UNIVERSITY OF CALIFORNIA

Los Angeles

Towards Intelligent Robotic Systems: Unifying Model-based Optimization and Machine Learning for Planning, Control, and Estimation

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Mechanical Engineering

by

Alexander Vaschauner Schperberg

2024

© Copyright by

Alexander Vaschauner Schperberg

2024

ABSTRACT OF THE DISSERTATION

Towards Intelligent Robotic Systems: Unifying Model-based Optimization and Machine Learning for Planning, Control, and Estimation

by

Alexander Vaschauner Schperberg
Doctor of Philosophy in Mechanical Engineering
University of California, Los Angeles, 2024
Professor Dennis W. Hong, Chair

The goal of this work is to formulate algorithms that can address three key ingredients I believe are necessary towards making robots autonomous and smart: (1) The robot needs to be able to **react** in an energy-efficient manner to outside disturbances; (2) The robot needs to **understand** its location of its surroundings and evaluate the uncertainty of its location to optimally and safely achieve some goal state; (3) The robot should continuously **learn** from experience during operation. Throughout this work, we show algorithms that can achieve in obtaining these ingredients. First, we will demonstrate a simple algorithm that plans for the most energy-efficient trajectories for a quadruped robot by optimizing for parameters such as cost of transport, manipulability measures, and avoid non-slipping configurations. With this algorithm, we show that the robot only moves when necessary, and demonstrates behaviors of reacting to outside disturbances to ensure it does not fall while also not wasting unnecessary energy. The idea of understanding is demonstrated through an algorithm that combines an MPC, SLAM, RNN, and object detection using CNNs to generate paths for unknown and uncertain environments. This algorithm is evaluated not

only for a complex quadruped robot, but also for multi-agent robot teams consisting of a UGV and UAV. The feasibility of such complex algorithm is also evaluated. Lastly, the idea of continuous learning is addressed not only through use of learning-based algorithms such as RNNs, but also through auto-tuning algorithms that employ an UKF. Using a UKF, we show that we can automatically tune controller gains and even parameters of an online planner. Because the UKF can adapt parameters quickly and without heavy computational load, the robot can continuously adapt its control/planner parameters during online operation to continually learn from the environment. To summarize, we will first present a simple planner for energy-efficient locomotion, provide two examples of end-to-end frameworks for motion planning and state estimation that uses a hybrid approach consisting of model and learning-based methods, and then provide a method of calibrating such end-to-end frameworks (which often contain many various modules) through an auto-tuning technique. Lastly, I end with a discussion on Large Language Models, and how they may potentially affect the robotic field, and further contribute to the idea of *understanding* in significant ways.

The dissertation of Alexander Vaschauner Schperberg is approved.

Veronica Santos

Mohammad Jawed Khalid

Stefano Soatto

Dennis W. Hong, Committee Chair

University of California, Los Angeles

2024

With each deliberate and small step, no matter how slow and difficult to attain, I ascend towards my dreams, fueled by unstoppable perseverance.

TABLE OF CONTENTS

1 Introduction **1**

1.1 Motivation 1

1.2 Objective 4

1.3 Contributions 6

1.4 Necessary Background Knowledge 9

1.4.1 Model Predictive Control 9

1.4.2 SLAM 11

1.4.3 Reinforcement Learning 12

1.4.4 Neural Networks and its Variants 13

1.4.5 Kalman Filtering and its Varients 16

1.4.6 Legged Robot Terminology 19

2 Related Works **21**

2.1 Motion Planning Problem 21

2.2 Auto-Calibration 26

2.3 State Estimation 32

3 Energy-Efficient Locomotion **36**

3.1 Introduction 36

3.2 Footstep Planner 36

3.3 Parameter Design Framework 38

3.4 Parameter Study Results 39

3.5 Simulation and Experimental Results 39
3.5.1 Controller Architecture/Implementation 39
3.5.2 Gazebo Simulation Results 39
3.5.3 Hardware Validation 40
3.6 Conclusions 40

4 Risk-Averse MPC 45
4.1 Introduction 45
4.2 Methods 47
4.2.1 Architecture Overview 48
4.2.2 General MPC Formulation 49
4.2.3 MPC Constraints 51
4.2.4 Obstacle Detection 53
4.2.5 Recurrent Neural Networks for Learning Uncertainties 55
4.3 Experimental Results 57
4.3.1 Robot Model and Motion Tracking Controller 57
4.3.2 Analysis of Learning Components 57
4.3.3 Gazebo Simulation 58
4.4 Discussion and Future Work 59

5 Motion Planning for Heterogeneous Multi-Agents 68
5.1 Introduction 68
5.2 Methods 70
5.2.1 Stochastic Model Predictive Control Formulation 71

5.2.2 Cooperative Multi-Agent Localization 74

5.2.3 Recurrent Neural Network for Uncertainty Propagation 76

5.2.4 Deep Q-Learning (DQN) for Global Planning 77

5.3 Experimental Validation 78

5.3.1 Implementation details 78

5.3.2 Results 79

5.4 Conclusion 82

6 Auto-Calibrating Planning and Control Parameters 90

6.1 Preliminaries 91

6.1.1 Notation 91

6.1.2 Unscented Kalman Filter for Control Parameter Tuning 92

6.2 Robot Control Architecture 95

6.2.1 Swing Controller 95

6.2.2 Stance Controller 95

6.2.3 Problem Definition and Contributions 96

6.3 Auto-Tuning Controller and Reference Trajectories of Legged Robot 98

6.3.1 Training Objectives for Auto-Tuning Swing Controller 98

6.3.2 Training Objectives for Auto-Tuning Stance Controller 99

6.3.3 Generating Reference Trajectories 99

6.3.4 Training Objectives for Auto-Tuning Reference Trajectory 101

6.4 Results using Physics-Based Simulator 103

6.4.1 Auto-Tuning for Optimizing Step Clearance and Forward Progress . . 105

6.4.2 Auto-Tuning for Optimizing Step Clearance, Forward Progress, Slippage, and Energy Consumption 105
6.4.3 Auto-Tuning Stance Controller 105
6.4.4 Auto-Tuning Swing Controller 106
6.4.5 Computation Times 107
6.5 Conclusions 108

7 Adaptive Force Controller 110
7.1 Admittance control formulation 111
7.1.1 Controlling for grasping force and normal force offsets 116
7.2 Auto-tuning formulation 117
7.2.1 Training objectives 120
7.3 Experimental Validation 124
7.3.1 Experiment 1: Tracking grasping force 125
7.3.2 Experiment 2: Tracking wrenches for free-climbing 126
7.3.3 Experiment 3: Tracking fast reference forces 127
7.4 Conclusion 127

8 Real-to-Sim: Predicting Residual Errors 128
8.1 Introduction 128
8.2 Problem Definitions 131
8.2.1 General Model 131
8.2.2 Neural Network with Unscented Kalman Filtering 133
8.3 Implementation 136
8.3.1 Computer specifications 136

8.3.2 Differential Drive Robot 137
8.3.3 Manipulator Robot 141
8.4 Experimental Results 142
8.5 Conclusion 143

9 State Estimation of Legged Robots 145
9.1 Methods 147
9.1.1 Problem Definition 147
9.1.2 Measurements 150
9.1.3 Model 150
9.1.4 Kalman Filter 153
9.1.5 Improving Estimation through Learning 154
9.2 Experimental Validation 155
9.2.1 Results 156
9.3 Limitations and Conclusion 157

10 Current and Future Work, and Conclusion 161
10.1 Modification of Auto-Tuning 161
10.2 Resolving Model Complexity with Reinforcement Learning 161
10.2.1 Reward Equations 162
10.3 Rise of Large Language and Vision Models 167
10.4 Conclusion 172

Bibliography 173

LIST OF FIGURES

1.1 **Smart Robots.** 6

1.2 **General concept of Model Predictive Control** 10

3.1 **Hardware validation** with the Unitree A1. The left shows the concept of our ellipse-based planner (see Sec. 3.2). 37

3.2 **CoT and manipulability measure** for the baseline and optimized parameter selection for various velocities. 41

3.3 **Planner and Control Architecture** 42

3.4 **Gait transition results**. Transition from walk to trot gait shown with vertical leg positions. 43

3.5 **Teleoperation results** demonstrating disturbance robustness. The top graph shows commanded and actual velocities, with vertical leg positions below. 44

4.1 **Architecture Overview.** This figure demonstrates the training and testing procedures of our method. In training, we first select different maps, where obstacles in each map are randomly distributed. A simulation where the robot moves from an initial to a goal position is executed on this map. At each timestep an observation is taken (e.g., camera or on-board sensor data). These measurements are used as the input to our SLAM/Object Detection/Sensors system, which estimate the current position and uncertainty in position of the robot, and also location and size of obstacles. MPC accounts for this information and produces outputs entered into our motion tracking controller. For every map at every timestep, the current observations, state position, and positional uncertainty (among other variables outlined in Section 5.2.3) are entered into a large database to produce our RNN model. Lastly, in the testing phase, RNNs can predict the positional uncertainty (which provide our collision boundaries) of the robot at future timesteps of the MPC prediction horizon. 61

4.2 **Example Module Outputs.** *Left*: An example output image of our trained object detector using a custom-trained convolutional neural network model. We used the YOLOv3 [110] architecture with default initialized weights for fast training and inference. *Right*: Inlier (green +) and outlier tracks (red *) produced by XIVO on data collected from the Intel Realsense D435i. 62

4.3 **Recurrent Neural Network Architecture.** Our RNN architecture predicts the covariances at robot poses $[x_{t+n}, y_{t+n}]$ at timesteps $t{+}n$ for $n = 1, ..., N$ (where N is the length of the MPC's prediction horizon). During training, we used inputs collected from the output of XIVO to parameterize the network towards the four output units, as indicated by the first 18 input units and last four units in the figure above. Seven hidden layers were used with ReLU activation functions, with five recurrent layers (green) and two fully connected layers (purple), to learn the temporal structure for covariance propagation. 63

4.4 **Gazebo Simulation.** Our high-fidelity simulation accurately models the dynamics of the ALPHRED quadruped robot. 64

4.5 **Training Loss.** *Top:* Our CNN model's training loss, used in our object detection pipeline. We trained for 5,200 epochs but only display 300 in the figure above. Note that we verified avoidance of overfitting via a validaton set but did not plot the curve here. *Bottom:* Our RNN model's training loss, used to infer future localization uncertainty for the MPC. As with the CNN, we verified avoidance of overfitting using a validation set. 65

4.6 **Trajectory Comparison.** A comparison of the trajectories computed by three different approaches. The baseline method (red) is an MPC framework without our extensions to consider propagated future state uncertainty from an RNN, and we define the naive approach (blue) as artificially inflating a robot's boundary through all time. In comparison, our approach (green) can plan for a quick yet safe trajectory by predicting potential future collisions. 66

4.7 **ALPHRED Hardware.** The ALPHRED quadrupedal robot developed by Hooks *et al.* [60] of the RoMeLa robotics laboratory at the University of California, Los Angeles. This complex platform is an ideal model to apply our methods, as showing success on this platform also demonstrates the potential of applying our methods to a wide selection of robotic systems. Table 4.2 describes some physical properties of the system. 67

5.1 **SABER framework.** SABER combines controls (stochastic model predictive control), vision (simultaneous localization and mapping), and machine learning (RNN and DQN), to provide local and globally optimized solutions in unknown and uncertain environments. 69

5.2 **SABER Algorithm.** This figure demonstrates the overall SABER planning algorithm in the testing phase, which can plan paths for one or more robots simultaneously. At timestep k, the environment provides information to robots that either carry a LiDAR or RGB camera and IMU; for the LiDAR configuration, a particle-filter SLAM is implemented, while for the RGB configuration, Visual-Inertial Odometry SLAM (VIO SLAM) is implemented. The sensors provide either scans or distance to feature information to a recurrent neural network model (which serve as inputs), and outputs the propagation of state uncertainty for future timesteps. If two or more robots are within communication range, a distributed Kalman filter updates the current and future states and their uncertainties to a more accurate estimate. These updated states and uncertainties are used to update the chance constraints for obstacle avoidance. These constraints are then considered by a stochastic MPC controller, which follows a given target position, provided by a deep Q-learning (DQN) agent that aims to move the robot towards a global goal. DQN uses the relative distances between the robots and the respective obstacles as its states, provides a target position for all robots as its actions, and is trained on several different maps with obstacles randomly distributed in each. Note, that the SMPC, SLAM, and RNNs components run on each robot individually, however, the DQN is run on a centralized base (which may be on the robot itself). 83

5.3 **Network structures.** We show the RNN structure used to model an EKF from a VIO SLAM algorithm in (A), or a particle-filter SLAM algorithm in (B) (5.2.3). The inputs are shown in orange, and correspond to either features/robot position (using VIO SLAM) or LiDAR scans/robot position (using particle-filter SLAM). The outputs are shown in red, and correspond to the x-y covariance matrix (which represents uncertainty in x-y position). The layer type is color coded below, where green represents a simple RNN layer, and purple a dense layer. 84

5.4 **Training loss.** Here we show the training loss for the RNNs, which were trained on uncertainty covariance outputs (in position) of a Visual-Inertial SLAM in (A) and a particle-filter SLAM in (B). The training was done using 500 epochs and on 4 different maps. Note, that the noise observed in (B) may be due to the particle-filter estimations/simplifications done in [53]. 85

5.5 **DQN Training and Testing Procedure.** In (A), we show the neural network structure used in our DQN algorithm (5.2.4). The network maps the inputs (i.e., states or relative distances between robots and obstacles/goals) to the outputs (i.e., actions or next target positions for the robots). The states and actions are connected by a linear neural network model (blue). In (B) we visually show the training process of the DQN for a 2 to 5 robot team, where all robots were trained to go to the goal location while avoiding obstacles (obstacles are randomized for each episode). The average rewards (calculated from 25 episodes at a time and divided by number of robots) are shown across the 35,000 episodes of training (training time was 5 hours). In (C) we show an example of how the environment can be transcribed into a 2D plot and apply the DQN to traverse multiple robots toward their goals. We also allow the UAV to fly over the obstacles (and assume we know the height of the obstacle *a priori*) while the UGV must avoid it. In (D) we show another example of our DQN, but this time the 2-robot UAV and UGV team have separate goal locations, and we add a reward incentive when both robots are near each other at each timestep. 86

5.6 **SABER Algorithm Results.** This figure demonstrates the overall SABER planning algorithm. In (A) and (B) we first show the capability of the SMPC-RNN to navigate the UGV and UAV in a densely populated space. In (C), we show that the SMPC-RNN of the UGV cannot get to the goal state, because of the occurrence of a local minima. However, with a DQN (which provides a global path illustrated by triangles), the UGV (orange) can correctly maneuver around the obstacle. The UAV (purple) can simply use it's z-axis to fly above the obstacle. A more complex example is shown in *(D)*, where both robots are directed towards different goal locations simultaneously. 87

5.7 **RNN Results.** In this figure, we show the true covariance value (where 'xx' represents the covariance of the center of mass in the x position as an example) in blue and the predicted covariance in orange for about 430 seconds of data. This is done when modeling the uncertainty using VIO SLAM (A) and particle-filter SLAM (B) on a test map. Note, that the predicted and true values almost perfectly align, demonstrating the RNN's ability to make valid uncertainty predictions. Lastly, we show more explicitly in the graphs, that when more features are tracked (green arrow) the lower the estimated uncertainty, while fewer features corresponded to higher uncertainty (red arrow). 88

5.8 **System performance.** We compare the results of SABER (SMPC-RNN-DQN) using the same map as *(D)* in Fig. 5.6) against baselines for the UAV *(A)* and UGV *(B)* with distance to goal vs time as our metric. See 5.3.2.2 for details. . . 89

6.1 **Auto-Tuning reference trajectories in Gazebo.** The robot follows a reference trajectory that is being tuned by the auto-tuning formulation. The figure shows the robot following a desired velocity (forward progress), a desired foot height (step clearance), minimizing foot slippage (ground reaction forces), and minimizing energy consumption. 92

6.2 **Auto-Tuning results.** The auto-tuning method successfully calibrated control parameters that generate reference trajectories (**A-D**), and controller gains (**E-F**). To make this evaluation, we used the cost as calculated using (6.2). The cost decreases (**A** and **D**) as the control parameters (e.g., forward progress and step clearance) get to their desired reference values. In **E-F**, we show the difference between the cost when not auto-tuning with the cost when auto-tuning. 104

6.3 **Results for robust locomotion on uneven terrain.** After auto-tuning the stance and swing controllers, we employ the robot on several test cases to demonstrate robust locomotion. We show the robot traversing over and on large beams and planks in the left and top right with a trot gait, respectively, and demonstrate a successful jumping gait in the bottom right. The robot is not aware of the obstacles and must overcome them by using the auto-tuned controllers. . . 107

7.1 **Hardware validation tests.** Auto-calibrating admittance control for tracking wrenches for climbing, manipulation, and locomotion tasks. The blue arrows indicate forces, while the orange arrows indicate torques. We also indicate the Force/Torque (F/T) sensor at the wrist, and strain gauges located on the finger, measuring the finger compression force 111

7.2 **Auto-calibrating admittance control framework.** This figure shows the overall control architecture. The admittance controller takes as input the wrench and position profile (represented by $\mathcal{W}_{ref}$ and $\mathbf{x}_{ref}$) or the forces $\mathbf{f}_{ref}$ (if our MPC is used). The current wrenches are received using the relationship provided by equation (11) which requires force/torque sensor information, and also the current control inputs ($\mathbf{x}$ and $\dot{\mathbf{x}}$). The output of the admittance controller is the acceleration ($\ddot{\mathbf{x}}$), where its integration yields the control inputs for the next timestep (considered the inner loop). Since we use position-controlled motors, we use $\mathbf{x}$ as input to our inverse kinematics, which provides the joint motor angles ($\boldsymbol{\theta}_{1:n_{actuators}}$). A PD joint controller is used to track these angles (considered the outer loop). Lastly, the auto-tuning framework takes as input the reference and current wrenches and outputs new gains for the admittance controller ($\mathbf{M}_d$, $\mathbf{D}_d$, and $\mathbf{K}_f$). As the auto-tuning method makes use of the robot model (dependent on spring constants $\mathbf{K}_p$, and $\mathbf{K}_\theta$), we make use of the current wrenches to continually update these spring constants to a more accurate estimate as part of the auto-tuning process. 112

7.3 **Frame definitions.** Here we show the general frame definitions for the variables used in the admittance controller. Our controller will operate in the local gripper frame $\{\mathbf{G}\}$, which is considered the robot's wrist (or the location of the Force/Torque or F/T measurements). The gripper may have k number of end-effectors or fingertips, denoted by frame $\{\mathbf{F}\}$. Fingertip positions, $\mathbf{p}_k^G$, are all with respect to the gripper frame $\{\mathbf{G}\}$. Similar notations are given to the fingertip wrenches $\mathcal{W}_k^G$ and wrenches at the wrist $\mathcal{W}^G$. Note that to more easily solve for equations, we choose frame $\{\mathbf{F}\}$ to be the same orientation as frame $\{\mathbf{G}\}$. . . . 114

7.4 **Results: Tracking Grasping Force for Manipulation.** (**A**)-(**C**) shows the results for tracking the grasping force. Note, we do not show the results of auto-tuning versus no auto-tuning for (**A**)-(**C**) because without auto-tuning, the values can quickly diverge, see (**D**). (**D**) is the H2 Norm between auto-tuning (blue) and not auto-tuning (orange) the gains during operation (using the object shown in (**A**). (**E**)-(**F**) exemplify the tuning of the M_d, K_d gains of the admittance controller and K_p gain of the spring term of the model described in equation (7.8) (note for this test we only tune the z-component of the fingertip force). . . 119

7.5 **Results: Tracking Wrench during Climbing**. A Climbing wrench trajectory is tracked in (**A**) - (**F**) or $f_{1,2,x}$, $f_{1,z}$, and $\tau_{1,2,z}$. The experiment is depicted on the left of Figure (7.1). 120

7.6 **Results: Tracking Fast Reference Forces from an MPC.** An MPC is used to generate reference forces during a trot gait (blue). The actual force (tracked by our adaptive admittance controller), is measured by FT sensors attached to the feet of the robot. The experiment is depicted on the bottom right of Figure (7.1). 121

8.1 **Real-to-Sim.** Here we show our experimental setup, where we learn the residual error between the actual and simulated robot. 129

8.2 **Methods flowchart.** Here we show the overall methods of this paper as described in Section 8.2. Overall, the goal is to learn the residual model error between a simulator model and a dynamic model or Sim-to-Dyn (boxed in blue), the real robot and a dynamic model or Real-to-Dyn (boxed in green), and the real robot and the simulator model or Real-to-Sim (boxed in brown). The model is learned using a UKF method which calibrates weights and bias parameters of a neural network (the weights and bias variables are parameterized by $\delta(\mathbf{y}_t)$). Although not necessary, the learned residual model of one case may also be used as part of a warm-start procedure for the next case (e.g., the residual model for Sim-to-Dyn may be used as the starting residual error for the Real-to-Dyn case–which we found to reduce the amount of learning required to model any additional residual error). 130

8.3 **Neural Network with UKF.** We illustrate the UKF method as described in Section 8.2.2. The UKF calibrates weights and bias values of a neural network that is composed of a hidden layer. The result of this calibration is to produce an output or residual model error that minimizes the difference between a current and reference model. The green and blue arrows simply indicate that we are doing a two sequential for loop iterations. 134

8.4 **Manipulator trajectories.** For evaluating the modelling of residual errors on the manipulator arm, we use two trajectories. The first trajectory (shown in top half) consists of only x and y components–drawing a 2D circle, with a straight line in the middle. This was done to ensure we can model circular and also linear motions simultaneously. The second trajectory, shown on the bottom half, consists of x, y, and z components–drawing a circle in x and y while moving up and down in z. The red line on the last image of the trajectory shows the complete trajectory made by the arm. 137

8.5 **Differential drive robot results** The results of learning the residual error of our two-wheeled robot is shown here and described further in Section 9.4. In all cases, the H2 norm converges to a steady-state value near zero, indicating the residual model could be learned. We do note that for the real robot (Case C-E) we observed very noisy data due to our hardware (e.g., the wheels would stick and slip on the ground) and error-prone localization. By using a low-pass filter on the output of our neural network however, we could generate a more robust convergence even for this difficult setup. 139

8.6 **Manipulator arm results.** The results of learning the residual error of our manipulator robot is shown here, and described in more detail in Section 9.4. For the manipulator arm, we use two trajectories (one consists of 2D motion, i.e., Cases A-C, while another demonstrates 3D motion, i.e., Case D). The motion is described and visualized in Fig. 8.4. Overall, the H2 norm decreased for all cases. We also not only show the overall H2 norm but also the H2 norm of each individual component (i.e., $\dot{x}$,$\dot{y}$, and $\dot{z}$). Unlike the case for our wheeled robot, the localization of our manipulator arm was more stable (we used encoders for localization) and did not require any additional filters to our outputs. 140

9.1 Top left shows the attached body frame (B), and world frame (W). The trunk state $\mathbf{x}$ is in the world frame, while footstep positions $\mathbf{p}$ is relative to the body. Ground reaction forces (blue arrow) from our MPC control policy are given by $\mathbf{f}$, also in world frame. We verify our algorithm, OptiState, on slippery surfaces (top right), incline (bottom left), and rough terrain (bottom right). 146

9.2 Overall state estimation architecture as described in Sec. 9.1. 148

9.3 Transformer and Gated Recurrent Unit (GRU) network architecture. From (**A**), model 1 is the transformer model, model 2 is the GRU (δ) that predicts the robot's trunk state and uncertainty of its own prediction. Input/output state and hidden layer sizes indicated by the numbers. Training loss of model 1 shown in (**B**) and for model 2 in (**C**). MSE is the loss function (see Sec. 9.1.5). 158

9.4 Results during the online testing phase, as described in Sec. 9.2.1. In (**A**) we show the state estimation for all state components from OptiState, VIO SLAM, and the ground truth. We show 4 distinct trajectories connected by solid lines to symbolize the various terrain under evaluation, such as flat, slippery, incline, and rough terrain. The RMSE results over all 4 trajectories and per state component are shown in (**B**), and includes OptiState without the Kalman filter input, or vision input, and the Kalman filter alone. Lastly, we show the percentage improvement of RMSE over the VIO SLAM baseline for each state in (**C**) per estimation algorithm shown in the first column. 159

9.5 Example of predicting the uncertainty of the GRU's (OptiState) x position, or μ_x. The shaded blue represents $\bar{x} \pm \mu_x$. 160

10.1 **SCALER-B.** We show the hardware and motion capabilities of our SCALER-B robot, capable of (a) quadruped mode, (b) biped mode, (c) rolling, (d) climbing vertically, (e) pull-up, and (f) ceiling. 163

10.2 **Average Reward Results.** We show how the average rewards during training converges. 166

10.3 **Mujoco with MJX simulator.** We show a disturbance being applied while the robot is doing locomotion, as represented by the pink-red arrow. The green-blue arrows signify the ground reaction forces. 166

10.4 **VLM for Motion Planning** 171

LIST OF TABLES

3.1 Parameters for Parameter Study 39

4.1 Model Predictive Control Constraints 53

4.2 ALPHRED Configuration 60

6.1 Auto-Tuning Computation Times 108

9.1 Notation of Critical variables. Frames are defined in either world frame (W), body frame (B), or neither (N/A) 151

ACKNOWLEDGMENTS

Above all, I want to thank my family. My mother, Evelinde, father, Zolik, and brother Daniel. I am incredibly thankful to my mother for always believing in me, giving me hope in times that felt completely hopeless, and showing me the light, when things were dark. There were so many times where I complained to her about my life problems, days where I received negativity from those around me, and unfortunately, I ended up transferring this negativity into complaints to her. But no matter what I said, she would always listen and turn the negative into a positive – instantly making me feel better. Without her, there would be no chance I would have been able to attain a PhD, or ultimately, look forward to a brighter future. I want to thank my Father, who instilled in me discipline, and the desire to continuously learn and study – the art of Quality Management. Although I would sometimes put myself down or think I was not good enough, my Father always reminded me that I was great – this was needed for me to grow my confidence, so critically important when going on the long and difficult journey of doing a PhD. Although I did not always express it well, I am incredibly thankful to him for helping me no matter where I was. He would drive more than 8 hours from Walnut Creek near the Bay area to UC San Diego where I did my Bachelors degree, just to make sure I had food, water, and was okay. He would do this no matter which location I resided and did so repeatedly. I want to thank my brother, who since growing up has always been my best friend, and I cherish all our memories together. Immigrating from Austria to California as a child was a very difficult time for me, having to adapt to a totally new environment, culture, and people – but having my brother at my side during those times helped get me through this challenge. Just as my parents did, he too always believed in me. Without their guidance, and their love for me, I could not have done a PhD, and this PhD is ultimately their achievement as much as it is mine.

I also want to thank my aunts, uncles, and cousins from Austria – Monika (who essentially acted as my second mother and who I will always cherish), Elvira, Othmar, Hermi, Helmut,

Helmuti, Lydia, Christi, Stefan, Christian, and David (along with many others). I have such fond childhood memories of them, and whenever I feel down, I always think back on these nice memories. I want to thank Stefan in particular, as he would often do things with me every time I was in Austria, such as playing beach volleyball, as well as having fun while still having serious discussions about life in general. I feel fortunate that every time I visit Austria, I can be excited to see all of them again. I want to thank my friends from Austria as well, in particular my childhood friends Stefan Steinecker, Jacob, Martin, and Dario. It was so much fun hanging out with you as friends, and hope we can continue hanging out as adults as well. You guys shaped who I am today, and will never forget you no matter what. We now have a PhD in our family, and hopefully more to come in the future!

I want to thank Aghdas for being like a second mother to me in the US growing up. I have such fond memories of her positive outlook and great cooking. She would always give me gifts for both birthdays and Christmas, never forgetting a single time or year, even now when we are living far apart. I want to thank Michael for his incredible knowledge, wisdom, and deep conversations. Even as a child I was so fascinated by his intelligent discussions with my parents, and I would love listening to him. I feel I learned so much about business, and life in general through his discussions.

I want to thank my grandparents, Mutti, Vati, Nona, and Maurice, who I strongly believe have always watched and are continuously watching over me during my life's journey from heaven. When things get difficult, I always prayed to them, and believe they always found a way to help me. I hope I make them proud.

From the Robotics and Mechanisms Laboratory (RoMeLa), I want to give my utmost thanks and appreciation to my PhD advisor, Dr. Dennis Hong. He is an incredibly generous and honest person, and I could not have asked for a better advisor than him. He gave me nearly unlimited freedom to pursue my passion and allow me the freedom to study any topic I wish to study – without such freedom, I do not believe I would have had the success that I had throughout my studies. His popular phrase which is to *break robots*, has always stuck

with me – in fact, failure is a necessary step to success, and he taught me this. This allowed me to fiercely pursue my algorithms and employ them without fear, but if things fail, to make sure I know why it failed and learn from them – this was incredibly important advise when trying to employ algorithms successfully in hardware. I am also very inspired by his positive outlook on life and how he handles himself in difficult situations with calm but also with a smile on his face. I will try to strive each day to have that same mindset.

I want to thank the entire SCALER robot climbing team for giving me a home at RoMeLa where I can feel comfortable and do my research while having fun doing it. I want to thank Yusuke Tanaka for his kind support and helping me with so many of my research papers. He is one of the most talented people I know, capable of hardware and software – there is a reason why so many in the lab, including myself, would often go to him for questions and advice. He was always ready and happy to help without expecting anything in return. I feel incredibly grateful that we were able to go to so many conferences together – our time in Hawaii, Austin, and Japan is something I will forever cherish. It was also an honor working alongside him in difficult times as well, when the robot would break, or certain code would not work – I will never forget how we endured these times and overcame them together as a team, staying until 5 AM to fix any remaining issues. I also won't soon forget many of our food expeditions, which was always so much fun. I also want to thank Yuki Shirai, he was the one who introduced me to the SCALER team and also helped introduce me to important topics in robotics, which was incredibly helpful as I transitioned from Bioengineering to Mechanical Engineering. Throughout my PhD studies, we had so many great moments together in areas involving research but also just from our social interactions. We both not only did two internships at MERL together, but are also working at MERL as Postdocs as well – such a nice and fortunate coincidence. I feel so grateful to have shared so many memories with him, and feel incredibly thankful for the opportunity to create new memories with him in the near future in Boston. I want to thank Xuan Lin for his wisdom and knowledge in legged robots. I am particularly thankful for our discussions in force control, which helped me publish a

work in this domain.

Of course, I want to thank all other members in RoMeLa, for our discussions, and overall good times we had together: Dr. Jeffrey Yu, Dr. Joshua Hooks, Dr. Min Sung Ahn, Dr. Hosik Chae, Dr. Jesse Cha, Dr. Gabriel Fernandez, Dr. Aaron Zhang, Dr. Junjie Shen, Nick Liu, Donghun Noh, Kyle Gillespie, Hayato Kato, Varit Vichathorn, Feng Xu, and Alvin Zhu. Others from UCLA I want to thank are Stephanie Tsuei and Kenny Chen – thank you for your collaboration, insights, and contributions to our work together. Having such great collaborators especially on my first robotic paper was incredibly important as a jump-start to my future robotic ventures.

I want to thank my committee members Prof. Santos, Prof. Jawed, and Prof. Soatto. They have given me such valuable feedback on research and their classes truly inspired me as well. Prof. Santos for showing me the ways of control for the first time – my love for her class made me also fall in love with controls, a topic I now intensely pursue in most of my research works. I am also incredibly thankful for her support and recommendation to programs I applied. Prof. Jawed for introducing me for the first time on soft robotics, and how they can help shape the future. And to Prof. Soatto, who challenged me to think deeply about what I was working on, and to keep in mind the impact of my work in the future. I am incredibly grateful to his help and contribution of several of my works.

I want to thank Dr. Marcel Menner and Dr. Stefano Di Cairano from MERL for entrusting in me in not just one but two internships at MERL – because of this, I found my passion and desire to work at MERL. I thank Marcel for his excellent guidance during my internships, as he served as my host. I learned a lot from him, and his methods on Auto-Tuning greatly impacted several of my works in my dissertation. He provided the perfect balance of giving me the freedom during my internship to explore ideas, while at the same time providing quality feedback and deep research discussions which resulted in impactful work and ground our ideas towards significant directions. I want to thank Stefano for giving me the opportunity to do the internship at MERL and also for the opportunity to do a

Postdoc. His support and trust that I will do a great job for MERL is something I very much appreciate and I feel truly honored – I look forward to making the company proud with my future works in return.

I would like to thank Saviz Mowlavi, who not only helped me as a co-author for one of my works and provided insightful research discussions, but has become one of my best friends. I feel grateful to have such a friend who is not only loyal, but really supports and wants to see me do well – these kinds of friendships appear very rarely in life, and is something I will appreciate each and every single day. I am truly blessed to have him in my life, and I look forward to making more memories with him as I dive into the Postdoc position at MERL.

There are also so many people who I believe were instrumental in shaping me into a professional scientist at a earlier stage in my educational career. From UC San Diego, I want to thank the people working at the Swartz Center for Computational Neuroscience. This was my first lab that I joined – it helped me gain the experience necessary in what it means to be a researcher. I am incredibly grateful to Dr. Ying Wu, for teaching me the ways of research, and guiding me as I learned the wonderful world of neuroscience.

I want to thank the people at CureMatch, in particular Dr. Razelle Kurzrock, Blaise Barrelet, Dr. Stephane Richards, and Dr. Amelie Boichard (as well as all the people on that team). Dr. Kurzrock taught me what it takes to write and master paper writing, and to become detailed and quality orientated–she is one of the most talented and fierce people I know, I feel incredibly lucky to have worked with her. I owe my success of my publications to her incredible expertise and training. Blaise has been always wonderful to me, and always went out of his way to make sure my voice was heard – he made me feel like an intricate part of the team and I was incredibly honored and touched by his belief in me. Dr. Boichard was an incredible co-author, helping me in tremendous ways to facilitate my first published work. Her knowledge in biotech and cancer drugs is unmatched, and I learned a lot from her. And last but not least, I want to thank Dr. Richards, I will never forget when I got my first paycheck in my life, and it was from him. I still remember how happy I was when I

received my first paycheck. He was also the reason I got my first job, and strongly believed in my skills.

Finally, I'd like to thank the sources of my funding, which include the Office of Naval Research (ONR), National Science Foundation (NSF), and Amazon Science. And of course Dr. Dennis Hong for giving me the funding from these sources.

As I conclude this acknowledgement, I am struck by the realization of how many people, interactions, and efforts are essential to achieving any form of success. While the journey begins within oneself, the influence and support of those around me are invaluable. I am deeply grateful to a higher power for placing me in the right circumstances, guiding me to meet the right people, and steering me towards becoming a better and more successful person.

VITA

2018–2024	Ph.D. (expected) Mechanical Engineering University of California, Los Angeles – Samueli School of Engineering
2021-2022	Research Scientist Intern, Legged Locomotion Mitsubishi Electric Research Laboratories, Cambridge, Massachusetts
2016-2018	Research Scientist, Machine Learning CureMatch, Sorrento Valley, California
2013–2016	B.S., Bioengineering University of California, San Diego – Jacobs School of Engineering

PUBLICATIONS

- A. Schperberg, S. D. Cairano, M. Menner. *Auto-Tuning of Controller and Online Trajectory Planner for Legged Robots*. IEEE Robotics and Automation Letters (RA-L) with IROS option, June 2022.

- A. Schperberg, S. Tsuei, S. Soatto, D. Hong. *SABER: Data-Driven Motion Planner for Autonomously Navigating Heterogeneous Robots*. IEEE Robotics and Automation Letters (RA-L), May 2021.

- A. Schperberg, A. Boichard, I. Tsigelny, S. Richard, R. Kurzrock. *Machine learning model to predict oncologic outcomes for drugs in randomized clinical trials*. International Journal of Cancer. 2020; 1-13. https://doi.org/10.1002/ijc.33240.

- A. Schperberg, Y. Tanaka, S. Mowlavi, F. Xu, B. Balaji, D. Hong. *OptiState: State Estimation of Legged Robots using Gated Networks with Transformer-based Vision and Kalman Filtering*. IEEE International Conference on Robotics and Automation (ICRA), May 2024.

- A. Schperberg, Y. Shirai, X. Lin, Y. Tanaka, D. Hong. *Adaptive Force Controller for Contact-Rich Robotic Systems using an Unscented Kalman Filter*. IEEE/RSJ International Conference on Intelligent Robots and Systems (Humanoids), Dec. 2024.

- A. Schperberg, Y. Tanaka, F. Xu, A. Khan, B. Balaji, D. Hong. *Planner for Robotic Free-Climbing using Reinforcement Learning*. Southern California Robotics Symposium (SCR), Sept. 2023.

- A. Schperberg, Y. Tanaka, F. Xu, M. Menner, D. Hong. *Real-to-Sim: Predicting Residual Error of Robotic Systems using a Learning-based Unscented Kalman Filter*. IEEE 20th International Conference on Ubiquitous Robots (UR), Jun. 2023.

- A. Schperberg*, K. Chen*, S. Tsuei, M. Jewett, J. Hooks, S. Soatto, A. Mehta, D. Hong. *Risk-Averse MPC via Visual-Inertial Input and Recurrent Networks for Online Collision Avoidance*. 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Nov. 2020.

- Y. Tanaka, Y. Shirai, A. Schperberg, X. Lin, D. Hong. *SCALER: Versatile Multi-Limbed Robot for Free-Climbing in Extreme Terrains*. Under Review in IEEE Transactions on Robotics (T-RO), Dec. 2023.

- Y. Shirai, X. Lin, A. Schperberg, Y. Tanaka, H. Kato, V. Vichathorn, D. Hong. *Simultaneous Efficient Contact-Rich Grasping and Locomotion Optimization Enabling Free-Climbing for Multi-Limbed Robots*. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Mar. 2022.

CHAPTER 1

Introduction

1.1 Motivation

My goal is to facilitate the dream of one day seeing diverse sets of wheeled, aerial, legged, and underwater robots being used ubiquitously towards reducing the burdens of society. Robotics and AI technology have the enormous potential to support humanity by performing tasks too dangerous for human workers, or through human-robot interactions. Unfortunately, while the potential use of robotics is an exciting prospect, they are still not commonly used due to a justified concern for both their safety and cost. For example, to make robots safer typically demands high-fidelity sensor and computer components. Thus, these robots are very expensive and still seen as a luxury item rather than a product for everyday use. More troubling is that those from economically challenged and/or underprivileged groups may not have access and potentially cannot reap the benefit from this technology. Ideally, creating new robots using off-the-shelve or inexpensive components would greatly expand the robotic field and rapidly benefit society for all.

Still, in the past decades, we have seen enormous progress in realizing robotic systems beyond being merely conceptualized as drawings inside a notebook, but witnessing their actual form in reality— driving, swimming, flying, and even walking. As a society, we are gradually becoming accustomed to their presence, from vacuuming our floors, autonomous driving, recording ourselves using drones, or using wheeled robots for food delivery to our homes. Perhaps most frequently, robots are used beyond the sights of our eyes, from stacking

boxes in enormous warehouses, transporting medical goods in hospitals, and from teleoperation in Space. One of the primary reasons why robots are seen more today than ever before, is due to the development and research in three key components for realizing robotic motion – planning, control, and estimation (abbreviated as PCE throughout this prospective). However, while we are seeing more robots today than in the past, we are still far away from achieving their full potential. Specifically, only very few hospitals employ robots for transportation, most people still prefer manual vacuums, and autonomous driving is far from being implemented ubiquitously. There are two main issues why this is the case. First, fidelity in **planning**, **control**, and **estimation** still requires further research development before they are robust and useful enough to exceed human ability— beyond just being optimized for single purpose tasks. Second, while PCE algorithms can be enhanced through the best available sensor technology, this would make robotic systems too expensive for the average consumer and too much of an investment and risk for industry.

Still, perhaps one of the most exciting prospects in robotics is the research and development of legged robots, which is why I joined the Robotics and Mechanisms Laboratory. While many robots employed in the real world today can only operate in controlled environments, legged systems, which are created with the main purpose of emulating animals and humans, are expected to traverse diverse and unstructured environments. Further, as with some animals and all humans, they are capable and expected to manipulate the environment around them. This presents both a locomotion and manipulation challenge, in addition to the fact that legged systems are complex dynamical hybrid systems that constantly make and break contact with the ground. As these legged systems are developed further to fully realize their capabilities, we can be one day closer to creating machines that can be useful for humans in their everyday lives: as assistance in nursing homes, navigation of environments most easily accessible for humans, and even as companions to alleviate the sense of loneliness. Most importantly however, research and development in legged robots can be advantageous, as they can be used as platforms to test the robustness, safety, and efficiency

of algorithms within the PCE domains – this may provide a strong argument for their use in other less dynamically complex systems such as drones or wheeled robots.

I believe that advances in machine learning (ML) technology can alleviate the computational burden from PCE algorithms and make using less expensive equipment feasible. My goal is to formulate methods that apply both machine learning (ML) and traditional model-based planning and control toward making robots safer and easier to use. For instance, this safety can be considered through modeling residual errors using ML from exteroceptive and proprioceptive sensors, which, once modeled, can help alleviate the computational burden from planning and/or control algorithms and make using less expensive equipment more feasible. Additionally, the automation of PCE algorithms is greatly needed, as the amount of tuning and adjustment necessary to make these algorithms work can drastically slow down development time.

Ultimately, the main motivation throughout my PhD work is to demonstrate my best attempt at getting a little bit closer to making robots 'smarter' (Fig. 1). For instance, avoiding open loop systems that cannot adapt to unknown situations. Although these systems may be suitable enough for repetitive and simple tasks, they cannot lead to more general multi-task robots that can traverse our human-made environments. I define the following three components as necessary abilities for a smart robot:

(1) **Reacting:** When a robot is pushed, pulled, or disturbed with some amount of force, the robot should react autonomously to reduce this disturbance by performing the most energy-efficient action that aims to always stabilize its center of mass to avoid falling.

(2) **Understanding:** The robot must observe its environment and understand not only its own location relative to other beings and obstacles, but how its current and future actions can affect other beings and obstacles.

(3) **Learning:** During operation, the robot must utilize its experience interacting with

the environment to learn and continually improve itself autonomously over time

These components of a 'smart' robot can be achieved through implementation of new PCE algorithms. As I strive to produce methods in an attempt towards smarter robots, I believe can facilitate the progress and dream toward completely ubiquitous robots.

1.2 Objective

The objective of this work is to demonstrate various methods towards making robots 'smarter', where the definition of 'smart' is defined in the motivation Sec. 1.1. This is achieved through improving and automating parts of PCE. The first work as seen in Sec. 3 discusses how we can achieve the *reacting* component of a smart robot without requiring ML or complex algorithmic techniques. This is achieved through a simple planner for a quadruped robot that demonstrates how the robot can autonomously react to stabilize its center of mass when imposed by an external force while optimizing energy efficiency in its movement. Although the robot may stabilize itself and react to external forces using the method in Sec. 1.1, for the robot to traverse its environment, which for general real-world application must be assumed unknown, the robot must also have an understanding of its environment so it can either interact or avoid obstacles. This requires knowledge of how we can not only interpret uncertainty, e.g., where the state of the robot and/or obstacle is at the current and future time steps, but how we can harness it towards improved path planning capability. The problem described above is typically considered as solving the Active Simultaneous Localization and Mapping (Active SLAM) problem. Active SLAM is a research topic that deals with how the motion of the robot is affected while dealing with localization uncertainty (based on mapping the features of the robot's environment).

In Sec. 4 we demonstrate how we combine ML and traditional control techniques, along with mapping, to help provide robots with Active SLAM capability while being computationally efficient. This method was successfully deployed on a complex quadruped robot.

Because one of my main motivations is to demonstrate algorithms that can work on any robot and even robot teams, the next objective of this work is to expand the previous Active SLAM work for heterogeneous robot teams, see Sec. 5.

This end-to-end motion planning algorithm, named SABER (i.e., Stochastic model predictive control for Autonomous Bots in uncertainty Environments using Reinforcement learning), is demonstrated for a UAV-UGV system that fuses traditional controls, vision, and ML for collision avoidance and navigation. While this robot system moves to their individual goal points, which are generated using reinforcement learning, uncertainty is considered using SLAM algorithms, where the uncertainty from cross communications (when the robots are in range of each other) is also accounted for.

Because Active SLAM typically requires various components such as Kalman filtering and planners or controllers, these components need to be finely tuned for optimal performance. Tuning all of these components can be very challenging. Thus, I discuss a method to automate the tuning and calibration required for controlling and even planning for robotic systems, see Sec 6, Sec 7, and Sec 8. These methods help facilitate robots to continuously learn during operation, enabling constant adaptation and improvement as it interacts with its environment. The end goal of using this method, is to combine it with machine learning for guided and faster training through model-based approaches.

Because planning, control, or machine learning algorithms applied to hardware eventually demand a high-fidelity state estimation system. In Sec. 9, I describe a state estimation algorithm that combines model and learning-based methods to formulate a state estimation system, and apply it to a legged robot system for evaluation.

Finally, with the rise of Large Language Models, it would be irresponsible to ignore them in the context of the future development of robotic systems. I provide some explanation of them in Sec. 10, and also showcase a simple example of using them for high level planning.

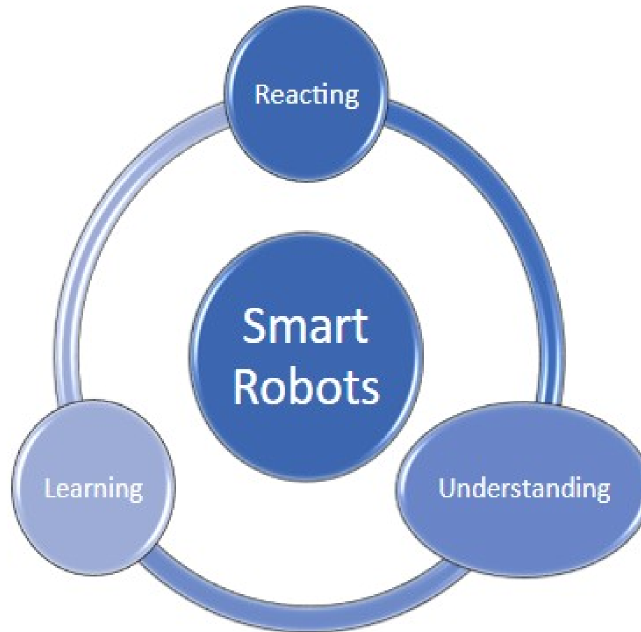


Figure 1.1: **Smart Robots.**

1.3 Contributions

(1) We will demonstrate how physical intelligence (reacting) can be imbued in robotic systems by simply optimizing energy efficiency for legged locomotion. The simplicity is demonstrated by not requiring any solver or machine learning methods.

(2) We demonstrate how understanding the robot's surroundings through a representation of state uncertainty leads to more intelligent decision-making for obstacle avoidance. This is shown through the following:

   (a) We evaluate the feasibility of an online end-to-end path planner that unifies MPC, SLAM, RNN, and an object detector using CNNs to generate paths for unknown and uncertain environments using a non-linear programming solver.

(b) We verify that our quadrupedal robot, ALPHRED, avoids collisions and computes a shorter trajectory while maintaining safety using our method as compared to a more conservative and naïve planner.

(c) We propose a novel use of RNNs to estimate positional uncertainties at all future timesteps of the MPC's prediction horizon.

(d) We integrate all components into a high-fidelity simulation using the quadruped dynamics of ALPHRED. Additionally, we test all components individually either online or offline using hardware.

(3) We expand the contribution in (2) for a heterogeneous team of robots using a UGV and UAV. Using the semantic learning that reinforcement learning can provide, we show how a team of robots can continually learn to collaborate towards some shared objective. This is shown through the formulation and implementation of a new planning algorithm called SABER: Stochastic model predictive control for Autonomous Bots in uncertainty Environments using Reinforcement learning:

(a) SABER is an end-to-end motion planning framework for a team of heterogeneous robots that unifies controls, vision, and machine learning approaches to plan paths that account for safety, optimality, and global solutions (our complete framework is shown on a UGV-UAV team).

(b) Cooperative localization algorithms are used for cross-communicating robots, which may include both non-Gaussian and Gaussian measurement noise, where uncertainty is modeled with recurrent neural networks (RNNs) for each agent's sensor configuration using outputs from simultaneous localization and mapping algorithms (SLAM).

(c) Instead of simple heuristics when sampling the map for target positions, we employ Deep Q-learning (DQN) for high-level path planning, which is easily modifiable for learning desired multi-agent behavior and finds global solutions (DQN scalability

for more than two robots is also evaluated).

(4) We employ and expand the method of auto-calibration of parameters towards planners, controllers, and for real to sim applications.

  (a) Specifically we provide an approach for auto-tuning feedback controllers and online trajectory planners to achieve robust locomotion of a legged robot. The auto-tuning approach uses an Unscented Kalman Filter (UKF) formulation, which adapts/calibrates control parameters online using a recursive implementation. In particular, this letter shows how to use the auto-tuning approach to calibrate cost function weights of a Model Predictive Control (MPC) stance controller and feedback gains of a swing controller for a quadruped robot. Furthermore, this letter extends the auto-tuning approach to calibrating parameters of an online trajectory planner, where the height of a swing leg and the robot's walking speed are optimized, while minimizing its energy consumption and foot slippage. This allows us to generate stable reference trajectories online and in real time.

  (b) We use this method to also calibrate the gains of an admittance controller for manipulation and climbing tasks, to help track reference wrench values.

  (c) Finally, the method was used to learn the residual errors between the real and simulator robot model. We use a neural network to do this, where we use the weights and bias values as our control parameters. This allows us to quickly calibrate these parameters during online operation directly on the real robot as the method is data efficient.

(5) We unify model and leaning-based modalities for state estimation of legged robots. We present the following contributions within the state estimation domain:

  (a) We combine a model-based Kalman filter with the ability of learning nonlinearities through a GRU and ViT. Our estimator provides both the robot's trunk state and the associated predictive uncertainty.

(b) For our Kalman filter, we consider joint encoder and IMU measurements, and reuse the control outputs from a convex MPC for the filter's system model.

(c) We demonstrate our state estimator on hardware using a quadrupedal robot on various terrains and demonstrate a 65% improvement against a state-of-the-art VIO Simultaneous Localization and Mapping (SLAM) baseline [56] using the Root Mean Squared Error (RMSE) as our validation metric.

1.4 Necessary Background Knowledge

Throughout this dissertation, I will include topics that span from Model Predictive Control to terminology typically used in Legged robot systems. For ease of reading, I briefly describe these topics here.

1.4.1 Model Predictive Control

Model Predictive Control (MPC) involves using current measurements to predict future values or outputs [62, 61]. Similar to *feedforward* control, MPC outputs solutions based on a future reference point. The key difference is that MPC can adapt to a changing reference trajectory and optimize the solution at every time interval through online calculations. This allows the robot to respond to both unexpected disturbances and a time-varying reference. The objective of MPC is to compute a series of *control values* (e.g., jerk) that adjust the input or *state* variable (e.g., position, velocity, and/or acceleration) to follow a *reference trajectory*. Figure 1.2 demonstrates the concept of MPC, where x is the predicted output and u is the control variable. At the current sampling instant, iteration k, MPC calculates a set of N control values $[u(k+i-1), i = 0, 1, 2, 3, 4, \ldots, N-1]$ over a time interval T, also known as the *prediction horizon* (equal to $N \cdot dt$). These predicted outputs, derived from control variables u, form a set of x values optimized to follow the reference trajectory $[x(k+i), k = 0, 1, 2, 3, 4, \ldots, N)]$. The optimization of x values based on control u is determined by the cost

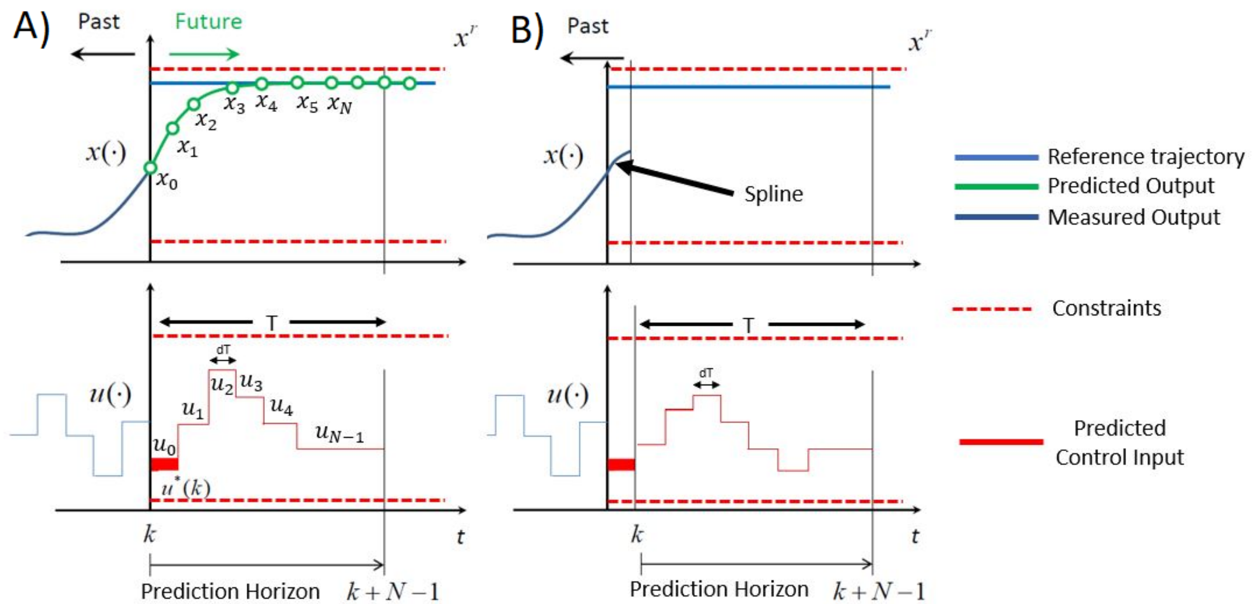


Figure 1.2: **General concept of Model Predictive Control**

[89]

function or objective function [62]. The objective function minimizes the weighted sum of squared predicted errors (the difference between predicted output and reference trajectory) and the squared control values, all subject to hard constraints.

Although N control values are computed at each time interval T, only the first control output u is applied, or $u(k)$ in Figure 1.2, because the system might not accurately track the reference. Predicted motion often differs from actual motion due to modeling errors, sensor noise, or unexpected disturbances. By using only the first control output u at each interval, the controller aims to provide the optimal output u that minimizes the cost function. By holding the first output solution $u(k)$ constant over dt, a spline can be created from $x(k)$ to $x(k+1)$. The MPC time horizon then shifts by one step and the process repeats.

To optimize their solutions, users can adjust the Q and R matrices (positive semi-definite gains), modify the prediction horizon N (higher N improves reference trajectory tracking at the cost of increased computation time), and set appropriate constraints. While MPC offers

many advantages over PD control, poor choices for N or constraints, especially if the plant model is not fully understood, can result in erroneous outcomes.

1.4.2 SLAM

Simultaneous Localization and Mapping or SLAM can be described mathematically as the following:

Given a series of control inputs u_t and sensor observations o_t over discrete time steps t, the SLAM problem aims to estimate the agent's state x_t and the map of the environment m_t. These quantities are typically probabilistic, thus the objective is to determine:

$$P(m_{t+1}, x_{t+1} \mid o_{1:t+1}, u_{1:t})$$

By applying Bayes' rule, we can sequentially update the location posteriors, given a map and a transition function $P(x_t \mid x_{t-1})$,

$$P(x_t \mid o_{1:t}, u_{1:t}, m_t) = \sum_{m_{t-1}} P(o_t \mid x_t, m_t, u_{1:t}) \sum_{x_{t-1}} P(x_t \mid x_{t-1}) P(x_{t-1} \mid m_{t-1}, o_{1:t-1}, u_{1:t})/Z$$

In a similar manner, the map can be updated sequentially by

$$P(m_t \mid x_t, o_{1:t}, u_{1:t}) = \sum_{x_t} \sum_{m_{t-1}} P(m_t \mid x_t, m_{t-1}, o_t, u_{1:t}) P(m_{t-1}, x_t \mid o_{1:t}, m_{t-1}, u_{1:t})$$

As with many inference problems, the solutions to estimating both variables together can be achieved, reaching a local optimum, by alternately updating the two beliefs in a form of an expectation-maximization algorithm. Further, in this prospectus, I will also describe Active SLAM, which can be thought of as Simultaneous Planning Localization and Mapping. In other words, we still use the traditional SLAM setup, however, we add a planner to this methodology. In other words, the localization of the robot as well as the environment directly affects the behavior of a motion planner.

