectories of Experiment A. Starting from the left corner and in the anti-clockwise direction, the five ball throwing positions in the xy plane in meters are $(2.8, -3.1)$, $(2.9, 0.0)$, $(2.5, 2.0)$, $(0.0, 2.8)$, $(-2.0, 2.0)$. Ball trajectories are colored according to its throwing location.	34
3.9	An example of (a) arm joint and (b) mobile base tracking performance. Dashed lines represent desired trajectories, while solid lines represent actual trajectories. For a better illustration effect, we subtract initial values from trajectory waypoints, and thus joint angles relative to their initial values are shown here.	35
4.1	The mujoco simulation environment for the mobile manipulator ball catching task.	43
4.2	The episode reward when training and testing in the nominal environment. The green and orange line represent the policy trained with dense rewards and the policy trained with sparse rewards respectively. The green and orange area represent the one-sigma uncertainty bound.	46
4.3	The success rate when training and testing in the nominal environment. The green and orange line represent the policy trained with dense rewards and the policy trained with sparse rewards respectively. The green and orange area represent the one-sigma uncertainty bound.	46
4.4	The posterior distribution learned by the Gaussian process. The red squares represent observations from the nominal environment, the dashed line represents the prediction, and the the blue area represents the 95% confidence bound.	47
4.5	The success rate and simulation optimizer's cost function of every iteration when doing a sim2sim transfer, represented by orange and green lines respectively.	47

Chapter 1

Introduction

1.1 Background & Motivation

The next generation of industry, Industry 4.0, is believed to have mass customization with lower cost and higher quality. Production lines should be capable of switching between different tasks quickly. However, the commonly used fixed-base manipulators, which play an important role in modern production lines, falls behind this flexibility requirement. Mounted along the production line, the motion of fixed-base manipulators is limited to its small workspace. In contrast, mobile manipulators can move in the manufacturing planet and apply its skills where it is required.

A mobile manipulator combines mobile platforms and robot arms. Unlike fixed-base manipulators that are mainly used for repeated tasks in static environments, mobile manipulators can be used for complex tasks where a larger workspace and higher flexibility are needed, like airplane assembly, wind turbine manufacturing and shipbuilding [3]. They can also be used for collaborative tasks in dynamic and unstructured environments, like house cleaning and warehouse pick-and-place.

Current research on mobile manipulators mainly focuses on techniques in slow-motion scenarios, such as optimal path planning with kinematics redundancy, contact-based interaction with the environment, and human-robot collaboration [3, 4, 5]. However, mobile manipulators are capable of high-speed motions, and thus have the potential to be used in applications such as industrial high-speed pick-and-place, or other challenging tasks such as table tennis [6]. In this thesis we focus on high-speed trajectory generation and accurate tracking for mobile manipulators, with an emphasis on not compromising accuracy at higher speeds.

We focus on a benchmark task for accurate high-speed motions: object catching, specifically, ball catching. Within a fraction of a second, the robot needs to *(i)* predict

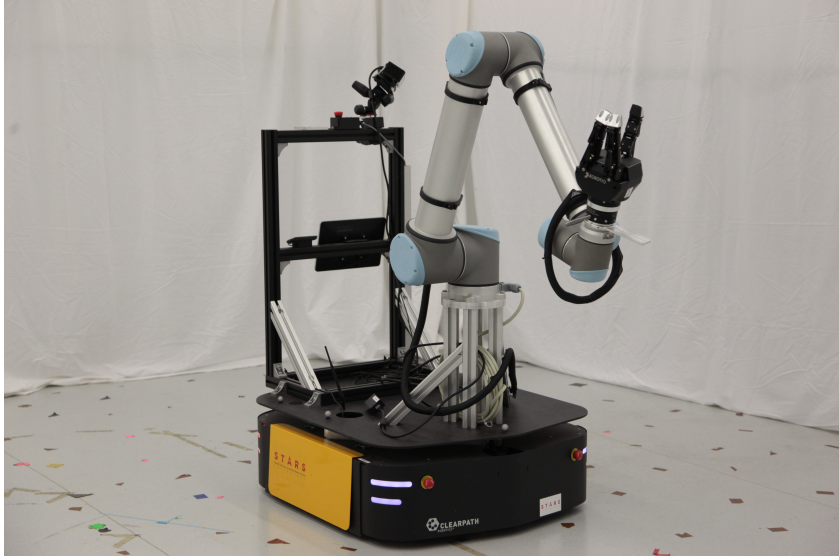


Figure 1.1: A mobile manipulator consisting of a UR10 arm and a Ridgeback mobile platform.

the ball's trajectory, *(ii)* generate and update a trajectory for graceful catching, and *(iii)* accurately track it with the mobile manipulator. There are numerous papers [7, 8, 9, 10] that address the ball catching problem with a *fixed-base* manipulator. However, using a *mobile* manipulator introduces additional challenges, due to the interplay between the following three characteristics:

Controlling wheel-driven bases: The mobile bases that are driven by Mecanum wheels [11], as used in this thesis, usually have lower position accuracy than manipulators. One example is the position accuracy of the Kuka KMR Quantec driven by Mecanum wheels of $\pm 5 \text{ mm}$ [12] while the accuracy of the UR10 arm is $\pm 0.05 \text{ mm}$ [13]. Thus extra efforts are needed to accurately control the base.

High precision in position and time: Precise end effector trajectory tracking, both spatially and temporally and in position and orientation, is a pre-requisite for successful catching. A precision of 16 mm in position and 3 ms in time must be achieved in this thesis. These parameters vary depending on the gripper [14].

Real-time computation: On average, the duration of the ball's free flight trajectory is one second, which puts a hard constraint on the computation time of ball prediction, motion planning and control. Motion planning is very challenging since complex constraints such as the nonlinear robot dynamics and forward kinematics must be considered.

Given the three challenges above, accurate high-speed motion generation for mobile manipulator ball catching has not been widely explored, with a few very notable exceptions [14]. In this thesis we present two different frameworks to resolve this prob-

lem: (i) a modular framework that contains a Kalman filter, a bi-level motion planner and a learning-based controller where Every module is optimized separately, and (ii) an end-to-end framework that contains a neural network policy and is optimized in an end-to-end fashion. We evaluate our modular framework on a real robot and the end-to-end framework in a simulated environment, achieving a success rate of 85.33% and 85.60% respectively.

1.2 Related Work

A large body of work has been devoted to the ball catching task with a fixed-base manipulator [7, 8, 9, 10, 15]. Early pioneering work in this field used heuristic methods for selecting the catching point and generated the joint trajectory via interpolation. For example, in [15], authors choose the closest point of the ball trajectory to the robot base as the catching position and orient the gripper perpendicular to the trajectory. Such methods are simple, as they restrict catching points to a certain area, but may not find the optimal robot motion for catching. Recent works use nonlinear optimization methods for selecting a catching pose, and parameterization methods, like trapezoidal velocity profile and third-order polynomials, for path generation for simplicity. In [7], a ball catching system consisting of a fixed-base 7 Degree-of-Freedom (DoF) arm and a four-finger hand is presented. It uses Sequential Quadratic Programming (SQP) for high-level goal selection, and restricts joint trajectories to trapezoidal velocity profiles. In contrast, in our modular framework, we do not parameterize the trajectory. Instead, we use QP to generate intermediate waypoints which define smooth and flexible trajectories. Besides sphere-like balls, manipulators are also capable of arbitrary-shape object catching. In [16], a dataset of human arbitrary-shape object catching is collected, and learned by a Gaussian Mixture Model. The learned model is then used for real-time motion generation for arbitrary-shape object catching using a manipulator. However, only fixed-base manipulators are used.

The work most similar to ours is the mobile humanoid system presented in [14], which consists of two light-weight 7 DoF arms and an omnidirectional mobile base. In their work, the mobile base is restricted to motion in the x axis only. Further, they use customized hardware capable of high-speed motions. In contrast, in our modular framework, we achieve high-speed motions in the $x - y$ plane with the mobile base, which brings the motion capacity of omnidirectional platforms into full play. Moreover, we use an off-the-shelf mobile manipulator, which may make our approach easier to generalize to other robotic platforms without the need for customized hardware and low-level access.

There are general motion planning and control algorithms for robots that can be applied to mobile manipulators. Common planning algorithms like potential field, rapidly exploring random trees (RRT), and probabilistic roadmaps (PRM) can be used for manipulator path planning [17]. These methods are good at finding feasible paths in cluttered environments, but they are computationally expensive. So they are not suitable here because real-time computation is needed. As for low-level control, there are numerous works that try to improve the tracking performance of high-speed and aggressive trajectories. Non-learning-based methods [5, 18] rely on accurate analytical system dynamics models. For example, the feedback linearization control uses an inverse dynamics model to cancel the non-linearity of the system and turn it into a simple double integrator [19]. However, in general, accurate analytical models are hard and time-consuming to obtain. Learning-based methods mainly focus on learning the system’s dynamics or inverse dynamics models [20, 21]. In [21], a Deep Neural Network (DNN) is used for direct inverse control of a quadrotor. However, guaranteeing stability with a DNN in the control loop can be very difficult. In contrast, the inverse dynamics learning method from [2] uses a DNN as an add-on module outside the original closed-loop system, which is safer in terms of stability. This method’s ability for improved, impromptu trajectory tracking has been demonstrated on quadrotors [2, 22], but not on mobile manipulators.

Out of the classical robot domain consisting of state estimation, planning and control techniques, there are works that achieve end-to-end mapping via learning one monotonic neural network policy from data, namely end-to-end learning. By end-to-end it means that the policy directly maps raw input measurements to robot low-level control commands. Compared with classical pipelines, end-to-end policies are learned from data in an end-to-end fashion. This saves a lot of engineering efforts in module design and may achieve better overall system performance. There have been several promising results in this direction. In [23], A PR2 robot learns a policy that maps raw image observations directly to torques at the robot’s joint. The robot can achieve challenging tasks like cube insertion and bottle opening. In [24], authors propose an end-to-end system capable to control an autonomous car directly from the pixels provided by the embedded cameras. With approximately 72 hours of driving data, the system was able to learn how to steer the car in different road types and weather conditions. But no one has tried the ball catching task, which requires high-speed motions in a dynamic environment.

End-to-end learning does have several drawbacks. The learned policy is usually a black-box model which is hard to be validated for safety requirements. Also, training an end-to-end system requires a huge amount of robot operation data, which is usually not feasible. To solve these problems, researchers tend to learn and validate policies in

simulation and then transfer the learned policy into reality, namely sim2real transfer, where simulators are usually a non-differential model with adjustable parameters. Current sim2real transfer techniques try to bridge the gap between simulation and reality from different perspective. There are methods that solve this problem in a system identification way. They are aimed at estimating accurate simulation parameters. Yu et al. [25] trained a system identification module to predict parameters of interest online, such as friction and length. The predicted parameters are then provided as input to a policy to compute the appropriate controls. Ramos et al. [26] trained a Gaussian mixture model to predict posterior distribution of simulator parameters, given a trajectory of states and actions from the real world. These two methods are promising, but are only tested in transferring between different simulation environments with a simple task.

There are methods try to achieve sim2real transfer in a robust control way. They are aimed at making the trained policy robust to modelling errors between simulation environments and the real world. Peng et al. [27] randomized the dynamics of a simulator using a fixed Gaussian distribution during training. The trained policy is then able to generalize to the dynamics of the real world without any training on the physical system. Following this idea, follow-up works try to adapt the randomization distribution smartly. Chebotar et al. [28] adapts the simulator parameter distribution to minimize the distance between simulation trajectories and real-world trajectories, using a gradient-free optimization method, which works well on real robot experiments

1.3 Contributions

The main contributions of this theses are as follows:

1. A modular framework that generates and accurately follows high-speed trajectories with a mobile manipulator. The proposed framework has a bi-level optimization scheme for motion planning, and a learning-based controller to improve the tracking performance of the mobile manipulator.
2. An end-to-end framework that directly maps raw measurements to robot control commands. The proposed framework use deep reinforcement learning algorithms for task policy training and bayesian optimization for sim2real transfer.

1.4 Thesis Overview

The thesis is organized as follows. Ch. 2 presents basic preliminaries. Ch. 3 presents the modular framework while Ch. 4 presents the end-to-end framework. Ch. 5 concludes this thesis and presents possible future work.

Chapter 2

Preliminaries

This chapter gives a brief introduction to mobile manipulator models, Bayesian optimization, neural networks and reinforcement learning algorithms.

2.1 Mobile manipulators

Manipulators can have different structures. They can be serial or parallel, the joints can be prismatic, revolute or ball joints, and either actuated or non-actuated, based on the required design and the application scenario [1]. In the past, a variety of robot kinematics structures have been developed. In this thesis, we assume the robot are serial and have actuated prismatic or revolute joints, for they can be used for a wide range of industrial applications.

2.1.1 Forward Kinematics

The forward kinematics is concerned with the relationship between the individual robot manipulator joints and the position and orientation of the end effector. A robot manipulator with n joints will have $n + 1$ links, because each joint connects two links. We number the links from 0 to n , starting from the ground, and we number the joints from 1 to n . Thus, joint i connects link $i - 1$ to link i . We will consider the location of joint i to be fixed with respect to link $i - 1$. When joint i is actuated, link i moves. Therefore, link 0 (the first link) is fixed and is usually attached with the ground. With the i th joint, we associate a joint variable, denoted by q_i . For a revolute joint, q_i is the angle of rotation, and for a prismatic joint, q_i is the joint displacement. We denote $\mathbf{q} = (q_1, q_2, \dots, q_n)$ as the *configuration vector*.

To perform the kinematic analysis, we rigidly attach a coordinate frame to each link.

Specifically, we attach $o_i x_i y_i z_i$ to link i , where o_i is the origin and $o_i x_i, o_i y_i, o_i z_i$ are the x, y, z axes of the i -th coordinate frame respectively. Therefore, whatever motion the robot executes, the coordinates of each point on link i are constant when expressed in the i th coordinate frame. Meanwhile, when joint i is actuated, link i and its attached frame, $o_i x_i y_i z_i$, have a same resulting motion. The frame $o_0 x_0 y_0 z_0$, which is attached to the ground, is referred to as the inertial frame.

Now suppose A_i is the homogeneous transformation matrix that expresses the orientation and position of $o_i x_i y_i z_i$ with respect to $o_{i-1} x_{i-1} y_{i-1} z_{i-1}$. The homogeneous transformation matrix A_i will be in the format of:

$$A_i = \begin{bmatrix} R_i^{i-1} & o_i^{i-1} \\ 0 & 1 \end{bmatrix} \quad (2.1)$$

where R_i^{i-1} and o_i^{i-1} is the relative rotation and translation from frame i to frame $i-1$. The matrix A_i is not constant, and it varies as the configuration of the robot is changed. However, the assumption that all joints are either revolute or prismatic means that A_i is a function of only a single joint variable, namely q_i . In other words:

$$A_i = A_i(q_i). \quad (2.2)$$

The D-H convention are commonly used to construct this function $A_i = A_i(q_i)$ [19]. Then the position and orientation of the end effector in the inertial frame are given by

$$H = H(q_1, q_2, \dots, q_n) = A_1(q_1) \cdots A_n(q_n), \quad (2.3)$$

where H is also a homogeneous transformation matrix and is in the format of:

$$H = \begin{bmatrix} R_n^0 & o_n^0 \\ 0 & 1 \end{bmatrix}. \quad (2.4)$$

Here R_n^0 and o_n^0 are the orientation and position of the end effector in the inertial frame, and they are functions of the robot configuration $\mathbf{q}$.

