2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 25-29, 2020, Las Vegas, NV, USA (Virtual)

Sim-to-Real with Domain Randomization for Tumbling Robot Control

Amalia Schwartzwald and Nikolaos Papanikolopoulos
{ schw1818 | papan001 }@umn.edu
Department of Computer Science and Engineering, University of Minnesota

***Abstract*— Tumbling locomotion allows for small robots to traverse comparatively rough terrain, however, their motion is complex and difficult to control. Existing tumbling robot control methods involve manual control or the assumption of flat terrain. Reinforcement learning allows for the exploration and exploitation of diverse environments. By utilizing reinforcement learning with domain randomization, a robust control policy can be learned in simulation then transferred to the real world. In this paper, we demonstrate autonomous setpoint navigation with a tumbling robot prototype on flat and non-flat terrain. The flexibility of this system improves the viability of nontraditional robots for navigational tasks.**

I. Introduction

In recent years, many different iterations of tumbling robots have been mechanically designed. As discussed in [1], tumbling locomotion has advantages over other methods such as increased mobility at small sizes and low levels of hardware complexity. These characteristics have allowed for tumbling robots to be designed for tasks such as underwater sampling [2] and jumping over obstacles [3].

Controllers for tumbling robots are notoriously complex, especially in diverse environments. Most utilize position control servos and assume certain friction coefficients or terrain properties. In [4], a set of motion primitives for locomotion across flat terrain is developed, with gaits defined to turn and move a robot forward. Climbing stepped terrain is discussed in [1] and [5], however no control policy is demonstrated. Previous work has shown that the motion resulting from an individual tumble is highly dependent on the surface the robot is in contact with, with friction and terrain slope limiting performance. A control policy that is adaptable and robust to diverse terrain would expand the capabilities of tumbling robots.

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 [6], [7], [8], [9]. Rather than attempting to perfectly replicate the real-world environment, parameters of the simulation can be varied through domain randomization. This variation results in a network with a greater ability to generalize making it more robust to differences between the real-world and the simulation [8]. With sufficient domain randomization, the real world is approximately a subset of all simulated environments. A prominent example where this methodology achieved success is [6] where a large set of randomized environments were used in simulation to real (sim-to-real) transfer.

Similarly, in [9], irregularly shaped branches were attached to servos, and movement was learned through sim-to-real.

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

This shows reinforcement learning's use in developing control policies for nontraditional robot designs, however, in that work tumbling motions were avoided as the robot was tethered. A locomotion policy was developed similarly in [7] for a quadruped. Tumbling robot motions were generated in [10], but are only tested with a simple robot in simulation, are only for movement in a 2D plane, and are not demonstrated in the real world. The present work explores the application of a reinforcement learning control policy learned in simulation to a physical tumbling robot platform. To the best of the authors' knowledge, this is the first use of reinforcement learning to control a physical tumbling robot, using the definition of a tumbling robot given in [1].

In the development of a tumbling robot sim-to-real pipeline, a variety of open-source libraries were used. The PyBullet [11] Python module provides physics simulations using the Bullet engine. It was also utilized in [7] and other work. The OpenAI Gym [12] provides a set of environments ready for applying reinforcement learning algorithms. It is also possible to create custom Gym environments for specific tasks and/or models, thus a custom tumbling robot Gym environment was developed. Stable Baselines [13] is a collection of reinforcement learning algorithms compatible with Gym environments. They are based on the OpenAI Baselines library [14]. Stable Baselines were utilized in this work due to their increased documentation and features.

The Stable Baselines implementation of Proximal Policy Optimization (PPO2) [15] was chosen as it allows recurrent policies, and flexible action spaces. A long short-term memory (LSTM) [16] policy was selected because sequences of motor commands are required for tumbles. In [6], PPO with an LSTM was used successfully in the task of manipulating cubes with a robotic hand. A recurrent policy can help with adapting to different environments, such as the transfer to

978-1-7281-6211-9/20/$31.00 ©2020 IEEE

real life.

