EXERCISE 4
Discretization and PID control

Tue Herlau
tuhe@dtu.dk

21 February, 2025

Objective: This exercise will introduce the rest of the control-toolbox which will be used in the course. It it is important to emphasize that this will not change how you specify models, which we looked at last week, but rather how you can take a model specification and either discretize it or turn it into an environment. Finally, we will take a look at PID control which is a simple but widely-used model free control method. (24 lines of code)
Exercise code: `https://lab.compute.dtu.dk/02465material/02465students.git`
Online documentation: 02465material.pages.compute.dtu.dk/02465public/exercises/ex04.html

Contents

1 Exam question: Euler discretization of control problem

We consider a control problem where a control signal $u(t) \in \mathbb{R}$ is applied to a variable $w(t) \in \mathbb{R}$. The variable measures an angle, and it satisfies the following differential equation:

$$\ddot{w} = \cos(u + w) \tag{1}$$

We introduce a state $\boldsymbol{x}(t) = \begin{bmatrix} w(t) \\ \dot{w}(t) \end{bmatrix}$ which allows us to re-write the system in the usual way as a first-order differential equation:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), u(t)) = \begin{bmatrix} x_2(t) \\ \cos(u(t) + x_1(t)) \end{bmatrix}$$

The problem is then discretized using Euler discretization with a time step of Δ to give states $\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \cdots$.

(a.) Assume that the initial conditions at the starting time $t_0 = 0$ is $\boldsymbol{x}_0 = \mathbf{0}$ and that no control signal is applied to the system ($u(t) = 0, t \geq 0$). According to Euler discretization, what is the value of the angle w at time $t_1 = \Delta$?

(b.) Continuing the previous problem and still assuming no control signal is applied. According to Euler discretization, what is the value of w at time $t_2 = 2\Delta$?

(c.) Assume the system is initialized in $\boldsymbol{x}_0 = \begin{bmatrix} \frac{\pi}{2} \\ 0 \end{bmatrix}$ and we apply a constant control signal u_0 to the system. What is $\boldsymbol{x}_1$ as a function of u_0? Explain what the value of angle, w, at time t_1 mean in terms of the accuracy of Euler-discretization.

2 Exam Question: PID

Consider PID control applied to steer a car along a straight track. The control signal u_k corresponds to the angle between the front wheel and the centerline of the track, the input signal x_k corresponds to the angle between the car body and the track in degrees, and the goal of the PID controller is to bring the angle between the car body and the track to a value of $x^* = 4$ degrees (corresponding to executing a turn). Figure 1 shows the behavior of both x_k and u_k at time steps $k = 0, 1, 2, \ldots$. Suppose the PID controller takes the form described in the lecture notes, and assume $K_d = K_i = 0$, which one of the following options are true?

a. $K_p = 1$

b. $K_p = 2$

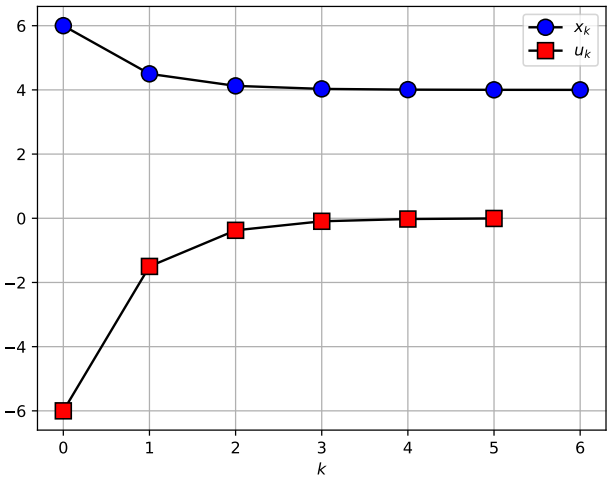


Figure 1: Plot of PID controller

c. $K_p = 3$

d. There is not enough information to determine the correct answer

e. Don't know.

3 Models, simulation, discretization and environments (`discrete_kuramoto.py`)

This exercise will be a continuation of the Kuromoto toy problem from last week where the focus will be on

- Creating a discretized model
- Turning it into an environment

3.1 Recap of the Kuramoto toy problem

Assume that $x(t) \in \mathbb{R}$ and $u(t) \in \mathbb{R}$ are one-dimensional. The Kuramoto oscillator is defined by the following differential equation:

$$\frac{dx(t)}{dt} = u(t) + \cos(x(t))$$

If we write this in our standard notation it looks as follows:

$$\dot{x} = f(x, u) = u + \cos(x). \tag{2}$$

We will assume that the cost function is just:

$$\{\text{cost}\} = \frac{1}{2}\int_0^{t_F} u(t)^2 dt$$

and that the system is subject to the constraint that $-2 \leq u(t) \leq 2$. The next sections will show how to implement and discretize this model.

