Tumbling Robot Control Using Reinforcement Learning

Andrew Schwartzwald, Matthew Tlachac, Luis Guzman, Athanasios Bacharis and Nikolaos Papanikolopoulos
{ schw1818 | tlach007 | guzma102 | bacha035 | papan001 }@umn.edu
Department of Computer Science and Engineering, University of Minnesota

***Abstract*— Tumbling robots are simple platforms that are able to traverse large obstacles relative to their size, at the cost of being difficult to control. Existing control methods apply only a subset of possible robot motions and make the assumption of flat terrain. Reinforcement learning allows for the development of sophisticated control schemes that can adapt to diverse environments. By utilizing domain randomization while training in simulation, a robust control policy can be learned which transfers well to the real world. In this paper, we implement autonomous setpoint navigation on a tumbling robot prototype and evaluate it on flat, uneven, and valley-hill terrain. Our results demonstrate that reinforcement learning-based control policies can generalize well to challenging environments that were not encountered during training. The flexibility of our system demonstrates the viability of nontraditional robots for navigational tasks.**

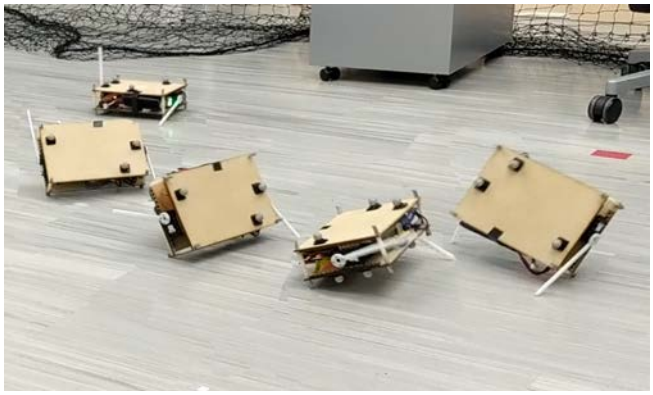


Fig. 1: One of our tumbling robot prototypes navigating to a setpoint (collage of several frames from a physical trial).

I. Introduction

In recent years, many distinct tumbling robot platforms have been designed. This class of robots uses gravity to assist their movement by rolling their body through a given terrain. These type of robots can be effective in applications where a high mobility-to-size ratio is required [1]. It is often the case that real world environments include complex terrains composed of varying surfaces, rapid changes in elevation, and impassable obstacles. Tumbling locomotion can allow robots to traverse these complex environments in which traditional locomotion fails. However, this comes with the cost of being difficult to control. For example, interactions with the terrain can be used to efficiently maneuver down a hill, but they can also cause the robot to roll in unanticipated directions. All these characteristics make tumbling robots an attractive option for mobile robot applications, and they open an interesting area of control research where a robot's motion cannot be modeled or predicted.

Tumbling motion is defined as being stochastic—for a given motor command, we are not able to predict the motion of the robot until after it has executed a tumble. Even seemingly simple maneuvers such as driving forward require constant adjustment as variations in surface friction and terrain height can easily perturb the robot significantly from its desired path. As a result, tumbling robots often need to segment their movement into discrete tumbles [2], where they can analyze the result of their previous action and correct for any disturbances that caused them to deviate from their goal. This represents a challenging control problem, as there is no direct mapping from the disturbance to a motor command that can correct for it. Tumbling robots require a much more sophisticated control policy that has only become feasible in recent years due to advancements in reinforcement learning (RL).

Our previous work in [2] takes the first steps of exploring the importance of RL for tumbling locomotion. It proposes the use of RL methods for tumbling robot control, due to controller complexity and environmental uncertainty. In this work, we examine how well RL policies perform in comparison with a simple controller and a random policy. Also, we test the RL policies in different environments than they were trained in, and measure their performance. We focus on the task of setpoint navigation, and evaluate trained policies on three different environments: flat ground, an uneven surface, and a valley-hill terrain (shown in Figure 2). The testing of the policies takes place in simulated and real-world environments, demonstrating promising results for deploying tumbling robots outside of the lab.

II. Related Work

A. Control of Tumbling Robots

The benefits of tumbling locomotion include terrain-body interactions that allow tumbling robots to successfully navigate many terrains with few actuated degrees of freedom [3]. As discussed in [2], [3], tumbling locomotion has several advantages over other methods, such as simple hardware and increased mobility relative to size. These characteristics have led to tumbling robots being designed for applications that require small and light robots such as underwater sampling [4] and jumping over obstacles [5]. More importantly for our application, tumbling locomotion has proven to be a difficult control problem that we can use to test the capabilities of reinforcement learning-based control policies.

