A Implementation Details for Motion Controller

A.1 Choice of Inertial Frame

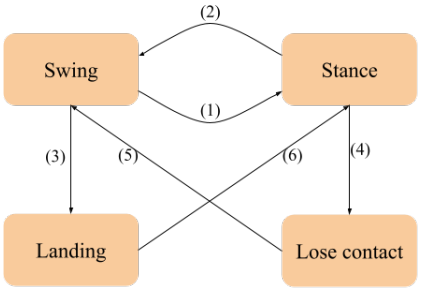
Unless otherwise mentioned, all quantities are specified in an inertial frame with a positive upward gravity-aligned z-axis and a x-axis which is instantaneously aligned with the robot's heading direction at every time step. Also, the origin of this inertial frame always sits at the robot's center of mass (CoM). Within this representation, the yaw angle of the robot is 0 at any given moment; Thus this design eliminates the need to track the yaw angle of the robot in the ground attached world frame without SLAM. Compared with a purely body attached frame that rotates with the robot, the *heading aligned inertial frame* makes dynamics computation and estimations easier.

A.2 Gait Scheduler

The locomotion gait refers to the pattern of limb movements during locomotion. In this work, we classify the movement of each robot limb into one of two categories: swing leg, where the foot of the leg is moving in the air to a target location, and stance leg, where the foot of the leg remains in contact with the ground. By modulating the timings where each leg switches between swing and stance, we can create different locomotion gaits for the quadruped robot. In this work, we use three types of locomotion gaits: trotting (diagonal legs in swing or stance synchronously), pacing (legs on the same side in swing or stance synchronously), and walking (one swing leg at a time).

We design a gait scheduler to output the desired leg state at each timestep, which then determines whether a leg is to be controlled by the swing leg controller, or the MPC-based stance leg controller. For each leg, the gait scheduler employs a finite state machines (FSM) driven by time and robot states, see Figure 9. Given an initial gait phase and the gait parameters (stance duration and swing duration), the FSM will output the current desired leg state. Different locomotion gaits are then realized by specifying different initial gait phases and gait parameters for different legs. Figure 10 illustrates the leg states over time for three types of locomotion gaits.

Note that our FSM contains two additional states other than swing and stance: landing and lose contact. The first is to handle cases where the swing leg finishes the entire swing trajectory but has not established a contact with ground. In this case, we will control the swing leg to move downward at a constant speed until it reaches the ground. During this period, we will also freeze the gait phase increment on the other legs such that the relative phase between different legs are kept the same. The second additional state, lose contact, is to handle when the stance leg loses the contact due to unexpected situations such as moving stepstones, etc. In this case, we send zero torque to the leg until it is ready to begin the next swing motion.



(1) If foot establishes contact with the ground
(2) If gait phase > stance duration
(3) If gait phase > swing duration and foot not in contact with the ground
(4) If foot loses contact with the ground
(5) If gait phase > stance duration
(6) If foot establishes contact with the ground

Figure 9: Finite state machine used to specify the state for one leg (Left). The conditions used for each state transition (Right).

A.3 Swing Leg Control

Given the lift-off and the target landing location of the swing leg in the heading-aligned inertial frame, our swing leg controller computes an interpolation of the foot position throughout the swing process, and converts it into the target joint angles for the leg. Specifically, we represent the target foot trajectory using five connected line segments, as shown in Figure 11: the swing foot will first lift up by 60% of the swing clearance H and move back by 5% of the step length L, then move

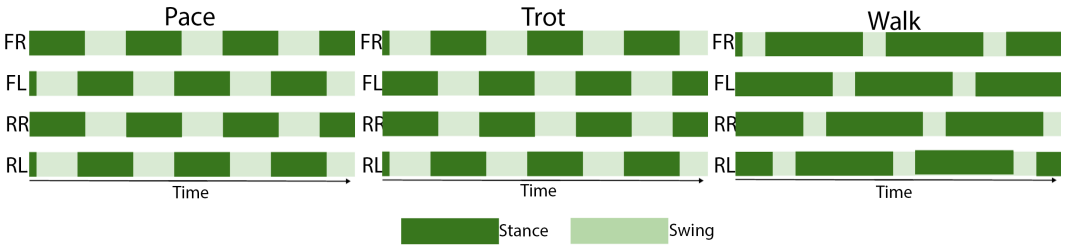


Figure 10: Diagram of the three gaits used in our experiments.

