Serpentile Locomotion

Farzad Abdolhosseini
University of British Columbia
Vancouver, BC, Canada
farzadab@cs.ubc.ca

Ainaz Hajimoradlou
University of British Columbia
Vancouver, BC, Canada
ainaz@cs.ubc.ca

ABSTRACT

UPDATED—May 6, 2019. Snakes are capable of moving on both steady surfaces and cluttered terrain while most models that use legs or wheels to obtain mobility perform poorly on cluttered exteriors. This project focuses on simulating the snake's motion by different techniques. First, different structures that can be used to construct snake's body are explored. Then, we try to create a decent motion using sine waves; and finally we use reinforcement learning, specifically policy gradient algorithms, to achieve controllable movement. The model is supposed to move toward a specified goal that is determined by a random objective. Different action encodings, including one based on Discrete Fourier Transforms, are introduced to make the task more feasible.

KEYWORDS

Locomotion; Modeling; Snake; Reinforcement Learning; Policy Gradients, DFT

ACM Reference format:
Farzad Abdolhosseini and Ainaz Hajimoradlou. 2019. Serpentile Locomotion. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Figure 1: Snake locomotion.

INTRODUCTION

The problem addressed in this work is to model a snake's movement. A snake mainly has four types of motion; Lateral undulation, Concertina locomotion, Rectilinear crawling, and Sidewinding. Lateral undulation is when various areas of the snake's body are pushing against a number of fixed points at the same time. This kind of motion is particularly used in desert or water. Concertina locomotion can be seen when a snake crawls through a tube which is done by making the back half of the body act as an anchor. Rectilinear crawling involves waves of bilaterally symmetrical muscle contractions and is commonly used by large snakes. Sidewinding is similar to concertina locomotion except that in this case, the angle of movement is 45 degrees.

We have only considered a Lateral undulation which is the most common mode of motion that snakes use to move through a wide range of environments. Cubes are used to build each part of the snake's body, and these parts are then connected through joints. By feeding a sinusoid to the body parts, a simple working model is constructed. Different results of the simulation are generated by other shapes like cylinders and also varying the sizes of the cubes.

In order to see if we can achieve a reasonable speed with the current model we run a grid search and optimize the movement by using a simple sinusoidal motion. As you can see in Figure 10, we achieve a speed of more than $0.78^{m/s}$ which is a promising result.

As the final step, we have used policy gradient for learning the movements. In each episode, we specify a randomly chosen objective and a corresponding reward function. This reward is based on the velocity and the distance between the center of mass and the goal. Goal has a high reward of 100, and in each step before reaching the goal, we receive a negative reward. We have defined several controllers (action encodings) for action space. These controllers are responsible for constructing the desired joint angles. In each observation, we find the position and orientation of the object in the space. At the start of each episode the environment is reset. The final results are then generated after 40 iterations on several controllers.

TOOLS

We choose Python programming language for our work, because of the vast range of available libraries and its ease of use. After creating the model, we need a physics-based simulation engine to test its different aspects. In order to do this, we use *PyBullet*. To achieve our goal of creating controllable motion, we use *RL-Lab* which is a framework for reinforcement learning containing the implementation of many state of the art algorithms in this field.

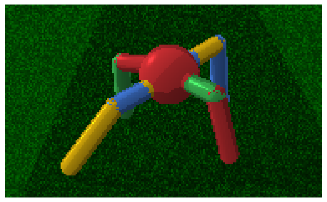

Figure 2: Ant Bullet environment

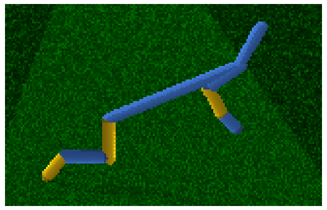

Figure 3: Half Cheetah Bullet environment

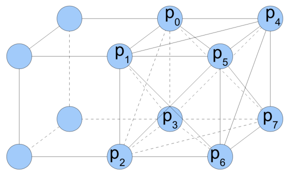

Figure 4: Part of snake PBD configuration for one body segment.