Developing a control policy for tumbling robots can be a notoriously complex task, especially in diverse environments. Usually, these methods utilize feedback from the robot's

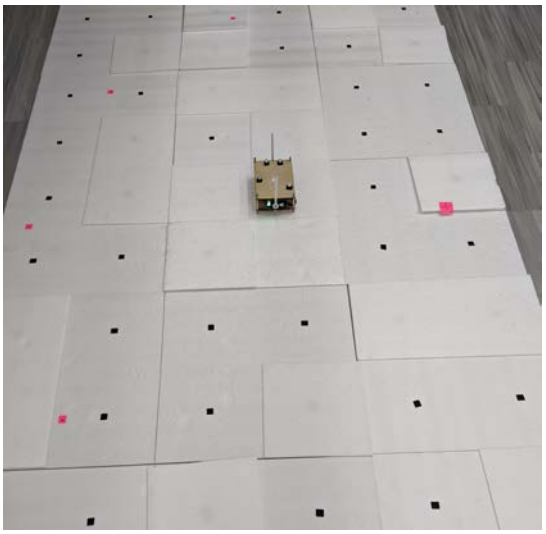
motors and make assumptions about the environment's characteristics such as specific surface friction and zero elevation gain. Previous work has focused on motion primitives for locomotion on a flat surface [1], while there have also been discussions for climbing step locomotion in [3] and [6], though neither of those have demonstrated a control policy for tumbling locomotion. The movement of these robots is highly dependent on environmental conditions, with varying friction, terrain slope, and an uneven surface limiting its performance. An adaptable and robust control policy that can leverage complex maneuvers and handle diverse terrain would greatly expand the capabilities of tumbling robots.

Reinforcement learning has proven to be a successful control technique when we lack a full model of a robot's locomotion; furthermore, the use of deep reinforcement learning to train a model in simulation and then transfer it to real-life control has been demonstrated in many different contexts [7]–[10]. The real-world environment can never be perfectly replicated in simulation, so simulation parameters such as friction, gravity, and motor responses are varied through a process called domain randomization. This results in a network more capable of generalizing to different environments, thereby making it more robust to transfer from simulation to the real world [9]. With sufficient domain randomization, the real world is a subset of the simulated environments used for training. A prominent example where this methodology achieved success is [7] where a large set of randomized environments were used in simulation to real (sim-to-real) transfer.

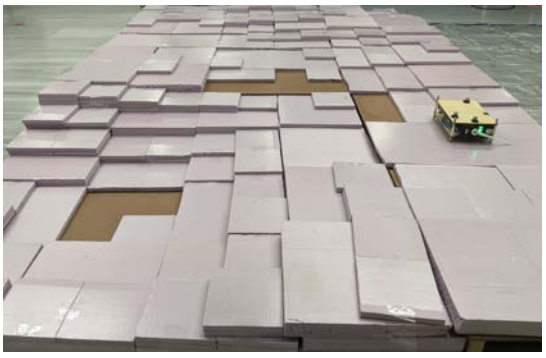
Similarly, Maekawa et al. attached irregularly shaped branches to servos and implemented a sim-to-real locomotion controller [10]. This demonstrates the use of reinforcement learning in developing control policies for nontraditional robot designs. However, in that work, tumbling motions were avoided as the robot was tethered. Tan et al. similarly developed a locomotion policy for a quadruped [8]. Tumbling robot motions were generated in [11], but are only tested with a simple robot in simulation and are only applied to movement in a 2D plane. The first time that RL was used to control a physical tumbling robot was in [2], and in this paper we explore how well RL control policies generalize to new and complex real world environments.

B. RL Method & Robot Design

Proximal Policy Optimization (PPO) [12] is an on-policy RL algorithm that belongs to the family of policy gradient methods [13]. In [12], PPO establishes its importance over other methods, such as Actor Critic approaches [14], due to the algorithmic simplicity and better convergence rates. Many recent works have demonstrated success using this method for robot control applications, including the control of a humanoid robot [15]. Although methods like soft-actor-critic can offer a better sample efficiency, the improved stability of on-policy algorithms, along with the relatively inexpensive sampling process of training within a simulated environment, makes PPO a more attractive method.



(a) Uneven surface constructed with randomly placed foam tiles. Each tile is 1 ft x 1 ft x 0.25 in.

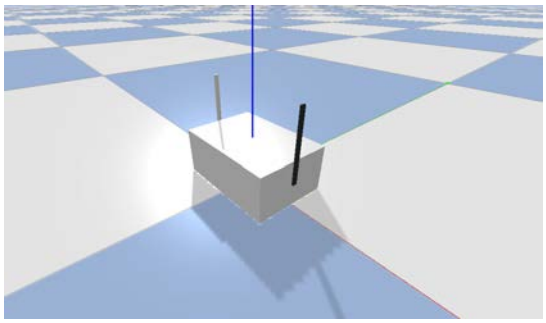


(b) Valley-hill terrain constructed with foam tiles. Maximum terrain height is 10 inches. Each tile is 0.5 ft x 0.5 ft x 0.5 in. A tarp (not pictured) was laid over the terrain during physical trials to prevent the tumblebot's legs from getting stuck in gaps between the tiles.

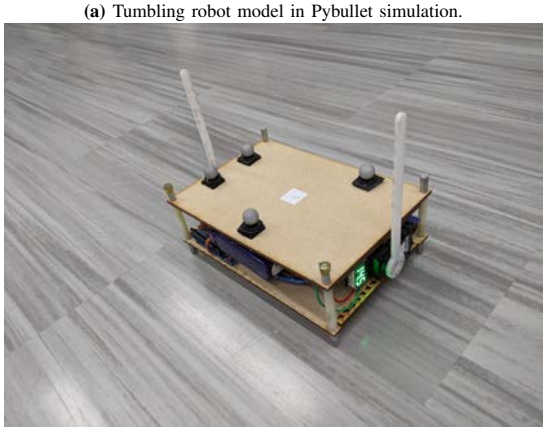
Fig. 2: The two different terrains used for the real world experiments.