2.1.2 Inverse Kinematics

The inverse kinematics problem is concerned with finding the joint variables in terms of the given end effector position and orientation in the inertial frame. This problem is, in general, harder than the forward kinematics problem. Formally, given a homogeneous

transformation matrix H representing the pose of the end effector, we want to find a configuration vector $(q_1, q_2, \dots, q_n)$ that satisfies (2.3):

$$A_1(q_1) \cdots A_n(q_n) = H. \quad (2.5)$$

Equation (2.5) provides six independent equations (three for rotation and three for translation) in n unknown variables. These forward kinematics equations are in general complex non-linear non-convex functions and hard to solve. It is possible that no solution exists or multiple solutions exist for a given end effector pose.

Current inverse kinematics methods can be divided into analytical methods and numerical methods. Analytical methods are aimed at finding closed-form solutions to (2.5) via geometry analysis. The resulting solution is fast, but it is hard to have an analytical solution that works for every robot. Numerical methods treat (2.5) as an optimization problem with forward kinematics constraints. This method is more computationally expensive, but it is a generic algorithm that works for every robot.

2.1.3 Dynamics Model

Dynamics models describe the relationship between robot motion and the forces or moments that generates the motion, while the kinematics models, introduced in the previous section, only deals with robot motion without consideration of forces. There are two categories of dynamics models, namely forward dynamics and inverse dynamics. Forward dynamics models calculate robot motion from known forces and/or torques that applied on the robot, and this is usually used for motion prediction and planning. Inverse dynamics models calculate applied forces and/or torques on the robot from known robot motion, and this is usually used for feedback linearization control.

For a rigid-body mechanical system that subjects to holonomic constraints [19], its dynamics model is governed by the Euler-Lagrange dynamics equation:

$$M(\mathbf{q})\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + G(\mathbf{q}) = \boldsymbol{\varsigma}, \quad (2.6)$$

where M is a symmetric and positive definite matrix for inertial force terms, C is a matrix for centrifugal and Coriolis force terms, and G is a matrix for potential energy [19], $\dot{\mathbf{q}}, \ddot{\mathbf{q}}$ are joint velocities and accelerations, and $\boldsymbol{\varsigma}$ are generalized forces that applied on the joints. $\boldsymbol{\varsigma}$ are forces for prismatic joints and torques for revolute joints.

The Euler-Lagrange dynamics equation requires the values of certain parameters, like link mass, moments of inertia, etc. Determining these values from experiments would be

a challenging task due to the complexity of equations. Fortunately, the Euler-Lagrange equation is linear in these inertia parameters [19, 29]. This means that we can rewrite the equation in the following format:

$$M(\mathbf{q})\ddot{\mathbf{q}} + C(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + G(\mathbf{q}) =: Y(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}})\beta = \varsigma. \quad (2.7)$$

The function $Y(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) \in \mathbb{R}^{n \times 10}$ is called regressor, and β is the parameter vector consisting of ten inertial parameters of every link: the total mass, the six independent entries of the inertia tensor, and the three coordinates of the center of mass. This linearity property can be used for inertial parameter identification [29].

2.2 Bayesian Optimization

Bayesian optimization (BO) is an approach to optimizing objective functions whose analytical form is initially unknown. It is best-suited for optimization over continuous domains of less than 20 dimensions, and can handle stochastic noise in function evaluations. BO has no requirement for the objective function, like continuity and differentiability, and thus it can be regarded as a black-box optimization approach.

The BO algorithm is simple. It consists of two major modules: a Bayesian statistical model, Gaussian Process (GP), for modeling the objective function, and an acquisition function for deciding the next sample position. After evaluating the objective according to an initial space-filling experimental design, which often consists of points chosen uniformly at random, acquisition functions are used iteratively to allocate the remainder of a budget of N function evaluations [30].

2.2.1 Gaussian Process

A GP is a nonparametric regression method that is used to approximate a nonlinear objective function $f(x) : \mathcal{X} \rightarrow \mathbb{R}$ at a candidate input $x \in \mathcal{X}$. It assumes that function values $f(x)$ associated with different values of x are random variables, and any finite number of these variables have a joint Gaussian distribution. GP predicts the value of objective function $f(x)$, at a new input $x \in \mathcal{X}$ based on a set of n observations $\mathcal{D}_n = \{x_i, \hat{f}(x_i)\}_{i=1}^n$, where $\hat{f}(x_i)$ are noisy measurements of the function's true values: $\hat{f}(x_i) = f(x_i) + \omega$. Here ω is a zero mean Gaussian noise with variance σ_ω^2 . Assuming a zero prior mean of the GP and conditioned on the previous observations, the mean and

variance of the GP prediction are given by:

$$\begin{aligned}\mu_n(x) &= \mathbf{k}_n(x)(\mathbf{K}_n + \mathbf{I}_{n \times n} \sigma_\omega^2)^{-1} \hat{\mathbf{F}}_n, \\ \sigma_n^2(x) &= \kappa(x, x) - \mathbf{k}_n(x)(\mathbf{K}_n + \mathbf{I}_{n \times n} \sigma_\omega^2)^{-1} \mathbf{k}_n^\top(x),\end{aligned}\tag{2.8}$$

where $\hat{\mathbf{F}}_n = [\hat{f}(x_1), \hat{f}(x_2), \dots, \hat{f}(x_n)]$ is the vector of observed noisy function values. The matrix $\mathbf{K}_n \in \mathbb{R}^{n \times n}$ is the covariance matrix with entries $[\mathbf{K}_n]_{(i,j)} = \kappa(x_i, x_j)$, $i, j \in \{1, \dots, n\}$, where $\kappa(x_i, x_j)$ is the covariance function defining the covariance between two function values. $\kappa(x_i, x_j)$ is also called a kernel function. The vector $\mathbf{k}_n(x) = [\kappa(x, x_1), \dots, \kappa(x, x_n)]$ is the covariances between the new input and the observed data points, and $\mathbf{I}_{n \times n}$ is the identity matrix.

The choice of kernel functions $\kappa(x_i, x_j)$ and hyperparameters of kernel function has a large influence on the GP regression quality. There are two requirements for the kernel function. First, points closer in the input space are more strongly correlated. In other words, if $\|x - x'\| < \|x - x''\|$ for some norm $\|\cdot\|$, then $\kappa(x, x') > \kappa(x, x'')$. The other requirement is that kernels are required to be positive semi-definite functions, i.e. $\kappa(x, x') \geq 0, \forall x, x' \in \mathcal{X}$. One common used and simple kernel is the Gaussian kernel:

$$\kappa(x, x') = \exp\left(-\frac{\|x - x'\|_2^2}{2l^2}\right),\tag{2.9}$$

where l is the length scale parameter. This kernel is infinitely differentiable, which implies that GPs with this kernel as covariance function have mean square derivatives of all orders, and are thus very smooth. Another common used kernel is the Matern kernel:

$$\kappa(x, x') = \frac{1}{\Gamma(\nu)2^{\nu-1}} \left(\frac{\sqrt{2\nu}}{l} \|x - x'\|_2\right)^\nu K_\nu\left(\frac{\sqrt{2\nu}}{l} \|x - x'\|_2\right),\tag{2.10}$$

where $K_\nu(\cdot)$ is a modified Bessel function and $\Gamma(\cdot)$ is the gamma function. The parameter ν controls the smoothness of the learned function. The smaller ν , the less smooth the approximated function will be. The most common choice for ν is $[0.5, 1.5, 2.5, \infty]$

2.2.2 Acquisition Function

The acquisition function predicts the new point x that would maximize or minimize the objective function, based on the posterior distribution over f provided by the GP model. There are different acquisition functions for different purposes, like Expected Improvement (EI), Upper Confidence Bound (UCB), Entropy Search (ES) and Knowledge

Gradient (KG). Here we give a brief introduction to EI and UCB for their wide usage. Without loss of generality, we now assume that we want to maximize our objective function.

The EI method seeks the point that has the maximum expected improvement. Denote the optimal point among the existing n observations is (x_n^*, f_n^*) , EI chooses the point that maximizes the *expected improvement*:

$$EI_n(x) := \mathbb{E}_n [[f(x) - f_n^*]^+], \quad (2.11)$$

where $[a]^+ = \max(a, 0)$ indicates the positive part, and $\mathbb{E}_n$ indicates the expectation taken under the posterior distribution predicted by the GP in (2.8).

The UCB method is more optimistic in face of uncertainty. It chooses the point that maximizes the upper bound:

$$UCB_n(x) := \mu_n(x) + \alpha \sigma_n(x), \quad (2.12)$$

where α is a weight parameter, $\mu_n(x)$ and $\sigma_n(x)$ are mean and covariance functions as shown in (2.8).

2.3 Neural Networks

Artificial neural networks are biologically inspired computational frameworks, which consist of a series of connected groups of neurons known as layers. Neural networks are a powerful learning model. It has been proven that with enough data and neurons, a neural network can approximate any function [31].

2.3.1 Multi-Layer Perceptron

The multi-layer perceptron (MLP) should be the most classical neural network structures. MLPs consist of input layer, hidden layers and output layer. In the input layer, neurons' values are determined by inputs. In the hidden layer and output layer, each neuron's value is a summation of the previous layer with the weights of the current layer under a non-linear (usually for hidden layer) or linear (usually for output layer) activation function g . The non-linearity of hidden layer activation functions is the key of the MLP's learning capability. The value of a neuron of a hidden layer l is given by:

$$h_i^l = g\left(\sum_j w_{ij}^l h_j^{l-1}\right) = g(W^l h^{l-1}), \quad (2.13)$$

where w_{ij}^l is the weight for the connection between the neuron i in layer l and the neuron j in layer $l - 1$.

The training of MLP is usually achieved through minimizing a loss function via the backpropagation algorithm [32]. A loss function is defined by the final output layer $\hat{y}$ and the target value prediction y given by the MLP model. As the output layer is a composite differentiable function of model weights, weights can be updated by differentiating the loss function with respect to (w.r.t) the weights using the chain rule.

2.3.2 Activation Functions

Many different non-linear activation functions have been proposed since the advent of neural networks. They have different advantages and disadvantages, and are used in different scenarios. Here we give a brief introduction to the most common activation functions.

The most popular activation functions through the history of neural networks are probably the *tanh* and *sigmoid* functions, which are given by the following equations:

$$\begin{aligned} g(x) = \tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}}, \\ g(x) = \text{sigmoid}(x) &= \frac{1}{1 + e^{-x}}. \end{aligned} \tag{2.14}$$

The tanh and sigmoid functions constrain the output value to be within the boundaries $[0, 1]$ and $[-1, 1]$ respectively. This would lead to vanishing gradients for deep neural networks [33]. This is because that gradients of these two functions are constrained within $[-1, 1]$. When propagating gradients via the chain rule, these gradients may decay quickly towards 0 and the neural network will have few updates.

The Rectified Linear Unit (ReLU) can solve the vanishing gradient problem, and may be the current most popular activation choice in deep learning [33]. It is given by the following equation:

$$g(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ x, & \text{if } x > 0 \end{cases} \tag{2.15}$$

Because ReLU does not bound output values, the vanishing gradient problem can be alleviated. Meanwhile, when the input $x \leq 0$, the output of ReLU will always be zero. This brings sparsity to the neural network, which can further increase the network's learning ability [33].

2.4 Reinforcement Learning

Reinforcement learning (RL) is a multidisciplinary area combining methods from neuroscience, computer science, control theory and economy, where an agent learns to interact with an environment by taking actions and receiving rewards. RL has achieved remarkable progress since 1950s. It has beaten professional human players in checkers, Atari-2600 games, GO and most recently OpenAI's DOTA II [34]. However, most of the RL achievements are limited to video games and simulation environments. This is because most of RL algorithms are sample-inefficient. It is common for RL algorithms to take millions of training steps to effectively learn to interact with its environment, which is almost impossible in the domain of robotics. Therefore, as mentioned above, researchers usually train a policy in simulation environments and then utilize sim2real transfer techniques to transfer the policy to real robots.

2.4.1 Fundamentals

We present a brief introduction to RL algorithms here. For a more detailed and accurate tutorial, please see [35] as a reference. Let $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, p_0, \gamma, T)$ be a finite-horizon Markov Decision Process (MDP), where $\mathcal{S}$ and $\mathcal{A}$ are state and action spaces, $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}_+$ is a state-transition probability function or probabilistic system dynamics, $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ a reward function, $p_0 : \mathcal{S} \rightarrow \mathbb{R}_+$ an initial state distribution, γ a reward discount factor, and T a fixed horizon. The MDP has the Markov chain property which means that the next state s_{t+1} is only dependent on the current state s_t :

$$p(s_{t+1}|s_0, s_1, \dots, s_t) = p(s_{t+1}|s_t). \quad (2.16)$$

Let $\tau = (s_0, a_0, \dots, s_T, a_T)$ be a trajectory of states $s_i \in \mathcal{S}$ and actions $a_i \in \mathcal{A}$, and let $R(\tau) = \sum_{t=0}^T \gamma^t R(s_t, a_t)$ be the trajectory reward. The goal of reinforcement learning methods is to find parameters θ of a policy $\pi_\theta(a|s)$ that maximize the expected discounted reward over trajectories induced by the policy:

$$\theta^* = \arg \max_{\theta} \mathbb{E}_{p_0, P, \pi_\theta} [R(\tau)], \quad (2.17)$$

where $s_0 \sim p_0$, $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$ and $a_t \sim \pi_\theta(a_t|s_t)$.

The value function for a given state s under the policy π is given by the expected future discounted reward starting at s and following π until the final state. Similarly the state-action value function, or Q function, is given by the expected future discounted

reward from s taking action a then following π after. The value function and Q function are given by:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[R_t | s_t = s] = \mathbb{E}_\pi\left[\sum_{t'=t}^T \gamma^{t'-t} r_{t'} | s_t = s\right] \\ Q^\pi(s, a) &= \mathbb{E}_\pi[R_t | s_t = s, a_t = a]. \end{aligned} \quad (2.18)$$

For known MDPs with small discrete state and action spaces, it is possible to use dynamic programming to solve the MDP and obtain the global optimal policy. Unfortunately, most of real-world problems does not fall into this category. Therefore, modern RL algorithms usually use deep neural networks or other learning models to approximate certain components of MDPs. Based on this, RL algorithms can be divided into three categories: value-function-based, policy-gradient-based and actor-critic¹.

For value-function-based algorithms, deep neural networks are trained to approximate the true value function $V^\pi(s)$ or Q function $Q^\pi(s, a)$, and the optimal policy is obtained via greedy policy:

$$\begin{aligned} \pi^*(s_t) &= \arg \max_{a_t} \mathbf{E}_{P(s_{t+1}|s_t, a_t)} V^\pi(s_{t+1}), \quad \text{or} \\ \pi^*(s_t) &= \arg \max_{a_t} Q^\pi(s_t, a_t). \end{aligned} \quad (2.19)$$

For policy-gradient-based algorithms, deep neural networks are used to directly approximate the policy. This kind of RL algorithms updates policy parameters w.r.t the expected trajectory return by gradient descent. Actor-critic algorithms are a combination of the above two, where neural networks are used to learn both value functions and policies. Note that some policy-gradient-based algorithms can also use neural networks or other models as an approximation of value functions, and they are actually actor-critic algorithms then. So there may not be a clear boundary between policy-gradient-based algorithms and actor-critic algorithms.

2.4.2 Representative Algorithms

Now we present several representative RL algorithms. The first one is the Double Deep Q-function Network (DDQN), which is a popular value-function-based algorithm [36]. DDQN is aimed at addressing the overoptimism problem of Q-value networks, where Q-value networks tend to overestimate the expected reward of all actions. To solve this problem, DDQN utilizes two separate Q-value networks, each of which is used to update the other. Using these independent networks, DDQN can have unbiased Q-value

¹This is not an accurate definition and division. For a more precise definition, please see [35] as a reference.

estimates of the actions selected using the opposite Q-value network. DDQN achieved human-level performance in Atari games, but it is restricted to discrete state and action space.