1.4.3 Reinforcement Learning

Reinforcement Learning (RL) is a type of machine learning where an agent learns to make decisions by interacting with an environment to maximize cumulative reward. The agent takes actions in discrete time steps. At each time step t, the agent receives a state s_t, selects an action a_t based on a policy π, and receives a reward r_t from the environment. The goal is to learn a policy that maximizes the expected cumulative reward, represented as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

where γ is the discount factor.

1.4.3.1 Deep Q-Learning

Deep Q-Learning is an extension of Q-Learning that uses deep neural networks to approximate the Q-value function, $Q(s, a)$, which represents the expected cumulative reward of taking action a in state s. The Q-Learning update rule is given by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right]$$

In Deep Q-Learning, a neural network with parameters θ is used to approximate $Q(s, a; \theta)$. The loss function for training the neural network is:

$$L(\theta) = E_{(s_t, a_t, r_t, s_{t+1})} \left[\left(r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right]$$

where θ^- are the parameters of a target network that are periodically updated to stabilize training.

1.4.3.2 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a policy gradient method for RL that improves training stability and performance. PPO aims to optimize a surrogate objective function while ensuring the new policy does not deviate too much from the old policy. The PPO objective function is:

$$L^{CLIP}(\theta) = E_t \left[\min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t, \text{clip} \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon \right) \hat{A}_t \right) \right]$$

where π_θ is the new policy, $\pi_{\theta_{\text{old}}}$ is the old policy, $\hat{A}_t$ is the advantage estimate, and ϵ is a hyperparameter that controls the clipping range.

PPO strikes a balance between exploration and exploitation by ensuring the updates are within a small trust region, thereby maintaining the stability and efficiency of the learning process.

1.4.3.3 Conclusion

Reinforcement Learning encompasses various algorithms, each suitable for different types of problems. Deep Q-Learning and PPO are two powerful algorithms that leverage neural networks and advanced optimization techniques to solve complex RL tasks. These methods have been successfully applied to various domains, including robotics, games, and autonomous systems.

1.4.4 Neural Networks and its Variants

Neural networks are a class of machine learning models inspired by the human brain's architecture. They are particularly effective in handling complex data and tasks. Below, we summarize different types of neural networks, including Recurrent Neural Networks (RNNs) and their variant Gated Recurrent Units (GRUs), Convolutional Neural Networks (CNNs),

and Vision Transformer (ViT).

1.4.4.1 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are designed to handle sequential data. They maintain a hidden state that captures information about previous inputs in the sequence. The hidden state h_t at time step t is updated based on the current input x_t and the previous hidden state h_{t-1}:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b)$$

where W_h and W_x are weight matrices, b is the bias, and σ is an activation function, typically the hyperbolic tangent (tanh).

1.4.4.2 Gated Recurrent Units (GRUs)

Gated Recurrent Units (GRUs) are a variant of RNNs that aim to mitigate the vanishing gradient problem. GRUs use gating mechanisms to control the flow of information. The update and reset gates determine how much of the previous state should be passed to the current state. The GRU equations are:

$$z_t = \sigma(W_z x_t + U_z h_{t-1})$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1})$$

$$\tilde{h}_t = \tanh(W_h x_t + r_t \odot U_h h_{t-1})$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

where z_t is the update gate, r_t is the reset gate, $\tilde{h}_t$ is the candidate hidden state, and $\odot$ denotes element-wise multiplication.

1.4.4.3 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are specialized for processing grid-like data, such as images. They use convolutional layers to extract features from the input data. A convolutional layer applies a set of filters to the input, producing feature maps. The operation is defined as:

$$(f * x)(i, j) = \sum_m \sum_n x(i + m, j + n) \cdot f(m, n)$$

where f is the filter, x is the input, and (i, j) are the coordinates of the output feature map.

CNNs typically consist of multiple convolutional layers, followed by pooling layers and fully connected layers. The pooling layers reduce the spatial dimensions, retaining important features while reducing computational complexity.

1.4.4.4 Vision Transformer (ViT)

Vision Transformer (ViT) is an advanced neural network architecture that applies the transformer model to image data. Unlike CNNs, ViTs do not use convolutions to extract features. Instead, they divide the image into patches and process these patches using self-attention mechanisms. The steps involved in a ViT are:

1. Divide the image into fixed-size patches. 2. Flatten each patch and map it to a lower-dimensional embedding space. 3. Add positional embeddings to retain spatial information. 4. Process the sequence of patch embeddings using transformer encoder layers.

The self-attention mechanism in each transformer encoder layer is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where Q, K, and V are the query, key, and value matrices, respectively, and d_k is the dimensionality of the key vectors. They are employed in different ways throughout this

prospectus.

1.4.5 Kalman Filtering and its Varients

Kalman filtering is an algorithm that provides estimates of unknown variables over time by using a series of measurements observed over time, containing statistical noise and other inaccuracies. Kalman filters are widely used in various applications, such as navigation, tracking, and control systems. The standard Kalman Filter (KF), Extended Kalman Filter (EKF), and Unscented Kalman Filter (UKF) are its common variants.

1.4.5.1 Kalman Filter (KF)

The Kalman Filter is optimal for linear systems with Gaussian noise. It consists of two main steps: prediction and update.

Prediction Step:

$$\hat{x}_{k|k-1} = A\hat{x}_{k-1|k-1} + Bu_k$$

$$P_{k|k-1} = AP_{k-1|k-1}A^T + Q$$

Update Step:

$$K_k = P_{k|k-1}H^T(HP_{k|k-1}H^T + R)^{-1}$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(z_k - H\hat{x}_{k|k-1})$$

$$P_{k|k} = (I - K_kH)P_{k|k-1}$$

where $\hat{x}$ is the state estimate, P is the error covariance matrix, A is the state transition model, B is the control-input model, u_k is the control vector, Q is the process noise covariance, H is the observation model, R is the measurement noise covariance, K_k is the Kalman gain, and z_k is the measurement at step k.

1.4.5.2 Extended Kalman Filter (EKF)

The Extended Kalman Filter is used for non-linear systems by linearizing the system around the current estimate. The process and measurement models are given by:

$$x_k = f(x_{k-1}, u_k) + w_k$$

$$z_k = h(x_k) + v_k$$

where f and h are non-linear functions, and w_k and v_k represent process and measurement noise.

Prediction Step:

$$\hat{x}_{k|k-1} = f(\hat{x}_{k-1|k-1}, u_k)$$

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k$$

Update Step:

$$K_k = P_{k|k-1} H_k^T (H_k P_{k|k-1} H_k^T + R_k)^{-1}$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(z_k - h(\hat{x}_{k|k-1}))$$

$$P_{k|k} = (I - K_k H_k) P_{k|k-1}$$

where $F_k = \frac{\partial f}{\partial x}\Big|_{\hat{x}_{k-1|k-1}, u_k}$ and $H_k = \frac{\partial h}{\partial x}\Big|_{\hat{x}_{k|k-1}}$ are the Jacobian matrices of f and h.

1.4.5.3 Unscented Kalman Filter (UKF)

The Unscented Kalman Filter addresses the limitations of the EKF by using a deterministic sampling technique called the Unscented Transform to better capture the mean and covariance estimates.

Prediction Step:

Generate sigma points χ_{k-1} from $\hat{x}_{k-1|k-1}$ and $P_{k-1|k-1}$:

$$\chi_{k-1} = \hat{x}_{k-1|k-1} \pm \sqrt{(L+\lambda)P_{k-1|k-1}}$$

Propagate sigma points through the process model:

$$\chi_{k|k-1} = f(\chi_{k-1}, u_k)$$

Calculate predicted mean and covariance:

$$\hat{x}_{k|k-1} = \sum_i W_i^{(m)} \chi_{k|k-1}^{(i)}$$

$$P_{k|k-1} = \sum_i W_i^{(c)} (\chi_{k|k-1}^{(i)} - \hat{x}_{k|k-1})(\chi_{k|k-1}^{(i)} - \hat{x}_{k|k-1})^T + Q$$

Update Step:

Generate sigma points from the predicted state:

$$\chi_{k|k-1} = \hat{x}_{k|k-1} \pm \sqrt{(L+\lambda)P_{k|k-1}}$$

Propagate sigma points through the measurement model:

$$\zeta_k = h(\chi_{k|k-1})$$

Calculate predicted measurement mean and covariance:

$$\hat{z}_k = \sum_i W_i^{(m)} \zeta_k^{(i)}$$

$$S_k = \sum_i W_i^{(c)} (\zeta_k^{(i)} - \hat{z}_k)(\zeta_k^{(i)} - \hat{z}_k)^T + R$$

Calculate cross-covariance and Kalman gain:

$$P_{xz} = \sum_i W_i^{(c)} (\chi_{k|k-1}^{(i)} - \hat{x}_{k|k-1})(\zeta_k^{(i)} - \hat{z}_k)^T$$

$$K_k = P_{xz} S_k^{-1}$$

Update the state and covariance estimates:

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(z_k - \hat{z}_k)$$

$$P_{k|k} = P_{k|k-1} - K_k S_k K_k^T$$

Kalman filtering and its variants, including the Kalman Filter (KF), Extended Kalman Filter (EKF), and Unscented Kalman Filter (UKF), provide robust methods for estimating the state of a system over time. While KF is suitable for linear systems, EKF and UKF extend its application to non-linear systems, with UKF offering improved performance by better capturing the mean and covariance through the Unscented Transform.

1.4.6 Legged Robot Terminology

In this propsectus, there are certain terminology used that are specific to legged robot systems. For example, I will sometimes use phrases such as stance and swing to describe the various motions of a legged robot. Specifically, stance describe the feet of the robot currently in contact with the ground. As soon as this foot loses contact with the ground (i.e., leg is in the air), we describe this foot as being in swing mode. Essentially, stance and swing describe the discrete hybrid nature of a legged robot. Although we provide more details in terms of control of these legged systems, the act of balancing such a system is done through the stance leg – essentially, the robot uses its stance leg to push on the ground and ensure it can balance and avoid falling. To move in a desired direction, the robot will also push on the ground (to produce some force in that direction), while lifting one or more of its legs

simultaneously so that the base position can translate successfully to a new position. The amount of time a leg is either in stance or swing phase is called the gait timings. Finally, depending on the sequence of both timings and how many legs are on the ground/air at the same time, describe the specific gait. For example, for a quadruped (or 4 legged robot), if 1 leg is in the air while 3 legs are on the ground, the most typical gait is called a walking gait. If 2 legs are in the air while 2 legs are on the ground, the most common gait is called trot gait. Although there are various combinations of timings to produce other gaits, in this work we only use trot and walk gait.

CHAPTER 2

Related Works

2.1 Motion Planning Problem

The primary objective of this thesis work is to demonstrate various ways to make robots more intelligent or 'smart' by demonstrating implementation of *reacting*, *understanding*, and *learning* through formulation of new PCE algorithms. We first start with the importance of *reacting*. In real-world scenarios, the robot will be disturbed either by the environment or by others during operation. It is important that the robot can adequately react to these changes to keep its center of mass stable at all times. Looking towards the example of human motion, reaction and energy-efficiency are typically intertwined in the sense that the ability to react quickly and accurately to environmental stimuli depends on the efficiency of our motor control system. Specifically, energy-efficient motion is achieved through a combination of factors, including the optimization of movement patterns and the minimization of wasted energy. When the body moves in an energy-efficient manner, it requires less metabolic energy, allowing for sustained activity over longer periods.

In the context of reaction time, an efficient motor control system allows faster, precise reactions by minimizing wasted energy. For instance, quick directional changes to avoid obstacles become easier and less energy-consuming. An inefficient system leads to delayed reactions, more energy use, fatigue, and reduced performance. Training can improve motor control efficiency, resulting in quicker reactions and sustained activity.

This principle applies to legged robots, where stable motion is crucial. One simple,

effective method is the Raibert controller [107], though it's less suited for rough terrain. Optimization techniques like the Linear Inverted Pendulum use the Center of Mass to maintain balance [69], but require pre-specified footsteps, limiting workspace. Trajectory Optimization (TO) can predict future steps [143], but is computationally heavy for real-time use. Reinforcement Learning (RL) offers another approach [66, 85, 74], but demands extensive data and tuning, making it complex.

Our work aims for an easy-to-implement, low-cost, energy-efficient path planner. Similar work [126] minimizes motor energy while adhering to ZMP constraints, but requires predefined step heights and distances. Our approach dynamically adjusts step height and timing based on current velocity.

Another study [28] uses a neural network for terrain-based footstep planning, ensuring optimal energy efficiency and stability. Although we don't use vision-based data, our method also aims to avoid slipping using force ellipsoids.

Other research [145, 129, 75] focuses on optimizing locomotion parameters for energy efficiency but doesn't address gait transitions. Our objective is to determine the optimal velocity for transitioning gaits to save energy. Gait transitions are complex due to hybrid dynamics and contact dynamics, often requiring high computational costs. Recent RL studies like [49] tackle this by learning energy-efficient gaits. Unlike model-based controllers that rely on contact-implicit optimization [25, 90], our approach provides an analytical solution, offering clear physical relationships between dynamics, gaits, and energy consumption without needing predefined contact gaits.

Our method stands out for its simplicity, lack of learning or solver requirements, and general applicability across different gaits.

Although *reacting* to external disturbances can be done considering only the dynamical physics of a robot, *understanding* the surrounding demands perception and new techniques for considering uncertainty and guarantee some metric of safety during motion planning.

However, planning in environments that are not well-known is a daunting challenge because of issues related robustness [3].

For example, simultaneous planning, localization, and mapping (SPLAM, or more commonly referred to as Active SLAM) is an area of research that attempts to satisfy some of these requirements. The main challenges to Active SLAM consist of planning under uncertainty in an acceptable amount of time, bridging the gap between sensor data (semantic mapping), and ensuring that it is robust enough for a complex platform. Because of these challenges, there are many works addressing a subset of Active SLAM, namely simultaneous localization and planning [2] or works addressing SLAM, such as [98, 68, 41]. Other Active SLAM works also use very simple research platforms [70] or were only tested in simulation [81].

There are two common frameworks to address the problem of planning under uncertainty, which are explicitly modeling the posterior distributions in a Bayesian setting [47]. Unfortunately, modeling the posterior distributions in a Bayesian setting may be unrealistic, as the Bayesian setting is only computationally tractable for the simplest cases (e.g., Gaussian prior and Gaussian observations). One critical attribute of a POMDP is the assumption that states are not observable (unlike MDPs). Practically, this means that the agent cannot decide an action based on the current states alone – instead, it must consider the complete history of past actions and observations to choose its current action. Consequently, using POMDP to solve the Active SLAM problem becomes computationally intractable due to the curse of history and dimensionality (necessity of having to track a longer and longer history of past state and action pairs), and does not sufficiently model an agent's future intent [42]. Moreover, there is a need for new path planning architectures for unknown and uncertain environments that addresses the concerns of belief space planning or provides alternative methods that can be ubiquitously applied on most robotic systems.

Besides *reacting* and *understanding*, *learning* is the other component that I believe is necessary to making robots smarter and feasible to be useful in real-world applications. One

of the more obvious ways to integrate a sense of learning for robotic systems, is in the field of reinforcement learning. The primary reason why *learning* is important is in situations when the solution to some problem may not be so obvious that it can be adequately addressed through traditional planning methods while being computationally feasible. For example, how do we plan paths for a heterogeneous team of robots (i.e., robots with varying dynamics or locomotion abilities) that is both safe (e.g., considers uncertainty in agent/obstacle avoidance) and fast (e.g., finding the shortest path to the goal)? In addition to solving a path that is both safe and fast, how can this path be generated by also considering the communication abilities of multiple robots – for example, before reaching a desired goal state for each individual robot, for some amount of time, it may be desirable for them to be in proximity so they can combine their sensor measurement and decrease their overall state uncertainty. This level of complexity can be incredibly challenging without learning-based methods, which have the advantage (particularly when simulations are available) to fully explore the state-space of the system.

I now provide some examples of past works that address heterogeneous path planning and explain how our work (described in more detail in Sec. 5), using a combination of traditional planning/control methods with machine learning can provide additional functionality not so easily addressed with only traditional planning methods. For instance, works that examine planning for heterogeneous robots (typically composed of a two robot UGV-UAV team) have focused mainly on fusing different sensor data to build a unified global map [73], integrating several components such as path planning, sensor fusion, mapping, and motion control towards a single framework [103], or strictly analyzing multi-agent localization (i.e., multi-SLAM) [138]. While we also consider a UGV-UAV team as done in the above works, here, we are more concerned with the feasibility (i.e., computation time) of such a complex system, and also in how uncertainty is not only estimated for robots with different sensor configurations, but how it's tightly coupled with a local stochastic model predictive controller (SMPC) towards coordinating multi-agent behavior.

We also seek to address the major challenge for multi-agent (or even single-agent) planners, which is to estimate a path that is both safe (e.g., considers uncertainty in agent/obstacle avoidance) and fast (e.g., finding the shortest path to the goal). This is significant because conservative approaches to safety would lead to over avoidance or non-optimal solutions, and high-risk behavior may cause undesired collisions. Currently, few motion planners fully investigate this problem. For example, in [28], the cost of reaching a target position for each robot in a heterogeneous team depends on its individual characteristics (e.g., varying sensors, travelling speed, and payloads). However, by not considering uncertainty in their planner, their cost-to-go function can be significantly affected by disturbances. Conversely, a multi-agent planner that does consider probabilistically-safe motion planning can be found in [8]. Still, their planner may lead to conservative solutions as they assume a worst-case behavior approach to safety. SABER addresses the problem of avoiding obstacles without over avoidance by using an RNN, which predicts and propagates future state uncertainty dynamically and does not make the assumption that uncertainty increases when no future measurements are received [120]. The downside with an RNN (which is typical with learning-based approaches) is that the accuracy of the uncertainty estimates is directly correlated with the quality of the data collection.

The geometric representation of obstacles is also critical for planning. For example, FASTER [31] is a decentralized and asynchronous planner where obstacles are represented as outer polyhedrals (estimated from convex decomposition) and applied as constraints into the optimization. In SABER, we also represent obstacles as polyhedral constraints for each timestep, however, we decompose them into a disjunction of linear chance constraints (thus, obstacle 'size and location' are a function of exteroceptive and proprioceptive uncertainty). Using chance constraints in motion planning is not new, and has been shown with success in single-agent planners such as [124] and [83]. In this work, our chance constraints are also influenced by the cross-communication uncertainty of a heterogeneous robot team. Additionally, while obstacle constraints can be explicitly used in the optimization, other works show a

learning-based approach to avoid collisions by modeling the distribution of promising regions for travel [29] or predicting the separation distance between the robot and its surroundings [26]. Our work is a hybrid approach, where we use the RNN to predict uncertainty of state estimates (which affect the 'size' of polyhedral obstacles), but still use these obstacles as constraints in our SMPC optimization. This choice sacrifices computation time but may be more generalizable to environments not observed in training and prevent collisions.

Finding a suitable path to the goal also has a wide array of different solutions. Most commonly, sampling-based methods which do not consider workspace topology (such as grid-sampling [77] or rapidly-exploring random tree or RRT and its variants [50]) can lead to very dense roadmaps and may not scale well when the shortest path to the goal is desired. This issue has been investigated in [6], which uses a self-supervised learning approach to build sparse probabilistic roadmaps (PRM) for bias sampling (sampling only regions the robot is likely to safely travel). Moreover, using a learning-based approach for path planning also has the potential for integrating semantic behavior that can be gained from multi-agent coordination, as evidenced in [142] and [9]. Motivated by these works, we use a DQN for high-level planning, where simple modifications of a reward function can yield desired multi-agent behavior (e.g., rewarding agents based on proximity or reaching the goal concurrently). Nevertheless, the tradeoff of using a DQN compared to sampling-based methods is that it cannot guarantee asymptotic optimality (e.g., RRT-star) or probabilistic completeness (e.g., PRM). However, for our DQN, we are primarily concerned with the feasibility relative to our application (i.e., finding a near-optimal path in a computationally efficient manner while satisfying multi-agent behavior).

2.2 Auto-Calibration

Due to the inverse relationship between the complexity of the dynamic model used for locomotion and the computation time required for an optimization solver, a popular approach is

to employ Reinforcement Learning (RL) [136, 80, 66, 85, 74, 122]. E.g., in [136], a stochastic policy is realized with neural networks to simulate a foothold and base motion controller using Trust-Region Policy Optimization. This is done by considering a reward function that consists of several physical parameters such as the error between actual and reference footstep position, penalizing foot slippage or large swing-leg velocities, or sudden changes in base orientation. An RL approach has also been successfully demonstrated in [80], which learns a neural network that acts only on a stream of proprioceptive signals for locomotion on uneven terrain. As part of the *learning* part as described in the introduction, which naturally lends into the machine learning field, there are also alternative approaches to robustify and adapt from controller/planning errors. Although tremendous progress has been made in the RL literature to achieve legged robot locomotion, this approach is highly complex and requires significant amounts of data. For example, in [80], a two-stage training process is required that involves a teacher/student policy in addition to particle-filtering, which maintains certain terrain parameters used to classify what is and is not traversable during training. In [136], the reward function and corresponding weighing parameters required intricate fine-tuning and assumes the user has extensive computational resources available during training. Another work can be found in [51], which uses RL to create adaptive trajectories based on following a reference trajectory calculated offline first through TO. Similar to our work, they also use step clearance and slippage as objectives to their calibration. However, different from ours, while we also use TO methods, we only rely on the reference trajectory from TO for the first few footsteps, while future footsteps are planned online. Overall, the complexity of formulating the correct reward function in RL and the effort required for tuning hyper-parameters may cause significant delay in development time. This also may make RL difficult to reproduce or generalize to a wide-variety of robotic platforms without significant amounts of tuning.

Force control strategies are worthwhile pursuits as they have already been shown, compared to ignoring force feedback, to deliver adequate control for diverse tasks, from insertion

with tight tolerances to the polishing of non-flat surfaces [141]. So far, one area that is relatively less explored is the control of wrenches for dexterous manipulation, which typically involve one or more fingertips that must physically interact with the environment and objects (i.e., multi-finger robots) [79]. Instead, the research focus in dexterous manipulation has been on object recognition or localization with fully-actuated grippers, where grasping motion is applied through an on/off mechanism, where the gripper is either in a fully open or fully closed configuration [84]. More complex controls for grasping have been considered using tactile feedback and even learning-based approaches through teleoperation [82, 87]. While these works incorporate wrench information, their goal is to control the relative motion between the gripper and its target object than explicitly track a desired wrench. However, tracking wrenches can be necessary if we have an underactuated gripper or make contact in very compliant environments, as they are difficult to control, and are endowed with the ability to envelop objects of different geometric shapes. Still, control of underactuated grippers has been shown in past work by estimating force but not controlling for them directly in [10] or controlling force directly but without hardware results in [54]. In our work, we will primarily focus on tracking wrench profiles directly using force-feedback with an auto-calibrating admittance controller that can be used not only for single-point contacts, such as legged robots with point feet [125], but also for multi-finger robots, such as manipulators with grippers [10], on both stiff and compliant environments. Note, that we opt for an admittance force controller because we assume access to F/T sensors, and will use a change in task-space position for control after obtaining these sensor measurements.

The reason we employ a self-calibrating admittance controller instead of using traditional admittance control [59] is that the controller can quickly become unstable if no modifications are made. For instance, admittance control requires inverse kinematics, where the position of the end-effector as a control parameter can cause kinematic singularities. Also, task and joint space control require a complete dynamic model of the robot, contributing to the accuracy/robustness dilemma [59]. Recently, model error compensation techniques for

impedance/admittance control have been shown through adaptive control schemes, although knowledge of the dynamic model is still required. One approach to resolve model-based errors is demonstrated in [149], which modifies the admittance controller by using only the orientation components of the end-effector through relating orientation with joint angles, avoiding the need for inverse kinematics. However, this approach can only be used if the limb's joint space is R^6, ours is R^7. Additionally, deriving orientation with joint angles may be difficult if the gripper is too complex or underactuated. Other methods that are model-free, such as neural networks, may still result in imprecise control and demands a large number of training sets to achieve robustness under unseen conditions [71].

Alternatively, to modify our admittance controller, we use the method of [94] (which employs a UKF) over other adaptive or machine-learning methods as it does not require a trial-and-error implementation and can be directly applied to hardware without the need for complex simulation models–which may otherwise be difficult to achieve with a physically complex manipulator/gripper and in uncertain environments. The method also allows the freedom to choose any number of desired training objectives toward stabilizing the controller, such as closely tracking wrench profiles, predicting the spring constant of the environment to update the dynamic model, ensuring control outputs are within limits to minimize slipping, and avoiding kinematic singularity.

Although other works exist in tracking dynamic forces in uncertain environments, such as [36, 24], their experimental setup is relatively simple and they only show results for tracking force and not torque. By using a UKF, we can also more easily consider nonlinearities imposed by the system. However, the main limitation of our approach relative to those works, is that it is more challenging to provide mathematical guarantees on controller stability. For instance, the admittance controller gains must be positive semi-definite to be stable. To circumvent this issue, we add a small modification to [94], by adding a large cost term on the training objective when the sigma points or controller gains sampled by the UKF violate the semi-definite property. We perform a similar operation to avoid kinematic singularities

as well. Doing so, the algorithm will not select gains that violate these properties.

The UKF auto-tuning concept was also applied to real to sim applications. To provide context to this problem, the reality gap between policies trained in simulations and then applied to real-world tasks are typically addressed through domain randomization, domain adaptation, or system identification methods. In domain randomization, a model is trained across simulated environments that vary in its dynamics or visual information. The real-world environment is then assumed to be generalizable as a sub-variant of these randomized environments [35]. However, as sample complexity exponentially increases with the number of randomizations, the work in [92] has shown that domain randomization (in general) leads to suboptimal and high variance policies. To resolve the increase of sample complexity over time, [100] addresses this through automatic domain randomization, which expands the range of parameters autonomously during the agent's learning procedure. However, by not using real-world data, their final distribution of parameters may not sufficiently reflect the actual distribution of the real world – a common issue in this domain [35]. Another approach is to use real-world data directly to influence simulation parameters. For example, [27] uses continuous object tracking with real-world data to compare trajectories between the real and simulated model. Other works have also employed model-based RL and use real-world data to learn a policy that fits some probabilistic model [33, 38]. Still, using machine learning only on real-world data to address the reality gap can be time consuming and impractical depending on the task or environment complexity, as current methods rely on sufficient data collection and may lead to safety issues for real robots during the lengthy training execution.

A hybrid approach (between using real-world and simulated data) exists through domain adaptation methods. [67] achieves desired manipulation tasks by learning a policy on the real robot through a collection of unlabeled real-world images, which are then employed along with simulated images to adapt the policy (effectively decreasing the number of real-world data required). However, while [67] does not improve the simulator itself during the learning process, [35] does improve the simulator by matching reality and reduces (over time)

the need for real-world data. Although this hybrid approach has been used with success, it requires a careful and at times complex procedure for generating new simulation data, and the agent's policy may struggle to learn if the distribution of the simulation parameters grow too large [67, 101]. Thus, while RL has made significant progress in handling the error associated with the real and simulated robot model, resolving real-world data sparsity along with generalizing the training process across environments/robots remains a consistent issue.

Another approach is to apply methods of system identification and Bayesian optimization, such as in [63] which use prior knowledge of the system. These works typically only account for geometric parameters and do not consider non-linearities that arise from kinematic chains and noise in system propagation (although these methods benefit from being applicable on the real world directly without learning requirements). Further, to achieve the accuracy required in the task-relevant state-space domain, non-geometric errors need to be accounted for including friction, temperature, and compliance which is difficult and potentially infeasible to model without learning-based methods [5, 139].

In this work, our goal is to make it feasible to train on real-world data directly to continuously improve the simulated model, i.e., Real-to-Sim (or sometimes referred to as modelling a 'digital twin' [139]) using a learning-based UKF approach. Other works similar to our goal but different in approach can be found in [32], where the forward kinematics are modeled by optimizing Denavit–Hartenberg parameters through standard gradient descent (SGD) methods. Our work differs as we are not restricted to robots with only revolute or prismatic joins but is generalizable to any robot type (e.g., wheeled robots). Another work in this space is found in [101] which learns the non-parametric model error in addition to the existing (uncalibrated) forward model. Similar to our work, [101] applies a trade-off between modelling the non-geometric effects by hand and removing the available kinematic model completely. They achieve this through obtaining accurate forward models in the presence of non-linearities by learning residual errors using Gaussian Process Regression (GPR). However, because these residual errors (or in our case, model mismatch from proprioceptive and exteroceptive noise)

is not only non-linear but non-Gaussian, we can make use of neural networks instead [123, 121].

Ultimately, our approach is to directly learn the model or residual error between a desired or reference model and the current model. The reference or current model may be composed of a simple dynamic model, the state feedback of the simulator, or the state feedback of the real robot (in our case either a mobile or 6 degrees of freedom manipulator robot). To learn this model error we employ the method described in [93], which uses a UKF to predict control parameters with user-defined training objectives to evaluate the performance of a closed-loop system online through a recursive implementation. This method has previously shown success for multiple applications, from tuning the cost function weights of an LQR controller for autonomous vehicles [93] to tuning a model predictive controller and trajectory planner of a quadruped robot [117]. Here, we apply this method for tuning the weights of a neural network toward learning the model error. We use a neural network because we assume this model error is non-linear and non-Gaussian and may potentially learn semantic information that is difficult to account for manually. The motivation of using the method described in [93] (against RL or other adaptive methods for example), is because this approach has been shown to update control parameters with small amounts of data [117], and can handle non-linearity in the data due to the UKF formulation. Additionally, as this approach is applied online and recursively, the method can quickly adapt to model changes or sudden change in the localization result that affects the model error.

2.3 State Estimation

The spirit of unifying model-based and learning-base methods was also applied towards the domain of state estimation. We now describe using related works why such unification is useful to avoid the simplifications and assumptions of previous methods.

In [43], an Extended Kalman Filter (EKF) approach was used on the Atlas bipedal robot,

where the measurements include the velocity derived from the integration of leg odometry. An EKF was also employed in [1] to fuse leg kinematics and IMU information, and an a Unscented Kalman Filter (UKF) in [15] to better handle the nonlinearities of slippery terrain. However, these methods suffer from linearization errors [46]. Additionally, these works do not consider vision or other exteroceptive information which may impose drift over long term operation, which may be avoided in areas cameras can be applied.

One solution is to employ both proprioception and exteroceptive information through SLAM [37, 7], visual odometry [116, 48], or more currently, VIO SLAM [31, 23, 148]. However, while these works benefit from the simplicity and computational efficiency of Kalman filter methods, they are still limited by their requirement to approximate highly nonlinear systems and rely on sufficient initial conditions. In our case, employing a simple Kalman filter to estimate the robot's state, when utilized as input for the GRU, enhances the GRU's performance, especially concerning velocity components in unseen datasets (see Sec. 9.2.1).

Due to these issues associated with methods that rely purely on Kalman filtering, some works, such as in [147], employ state estimation through Quadradic Programming (QP), which considers both the modeling and measurement error within its cost function. However, while their QP works well in simulation, where their measurement noise appears relatively low, they do not compare their algorithm against ground truth on the real robot (e.g., using a motion capture system). Instead, they only compare their QP against a Kalman filter on the real robot without ground truth. Although our use of QP differs from the work in [147], using QP for state estimation is a realistic option due to the recent advancement of computers. In our case, we use a convex MPC QP program [34] which computes ground reaction forces that actuate a robot to help follow a user's reference trajectory. Thus, from their model formulation, these forces embed information of the user's reference trajectory (e.g., velocity command), and also the previous state estimator output. These forces are then reused as part of our Kalman filter's system model to propagate the state to the next time step.

In this work, we further correct any errors due to nonlinearities that occur from the model and measurements through a GRU. Using learning to correct for model errors is validated by several works. For example, using IMU-only state estimation on pedestrian motion [19] using a neural network, learning model/measurement errors through a Recurrent Neural Network (RNN) by removing bias errors and learning IMU noise parameters [151], and combing a UKF with a neural network to learn the residual errors between a dynamic model and the real robot [119]. However, many of these works, such as [151, 119, 115, 123], focus more on smoothing, imputation, or error prediction, rather than providing an end-to-end state estimation system.

To incorporate a history of states and measurements for estimating the current state can be done through factor graphs. For example, [20] uses factor graphs to combine motion priors with kinematics based on historical state and measurement data. While contact state estimation can be useful, it's not always required as shown in [144], which uses factor graphs with inertial, leg, and visual odometry on soft and slippery surfaces without requiring force/torque sensors. However, factor graphs come with certain limitations. They necessitate an accurate dynamic model and encounter scalability challenges when dealing with significantly expanded input state spaces, which can result in substantial computational overhead. We address these issues using a GRU, which allows us to employ a greater number of state variables while also considering time-series inputs without significant computational effort. Additionally, instead of using the entire depth image as input, we use a ViT to down-scale the image into a smaller latent space, making the overall GRU faster to train.

Perhaps the work most similar to ours can be found in [111], which learns the Kalman gain parameters through a GRU for state estimation. However, there are several distinct differences besides the methodology itself. For one, we explicitly learn not only the state estimate but also the potential error associated with the networks' prediction at each time step. This serves as an indicator for situations involving high prediction errors where we can still utilize the Kalman filter's output, as it operates independently of any learning

components. Lastly, we incorporate vision as part of the error prediction, and apply our methods on complex hardware online. To our knowledge, we are the first to develop an end-to-end nonlinear state estimation system for legged robots, which directly employs the output of a machine learning (ML) model while enhanced by the integration of the Kalman filter and depth encoder into its input space. We outperform a state-of-the-art VIO SLAM solution with our approach.

CHAPTER 3

Energy-Efficient Locomotion

3.1 Introduction

Legged robots excel in uneven terrains but often face energy inefficiency compared to wheeled robots due to energy loss in foot-ground collisions. Solutions include hardware improvements, such as Ranger [11] using passive dynamics for low Cost of Transport (CoT) and ANYmal [64] using mechanical springs to save power. Alternatively, software-based motion planning, like our proposed geometry-based planner, can enhance efficiency. Our approach uses ellipses under each hip joint to determine foot placement, optimizing shape, swing time, and step height offline for different speeds to minimize CoT and maintain stability.

3.2 Footstep Planner

Our planner uses the robot's center of mass and foot positions, with ellipses under each hip joint as placement sets. Defined by major/minor axes and hip positions, we calculate the difference between current foot positions and hip positions:

$$p_{ell}^{dx} = p_{cur}^{x} - p_{hip}^{x} \tag{3.1a}$$

$$p_{ell}^{dy} = p_{cur}^{y} - p_{hip}^{y} \tag{3.1b}$$

Using these, we check if feet are outside the ellipse:

$$\frac{(p_{ell}^{dx})^2}{(r_{ell}^{x})^2} + \frac{(p_{ell}^{dy})^2}{(r_{ell}^{y})^2} > 1 \tag{3.2}$$

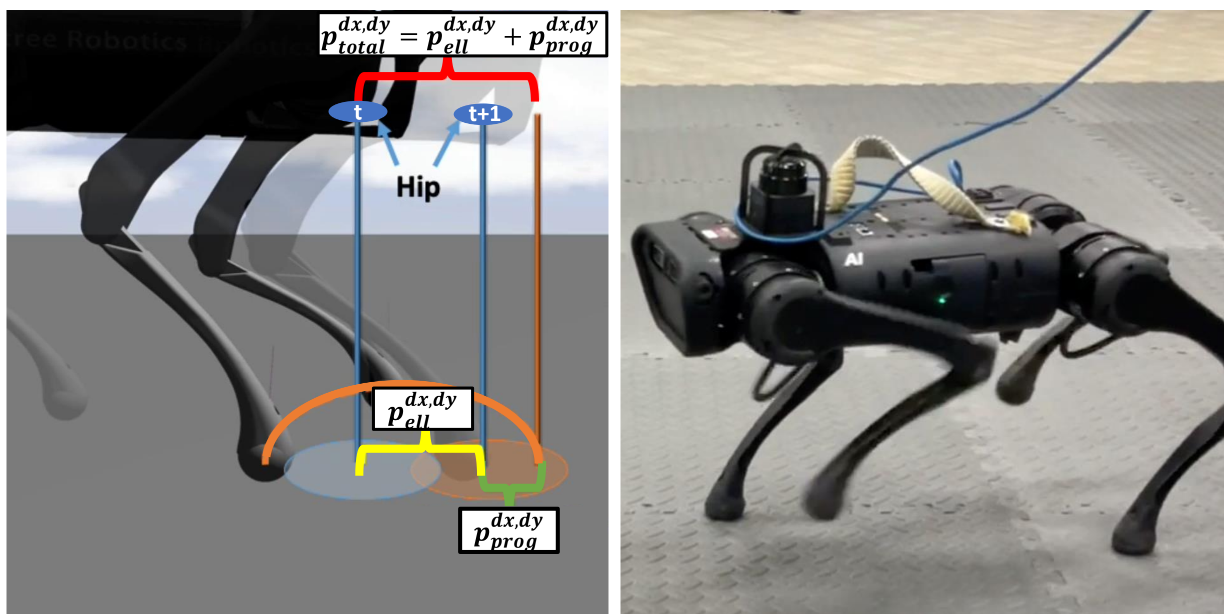


Figure 3.1: **Hardware validation** with the Unitree A1. The left shows the concept of our ellipse-based planner (see Sec. 3.2).

If true, we plan the next footstep and trunk trajectory using commanded velocity and Euler discretization. Algorithm 1 outlines this process.

When all feet are within their ellipses, the trajectory remains unchanged. If the user's commanded velocity exceeds the ellipse boundary, the planner triggers a swing trajectory, calculated to minimize energy use and maintain stability using a sinusoidal implementation:

$$\bar{a}_z = p_{step}\frac{1}{2}\left(\frac{2\pi}{\Delta T}\right)^2 \tag{3.3a}$$

$$a_{z,t} = \bar{a}_z \cos\left(\frac{2\pi}{\Delta T}t\right) \tag{3.3b}$$

$$v_{z,t} = \bar{a}_z \frac{\Delta T}{2\pi} \sin\left(\frac{2\pi}{\Delta T}t\right) \tag{3.3c}$$

$$p_{z,t} = \bar{a}_z \left(\frac{\Delta T}{2\pi}\right)^2 \left(1 - \cos\left(\frac{2\pi}{\Delta T}t\right)\right) \tag{3.3d}$$

This implementation avoids unnecessary stepping, conserving energy.

Algorithm 1: Footstep Planner

```
1  while Robot in Operation do
2      X_cur, p_cur ← StateEstimation(f_cur, q_cur, q̇_cur, IMU)
3      V_des ← Joystick
4      p_ell^dx = p_cur^x − p_hip^x
5      p_ell^dy = p_cur^y − p_hip^y
6      if checkEllipse(p_ell^dx, p_ell^dy, r_ell^x, r_ell^y) then
7          p_prog^dx = V_{x,des}^B ΔT_sw + sqrt(X^z/g)(V_{x,cur}^B − V_{x,des}^B)
8          p_prog^dy = V_{y,des}^B ΔT_sw + sqrt(X^z/g)(V_{y,cur}^B − V_{y,des}^B)
9          p_ref, ṗ_ref, p̈_ref, C_ref ← Sinusoidal(dt, ΔT_sw, gait, p_prog^{dx,dy}, p_ell^{dx,dy}, h_step)
10         X_ref ← interp(X_cur, V_des)
11     end
12     else if Swing Phase Complete then
13         X_next, p_next, ṗ_next, p̈_next, C_next ← X_cur, p_cur, 0^12, 0^12, 1^4
14     end
15     else
16         X_next, p_next, ṗ_next, p̈_next, C_next ← X_ref[it], p_ref[it], ṗ_ref[it], p̈_ref[it], C_ref[it]
17     end
18 end
```

3.3 Parameter Design Framework

To optimize energy efficiency, we use the Cost of Transport (CoT) metric, calculated with joint velocity, torque, swing time, and distance:

$$\text{CoT} = \frac{\sum_{t=k-N}^{k} \max\{\dot{\mathbf{q}}_t^\top \boldsymbol{\tau}_t, 0\} dt}{\Delta D} \tag{3.4}$$

We aim to minimize CoT while maximizing force manipulability using ellipsoids. Parameters studied include swing time, body velocity, swing height, robot height, and ellipse size.

Table 3.1: Parameters for Parameter Study

Name	Description	min	max	units
V^x_{des}	Desired Body Velocity	0.05	1.50	$\frac{m}{s}$
ΔT	Swing Time	0.10	0.25	s
h_{step}	Swing Height	0.05	0.15	m
X^z	Robot Height	0.28	0.311	m
r^x_{ell}	Ellipse axis in x direction	0.01	0.15	m

Table 3.1 lists the parameters tested.

3.4 Parameter Study Results

We tested various parameters over 20 seconds of locomotion to minimize CoT and maximize manipulability. Results showed significant CoT savings using optimal parameters compared to a baseline, especially for trot gait (34.5% improvement) and walk gait (13.3 percent improvement), as seen in Fig. 3.2.

3.5 Simulation and Experimental Results

3.5.1 Controller Architecture/Implementation

The control architecture (Fig. 3.3) runs four parallel processes for efficient computation. Velocity commands are provided via joystick, and state estimation uses a Kalman filter.

3.5.2 Gazebo Simulation Results

Gazebo simulations demonstrated smooth transitions from walk to trot gait (Fig. 3.4), optimizing for CoT and maintaining stability. Walk gait required fewer steps and less energy

compared to trot gait.

3.5.3 Hardware Validation

Hardware experiments (Fig. 3.5) validated the planner's effectiveness in real-world conditions, showing energy savings and robustness against disturbances. The walk gait used 81% of the energy of the trot gait.

3.6 Conclusions

We developed an energy-efficient motion planner for legged robots, showing significant CoT improvements and robust performance in both simulations and hardware experiments. Our planner is simple, computationally efficient, and effective in saving energy by only stepping when necessary.

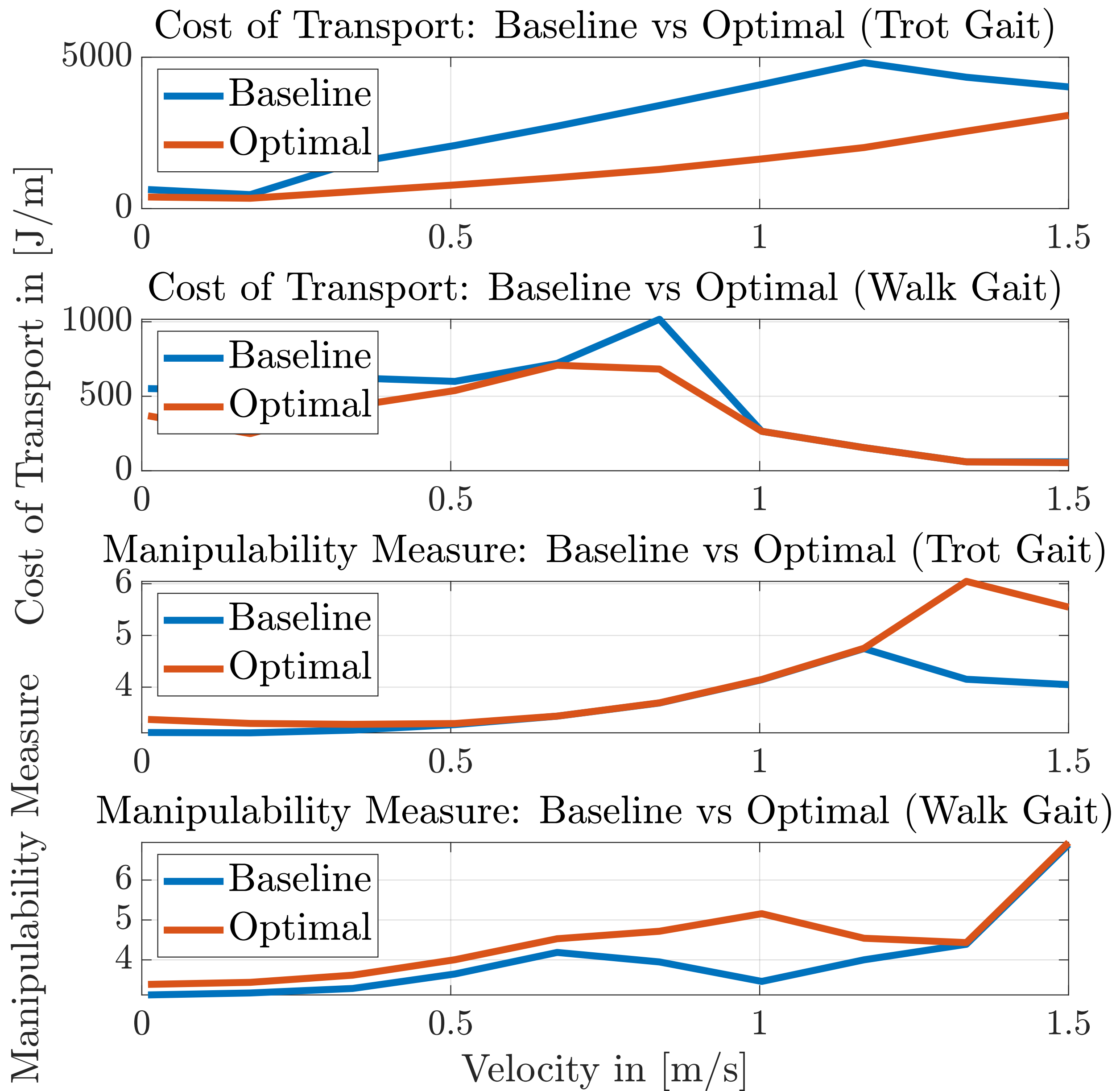


Figure 3.2: **CoT and manipulability measure** for the baseline and optimized parameter selection for various velocities.

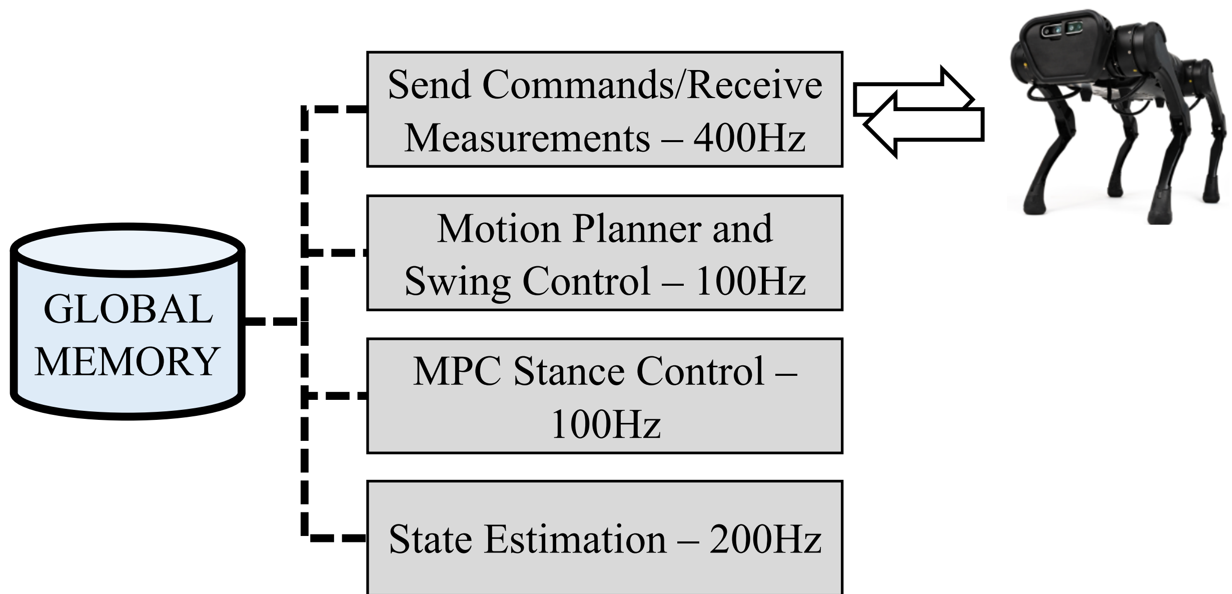


Figure 3.3: **Planner and Control Architecture**

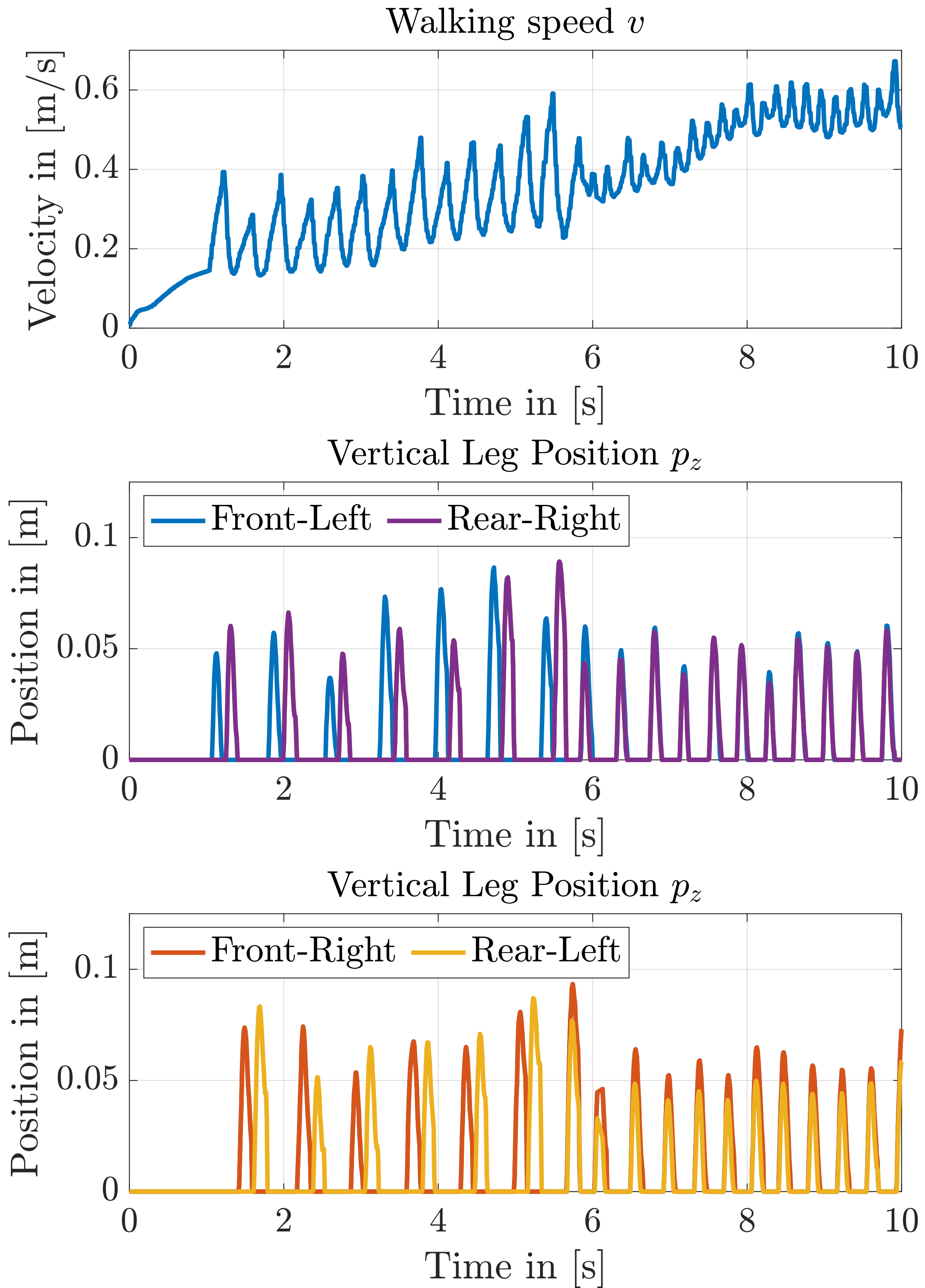
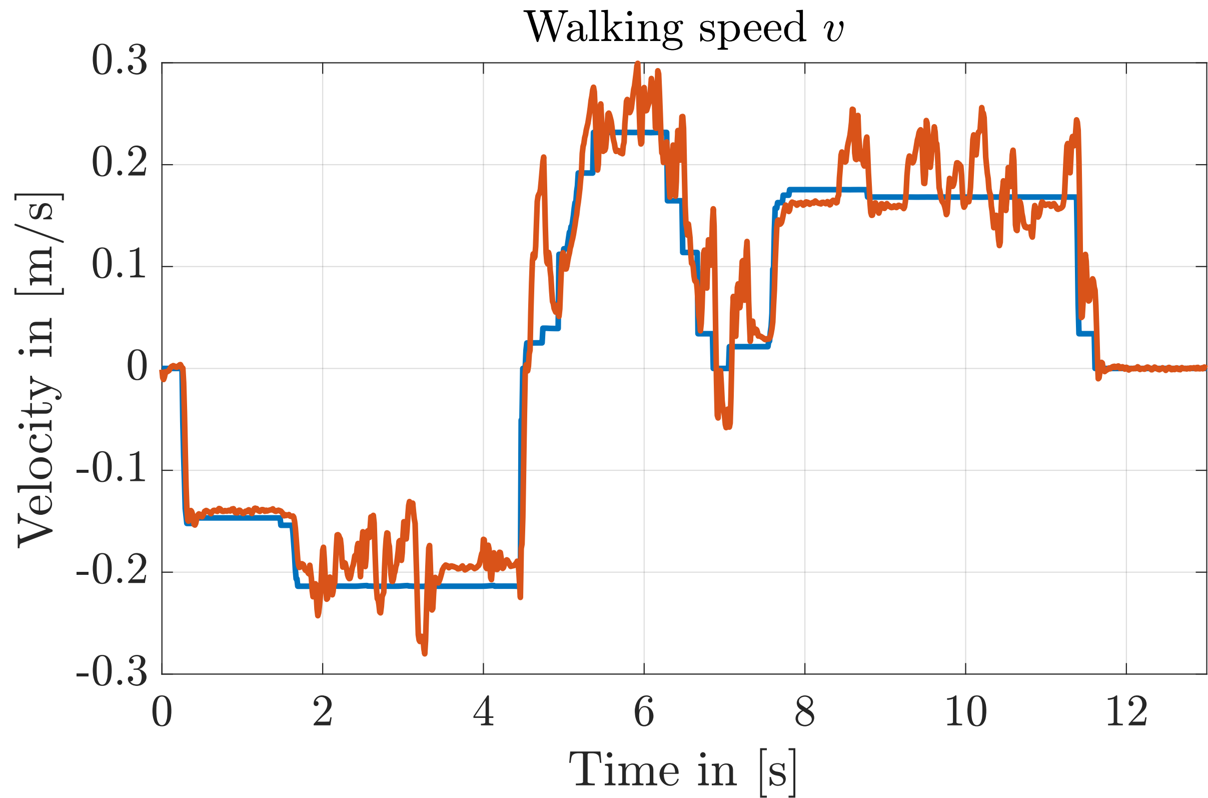


Figure 3.4: **Gait transition results**. Transition from walk to trot gait shown with vertical leg positions.



