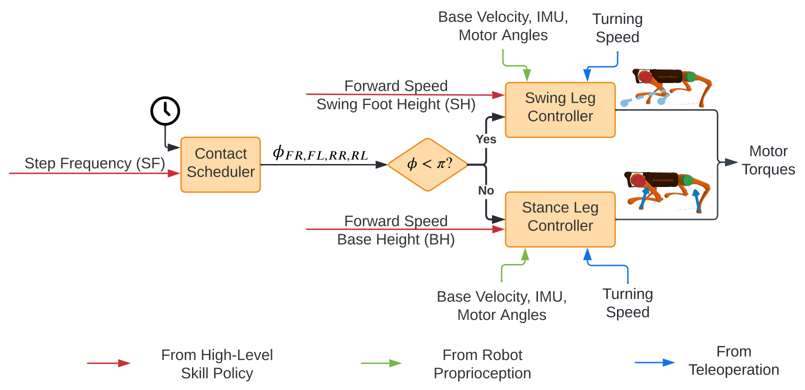


Figure 7: Overview of our low-level convex MPC controller. We use separate control strategies for swing and stance legs, and use a phase-based contact scheduler to determine the contact state of each leg.

A Details of the Convex MPC Controller

Our low-level convex MPC controller is similar to previous works [28, 12], which includes a contact scheduler to determine the contact state of each foot (*swing* or *stance*), and separate controllers for swing and stance legs. In addition, we implement a slope detector to react to changes in ground level and an impedance controller to prevent excessive foot slipping.

A.1 Phase-based Contact Scheduler

The contact scheduler determines the contact state of each leg (swing or stance). Similar to previous work by Yang et al. [12], we adopt a phase-based representation for contact schedule. More specifically, we introduce four phase variables $\phi_{FR,FL,RR,RL} \in [0, 2\pi)$, one for each leg. The subscripts denote the identity of each leg (Front-Right, Front-Left, Rear-Right, Rear-Left). The phase of each leg denotes its progress in the current locomotion cycle. As a leg moves, its phase ϕ increases monotonically from 0 to 2π, and wraps back to 0 as the next locomotion cycle starts.

At each control step, we determine the leg phases using the following procedure. First, we progress the phase of the front-right leg based on the stepping frequency (SF), f, which is obtained from the gait selector (Section 5):

$$\phi_{FL} \leftarrow \phi_{FL} + 2\pi f \Delta t \tag{1}$$

where Δt is the control timestep (0.0025s). We then determine the remaining leg phases according to the trotting pattern, where diagonal legs move together, and 180° out-of-phase with the other diagonal:

$$\begin{aligned} \phi_{FL} &= \phi_{FR} + \pi \\ \phi_{RR} &= \phi_{FR} + \pi \\ \phi_{RL} &= \phi_{FR} \end{aligned} \tag{2}$$

Depending on the phase of each leg, its motor command is computed by either the swing controller ($\phi < \pi$) or the stance controller ($\phi \geq \pi$).

A.2 Swing and Stance Controller

We use separate control strategies for swing and stance legs, where the swing controller uses a PD control to track a desired swing foot trajectory, and the stance controller solves a model predictive control (MPC) problem to optimize for foot forces. Our controller design is based on the work by Di Carlo et al. [28] with modifications. We briefly summarize them here. Please refer to the original work [28] for further details.

Swing Controller The goal of the swing controller is for the swing legs to track the desired swing trajectory. At each control step, the swing controller computes the swing trajectory by interpolating between three key positions, namely, the lift-off position $\boldsymbol{p}_{\text{lift-off}}$, the highest position in the air $\boldsymbol{p}_{\text{air}}$, and the landing position $\boldsymbol{p}_{\text{land}}$. $\boldsymbol{p}_{\text{lift-off}}$ is the recorded lift-off position at the beginning of the swing phase. $\boldsymbol{p}_{\text{air}}$ is the foot's highest position in the swing phase, where the height is determined by the swing height (SH) from the gait selector (Section 5). $\boldsymbol{p}_{\text{land}}$ is the estimated landing position computed by the Raibert Heuristic. Once the desired swing trajectory is computed, the controller computes the leg's desired position based on its progress in the current swing state. The controller then converts this desired position into joint angles using inverse kinematics, and uses a joint PD controller to track the desired position.

