Agile and Versatile Robotic Locomotion via Kernel-based Residual Learning

Milo Carroll

Master of Science

School of Informatics

University of Edinburgh

2022

Abstract

This work develops a robotic locomotion framework for quadrupedal robots using kernel-based residual learning. Initially, learning a kernel that replicates an MPC controller to produce trajectories, which a reinforcement learning agent then adapts. We show that with relatively little training, the robot can traverse a range of unseen terrains, which the MPC controller can not, and demonstrate symmetry in the locomotion without additional reward function engineering. Furthermore, the kernel learned can produce a range of functional locomotion behaviors and has the potential to generalize to unseen gaits.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Milo Carroll*)

Acknowledgements

First and foremost i would like to thank my supervisors, Mohammad Kasaei, Zhaocheng Liu, and Zhibin Li, for the insightful conversations, guidance, and support through the challenging aspects of this project.

A special thanks goes out to the BB's, for the great times, high spirits, and for providing me with memories for life.

Finally, and most importantly, i would like to thank my family, without you, this year could not have happened, and what an amazing year it has been.

Table of Contents

Chapter 1

Introduction

1.1 Project Motivations

Challenging terrains come in many forms, yet quadrupedal animals can traverse almost any, even those not previously experienced, allowing them to access the most remote locations [27]. For this reason, researchers have paved the way in developing controllers for legged robots. Their versatility far exceeds that of other forms of locomotion, such as wheeled locomotion, which requires continuous ground support and cannot feasibly adapt to challenging terrains [4, 49].

Quadrupedal robotic locomotion is a highly complex control problem that requires optimizing a non-linear dynamical system. Nevertheless, researchers have demonstrated the ability to create highly effective controllers [7, 22, 23]. Challenges still pertain, with researchers actively tackling locomotion over slippery, obstructed, irregular, and deformable terrains. Conventional controllers struggle to maintain stable balance of the robot in such challenging situations. Furthermore, they have become increasingly complex, requiring bolt-on packages to handle issues such as irregular ground contact and foot slippage [13]. These methods increase the computational cost and become unreliable in field tests where conditions are uncertain [27].

Deep reinforcement learning (RL) has become an increasingly popular method for robotic locomotion due to removing much of the engineering effort [13, 27]. RL has been highly successful in many cases and can learn more robust locomotion control [13, 33, 62]. At the same time, It can be challenging to produce desirable locomotion strategies, often we see asymmetric gait patterns [1] and *lazy limbs*, where one limb is under utilized and the motion appears unnatural [67]. Furthermore, the learned policies do not necessarily generalize to unseen terrains, and require many millions of training times-steps which is both costly and prevents us from learning on physical systems, forcing us to learn in simulation, which results in a performance gap on physical systems (the reality-gap) [55].

RL is inspired by biological systems' ability to learn [60]. Research in robotic locomotion again looks to nature, finding that numerous mammalian species demonstrate the ability to walk and even trot within minutes of birth [12]. Their nervous systems have pre-developed neural circuits that are rapidly refined and used to learn additional skills. Researchers aim to replicate this characteristic with RL control methods, introducing motion priors that and enhancing the skills with RL [8, 61]. Thus, reducing the sample efficiency issue and promoting desirable locomotion characteristics such as symmetry.

1.2 Project Objectives

This project seeks to develop a robust and controllable omnidirectional robotic locomotion framework; It aims to traverse challenging terrains and recover the robot's balance after large perturbations. Leveraging conventional control methods to learn practical and controllable motion priors using neural network (kernel). We adapt the motion priors to achieve the framework's described characteristics using RL for residual learning (Section 3.3). Below lists the core objectives of this work:

- Learn a trajectory based locomotion controller (kernel) from expert data.
- Use RL to enhance the abilities of the kernel via residual learning.
- Demonstrate traversing unseen and challenging terrains.
- Demonstrate robustness against large perturbations.
- Demonstrate the frameworks superior performance over baselines.

1.3 Document Structure

- Background: Core technical concepts required for understanding the paper.
- Related Works: Papers inspiring this work, and advances within the domain.
- Methodology: The proposed framework, experimental results and analysis.
- Evaluation: Analysis of the frameworks performance compared to baselines.
- Conclusion: Core findings, highlighting strengths, weaknesses and future research directions.

Chapter 2

Background

2.1 Robotic Locomotion Control

Controllers are responsible for producing the locomotion behavior by actuating the robot's joints. There are four core controller categories; model-based, model-free, Central-Pattern-Generators (CPG), and hybrids [21]. Conventional model-based controllers require a dynamics model of the robot, using model predictive control (Section 2.3) to generate motor commands for optimal control. Typically, the model is a reduced order (simplified) *template* [10] model that captures the system's essential dynamics. This increases computational efficiency, although it introduces inaccuracies. Conversely, model-free controllers, that are typically learned via reinforcement learning (Section 2.5), do not require a dynamics model of the system [27]; Rather control is achieved by a learned mapping between sensory input to control values. CPG-based controllers simulate biological neuronal groups that produce rhythmic activity to produce rhythmic locomotion strategies [59]. They are feed-forward, efficient, adaptable, and allow for online gait generation; however, they are difficult to parameterize and to incorporate sensory information [21]. Hybrid controllers utilize aspects multiple control categories, leveraging advantages of one to overcome challenges of another [21, 25, 65]. Typically we see the combination of model-free methods with either model-based or CPG-based methods.

2.2 Kinematics

A kinematics structure contains a tree of links and joints, which can rotate about an axis and connect two links. A kinematic tree is described by its configuration denoting the angle of each joint's rotation. With this knowledge we can determine the position of the last link in the kinematic tree (end-effector) using forward kinematics. Conversely, if we know the end-effector's current or target position, we can calculate the configurations that achieve this position with inverse-kinematics.

2.3 Model Predictive Control

Model predictive control (MPC) is an iterative optimization technique that requires a system dynamics model [7]. Given the current control inputs, trajectory, and cost function, it finds the optimal sequence of control outputs and the corresponding trajectory over a finite-time horizon. The first control outputs of the computed sequence are executed. At the next time-step, the optimization is run again given the new state of the robot. MPC is computationally expensive, hence the use of reduced order dynamics models.

2.4 Proportional Derivative Control

Proportional derivative (PD) control is a subset of the proportional integral derivative (PID) controller, which is an open-loop feedback controller. The PID controller produces control outputs u_t to apply to a system to reach some target position q_{target}. The control outputs are responsive to the current position q_t by way of the error $q_e = q_{target} - q_t$. The controller is parameterized by the variables K_p, K_i, K_d, as in Equation 2.1. Note that a PD controller simply assigns $K_i = 0$.

$$u_t = K_p * q_e + K_i * \int q_e \, dt + K_d * \dot{q_e} \tag{2.1}$$

2.5 Reinforcement Learning

Reinforcement Learning (RL) is a paradigm of machine learning, that through trial and error, learns to make sequential decisions. RL algorithms learn to maximize the total expected reward collected by taking actions according to its policy [2]. RL is modeled as a Markov Decision Process (MDP) [40] described as the tuple.$\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \rho(s_0) \rangle$; Where $\rho(s_0)$ is the set starting states [2], $\mathcal{S}$ is All possible states and $\mathcal{A}$ are actions. Actions a_t taken in a state s_t produce rewards r_t depending on the next state s_{t+1} as defined by the reward function $\mathcal{R}(s_t, a_t, s_{t+1})$. The next state is determined according to the transition dynamics $\mathcal{T}(s_{t+1}|s_t,a_t)$, which may be non-deterministic. RL algorithms aim to learn the optimal policy $\pi*(a_t|s_t))$ [2] that maximises the expected sum of discounted rewards $\pi^* = \text{argmax}_\pi \mathbb{E}[\sum_{t=0}^{\infty}\gamma^t r_{t+1}|\pi]$, where γ is the discount factor $\in [0,1]$.

2.6 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is considered to be easy to implement and stable compared to alternative algorithms. It is On-policy, meaning the experiences used to update π_θ are generated by the policy π_θ [46, 54]. The algorithm learns a policy $\pi(a_t|s_t)$ and a value function $\mathcal{V}(s_t)$. The policy learns via policy gradients, learning it directly by estimating the gradient (Equation 2.2) using the generalized advantage estimator [45] (Equation 2.3), which values the actions taken against the expected return of a specific state; The advantage estimate is positive when the policy is improving and negative otherwise.

$$\mathbb{E}[\sum_{t=0}^{\infty} \nabla_\theta log\ \pi_\theta(a_t|s_t)\ \hat{A}_t] \tag{2.2}$$

$$\hat{A}_t = -\mathcal{V}(s_t) + r_t + \gamma r_{t+1} + \ldots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t}\mathcal{V}(s_T) \tag{2.3}$$

Schulman et al. [46] innovated by removing the optimization constraints of TRPO[44] while preventing the policy update being to large. Achieved using a loss function that clips the ratio r of the change in the policy (Equations 2.4, 2.5) if it exceeds a threshold.

$$L_t^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t,\ clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t] \tag{2.4}$$

$$where\ \ r_t(\theta) = \frac{\pi(a_t|s_t\ ;\theta)}{\pi(a_t|s_t\ ;\theta_{old})} \tag{2.5}$$

Note, we can share parameters between the value function and the policy. Doing so introduces additional terms for the value function error, and an entropy bonus to promote policy exploration (Equation 2.6).

Clipping forces the ratio r in the range $[1-\varepsilon, 1+\varepsilon]$ and thus preventing large deviations between the old and the new policy, by limiting the estimated gradient. As the loss function takes the minimum, it produces a lower bound of the unclipped objective function. Figure 2.1 shows the effect of the clipping function.

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta(s_t)]] \quad (2.6)$$

2.6: where c1 and c2 are coefficients, S is an entropy bonus, and
$L_t^{VF} = (\mathcal{V}_\theta(s_t) - \mathcal{V}_t^{target})$

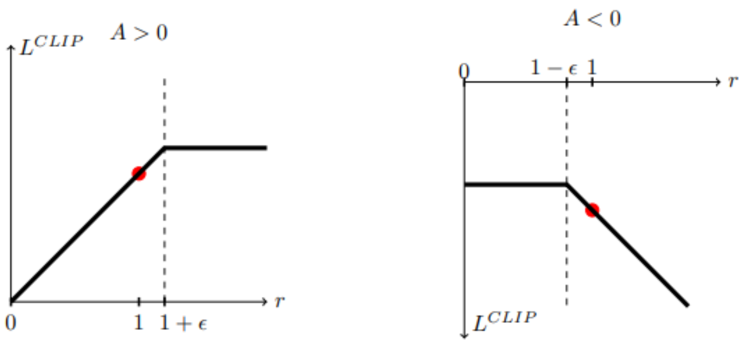


Figure 2.1: "Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., r = 1. Note that L^{CLIP} sums many of these terms" [46]

As shown in Algorithm 1, PPO computes multiple updates of the policy π_θ with experiences collected using $\pi_{\theta_{old}}$. This highlights the necessity for clipping, as large changes will break the requirements of an on-policy algorithm, which causes the instability found without clipping. PPO is equally stable and reliable compared to TRPO, while showing improved results, exhibiting better sample efficiency, achieving greater total cumulative reward, and is more broadly applicable.

Algorithm 1 PPO, Actor-Critic style [46]

```
1: for Iteration: 1,2,...,N do
2:     for actor=1,2,...,N do
3:         Run policy $\pi_{\theta_{old}}$ in environment for $T$ timesteps
4:         Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
5:     end for
6:     Optimize surrogate $L$ w.r.t $\theta$, with K epochs and minibatch size $M \leq NT$
7:     $\theta_{old} \leftarrow \theta$
8: end for
```

Chapter 3

Related Work

3.1 Learning Trajectory-based Controllers

Primitives frequently appear in the literature as a method to learn trajectory-based locomotion controllers from expert data. Neural networks are applied in recent works, with more effort towards generative instead of discriminative models.

Primitives: Kinematic Motion Primitives (kMPs) [35, 52] produce static controllers, showing the ability to execute stable gaits on physical robots [36, 50]. kMPs are derived using Principal Component Analysis (PCA [19]) for dimensionality reduction. The method extracts eigenvalues corresponding to each motor-controlled joint. Typically a subset of the eigenvalues can reconstruct joint angle trajectories with a high degree of accuracy. Singla et al. [50] show that a single set of kMPs learned from one gait (trot) is sufficient to generate multiple gaits (walk, bound, gallop) through *time-shifting*. Moro et al. [36] create several kMP controllers for different gait speeds, demonstrating gait synthesis through a linear combination of the kMPs.

Periodic Dynamic Movement Primitives [16, 17, 43] (dMPs) also produce static controllers. They are modeled as a dynamical system using a linear second-order differential equation $\tau\ddot{y} = \alpha[\beta(g-y)-\dot{y}]+f(\phi)$ where τ is a time constant, α and $\beta > 0$ are tuned and fixed constants, y is the trajectory to fit, and $f(\phi)$ (Equation 3.1) is a forcing function given the phase ϕ that creates oscillations around the goal $g = 0$. Given a single demonstration, the weights ω_i can be trained using locally weighted regression to fit f_{target} (Equation 3.2), reproducing the trajectories y_{demo}.

Rosado et al. [42] and Liu et al. [32] show the adaptability of this formulation. Specifically, [42] shows that modifying the parameters r, ω, and g can alter the amplitude, frequency, and offset of the trajectories, which can enable adaptive control.

$$f(\phi) = \frac{\sum_{i=1}^{n} \psi_i(\phi)\omega_i}{\sum_{i=1}^{n} \psi_i} r \qquad \psi_i(\phi) = exp(h_i(cos(\phi - c_i) - 1) \tag{3.1}$$

3.1: r characterises the amplitude of the oscillations, ψ_i are Mises basis function, and N, h_i, c_i are selected hyper-parameters.

$$f_{target} = \tau^2 \ddot{y}_{\phi,demo} - \alpha[\beta(g - y_{\phi,demo}) - \tau\dot{y}_{\phi,demo} \tag{3.2}$$

FastMimic [30] utilizes this attribute, optimizing the parameters r and g to transform dMPs fitted to motion capture data re-targeted [37] to the A1 robot. They demonstrate rapid imitation learning and impressive transferability to physical systems.

Both kMPs and dMPs provide limited control, only producing static controllers (e.g. a single gait). Furthermore, dMPs require a unique dMP for each degree of freedom of a trajectory. For example, FastMimic [30] requires 24 unique dMPs per gait, 12 to model the foot position trajectories. However, dMPs can be task-parameterized [38, 39]; therefore, it may be possible to create a dynamic controller, although this has not been applied to rhythmic dMPs, which are required for locomotion.

Discriminative Neural Networks (NN) are used frequently for trajectory prediction tasks. Nevertheless, we seldom see them applied for locomotion control, but typically as a component within a larger system [24, 57].To our knowledge, there are only two examples of discriminative neural networks being used directly for trajectory prediction in robot locomotion[28, 63].