New tumbling robots were designed to serve as testing platforms. They were inspired by the Adelopod [17] and use two single degrees of freedom legs attached to continuous rotation servos. No servo position feedback was utilized, and encoders were not added to provide it. ROS (Robot Operating System) [18] was used onboard to allow for a modular configuration.

After training the policies to convergence in simulation, their performance was evaluated in laboratory environments on the testing platform. One was also tested in a simulation with sim-to-sim transfer. The results demonstrate successful transfer and the feasibility of using this sim-to-real pipeline for tumbling robot locomotion. The control policy does not require servo position feedback and is not as sensitive to environmental changes as existing methods.

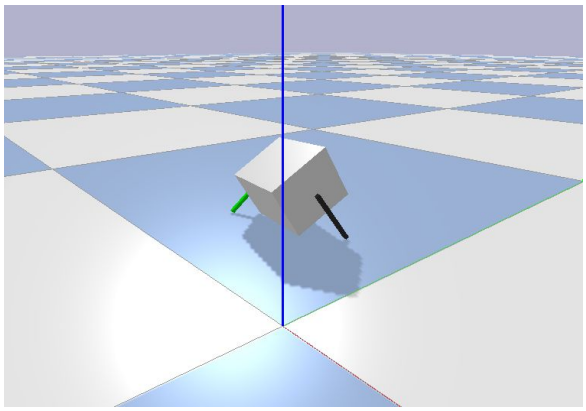

Fig. 2: Tumbling robot model navigating in Pybullet simulation.

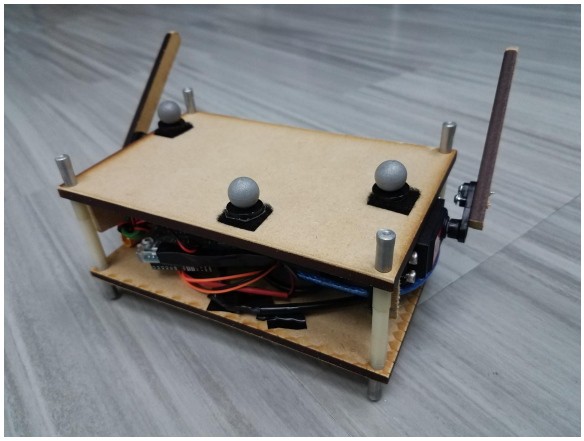

Fig. 3: Tumbling robot prototype.

II. Approach

A. Robotic Platform

1) Hardware: Two simple and low cost tumbling robot platforms were constructed. An image of the first prototype is shown in Figure 3. In [1], low leg friction with high body friction was found to increase performance in step climbing. 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

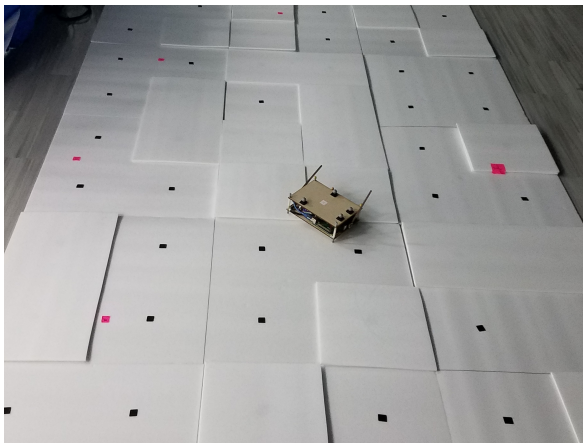

Fig. 4: Non-flat terrain constructed with randomly placed foam tiles. Each tile is 1 ft x 1 ft x 0.25 in.

