Simulating Snakes

William Corsel Wouter Stokman

May 25, 2021

1 Introduction

Over the years, snake robots have been researched extensively [1]. These robots try to emulate the natural movement of snakes using multiple joint actuators, giving them more degrees of freedom. This allows such robots to excel in the fields of exploration or search and rescue, where their slim size and versatile movement capabilities gives them an advantage over their traditional wheeled counterparts.

In this work, we simulate a snake robot using the PyBullet simulation engine [2]. We implement a movement function that simulates the lateral undulation locomotion often observed by real-life snakes. Furthermore, we embed our snake simulation into an OpenAI Gym reinforcement learning environment [3], and test the performance of the Proximal Policy Optimisation (PPO) algorithm on the environment to automate the movement function creation process.

2 Methods

This section describes implementation details for our snake simulation, the manual movement function, and the reinforcement learning environment.

2.1 Simulation Environment

We implement the snake robot using the PyBullet simulation library. This library includes a real-time physics engine to be used for games, robotics, or reinforcement learning applications. We develop a simple snake model to be used in this engine, consisting of a number of cubes, connected using revolute joints. The angle position of each of these joints can be set, after which the joint will turn until the position is reached with a constant applied force of 10N. This was done to mimic the design of a real-life servo motor.

To further improve the realism of the simulation, various PyBullet parameters were tuned. The gravity coefficients were set to $[0, 0, -9.81]$, anistropic friction coefficients to $[1, 0.01, 0.01]$, and lateral friction to 5. Each cube of the snake was assigned a weight of $60g$. We empirically found these values to allow the snake to move smoothly.

2.2 Manual Movement Function

Lateral undulation is an often seen movement pattern of real-life snake. This pattern is characterised by a wave-like movement propelling the snake forwards using the forces generated by the friction between the snake's body and the ground. We aim to mimic this movement for our simulated snake, by setting the positions of the joints based on a simple wave function. This is done by tracking a wave during the simulation run and mapping the location of each joint in the chain to a curvature based on the location

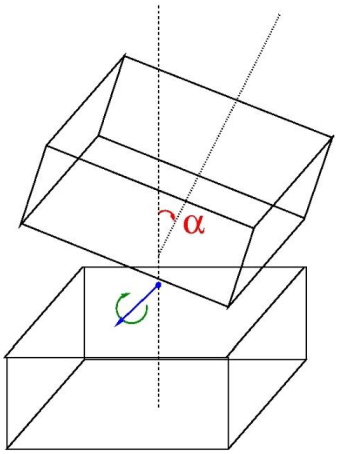

Figure 1: Revolute joint were angle position α can be set.

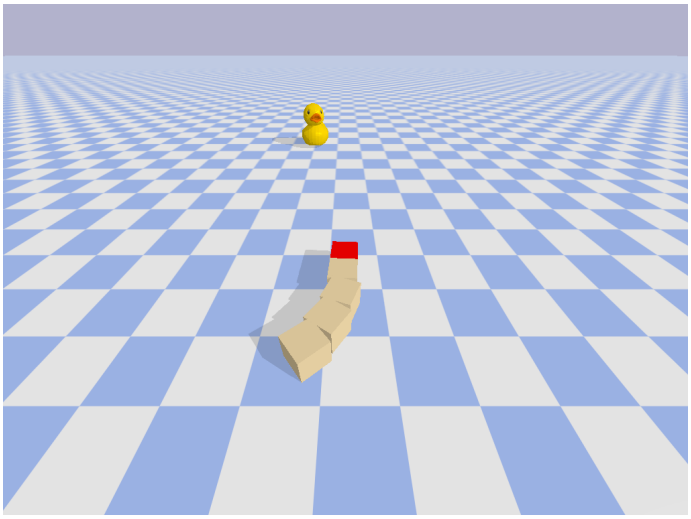

Figure 2: Snake simulation example.

of the wave. The joint angle α of the i^{th} joint can be calculated as follows:

$$\alpha_i(t) = A \sin(\omega \phi_i(t)) \tag{1}$$

$$\phi_i(t) = 2\pi \left[\!\left[\frac{t - (i+1)}{T} \right]\!\right] \tag{2}$$

with amplitude $A = 0.4$, period $T = 1.5$, and phase $\phi(t)$. $[\![\ldots]\!]$ denotes the fractional part of a number. We found that these values result in a smooth wave-like snake motion for snakes of lengths 5 and 10. However, we observed that in some cases, longer snakes (> 20 joints) can struggle to move smoothly.