Yamamoto et al. [63] use auto-encoders to reconstruct the robot's state from a three-dimensional latent encoding. They found that the latent variables periodically oscillated between zero and one during locomotion. Furthermore, they generate gait patterns outside of the training distribution by injecting oscillations into the decoder. However, the additional gaits could not be used to control the physical robot effectively.

Li et al. [28] use a fully connected neural network for trajectory prediction. Despite achieving a low validation loss, the model could not generate functional locomotion behavior in simulation. This is likely because the inputs to the network did not include a concept of time, unlike [63], which used time-dependent oscillatory latents. Therefore similar network inputs could have very different labels confusing the model. It is worth noting that this model was trained to seed the neural network of an RL agent. Therefore, the controllers' functionality was not of primary concern.

Generative Models enable plausible motion synthesis, for example, by creating realistic motion between two gaits provided during training. Surovik et al. [53] use conditional variational auto-encoders (cVAE [51]) to learn a generative model for quadrupedal locomotion in a receding-horizon fashion. The model can navigate obstacles, gaps, and other challenging terrains. The cVAE reconstructs a sequence of trajectories τ conditioned on a sequence of sensory observations and task parameters x. During the roll-out the encoder is not used, therefore, latent variables z are determined using a nearest-neighbors look-up between x and $x' \in \mathcal{D}$, such that $g(x) \rightarrow x'$ and $f(x') \rightarrow z$. This limits the potential of the cVAE by restricting the latent space, and assumes that there exists an x' sufficiently similar to x such that the latent z are appropriate.

VAE-Loco [34] uses disentangled variational autoencoder (VAE) for trajectory prediction. Given a sequence of previous robot states, the VAE reconstructs the current state and predicts the following k states. They observe that the latents move in a cyclic pattern during locomotion. Furthermore, they found that injecting a drive signal into the latent space with the least variance can control the locomotion gait. The signal is parameterized by A_k, ε_k, T_k, determining the step height, frequency, and stance duration, respectively. Impressively, this method can generate continuous variations of the trot gait (given a single style of the trot gait as a demonstration); However, it was not shown to produce multiple gaits.

3.2 Locomotion via Reinforcement Learning

As our framework does not rely soley on RL techniques we briefly discuss some of the advanced approaches applying RL for robotic locomotion. The field is rich with successful implementations [13, 15, 26, 58, 62], most notably, Lee et al. [27] train a policy that effectively transfers to a physical system, traversing unseen muddy and rocky terrains without any visual system. This is achieved by gradually increasing the terrain difficulty during training, teacher guiding via a privileged learner[5], and many millions of training time-steps.

Efforts to reduce the sample efficiency bottleneck have been primarily focused around Imitation learning, and residual learning (Section 3.3). Imitation learning techniques typically try to replicate reference motions, for example motion capture data that has been retargeted to the robot [37] or trajectories provided by an expert controller

[47]. Alternatively, Escontrela et al. [9] and Li et al. [29] aim to produce trajectories of the same distribution through adversarial rewards, training a discriminator to determine if the trajectory was produced by the agent or a reference. Yang et al. [64] made a significant contribution with MELA, a hierarchical system capable of generating diverse motions including fall recovery. The agent synthesises a policy from several expert policies trained for specific skills through imitation, demonstrating rapid learning of a multi-skilled system.

3.3 Locomotion via Residual Learning

Residual Learning [18, 68] methods train an RL agent to produce target trajectory deltas and sum this with a base trajectory (reference) typically generated from an expert controller. We see this applied to control both bipedal [8, 20, 21, 61] and quadrupedal robots [11, 48] to improve the sample efficiency and robustness.

Xie et al. [61] implemented an early residual learning framework on Cassie[1], a bipedal robot. Using a looped, two-step reference trajectory and a residual agent, they demonstrate traversing over challenging terrains, recovering balance from significant perturbation, and correcting infeasible reference trajectories. The agent receives the reference motion and the robot's state, outputting the residual trajectories in the joint angle space. However, walking at a range of speeds required training multiple agents and longer training times as the target velocity increased.

Duan et al. [8] improve upon the work from Xie et al. [61] by providing a greater variety of reference trajectories; however, similarly , they are not dynamic. Trajectories are queried from a library by the desired velocity. They display the significant sim-to-real capabilities of residual learning, and that residuals in the joint space perform poorly compared to positional residuals; Which requires less training and achieves greater asymptotic cumulative rewards. Both works [8, 61] are significantly limited by lacking the ability to turn due to not including any rotational reference trajectories.

Overcoming the inability to turn, Kasaei et al. [21] use a CPG controller to generate the reference motions enabling omnidirectional and continuous velocity control. The agent is unaware of the reference trajectory but, in addition to residuals, produces the CPG control variables; step lengths, rotation, target CoM, step duration, and PD gains,

[1]https://github.com/agilityrobotics/cassie-doc

while demonstrating impressive robustness with the more complex COMAN[2] robot.

Similarly, Gangapurwala et al. [11] use a model-based controller to generate dynamic omnidirectional reference trajectories with the ANYmal[3] robot without passing reference trajectories to the agent. However, they pass information about the terrain. Using a convolutional auto-encoder, the latent space, which represents the input depth map, is included in the agents' state space.

Recent works learn to generate reference trajectories. Shi et al. [48] produce reference trajectories using a CPG-RBF [56] with an additional linear network optimized via evolutionary strategies[3] for reference trajectory exploration. The linear network and residual agent are optimized interchangeably. The trained linear network modifies the CPG-RBF trajectories in a way best suited for the task, producing more effective reference trajectories.

Yu and Rosendo [66] develop a multi-modal locomotion framework that can control a quadrupedal robot to walk on two legs via residual learning. They provide four primitive reference trajectories parameterized by $\pi_{\theta'}$, which are optimized in parrallel with the residual agent using various non-gradient optimization methods, achieving the best results using CMA-ES [14].

Most impressively, Jungdam et al. [20] train a kernel using a cVAE to produce a range of locomotion strategies for a humanoid. The kernel behavior alone is somewhat random. The residual agent retargets the trajectories enabling omnidirectional control. However, this controls a physically-simulated character, simplifying the challenges faced when dealing with robots.

In an alternative to learning residuals, Li et al. [31] train the policy to directly output the target trajectories, arguing that residual learning limits the controller's capability. The policy receives a reference trajectory from a gait library, consisting of predetermined trajectories that can be selected according to the target frontal, lateral, and angular velocity commands. This method shows slightly improved performance compared to residual baselines. However, it is less capable of handling large perturbations.

[2]https://robots.ieee.org/robots/coman/
[3]https://robots.ieee.org/robots/anymal/

Chapter 4

Methodology

4.1 Problem Formulation

This project aims to create a robust locomotion framework with omnidirectional control. Agility and versatility being at the core of the project, we aim to produce a controller that can navigate across a broad range of unseen terrains quickly. As such, the core behavioural characteristics of concern are the robots falling frequency (robustness), and the number of targets the robot can reach within a time limit. Given a target location pos_{target} the controller must autonomously navigate a robot across the terrain, such that the distance D_{target} between the robots position pos_{base} and pos_{target} is less than a minimum threshold D_{min}.

4.2 Proposed Architecture

The proposed locomotion framework consists of three core components (Figure 4.1); A kernel, an RL agent, and a low-level PD controller. The kernel is an MLP trained to replicate the trajectories produced by a model-based MPC controller. Given a set of velocity commands, it outputs foot target positions in cartesian space relative to the robot's base.

The role of the RL agent is to make the locomotion strategy both agile and versatile to the environment via residual learning, modifying the trajectories produced by the kernel. Specifically, it produces foot target deltas in cartesian space that are summed with the kernel output, as shown to be most effective by Duan et al. [8]. As such, it learns the robot's dynamics, and skills such as balance recovery.

Given the aggregated foot target positions, we convert these into target joint angles using inverse-kinematics and pass this to the low-level PD controller. The low-level PD controller is responsible for optimizing the torque values applied to each joint that realizes the desired behavior of the robot.

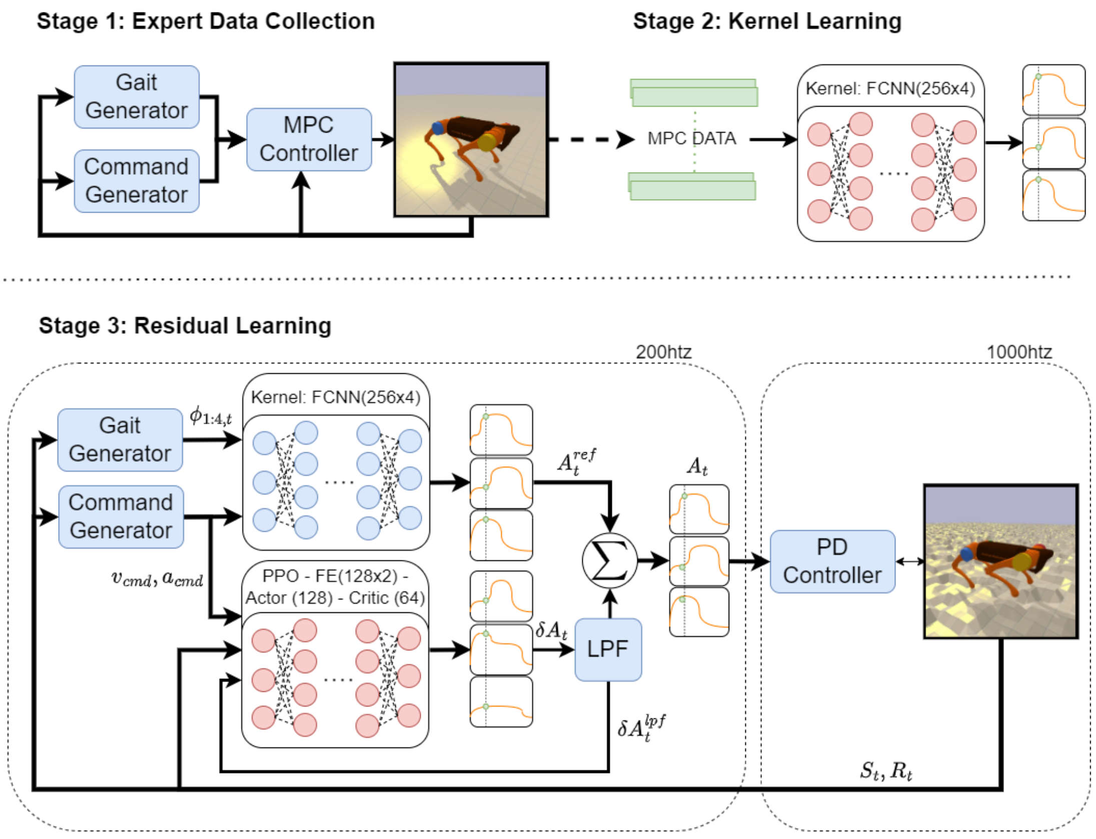


Figure 4.1: Overview of the multi-stage robot locomotion framework , where the red components represent trainable modules, and blue components represent fixed modules.

4.2.1 Environment

We simulate the A1 uni-tree quadruped[1] (Figure 4.2a) in the PyBullet[2] physics simulator for all the work carried out in this project. In addition, we wrap the PyBullet simulation in an Open-Ai Gym[3] environment for all RL experiments.

Several environments are used for different aspects of the training and optimization process. For training the kernel, data is collected from an expert controller navigating over a flat terrain (Figure 4.2a). Optimization of the RL agent is carried out using a moderately difficult height-field terrain with perturbations 0.35cm (Figure 4.2b), which

[1]https://www.unitree.com/products/a1/
[2]https://pybullet.org/wordpress/
[3]https://www.gymlibrary.ml/

is easy for the agent to learn on without requiring long training times. The final RL agent is trained on a variety of terrains, including height-fields with more significant perturbations (Figure 4.2c) and terrains simulating hills and mounds (Figure 4.2d); For all experiments the lateral friction of the ground is set to 1, and the navigation precision is set as $D_{min} = 0.5m$. Furthermore, through out all experiments we use the same four seeds ($\{150, 215, 345, 556\}$), which will be referenced throughout this paper.

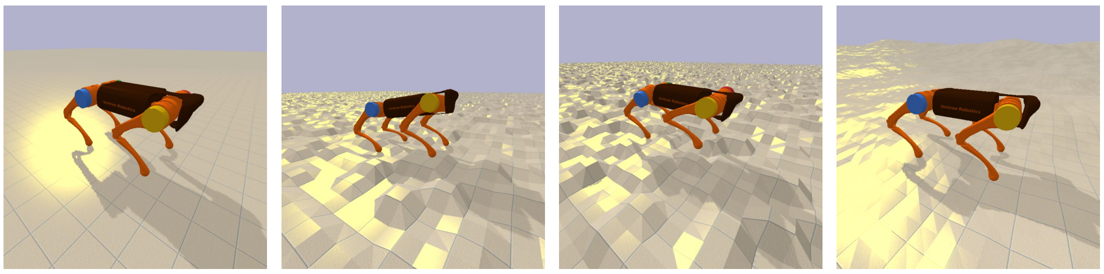

(a) A1 PyBullet (b) Height-field (3.5cm) (c) Height-field (6cm) (d) Perlin

Figure 4.2: Training terrains in the PyBullet simulation.

4.2.2 Command Generator

The command generator is responsible for generating target frontal, lateral, and yaw velocity commands. Given the target location pos_{target}, the robot's current location pos_{base}, and orientation orn_{base}, we can determine the desired velocity commands that will direct the robot to the target (Algorithm 2). It gradually increments the velocity commands at a frequency of 20_{htz}, with a maximum change of ± 0.005 to ensure smooth transitions in the velocity commands after a change in the pos_{target}. This benefits the controllers' performance, forcing it to execute a series of trajectories given a constant set of commands, which helps to prevent foot slippage.

4.2.3 Gait Generator

The gait generator developed is a modification of that from the work of Peng et al. [37], inspired by the simplicity and versatility Yang et al. [65] introduced. It is responsible for producing desired locomotion gait patterns in the form of a contact schedule (Figure 4.3b) according to its internal parameters: leg phases $\phi_{1:4} \in (0, 1]$, initial phases $\theta_{1:4} \in (0, 1]$, swing ratio $r_{swing} \in (0, 1]$, and stance duration τ_{stance}. .