Proximal Policy Optimization (PPO) [37] is a representative policy-gradient-based algorithm. It optimizes a surrogate loss function that encourages the policy to choose actions at each state that generate the highest reward, relative to the expected reward of said state. Additionally PPO enforces a clipping constraint on the update such that each update on the policy does not change the policy too much.

The last one is the Twined Delayed Deep Deterministic Policy Gradient (TD3) [38], which should be the state-of-the-art actor-critic algorithm. TD3 specifically addresses a common error mode in previous actor-critic algorithms where the Q-value network begins to overestimate the expected return from a certain action. During training, this would lead the agent to select the overestimated action only, which may cause a policy breakdown. TD3 uses both a twin set of Q-value networks and delayed policy updates to reduce this overestimation problem, resulting in significant improvements in the learning performance.

2.4.3 Sim2real Transfer

Sim2real transfer is aimed at minimizing the difference between the simulation and the real world. And the most important difference may lie in the system dynamics, or the state-transition probability function P . In general, simulation dynamics will not be exactly the same as the real world dynamics. Inaccurate simulation model parameters, unmodeled dynamics and stochastic noise would make simulated robot behaviors deviate from real-world trajectories. Therefore, policies trained in simulation may perform worse or even fail in the real world. In general, the simulation dynamics model is a known parameterized model or a black-box model, with adjustable parameters ξ . Denote the simulation dynamics as $P_\xi = P(s_{t+1}|s_t, a_t, \xi)$, while the real world dynamics is denoted as P_{real} . The goal of sim2real transfer is to find policy parameters θ and simulation parameters ξ that maximize the expected real-world trajectory rewards of a policy which is trained using the simulation dynamics. This can be expressed in a bi-level formulation:

$$\begin{aligned} \xi^* &= \arg \max_{\xi} \mathbb{E}_{p_0, P_{real}, \pi_{\theta^*(\xi)}}[R(\tau)] \quad \text{with,} \\ \theta^*(\xi) &= \arg \max_{\theta} \mathbb{E}_{p_0, P_\xi, \pi_\theta}[R(\tau)]. \end{aligned} \tag{2.20}$$

This method uses a point estimate of simulation parameters, and thus is referred to *point estimate* method. Besides this, there are methods that use a probability density estimate $\xi \sim p_\phi$, where ϕ is the probability function parameters. This method is called *domain randomization*, and it can be expressed in the following formulation:

$$\begin{aligned}\phi^* &= \arg \max_{\phi} \mathbb{E}_{p_0, P_{real}, \pi_{\theta^*(\phi)}}[R(\tau)] \quad \text{with,} \\ \theta^*(\phi) &= \arg \max_{\theta} \mathbb{E}_{p_0, P_{\xi \sim p_\phi}, \pi_\theta}[R(\tau)].\end{aligned}\tag{2.21}$$

During the training phase, instead of a single environment, the policy needs to interact with multiple environments sampled from p_ϕ simultaneously. Therefore, policies are encouraged to yield actions that can achieve better average rewards across different environments. This will make policies more robust to the discrepancy between the simulation world and the real world, but also more conservative and computationally expensive. Because we have an extra sampling of simulation parameters from the distribution now.

The optimization problem in (2.20) and (2.21) would entail training and real robot evaluation of the policy $\pi_{\theta(\phi)}$ for each ξ or ϕ . This would require a large amount of reinforcement learning iterations and, more critically, real robot trials. Typical sim2real transfer methods solve this problem in an iterative way [28, 39, 40]. Their common framework is summarized in Alg. 1 and visualized in Fig. 2.1, using the point estimate method as an example. In this framework, for the i -th iteration, we have an acquisition policy² π_{a_i} that is responsible for collecting trajectories τ_{real}^{acq} in the real world. These trajectories, along with our current prior knowledge of simulation parameter ξ_i , are fed into a simulation optimizer $SimOpt(\cdot)$ to obtain the new estimate ξ_{i+1} . Then we use the policy optimizer $PolicyOpt(\cdot)$ to train our task policy $\pi_{\theta_{i+1}}$ with simulation environments parameterized by ξ_{i+1} . This newly trained policy $\pi_{\theta_{i+1}}$ is then tested in the real world to get the task performance $R[\tau_{real}^{task}]$. Finally, we can update our acquisition policy $\pi_{a_{i+1}}$ if necessary.

²This is different from the acquisition function in the Bayesian optimization.

Algorithm 1 A generic sim2real transfer algorithm

Input: initial acquisition policy π_{a_0} , initial task policy π_{θ_0} , initial simulation distribution parameter ξ_0 , simulation optimizer $SimOpt(\cdot)$, policy optimizer $PolicyOpt(\cdot)$.

Output: task policy π_{θ_N} , simulation parameter ξ_N .

- 1: **for** iteration $i \in \{0, 1, 2, \dots, N\}$ **do**
- 2: Roll out acquisition policy π_{a_i} with real-world dynamics P_{real} , and obtain acquisition trajectory τ_{real}^{acq}
- 3: Estimate simulation distribution parameter: $\xi_{i+1} \leftarrow SimOpt(\xi_i, \tau_{real}^{acq})$
- 4: Update task policy parameter: $\theta_{i+1} \leftarrow PolicyOpt(\theta_i, \xi_{i+1})$
- 5: Roll out $\pi_{\theta_{i+1}}$ with real-world dynamics, obtain real-world task performance $R[\tau_{real}^{task}]$
- 6: Update the acquisition policy to obtain $\pi_{a_{i+1}}$
- 7: **end for**

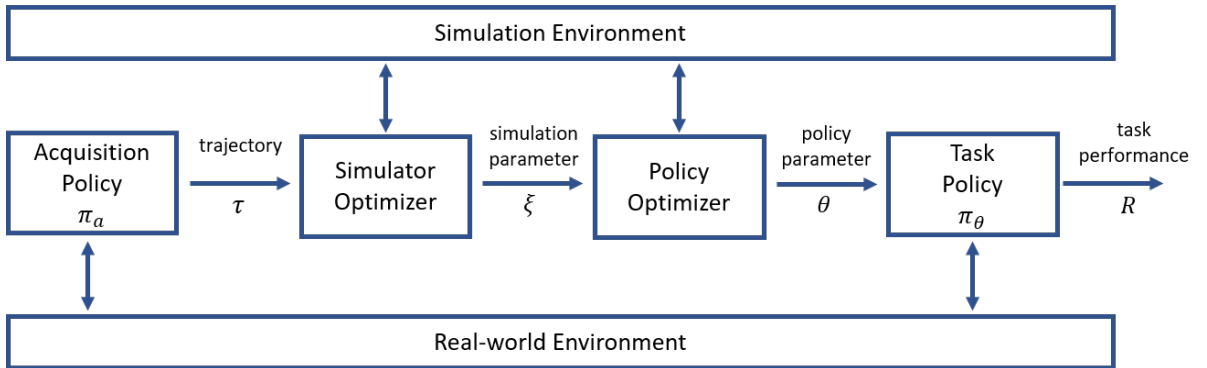


Figure 2.1: A generic domain randomization algorithm framework.

Chapter 3

Modular Framework

This chapter presents the modular framework, which consists of a modified Kalman filter, a bi-level motion planer and a learning-based low-level controller. We first present the problem statement and the methodology of this framework, then implementation details and experiment results.

3.1 Problem Statement

3.1.1 System Model

Previous sections present general kinematics and dynamics models of mobile manipulators, and here we show the robot model used in this thesis. It is a 6-DoF manipulator rigidly mounted on a 3-DoF omnidirectional mobile base, as shown in Fig. 3.1. The yaw rotation of the mobile base is not used in this work since it is too slow for the application at hand. Thus the system has 8 DoF in total. We refer to these DoF as robot joints, for brevity. The joint configuration vector of the robot is $\mathbf{q} = [\mathbf{q}_a^T, \mathbf{q}_b^T]^T$, where $\mathbf{q}_a = [\theta_1, \theta_2, \dots, \theta_6]^T$ is a vector containing the arm joint angles and $\mathbf{q}_b = [x_b, y_b]^T$ represents the Cartesian position of the mobile base. The forward kinematics functions $R_n^0(\mathbf{q}), o_n^0(\mathbf{q})$ that calculates the orientation and position of the end effector are assumed to be known.

The flying ball's motion is modeled as a point mass under the influence of aerodynamic drag and gravity. The aerodynamic drag is proportional to the square of the speed [41]. The ball's equation of motion can be written as follows:

$$\ddot{\mathbf{b}} = -\mathbf{g} - K_D \|\dot{\mathbf{b}}\| \dot{\mathbf{b}}, \quad (3.1)$$

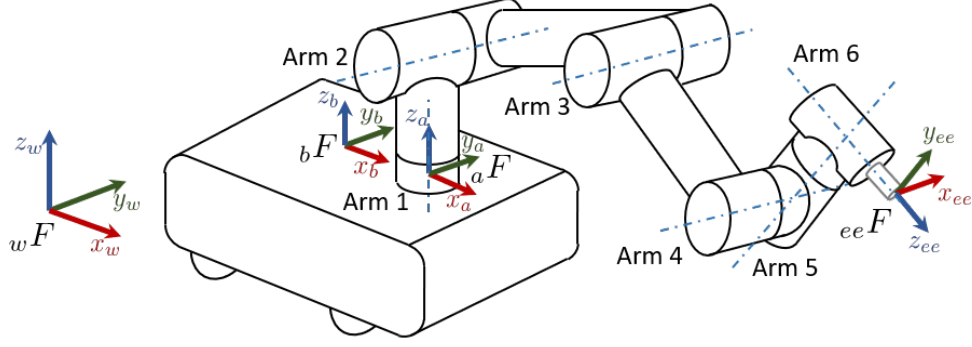


Figure 3.1: The mobile manipulator with the world frame ${}_wF$ (or $o_0x_0y_0z_0$), base frame ${}_bF$ (or $o_2x_2y_2z_2$), arm frame ${}_aF$ (or $o_3x_3y_3z_3$) and end effector frame ${}_{ee}F$ (or $o_8x_8y_8z_8$). Rotation axes of the six arm joints are also presented. Adapted from [1].

where $\mathbf{b} = [b_x, b_y, b_z]^\top$ is the ball's Cartesian position in the world frame and $\dot{\mathbf{b}}, \ddot{\mathbf{b}}$ are the corresponding velocity and acceleration, $\mathbf{g} = [0, 0, g]^\top$ ($g = 9.81 \text{ m/s}^2$) is the gravitational acceleration, K_D is the aerodynamic drag coefficient, and $\|\cdot\|$ is the Euclidean norm.

3.1.2 Problem Statement

Given initial robot configuration $\mathbf{q}_0$, ball state $[\mathbf{b}_0^T, \dot{\mathbf{b}}_0^T]^T$, and time t_0 , the goal is to catch the ball with the robot's end effector at a certain catch time $t_f > t_0$. To achieve this, two conditions need to be satisfied: 1) at catch time t_f , the origin of the end effector frame should coincide with the ball's position; and 2) the z axis of the end effector frame should be aligned with the ball's velocity vector. Formally, these constraints can be written as:

$$\begin{aligned} {}_w\mathbf{P}_{ee}(\mathbf{q}_f) &= \mathbf{b}(t_f) \\ \cos(\angle {}_w\mathbf{z}_{ee}(\mathbf{q}_f), -\dot{\mathbf{b}}(t_f)) &= 1, \end{aligned} \quad (3.2)$$

where $\mathbf{q}_f$ is the robot joint configuration at catch time t_f , ${}_w\mathbf{P}_{ee}$ is the position of the end effector frame's origin in the world frame, ${}_w\mathbf{z}_{ee}$ is the vector of the z axis of the end effector in the world frame. ${}_w\mathbf{P}_{ee}$ and ${}_w\mathbf{z}_{ee}$ can be obtained by the forward kinematics function $o_n^0(\mathbf{q}), R_n^0(\mathbf{q})$ respectively. The operator $\angle \cdot, \cdot$ calculates the angle between two vectors.

3.2 Methodology

This section presents the methodology of the proposed modular framework, which includes three major components: (i) ball estimation and prediction, (ii) bi-level motion

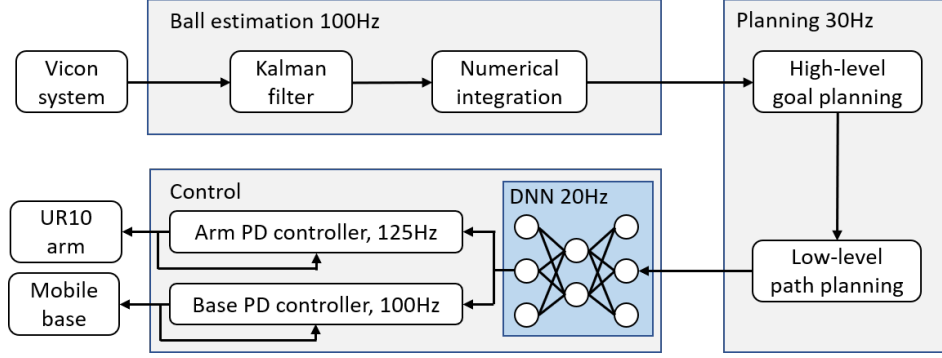


Figure 3.2: Diagram of the overall framework for accurate high-speed ball catching. Note that the estimation module shown here is for ball state estimation, with separate, but not shown estimation modules for arm and base controllers.

planning, and (iii) low-level control with inverse dynamics learning. The system diagram is shown in Fig. 3.2.

3.2.1 Ball Estimation and Prediction

The first component estimates the ball state and uses it to generate trajectory predictions within a finite horizon. To estimate the ball state $\mathbf{s}_b = [\mathbf{b}^T, \dot{\mathbf{b}}^T]^T$, we use a modified discrete-time Kalman filter that considers the influence of air drag in the state prediction step, as presented in [41]. In the state prediction step at the discrete time index k , we predict the ball state $\mathbf{s}_b[k+1|k]$ at time step $k+1$ given measurements up to time k according to the estimated state at the previous time step $\mathbf{s}_b[k|k]$ and motion model in (3.1) as follows:

$$\begin{aligned}\ddot{\mathbf{b}}[k|k] &= -\mathbf{g} - K_D \|\dot{\mathbf{b}}[k|k]\| \dot{\mathbf{b}}[k|k] \\ \dot{\mathbf{b}}[k+1|k] &= \dot{\mathbf{b}}[k|k] + \delta_k \ddot{\mathbf{b}}[k|k] \\ \mathbf{b}[k+1|k] &= \mathbf{b}[k|k] + \delta_k \dot{\mathbf{b}}[k|k] + \frac{1}{2} \delta_k^2 \ddot{\mathbf{b}}[k|k],\end{aligned}\tag{3.3}$$

where δ_k is the time interval between time index $k+1$ and k . The update step and variance propagation and update are the same as the vanilla Kalman filter [42].

An accurate estimation of the aerodynamic drag coefficient K_D is critical for the above ball state estimation method. Analytical methods can calculate drag coefficient for an ideal sphere. However, balls will not be perfect spheres, and their uneven surface will have complex influence on the drag coefficient. Therefore, we use a Recursive Least-square estimator [41] to estimate the drag coefficient through experiments.

