Quadruped Locomotion with Central Pattern Generators and Deep Reinforcement Learning

Contents

1 Introduction

In this project we will explore two methods for quadruped locomotion. In the first part, we will implement different gaits with Central Pattern Generators. In the second part, we will design the Markov Decision Process (MDP) to be used by state-of-the-art deep reinforcement learning (DRL) algorithms. Note that the two parts can be completed in either order, but make sure to start early on part 2 to allow time to run multiple DRL experiments.

1.1 Code Structure

The code is hosted at `https://gitlab.epfl.ch/bellegar/lr-quadruped-sim.git`. Here we give details in addition to the `README.md`.

1.1.1 env

`env` contains all files needed for simulating the quadruped robot in pybullet. The structure is similar to the `leg` environment we used in Weeks 1-5.

- `quadruped_gym_env.py` contains the Gym interface where you will implement your MDP design decisions (i.e. state space, action space, reward function). **You will need to modify this file, see [TODO] tags**.
- `quadruped.py` implements many helpful functions for accessing various robot states, solving inverse kinematics, and computing leg Jacobians. **Please review this file carefully**.
- `configs_a1.py` contains configuration variables relating to initial positions, kinematics, joint limits, etc.

1.1.2 run_cpg.py and hopf_network.py

`hopf_network.py` provides a CPG class skeleton for various gaits, and you will need to map these to joint commands in `run_cpg.py` to be executed on an instance of the `quadruped_gym_env` class. **Please fill in these files carefully**.

1.1.3 run_sb3.py and load_sb3.py

`run_sb3.py` and `load_sb3.py` provide an interface to training RL algorithms based on stable-baselines3. You should review the documentation carefully for information on the different algorithms and training hyperparameters. However, feel free to use alternative reinforcement learning libraries.

1.2 Report Structure

Please format your report in the standard scientific format. This project is more open-ended than the previous project, so be sure to include all relevant parts: introduction, methods, results, discussion, conclusion. The assignment sections below will clarify specific questions you should include, but the overall contents should be self-contained and clear to someone who has not read this document. Be sure to cite any ideas, formulas, reward functions, etc. that you use from the literature. The target report length is 20 pages, with a max limit of 30 pages.

2 Quadruped Modeling

In this section we discuss the quadruped modeling used in pybullet. The quadruped has 4 legs, each with 3 joints (hip, thigh, calf), for a total of 12 motors. When querying joint states or sending torques to the motors, we thus have an array of length 12. The legs are numbered 0-3 in order Front Right (FR, 0), Front Left (FL, 1), Rear Right (RR, 2), Rear Left (RL, 3). The motor order is always hip, thigh, calf. See the slides for a clearer visualization.

In many cases, we will be interested in the location of the foot in the leg frame. As an example, let's consider the position of the front right (FR) foot at a "neutral" position (i.e. the foot is directly under the hip) when standing at 0.25 m. The foot position in the leg frame is then $(x, y, z) = (0, \text{-hip\_length}, -0.25)$ m. Please think about whether the other legs will have the same leg frame coordinates (why is y for FR negative? What are the coordinates for our simulation?). The functions `ComputeJacobianAndPosition` and `ComputeInverseKinematics` will return/expect this format.

On this note, recall from lecture the relationship between the joint angles $\boldsymbol{q}$ and foot position $\boldsymbol{p}$ for leg i:

$$\boldsymbol{p} = f(\boldsymbol{q}) \tag{1}$$

where the function $f(\cdot)$ denotes the forward kinematics of a single leg with respect to the leg frame. Differentiating this relationship gives the foot velocity $\boldsymbol{v}$:

$$\boldsymbol{v} = \boldsymbol{J}(\boldsymbol{q}) \cdot \dot{\boldsymbol{q}} \tag{2}$$

where $\boldsymbol{J}(\boldsymbol{q}) \in \mathbb{R}^{3\times3}$ is the single-leg Jacobian of the foot position w.r.t. the leg frame. The Jacobian can also be used to map a desired force $\boldsymbol{F}$ at the end effector (foot) to joint torques:

$$\boldsymbol{\tau} = \boldsymbol{J}^T(\boldsymbol{q})\boldsymbol{F} \tag{3}$$

These three relationships will be used repeatedly throughout this project.