Leg phase variables define the completion of a full step cycle for each leg; they are updated at each time-step (200_{htz}). Step cycles consist of two states: *stance* ($\phi_i > r_{swing}$), where the feet are in contact with the ground, and *swing* ($\phi_i \leq r_{swing}$) when not (Figure 4.3a). The parameter stance duration determines the amount of time the leg remains in the stance state. Thus we can determine the swing duration τ_{swing} (Equation 4.1) and the step cycle duration τ_{step} (Equation 4.2). Given the initial phases and the current time, we calculate the current phases using Equation 4.3.

$$\tau_{swing} = \tau_{stance}/(1 - r_{swing}) * r_{swing} \tag{4.1}$$

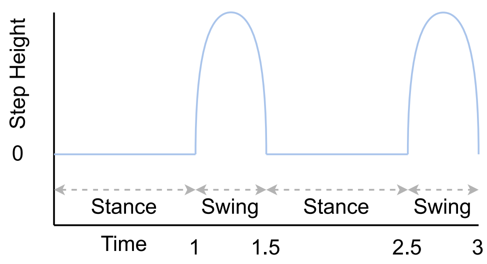
$$\tau_{step} = \tau_{stance} + \tau_{swing} \tag{4.2}$$

$$\phi_i = \theta_i + (\tau/\tau_{step}) \ mod \ 1 \tag{4.3}$$

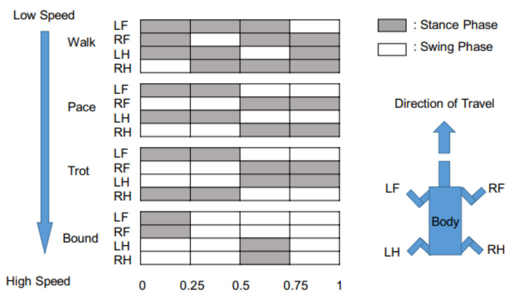
The initial phase variables are primarily responsible for coordinating each leg to produce the desired gait patterns (Table 4.1). While the parameters τ_{stance} and r_{swing} are most responsible for the feasibility of the gait pattern, for example, an excessively long stance duration can cause the robot to fall.

Gaits	θ_1	θ_2	θ_3	θ_4	τ_{stance}	r_{swing}
walk	0.	0.5	0.75	0.25	0.3	0.25
trot	0.9	0.4	0.4	0.9	0.3	0.4
bound	0.4	0.4	0.9	0.9	0.1	0.3

Table 4.1: Gait generator parameters for different gaits.



(a) Illustration of foot trajectory over step cycle phases.



(b) Illustration of common quadrupedal walking gaits, as a contact schedule [63].

Figure 4.3: Illustrations of gaits schedules and state characteristics.

4.2.4 PD controller

Given the foot positional target p_{ref}, we generate joint angle targets as inputs for the PD controller via inverse kinematics. We apply the torque values produced by the

PD controller to each corresponding joint motor at a rate of 1000_{htz}, as shown to be effective in MELA [64]. Figure 4.4 demonstrates the effect of implementing PD control at different rates, given a fixed target generation rate of 200_{htz} using the K_p and K_d parameters displayed in Table 4.2. We observe a significant error at a control rate of 1000_{htz}, which increases as the control rate decreases. Note that implementing a higher control rate requires considerably more compute.

Gains	abductor	hip	knee
K_p	100	100	100
K_d	1	2	2

Table 4.2: Parameters of the PD controller.

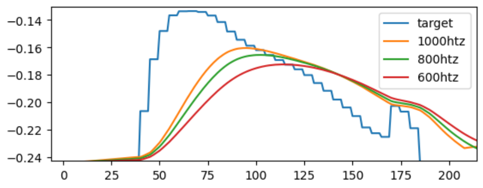


Figure 4.4: Demonstration of the foot position error in the z axis at different PD control rates. The blue line shows the target position, while the others show the true foot position after each control step.

4.2.5 Low Pass Filter

The role of the low pass filter (LPF) is to smooth noisy trajectories, which would otherwise result in poor performing and jittery locomotion strategies that are likely to be infeasible. The Kernel replicates the MPC controllers' trajectories, which are known to be feasible. Therefore, we do not smooth the trajectories produced by the kernel, only the residual outputs of the agent (Equation 4.4), as shown to be effective by Kasaei et al. [21].

$$\delta\mathcal{A}_t^{lpf} = \alpha * \delta\mathcal{A}_t + (1-\alpha) * \delta\mathcal{A}_{t-1}^{lpf} \tag{4.4}$$

Equation 2.6: Where $\delta\mathcal{A}_t^{lpf}$ is the residual after passing through the LPF, $\delta\mathcal{A}_t$ is the residual produced at the current time-step, and α is the smoothing factor.

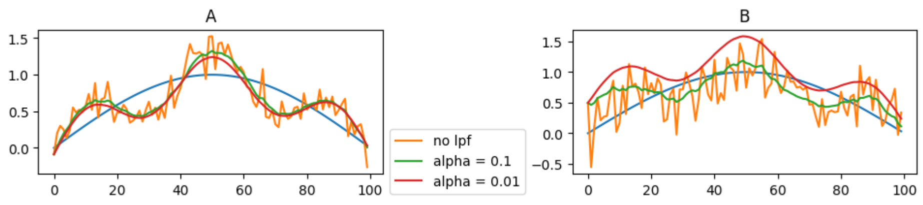


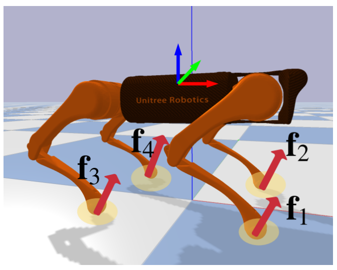
Figure 4.5: Illustration of the effect of the LPF on the residual outputs. The blue line showing the kernel trajectory, the orange showing unpassed residual outputs summed with the kernel trajectories, the green and red show passed residual outputs.

We set the smoothing factor to 0.1 for all experiments; this sufficiently removes most of the noise (Figure 4.5A). Note that leaving some noise can be beneficial for policy exploration. Furthermore, it can improve responsiveness especially during highly noisy instances where over smoothing can introduce bias (Figure 4.5B).

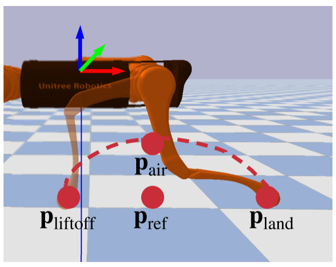
4.2.6 Kernel

4.2.6.1 Expert MPC Controller

We use the MPC controller from [37, 65] as our teacher model to train the kernel, the code for which is openly available[4]. To collect the expert data, we use the command generator (Section 4.2.2), gait generator (Section 4.2.3), and PD controller (Section 4.2.4) with the MPC controller described in this section.



(a) Stance controller, optimizing the ground reaction forces [65]


$\mathbf{p}_{\text{air}}$
$\mathbf{p}_{\text{liftoff}}$
$\mathbf{p}_{\text{ref}}$
$\mathbf{p}_{\text{land}}$

(b) Swing controller, producing parabolic foot position trajectories [65]

Figure 4.6: Illustration of the MPC controller functions during the swing and stance states, using a different controller for each state.

While in the *swing* state, as determined by the gait generator, we use Raibert Heuristics [41] to create positional foot target trajectories and the PD controller for

[4]https://github.com/google-research/motion_imitation

actuating the joints. Raibert Heuristics generates a trajectory following a parabola fitted to the points (Figure 4.6b): lift-off position $p_{liftoff}$, landing position p_{land}, and the mid-point. $p_{liftoff}$ is the foot's position at the point of entering the swing state. p_{land} is determined by the target velocity commands and current velocities (Equation 4.5) and is calculated in an open-loop fashion. The mid-point is at the height of p_{air} above p_{ref}, which is the default foot position; p_{air} is a controllable parameter that determines the step height. We execute 80% of the trajectory during the first 50% of the *swing* phase. As such, we have more time to execute the remaining swing trajectories resulting in more accurate landing positions and preventing early touchdowns.

$$p_{land} = p_{ref} + v_{hip} * (v_{cmd,hip} - v_{hip}) * \tau_{stance}/2 \tag{4.5}$$

During the *stance* state we use short-horizon MPC control to optimize the ground reaction forces of the feet in contact with the ground (Figure 4.6a). The MPC ensures the robot base tracks a given reference trajectory, which is determined by the velocity commands. We use the *template*, Central Dynamics Model [6] to simplify the complexity of the optimization, where the robot is considered as a rigid-body with massless legs.

4.2.6.2 Data Collection Process

Using the MPC controller (Section 4.2.6.1) described above, we collect a data set to train the kernel. We run the controller to implement the trot gait, navigating the robot to 500 consecutive target locations at a minimum distance of 2.5m in a random direction. During this, we collect the data $\{v_{base},\ a_{base},\ v_{cmd},\ a_{cmd},\ q,\ \dot{q},\ \phi_{1:4},\ p_{1:4},\ CoM\}$ before actions are taken, $\{p^{ref}_{1:4},\ p'_{1:4}\}$ and after each time-step($200htz$); See Table A.1 for definitions of the variables.

4.2.6.3 Training and Analysis

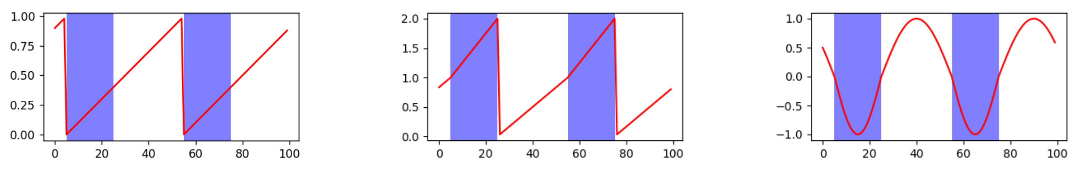
Yamamoto et al. [63] achieved successful locomotion using time-dependant oscillatory inputs to the decoder. As such, we hypothesised that the locomotion phases $\phi_{1:4}$, which are time-dependant, and velocity commands v_{cmd} and a_{cmd}, would be an adequate signal for an MLP to reproduce functional locomotion trajectories. With the velocity commands allowing us to improve upon the work of Yamamoto et al. [63], enabling omni-directional and continuous velocity control. Furthermore, we could improve the kernel performance by giving additional inputs such as the center of mass and joint level information as Mitchell et al. [34] and Surovik et al. [53] showed to be effective with their generative models.

Training Labels: The kernel aims to replicate the realized foot trajectories of the robot. Therefore, when the MPC controller uses PD control (swing states), we use the generated target trajectories p_{swing}^{ref} as labels. Otherwise, the realized kernel trajectories would exhibit additional error from the targets due to the lag of the PD controller (Figure 4.4). As we do not have target trajectories for the stance legs, we instead use the resulting foot positions after taking actions p'_{stance} for stance trajectory labels.

Phase Representation: Passing raw leg phases to the network would prevent it from being able to generalize to gaits with a different r_{swing} (Figure 4.7a). The network would associate specific phase values with either swing or stance trajectories. Therefore, we use a transformed normalized phase $|\phi_i|$ (Equation 4.6), where the phase is greater than one during the state swing (Figure 4.7b), irrespective of gait parameters. We cannot simply use the normalized phase, which defines the completion of a step cycle state ($\in (0,1]$), as there would be no differentiation between the swing and stance states.

$$|\phi_i| = \begin{cases} 1 + (\phi_i / r_{swing}), & \text{if } \phi_i <= r_{swing} \\ (\phi_i - r_{swing})/(1 - r_{swing}), & \text{otherwise} \end{cases} \tag{4.6}$$

Alternatively, we could represent the phase as a wave by applying a sin function to the phase ($sin(|\phi_i| * \pi)$). This removes the sudden drop in the phase as the state transitions from swing to stance, and transitions between states occurs at zero. However, this introduces issues; each point in the phase, for both swing and stance, correspond to two trajectories (Figure 4.7c). Therefore, this would be most appropriate with a time series model.



(a) Global phase (b) Normalized phase (c) Normalized sin phase

Figure 4.7: Visualisation of phase representation options, where the shaded blue region corresponds to a swing state.

Initial experiments When training the kernel with expert trot data, as in Table 4.1.

The network learns, however, overfits with a high validation loss (Table 4.3), resulting in a dysfunctional locomotion controller. Analyzing the behavior of the MPC controller, it is clear that there is instability in its motion, which can be seen in the robot *rocking* as it works to maintain balance. Reducing the stance duration prevents this behavior, shown by the reduction in the variance of it's center of mass (Figure 4.8). Long stance durations cause longer swing durations. Therefore, the robot must maintain a stance with only two grounded feet for longer, where the support polygon is small, making it easy to lose balance. As the robot works to maintain balance, the relationship between $\phi_{1:4}$, v_{cmd}, a_{cmd} and the trajectories becomes weak. As such, multiple distinct labels exist for given inputs in the dataset, causing model confusion. For this reason we learn a better kernel with data collected using a lower stance duration, as shown in Table 4.3.

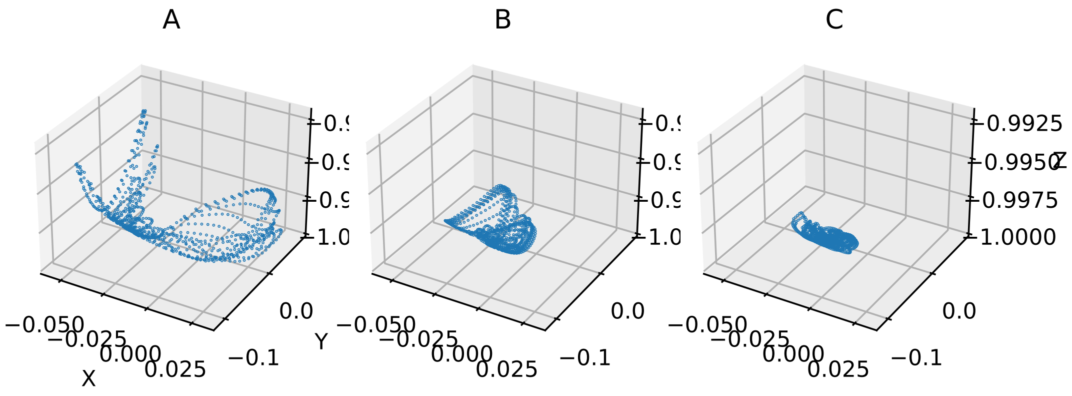


Figure 4.8: Center of Mass of the robot using the MPC controller varying the stance duration: A) $\tau_{stance} = 0.3$, B) $\tau_{stance} = 0.25$, C) $\tau_{stance} = 0.2$

Optimization (Kernel-base): With the data collected using a stance duration of 0.2, we use Bayesian Optimization via Optuna[5] for hyperparameter tuning, training for ten epochs, and minimizing the final validation loss. The search space and results for this optimization are shown in Table A.2. The optimized model achieves an extremely low validation loss (L1=$6.2e-4$, Table 4.4), demonstrated by the closely aligned predicted trajectories in Figure 4.9. The trained kernel produces a high-performing locomotion controller in simulation (video available), validating our initial hypothesis.