Pybullet

PyBullet is the official Python interface to the *Bullet Physics SDK* which for our purposes provides physics-based simulation and collision detection as well as rendering and visualization. You can use this engine either in a *GUI*, or a *DIRECT* mode which is useful when training or running computations on a remote server. One thing to note here is that the DIRECT mode does not make use of the GPU.

PyBullet can also be used for Virtual Reality and a suite of environments that can be used with *OpenAI Gym* for reinforcement learning. In addition, there are a set of pre-coded examples of training using these environments and some pre-trained models that can be enjoyed out of the box (See Figures 2 and 3).

Jinja Model Templates. To describe the model, Bullet lets us use different file formats like SDF, URDF, MJCF, and the native Bullet format. We chose to use the Universal Robot Description File (URDF) format which is used by the ROS project (Robot Operating System). This format lets us describe a robot using links plus joints that connect them. Unfortunately, this format is not really flexible and changing simple parameters like the number of links needs manual intervention. To remove this manual part, we use a templating language called Jinja2 to create the URDF files from Python, and then load these into PyBullet. This let's us create really flexible models using high-level manipulations.

Joints and Motor Control. PyBullet provides three different modes of motor-control with fixed position, velocity, or torque. The documentations clearly specify that the first two modes (fixed position and velocity) are implemented as constraints, but this might lead to multiple infeasible constraints which lead to physically impossible phenomena, such as a flying snake.

To avoid these issues, we use fixed torque control along with simple PD-Controllers. Getting these controllers to work when you have friction can be hard. The problem here is that if the applied force is too low, the force of friction will dominate and the joint won't move, but if it's too high, you might experience high velocities and therefore non-converging oscillations. To alleviate this problem, one can decrease the size of the time-step for the simulation, but the problem here was that by setting time-step below a certain threshold, the joint would not move at all. Unfortunately, we haven't been able to identify the cause of this problem, therefore, we found a time-step that worked and stuck with it.

RL-Lab

RL-Lab is a framework for developing and evaluating reinforcement learning algorithms. It includes a range of state of the art algorithms for continuous control. In addition, it is fully compatible with OpenAI Gym environments. Unfortunately, there are no official installation scripts or even manuals except for Anaconda, which still needs manual work for using this library every single time. We have tried to gather all the necessary requirements in a single *requirements . txt* file, but it still might prove to be incomplete.

To use this package, we had to create a Gym environment to be used as a test-bed for running the learning algorithms. By doing this, we now have a fairly standardized environment that can be used with already existing codes that use Gym environments. Therefore, it's also possible to contribute this model to PyBullet or OpenAI Gym to be used alongside humanoid, ant, and other models.

RELATED WORKS

There are generally different models to simulate a snake's body. [Miller, 1988] models the body with springs. Each part of the snake's body is modeled with a mass object and springs are used to connect those parts with each other. By writing the Newtons second law equations and dividing it by the total mass, the position of each part of the snake is obtained. For collisions, a simple normal and tangent reflection is used and the final position is updated. As a result, snake is capable of moving forward by oscillating its spring length as a function of time.

On the other hand, [Waszak, 2015] uses a Position-Based Dynamics (PBD) model. They use two different distance constraints instead of springs to simulate the body of the snake: strict distances and distance range. So each two body segments are connected by 14 constraints as shown in figure 3. Friction constraint is modeled by making use of the fact that the snake's friction properties are big when pushing in the lateral or backward direction and small when moving forward. Some External forces with sinosudial patterns are then applied to the head of the snake.

OUR MODEL

To make things simple, we use a model consisting of cubic links and each two consecutive links are connected using a hinge joint that rotates along the z-axis. We actually allow a gap between the links so that the joint have some room for movement[1]. The joint limits are set in a way that these two links can't collide, but collision between links that are further apart is still possible and should be handled by the simulator.