Usually, the policy network of PPO is a fully connected neural network, but recurrent neural network policies can also be used in RL algorithms to improve robustness when using noisy data [16]. Recurrent neural networks allow the policy to observe its past actions, which can lead to more sophisticated behavior such as attempting a different action if the first was unsuccessful. [7] applied this concept for the task of manipulating cubes with a robotic hand. In the aforementioned application, PPO was successfully used to optimize a recurrent neural network; in particular, long short-term memory (LSTM) [17] was used. For our application, we chose a LSTM policy due to its robustness to noise [7], [16], and we trained the policy with PPO due to its fast convergence and effectiveness in continuous state spaces [12].

We designed a new tumbling robot to serve as a testing platform. Like the one in [2] this was inspired by the Adelopod [18], and it uses two single degrees of freedom legs attached to continuous rotation servos. Similarly, no servo position feedback was utilized, and encoders were not added to provide it. The new robot features higher body friction, which improves its ability to overcome uneven terrain and obstacles. ROS (Robot Operating System) was used onboard



(a) Tumbling robot model in Pybullet simulation.



(b) Tumbling robot prototype.

Fig. 3: The two robots used in simulation and real world experiments.

to allow modular configuration.

After training the policies to convergence in simulation, we evaluated their performance in various real-world environments, as well as in simulation. Our results demonstrate a successful sim-to-real pipeline for tumbling locomotion, and a robust control policy able to adapt to different terrain environments.

III. Approach

A. Robotic Platform

1) Hardware: First, we constructed a simple and low cost tumbling robot platform. An image of the prototype is shown in Subfigure 3b, and its simulation version in Subfigure 3a. In [3], low leg friction with high body friction was found to increase performance in step climbing. Accordingly, the legs were cut from low-friction Delrin plastic and the exterior aluminum standoffs were coated in rubber to increase body friction. Two Power HD continuous rotation servos were used for actuation. These servos are inexpensive yet provide sufficient torque and velocity at a low mass. An inertial measurement unit (IMU) was included, however, its data was not used. Rather, we used a Vicon motion capture system to obtain the pose and ensure accurate position feedback to the robot. Since the purpose of the IMU is to enable tumbling locomotion outside of the laboratory setting, use of filtered IMU data instead is left to future work. Hardware connections are shown in Figure 5 of [2].

This platform differs from existing tumbling robots in its simple blocky design which allows for more robot parameters to be easily modified in simulation. Randomized Unified Robot Description Format (URDF) models can be generated programmatically, increasing the scope of possible domain randomization. We generate a new robot each time one is spawned in simulation during training by randomizing 37 total parameters. Each generated tumble bot has varying dimensions, inertia characteristics, motor responses, and friction coefficients. Every parameter is randomized with a standard deviation of 5% of the nominal values. This amount of domain randomization is necessary to ensure that the trained policy is robust to small errors in these parameters. Since the real tumble bot is within the subset of these generated robots, the policy is expected to transfer well to the real world.

2) Software: For the physical prototype, the main onboard computer is a Raspberry Pi 3 Model B+, running Ubuntu 16.04 with ROS Kinetic. It is configured as an access point, executing motor commands produced by the trained network running on a separate computer and receiving a stream of Vicon motion capture data. The tumble bot uses an Arduino Uno as a motor controller between the Pi and the servos to allow for future expansion and control of different servos. We chose the Raspberry Pi for its compatibility with baselines models. For future work, all computation would be performed by the Raspberry Pi, with the only external communication being setpoint commands.

The flow of data is shown in Figure 6 of [2]. We configured the training environment using OpenAI Gym. For each episode step, the servo outputs are sent from Gym using a TCP socket. The robot returns a position and orientation observation using the most recently received mocap data. A timer is used to ensure that communication is performed no faster than the rate it was modeled as in simulation. By choosing a relatively slow rate of 1 Hz effects of communication latency are minimized. We designed the system for a latency of about 20 ms, allowing theoretical rates of about 50 Hz.

The system can easily switch between manual and autonomous control, with a safety override to stop movement. This was accomplished through the use of ROS. ROS is a communication framework that allows different robots to use the same software packages and makes it easy for these packages to interact with each other. This package and node system enables a modular software approach. The tumbling robot's servo outputs are controlled via commands from the Gym environment though the Gym link node, when they are not being manually controlled with a joystick through the joystick node. While IMU data was not used, the mocap node could be swapped for the IMU node. This would shorten the time required to move the lab experiments to an outdoor environment.

B. Policy Training

Training a policy in simulation rather than the real world decreases training time [9], as long as the reality gap between the simulated and real environments is not too large.

1) Simulation Environment: Simulation was performed using PyBullet, a Python interface to the Bullet physics

Parameter	Value
learning rate	3e-4
episode length	20 steps
batch size	128 steps
discount (γ)	0.99
clipping parameter	0.2
optimizer	Adam

TABLE I: PPO2 hyperparameters used in training.

engine. The simulation's numerical solver's rate was left at the default of 240 Hz. Between every action output, the simulation was stepped 240 times. This results in an action frequency of 1 Hz. While a much higher frequency is possible, it is not required for effective locomotion. Each PPO episode was limited to 20 steps, which corresponds to episodes of 20 seconds and 4,800 simulation steps.

The simulator was sensitive to changes in the solver's rate, due to the many collisions involved in the simulation of tumbling motion. With too low of a rate, simulated joint control was inconsistent at low velocities. The rate, number of simulation steps per episode step, and number of steps per episode were empirically chosen to result in fast training times, a policy that converges, and consistent simulator behavior.