Data dependency: The results discussed above were achieved training on 80% of the data, corresponding to walking to 400 target locations and 2.1hrs of locomotion data.

[5]https://optuna.org/

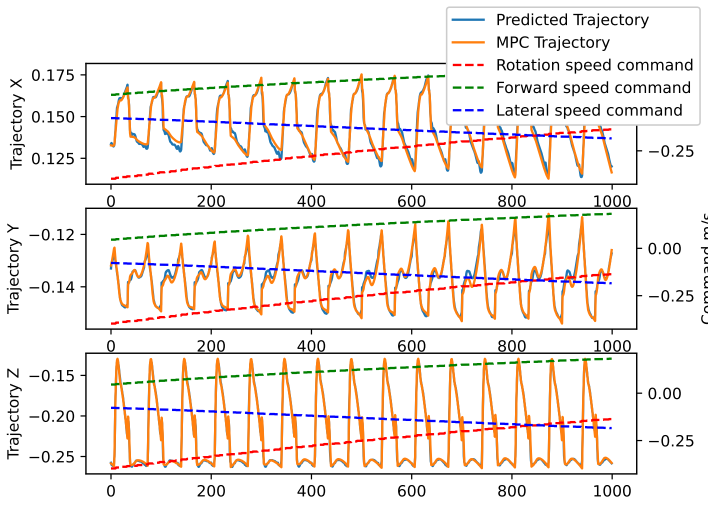


Figure 4.9: Front right foot trajectories produced by the MPC control compared to the kernel-base as the input velocity commands change given a stance duration of 0.2

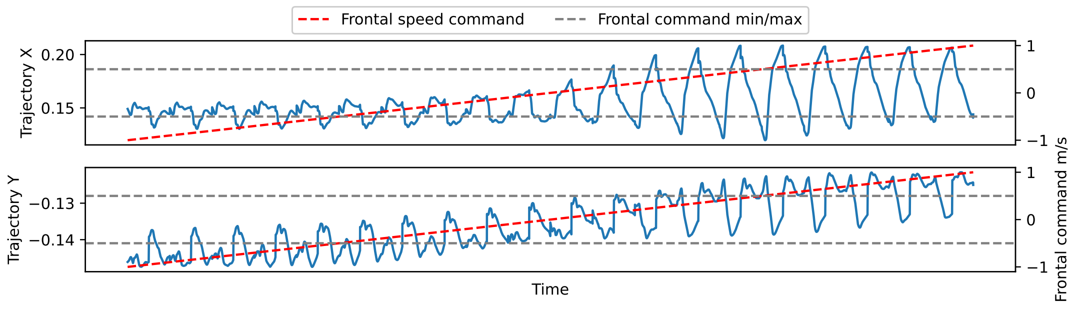
Collecting this amount of data may not be possible; Ideally, we can learn a controller with relatively little expert data. Table 4.5 shows the results of training with less data, with each experiment using the same validation set. We observe a deterioration in the validation performance as the number of targets decreases. However, training with only ten target locations (3.1 minutes), the kernel achieves a validation loss of $1e-3$, which results in a functional controller in simulation.

4.2.6.4 Kernel Trajectory Analysis

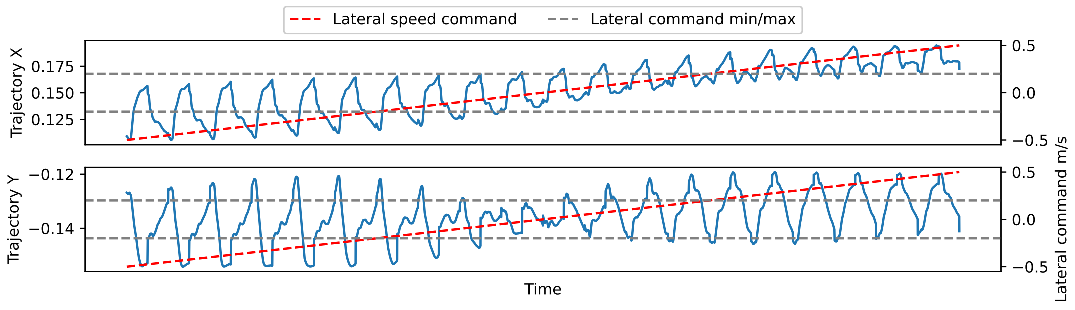
Dynamic Control: The kernel demonstrates the ability to act as a dynamic controller by producing trajectories that vary depending on the commands we pass (Figure 4.10). We observe that the trajectories produced on the z-axis appear independent of all of the velocity commands, as expected.

Increasing the frontal velocity commands produces trajectories with greater step length, as shown by the increasing magnitude of the oscillation in the x-dimension (Figure 4.10a). Also, the trajectories move closer to the body, as shown by the range of the oscillations in the y-dimension reducing. Lateral velocity commands also affect both x and y-dimension trajectories. Contrary, lower lateral commands result in greater step lengths (Figure 4.10b). Additionally, lateral commands of greater magnitude cause the oscillations in the y-dimension trajectories to increase symmetrically. We see little change in the x-dimension trajectories by changing the angular velocity commands; However, we note that the effect in the y-dimension also appears symmetric (Figure

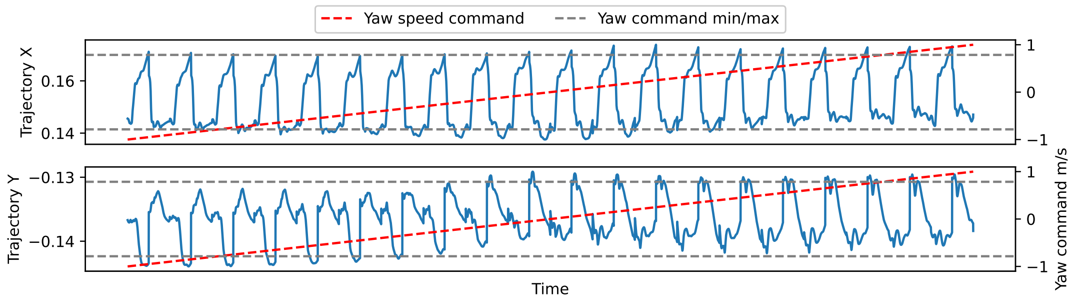
4.10c).



(a) Frontal Velocity (0 to 1m/s)



(b) Lateral velocity (-0.5 to 0.5m/s)



(c) Yaw velocity (-1 to 1m/s)

Figure 4.10: Kernel-base front right foot trajectories with varying velocity commands. All commands fixed to zero other than the command being varied. The grey lines indicate the min and max values passed in training. The red line shows the velocity command input.

The observed trends in the trajectories deteriorate when the commands move outside the observed limits during training (grey lines Figure 4.10). This suggests that the model cannot generalize effectively to such commands. Therefore, we would not be able to produce a gait that moves at significantly greater velocities than provided during training without speeding up the step cycle rate.

Symmetry in legged locomotion is expected, considered good, and often targeted as a behavioral feature from RL-controlled robots [1, 67]. The gait generator enforces some degree of symmetry in the trajectories shown by the alignment in the x and z-dimension. However, true symmetry would imply that the front-right and rear-left y-dimension trajectories should be the inverse of each other when there are no yaw commands (e.g. walking straight or in place). This is not observed, therefore, trajectories produced by the kernel are not perfectly symmetric (Figure 4.11). However, the symmetry improves as the frontal velocity increases, suggesting the trained kernel has some deficiencies, these may be inherited from the MPC controller, or a consequence of low data in this region of input space.

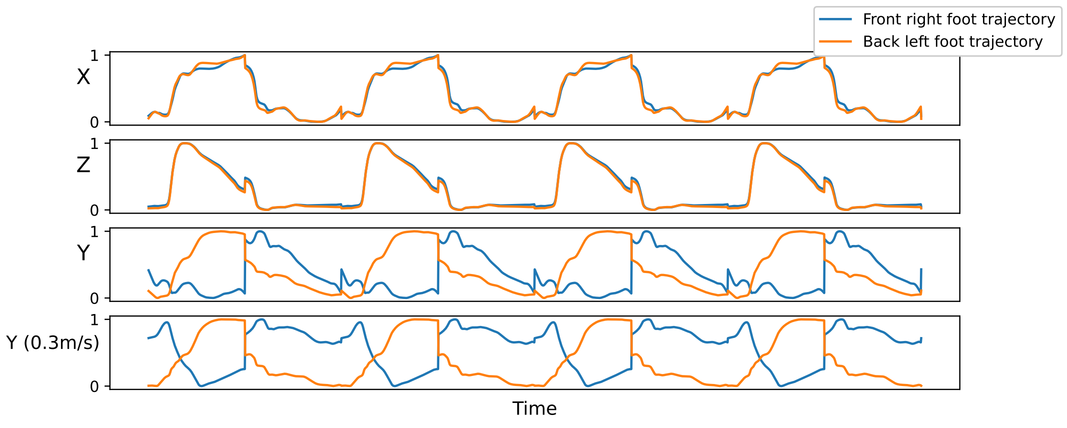
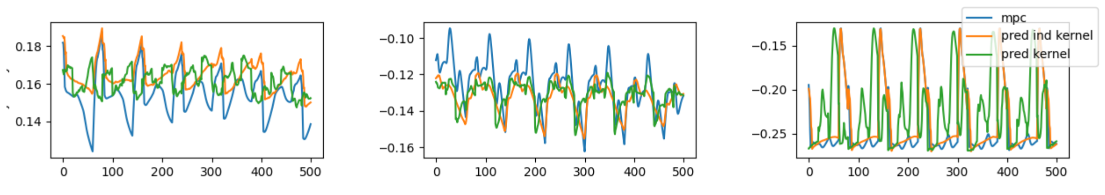


Figure 4.11: Normalized kernel-base trajectories for the front right and back left legs, showing asymmetry in the y-dimension at 0m/s velocity commands, with symmetry becoming apparent at a frontal velocity of 0.3m/s

4.2.6.5 Further Experiments

Gait generalization (Kernel-ind): Yamamoto et al. [63] show the capability to generalize to unseen gait patterns. This attribute would be beneficial for promoting the versatility of the framework by enabling multi-modal locomotion. We hypothesized that the kernel-base would fail to generalize for other gait patterns due to using a fully connected network, thus modeling the relative phases of the gait. Using other gait parameters to generate the phases would result in poor predictions due to the unseen relative phases. We show this to be true in Figure 4.12, where the kernel produces erratic trajectories in the z-dimension.

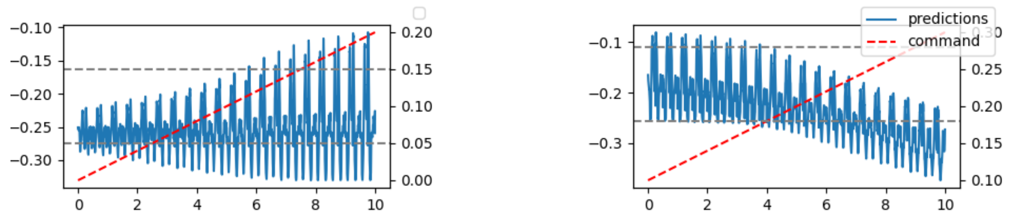


(a) X-Dimension (b) Y-Dimension (c) Z-Dimension

Figure 4.12: Comparison of the front right foot trajectory predictions between the kernel-base model and kernel-ind. Showing that the kernel-ind can potentially produce trajectories for unseen gaits.

Modeling each leg individually significantly reduces the z-dimension errors (Figure 4.12c). This version (kernel-ind) receives velocity commands, a single leg phase, a one-hot encoding representing the target leg, and only predicts the positional trajectory for that target leg. As such, the relative phases are ignored. This has a substantial effect on the z-trajectories, although significant errors in the x and y-trajectory predictions persist (Figures 4.12a and 4.12b), resulting in weak simulation performance (executing the walk gait), albeit improved upon kernel-base (video available).

Furthering the Kernel capabilities (Kernel-ext): The MPC controller with a stance duration of 0.2, shows robust stability when altering step and ride heights, exhibiting minimal change in the variance in the center of mass (Figures A.2 and A.3). We collect data, randomly selecting step height (∈[0.05,0.18]) or ride height (∈[0.18,0.28]) settings before walking to each target location. Training the kernel on this data (with the settings as additional inputs) causes a minimal increase in the validation loss (L1=$7.1e-4$, Table 4.4), while allowing us to control the ride and step heights when using it as a controller (video available). Figure 4.13 shows how the z-trajectories change with respect to the step and ride height commands.

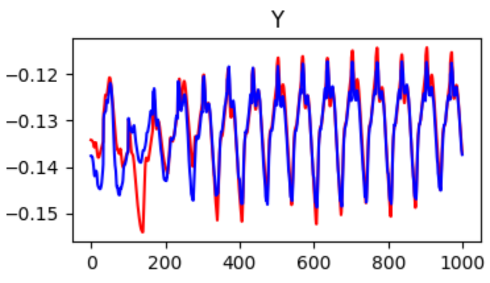


(a) Step height commands (b) Ride height commands

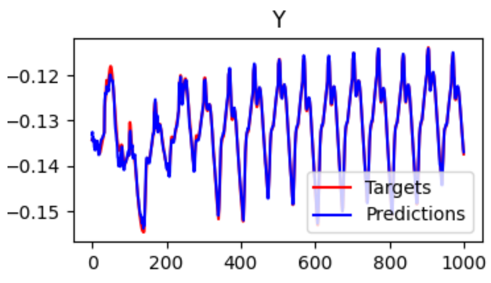
Figure 4.13: Kernel-ext front right foot z-trajectories with varying step and ride height commands, where the grey lines show the min and max values provided during training.

Addressing dynamic capture: Kernel variants thus far do not capture the robot's dynamics but regresses the trajectories produced, limiting the kernels' versatility. Hence we cannot learn a kernel of MPC controllers that produce high-variance trajectories. Conversely, Surovik et al. [53] and Mitchell et al. [34] achieve this while providing additional features of the robot state. We found including additional variables $\{v_{base}, a_{base}, q, \dot{q}, CoM\}$ (**Kernel-dyn1**) reduced the validation loss (L1=$2.5e-4$, Table 4.4); however, the kernel does not produce functional locomotion in simulation.

We hypothesised that poor trajectory prediction at the start of locomotion (Figure 4.14a) caused the robot to enter states outside the training distribution resulting in insufficient control performance. We resolve this (Figure 4.14b) and significantly reduce the error (L1=$3.1e-4$, Table 4.4) by collecting data where the robot is reset after reaching each target location (**Kernel-dyn2**). Furthermore, we stack a history of 5 input states and use the normalized sin phase representation as we have a time series input. Yet, we still do not have a functional controller that captures the robot's dynamics (video available).