forward by $15\%L$ while reaching H in z direction. The target foot position will then move forward until $90\%L$ at the same height, followed by a landing trajectory mirroring the lifting one. The shape of the swing trajectory is designed such that the foot is less likely to hit obstacles during the swing process. After obtaining the target foot position, we map it back to the joint angles using Inverse Kinematics (IK) and then track the desired joint angles using a proportional-derivative (PD) controller.

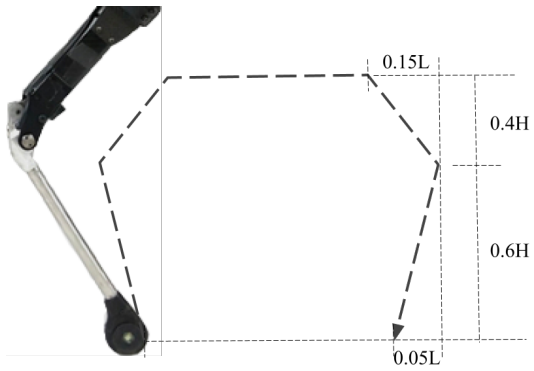


Figure 11: Swing foot target interpolation between the lift-off and landing position.

A.4 Stance Leg Control

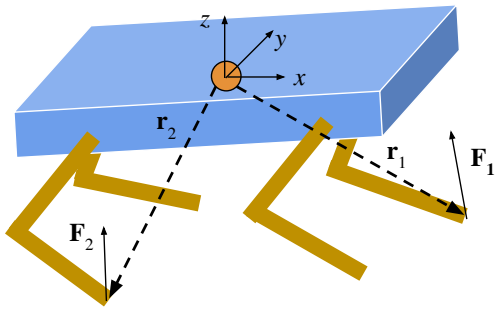


Figure 12: The centroidal dynamics model used to formulate the model predictive control problem.

We employee a model predictive control (MPC) based method in our stance leg controller. Using centroidal dynamics [9], the MPC algorithm calculates the feet contact force on each stance foot to track the desired center of mass (CoM) trajecotry obtained from the high level policy. In the centroidal dynamic model, we treat the whole robot as a single rigid body, and assume that the inertia contribution from the leg movements is negligible (Figure 12). This is a reasonable assumption since Laikago's legs are thin and have low masses. With these simplifications, the system dynamics can

be simply written using the Newton-Euler equations:

$$m\ddot{\mathbf{q}} = \sum_{i}^{4} \mathbf{F}_i - \mathbf{g}, \tag{2}$$
$$\frac{d}{dt}(\mathbf{I}\omega) = \sum_{i}^{4} \mathbf{r}_i \times \mathbf{F}_i,$$

where m is the whole body mass, $\mathbf{q} = (p_x, p_y, p_z, \Phi, \Theta, \Psi)$ denotes the CoM translation and rotation, $\mathbf{F}_i = (f_x, f_y, f_z)_i$ denotes the contact force applied on the i-th foot (set to zero if the i-th foot is not in contact with the ground), and $\mathbf{r}_i$ is the displacement from the CoM to the contact point. We follow the same ZYX Euler angle conventions in [9] to represent the CoM rotation.

The MPC algorithm solves an optimization problem over a finite horizon H to track a desired pose and velocity $\mathbf{x} = (\mathbf{q}, \dot{\mathbf{q}})$. Given a desired state $\mathbf{x}^d$, the system dynamics can be further simplified by globally linearized and discretized in time with an explicit Euler formulation:

$$\mathbf{x}_{t+1} = A(\mathbf{x}^d)\mathbf{x}_t + B(\mathbf{x}^d)\mathbf{u}_t,$$

where $\mathbf{u}_t = (\mathbf{F}_{1,t}, \mathbf{F}_{2,t}, \mathbf{F}_{3,t}, \mathbf{F}_{4,t})$, is the concatenated force vectors from all feet. The matrices A and B are derived from Eq (3) and already absorb the time discretization dt. We formally write the optimization target as a quadratic problem (QP):