Taking the time derivative of the ball's mechanical energy $E_b = \frac{1}{2} \|\dot{\mathbf{b}}\|^2 + \mathbf{g}^T \mathbf{b}$, this

yields the specific power extracted from the system via the aerodynamic drag force:

$$\dot{E}_b = -K_D ||\dot{\mathbf{b}}||^3. \quad (3.4)$$

If we use the ball state estimate $\mathbf{s}_b[k+1|k+1]$ from the ball state Kalman filter to calculate the ball's speed $\bar{v}[k+1]$, then (3.4) can be approximated as a measurement for $K_D[k+1|k]$:

$$\begin{aligned} \bar{v}[k+1] &= \sqrt{\mathbf{s}_b[k+1|k+1]^T \begin{bmatrix} \mathbf{0}_{3 \times 3} & \mathbf{0}_{3 \times 3} \\ \mathbf{0}_{3 \times 3} & \mathbf{I}_{3 \times 3} \end{bmatrix} \mathbf{s}_b[k+1|k+1]} \\ \bar{E}[k+1] &= \frac{1}{2}(\bar{v}[k+1])^2 + \begin{bmatrix} \mathbf{g}^T & \mathbf{0}_{1 \times 3} \end{bmatrix} \mathbf{s}_b[k+1|k+1] \\ K_D[k+1|k] &= -\frac{\bar{E}[k+1] - \bar{E}[k]}{\delta_k(\frac{1}{2}(\bar{v}[k+1])^2 + \bar{v}[k])^3}, \end{aligned} \quad (3.5)$$

where $\bar{E}[k+1]$ is the estimate of the ball's mechanical energy at time step k , and a two step average of the ball's speed is used here. These measurements can now be used to form an estimate of the drag value $K_D[k+1|k+1]$ using a recursive least squares estimator. We denote $R_D[k]$ as the coefficient measurement noise variance, $P_D[k] = \text{var}(K_D[k|k])$ as the estimate variance, and we introduce an intermediate gain $C_D[k]$:

$$\begin{aligned} R_D[k+1] &= \frac{R_{D,0}}{(\frac{1}{2}(\bar{v}[k+1])^2 + \bar{v}[k])^3} \\ C_D[k+1] &= \frac{P_D[k]}{P_D[k] + R_D[k+1]} \\ K_D[k+1|k+1] &= K_D[k|k] + C_D[k+1](K_D[k+1|k] - K_D[k|k]) \\ P_D[k+1] &= \frac{P_D[k]R_D[k+1]}{P_D[k] + R_D[k+1]}. \end{aligned} \quad (3.6)$$

In experiments, we discard measurements when the ball is below a certain minimum height or when the ball's speed approaches zero for numerical stability. We take the converged air drag coefficient estimate as the final estimation result.

For the trajectory prediction, given the estimated ball state $\hat{\mathbf{s}}_b$ at time t_0 , future ball trajectory predictions are made via numerically integrating (3.1) for a finite horizon. Hence, ball trajectory predictions will change when a new state measurement comes, and a new motion plan should be made. This requires that the motion planner can make new plans in a very short time (typically 20 ms [10]) so that the information of the latest ball trajectory prediction can be used in the latest robot motion plan.

3.2.2 Motion Planning

In order to achieve better performance, the mobile platform trajectory needs to be updated when a new ball trajectory prediction is available. The motion planner is bi-level: 1) the *high-level* planner calculates a feasible and optimal catching configuration $\mathbf{q}_f$ and catch time t_f that enable the end effector to intercept the ball's trajectory at time t_f , and 2) the *low-level* planner generates a smooth joint trajectory $\mathbf{q}_f(t)$ that takes the robot from its current to the desired catch configuration. Moreover, the motion planner must take into account kinematic redundancy, collision avoidance, and robot kinematic and dynamic constraints (e.g. position, velocity and torque constraints). This problem is a nonlinear and non-convex optimization problem [10]. Solving such an optimization problem even off-line is still an open question in the field [7, 10].

To tackle this planning problem in an on-line fashion, we need to make the following three simplifications. Similar simplifications are also made in the ball catching literature and work well in experiments [7, 14, 10, 16].

Simplification 1 (Arm-base collision) *There are no nearby obstacles and no arm self-collision check; hence, no obstacle or self-collision check is needed. The only collision that may happen is the collision between the arm end effector and the mobile base.*

This simplification is reasonable because during robot catching experiments, usually the arm will extend to reach out for a catching. In such a scenario, arm self-collision is rare but collisions between the arm and mobile base are possible. Specifically, to prevent such a collision, we require the arm end effector to stay inside a semi-cylinder centered on the arm's base, as shown in Fig. 3.3:

$$\begin{aligned} {}_aP_{ee,x}^2(\mathbf{q}) + {}_aP_{ee,y}^2(\mathbf{q}) &\leq R_{coll}^2 \\ 0 &\leq {}_aP_{ee,z}(\mathbf{q}) \leq H_{coll}, \end{aligned} \tag{3.7}$$

where $[{}_aP_{ee,x}, {}_aP_{ee,y}, {}_aP_{ee,z}]$ is the end effector's position in the arm frame which can be calculated via forward kinematics functions given the robot configuration $\mathbf{q}$, R_{coll} is the cylinder's radius and H_{coll} is the cylinder's height.

Through experiments, we noticed that as long as the end effector is inside this semi-cylinder space at catch time, the whole generated end effector trajectory usually will stay in that space. Thus constraints in (3.7) is only checked for $\mathbf{q}_f$ instead of every $\mathbf{q}(t)$ for $t_0 \leq t \leq t_f$.

Simplification 2 (Kinematic planning) *We use kinematic models to depict the joint*

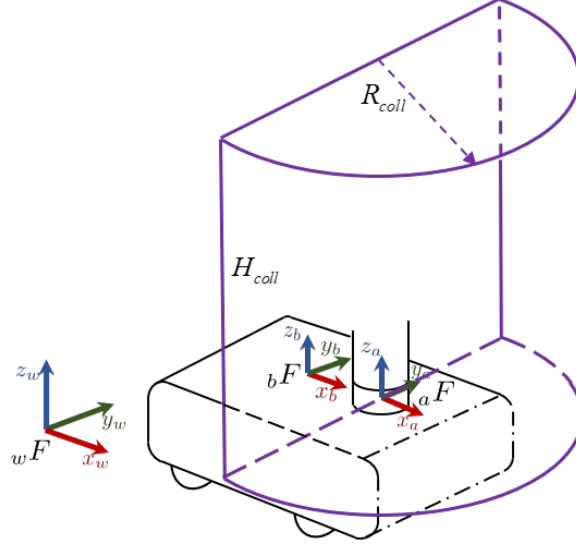


Figure 3.3: The geometric model of the collision-free space. This is a semi-cylinder with its bottom centered in the arm frame's origin.

motion, specify objective functions and constraints on the joint accelerations, instead of torques.

This simplification works in experiments as long as maximum joint acceleration values are chosen conservatively and low-level joint controls work properly. We use a double integrator as the kinematic model of a single joint and take joint acceleration as the control input:

$$\begin{bmatrix} q_i[k+1] \\ \dot{q}_i[k+1] \end{bmatrix} = \begin{bmatrix} 1 & \delta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} q_i[k] \\ \dot{q}_i[k] \end{bmatrix} + \begin{bmatrix} \frac{1}{2}\delta^2 \\ \delta \end{bmatrix} u_i[k], \quad (3.8)$$

where index i represents the i -th joint, δ is the step size¹, and $u_i[k]$ is the control input. To ensure the trajectory feasibility, we have the following box constraints:

$$\begin{aligned} \mathbf{q}_{min} &\leq \mathbf{q}[k] \leq \mathbf{q}_{max}, \\ \dot{\mathbf{q}}_{min} &\leq \dot{\mathbf{q}}[k] \leq \dot{\mathbf{q}}_{max}, \\ \ddot{\mathbf{q}}_{min} &\leq \mathbf{u}[k] \leq \ddot{\mathbf{q}}_{max}, \end{aligned} \quad (3.9)$$

where $\mathbf{q}_{min}$, $\dot{\mathbf{q}}_{min}$, $\ddot{\mathbf{q}}_{min}$, $\mathbf{q}_{max}$, $\dot{\mathbf{q}}_{max}$, $\ddot{\mathbf{q}}_{max}$ are minimum and maximum value of joint positions, velocities and accelerations, respectively.

Simplification 3 (Quadratic cost function) *The cost functions in the high- and low-level planners are quadratic. We use quadratic cost functions because they are popular in*

¹The step size δ used for joint motion equations is not necessarily the same as the step size δ_k used for the Kalman filter

the optimization field and usually easy to deal with.

With these three simplifications, we solve the motion planning problem in a bi-level fashion. The high-level planner selects the final catching configuration $\mathbf{q}_f$ and catch time t_f , while including the nonlinear non-convex constraints in (3.2) and (3.7). Given the solution $\mathbf{q}_f^*, t_f^*$, the low-level path planner determines the intermediate trajectory waypoints $\mathbf{q}(t_k), t_0 < t_k < t_f$, but is restricted only by the linear constraints in (3.8) and (3.9). This is a quadratic program that can be solved very efficiently. To ensure that the optimal values $(\mathbf{q}_f^*, t_f^*)$ calculated by the high-level planner are feasible for the low-level planning problem, a box constraint on $\mathbf{q}_f$ which depends on t_f and initial robot conditions is defined.

Compared to the parameterization methods used in literature for the low-level planning problem, like trapezoidal velocity profiles [7, 14] and third-order polynomials [6], the QP technique we use allows more flexible trajectories due to the increased number of decision variables. Furthermore, via constraints of kinematic models in the quadratic program formulation, the generated trajectories are smoother and should be closer to the robot's actual behaviors when compared to trapezoidal velocity profiles and third-order polynomials. Details on the high- and low-level planners follow.

High-level goal planner

Given the initial robot configuration and velocity $(\mathbf{q}_0, \dot{\mathbf{q}}_0)$, ball position and velocity trajectory prediction $(\mathbf{b}(t), \dot{\mathbf{b}}(t))$ and initial time t_0 , the high-level planner aims to find the optimal final catching configuration and catch time $(\mathbf{q}_f^*, t_f^*)$. Formally, this can be written as the following optimization problem:

$$\begin{aligned}
 (\mathbf{q}_f^*, t_f^*) = & \arg \min_{\mathbf{q}_f, t_f} \Gamma_h(\mathbf{q}_f, t_f) \\
 \text{s.t. } & {}_a P_{ee,x}^2(\mathbf{q}_f) + {}_a P_{ee,y}^2(\mathbf{q}_f) \leq R_{coll}^2, \\
 & 0 \leq {}_a P_{ee,z}(\mathbf{q}_f) \leq H_{coll}, \\
 & {}_w \mathbf{P}_{ee}(\mathbf{q}_f) = \mathbf{b}(t_f) \\
 & \cos(\angle({}_w \mathbf{z}_{ee}(\mathbf{q}_f), -\frac{\dot{\mathbf{b}}(t_f)}{\|\dot{\mathbf{b}}(t_f)\|}) >) = 1 \\
 & |\mathbf{q}_f - \mathbf{q}_0| \leq \Delta \mathbf{q}(\mathbf{q}_0, \dot{\mathbf{q}}_0, t_f - t_0),
 \end{aligned} \tag{3.10}$$

where Γ_h is a user-defined quadratic cost function for the high-level planner, and $\Delta \mathbf{q}(\mathbf{q}_0, \dot{\mathbf{q}}_0, t_f - t_0)$ is the box constraint on $\mathbf{q}_f$ given initial joint position, velocity and duration time.

Accurately calculating $\Delta \mathbf{q}(\mathbf{q}_0, \dot{\mathbf{q}}_0, t_f - t_0)$ requires calculating the optimal intermediate waypoints $\mathbf{q}(t)$ between $\mathbf{q}(t_0)$ and $\mathbf{q}(t_f)$ under kinematic constraints (3.10) for every catch time guess t_f need to be calculated. This is too time-consuming and an approximation is needed. In this paper, we use the following approximation:

$$\Delta q_i(q_{0,i}, \dot{q}_{0,i}, \Delta t) = \lambda \left(\dot{q}_{max,i} \left(\Delta t - \frac{\dot{q}_{max,i} - \dot{q}_{0,i}}{\ddot{q}_{max,i}} \right) + \frac{\dot{q}_{max,i}^2 - \dot{q}_{0,i}^2}{2\ddot{q}_{max,i}} \right), \quad (3.11)$$

where $\Delta t = t_f - t_0$ is the duration time, $\dot{q}_{max,i}, \ddot{q}_{max,i}$ are the maximum velocity and acceleration of the i -th joint, $\lambda \in (0, 1)$ is a proportion parameter that needs to be tuned. The underlying assumption is that the maximum joint movement is proportional to the distance that the joint can traverse under the trapezoidal velocity profile [7]. Namely the joint first accelerates to its maximum speed and holds that speed until the end. We choose this approximation because of its simplicity and few corner cases needed to be considered in implementations. Other trajectory profiles like third-order polynomial can also be used for approximation [6]. The SQP algorithm is used to solve the problem specified in (3.10).

Low-level path planner

Given the desired configuration $\mathbf{q}_f^*$ and catch time t_f^* from the high-level goal planner, the task of the low-level path planner is to generate a sequence of intermediate trajectory waypoints for each joint.

For the i -th joint, this problem can be formulated as:

$$\begin{aligned} \mathbf{U}_i^* &= \arg \min_{\mathbf{U}_i} \Gamma_l(\mathbf{P}_i, \mathbf{U}_i) \\ \text{s.t.} \quad & \begin{bmatrix} q_i[k+1] \\ \dot{q}_i[k+1] \end{bmatrix} = \begin{bmatrix} 1 & \delta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} q_i[k] \\ \dot{q}_i[k] \end{bmatrix} + \begin{bmatrix} \frac{1}{2}\delta^2 \\ \delta \end{bmatrix} u_i[k] \\ & q_{min,i} \leq q_i[k] \leq q_{max,i} \\ & \dot{q}_{min,i} \leq \dot{q}_i[k] \leq \dot{q}_{max,i} \\ & \ddot{q}_{min,i} \leq u_i[k] \leq \ddot{q}_{max,i} \\ & q_i[0] = q_{0,i}, \dot{q}_i[0] = \dot{q}_{0,i} \\ & q_i[K] = q_{f,i}^*, \dot{q}_i[K] = 0, \end{aligned} \quad (3.12)$$

where Γ_l is a user-defined quadratic cost function for the low-level planner, $K = \text{floor}((t_f^* - t_0)/\delta)$ is the finite horizon and $\text{floor}(x)$ returns the greatest integer less than or equal to x , $\mathbf{P}_i = [q_i[1], \dots, q_i[K]]^T$ and $\mathbf{U}_i = [u_i[1], \dots, u_i[K]]^T$ are the position and control

input sequence of the i -th joint.

This is an optimal control problem that consists of quadratic cost functions and linear constraints. Standard quadratic programming methods can be used to solve this problem efficiently [43]. With the solution of optimal control input sequence $\mathbf{U}_i$ and initial conditions $q_i[0] = q_{0,i}$, $\dot{q}_i[0] = \dot{q}_{0,i}$, the optimal position sequence $\mathbf{P}_i$ can be constructed via the kinematic model in (3.8).

3.2.3 Low-level Joint Control

Given the desired position trajectories generated by the motion planners, the task of the low-level joint controllers is to accurately track the trajectory. Accurate trajectory tracking is challenging due to the high-speed joint trajectories required for successful ball catching. Moreover, joint trajectory replanning may introduce discontinuities into the desired trajectories, which may be harder to track. To overcome these problems, we use the inverse dynamics learning technique developed in [2], shown in Fig. 3.4. It uses DNNs as an add-on block to baseline feedback control systems to achieve a unity map between desired and actual outputs.

This method learns an inverse dynamics model of the baseline closed-loop system: $\hat{y}[k] = f(x[k], x[k+r])$, where $x[k]$ is the system's discrete state at time index k , $\hat{y}[k]$ is the reference signal to the baseline control system, and r is the relative degree [2]. For discrete-time systems, the vector relative degree can be intuitively understood as the number of time steps before a change in the input is reflected in the output, which can be easily determined experimentally from the system's step response [2].