(a) Kernel-dyn1 predictions



(b) Kernel-dyn2 predictions

Figure 4.14: The kernel-dyn1/2 front right foot trajectory predictions at the start of locomotion compared to the MPC controller targets.

4.2.7 Kernel Results

τ_{stance}	0.3	0.25	0.2
mean validation loss	5.9e-3	2e-3	1.6e-3
functional locomotion	False	False	True

Table 4.3: Performance of the kernel given different stance durations used for data collection, using the manually tuned kernel hyperparameters: LR 0.001, Linear LR decay 0.8, Dropout 0.2, Batch-normalization True, Batch-size 100, Network (256x3), Activation Tanh, Loss L1

Kernel-variant	Mean Validation Loss	Standard Deviation
Kernel-base	$6.2e-4$	$6.9e-6$
Kernel-ind	$7.2e-4$	$4.9e-6$
Kernel-ext	$7.1e-4$	$1e-5$
Kernel-dyn1	$3.1e-4$	$6.1e-6$
Kernel-dyn2	$2.5e-4$	$5.6e-6$

Table 4.4: Performance of kernel variants, showing the mean minimum achieved validation loss and the standard deviation. Hyperparameters: LR 0.0024, Linear LR decay 0.7, Dropout 0.0002, Batch-normalization False, Batch-size 200, Network (256x4), Activation ReLU, Loss L1

Number of Targets	10	25	50	100	200	400
Mean Validation Loss	$1e-3$	$9.1e-4$	$8.4e-4$	$7.7e-4$	$6.7e-4$	$6.2e-4$
Standard Deviation	$3.1e-6$	$9.8e-6$	$5.3e-6$	$6.2e-6$	$5.5e-6$	$6.9e-6$

Table 4.5: Kernel-base performance as the amount of data increases.

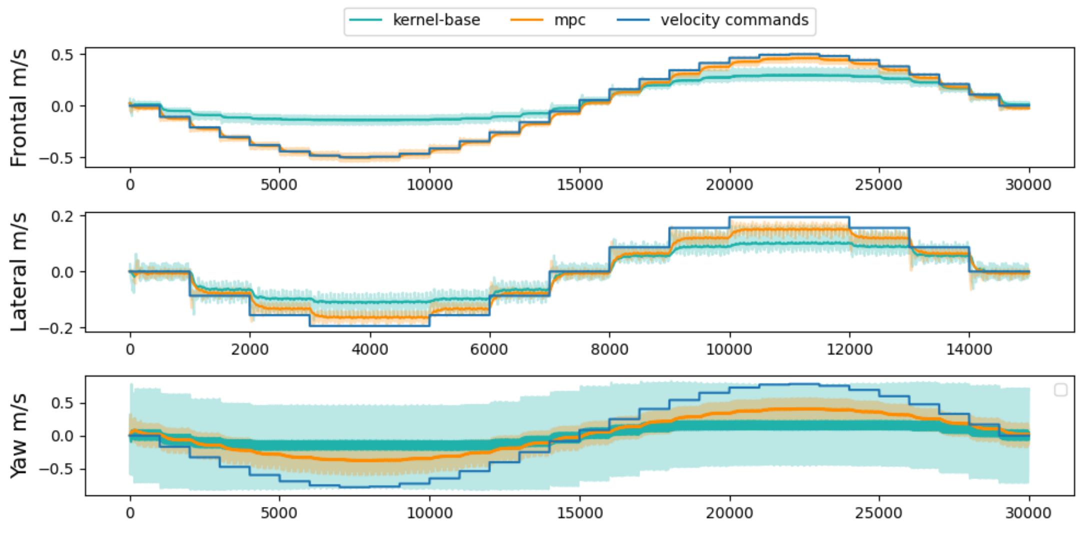


Figure 4.15: The realized velocities of the robot given velocity commands for the MPC controller and kernel base.

Figure 4.15 shows the performance gap between the kernel-base and the MPC controller. The kernel cannot move at negative frontal velocities while not being able to match the maximum positive frontal and all lateral velocities of the MPC controller. This verifies that the kernel cannot generalize to velocities greater than experienced in training. Furthermore, the kernel experiences extremely high variance when turning, showing a significant performance gap in the realized yaw velocities compared to the MPC controller.

4.2.8 RL Agent

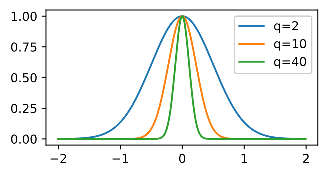
We chose an initial architecture, reward function and state feature set to be improved, using kernel-base to provide the reference trajectories. Each aspect is optimized individually for efficiency, starting with the PPO hyper-parameters, reward function, and finally the state features.

Initial state features: The initial state feature set consists of core attributes used throughout all the experiments and optional attributes. The core attributes are selected as they appear frequently in the literature, or are required for our formulation.

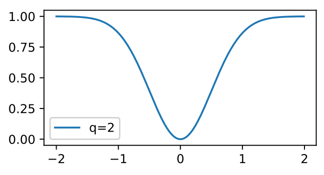
- Core attributes: $\{v_{base}, a_{base}, v_{cmd}, a_{cmd}, q, \dot{q}, CoM, pitch_{base}, roll_{base}\}$.
- Optional attributes: $\{\phi_{1:4}, footcontacts_{1:4}\}$

We choose to use the phase variables $\phi_{1:4}$ rather then the reference trajectories typical used in residual methods [8, 20, 61]. This provides consistency between the kernel and agent inputs. Furthermore, as the kernel is deterministic, and the agent receives all other kernel inputs, the kernel trajectories (reference motions) are fully described.

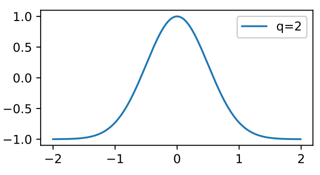
Reward Function Formulation: Reward functions have a large impact on the performance of reinforcement learning systems. We use radial basis functions (RBF 4.16a) to define each feature of the reward function as they have shown effectiveness in other works [30, 64]. They limit the maximum reward to one (by default), allowing us to determine the maximum reward per episode. They provide a smooth reward curve for the algorithm to follow, and can prevent unwanted penalties which may occur using nominal rewards. Typically we use positive RBFs, but we can also use negative 4.16b, and mixed 4.16c RBFs. A steeper RBF function (Figure 4.16a) incentivises learning, and accommodates for attributes with small numeric errors.



(a) RBF positive rewards: $r_i = exp(-(\gamma' - \gamma)^2 * q)$



(b) RBF negative rewards: $r_i = -exp(-(\gamma' - \gamma)^2 * q) + 1$



(c) RBF mixture of rewards: $r_i = 2 * exp(-(\gamma' - \gamma)^2 * q) - 1$

Figure 4.16: RBF reward functions: γ' is target value, γ is the measured value, and q is a parameter to determines the steepness of the reward function.

Initial reward function: The initial reward function consists of four features ($f_i \in \mathcal{F}$): the center of mass (CoM), the linear velocity, the angular velocities, and the distance to the target location. It is worth noting that the CoM and velocity features are vectors, which can be integrated to the RBF by taking the euclidean distance between the target and current value. Table 4.6 details the parameters and Equation 4.7 shows the baseline reward function, where ϕ represents the radial basis function.

Reward Feature	γ'	γ	q	weight (ω)
Linear velocity	v_{cmd}	v_{base}	18.42	0.01
Angular velocity	a_{cmd}	a_{base}	7.47	0.01
Center of mass	[0,0,-1]	CoM	2.35	0.01
Distance to target	0	D_{target}	0.74	0.01

Table 4.6: Initial reward function and parameters, RBF parameters taken from [64]

$$R_t = \sum_{f_i \in \mathcal{F}} \omega_i * \phi(\gamma'_i, \gamma_i, q_i) \tag{4.7}$$

4.2.8.1 hyper-parameter tuning

Given the initial state feature set and reward function, we first optimize the PPO algorithm's hyper-parameters to ensure that further optimizations will train effectively. We identify candidate solutions using a random search of one hundred trials over the hyper-parameter search space (Table A.3), training for three million time steps. Taking the top 5 candidates, we train them for five million time-steps with four seeds 4.2.1, as shown in Figure 4.17. Trail 3 is selected, and used for all remaining experiments due to achieving on par performance while using the smallest network, with few epochs.

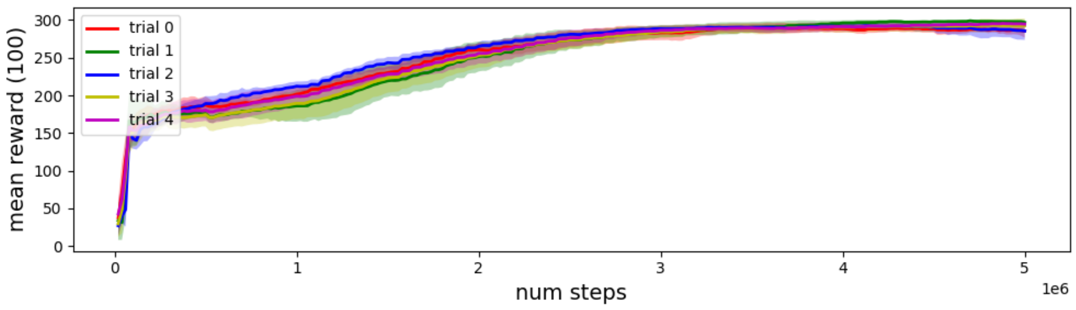


Figure 4.17: Mean and standard deviation of the top 5 hyper-parameter candidates. Selecting Trial 3: learning rate (lr): 0.001 , lr decay: $1e-7$, batch-size: 4000 , number of epochs: 10, roll-out length: $2e5$, entropy coefficient: $5e-6$, network: Feature extractor (128x2) Actor (128) Critic (64)

4.2.8.2 Reward Function Design

Feature Experiments: Second, we optimize the reward function features as the included terms are critical for effective learning, ensuring it promotes learning the correct behaviors in later optimizations; Table 4.7 lists the experiments conducted. The mean cumulative reward is a uninformative metric as each experiment affects the available reward per episode, instead we track the metrics below.

- **Target count**: This is the primary metric we optimize for, tracking the number of target locations the robot can navigate to in 60 seconds.
- **Success rate**: Navigating to at least two target locations and not falling.

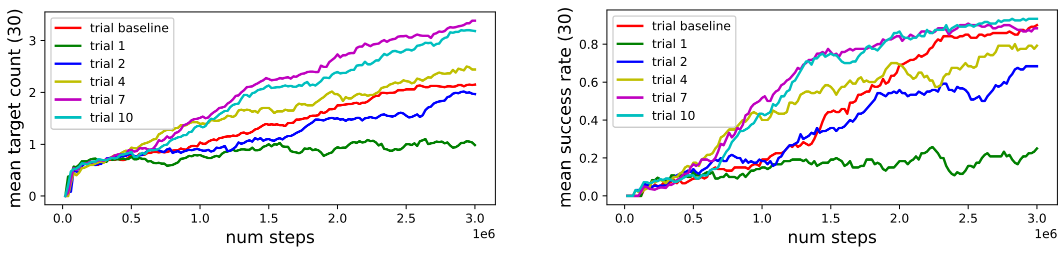
Experiment group one validated the importance of the CoM and target distance features; removing these in trials 1 and 2 reduced the average target count and success rate. Furthermore, Trial 4 shows that relaxing the velocity constraints in the reward function improves performance (Figure 4.18a). However, it achieves a lower success rate (Figure 4.18b); this is not necessarily bad as it may result in greater state space exploration, and make the model more robust when training for longer. Experiment group two, which builds on the reward function from trial 4, significantly improves the success rate and target counts by including the nominal reward features for falling and reaching targets, with Trial 7 performing best.

Experiment group three aimed to eradicate foot slippage observed in the motion behavior (learned using Trial 7) to improve the performance. Of these experiments, foot slippage was reduced by trials 9 and 10, with 10 performing best. We see a minor reduction in the target count, and the motion displayed unfavorable characteristics, such

Trial ID	Description
Exp group 1	The following builds upon the initial reward function described in Section 4.2.8
Trial 1	Removes the CoM feature.
Trial 2	Removes the target distance feature.
Trial 3	Introduces additional velocity constraints, tracking the vertical, pitch, and roll velocities with targets set to zero.
Trial 4	Removes velocity constraints, prevents losing reward for exceeding the velocity commands.
Exp group 2	The following trials build upon Trial 4 as this achieved the best results.
Trial 5	Includes a penalty of 5 for falling.
Trial 6	Includes a reward of 5 for reaching a target.
Trial 7	Includes both the penalty and reward used in trials 5 and 6.
Exp group 3	The following trials build upon Trial 7 as this achieved the best results.
Trial 8	Foot slippage reward (RBF), tracking the distance between stance leg contact positions at timestep t and t-1, setting the target to 0.
Trial 9	Foot slippage penalty (RBF), tracking the distance between stance leg contact positions at timestep t and t-1, setting the target to 0.
Trial 10	Foot slippage penalty (RBF), tracking the distance between stance leg initial contact positions and at timestep t, setting the target to 0.

Table 4.7: Reward function features experiment list, grouped into batches.

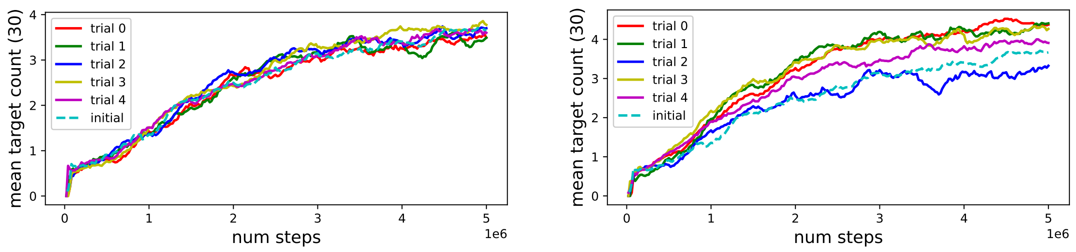
as walking on the two front legs. Therefore for further optimization experiments, we continue with the reward function from trial 7.



(a) Mean number of targets reached (b) Mean success rates

Figure 4.18: Performance of significant reward function feature experiments.

Reward Function optimization: The RBF parameters and the reward feature weights of the reward function are optimized individually over two random search experiments of 100 trials training for three million timesteps. The RBF parameters search space can be found in Table A.4, while the search space over the weights is the continuous range $[0.001, 0.005]$. Figures 4.19a and 4.19b shows the mean target counts e of the 5 best candidate trials from the searches, trained using each seed 4.2.1. The RBF parameter space optimization failed to improve performance. However, we find the best-performing reward function from the search over the weight space (Figure 4.19b). The final reward function is detailed in Table 4.8 and Equation 4.8.



(a) Top five RBF parameters (b) Top 5 reward feature weights

Figure 4.19: Optimization performance of the top five parameters found in the random search for the search over the RBF parameters and reward feature weights.