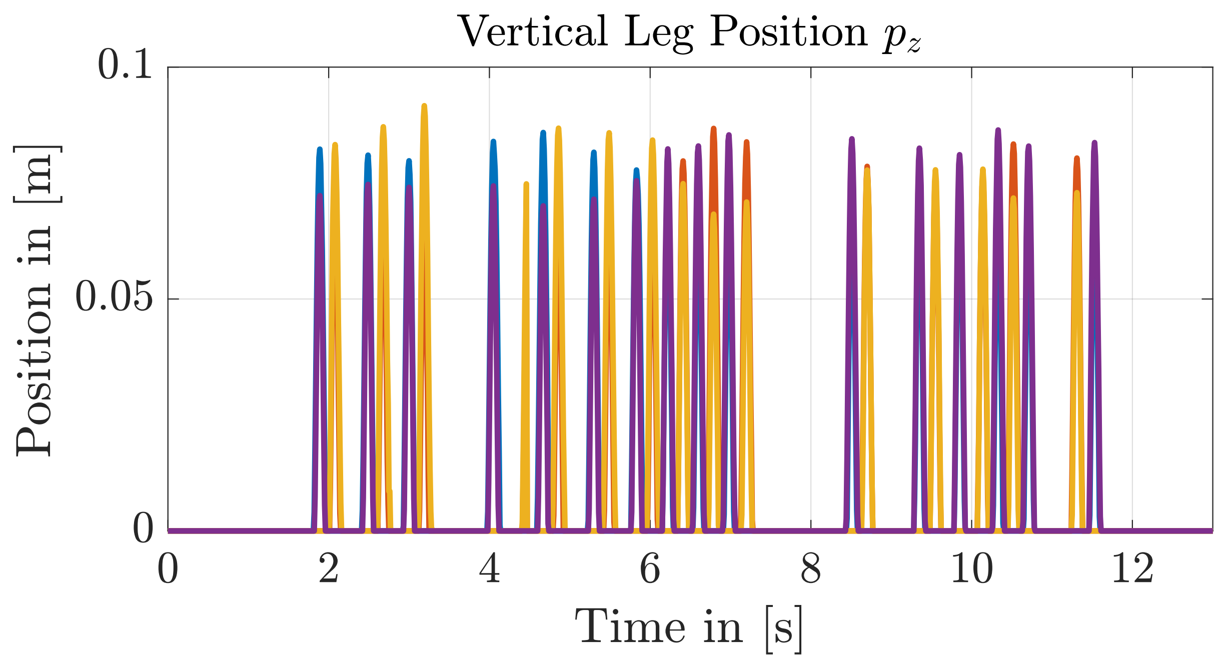


Figure 3.5: **Teleoperation results** demonstrating disturbance robustness. The top graph shows commanded and actual velocities, with vertical leg positions below.

CHAPTER 4

Risk-Averse MPC

4.1 Introduction

For robots to truly become a viable option for unmanned tasks such as search and rescue operations and unknown environmental exploration, autonomous and safe trajectory planning must be considered as a fundamental design goal during algorithmic design and systems integration. Necessary requirements for robots to autonomously perform such complex tasks include, but are not limited to, online low-level feedback controls, localization, vision, motion planning, high-level reasoning, and reasoning under uncertainty. Currently, all individual components are well-developed, but integrating multiple pieces together into a single system, especially for environments that are not well-known, has proven to be a daunting challenge because of issues related to robustness [3]. For example, simultaneous planning, localization, and mapping (SPLAM, or 'Active SLAM') is an active area of research that attempts to satisfy some of these requirements. The main challenges to Active SLAM consist of planning under uncertainty in an acceptable amount of time, bridging the gap between sensor data ('semantic mapping'), and ensuring that it is robust enough for a complex platform. Because of these challenges, there are many works addressing a subset of Active SLAM, namely simultaneous localization and planning or works addressing SLAM, such as [98, 68, 41]. Other Active SLAM works also use very simple research platforms [70] or were only tested in simulation [81].

To resolve the above issues, we propose a multifaceted approach that uses model pre-

dictive control (MPC), SLAM[1], and recurrent neural network (RNN) algorithms to address the problem of Active SLAM and account for uncertainties in both current and future robot positions. Our architecture is based on MPC because MPC operates online, continually satisfies the dynamic state of the robot over a prediction horizon N, and naturally offsets estimation errors [146]. The MPC is augmented to be 'risk-averse' by considering uncertainty in position from timestep k to $k+N$. This uncertainty is inferred by an RNN, which has been demonstrated to handle time series data, account for temporal factors that directly affect predictions, have shown promise in modeling complex interactions between agents and their environment [42, 150] and previously applied to MPC but for industrial processes [78]. Our RNN model is trained on the positional covariance estimations of a visual-inertial odometry (VIO) system taking readings from an inertial measurement unit (IMU) and camera data as input. By considering the current and future positional uncertainties in the MPC optimization problem, our method can solve for more optimal control actions at each timestep. To facilitate object avoidance, we additionally incorporate an object detection pipeline that uses a deep convolutional neural network (CNN) to recognize obstacles and a feature detector with RGB and depth images to estimate the distance and size of nearby obstacles. We show that by using a trained RNN model to infer positional uncertainties at future timesteps, a robot can demonstrate more evasive behavior to better guarantee collision avoidance without becoming too conservative. Our linearized path planning framework is applied and tested on a complex quadruped robot, which demonstrates our algorithm's robustness and efficiency in computation, showing the feasibility of extending our work to a wide range of robotic platforms.

[1] In this paper, SLAM includes visual-inertial odometry with sparse mapping in addition to algorithms that produce denser maps.

Summary of Our Contributions

(1) We evaluate the feasibility of an online end-to-end path planner that unifies MPC, SLAM, RNN, and an object detector using CNNs to generate paths for unknown and uncertain environments using a non-linear programming solver.

(2) We verify that our quadrupedal robot, ALPHRED [60], avoids collisions and computes a shorter trajectory while maintaining safety using our method as compared to a more conservative and naive planner.

(3) We propose a novel use of RNNs to estimate positional uncertainties at all future timesteps of the MPC's prediction horizon.

(4) We integrate all components into a high-fidelity simulation using the quadruped dynamics of ALPHRED (Figure 4.4). Additionally, we test all components individually either online or offline using hardware (Figure 4.7).

In the following sections, we will explicitly refer to the simulation or hardware data. The main difference between the model of ALPHRED in simulation versus hardware is that in simulation the model is equipped with an idealistic RGB + dense depth Microsoft Kinect camera, while the actual hardware is equipped with Intel's Realsense D435i. The idealistic camera publishes both RGB and dense depth images at arbitrarily fast speeds while the RealSense publishes RGB images at 30Hz and dense depth images at only 2Hz.

4.2 Methods

In this section, we provide an overview of our architecture and how our risk-averse MPC propagates uncertainty through its finite time horizon trajectory. In Section 4.2.1, we provide a high-level overview of our path planning algorithm. In Section 4.2.2, we describe our MPC's mathematical framework for planning and tracking. In Section 4.2.3, we describe

the constraints used in our cost functions. In Section 4.2.4, we describe our object detection system using CNNs and keypoint detection on RGB and depth images, and finally in Section 5.2.3, we detail our RNN training and inference procedures (utilizing our SLAM algorithm) for predicting future positional uncertainties used to create collision boundaries.

4.2.1 Architecture Overview

Our path planner is formulated as an MPC optimization problem using a non-linear programming solver [4]. We divide our MPC framework into a planning phase and a tracking phase, with different cost functions for each. In the planning phase, the goal of our MPC is to create waypoints that move a robot closer to a desired position while detecting and avoiding obstacles through measurement updates. Specifically, the object detection algorithm feeds the MPC with the position and size of surrounding obstacles, while the positional uncertainty of the robot at all future timesteps in the MPC prediction horizon is inferred by RNNs. In the tracking phase, we discretize the generated path into segments of fixed temporal length using a cubic polynomial to create a smooth reference trajectory. MPC is used to track this reference trajectory and outputs our desired planar velocity values (v_d and $\dot{\psi}_d$). These velocity values are used by our motion tracking controller to generate stable footstep trajectories. Note that dividing our MPC formulation into two phases facilitates lower computation time, and allows for separate control on waypoint generation and creation of custom reference trajectories if desired (see Algorithm 2, Fig. 5.2, or our accompanied video[2] for a general overview of our path planning architecture).

[2] https://www.youtube.com/watch?v=faurQ1LpNVI

Algorithm 2: Risk-Averse MPC

1 Initialize state X, control U, dt_{plan}, dt_{track}, horizon N, robot collision boundary $r_{\Sigma_{k:k+N}}$, and timestep k

```
// Planning Phase (waypoints to goal)
```

2 **while** $\|X_{curr} - X_{goal}\|_2 > 0$ **do**

3 $X, U_{sols} \leftarrow \text{MPC}(X_{curr}, X_{goal}, r_{\Sigma_{k:k+N}})$

4 $X_{ref}, U_{ref} \leftarrow \text{CubicSpline}(X, U_{sols})$

5 $[pixel_x, pixel_y]_{1:f} \leftarrow \text{FeatureExtractor(RGB)}$

6 $bboxes \leftarrow \text{CNN(RGB)}$

7 $[x, y, z]_{1:l} \leftarrow \text{ObjectDetector(RGB-D}, [pixel_x, pixel_y]_{1:f}, bboxes)$

8 $X_{curr} \leftarrow \text{RobotEstimator}(U, \text{IMU}, \text{joint encoders})$

9 $r_{\Sigma_{k:k+N}} \leftarrow \text{RNN}(X_{sols}, [x, y, z]_{1:l})$

// Tracking Phase (follow X_{ref} and U_{ref})

10 **while** $dt \leq dt_{plan}$ **do**

11 $U \leftarrow \text{MPC}(X_{curr}, X_{ref(dt)}, U_{ref(dt)})$

12 $X_{curr} \leftarrow \text{MotionTrackingController}(U)$

13 $dt \mathrel{+}= dt_{track}$

14 **end**

15 $dt = 0$

16 **end**

4.2.2 General MPC Formulation

4.2.2.1 Planning Phase

MPC in the planning phase has the following time-invariant linear discretized model:

$$f(X_k, U_k) = X_{k+1} = AX_k + BU_k + w_k \tag{4.1}$$

where $X = \left[\begin{array}{cc} x, & y \end{array}\right]^\top$ represents our state variables (planar waypoint position), and $U = \left[\begin{array}{c} v_x, v_y \end{array}\right]^\top$ represents our control variables (planar velocity). We also initialized our state and control variables to zero before run-time.

Because we have a motion tracking controller to incorporate robot dynamics (see Section 4.3.1), our A and B matrices can assume a simple point mass:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, B = \begin{bmatrix} dt_{plan} & 0 \\ 0 & dt_{plan} \end{bmatrix}$$

where dt_{plan} is the time between taking proprioceptive and exteroceptive sensor measurements (e.g., RGB-D images and odometer readings), and w_k represents a non-unit variance random Gaussian noise ($w_k \sim \mathcal{N}(0, \sigma^2)$, where σ represents the standard deviation of planar velocity).

The goal of our cost function in the planning phase (equation (4.2)) is to find the optimal control value that minimizes the distance from the current and predicted states ($X_{k=0 \to N}$) to the goal state (X_{goal}) – where $X_{k=0}$ is given by the results of localization. Note, that we use $\hat{U}_k$ instead of U_k in our cost function to represent the inclusion of a slack decision variable, ϵ (the slack variable has no role in our discretized model equation, but does affect the cost function through R - see 4.2.3), so that $\hat{U} = \left[\begin{array}{c} v_x, v_y, \epsilon \end{array}\right]^\top$.

$$\min_{U_{k:k+N}} \sum_{k=0}^{N} \left(X_{k+1} - X^{goal}\right)^\top Q \left(X_{k+1} - X^{goal}\right) + \hat{U}_k^\top R \hat{U}_k \tag{4.2}$$

s.t. 1, 2, 3, 5 (see Table 4.1)

4.2.2.2 Tracking Phase

MPC in the tracking phase has the following time-invariant linear discretized model:

$$f\left(X_k, U_k\right) = X_{k+1} = AX_k + BU_k \tag{4.3}$$

where $X = \begin{bmatrix} x, & y, & \psi \end{bmatrix}^\top$ represents our state variables (desired planar position and yaw or heading angle), and $U = \begin{bmatrix} v_x, v_y, \dot{\psi} \end{bmatrix}^\top$ represents our control variables (desired planar velocity and yaw rate). Matrices A and B are the same as shown in (4.1), except for an additional row/column for yaw and yaw rate.

The goal of our cost function in the tracking phase (equation (4.4)) is to output desired planar velocity and yaw rate (v_d and $\dot{\psi}_d$) values that follow a reference trajectory.

$$\begin{aligned} \min_{U_{k:k+N}} \sum_{k=0}^{N} & \left(X_k - X_k^{ref}\right)^\top Q \left(X_k - X_k^{ref}\right) \\ & + \left(U_k - U_k^{ref}\right)^\top R \left(U_k - U_k^{ref}\right) \end{aligned} \tag{4.4}$$

s.t. 1, 2, 3, 4 (see Table 4.1)

X^{ref} and U^{ref} are obtained by cubic interpolation (equation (4.5)) with end points specified by the MPC planning phase from $X_k ... X_{k+2}$ and $U_k ... U_{k+2}$ (the reason we discretize to $k+2$ instead of $k+1$ is to ensure there are enough reference points for MPC to 'look-ahead').

$$\begin{aligned} X^{ref}(t), U^{ref}(t) &= a_0 + a_1 t + a_2 t^2 + a_3 t^3, \\ a_i &= f\left(dt_{track}, X_k, U_k, X_{k+2}, U_{k+2}\right) \end{aligned} \tag{4.5}$$

4.2.3 MPC Constraints

4.2.3.1 Constraints 1 - 4

Constraint 1 represents multiple shooting constraints which facilitate solving non-linear programs [58]. The limits on state variables (i.e., map constraints), and control variables (limits on velocity) are represented by Constraint 2 (note that the slack decision variable should be set as $0 \leq \epsilon$). If there is apparent jerk during path planning, it may be necessary to

include Constraint 3, where α^{limit} represents the limit on acceleration (a_x, a_y, and $\ddot{\psi}$) and U represents velocity (v_x, v_y and $\dot{\psi}$). Orienting the robot along the planned trajectory can be achieved using Constraint 4, and setting the limit on v_y to be much smaller than the limit on v_x (which points directly along the path) in the body frame. Because MPC outputs velocities in the inertial reference frame (irf), a rotation matrix is required to transform these velocities into the correct frame of reference.

4.2.3.2 Constraint V - Collision Boundary with Slack Variable

Our obstacle avoidance constraints are given by Constraint 5, which ensure that the collision boundary of the robot does not collide with detected obstacles (note, that because these constraints are updated at every timestep dt_{plan}, moving obstacles can also be considered). x_{o_i} and y_{o_i} represent the x and y center positions of all obstacles detected by the robot ($i \rightarrow M$: where M is the number of obstacles currently in range). x_k and y_k represent the x and y positions of the robot from timestep k to timestep $k + N$ (future positions can be received from the MPC solution). r_{Σ_k} represents the radius of the collision boundary of the robot, and r_{o_i} represents the radius of the collision boundary of the obstacle. The collision boundary radius of the robot is calculated by using the major axis of the covariance uncertainty ellipse (Σ) estimated from RNN and is added to the radius or size of the robot itself (thus, we assume a more conservative collision boundary, which, combined with the slack variable—see below—provides some tolerance to ensure the planner does not fail while at the same time lowering the probability of collision). Note, from timestep k to timestep $k + N$, $\Sigma_{k \rightarrow N}$ is predicted by our RNN (see Section 5.2.3).

Without a slack variable and because of sensor measurement noise, the measured state of the robot may suddenly find itself very near or slightly inside the collision boundary of the obstacle and cause the solver to fail. To accommodate for this issue, Constraint 5 includes a slack variable to allow for some degree of constraint violation in our optimization problem. In other words, we effectively separate the covariance into a constraint and slack variable,

Table 4.1: Model Predictive Control Constraints

No.	Constraint
I	$X_{k+1} - f(X_k, U_k) = 0$
II	$X^{limit} \geq \lvert X_k \rvert,\ U^{limit} \geq \lvert U_k \rvert$
III	$\alpha^{limit} \geq \lvert [\mathrm{U_{k+1}} - \mathrm{U_k}] \rvert / \mathrm{dt}$
IV	$\begin{bmatrix} v_x^{limit} \\ v_y^{limit} \end{bmatrix}_{body} \geq \left\lvert \begin{bmatrix} \cos\theta_k & \sin\theta_k \\ -\sin\theta_k & \cos\theta_k \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix}_{irf} \right\rvert$
V	$-\sqrt{(x_k - x_{o_i})^2 + (y_k - y_{o_i})^2} + r_{\Sigma_k} + r_{o_i} - \epsilon \leq 0$

where we tune the confidence attributed to the posterior estimate and break it into a nominal estimate, and a controllable slack parameter. Specifically, slack can be tuned through the R matrix, where a high cost for ϵ will ensure that the majority of solutions will not violate Constraint 5, while lower values allow for greater violation (the cost on the slack variable is largely dependent on user-experience during implementation).

4.2.4 Obstacle Detection

4.2.4.1 Convolutional Neural Networks

To support the avoidance of incoming obstacles as our robot traverses the environment, we use a custom-trained CNN model for real-time object detection. More specifically, using

Redmon et al.'s YOLOv3 [110] fast CNN architecture because of its maturity (although other methods could be used, such as [44]), we trained two custom models. One localized brown boxes within an RGB camera frame using 1,500 hand-labeled images and the other localized black $1m$ x $1m$ boxes in a mostly empty Gazebo environment using 300 images. Weights were initialized using YOLOv3's default weights and trained for 5,200 epochs using stochastic gradient descent with a batch size of 64, momentum of 0.9, and learning rate of 0.001 for both models. We validated our model on labeled data withheld from the training data and we verified empirically that our object detector could successfully draw tight bounding boxes around our brown boxes (Fig. 4.2).

4.2.4.2 From Bounding Boxes to 3D Obstacles

For our end-to-end Gazebo simulation, we implemented simple classical feature detection over the simulated RGB and dense depth images to transform the bounding boxes from the object detector into useful 3D obstacles for the motion planner. The scheme described below assumes that features are cubes and that ALPHRED is directly facing all existing boxes. It is executed only once, at the beginning of the simulation. Note that instead of fully addressing the semantic mapping problem, we use simple placeholder computer vision components designed for our specific test scenarios; for now, we only utilize SLAM as training data for the RNN.

First, ORB features [114] are extracted. Let x_p and y_p be the pixel coordinates of a single feature, and Z_c be its depth. Then, let $g_{sb} = (R_{sb}, T_{sb})$ be the body-to-spatial transformation, $g_{bc} = (R_{bc}, T_{bc})$ be the camera-to-body transformation, and K be the intrinsics matrix. Then, the position of the feature in the spatial frame, X_s (a 3×1 vector) can be calculated as:

$$
\begin{aligned}
\begin{bmatrix} x_c \\ y_c \\ 1 \end{bmatrix} &= K^{-1} \begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} \\
X_c &= Z_c \begin{bmatrix} x_c \\ y_c \\ 1 \end{bmatrix} \\
X_b &= R_{bc} X_c + T_{bc} \\
X_s &= R_{sb} X_b + T_{sb}
\end{aligned}
\tag{4.6}
$$

Next, for each bounding box captured by the CNN object detection process, we determine which features are in each bounding box. The size of the box is the maximum distance (in meters) between any two points. Half of that size then becomes the 'radius' of the obstacle's collision boundary (the MPC assumes that the collision boundary are circles).

We note that our classical feature detection approach is computationally efficient but also simple (i.e., not as robust). For example, most features exist near corners, where rounding errors could lead to a very different depth value. For the purpose of our end-to-end Gazebo simulation, we discarded any obstacle detections that were more than 5 meters away.

4.2.5 Recurrent Neural Networks for Learning Uncertainties

The RNN, shown in Figure 4.3, uses a combination of feedforward layers and simple RNN layers. The hidden layers all use ReLU activations. The network's 18 inputs are the robot's x, y, and z positions. The next 15 inputs consist of the x, y, and z positions of the five closest tracked features at any given state. The four output layer neurons correspond to the four values of the robot's 2×2 x-y covariance matrix, which is then used in Constraint V of the motion planning MPC. Unlike the hidden layers, the output layer uses a linear activation function because the outputs themselves are not restricted. Note that even though our MPC

plans in only two dimensions, the inputs to the neural network are three-dimensional because the state estimation in our experiment is three-dimensional.

We used the Mean Squared Error (MSE) as the loss function:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\Sigma_i - \hat{\Sigma}_i)^2$$

Here N is the total number of timesteps, Σ_i is the covariance matrix of the planer position computed by a SLAM system at timestep i and $\hat{\Sigma}_i$ is the covariance matrix predicted by the RNN. Conceptually, the covariance matrix is a 2×2 matrix, but the implementation of the RNN treats it as a 4×1 flattened matrix when it makes predictions and propagates error. Lastly, we note that covariance matrices are positive semidefinite by definition. However, the above training procedure does not constrain the output of the RNN to positive semidefinite; the outputs were indeed arbitrary 2×2 matrices. To account for this, we zeroed out off-diagonal elements and negated any negative diagonal elements.

Training data (robot position, position of tracked features, and covariance matrices) for the RNN was collected from running XIVO,[3] a simplified and modernized implementation of the SLAM system described in [68], on time-synchronized RGB and IMU data collected from an Intel RealSense D435i mounted onto ALPHRED's head. We collected four ~40-second training sequences in total (using 100 epochs for training on the four sequences). The right side of Figure 4.2 displays tracked features and localization estimates from the collected data.

One key assumption that XIVO makes is that disturbances to the angular velocity and acceleration measurements (bias + noise) are a random walk (i.e. white, zero-mean, and Gaussian). This is not true for a walking robot, where each step produces a large periodic disturbance. Thus, XIVO's generic motion model is best suited to a flying robot. However, to adapt XIVO for our quadruped, we limited the acceleration and angular rate measurements to 'realistic' values and then 'de-tuned' the filter by setting large bounds on expected IMU measurement noise. This hampered accuracy, but ultimately enabled convergence.

[3] Code available: https://github.com/ucla-vision/xivo

4.3 Experimental Results

In this section, we provide an overview of our robot model and its motion tracking controller, describe the training and testing results of the CNN and RNN neural networks, and then summarize our end-to-end results using our Gazebo simulation environment.

4.3.1 Robot Model and Motion Tracking Controller

The robot used in this study is ALPHRED from Hooks *et al.* [60], a full-sized quadrupedal robot that has unique kinematic configurations which enable several dynamic modes of operation as shown in Fig. 4.7 and Table 4.2. Our path planner is tested on a highly accurate simulation of ALPHRED using Gazebo software [76] (Fig. 4.4). The robot is modeled as several interconnected rigid-bodies in PyBullet so that the state includes not only joint angular velocities, but sensor and actuator noise due to motor temperature. The camera model used is a standard perspective projection with the same intrinsics as the Intel RealSense camera used to collect RNN training data, but without distortions. ALPHRED uses an Extended Kalman Filter (EKF) that fuses kinematic encoder data with on-board IMU measurements to provide full state estimation [14]. A Raibert-style controller [106] is used to track desired trajectories, where the input to the controller is desired planar velocities (v_d) and a desired yaw rate ($\dot{\psi}_d$) in the body frame. The controller operates by planning footsteps using powerful heuristics based on velocity feedback and corrects velocity and orientation errors by adjusting the length of the limbs in support. Further details of the ALPHRED platform and its low-level motion tracking controller can be seen in [60].

4.3.2 Analysis of Learning Components

Training loss for both the CNN and RNN are shown in Figure (Fig. 4.5). CNN and RNN networks were trained for 5,300 and 100 epochs, respectively, but only a limited range was plotted for visualization. To avoid overfitting, we used cross-validation and ensured that the

validation loss was close to the training loss during the training process for both networks. Additionally, we observed that as ALPHRED tracked more features (i.e., the corners of an obstacle), the RNN's covariance estimates decreased. Conversely, as tracked features went out of view, estimates would increase. This is expected from the behavior of a visual-inertial odometry algorithm.

4.3.3 Gazebo Simulation

To test our proposed method, we used a custom Gazebo environment loaded with a high-fidelity model of our quadrupedal robot equipped with a Microsoft Kinect sensor. For localization, we used the motion tracking controller as described in Section 4.3.1. Our 3D environment consisted of a $1m^3$ box obstacle with the objective to command ALPHRED to move from its initial position at [0,0] to the goal position at [8,0]. We compared our method against a baseline approach, in which only an MPC was used for trajectory planning (with the obstacle explicitly hardcoded), and a naive approach for safer traversal, in which the robot's radius was artificially inflated to twice the original size (from 0.7m to 1.4m).

In the illustrative example shown in Fig. 4.6, we observed that when using a classic MPC controller, which assumes that the robot's state estimation is perfect, the resulting trajectory is too close to the obstacle and ALPHRED crashes (red). On the other hand, when using a conservative MPC controller, in which the assumed value of ALPHRED's radius is twice its actual size, the resulting trajectory over-avoids collisions and ALPHRED moves slowly towards the goal point (blue). However, when using our full risk-aware MPC in this scenario, we observed that ALPHRED not only avoids collision, but executes a tighter trajectory than the conservative approach and requires less time to move to the goal. Note that the simulation was run on a laptop with an Intel Core i7 6700 HQ CPU and a NVIDIA GeForce GTX 970M GPU in real time with $dt_{plan} = 0.1s$ and $dt_{track} = 0.005s$.

4.4 Discussion and Future Work

Collision-free path planning within unknown and unexplored environments requires the daunting integration of several components, such as sensor processing, control algorithms, and uncertainty resolution, into a fast and online end-to-end framework. To this end, we propose an architecture that unifies these modalities which attempts to address the fundamental problem of uncertainty in Active SLAM. By inferring the future positional uncertainty for an MPC using an RNN, we can substitute typical belief space planners with a more computationally efficient approach. Our work can also pave the way towards using RNNs to address problems with temporal structure which are difficult for classic robotic algorithms.

Overall, our architecture addresses Active SLAM by combining MPC, SLAM, RNN, and CNN algorithms. We demonstrate that by inferring future positional uncertainties of the robot using our RNN prediction model, the robot can reach a goal state faster than when assuming a fixed uncertainty while still safely avoiding obstacles. This is significant because modeling uncertainties within a neural network framework, rather than belief space planning (i.e., POMDP), sufficiently shortens the computation time for hardware implementation. Future work will entail improvement of individual components in our architecture and modifying parts for complete hardware compatibility if necessary. For example, we would combine the classical feature detection and ALPHRED's state estimation (described in Section 4.2.4.2) into a VIO algorithm.[4] This would require us to modify generic VIO equations, such as the ones present in XIVO, by explicitly modeling a walking gait, expanding the size of the gait space, and recomputing the Jacobians to incorporate robot dynamics into visual modeling instead of assuming a simple random walk. Then the performance-limiting detuning and signal clipping described in Section 4.2.5 will become unnecessary. Finally, we also aim to replace the CNN + classical feature detection and unprojection with a modern

[4]We prefer a VIO algorithm over other SLAM algorithms because they can directly observe scale and because range sensors (e.g. LiDAR) are more expensive in both cost and computation and typically only produce sparse images. The motions of a walking robot should be sufficiently exciting, such that the VIO problem is observable.

Table 4.2: ALPHRED Configuration

Parameter	Value
Degrees of Freedom	12 (3 per leg)
Weight	17.9 kg
Max Velocity	1.5 m/s
IMU	VectorNav 200
Camera	RealSense D435i

semantic mapping algorithm, such as [44].

Future directions also include: (1) formulating our RNN to infer semantics and feed object-dependent margins to the planner (e.g., the robot can get close to safe objects but not to dangerous ones); (2) exploring additional inputs to the RNN, such as the estimated covariances of features or other states; (3) incorporating a z state/control in the MPC rather than assume planar motion for more complex path planning; (4) comparing our formulations to belief space planners (e.g., stochastic MPC) as well as other state-of-the-art path planners; and (5) constraining the RNN training process such that outputs are guaranteed positive semidefinite. We believe that implementing the above modifications could lead to closing the gap in achieving the futuristic vision for complete autonomous robotic systems.

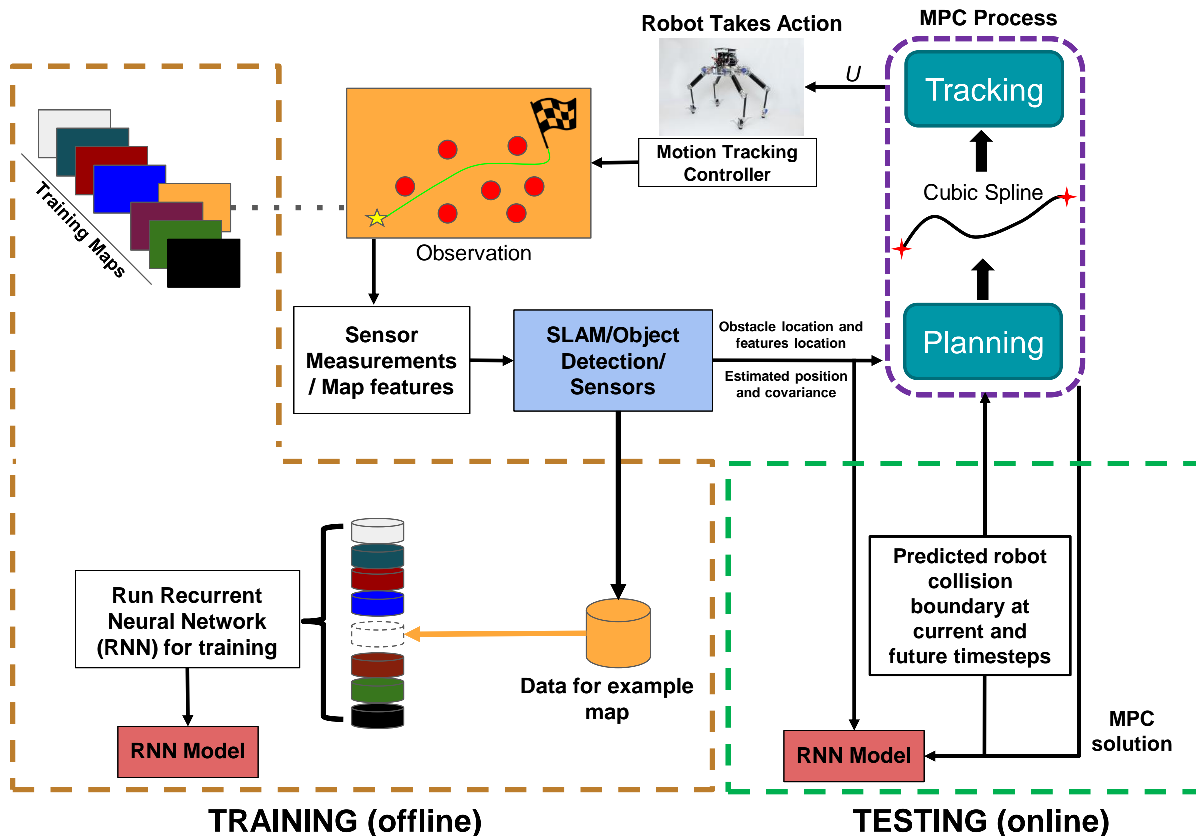
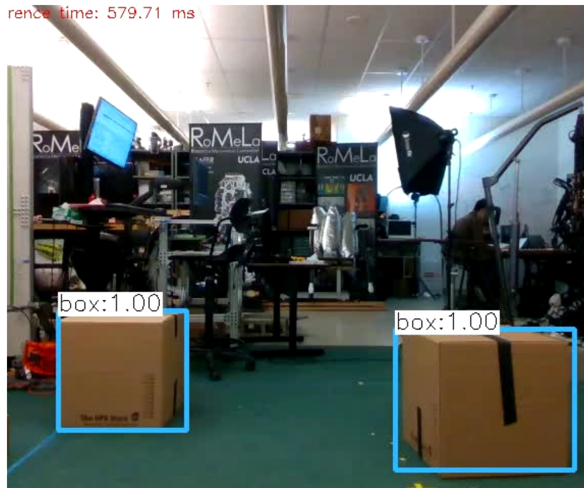


Figure 4.1: **Architecture Overview.** This figure demonstrates the training and testing procedures of our method. In training, we first select different maps, where obstacles in each map are randomly distributed. A simulation where the robot moves from an initial to a goal position is executed on this map. At each timestep an observation is taken (e.g., camera or on-board sensor data). These measurements are used as the input to our SLAM/Object Detection/Sensors system, which estimate the current position and uncertainty in position of the robot, and also location and size of obstacles. MPC accounts for this information and produces outputs entered into our motion tracking controller. For every map at every timestep, the current observations, state position, and positional uncertainty (among other variables outlined in Section 5.2.3) are entered into a large database to produce our RNN model. Lastly, in the testing phase, RNNs can predict the positional uncertainty (which provide our collision boundaries) of the robot at future timesteps of the MPC prediction horizon.

CNN XIVO



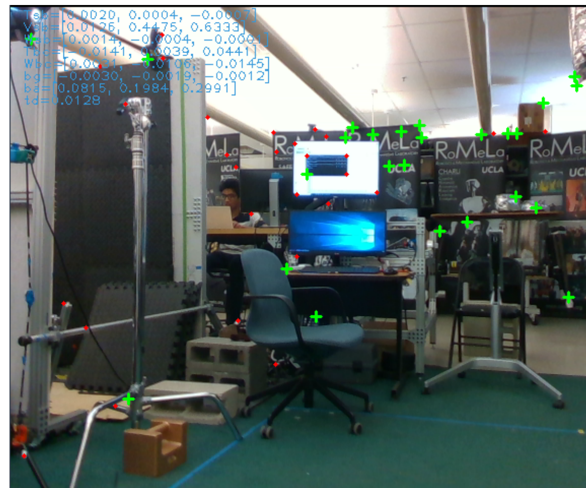


Figure 4.2: **Example Module Outputs.** *Left*: An example output image of our trained object detector using a custom-trained convolutional neural network model. We used the YOLOv3 [110] architecture with default initialized weights for fast training and inference. *Right*: Inlier (green +) and outlier tracks (red *) produced by XIVO on data collected from the Intel Realsense D435i.

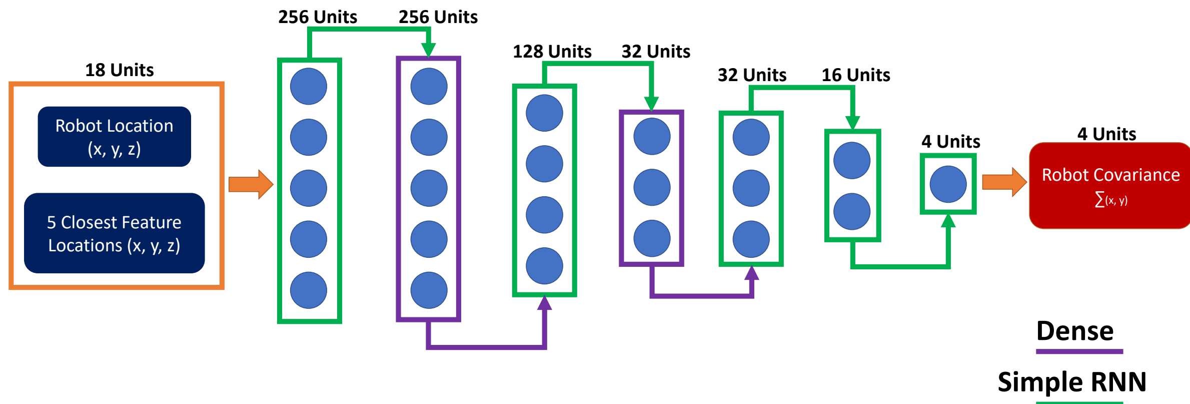


Figure 4.3: **Recurrent Neural Network Architecture.** Our RNN architecture predicts the covariances at robot poses $[x_{t+n}, y_{t+n}]$ at timesteps $t+n$ for $n = 1, ..., N$ (where N is the length of the MPC's prediction horizon). During training, we used inputs collected from the output of XIVO to parameterize the network towards the four output units, as indicated by the first 18 input units and last four units in the figure above. Seven hidden layers were used with ReLU activation functions, with five recurrent layers (green) and two fully connected layers (purple), to learn the temporal structure for covariance propagation.

Figure 4.4: **Gazebo Simulation.** Our high-fidelity simulation accurately models the dynamics of the ALPHRED quadruped robot.

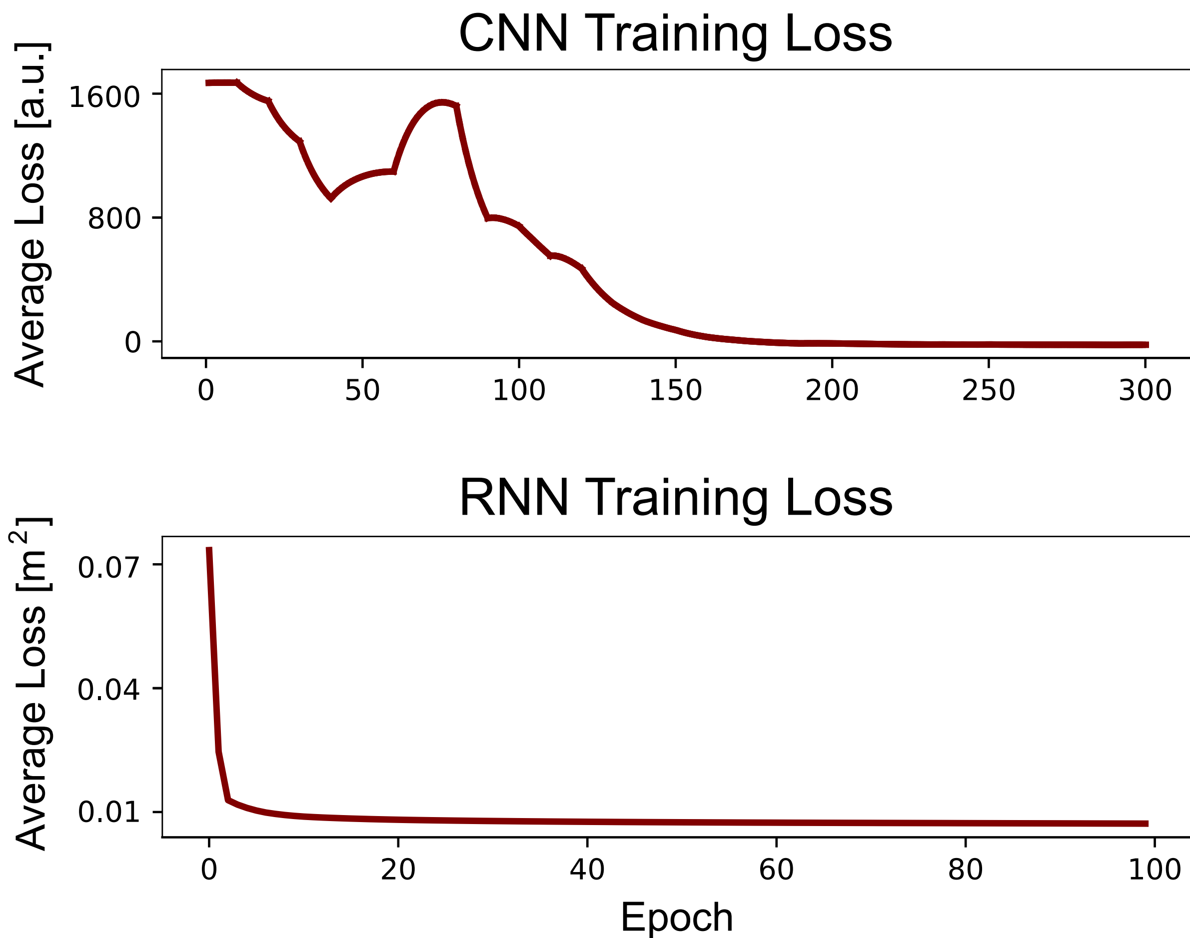


Figure 4.5: **Training Loss.** *Top:* Our CNN model's training loss, used in our object detection pipeline. We trained for 5,200 epochs but only display 300 in the figure above. Note that we verified avoidance of overfitting via a validaton set but did not plot the curve here. *Bottom:* Our RNN model's training loss, used to infer future localization uncertainty for the MPC. As with the CNN, we verified avoidance of overfitting using a validation set.

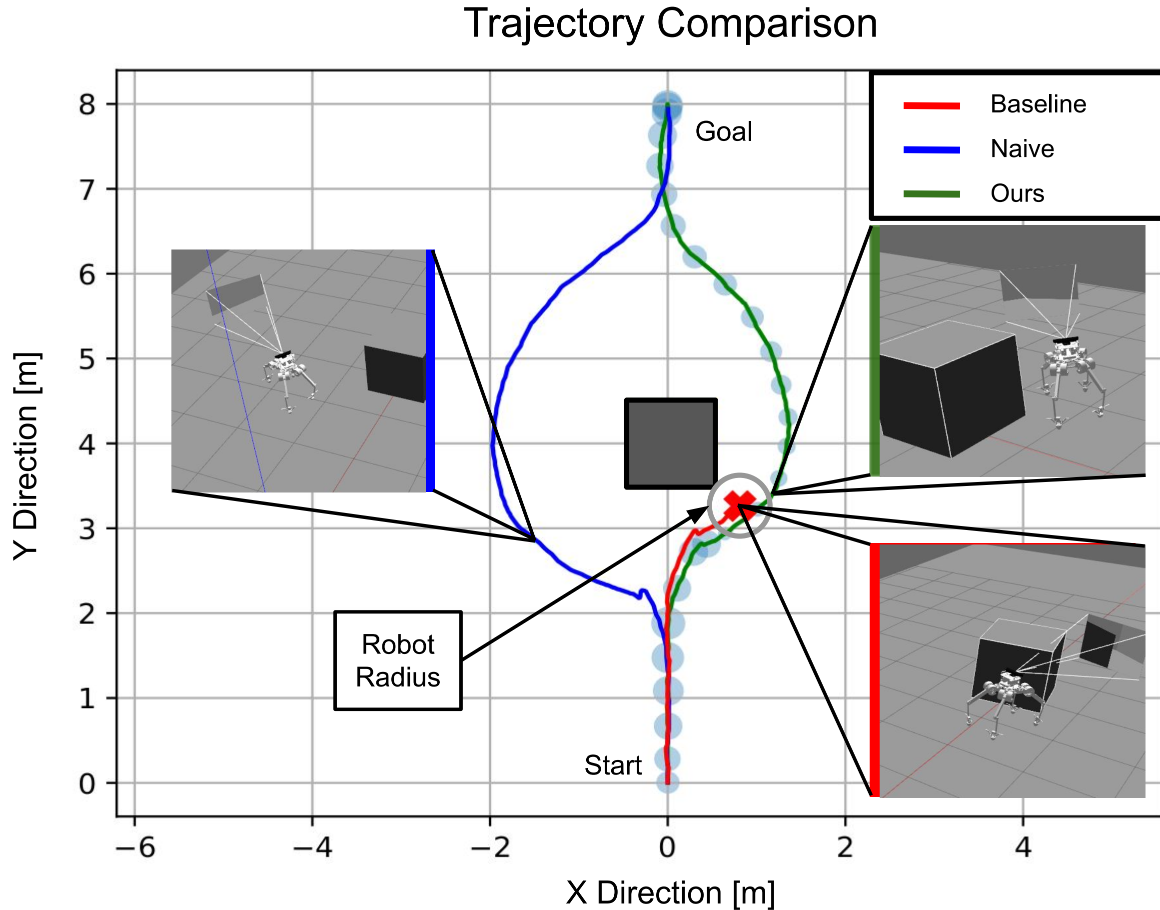


Figure 4.6: **Trajectory Comparison.** A comparison of the trajectories computed by three different approaches. The baseline method (red) is an MPC framework without our extensions to consider propagated future state uncertainty from an RNN, and we define the naive approach (blue) as artificially inflating a robot's boundary through all time. In comparison, our approach (green) can plan for a quick yet safe trajectory by predicting potential future collisions.

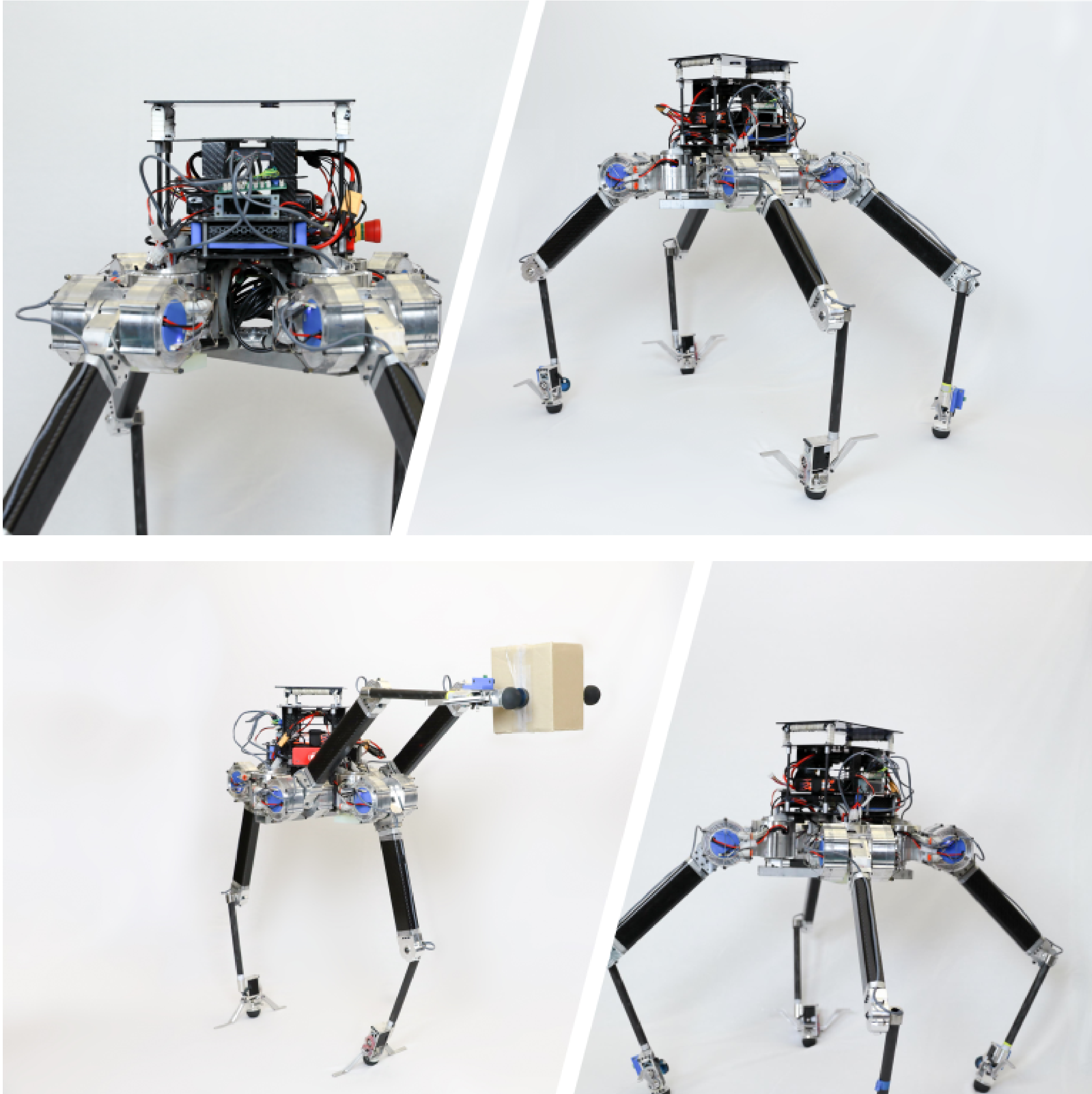


Figure 4.7: **ALPHRED Hardware.** The ALPHRED quadrupedal robot developed by Hooks *et al.* [60] of the RoMeLa robotics laboratory at the University of California, Los Angeles. This complex platform is an ideal model to apply our methods, as showing success on this platform also demonstrates the potential of applying our methods to a wide selection of robotic systems. Table 4.2 describes some physical properties of the system.

CHAPTER 5

Motion Planning for Heterogeneous Multi-Agents

5.1 Introduction

The field of robotics has made remarkable progress in providing diverse sets of robotic platforms with different physical properties, sensor configurations, and locomotion capabilities (e.g., climbing, running, or flying). Thus, developing new planning algorithms that can be ubiquitously applied to a team of heterogeneous robots is a worthwhile endeavor, and applicable to a wide range of tasks from search and rescue to space exploration. However, for multi-agent planners to be used in unknown and uncertain environments, they should consider complex robot dynamics, uncertainty from imperfect exteroceptive and proprioceptive sensor measurements, update uncertainty when robots are in communication range, avoid obstacle collisions, and address desired multi-agent behavior.

One common approach is to fully address some but not all of the described requirements while assuming the rest can either be satisfied in future work, or can be combined with other existing methods. However, combining multiple approaches towards a unified motion planning framework satisfying all requirements can be nontrivial, requiring close examination of the overall feasibility and performance of such a complex system. Thus, in this work, we will examine the feasibility and performance of an end-to-end motion planning framework that addresses the above requirements termed as 'Stochastic model predictive control for Autonomous Bots in uncertain Environments using Reinforcement learning' or SABER.

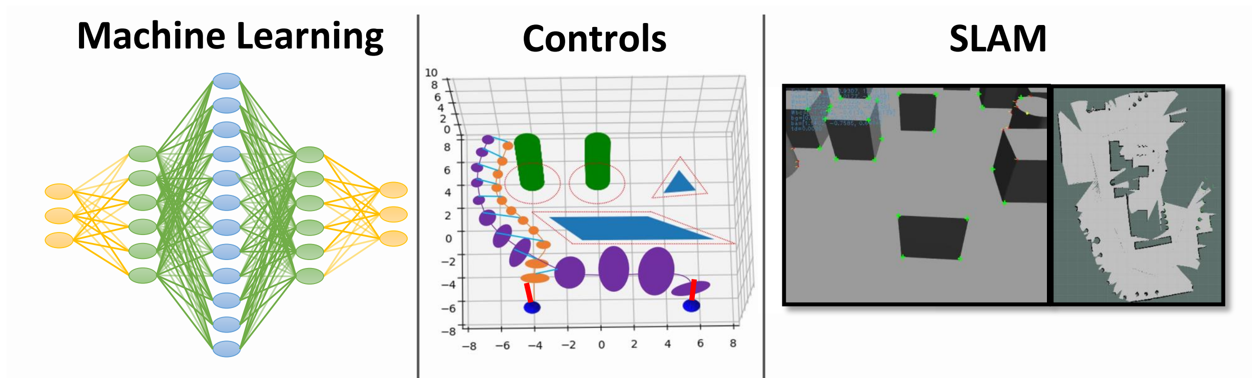


Figure 5.1: **SABER framework.** SABER combines controls (stochastic model predictive control), vision (simultaneous localization and mapping), and machine learning (RNN and DQN), to provide local and globally optimized solutions in unknown and uncertain environments.

Summary of Our Contributions

(1) SABER is an end-to-end motion planning framework for a team of heterogeneous robots that unifies controls, vision, and machine learning approaches to plan paths that account for safety, optimality, and global solutions (our complete framework is shown on a UGV-UAV team).

(2) Cooperative localization algorithms are used for cross-communicating robots, which may include both non-Gaussian and Gaussian measurement noise, where uncertainty is modeled with recurrent neural networks (RNNs) for each agent's sensor configuration using outputs from simultaneous localization and mapping algorithms (SLAM).

(3) Instead of simple heuristics when sampling the map for target positions, we employ Deep Q-learning (DQN) for high-level path planning, which is easily modifiable for learning desired multi-agent behavior and finds global solutions (DQN scalability for more than two robots is also evaluated).

Algorithm 3: SABER

1 Initialize state $X_k^{i:n_r}$, goal $X_{goal}^{i:n_r}$, dt, horizon N, robot size $r_{i:n_r}$, timestep k, empty $Map^{i:n_r}$, uncertainty $\Sigma_{k:k+N+1}^{i:n_r}$, error to goal ϵ, number of robots n_r

2 **while** $\|X_k^{i:n_r} - X_{goal}^{i:n_r}\|_2 > \epsilon$ **do**

3 **if** LiDAR configuration:

4 $X_k^{i:n_r}, \Sigma_k^{i:n_r}, Map^{i:n_r} \leftarrow$ Particle-filter SLAM(odom, scans, $Map^{i:n_r}$)

5 $\Sigma_{k+1:k+N+1}^{i:n_r} \leftarrow \text{RNN}^{i:n_r}(X_k^{i:n_r}, \text{scans})$

6 **if** RGB camera configuration:

$X_k^{i:n_r}, \Sigma_k^{i:n_r}, Map^{i:n_r} \leftarrow$ VIO SLAM(IMU, RGB, $Map^{i:n_r}$)

7 $\Sigma_{k+1:k+N+1}^{i:n_r} \leftarrow \text{RNN}^{i:n_r}(X_k^{i:n_r}, \text{ features})$

8 **continue:**

9 $X_{ref}^{i:n_r} \leftarrow$ Deep Q-Learning($X_k^{i:n_r}, X_{goal}^{i:n_r}, Map^{i:n_r}$)

10 $\mathcal{O}_{j:n_{obs}} \leftarrow$ checkObstacles($X_k^{i:n_r}, r_{i:n_r}, Map^{i:n_r}$)

11 $X_{k+1:k+N+1}^{i:n_r}, U_{k+1}^{i:n_r} \leftarrow$ SMPC($X_{ref}^{i:n_r}, X_k^{i:n_r}, \mathcal{O}_{j:n_{obs}}, \Sigma_{k:k+N+1}^{i:n_r}$)

12 **if** $\forall robot\ i, j : n_r$ within communication range:

13 $\Sigma_{k:k+N+1}^{i:n_r}, X_{k:k+N+1} \leftarrow$ CoopLocalization($\Sigma_{k:k+N+1}^{i:n_r}, X_{k:k+N+1}$)

14 **end**

5.2 Methods

The SABER framework contains both learning (requiring data collection) and non-learning components (traditional control schemes). The non-learning components consist of an SMPC and a distributed Kalman filter, while the learning components consist of an RNN and DQN agent. The RNN and DQN components are trained separately and offline before being implemented into the overall system for online deployment (note, that the RNN is supervised by the SMPC controller on each robot, while the DQN is not integrated with any other component during training). Overall, the algorithm is structured as an SMPC

problem, which moves a robot toward a target location as formulated in 5.2.1. By using state uncertainties and obstacle locations, obstacles are represented as chance constraints within the SMPC cost function (5.2.1.1, 5.2.1.2). If two or more robots are in communication range, their state and uncertainty values are updated using a distributed Kalman filtering approach as described in 5.2.2. To quickly propagate state uncertainties for future timesteps, we use different RNN models based on the robot's sensor configuration, as explained in 5.2.3. In 5.2.4, we formulate a DQN approach, providing the SMPC with target locations which help generate trajectories that move the robots toward a global goal and prevent local minima solutions. See Algorithm (3), Fig. 5.2, or the attached video[1] for an overview of the methods, and 8.3 for implementation details.

