A guide to creating basic physics simulations using Pymunk

Author: Jake Ward

1 Introduction

A goal for the 2020 machine learning subgroup was to provide useful documentation of key algorithms and packages important in both testing and implementation. More specifically, see the GitHub file `Environment.py` which refers to the Pymunk physics simulation developed to test machine learning algorithms quickly and efficiently using a step-based protocol similar to that of OpenAI's Gym. The intention is for future years to use this guide to understand the principles of Pymunk and advance the swing simulation created by the 2020 group.

2 Gym

OpenAI's Gym is *a toolkit for developing and comparing reinforcement learning algorithms.* It was found that the CartPole environment was best representative of the robot swing problem. This is because, unlike many of the other environments that guides will utilise, the CartPole is a *continuous* system. However the main difference between that and our system is that our system is naturally stable, whereas CartPole is naturally unstable. The main impact this has is that the only termination condition for the swing is a timeout, whereas CartPole is terminated when the system fails (i.e. the pole moves beyond 90 degrees). To learn how to set up a Gym environment see the documentation: '*Solving the CartPole problem with a Deep Q-Network*'. Gym uses *steps* to make an action and increment the physics simulation by a specified period of time on command.[1]. The `step(action)` method returns the list: `[observation, reward, done, info]`. The `observation` is the new state of the system, the `reward` is the current reward associated with that new state, `done` is a boolean set to `True` when the simulation is terminated and `info` provides additional information regarding the system. The advantage of using a time-step based interval system is that reinforcement learning algorithms can be implemented in a logical manor that relies very little on the simulation itself. Therefore we have decided to design the Pymunk simulation to replicate a Gym environment. Implementing various machine learning algorithms on these environments is heavily documented online.

[1]Much dissimilar to Unity that runs the physics engine in real time. This became a problem for the machine learning subgroup as the algorithms had to be changed to work much more closely with the simulation itself. This is doable, however for testing purposes provides unnecessary areas of error.

3 Pymunk

Pymunk is a relatively simple and well documented pythonic 2D physics library. It is built on top of the *Chipmunk* 2D physics engine written in C++. Alternatives to Pymunk include the *pybox2D* library, however it was decided to not use this as there is less online documentation. Installing pymunk to an Anaconda environment is done as follows.

- Open up Anaconda launcher.
- Open the 'Environment' tab.
- Create a new appropriately named environment.
- Open the anaconda prompt and type: `conda activate ENVIRONMENT_NAME`
- If this does not work type: `set PATH=C:\Users\USER\Anaconda3\envs\ENVIRONMENT_NAME \Scripts;C:\Users\USER\Anaconda3\envs\ENVIRONMENT_NAME;%PATH%`.
- The current directory should now look like: `ENVIRONMENT_NAME C:\Users\USER\SomeDirectory`.
- To install pymunk type: `conda install -c conda-forge pymunk`.
- If you also want pyglet type: `conda install -c conda-forge pyglet`.
- You will now need to install the other modules that you might need (Numpy, Matplotlib, Scipy etc.). Most of these should be found in the anaconda prompt in the environment tab.
- Make sure the correct environment is selected (green bar) and search for the necessary modules within the right-most tab.
- Select all required and press install. It will ask you to install some dependencies, do this.
- If the module is not present, you will have to install it in a similar way to Pyglet and Pymunk.

3.1 Bodies

There three types of Pymunk bodies. *Dynamic bodies* are the most common, and can react to collisions and are affected by forces and gravity. These have finite mass and are typically bodies you wish the physics engine to simulate for you. *Kinetic bodies* are similar to dynamic bodies, however are controlled by your code rather than by the physics engine. They aren't affected by gravity and they have infinite mass so they don't react to collisions or forces with other bodies. *Static bodies* are bodies that never (or rarely) move. They have infinite mass and do not need to react to collisions, providing a major performance boost over other body types.

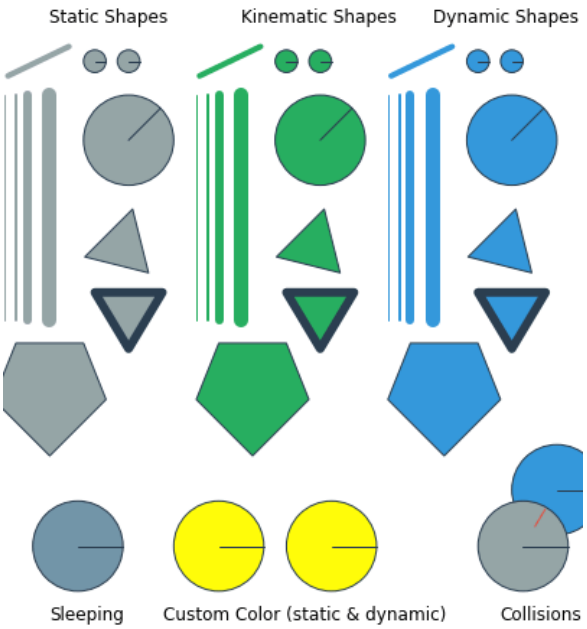

Figure 1: Different types of body that can be created using Pymunk.