RBF reward features $f_i \in \mathcal{F}$	γ'	γ	q	weight
Linear velocity	v_{cmd}	v_{base}	18.42	0.0076
Angular velocity	a_{cmd}	a_{base}	7.47	0.0264
Center of mass	[0,0,-1]	CoM	2.35	0.0298
Distance to target	0	D_{target}	0.74	0.0169
Nominal reward features $r_i \in \mathcal{F}_{nom}$	Reward function			weight
Falling penalty	$r = \begin{cases} -19.8, & \text{if the robot fell} \\ 0, & \text{otherwise} \end{cases}$			1
Target reached	$r = \begin{cases} 8.75, & \text{if } D_{target} \leq D_{min} \\ 0, & \text{otherwise} \end{cases}$			1

Table 4.8: The optimized reward function parameters.

$$R_t = \sum_{f_i \in \mathcal{F}} \omega_i * \phi(\gamma'_i, \gamma_i, q_i) + \sum_{r_i \in \mathcal{F}_{nom}} r_i \tag{4.8}$$

4.2.8.3 State Space Search

The optional state features; leg phase and foot contacts are proved beneficial for learning (Figure 4.20). Additionally the locomotion phase results in improved learning compared to reference trajectories from the kernel (which are typically used [8, 61]).

As the agent outputs are passed through the LPF, we break the Markov Property of the MDP. Such that future states and rewards are no longer independent of past actions given the current state [54]; Including the residuals in the state resolves this and provides a performance boost. Although this is not done in other work, we often see the actions taken at previous timesteps being passed [26], which is a less elegant approach to resolve the same issue. Interestingly, once introducing the residual into the state space, the leg phase is no longer beneficial (Figure 4.20). As such, the best-performing state space includes only the residual and the foot contact state as the optional features.

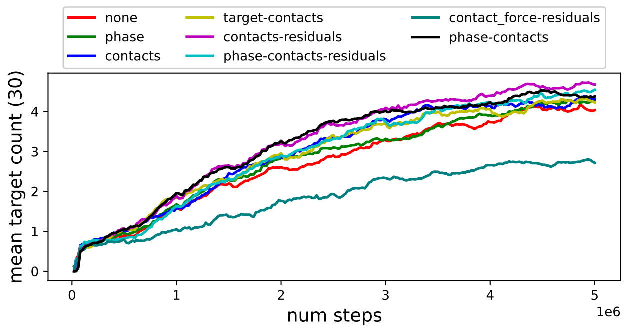


Figure 4.20: The performance using alternative state features. Contacts: The foot contact states of the robot, Targets: The kernel trajectories predicted A_t^{ref}, Contact_force: The force the foot exerts on the ground, Residuals: δA_{t-1}^{lpf}, Phase: $\phi_{1:4}$.

4.2.8.4 Alternative architectures

As the agent learns most effectively with the residual and foot contact state as optional state features, there is potential for a greater degree of flexibility with the architecture of the framework. Inspired by Kasaei et al. [21], we use the agent to generate the velocity commands passed to the kernel.

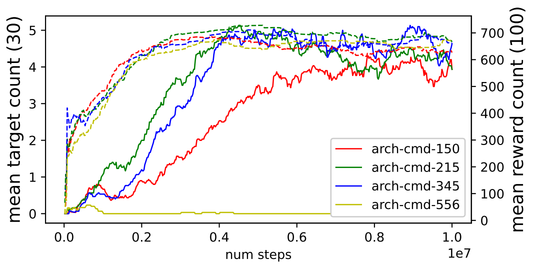


Figure 4.21: Alternative architecture, where the agent generates velocity commands: showing the training rewards (dashed line) and target counts.

The alternative architecture shows promise with some runs; however, others fail to produce the desired behavior, as shown by the inability to navigate to target locations

(Figure 4.21). Furthermore, the runs that fail to reach targets consistently increase the accumulated reward, suggesting the reward function is unsuitable. The model is not analyzed further due to time constraints; however, it is likely that the emphasis on the CoM in the reward function incentivises the agent to produce zeros as velocity commands to the kernel, where the robot is most stable.

4.2.9 Training The Final Agent

Terrains: We train the final agent in a randomly selected terrain, with a 75% probability of choosing a height-field terrain (Figure 4.2c) and 25% chance of perlin (Figure 4.2d. Height-field terrains are further varied, sampling the height perturbation uniformly ([3cm, 4.5cm]). Selected terrains are used for five consecutive episodes before resampling a new terrain.

Force perturbations are applied to the robot randomly, sampling the interval between force application ($\in [5,8]$ seconds), the magnitude of the force applied ($\in [100,350]$ newtons), the location of the applied force on the robot's body, and the direction in which it is applied. The force is applied for 0.3 seconds and directed slightly upwards to make falling more likely. This aims to ensure thorough state space exploration during training and allows the agent to learn a more robust policy.

Implementation details: Policy roll-out, is executed over five cpu's in parallel, for a total of $2e7$ timesteps. We train 4 policies using the four seeds 4.2.1, shown in Figure 4.23. Furthermore, we stop decaying the learning rate after $3e6$ timesteps to prevent policy stagnation. We observed continued decay resulted in the KL divergence of the policy to be ~ 0, therefore nothing is being learned past this point. Stopping the decay at $3e6$ timesteps ensured consistent non-zero KL divergence that resulted in a reasonable amount of clipping ($\sim 10-20\%$).

Debugging: The reward function developed (**agent-base**, Table 4.8) does not maintain a level base when on a slope, causing the robot to lose balance and take extreme actions to regain its center of mass (Figure 4.22). Adding a roll and pitch reward term with the target set at zero solved this (**agent-rp**). Achieving the best performance using the gravity terms' weight and the angular velocity terms' steepness (Table 4.8).

Additionally, we note that using a large target threshold D_{min} during training limits the precision of the framework's navigation capability. Attempting to navigate closer

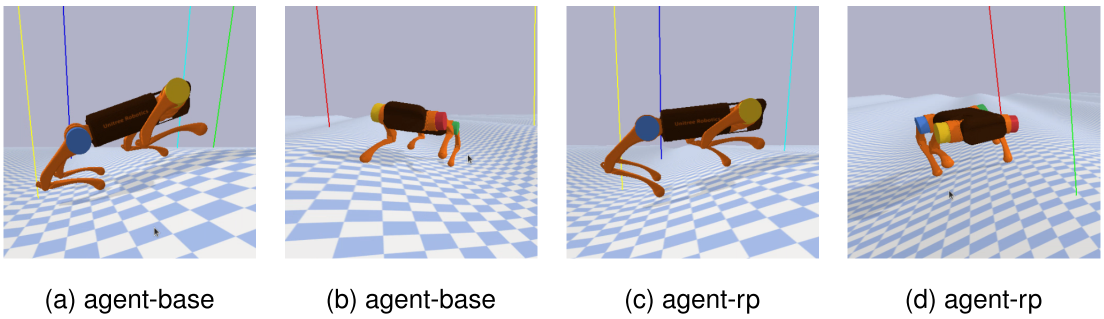


Figure 4.22: A and B show how the agent trained with the initial reward function (agent-base) failed to maintain a level base on slopes. C and D show the result after introducing the roll and pitch term to the reward function (agent-rp).

than the training threshold causes the robot's CoM to move over its front legs (Figure A.4) and fall over if on a negative slope.

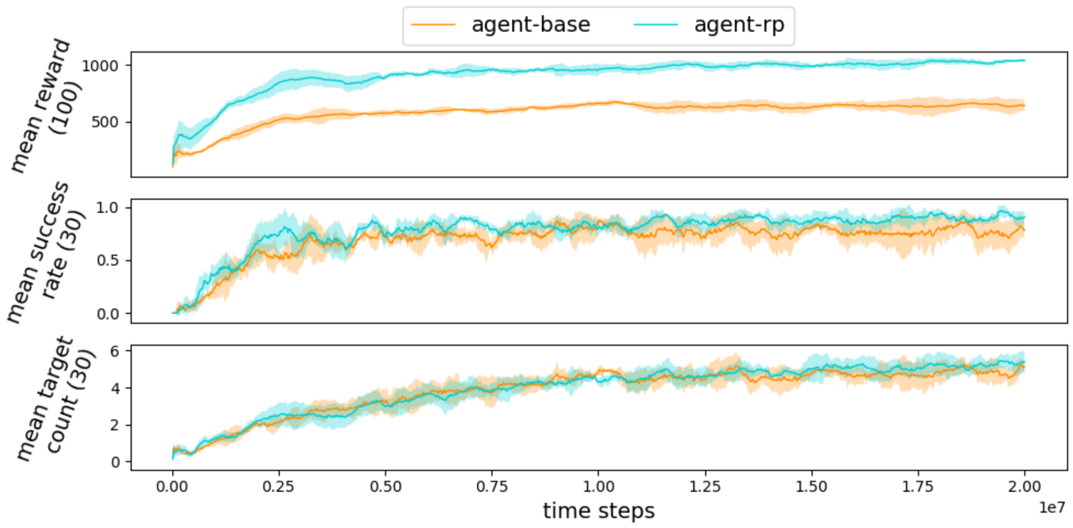


Figure 4.23: Training of agent-base and agent-rp used for evaluation. Showing greater success rates, with less variance using agent-rp.

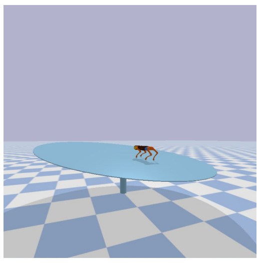
Chapter 5

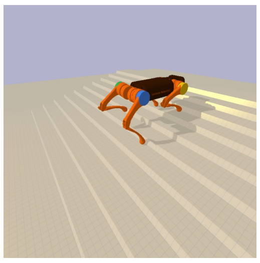
Evaluations

5.1 Evaluation Environments and Metrics

Environments: We evaluate the framework across four environments similar to those used by Kasaei et al. [21] and Shi et al. [48]; a sinusoidal terrain (Figure 5.1d) with a maximum incline of 0.2rad, stairs (Figure 5.1c) with a step height of 4cm, a pivoting table-top (Figure 5.1b) with maximum rotations around the pivot of 0.087rad, and a seesaw (Figure 5.1a) with an decline/incline of 0.1rad. We predefined a set of way-points for the robot to follow, such that the robot completes a course planned to test challenging situations, such as turning on the staircase (video available).

(a) Seesaw


(b) Tabletop


(c) Stairs (4cm)

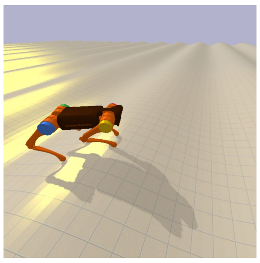

(d) Sinusoidal

Figure 5.1: The evaluation environments

Metrics: We measure the average reward per time-step, using the final reward function (Table 4.8) with the additional roll and pitch features; The success rate, defined as the proportion of complete runs (reaching all the targets), and the fall rate defined as the proportion of runs where the robot falls.

5.2 Results

Tables 5.1 to 5.4 show the performance of the MPC controller given different stance durations, the kernel acting alone, and the two versions of the trained agents in the evaluation environments. For the trained agents we take the mean and standard deviation across the policy trained using each seed 4.2.1, for the kernel, we only evaluate the kernel used to provide the reference motions to the agent. Furthermore, we note that each environment has 5 target locations to reach, using 4 different starting locations.

As shown, the agents successfully outperform the MPC controllers and kernel in all evaluation environments. Agent-rp performs best, only falling a single time out of 16 runs in the most challenging environments, stairs, and sin, significantly outperforming agent-base. We note that it achieves the most reward per timestep and is also the most versatile, showing locomotion skills that have generalized to the full range of terrains.

Models	**Reward/steps**	**Num Targets**	**Success Rate**	**Fall Rate**
MPC (τ_{stance} =0.3)	0.013±0.072	0.25±0.5	0	1
MPC (τ_{stance} =0.2)	0.016±0.11	2.5±2.89	0.5	0.5
Kernel	0.065±0.0085	2.25±1.5	0	1
Agent-base	0.091±0.0076	4.5±1.1	0.8125	0.1875
Agent-rp	0.097±0.001	5±0.0	1	0

Table 5.1: Tabletop

Models	**Reward/steps**	**Num Targets**	**Success Rate**	**Fall Rate**
MPC (τ_{stance} =0.3)	0.041±0.0722	0.0±0.0	0	1
MPC (τ_{stance} =0.2)	0.065±0.018	0.0±0.0	0	1
Kernel	0.043±0.00245	0.0±0.0	0	1
Agent-base	0.087±0.0024	5±0.0	1	0
Agent-rp	0.091±0.0006	5±0.0	1	0

Table 5.2: Seesaw

Models	**Reward/steps**	**Num Targets**	**Success Rate**	**Fall Rate**
MPC (τ_{stance} =0.3)	0.067±0.00098	0.0±0.0	0	1
MPC (τ_{stance} =0.2)	0.073±0.0022	0.0±0.0	0	1
Kernel	0.047±0.0019	0.0±0.0	0	0
Agent-base	0.078±0.0046	2.25±1.8	0.25	0.75
Agent-rp	0.089±0.0024	4.75±1.0	0.9375	0.0625

Table 5.3: Stairs

Models	**Reward/steps**	**Num Targets**	**Success Rate**	**Fall Rate**
MPC (τ_{stance} =0.3)	0.07±0.0078	0.5±1	0	1
MPC (τ_{stance} =0.2)	0.082±0.0057	3±1.83	0.25	0.75
Kernel	0.042±0.0021	0±0.0	0	0
Agent-base	0.081±0.006	3.87±1.4	0.5	0.5
Agent-rp	0.089±0.0026	4.75±1.0	0.9375	0.0625

Table 5.4: Sinusoidal

5.3 Evaluating Perturbation Robustness

The agent is tasked with walking to a single target location, during this task, a force is applied to a random point of the body, in a random direction, at a random point of time, for a duration of 0.3 seconds. We determine success by the robots ability to reach the target location. For each magnitude of force applied, we run 10 attempts and record the percentage of successfully completed tasks, as shown in Table 5.5.

We observe that the MPC controller falls at relatively low force magnitudes (300N), where even the kernel can recover balance. The MPC controller outperforms the kernel at higher forces, however agent-rp has shown to be the most robust, regularly being able to recover its balance after perturbations of 800N where the MPC controller can not.