2.1 Control

In the previous project we abstracted away the control for interfacing with Atlas' motors. In this project you will implement both joint PD control and Cartesian PD control. As a reminder from earlier practicals, for position control in joint space for leg i, we can compute the torques with the following formula:

$$\boldsymbol{\tau}_{\text{joint}} = \boldsymbol{K}_{p,joint}(\boldsymbol{q}_d - \boldsymbol{q}) + \boldsymbol{K}_{d,joint}(\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{q}}) \tag{4}$$

where $\boldsymbol{q}_d$ are the desired joint angles, $\dot{\boldsymbol{q}}_d$ are the desired joint velocities, and $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$ are the current joint angles and joint velocities, respectively. $\boldsymbol{K}_{p,joint}$ and $\boldsymbol{K}_{d,joint}$ are vectors of proportional and derivative gains. Please note the dimensions of each of these vectors.

To improve tracking performance, Cartesian PD controllers can be added (or used as an alternative, for example usually used in MPC [1]). From the desired joint angle ($\boldsymbol{q}_d$) and joint velocity ($\dot{\boldsymbol{q}}_d$) trajectories, we can extract desired foot positions ($\boldsymbol{p}_d$) and foot velocities ($\boldsymbol{v}_d$) in the leg frame. Given the current foot position ($\boldsymbol{p}$) and foot velocity ($\boldsymbol{v}$), the corresponding motor torques to track the desired quantities can be computed with:

$$\boldsymbol{\tau}_{\text{Cartesian}} = \boldsymbol{J}^T(\boldsymbol{q})\left[\boldsymbol{K}_{p,Cartesian}(\boldsymbol{p}_d - \boldsymbol{p}) + \boldsymbol{K}_{d,Cartesian}(\boldsymbol{v}_d - \boldsymbol{v})\right] \tag{5}$$

where $\boldsymbol{J}(\boldsymbol{q})$ is the foot Jacobian at joint configuration $\boldsymbol{q}$, and $\boldsymbol{K}_{p,Cartesian}$ and $\boldsymbol{K}_{d,Cartesian}$ are diagonal matrices of proportional and derivative gains.

3 Central Pattern Generators

In part 1 of this project we will implement our locomotion controllers based on Central Pattern Generators consisting of coupled Cartesian space Hopf oscillators [2–4]. The equations for oscillator i can be written as follows:

$$\dot{r}_i = \alpha(\mu - r_i^2)r_i \tag{6}$$

$$\dot{\theta}_i = \omega_i + \sum_{j=0}^{3} r_j w_{ij} \sin(\theta_j - \theta_i - \phi_{ij}) \tag{7}$$

where r_i is the current amplitude of the oscillator, θ_i is the phase of the oscillator, $\sqrt{\mu}$ is the desired amplitude of the oscillator, α is a positive constant that controls the speed of convergence to the limit cycle, ω_i is the natural frequency of oscillations, w_{ij} is the coupling strength between oscillators i and j, and ϕ_{ij} is the desired phase offset between oscillators i and j.

Here we will split the natural frequency ω_i between ω_{swing} and ω_{stance}, where the phase variable for oscillator i will dictate the stance/swing status of leg i. We will define swing phase (foot in air) as $0 \leq \theta_i \leq \pi$, and stance phase (foot in contact with the ground) as $\pi < \theta_i \leq 2\pi$. Make sure to use the mod operator (% in python) to keep your phase variables between 0 and 2π.

To evaluate the gait, we define the duty cycle as the time it takes to go through one full cycle of stance + swing phases. The duty ratio D is the duration of one cycle in which the foot is in stance phase T_{stance} over the duration of one cycle in which the foot is in swing phase T_{swing}, or $D = T_{stance}/T_{swing}$ [5].

There exist different methods for mapping the CPG states to joint commands. In this assignment we suggest mapping the states to Cartesian foot positions in the leg xz plane. This can be accomplished as follows:

$$x_{\text{foot}} = -d_{step} r_i \cos(\theta_i) \tag{8}$$

$$z_{\text{foot}} = \begin{cases} -h + g_c \sin(\theta_i) & \text{if } \sin(\theta_i) > 0 \\ -h + g_p \sin(\theta_i) & \text{otherwise} \end{cases} \tag{9}$$

where d_{step} is the step length (why do we need this?), h is the robot height, g_c is the max ground clearance during swing, and g_p is the max ground penetration during stance.