Stance Controller To find desired foot forces, the stance controller solves an MPC problem, where the objective is for the base to track the desired trajectory. The trajectory is generated by numerically integrating the desired forward speed and steering speed, where the forward speed is provided by the speed policy (Section 4) and the steering speed is provided by an external teleoperator. In addition, the height of the robot in this trajectory is determined by the base height (BH), which is provided by the gait selector (Section 5). To solve this MPC problem efficiently, we cast it as a quadratic program (QP), where the objective is a quadratic trajectory tracking loss, and the constraints include the linearized robot dynamics, actuator limits and approximated friction cones. Once the foot force is solved, we convert it into joint torque commands using jacobian transpose. Note that the final joint torque command is the sum of this torque computed by MPC and an additional torque from position feedback. Please see Appendix A.4 for details.

A.3 Uneven Terrain Detection and Adaptation

To operate in complex offroad terrains, it is crucial for the robot to adapt its base pose and foot swing range based on terrain shape. Therefore, we implement an uneven terrain detector that estimates terrain orientation from foot contact information and use it to adjust the robot's pose on steep slopes.

Ground Orientation Estimation The orientation of the ground plane is estimated based on foot contact position and imu readings, similar to previous work by Gehring et al. [31]. To find the ground orientation, we first find the ground normal vector in the robot frame, $\boldsymbol{n}_{\text{robot}}$, from foot contact positions. More specifically, we keep track of the last contact position of each leg, $\boldsymbol{p}_{1,\dots,4}$, which is represented in *robot frame*. Assuming that the contact positions lie in the same ground plane, the relationship between the contact positions $\boldsymbol{p}_{1,\dots,4}$ and the ground normal vector $\boldsymbol{n}_{\text{robot}}$ can be represented by:

$$\boldsymbol{n}_{\text{robot}}^T\boldsymbol{p}_1 = \boldsymbol{n}_{\text{robot}}^T\boldsymbol{p}_2 = \boldsymbol{n}_{\text{robot}}^T\boldsymbol{p}_3 = \boldsymbol{n}_{\text{robot}}^T\boldsymbol{p}_4 \tag{3}$$

We find $\boldsymbol{n}_{\text{robot}}$ using the following least-squares formulation:

$$\min_{\boldsymbol{n}_{\text{robot}}} \left\| \begin{bmatrix} | & | & | & | \\ \boldsymbol{p}_1 & \boldsymbol{p}_2 & \boldsymbol{p}_3 & \boldsymbol{p}_4 \\ | & | & | & | \end{bmatrix}^T \boldsymbol{n}_{\text{robot}} - \mathbf{1} \right\|_2^2 \tag{4}$$

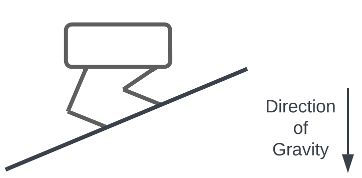
Finally, we convert the ground normal vector to the world frame based on the robot's orientation, which is provided by the onboard IMU.

Adaptation for Stance Legs To walk on uneven terrains, the robot needs to maintain its body to be *ground-level*, instead of *water-level*, so that each leg has an equal amount of swing space (Fig. 8). To achieve that, we express the base pose in the ground frame in the stance-leg MPC controller (Appendix A.2) and adapt the direction of gravity based on the estimated ground orientation.

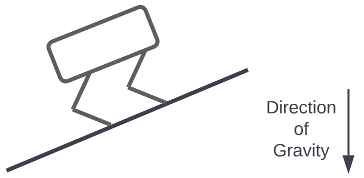
Adaptation for Swing Legs For swing legs, we adapt the *neutral swing position* to be aligned with the direction of gravity. This configuration allows the base to be controlled more easily even on slippery surfaces, as shown in previous work by Gehring et al. [31].

A.4 Impedance Controller to Counter Foot Slipping

Since the MPC-based stance leg controller assumes static foot contacts, an additional slip-handling technique is usually required for the robot to walk on slippery surfaces. For example, to prevent foot



(a) Water Level



(b) Ground Level

Figure 8: Comparison of stance leg strategies on an upward slope. **Left**: If the robot remains water-level, its front legs are overly retracted, and its rear legs are overly extended, which makes it control both legs. **Right**: If the robot remains ground-level, both front and rear legs have an equal amount of extension. We keep the robot ground-level and convert the direction of gravity to the ground frame based on estimated ground orientation.