2.3 Reinforcement Learning Environment

To automate the creation of snake movement functions, we embed the PyBullet snake simulation into an OpenAI gym environment. This widely used reinforcement learning (RL) framework allows us to apply a variety of RL algorithms to our problem. The aim of this environment is for the agent to learn how to move the snake robot to a certain goal. The environment consists of the following elements:

1. **Action-space:** The space from which the agent can sample its actions. For our environment, this consists of n-dimensional, continuous, vector of values between -1 and 1, where n is the number of joints in the snake.

2. **State-space:** Representation of the simulation-state. These variables are passed to the agent as an observation which is used to decide the next action to take. For our environment, this vector consists of the position of the snake head, and the position and velocity values for each joint.

3. **Reward function:** The agent is assigned a reward based on its current state, encouraging the agent to improve its state by applying the best actions. Our current environment implementation uses a simple reward function based on the distance to the goal:

   $$r = -\sqrt{(g_x - s_x)^2 + (g_y - s_y)^2}$$

   where g represents the coordinate of the goal, and s the coordinate of the head of the snake. We assign an extra reward of 100 if the goal is reached.

At the start of each simulation run, we apply a small random offset to both the starting position and angle of the snake.

2.4 Proximal Policy Optimisation

To investigate the created environment, we apply Proximal Policy optimisation (PPO) [4], a popular reinforcement learning algorithm. The main idea of this algorithm is to add a constraint to the surrogate objective function to limit the update size during gradient descent. The loss function of PPO is defined as follows:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right] \tag{3}$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} \tag{4}$$

where $\hat{A}_t$ is the estimated advantage function at timestep t, π a policy, ϵ the clipping factor, a an action and s a state. For more information on PPO, we refer to the original paper [4]. As our environment implementation is compatible with the OpenAi gym API, we utilise the Stable-Baselines3 PPO implementation [5]. This implementation uses a simple actor-critic model. We test the following two PPO architectures:

1. **PPO-small:** Uses the standard model architecture of two dense layers of size 64 for both the actor and critic models.
2. **PPO-large:** Increases the size of the PPO-small model by adding two shared layers of size 128 before the split layers.

3 Experiments

We investigate the performance of PPO on the snake environment. Both PPO variants described in Section 2.4 were trained for 20M timesteps on snakes of size 5 and 10. All other training parameters can be seen in Table 1.

Table 1: PPO training parameters.

Parameter	**Value**
Learning rate (Actor/Critic)	0.0003 / 0.0003
PPO steps	4096
Batch size	64
Epochs per update	10
Discount factor (γ)	0.99
GAE lambda (λ)	0.95
Loss Clipping (ϵ)	0.2

Figure 3 shows the average return the agent received when evaluating over 10 games during the training procedure. We see that for snakes of size 5, PPO manages to reach an average reward of -10000, which translates to the snake reaching the goal successfully in most cases. After investigating the behaviour of this agent after training, we found that it managed to mimic the lateral undulation we expected quite well, although its movements were less smooth than our manual method. We can see that the agent learned to orient itself successfully after its initial placement with a random orientation offset. It is also interesting to see that the last block is moved much more frequently compared to the others. This was found to propel the snake forward, similar to how a fish uses its tail to move forward (see Figure 4).

With longer snakes of size 10, we notice that the PPO agent does not manage to reach the goal after 20M timesteps. Furthermore, the larger PPO architecture did not lead to increases performance. A

likely explanation for the poor performance on larger snakes is the significantly increased complexity caused by the addition of more joints. Each additional joint adds extra variables to both the state and action spaces, making it more difficult for the agent to find a good mapping between input variables and the proper action to take, lowering the reached reward. The introduced complexity also makes it less likely for good action-sequences to happen when exploration takes place (e.g.: it is much easier to move forward with a small snake when moving randomly), making the training process more difficult.