3.1 Assignment

By tuning the different parameters above, we can produce stable locomotion controllers for the following gaits: walk, trot, pace, and bound (please see the slides). In this assignment, you will need to follow the starter code in `hopf_network.py` to specifically implement/select:

- each of the coupling matrices ϕ_{gait}
- the swing and stance frequencies ω_{swing} and ω_{stance} (which are gait dependent)
- the mapping from the CPG states to desired quadruped foot positions
- the joint and Cartesian PD controllers to track these desired foot positions

Please include the following in your report:

1. A plot of the CPG states $(r, \theta, \dot{r}, \dot{\theta})$ for a trot gait (plots for other gaits are encouraged, but not required). We suggest making subplots for each leg, and make sure these are at a scale where the states are clearly visible (for example 2 gait cycles).
2. A plot comparing the desired foot position vs. actual foot position with/without joint PD and Cartesian PD (for one leg is fine). What gains do you use, and how does this affect tracking performance?
3. A plot comparing the desired joint angles vs. actual joint angles with/without joint PD and Cartesian PD (for one leg is fine). What gains do you use, and how does this affect tracking performance?
4. A discussion on which hyperparameters you found necessary to tune, the highest and lowest resulting body velocity you achieved, the resulting duty cycle/ratio, time duration of one step (time in stance/swing), and resulting Cost of Transport.
5. Two videos of the robot locomoting at different speeds named `GAIT_NAME_HIGH_XXms.mp4` and `GAIT_NAME_LOW_XXms.mp4`, replacing `GAIT_NAME` and XX with the average body velocity (after convergence) in m/s. At minimum, please show the `TROT` gait, but we encourage videos of other gaits as well (bonus). You can use the `record_video` flag.
6. Discuss how the controllers could potentially be extended with different feedback loops and with descending control signals for modulating locomotion (i.e. change of speeds, gaits, heading, foot placement). The actual implementation is left as a bonus (see Section A.3).

3.2 Tips

- Use the `on_rack` flag to `QuadrupedGymEnv` to freeze the robot base off of the ground. This is useful for visual debugging with low frequency gaits to ensure your coupling is as intended, and that your mapping from the CPG states to joint commands produce reasonable motions. You may also find it helpful to plot/visualize the desired foot trajectories to ensure they are reasonable.
- So far this framework results in open-loop commands sent to joints (i.e. there is no feedback). For keeping balance, we suggest starting with high frequencies during stance, and 2-4× higher for swing, depending on the gait. What are the slowest/fastest resulting gaits you can produce?

4 Deep Reinforcement Learning

In part two of this project you will design a Markov Decision Process (MDP) for the quadruped locomotion task, and then train control policies to solve it with state-of-the-art deep reinforcement learning algorithms. Implementing these algorithms is outside the scope of this course, so we provide an interface to two state-of-the-art algorithms, PPO [6] and SAC [7], with a popular reinforcement learning library (stable-baselines3) [8]. However, please feel free to do your own research and pick another library.

Here you will start from an OpenAI Gym [9] environment which defines an interface to your MDP for the quadruped robot. Notably, you need to determine what to put in the observation space, action space, and reward function. Please check `quadruped_gym_env.py` for the `[TODO]` tags to fill in. We give you a minimal example with the following:

- The observation space `DEFAULT` includes joint angles/velocities and body orientation. Note the upper and lower measurement limits you need to provide for the Gym interface. What other measurements are available on the real robot? What might help the agent choose better actions? **Important**: the observation space is normalized before using for RL. Why do you think this is? (Hint: what is the scale of different measurements, and how does this affect the ability of the network to learn)?

- The action space (joint) `PD` scales the policy network output from [-1,1] to joint positions within a smaller subset of the possible range. **Important**: having the policy network output actions in [-1,1] is currently standard practice in RL. Why do you think this is? (Hint: what is the scale of different actions we might take in general, and what is happening in the neural network?)

- The reward function `FWD_LOCOMOTION` currently has several terms to track a desired linear velocity while minimizing energy and other body velocities/orientations. Note that ultimately the reward is a single scalar, so weighting different terms properly is very important! The ultimate goal would be to create a smooth, omnidirectional locomotion policy that is energy efficient and capable of moving within a continuous range of forward, backward, and lateral velocities and yaw rates over varying terrains. You can start with training to achieve energy-efficient forward locomotion and consider other parts as an extension.