	Trail 1	Trail 2	Trail 3	Trail 4
Trail Length (km)	0.45	0.41	0.2	0.51
Terrain Type	Dirt	Mixed	Asphalt	Mud
Average Speed (m/s)	0.59	0.74	0.94	0.59

Table 2: Summary of test trails. Our framework selects different locomotion skills (speed and gait) based on terrain type.

slipping, Jenelten et al. [32] implemented a probabilistic slip detector and used impedance control with different gains for slip and non-slip legs. Similar to this work, we also implement an impedance controller to handle foot slips. However, our approach adopts a unified gain for both slip and non-slip legs and does not require a slip detector.

In our impedance controller design, the desired torque for each actuated DoF is the sum of the torque computed by MPC and additional position feedback:

$$\tau = \tau_{\text{MPC}} + k_p(q_{\text{ref}} - q) \tag{5}$$

where τ_{MPC} is the desired torque computed by the MPC solver (Appendix A.2), q is the joint position, and k_p is the position gain. We set the reference, q_{ref} to be the expected joint position as if the foot is in static contact with the ground. Intuitively, when the foot is not slipping ($q \approx q_{\text{ref}}$), the additional torque from the position feedback is small, and our controller falls back to the standard MPC controller. When the foot is slipping, the additional torque from position feedback will bring the leg to the non-slipping position. To determine q_{ref}, we first compute the desired position of the corresponding leg in Cartesian coordinates, $\boldsymbol{p}_{\text{ref}}$. Assuming that the base is moving at the commanded velocity $\bar{\boldsymbol{v}}$ with no foot slip, the foot velocity in the robot frame is just the negated base velocity, and the foot position can be computed by numerical integration:

$$\boldsymbol{p}_{\text{ref}} \leftarrow \boldsymbol{p}_{\text{ref}} - \bar{\boldsymbol{v}}\Delta t \tag{6}$$

where Δt is the control timestep. We then convert from foot position $\boldsymbol{p}_{\text{ref}}$ to joint position q_{ref} using inverse kinematics.

B Additional Experiment Results

B.1 Generalization to Unseen Terrain Instances

We test the performance of our framework in a number of trails not seen during training. Please see Table 2 for some examples. These trails include a number of terrain types that are not seen during training, such as mud, moss, mulch, and dirt (Fig. 9). Our framework generalizes well to these terrains and enables the robot to traverse through them quickly and safely.

B.2 Details about Robot Failures

For each policy tested in the ablation study, we plot its GPS tracking and failure locations in Fig. 10. We find that the incorrect selection of speed and gait can both lead to robot failures. For example, a faster forward speed makes the robot more susceptible to small unevenness on the ground.

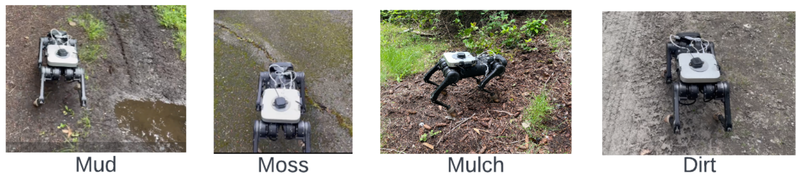


Figure 9: Our framework generalizes to unseen terrain types, such as mud, moss, mulch and dirt.

	Finetuned on Hidden Layers	Finetuned on Class Label	Trained from Scratch
Validation Loss	$\mathbf{0.061 \pm 0.002}$	0.075 ± 0.003	0.088 ± 0.013

Table 3: Comparison of performance on different ways of training the speed policy.

Therefore, non-adaptive policies with higher speeds fail more frequently, especially in areas such as grass or gravel. Moreover, gait parameters such as swing frequency and swing height also affect the controller's stability. For example, a low swing height (SH) can get the legs trapped in deep grass, and a low swing frequency (SF) can result in a large number of foot swings at each step, which can go beyond the robot's capability.