Specifically, for a single robot joint of the mobile manipulator, the DNN's inputs and outputs are:

$$\hat{y}_j^d[k+1] = NN(q[k], y_j^d[k+r]), \quad (3.13)$$

where $y_j^d[k]$ is the reference signal sampled from desired trajectories for the j -th joint at time index k , and $\hat{y}_j^d[k+1]$ is the DNN's modified reference signal. As mentioned in [2], subtracting $y_j^d[k+1]$ from both the inputs and outputs of the neural network can improve the DNN's generalization ability since only the reference signal offsets $\tilde{y}_j^d[k+1] = \hat{y}_j^d[k+1] - y_j^d[k+1]$ instead of the reference signal itself $\hat{y}_j^d[k+1]$ are learned:

$$\tilde{y}_j^d[k+1] = NN(q[k] - y_j^d[k+1], y_j^d[k+r] - y_j^d[k+1]). \quad (3.14)$$

The modified position reference signal can then be sent to the baseline control system to actuate the robot joint.

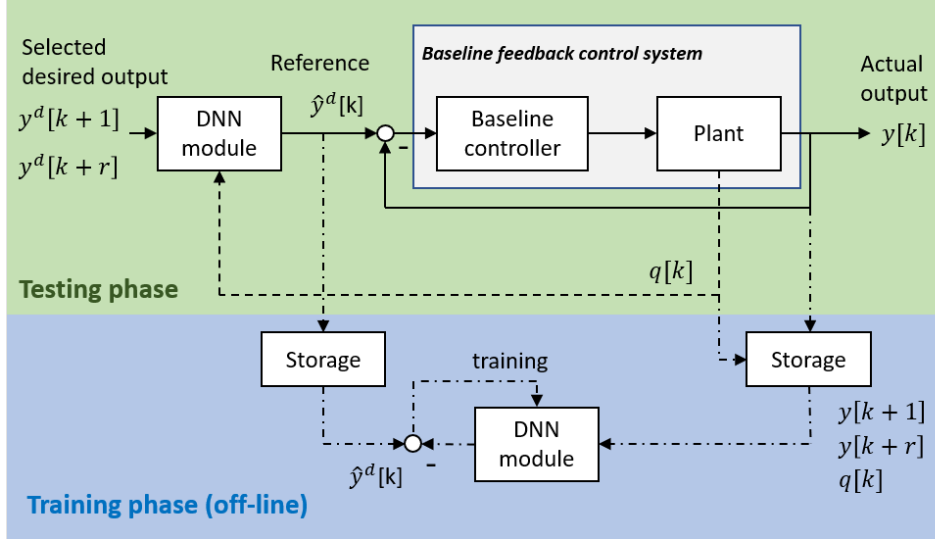


Figure 3.4: A diagram of the inverse dynamics learning framework and the training phase. The DNN module modifies the reference signal to the baseline system for trajectory tracking performance improvement. Adapted from [2].

3.3 Framework Implementation and Evaluation

This section presents implementation details and evaluation of each of the three framework components.

3.3.1 Ball Estimation and Trajectory Prediction

First, we describe the implementation details of the ball state estimation and trajectory prediction. The modified Kalman filter, stated in Section 3.2.1, is used to reduce noise in VICON position and velocity measurements. In experiments, VICON position measurements are very precise. Therefore, ball state estimation consists of position measurements from the VICON system and velocity estimation from the Kalman filter. Trajectory prediction is done through numerical integration with a step size of 0.1 ms . Both the Kalman filter (3.3) and trajectory prediction (3.1) need an estimate of the aerodynamic drag coefficient K_D . To estimate it, we record ball state data from 20 random ball tosses and use the recursive least-squares estimator in (3.5). The estimated value of K_D is 0.0238 m^{-1} .

We use the trajectory prediction quality to evaluate the modified Kalman filter and the recursive least-squares estimator, because a precise trajectory prediction relies on both initial state estimate and air drag coefficient estimate. Specifically, we care about the trajectory prediction quality around the expected catching point, because trajectory prediction in other segments of the trajectory will not have an influence on the catching

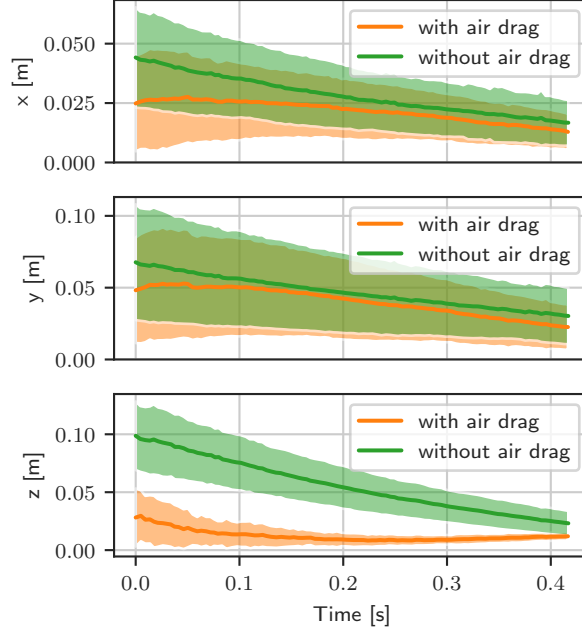


Figure 3.5: The ball trajectory prediction error of the 75 ball catching experiments in Experiment A, described in Sec. 3.4.2. The dashed area represents the standard deviation.

point selection. So we calculate the prediction rooted-mean-squared error (RMSE) of the last 50 time steps before the final expected catching time, and use this as a metric. Note that for every new ball state estimate, we will have a new trajectory prediction, so the prediction error varies among time. Fig. 3.5 shows the prediction error of 75 ball catching experiments in Experiment A, as described in Sec. 3.4.2, when the Kalman filter and trajectory prediction are using and not using the estimated air drag coefficient (i.e. setting $K_D = 0.0$). It shows that using the estimated air drag coefficient can largely reduce the error, especially for the z axis. This validates the necessity and correctness of the air drag coefficient estimator. It also shows that when using air drag coefficient the prediction error is smaller than 0.025 m after 0.4 s , which satisfies the position accuracy requirements of the catching task.

3.3.2 Motion Planning

We now describe the implementation details of the high-level and low-level motion planner. The goal of the high-level planner is to find a catch configuration $\mathbf{q}_f$ and catch time t_f while minimizing the robot movement:

$$\Gamma_h(\mathbf{q}_f, t_f) = w_a \|\mathbf{q}_{a,0} - \mathbf{q}_{a,f}\|_2^2 + w_b \|\mathbf{q}_{b,0} - \mathbf{q}_{b,f}\|_2^2, \quad (3.15)$$

where w_a, w_b are weight parameters that penalize movement of the arm joints and the mobile base, respectively. As mentioned in Sec. 1.2, control accuracy of the mobile base is usually lower than the arm [12, 13]. Therefore, we select $w_a = 1, w_b = 5$ to reduce the mobile base motion. The proportion parameter used in the high-level planner (3.11) is set to be $\lambda = 0.4$. The high-level planner uses the SQP algorithm which we implemented using the NLOpt c++ library [44]. The initial guess for the optimization variables $(\mathbf{q}_f, t_f)$ are the robot's current configuration and a hard-coded catch time guess $t_f = 0.5$ s. While multiple start is used to find a good initial guess in literature [14], our simple initialization scheme works well in experiments.

The goal of the low-level planner is to generate a smooth trajectory from current configuration $\mathbf{q}_0$ to catch configuration $\mathbf{q}_f$. This is achieved by penalizing joint acceleration change from one time step to the next during the trajectory, which can be written as follows for the i -th joint:

$$\Gamma_l(\mathbf{P}_i, \mathbf{U}_i) = \sum_{k=0}^{K-1} \|u_i[k+1] - u_i[k]\|_2^2, \quad (3.16)$$

where K is calculated according to Sec. 3.2.2, and the discretization step size is $\delta = 0.05$ ms. We solve the QP in the low-level planner with qpOASES c++ library [45].

We use numerical experiments to evaluate the motion planner's performance since we cannot control the ball initial state in real experiments when the ball is thrown by a human operator. We generated 2320 initial ball state samples, whose positions are uniformly distributed in a circle around the robot. In order to ensure that the resulting ball trajectories are feasible given the robot's motion constraints, we first note that the end effector can move 0.65 m in 0.86 s, as described in Sec. 3.4.2. We restrict the initial ball states to be such that after 0.7 s the ball is predicted, via numerical integration of (3.1), to be located inside a 0.50 m diameter sphere centered around the end effector of the given robot configuration.

The proposed motion planner can find a trajectory for 89.39% of all ball throws, and this success rate drops to 58.36% if we keep the base static, which shows the necessity of base movement. The average computation time is 5.10 ms, with 4.80 ms used for the high-level nonlinear planner, and 0.3 ms used for the low-level (quadratic program) planner on an Intel Xeon CPU i7-8850H 2.60GHz computer. No sophisticated techniques like multi-threaded programming are used to speed up computation time.

3.3.3 Low-level Control

We now describe the implementation of the learning-based low-level controller. We first estimate experimentally the relative degree r of each joint, which is 8 for arm joints and 14 for mobile base joints.

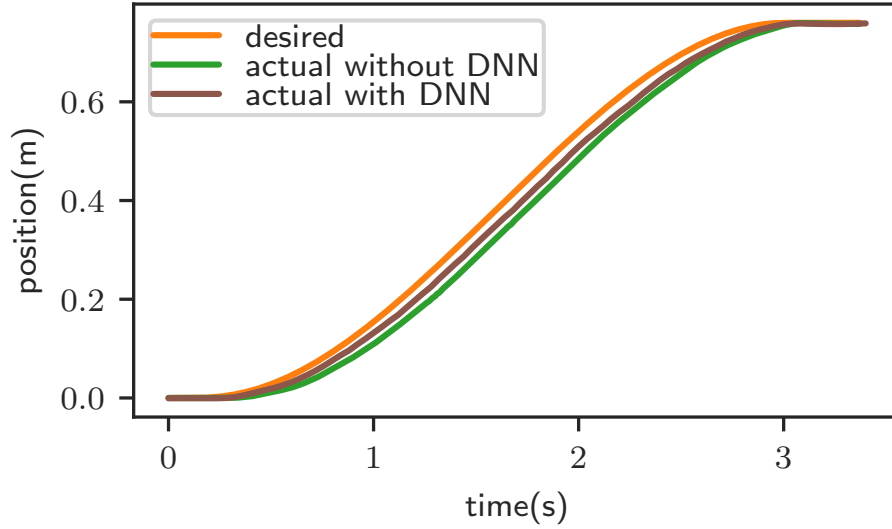
To learn the mapping function in (3.14), we train 8 DNNs, one for each joint. Each DNN consists of two hidden layers and each layer contains 10 neurons. The ReLU function is used as the activation function. To create the training and testing dataset, we use the low-level path planner (described in Sec. 3.2.2) to generate trajectories similar to those used in real ball catching experiments. We characterize these trajectories by bounding the maximum joint displacement $\Delta \mathbf{q}$ in a given amount of time Δt . In our experiments we set $\Delta t = 2s, 3s, 4s$, to represent trajectories with different speeds and $\Delta \mathbf{q} = [1.36, 1.36, 1.36, 1.75, 1.75, 2.53, 0.76, 0.76]^T$, where the first six joint displacements correspond to the arm and are specified in radians, and the last two joint displacements correspond to the mobile base and are specified in meters.

The trajectories are executed using the robot’s baseline control system. It consists of off-board PD controllers that generate desired joint velocities and on-board controllers of the UR10 arm and mobile base that turn desired velocities into joint torques (UR10 arm) or wheel rotation velocities (mobile base). In this way we collect a dataset with 1000 data pairs for each joint. The dataset is divided with 80% is used for training and the rest is used for validation. We use Adam optimizer implemented in *Tensorflow* to train the neural network with 50 training iterations. The trained DNNs are implemented via a light-weight neural network library c++ for real-time performance [46].

To test the influence of inverse dynamics learning on tracking performance, we generate testing trajectories by setting $\Delta \mathbf{q} = [0.8, 0.8, 0.8, 1.1, 1.1, 1.1, 0.20, 0.20]^T$ and $\Delta t = 1$ s, which are different from training and validation trajectories in the dataset. The testing trajectories are executed ten times each when the robot uses the learning technique and ten times each when no learning technique is used. The trajectory tracking RMSE over the ten trajectories is shown in Tab. 3.1. It can be seen that the inverse dynamics learning method can reduce tracking errors between 35% $\sim$ 80% with the highest error reduction corresponding to the mobile base. As an example, Fig. 3.6 shows the tracking performance for the x position of the mobile base when the robot is using and not using the DNN module. It can be seen that the trained DNN module can largely reduce the tracking error.

Table 3.1: Joint trajectory tracking error comparison

Joint	With DNN [$^{\circ}$]	Without DNN [$^{\circ}$]	Reduction
arm 1	0.0218	0.0508	57.2%
2	0.0300	0.0508	40.9%
3	0.0264	0.0506	47.8%
4	0.0309	0.0561	45.0%
5	0.035	0.0562	36.5%
6	0.034	0.0561	37.1%
Joint	With DNN [m]	Without DNN [m]	Reduction
base x	0.0041	0.0196	79.1%
y	0.0047	0.0183	74.0%

Figure 3.6: Tracking performance comparison in the x position of the mobile base.

3.4 Experiments

This section presents the experimental setup and ball catching experimental results. A video demonstrating the real robot experiments can be found at http://tiny.cc/ball_catch.

3.4.1 Experimental Setup

The mobile manipulator used in this experiment consists of a 6-DOF UR10 arm and a 3-DOF Ridgeback omnidirectional mobile base driven by four Mecanum wheels. We use symmetric joint motion boundary values, i.e. $\mathbf{q}_{min} = -\mathbf{q}_{max}$, $\dot{\mathbf{q}}_{min} = -\dot{\mathbf{q}}_{max}$, $\ddot{\mathbf{q}}_{min} =$



Figure 3.7: The 61.76 mm diameter ball and 94.90 mm diameter cup used in this paper. The position accuracy requirement in the x, y, z axes is approximately $(94.90 - 61.76)/2 = 16.57$ mm. The typical speed at the catch time is around 5 m/s, leading to the $16.57/5 \approx 3$ ms time precision requirement.

$-\ddot{\mathbf{q}}_{max}$. The six arm joints have the same maximum joint positions of 180° and accelerations of $458^\circ/s^2$. Their maximum joint velocities are $103^\circ/s$, $103^\circ/s$, $103^\circ/s$, $126^\circ/s$, $126^\circ/s$, and $180^\circ/s$ for arm joint 1 $\sim$ 6 respectively. The maximum positions, velocities and accelerations of the mobile base in the x and y axes are 3m, 1.0m/s, $2.5m/s^2$, respectively. The three-finger end effector grasps a 94.90 mm diameter metal cup that is used to catch the ball, as shown in Fig. 3.7. The proposed framework is implemented via Robot Operation System (ROS) and runs on a ThinkPad P52 laptop. The ball used in these experiments is a standard 61.76 mm tennis ball with a weight of 81.76 g. The ball is wrapped with retro-reflective tape in order to be visible to the VICON motion capture system, which provides position and velocity measurements to be used in the ball state estimation module.

3.4.2 Ball Catching Experiment

We conducted experiments to test three main characteristics of the proposed framework: (i) ball catching success rate, (ii) impact of the learning-based controller, and (iii) impact of QP-based low-level planner.