5.2.1 Stochastic Model Predictive Control Formulation

The goal of the cost function (equation (5.1)) is to find the optimal control value U_k that minimizes the distance between the current and predicted states ($X_{k \to N+1}$) with a reference state or trajectory ($X^{(ref)}$) while under equality and inequality constraints – where X_k^i is given by the results of localization for each robot i, and k is the current timestep (Q and R are control matrices, and P is described further below):

$$\min_{U^i_{k:k+N\text{-}1}} \sum_{k}^{k+N-1} \|X^i_{k+1} - X^{(ref)i}_{k+1}\|^2_{Q^i} + U^{i\top}_k R^i U^i_k + \|X^i_{k+N+1} - X^{(ref)i}_{k+N+1}\|^2_{P^i} \tag{5.1}$$

$$\text{s.t.} \quad X^i_{k+1} = f^i(X_k, U_k) = A^i X^i_k + B^i U^i_k + W^i_k \tag{5.2}$$

$$X^i_k \sim \mathcal{N}(\bar{X}^i_k, \Sigma^i_k), W^i_k \sim \mathcal{N}(0, \sigma^{2i}) \tag{5.3}$$

$$X^i_{limit} \geq |X^i_k|, U^i_{limit} \geq |U^i_k| \tag{5.4}$$

[1] https://youtu.be/EKCCQtN5Z6A

Obstacle Constraints:

$$\Pr\left(\bigwedge_{j=1}^{N_O} \mathcal{O}_j\right) \geq 1 - \Delta \tag{5.5}$$

Constraint (2) represents multiple shooting constraints, which ensure that the next state is equivalent to a time-invariant linear discretized model, where A and B represent the robot's dynamic matrices. Uncertainty in state as well as the addition of a non-unit variance random Gaussian noise (W) is shown in (3). Note, that the propagation in state uncertainty ($\Sigma^i_{k+1\rightarrow N+1}$) is received from an RNN model (Section 5.2.3), and can be affected by cooperative localization algorithms (Section 5.2.2) at timestep k, if multiple robots are in communication range at timestep k.

Limits on state and controls variables are imposed by constraint (4). For robustness [109], a terminal cost is included with a weighting matrix P which can be obtained by solving the discrete-time Riccati equation (5.6):

$$\begin{aligned} A^{i\top} P^i A^i - P^i - A^{i\top} P^i B^i (B^{i\top} P^i B^i + R^i)^{-1} B^{i\top} P^i A^i + \\ Q^i = 0 \end{aligned} \tag{5.6}$$

5.2.1.1 Chance Constraints for Obstacle Avoidance

Constraint (5) represents chance constraints that enable obstacle avoidance subject to uncertainty in convex regions as done in [12]. This constraint can be rewritten as a disjunction of linear constraints for obstacle $\mathcal{O}_j$:

$$\mathcal{O}_j \Longleftrightarrow \bigvee_{k \in T(\mathcal{O}_j)} \bigwedge_{i \in G(\mathcal{O}_j)} a_i^\top \bar{X}_k - b_i \geq c_i \tag{5.7}$$

where $G(\mathcal{O}_j)$ is the set of linear constraints (indexed by i) for each obstacle (indexed by j), $T(\mathcal{O}_j)$ is the set of timesteps in the MPC prediction horizon (indexed by k), a_i is the vector normal to each line constraint and directed toward state $\bar{X}_k$, r is the radius/size of the robot, and c_i is given by:

$$c_i = \sqrt{2a_i^\top \Sigma_k a_i} \cdot \mathrm{erf}^{-1}(1 - 2\delta_j),\ \delta_j \leq 0.5 \tag{5.8}$$

Important to consider is that the degree of ‘risk’ can be controlled by changing the values of δ_j (related to Δ) for each obstacle $\mathcal{O}_j$. Lower values lead to more evasive behavior (robot moves further away from obstacles) while higher values lead to more risky behavior (robot moves closer to obstacles).

If obstacles are assumed circular (centered at x_{o_j}, y_{o_j}), we can use the following equation, where a_i is equal to an identity vector, x_k and y_k is the center position of the robot, and only a single c_j value needs to be calculated per obstacle:

$$-\sqrt{\left(x_k - x_{o_j}\right)^2 + \left(y_k - y_{o_j}\right)^2} + (r + c_j) \leq 0 \tag{5.9}$$

5.2.1.2 Mixed-Integer Nonlinear Programming

To more effectively consider the disjunctive convex program for polygon-shaped obstacles, as introduced by (7), we can change these constraints into a mixed integer format (we assume the line constraints are in the x-y plane, however, the same equations can be used for the x-z, and y-z planes respectively):

$$\mathcal{O}_j \Longleftrightarrow \bigvee_{k \in T(\mathcal{O}_j)} \bigwedge_{i \in G(\mathcal{O}_j)} I_{i,j} \mathrm{dist}(\bar{X}_k, a_i, m_i, b_i) \geq I_{i,j}(r + c_i) \tag{5.10}$$

$$x_l = a_i^* x_k - y_k + b_i / (a_i^* - m_i) \tag{5.11}$$

$$y_l = m_i x_k + b_i \tag{5.12}$$

$$\mathrm{dist}(*) = \left| -m x_k + y_k - b_i \right| / \sqrt{m_i^2 + 1} \tag{5.13}$$

$$
\text{dist}(*) \begin{cases} \textbf{IF } \text{sign}(y_l - y_k) = \text{sign}(a_y) \bigvee & \\ \quad \text{sign}(x_l - x_k) = \text{sign}(a_x), & -\text{dist}(*) \\ \textbf{ELSE} & \text{dist}(*) \end{cases} \tag{5.14}
$$

$$
I_{i,j} = \{0, 1\} \forall i, j \tag{5.15}
$$

$$
\sum_{i=1}^{size(I_j)} I_{i,j} \geq 1 \quad \forall j \tag{5.16}
$$

where m_i and b_i are the slope and y-intercept of each line constraint i belonging to obstacle $\mathcal{O}_j$, a_i^* is a_y/a_x, x_k and y_k are the x and y position retrieved from robot state $\bar{X}_k$, and the coordinates of the point on the line constraint closest to $\bar{X}_k$ is represented by x_l and y_l (equations (11) and (12)). The dist(*) function approximates the distance between $\bar{X}_k$ and one of the linear constraints of an obstacle as shown in equation (13). Equation (14) returns a positive distance if the robot is 'outside' the obstacle boundary, and a negative distance if the robot is 'inside' the obstacle boundary (a negative distance would cause the line constraint to fail). By definition of constraint (10), only one or more of the line constraints need to be satisfied per obstacle $\mathcal{O}_j$, which is ensured by using binary integer variables under constraints (15) and (16) (e.g., for line constraint i belonging to $\mathcal{O}_j$, if $I_{i,j} = 1$, the robot is outside this obstacle). For simplicity, we assume we have a 'perfect' object detection system. If the robot is close enough to an obstacle, the obstacle is automatically 'seen' and embedded into the SMPC cost function.

5.2.2 Cooperative Multi-Agent Localization

While the propagation of uncertainty for each robot is calculated using an RNN (see 5.2.3), updating the uncertainty after information is exchanged with another robot is done using a distributed Kalman filtering approach [113]. Thus, when two or more robots are in communication range (as pre-specified by the user), their individual uncertainty estimates should

be updated to correctly reflect the gain from additional sensor information.

Equations (17) - (25) describe how the pose for robot i is updated while in communication range of another robot j. The same equations can be further extrapolated to consider additional robots as explained in [113].

For $\forall i, j$ *and* $k \rightarrow k + N + 1$*:*

Propagation:

$$\Sigma^i_{k+1}, \Sigma^j_{k+1} = RNN^i(*), RNN^j(*) \tag{5.17}$$

$$X^i_{k+1}, X^j_{k+1} = f^i(X^i_k, U^i_k), f^j(X^j_k, U^j_k) \tag{5.18}$$

Update:

$$\bar{X}^{+i}_{k+1} = \bar{X}^i_{k+1} + K^i_{k+1}(Z^{ij}_{k+1} - (\bar{X}^i_{k+1} - \bar{X}^j_{k+1})) \tag{5.19}$$

$$S^{ij}_{k+1} = \Sigma^i_{k+1} + \Sigma^j_{k+1} + R^{ij}_{k+1} \tag{5.20}$$

$$Z^{ij}_{k+1} = X^i_{k+1} - X^j_{k+1} \tag{5.21}$$

Update A (first time robots meet):

$$\Sigma^{+ij}_{k+1} = \Sigma^i_{k+1}(S^{ij}_{k+1})^{-1}\Sigma^j_{k+1} \tag{5.22}$$

$$K^i_{k+1} = \Sigma^i_{k+1}(S^{ij}_{k+1})^{-1} \tag{5.23}$$

Update B (all other times robots meet):

$$\Sigma^{+ij}_{k+1} = \Sigma^{ij}_{k+1} - [\Sigma^i_{k+1} - \Sigma^{ij}_{k+1}](S^{ij}_{k+1})^{-1}\Sigma^j_{k+1}[\Sigma^{ij}_{k+1} - \Sigma^j_{k+1}] \tag{5.24}$$

$$K^i_{k+1} = (\Sigma^i_{k+1} - \Sigma^{ij}_{k+1})(S^{ij}_{k+1})^{-1} \tag{5.25}$$

where Z^{ij}_{k+1} is the relative pose measurement between robot i and robot j (X_k is received by current localization, and $X_{k+1 \rightarrow k+N+1}$ can be retrieved by the SMPC solution), and R^{ij} is the relative measurement noise between robot i and robot j.

5.2.3 Recurrent Neural Network for Uncertainty Propagation

Because our SMPC calculates the optimal control (U_k) based on a prediction horizon ($X_{k+1\rightarrow k+N+1}$, $U_{k\rightarrow k+N}$), we must also provide as input the propagation of uncertainty ($\Sigma_{k+1\rightarrow k+N+1}$) at each timestep. As described in more detail in [120], an RNN can provide a computationally fast prediction of future state uncertainties (as it only requires a small network) and can operate in continuous space, making it ideal for online replanning in complex environments. This is achieved as the RNN can model the behavior of a filter (e.g., particle filter or EKF) from SLAM algorithms. However, in this study, we have multiple robots with different sensor configurations, which requires training separate RNN models for each.

In this work, we estimate state uncertainty using two different SLAM algorithms, a Rao-Blackwellized particle-filter SLAM (LiDAR camera configuration) and a Visual-Inertial Odometry (VIO) SLAM (RGB camera configuration). In the particle-filter case, the following equation is used for factorization:

$$\begin{gathered} p\left(x_{1:k}, m \mid z_{1:k}, u_{1:k-1}\right) = \\ p\left(m \mid x_{1:k}, z_{1:k}\right) \cdot p\left(x_{1:k} \mid z_{1:k}, u_{1:k-1}\right) \end{gathered} \tag{5.26}$$

where $x_{1:k} = x_1, ..., x_k$ is the robot's trajectory, $z_{1:k} = z_1, ..., z_t$ are the given observations, $d_{1:k-1} = d_1, ..., d_{k-1}$ are the odometry measurements, and $p(x_{1:k}, m \mid z_{1:k}, d_{1:k-1})$ is the joint posterior estimate about map m (see [134]). For training the RNN for the particle-filter SLAM case, we have 362 input units for each timestep k. The first 2 units is the center position of the robot (x and y), and the 360 other units represent the range distances from LiDAR scans (e.g., for timestep k, we have $d_k^{1:360}$ relative scan distances). The output layers, which use a linear activation function, correspond to the robot's 2×2 x-y covariance matrix which is flattened into a 4×1 array or 4 output units. For the VIO SLAM configuration, we used the same methods for training the RNN as described in [120]. See networks for an overview of the network structure, and 8.3 for further implementation details.

5.2.4 Deep Q-Learning (DQN) for Global Planning

5.2.4.1 DQN formulation

To provide local target positions ($X_{ref}^{i:n_r}$) that direct the robots toward a global goal ($X_{goal}^{i:n_r}$), we implement a DQN agent and use the Bellman equation:

$$Q(s_k, a_k) = r + \gamma \max_{a_{k+1}} Q\left(s_{k+1}, a_{k+1}\right) \tag{5.27}$$

where the state is represented by $s_k \in R^{(N_j \times n_r)+n_r}$, action by $a_k \in R^{9^{n_r}}$, learning rate by γ, n_r is the number of robots, and reward function by r (where N_j is the number grid spaces required to represent each obstacle; described further below). The idea of DQN is to use the Bellman equation (5.27) and a function approximator (with neural networks) to reduce the loss function (we use the Adam optimizer) [97].

Ultimately, the goal of the DQN-agent is to generate target positions at each timestep k for multiple robots and to move them towards a global goal position. To accomplish this and generalize our methods to any map (or changing the location of obstacles after each episode), we use the relative distances between the robot and surrounding obstacles and the relative distance between the robot's position and the global goal as our states: $s_k = (d_j^{i:n_r}, d_{j+1}^{i:n_r}, d_{j+2}^{i:n_r}, ...d_{N_O}^{i:n_r}, d_{goal}^{i:n_r})$, where $d_j^{i:n_r}$ is the relative distance between the robot and obstacles and can be described by $\|X_k^{i:n_r} - \mathcal{O}_j^{i:n_r}\|$, and $d_{goal}^{i:n_r}$ by $\|X_k^{i:n_r} - X_{goal}^{i:n_r}\|$ (X_k is assumed to be the x and y position of the robot). Our DQN agent is trained on a 2D-grid map, where the robots and obstacles are represented as squares (1m^2) within the grid. Thus, the actions permitted by the robot is an 8-directional x-y movement (or no movement) at each timestep. The reward function is simply formulated by providing a positive reward (+1) if the robot gets to the goal and a negative reward (-1) if the robot hits an obstacle, which results in termination of the episode (we allow the UAV to 'fly' over some obstacles by modifying its reward function to not receive a penalty for hitting that obstacle). To test different behaviors, we also added a reward (+0.1) at each timestep when both robots are within a 2 meter distance (see (D) in Fig. 5.5).

We also assume the dimensions of each obstacle are known *a priori* (to estimate this without this assumption may require an object detection pipeline). Thus, by knowing the height of each obstacle, a UAV can use this value in its chance constraint to fly over obstacles. Finally, mapping our states to our actions is done through a linear neural network. For an overview of the network structure, and the training/testing process, see Fig. 5.5.

5.3 Experimental Validation

5.3.1 Implementation details

SABER is demonstrated on a UGV (Turtlebot3) equipped with a 360 degree LiDAR camera, and a UAV (Quadrotor) equipped with a RGB camera in a Gazebo simulation running in real-time with additive noise. The dynamic equation of motion for the UGV assumes states composed of center of mass position and heading angle ($X = [x, y, \theta]$), actions composed of linear and angular velocity ($U = [v, \omega]$) and the following matrices for the discretized equation:

$$A = I \in R^{3\times3},\ B = \begin{bmatrix} \cos(\theta)\delta t & 0 \\ \sin(\theta)\delta t & 0 \\ 0 & \delta t \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix}$$

The UAV assumes similar dynamics as in [18], where the states are center of mass position, linear velocity (x, y, and z components), angle, and angular velocity (we consider pitch and roll but keep yaw fixed) or $X = [x, v_x, \theta_1, \omega_1, y, v_y, \theta_2, \omega_2, z, v_z]$, and actions composed of thrust $U \in R^{3\times1}$. The A and B matrices and their parameters are fully described in [18], where $A \in R^{10\times10}$ and $B \in R^{10\times3}$.

To solve the cost function (5.1), we use the mixed-integer nonlinear programming (MINLP) solver 'bonmin' [16], as we need to consider both integer and continuous variables. Constraints and problem formulation were setup using CasADi [4] running on a laptop with an

Intel Core i7-8850H CPU, and NVIDIA Quadro P3200 GPU.

To collect data for training our RNN model for the UGV, we used the GMapping package [53] to create a map of the environment, and then implemented the AMCL package [134] to track the robot's pose and receive the uncertainty covariances using this map with particle-filter SLAM (uncertainty outputs from the filter are considered as the 'ground truth'). In the UAV case, we used the XIVO SLAM package [44] to make localization and covariance estimations (XIVO uses an Extended-Kalman Filter). The Adamax optimizer was used for training and the Mean Squared Error (MSE) was implemented as the loss function for both RNN models (covariance matrices need to be converted to be positive semi-definite during usage, see [120]). For training (for both models), we created 4 different maps with obstacles randomly distributed, where robots traverse about these maps via the SMPC. Note, that by definition of a particle-filter and EKF, the former assumes non-Gaussian noise while latter assumes Gaussian noise.

5.3.2 Results

5.3.2.1 Analysis of learning components and their significance

The training loss for the RNN networks are shown in Fig. 5.4. The RNN networks for both SLAM algorithms were trained for 500 epochs (validation loss was observed to be close to the training loss), and shows a strong correlation between truth (output of SLAM in training) and predicted covariance. We also demonstrate that our RNN can model the behavior of covariance outputs generated by different sensor configurations (i.e., SLAM algorithms), as seen in (A) and (B) of Fig. 5.7 (e.g., we observed an increase in uncertainty for VIO SLAM when too close to obstacles, while seeing the opposite behavior for a particle-filter SLAM). Important to note, is that this result indicates that the RNN can model both non-Gaussian (particle-filter) and Gaussian (EKF) noise. By modeling the propagation of uncertainty of different measuring systems and integrating them into the SMPC prediction

horizon through chance constraints, we ensure control values that avoid obstacle collision (without over avoidance) for each individual robot.

In (B) of Fig. 5.5, we show that during training of our DQN, the rewards would increase over a majority of the 35,000 episodes, reaching a steady-state at approximately 27,000 (except for the 5-robot team). In Table 5.3.2.1, we also compared our DQN to other 2D baseline algorithms (we chose resolution settings for RRT, RRT-star, and A-star so that their path lengths were similar to the DQN, then evaluated their computation time). The results indicate that on average, the DQN had the lowest computation time and was comparable to A-star in terms of its path length to the goal (considered optimal for our map resolution). Unlike the baseline algorithms, our DQN also has the additional functionality of learning multi-agent semantic behavior–it was successful at moving the robots simultaneously to their goal points as shown in (C), and, as expected, stayed closer to each other when rewarding them based on close proximity as shown in (D) of Fig. 5.5. Although it would be possible to modify the baseline planners to consider multi-agent planning, the learning-based approach considers potential semantics between agents that would be difficult to quantify and define beforehand within a cost function.

High-level planner analysis (100 trials on map (D) of Fig. 5.6)

Planner	Path length (m)	std dev (m)	Solve time (s)	std dev (s)
RRT	10.83	±1.28	0.120	±0.091
RRT-star	10.40	±1.20	0.084	±0.073
A-star	9.66	±0.00	0.154	±0.002
DQN	9.68	±0.00	0.051	±0.001

Average Computation time of SABER components (10 minutes of data)					
Component	SMPC	RNN	VIO SLAM	PF SLAM	SABER
Solve time (s)	0.0559	0.0212	0.030	0.073	0.193
std dev (s)	±0.0187	±0.0077	±0.003	±0.005	±0.110

The limitation of our DQN is that it's currently most capable in planning in 2D rather than 3D space, which helps lower training time and increase convergence (more complex DQN formulations may be necessary if the problem is either scaled to 3D, assumes a map bigger than 10×10m, or uses more than 3 robots, see (B) of Fig. 5.5). However, since we use an SMPC to avoid obstacles in 3D space using chance constraints, a 2D planner is sufficient for our application.

5.3.2.2 Validation of the SABER algorithm

The complete planner is exemplified in Fig. 5.6 on a UGV-UAV team. We show that the SMPC of the UGV and UAV uses the state uncertainties estimated by an RNN to avoid colliding in obstacles in dense maps (see (A), and (B)). A special case is also shown in (C), where the SMPC of the UGV (without the DQN) reaches a local minima solution and is stuck behind the obstacle. However, when using the DQN's proposed path, the UGV can successfully reach its global goal. Note, that the SMPC and not the DQN considers both the dynamics of the robots and the uncertainty provided by their RNN models, thus, the actual path (shown in orange/purple) will differ slightly from the proposed DQN path (marked by triangles). We also evaluate the average computation time of the SABER algorithm (and its individual components) in Table 5.3.2.1, showing a computation time of $\simeq 0.19$ seconds/timestep. Lastly, in Fig. 5.8, we verify that SABER performs best compared to several baselines (using distance to goal vs time as our metric on map (D) of Fig. 5.6). The figure shows that MPC alone (no uncertainty is considered) causes an obstacle collision for the UGV and UAV (this likely occurred as the simulation contains noise, and the MPC

is unaware of this noise during optimization). Both robots are able to get to their goals using a naïve stochastic MPC (uncertainty is considered by artificially inflating all obstacle boundaries), but due to over avoidance take longer to reach their goals. By adding our RNN to the SMPC allows both robots to reach their goals more quickly (uncertainty is now more accurately propagated within the SMPC prediction horizon). However, we observed that without a global planner, both robots run into local minima issues (i.e., robots would sometimes get stuck behind an obstacle for some time before reaching their goals). With a global planner (e.g., DQN, A-star), the robots reach their goals in the quickest manner (avoiding local minima issues). Although A-star and DQN provide near-optimal paths, the reason the DQN shows slightly better improvements (including better computation time) over A-star is because the DQN also accounts for multi-agent behavior, where robots were trained to stay close to each other when possible before reaching their respective goals (staying in close proximity decreases uncertainty via the Kalman filter).

5.4 Conclusion

In this work, we demonstrated that the SABER algorithm (which combines several fields of robotics including controls, vision, and machine learning into a single framework) is computationally feasible ($\simeq$ 0.19 seconds/timestep) and plans paths for heterogeneous robots to reach a global goal while satisfying diverse dynamics, constraints, and consideration of uncertainty. In future work, we plan to relax the assumption of a perfect object detection system and will focus on expanding our DQN to consider more complex behavior and tasks.

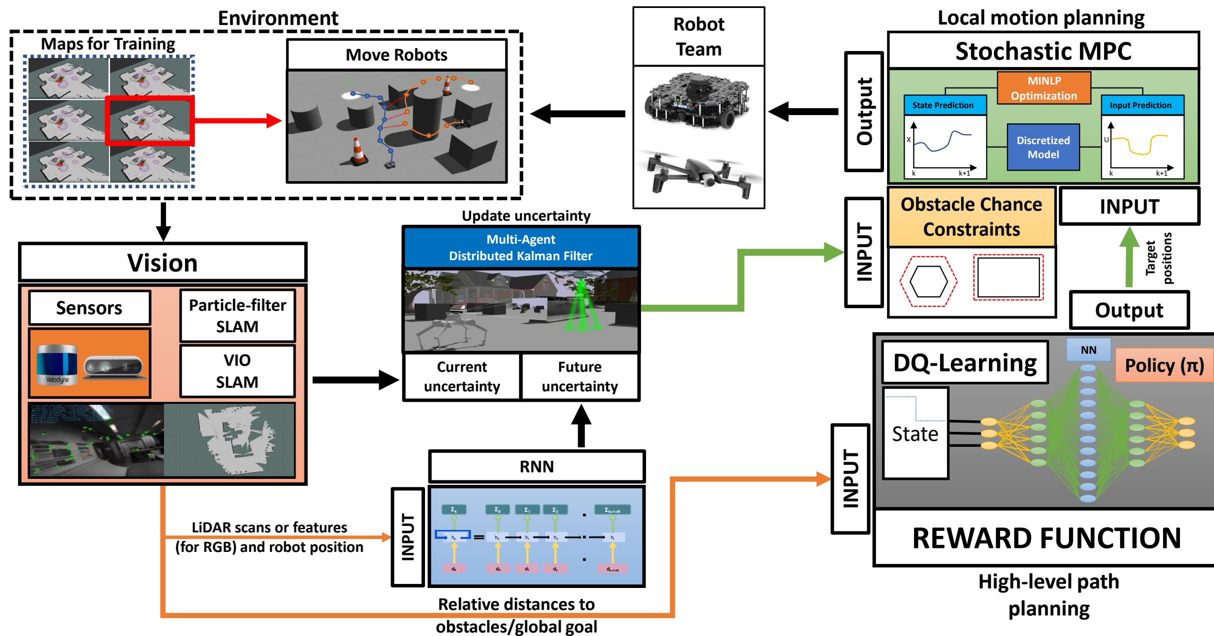
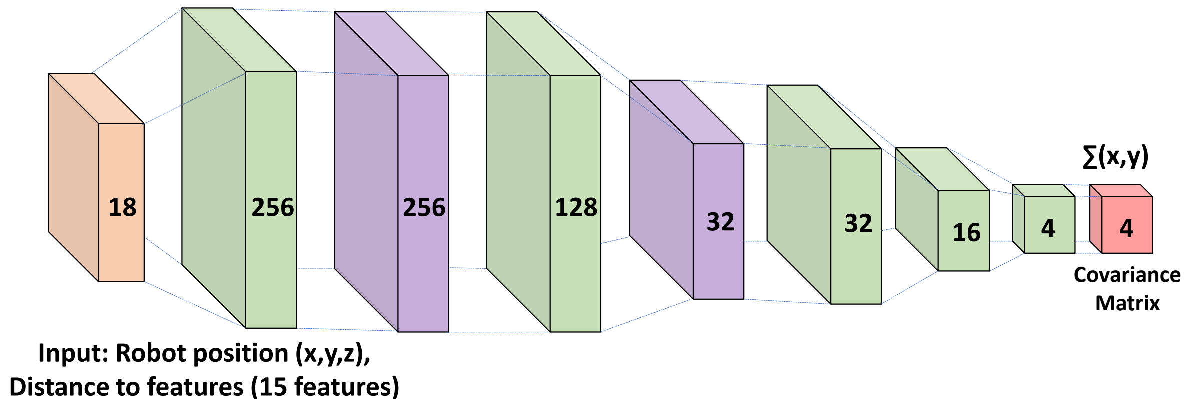


Figure 5.2: **SABER Algorithm.** This figure demonstrates the overall SABER planning algorithm in the testing phase, which can plan paths for one or more robots simultaneously. At timestep k, the environment provides information to robots that either carry a LiDAR or RGB camera and IMU; for the LiDAR configuration, a particle-filter SLAM is implemented, while for the RGB configuration, Visual-Inertial Odometry SLAM (VIO SLAM) is implemented. The sensors provide either scans or distance to feature information to a recurrent neural network model (which serve as inputs), and outputs the propagation of state uncertainty for future timesteps. If two or more robots are within communication range, a distributed Kalman filter updates the current and future states and their uncertainties to a more accurate estimate. These updated states and uncertainties are used to update the chance constraints for obstacle avoidance. These constraints are then considered by a stochastic MPC controller, which follows a given target position, provided by a deep Q-learning (DQN) agent that aims to move the robot towards a global goal. DQN uses the relative distances between the robots and the respective obstacles as its states, provides a target position for all robots as its actions, and is trained on several different maps with obstacles randomly distributed in each. Note, that the SMPC, SLAM, and RNNs components run on each robot individually, however, the DQN is run on a centralized base (which may be on the robot itself).

A) – RNN (input/output using VIO SLAM on UAV)



B) – RNN (input/output using particle-filter SLAM on UGV)

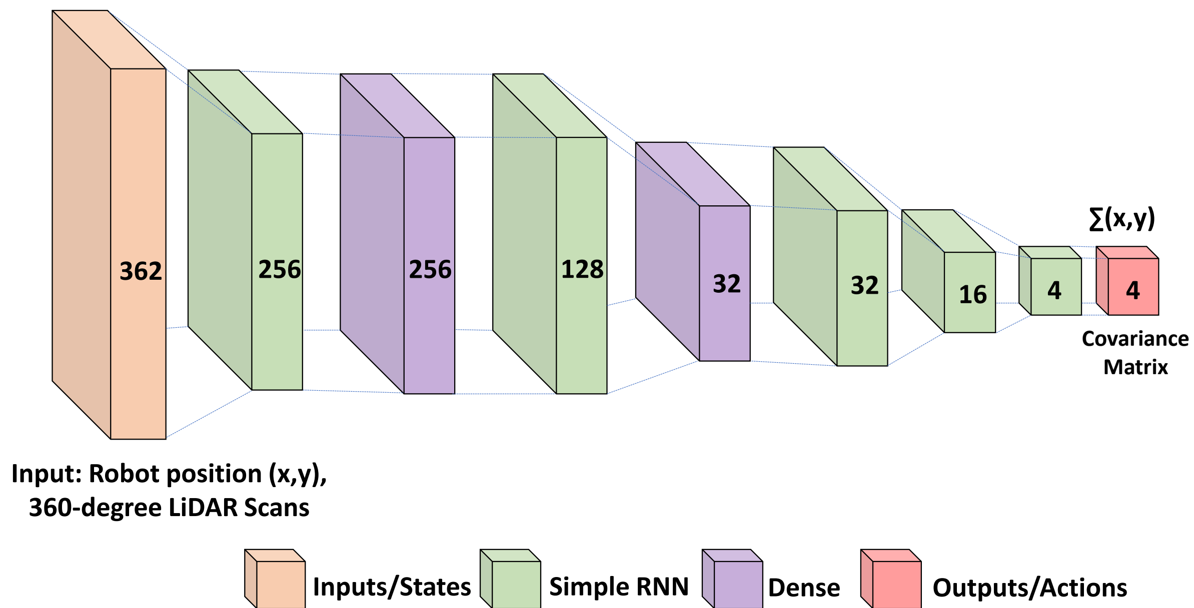


Figure 5.3: **Network structures.** We show the RNN structure used to model an EKF from a VIO SLAM algorithm in (*A*), or a particle-filter SLAM algorithm in (*B*) (5.2.3). The inputs are shown in orange, and correspond to either features/robot position (using VIO SLAM) or LiDAR scans/robot position (using particle-filter SLAM). The outputs are shown in red, and correspond to the x-y covariance matrix (which represents uncertainty in x-y position). The layer type is color coded below, where green represents a simple RNN layer, and purple a dense layer.

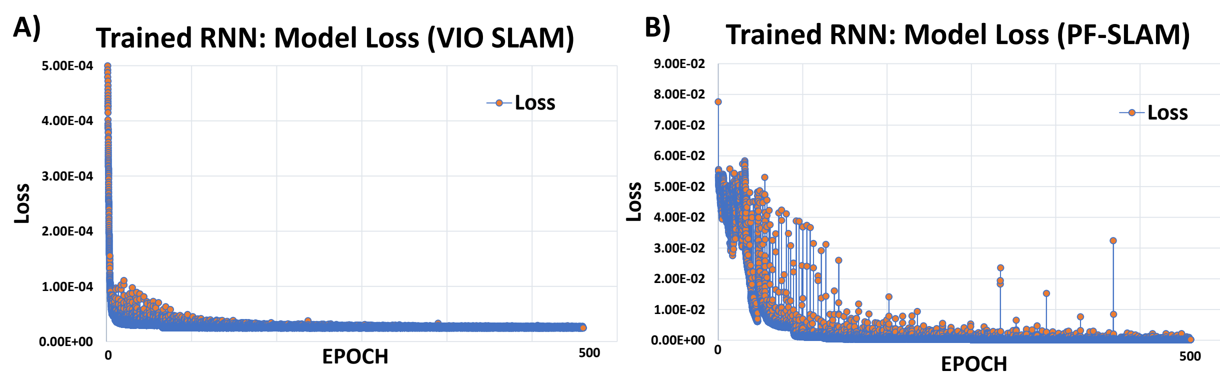


Figure 5.4: **Training loss.** Here we show the training loss for the RNNs, which were trained on uncertainty covariance outputs (in position) of a Visual-Inertial SLAM in (A) and a particle-filter SLAM in (B). The training was done using 500 epochs and on 4 different maps. Note, that the noise observed in (B) may be due to the particle-filter estimations/simplifications done in [53].

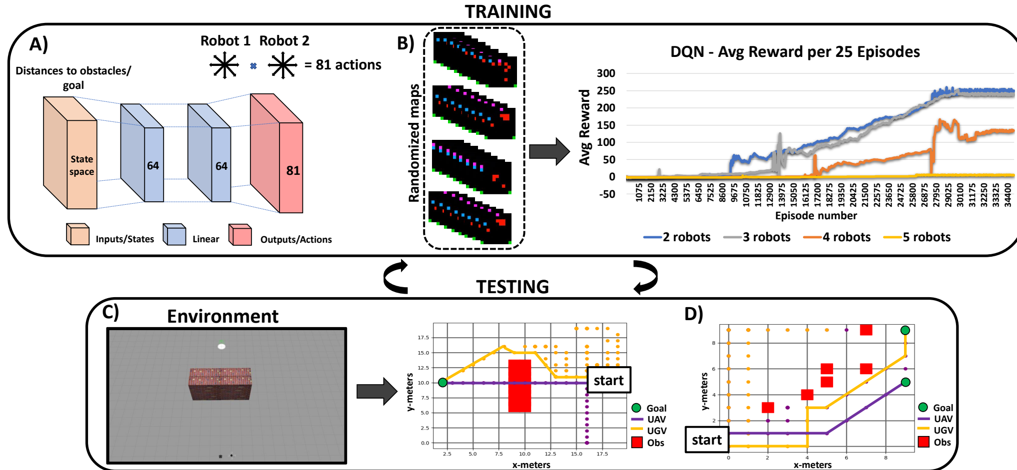


Figure 5.5: **DQN Training and Testing Procedure.** In (A), we show the neural network structure used in our DQN algorithm (5.2.4). The network maps the inputs (i.e., states or relative distances between robots and obstacles/goals) to the outputs (i.e., actions or next target positions for the robots). The states and actions are connected by a linear neural network model (blue). In (B) we visually show the training process of the DQN for a 2 to 5 robot team, where all robots were trained to go to the goal location while avoiding obstacles (obstacles are randomized for each episode). The average rewards (calculated from 25 episodes at a time and divided by number of robots) are shown across the 35,000 episodes of training (training time was 5 hours). In (C) we show an example of how the environment can be transcribed into a 2D plot and apply the DQN to traverse multiple robots toward their goals. We also allow the UAV to fly over the obstacles (and assume we know the height of the obstacle *a priori*) while the UGV must avoid it. In (D) we show another example of our DQN, but this time the 2-robot UAV and UGV team have separate goal locations, and we add a reward incentive when both robots are near each other at each timestep.

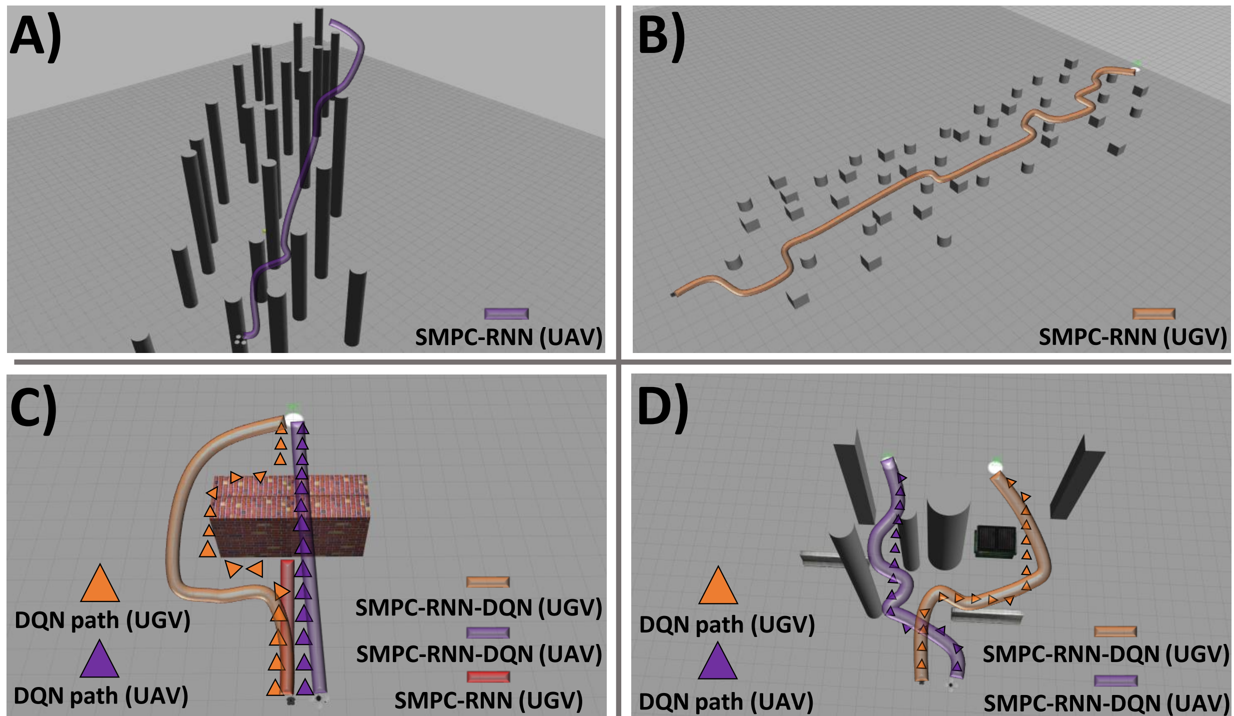


Figure 5.6: **SABER Algorithm Results.** This figure demonstrates the overall SABER planning algorithm. In (A) and (B) we first show the capability of the SMPC-RNN to navigate the UGV and UAV in a densely populated space. In (C), we show that the SMPC-RNN of the UGV cannot get to the goal state, because of the occurrence of a local minima. However, with a DQN (which provides a global path illustrated by triangles), the UGV (orange) can correctly maneuver around the obstacle. The UAV (purple) can simply use it's z-axis to fly above the obstacle. A more complex example is shown in *(D)*, where both robots are directed towards different goal locations simultaneously.

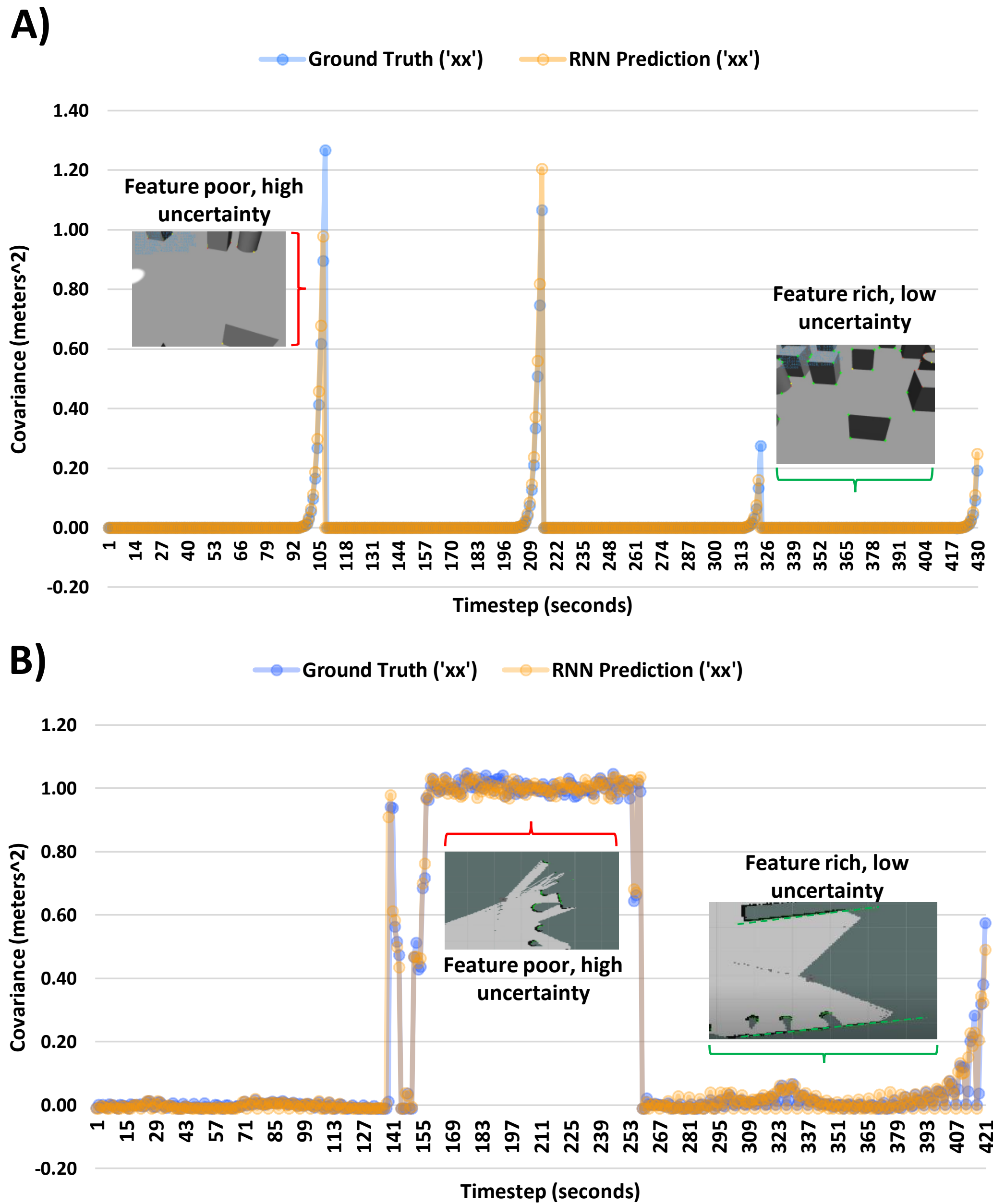


Figure 5.7: **RNN Results.** In this figure, we show the true covariance value (where 'xx' represents the covariance of the center of mass in the x position as an example) in blue and the predicted covariance in orange for about 430 seconds of data. This is done when modeling the uncertainty using VIO SLAM (*A*) and particle-filter SLAM (*B*) on a test map. Note, that the predicted and true values almost perfectly align, demonstrating the RNN's ability to make valid uncertainty predictions. Lastly, we show more explicitly in the graphs, that when more features are tracked (green arrow) the lower the estimated uncertainty, while fewer features corresponded to higher uncertainty (red arrow).

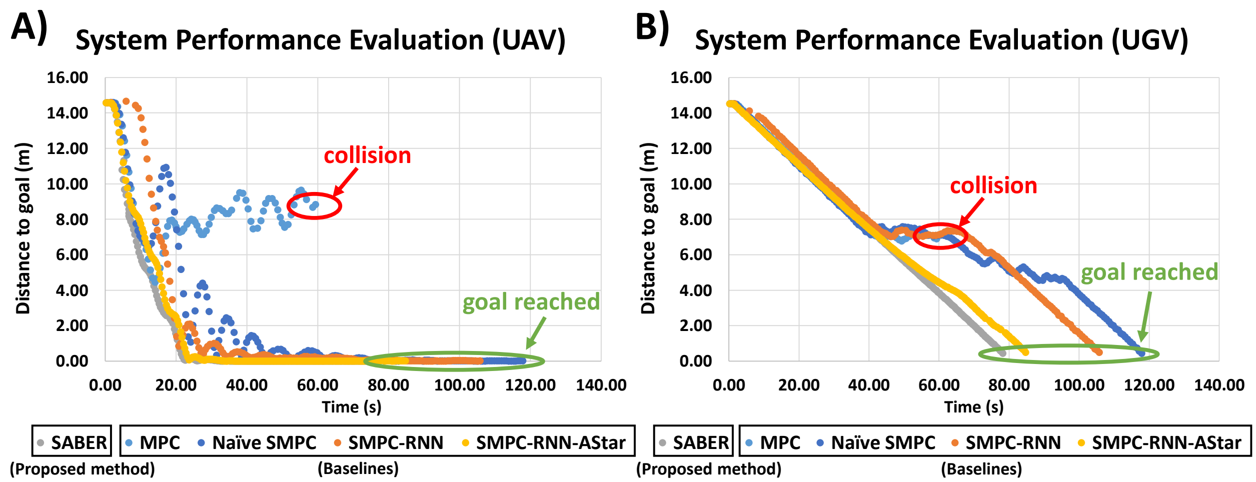


Figure 5.8: **System performance.** We compare the results of SABER (SMPC-RNN-DQN) using the same map as *(D)* in Fig. 5.6) against baselines for the UAV *(A)* and UGV *(B)* with distance to goal vs time as our metric. See 5.3.2.2 for details.

CHAPTER 6

Auto-Calibrating Planning and Control Parameters

This letter proposes to use an alternative method, see Fig. 7.1, to the above approaches for achieving robust locomotion. In [93, 94], a Kalman filter technique was used to estimate control parameters with a training objective that evaluates the performance of a closed-loop system online using a recursive implementation. This approach was successfully used to automatically tune a variety of control architectures such as the weights of a neural network, cost function weights of an LQR, or gains of a PID controller to achieve stable motion of an autonomous vehicle. Although auto-tuning methods have been applied before to tune various controller gains, typically using some form of Bayesian optimization (BO) [91, 99], the applied method differs as we do not require a trial-and-error implementation, do not need a surrogate function, and can handle disturbances due to the recursive nature of the applied method [93, 94]. One of the issues with BO is that their performance decreases in higher dimensions [22]. In [105], this problem was overcome by including domain knowledge into BO for tuning walking controllers of humanoid robots. In [52], the parameters of running and jumping motions are tuned through a policy search of generic sets of motion primitives and their cost functions. Both [105, 52] require a trial-and-error implementation which requires extensive training using simulators before application to hardware. Our method however, is able to calibrate control parameters on a single run without relying on trial-and-error.

Additionally, while other algorithms have been proposed to tune controller gains [104, 30], these algorithms have never been applied (or evaluated) to directly tune reference trajectories without having to include a predefined set of reference trajectories [55].

Here, we employ the method of auto-tuning initially proposed in [93, 94] on a legged system using an Unscented Kalman Filter (UKF) implementation and apply it not only to controller gains, but directly on physically-meaningful parameters such as 'step clearance', 'forward progress', 'energy efficiency', and 'dynamic and kinematic gait stability', see Fig. 6.1. This is achieved by online tuning the robot's future reference trajectories which uses a training objective to satisfy user-defined specifications or requirements. We demonstrate the method on a quadruped robot, the Unitree A1, in a high-fidelity physics simulation provided by the robot manufacturer. We demonstrate walking on both even and uneven terrain, and present several test-cases such as tuning reference trajectories that satisfy the physical parameters mentioned previously, and controller weights of a swing and stance controller. The presented method for automating the controller calibration and the trajectory planner can save development/deployment time as manually tuning control parameters can be tedious and time consuming, facilitates in abstracting the tuning problem into physically meaningful parameters, and enables greater autonomy of robotic systems.

6.1 Preliminaries

6.1.1 Notation

Given two integer indices n, m with $m < n$ and $\mathbf{x}_i \in \mathbf{R}^{n_x}$, we define $\mathbf{x}_{m|n} \in \mathbf{R}^{n_x(n-m+1)}$ as the vectorized sequence that comprises $\mathbf{x}_i$ from $i = m$ through $i = n$,

$$\mathbf{x}_{m|n} := \begin{bmatrix} \mathbf{x}_m \\ \vdots \\ \mathbf{x}_n \end{bmatrix}.$$

We define $\|\mathbf{x}\|_{\mathbf{\Sigma}} := \mathbf{x}^T \mathbf{\Sigma} \mathbf{x}$ and $\mathcal{N}(\boldsymbol{\mu}, \mathbf{\Sigma})$ as the Gaussian distribution with mean vector $\boldsymbol{\mu}$ and covariance matrix $\mathbf{\Sigma}$. The notation $\mathbf{x} \sim \mathcal{N}(\mu, \mathbf{\Sigma})$ means $\mathbf{x}$ sampled from $\mathcal{N}(\mu, \mathbf{\Sigma})$. Further, $diag(\boldsymbol{\lambda}) \in \mathbf{R}^{n_\lambda \times n_\lambda}$ is a matrix, whose diagonal entries are the entries of a vector $\boldsymbol{\lambda} \in \mathbf{R}^{n_\lambda}$.

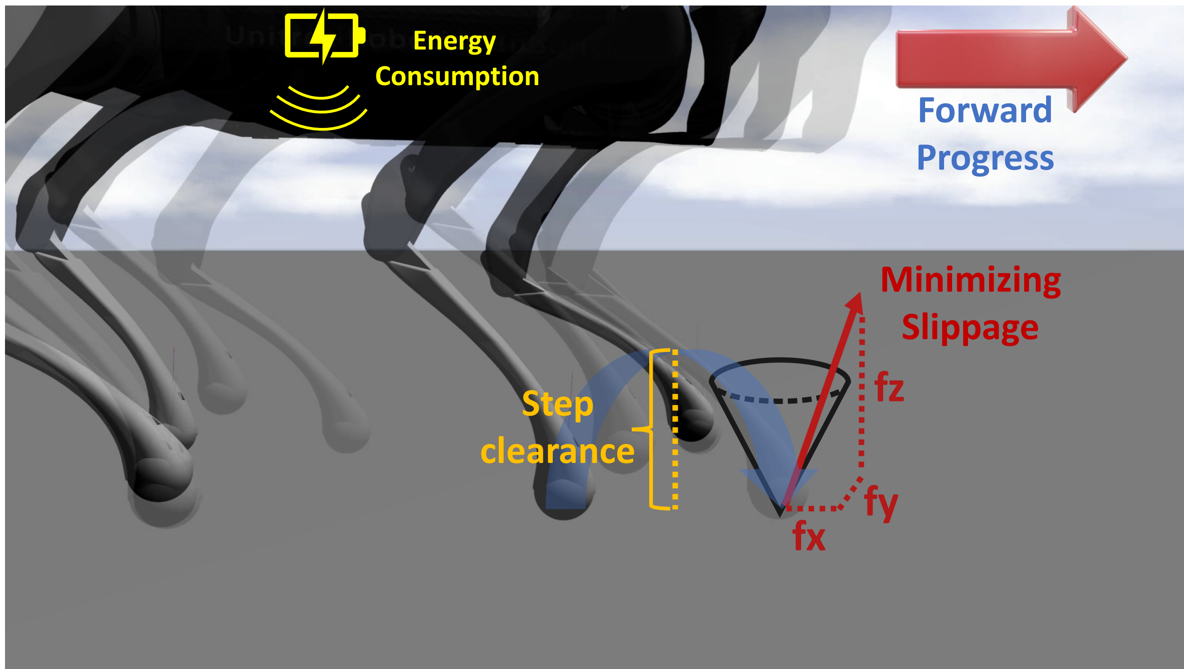


Figure 6.1: **Auto-Tuning reference trajectories in Gazebo.** The robot follows a reference trajectory that is being tuned by the auto-tuning formulation. The figure shows the robot following a desired velocity (forward progress), a desired foot height (step clearance), minimizing foot slippage (ground reaction forces), and minimizing energy consumption.

6.1.2 Unscented Kalman Filter for Control Parameter Tuning

This letter applies the auto-tuning method proposed in [93, 94] to calibrate control parameters of the Model Predictive Control (MPC) stance controller, the PD swing controller, as well as the reference trajectory. The auto-tuning method is based on an UKF, which "estimates" the optimal control parameters measured with respect to a training objective. The UKF uses deterministic samples (called sigma points) around the mean, which are propagated and used to update the mean and covariance estimates [133]. Further, it uses a model of the system dynamics in order to obtain evaluations of the sigma points, which are then used to update the control parameters.

The method is model-based, i.e., it uses a model of a dynamical system (denoted by

$\mathbf{dyn}(\mathbf{x}_k, \mathbf{u}_k)$),

$$\mathbf{x}_{k+1} = \mathbf{dyn}(\mathbf{x}_k, \mathbf{u}_k) + \mathbf{w}_k \tag{6.1}$$

with the state $\mathbf{x}_k \in R^n$, the control input $\mathbf{u}_k \in R^m$, and process noise or model mismatch $\mathbf{w}_k$ at time k. The method calibrates control parameters, $\boldsymbol{\theta} \in R^L$, of a generic controller, $\mathbf{u}_k = \kappa_\theta(\mathbf{x}_k)$ to minimize the training objective

$$\|\mathbf{y}_k - \mathbf{h}(\boldsymbol{\theta}_k)\|_{\mathbf{C}_y^{-1}} \tag{6.2}$$

with a positive definite $\mathbf{C}_y$, desired nominal values $\mathbf{y}_k$, and specification function $\mathbf{h}(\boldsymbol{\theta})$, where

$$\mathbf{h}(\boldsymbol{\theta}) := \mathbf{r}(\mathbf{x}_{k-N}, \mathbf{x}_{k-N+1}, ..., \mathbf{x}_k, \mathbf{u}_{k-N}, ..., \mathbf{u}_{k-1}),$$

i.e., $\mathbf{y}_k = \mathbf{h}(\boldsymbol{\theta}_k)$ when the dynamical system satisfies all the specifications in the training objective, exactly.

For the legged robot, the control parameters are tuned during operation (episodically, from time step $k-N$ to k) according to

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-N} + \Delta\boldsymbol{\theta}_k, \tag{6.3}$$

where the update $\Delta\boldsymbol{\theta}_k$ is computed based on sensor measurements, $\mathbf{x}_k$, $\mathbf{u}_k$, and the training objective in (6.2).

The idea of the auto-tuning method is to treat the control parameter adaptation problem as an optimal estimation problem with prior distributions $\Delta\boldsymbol{\theta}_k^{prior} \sim \mathcal{N}(\mathbf{0}, \mathbf{C}_\theta)$ and $\mathbf{y}_k \sim \mathcal{N}(\mathbf{h}(\boldsymbol{\theta}_k), \mathbf{C}_y)$. Thus, the parameter tuning law in (6.3) results from the corresponding posterior distribution,

$$\Delta\boldsymbol{\theta}_k = \mathbf{K}_k \left(\mathbf{y}_k - \hat{\mathbf{h}}_k\right) \tag{6.4a}$$

with the Kalman gain, $\mathbf{K}_k$, computed as

$$\hat{\boldsymbol{\theta}}_k = \sum_{j=0}^{2L} v_j^a \boldsymbol{\theta}_k^{sp,j} \tag{6.4b}$$

$$\hat{\boldsymbol{h}}_k = \sum_{j=0}^{2L} v_i^a \mathbf{h}_k^{sp,j} \tag{6.4c}$$

$$\mathbf{h}_k^{sp,j} = \mathbf{h}(\boldsymbol{\theta}_k^{sp,j}) \tag{6.4d}$$

$$\mathbf{S}_k = \mathbf{C}_v + \sum_{j=0}^{2L} v_j^c (\mathbf{h}_k^{sp,j} - \hat{\boldsymbol{h}}_k)(\mathbf{h}_k^{sp,j} - \hat{\boldsymbol{h}}_k)^\top \tag{6.4e}$$

$$\mathbf{Z}_k = \sum_{j=0}^{2L} v_j^c (\boldsymbol{\theta}_k^{sp,j} - \hat{\boldsymbol{\theta}}_k)(\mathbf{h}_k^{sp,j} - \hat{\boldsymbol{h}}_k)^\top \tag{6.4f}$$

$$\mathbf{K}_k = \mathbf{Z}_k \mathbf{S}_k^{-1} \tag{6.4g}$$

and the posterior covariance, which is used to generate the sigma points, computed as

$$\mathbf{P}_{k|k-N} = \mathbf{C}_\theta + \sum_{j=0}^{2L} v_i^c (\boldsymbol{\theta}_k^{sp,j} - \hat{\boldsymbol{\theta}}_k)(\boldsymbol{\theta}_k^{sp,j} - \hat{\boldsymbol{\theta}}_k)^\top \tag{6.4h}$$

$$\mathbf{P}_{k|k} = \mathbf{P}_{k|k-N} - \mathbf{K}_k \mathbf{S}_k \mathbf{K}_k^\top, \tag{6.4i}$$

where $\boldsymbol{\theta}_k^{sp,j}$ is the jth sigma point, v_j^c and v_j^a denote weights associated with the sigma points, $\mathbf{Z}_k$ is the cross-covariance matrix, $\mathbf{S}_k$ is the innovation covariance, and $\mathbf{P}_{k|k}$ is the estimate covariance. Hence, the UKF implementation uses $2L+1$ sigma points. The reader is referred to [93, 94] for more details.

For each sigma point evaluation (6.4d), the system dynamics (6.1) is simulated, where the model mismatch, $\mathbf{w}_k$, is calculated using measured data, $\mathbf{x}_{k-N}$ through $\mathbf{x}_k$.

We choose the weights $v_0^a = v_0^c = 0$, $v_i^a = v_i^c = (1-v_0^a)/(2L)$ and the sigma points as $\boldsymbol{\theta}_k^{sp,0} = \boldsymbol{\theta}_k$, $\boldsymbol{\theta}_k^{sp,j} = \boldsymbol{\theta}_k + \sqrt{L/(1-v_0^a)}\boldsymbol{\Gamma}^j$ for $j=1, ..., L$, and $\boldsymbol{\theta}_k^{sp,j} = \boldsymbol{\theta}_k - \sqrt{L/(1-v_0^a)}\boldsymbol{\Gamma}^j$ for $j=L+1, ..., 2L$ with $\boldsymbol{\Gamma}^j$ being the jth column of $\boldsymbol{\Gamma}$ and $\mathbf{P}_{k-N|k-N} = \boldsymbol{\Gamma}\boldsymbol{\Gamma}^\top$, i.e., $\boldsymbol{\Gamma}$ is calculated using the Cholesky decomposition.