Measurements of the physical prototype were translated into simulation through URDF models, similar to the process described in [9]. We segmented the structure of the robot into three parts: the body and two legs. For modeling inertial characteristics, calculations were done assuming uniform density, using the measured mass and dimensions. A high friction coefficient, 0.9 for lateral friction, was chosen empirically to minimize behavior such as sliding that would be unlikely to transfer well [10]. As described in Section III-A.1, these values were used as the means for randomized generated URDF models. This randomization affects the dynamics of the tumbling robot's movement.

The tumbling bot's servo controllers do not provide a consistent torque. At near-zero velocities, torque is significantly reduced. This behavior is approximated in simulation by setting the motor's force to 0.1 N with Gaussian noise, where standard deviation is 0.1 N. At higher velocities, servo torque is set to 10 N. Servo velocity was also randomized, with a standard deviation of 0.8 rad/s. This randomization is significant relative to the commanded velocities, but was essential to having a robust policy.

2) Reinforcement Learning: We used the Stable Baselines implementation of PPO with an MLP LSTM policy. We set the learning rate to 3e-4, the same as was used in [7]. Nminibatches was set to 1. For the hyperparameter selection, we used the same ones proposed in [12], due to high performance in their testing/evaluation. A selection of values are shown in Table I.

Table II, shows initial results that support our choice of a recurrent policy. Both policies performed similarly, although the LSTM achieved slightly higher rewards at times. We also tested different RL algorithms, including Advantage Actor Critic (A2C) [19] and Trust Region Policy Optimization (TRPO) [20]. These are the only other algorithms within Stable Baselines that are compatible with the MultiDiscrete

Algorithm	Reward
PPO-LSTM	9.342
PPO-MLP	7.758
A2C	9.333
TRPO	8.089

TABLE II: Asymptotic Reward of Different Reinforcement Learning Methods

action space, so testing other algorithms would require modifying the tumble bot action space. In Table II, we see that A2C achieved a large asymptotic reward, which is similar to PPO-LSTM. On the other hand, TRPO did not perform that well, followed by PPO-MLP. Each algorithm was able to train a successful policy, which is expected due to the small action space, but the higher reward values indicate a policy that can more quickly and accurately reach the target. Since each RL algorithm we tested performed equally well or worse than PPO-LSTM, we chose to apply this method for our task; in general, it is also the most stable and likely to converge [12].

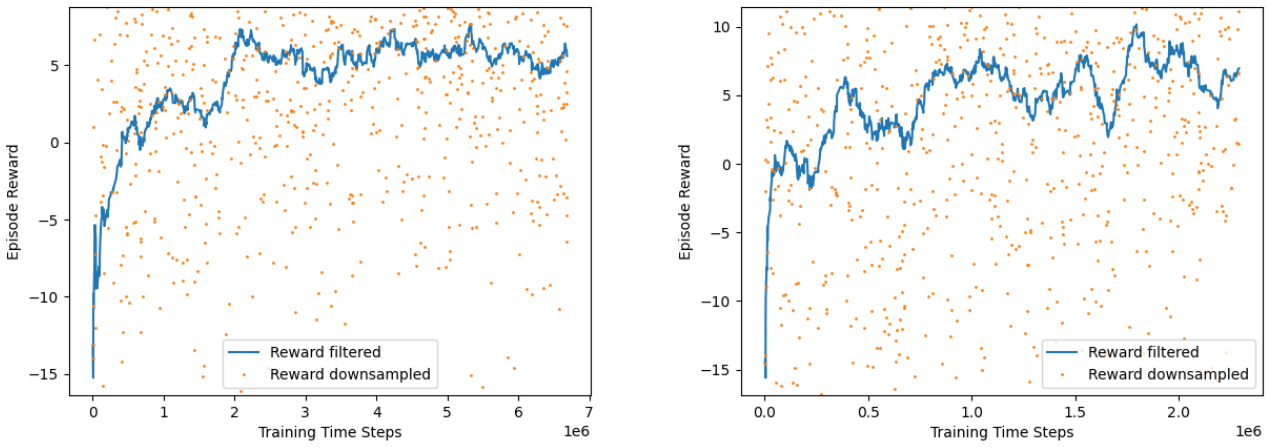
PPO allows discrete or continuous action and observation spaces. A discrete action space was used, with three possible outputs for each of the two legs. These outputs are velocities of -5.25 rad/s, 0 rad/s and 5.25 rad/s. This small action space allows for a simple approximation of the real world servos in simulation. Additionally, limiting the number of actions greatly limits the dimensionality of the policy optimization. Adding the full range of servo velocities would exponentially increase the amount of exploration needed during training and would not add significant physical capability to the robot agent.

The observation space is continuous and includes the position of the center of mass, orientation of the robot torso, and the intended output velocities. Center of mass velocity is included in reward calculation, but not in the observation. This way, after transfer, the learned policy can still be loaded, and velocity data does not need to be estimated online. Decreasing the dimension of the observation space was found to improve transfer success in [8].

Adding noise to the output velocities, as described in Section III-B.1, prevents the policy from easily approximating servo position. The velocity and torque changes applied are not directly made visible to the policy. However, it is possible for leg positions to be inferred through body orientation along with previous servo commands. By creating a network that is resistant to unpredictable servos, we predict that unmodeled phenomena are covered by these randomizations. For example, no battery model is used, but on the real robot, battery state affects servo speed. This would increase the likelihood of successful transfer without requiring extensive domain randomization.