In order to drive the joints in our model, we use a sinusoid which is realized by the PD-Controllers. Denoting by $-L_i$ and L_i the lower and higher limits of the i'th joint, we set the goal of our controllers in time-step t to be the angle:

$$\theta_i(t) = L_i \cdot \sin(\omega \cdot t + \varphi \cdot i),$$

for pre-defined values of ω and φ.

Looking at this formula, one can see that for a fixed time-step t, different joints take the values of the sinusoid with a fixed difference in the phase. Then, with each time-step, the sinusoid "moves" along

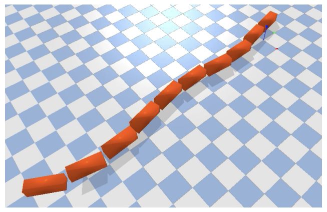

Figure 5: cube with larger width.

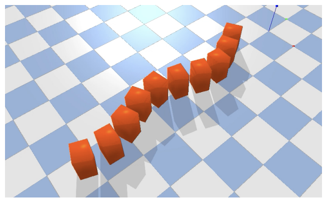

Figure 6: cube with larger height.

[1]In real life, this can be modeled with triangular shapes that meet in between, but we didn't see any reason to do such a thing at this point.

the body. It's also possible to show that under some assumptions, the body should also approximate a sinusoid at each time-step[2].

Different Body Parts

We tried experimenting with different body types for the model. We varied sizes of the width, length and height of the cubes. Having a bigger height resulted in an unstable snake movement and a snake with larger width didn't look natural. Larger length resulted in some interesting movements. Moreover, we experimented with different shapes such as cylinders. The snakes could move faster but they were sensitive to the sizes of the cylinders. One major problem was that cylinders could rotate easily which produced some strange behaviors. Figure 5 to 8 show different body types used for simulating snakes. Our model uses a cube with the same sizes for simulation.

Axes of Rotation

We have also experimented with multiple axes of rotation. The sinudoid is applied to both x and z axes. When having multiple axes, it is very critical to fully constrain the rotations. Otherwise, the model starts lifting up from the ground. We were able to achieve a stable model only when the rotation along z-axis was negligible.

Optimization

As we saw earlier, we can actually drive the snake with a sinusoid by using fixed variables for ω and φ. To get the fastest motion that we can achieve using these variables, we run a grid search to different combinations of these values. The results can be seen in Figure 10. There was a particular setting of these values in which the snake moves 4.7 meters in the opposite direction. This amounts to a speed of $0.78^{m/s}$, which is pretty decent. This result can be used as a benchmark for the reinforcement learning part.

ACTION REPRESENTATION ENCODINGS

Reinforcement learning is a hard task in and of itself, but when we are using learning algorithms, specifically policy search and policy gradients, the way we represent the observations and actions can greatly influence the results (like in [Merel et al., 2017]). For this reason, we represent the observations relative to the objects position. But for the actions, it is not clear which representation works best. So, we create several encodings, which we call low-level-controllers, some of which are only used as a reference for comparison and debugging.

Simple. This encoding is just an identity and is mainly used for comparison. Here, we simply have a single input signal per each joint and simply pass the input action to the designated joint.

[2]The idea is that, the joints angles are actually the second derivatives of the curve which we see as the body of the snake. So, if the second derivatives are sinusoids, then the actual function is also a sinusoid.

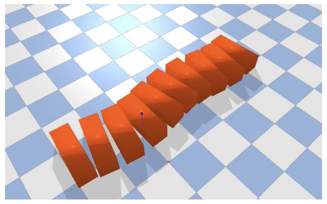

Figure 7: cube with larger length.

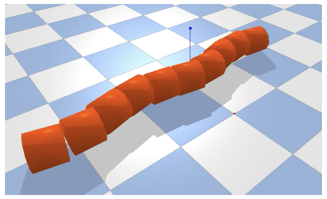

Figure 8: cylinder.