6.2 Robot Control Architecture

6.2.1 Swing Controller

The swing controller in this letter is similar to [34], which is used to compute the torque for each foot i for all three joints of the robot as (time step k omitted for simplicity)

$$\begin{aligned} \boldsymbol{\tau}_i = \mathbf{J}_i^\top \left[\mathbf{K}_p \left(\mathbf{p}_{i,\text{ ref}}^b - \mathbf{p}_i^b\right) + \mathbf{K}_d \left(\mathbf{v}_{i,\text{ ref}}^b - \mathbf{v}_i^b\right)\right] \\ + \mathbf{J}_i \boldsymbol{\Lambda}_i \left(\mathbf{a}_{i,\text{ ref}}^b - \dot{\mathbf{J}}_i^\top \dot{\mathbf{q}}_i\right) + \mathbf{V}_i \dot{\mathbf{q}}_i + \mathbf{G}_i \end{aligned} \tag{6.5}$$

where $\boldsymbol{\tau}_i \in R^3$ is the joint torque, $\mathbf{q}_i \in R^3$ and $\dot{\mathbf{q}}_i \in R^3$ are the current joint position and velocity of foot i, $\mathbf{J}_i \in R^{3\times3}$ is the foot Jacobian, $\mathbf{K}_p$ and $\mathbf{K}_d$ are the proportional and derivative (PD) gain matrices (3×3 diagonal positive semi-definite), $\mathbf{p}_{i,\text{ ref}}^b \in R^3$ and $\mathbf{p}_i^b \in R^3$ are the reference and current footstep positions in the body frame, $\mathbf{v}_{i,\text{ ref}}^b \in R^3$ and $\mathbf{v}_i^b \in R^3$ are the reference and current footstep velocities in the body frame, $\mathbf{a}_{i,\text{ ref}}^b \in R^3$ is the reference footstep acceleration in the body frame, $\mathbf{V}_i \in R^3$ is the torque due to the coriolis and centrifugal forces, $\mathbf{G}_i \in R^3$ is the torque due to gravity, and $\boldsymbol{\Lambda}_i \in R^{3\times3}$ is the operational mass matrix.

6.2.2 Stance Controller

The stance MPC calculates ground reaction forces $\mathbf{f}_i$ for all feet i and is formulated as in [34],

$$\min_{\mathbf{x},\mathbf{f}} \sum_{k=0}^{N_{MPC-1}} \|\mathbf{x}_{k+1} - \mathbf{x}_{k+1,\text{ref}}\|_{\mathbf{Q}} + \|\mathbf{f}_k\|_{\mathbf{R}} \tag{6.6a}$$

$$\text{subject to} \quad \begin{aligned} f_{k,\min} \leq f_{k,z} \leq f_{k,\max} \\ -\mu f_{k,z} \leq \pm f_{k,x} \leq \mu f_{k,z} \\ -\mu f_{k,z} \leq \pm f_{k,y} \leq \mu f_{k,z} \\ \mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{f}_k \\ \mathbf{D}_k \mathbf{f}_k = 0 \end{aligned} \tag{6.6b}$$

with the state

$$\mathbf{x}_k = \begin{bmatrix} \boldsymbol{\Theta}_k \\ \mathbf{r}_k \\ \boldsymbol{\omega}_k \\ \mathbf{v}_k \end{bmatrix}, \tag{6.6c}$$

where $\boldsymbol{\Theta}$ is the robot's orientation, $\mathbf{r}$ is the CoM base position, $\boldsymbol{\omega}$ is the angular velocity, $\mathbf{v}$ is the linear velocity, μ is the friction coefficient, $\mathbf{A}$ and $\mathbf{B}$ are the dynamic matrices for state propagation, $\mathbf{D}_k$ is a force selection matrix (selecting forces that are not in contact to be equal to zero), and $\mathbf{Q} \in R^{12\times 12}$ and $\mathbf{R} \in R^{12\times 12}$ are diagonal positive semi-definite cost matrices, see [34] for further details. The joint torques, which are the input to the torque-controlled motors, are obtained using the forces $\mathbf{f}_k$ resulting from (6.6) as

$$\boldsymbol{\tau}_{i,k} = \mathbf{J}_{i,k}^{\top} \mathbf{R}_{b,i,k}^{w,\top} \left(-\mathbf{f}_{i,k}\right), \tag{6.7}$$

where $\mathbf{f}_{i,k} \in R^3$ is the force vector associated with leg i as subset of $\mathbf{f}_k$, and $\mathbf{R}_{b,i,k}^{w}$ is the rotation matrix from world to body frame of leg i at time step k.

6.2.3 Problem Definition and Contributions

In this letter, we implement model-based controllers and auto-tune their control parameters. Model-based control offers the advantages that the physics of a dynamical system are utilized to make informed decisions, less data are needed to improve the robot's performance, safety guarantees may be provided, and that the computational structure remains fixed. This control philosophy stands in contrast to black-box controllers and reduces the amount of required tuning to a few selected control parameters.

Problem A

Auto-tune the gains of the swing controller in (6.5), i.e., $\mathbf{K}_p$, $\mathbf{K}_d$. Section 6.3.1 addresses this problem by adjusting the controller calibration method in [93, 94] to a swing controller of a

legged robot. This is achieved by defining a training objective suited for a swing controller as well as deriving a motion model suited to model the swing motion of a legged robot.

Problem B

Auto-tune the cost function weights of the stance controller in (6.6), i.e., $\mathbf{Q}$, $\mathbf{R}$. Section 6.3.2 addresses this problem by adjusting the controller calibration method in [93, 94] to a stance controller of a legged robot. This is achieved by defining a training objective suited for a stance controller as well as deriving a motion model suited to model the stance phase of a legged robot.

Problem C

Develop concept for online computation of reference trajectories. Section 6.3.3 addresses this problem by continuously generating reference trajectories for continuous operation of a legged robot.

Problem D

Develop concept for adaptively refining reference trajectory to the current task and the current environment. Section 6.3.4 addresses this problem by adjusting the method in [93, 94] to calibrate parameters of reference trajectories. This is achieved by parameterizing the reference trajectories, defining a suitable training objective for the operation of the legged robot, and deriving a motion model for the robot's movements.

Finally, Section 9.4 presents the application of the proposed algorithms to a third-party high-fidelity simulator of the Unitree A1 in Gazebo. The high-fidelity simulator is provided by Unitree [137] and it includes contact dynamics, multi-body dynamics, sensor noise, etc.

6.3 Auto-Tuning Controller and Reference Trajectories of Legged Robot

The overall implementation of the auto-tuning method is as follows: (1) we generate a feasible motion plan for the first few footsteps using TO; (2) we use this initialization as part of the reference-generator function to estimate new trajectories without optimization; (3) the auto-tuning formulation is employed for both improved trajectory tracking (i.e., auto-tuning the swing or stance controller) and modifying the trajectories directly in order to minimize energy consumption, produce forces that minimize slippage, or satisfy a desired step clearance (i.e., step height) or forward progress (i.e., base velocity), see Fig. 6.1

6.3.1 Training Objectives for Auto-Tuning Swing Controller

For auto-tuning the swing controller in (6.5), we parametrize the gains, i.e., $\mathbf{K}_p = \mathbf{K}_p(\boldsymbol{\theta})$, $\mathbf{K}_d = \mathbf{K}_d(\boldsymbol{\theta})$, and we use the dynamical model using the notation in (6.1) for leg i,

$$\mathbf{x}_{i,k} = \begin{bmatrix} \mathbf{q}_{i,k} \\ \dot{\mathbf{q}}_{i,k} \end{bmatrix} \tag{6.8a}$$

$$\mathbf{dyn}(\mathbf{x}_{i,k}, \boldsymbol{\tau}_{i,k}) = \begin{bmatrix} \mathbf{q}_{i,k} + dt\dot{\mathbf{q}}_{i,k} + dt^2\ddot{\mathbf{q}}_{i,k} \\ \dot{\mathbf{q}}_{i,k} + dt\ddot{\mathbf{q}}_{i,k} \end{bmatrix} \tag{6.8b}$$

$$\ddot{\mathbf{q}}_i = \mathbf{M}_i(\mathbf{q}_i)^{-1}(\boldsymbol{\tau}_i - \mathbf{V}_i(\mathbf{q}_i, \dot{\mathbf{q}}_i) - \mathbf{G}_i(\mathbf{q}_i)), \tag{6.8c}$$

where $\mathbf{M}_i \in R^{3\times 3}$ is the mass matrix in the joint space, and $\ddot{\boldsymbol{q}}_i \in R^{3\times 3}$ is the current joint acceleration of foot i. The training objective (6.2) is defined to improve reference trajectory tracking as

$$\mathbf{y}_k = \begin{bmatrix} \mathbf{p}^{ref}_{i,k-N|k} \\ \mathbf{v}^{ref}_{i,k-N|k} \end{bmatrix}, \quad \mathbf{h}(\boldsymbol{\theta}) = \begin{bmatrix} \mathbf{p}_{i,k-N|k} \\ \mathbf{v}_{i,k-N|k} \end{bmatrix}, \tag{6.8d}$$

where $\mathbf{p}^{ref}_{i,k-N|k}$ and $\mathbf{v}^{ref}_{i,k-N|k}$ are positions and velocities of foot i from time step $k-N$ through k as in (6.10) and (6.11).

The model mismatch $\mathbf{w}_{k-N|k} \in R^6$ for the swing controller is computed using (6.8) and the measured states for the past swing execution. The UKF-based control parameter adaptation method uses (6.5) and (6.8) to "simulate" an execution of a swing using the gains defined by the sigma points, $\mathbf{K}_p(\boldsymbol{\theta}^{sp,j})$ and $\mathbf{K}_d(\boldsymbol{\theta}^{sp,j})$, considering the model mismatch, $\mathbf{w}_{k-N|k}$. Note that $\mathbf{p}^b_{i,k}$ and $\mathbf{v}^b_{i,k}$ in (6.5) can be calculated using the robot forward kinematics, which uses $\mathbf{q}_{i,k}$ as input, and the footstep Jacobian by $\mathbf{v}^b_{i,k} = \mathbf{J}_i(\mathbf{q}_{i,k})\dot{\mathbf{q}}_{i,k}$.

6.3.2 Training Objectives for Auto-Tuning Stance Controller

For auto-tuning the stance controller cost function weights in (6.6), i.e., $\mathbf{Q} = \mathbf{Q}(\boldsymbol{\theta})$, $\mathbf{R} = \mathbf{R}(\boldsymbol{\theta})$, we use the dynamical model (6.1) as

$$\mathbf{dyn}(\mathbf{x}_k, \mathbf{f}_k) = \begin{bmatrix} \boldsymbol{\Theta}_k + \mathbf{R}^w_b \boldsymbol{\omega}_k \\ \mathbf{r}_k + dt\mathbf{v}_k \\ \boldsymbol{\omega}_k + dt(\sum_{i=1}^4 \hat{\mathbf{I}}^{-1} \left[\mathbf{p}^b_{i,k}\right]_\times \mathbf{f}^i_k) \\ \mathbf{v_k} + dt(\sum_{i=1}^4 \frac{\mathbf{f}^i_k}{m} + \mathbf{g}) \end{bmatrix} \tag{6.9a}$$

with the state $\mathbf{x}_k$ defined as in (6.6), where $\hat{\mathbf{I}}$ represents the inertia tensor in the world frame ($\times$ indicates a skew matrix) [34], $\mathbf{R}^w_b$ is the rotation matrix from world to body frame, and $\mathbf{g}$ is the gravity vector. The training objective in (6.2) is defined to improve reference trajectory tracking as

$$\mathbf{y}_k = \left[\mathbf{x}^{ref}_{k-N|k}\right], \quad \mathbf{h}(\boldsymbol{\theta}) = \left[\mathbf{x}_{k-N|k}\right] \tag{6.9b}$$

Here, too, the UKF-based control parameter adaptation method uses (6.6) and (6.9) to "simulate" the MPC using the cost function weights defined by the sigma points $\mathbf{Q}(\boldsymbol{\theta}^{sp,j})$, considering the model mismatch $\mathbf{w}_{k-N|k}$.

6.3.3 Generating Reference Trajectories

To ensure kinematic and dynamically feasible motion for the first few footsteps, we apply the methods proposed in [143], which uses trajectory optimization.

6.3.3.1 Reference Trajectory for Feet

In order to continuously generate new reference trajectories, we use

$$\mathbf{p}_{i,k}^{ref} = \mathbf{p}_{i,k-N}^{ref} + \mathbf{v}_{i,k-N}^{ref}\Delta T \tag{6.10a}$$

$$\mathbf{p}_{i,k}^{ref} = \begin{bmatrix} p_{i,k}^{x,ref} \\ p_{i,k}^{y,ref} \\ p_{i,k}^{z,ref} \end{bmatrix}, \quad \mathbf{v}_{i,k}^{ref} = \begin{bmatrix} v_{i,k}^{x,ref} \\ v_{i,k}^{y,ref} \\ v_{i,k}^{z,ref} \end{bmatrix} \tag{6.10b}$$

where $\mathbf{p}_{i,k-N}^{ref}$ denotes the position of foot i on the ground at time step $k-N$, $\mathbf{p}_{i,k}^{ref}$ is the next position of foot i on the ground at time step k, which are determined by a desired CoM body velocity, $\mathbf{v}_{i,k}^{ref}$, and the phase time ΔT, i.e., the time that the foot is in stance or swing phase. Throughout, we assume $v_{i,k}^{z,ref}=0$.

Then, we use $s(t) = 2\pi\frac{N-k+t}{N}$ with $t=k-N, k-N+1, ..., k$ to create cycloidal footstep reference trajectories in between the ground positions in (6.10) with

$$p_{i,t}^{x,ref} = p_{i,k-N}^{x,ref} + \left(p_{i,k}^{x,ref} - p_{i,k-N}^{x,ref}\right)\frac{s(t)-\sin(s(t))}{2\pi} \tag{6.11a}$$

$$p_{i,t}^{y,ref} = p_{i,k-N}^{y,ref} + \left(p_{i,k}^{y,ref} - p_{i,k-N}^{y,ref}\right)\frac{s(t)-\sin(s(t))}{2\pi} \tag{6.11b}$$

and

$$p_{i,t}^{z,ref} = \begin{cases} p_{i,k-N}^{z,ref} + p_{\max}^{z}\frac{1-\cos(s(t))}{2} \\ \quad if\ s(t) \leq \pi \\ p_{i,k}^{z,ref} + \left(p_{\max}^{z} + p_{i,k-N}^{z,ref} - p_{i,k}^{z,ref}\right)\frac{1-\cos(s(t))}{2} \\ \quad if\ s(t) > \pi \end{cases} \tag{6.11c}$$

where $p_{\max}^{z}$ is the step clearance/apex height of the swing trajectory, i.e., the maximum p^z component of the footstep in swing. If the foot is in stance, $p_{\max}^{z}=0$.

6.3.3.2 Reference Trajectory for Center of Mass

The reference trajectory of the body's CoM is specified by

$$\begin{bmatrix} \mathbf{r}_k^{ref} \\ \boldsymbol{\Theta}_k^{ref} \end{bmatrix} = \begin{bmatrix} \mathbf{r}_{k-N}^{ref} \\ \boldsymbol{\Theta}_{k-N}^{ref} \end{bmatrix} + \begin{bmatrix} \mathbf{v}_{k-N}^{ref} \\ \boldsymbol{\omega}_{k-N}^{ref} \end{bmatrix} \Delta T \tag{6.12}$$

with the current body position, $\mathbf{r}_{k-N}^{ref} \in R^3$, and orientation, $\boldsymbol{\Theta}_{k-N}^{ref} \in R^3$, and $\mathbf{r}_k^{ref}, \boldsymbol{\Theta}_k^{ref}$ are composed of the next desired body position, and $\mathbf{v}_{k-N}^{ref}, \boldsymbol{\omega}_{k-N}^{ref}$ is the desired velocity (linear and angular) over the phase duration ΔT. Here, we use $\bar{s}(t) = \frac{N-k+t}{N}$ with $t = k-N, k-N+1$, ..., k to linearly interpolate between current and desired states with

$$\begin{bmatrix} \mathbf{r}_t^{ref} \\ \boldsymbol{\Theta}_t^{ref} \end{bmatrix} = \bar{s}(t) \begin{bmatrix} \mathbf{r}_k^{ref} \\ \boldsymbol{\Theta}_k^{ref} \end{bmatrix} + (1 - \bar{s}(t)) \begin{bmatrix} \mathbf{r}_{k-N}^{ref} \\ \boldsymbol{\Theta}_{k-N}^{ref} \end{bmatrix}.$$

6.3.4 Training Objectives for Auto-Tuning Reference Trajectory

For auto-tuning the reference trajectory, we use the same states and dynamic model as for the swing controller in (6.8). The main difference is in the implementation of the training objective and control parameters, i.e., we do not auto-tune the $\mathbf{K}_p$ and $\mathbf{K}_d$ gain matrices. Instead, we tune two parameters often desirable for legged robots, which are to achieve a desired step clearance in order to step over an obstacle and reach a desired velocity. Hence for leg i, we parametrize the step clearance and the desired forward velocity,

$$p^z_{\text{max}} = \theta_z, \quad v_{i,k}^{x,\text{ref}} = \theta_v, \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_z \\ \theta_v \end{bmatrix}$$

and the training objective for adapting such parameters is

$$\mathbf{y}_k = \begin{bmatrix} p^z_{\text{des}} \\ v^x_{\text{des}} \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{h}(\boldsymbol{\theta}) = \begin{bmatrix} h_z(\boldsymbol{\theta}) \\ h_v(\boldsymbol{\theta}) \\ h_f(\boldsymbol{\theta}) \\ h_e(\boldsymbol{\theta}) \end{bmatrix}, \tag{6.13}$$

where $h_z(\boldsymbol{\theta})$ is the achieved step clearance, $h_v(\boldsymbol{\theta})$ is the achieved velocity, $h_f(\boldsymbol{\theta})$ computes lateral forces of the robot and is used to reduce slippage, and $h_e(\boldsymbol{\theta})$ is the energy consumption. Note that p^z_{des} measures the achieved step clearance in closed loop, whereas p^z_{max} is the control parameter that defines the reference trajectory. Thus, p^z_{des} and p^z_{max} may be different, e.g., due to a model mismatch of the physical robot and the dynamical system model.

6.3.4.1 Step clearance optimization $h_z(\boldsymbol{\theta})$

To propagate the states for optimizing the step clearance, we use the same method as in Section 6.3.1. However, here we "simulate" the various reference trajectories defined using the sigma points for the swing leg, where we do not change the controller gains of $\mathbf{K}_p$, $\mathbf{K}_d$. Hence, we modify the reference trajectory of the footstep using the reference generator function described in Section 6.3.3. Thus, $h_z(\boldsymbol{\theta}) = max\,(p^z_{k-N|k})$, i.e., the maximum value of the z component of the footstep trajectory (after propagation with $\mathbf{w}_{k-N|k}$).

6.3.4.2 Forward progress optimization $h_v(\boldsymbol{\theta})$

For forward progress, we propagate the CoM position using the desired velocity, v^x_{des}, with (6.12), where the initial state is the actual state.

6.3.4.3 Slippage optimization $h_f(\boldsymbol{\theta})$

Representing slippage of the foot first requires the propagation of ground reaction forces along the trajectory, which can be achieved by rearranging the MPC state-space equations in (6.6),

$$\mathbf{f}_k = \mathbf{B}^+(\mathbf{x}_{k+1} - \mathbf{A}\mathbf{x}_k), \tag{6.14}$$

where $\mathbf{B}^+$ is the Moore-Penrose inverse and $\mathbf{x}_k$ is propagated using the sigma points. We chose (6.14) due to its similarity to (6.6), but other slippage formulations are possible, too,

e.g., as in [13]. If the foot is not in contact, $\mathbf{f}_k = \mathbf{0}$. Then, the foot slippage is computed using similar ideas to friction cone constraints (see, e.g., [143]) as

$$h_f(\boldsymbol{\theta}) = \sum_{i=1}^{4} \frac{\sqrt{f_{i,\text{avg}}^x + f_{i,\text{avg}}^y}}{f_{i,\text{avg}}^z},$$

$$\begin{bmatrix} f_{i,\text{avg}}^x \\ f_{i,\text{avg}}^y \\ f_{i,\text{avg}}^z \end{bmatrix} = \frac{1}{N} \sum_{t=k-N}^{k} \begin{bmatrix} |f_{i,t}^x| \\ |f_{i,t}^y| \\ |f_{i,t}^z| \end{bmatrix}.$$

6.3.4.4 Energy consumption optimization $h_e(\boldsymbol{\theta})$

The energy consumption is included as it may be desirable to find the best balance between maximizing step clearance and forward progress, while simultaneously minimizing the energy consumption required. The energy consumption can be computed using the joint velocity $\dot{\mathbf{q}}$, torque $\boldsymbol{\tau}_t$, and phase time ΔT, as

$$h_e(\boldsymbol{\theta}) = \sum_{t=k-N}^{k} |\dot{\mathbf{q}}_t|^\top |\boldsymbol{\tau}_t| \Delta T.$$

To ensure we do not violate kinematic or dynamic constraints as we simulate new trajectories using $\boldsymbol{\theta}$, we can use $\left|\mathbf{R}_w^b \left[\mathbf{p}_{i,k} - \mathbf{r}_k\right] - \overline{\mathbf{p}}_i^b\right| < \mathbf{b}$, where $\mathbf{R}_w^b$ is the rotation matrix from base to world frame, and $\overline{\mathbf{p}}_i^b$ is the nominal footstep position centered in the kinematic bounding box specified by $\mathbf{b} \in R^3$. If the bounding box constraint does not hold, we can impose a cost for that particular simulation guiding the auto-tuner away from such control parameters.

6.4 Results using Physics-Based Simulator

We implemented the auto-tuning algorithms for the Unitree A1 robot [137] within a realistic simulation using Gazebo with the bullet physics engine. The Unitree A1 simulator is made available by the robot manufacturer. The simulation considers not only localization noise but also friction within each torque-controlled motor, where torque commands are sent using the

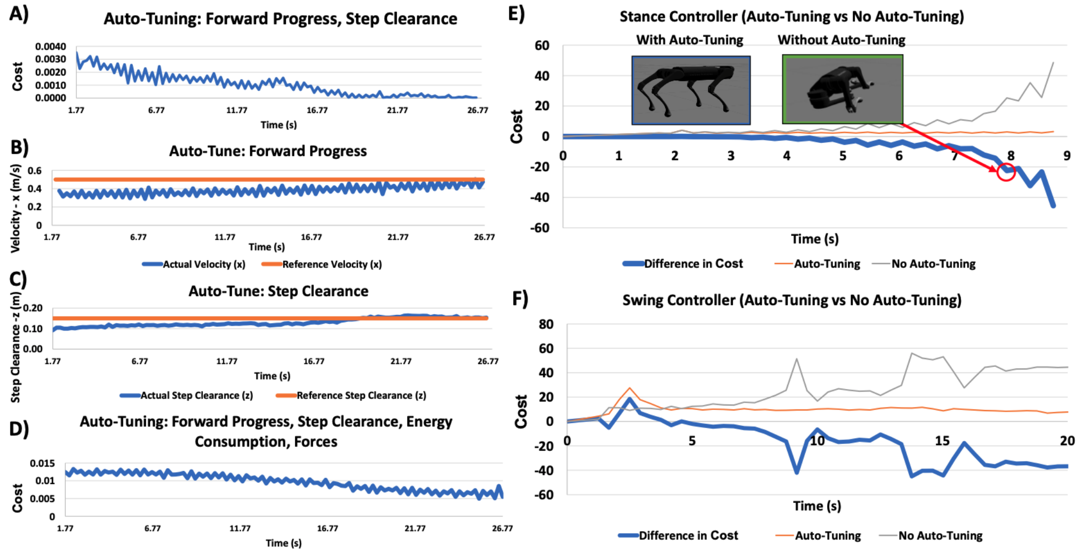


Figure 6.2: **Auto-Tuning results.** The auto-tuning method successfully calibrated control parameters that generate reference trajectories (**A-D**), and controller gains (**E-F**). To make this evaluation, we used the cost as calculated using (6.2). The cost decreases (**A** and **D**) as the control parameters (e.g., forward progress and step clearance) get to their desired reference values. In **E-F**, we show the difference between the cost when not auto-tuning with the cost when auto-tuning.

robotic operating system (ROS). Further, the Gazebo simulation environment is executed in a parallel thread, and hence the controller and the auto-tuning algorithm need to be executed during run-time of the simulation. Hence, being able to execute the applied algorithms using this high-fidelity simulator indicates their applicability to the physical robot. The A1 Unitree is a 12.7 kg quadruped robot with 3 degrees of freedom per leg. For obtaining an initial trajectory as in [143], we use the nonlinear programming solver IPOPT [140]. Optimization of the MPC stance controller was done using the quadratic programming solver qpOASES [45]. Constraints and overall problem formulation were setup using the CasADi [4] software. The phase time was chosen as 0.2 s, which is also the update frequency of the parameters. The swing and stance controller are executed at sampling frequency at $dt = 0.01$ s, and $N_{MPC} = 5$ in (6.6).

6.4.1 Auto-Tuning for Optimizing Step Clearance and Forward Progress

The first test case is to auto-tune reference trajectories aiming at providing a desired forward progress and step clearance. Here, we consider only the first and second element in (6.13), where we use $p^z_{des} = 0.15$ m and $v^x_{des} = 0.45$ m/s. The results are shown in Fig. 6.2-**A)**, **B)**, **C)**. In **A)**, we show the cost defined as $||\mathbf{y}_k - \mathbf{h}(\boldsymbol{\theta})||_{C_y^{-1}} = \| \begin{bmatrix} p^z_{des} - h_z(\boldsymbol{\theta}) & v^x_{des} - h_v(\boldsymbol{\theta}) \end{bmatrix} \|_{C_y^{-1}}$ as in (6.2). Thus, as the cost decreases, the closer the actual step clearance and forward progress get to the desired value. We decompose the results in **A)** and show how the actual velocity reaches the desired velocity in **B)**, and how the actual step clearance reaches the desired step clearance in **C)**.

6.4.2 Auto-Tuning for Optimizing Step Clearance, Forward Progress, Slippage, and Energy Consumption

In **D)**, we show the cost when setting a desired step clearance and forward progress with $p^z_{des} = 0.1$ m and $v^x_{des} = 0.4$ m/s, while minimizing energy consumption and foot slippage as in (6.13). The cost decreases quickly within a few time steps, which indicates that the auto-tuning method was able to find a good balance between reaching its desired step clearance and forward progress, while ensuring a minimization of both energy consumption and foot slippage.

6.4.3 Auto-Tuning Stance Controller

Next, we demonstrate the auto-tuning method to calibrate the gains of the stance controller. For simplicity, here we auto-tune only the diagonal elements of $\mathbf{Q} = \mathbf{Q}(\boldsymbol{\theta})$ in (6.6), but $\mathbf{R}$ can similarly be tuned. We initialize $\mathbf{Q} = \text{diag}([1, 1, Q_3, Q_4, Q_5, Q_6, 1000, 1, 1, 1, 1, 1])$, where Q_3, Q_4, Q_5, Q_6 are initialized to 300, and will be further calibrated, i.e., $\mathbf{Q}(\boldsymbol{\theta})$ with $\boldsymbol{\theta} \in R^4$. The cost function is tuned online for 20 s of locomotion using trot gait. In Fig. 6.2-**E)**, we present the cost difference between the auto-tuning case and the no auto-tuning

case. The difference decreases over time, which shows that the auto-tuning improves the cost using auto-tuning decreased significantly compared to the cost without using auto-tuning. Additionally, without auto-tuning, the robot eventually falls after about 8 seconds of locomotion, because the initial gains of the MPC controller have not been properly hand-tuned. Note that some hand-tuning was necessary to ensure initial gains that at least allow the robot to start moving a few steps without falling. After auto-tuning the stance controller for 20 s, $Q_3 = 5245.53$, $Q_4 = 1172.34$, $Q_5 = 662.45$, and $Q_6 = 1172.25$. Further, we applied the tuned controller along with the tuned swing controller in Section 6.4.4 and demonstrated robust locomotion on uneven terrain, see Fig. 6.3.

6.4.4 Auto-Tuning Swing Controller

Lastly, we apply the auto-tuning method to calibrate the gains of the swing controller, $\mathbf{K}_p$ and $\mathbf{K}_d$. We choose $\boldsymbol{\theta} \in R^6$ consisting of the three diagonal elements of the proportional gain matrix, $\mathbf{K}_p$, and the three diagonal elements of the derivative gain matrix, $\mathbf{K}_d$. We initialized $\mathbf{K}_d = \text{diag}([0.1, 10, 10])$ and $\mathbf{K}_p = \text{diag}([150.11, 16.11, 10.11])$ to be equal to the leg's natural frequency using the inverted pendulum model multiplied by the operational mass matrix, see [34] for more detail. Fig. 6.2-**F)**, shows the difference of the cost when not auto-tuning with the cost when auto-tuning for over 20 seconds of locomotion. The cost curve follows a downward trend, demonstrating that the cost when auto-tuning is smaller than without auto-tuning. The gains using auto-tuning outperformed the gain selection without auto-tuning, which was based on the current natural frequency calculation and was shown to perform well in [34]. The final gains after tuning was $\mathbf{K}_d = \text{diag}([13.82, 12.42, 17.80])$ and $\mathbf{K}_p = \text{diag}([166.26, 22.75, 14.03])$.

Fig. 6.3 shows the robot walking on uneven ground using auto-tuning of the swing controller gains and the stance controller cost function.

While this letter presents one simulation run, we have obtained similar results for repeated test cases, which is expected as the auto-tuning formulation is robust to noise/disturbances

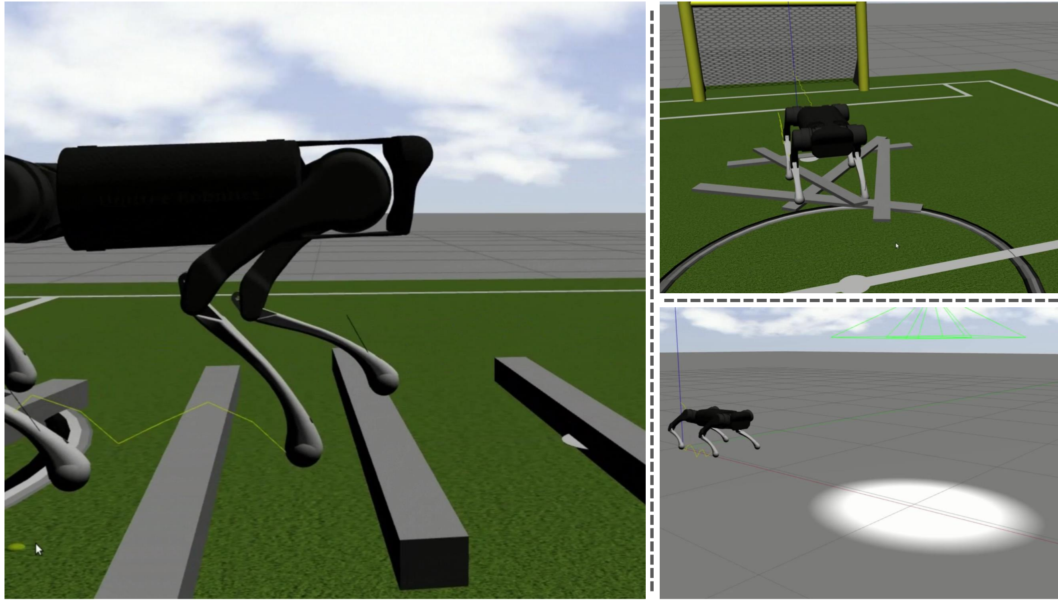


Figure 6.3: **Results for robust locomotion on uneven terrain.** After auto-tuning the stance and swing controllers, we employ the robot on several test cases to demonstrate robust locomotion. We show the robot traversing over and on large beams and planks in the left and top right with a trot gait, respectively, and demonstrate a successful jumping gait in the bottom right. The robot is not aware of the obstacles and must overcome them by using the auto-tuned controllers.

due to its filter-based design.

6.4.5 Computation Times

The computation times of auto-tuning the swing and stance controllers, as well as the reference trajectory are shown in Table 6.1. Table 6.1 lists the maximum, minimum, and median computation times of the auto-tuning algorithm for tuning the gains of the swing controller, the MPC weights of the stance controllers and the parameters of the reference trajectory (Reference Traj. $\mathbf{A}-\mathbf{D}$ in Table 6.1 refers to the training objectives specified by $\mathbf{A}-\mathbf{D}$ in Fig. 6.2). The computation times were obtained while running on a laptop using 4 CPU cores (Intel core i7-8850H CPU at 2.60 Ghz) with a Quadro P3200 GPU. Further, the controller

Table 6.1: Auto-Tuning Computation Times

Auto-Tune	Min. Time	Median Time	Max. Time
Reference Traj. **A)**	0.0108s	0.0143s	0.0370s
Reference Traj. **B)**	0.0072s	0.0089s	0.0121s
Reference Traj. **C)**	0.0086s	0.0093s	0.0235s
Reference Traj. **D)**	0.0632s	0.0662s	0.0667s
Stance Controller **E)**	0.0409s	0.0465s	0.0807s
Swing Controller **F)**	0.0078s	0.0569s	0.1006s

ran in parallel with a Gazebo simulation, which requires more computational resources than an implementation on hardware would. As the computation times are below the update rate of the auto-tuner updating at the end of the phase time with 0.2 s, we can conclude that the algorithms in this letter can be executed in real time on comparable computational resources.

The computation times scale linearly with the amount of parameters to be tuned. Although the applied algorithms can be executed online, depending on user specifications, there is the option to execute the algorithms offline or in parallel. E.g., the auto-tuner for the swing and stance controllers can update the gains on a different thread, which runs in parallel to the low-level/high-level controllers of the robot. Additionally, the methods could also be implemented offline if computational resources are limited.

6.5 Conclusions

In this work we successfully demonstrated robust locomotion. An auto-tuning method was implemented, which is based on an unscented Kalman filter formulation, to calibrate the gains of the swing controller and the cost function weights of a stance controller. We also

demonstrated that the method can be applied for directly auto-tuning the reference trajectory, i.e., reference trajectories are calibrated to make the robot achieve a desired forward progress and step clearance, while also minimizing energy consumption and foot slippage. We showed that the method can be easily generalized to consider diverse control parameter by demonstrating the auto-tuning on both controller gains and on physically meaningful parameters of a reference trajectory. Future work may include experimental validation on the robot platform and extending the method to consider tuning for optimal footstep timings of various gait sequences, consider dynamic environments that require the auto-tuner to calibrate changing step heights and base velocities, and also adapt the aggressiveness of the tuning through the unscented Kalman filter's covariance matrices based the robot's environment or desired task. Future work may also include more sudden changes in the robot's motion, e.g., by leveraging the "prediction model" of the Kalman filter in addition to the "measurement model" used in this letter.

CHAPTER 7

Adaptive Force Controller

Forces and torques are exchanged in nearly every type of interaction between robots and their environments. From robotic manipulators that perform peg-hole-insertion tasks [132], pushing/pulling objects [40, 102], to the ground reaction wrenches generated by foot contacts for robotic climbing [127]. Thus, state-of-the-art algorithms for locomotion and manipulation estimate desired wrench profiles which can achieve stable balancing of an object [125] or satisfy kinematic and dynamic constraints [143]. Additionally, current controllers typically use forces as control variables, optimizing them to either stabilize the Center of Mass trajectory of the robot's base [108] or achieve compliance control during manipulation [79]. In all the above cases, the success of the robot's motion depends on how well a controller can comply with external perturbation and ideally, track the desired output wrench profiles.

In this paper, our objective is to formulate an auto-calibrating admittance controller for tracking the wrench profile using F/T sensors in a closed-loop fashion. The main contribution of this paper is to demonstrate this using an Unscented Kalman Filter (UKF) to track wrench profiles that includes the additional torque due to rotational friction, while considering training objectives that facilitate controller robustness and adaptability during online operation for multi-point contact grippers.

Additionally, we validate our controller on hardware (Figure. 7.1) that consists of underactuated grippers, and demonstrate tracking of reference wrench profiles for various tasks, including wall climbing, grasping of objects, and locomotion.

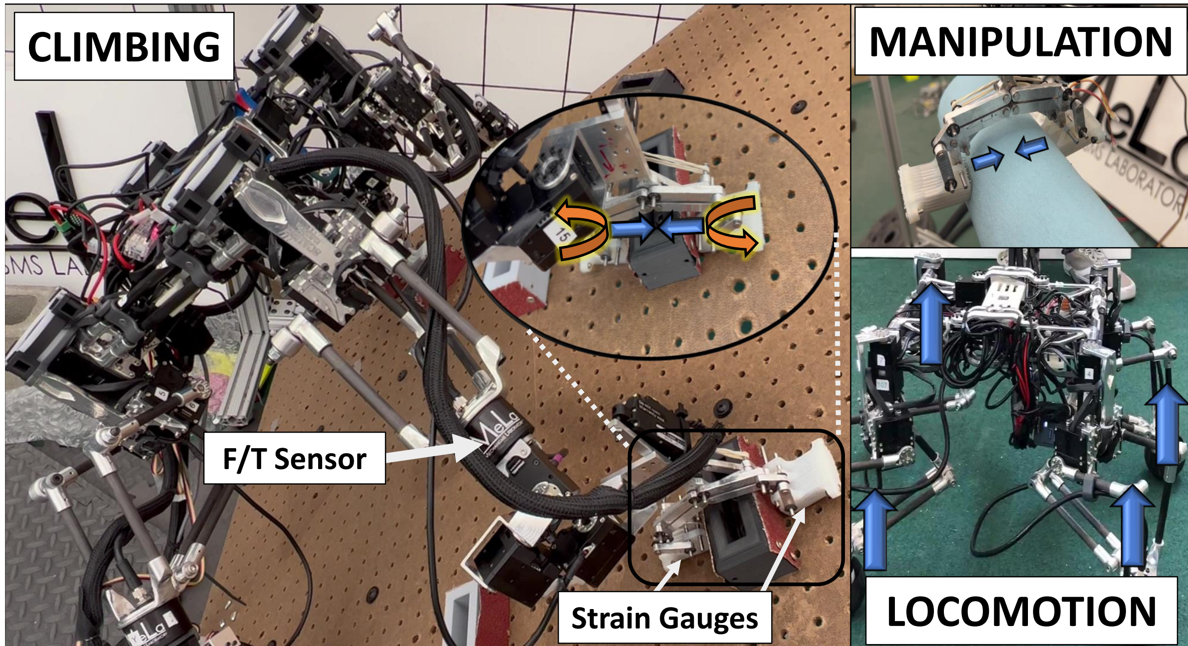


Figure 7.1: **Hardware validation tests.** Auto-calibrating admittance control for tracking wrenches for climbing, manipulation, and locomotion tasks. The blue arrows indicate forces, while the orange arrows indicate torques. We also indicate the Force/Torque (F/T) sensor at the wrist, and strain gauges located on the finger, measuring the finger compression force

7.1 Admittance control formulation

The fundamental nature of an admittance controller is that it converts a force input into a motion defined by a change in position. Admittance control may be necessary for robots that have positioned-controlled motors but still need to simulate the properties of compliance when interacting with its environment. For clarity in wording, we will call the end-effector as being the fingertip, and the wrist as being the base of the robot's gripper where the fingertips are attached. Our admittance control formulation can be described by:

$$\ddot{\mathbf{x}} = \mathbf{M}_d^{-1}(-\mathbf{D}_d\dot{\mathbf{x}} - \mathbf{K}_d(\mathbf{x} - \mathbf{x}_{ref}) + \mathbf{K}_f(\mathcal{W}_{meas} - \mathcal{W}_{ref})) \tag{7.1}$$

where $\mathcal{W}_{meas} \in R^{6k}$ is the current wrench and $\mathcal{W}_{ref} \in R^{6k}$ is the desired reference wrench for fingertip or contact point k, 6 represents the x, y, and z components of force, $\mathbf{f} \in R^{3k}$,

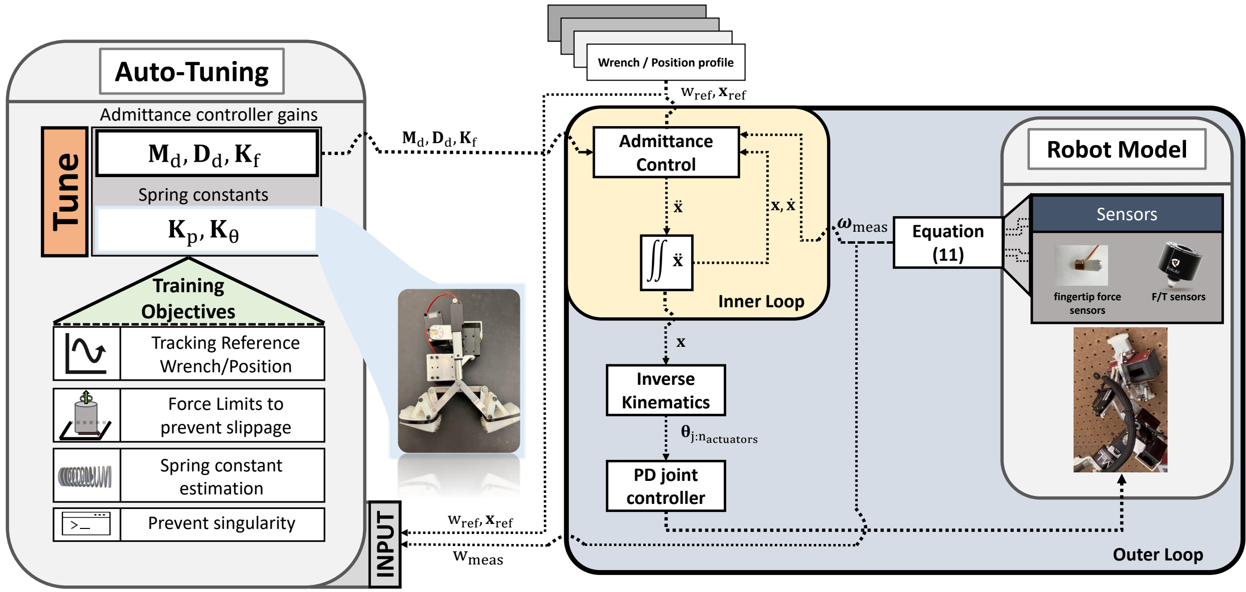


Figure 7.2: **Auto-calibrating admittance control framework.** This figure shows the overall control architecture. The admittance controller takes as input the wrench and position profile (represented by $\mathcal{W}_{ref}$ and $\mathbf{x}_{ref}$) or the forces $\mathbf{f}_{ref}$ (if our MPC is used). The current wrenches are received using the relationship provided by equation (11) which requires force/torque sensor information, and also the current control inputs ($\mathbf{x}$ and $\dot{\mathbf{x}}$). The output of the admittance controller is the acceleration ($\ddot{\mathbf{x}}$), where its integration yields the control inputs for the next timestep (considered the inner loop). Since we use position-controlled motors, we use $\mathbf{x}$ as input to our inverse kinematics, which provides the joint motor angles ($\boldsymbol{\theta}_{1:n_{actuators}}$). A PD joint controller is used to track these angles (considered the outer loop). Lastly, the auto-tuning framework takes as input the reference and current wrenches and outputs new gains for the admittance controller ($\mathbf{M}_d$, $\mathbf{D}_d$, and $\mathbf{K}_f$). As the auto-tuning method makes use of the robot model (dependent on spring constants $\mathbf{K}_p$, and $\mathbf{K}_\theta$), we make use of the current wrenches to continually update these spring constants to a more accurate estimate as part of the auto-tuning process.

and torque, $\boldsymbol{\tau} \in R^{3k}$, with $\mathcal{W} = [\mathbf{f}^\top, \boldsymbol{\tau}^\top]^\top$. $\mathbf{M}_d \in R^{6k\times 6k}$, $\mathbf{D}_d \in R^{6k\times 6k}$, $\mathbf{K}_d \in R^{6k\times 6k}$, are the diagonal gain matrices that can be described as the desired mass, which must be

invertible, damping, and spring coefficients respectively. $\mathbf{K}_f \in R^{6k\times 6k}$ does not have any physical meaning, but may be tuned depending on the desired sensitivity to changes in wrench. $\dot{\mathbf{x}}$, and $\mathbf{x}$ are the control outputs, where $\mathbf{x} = [\mathbf{p}^\top, \boldsymbol{\Theta}^\top]^\top$ represents the fingertip position, $\mathbf{p} \in R^{3k}$, in x, y and z, and orientation, $\boldsymbol{\Theta} \in R^{3k}$, in x, y, and z, which are solved by integrating $\ddot{\mathbf{x}}$ using Euler discretization. $\mathbf{x}_{ref}$ is a desired position and orientation from a reference trajectory. Note, that subtracting Euler angles, as required by the $\mathbf{x} - \mathbf{x}_{ref}$ term, we can employ the techniques found in [21].

To estimate the current wrench, $\mathcal{W}_{meas}$ (as used in (7.1)), we must first derive the relationship between wrenches of the robot's wrist and the wrenches of each fingertip of the gripper (frames defined in Figure 7.3):

$$\mathcal{W}_k^G = \begin{pmatrix} \mathbf{f}_k^G \\ \mathbf{p}_k^G \times \mathbf{f}_k^G \end{pmatrix} + \begin{pmatrix} \mathbf{0}^3 \\ \boldsymbol{\tau}_k^G \end{pmatrix} \tag{7.2}$$

where $\mathcal{W}_k^G$ is the fingertip wrench of fingertip k relative to the gripper frame $\{\mathbf{G}\}$, $\mathbf{p}_k^G$ is the position of the fingertip, $\mathbf{f}_k^G$ is the reaction force, and $\boldsymbol{\tau}_k^G$ is the additional torque due to patch contact at the fingertip relative to the gripper frame. We assume gripper and fingertip frames are the same, or $\{\mathbf{G}\} = \{\mathbf{F}\}$ in Figure. 7.3.

When the fingertips are in contact with the environment, we will assume mathematically that the force/torque at the wrist of the gripper and the sum of forces/torques received from the fingertips are equal. Thus, we have:

$$\mathbf{f}^G = \sum_{k=1}^{n_f} \mathbf{f}_k^G \tag{7.3}$$

$$\boldsymbol{\tau}^G = \sum_{k=1}^{n_f} (\mathbf{p}_k^G \times \mathbf{f}_k^G) + \sum_{k=1}^{n_f} \boldsymbol{\tau}_k^G \tag{7.4}$$

where $\mathbf{f}^G$ and $\boldsymbol{\tau}^G$ are the forces and torques at the wrist, and n_f is the number of fingertips. In our case, we have a two-finger gripper ($n_f = 2$), which would yield a total of 6 equations and potentially 18 variables.

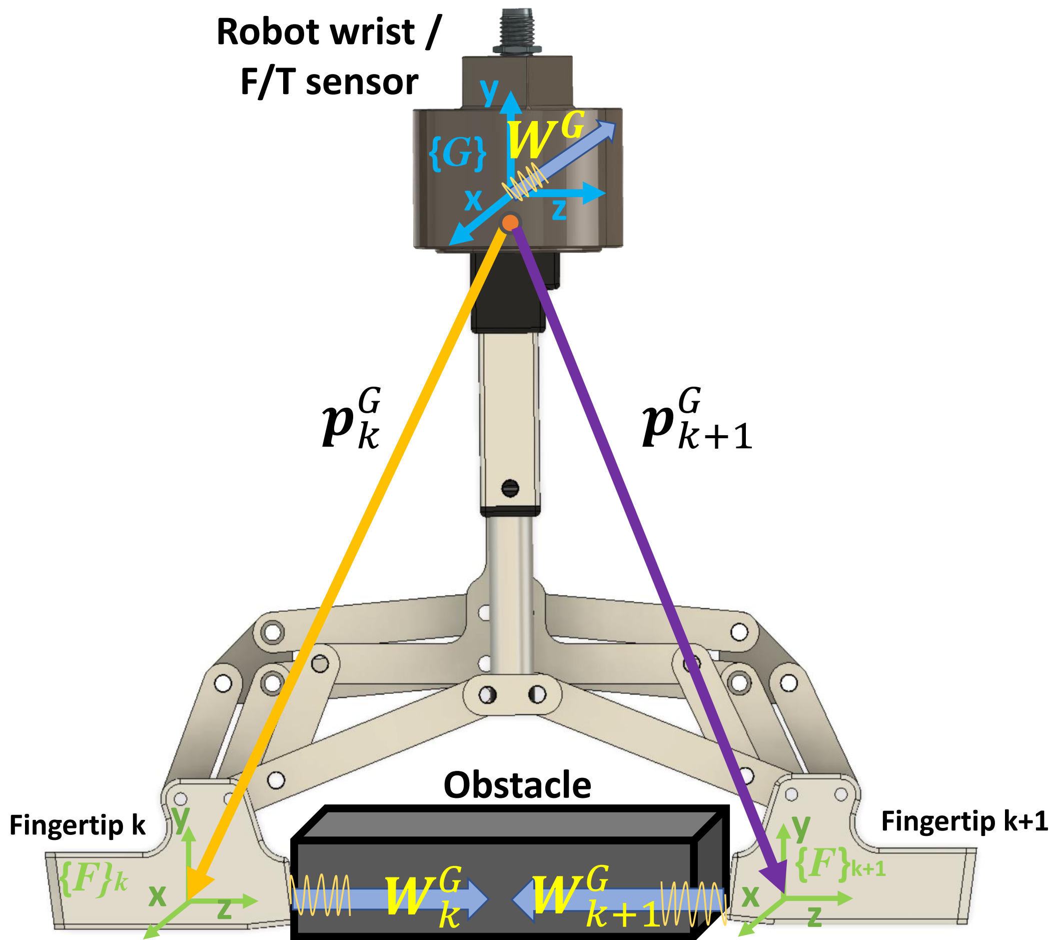


Figure 7.3: **Frame definitions.** Here we show the general frame definitions for the variables used in the admittance controller. Our controller will operate in the local gripper frame $\{\mathbf{G}\}$, which is considered the robot's wrist (or the location of the Force/Torque or F/T measurements). The gripper may have k number of end-effectors or fingertips, denoted by frame $\{\mathbf{F}\}$. Fingertip positions, $\mathbf{p}_k^G$, are all with respect to the gripper frame $\{\mathbf{G}\}$. Similar notations are given to the fingertip wrenches $\mathcal{W}_k^G$ and wrenches at the wrist $\mathcal{W}^G$. Note that to more easily solve for equations, we choose frame $\{\mathbf{F}\}$ to be the same orientation as frame $\{\mathbf{G}\}$.

In this paper, we assume that when the end-effector comes into contact with its environment, some non-negligible amount of deformation occurs between the end-effector material

and its environment, resulting in a contact 'patch', which we assume is circular. Thus, the effect of deformation when adapted to a typical rigid body point contact model is an additional torque ($\boldsymbol{\tau}^n \in R^3$) that acts about the contact normal [72], with Coulomb friction acting on each point of the patch. Using the derivation as done in [72], the norm of this torque is bounded by $|\boldsymbol{\tau}^n| \leq \boldsymbol{\lambda}\mathbf{f}^n$, where $\mathbf{f}^n \in R^3$ is the normal force and $\boldsymbol{\lambda} > 0$ is the friction coefficient due to rotation. Throughout this paper, we will assume that the normal force is in the same direction as the z component of fingertip k in the gripper frame $\{\mathbf{G}\}$ (see Figure. 7.3) or τ^n=$\tau^G_{k,z}$ and $f^n = f^G_{k,z}$ [72].

We also assume access to force/torque sensors at the wrist and 1-axis strain gauges on each fingertip measuring the z component of the force, which we approximate is along the same direction as the surface normal. Thus, we can directly measure f^G_x,f^G_y,$f^G_{1,z}$, $f^G_{2,z}$,τ^G_x,τ^G_y, and τ^G_z. A limitation of this work is that we don't directly measure all components of the wrench for each fingertip, and thus, need to estimate these wrenches and also impose additional constraints for which components we can independently control. For one, we have an underactuated gripper, thus, in our case, we cannot independently control both fingertip positions in the x, and y directions but can for z using the robot's arm and by closing/opening the gripper. Second, we impose constraints in our controller, where we assume control of each rotational axis for torque for both fingertips simultaneously. In other words, taking the above gripper configuration and constraints into account, we have that $f^G_{1,x} = f^G_{2,x}$, $f^G_{1,y} = f^G_{2,y}$, $\tau^G_{1,x} = \tau^G_{2,x}$, $\tau^G_{1,y} = \tau^G_{2,y}$, and $\tau^G_{1,z} = \tau^G_{2,z}$, where only $f^G_{1,z}$ and $f^G_{2,z}$ can be independently controlled per fingertip–see Sec. 7.1.1 for details.

Taking the above simplification and sensor readings into account, we reduce equations

(7.3) and (7.4) into the following $\mathbf{Au} = \mathbf{b}$ form used to solve for $\mathbf{u}$:

$$\begin{aligned}
\mathbf{A} &= [\mathbf{A}^1, \mathbf{A}^2, \mathbf{A}^3, \mathbf{A}^4, \mathbf{A}^5, \mathbf{A}^6, \mathbf{A}^7] \\
\mathbf{A}^1 &= [2, 0, 0, 0, 0, p^G_{1,z} + p^G_{2,z}, -p^G_{1,y} - p^G_{2,y}]^\top \\
\mathbf{A}^2 &= [0, 2, 0, 0, -p^G_{1,z} - p^G_{2,z}, 0, p^G_{1,x} + p^G_{2,x}]^\top \\
\mathbf{A}^3 &= [0, 0, 1, 0, p^G_{1,y}, -p^G_{1,x}, 0]^\top \\
\mathbf{A}^4 &= [0, 0, 0, 1, p^G_{2,y}, -p^G_{2,x}, 0]^\top \\
\mathbf{A}^5 &= [0, 0, 0, 0, 2, 0, 0]^\top \\
\mathbf{A}^6 &= [0, 0, 0, 0, 0, 2, 0]^\top \\
\mathbf{A}^7 &= [0, 0, 0, 0, 0, 0, 2]^\top \\
\mathbf{u} &= [f^G_{1,2,x}, f^G_{1,2,y}, f^G_{1,z}, f^G_{2,z}, \tau^G_{1,2,x}, \tau^G_{1,2,y}, \tau^G_{1,2,z}]^\top \\
\mathbf{b} &= [f^G_x, f^G_y, f^G_{1,z}, f^G_{2,z}, \tau^G_x, \tau^G_y, \tau^G_z]^\top
\end{aligned} \tag{7.5}$$

where $f^G_{1,2,x}$, $f^G_{1,2,y}$, $\tau^G_{1,2,x}$, $\tau^G_{1,2,y}$, $\tau^G_{1,2,z}$ implies that the force and torque of fingertips 1 and 2 are assumed the same during control. Because the determinant($\mathbf{A}$) = 32 and the matrix rank is 7, $\mathbf{A}$ can be inverted for all cases. Note that $\mathbf{u}$ is used for $\mathcal{W}_{meas}$ in (7.1), and $\mathbf{b}$ consists of the direct measurements from our sensors.

