Supplementary Material: Coupling Vision and Proprioception for Navigation of Legged Robots

Zipeng Fu*[1] Ashish Kumar*[2] Ananye Agarwal[1] Haozhi Qi[2] Jitendra Malik[2] Deepak Pathak[1]
[1]Carnegie Mellon University [2]UC Berkeley

1. Locomotion Policy Details

Base Policy & Env-Factor Encoder Architecture: We follow the implementation of [3]. The base walking policy is a multi-layer perceptron (MLP) with 3 hidden layers. The input is the current state $x_t \in \mathbb{R}^{30}$, previous action $a_{t-1} \in \mathbb{R}^{12}$ and the extrinsics vector $z_t \in \mathbb{R}^8$ and the output is 12-dim target joint angles. The dimension of hidden layers is 128. The extrinsics vector z_t is estimated by an environment factor encoder. The environment factor encoder is a 3-layer MLP (256, 128 hidden layer sizes) and encodes $e_t \in \mathbb{R}^{17}$ into $z_t \in \mathbb{R}^8$.

Adaptation Module Architecture: The adaptation module first embeds states and actions into 32-dim vector using a 2-layer MLP. Then, a 3-layer 1-D CNN convolves the representations across the time dimension to capture temporal correlations in the input. The input channel number, output channel number, kernel size, and stride of each layer are $[32, 32, 8, 4]$, $[32, 32, 5, 1]$, $[32, 32, 5, 1]$. The flattened CNN output is linearly projected to estimate $\hat{z}_t$.

Learning the Walking Policy: We jointly train the base policy and the environment encoder network using PPO [5] for $15,000$ iterations (1.2B sample, 24 hours) each of which uses batch size of $80,000$ split into 4 mini-batches. We then train the adaptation module using supervised learning with on-policy data. We run the optimization process for 1000 iterations (80M samples, 3 hours) and use Adam optimizer [2] to minimize MSE loss. The batch size is $80,000$ split up into 4 mini-batches.

Reward Function: The reward at time r_t is defined as the sum of the following quantities:

- Velocity Matching: $-|v_x - v_x^{\text{cmd}}| - |\omega_{\text{yaw}} - \omega_{\text{yaw}}^{\text{cmd}}|$
- Energy Consumption: $-\boldsymbol{\tau}^T \dot{\boldsymbol{q}}$
- Lateral Movement: $-|v_y|^2$
- Hip Joints: $-\|\boldsymbol{q}_{\text{hip}}\|^2$

The corresponding scalings are 20, 0.075, 1 and 0.2. The survival bonus is set by a simple rule as $10 + 20(v_x^{\text{cmd}} + \omega_{\text{yaw}}^{\text{cmd}})$.

We list the ranges of command linear velocity and angular velocity in Supplementary Table 1. We re-sample the command velocities within a single episode with probability 0.004.

Task	Command Linear Velocity Range (m / s)	Command Angular Velocity Range (rad / s)
Curve Following	[0.15, 1.0]	[-0.4, 0.4]
In-Place Turning	[0, 0.15]	[-0.6, 0.6]

Table 1. Command velocity range for curve following and in-place turning.

2. Safety Advisor Details

Hyperparameters: Velocity changes in Fall Predictor and the size of obstacles in Collision Detector are set by simple rules. For instance, (a) if a fall is predicted, the safety advisor module decreases the velocity limit by a large amount (we pick 0.2 m/s), so the robot can slow down quickly; (b) otherwise, it increases the velocity limit by a small amount (we pick 0.05 m/s) for conservative speed up; (c) the size of obstacles in Collision Detector (9cm x 3cm) is set to roughly be the size of the head of the robot. Additional real-world experiments [link] show that if the obstacle is set to be larger, the robot will take a more conservative path around the unexpected obstacle. If the obstacle is set to be smaller, the robot takes a shorter path but risks colliding legs with the unexpected obstacle.

Network Structure: Similar to the adaptation module, both the collision detector and fall predictor module share the same architecture and embed states and actions into a 32-dim vector using a linear layer. Then, we use 3 layers of 1D convolutions with input channels, output channels and strides $[32, 32, 8, 4]$, $[32, 32, 5, 1]$, $[32, 32, 5, 1]$. The output is a sigmoid scalar.