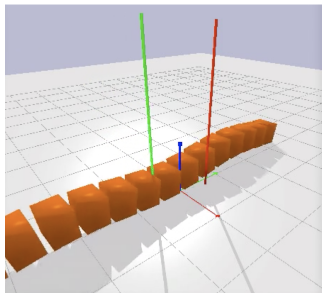

Figure 9: Rotation along multiple axes; green line shows the center of mass and the red line is the objective goal that the snake is trying to reach.

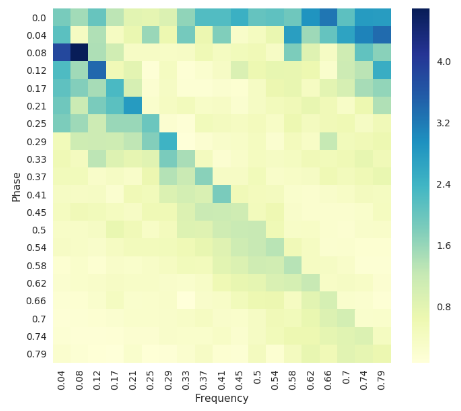

Figure 10: Distance traveled by the snake in meters by using fixed variables for the sinusoid in 6 seconds.

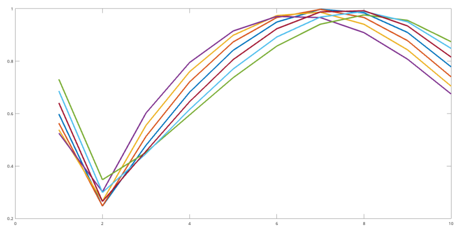

Figure 11: Shifting a signal with fractional values using DFT.

Sinusoid. This encoding only has two parameters, ω and φ, that we saw before. This is a really simple encoding and doesn't have a lot of variance, but we've already that it can actually achieve a really fast motion, which can be good reference for debugging the whole learning framework. One thing to note here is that, this controller actually needs the time-step t to work, and we will explore this idea further down the road.

DFT. We propose a sort of encoding scheme that has some nice properties and feels natural for the snake motion. The idea is to represent the actions in the frequency domain, and then use Fast Fourier Transform (FFT) to revert it back. By doing this, the range of motion will not be affected.

Just doing might be enough to make a difference, since changing even a single input, does make a change in the overall motion in a meaningful way. But, this is not the whole story. The second property that is useful here is that, we can actually decrease the dimensionality of the action space as much as we want, without losing much of our control over the possible motions. Even if the input space is as low as two or three dimensions, it can still be thought of as a sinusoidal controller which we already know is powerful. This property is really useful when the number of links in our snake model is high. You can think of it as creating the movement using a linear combination of sinusoids.

Phase-dependent Controllers. We have seen in the past that phase plays a critical role in locomotion tasks (see [Peng et al., 2017] and [Holden et al., 2017]). So, another idea that we wanted to experiment with is to actually change what an action does for each time-step based on the phase variable. Obviously, before doing this, we need to include the phase as one of the state variables, otherwise the states would not be Markovian. We actually included the phase in the state in other cases as well.

Doing this allows us to use the sinusoidal controller that we mentioned earlier. Also, this idea integrates nicely with the DFT controller. From basic properties of the DFT, we know that if we have a signal x and its corresponding transformed signal X, in order to shift the original signal by D we use an exponential as shown in the following equation:

$$x[n] \leftrightarrow X[k]$$
$$x[n-D] \leftrightarrow e^{\frac{-j2\pi kD}{N}} X[k]$$

Now, even if we use fractional values for the variable D on the right, we still get a nicely interpolated result for shifting x (see Figure 11). Using this, we can actually create action signals by which even a fixed action results in a nice motion.

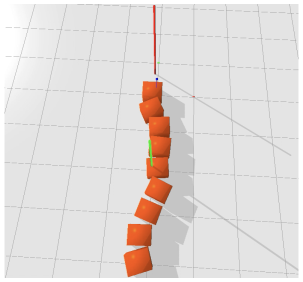
Figure 12: Trained model with DFT controller on 40 iters.