Experiment A: Ball Catching Success Rate

In this experiment, we use the framework proposed in Sec. 3.2 and throw balls from different locations while the robot has different initial configurations. Three initial robot

configurations are chosen. For each initial robot configuration, we throw the ball from five different locations, five throws for each location, for a total of 75 ball catching experiments. All ball trajectories are shown in Fig. 3.8. Of the 75 ball throws, 64 balls are caught successfully by the robot, which is a 85.33% success rate. The end effector position trajectory tracking RMSE over the 75 experiments are 11.03 *mm*, 14.01 *mm*, 13.25 *mm* for x, y, z axes respectively. Among the successful catches, the maximum end effector displacement in a catch is 0.65 *m* in 0.86 *s* with a mobile base displacement of 0.26 *m* in that catch. The corresponding arm and base position trajectories for this catch are shown in Fig. 3.9a and 3.9b. It can be seen that our mobile base is able to move accurately in both x and y directions while the humanoid in [14] can only move in one direction.

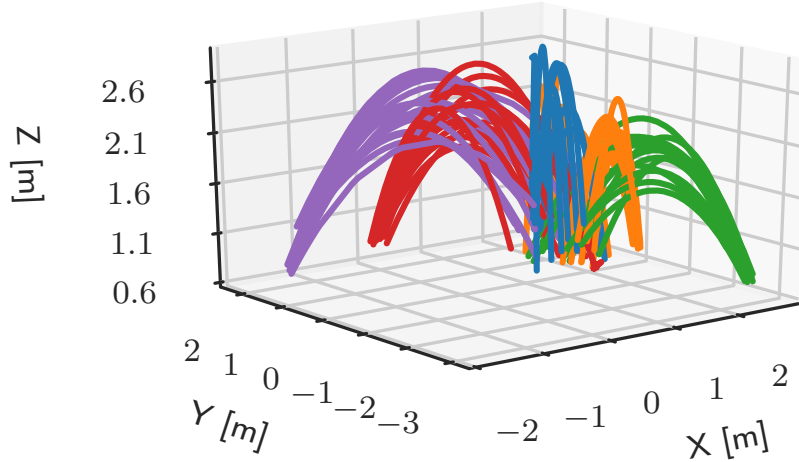


Figure 3.8: Ball trajectories of Experiment A. Starting from the left corner and in the anti-clockwise direction, the five ball throwing positions in the xy plane in meters are $(2.8, -3.1)$, $(2.9, 0.0)$, $(2.5, 2.0)$, $(0.0, 2.8)$, $(-2.0, 2.0)$. Ball trajectories are colored according to its throwing location.

Experiment B: Impact of Learning-based Controller

In this experiment, we are interested in showing that the learning-based controller improves tracking performance, therefore improving the success rate of the overall framework. It is hard to repeat a ball trajectory twice because a human throws it. To do a fair comparison in this experiment we throw balls from the same location $(2.5, 2.0, 0)$ while the robot has the same initial configuration, as shown in Fig. 1.1. We make 40 throws when the robot uses inverse dynamics learning and 40 when the robot does not use inverse dynamics learning for a total of 80 throws. The success rate for this particular

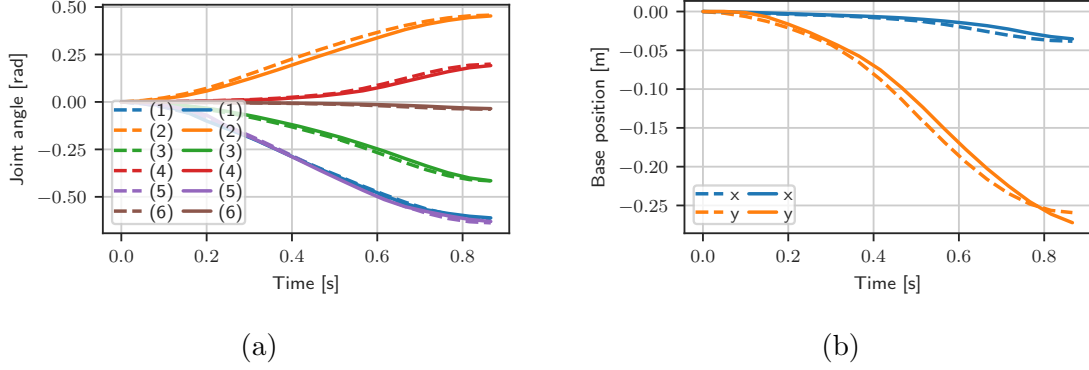


Figure 3.9: An example of (a) arm joint and (b) mobile base tracking performance. Dashed lines represent desired trajectories, while solid lines represent actual trajectories. For a better illustration effect, we subtract initial values from trajectory waypoints, and thus joint angles relative to their initial values are shown here.

configuration is 75% when using inverse dynamics learning and 65% when not using it. The ten percent decrease shows the effectiveness of inverse dynamics learning. Note that the success rate for this particular configuration is lower than the success rate in Experiment A. Visual inspection of the data and video show that the geometrical setup, i.e. ball throwing location and robot initial configuration, used in this experiment requires more robot maneuvers than other geometrical setups used in Experiment A.

Experiment C: Impact of QP-based Low-level Planner

In this experiment, we are interested in showing the impact of the QP-based low-level planner in the ball catching success rate. To do this, we implemented the trapezoidal velocity profile method in [7] for the low-level planner, and tested it by throwing 40 balls with the geometrical setup as in Experiment B. Note that the inverse dynamics learning is still used. The robot achieves a success rate of 72.5%, similar to the success rate of 75% when using the QP method.

The QP method does not have a significant impact on ball catching success rate. One possible reason is that the way the robot accelerates and decelerates does not have a large impact on the ball catching success due to the short duration of the trajectory [7]. However, the QP method does allow for more flexible and smooth trajectories, which may improve the performance of tasks with longer duration. Finally, similar success rates with the two different low-level planners may be due to the high accuracy of the learning-based controller when tracking arbitrary trajectories.

3.4.3 Failure analysis

The success of ball catching tasks requires a tight integration of every system component. The malfunction of any component will result in a failure. Among all throws in Experiment A, B, C, failure has occurred for the following reasons: inaccurate ball trajectory prediction, large tracking errors, and failure to find feasible solutions of the high-level planning problem. Large tracking errors occur when maneuvers are too aggressive and overshooting happens. We could improve this by penalizing aggressive trajectories.

3.5 Conclusion

In this chapter we presented a framework for accurate high-speed motion generation and execution on mobile manipulators with the application scenario of ball catching. A modified Kalman filter that considers aerodynamics is used for estimating the ball's velocity, and ball trajectory prediction is done via numerical integration. A bi-level optimization scheme is proposed to handle complex trajectory requirements in real-time, with an average run time of 5.1 *ms*. The high-level goal planning problem is formulated as a non-linear optimization problem and solved by SQP. The low-level path planning problem is solved by QP. The joint control is driven by an inverse dynamics learning method. This learning-based controller reduces joint tracking errors by up to 79.1%, which translate into a 10% increase in success rate compared to a non-learning-based joint controller. The proposed framework is validated via extensive real robot catching experiments under different configurations and achieves a success rate of 85.33%, setting a new state of the art.

Chapter 4

End-to-end Framework

As shown in Ch. 3, the modular framework mainly consists of three components: ball estimation, motion planning and joint control. Designing and implementing each component requires a lot of engineering efforts and experience. Meanwhile, modification in one component usually will result in changes in upstream and downstream components. As an example, instead of using discrete numerical integration for the ball trajectory prediction, we can fit a parabola curve using all existing ball state measurement. Then the high-level motion planner can search in continuous catching time instead of discrete time. This may change the SQP’s behaviors and require another round of SQP parameter fine-tuning, which is notoriously time-consuming.

Therefore it is interesting to utilize end-to-end learning method for the ball catching task, where a monotonic neural network can directly map raw ball estimations to low-level joint control commands. We do not need to design and implement each component, which is faster and does not require a lot of domain knowledge. Meanwhile, the system now can be optimized in an end-to-end fashion, which may result in better overall system performance. Therefore, in this chapter we present our exploration of using deep reinforcement learning and sim2real transfer technique for the ball catching task. Considering that even accurate motion for static tasks like pick-and-place is very challenging for deep RL algorithms now, we have to simplify the ball catching task. Thus we choose to use a fixed-base manipulator in this chapter, but the proposed algorithm framework should be able to be easily extended to the mobile manipulator case.

The rest of this chapter is organized as follows: Sec. 4.1 presents the problem statement, Sec. 4.2 shows the methodology including the task policy, policy optimizer, simulation optimizer and the acquisition policy, and Sec. 4.3 presents the simulation experiment results. Sec. 4.4 concludes this chapter.

4.1 Problem Statement

The system model and problem statement here are similar to the modular framework, as presented in Sec. 3.1, with slight modifications. Here we consider a 6-DoF manipulator mounted on a fixed base. The robot configuration vector is $\mathbf{q} = [\mathbf{q}_a^T, \bar{\mathbf{q}}_b^T]^T$ where $\bar{\mathbf{q}}_b$ is the fixed base’s Cartesian position. Given initial robot configuration $\mathbf{q}_0$, ball state $[\mathbf{b}_0^T, \dot{\mathbf{b}}_0^T]^T$, and time t_0 , we want to find a certain catch time t_f :

$$\begin{aligned} {}_w\mathbf{P}_{ee}(\mathbf{q}_f) &= \mathbf{b}(t_f) \\ \cos(\angle {}_w\mathbf{z}_{ee}(\mathbf{q}_f), -\dot{\mathbf{b}}(t_f)) &= 1. \end{aligned} \tag{4.1}$$

4.2 Methodology

This section presents our algorithm design for the end-to-end framework. We use the generic sim2real transfer framework as presented in Alg. 1, where the simulation environment is parameterized by a set of parameters ξ . In the following subsections, we present our designs for task policy, policy optimizer, simulation optimizer and acquisition policy.

4.2.1 Ball Catching MDP

We first present a detailed definition of MDP for the ball catching task, while a generic definition of MDP can be viewed in Sec. 2.4.1. We focus on the design of state space, action space, and reward functions, which have a large influence on the task policy’s training and evaluation performance.

The state s_t consists of current arm joint positions, current arm joint velocities, and the ball’s 3D positions in the world frame over the last three time steps:

$$s_t = [\mathbf{q}_{a_t}, \dot{\mathbf{q}}_{a_t}, \mathbf{b}_t, \mathbf{b}_{t-1}, \mathbf{b}_{t-2}]. \tag{4.2}$$

Having past ball positions in the state can help the neural network predict future ball trajectories implicitly. The action a_t of the MDP is the arm joint position movement between the current time step and next time step:

$$a_t = \Delta \mathbf{q}_{a,t}. \tag{4.3}$$

In other words, the desired arm joint position at the next time step $t+1$ would be $\mathbf{q}_a + a_t$. Only learning the relative movement instead of absolute joint position can increase the learned policy’s generalization ability.

The reward function design of the ball catching MDP is not a trivial task. The most intuitive way is to only give the robot rewards when it catches the ball. This is a sparse reward where most of the agent’s actions will receive no feedback signals, which is an open problem in the reinforcement learning community. With sparse rewards, it is extremely difficult for RL algorithms to connect a long series of actions to a distant future reward [47]. For extremely sparse rewards, the agent may never find a reward signal, and thus will never learn how to perform a given task. There have been a lot of work trying to solve this problem, like experience replay [47], reward shaping [48] and intrinsic rewards [49], but there is no general solution yet.

In this thesis we choose to use reward shaping, where the reward function needs to be carefully redesigned to encourage the expected optimal behaviors. Compared with other methods, reward shaping is simple and easy to incorporate prior knowledge of the task. One common reward shaping approach for catching-like tasks, like reaching and grasping, is to add a reward based on the distance between the target location and the robot’s end effector. Unfortunately, preliminary experiments with rewards based on distance between the end effector and the ball failed to produce meaningful results. Possible reason is that being close to the current ball position at every time step is not an optimal strategy. Instead, the agent should predict the ball’s trajectory and moves towards the desired catching location instead of the current ball position. To this end, our reward function is as follows:

$$R(s_t, a_t) = -w_1 \|\mathbf{q}_{a,f}^* - \mathbf{q}_{a,t}\|_2^2 - w_2 \|\dot{\mathbf{q}}_{a,t}\|_2^2 + w_3 I(s_t) \quad (4.4)$$

where w_1, w_2, w_3 are weight parameters, $\mathbf{q}_{a,t}, \dot{\mathbf{q}}_{a,t}$ are the current arm joint position and velocity respectively. The term $\mathbf{q}_{a,f}^*$ is the optimal catching configuration for the ball trajectory, which is calculated using the nonlinear high-level motion planner presented in (3.10) based on the initial ball position and velocity. Therefore $\mathbf{q}_{a,f}^*$ would be calculated at the beginning of one ball catching episode and remains constant in that episode. The term $I(s_t)$ is a success verification function:

$$I(s_t) = \begin{cases} 1, & \text{If catches the ball} \\ 0, & \text{Otherwise} \end{cases} \quad (4.5)$$

Intuitively, the first reward term $\|\mathbf{q}_{a,f}^* - \mathbf{q}_{a,t}\|_2^2$ is to encourage the policy to move to the predicted catching pose, which can help the policy converge faster during the early training phase. The second term $\|\dot{\mathbf{q}}_{a,t}\|_2^2$ is to penalize large joint velocity. This can also discourage unnecessary oscillation during catching. The third term $I(s_t)$ is the original sparse reward signal. We set the reward weight parameter w_3 much larger than the other

two parameters w_1, w_2 , to encourage the agent to find catching policies different from the high-level motion planner. For convenience, in this thesis, we refer to reward function in (4.4) as dense reward, and refer to the reward function that only contains (4.5) as sparse reward.

4.2.2 Task Policy and Policy Optimizer

We use a deep neural network to learn the task policy because of the neural network's remarkable learning capability. For policy optimizers, Sec. 2.4.2 presents several representative RL algorithms. Among them, DDQN is only suitable to tasks with discrete state and action space, while we have continuous state and action space here. Preliminary experiment results show that TD3 and PPO have similar task performance, but TD3 has better sample efficiency. Therefore we choose to use TD3 as the policy optimizer.

4.2.3 Simulation Optimizer

The goal of simulation optimizer is, given real-world trajectory τ_{real}^{acq} , previous simulation parameter estimate ξ_i , and possibly interaction with simulation environments, to generate a new parameter estimate ξ_{i+1} that will result in a better task policy performance. Simulation optimizers can be dynamics-oriented or task-oriented. Dynamics-oriented optimizer is aimed at finding estimate ξ_{dyn} that minimizes the difference between simulation dynamics $P_{\xi_{dyn}}$ and the real-world dynamics P_{real} . And task-oriented optimizer is aimed at finding estimate ξ that can result in the best task performance, and ξ is not necessarily close to ξ_{dyn} .

Compared with task-oriented optimizers, the dynamics-oriented optimizer's estimate can be used in other tasks, since it is searching for a globally optimal estimate while task-oriented methods are searching a locally optimal one. Searching the global optimum usually require more data and computation time. However, task-oriented optimization is a bi-level optimization. For each parameter ξ , we need to train a new policy π_θ and then roll the policy out in the real world, which is also time-consuming. So it is hard to give a general conclusion about which method is more computationally expensive.

In this thesis we choose to use the task-oriented optimizer since we can use well-established optimization methods. Specifically, we want to find the parameter ξ that brings the simulation trajectories closer to the real-world trajectories, when the simulated environment and real-world environment have same initial conditions and control policy [28]. The distance between simulation trajectory τ_ξ and real-world trajectory τ_{real}

is:

$$D(\tau_\xi, \tau_{real} | s_0, \pi_a) = \sum_{i=1}^T \|s_{i,\xi} - s_{i,real}\|_2^2, \quad (4.6)$$

where s_0 is the initial state, π_a is the acquisition policy for rollout, $s_{i,\xi}$ is the simulated state at time step i , and $s_{i,real}$ is the real-world state.