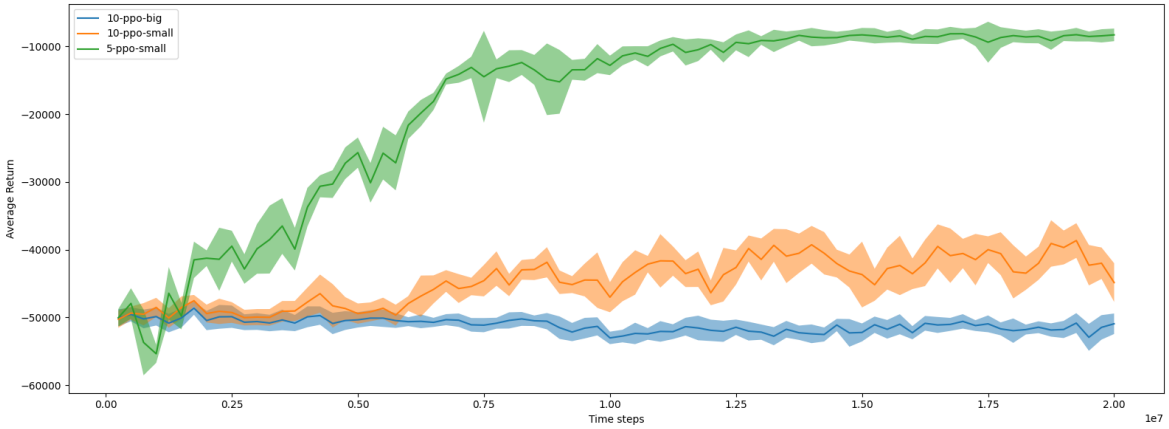

Figure 3: Average return over 10 evaluation runs of PPO agents during training.

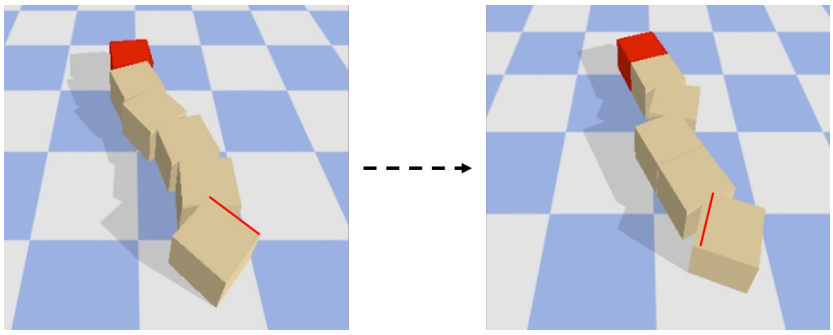

Figure 4: Movement of the trained *PPO-small* agent on a snake of length 5. The red line is used to indicate the angle of the block.

4 Conclusion & Future Work

In this work, we implement a simulated snake robot using PyBullet and implement a movement function that manages to mimic the lateral undulation often seen in real-life snakes. Furthermore, we adapt our simulation to an OpenAI gym environment and investigate the behaviour of Proximal Policy Optimisation (PPO). We found that PPO manages to learn how to move the snake towards the goal successfully for smaller snakes, but struggles when the snake grows in size due to the significant increase in complexity caused by the addition of extra joints.

Future research could look into further fine-tuning the reward function to give the agent a more specific task. Examples for this could be to add a smoothness reward based on the difference in position of the motors, discouraging the agent to move the positions of the motors too drastically at each step, smoothing out the robot's motion. The complexity of the larger snake models could be addressed by teaching the action network to use a simple sine function at initialisation, after which it can be trained to finetune this movement. This could allow the model to escape low-performance local optima. Another possibility for further research would be designing a more realistic snake model, and adapting the frictional behaviour of the Bullet engine to more closely reflect a real-life situation.

References

[1] Matthew Tesch, Kevin Lipkin, Isaac Brown, Ross Hatton, Aaron Peck, Justine Rembisz, and Howie Choset. Parameterized and scripted gaits for modular snake robots. *Advanced Robotics*, 23(9):1131–1158, 2009.

[2] Erwin Coumans et al. Bullet physics library. *Open source: bulletphysics. org*, 15(49):5, 2013.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. arXiv:1606.01540.

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

[5] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. https://github.com/DLR-RM/stable-baselines3, 2019.