Fortunately, there are many papers using deep reinforcement learning-based control for quadruped robots, for example [2, 10–15]. It would be a good idea to check some of these papers (some have already been presented in class!) for inspiration.

4.1 Assignment

Here you will design the observation space, action space, and reward function to train control policies with off-the-shelf deep reinforcement learning libraries. Your report should discuss all of your design choices for the MDP, any modifications to the environment (i.e. sensor noise, dynamics randomization, adding rough terrain, resets), as well as justification of the learning algorithm hyperparameters you used. We provide hyperparameters for PPO [6] and SAC [7], however these may not be optimal and you should seek to understand the effects of these parameters on the algorithm and resulting policy. To summarize, develop and include the following in your report:

- Action space: since we have limited computing resources, we recommend to think carefully about how to strucure your action space. Some examples could include (1) joint position action space with joint PD control, (2) Cartesian space action space with Cartesian PD control, (3) learning to modulate CPG parameters and map these to joint commands (CPG-RL) [2]. We provide an interface for all three of these methods, and recommend starting with (3) for computational reasons. For the same training conditions (i.e. observation spaces, reward functions, RL hyperparameters), how do your training curves compare? Qualitatively, how do the resulting control policies look? Include plots and videos showing the performance.

- Observation space: what do you include? Why do you think each measurement was important? Did you see performance improvements with adding certain terms? How noisy would these measurements be on hardware? If using CPG-RL, consider placing the CPG states in the observation space.

- Reward function: what is your reward function? What terms do you include? How do you weight the terms to achieve a balanced result?
- Deep reinforcement learning: which algorithm(s) did you use? Which hyperparameters did you find important? What are some differences between the algorithms, and why/how did this affect the resulting control policy? At minimum please explain the provided hyperparameters and neural network architectures.
- Environmental details: what modifications do you make to the environment? How did this affect training and policy robustness?
- Discuss: how robust do you think your approach is, and will it transfer successfully to the real world (sim-to-real)? Discuss all potential difficulties you may encounter with the transfer to hardware.
- Quantify your resulting control policy: what is the average body velocity? Please include a plot of relevant robot states during the execution of one of your learned policies to go along with a video. What do you conclude about your reward function? Can you train a policy that you can command at test time to achieve any arbitrary speed (if not, discuss how you would do so)? How would/do you deal with uneven terrain? What is the Cost of Transport? Include the same discussion as in the CPG section relating to: resulting duty cycle/ratio, time duration of one step (time in stance/swing), etc.

4.2 Tips

- If you have installation issues or training is taking a long time on your local machine, you may want to consider resources such as Google Collab (free access to a GPU).
- During training you can monitor `ep_len_mean` (average episode length) and `ep_rew_mean` (average episode reward). The environment automatically resets after 10 simulated seconds, so a policy that does not cause the agent to fall should average 1000 for `ep_len_mean`. The average episode reward will depend on your reward function.
- You can periodically use `load_sb3.py` to check the training curves for `ep_len_mean` and `ep_rew_mean`, as well as load the most recent intermediate control policy being saved at the stated path. Training should converge relatively quickly (i.e. within 1 million time steps) in most ideal cases. If you are training for much longer without improvement you should check that your MDP is reasonable.
- Your resulting control policies may not look optimal. This is ok! Bloopers are encouraged. Why do you think the agent has converged to the policy you see, and what could you do to improve it?
- When reporting results, make sure to train several times, varying the random seed, and average the training curves. This is very important! Research has shown that this highly affects the results, see [16].
- Researchers currently use many techniques to try to ensure the agent learns a robust controller that will maximize its probability of success on a new environment. You may want to change the environment with some of the following possible ideas: domain randomization (adding/subtracting mass/inertia from various links), varying the coefficient of friction, varying the terrain (add boxes), adding noise to the observation, etc.

References

[1] J. Di Carlo, P. M. Wensing, B. Katz, G. Bledt, and S. Kim, "Dynamic locomotion in the mit cheetah 3 through convex model-predictive control," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 1–9.

[2] G. Bellegarda and A. Ijspeert, "CPG-RL: Learning central pattern generators for quadruped locomotion," *IEEE Robotics and Automation Letters*, 2022.

[3] L. Righetti and A. J. Ijspeert, "Pattern generators with sensory feedback for the control of quadruped locomotion," in *2008 IEEE International Conference on Robotics and Automation*, 2008, pp. 819–824.