$$\begin{aligned} \min_{\mathbf{u}_t} \quad & \sum_{t=1}^{H} [(\mathbf{x}_t - \mathbf{x}_t^d)^T \mathbf{Q} (\mathbf{x}_t - \mathbf{x}_t^d) + \mathbf{u}_t^T \mathbf{R} \mathbf{u}_t], \\ \text{s.t.} \quad & \mathbf{x}_{t+1} = A\mathbf{x}_t + B\mathbf{u}_t, \\ & \mathbf{F}_{i,t} = 0 \text{ if } i\text{-th foot is not in contact}, \\ & 0 \le f_{z,t} \le f_{max}, \quad \text{contact normal force for each foot}, \\ & -\mu f_{z,t} \le f_{x,t} \le \mu f_{z,t}, \\ & -\mu f_{z,t} \le f_{y,t} \le \mu f_{z,t}, \quad \text{friction cone}, \end{aligned} \tag{3}$$

where we used a diagonal $\mathbf{Q}$ and $\mathbf{R}$ matrix with the weights detailed in [9]. At each control step, we solve this QP to obtain a sequence of optimal contact forces over the prediction horizon. The we take the first predicted contact force $F_{i,t=1}$, and convert them to motor torques using the Jacobian matrix.

B Action Space Adjustments

To help the learning process, we designed two critical features to augment the output of the NN policy before feeding them to the low level pose controller. The first is what we call the "inverse Raibert control" that applies to the desired CoM velocity (v_x, v_y) in the horizontal plane. In the regular Raibert formulation [35], the foothold in x, y can be uniquely determined by desired and actual CoM velocity (strictly speaking, for multi-legged robots one should use the hip velocity of each leg):

$$r^d_{x,y} = \frac{\dot{p}_{x,y} T_s}{2} + k_p(\dot{p}_{x,y} - \dot{p}^d_{x,y}), \tag{4}$$

where T_s is the swing duration, k_p is the gain. However, if foothold is directly controlled, i.e. as an output from the visual policy in our case, then in principle, we can invert this formula and find the compatible unique hip velocity for a given foot step size. There comes the inverse Raibert equation:

$$\dot{p}^d_{x,y} = \dot{p}_{x,y} + K_p(\frac{2r^d_{x,y}}{T_s} - \dot{p}_{x,y}) \tag{5}$$

Where K_p is positive vector gain. The desired CoM speed is then the average of the desired hip velocities for all swing legs. The intuition behind this formulation is to stabilize the CoM speed around the neutral velocity point $\frac{2r^d_{x,y}}{T_s}$, for given target swing footholds $r^d_{x,y}$. On top of the inverse

Raibert formulation, our visual policy also outputs a delta adjustment $\delta\dot{p}_{x,y}$, so the final desired COM speed passed to the motion controller is $\dot{p}^d + \delta\dot{p}$.

The second feature is pitch adaptation, which is especially useful to train stair climbing tasks. The idea is simple: in order to climb up or down steps, the robot will need to tilt according to the slope of the terrain, i.e. the "neutral tiling angle". The quadruped robot can sense this angle by computing the differences of the leg extension height (with respect to the hip joint) between the front and rear stance legs, δh. If $\delta h < 0$, the robot is moving up a slope and still needs to tilt further up, and vice versa. The desired pitch angle for the low-level motion controller is then:

$$\Theta^d = \Theta^s + \arctan\frac{\delta h}{L} + \delta\Theta, \tag{6}$$

where Θ^s is the currently measured pitch angle, L the horizontal distance between the landing positions of front and rear feet, and $\delta\Theta$ is the output from the visual policy.

C State Estimator

In order to plan the robot motion into the future, we need to use the current pose of the robot base, including its CoM height, CoM velocity, base roll and pitch, and angular velocities. Base orientation and angular velocities can be directly measured from an onboard IMU, while CoM height and velocity are estimated without motion capture systems. To estimate the CoM height, we compute average height of stance legs in the heading-aligned inertial frame.