7.1.1 Controlling for grasping force and normal force offsets

Although we can control $f^G_{1,2,x}$, $f^G_{1,2,y}$, $\tau^G_{1,2,x}$, $\tau^G_{1,2,y}$, $\tau^G_{1,2,z}$ by using (9.7) directly, where the control output from (7.1) assumes force/torques are equal and in the same direction for both fingertips, controlling $f^G_{1,z}$ and $f^G_{2,z}$ requires extra care as we have the freedom to control these force components independently. To control for these independent normal forces while considering that both fingertips still move either 'inwards' and the gripper is closing, or 'outwards' and the gripper is opening, one solution is to first control for a 'grasping' force, or the amount of normal force exerted on an object, and then separately control for an offset between this grasping force and desired fingertip force. For example, let $f^G_{1,z,ref}$, $f^G_{2,z,ref}$ be

the reference force for fingertips 1 and 2, and $f_{1,z}^G$, $f_{2,z}^G$ be the measured force for fingertips 1 and 2 from our strain gauges. We then define the reference grasping force as the magnitude $f_{grasp,ref}^G = \frac{|f_{1,z,ref}^G|+|f_{2,z,ref}^G|}{2}$. Note, that the measured grasping force is achieved with this same definition but using $f_{1,z}^G$ and $f_{2,z}^G$ instead. We then use (7.1) but with the previous definition of grasping force, where the output $\mathbf{x}$ constitutes the fingertip 1 and 2 positions for z, or $p_{1,z}^G$ and $p_{2,z}^G$, where both fingertip positions change with the same magnitude but in opposite directions according to frame $\{\mathbf{G}\}$. However, note that some offset force may exist if $f_{1,z}^G \neq f_{2,z}^G$ or we desire different reference fingertip forces. To account for this offset force, we let the reference offset force be $f_{z,ref}^G = f_{1,z,ref}^G - f_{grasp,ref}^G$, where the measured force is received from the force/torque sensors at the wrist (or f_z^G). For this case, the control output $\mathbf{x}$ will move both fingertip positions ($p_{1,z}^G$ and $p_{2,z}^G$) with the same magnitude and same direction. In other words, the robot moves its arm to compensate for offsets between fingertip normal forces as defined by $\{\mathbf{G}\}$. While we can essentially think of having 3 admittance controllers, one for controlling all fingertip components except for $f_{1,z}$, and $f_{2,z}$ one for grasping force, and another for the offset in fingertip normal force, for simplicity, we assume the same set of gains across each. In other words, $\mathbf{K}_f$ for controlling the offset normal force will be the same as $\mathbf{K}_f$ for controlling the grasping force.

7.2 Auto-tuning formulation

In this paper, we use a UKF (as done in 6) [94] to tune the control parameters of an admittance controller to track reference fingertip wrenches (Sec. 7.2.1.1) while following desired properties such as updating the spring constant of fingertip force and orientation to increase model accuracy (Sec. 7.2.1.2), ensuring output wrenches that do not cause slipping motion (Sec. 7.2.1.3), and are within kinematic boundaries to prevent singularity configurations (Sec. 7.2.1.4). While the method was used to calibrate controller gains for moving a vehicle, we apply it for the first time here for dexterous manipulation tasks, with

modification on the training objective through use of large costs to avoid instability.

The goal of the auto-tuning method in this work is to calibrate control parameters of a generic controller $\mathbf{x}^G_{k,t} = \boldsymbol{\kappa}_\theta(\mathcal{W}^G_{k,t})$, based on the state, $\mathcal{W}^G_{k,t}$ or wrench of fingertip k at timestep t, the control input, $\mathbf{x}^G_{k,t}$, where $\mathbf{x}^G_{k,t} = [\mathbf{p}^{G\top}_{k,t}, \boldsymbol{\Theta}^{G\top}_{k,t}]^\top$, which is the position and orientation of fingertip k at timestep t, sensor measurements, $\mathcal{W}^G_{k,t,meas}$ or current actual wrench of fingertip k at timestep t using (9.7), and training objectives. The control parameters are represented by $\boldsymbol{\theta}_{k,t}$, which consists of the diagonal gains of the admittance controller, $\mathbf{M}_d$, $\mathbf{D}_d$, and $\mathbf{K}_f$ and spring constants, $\mathbf{K}_p$ and $\mathbf{K}_\theta$ from (7.8) and (7.9).

For our admittance controller, we use the following to represent our dynamic mode:

$$\mathcal{W}^G_{k,t} = \mathbf{Dyn}(\mathcal{W}^G_{k,t-1}, \mathbf{x}^G_{k,t-1}, \mathcal{W}^G_{k,t-1,meas}) - \hat{\mathcal{W}}^G_{k,t} \tag{7.6}$$

where $\mathbf{Dyn}(\mathcal{W}^G_{k,t-1}, \mathbf{x}^G_{k,t-1}, \mathcal{W}^G_{k,t-1,meas})$ is the dynamic model with its input state being the estimated wrench, $\mathcal{W}^G_{k,t-1}$, of fingertip k at timestep $t-1$, the control input as the fingertip position/orientation, $\mathbf{x}^G_{k,t-1}$, of fingertip k at timestep $t-1$, and actual wrench, $\mathcal{W}^G_{k,t-1,meas}$, of fingertip k at timestep $t-1$. Because the auto-tuning method is performed recursively online after measurements are received, we can calculate the process noise to be:

$$\hat{\mathcal{W}}^G_{k,t} = \mathbf{Dyn}(\mathcal{W}^G_{k,t-1}, \mathbf{x}^G_{k,t-1}, \mathcal{W}^G_{k,t-1,meas}) - \mathcal{W}^G_{k,t,meas} \tag{7.7}$$

However, to calculate $\hat{\mathcal{W}}^G_{k,t}$ we still need to formulate a model of our system, i.e., estimate or predict the value of $\mathbf{Dyn}(\mathcal{W}^G_{k,t-1}, \mathbf{x}^G_{k,t-1}, \mathcal{W}^G_{k,t-1,meas})$ using measurements at timestep $t-1$. To do so, we first use the control output $\ddot{\mathbf{x}}$ from (7.1) at timestep $t-1$ and integrate to get fingertip position/orientation at t or $\mathbf{x}^G_{k,t}$. We then model the fingertip forces at timestep t as a virtual spring/mass system:

$$\mathbf{f}^G_{k,t} = \mathbf{K}_p(\Delta\mathbf{p}^G_{k,t}) \tag{7.8}$$

where $\mathbf{K}_p \in R^3$ is a spring constant for each of the x, y, and z displacements of fingertip positions. Note, that $\Delta\mathbf{p}^G_{k,t} = \mathbf{p}^G_{k,t} - \mathbf{p}^G_{k,t-1}$, where $\mathbf{p}^G_{k,t}$ is the control output at timestep t

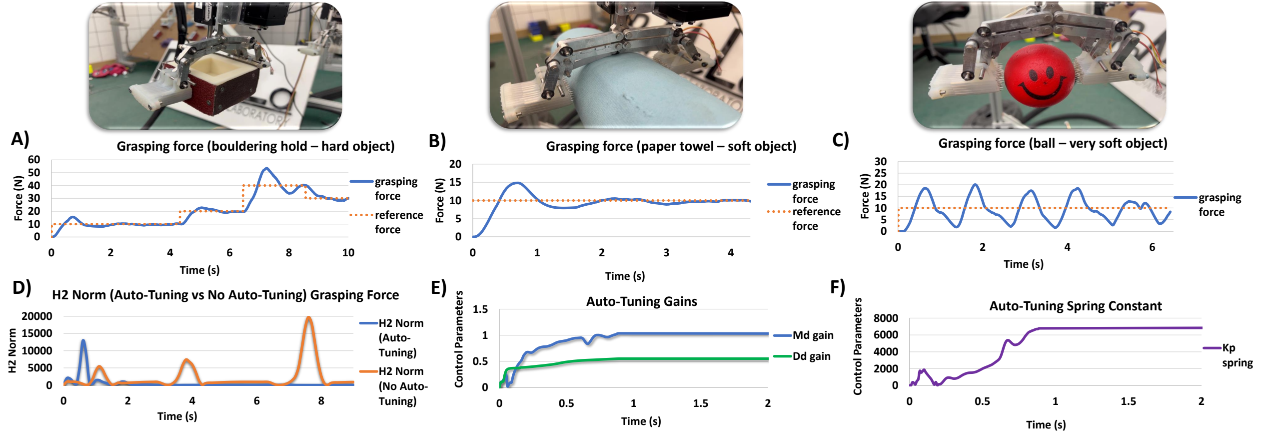


Figure 7.4: **Results: Tracking Grasping Force for Manipulation.** (**A**)-(**C**) shows the results for tracking the grasping force. Note, we do not show the results of auto-tuning versus no auto-tuning for (**A**)-(**C**) because without auto-tuning, the values can quickly diverge, see (**D**). (**D**) is the H2 Norm between auto-tuning (blue) and not auto-tuning (orange) the gains during operation (using the object shown in (**A**). (**E**)-(**F**) exemplify the tuning of the M_d, K_d gains of the admittance controller and K_p gain of the spring term of the model described in equation (7.8) (note for this test we only tune the z-component of the fingertip force).

and $\mathbf{p}^G_{k,t-1}$ is the control output at timestep $t-1$ for fingertip position. Applying the same principle for modeling fingertip forces but now for torques (with a spring constant $\mathbf{K}_\Theta \in R^3$):

$$\boldsymbol{\tau}^G_{k,t} = \mathbf{K}_\Theta(\Delta\boldsymbol{\Theta}^G_{k,t}) \tag{7.9}$$

We can use (7.2), (7.8), and (7.9) to solve for the dynamic model through the following relationship:

$$\mathbf{Dyn}(\mathcal{W}^G_{k,t-1}, \mathbf{x}^G_{k,t-1}, \mathcal{W}^G_{k,t-1,meas}) = \begin{pmatrix} \mathbf{K}_p(\Delta\mathbf{p}^G_{k,t}) \\ \mathbf{p}^G_{k,t} \times \mathbf{K}_p(\Delta\mathbf{p}^G_{k,t}) \end{pmatrix} + \begin{pmatrix} \mathbf{0}^{3\times 1} \\ \mathbf{K}_\Theta(\Delta\boldsymbol{\Theta}^G_{k,t}) \end{pmatrix}$$

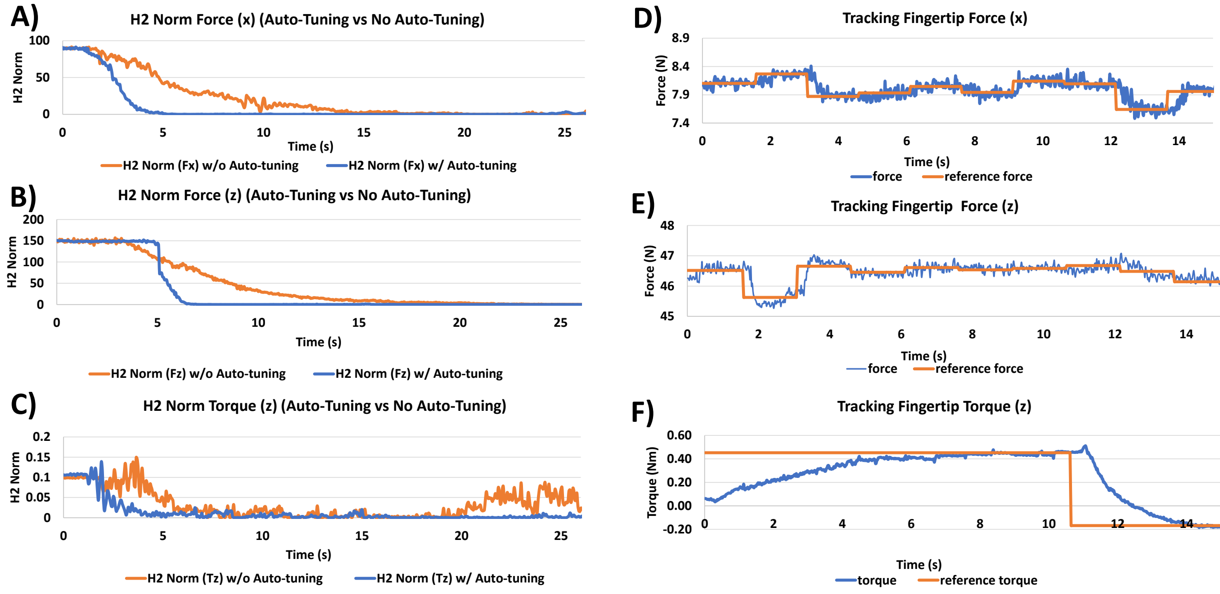


Figure 7.5: **Results: Tracking Wrench during Climbing**. A Climbing wrench trajectory is tracked in (**A**) - (**F**) or $f_{1,2,x}$, $f_{1,z}$, and $\tau_{1,2,z}$. The experiment is depicted on the left of Figure (7.1).

7.2.1 Training objectives

As described in Sec. 7.2, based on a combination of different $\boldsymbol{\theta}^i_{k,t}$ sigma points, which represent different sets of admittance controller gains and spring constants, and simulating them using our proposed model propagation as estimated by (7.6), we then evaluate the performance of these sigma points using the evaluation function $\mathbf{h}(\boldsymbol{\theta}^i_{k,t})$. The objective is to drive our control parameters towards minimizing the difference between the outputs of our evaluation function with a user-defined desired output, specified by $\mathbf{y}^{des}_{k,t}$ from (8.9). In this section, we will now explicitly state our control parameters, $\boldsymbol{\theta}^i_{k,t}$, for each fingertip k, and

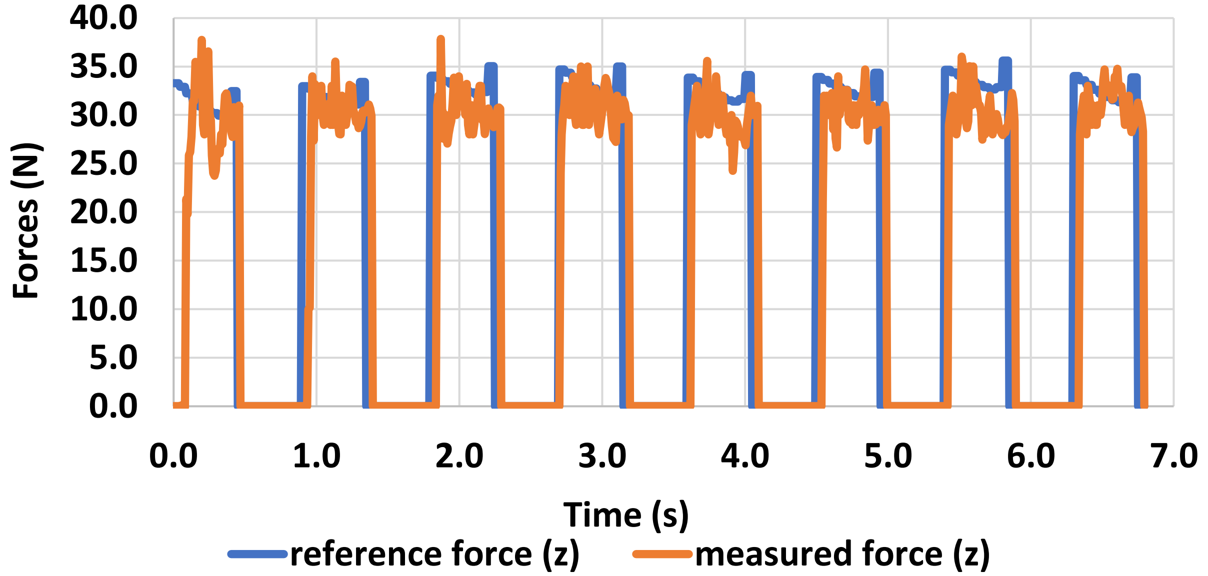


Figure 7.6: **Results: Tracking Fast Reference Forces from an MPC.** An MPC is used to generate reference forces during a trot gait (blue). The actual force (tracked by our adaptive admittance controller), is measured by FT sensors attached to the feet of the robot. The experiment is depicted on the bottom right of Figure (7.1).

describe our evaluation function $\mathbf{h}_{k,t}(\boldsymbol{\theta}^i)$, which can be summarized as:

$$\boldsymbol{\theta}^i_{k,t} = \begin{bmatrix} \mathbf{M}_d(\boldsymbol{\theta}^i_{M_d}) \\ \mathbf{D}_d(\boldsymbol{\theta}^i_{D_d}) \\ \mathbf{K}_f(\boldsymbol{\theta}^i_{K_f}) \\ \mathbf{K}_p(\boldsymbol{\theta}^i_p) \\ \mathbf{K}_\Theta(\boldsymbol{\theta}^i_\Theta) \end{bmatrix}, \mathbf{h}_{k,t}(\boldsymbol{\theta}^i) = \begin{bmatrix} \mathbf{h}_{ref}(\boldsymbol{\theta}^i) \\ \mathbf{h}_{spring}(\boldsymbol{\theta}^i) \\ h_{forces}(\boldsymbol{\theta}^i) \\ h_{const}(\boldsymbol{\theta}^i) \end{bmatrix} \tag{7.10}$$

$$\mathbf{y}^{des}_{k,t} = \begin{bmatrix} \mathbf{y}^{des}_{ref} \\ \mathbf{y}^{des}_{spring} \\ y^{des}_{forces} \\ y^{des}_{const} \end{bmatrix}$$

where the control parameters, $\boldsymbol{\theta}^i_{k,t} \in R^{24k}$, include the gains of the admittance controller, $\mathbf{M}_d$, $\mathbf{D}_d$, and $\mathbf{K}_f$, which in total includes 18 gains to be tuned per fingertip, and the spring constants $\mathbf{K}_p$ and $\mathbf{K}_{\boldsymbol{\Theta}}$, which in total includes 6 constants to be tuned per x, y and z components. Note, we do not tune $\mathbf{K}_d$ in (7.1) because in this work we are primarily interested in tracking force not position, i.e., the gripper must be in contact and we set $\mathbf{K}_d$ to be zero. The evaluation function $\mathbf{h}_{k,t}(\boldsymbol{\theta}^i) \in R^{14k}$ contains 4 main components, including the cost of following a reference wrench trajectory ($\mathbf{h}_{ref} \in R^{6k}$), the cost of estimating the spring constants ($\mathbf{h}_{spring} \in R^{6k}$), the cost of ensuring that the control output generates forces that do not cause slipping ($h_{forces} \in R^k$), and the cost of satisfying kinematic and controller constraints ($h_{const} \in R^k$). The costs of each will now be described in the following subsections and also include the corresponding desired values, represented by $\mathbf{y}^{des}_{k,t}$ (which must be the same size and format as $\mathbf{h}_{k,t}(\boldsymbol{\theta}^i)$).

7.2.1.1 Reference trajectory

One of the main goals for auto-tuning the admittance controller is to tune the gains such that a reference wrench trajectory, which was calculated using the methods found in [127], can be closely followed. To do so, for fingertip k at timestep t, where k, t notation are omitted for simplicity, we let $\mathbf{h}_{ref}(\boldsymbol{\theta}^i) = [\mathcal{W}(\boldsymbol{\theta}^i)^G]$ or specifically, $\mathbf{h}_{ref}(\boldsymbol{\theta}^i) = [\mathbf{f}(\boldsymbol{\theta}^i)^{G\top}, \boldsymbol{\tau}(\boldsymbol{\theta}^i)^{G\top}]^\top$, which represent the fingertip force and torque values based on the model propagation using (7.6) to evaluate $\mathcal{W}(\boldsymbol{\theta}^i)^G$. Note, that both are functions of the sigma points or $\boldsymbol{\theta}^i$ which provide the current gains of the admittance controller in (7.1). If we then let our desired parameters of (8.9) or $\mathbf{y}^{des}_{ref} = [\mathcal{W}^G_{ref}]$, where $\mathcal{W}^G_{ref}$ is received directly from the reference trajectory from (7.1), then the training objective is to produce control outputs that follow this trajectory as closely as possible.

7.2.1.2 Spring evaluation

One of the challenges of the model propagation described in (7.6) is in propagating the fingertip wrenches from timestep $t-1$ to t. The approach used in this work is to assume a spring model, where we estimate the wrench at timestep t by propagating the fingertip position and orientation instead, as shown in (7.8) and (7.9). However, to use this model we must know the spring constants $\mathbf{K}_p$ and $\mathbf{K}_\Theta$ for the x, y, and z displacements, which requires extensive system identification methods. To automate this process, we can also use our auto-tuning method to evaluate $\mathbf{K}_p$ and $\mathbf{K}_\Theta$ during system operation directly. Thus, we have $\mathbf{h}_{spring}(\boldsymbol{\theta}^i) = [\mathbf{K}_p(\boldsymbol{\theta}^i)\Delta\mathbf{p}(\boldsymbol{\theta}^i)^{G\top}, \mathbf{K}_\Theta(\boldsymbol{\theta}^i)\Delta\boldsymbol{\Theta}(\boldsymbol{\theta}^i)^{G\top}]^\top$, where the values of the sigma points $\boldsymbol{\theta}^i$ are equivalent to the spring constant $\mathbf{K}_p$ and $\mathbf{K}_\Theta$ itself. Our desired parameter then becomes $\mathbf{y}_{spring}^{des} = [\mathbf{f}_{meas}^{G\top}, \boldsymbol{\tau}_{meas}^{G\top}]^\top$, where $\mathbf{f}_{meas}^{G}$ and $\boldsymbol{\tau}_{meas}^{G}$ are the actual fingertip force and torque. By minimizing the difference between $\mathbf{h}_{spring}(\boldsymbol{\theta}^i)$ and $\mathbf{y}_{spring}^{des}$ (in (8.9)), the training objective of the auto-tuner is to find parameter $\boldsymbol{\theta}^i$ which best estimates the value of $\mathbf{K}_p$ and $\mathbf{K}_\Theta$ such that the spring model gets closer to the actual current values from (9.7). Note, that by improving the model through updating the spring constants, we also improve the performance of the auto-tuner over time as the auto-tuning method relies on estimating the model mismatch in (7.6) for propagation.

7.2.1.3 Force limits

During the evaluation of our costs, or $\mathbf{h}_{k,t},(\boldsymbol{\theta}^i)$, which requires simulating each sigma point using our dynamics model from timestep $t-1$ to t, it is possible that some combination of controller gains may propagate forces that are near or at the boundary of slipping (i.e., slipping between the fingertip contact point and its environment). As described previously, and written in more detail in [72], we assume a contact model which has a finite contact patch that includes frictional and normal forces, and a torsional moment with respect to the

contact normal. To prevent control outputs that cause slippage, we can make use of (7.11):

$$-\lambda f_{k,z}^{G} \leq \tau_{k,z}^{G} \leq \lambda f_{k,z}^{G} \tag{7.11}$$

where the fingertip is least likely to slip when $\tau_{k,z}^{G}$ is between $-\lambda f_{k,z}^{G}$ and $\lambda f_{k,z}^{G}$. This behavior can be achieved by setting an arbitrary high cost (say δ) when (7.11) is not satisfied, and if (7.11) is satisfied, we can set the cost to be a function of how 'far' away the system is from slipping. Thus, we can let $h_{forces}(\boldsymbol{\theta}^i) = [\delta]$ if (7.11) is not satisfied, and if it is satisfied, we can let $h_{forces}(\boldsymbol{\theta}^i) = [\frac{1}{|\lambda f_{k,z}^{G}|-|\tau_{k,z}^{G}|}]$. In both cases, our desired parameter is $y_{forces}^{des} = [0]$

7.2.1.4 Kinematic and controller constraints

As we propagate our admittance control output $\mathbf{x}$ based on different values for $\boldsymbol{\theta}^i$ during our propagation from $t-1$ to t, we may get unreasonable values for $\mathbf{x}$, due to 'bad' choices of $\boldsymbol{\theta}^i$, that are kinematically infeasible or outside the workspace of the robot causing singularity, or obtain controller gains which are not positive semi-definite. A simple solution to avoid our auto-tuning method to choose these $\boldsymbol{\theta}^i$ values, is to impose a large cost, which we arbitrarily term as ζ, when $\mathbf{x}$ is infeasible or violate the semi-definite property. Thus, we can write that if $\boldsymbol{\theta}^i$ are chosen such that it causes a kinematic infeasibility or violates the positive semi-definite property then $\mathbf{h}_{const}(\boldsymbol{\theta}^i) = [\zeta]$, and if the solution does not cause violations, then $\mathbf{h}_{const}(\boldsymbol{\theta}^i) = 0$, where our desired value $y_{const}^{des} = [0]$.

7.3 Experimental Validation

We implement our controller on the SCALER [130] quadruped climbing robot. SCALER weighs 9.6 kg, and has two configurations, a walking configuration used to track ground reaction forces from an MPC [34], 3 DoF per leg, and a climbing configuration to track fingertip wrenches, 6 DoF per leg in addition to a 1 DoF two-finger gripper, which totals 7 DoF per limb [130]. At the wrist of the robot, or endpoint of 3 DoF or 7 DoF configuration,

we use BOTA's force/torque sensor [17], and have 1-axis strain gauges at each fingertip of our gripper, where actuation for grasping is done using a linear actuator. We demonstrate the efficacy of auto-tuning our gains using the H2 norm defined as $||\mathcal{W}_{meas} - \mathcal{W}_{ref}||_{H_2}$ (defined as a scalar value). Thus, a decrease in the H2 norm indicates better tracking of the reference wrench trajectory. Overall, we show our results through three different experiments as shown in Figures 7.4, 7.5 and 7.6. In Sec. 7.3.1, we discuss the tuning process for grasping, and apply our method toward tracking grasping force on objects with different degrees of compliance. In Sec. 7.3.2, we discuss the tuning and then tracking of not only forces, but also torques, which is necessary for free-climbing tasks. In Sec. 7.3.3, to show how our admittance controller can handle tracking reference forces that rapidly changes, we apply our controller for tracking the ground reaction force outputs of an MPC controller during a fast moving trot gait. For 7.3.1 and 7.3.2, our baseline or no auto-tuning, is an admittance controller that has been hand-tuned to the best of our abilities to achieve convergence according to [112]. We do not show a baseline for 7.3.3, as we were not able to hand-tune the gains of the admittance controller that could successfully track or converge for fast moving reference forces. Finally, we note that we ran the admittance controller and auto-tuner at 100 Hz, and applied the auto-tuning procedure for one fingertip only.

7.3.1 Experiment 1: Tracking grasping force

As shown in (**A**)-(**C**) in Fig. 7.4, we were able to track the grasping force of a stiff object, or bouldering hold, as well as compliant objects, or paper towel and a soft ball. We show the tuning results in (**D**), where we see a lower H2 norm with auto-tuning (blue) compared to not using auto-tuning (orange), using the bouldering hold as seen in (**A**) as our example. Additionally, without auto-tuning the system diverges and becomes unstable as shown by the large peaks. In (**E**) we show how the auto-tuner updates the admittance controller gains, here we only update M_d and D_d, which show that both gains initialized at 0.1 converge to a constant value once the controller becomes stable or follows the reference trajectory. We

also show in (**F**) how the K_p spring constant, here just a single value, changes as the gripper is increasing its grasping force, where eventually it reaches a very high value as the gripper's change in position decreases with respect to increasing grasping force. Lastly, we note that the auto-tuner only needed 0.8 seconds to converge to optimal gains for this test.

7.3.2 Experiment 2: Tracking wrenches for free-climbing

To auto-tune other force/torque components, we update $\mathbf{M}_d$, $\mathbf{D}_d$, $\mathbf{K}_f$ and spring constants by hanging the robot on a bar and having it grasp a bouldering hold while following a constant reference wrench trajectory. In total, we tune the following components of wrench: $f^G_{1,2,x}$, $f^G_{1,2,z}$, $\tau^G_{1,2,x}$, $\tau^G_{1,2,y}$, and $\tau^G_{1,2,z}$. (**A**)-(**C**) of Fig. 7.5 shows that with auto-tuning we demonstrate quicker convergence to our reference compared to without auto-tuning for $f^G_{1,2,x}$, $f^G_{1,2,z}$, and $\tau^G_{1,2,z}$ as an example. We note that with and without auto-tuning the H2 norm eventually converged for the case of tracking force, likely as the initialized gains were already sufficiently stable. However, without auto-tuning, the H2 norm could not converge for tracking the additional torque (see (**C**) of Fig. 7.5), $\tau^G_{1,2,z}$, while convergence occurred using auto-tuning. Additionally, the auto-tuner took about 3 seconds to converge for fingertip force in the x direction, 7 seconds for force in the z direction, and 5 seconds for torque in the z direction.

After using auto-tuning to find optimal gains, we use these updated gains to track a changing wrench trajectory, namely shear force, grasping force with fingertip normal force offset as described in Sec. 7.1.1, and additional torque due to rotational friction along the yaw axis, or $f^G_{1,2,x}$, $f^G_{1,z}$, $f^G_{2,z}$, and $\tau^G_{1,2,z}$ simultaneously (results for $f^G_{1,z}$ are shown for brevity). The results of tracking this wrench is demonstrated in (**D**)-(**F**) in Fig. 7.5 for the robotic free-climbing task.

7.3.3 Experiment 3: Tracking fast reference forces

Lastly, we demonstrate the speed of our admittance controller when applied to a single point contact robot during a trot gait as shown in Fig. 7.6. Output ground reaction forces from a point contact (f_z^G) are produced by an MPC controller, formulated identically to [34], and we show a step response of ≈ 0.03 seconds across 8 steps of a trot gait (i.e., it took about 0.03 seconds for the admittance controller to track the force profile per step). We note that while the step response was adequate for climbing and even for locomotion tasks, the MPC only needs to run every ≈ 0.03 seconds, faster response rates may be achievable with motors that have a lower gear ratio, as our motors have a high gear ratio.

7.4 Conclusion

We demonstrated an auto-calibrating admittance controller that could track wrench trajectories during locomotion and manipulation tasks. We show that we could successfully track the additional torque due to rotational friction simultaneously with other force components for an extreme manipulation case trajectory - namely for robotic wall-climbing. In future work, we aim to use the auto-tuner to derive various spring constants for different surfaces, and examine more complex training objectives such as recovering from slipping.

CHAPTER 8

Real-to-Sim: Predicting Residual Errors

8.1 Introduction

One of the primary challenges in the field of robotics is to successfully exploit simulation-based results and apply these results to real-world applications. Simulations are convenient as they provide data with low-cost, and can be run safely for long periods of time without risking potentially expensive hardware. However, in almost all cases (even for high-fidelity and expensive simulators), the simulation and reality will differ, i.e., 'reality gap', due to error or mismatch from sensors and actuators that are difficult to correctly model. Because of this reality gap, it typically takes a significant amount of engineering effort to successfully train an agent in simulation and then transfer this learning to hardware. In previous work (see Section **??**), there have been numerous approaches to close this reality gap [152], however, these approaches still require intricate hand-tuning and substantial data collection. Ideally, one way to close the reality gap is to either have an accurate dynamic model of the robot itself and/or directly use hardware data to inform and improve the simulator. Still, the challenge remains in that access to a highly accurate dynamic model is difficult and typically only sparse amounts of data can be received from hardware experiments. Thus, methods that can quickly improve a dynamic model, or a simulator using sparse amounts of data are needed.

In this work, our objective is to provide a method to learn the residual errors assumed to be non-linear and non-Gaussian between a dynamic or simulator model, and the real robot.

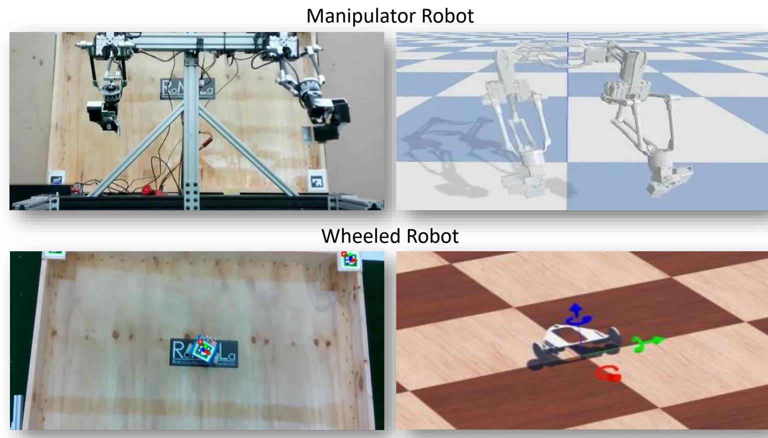


Figure 8.1: **Real-to-Sim.** Here we show our experimental setup, where we learn the residual error between the actual and simulated robot.

By learning these residual errors, we can either improve an existing dynamic model to be closer to the real robot, or improve the simulator to be closer to the real robot. The former may be useful for model-based controls (e.g., model predictive control) or for model-based planners, while the latter can be used for reinforcement learning (RL) applications. The residual errors are learned using a neural network, where the parameters of a neural network are updated through an Unscented Kalman Filter (UKF) which can quickly converge to desired parameters (i.e., minimizing the difference between the current and reference model) [93]. A UKF is useful (relative to other filtering methods) as they can better handle non-linear and non-Gaussian residual errors during the update step.

Summary of our contributions

1. An implementation of neural networks and a UKF to update network parameters are used to quickly learn residual errors between the current and reference model.

2. We show how our approach learns the residual error for several test cases, in particular Sim-to-Dyn (simulator is the reference model and the dynamic model with residual error is the current model), Real-to-Dyn (the real robot is the reference model and

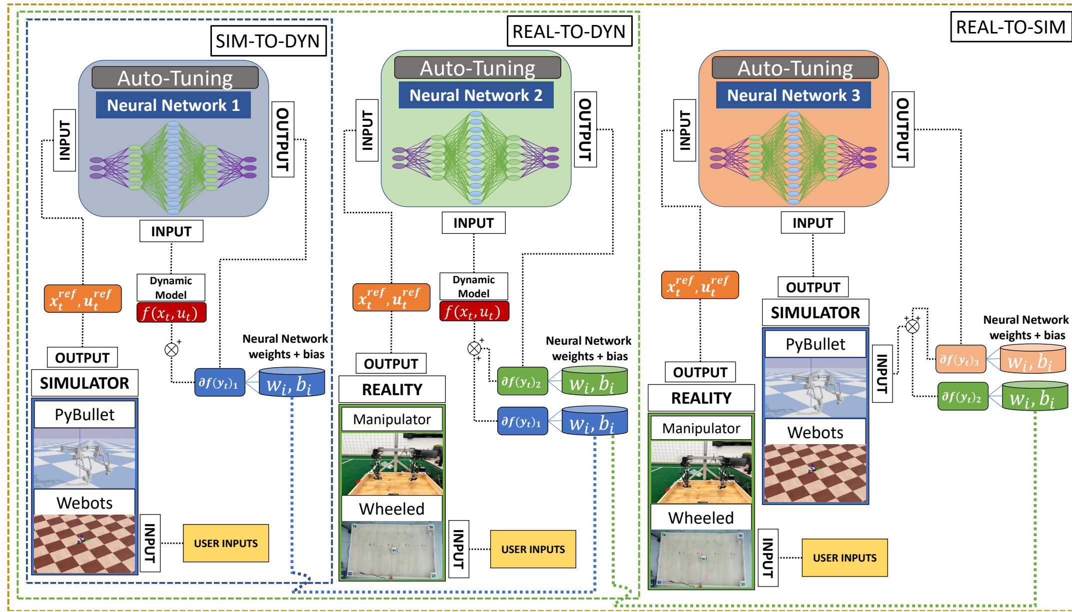


Figure 8.2: **Methods flowchart.** Here we show the overall methods of this paper as described in Section 8.2. Overall, the goal is to learn the residual model error between a simulator model and a dynamic model or Sim-to-Dyn (boxed in blue), the real robot and a dynamic model or Real-to-Dyn (boxed in green), and the real robot and the simulator model or Real-to-Sim (boxed in brown). The model is learned using a UKF method which calibrates weights and bias parameters of a neural network (the weights and bias variables are parameterized by $\delta(\mathbf{y}_t)$). Although not necessary, the learned residual model of one case may also be used as part of a warm-start procedure for the next case (e.g., the residual model for Sim-to-Dyn may be used as the starting residual error for the Real-to-Dyn case–which we found to reduce the amount of learning required to model any additional residual error).

the dynamic model with residual error is the current model), and Real-to-Sim (the real robot is the reference model and the simulator with residual error is the current model).

3. Our results are demonstrated in simulation and hardware using a mobile and stationary manipulator robot.

8.2 Problem Definitions

8.2.1 General Model

We assume the following model propagation from time step t to $t+1$:

$$\mathbf{x}_{t+1} = \mathbf{Dyn}(\mathbf{x}_t, \mathbf{u}_t) + \delta(\boldsymbol{y}_t) \tag{8.1}$$

where $\mathbf{x}_t$ is the state, $\mathbf{u}_t$ is the control input, $\mathbf{Dyn}(\mathbf{x}_t, \mathbf{u}_t)$ is some dynamic model of the robot to help propagate the states of the robot from timestep t to $t+1$, and $\delta(\boldsymbol{y}_t)$ is considered the model mismatch (or residual error of the model), which we assume to be a non-Gaussian and non-linear function parameterized by $\boldsymbol{y}_t$. Note, throughout this paper, we will model the derivative terms of the system dynamics since our environment is assumed uniform, and the residual term is not a function of positional space. The objective of this paper is to learn the function $\delta(\boldsymbol{y}_t)$ through a neural network combined with a UKF (see Section 8.2.2). Here, we will learn the residual error for several different cases, and transfer the learned residual from one case to the next case to help decrease the overall learning time. Note that from this point, we will use the hat notation on states that result from a predicted model (i.e., dynamic model with the addition of our residual error function) and a bar notation on states that are received directly from the localization result of the simulation or the real robot.

Our first test case will be Sim-to-Dyn, where we use a predefined dynamic model, $\mathbf{Dyn}(\mathbf{x}_t, \mathbf{u}_t)$, and learn the residual error $\delta(\boldsymbol{y}_t)_1$ to predict the state of the model of a simulation, $\hat{\mathbf{x}}^{sim}_{t+1}$. Thus, we can write the Sim-to-Dyn case as the following:

$$\hat{\mathbf{x}}^{sim}_{t+1} = \mathbf{Dyn}(\mathbf{x}_t, \mathbf{u}_t) + \delta(\boldsymbol{y}_t)_1 \tag{8.2}$$

The next test case will be Real-to-Dyn, where we try to learn the error between the dynamic model of the robot and the results from the real robot. For this case, instead of starting with just the dynamic model, $\mathbf{Dyn}(\mathbf{x}_t, \mathbf{u}_t)$, we instead apply the learned residual from equation (8.2) in addition to the dynamic model, and learn the remaining (new) residual error specified

by $\delta(\boldsymbol{y}_t)_2$. Note, by using the previously learned model as the updated dynamic model, we may decrease the amount of training time needed to find the remaining residual error (i.e., warm-start) – however, this is only true if we assume that the simulator model is more accurate than the initialized dynamic model as we do in this paper (although this is not a prerequisite to using our methods and the previously learned model error may also be set to zero if equation (8.3) is not true):

$$|\bar{\mathbf{x}}_{t+1}^{real} - \mathbf{Dyn}(\mathbf{x}_t, \mathbf{u}_t)| > |\bar{\mathbf{x}}_{t+1}^{real} - \hat{\mathbf{x}}_{t+1}^{sim}| \tag{8.3}$$

The Real-to-Dyn case can then be specified as:

$$\hat{\mathbf{x}}_{t+1}^{real} = \mathbf{Dyn}(\mathbf{x}_t, \mathbf{u}_t) + \delta(\boldsymbol{y}_t)_1 + \delta(\boldsymbol{y}_t)_2 \tag{8.4}$$

where the goal is to learn the residual error, $\delta(\boldsymbol{y}_t)_2$, while $\delta(\boldsymbol{y}_t)_1$ was already learned from the previous Sim-to-Dyn case.

Lastly, we will test the Real-to-Sim case, where our goal is to match the simulator model to the real robot. As before, we can make use of the previously learned cases to inform and facilitate the training procedure. For example, we can combine equations (8.2) and (8.4) to get the following relationship (with the goal of finding the remaining new residual error $\delta f(\boldsymbol{y}_t)_3$):

$$\hat{\mathbf{x}}_{t+1}^{real} = \mathbf{Dyn}(\mathbf{x}_t, \mathbf{u}_t) + \delta(\boldsymbol{y}_t)_1 + \delta(\boldsymbol{y}_t)_2 + \delta(\boldsymbol{y}_t)_3 \tag{8.5}$$

which can be identically represented as:

$$\hat{\mathbf{x}}_{t+1}^{real} = \hat{\mathbf{x}}_{t+1}^{sim} + \delta(\boldsymbol{y}_t)_2 + \delta(\boldsymbol{y}_t)_3 \tag{8.6}$$

However, we note that in this test case we aim to compare the state estimation of the simulator to the state estimation of the real robot. Thus, we can replace the predicted simulator model, $\hat{\mathbf{x}}_t^{sim}$, with the actual simulator values defined by $\bar{\mathbf{x}}_t^{sim}$ (in this case, $\delta(\mathbf{y}_t)_2$ may act as a 'warm-start' for modeling the behavior of the real robot, since the term was learned during Real-to-Dyn):

$$\hat{\mathbf{x}}_{t+1}^{real} = \bar{\mathbf{x}}_t^{sim} + \delta(\boldsymbol{y}_t)_2 + \delta(\boldsymbol{y}_t)_3 \tag{8.7}$$

The overall method is also demonstrated in Fig. 8.2.

8.2.2 Neural Network with Unscented Kalman Filtering

As described previously, the goal is to learn the residual error $\delta(\boldsymbol{y}_t)$ from equation (8.1) in order to minimize the difference between the current and a reference model. Depending on the test case, what is designated as the current and reference model may differ. For example, in Sim-to-Dyn the reference model is the simulator while the current model is the specified dynamic model, in Real-to-Dyn, the reference model is the real robot while the current model is the specified dynamic model, and in Real-to-Sim, the reference model is the real robot while the current model is the simulator model. Each of these cases and which specific residual model we wish to learn is described in Section (8.2.1).

To learn this residual error $\delta(\boldsymbol{y}_t)$ we use a fully connected neural network (see Fig. 9.3) that has one input layer (with either 5 inputs when using our mobile robot or 6 inputs when using our manipulator robot), one hidden layer with 10 nodes, and one output layer (with 3 outputs when using our mobile or manipulator robot). We can write this network as the following:

$$\delta(\boldsymbol{y}) = \boldsymbol{y}_{\text{out}}\, \sigma\left(\boldsymbol{y}_{\text{lay}}\, \sigma\left(\boldsymbol{y}_{\text{in}}\, \mathbf{z} + \boldsymbol{y}_{\text{in}\ ,0}\right) + \boldsymbol{y}_{\text{lay}\ ,0}\right) + \boldsymbol{y}_{\text{out}\ ,0} \tag{8.8}$$

where $\boldsymbol{y}_{out} \in R^{3\times10}$ are the weights of the output layer, $\boldsymbol{y}_{out,0} \in R^{3\times1}$ are the bias of the output layer, $\boldsymbol{y}_{lay} \in R^{10\times10}$ are the weights of the hidden layer, $\boldsymbol{y}_{lay,0} \in R^{10\times1}$ are the bias of the hidden layer, $\boldsymbol{y}_{in} \in R^{10\times5}$ or $\in R^{10\times6}$ are the weights of the input layer, and $\boldsymbol{y}_{in,0} \in R^{10\times1}$ are the bias of the input layer, and lastly, the inputs are represented by $\mathbf{z} \in R^{5\times1}$ or $\in R^{6\times1}$. Thus, in total, $\delta(\boldsymbol{y})$ is a function parameterized by $\boldsymbol{y} \in R^{198\times1}$ for the mobile robot, and $\boldsymbol{y} \in R^{209\times1}$ for the robot manipulator. The σ represent leaky ReLu activation functions $\mathbf{a} = \sigma(\mathbf{b})$ with $\mathbf{a} = \max(0.01\mathbf{b}, \mathbf{b})$, where $\mathbf{b}$ is the input to the activation function.

Equation (8.8) is considered the feed-forward procedure of a neural network. However, in this paper, instead of using typical back-propagation methods with optimization functions

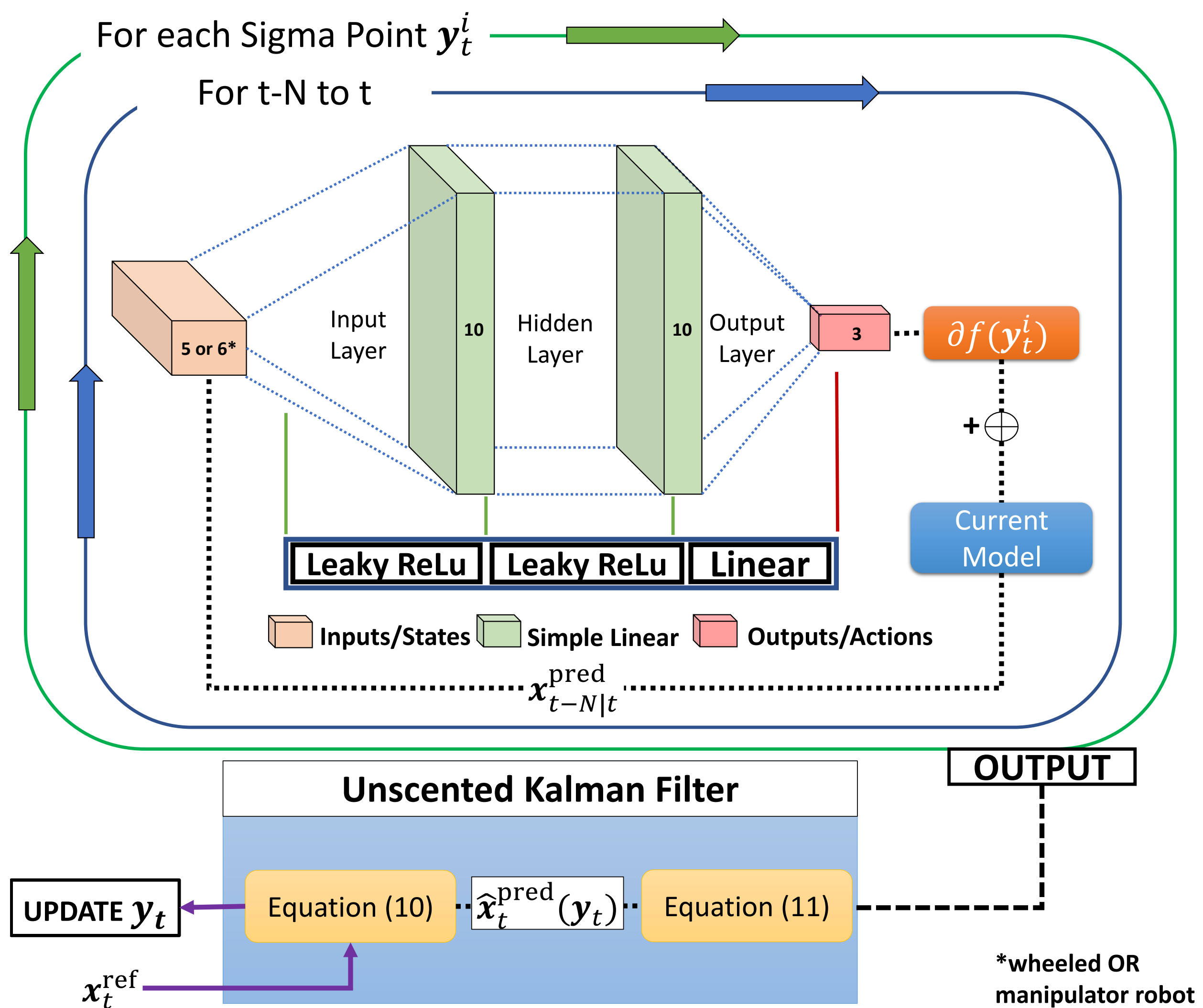


Figure 8.3: **Neural Network with UKF.** We illustrate the UKF method as described in Section 8.2.2. The UKF calibrates weights and bias values of a neural network that is composed of a hidden layer. The result of this calibration is to produce an output or residual model error that minimizes the difference between a current and reference model. The green and blue arrows simply indicate that we are doing a two sequential for loop iterations.

such as Batch Gradient Descent, we instead apply the UKF method described in [93]. The motivation of using this method opposed to other more typical back-propagation methods is that it's model-based and has been shown in previous works [117, 118] to tune parameters

quickly with sparse amounts of data. Additionally, compared to gradient-descent methods, a UKF can be more robust to the presence of outliers in the data (i.e., uses a weighted average of sigma points), has reduced computational complexity as it uses a fixed set of sigma point to represent the system state rather than requiring the calculation of higher-order derivatives, and shows improved convergence as it uses deterministic sampling process to propagate the sigma points through the system.

While we refer readers to [93] for a full description of the UKF method, the main objective of how we apply this method in this paper, is to update the neural network weights/bias parameters in (8.8) or $\boldsymbol{y}_t$ (where t is the current time step) such that the difference between a predicted model, $\mathbf{x}_t^{pred}$, and a reference model, $\mathbf{x}_t^{ref}$, is minimized. This calibration is done through finding a Kalman gain $\mathbf{K}_t$ of the UKF using a recursive implementation. In other words, we first collect the history of past reference and predicted values using a time horizon specified by $t-N$ (where N is the number of time steps in the past time horizon), and make the Kalman gain update at the current time step t. Thus, we formulate the update of our neural network parameters as:

$$\boldsymbol{y}_t = \boldsymbol{y}_{t-N} + \Delta \boldsymbol{y}_t, \tag{8.9}$$

where the update $\Delta \mathbf{y}_t$ is computed based on the following equation:

$$\Delta \boldsymbol{y}_t = \mathbf{K}_t \left(\mathbf{x}_t^{ref} - \hat{\mathbf{x}}^{pred}(\boldsymbol{y}_t) \right) \tag{8.10}$$

Finally, the Kalman gain ($\mathbf{K}_t$) is calculated using the following UKF formulation:

$$\mathbf{K}_t = \mathbf{C}_t^{sz}\mathbf{S}_t^{-1} \tag{8.11a}$$

$$\mathbf{S}_t = \mathbf{C}_v + \textstyle\sum_{i=0}^{2L} \mathbf{w}^{c,i}(\mathbf{x}_t^{pred,i} - \hat{\boldsymbol{x}}_t^{pred}) \tag{8.11b}$$

$$(\mathbf{x}_t^{pred,i} - \hat{\boldsymbol{x}}_t^{pred})^\top \tag{8.11c}$$

$$\mathbf{C}_t^{sz} = \textstyle\sum_{i=0}^{2L} \mathbf{w}^{c,i}(\boldsymbol{y}_t^i - \hat{\boldsymbol{y}}_t)(\mathbf{x}_t^{pred,i} - \hat{\boldsymbol{x}}_t^{pred})^\top \tag{8.11d}$$

$$\hat{\boldsymbol{x}}_t^{pred} = \textstyle\sum_{i=0}^{2L} \mathbf{w}^{a,i}\mathbf{x}_t^{pred,i} \tag{8.11e}$$

$$\mathbf{x}_t^{pred,i} = \mathbf{x}^{pred}(\boldsymbol{y}_t^i) \tag{8.11f}$$

$$\mathbf{P}_{t|t-1} = \mathbf{C}_\theta + \textstyle\sum_{i=0}^{2L} \mathbf{w}^{c,i}(\boldsymbol{y}_t^i - \hat{\boldsymbol{y}}_t)(\boldsymbol{y}_t^i - \hat{\boldsymbol{y}}_t)^\top \tag{8.11g}$$

$$\hat{\boldsymbol{y}}_t = \textstyle\sum_{i=0}^{2L} \mathbf{w}^{a,i}\boldsymbol{y}_t^i \tag{8.11h}$$

$$\mathbf{P}_{t|t} = \mathbf{P}_{t|t-1} - \mathbf{K}_t\mathbf{S}_t\mathbf{K}_t^\top \tag{8.11i}$$

where $\boldsymbol{y}_t^i$ with $i = 0, ..., 2L$ are the sigma points, $\mathbf{w}^{c,i}$ and $\mathbf{w}^{a,i}$ are the weights of the sigma points, $\mathbf{C}_t^{sz}$ is the cross-covariance matrix, $\mathbf{S}_t$ is the innovation covariance, and $\mathbf{P}_{t|t}$ is the estimate covariance. The weights are user defined, and in this paper, we use the same weights as described in *remark 6* of [93]. Lastly, note that the covariance matrix $\mathbf{C}_\theta$ (which is initialized by the user) defines the aggressiveness of the update, while $\mathbf{C}_v$ defines the 'weight' given to the components of $\mathbf{x}_t^{ref}$ [93].

8.3 Implementation

8.3.1 Computer specifications

Our method was performed on a laptop with 4 CPU cores (Intel core i7-8850H CPU at 2.60 Ghz) with a Quadro P3200 GPU. We note that the computation time of our method described in Section 8.2.2 was ≈ 0.3 seconds for each update of our neural network parameters, which includes a time horizon of $N = 20$. However, the computation time of the UKF update does increase with either a greater time horizon or number of parameters (i.e., larger neural

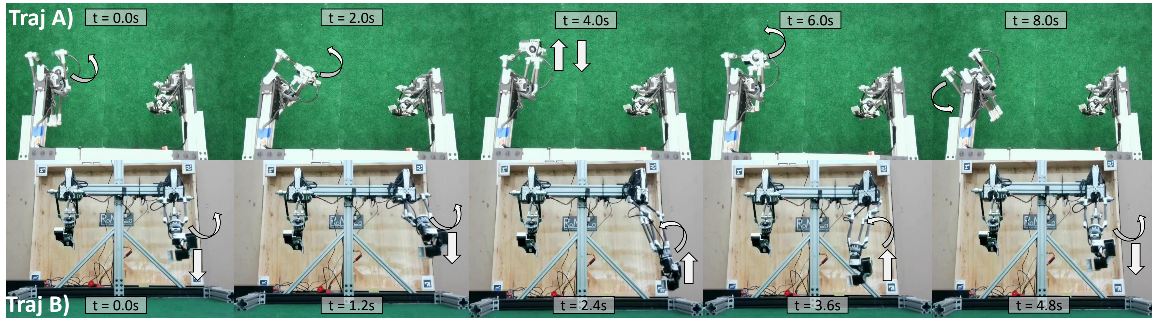


Figure 8.4: **Manipulator trajectories.** For evaluating the modelling of residual errors on the manipulator arm, we use two trajectories. The first trajectory (shown in top half) consists of only x and y components–drawing a 2D circle, with a straight line in the middle. This was done to ensure we can model circular and also linear motions simultaneously. The second trajectory, shown on the bottom half, consists of x, y, and z components–drawing a circle in x and y while moving up and down in z. The red line on the last image of the trajectory shows the complete trajectory made by the arm.

networks). Although we chose a relatively small network in this work (since we only needed a small network to learn the residual errors for our application), larger networks can still be chosen because the computation time of our method is not critical and can be computed (as we do here) on its own CPU core through multi-processing. In other words, the update to our parameters can be performed at any point during the experiment operation and does not need to be updated at any specific frequency. To show the generality of our method, we apply our method on two different robot configurations (both in hardware and in their state localization methods).

8.3.2 Differential Drive Robot

We first demonstrate our methods on a differential drive two-wheeled robot as validation (before applying our methods on our manipulator robot). Localization for the real robot is done using April tags on each corner of a 1 by 1 meter square box and on the robot

itself, with an Intel RealSense D435i RGB-D camera in a bird's eye view configuration. The localization provides state estimation of the robot's position and heading angle in addition to their corresponding velocities. For simulating our robot, we use the Webots software [95] which includes system noise and contact dynamics.

To propagate our states (where the residual error is an error on velocity), we use the following differential drive model:

$$\begin{bmatrix} \dot{x}_t \\ \dot{y}_t \\ \dot{\Theta}_t \end{bmatrix} = R \begin{bmatrix} \frac{c_t}{2} & \frac{c_t}{2} \\ \frac{s_t}{2} & \frac{s_t}{2} \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix} \mathbf{u_t} + \delta(\boldsymbol{y}_t) \tag{8.12}$$

where $\dot{x}$, $\dot{y}$, $\dot{\Theta}$ represent the linear and angular velocity (Θ is yaw heading angle), R is the radius of the wheel, L is the length from the left to the right wheel, and $\mathbf{u} \in R^{2\times1}$ is the control input, wheel angular velocity for the left (u_t^l) and right (u_t^r) wheel. Lastly, $\delta(\boldsymbol{y}_t)$ represents the residual error predicted by the neural network (see Section 8.2.2). For the differential drive robot, our input to this neural network is $\mathbf{z}_t$=$[\dot{x}_t, \dot{y}_t, \dot{\Theta}_t, u_t^l, u_t^r]^\top$. Note, that our residual error is an error on the velocity.