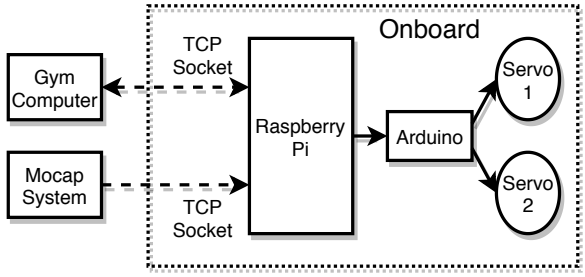

Fig. 5: Diagram of hardware components and connections. Dashed lines represent wireless communication, and solid lines represent wired communication.

yet provide sufficient torque and velocity at a low mass. An inertial measurement unit (IMU) was included, however, its data was not used. Rather, a Vicon motion capture system was used to obtain the pose. Use of filtered IMU data instead is left to future work. Hardware connections are shown in Figure 5. For the second prototype, the IMU was removed, and stronger servos were used along with a larger battery.

These robots differ from existing tumbling robots in their simple blocky design which allows for more robot parameters to be easily modified in simulation. Many Unified Robot Description Format (URDF) models can be generated programmatically, shortening simulation building time, and increasing the scope of domain randomization possible. This was performed with the second prototype. 100 models were generated, with the dimensions, inertia characteristics, and friction coefficients randomized with a standard deviation of 5% of the nominal values.

However, for the first prototype, the same crude model was used for all trials. It is shown in Figure 2. Some unintentional differences between the model and prototype were left uncorrected. The difference in mass was about 70 grams, and the asymmetry of servo positions was ignored.

2) Software: For both prototypes, 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. An Arduino Uno was utilized as a motor controller between the Pi and the servos to

allow for future expansion and control of different servos. The Raspberry Pi was chosen 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 IMU data would be integrated to approximate position, in place of mocap data.

The flow of data is shown in Figure 6. 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. The system was designed for a latency of about 20 ms, allowing theoretical rates of about 50 Hz.

The system was designed to 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 prototype tumbling robots' 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.

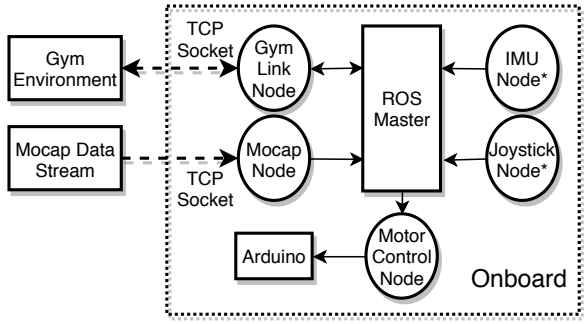

Fig. 6: Diagram of software communication. The Raspberry Pi is running ROS, and all communications are handled through ROS nodes. IMU and joystick were not used.

B. Policy Training

Training a policy in simulation rather than the real world decreases training time [8], 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 engine [11]. 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.

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 [13] used in training.

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 each of the physical prototypes were translated into simulation through URDFs, similar to the process described in [8]. The structure of the robot was broken 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 [9]. As described in Section II-A.1, these values were used as the mean for randomized generated URDFs for the second prototype. This randomization affects the dynamics of the tumbling robot's movement.

The prototypes' 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 a Gaussian noise, standard deviation of 0.1 N. At higher velocities, servo torque is set to 10 N. Servo velocity was also randomized, with a standard deviation of 1.25 rad/s for the first prototype, and 0.75 rad/s for the second. This randomization is significant relative to the commanded velocities, but was essential to having a robust policy.

2) Reinforcement Learning: Stable Baselines implementation of PPO [13] was used with an MLP LSTM policy. The learning rate was set to 3e-4, the same as was used in [6]. Nminibatches was set to 1. All other hyperparameters were left as defaults. A selection of values are shown in Table I. Training was performed on a Linux machine with an Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz and 32 GiB of memory. The network was trained to convergence, which occurred after 3 hours and 55 minutes of training for the training on flat terrain. Episode reward over time is shown in Figure 7. Training the second prototype's network was significantly slower due to non-flat terrain computations in the simulation.