	250	300	350	400	450	500	550	600	650	700	750	800	850	900
MPC	1	0.8	0.9	0.7	0.5	0.3	0.4	0.2	0.4	0.1	0.2	0	0	0
Kernel	1	1	1	0.8	0.5	0.3	0.3	0	0	0	0	0	0	0
Agent-rp	1	1	1	1	0.9	0.8	0.9	0.8	0.7	0.4	0.8	0.6	0.3	0.2

Table 5.5: Robustness against perturbations, using the MPC with (τ_{stance} =0.2)

Chapter 6

Conclusions

Core Findings: We have successfully learned a kernel that replicates the trajectories of an expert MPC controller. We demonstrate that we can produce dynamic trajectories, making the kernel more controllable and useful than typical methods using kMPs, dMPs, and other discriminative neural network methods that are not controllable. Furthermore, we can dynamically change the step and ride height during locomotion and use the kernel to produce gait patterns not provided during training, albeit poor at producing locomotion. Most importantly, we find that we only need minimal amounts of expert data in order to produce a functional kernel controller.

Using the kernel to provide reference motions and a residual agent adapting them, we successfully demonstrate the ability to navigate a range of highly challenging unseen terrains without falling, far exceeding the MPC controller used for training the kernel. Furthermore, our framework demonstrates superior robustness to external perturbations. All of which, with relatively little training. Of note, we found the agent performs best without knowledge of the reference motions and with residual feedback.

Limitations: The most notable limitation of the framework is the inability to learn a kernel that successfully captures the robot's dynamics. This limits our ability to learn a kernel when the control of the robot is unstable and produces trajectories with high variance. Therefore, we can only use the framework given that we have a highly stable controller. As such, implementation with bipedal locomotion may be more challenging.

The framework is limited by the velocity command caps introduced by the command generator. Nevertheless, higher velocities are likely unachievable given the framework's fixed gait stance duration. To achieve higher velocities, we would typically increase the stepping frequency. Finally, step height is the primary concern for the framework's versatility. The agent cannot take steps significantly greater than required in training and also possesses no capacity to determine when a greater step height may be necessary.

Future Research Directions: Although we do not expect capturing the robot's dynamics in the kernel to improve the framework's performance, it should be addressed first to ensure broader scope and applicability. We note that Mitchell et al. [34] and Surovik et al. [53] do this, however their frameworks lacks interpretable control. We hypothesize that we could expand upon the work from Surovik et al. [53]; Rather, using the commands from our framework as the conditioning set of the cVAE. This would also open up a new research question within residual learning; Could a residual agent interpolate the latent space rather than modifying positional trajectories?

We found that the residual agent could not correct the zero-shot walk gait trajectories produced by the kernel-ind. Although it could improve the performance, it was still sub-expert level and fell even on flat terrains. Due to the poor reference trajectories, it would be valuable to explore zero-shot trajectory prediction further, as this would allow us to rapidly learn a broader range of expert gaits with very little data.

The trained agent only produces locomotion for a single gait. However, training it on a more diverse range of gaits may allow it to generalize to unseen gaits. Furthermore, if this is the case, it may also be effective to allow the agent to choose the gait parameters, allowing for gait transitioning and the selection of more stable gaits under uncertain conditions. Finally, we believe the agent could be enhanced with a vision system as in the work of Gangapurwala et al. [11]. This could present an opportunity to utilize the kernel's ability to adjust the step and ride height, where the agent could select appropriate kernel step heights for traversing stairs with high steps or selecting the ride height allowing it to traverse low passes. Alternatively, the agent could lower its ride height after large perturbations to lower its center of mass and regain stability.

Bibliography

[1] Farzad Abdolhosseini, Hung Yu Ling, Zhaoming Xie, Xue Bin Peng, and Michiel van de Panne. On learning symmetric locomotion. In *Motion, Interaction and Games*, MIG '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369947. doi: 10.1145/3359566.3360070. URL https://doi.org/10.1145/3359566.3360070.

[2] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Process. Mag.*, 34(6):26–38, Nov 2017. doi: 10.1109/MSP.2017.2743240.

[3] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies - a comprehensive introduction. *Natural Computing*, 1:3–52, 03 2002. doi: 10.1023/A:1015059928466.

[4] J. Bhatti, A. R. Plummer, P. Iravani, and B. Ding. A survey of dynamic robot legged locomotion. In *2015 International Conference on Fluid Power and Mechatronics (FPM)*, pages 770–775, 2015. doi: 10.1109/FPM.2015.7337218.

[5] Dian Chen, Brady Zhou, Vladlen Koltun, and Philipp Krähenbühl. Learning by cheating, 2019. URL https://arxiv.org/abs/1912.12294.

[6] Jared Di Carlo, Patrick M Wensing, Benjamin Katz, Gerardo Bledt, and Sangbae Kim. Dynamic locomotion in the mit cheetah 3 through convex model-predictive control. In *2018 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 1–9. IEEE, 2018.

[7] Jared Di Carlo, Patrick M. Wensing, Benjamin Katz, Gerardo Bledt, and Sangbae Kim. Dynamic locomotion in the mit cheetah 3 through convex model-predictive control. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–9, 2018. doi: 10.1109/IROS.2018.8594448.

[8] Helei Duan, Jeremy Dao, Kevin Green, Taylor Apgar, Alan Fern, and Jonathan Hurst. Learning task space actions for bipedal locomotion. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1276–1282, 2021. doi: 10.1109/ICRA48506.2021.9561705.

[9] Alejandro Escontrela, Xue Bin Peng, Wenhao Yu, Tingnan Zhang, Atil Iscen, Ken Goldberg, and Pieter Abbeel. Adversarial motion priors make good substitutes for complex reward functions, 2022. URL https://arxiv.org/abs/2203.15103.

[10] Siddhant Gangapurwala, Mathieu Geisert, Romeo Orsolino, Maurice Fallon, and Ioannis Havoutis. Rloc: Terrain-aware legged locomotion using reinforcement learning and optimal control, 2020. URL https://arxiv.org/abs/2012.03094.

[11] Siddhant Gangapurwala, Mathieu Geisert, Romeo Orsolino, Maurice Fallon, and Ioannis Havoutis. Real-time trajectory adaptation for quadrupedal locomotion using deep reinforcement learning. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5973–5979, 2021. doi: 10.1109/ICRA48506.2021.9561639.

[12] Sten Grillner and Peter Wallén. Innate versus learned movements—a false dichotomy? In *Brain Mechanisms for the Integration of Posture and Movement*, volume 143 of *Progress in Brain Research*, pages 1–12. Elsevier, 2004. doi: https://doi.org/10.1016/S0079-6123(03)43001-X. URL https://www.sciencedirect.com/science/article/pii/S007961230343001X.

[13] Tuomas Haarnoja, Sehoon Ha, Aurick Zhou, Jie Tan, George Tucker, and Sergey Levine. Learning to walk via deep reinforcement learning, 2019.

[14] Nikolaus Hansen. The cma evolution strategy: A tutorial, 2016. URL https://arxiv.org/abs/1604.00772.

[15] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26), jan 2019. doi: 10.1126/scirobotics.aau5872. URL https://doi.org/10.1126%2Fscirobotics.aau5872.

[16] A.J. Ijspeert, J. Nakanishi, and S. Schaal. Movement imitation with nonlinear dynamical systems in humanoid robots. In *Proceedings 2002 IEEE International*

Conference on Robotics and Automation (Cat. No.02CH37292), volume 2, pages 1398–1403 vol.2, 2002. doi: 10.1109/ROBOT.2002.1014739.

[17] Auke Jan Ijspeert, Jun Nakanishi, and Stefan Schaal. Learning rhythmic movements by demonstration using nonlinear oscillators. In *Proceedings of the ieee/rsj int. conference on intelligent robots and systems (iros2002)*, number CONF, pages 958–963, 2002.

[18] Tobias Johannink, Shikhar Bahl, Ashvin Nair, Jianlan Luo, Avinash Kumar, Matthias Loskyll, Juan Aparicio Ojea, Eugen Solowjow, and Sergey Levine. Residual reinforcement learning for robot control, 2018. URL https://arxiv.org/abs/1812.03201.

[19] Ian Jolliffe. *Principal Component Analysis*, pages 1094–1096. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-04898-2. doi: 10.1007/978-3-642-04898-2_455. URL https://doi.org/10.1007/978-3-642-04898-2_455.

[20] Won Jungdam, Gopinath Deepak, and Hodgins Jessica. Physics-based character controllers using conditional vaes. *ACM Transactions on Graphics (SIGGRAPH 2022)*, 2022.

[21] Mohammadreza Kasaei, Miguel Abreu, Nuno Lau, Artur Pereira, and Luis Paulo Reis. A cpg-based agile and versatile locomotion framework using proximal symmetry loss, 2021. URL https://arxiv.org/abs/2103.00928.

[22] Benjamin Katz, Jared Di Carlo, and Sangbae Kim. Mini cheetah: A platform for pushing the limits of dynamic quadruped control. In *2019 international conference on robotics and automation (ICRA)*, pages 6295–6301. IEEE, 2019.

[23] Donghyun Kim, Jared Di Carlo, Benjamin Katz, Gerardo Bledt, and Sangbae Kim. Highly dynamic quadruped locomotion via whole-body impulse control and model predictive control. *arXiv preprint arXiv:1909.06586*, 2019.

[24] Yo Kondo and Yasutake Takahashi. Real-time whole body imitation by humanoic robot based on particle filter and dimension reduction by autoencoder. In *2017 Joint 17th World Congress of International Fuzzy Systems Association and 9th International Conference on Soft Computing and Intelligent Systems (IFSA-SCIS)*, pages 1–6. IEEE, 2017.

[25] Ivan Koryakovskiy, Manuel Kudruss, Heike Vallery, Robert Babuska, and Wouter Caarls. Model-plant mismatch compensation using reinforcement learning. *IEEE Robotics and Automation Letters*, 3(3):2471 – 2477, 2018. ISSN 2377-3766. doi: 10.1109/LRA.2018.2800106. Accepted Author Manuscript.

[26] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. Rma: Rapid motor adaptation for legged robots, 2021. URL https://arxiv.org/abs/2107.04034.

[27] Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning quadrupedal locomotion over challenging terrain. *Science Robotics*, 5(47):eabc5986, 2020. doi: 10.1126/scirobotics.abc5986. URL https://www.science.org/doi/abs/10.1126/scirobotics.abc5986.

[28] Anqiao Li, Zhicheng Wang, Jun Wu, and Qiuguo Zhu. Efficient learning of control policies for robust quadruped bounding using pretrained neural networks, 2020. URL https://arxiv.org/abs/2011.00446.

[29] Chenhao Li, Marin Vlastelica, Sebastian Blaes, Jonas Frey, Felix Grimminger, and Georg Martius. Learning agile skills via adversarial imitation of rough partial demonstrations, 2022. URL https://arxiv.org/abs/2206.11693.

[30] Tianyu Li, Jungdam Won, Sehoon Ha, and Akshara Rai. Fastmimic: Model-based motion imitation for agile, diverse and generalizable quadrupedal locomotion, 2021. URL https://arxiv.org/abs/2109.13362.

[31] Zhongyu Li, Xuxin Cheng, Xue Bin Peng, Pieter Abbeel, Sergey Levine, Glen Berseth, and Koushil Sreenath. Reinforcement learning for robust parameterized locomotion control of bipedal robots, 2021.

[32] Chengju Liu, Wandong Geng, Ming Liu, and Qijun Chen. Workspace trajectory generation method for humanoid adaptive walking with dynamic motion primitives. *IEEE Access*, 8:54652–54662, 2020. doi: 10.1109/ACCESS.2020.2976098.

[33] Gabriel B Margolis, Ge Yang, Kartik Paigwar, Tao Chen, and Pulkit Agrawal. Rapid locomotion via reinforcement learning, 2022. URL https://arxiv.org/abs/2205.02824.

[34] Alexander L. Mitchell, Wolfgang Merkt, Mathieu Geisert, Siddhant Gangapurwala, Martin Engelcke, Oiwi Parker Jones, Ioannis Havoutis, and Ingmar Posner.

Vae-loco: Versatile quadruped locomotion by learning a disentangled gait representation, 2022. URL https://arxiv.org/abs/2205.01179.

[35] Federico Moro, Nikos Tsagarakis, and Darwin Caldwell. On the kinematic motion primitives (kmps) - theory and application. *Frontiers in Neurorobotics*, 6, 2012. ISSN 1662-5218. doi: 10.3389/fnbot.2012.00010. URL https://www.frontiersin.org/articles/10.3389/fnbot.2012.00010.

[36] Federico L. Moro, Nikos G. Tsagarakis, and Darwin G. Caldwell. A human-like walking for the compliant humanoid coman based on com trajectory reconstruction from kinematic motion primitives. In *2011 11th IEEE-RAS International Conference on Humanoid Robots*, pages 364–370, 2011. doi: 10.1109/Humanoids.2011.6100862.

[37] Xue Bin Peng, Erwin Coumans, Tingnan Zhang, Tsang-Wei Edward Lee, Jie Tan, and Sergey Levine. Learning agile robotic locomotion skills by imitating animals. In *Robotics: Science and Systems*, 07 2020. doi: 10.15607/RSS.2020.XVI.064.

[38] Affan Pervez and Dongheui Lee. Learning task-parameterized dynamic movement primitives using mixture of gmms. *Intelligent Service Robotics*, 11(1):61–78, 2018.

[39] Affan Pervez, Yuecheng Mao, and Dongheui Lee. Learning deep movement primitives using convolutional neural networks. In *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pages 191–197, 2017. doi: 10.1109/HUMANOIDS.2017.8246874.

[40] Martin L. Puterman. Chapter 8 Markov decision processes. In *Handbooks in Operations Research and Management Science*, volume 2, pages 331–434. Elsevier, Walthm, MA, USA, Jan 1990. doi: 10.1016/S0927-0507(05)80172-0.

[41] Marc H Raibert. *Legged robots that balance*. MIT press, 1986.

[42] José Rosado, Filipe Silva, and Vítor Santos. Adaptation of robot locomotion patterns with dynamic movement primitives. In *2015 IEEE International Conference on Autonomous Robot Systems and Competitions*, pages 23–28, 2015. doi: 10.1109/ICARSC.2015.9.

[43] Matteo Saveriano, Fares J. Abu-Dakka, Aljaz Kramberger, and Luka Peternel. Dynamic movement primitives in robotics: A tutorial survey, 2021. URL https://arxiv.org/abs/2102.03861.

[44] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2015. URL https://arxiv.org/abs/1502.05477.

[45] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015. URL https://arxiv.org/abs/1506.02438.

[46] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

[47] Yecheng Shao, Yongbin Jin, Xianwei Liu, Weiyan He, Hongtao Wang, and Wei Yang. Learning free gait transition for quadruped robots via phase-guided controller. *IEEE Robotics and Automation Letters*, 7(2):1230–1237, 2022. doi: 10.1109/LRA.2021.3136645.

[48] Haojie Shi, Bo Zhou, Hongsheng Zeng, Fan Wang, Yueqiang Dong, Jiangyong Li, Kang Wang, Hao Tian, and Max Q. H. Meng. Reinforcement learning with evolutionary trajectory generator: A general approach for quadrupedal locomotion, 2021. URL https://arxiv.org/abs/2109.06409.