B.3 Ablation Study on Perception Module

Model Architecture and Training We use the FCHarDNet-70 architecture [29], which is obtained from the paper's open-sourced code base. For pre-training on RUGD dataset [30], we train the model for 100 epochs with a batch size of 10 using the Adam optimizer and a learning rate of 0.001. For more robust training, we augment the images from RUGD with random crops and color adjustments. For fine-tuning on demonstration data, we train the model for 60 epochs with a batch size of 32, using the Adam optimizer with the same learning rate of 0.001. Both the pre-training and fine-tuning are conducted on a desktop computer with an Nvidia 2080Ti GPU, where pre-training takes around 6 hours and fine-tuning takes around 20 minutes.

Baselines As discussed in Section. 4.1, we train the speed policy by finetuning on the pixel-wise semantic embedding, which is extracted from the output of the last hidden layer. To justify this design choice, we compare our way of training the speed policy with two baselines. For the model *finetuned on class labels*, we extract the embedding of each pixel from the one-hot encoding of the model's predicted semantic class. For the model *trained from scratch*, we train the FCHarDNet from scratch on the demonstration data without pre-training.

Results We train our method and the baselines on the demonstration data and test the model's performance on a small validation set, where the data is collected on a different trail. For each model, we repeat the experiment 5 times with different random seeds and report the mean and standard deviation of the loss function (mean-squared loss). The result is summarized in Table. 3.

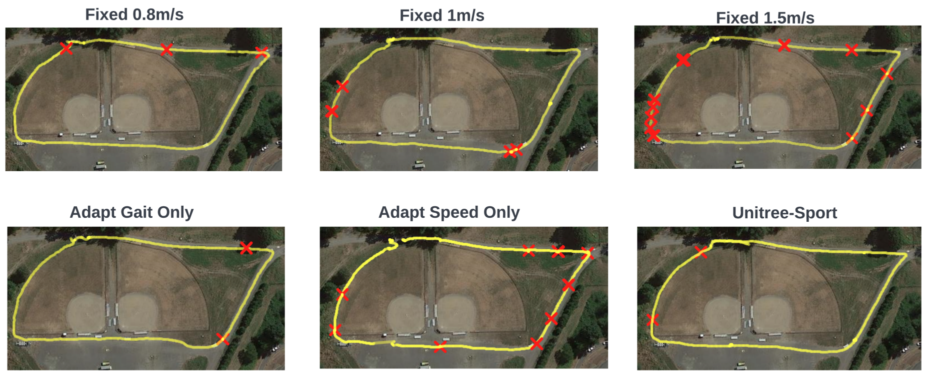


Figure 10: GPS logs (yellow) and failure locations (red cross) for the policies tested in Table. 1

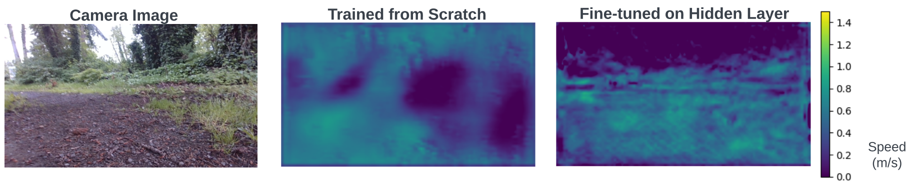


Figure 11: Comparison of different ways to predict the speed map. **Left**: the camera image contains multiple semantic classes, including mulch, grass, and trees. **Middle**: the speed model trained from scratch has a low resolution and cannot identify different semantic classes. **Right**: the speed model fine-tuned from semantic embedding accurately identifies different terrain types and computes the desired speed for each terrain.

Our method, which is finetuned on the output from the hidden layer, achieves the lowest error on the validation set. The model fine-tuned on class label achieves a big loss on both datasets. This is likely due to noisy label prediction, which results from the distribution shift between the model's training data (RUGD) and testing data (robot images). Since the model trained from scratch tunes the entire FCHarDNet on the small set of demonstration data, it overfits to the training data and does not generalize well to the validation set. Moreover, a closer look at the models' predictions shows that the model trained from scratch predicts a blurry speed map with incorrect speed predictions for several regions, compared to our fine-tuned model (Fig. 11). This is likely due to the lack of granularity in demonstration data, which only labels the desired average speed over a fixed region.