To demonstrate navigation to a setpoint, the setpoint was fixed to the origin while starting location was varied. Navigation to other setpoints could be achieved by shifting the coordinate system. At the start of each episode, the robot's location was reset to a uniformly random position along a half circle of radius 1.5 m.

The reward function used is the same as [2], and is given by:



(a) Trained on flat ground.

(b) Trained on uneven surface.

Fig. 4: Plots of episode reward during training. Data has been downsampled to 1000 samples, and filtered with a window size of 50.

$$R(s_t) = \frac{1}{1+\sqrt{v_x^2+v_y^2}} - \sqrt{p_x^2+p_y^2}$$

where v_x and v_y are the x and y components of velocity, and p_x and p_y are the x and y components of position. The velocity component discourages circling the setpoint, and encourages lower energy consumption.

One challenge with tumbling robots is that there is not a clear failure state - other systems, such as a quadruped robot [8], may terminate an episode when the simulated robot falls. We chose to treat travel past 3 m away from the goal as a failure state for the tumbling robot. In simulation, this causes a reset. For real-world transfer, this failure state was unchecked, since the network had learned to avoid the boundary post-training.

3) Training Results: Training was performed on a Linux machine with an Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz and 32 GiB of memory. We note that training on a GPU had negligible performance gains due to the complexity of the physics simulation, so all training was CPU-based. One network was trained on flat ground, while the other trained on a randomly generated surface that mimics the foam tiles shown in Figure 2a; their episode rewards during training are plotted in Figure 4.

We compare our training results against a random policy and a simple control approach. This controller treats the tumble bot as a differential drive wheeled robot, which is the control method that most closely matches the locomotion of the tumble bot. In more detail, given the relative angle to the goal, the robot sends motor commands to account for errors in its trajectory. Although wheeled robots can use a proportional controller, the tumble bot hardware and simulation environment only support discrete actions, so we chose an angle threshold where the robot would update its movement to either drive straight, left, or right. As expected, this simple control approach does not transfer well to the tumble bot due to the unpredictability of tumbling motion, so this serves as an important baseline for our RL approach to beat.

Due to the increased number of collisions being calculated by the physics engine, the uneven network took significantly longer to train than the flat network. As a result, the flat network trained for 10.86 times as many time steps, despite additional training time being allotted to the uneven network. Any potential robustness increase from training the network on terrain comes with a tradeoff in required training time or resources.

C. Evaluation Environments

We evaluated each policy on three different environments: flat ground, the randomized surface pictured in Figure 2a, and the valley-hill terrain in Figure 2b. The valley-hill terrain was generated using perlin noise, with a maximum height of 25.4 cm (10 inches). We set this height value to be the maximum value which the tumble bot could physically overcome. Preliminary testing on both the physical and simulated tumble bot (shown in Figure 5) showed that steps greater than 5cm are difficult for the tumble bot to overcome due to its body dimensions. We chose the 25.4 cm terrain height since it had no single step size greater than 5cm. We utilized two octaves of perlin noise to produce a terrain that features a low-frequency slope and relatively high-frequency valleys and hills. The terrain is then quantized so that it can be represented using collision boxes. This representation simplifies the terrain construction both in simulation and for the laboratory trials. An example of the simulated terrain environment is shown in Figure 6. We generated a similar environment for the uneven surface, using uniformly random numbers to specify the terrain height within a given maximum value.

IV. Experimental Results

After training the policies in simulation as described in Section III-B, we evaluated their performance both in

Policy	Flat Sim Setpoint Error		Flat Real Setpoint Error		Uneven Sim Setpoint Error		Uneven Real Setpoint Error		Valley-Hill Sim Setpoint Error		Valley-Hill Real Setpoint Error	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Flat	0.104	0.064	0.250	0.158	0.183	0.173	0.269	0.206	0.620	0.314	0.554	0.339
Uneven	0.219	0.164	0.433	0.407	0.312	0.307	0.335	0.243	0.685	0.420	0.557	0.307
Simple Controller	1.081	0.890	-	-	1.166	0.879	-	-	1.123	0.770	-	-
Random Policy	1.395	0.642	-	-	1.374	0.602	-	-	1.367	0.758	-	-

TABLE III: Setpoint accuracy for simulated and real-world trials. Networks trained on flat ground and uneven surfaces were evaluated in three environments. All values are in meters.

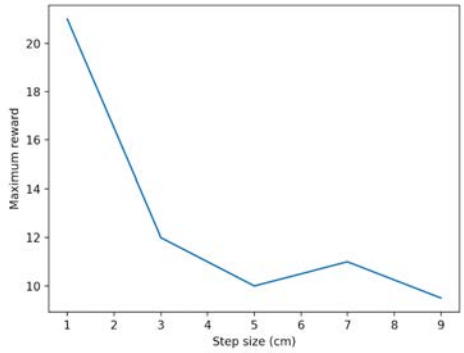


Fig. 5: The maximum reward for various height values on a random terrain. Reward plateaus at 5cm, indicating this is the maximum step size that the robot can overcome.