Given the initial state and the acquisition policy, $D(\tau_\xi, \tau_{real})$ in (4.6) is a cost function to be minimized w.r.t the simulation parameter ξ . If we have a differentiable simulation dynamics P_ξ , $D(\tau_\xi, \tau_{real})$ is differentiable w.r.t ξ and gradient-based optimization methods like gradient descent and Newton method can be applied here. Otherwise, we have to use gradient-free methods like BO and Evolutionary Strategy (ES). It is usually more desired to use gradient-based optimization methods since they are, in general, much faster. However, differentiable simulators for robots are hard to obtain, mostly because the mathematical model of contact and friction force is usually non-differentiable [50]. Thus we choose to use gradient-free methods, in particular bayesian optimization.

As mentioned in Sec. 2.2, BO is best-suited for optimization over continuous domains of less than 20 dimensions, and can handle stochastic noise in function evaluations. Our target function in (4.6) falls into this category. For each evaluation of ξ , we need to do one rollout in the simulation which requires a fair amount of computation time. And we do have stochastic noise in our simulation dynamics and thus noise in function evaluations. The number of critical simulation parameters is usually not large, which includes link mass, friction coefficients, etc. So BO is a reasonable choice here.

4.2.4 Acquisition Policy

The goal of acquisition policy is to actively generate action sequence used in the simulation optimizer (4.6). We first need to define a utility function over the state-action space, and then select the action sequence according to our utility function. The design of utility function depends on the simulation optimizer. For dynamics-oriented simulation optimizers, acquisition policies may want to generate action sequences that lead to states where the simulation dynamics uncertainty is large. For task-oriented simulation optimizers, acquisition policies may prefer action sequences that lead to states where the task performance uncertainty is large.

This topic is related with the exploration policy of RL algorithms, where agents need to actively explore states and actions that help agents better learn the environment's dynamics model or help agents find states with higher value [51, 52]. But it is non-trivial to transfer these methods into the sim2real transfer domain. There are few paper

trying to address this problem. In [39], an acquisition policy is trained by deep RL algorithms. The policy’s reward is the downstream task policy’s performance. However, it turns out that training such an acquisition policy is very computationally expensive, and the improvement on the final task performance is marginal. Therefore, like many other sim2real transfer methods, we simply reuse the task policy as the acquisition policy. This is simple, requires no extra training phase, and it can cover the frequently visited state-action pairs of the task.

4.3 Simulation Experiment

This section presents the simulation experiment setup, framework implementation details and experiment results.

4.3.1 Environment Setup and Framework Implementation

The ball catching simulation environment is built using the Multi-Joint dynamics with Contact (Mujoco) simulation engine [53]. Mujoco uses symbolic algorithms to calculate the rigid-body dynamics in (2.6), and uses convex optimization techniques to solve contact forces. Mujoco is suitable for computationally-intensive techniques such as optimal control and physically-consistent state estimation. Experiments shows that it can run 100 times faster than real-time on a single core of a desktop processor [53].

We build up the mobile manipulator model based on nominal kinematics and dynamics parameters. A screenshot of the simulation environment is shown in Fig. 4.1. We rigidly attach a cylinder onto the end effector to capture the ball. The cylinder’s radius is 0.1 m and its height is 0.2 m , while the ball’s radius is 0.025 m . This is similar to the geometry shape of the real bottle and ball. The control action inputs to the Mujoco simulator $a_t = \Delta \mathbf{q}_{a,t}$ is translated to joint torques by a position controller $\varsigma = K_p \Delta \mathbf{q}_{a,t}$, where the vector $K_p = [K_{p,1}, K_{p,2}, \dots, K_{p,6}]^T$ is the control gain.

We set the maximum episode length T to be 50. In each episode, we will reset the arm to a fixed initial configuration $\mathbf{q}_{a,0} = [1.0, -0.3, -1.0, 0.0, 1.6, 0.0]$, while the base is fixed at $\mathbf{q}_b = [0.0, 0.0]$ in the world frame, as shown in Fig. 4.1. The initial ball position and velocity are sampled from Gaussian distributions:

$$\begin{aligned} \mathbf{b}_0 &\sim \mathcal{N}([1.22, -1.60, 1.20]^\top, 0.01\mathbf{I}_{3 \times 3}), \\ \dot{\mathbf{b}}_0 &\sim \mathcal{N}([-1.00, 1.50, 4.50]^\top, 0.01\mathbf{I}_{3 \times 3}). \end{aligned} \tag{4.7}$$

The position and velocity mean are chosen such that the initial robot end effector can

intersect the resulting ball trajectory. In other words, the resulting ball trajectory is feasible for the robot to catch in high probability.

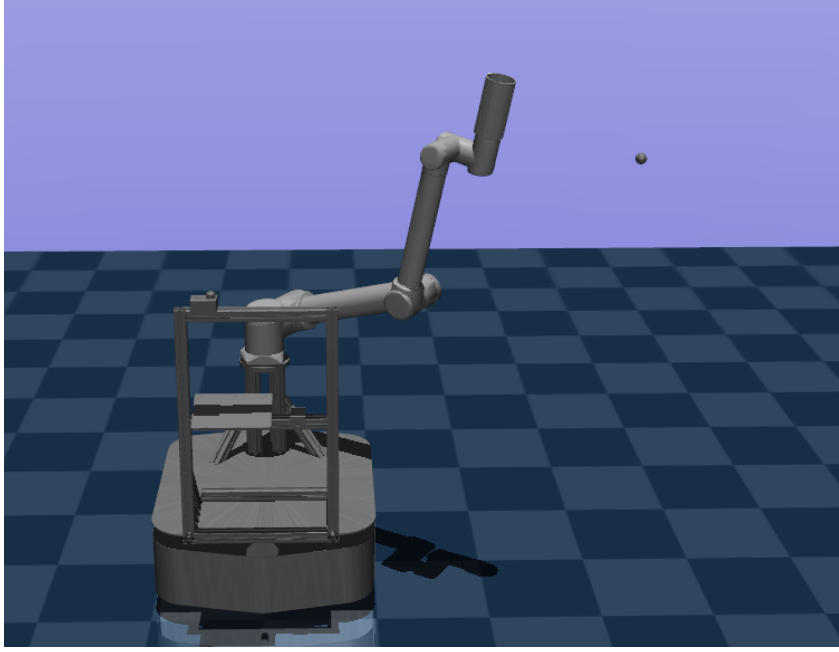


Figure 4.1: The mujoco simulation environment for the mobile manipulator ball catching task.

As for the simulation optimizer, we choose the following simulation parameters to optimize: (i) six arm link mass $[m_1, m_2, \dots, m_6]$; (ii) joint friction coefficient μ_{fric} ; (iii) joint position controller gain K_p . Thus, we have:

$$\xi = [m_1, m_2, \dots, m_6, \mu_{fric}, K_{p,1}, K_{p,2}, \dots, K_{p,6}]^T, \quad (4.8)$$

which is a 13-dimensional vector. It covers the robot's inertial parameters, joint friction and control gains. Note that we assume every arm joint has the same friction coefficient μ_{fric} . The simulation optimizer is implemented via an open source BO library [54]. For the GP model used in the BO, we use the Matern kernel with $\nu = 2.5$ and length scale $l = 1$, and the noise covariance σ_ω is set to be 0.1. The upper confidence bound function is used for data acquisition.

As for the task policy, we use a four-layer MLP. Each hidden layer has 256 neurons with ReLU activation function. The output layer first uses tanh activation function to map the action to be $[-1, 1]$, and then multiply the output with the max action value to get the final action output. We set the reward function weight parameters in (4.4) to be $w_1 = 0.1, w_2 = 0.01, w_3 = 10.0$. The large weight for the sparse reward $w_3 I(s_t)$ is to

encourage the agent to discover catching policies different from the non-linear high-level motion planning algorithm. The task policy is trained via the TD3 algorithm in Pytorch, with default hyper-parameters provided in the original paper.

4.3.2 Experiment Result

We now present the experiment results of the proposed end-to-end framework. Here we use a nominal simulation environment as the "real world". Compared with the real robot, a simulated real-world environment can help us better control the initial experiment conditions and have a fair comparison. The nominal environment's simulation parameters are shown in Tab. 4.1.

First we evaluate the performance of our task policy and policy optimizer in the nominal environment. In other words, the policy is trained and tested in the same environment. To facilitate the training, we take 1000 random initial ball state samples from (4.7), and we randomly pick one initial state from this dataset for each training episode. In this way, the policy will not need to keep adapting to new ball trajectories, and focus on learning the catching strategy. We repeat this experiment with five different randomly-picked random seeds. For each experiment, we use one Nvidia Tesla P100 GPU and four CPUs for training, and it takes about seven hours to complete 50000 episodes. As a comparison, we also run experiments which only uses the sparse reward term $w_3 I(s_t)$ in (4.4) for training. The policy trained with sparse reward is evaluated with the dense reward function in (4.4) so that we can compare policies' task performance under the same metric. The overall training performance is presented in Fig. 4.2 and 4.3. It can be seen that after 50000 episodes, the agent with dense reward can achieve really good task performance. The highest reward and success rate are 5.30 and 85.6% respectively. But the agent with sparse reward still learns nothing, which validates our reward function design.

Now we evaluate the performance of our simulation optimizer. To better visualize the optimization progress, here we consider a simpler optimization problem where the simulation parameter only contains the joint friction coefficient μ_{fric} . We use the policy trained in the above task policy evaluation experiment. We set the initial guess to be $\mu_{fric} = 30$, while the nominal value is $\mu_{real} = 65$. The posterior distribution learned by the Gaussian process after 240 data points is shown in Fig. 4.4. It is obvious that the Gaussian process successfully predicts that the cost function in (4.6) can achieve its minimum when $\mu_{fric} = 65$.

Finally, we evaluate our proposed end-to-end framework. The initial guess and BO

search bound of simulation parameters are shown in Tab. 4.1. It can be seen that there is a relatively large distance between the initial parameter guess, and the nominal parameter. We set the max iteration number to be $N = 10$. In each iteration, the max training episode for the policy optimizer is 15000 and the maximum number of real-world observation for BO is 200. Fig. 4.5 presents the trained policy’s task performance and the simulation optimizer’s cost function values in each iteration. The cost function value keeps decreasing, which means that our simulation environment is getting closer to the nominal environment. But our task performance is not increasing all the time, and there is a sudden drop in the middle. Possible reason is that the TD3 algorithm is not a non-regret algorithm. It may slide into a sub-optimal update direction during training. The final optimal success rate is 76.00% while the initial success rate is 10.00%, which demonstrates our proposed end-to-end framework.

Table 4.1: The simulation environment parameters.

Name	Nominal value	Initial guess	Search bound	Final estimate
m_1	7.78	3.00	[1, 10]	1.00
m_2	12.93	6.00	[5, 15]	8.97
m_3	3.87	7.50	[1, 10]	1.00
m_4	1.96	4.50	[1, 10]	1.00
m_5	1.96	4.50	[1, 10]	1.00
m_6	0.20	2.00	[0.01, 5]	2.00
μ_{fric}	65	30	[10, 100]	100
$K_{p,1}$	775	1000	[500, 1500]	1294
$K_{p,2}$	750	1000	[500, 1500]	1499
$K_{p,3}$	725	1000	[500, 1500]	1020
$K_{p,4}$	850	1000	[500, 1500]	1311
$K_{p,5}$	850	1000	[500, 1500]	1213
$K_{p,6}$	850	1000	[500, 1500]	1296

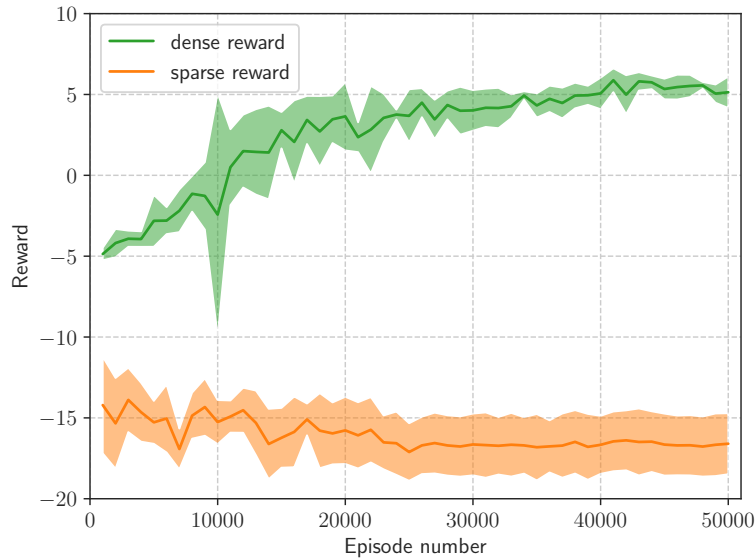


Figure 4.2: The episode reward when training and testing in the nominal environment. The green and orange line represent the policy trained with dense rewards and the policy trained with sparse rewards respectively. The green and orange area represent the one-sigma uncertainty bound.

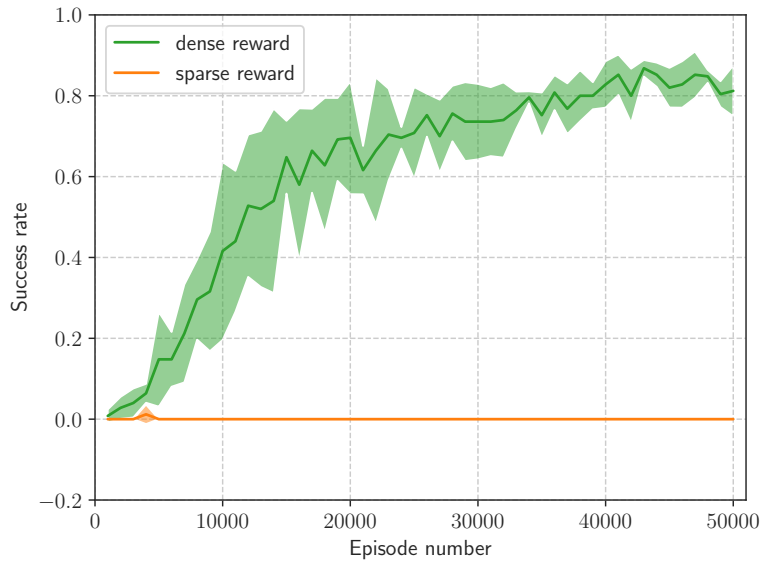


Figure 4.3: The success rate when training and testing in the nominal environment. The green and orange line represent the policy trained with dense rewards and the policy trained with sparse rewards respectively. The green and orange area represent the one-sigma uncertainty bound.

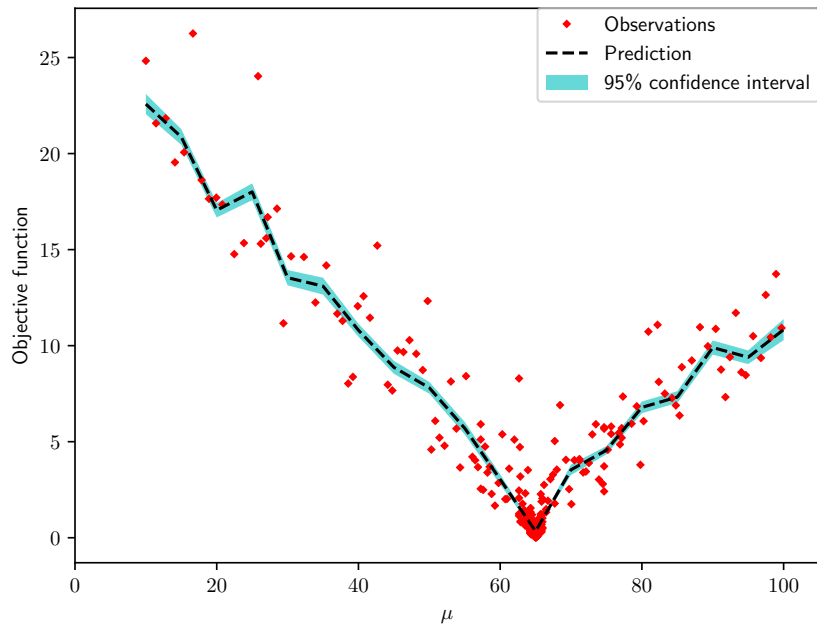


Figure 4.4: The posterior distribution learned by the Gaussian process. The red squares represent observations from the nominal environment, the dashed line represents the prediction, and the the blue area represents the 95% confidence bound.