PPO allows discrete or continuous action and observation spaces [13]. 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. The observation space is

continuous and includes the position of the center of mass, 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 [7]. Velocity shown in Figures 8, 9, and 10 was estimated post trials.

Adding noise to the output velocities, as described in Section II-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, it is predicted that unmodeled phenomenon is covered by these randomizations. For example, no battery model is used, but on the real robot, battery state affected 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 varying the starting location. 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 policy reward function rewards low velocity and proximity to the origin. The reward function is given by:

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

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 [7], may terminate an episode when the simulated robot falls. A failure state for the tumbling robot was chosen to be travel past a radius away from the goal location. 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.

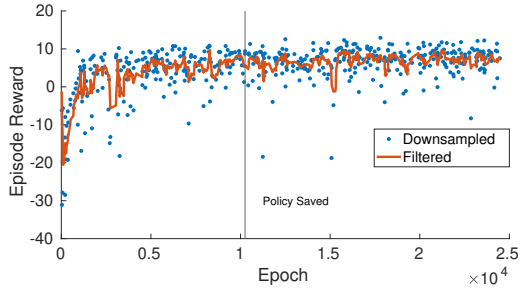

Fig. 7: Plot of the sum of rewards per episode for the first network, with flat terrain. Data has been downsampled to 500 samples, and filtered with a window size of 5. Network was trained for 3 hours and 55 minutes to convergence, then transferred to the first prototype robot.

Training	Testing	Setpoint Error [m]
Flat	Sim Flat	0.2381
Flat	Sim Flat Easy	0.1769
Flat	Sim Non-Flat 0.1	0.2859
Flat	Sim Non-Flat 0.2	0.4309
Flat	Sim Non-Flat 0.3	0.5851
Flat	Real World Flat	0.3302
Non-Flat	Real World Non-Flat	0.5994

TABLE II: Setpoint accuracy. Simulation with original training randomizations, simulation without servo randomizations, simulation without servo randomizations with rough terrain, real-world performance.

III. Experimental Results

After training the policies in simulation as described in Section II-B, their performance was evaluated in the real-world, and in variations of the simulation environment. With the first prototype, six consecutive trials were performed from five different starting positions. During one of the trials, a hardware error occurred, and localization data was not recorded. This trial is excluded. The trajectories of the remaining five trials (trials A-E) are shown in Figure 9. The average final horizontal distance from the setpoint was 0.3302 m. The most successful trial had a distance of 0.1485 m. Details of this trial are shown in Figures 1, 11, 12, and 13. With the second prototype, twenty trials were performed on non-flat terrain, with results shown in Figure 10.

As mentioned previously, a major weakness of existing control policies is their assumption of flat land. For the first prototype, training was performed in a flat simulation environment, with testing on flat real-world terrain and non-flat simulation terrain. Rough terrain was generated using an adapted version of PyBullet's terrain.py script. The wave height of each peak was multiplied by the square of a random sample [0, 1). This produces terrain that does not repeat and appears realistic. A larger wave height produces less navigable terrain. Wave heights of 0.1, 0.2, and 0.3 meters were used. For the second prototype, a custom tile-based terrain generation script was used, generating terrain similar the the real-world terrain shown in Figure 4. Both the training and real world trials were performed on this terrain.

Performance of the policies in a variety of different environments is evaluated. The results are shown in Table II. For the first prototype, the training environment was a simulation with flat terrain, while the testing environment varied. For the second prototype, both training and testing were on non-flat terrain, with training in simulation and testing in the real world. For prototype one, Sim Flat refers to the policy tested with the same simulation environment it was trained in. Trajectories from this are shown in Figure 8. Sim Flat Easy is an easier version of the training environment, where outputs were not randomized. This had the best performance. Sim Non-Flat 0.1-0.3 is the policy tested on randomized terrain without output randomization. Fifty trials were performed for each of the simulation testing environments, and the average setpoint error was calculated. The performance of the policy in the real-world on flat terrain is between that of Sim Non-Flat 0.1 and 0.2. Real-world trajectories are shown in Figure 9.