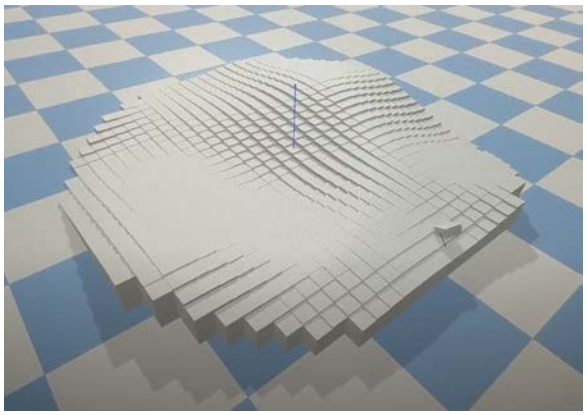


Fig. 6: The simulated valley-hill terrain, generated from perlin noise.

simulation and in the real world. In simulation, 100 trials were performed per network. The robot starting location was randomized, and all domain randomization parameters were set to the corresponding measurements of the physical robot and testing environment. Results are reported in Table III.

100 trials were also performed per network in the real world environments. The tile floor was significantly more slippery than the foam used to construct the terrain. As pictured in Figure 7, physical trials were started from one of five locations, spaced 30° apart around a semicircle of radius 1.5 meters. Note how the initial robot orientation remains fixed as the starting position is moved around the semicircle, resulting in a variety of angles from which the robot must approach the origin. Raw pose data from the VICON system for a tumbling platform results in messy-looking trajectories, so smoothed trajectories are plotted instead. Smoothed trajectories are generated using B-splines from the `scipy` package with the smoothing parameter set to 0.2. An example of raw vs. smoothed trajectories

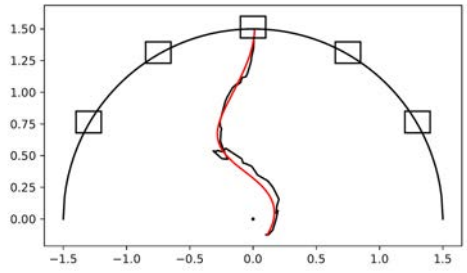


Fig. 7: Raw data and a smoothed trajectory for one physical trial. Each trial begins at one of the five indicated positions around the semicircle. All distances are in meters.

is also compared in Figure 7. The smoothed trajectories for every physical trial are plotted in Figures 8 and 9, separated by policy and evaluation environment (flat ground, uneven surface, or valley-hill terrain). Setpoint errors are again reported in Table III.

Table III shows that the network trained on flat ground performed better than the network trained on terrain in every trial, real-world and simulation alike. This is an interesting outcome because a policy trained on a more complex environment would be expected to be more robust. However, including the terrain during training also results in training from noisy data, since the robot's trajectories are less predictable. This suggests that any robustness gained from a more challenging environment is offset by a less stable training process.

Performance was better on flat ground than the uneven surface except for the real-world evaluation of the network trained on terrain. This may be explained by the difference in friction between the flat tile floor and the foam terrain—the slicker floor resulted in the robot sliding more, which makes the results of tumbles harder to predict, and the terrain network may have been less robust to friction changes due to the fewer number of training iterations. The flat network, in contrast, had more time to adjust to different coefficients of friction, but had never seen uneven terrain before the evaluation stage. On average, the setpoint errors in the real-world trials were only 1.13 times those in the corresponding simulated trials, which demonstrates a robust system that can transfer to real life remarkably well.

We also note that in the valley-hill terrain, both networks performed better in real life than in simulation. We attribute this to the real robot's rubber stand-offs, which were not modeled in simulation and resulted in a better step-climbing

ability. This analysis agrees with [3], which identified ideal contact as a necessity for efficient step-climbing. Additionally, the simulated setpoint error was calculated on various terrain generated from perlin noise, which may have been more challenging than the single terrain instance we constructed for the real-world tests. Both policies had similar setpoint error in the physical trials, which suggests that physical limitations of the tumblebot (such as step-climbing ability) may have a greater role in maneuvering terrain than more robust policies.

Some additional insights can be derived from the plots in Figures 8 and 9. It appears as though the beginnings of trajectories for the uneven network are significantly more consistent than those of the flat network trajectories. Despite this difference, the flat network performed much better on average. One interpretation is that the uneven network learned a policy that is more robust to noisy data, despite converging to a sub-optimal setpoint error. For evidence of this, consider the trials starting from the leftmost initial position. In Subfigure 8b, many trajectories follow the same general strategy - initially heading towards the origin, passing above it, and trying to recover with a large circle. Subfigure 8a, in contrast, shows a policy with significantly more varied and effective strategies for recovering from missing the origin.

Setpoint error aside, comparing the network performance when evaluated on the uneven surface implies that the uneven network was more robust in its presence. The flat network has significantly messier paths, but a better strategy for recovering from unexpected setbacks. The comparative consistency of the terrain network shows promise, and it may well be worth using in situations where path consistency is more important than setpoint error.

The results in Figures 9a and 9b follow more unpredictable paths since the terrain height has a much greater effect on the tumblebot's motion. By comparing the overlay of terrain heights, we note how the tumblebot tends to follow the terrain slope, often falling into the darker regions of the plot (valleys) and avoiding the lighter regions (hills). We observe that the flat network visited a larger area of the terrain and tended to climb straight over hills. In contrast, the uneven network avoided hills by traversing around them when possible. This behavior can be attributed to the uneven network encountering impassable areas during training, which the tumblebot must learn to avoid using the LSTM policy. Essentially, if the uneven network detects that the robot is blocked, its next action will be to turn another direction instead of continuing forwards.