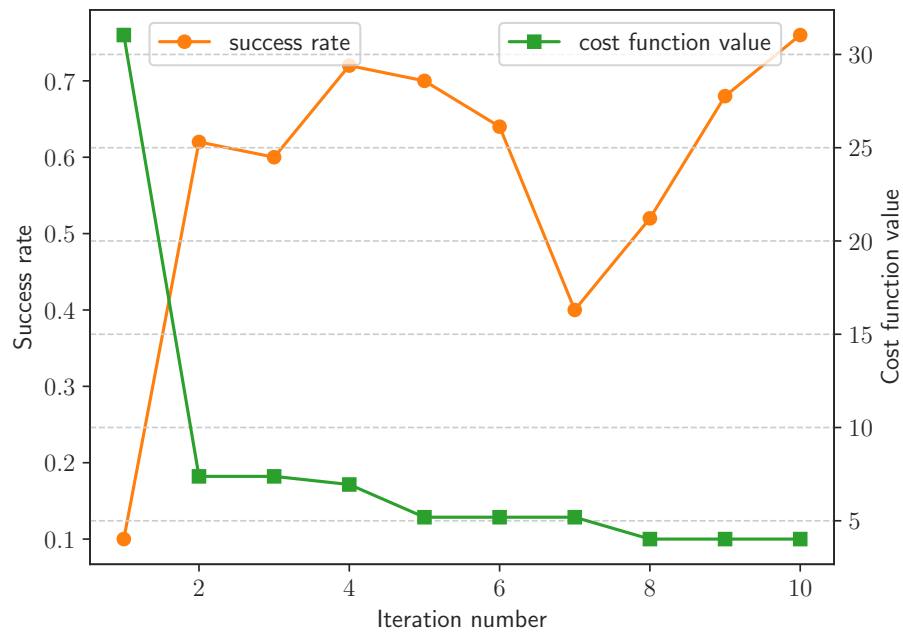


Figure 4.5: The success rate and simulation optimizer's cost function of every iteration when doing a sim2sim transfer, represented by orange and green lines respectively.

4.4 Conclusion

In this chapter we presented an end-to-end framework for the ball catching task, which utilizes deep reinforcement learning algorithms and sim2real transfer techniques. We use a four-layer MLP to directly map ball measurements to low-level joint control commands, and this MLP is trained via the TD3 algorithm. We carefully designed the original sparse reward function, using the high-level motion planner proposed in Ch. 3 for dense rewards. Simulation experiment results show that this modification can improve the success rate from 0.00% to 85.6%. We validate our proposed framework in sim2sim transfer experiments. Results show that our algorithm can improve the success rate from 10.00% to 76%.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis focuses on high-speed motion generation methods for mobile manipulators. We use the challenging ball catching task as the benchmark. Within a fraction of a second, the robot needs to predict the ball’s trajectory, generate aggressive motion and accurately track it in high-speed. We proposed two different frameworks for this problem, a modular framework and an end-to-end framework. In the modular framework, as presented in Ch. 3, a modified Kalman filter that considers aerodynamics is used for estimating the ball’s velocity, and we use numerical integration for ball trajectory prediction. A bi-level optimization scheme is proposed to handle non-linear trajectory requirements in real-time. The high-level goal planning problem is formulated as a non-linear optimization problem and solved by SQP, while the low-level path planning problem is solved by QP. Numerical experiment shows that this planner scheme has an average run time of 5.1 *ms*, which satisfies the real-time computation requirement. The joint control is driven by an inverse dynamics learning method, which can reduce joint tracking errors by up to 79.1%. This modular framework is validated via extensive real robot catching experiments under different configurations and achieves a success rate of 85.33%, setting a new state of the art.

The end-to-end framework presented in Ch. 4 addresses the ball catching task from a different perspective. It utilizes the TD3 algorithm to train a four-layer MLP as the policy in simulation, and uses the bayesian optimization to adapt the simulation environment. We carefully designed the policy’s state space, action space, and most importantly, the reward function. We use the non-linear high-level motion planner in the modular framework to provide a dense reward, in addition to the original sparse ball catching reward. Simulation experiment shows that this improves the success rate from 0.00% to 85.60%.

We tested the end-to-end framework in sim2sim transfer experiments. Results show that our algorithm achieve a success rate of 76.00% in the unknown nominal environment.

5.2 Future Work

A possible extension for the modular framework is to use dynamics-level planning instead of kinematics-level planning to have more aggressive motions. The current motion planner in the modular framework uses a double integrator model to describe the robot’s motion. This is easy and fast. However, to ensure the double integrator model work, the motion planner needs to choose conservative maximum velocity and acceleration values. Meanwhile, compared with dynamic models, motions generated using the kinematic model may be harder for the joint controller to track, because the dynamic model has a more accurate prediction of the robot’s motion. However, using the dynamic model will largely increase the planning algorithm’s computation time. The parallel computing and GPU acceleration techniques may alleviate this problem and achieve real-time computation.

The next step for the end-to-end framework is to extend it to mobile-base manipulator scenarios and validate it on the real robot, which would require more efforts in the implementation side. It is also interesting to extend the task from ball catching to arbitrary object catching, which can be used in warehouse pick-and-place. The new challenge here is to determine the optimal grasping pose for arbitrary objects in real-time. To this end, we would need to use vision instead of the VICON system for perception. Learning from human demonstration should improve the policy’s training performance.

Maybe in the future, the collaboration between robots in the warehouse center is throw-and-catch instead of pick-and-place. This will largely increase the warehouse efficiency, but it will involve a number of new challenging problems to be solved.

Bibliography

- [1] M. Jakob, “Position and force control with redundancy resolution for mobile manipulators,” Master’s thesis, University of Stuttgart, Stuttgart, Germany, 2018.
- [2] S. Zhou, M. K. Helwa, and A. P. Schoellig, “Design of deep neural networks as add-on blocks for improving impromptu trajectory tracking,” in *Proc. of the IEEE Conference on Decision and Control (CDC)*, pp. 5201–5207, 2017.
- [3] D. M. Bodily, T. F. Allen, and M. D. Killpack, “Motion planning for mobile robots using inverse kinematics branching,” in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5043–5050, IEEE, 2017.
- [4] A. De Luca, G. Oriolo, and P. R. Giordano, “Kinematic modeling and redundancy resolution for nonholonomic mobile manipulators,” in *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1867–1873, IEEE, 2006.
- [5] L.-P. Ellekilde and H. I. Christensen, “Control of mobile manipulator using the dynamical systems approach,” in *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1370–1376, IEEE, 2009.
- [6] O. Koç, G. Maeda, and J. Peters, “Online optimal trajectory generation for robot table tennis,” *Robotics and Autonomous Systems*, vol. 105, pp. 121–137, 2018.
- [7] B. Bäuml, T. Wimböck, and G. Hirzinger, “Kinematically optimal catching a flying ball with a hand-arm-system,” in *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2592–2599, 2010.
- [8] U. Frese, B. Bauml, S. Haidacher, G. Schreiber, I. Schäfer, M. Hahnle, and G. Hirzinger, “Off-the-shelf vision for a robotic ball catcher,” in *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3, pp. 1623–1629, 2001.

- [9] S. S. Mirrazavi Salehian, M. Khoramshahi, and A. Billard, “A dynamical system approach for catching softly a flying object: Theory and experiment,” *IEEE Transactions on Robotics*, 2016.
- [10] R. Lampariello, D. Nguyen-Tuong, C. Castellini, G. Hirzinger, and J. Peters, “Trajectory planning for optimal robot catching in real-time,” in *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3719–3726, IEEE, 2011.
- [11] P. A. Flores Álvarez, “Modelling, simulation and experimental verification of a wheeled-locomotion system based on omnidirectional wheels,” Master’s thesis, Pontificia Universidad Católica del Perú, Ilmenau, Germany, 2016.
- [12] KUKA, “Mobile robotics kmr quantec.” <https://www.kuka.com/en-ca/products/mobility/mobile-robots/kmr-quantec>. [Online; accessed 20-February-2020].
- [13] U. Robots, “Technical specifications ur10.” https://www.universal-robots.com/media/50880/ur10-{_}bz.pdf. [Online; accessed 20-February-2020].
- [14] B. Bäuml, O. Birbach, T. Wimböck, U. Frese, A. Dietrich, and G. Hirzinger, “Catching flying balls with a mobile humanoid: System overview and design considerations,” in *Proc. of the IEEE/RAS International Conference on Humanoid Robots*, pp. 513–520, IEEE, 2011.
- [15] W. Hong and J.-J. E. Slotine, “Experiments in hand-eye coordination using active vision,” in *Experimental Robotics IV*, pp. 130–139, Springer, 1997.
- [16] S. Kim, A. Shukla, and A. Billard, “Catching objects in flight,” *IEEE Transactions on Robotics*, vol. 30, pp. 1049–1065, Oct 2014.
- [17] S. M. LaValle, *Planning algorithms*. Cambridge university press, 2006.
- [18] H. Seraji, “A unified approach to motion control of mobile manipulators,” *The International Journal of Robotics Research*, vol. 17, no. 2, pp. 107–118, 1998.
- [19] M. W. Spong, S. Hutchinson, and V. Mathukumalli, *Robot Modeling and Control*. John Wiley and Sons, 2006.
- [20] A. Punjani and P. Abbeel, “Deep learning helicopter dynamics models,” in *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3223–3230, IEEE, 2015.

- [21] M. T. Frye and R. S. Provence, “Direct inverse control using an artificial neural network for the autonomous hover of a helicopter,” in *Proc. of the IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 4121–4122, IEEE, 2014.
- [22] Q. Li, J. Qian, Z. Zhu, X. Bao, M. K. Helwa, and A. P. Schoellig, “Deep neural networks for improved, impromptu trajectory tracking of quadrotors,” in *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5183–5189, IEEE, 2017.
- [23] S. Levine, C. Finn, T. Darrell, and P. Abbeel, “End-to-end training of deep visuomotor policies,” *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1334–1373, 2016.
- [24] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, “End to end learning for self-driving cars,” *arXiv preprint arXiv:1604.07316*, 2016.
- [25] W. Yu, J. Tan, C. K. Liu, and G. Turk, “Preparing for the unknown: Learning a universal policy with online system identification,” in *Proc. of Robotics: Science and Systems (RSS)*, July 2017.
- [26] F. Ramos, R. C. Possas, and D. Fox, “Bayessim: adaptive domain randomization via probabilistic inference for robotics simulators,” in *Proc. of Robotics: Science and Systems (RSS)*, 2019.
- [27] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, “Sim-to-real transfer of robotic control with dynamics randomization,” in *Proc. of the IEEE international conference on robotics and automation (ICRA)*, pp. 1–8, IEEE, 2018.
- [28] Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. Ratliff, and D. Fox, “Closing the sim-to-real loop: Adapting simulation randomization with real world experience,” in *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 8973–8979, IEEE, 2019.
- [29] C. G. Atkeson, C. H. An, and J. M. Hollerbach, “Estimation of inertial parameters of manipulator loads and links,” *The International Journal of Robotics Research*, vol. 5, no. 3, pp. 101–119, 1986.
- [30] P. I. Frazier, “A tutorial on bayesian optimization,” *arXiv preprint arXiv:1807.02811*, 2018.

- [31] K. Hornik, M. Stinchcombe, H. White, *et al.*, “Multilayer feedforward networks are universal approximators,” *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.
- [32] Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski, “A theoretical framework for back-propagation,” in *Proc. of the connectionist models summer school*, vol. 1, pp. 21–28, 1988.
- [33] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [34] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, *et al.*, “Dota 2 with large scale deep reinforcement learning,” *arXiv preprint arXiv:1912.06680*, 2019.
- [35] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
- [36] H. Van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” in *Proc. of the AAAI Conference on Artificial Intelligence*, pp. 2094–2100, 2016.
- [37] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv preprint arXiv:1707.06347*, 2017.
- [38] S. Fujimoto, H. Van Hoof, and D. Meger, “Addressing function approximation error in actor-critic methods,” in *Proc. of the International Conference on Machine Learning (ICML)*, 2018.
- [39] J. Liang, S. Saxena, and O. Kroemer, “Learning active task-oriented exploration policies for bridging the sim-to-real gap,” in *Proc. of Robotics: Science and Systems (RSS)*, 2020.
- [40] F. Muratore, C. Eilers, M. Gienger, and J. Peters, “Bayesian domain randomization for sim-to-real transfer,” *arXiv preprint arXiv:2003.02471*, 2020.
- [41] M. Müller, S. Lupashin, and R. D’Andrea, “Quadrocopter ball juggling,” in *Proc. of the IEEE/RSJ international conference on Intelligent Robots and Systems (IROS)*, pp. 5113–5120, IEEE, 2011.
- [42] T. D. Barfoot, *State estimation for robotics*. Cambridge University Press, 2017.

- [43] S. Boyd, S. P. Boyd, and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.
- [44] S. G. Johnson, “The nlopt nonlinear-optimization package.” <http://github.com/stevengj/nlopt>. [Online; accessed 20-February-2020].
- [45] H. Ferreau, C. Kirches, A. Potschka, H. Bock, and M. Diehl, “qpOASES: A parametric active-set algorithm for quadratic programming,” *Mathematical Programming Computation*, vol. 6, no. 4, pp. 327–363, 2014.
- [46] K. Dong, “A light-weight mlp library for fast online inference.” https://github.com/KevinDong0810/mlp_cpp_inference. [Online; accessed 20-February-2020].
- [47] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. P. Abbeel, and W. Zaremba, “Hindsight experience replay,” in *Proc. of the Advances in Neural Information Processing Systems (NeurIPS)*, pp. 5048–5058, 2017.
- [48] A. Y. Ng, D. Harada, and S. Russell, “Policy invariance under reward transformations: Theory and application to reward shaping,” in *Proc. of the International Conference on Machine Learning (ICML)*, vol. 99, pp. 278–287, 1999.
- [49] N. Chentanez, A. G. Barto, and S. P. Singh, “Intrinsically motivated reinforcement learning,” in *Proc. of the Advances in Neural Information Processing Systems (NeurIPS)*, pp. 1281–1288, 2005.
- [50] F. de Avila Belbute-Peres, K. Smith, K. Allen, J. Tenenbaum, and J. Z. Kolter, “End-to-end differentiable physics for learning and control,” in *Proc. of the Advances in Neural Information Processing Systems (NeurIPS)*, pp. 7178–7189, 2018.
- [51] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel, “Vime: Variational information maximizing exploration,” in *Proc. of the Advances in Neural Information Processing Systems (NeurIPS)*, pp. 1109–1117, 2016.
- [52] I. Osband, C. Blundell, A. Pritzel, and B. Van Roy, “Deep exploration via bootstrapped dqn,” in *Proc. of the Advances in Neural Information Processing Systems (NeurIPS)*, pp. 4026–4034, 2016.
- [53] E. Todorov, T. Erez, and Y. Tassa, “Mujoco: A physics engine for model-based control,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5026–5033, IEEE, 2012.

- [54] F. Nogueira, “Bayesian Optimization: Open source constrained global optimization tool for Python.” <https://github.com/fmfn/BayesianOptimization>, 2014. [Online; accessed 20-February-2020].