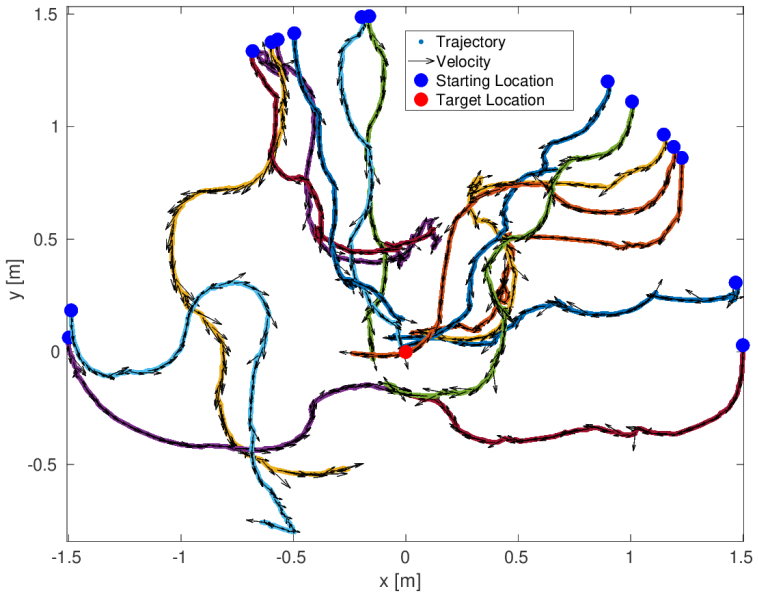

Fig. 8: Learned policy trajectories in simulation, from 15 random starting locations.

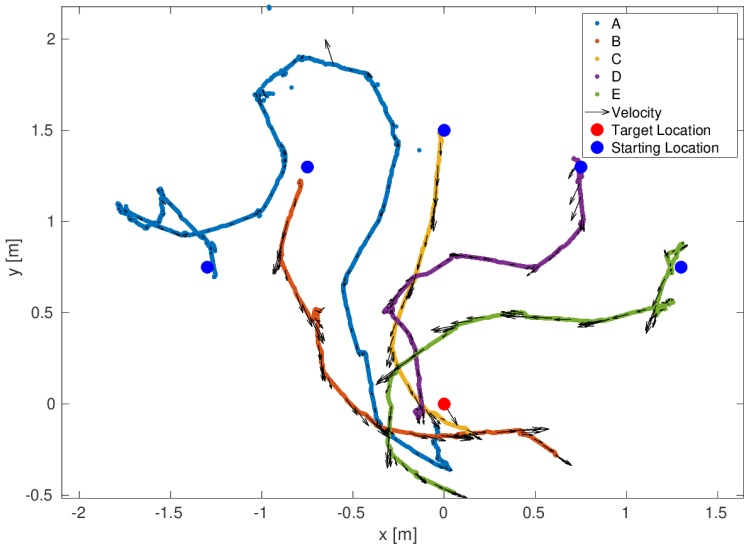

Fig. 9: Learned policy trajectories in the real-world, after transfer. Starting from 5 different locations.

With randomization, the trajectories were less precise at getting to the setpoint. As actuator output randomization increases, the probability of a given action being beneficial decreases. When near the setpoint, any overshoot decreases the reward received by increasing distance from the setpoint, and velocity. This lower precision can be observed in Figure 8. The randomization likely also contributed to reward function noise. Reduced performance in this case relative to a non-randomized domain indicates the domain randomization was effective in producing a robust model.