To apply the Kalman gain update as described in equation (8.9), we will use the following for the differential drive robot:

$$\boldsymbol{x}_t^{ref} = \begin{bmatrix} \dot{x}_{t-N|t}^{ref} C_{\dot{x}} \\ \dot{y}_{t-N|t}^{ref} C_{\dot{y}} \\ \dot{\Theta}_{t-N|t}^{ref} C_{\dot{\Theta}} \end{bmatrix}, \hat{\mathbf{x}}_t^{pred} = \begin{bmatrix} \dot{x}_{t-N|t} C_{\dot{x}} \\ \dot{y}_{t-N|t} C_{\dot{y}} \\ \dot{\Theta}_{t-N|t} C_{\dot{\Theta}} \end{bmatrix} \tag{8.13}$$

where $\mathbf{x}_t^{ref}$ is received directly from state estimation of the simulator or the real robot. $\hat{\mathbf{x}}_t^{pred}$ are the values received from the UKF equations described in (8.11), and utilizes the dynamic model or the simulator (with the appropriate residual error(s) depending on the test case) from equation (8.1). To clarify the notation, and using Real-to-Dyn as the example (8.4), $\mathbf{x}_t^{ref}$ would be equivalent to $\bar{\mathbf{x}}_t^{real}$, and $\hat{\mathbf{x}}_t^{pred}$ equivalent to $\hat{\mathbf{x}}_t^{real}$.

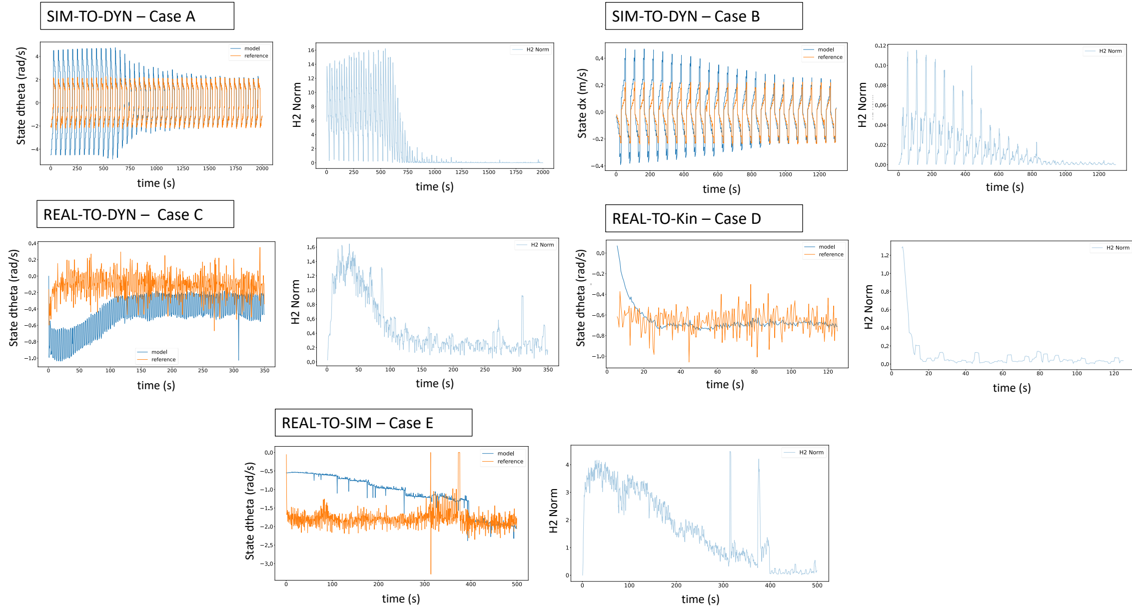


Figure 8.5: **Differential drive robot results** The results of learning the residual error of our two-wheeled robot is shown here and described further in Section 9.4. In all cases, the H2 norm converges to a steady-state value near zero, indicating the residual model could be learned. We do note that for the real robot (Case C-E) we observed very noisy data due to our hardware (e.g., the wheels would stick and slip on the ground) and error-prone localization. By using a low-pass filter on the output of our neural network however, we could generate a more robust convergence even for this difficult setup.

Moreover, we also include a cost term, $C_{\dot{x}}$, $C_{\dot{y}}$, $C_{\dot{\theta}}$, on each component of $\mathbf{x}_t^{ref}$ and $\mathbf{x}_t^{pred}$—these costs can (if desired by the user) bias learning residual errors of certain components over others (e.g., if the robot's reference trajectory is composed of mainly turning in place, putting a higher cost on $\dot{\Theta}$ may be preferable over other components). In this work, we chose a cost of one for all components and test cases.

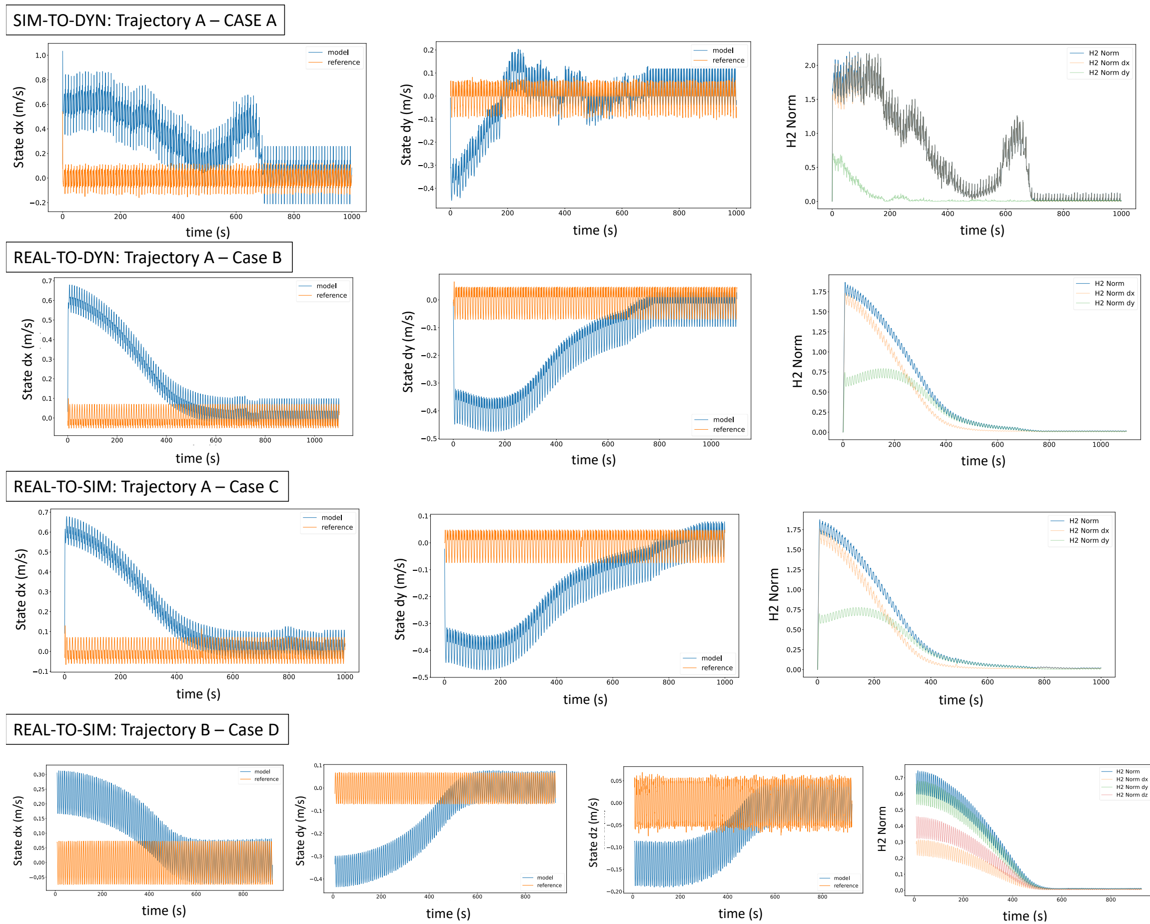


Figure 8.6: **Manipulator arm results.** The results of learning the residual error of our manipulator robot is shown here, and described in more detail in Section 9.4. For the manipulator arm, we use two trajectories (one consists of 2D motion, i.e., Cases A-C, while another demonstrates 3D motion, i.e., Case D). The motion is described and visualized in Fig. 8.4. Overall, the H2 norm decreased for all cases. We also not only show the overall H2 norm but also the H2 norm of each individual component (i.e., $\dot{x}$,$\dot{y}$, and $\dot{z}$). Unlike the case for our wheeled robot, the localization of our manipulator arm was more stable (we used encoders for localization) and did not require any additional filters to our outputs.

8.3.3 Manipulator Robot

After validating our methods on the mobile wheeled robot, we then demonstrate our methods on one of the arms of our manipulator robot SCALER [130]. The arm has 6 degrees of freedom, and is composed of a five-bar linkage combined with a shoulder joint and a spherical wrist joint. Localization for the real robot is done using joint encoders from the motors placed at each joint. For simulating our robot, we use the PyBullet physics engine [39]. Taking advantage of the rigid body dynamics constraint solver in Pybullet, we simulate the manipulator robot with a closed loop kinematics chain. In the simulator, we use the in-built position control to control the actuators, and set the simulation time step to be 0.01 seconds. Our objective will be to add the residual error in task space (i.e., on end-effector velocities). The velocities in task space are then used as part of our estimation of $\hat{\mathbf{x}}_t^{pred}$. As was done in the implementation of the differential drive robot, we compared $\hat{\mathbf{x}}_t^{pred}$ to the reference value $\mathbf{x}_t^{ref}$. Thus, to propagate the states of our manipulator robot, we have:

$$\begin{bmatrix} \dot{x}_t \\ \dot{y}_t \\ \dot{z}_t \end{bmatrix} = \frac{FK(\boldsymbol{\theta}_{t-1} + \dot{\boldsymbol{\theta}}_{t-1}\Delta T) - FK(\boldsymbol{\theta}_{t-1})}{\Delta T} + \delta f(\mathbf{y}_{t-1}) \tag{8.14}$$

where $FK(\boldsymbol{\theta}_{t-1})$ represents the forward kinematics of our manipulator robot with joint angles $\boldsymbol{\theta}_t \in R^{6\times1}$, joint velocities $\dot{\boldsymbol{\theta}}_t \in R^{6\times1}$, and $\dot{x}_t$, $\dot{y}_t$, and $\dot{z}_t$ are the end-effector velocities (a Jacobian can also be used in place of our differentiation). The residual error for the manipulator robot, $\delta(\boldsymbol{y}_t)$, to be predicted by the neural network with the UKF update (see Section 8.2.1), will have the following input $\mathbf{z}_t = [\dot{\theta}_{t,1}, \dot{\theta}_{t,2}, \dot{\theta}_{t,3}, \dot{\theta}_{t,4}, \dot{\theta}_{t,5}, \dot{\theta}_{t,6}]^\top$, where $\dot{\theta}_{t,1} - \dot{\theta}_{t,3}$ are the shoulder joint velocities, and $\dot{\theta}_{t,4} - \dot{\theta}_{t,6}$ are the spherical joint velocities. Similar to equation (8.13) for the differential drive robot, we have the following definitions for $\hat{\mathbf{x}}_t^{pred}$ and $\mathbf{x}_t^{ref}$ (with cost $C = 1$ for our case):

$$\boldsymbol{x}_t^{ref} = \begin{bmatrix} \dot{x}_{t-N|t}^{ref} C_{\dot{x}} \\ \dot{y}_{t-N|t}^{ref} C_{\dot{y}} \\ \dot{z}_{t-N|t}^{ref} C_{\dot{z}} \end{bmatrix}, \hat{\mathbf{x}}_t^{pred} = \begin{bmatrix} \dot{x}_{t-N|t} C_{\dot{x}} \\ \dot{y}_{t-N|t} C_{\dot{y}} \\ \dot{z}_{t-N|t} C_{\dot{z}} \end{bmatrix} \tag{8.15}$$

8.4 Experimental Results

Our results are demonstrated for the mobile wheeled robot in Fig. 8.5 and for our manipulator robot in Fig. 8.6. For both robots, we evaluate the results of our method by comparing the current model (i.e., dynamic or simulator model) with a reference model (simulator model or the real robot) by calculating the H2 norm for each time step during the training process (i.e., defined as $||\mathbf{x}_t^{ref} - \mathbf{x}_t^{pred}||_2$). Thus, if the H2 norm decreases over time and reaches a steady-state value (ideally close to zero), we can assume convergence of the UKF update procedure. In Fig. 8.5, we show that the H2 norm decreases for each test case (in light blue) which is estimated based on the model values (dark blue) and reference values (orange). A and B present the Sim-to-Dyn cases, where in A we use a reference trajectory where the robot spins in place and $\dot{\Theta}$ changes from -2 to 2 rad/s, and in B we use a reference trajectory that drives the robot back and forth in the x-direction (where $\dot{x}$ ranges from -0.2 to 0.2 m/s). The Real-to-Dyn cases are shown in C and D (both used a reference trajectory which only changes the angular velocity). However, we note that for the real robot we faced several issues due to localization and hardware. For example, the localization would at times cause large amplitude spikes when estimating the state, and our wheeled robot was made out of low-cost material causing sticking and slipping behavior (as seen by the orange graph in case C). This noise affected the predicted model produced by our UKF update (graph in blue), which still managed to converge but to a local optima solution (as shown by the offset). One option is to introduce a low-pass filter, which we applied on the predicted model output as seen in Case D (we could have also applied the filter on the localization output instead, however, this would not demonstrate as strong of a case for controller robustness under large uncertainty). With the filter, we show a convergence without offset. Lastly, we test the Real-to-Sim case in E (using the filter as done in D) and with a reference trajectory that imposes a constant angular velocity. Some spikes are observed (likely due to bad localization values as seen in the graph in orange) but was able to converge within approximately 17 minutes.

The results from using our manipulator robot are shown in Fig. 8.6. For cases A - C we used the same reference trajectory as illustrated in the top half or Traj. A of Fig. 8.4 (circular 2D motion and drawing a line through the circle—this trajectory was chosen as transitioning from a circular to linear motion causes additional residual error, which we plan to account for with our methods). Note, that the Real-to-Dyn case (or B) produced more robust results (i.e., less transient errors) compared to the Sim-to-Dyn case (or A). One explanation is that as formulated in equation (8.4), the residual model error trained in the Sim-to-Dyn case is used as part of the formulation in equation (8.4). Thus, this additional knowledge may serve as a good initialization for the learning parameters when training the next test case. Finally, we demonstrate two Real-to-Sim cases (C and D), where C is trained on the same reference trajectory as A-B, and D is trained on a reference trajectory shown in the bottom half or Traj. B of Fig. 8.4 (A circle for the x, and y components and moving up and down in z). In both of these cases, the H2 norm reaches near zero and converges. Lastly, we note that convergence typically occurs in approximately 8 minutes of data, and low-pass filtering on model output was not required for our manipulator robot (due to good localization values through joint encoders).

8.5 Conclusion

In this paper we demonstrated a method that can learn and predict the residual model errors between dynamic/simulator models and the real robot. Approximately 17 minutes of experimental data for the wheeled robot and 8 minutes of experimental data for the manipulator robot was required to achieve convergence (i.e., learn the residual model error). Thus, this method is feasible for employment on hardware and with sparse amounts of data. Although the wheeled robot imposed hardware limitations (i.e., wheels would stick/slide on the surface), and we required low-pass filtering to generate more robust convergence (although convergence was received even without filters), this result showed that experimenting with

filtering techniques as part of the parameter calibration process may be promising to increase robustness. One limitation of this work is that the computation time does scale exponentially with an increased neural network. Some analysis in how to reduce the computation time of the algorithm may be needed, so that we can apply our method for more complex tasks (i.e., modelling residual error while grasping an object while considering contact) and legged systems (i.e., quadrupeds and bipeds), and employing larger neural network structures that consists of more challenging data-types (i.e., vision-based data).

CHAPTER 9

State Estimation of Legged Robots

So far, we have demonstrated algorithms capable of motion planning for legged, wheeled, and aerial robots. Shown that an Unscented Kalman filter can be used to calibrate gains of controllers and planners, and learn residual errors to bridge the real-to-sim gap. However, in all of these applications, some form of dependence on an accurate state estimation system is required. Based on previous literature, there is an opportunity to employ the unification of model-based and learning-based methods towards state estimation as well to correct for the underlying assumptions and errors associated with using only model-based or only learning-based algorithms.

Autonomous legged locomotion typically employs sophisticated planners and/or controllers, whose success and stability are directly dependent on an accurate state estimation system. However, state estimation for legged robots is difficult because dynamic motion produced by footsteps can cause camera motion blur, making vision data less reliable. Due to constantly making and breaking contact, along with slipping, kinematic information can be error prone as well, particularly in environments with large disturbances (i.e., obstacles) or compliant surfaces. Because legged robots produce dynamic motion which demands high frequency control, low computational cost then becomes a necessary design requirement for state estimation systems.

Overall, state estimation methods on legged robots can be subdivided into several design approaches. For example, methods that employ Kalman filtering on either proprioception alone [43], combining proprioception with exteroceptive information [37], using optimization

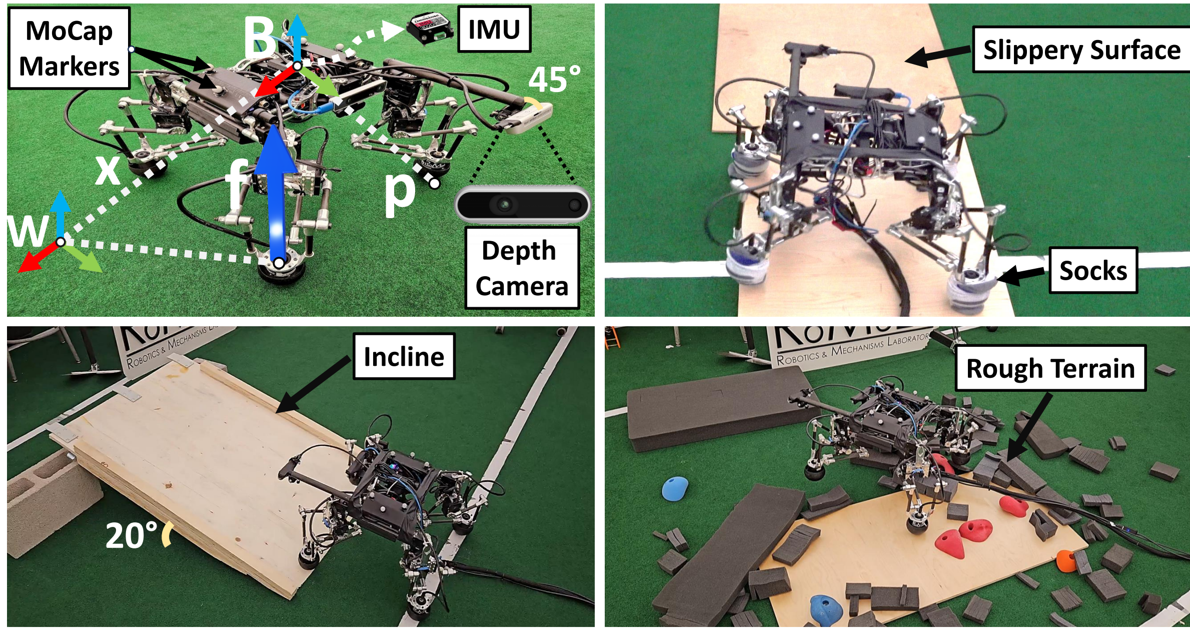


Figure 9.1: Top left shows the attached body frame (B), and world frame (W). The trunk state **x** is in the world frame, while footstep positions **p** is relative to the body. Ground reaction forces (blue arrow) from our MPC control policy are given by **f**, also in world frame. We verify our algorithm, OptiState, on slippery surfaces (top right), incline (bottom left), and rough terrain (bottom right).

[147] or neural networks [19] on some part of the overall estimation framework, and/or with the addition of factor graphs [20].

In this work, we take a hybrid approach that combines model-based Kalman filtering, optimization, and learning methods for state estimation using proprioception and exteroceptive information. The motivation of this approach is to facilitate fast nonlinear modeling in Visual-Inertial Odometry (VIO) estimation while addressing the limitations of pure model-based methods in capturing error from inaccurate state or measurement models, and achieve generalization in learning by incorporating domain knowledge.

Specifically, we use a Kalman filter that takes as input joint encoder and IMU measurements, and employ a single-rigid body model to propagate states by reusing the ground reaction force control outputs from a convex Model Predictive Control (MPC) optimization

[34]. The output of this Kalman filter, along with the state input history over a receding time horizon, is then fed as input to a Gated Recurrent Unit neural network (GRU).

Additionally, the GRU uses as input the latent space representation of depth images through a Vision Transformer (ViT) [57], which helps provide information not only on robot height, but can infer semantic information about its environment to help the estimator. After an offline training phase using a loss function which compares its prediction with the ground truth state output provided by a motion capture system, the goal of the GRU is to update (or correct) the Kalman filter state output to help alleviate the nonlinearities that may exist from noisy measurements and inaccuracies of our single-rigid body model. Using this hybrid approach, we will demonstrate the improved generalization capabilities of our GRU. Specifically, our GRU leverages the Kalman filter's output knowledge and image latent space vector, while remaining robust in predicting state components even in scenarios where our model-based Kalman filter assumptions break down, such as assuming non-slipping conditions and small angle approximations in roll and pitch, see Sec. 9.1.3.

9.1 Methods

The rest of the paper is organized as follows: an introduction to our overall framework, called OptiState, is given in Sec. 9.1.1, the measurements and model we employ in Sec. 9.1.2 and Sec. 9.1.3 respectively, the Kalman filter equations in Sec. 9.1.4, and the GRU to update our Kalman filter estimation and uncertainty in its prediction as well as the ViT to encode the depth image into latent space in Sec. 9.1.5.

9.1.1 Problem Definition

Our state estimation framework estimates the trunk state (robot's CoM) orientation, position, angular, and linear velocity in the world frame. We assume access to a 6-axis IMU, depth camera, joint encoder position and velocity, and during the training procedure of the

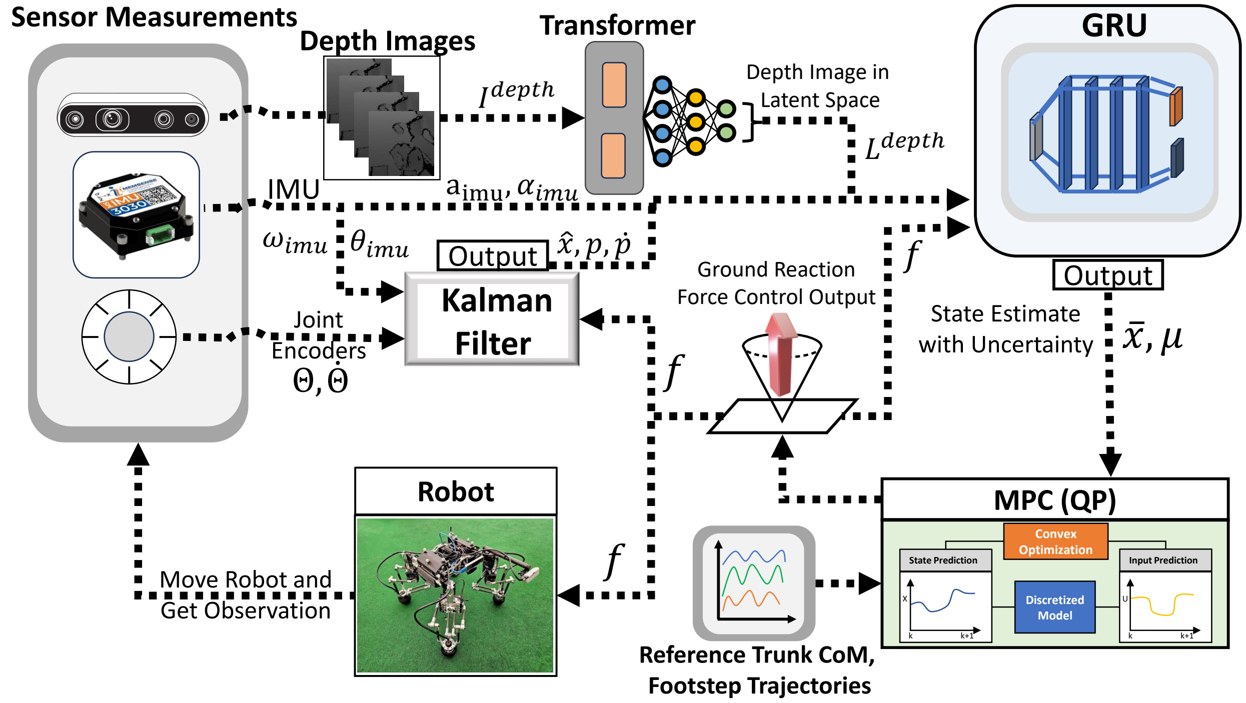


Figure 9.2: Overall state estimation architecture as described in Sec. 9.1.

GRU, access to the ground truth (i.e., motion capture system). The variables used and their size are compiled in Table 9.1. Our framework can be summarized in two parts: first, a Kalman filter is formulated that provides an initial estimate of the robot's trunk state:

$$\hat{\mathbf{x}}_k = \mathrm{KF}(\boldsymbol{\Theta}_k, \dot{\boldsymbol{\Theta}}_k, \boldsymbol{\theta}_k^{imu}, \boldsymbol{\omega}_k^{imu}, n_k^c, \mathbf{f}_k, \hat{\mathbf{x}}_{k-1}) \tag{9.1}$$

where $\boldsymbol{\Theta}_k$, $\dot{\boldsymbol{\Theta}}_k$ are the joint encoder positions and velocities for all feet, $\boldsymbol{\theta}_k^{imu}$, $\boldsymbol{\omega}_k^{imu}$ are the Euler orientation (converted from raw quaternion values) and angular velocity from the IMU, n_k^c is the number of feet currently in contact with the ground given by reference contact matrix $\mathbf{C}$, $\mathbf{f}_k$ are the ground reaction forces generated by an MPC controller formulated identically to [34] (model described in Sec. 9.1.3), k is the current time step, and the output of the filter is $\hat{\mathbf{x}}_k$ or the trunk state of the robot which includes the following order of components: heading angle in roll, pitch, and yaw, position in x, y, and z, angular velocity in roll, pitch, and yaw, and linear velocity in x, y, and z, or $\hat{\mathbf{x}}_k = [\theta_x, \theta_y, \theta_z, r_x, r_y, r_z, \omega_x, \omega_y, \omega_z, v_x, v_y, v_z]^\top$.

Note, because many legged robots [34] use ground reaction force control outputs to sta-

bilize the robot's trunk along a given reference trajectory, we simply can reuse these control outputs within the model of our Kalman filter without re-solving another optimization problem. A key component of our algorithm is that due to our learning module described in Sec. 9.1.5, we do not explicitly require a contact detection algorithm (i.e., n_k^c can be received from the reference instead of explicitly measuring it), as errors associated with our Kalman filter are corrected by the second step described by:

$$\begin{aligned} \mathbf{O}_k = \delta(&\hat{\mathbf{x}}_{k-N:k}, \mathbf{L}_{k-N:k}^{depth}, \mathbf{p}_{k-N:k}, \dot{\mathbf{p}}_{k-N:k}, \\ &\mathbf{a}_{k-N:k}^{imu}, \boldsymbol{\alpha}_{k-N:k}^{imu}, \mathbf{f}_{k-N:k}) \end{aligned} \tag{9.2}$$

where δ is a function describing the GRU (see Sec. 9.1.5), with the following inputs: $\hat{\mathbf{x}}_{k-N:k}$, which is the estimated trunk state from (9.1), $\mathbf{L}_{k-N:k}^{depth}$ is the latent space of the depth image from our ViT described in Sec. 9.1.5, $\mathbf{p}_{k-N:k}$ and $\dot{\mathbf{p}}_{k-N:k}$ are the estimated footstep positions and velocities (obtained from the joint encoder positions and velocities, as explained in Sec. 9.1.2), $\mathbf{a}^{imu}$ and $\boldsymbol{\alpha}^{imu}$ are the IMU linear and angular accelerations respectively (we use these IMU accelerations because they may indicate instances of slipping or sudden falling), $\mathbf{f}_{k-N:k}$ are the same ground reaction forces described in (9.1), the GRU output is given by $\mathbf{O}_k = [\bar{\mathbf{x}}_k, \boldsymbol{\mu}_k]$, where $\bar{\mathbf{x}}_k$ is the updated trunk state and $\boldsymbol{\mu}_k$ is the absolute error between the GRU prediction and ground truth ($\mathbf{x}^{mocap}$) written as $|\bar{\mathbf{x}}_k - \mathbf{x}_k^{mocap}|$, and we use N time steps to indicate the recurrent nature of the GRU. We include ground reaction forces as inputs as they contain information on the reference trajectory and contact sequence from the MPC optimization, and depth images to convey robot height and environmental semantics in our training procedure.

The main motivation for our use of the Kalman filter within our learning algorithm is the following: (1) The filter offers a reasonable initial estimation of the state, serving as a warm-start for our network and leverages the received domain knowledge, useful for generalization. We substantiate this claim through an ablation study in which we exclude the Kalman filter state as input to the GRU, resulting in a noticeable decline in overall performance (refer to Section 9.2.1); (2) As we also acquire knowledge of the uncertainty associated with

the GRU prediction, we can identify estimates that may carry significant errors. In such instances, users can rely on the Kalman filter estimate, as it remains unaffected by any learning components.

9.1.2 Measurements

For our Kalman filter described in Sec. 9.1.4, we use IMU measurements that provide the Euler angles of the trunk (converted from raw quaternion) $\boldsymbol{\theta}^{imu} = [\theta_x, \theta_y, \theta_z]^\top$ (roll, pitch, and yaw), and corresponding angular velocity $\boldsymbol{\omega}^{imu} = [\omega_x, \omega_y, \omega_z]^\top$. To measure the linear velocity of the trunk we first convert joint encoder position ($\boldsymbol{\Theta}$) and velocity ($\dot{\boldsymbol{\Theta}}$) using the Jacobian ($\mathbf{J}$) into footstep velocities or $\dot{\mathbf{p}} = \mathbf{J}(\boldsymbol{\Theta})\dot{\boldsymbol{\Theta}}$. Leg odometry can then measure linear velocity of the trunk, $\mathbf{v}^{odom} = [v_x, v_y, v_z]^\top$, and robot height, r_z^{odom}, using the following relationship:

$$\mathbf{v}^{odom} = -\frac{1}{n_c}\sum_{i=1}^{n}\mathbf{R}_b^w(\dot{\mathbf{p}}^i + \boldsymbol{\omega}^{imu} \times \mathbf{p}^i) \text{ (foot } i \text{ in contact)} \tag{9.3}$$

$$r_z^{odom} = -\frac{1}{n_c}\sum_{i=1}^{n} p_z^i \text{ (foot } i \text{ in contact)} \tag{9.4}$$

where n is the number of feet, and n_c is the number of feet in contact with the ground, and p_z is the height component of the footstep (received from forward kinematics of joint encoder position), and $\mathbf{R}_b^w$ is the rotation matrix from body to world frame. Note, this measurement assumes no foot slippage. Lastly, for the training module described in Sec. 9.1.5, we will measure motion capture data symbolized by $\mathbf{x}^{mocap}$, and the raw depth image as $\mathbf{I}^{depth}$.

9.1.3 Model

The model used for our Kalman filter is based on the single-rigid body that is subject to forces at a contact patch. While ignoring leg dynamics and assuming small angle approximations in roll and pitch [34] does simplify the actual robot dynamics, we argue that the decreased

Table 9.1: Notation of Critical variables. Frames are defined in either world frame (W), body frame (B), or neither (N/A)

Name	Description	Size	Frame
$\mathbf{x}$	Trunk State from System Model	$R^{12\times1}$	W
$\hat{\mathbf{x}}$	Trunk State from KF	$R^{12\times1}$	W
$\bar{\mathbf{x}}$	Trunk State from GRU	$R^{12\times1}$	W
$\boldsymbol{\mu}$	Trunk State GRU Prediction Error	$R^{12\times1}$	W
$\boldsymbol{\theta}$	Trunk Orientation	$R^{3\times1}$	W
$\mathbf{r}$	Trunk Position	$R^{3\times1}$	W
$\boldsymbol{\omega}$	Trunk Angular Velocity	$R^{3\times1}$	W
$\mathbf{v}$	Trunk Linear Velocity	$R^{3\times1}$	W
$\mathbf{p}$, $\dot{\mathbf{p}}$	Footstep Position, Velocity	$R^{12\times1}$	B
i	Foot Number	R^1	N/A
k	Time Step	R^1	N/A
n, n_c	# of Feet, # of Feet in contact	R^1	N/A
C^i	Reference Contact of Foot i	R^1	N/A
$\mathbf{f}^i$	Ground Reaction Force	$R^{3\times1}$	W
$\boldsymbol{\Theta}^i$, $\dot{\boldsymbol{\Theta}}^i$	Joint Encoder Position, Velocity	$R^{3\times1}$	N/A
$\mathbf{L}^{depth}$	Latent Space of Depth Image	$R^{128\times1}$	N/A
$\mathbf{I}^{depth}$	Raw Depth Image	$R^{224\times224}$	N/A
$\hat{\mathbf{I}}$	Inertia Tensor of Trunk	$R^{12\times12}$	W
$\mathbf{I}_n$	Identity Matrix	$R^{n\times n}$	N/A

computational cost outweighs this issue. The added error due to the model's simplification can also be accounted for by the introduction of our learning module discussed in Sec. 9.1.5. Simplifying this model's nonlinear dynamics through approximations [34] and expressing it in discrete-time form yields:

$$\mathbf{x}_{k+1} = \mathbf{Dyn}(\mathbf{x}_k, \mathbf{f}_k) \tag{9.5a}$$

$$\mathbf{Dyn}(\mathbf{x}_k, \mathbf{f}_k) = \begin{bmatrix} \boldsymbol{\theta}_k + \Delta T \mathbf{R}_b^{w\top} \boldsymbol{\omega}_k \\ \mathbf{r}_k + \Delta T \mathbf{v}_k \\ \boldsymbol{\omega}_k + \Delta T (\sum_{i=1}^{n} \hat{\mathbf{I}}^{-1} \left[\mathbf{p}_{i,k}^b\right]_\times \mathbf{f}_k^i) \\ \mathbf{v}_k + \Delta T (\sum_{i=1}^{n} \frac{\mathbf{f}_k^i}{m} + \mathbf{g}) \end{bmatrix} \tag{9.5b}$$

where $\mathbf{x}_k$ is the trunk state of the robot, the ground reaction forces are represented by $\mathbf{f}_k^i$ for each foot i, m is the total mass of the robot, $\mathbf{g}$ is the gravity vector, $\hat{\mathbf{I}}$ is the inertia tensor matrix in world coordinates, $\mathbf{p}_{i,k}^b$ is the footstep position of foot i in the body frame, ΔT is the discretized time step, and lastly, $[\bullet]_\times$ is defined as the skew-symmetric matrix (same definition as used in [34]). We can rewrite (9.5):

$$\mathbf{x}_{k+1} = (\mathbf{I}_{12} + \mathbf{A}_k \Delta T)\mathbf{x}_k + (\mathbf{B}_k \Delta T)\mathbf{f}_k + \mathbf{g}\Delta T \tag{9.6}$$

where $\mathbf{I}_{12} \in R^{12\times 12}$ is an identity matrix (not to be confused notationally with the inertia tensor matrix $\hat{\mathbf{I}}$), $\mathbf{f}_k \in R^{3n\times 1}$ is the ground reaction force vector (consisting of the x, y, and z components of force), stacked vertically for each of the n feet, and $\mathbf{g} \in R^{12\times 1}$ is the gravity vector, or $\mathbf{g} = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -9.81]^\top$, and $\mathbf{A}_k \in R^{12\times 12}$ and $\mathbf{B}_k \in R^{12\times 3n}$ represent the discrete-time system dynamics:

$$\mathbf{A}_k = \begin{bmatrix} \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{R}_{b,k}^{w\top} & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{I}_3 \\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 \end{bmatrix} \tag{9.7}$$

$$
\mathbf{B}_k = \begin{bmatrix} \mathbf{0}_3 & \dots & \mathbf{0}_3 \\ \mathbf{0}_3 & \dots & \mathbf{0}_3 \\ \hat{\mathbf{I}}^{-1}\left[\mathbf{p}^b_{1,k}\right]_\times & \dots & \hat{\mathbf{I}}^{-1}\left[\mathbf{p}^b_{n,k}\right]_\times \\ \mathbf{I}_3/m & \dots & \mathbf{I}_3/m \end{bmatrix} \tag{9.8}
$$

During operation of the robot, the MPC QP outputs ground reaction forces of the feet in contact with the ground, which are assigned into joint motor torques. These outputs enable the robot trunk to follow a reference state over a time horizon [34]. Thus, we can simply use these forces with the trunk state and footstep positions at time step k in the model described by (9.5), to predict the trunk state at time step $k+1$.

9.1.4 Kalman Filter

From Sections 9.1.2 and 9.1.3, we have the necessary information to use the Kalman filter equations, which are:

9.1.4.1 Prediction

$$
\hat{\mathbf{x}}_{k|k-1} = \mathbf{Dyn}(\mathbf{x}_{k-1|k-1}, \mathbf{f}_{k-1|k-1}) \tag{9.9}
$$

$$
\mathbf{P}_{k|k-1} = \mathbf{F}_k \mathbf{P}_{k-1|k-1} \mathbf{F}_k^\top + \mathbf{Q} \tag{9.10}
$$

9.1.4.2 Update

$$
\hat{\mathbf{y}}_k = \mathbf{z}_k - \mathbf{H}\hat{\mathbf{x}}_{k|k-1} \tag{9.11}
$$

$$
\mathbf{S}_k = \mathbf{H}\mathbf{P}_{k|k-1}\mathbf{H}^\top + \mathbf{R} \tag{9.12}
$$

$$
\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}\mathbf{S}_k^{-1} \tag{9.13}
$$

$$
\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k \hat{\mathbf{y}} \tag{9.14}
$$

$$
\mathbf{P}_{k|k} = (\mathbf{I}_{12} - \mathbf{K}_k \mathbf{H})\mathbf{P}_{k|k-1} \tag{9.15}
$$

where $\mathbf{F}_k = e^{(\mathbf{A}_k \Delta T)}$, $\mathbf{z}_k \in R^{10\times1}$ are the measurements received directly from the heading angles of the IMU and odometry (see Sec. 9.1.2) written as $\mathbf{z}_k = [\boldsymbol{\theta}_k^{imu}, r_{k,z}^{odom}, \boldsymbol{\omega}_k^{imu}, \mathbf{v}_k^{odom}]^\top$. Thus, $\mathbf{H} \in R^{10\times12}$, selecting the components of $\hat{\mathbf{x}}_k$ that match the components of $\mathbf{z}_k$, or all components except for r_x and r_y horizontal trunk position.

9.1.5 Improving Estimation through Learning

A ML approach is employed to consider nonlinearities associated with model and measurement error to better estimate the state of the robot's trunk (see Fig. 9.3). The approach includes training a ViT and then using the ViT model to train a GRU offline, before employing them simultaneously in the testing phase online. We first collect the following data used for our learning modules (the robot moved randomly in all terrains except for slippery and inclined surfaces, where it was commanded to move straight): the ground truth state of the robot's trunk using a motion capture system or $\mathbf{x}^{mocap}$, joint positions and velocities, IMU data or $\mathbf{a}^{imu}$ and $\boldsymbol{\alpha}^{imu}$, depth images or $\mathbf{I}^{depth}$, control outputs of our MPC controller, or $\mathbf{f}$ (see Sec. 9.1.3), and foot contact reference matrix $\mathbf{C}$. We can then make use of (9.1) and the data collected to calculate the output of the Kalman filter, or $\hat{\mathbf{x}}$, at each time step during collection. We use an autoencoder to convert the raw depth image $\mathbf{I}^{depth}$ to the latent space $\mathbf{L}^{depth}$ through a ViT architecture similar to [57]. By applying various operations such as rotation, flipping, and zooming on the collected depth images (to facilitate generality of the images for training), we convert these depth images into grayscale and then input them to the encoder. The encoder consists of a patch size equal to 16, embedding dimension of 128, depth of 4, multi-layer perception ratio of 4, and number of heads equal to 4. The decoder consists of these same hyperparameters. The AdamW [88] optimizer is used with weight decay of 0.1, learning rate of $4e^{-4}$, and $\beta_1 = 0.90$, $\beta_2 = 0.95$. The Mean Squared Error (MSE) loss function is used to minimize the error between the original depth image, and the reconstructed depth image from the output of our decoder. After training, we use the encoder output to represent the latent space of our depth image ($\mathbf{L}^{depth}$).

We then use (9.2) to predict the state of the robot's trunk using the GRU (δ), which requires the Kalman filter estimate of the trunk, latent space of the depth image, odometry (i.e., $\mathbf{p}$ and $\dot{\mathbf{p}}$), IMU, and ground reaction forces as input, over a receding horizon of N time steps (we chose $N = 10$). δ also outputs the error between the predicted state output and the ground truth, or $\boldsymbol{\mu} = |\bar{\mathbf{x}} - \mathbf{x}^{mocap}|$, which is learned during training. In other words, $\boldsymbol{\mu}$ helps provide an indicator of uncertainty of our GRU model. This uncertainty may be useful in a stochastic control/planning framework, or used as an indicator to employ the estimate from the Kalman filter instead of the GRU model in certain cases (i.e., $\hat{\mathbf{x}}$ instead of $\bar{\mathbf{x}}$). The GRU uses a learning rate of $1e^{-5}$, number of hidden layers of 4 with each layer of size 128, batch size of 64, Adam optimizer with weight decay of $1e^{-5}$, and with MSE as our loss function. We also normalize the input and output data from 0 to 1 (using min/max of dataset) to help stabilize the training. See (**B**) and (**C**) in Fig. 9.3 for the training loss of the autoencoder and GRU respectively.

9.2 Experimental Validation

For data collection, we used six VICON Vero v2.2 cameras running at 330 Hz to estimate the ground truth trunk state of our SCALER robot [131]. We compare OptiState with VIO SLAM from the the Intel RealSense T265 camera [56] running at 200 Hz. For collecting depth images, we use the Intel RealSense D435 camera pointed at a 45 angle to the ground, running at 60 Hz. Our encoder runs up to 500 Hz, which is well above the frequency of our camera. Note, our GRU estimator (i.e., up to 1600 Hz) is limited by the frequency of our motion capture system (i.e., 330 Hz), where lower frequency components are simply repeated during the training procedure. Evaluation was done on an Intel i7 8850H CPU with a Quadro 3200 GPU. As the T265 camera is our main baseline, and to compare this baseline with our architecture, we down-sample our GRU estimation to 200 Hz, with ΔT=0.005 s when plotting results.

9.2.1 Results

Overall, we collected 16 different trajectories for training the GRU using a trot gait, where each trajectory lasted between 1 to 2 minutes (total of ≈ 288,000 data points) and included flat, slippery, incline, and rough terrain. See Fig. 9.1 and accompanied video for the robot motion during collection. After training, we evaluate on four trajectories previously unseen during training (one trajectory per surface). The results of the testing phase are shown in **(A)**, **(B)** and **(C)** of Fig. 9.4. Note that in **(A)**, we display only a portion of the four trajectories and baseline comparison for easier visibility. However, in **(B)**, we computed the Root Mean Squared Error (RMSE) between the ground truth and the estimation algorithm over the entire evaluation testing set, which totals 72,000 data points at ΔT=0.005 s. In **(A)**, we see that both OptiState and VIO SLAM performed well, though VIO SLAM struggled more with predicting linear velocity and, notably, the robot's z height which becomes apparent during the incline and especially on rough terrain. This may be due to the fact that our rough terrain contains compliant surfaces, which can easily disrupt estimation systems and produce drift particularly in z height.

In **(B)**, we compared our estimation algorithm, OptiState, under three different conditions: one without the Kalman filter state $\hat{\mathbf{x}}$ as input to the GRU, another without the vision component or $\mathbf{L}^{depth}$ as input to the GRU, and using the Kalman filter alone as described in Sec. 9.1.4. We also included a baseline comparison with the VIO SLAM method (T265). In **(C)**, we compared RMSE percent improvements over our VIO SLAM baseline for each state component, with the average percentage across all state components shown in the right-most column. OptiState's percent improvement was generally higher by not excluding the Kalman filter state or depth image input, except for $\dot{\mathbf{y}}$, confirming the importance of including these inputs into our GRU. OptiState outperformed both the Kalman filter and VIO SLAM for all components. The Kalman filter had the poorest performance at average RMSE improvement of -52%, while OptiState performed the best at 65%. The Kalman filter's errors may be due to slippery and rough terrain conditions, which can violate the model's assumptions

(see Sec. 9.1.3). However, the Kalman filter showed less error in velocities. Note that during evaluation, the trace of $\mathbf{P}$ (covariance matrix) of the Kalman filter never diverged. The deviation between trace values throughout the entire testing trajectories remained within a small range, always less than $1e^{-5}$. In Fig. 9.5, we show one example on how $\boldsymbol{\mu}$, which predicts the absolute error between the GRU estimate and the ground truth, can highlight potential prediction errors. It correctly predicts higher uncertainty when the GRU estimate is less accurate.

9.3 Limitations and Conclusion

In this work, we demonstrated state estimation of legged robots using joint encoder information, IMU measurements, a ViT autoencoder, and outputs of a Kalman filter that reuses the control forces from MPC in its system model, wrapped within a GRU framework. In addition to predicting the trunk state of the robot, we also showed that we can predict the uncertainty or error of the GRU's prediction. Although OptiState showed improved performance for all states compared to VIO SLAM, there are several limitations to using our approach. First, although the GRU can reach up to 1600 Hz, we must note that the quality and frequency of our estimation is predicated on the ground truth setup (i.e., in our setup, up to 330 Hz). Because we employed a motion capture system for ground truth with a limited training space, accurate prediction of x and y world coordinate positions is challenging when the robot goes beyond this space—although velocity components can be predicted normally. In these cases, the user may integrate the velocity components of the GRU predictions to get a more accurate position, although accumulation of drift would occur. Overall, we have created a robust state estimation system for legged robots using a hybrid ML-Kalman filter approach, and emphasize ground truth motion capture collection as a key area of future improvement.

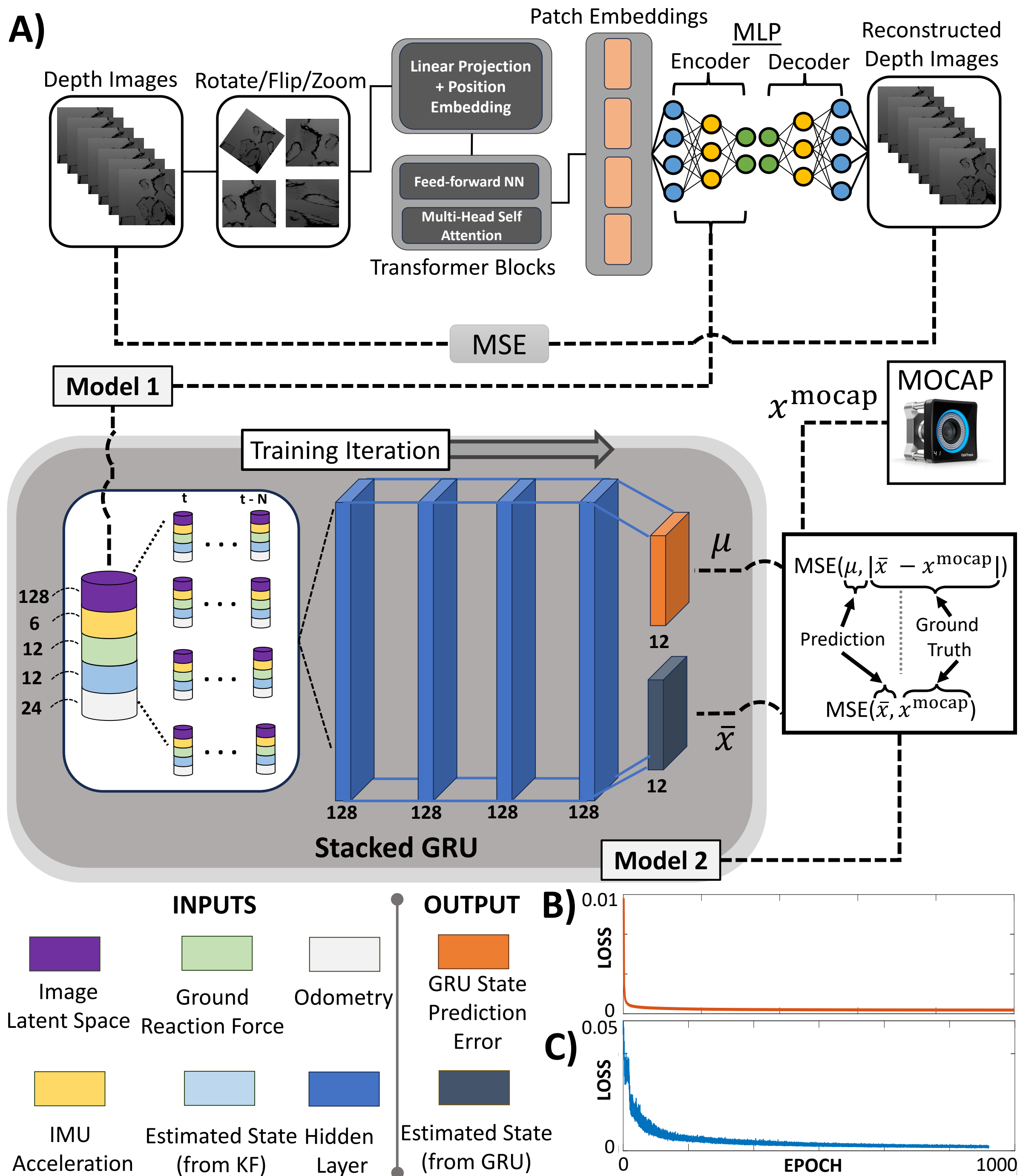
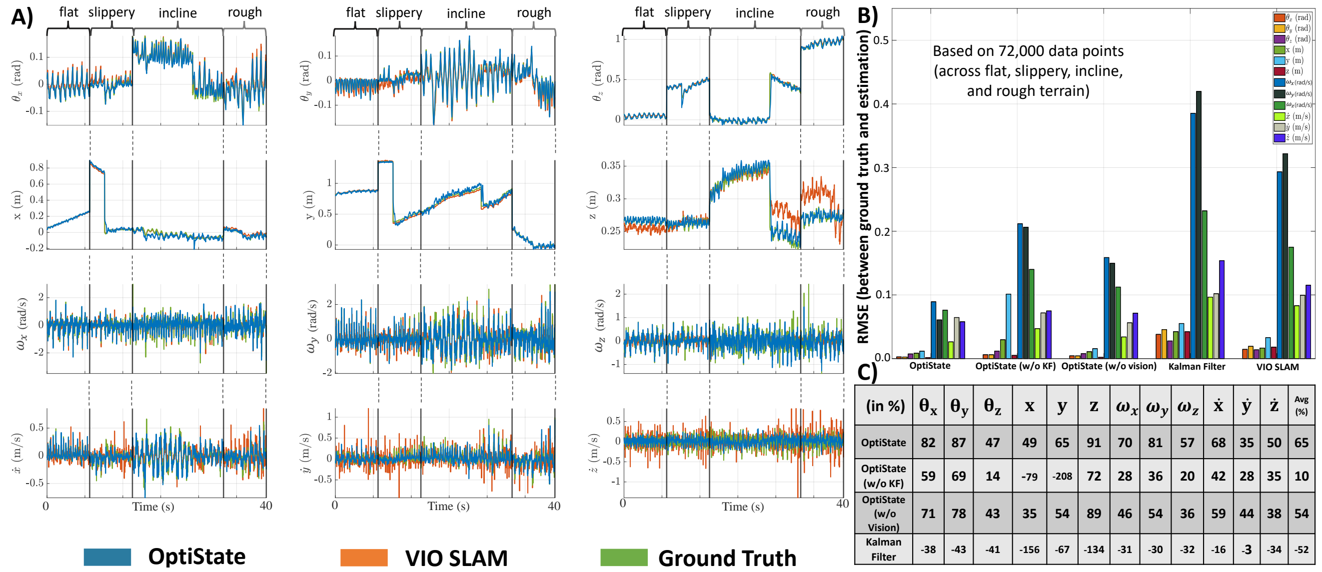


Figure 9.3: Transformer and Gated Recurrent Unit (GRU) network architecture. From (**A**), model 1 is the transformer model, model 2 is the GRU (δ) that predicts the robot's trunk state and uncertainty of its own prediction. Input/output state and hidden layer sizes indicated by the numbers. Training loss of model 1 shown in (**B**) and for model 2 in (**C**). MSE is the loss function (see Sec. 9.1.5).



(in %)	θ_x	θ_y	θ_z	x	y	z	ω_x	ω_y	ω_z	$\dot{x}$	$\dot{y}$	$\dot{z}$	Avg (%)
OptiState	82	87	47	49	65	91	70	81	57	68	35	50	65
OptiState (w/o KF)	59	69	14	-79	-208	72	28	36	20	42	28	35	10
OptiState (w/o Vision)	71	78	43	35	54	89	46	54	36	59	44	38	54
Kalman Filter	-38	-43	-41	-156	-67	-134	-31	-30	-32	-16	-3	-34	-52

Figure 9.4: Results during the online testing phase, as described in Sec. 9.2.1. In (**A**) we show the state estimation for all state components from OptiState, VIO SLAM, and the ground truth. We show 4 distinct trajectories connected by solid lines to symbolize the various terrain under evaluation, such as flat, slippery, incline, and rough terrain. The RMSE results over all 4 trajectories and per state component are shown in (**B**), and includes OptiState without the Kalman filter input, or vision input, and the Kalman filter alone. Lastly, we show the percentage improvement of RMSE over the VIO SLAM baseline for each state in (**C**) per estimation algorithm shown in the first column.

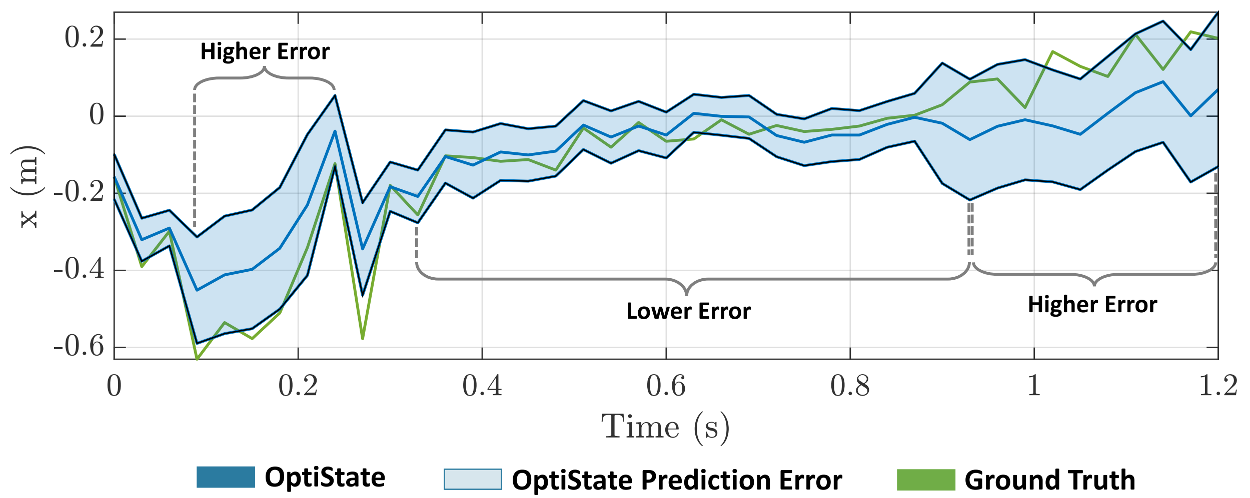


Figure 9.5: Example of predicting the uncertainty of the GRU's (OptiState) x position, or μ_x. The shaded blue represents $\bar{x} \pm \mu_x$.