To estimate the CoM velocity of the robot, we fuse the on-board IMU sensors and the stance feet velocities with a Kalman Filter. In a Kalman Filter, the true state to be estimated $\mathbf{x}$ is assumed to follow a linear dynamics: $\mathbf{x}' = F\mathbf{x} + B\mathbf{u} + \mathbf{w}$, where $\mathbf{x}'$ and $\mathbf{x}$ denote the current and previous state of the robot, F is the state transition matrix, B is the control-input matrix, and w is a noise sampled from $\mathcal{N}(0, Q)$. In addition, it also asusmes that we can only obtain a noisy observation of the true state from $\mathbf{z} = H\mathbf{x} + \mathbf{x}$, where $\mathbf{z}$ is the observed state, H is the observation matrix, and $\mathbf{v}$ is the observation noise sampled from $\mathcal{N}(0, R)$. In this particular case, $\mathbf{x}$ represents the estimated CoM velocity, $F = H = I$ are identity matrices, $B = \Delta t I$, $\mathbf{u}$ is the measured CoM acceleration from the accelerometer, and the noisy observation $\mathbf{z}$ is computed from the negative average velocities of the stance feet.

D Domain Randomization

We perform Domain Randomization (DR) to improve the robustness of our trained policy. Table 1 lists all the parameters and the ranges we used during training.

Table 1

Parameter	Minimal value	Maximum value
Mass	90%	110%
Inertial	90%	110%
Ground Friction	0.45	0.55
Motor Position Gain	200	220
Motor Velocity Gain	3.6	4.8

E Training Details

In our work, we use a large-scale implementation of the Augmented Random Search (ARS) algorithm [29] to train our vision policies. During each learning iteration, ARS samples N perturbations to the current policy parameter θ from a multivariate normal distribution $\mathcal{N}(0, \sigma I)$, where σ is the standard deviation of the parameter-space exploration, and then use the best performing $k\%$ of the perturbation directions to estimate the policy gradient $\frac{\partial J}{\partial \theta}$. The policy parameter is then updated using gradient descent, i.e. $\theta_{i+1} = \theta_i + \delta\frac{\partial J}{\partial \theta}$, where δ is the learning rate.

In this work, we use $N = 256$ perturbations per ARS iteration and run the algorithm until convergence with a maximum of 2000 training iterations. For each experiment, we perform a hyperparameter search on $\sigma \in \{0.005, 0.01, 0.025\}$, $k \in \{50, 100\}$, and $\delta \in \{0.005, 0.015\}$, resulting in 12 training trials per experiment and $1,024,000$ simulation episodes per trial.

Our policy is executed on a Zotac Magnus EN2070 mobile PC, which is 0.2x0.06x0.21m in size and weighs around 3kg. During our experiments, we placed the PC on the safety rack. However, it is possible to mount the PC on the robot given the weight and size.

E.1 Vision Policy Architecture

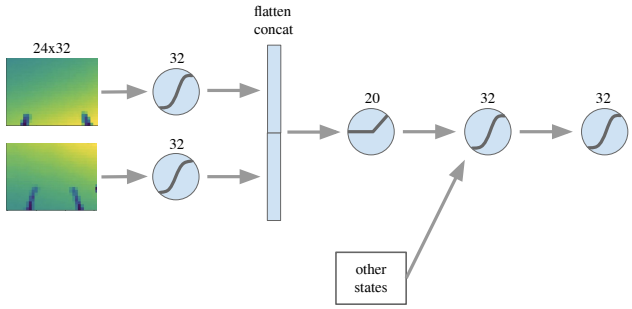


Figure 13: The network architecture for the vision policy. Each circle represents a fully connected layer with activation function drawn inside.

The vision inputs from both cameras are processed using MLP encoders (no weight sharing), which are then concatenated with the proprioception data from the robot. This is then fed into an MLP to produce the high-level action (Figure E.1). Therefore, our vision policy is fully MLP-based. Learning an MLP for processing vision input is feasible potentially due to the relatively low resolution images we use in our experiments. We have also experimented with other network architectures such as CNN and found them perform similarly in the simulation tasks. It is likely that for tasks that require a larger dimensional input the advantage of CNN-like architecture would show more advantage.