Figure 13 shows the height of the center of mass of the robot over time during trial D. In the plot, discrete tumbles, as defined in [1], can be observed. Each large peak, or major tumble, is composed of two smaller peaks, occurring when the center of mass is directly above the tumble axis. The falling edges of the large peaks are often irregular. This is caused by parts of the robot colliding with the ground, and becoming part of the support set. These collisions, which turn the robot, are difficult to predict without knowing the leg locations. The learned policy was able to overcome the lack of leg position feedback. This has the potential to reduce hardware complexity and cost.

Similar to a heuristic described in [19], a process of turning to face the target location, then driving in a straight line, was sometimes observed in simulation. Trajectories are often sets of arcs, created by one leg flipping the robot forward, and the other colliding with the ground and turning it.

Figure 14 shows the motor commands and position data for a trial performed on non-flat terrain. The motor com-

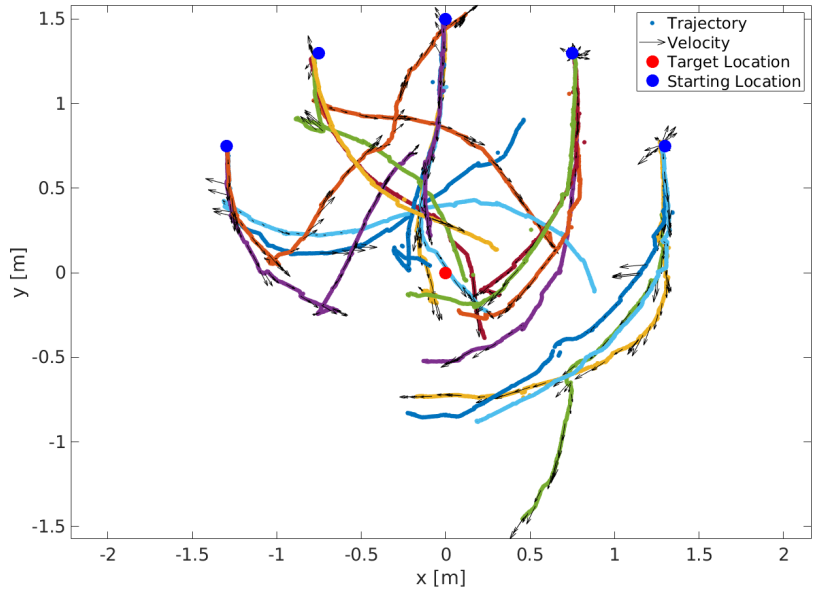

Fig. 10: Learned policy trajectories in the real-world on non-flat terrain, after transfer. 20 trials starting from 5 different locations.

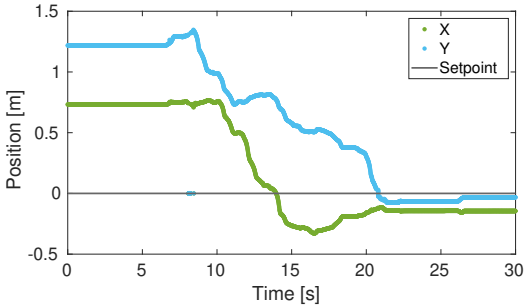

Fig. 11: X and y components of position over time, for trial D.

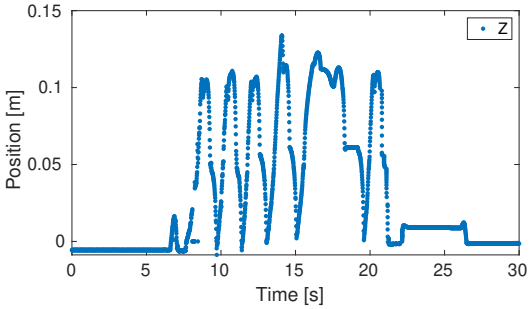

Fig. 13: Z component of position (height of the center of mass above ground, relative to the starting position) over time, for trial D.