Training Data and Environments: The scalar sigmoid output predicts a probability value, indicating whether the robot collides with an obstacle in the Obstacle Detector, or if the robot falls at time $t + 100$ and 0 otherwise (note that one simulation time-step is 0.01s) in the Fall Predictor. We train both modules in an self-supervised fashion by collecting data from robot walking / colliding with the obstacles / falling down. Data are collected given random command linear/angular velocity commands in environments with randomly sampled frictions, terrain roughness and payload values from the following list:

- Coefficient of Friction: $[0.1, 0.6, 1.1, 1.6, 2.1]$.
- Payload: $[1.2, 2.4, 3.6, 4.8, 6.0]$ (kg).
- Rough Terrain z-scale: $[0.01, 0.08, 0.14, 0.23]$ (m).
- Linear Velocity: $[0, 0.5, 1.0]$ (m/s).
- Angular Velocity: $[-0.4, 0.0, 0.4]$ (rad/s).

We train both Obstacle Detector and Fall Predictor for 145k iterations with a batch size of 1000. At simulation test time, we run both the collision detector and fall predictor at 5Hz whereas for deployment on robot we train a lightweight version using only the last 20 timesteps of observation history and run it at 10Hz.

3. Visual Planner Details

We command the angular velocity for our robot and the baseline LoCoBot using the following equation:

$$\omega_t^{\text{cmd}} = K_p \cdot (\theta_t^{\text{target}} - \theta_t) + K_d \cdot (\omega_t^{\text{target}} - \omega_t) \quad (1)$$

where $K_p = 1, K_d = 0.02, \omega^{\text{target}}$ is set to 0. The command angular velocity is clipped to the range in Supplementary Table 1 before being sent to the locomotion policy in order to be consistent with the training setting. We also observe that when the linear speed is low (less than 0.1m/s), the locomotion policy is unable to make in-place turns with a small commanded angular velocity, due to the imperfection of our locomotion policy. Thus in this case we clip the absolute value of the commanded angular velocity to be at least 0.4 to compensate this imperfection. We empirically observe a higher performance even when the command is sub-optimal, mainly because our planning algorithm operates in a relatively high frequency and can soon correct the angular velocity command as soon as the linear velocity becomes large enough.

4. LoCoBot Baseline & Discrete Planner Details

We import the PyRobot URDF model [4]. Both our method and the LoCoBot use a control frequency of 100Hz and a planning frequency of 10Hz. We follow [1] to convert commanded linear and angular velocity to the angular speed of the left and right wheel of the LoCoBot. We set the forward action of the discrete planner at 0.6 m/s after we measured the average speed of the continuous planner in the same evaluation environment is around 0.6 m/s. The low level controller is also a PD controller with $K_p = 10$, $K_d = 0.05$. The controller gain is adjusted so that no obvious motion jerk happens during movement. Since the control of wheeled robot is simpler and more accurate, we

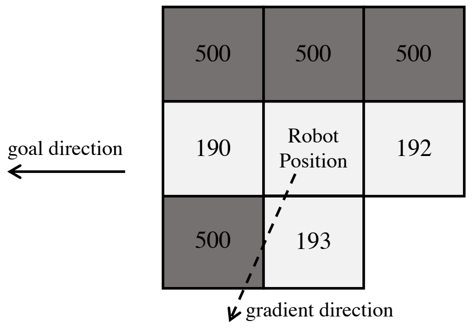

Figure 1. An example of local minima produced by the map. The gray square represents the non-traversible areas. The white square represents the traversible regions. The number in each square represents the cost on that point. At the current position, the robot will orientes to the bottom left which the linear velocity commands will command 0 velocity, in which case the robot got stuck in the local minima.

do observe the LoCoBot being more likely to stuck in local minima in the cost map (an illustration is shown in Supplementary Figure 1). For our robot, since the locomotion policy is not perfect and the legged robot is harder to control compared with LoCoBot, it sometimes can get out of the local minima due to the noisy movement, which is the reason why we perform better in the perfect flat ground (Table 4 (a) and (b) in the main text). However, we want to emphasize again that our point here is not to show our robot performs slightly better than baseline in the flat ground. Instead, what we show is the ability to traverse and navigate over difficult terrains where LoCoBot easily fail (Table 4 (c), (d), and (e) in the main text).

References

[1] Gregory Dudek and Michael Jenkin. *Computational principles of mobile robotics*. Cambridge University Press, 2010. 2

[2] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015. 1

[3] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. RMA: Rapid Motor Adaptation for Legged Robots. In *RSS*, 2021. 1

[4] Adithyavairavan Murali, Tao Chen, Kalyan Vasudev Alwala, Dhiraj Gandhi, Lerrel Pinto, Saurabh Gupta, and Abhinav Gupta. Pyrobot: An open-source robotics framework for research and benchmarking. *arXiv:1906.08236*, 2019. 2

[5] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv:1707.06347*, 2017. 1