CHAPTER 10

Current and Future Work, and Conclusion

10.1 Modification of Auto-Tuning

Although the auto-tuning framework can facilitate tuning controller/planning parameters, once we have too many parameters to optimize over, the computation can get significant. Additionally, the covariance matrices Q, and R associated with the UKF must be properly initialized for robust and converging performance. For example, the scale of these matrices can directly determine the aggressiveness of the parameter calibration, and also the relative selection of which parameters to prioritize in the update. If not initialized correctly, the Auto-tuner may diverge. Knowing the most optimal initialization, or perhaps even changing these parameters online depending on the current situation, may be an interesting avenue for future work.

10.2 Resolving Model Complexity with Reinforcement Learning

So far, in this dissertation, we mainly used model-based methods for control, such as model predictive control. In the majority of cases, this perfectly suffice for most robot types. However, in my experience, there can be cases where it becomes very challenging to employ model-based control on increasingly difficult dynamical systems. For instance, we developed a new robot called SCALER-B, which is a modification of our climbing robot SCALER [131], and is now capable of not only climbing tasks, but locomotion, manipulation, and grasping through its multi-modal design. This multi-modality enables the robot to go into biped mode,

where it has two larger forearms for manipulation and grasping tasks. Additionally, the robot can go into quadruped mode, being able to crawl and even peforming rolling motion (see Fig. 10.1). Although SCALER-B has such multi-modal capability, it is not inherently designed for only say, bipedal motion. Thus, the dynamics of this robot is incredibly challenging to model, and I found that model-based control such as MPC is difficult to employ on this particular system. Due to this, using reinforcement learning is a viable potential technology for this system. As some simulators such as Isaac Gym and more recently Mujoco with MJX are becoming capable of training using a GPU with parallel environments, several works [96, 65] have shown enormous success in quickly training legged robot locomotion to overcome challenging terrain. These algorithms can be applied zero-shot through sufficient domain randomization. I now present a typical set of rewards used to do both planning and control of our SCALER-B robot for bipedal locomotion:

10.2.1 Reward Equations

To design an effective reward function for reinforcement learning, we need to consider various aspects of the agent's behavior. Below are the reward equations for different aspects of the agent's performance, including penalizing undesired behaviors and rewarding desired outcomes.

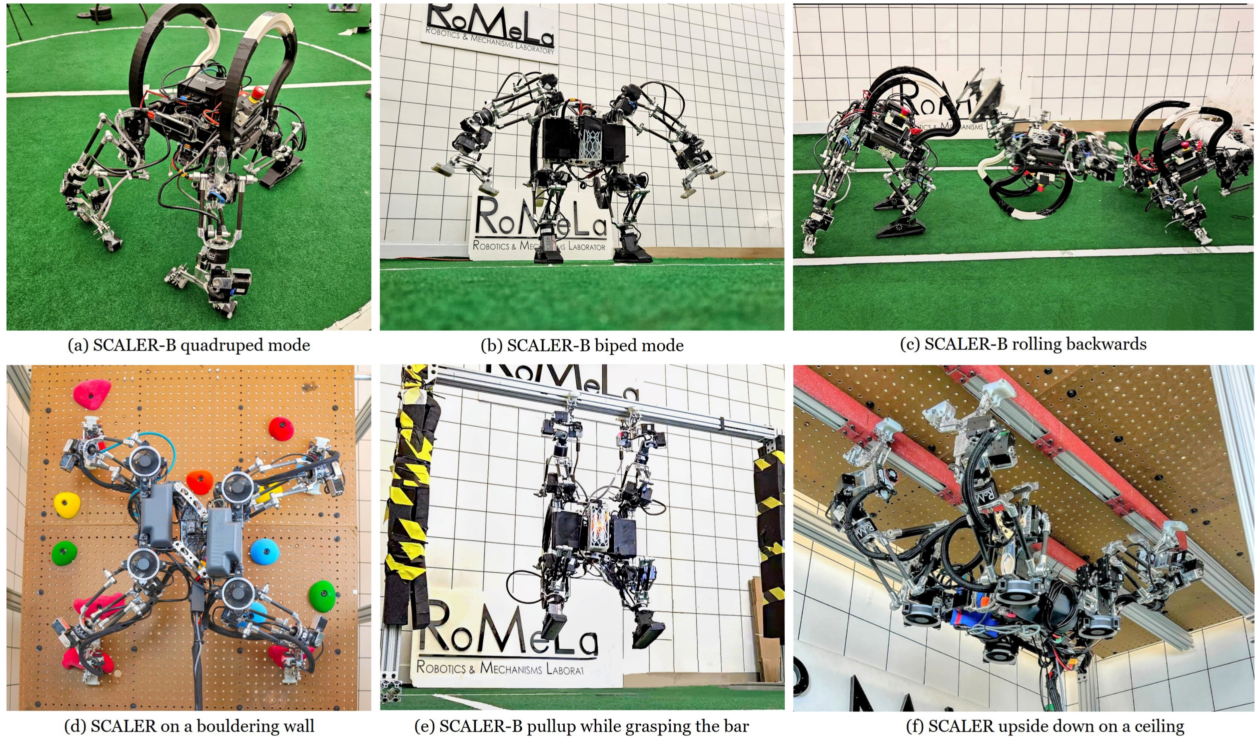
10.2.1.1 Penalize Velocity in z

$$R_{v_z} = -\alpha_v \left| v_z \right|$$

where v_z is the velocity in the z-direction and α_v is a penalty coefficient.

10.2.1.2 Penalize Angular Velocity in x-y

$$R_{\omega_{xy}} = -\alpha_\omega \left(|\omega_x| + |\omega_y| \right)$$



(a) SCALER-B quadruped mode
(b) SCALER-B biped mode
(c) SCALER-B rolling backwards
(d) SCALER on a bouldering wall
(e) SCALER-B pullup while grasping the bar
(f) SCALER upside down on a ceiling

Figure 10.1: **SCALER-B.** We show the hardware and motion capabilities of our SCALER-B robot, capable of (a) quadruped mode, (b) biped mode, (c) rolling, (d) climbing vertically, (e) pull-up, and (f) ceiling.

where ω_x and ω_y are the angular velocities around the x- and y-axes, respectively, and α_ω is a penalty coefficient.

10.2.1.3 Reward Tracking Orientation

$$R_\theta = \beta_\theta \left(1 - \|\theta_{\text{desired}} - \theta\|\right)$$

where θ_{desired} is the desired orientation, θ is the current orientation, and β_θ is a reward coefficient.

10.2.1.4 Penalize Torques

$$R_\tau = -\alpha_\tau \sum_{i=1}^{n} |\tau_i|$$

where τ_i are the torques applied at each joint, and α_τ is a penalty coefficient.

10.2.1.5 Penalize Change in Actions

$$R_{\Delta a} = -\alpha_{\Delta a} \sum_{i=1}^{n} |a_{i,t} - a_{i,t-1}|$$

where $a_{i,t}$ is the action at time step t for joint i, $a_{i,t-1}$ is the action at the previous time step, and $\alpha_{\Delta a}$ is a penalty coefficient.

10.2.1.6 Reward Tracking Linear Velocity

$$R_{v_{\text{linear}}} = \beta_{v_{\text{linear}}} (1 - \|v_{\text{desired}} - v_{\text{linear}}\|)$$

where v_{desired} is the desired linear velocity, v_{linear} is the current linear velocity, and $\beta_{v_{\text{linear}}}$ is a reward coefficient.

10.2.1.7 Reward Tracking Angular Velocity

$$R_{v_{\text{angular}}} = \beta_{v_{\text{angular}}} (1 - \|\omega_{\text{desired}} - \omega\|)$$

where ω_{desired} is the desired angular velocity, ω is the current angular velocity, and $\beta_{v_{\text{angular}}}$ is a reward coefficient.

10.2.1.8 Reward Feet Air-Time

$$R_{\text{air-time}} = \beta_{\text{air-time}} \sum_{i=1}^{n} T_{i,\text{air}}$$

where $T_{i,\text{air}}$ is the air time for foot i, and $\beta_{\text{air-time}}$ is a reward coefficient.

10.2.1.9 Reward Stand Still

$$R_{\text{stand-still}} = \beta_{\text{stand-still}} I\{v_{\text{linear}} = 0\}$$

where I is the indicator function that returns 1 when $v_{\text{linear}} = 0$ and 0 otherwise, and $\beta_{\text{stand-still}}$ is a reward coefficient.

10.2.1.10 Penalize Foot Slip

$$R_{\text{foot-slip}} = -\alpha_{\text{foot-slip}} \sum_{i=1}^{n} |v_{i,\text{foot}}|$$

where $v_{i,\text{foot}}$ is the slipping velocity of foot i, and $\alpha_{\text{foot-slip}}$ is a penalty coefficient.

10.2.1.11 Total Reward

The total reward R_{total} is the sum of all individual rewards and penalties:

$$R_{\text{total}} = R_{v_z} + R_{\omega_{xy}} + R_\theta + R_\tau + R_{\Delta a} + R_{v_{\text{linear}}} + R_{v_{\text{angular}}} + R_{\text{air-time}} + R_{\text{stand-still}} + R_{\text{foot-slip}}$$

Using Mujoco [135] but with the capability to run in parallel environments on a single GPU (Mujoco with MJX), we show that the agent is capable of learning both planning and control while following a desired joystick velocity command, see Fig. 10.2 and Fig. 10.3. Training time was approximately 10 minutes using PPO, which results in $300,000$ timesteps. This result is quite remarkable considering the difficulty of applying MPC to such a complex dynamical system. With reinforcement learning however, we were able to demonstrate a working result, as the motion can be learned through experience and using the model based on STL files with more accurate dynamics.

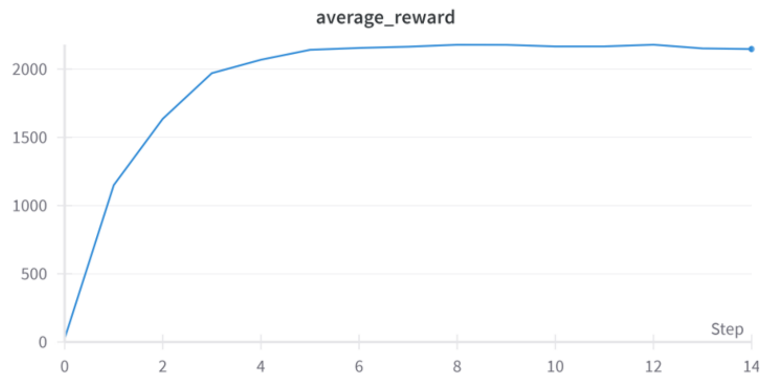


Figure 10.2: **Average Reward Results.** We show how the average rewards during training converges.

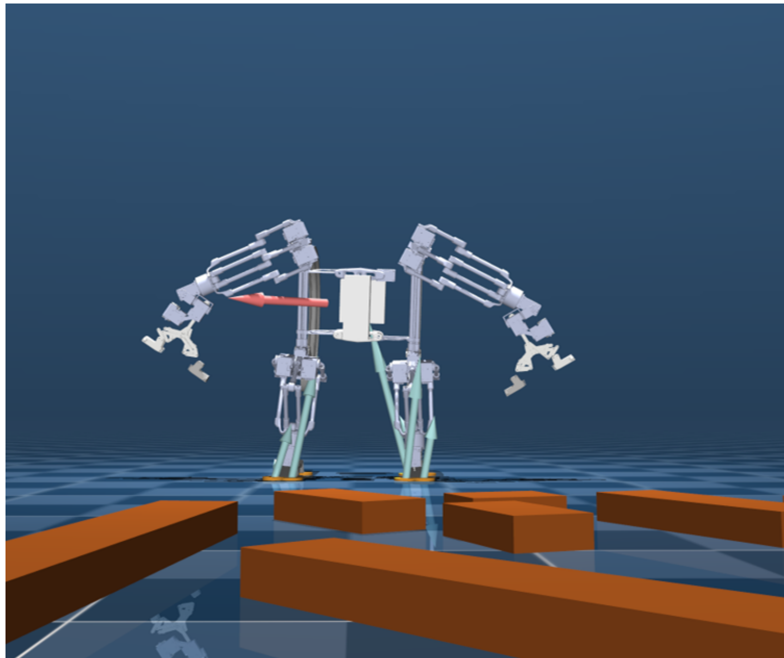


Figure 10.3: **Mujoco with MJX simulator.** We show a disturbance being applied while the robot is doing locomotion, as represented by the pink-red arrow. The green-blue arrows signify the ground reaction forces.

10.3 Rise of Large Language and Vision Models

Finally, I conclude this dissertation with some comments on Large Language Models (LLM) and Vision Language Models (VLM). With the rise of GPT models, it is becoming indisputable that they are having a large impact on our daily lives – and this includes robotics [86, 128]. How far this technology will go in affecting robotic research is currently unknown, although in the next decade from the publication of this thesis, we may soon have an answer or at least a better idea of it. Nevertheless I can only postulate, in the reference frame of the contents of this thesis, how VLM and LLM can potentially affect the Active SLAM field.

10.3.0.1 Enhanced Perception and Understanding

VLMs integrate visual and language processing, improving the robot's ability to interpret complex environments. This integration leads to better recognition and categorization of objects and scenes, resulting in more detailed maps.

10.3.0.2 Semantic Mapping

VLMs enable the inclusion of semantic information in the maps created by Active SLAM. Instead of producing purely geometric maps, robots can generate semantic maps that provide contextual details about objects and their relationships. For instance, a VLM can help a robot distinguish between a chair, a table, and a person, thereby enriching the spatial data with meaningful context. Critically, VLM can make this decision based on a very general model - unlike typical object detection system which would require pre-trained models on known objects.

10.3.0.3 Improved Decision-Making

In Active SLAM, the robot must decide where to move next to enhance its map and localization. VLMs can improve this decision-making process by providing a deeper understanding of the environmental context and identifying the most informative areas for exploration, leading to more efficient exploration strategies. For example, it is very difficult to train a basic neural network model to actually understand that a red stop sign means the vehicle needs to stop, unless it is specifically programmed to do so and trained on images of stop signs. A LLM in combination with a VLM can first recongize the stop sign, and then use its contextual understanding to know the vehicle needs to stop.

10.3.0.4 Natural Language Interaction

VLMs facilitate natural language understanding and generation, enabling intuitive human-robot interaction. In Active SLAM, this capability can be used for issuing high-level commands, requesting clarifications, or giving real-time updates. For example, a user might instruct the robot to "map the kitchen" or "avoid the crowded area," and the robot can comprehend and act on these instructions.

10.3.0.5 Cross-Modal Integration

By combining visual and linguistic data, VLMs offer richer contextual interpretations of the environment. In Active SLAM, this integration helps the robot understand annotations, signs, or instructions within the environment, adding valuable layers of information that purely visual or language-based models might overlook.

10.3.0.6 Learning from Language

VLMs can leverage large datasets of visual and textual information for pre-deployment learning about different environments and objects. This pre-training provides the robot with extensive knowledge, enhancing initial performance and reducing the training time required in new environments.

10.3.0.7 Context-Aware Planning

The contextual awareness provided by VLMs can improve planning and navigation strategies in Active SLAM. For instance, understanding that a kitchen is likely to contain obstacles such as tables and chairs can help the robot plan more efficient paths.

10.3.0.8 Challenges and Considerations

While VLMs offer numerous advantages for Active SLAM, several challenges need to be addressed:

- **Computational Complexity:** Integrating VLMs into SLAM systems requires significant computational resources, which may be challenging for real-time applications.
- **Robustness and Reliability:** Ensuring reliable performance of VLMs in diverse and dynamic real-world environments is crucial for practical deployment.

VLMs have the potential to significantly enhance Active SLAM by providing richer semantic understanding, improving decision-making, enabling natural language interaction, and offering context-aware planning. However, these benefits come with challenges that must be carefully managed. In particular, the question of how to make VLM or LLM robust, and repeatable is an open problem. Somehow framing these algorithms in a control context may be an interesting avenue for future research.

10.3.0.9 Example of Implementing a Simple VLM-based Motion Planner

I provide one example for how a VLM can be used for motion planning. As seen in Fig. 10.4, I first use an RGB image and perform segmentation through typical computer vision algorithms. For each segmentation (as seen by the various colors in the image), I place a single marker at the center position of the segmented part. Around this center, I place a few markers around it, evenly spaced (unless the marker reaches outside the image bound, in which case we remove this marker from the image). The markers can be seen in the image below the segmented image on the right corner of the figure. I then use GPT4-Vision to query the image with markers. Besides an input image, we must also provide a general prompt alongside this image. I use the following as an input prompt: *You are a walking robot with an RGB-D camera attached on the head. You are navigating an environment and want to avoid colliding into obstacles, and are searching for open areas. You cannot climb, fly, or walk over obstacles. You can only walk on flat surfaces, and want to avoid high areas. It is absolutely critical you do not collide in nearby obstacles and prefer walking on floors away from obstacles. I provide an image, where on the left side is a depth image, and on the right side is a rgb image. Both images are based on the exact same scene. Please note that seeing black means the depth information for that black pixel is uncertain, so the pixel location may be far away or close, try to avoid making decisions based on black pixels. Try to mainly make decision on rgb, but still use depth to try and understand how far or close objects are. "You will pick the next appropriate joystick command for the robot which are the following: Left, Right, Backward, Forward, Stop, Turn Left, Turn Right. You MUST select only among those commands, and your output is in the same words as those commands. After you pick the command, the robot will spend about 5 seconds for a linear velocity command (which is Left, Right, Backward, Forward), and 2 seconds for angular velocity command (Turn Left, or Turn Right). Ideally, you want to avoid turning if possible and you may decide to stop, but do not idle too long while just stopping unless in front of box, but make the turn if needed. The linear velocity is at most 0.3 m/s, and angular velocity is 1.5 rad/s.*

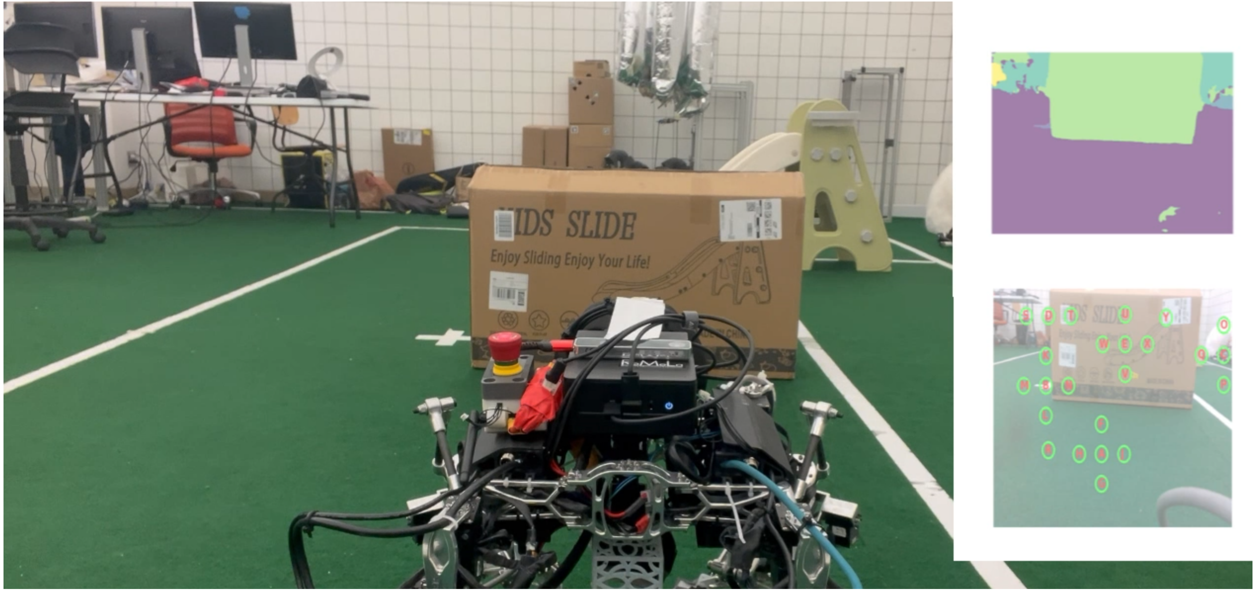


Figure 10.4: **VLM for Motion Planning**

Additionally, to standardize the output of the GPT4-Vision, we also include the desired output prompt format as an example:

```
outputPrompt = {
    "selected_keypoint": "A",
    "selected_command": "Forward",
    "reasoning": "Empty space in front, safe to go forward."
}
```

With such a simple setup and query, I can show motion planning for our robot, where it can do some form of obstacle avoidance and intelligent decision making. I note that the reason for using markers as a setup for motion planning, is that currently VLM are most adapt at multiple-choice questions [86]. This shows the potential these VLM models have in the near future. Of course, as this is just a quick and simple example, the objective in future work is to include some forms of safety to this architecture, which may still include some form of obstacle avoidance algorithm (to ensure collision do not occur) or safety, in

cases the VLM makes the wrong choice. I believe that knowledge in traditional controls and optimization can still be useful to provide stronger metrics of guarantees to this general models.

10.4 Conclusion

In this dissertation, I started with a simple energy-efficient motion planner, which required no optimization or machine learning – this was done to demonstrate how reactivity can be achieved through purely analytical means. When including uncertainty calculation or more complex scenarios, I demonstrated how a recurrent neural network can be used to quickly propagate and predict uncertainty covariances which can in turn be used within a Stochastic Model Predictive Control format. This allows us to use traditional optimization algorithms while reducing computational complexity but still show robustness. I then show that we need algorithms that can automatically calibrate various components used in Active SLAM, such as MPC or Kalman filters, as it is challenging to manually tune all of these components individually. One option is presented through the use of a UKF approach. A state estimation system that employs a hybrid model and learning-based approach is then shown, showcasing the effectiveness of including dynamic models within a machine learning context. Finally, I provide an example of using VLM for motion planning, to demonstrate the potential effectiveness for a robot to truly understand its environment and make autonomous and *smart* decision-making. I believe that research in model-based control and optimization along with VLM (e.g., formulating a VLM within an optimization or control context) to be the key for safe but truly autonomous behavior in robotic systems.

Bibliography

[1] Priyanshu Agarwal et al. "State Estimation for Legged Robots: Consistent Fusion of Leg Kinematics and IMU". In: *Robotics: Science and Systems VIII*. 2013, pp. 17–24.

[2] A. Aghar-Mohammadi et al. "Simultaneous Localization and Planning for Physical Mobile Robots via Enabling Dynamic Replanning in Belief Space". In: *CoRR* abs/1510.07380 (2015). arXiv: 1510.07380.

[3] M. S. Ahn, H. Chae, and D. W. Hong. "Stable, Autonomous, Unknown Terrain Locomotion for Quadrupeds Based on Visual Feedback and Mixed-Integer Convex Optimization". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 3791–3798.

[4] J. Andersson et al. "CasADi – A software framework for nonlinear optimization and optimal control". In: *Mathematical Programming Computation* 11.1 (2019), pp. 1–36. DOI: 10.1007/s12532-018-0139-4.

[5] Seiji Aoyagi et al. "Improvement of robot accuracy by calibrating kinematic model using a laser tracking system-compensation of non-geometric errors using neural networks and selection of optimal measuring points using genetic algorithm-". In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, October 18-22, 2010, Taipei, Taiwan*. IEEE, 2010, pp. 5660–5665. DOI: 10.1109/IROS.2010.5652953. URL: https://doi.org/10.1109/IROS.2010.5652953.

[6] Felipe Felix Arias et al. "Avoidance Critical Probabilistic Roadmaps for Motion Planning in Dynamic Environments". In: *IEEE Inter. Conf. on Robot. and Auto. (ICRA)*. 2021.

[7] T. Bailey and H. Durrant-Whyte. "Simultaneous localization and mapping (SLAM): part II". In: vol. 13. 3. 2006, pp. 108–117. DOI: 10.1109/MRA.2006.1678144.

[8] A. Bajcsy et al. “A Scalable Framework For Real-Time Multi-Robot, Multi-Human Collision Avoidance”. In: *2019 Int. Conf. on Robot. and Auto.* 2019, pp. 936–943. DOI: 10.1109/ICRA.2019.8794457.

[9] Brown Baker et al. “Emergent Tool Use From Multi-Agent Autocurricula”. In: *Int. Conf. on Learn. Repre.* 2020.

[10] Joaquin Ballesteros et al. “Proprioceptive Estimation of Forces Using Underactuated Fingers for Robot-Initiated pHRI”. In: *Sensors* 20.10 (2020), p. 2863.

[11] Pranav A. Bhounsule, Jason Cortell, and Andy Ruina. “Design and Control of Ranger: An Energy-Efficient, Dynamic Walking Robot”. In: *Adaptive Mobile Robotics*, pp. 441–448. DOI: 10.1142/9789814415958_0057. URL: https://www.worldscientific.com/doi/abs/10.1142/9789814415958_0057.

[12] Lars Blackmore, Masahiro Ono, and Brian C. Williams. “Chance-Constrained Optimal Path Planning With Obstacles”. en. In: *IEEE Trans. on Robot.* 27.6 (Dec. 2011), pp. 1080–1094. ISSN: 1552-3098, 1941-0468. DOI: 10.1109/TRO.2011.2161160. (Visited on 09/13/2020).

[13] Gerardo Bledt et al. “Contact model fusion for event-based locomotion in unstructured terrains”. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018, pp. 4399–4406.

[14] M. Bloesch et al. “State estimation for legged robots-consistent fusion of leg kinematics and IMU”. In: *Robotics* 17 (2013), pp. 17–24.

[15] Michael Bloesch et al. “State estimation for legged robots on unstable and slippery terrain”. In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013, pp. 6058–6064. DOI: 10.1109/IROS.2013.6697236.

[16] Pierre Bonami et al. “An algorithmic framework for convex mixed integer nonlinear programs”. In: *Discr. Opt.* 5.2 (May 2008), pp. 186–204. ISSN: 1572-5286. DOI: 10.1016/j.disopt.2006.10.011. (Visited on 09/13/2020).

[17] *BOTA Systems*. URL: https://www.botasys.com/force-torque-sensors (visited on 09/30/2010).

[18] Patrick Bouffard. "On-board Model Predictive Control of a Quadrotor Helicopter: Design, Implementation, and Experiments". MA thesis. EECS Depart., Uni. of Cal., Berkeley, 2012.

[19] Martin Brossard, Axel Barrau, and Silvère Bonnabel. "AI-IMU Dead-Reckoning". In: *IEEE Transactions on Intelligent Vehicles* 5.4 (2020), pp. 585–595. DOI: 10.1109/TIV.2020.2980758.

[20] Russell Buchanan et al. "Learning Inertial Odometry for Dynamic Legged Robot State Estimation". In: *Conference on Robot Learning* (2021). eprint: 2111.00789 (cs.RO).

[21] Francesco Bullo and Richard M. Murray. "Proportional Derivative (PD) Control on the Euclidean Group". In: 1995.

[22] Roberto Calandra et al. "Bayesian optimization for learning gaits under uncertainty". In: *Annals of Mathematics and Artificial Intelligence* 76.1 (2016), pp. 5–23.

[23] Marco Camurri et al. "Pronto: A Multi-Sensor State Estimator for Legged Robots in Real-World Scenarios". In: *Frontiers in Robotics and AI* 7 (2020). ISSN: 2296-9144. DOI: 10.3389/frobt.2020.00068. URL: https://www.frontiersin.org/articles/10.3389/frobt.2020.00068.

[24] Hongli Cao et al. "Dynamic Adaptive Hybrid Impedance Control for Dynamic Contact Force Tracking in Uncertain Environments". In: *IEEE Access* 7 (2019), pp. 83162–83174. DOI: 10.1109/ACCESS.2019.2924696.

[25] Nilanjan Chakraborty et al. "A geometrically implicit time-stepping method for multibody systems with intermittent contact". In: *The International Journal of Robotics Research* 33.3 (2014), pp. 426–445. DOI: 10.1177/0278364913501210. eprint: https:

//doi.org/10.1177/0278364913501210. URL: https://doi.org/10.1177/0278364913501210.

[26] J. Chase Kew et al. "Neural Collision Clearance Estimator for Batched Motion Planning". In: *Algorithmic Foundations of Robotics XIV*. Cham: Springer International Publishing, 2021, 2021, pp. 73–89. ISBN: 978-3-030-66723-8.

[27] Yevgen Chebotar et al. "Closing the Sim-to-Real Loop: Adapting Simulation Randomization with Real World Experience". In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 8973–8979. DOI: 10.1109/ICRA.2019.8793789.

[28] Lu Chen et al. "CNNs based Foothold Selection for Energy-Efficient Quadruped Locomotion over Rough Terrains". In: *2019 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. 2019, pp. 1115–1120. DOI: 10.1109/ROBIO49542.2019.8961842.

[29] Hao-Tien Lewis Chiang et al. "Fast Swept Volume Estimation with Deep Learning". In: *Algorithmic Foundations of Robotics XIII*. Springer International Publishing, 2020, pp. 52–68. ISBN: 978-3-030-44051-0.

[30] Debora Clever et al. "Inverse optimal control based identification of optimality criteria in whole-body human walking on level ground". In: *2016 6th IEEE International Conference on Biomedical Robotics and Biomechatronics (BioRob)*. 2016, pp. 1192–1199. DOI: 10.1109/BIOROB.2016.7523793.

[31] Alejo Concha et al. "Visual-inertial direct SLAM". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1331–1338. DOI: 10.1109/ICRA.2016.7487266.

[32] Alberto Dalla Libera et al. "Autonomous Learning of the Robot Kinematic Model". In: *IEEE Transactions on Robotics* 37.3 (2021), pp. 877–892. DOI: 10.1109/TRO.2020.3038690.

[33] Marc Deisenroth and Carl Rasmussen. "PILCO: A Model-Based and Data-Efficient Approach to Policy Search". In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. Ed. by Lise Getoor and Tobias Scheffer. ICML '11. Bellevue, Washington, USA: ACM, 2011, pp. 465–472. ISBN: 978-1-4503-0619-5.

[34] Jared Di Carlo et al. "Dynamic Locomotion in the MIT Cheetah 3 Through Convex Model-Predictive Control". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 1–9. DOI: 10.1109/IROS.2018.8594448.

[35] Yuqing Du et al. "Auto-Tuned Sim-to-Real Transfer". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021, pp. 1290–1296. DOI: 10.1109/ICRA48506.2021.9562091.

[36] Jinjun Duan et al. "Adaptive variable impedance control for dynamic contact force tracking in uncertain environment". In: *Robotics and Autonomous Systems* 102 (2018), pp. 54–65. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2018.01.009.

[37] H. Durrant Whyte and T. Bailey. "Simultaneous localization and mapping: part I". In: vol. 13. 2. 2006, pp. 99–110. DOI: 10.1109/MRA.2006.1638022.

[38] Hugh Durrant-Whyte, Nicholas Roy, and Pieter Abbeel. "Learning to Control a Low-Cost Manipulator Using Data-Efficient Reinforcement Learning". In: *Robotics: Science and Systems VII*. 2012, pp. 57–64.

[39] Benjamin Ellenberger. *PyBullet Gymperium*. https://github.com/benelot/pybullet-gym. 2018–2019.

[40] Sarah Elliott, Michelle Valente, and Maya Cakmak. "Making objects graspable in confined environments through push and pull manipulation with a tool". In: *IEEE Int. Conf. on Robot. and Auto. (ICRA)*. 2016, pp. 4851–4858.

[41] Jakob Engel, Thomas Schöps, and Daniel Cremers. "LSD-SLAM: Large-Scale Direct Monocular SLAM". In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Cham: Springer International Publishing, 2014, pp. 834–849. ISBN: 978-3-319-10605-2.

[42] M. Everett, Y. Chen, and J. How. "Motion Planning Among Dynamic, Decision-Making Agents with Deep Reinforcement Learning". In: (Oct. 2018), pp. 3052–3059. DOI: 10.1109/IROS.2018.8593871.

[43] Maurice F. Fallón et al. "Drift-free humanoid state estimation fusing kinematic, inertial and LIDAR sensing". In: *2014 IEEE-RAS International Conference on Humanoid Robots*. 2014, pp. 112–119. DOI: 10.1109/HUMANOIDS.2014.7041346.

[44] Xiaohan Fei and Stefano Soatto. "Visual-Inertial Object Detection and Mapping". In: *Computer Vision – ECCV 2018*. Ed. by Vittorio Ferrari et al. Cham: Springer International Publishing, 2018, pp. 318–334.

[45] Hans Joachim Ferreau et al. "qpOASES: A parametric active-set algorithm for quadratic programming". In: *Math. Program. Computation* 6.4 (2014), pp. 327–363.

[46] Geoff Fink and Claudio Semini. "Proprioceptive Sensor Fusion for Quadruped Robot State Estimation". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 10914–10920. DOI: 10.1109/IROS45743.2020.9341521.

[47] J. Fisac et al. "Probabilistically Safe Robot Planning with Confidence-Based Human Predictions". In: *Robotics Science and Systems* (2018).

[48] Friedrich Fraundorfer and Davide Scaramuzza. "Visual Odometry : Part II: Matching, Robustness, Optimization, and Applications". In: vol. 19. 2. 2012, pp. 78–90. DOI: 10.1109/MRA.2012.2182810.

[49] Zipeng Fu et al. "Minimizing Energy Consumption Leads to the Emergence of Gaits in Legged Robots". In: *Conference on Robot Learning (CoRL)*. 2021.

[50] Jonathan D. Gammell, Siddhartha S. Srinivasa, and Timothy D. Barfoot. "Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic". In: *2014 IEEE/RSJ Int. Conf. on Intell. Robots*

and Syst. ISSN: 2153-0866. Sept. 2014, pp. 2997–3004. DOI: 10.1109/IROS.2014.6942976.

[51] Siddhant Gangapurwala et al. "Real-Time Trajectory Adaptation for Quadrupedal Locomotion using Deep Reinforcement Learning". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021, pp. 5973–5979. DOI: 10.1109/ICRA48506.2021.9561639.

[52] Christian Gehring et al. "Practice Makes Perfect: An Optimization-Based Approach to Controlling Agile Motions for a Quadruped Robot". In: *IEEE Robotics Automation Magazine* 23.1 (2016), pp. 34–43.

[53] G. Grisetti, C. Stachniss, and W. Burgard. "Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters". In: *IEEE Trans. on Robot.* 23.1 (2007), pp. 34–46.

[54] Xuan Vinh Ha, Cheolkeun Ha, and Dang Khoa Nguyen. "A General Contact Force Analysis of an Under-Actuated Finger in Robot Hand Grasping". In: *Int. Journal of Adv. Robot. Syst.* 13.1 (2016), p. 14.

[55] F. L. Haufe, S. Maggioni, and A. Melendez-Calderon. "Reference Trajectory Adaptation to Improve Human-Robot Interaction: A Database-Driven Approach". In: *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. 2018, pp. 1727–1730. DOI: 10.1109/EMBC.2018.8512604.

[56] Philipp Hausamann et al. "Evaluation of the Intel RealSense T265 for tracking natural human head motion". In: *Scientific Reports* 11.1 (June 2021), p. 12486. DOI: 10.1038/s41598-021-91861-5.

[57] Kaiming He et al. "Masked Autoencoders Are Scalable Vision Learners". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2022), pp. 16000–16009.

[58] A. Hereid et al. "Hybrid zero dynamics based multiple shooting optimization with applications to robotic walking". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 5734–5740. DOI: 10.1109/ICRA.2015.7140002.

[59] Neville Hogan. "Impedance Control: An Approach to Manipulation". In: *1984 American Control Conference*. 1984, pp. 304–313.

[60] J. Hooks et al. "ALPHRED: A Multi-Modal Operations Quadruped Robot for Package Delivery Applications". In: *IEEE Robotics and Automation Letters (RA-L)* (2020). DOI: 10.1109/LRA.2020.3007482.

[61] Marwaha, Lather, Dhillon. "MULTI AREA LOAD FREQUENCY CONTROL USING MODEL PREDICTIVE CONTROLLER". In: *Proceedings of International Conference on Electrical and Electronics Engineering* (2015). DOI: ISBN:9788193137307. URL: https://www.researchgate.net/publication/290439146_MULTI_AREA_LOAD_FREQUENCY_CONTROL_USING_MODEL_PREDICTIVE_CONTROLLER.

[62] Mohamed, Bevrani, Hassan, Hiyama. "Decentralized model predictive based load frequency control in an interconnected power system". In: *Elsevier, Energy Conversion and Management* (2010). DOI: 10.1016/j.enconman.2010.09.016. URL: https://www.sciencedirect.com/science/article/pii/S0196890410004188.

[63] Uwe Hubert, Jörg Stückler, and Sven Behnke. "Bayesian calibration of the hand-eye kinematics of an anthropomorphic robot". In: *2012 12th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012)*. 2012, pp. 618–624. DOI: 10.1109/HUMANOIDS.2012.6651584.

[64] Marco Hutter et al. "ANYmal - a highly mobile and dynamic quadrupedal robot". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 38–44. DOI: 10.1109/IROS.2016.7758092.

[65] Jemin Hwangbo et al. "Learning agile and dynamic motor skills for legged robots". In: *Science Robotics* 4.26 (2019), eaau5872. DOI: 10.1126/scirobotics.aau5872.

eprint: https://www.science.org/doi/pdf/10.1126/scirobotics.aau5872. URL: https://www.science.org/doi/abs/10.1126/scirobotics.aau5872.

[66] K. Ito and F. Matsuno. "A study of reinforcement learning for the robot with many degrees of freedom - acquisition of locomotion patterns for multi-legged robot". In: *2002 IEEE International Conference on Robotics and Automation*. Vol. 4. 2002, pp. 3392–3397.

[67] Rae Jeong et al. *Self-Supervised Sim-to-Real Adaptation for Visual Robotic Manipulation*. 2019. DOI: 10.48550/ARXIV.1910.09470. URL: https://arxiv.org/abs/1910.09470.

[68] E. Jones and S. Soatto. "Visual-inertial navigation, mapping and localization: A scalable real-time causal approach". en. In: *The International Journal of Robotics Research* 30.4 (Apr. 2011), pp. 407–430. ISSN: 0278-3649. DOI: 10.1177/0278364910388963.

[69] S. Kajita et al. "Biped walking pattern generation by using preview control of zero-moment point". In: *2003 IEEE International Conference on Robotics and Automation*. Vol. 2. 2003, pp. 1620–1626.

[70] V. Kalogeiton et al. "Real-Time Active SLAM and Obstacle Avoidance for an Autonomous Robot Based on Stereo Vision". In: *Cybernetics and Systems* 50.3 (2019), pp. 239–260. DOI: 10.1080/01969722.2018.1541599. eprint: https://doi.org/10.1080/01969722.2018.1541599.

[71] Sang Hoon Kang et al. "A Solution to the Accuracy/Robustness Dilemma in Impedance Control". In: *IEEE/ASME Trans. on Mecha.* 14.3 (2009), pp. 282–294.

[72] Imin Kao, Kevin Lynch, and Joel W. Burdick. "Contact Modeling and Manipulation". In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 647–669. ISBN: 978-3-540-30301-5.

[73] Alonzo Kelly et al. "Toward Reliable Off Road Autonomous Vehicles Operating in Challenging Environments". In: *The Int. Jour. of Robot. Research* 25.5-6 (2006), pp. 449–483. DOI: 10.1177/0278364906065543.

[74] H. Kimura, T. Yamashita, and S. Kobayashi. "Reinforcement learning of walking behavior for a four-legged robot". In: *40th IEEE Conference on Decision and Control*. Vol. 1. 2001, pp. 411–416.

[75] Edin Koco. "Locomotion Control for Electrically Powered Quadruped Robot Dynarobin". In: 2014.

[76] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: (). DOI: 10.1109/IROS.2004.1389727.

[77] K. Kondo. "Motion planning with six degrees of freedom by multistrategic bidirectional heuristic free-space enumeration". In: *IEEE Trans. on Robot. and Auto.* 7.3 (1991), pp. 267–277. DOI: 10.1109/70.88136.

[78] Nicolas Lanzetti et al. "Recurrent neural network based MPC for process industries". In: *2019 18th European Control Conference (ECC)*. IEEE. 2019, pp. 1005–1010.

[79] Quentin Leboutet et al. "Tactile-Based Whole-Body Compliance With Force Propagation for Mobile Manipulators". In: *IEEE Trans. on Robot.* 35.2 (2019), pp. 330–342.

[80] Joonho Lee et al. "Learning quadrupedal locomotion over challenging terrain". In: *Science Robotics* 5.47 (2020).

[81] C. Leung, S. Huang, and G. Dissanayake. "Active SLAM using Model Predictive Control and Attractor based Exploration". In: (Nov. 2006), pp. 5026 –5031. DOI: 10.1109/IROS.2006.282530.

[82] Miao Li et al. "Learning object-level impedance control for robust grasping and dexterous manipulation". In: *2014 IEEE Int. Conf. on Robot. and Auto.* 2014, pp. 6784–6791.

[83] P. Li, M. Wendt, and G. Wozny. “A probabilistically constrained model predictive controller”. In: *Automatica* 38.7 (July 2002), pp. 1171–1176. ISSN: 0005-1098. DOI: 10.1016/S0005-1098(02)00002-X.

[84] Hongzhuo Liang et al. “PointNetGPD: Detecting Grasp Configurations from Point Sets”. In: *Int. Conf. on Robot. and Auto.* 2019, pp. 3629–3635.

[85] Zhiwei Liang, Songhao Zhu, and Xin Jin. “Walking parameters design of biped robots based on reinforcement learning”. In: *30th Chinese Control Conference*. 2011, pp. 4017–4022.

[86] Fangchen Liu et al. “MOKA: Open-Vocabulary Robotic Manipulation through Mark-Based Visual Prompting”. In: (2024). arXiv: 2403.03174 [cs.RO].

[87] John Lloyd and Nathan F. Lepora. “Goal-Driven Robotic Pushing Using Tactile and Proprioceptive Feedback”. In: *IEEE Trans. on Robot.* (2021), pp. 1–12.

[88] Ilya Loshchilov and Frank Hutter. “Decoupled Weight Decay Regularization”. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. 2019.

[89] M. Said. “Optimization Based Solutions for Control and State Estimation in Non-holonomic Mobile Robots: Stability, Distributed Control, and Relative Localization”. In: *Cornell University, arXiv.org* (2018). DOI: 10.13140/RG.2.2.22309.60644. URL: https://arxiv.org/abs/1803.06928.

[90] Zachary Manchester et al. “Contact-implicit trajectory optimization using variational integrators”. In: *The International Journal of Robotics Research* 38.12-13 (2019), pp. 1463–1476. DOI: 10.1177/0278364919849235. eprint: https://doi.org/10.1177/0278364919849235. URL: https://doi.org/10.1177/0278364919849235.

[91] Alonso Marco et al. “Automatic LQR tuning based on Gaussian process global optimization”. In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 270–277.

[92] Bhairav Mehta et al. "Active Domain Randomization". In: *CoRL*. 2019.

[93] Marcel Menner, Karl Berntorp, and Stefano Di Cairano. "A Kalman Filter for Online Calibration of Optimal Controllers". In: *2021 IEEE Conference on Control Technology and Applications (CCTA)*. 2021, pp. 441–446.

[94] Marcel Menner, Karl Berntorp, and Stefano Di Cairano. "Automated Controller Calibration by Kalman Filtering". In: *arXiv preprint arXiv:2111.10832* (2021).

[95] O. Michel. "Webots: Professional Mobile Robot Simulation". In: *Journal of Advanced Robotics Systems* 1.1 (2004), pp. 39–42. URL: http://www.ars-journal.com/International-Journal-of-Advanced-Robotic-Systems/Volume-1/39-42.pdf.

[96] Takahiro Miki et al. "Learning robust perceptive locomotion for quadrupedal robots in the wild". In: *Science Robotics* 7.62 (2022), eabk2822. DOI: 10.1126/scirobotics.abk2822. eprint: https://www.science.org/doi/pdf/10.1126/scirobotics.abk2822. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abk2822.

[97] Volodymyr Mnih, et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602.

[98] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós. "ORB-SLAM: A Versatile and Accurate Monocular SLAM System". In: *IEEE Transactions on Robotics* 31.5 (2015), pp. 1147–1163. ISSN: 1941-0468. DOI: 10.1109/TRO.2015.2463671.

[99] Matthias Neumann-Brosig et al. "Data-Efficient Autotuning With Bayesian Optimization: An Industrial Control Study". In: *IEEE Transactions on Control Systems Technology* 28.3 (2020), 730–740.

[100] OpenAI, Ilge Akkaya, et al. *Solving Rubik's Cube with a Robot Hand*. 2019. DOI: 10.48550/ARXIV.1910.07113. URL: https://arxiv.org/abs/1910.07113.

[101] Peter Pastor et al. "Learning task error models for manipulation". In: *2013 IEEE International Conference on Robotics and Automation*. 2013, pp. 2612–2618. DOI: 10.1109/ICRA.2013.6630935.

[102] Matteo Parigi Polverini et al. "Multi-Contact Heavy Object Pushing With a Centaur-Type Humanoid Robot: Planning and Control for a Real Demonstrator". In: *IEEE Robot. and Auto. Letters* 5.2 (2020), pp. 859–866.

[103] Fankhauser Péter et al. "Collaborative navigation for flying and walking robots". In: *2016 IEEE/RSJ Int. Conf. on Intell. Robots and Syst. (IROS)*. 2016. DOI: 10.1109/IROS.2016.7759443.

[104] Yousaf Rahman et al. "A tutorial and overview of retrospective cost adaptive control". In: *2016 American Control Conference (ACC)*. 2016, pp. 3386–3409. DOI: 10.1109/ACC.2016.7525440.

[105] Akshara Rai et al. "Bayesian optimization using domain knowledge on the ATRIAS biped". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018, pp. 1771–1778.

[106] M. Raibert. *Legged robots that balance*. MIT press, 1986.

[107] Marc H. Raibert. *Legged Robots That Balance*. USA: Massachusetts Institute of Technology, 1986. ISBN: 0262181177.

[108] Niraj Rathod et al. "Model Predictive Control With Environment Adaptation for Legged Locomotion". In: *IEEE Access* 9 (2021), 145710–145727. ISSN: 2169-3536.

[109] James B. Rawlings, David Q. Mayne, and Moritz M. Diehl. *Model predictive control: theory, computation, and design*. en. 2nd ed. Nob Hill Publ., 2017. ISBN: 978-0-9759377-3-0.

[110] J. Redmon and A. Farhadi. "YOLOv3: An Incremental Improvement". In: *arXiv* (2018).

[111] Guy Revach et al. "KalmanNet: Neural Network Aided Kalman Filtering for Partially Known Dynamics". In: *Trans. Sig. Proc.* 70 (2022), 1532–1547. ISSN: 1053-587X. DOI: 10.1109/TSP.2022.3158588. URL: https://doi.org/10.1109/TSP.2022.3158588.

[112] Daniel Reyes-Uquillas and Tesheng Hsiao. "Safe and intuitive manual guidance of a robot manipulator using adaptive admittance control towards robot agility". In: *Robot and Computer-Integrated Manuf.* 70 (2021), p. 102127. ISSN: 0736-5845. DOI: https://doi.org/10.1016/j.rcim.2021.102127.

[113] Stergios I. Roumeliotis. "Robust mobile robot localization: From single-robot uncertainties to multi-robot interdependencies". PhD thesis. Uni. of Southern Cal., Oct. 2000.

[114] E. Rublee et al. "ORB: An efficient alternative to SIFT or SURF". In: *2011 International Conference on Computer Vision*. 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544.

[115] Victor Garcia Satorras, Zeynep Akata, and Max Welling. "Combining Generative and Discriminative Models for Hybrid Inference". In: *arXiv* (2019). eprint: 1906.02547 (stat.ML).

[116] Davide Scaramuzza and Friedrich Fraundorfer. "Visual Odometry [Tutorial]". In: vol. 18. 4. 2011, pp. 80–92. DOI: 10.1109/MRA.2011.943233.

[117] Alexander Schperberg, Stefano Di Cairano, and Marcel Menner. "Auto-Tuning of Controller and Online Trajectory Planner for Legged Robots". In: *IEEE Robotics and Automation Letters* (2022), pp. 1–8. DOI: 10.1109/LRA.2022.3185387.

[118] Alexander Schperberg et al. "Adaptive Force Controller for Contact-Rich Robotic Systems using an Unscented Kalman Filter". In: *arXiv preprint arXiv:2207.01033* (2022).

[119] Alexander Schperberg et al. "Real-to-Sim: Predicting Residual Errors of Robotic Systems with Sparse Data using a Learning-Based Unscented Kalman Filter". In: *2023 20th International Conference on Ubiquitous Robots (UR)*. 2023, pp. 27–34. DOI: 10.1109/UR57808.2023.10202521.

[120] Alexander Schperberg et al. "Risk-Averse MPC via Visual-Inertial Input and Recurrent Networks for Online Collision Avoidance". In: *2020 IEEE/RSJ Int. Conf. on Intell. Robots and Syst. (IROS)*. Nov. 2020.

[121] Alexander Schperberg et al. "Risk-Averse MPC via Visual-Inertial Input and Recurrent Networks for Online Collision Avoidance". In: (2020), pp. 5730–5737. DOI: 10.1109/IROS45743.2020.9341070.

[122] Alexander Schperberg et al. "SABER: Data-Driven Motion Planner for Autonomously Navigating Heterogeneous Robots". In: *IEEE Robotics and Automation Letters* 6.4 (2021), pp. 8086–8093. DOI: 10.1109/LRA.2021.3103054.

[123] Alexander Schperberg et al. "SABER: Data-Driven Motion Planner for Autonomously Navigating Heterogeneous Robots". In: *IEEE Robotics and Automation Letters* 6.4 (2021), pp. 8086–8093. DOI: 10.1109/LRA.2021.3103054.

[124] A. Schwarm and M. Nikolaou. "Chance-constrained model predictive control". In: *AIChE Journal* 45.8 (1999), pp. 1743–1752. DOI: 10.1002/aic.690450811.

[125] Fan Shi et al. "Circus ANYmal: A Quadruped Learning Dexterous Manipulation with Its Limbs". In: *2021 IEEE Int. Conf. on Robot. and Auto. (ICRA)*. 2021, pp. 2316–2323.

[126] Hyeok-Ki Shin and Byung Kook Kim. "Energy-Efficient Gait Planning and Control for Biped Robots Utilizing the Allowable ZMP Region". In: *IEEE Transactions on Robotics* 30.4 (2014), pp. 986–993. DOI: 10.1109/TRO.2014.2305792.

[127] Yuki Shirai et al. “Risk-Aware Motion Planning for a Limbed Robot with Stochastic Gripping Forces Using Nonlinear Programming”. In: *IEEE Robot. and Auto. Letters* 5.4 (2020), pp. 4994–5001. DOI: 10.1109/LRA.2020.3001503.

[128] Lingfeng Sun et al. “Interactive Planning Using Large Language Models for Partially Observable Robotic Tasks”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. May 2024. URL: https://www.merl.com/publications/TR2024-052.

[129] Zhenglong Sun and Nico Roos. “An energy efficient dynamic gait for a Nao robot”. In: *2014 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*. 2014, pp. 267–272. DOI: 10.1109/ICARSC.2014.6849797.

[130] Yusuke Tanaka et al. “SCALER: A Tough Versatile Quadruped Free-Climber Robot”. In: *Proc. IEEE/RSJ Int. Conf. Intell. Rob. Syst* (2022).

[131] Yusuke Tanaka et al. “SCALER: A Tough Versatile Quadruped Free-Climber Robot”. In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2022, pp. 5632–5639. DOI: 10.1109/IROS47612.2022.9981555.

[132] Te Tang et al. “A Learning-Based Framework for Robot Peg-Hole-Insertion”. In: Dyn. Syst. and Con. Conf. Oct. 2015.

[133] Gabriel A Terejanu. “Unscented Kalman filter tutorial”. In: *University at Buffalo, Buffalo* (2011).

[134] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intell. Robot. and Autonomous Agents)*. The MIT Press, 2005. ISBN: 0262201623.

[135] Emanuel Todorov, Tom Erez, and Yuval Tassa. “MuJoCo: A physics engine for model-based control”. In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[136] Vassilios Tsounis et al. "DeepGait: Planning and Control of Quadrupedal Gaits Using Deep Reinforcement Learning". In: *IEEE Robotics and Automation Letters* 5.2 (2020), pp. 3699–3706. DOI: 10.1109/LRA.2020.2979660.

[137] Unitree Robotics. "GitHub repository for Unitree A1". In: *https:// github.com/unitreerobotics* (accessed 2021-09).

[138] Teresa A. Vidal-Calleja et al. "Large scale multiple robot visual mapping with heterogeneous landmarks in semi-structured terrain". In: *Robot. and Auto. Sys.* 59.9 (2011), pp. 654–674. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2011.05.008.

[139] Rok Vrabič et al. "An architecture for sim-to-real and real-to-sim experimentation in robotic systems". In: *Procedia CIRP* 104 (2021). 54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0, pp. 336–341. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2021.11.057. URL: https://www.sciencedirect.com/science/article/pii/S2212827121009550.

[140] Andreas Wächter and Lorenz T Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Math. Program.* 106.1 (2006), pp. 25–57.

[141] Amanhoud Walid, Mahdi Khoramshahi, and Aude Billard. "A Dynamical System Approach to Motion and Force Generation in Contact Tasks". In: *Proceedings of Robotics: Science and Systems*. Freiburg, Germany, 2019.

[142] Rose E. Wang et al. "Model-based Reinforcement Learning for Decentralized Multiagent Rendezvous". In: *Conf. on Robot Learn. (CoRL)*. 2020.

[143] Alexander W. Winkler et al. "Gait and Trajectory Optimization for Legged Systems Through Phase-Based End-Effector Parameterization". In: *IEEE Robotics and Automation Letters* 3.3 (2018), pp. 1560–1567. DOI: 10.1109/LRA.2018.2798285.

[144] David Wisth, Marco Camurri, and Maurice Fallon. "Robust Legged Robot State Estimation Using Factor Graph Optimization". In: *IEEE Robotics and Automation Letters* 4.4 (2019), pp. 4507–4514. DOI: 10.1109/LRA.2019.2933768.

[145] Weitao Xi and C. David Remy. "Optimal gaits and motions for legged robots". In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2014, pp. 3259–3265. DOI: 10.1109/IROS.2014.6943015.

[146] S. Xin, R. Orsolino, and N. Tsagarakis. "Online Relative Footstep Optimization for Legged Robots Dynamic Walking Using Discrete-Time Model Predictive Control". In: (Mar. 2019). DOI: 10.13140/RG.2.2.36298.21447.

[147] X Xinjilefu, Siyuan Feng, and Christopher G. Atkeson. "Dynamic state estimation using Quadratic Programming". In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2014, pp. 989–994. DOI: 10.1109/IROS.2014.6942679.

[148] Shuo Yang et al. "Cerberus: Low-Drift Visual-Inertial-Leg Odometry For Agile Locomotion". In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. 2023, pp. 4193–4199. DOI: 10.1109/ICRA48891.2023.10160486.

[149] Wen Yu and Adolfo Perrusquía. "Simplified Stable Admittance Control Using End-Effector Orientations". In: *Intern. Journal of Social Robot.* 12.5 (Nov. 1, 2020), pp. 1061–1073. ISSN: 1875-4805.

[150] J. Yuan et al. "A Novel GRU-RNN Network Model for Dynamic Path Planning of Mobile Robot". In: *IEEE Access* 7 (2019), pp. 15140–15151. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2894626.

[151] Ming Zhang et al. "IMU Data Processing For Inertial Aided Navigation: A Recurrent Neural Network Based Approach". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021, pp. 3992–3998. DOI: 10.1109/ICRA48506.2021.9561172.

[152] Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. "Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: a Survey". In: *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. 2020, pp. 737–744. DOI: 10.1109/SSCI47803.2020.9308468.