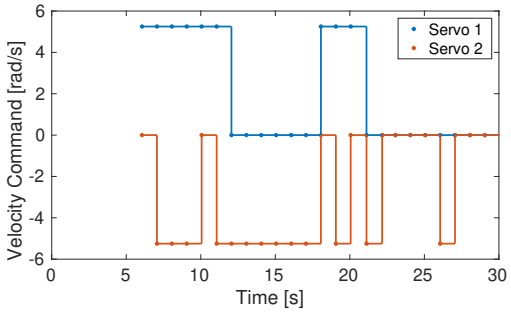

Fig. 12: Servo outputs over time, for trial D.

mands for the first five seconds would send a wheeled robot on flat terrain forward. However, the tumbling robot drifts in the negative x direction away from the setpoint before the policy corrects the trajectory. This shows that movement on the non-flat terrain with a tumbling robot is difficult to predict, and that navigation requires feedback for trajectory correction.

IV. Conclusion

This work demonstrates successful real world transfer of a control policy to a tumbling robot. It was accomplished with no training on real world data, only a crude simulation model. Minimal domain randomization was necessary to achieve successful sim-to-real transfer with a primitive robot, without the need for existing complex robot control schemes. This indicates the potential for training nontraditional robot designs in adverse environments using this method. This work is the first step in the development of a sim-to-real pipeline for locomotion.

V. 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 [2], would enable the robot to perform useful work.

While in this work only a small amount of domain parameters needed randomization for transfer to work in 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

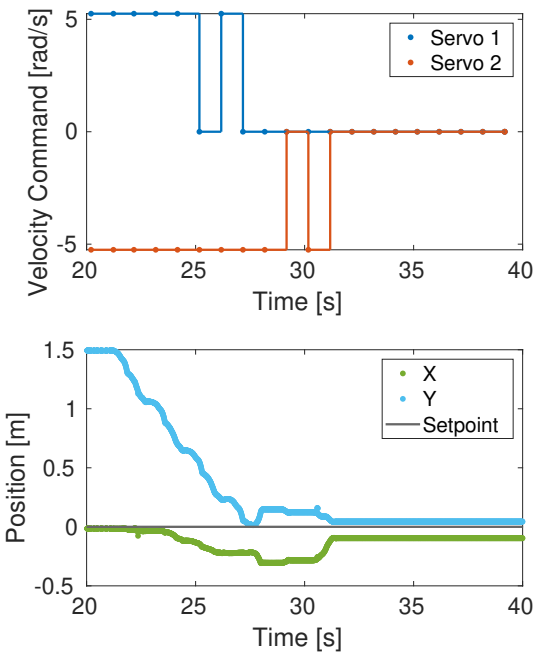

Fig. 14: X and y components of position and servo outputs over time for a trial on non-flat terrain. X position drifts but is corrected, indicating need for pose feedback in the system, and the unpredictability of tumbling robots on non-flat terrain.

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 [20]) 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. Implementing a PID controller that works with a tumbling robot may be challenging due to the system's unpredictability, but doing so would provide a better comparison for sim-to-real methods.

VI. 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. Canelon, J. Dancs, and N. Papanikolopoulos, "Robotic tumbling locomotion," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 5063–5069.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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

[7] 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

[8] 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

[9] A. Maekawa, A. Kume, H. Yoshida, J. Hatori, J. Naradowsky, and S. Saito, "Improvised robotic design with found objects," 2018.

[10] 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.

[11] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," http://pybullet.org, 2016–2019.

[12] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[13] A. Hill, A. Raffin, M. Ernestus, A. Gleave, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines," https://github.com/hill-a/stable-baselines, 2018.

[14] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines," https://github.com/openai/baselines, 2017.

[15] 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

[16] 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

[17] 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.

[18] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[19] B. Hemes and N. Papanikolopoulos, "A new modular schema for the control of tumbling robots," in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2009, pp. 5659–5664.

[20] 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.