During flat and uneven surface trials, the policies rarely leveraged simple strategies like turning in place and then driving forwards. Instead, trajectories are often sets of arcs, created by one leg flipping the robot forward, and the other colliding with the ground and turning it. The specifics of the leg positions, and whether the leg responsible for turning the robot is also moving, result in different degrees of turning and forward motion. These complex maneuvers allow for a fine degree of robot control, and are more sophisticated than typical motion primitives as explored in [1] or anything a human driver would be likely to achieve.

The results on the valley-hill terrain show a great adaptability of the policies trained in simulation. Not only can they transfer to a real robot and successfully navigate to a setpoint, but they can do so in an environment that is much more challenging than anything they experienced during training. Although the tumblebot can occasionally get stuck in a valley, we attribute this to physical limitations of the robot, since both policies had similar success in this scenario. Our results demonstrate that the policies are able to successfully recover from the unpredictable trajectories of the valley-hill terrain, which is the first time tumbling robots have been able to maneuver non-flat terrain.

V. Conclusion

This work demonstrates successful real world transfer of a control policy to a tumbling robot and shows that learned policies can generalize well to unknown terrain. It was accomplished with no training on real world data and only a crude simulation model. We achieved a successful sim-to-real control policy pipeline using domain randomization, avoiding the use of complex traditional robot control schemes. Our results demonstrate that reinforcement learning can provide an adaptive and robust policy for controlling unpredictable locomotion in complex environments. This work is the first step towards developing a tumbling robot control policy that is capable of navigating unknown environments autonomously.

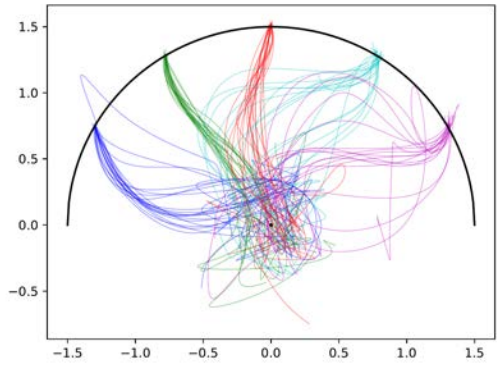
VI. Future Work

Future work for outside of lab deployment includes moving the rest of the computation to the onboard computer and using IMU data in place of mocap data. Adding environmental sensors to the prototype or using the policy to control an already outfitted robot such as the Aquapod [4] would enable the robot to perform useful work.

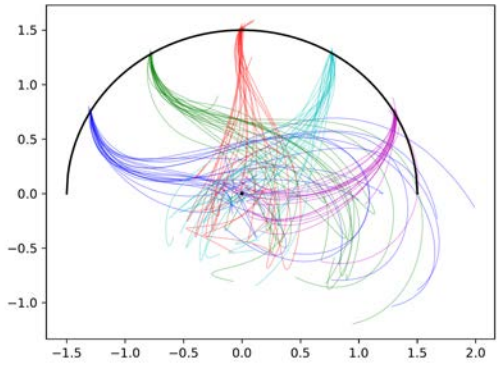
While in this work only a small amount of domain parameters needed randomization for transfer to a laboratory environment, robustness could be improved through a larger scope of domain randomization.

A limitation of the present work is that a constant starting orientation was used, but due to the robot's tumbling, the final orientation is difficult to predict, which influences future paths. Training with a greater range of initial poses (possibly through the use of automatic domain randomization [21]) may produce a policy which is better suited for navigation with multiple setpoints. Path planning with these setpoints should allow for navigation to any location assuming that the terrain allows it. Demonstrating this is left to future work.

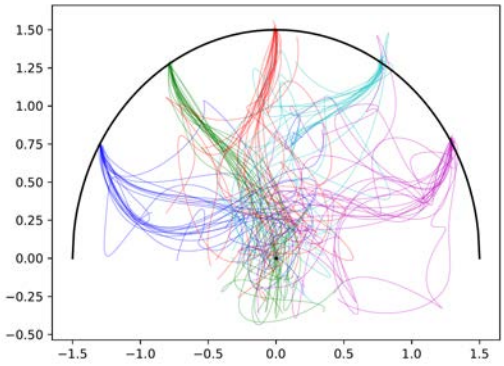
Other future improvements include increasing the communication rate for more precise control, and modelling latency rather than forcing a fixed delay. Modifying the reward function to encourage greater energy efficiency, speed, or precision could benefit specific use cases.



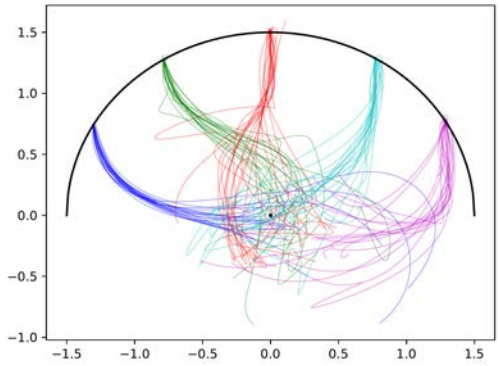
(a) Trained on flat ground, evaluated on flat ground.



(b) Trained on uneven surface, evaluated on flat ground.

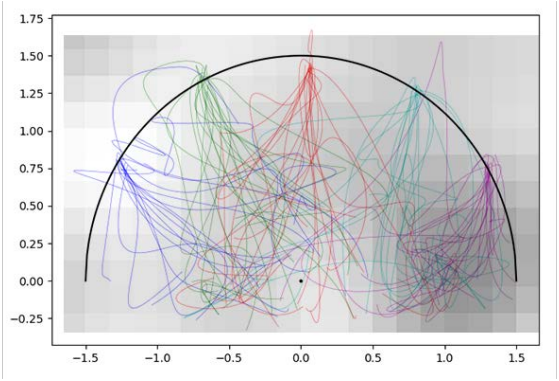


(c) Trained on flat ground, evaluated on uneven surface.