RESULTS

Reinforcement Learning. We have used PPO, the policy gradient algorithm, for reinforcement learning. We were able to train the model with batch size of 80000, max path length of 2000 and 0.999 for the discount value.

In each episode, the model is reset and a random objective is created within 5 to 7 meters from the snake's head position. Then the distance between the head position and the objective is calculated at each time step.

The reward of reaching the goal is considered 100. Any other point will be given a negative reward. The final reward is calculated by following equation:

$$r = -1 + r_{velocity} - p_{distance} \tag{1}$$

$p_{distance}$ is the distance penalty of the center of mass from the objective. $r_{velocity}$ is the velocity reward and is obtained by the dot product between velocity of the the center of mass and the goal.

Action space is then defined by several controllers as previously explained in Action Presentation Encodings section. Some captures of the trained models are shown in figures 12 and 13. Red line shows the objective and green line is the center of mass.

CONCLUSION

We construct a working model of a snake using a series of links and joints. Some experiments are conducted to obtain the most promising structure. Using the final model, we optimize over a finite set of movements and are able to achieve decent motion with a speed of $0.78^{m/s}$ in the opposite direction.

Afterward we move on to create a reusable environment that can make use of existing libraries of reinforcement learning algorithms. A simple reward mechanism based on both velocity in and distance to the objective is constructed. After some cleaning up, this environment can be made available for public use, and can even be integrated into the existing PyBullet project.

To make the task of learning more feasible, different action encodings are implemented that can create natural movements and have the ability to decrease the dimensionality of the input signal.

Finally, we use policy gradient algorithms to learn to move towards a specific randomized goal and try to use this as a measure of how successful each action encoding really is. Given the limited amount of time that we had, the results for this part are not yet conclusive, but we plan to spend more time on this in the future.

FUTURE WORKS

We are going to make the model more robust for future work as the parameters for PD controller and the time step are sensitive to initialization and some actions may result in strange behaviors. In

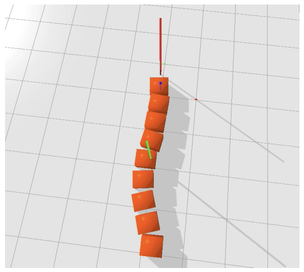

Figure 13: Trained model with rotating DFT controller on 100 iters.

addition, we have only considered one type of motion for the movement. We are going to train the model on different styles of motion such as sliding.

Besides, we have some major problems with the trained models. They are not efficient and the movement is very slow. One reason may be the fact that we were not able to train the model on many iterations. So, working on reinforcement learning will be one of our important objectives.

We are planning to test the model on different terrain such as ramps or sand. Moreover, one of our future goal is making use of the environment; building a model of the snake that is capable of moving by pushing itself to the objects.

REFERENCES

Daniel Holden, Taku Komura, and Jun Saito. 2017. Phase-functioned Neural Networks for Character Control. *ACM Trans. Graph.* 36, 4, Article 42 (July 2017), 13 pages. https://doi.org/10.1145/3072959.3073663

Josh Merel, Yuval Tassa, Dhruva TB, Sriram Srinivasan, Jay Lemmon, Ziyu Wang, Greg Wayne, and Nicolas Heess. 2017. Learning human behaviors from motion capture by adversarial imitation. *CoRR* abs/1707.02201 (2017). arXiv:1707.02201 http://arxiv.org/abs/1707.02201

Gavin S. P. Miller. 1988. The Motion Dynamics of Snakes and Worms. *SIGGRAPH Comput. Graph.* 22, 4 (June 1988), 169–173. https://doi.org/10.1145/378456.378508

Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel van de Panne. 2017. DeepLoco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning. *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)* 36, 4 (2017).

Bartlomiej Waszak. 2015. Snake Locomotion Using Position-based Dynamics. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games (i3D '15)*. ACM, New York, NY, USA, 136–136. https://doi.org/10.1145/2699276.2721401