[49] Manuel F. Silva and J.A. Tenreiro Machado. A historical perspective of legged robots. *Journal of Vibration and Control*, 13(9-10):1447–1486, 2007. doi: 10.1177/1077546307078276. URL https://doi.org/10.1177/1077546307078276.

[50] Abhik Singla, Shounak Bhattacharya, Dhaivat Dholakiya, Shalabh Bhatnagar, Ashitava Ghosal, Bharadwaj Amrutur, and Shishir Kolathaya. Realizing learned quadruped locomotion behaviors through kinematic motion primitives, 2018. URL https://arxiv.org/abs/1810.03842.

[51] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. Learning structured output representation using deep conditional generative models. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing*

Systems, volume 28. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper/2015/file/8d55a249e6baa5c06772297520da2051-Paper.pdf.

[52] Alexander Sprowitz, Mostafa ajallooeian, Alexandre Tuleu, and Auke Ijspeert. Kinematic primitives for walking and trotting gaits of a quadruped robot with compliant legs. *Frontiers in Computational Neuroscience*, 8, 2014. ISSN 1662-5188. doi: 10.3389/fncom.2014.00027. URL https://www.frontiersin.org/articles/10.3389/fncom.2014.00027.

[53] David Surovik, Oliwier Melon, Mathieu Geisert, Maurice Fallon, and Ioannis Havoutis. Learning an expert skill-space for replanning dynamic quadruped locomotion over obstacles. In Jens Kober, Fabio Ramos, and Claire Tomlin, editors, *Proceedings of the 2020 Conference on Robot Learning*, volume 155 of *Proceedings of Machine Learning Research*, pages 1509–1518. PMLR, 16–18 Nov 2021. URL https://proceedings.mlr.press/v155/surovik21a.html.

[54] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[55] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots, 2018. URL https://arxiv.org/abs/1804.10332.

[56] Mathias Thor, Tomas Kulvicius, and Poramate Manoonpong. Generic neural locomotion control framework for legged robots. *IEEE Transactions on Neural Networks and Learning Systems*, 32(9):4013–4025, 2021. doi: 10.1109/TNNLS.2020.3016523.

[57] Sashank Tirumala, Sagar Gubbi, Kartik Paigwar, Aditya Sagi, Ashish Joglekar, Shalabh Bhatnagar, Ashitava Ghosal, Bharadwaj Amrutur, and Shishir Kolathaya. Learning stable manoeuvres in quadruped robots from expert demonstrations, 2020. URL https://arxiv.org/abs/2007.14290.

[58] Vassilios Tsounis, Mitja Alge, Joonho Lee, Farbod Farshidian, and Marco Hutter. Deepgait: Planning and control of quadrupedal gaits using deep reinforcement learning. *IEEE Robotics and Automation Letters*, 5(2):3699–3706, 2020. doi: 10.1109/LRA.2020.2979660.

[59] Caiwang Wang, Guangming Xie, Xinyan Yin, Liang Li, and Long Wang. Cpg-based locomotion control of a quadruped amphibious robot. In *2012 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, pages 1–6, 2012. doi: 10.1109/AIM.2012.6265897.

[60] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3): 279–292, 1992.

[61] Zhaoming Xie, Glen Berseth, Patrick Clary, Jonathan Hurst, and Michiel van de Panne. Feedback control for cassie with deep reinforcement learning, 2018. URL https://arxiv.org/abs/1803.05580.

[62] Zhaoming Xie, Patrick Clary, Jeremy Dao, Pedro Morais, Jonanthan Hurst, and Michiel van de Panne. Learning locomotion skills for cassie: Iterative design and sim-to-real. In Leslie Pack Kaelbling, Danica Kragic, and Komei Sugiura, editors, *Proceedings of the Conference on Robot Learning*, volume 100 of *Proceedings of Machine Learning Research*, pages 317–329. PMLR, 30 Oct–01 Nov 2020. URL https://proceedings.mlr.press/v100/xie20a.html.

[63] Hiroki Yamamoto, Sungi Kim, Yuichiro Ishii, and Yusuke Ikemoto. Generalization of movements in quadruped robot locomotion by learning specialized motion data. *ROBOMECH Journal*, 7(1):1–14, 2020.

[64] Chuanyu Yang, Kai Yuan, Qiuguo Zhu, Wanming Yu, and Zhibin Li. Multi-expert learning of adaptive legged locomotion. *Science Robotics*, 5(49), dec 2020. doi: 10.1126/scirobotics.abb2174. URL https://doi.org/10.1126%2Fscirobotics.abb2174.

[65] Yuxiang Yang, Tingnan Zhang, Erwin Coumans, Jie Tan, and Byron Boots. Fast and efficient locomotion via learned gait transitions. In *Conference on Robot Learning*, pages 773–783. PMLR, 2022.

[66] Chen Yu and Andre Rosendo. Multi-modal legged locomotion framework with automated residual reinforcement learning, 2022. URL https://arxiv.org/abs/2202.12033.

[67] Wenhao Yu, Greg Turk, and C. Karen Liu. Learning symmetric and low-energy locomotion. *ACM Transactions on Graphics*, 37(4):1–12, aug 2018. doi: 10.1145/3197517.3201397. URL https://doi.org/10.1145%2F3197517.3201397.

[68] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics, 2019. URL https://arxiv.org/abs/1903.11239.

Appendix A

Appendix

A.1 Notation

Variable	Dimensions	Description
v_{base}	3	Velocity the base of the robot (x,y,z)
a_{base}	3	Angular velocity the base of the robot (roll, pitch, yaw)
v_{cmd}	2	Target velocity of the base of the robot (x,y)
$a_{/cmd}$	1	Target angular velocity of the base of the robot (yaw)
q	12	Angle of motor joints
$\dot{q}$	12	Angular velocity of motor joints
$\ddot{q}$	12	Angular acceleration of motor joints
$\phi_{1:4}$	4	Phase of each leg (0,1]
CoM	3	The center mass of the robot
$p_{1:4}$	12	Position of each foot link in the base frame (x,y,z)
$p_{1:4}^{ref}$	12	Target position of each foot link in the base frame (x,y,z)
$p'_{1:4}$	12	Resulting foot link positions after applying the motor torques (x,y,z)

Table A.1: Definitions of the variables collected

A.2 Optimizations and Search Spaces

Hyperparameter	Type	Values	Selected
Learning Rate(α)	Range	$(1e-5, 1e-2)$	0.0024
α_{decay}	Categorical	$[0.7,\ 0.8,\ 0.9]$	0.7
Dropout	Range	$(0, 0.3)$	0.00022
Batch size	Categorical	$[100, 200, 500, 1000, 2000]$	200
Activation	Categorical	$[tanh,\ relu]$	$relu$
Batch norm	Categorical	$[True, False]$	$False$
Network size	Categorical	$[(2,128), (3,128), (256,3), (256,4)]$	$(256,4)$
Optimizer	Fixed	Adam	Adam

Table A.2: Kernel-base hyperparameter optimization values and final selected hyperparameters

Hyper-parameter	Parameter type	Parameter Space	Selected
learning rate (lr)	Categorical	$[3e-3, 1e-3, 5e-4, 5e-5]$	$1e-3$
entropy coefficient	Range	$[1e-6, 1e-3]$	$5e-6$
Network size	Categorical	[large, medium, small]	small
Number of epochs	Categorical	[5,10,20]	10
Batch size	Categorical	[500,2000,4000]	4000
Rollout length	Fixed	20000	20000
lr decay	Fixed	$1e-7$	$1e-7$

Table A.3: PPO hyper-parameters search space, and selected parameters

Network sizes: We configure the PPO algorithm to share weight between the actor and critic as shown in A.1. We define each variant of the network (large, medium and small) to have a actor with one hidden layers of size 128 with 12 outputs nodes and a critic with one hidden layers of size 64 with one output node. We adjust the size of the network of the shared parameters for each variant; The large network has three hidden layers of size 256, the medium network has two hidden layers of size 256, and the small network has two hidden layers of size 128.

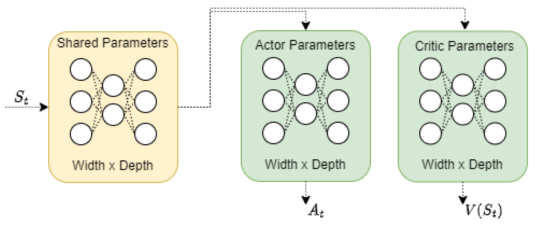


Figure A.1

Reward Feature	Type	Values	Selected
Linear velocity	range	[1,4]	18.42
Angular velocity	range	[1,4]	7.47
Center of mass	range	[15,20]	2.35
Distance to target	range	[5,10]	0.74

Table A.4: Reward function RBF parameter optimization space

A.3 Additional Figures

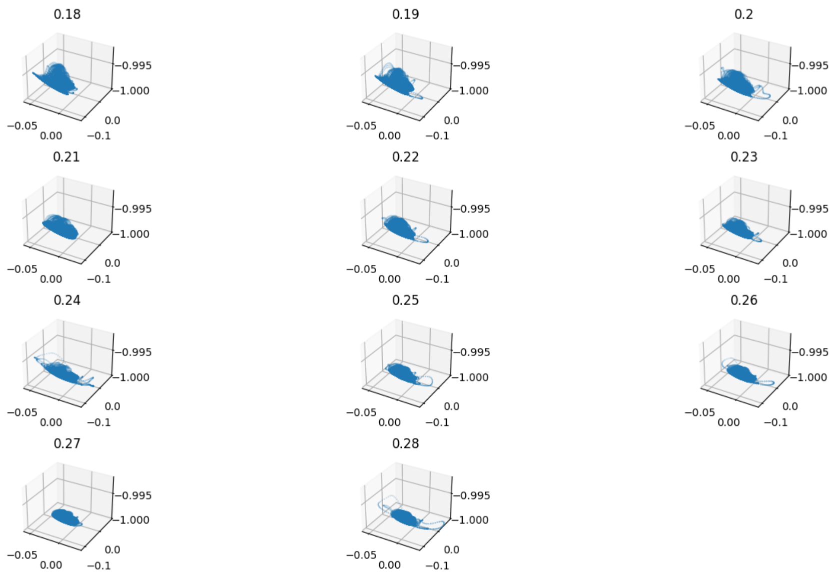


Figure A.2: Center of Mass of the robot using the MPC controller ($\tau_{stance} = 0.2$) varying the ride height

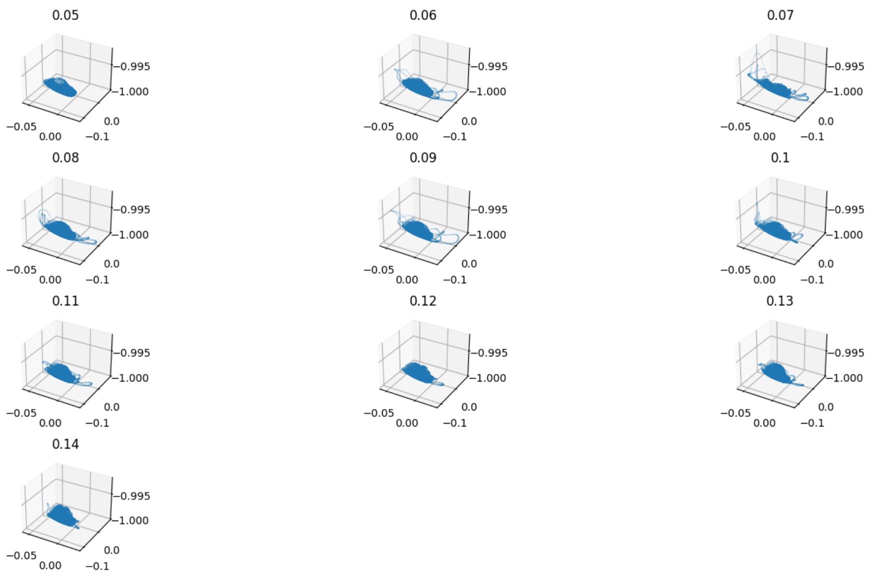


Figure A.3: Center of Mass of the robot using the MPC controller ($\tau_{stance} = 0.2$) varying the step height

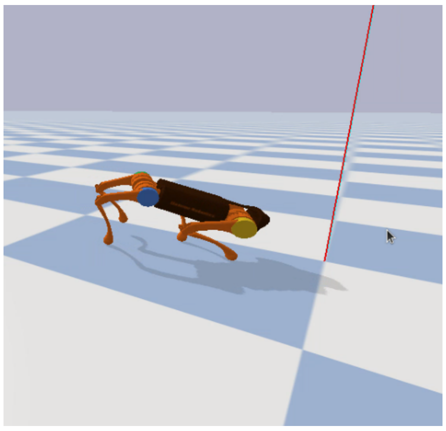


Figure A.4: Showing the robot leaning over its front legs as it approaches the target location.

A.4 Algorithms

Algorithm 2 Command Generator

```
1:  procedure COMMAND GENERATION PROCEDURE
2:      Inputs: - Target location $pos_{target}$ - Maximum Velocity Magnitude $vel_{max}$ - Velocity clip range $vel_{range}$ - Target threshold $dist_{min}$
3:
4:      Initialize a velocity command vector $vel_{cmd}$ as all zeros contaning the velocities:
5:      • frontal $vel_{frontal}$,
6:      • lateral $vel_{lateral}$
7:      • yaw $vel_{yaw}$
8:
9:      receive the robot location $pos_{base}$ and orientation $orn_{base}$
10:     while $dist(pos_{target}, pos_{base}) < dist_{min}$ do
11:         $vel_{cmd}$ = GENERATE COMMANDS($vel_{cmd}$, $vel_{max}$, $vel_{range}$)
12:         apply $vel_{cmd}$ to the robot
13:         receive the robot location $pos_{base}$ and orientation $orn_{base}$
14:     end while
15: end procedure

16: function GENERATE COMMANDS($pos_{target}$,$pos_{base}$,$orn_{base}$,$vel_{cmd}$,$vel_{max}$, $vel_{range}$)
17:     D = dist($pos_{target}$,$pos_{base}$)
18:     θ = DEG(arctan2($pos_{target}$,$pos_{base}$)-$orn_{base}$)
19:     $cmd$ = [D*cos( θ), D*sin( θ), 0.3*θ ]
20:     $cmd$ = $vel_{cmd}$ + clip($cmd$-$vel_{cmd}$, -0.005, 0.005)
21:     if $|cmd| > vel_{max}$ then
22:         $cmd$ = norm($cmd$)*$vel_{max}$
23:     end if
24:     $cmd$ = clip($cmd$, $vel_{range}$)
25:     return $cmd$
26: end function
```