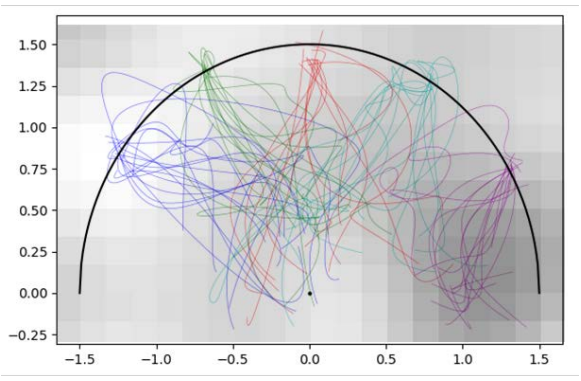


(d) Trained on uneven surface, evaluated on uneven surface.

Fig. 8: Robot trajectories on flat and uneven ground for real-world trials. All distances are in meters.



(a) Trained on flat ground, evaluated on valley-hill terrain. Darker areas represent valleys and lighter areas are hills.



(b) Trained on uneven surface, evaluated on valley-hill terrain. Darker areas represent valleys and lighter areas are hills.

Fig. 9: Robot trajectories on valley-hill ground for real-world trials. All distances are in meters.

VII. Acknowledgements

The authors would like to thank all the members of the Center for Distributed Robotics Laboratory for their help. This material is based upon work partially supported by the Corn Growers Association of MN, the Minnesota Robotics Institute (MnRI), Honeywell, and the National Science Foundation through grants #CNS-1439728, #CNS-1531330, and #CNS-1939033. USDA/NIFA has also supported this work through the grant 2020-67021-30755. The source code used for URDF generation is provided in a repository at https://github.com/MOLLYBAS/urdf_randomizer.

References

[1] B. Hemes, D. Fehr, and N. Papanikolopoulos, "Motion primitives for a tumbling robot," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sep. 2008, pp. 1471–1476.

[2] A. Schwartzwald and N. Papanikolopoulos, "Sim-to-real with domain randomization for tumbling robot control," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems*, May 2020, pp. 4411–4417.

[3] B. Hemes, D. Canelon, J. Dancs, and N. Papanikolopoulos, "Robotic tumbling locomotion," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 5063–5069.

[4] S. Dhull, D. Canelon, A. Kottas, J. Dancs, A. Carlson, and N. Papanikolopoulos, "Aquapod: A small amphibious robot with sampling capabilities," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2012, pp. 100–105.

[5] H. Sun, G. Song, J. Zhang, Z. Li, Y. Yin, A. Shao, J. Zhan, M. Xu, and Z. Zhang, "Design of a tumbling robot that jumps and tumbles for rough terrain," in *2013 IEEE International Symposium on Industrial Electronics*, May 2013, pp. 1–6.

[6] B. Hemes and N. Papanikolopoulos, "Frictional step climbing analysis of tumbling locomotion," in *2012 IEEE International Conference on Robotics and Automation*, May 2012, pp. 4142–4147.

[7] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Józefowicz, B. McGrew, J. W. Pachocki, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, "Learning dexterous in-hand manipulation," *CoRR*, vol. abs/1808.00177, 2018. [Online]. Available: http://arxiv.org/abs/1808.00177

[8] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, "Sim-to-real: Learning agile locomotion for quadruped robots," *CoRR*, vol. abs/1804.10332, 2018. [Online]. Available: http://arxiv.org/abs/1804.10332

[9] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," *CoRR*, vol. abs/1703.06907, 2017. [Online]. Available: http://arxiv.org/abs/1703.06907

[10] A. Maekawa, A. Kume, H. Yoshida, J. Hatori, J. Naradowsky, and S. Saito, "Improvised robotic design with found objects," in *NeurIPS Workshop on Machine Learning for Creativity and Design*, 2018.

[11] J. V. Albro and J. E. Bobrow, "Motion generation for a tumbling robot using a general contact model," in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, vol. 4, April 2004, pp. 3270–3275 Vol.4.

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[13] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[14] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12. MIT Press, 2000. [Online]. Available: https://proceedings.neurips.cc/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[15] L. C. Melo, D. C. Melo, and M. R. Maximo, "Learning humanoid robot running motions with symmetry incentive through proximal policy optimization," *Journal of Intelligent & Robotic Systems*, vol. 102, no. 3, pp. 1–15, 2021.

[16] D. Wierstra, A. Förster, J. Peters, and J. Schmidhuber, "Recurrent policy gradients," *Logic Journal of the IGPL*, vol. 18, no. 5, pp. 620–634, 09 2009. [Online]. Available: https://doi.org/10.1093/jigpal/jzp049

[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[18] B. Hemes, N. Papanikolopoulos, and B. O'Brien, "The adelopod tumbling robot," in *2009 IEEE International Conference on Robotics and Automation*, May 2009, pp. 1583–1584.

[19] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[20] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," *CoRR*, vol. abs/1502.05477, 2015. [Online]. Available: http://arxiv.org/abs/1502.05477

[21] OpenAI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang, "Solving rubik's cube with a robot hand," 2019.