3.2 Constraints

There are a few different types of constraints that provide a mechanical link between two bodies. These are best illustrated in this YouTube video: `https://youtu.be/ZgJJZTS0aMM`. The main joint used in our simulation is the *pivot joint*, that causes two bodies to pivot about a specified point. This point does not necessarily need to be on one of the bodies. Another joint commonly used in the swing environment is the rotary limit joint. This allows the relative rotations of two bodies to be constrained to certain minima and maxima. This was used to prevent certain body parts from moving beyond their physical limit.

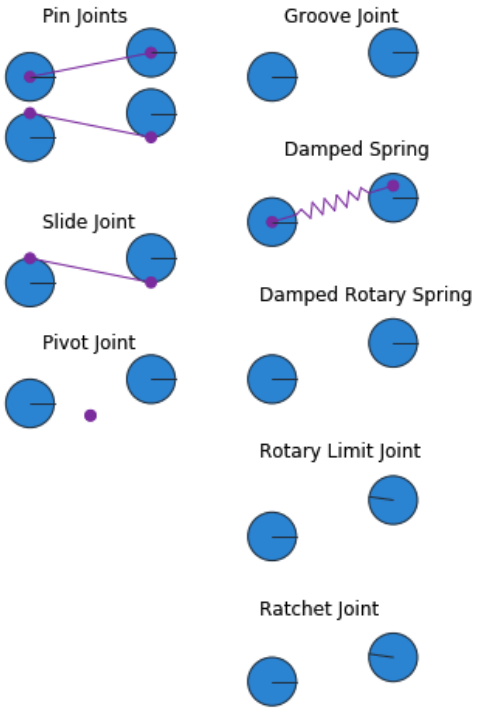

Figure 2: Different types of constraint that can be join two bodies together in different ways.

4 Code Structure

4.1 Defining a Swing

See the file `Environment.py`. This was created to allow swing-based Pymunk simulation to be interacted with as if it were a Gym environment. Future years may wish to make improvements to this simulation. The `Angle` class is purely used for convenience to allow certain key points to be defined in space based off an initial swing angle. Bodies and constraints can then be defined about these points. The `Swing` class is the main class that a ML algorithm would use to initialise the Gym-like environment. During initialisation the Pymunk space is defined with fundamental attributes such as gravity and damping.

The `create()` method is then called to generate a new swing / robot system in the Pymunk space. Each section of the swing requires two objects to be defined: a body and a shape. The body defines the sections mass and moment of inertia, as well as the type of body defined in Section 3.1. To avoid stability issues, the masses should be roughly the same throughout the simulation. The shapes are then defined for each section of the system. ShapeFilters are used to prevent all collisions for each section of the swing system. This is because collisions are not important in our domain, and negating them provides a performance boost. The positions of the bodies are defined using the `Angle` class described earlier. Pivots are then defined using the constraints defined in Section 3.2. These provide the connections between each part of the swing, and is the primary mechanism of which we wish to simulate. Finally the bodies, shapes and pivots can be added to the pre-defined Pymunk space.

The `delete()` method simply simply removes all bodies, shapes and pivots from the Pymunk space. The `reset()` method deletes the old swing system before creating a new system, which also resets the velocity and position variables. It returns the current state of the system, defined by the list: `[angle of main rod, velocity of main rod]`. This is useful for most ML algorithms, however future years may wish to alter how the state of the system is defined.

4.2 Making Actions

Actions in this environment are defined by an integer ranging from 0 to 4. This gives the agent 5 possible actions to choose from each time step, defined below.

0	Legs out
1	Legs in
2	Torso out
3	Torso in
4	Do nothing

In this simulation, the action is performed by applying a torque to the associated body part on the robot. The issue with this is that, on the Nao robot, a body part is moved by specifying a desired absolute angle and a time to reach this angle. Previously the `GearJoint` constraint was used to mimic this effect, however it was found that setting the angle between two bodies caused instantaneous movement and did not induce the desired torque. As a result it was decided to apply a torque directly to the body part. This may be an area for further improvement in this environment.

4.3 Giving Rewards

Rewards are derived from the *fitness function* you define. In this environment, the reward is defined as follows.

$$R = 2\theta^2 + v^2$$

The terms are squared to further incline the agent to achieve higher swing angles and velocities. The 2019 group simply used $R = \theta^2$, and rewarded the agent only at the apexes. The issue with this is that the agent often learns to create rapid motions of the main rod about $\theta = 0$, hence achieving many smaller rewards. As a consequence we decided to also reward the agent at $\theta = 0$ and include the velocity of the swing in the reward. The agent now learns to achieve faster swinging near the bottom point, hence encouraging more natural swinging throughout the domain. To further support this natural swinging behaviour, a penalty factor was introduced and deducted from the reward. This therefore functioned as a simple effort parameter that was set to 5 to actions that require motion, and set to 0 for the action where the agent does nothing.

4.4 Rendering the Simulation

Rendering the simulation is done using the Pyglet library. This is a very simple yet powerful python library that can be used to create games and applications. The `render(model)` method is used to generate a Pyglet window that simulates the Pymunk physics space environment. It does this by calling `pyglet.clock.schedule_interval` that runs the `update()` method at a fixed rate defined by the frame rate of your simulation. The `update()` method selects the action predicted to give the most favourable outcome (based on reward) according to the given neural network. It then steps the Pymunk space forward in time and draws the Pymunk space to the Pyglet window. Various text is included to provide information such as the current time, angle and velocity.