[4] A. J. Ijspeert, A. Crespi, D. Ryczko, and J.-M. Cabelguen, "From swimming to walking with a salamander robot driven by a spinal cord model," *science*, vol. 315, no. 5817, pp. 1416–1420, 2007.

[5] H.-W. Park, M. Y. Chuah, and S. Kim, "Quadruped bounding control with variable duty cycle via vertical impulse scaling," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014, pp. 3245–3252.

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[7] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870.

[8] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable baselines3," https://github.com/DLR-RM/stable-baselines3, 2019.

[9] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[10] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, "Learning agile and dynamic motor skills for legged robots," *Science Robotics*, vol. 4, no. 26, 2019.

[11] J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter, "Learning quadrupedal locomotion over challenging terrain," *Science Robotics*, vol. 5, no. 47, 2020.

[12] 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," in *Proceedings of Robotics: Science and Systems*, Pittsburgh, Pennsylvania, June 2018.

[13] A. Iscen, K. Caluwaerts, J. Tan, T. Zhang, E. Coumans, V. Sindhwani, and V. Vanhoucke, "Policies modulating trajectory generators," in *Conference on Robot Learning*. PMLR, 2018, pp. 916–926.

[14] X. B. Peng, E. Coumans, T. Zhang, T.-W. Lee, J. Tan, and S. Levine, "Learning agile robotic locomotion skills by imitating animals," 2020.

[15] G. Bellegarda and Q. Nguyen, "Robust high-speed running for quadruped robots via deep reinforcement learning," *arXiv preprint arXiv:2103.06484*, 2021.

[16] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[17] M. Ajallooeian, S. Pouya, A. Sproewitz, and A. J. Ijspeert, "Central pattern generators augmented with virtual model control for quadruped rough terrain locomotion," in *2013 IEEE International Conference on Robotics and Automation*, 2013, pp. 3321–3328.

[18] M. Ajallooeian, S. Gay, A. Tuleu, A. Spröwitz, and A. J. Ijspeert, "Modular control of limit cycle locomotion over unperceived rough terrain," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013, pp. 3390–3397.

[19] D. Owaki, M. Goda, S. Miyazawa, and A. Ishiguro, "A minimal model describing hexapedal interlimb coordination: The tegotae-based approach," *Frontiers in Neurorobotics*, vol. 11, p. 29, 2017.

[20] D. Owaki, T. Kano, K. Nagasawa, A. Tero, and A. Ishiguro, "Simple robot suggests physical interlimb communication is essential for quadruped walking," *Journal of The Royal Society Interface*, vol. 10, no. 78, p. 20120669, 2013.

A CPG Extensions (Bonus)

In this section we list possible extensions to the above required assignment. **Note that no part of this section is required, but we leave it as a bonus if you would like to improve the performance of your controllers.**

A.1 Parameters

Most of the parameters in Equations 6-9 have been pre-tuned to work for all of the gaits, however, we encourage you to explore these further for better gait-specific performance. Such parameters involve the mapping from CPG states to desired foot shape such as:

- `ground_clearance`, foot swing height
- `ground_penetration`, foot stance penetration into the ground
- `robot_height`, nominal robot height
- `des_step_len`, step length
- `coupling_strength`, weights of w_{ij} matrix in Equation 7.
- α, convergence to limit cycle
- joint PD and Cartesian PD gains

Which of these do you change, and how does this help with performance?

A.2 Gaits

What are some other gaits you think you could produce with the same framework (i.e. pronk, gallop)? How about omnidirectional locomotion (i.e. going backwards, laterally, etc.)? What needs to be modified?

A.3 Virtual Model Control

Virtual Model Control and reflexes have been used in [17, 18] to improve performance and robustness at variable speeds, as well as the ability to turn and locomote over uneven terrain. Please check in the above cited papers, where some of the key items to add would be:

- attitude control (attach springs to body to keep orientation - reject pitch and roll variations)
- lateral angle of attack control (helpful over rough terrain)
- direction control (maintain or change body heading)
- stumbling correction reflex (when hitting an obstacle during swing phase)

You can test the robustness of your implementation by varying the friction coefficient of the ground, and adding random boxes of varying height to the environment.

A.4 Force Feedback in CPG Equations

Another suggestion for improving robustness is to add feedback directly into the CPG equations. One biology-inspired idea is to add terms based on the normal forces ("Tegotae") in the phase equations, with more details in [19, 20].