3.2 Discretization

Discretization occurs by simply passing a model instance to the `DiscreteControlModel` class along with a discretization time step `dt` ($= \Delta$ in our notation):

```
# discrete_kuramoto.py
dmodel = DiscreteControlModel(KuramotoModel(), dt=0.5)
print(dmodel) # This will print details about the discrete model.
```

The class will automatically discretize the environment and create the function f_k such that $x_{k+1} = f_k(x_k, u_k) = x_k + \Delta f(x_k, u_k)$.

The online documentation contains an example of how to use this class, and this exercise will allow you to replicate the key steps using the Kuromoto environment – so use the online documentation in the following!

Problem 1 *Discrete model: Implement f_k*

Use the discrete model to compute the Euler-discretized step function f_k as the function `def fk(x,u)`.

Then implement the function `def dfk_dx(x,u)` which computes the Jacobian derivative with respect to $\boldsymbol{x}$:

$$\frac{\partial f_k(x,u)}{\partial x}$$

Info: If you are stuck with the first part, look at `https://02465material.pages.compute.dtu.dk/02465public/exercises/ex04.html#example-creating-a-discrete-pendulum` I.e., create a discrete model class and simply use the function `dmodel.f`, which evaluate f_k.

If you are stuck with the second part, look at `https://02465material.pages.compute.dtu.dk/02465public/exercises/ex04.html#computing-derivatives-jacobians-of-t`

When done, you should get this output.

```
The Euler-discretized version, f_k(x,u) = x + Delta f(x,u), is
f_k(x=0,u=0) = [0.5]
f_k(x=1,u=0.3) = [1.42015115]
The derivative of the Euler discretized version wrt. x is:
df_k/dx(x=0,u=0) = [[1.]]
```

Try to verify the last result by manually computing the derivative.

3.3 Part 2: Creating an environment

Our final task will be to create an environment. The environment requires a discrete model, and will automatically define a `reset` and `step` function. It can be specified as (see online documentation):

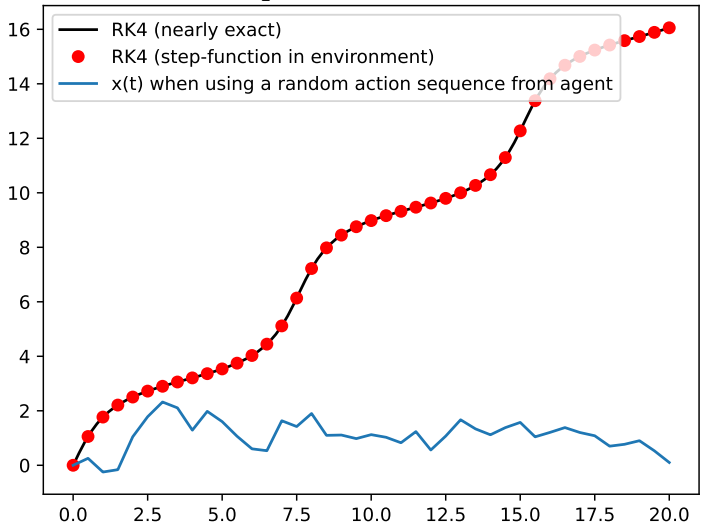
```python
# discrete_kuramoto.py
dmodel = DiscreteControlModel(KuramotoModel(), dt=0.5)
env = ControlEnvironment(dmodel, Tmax=20) # An environment that runs for 20 seconds.
```

The `step` function will, as usual, accept an action, and then simulate the model using RK4 (i.e., the code you just wrote) over a time period of `dt`. The environment will also track the current time as `env.time`.

Problem 2 *Simulate system using the step-function*

Complete the code to simulate the trajectory x_k over 20 seconds when using the `step`-function in the environment and a constant action $u = 1.3$.

Info: The code is nearly complete and again the online documentation contains plenty of hints. The point of the exercise is to familiarize you with how you can create a Control environment, and note that the outcome of the environment agrees with the high-order RK4 simulator we looked at last week – this is of course not a coincidence, as the step-function uses RK4! When done, you should get the following:



You should now be all set: You have defined a continuous-time variant of the environment, an Euler discretized variant, and an environment which is consistent with both. You can now build agents that uses either of the models to plan in your environment!. As an example, this is how you can use a random agent:

```python
# discrete_kuramoto.py
stats, trajectories = train(env, Agent(env), return_trajectory=True)
plt.plot(trajectories[0].time, trajectories[0].state, label='x(t) when using a random action sequence from agent')
```

4 PID Control

This section will consider PID control discussed in algorithm 19. The basic PID method accepts a single number as input and outputs a single number. We will implement this functionality as a class so it can later be reused.

4.1 PID and the locomotive environment (`pid.py`)

Our first example will be the locomotive-example from the notes. Since PID is a model-free method, you will only need to interact with the `def step(action)` function as usual, which will simulate the model exactly using RK4. You job is to implement a class, `PID`, which has a single function `PID.pi(x)` which takes a state as input, and return the action computed using the PID update formula. In other words, this is how you use the class to compute an action:

```
# pid.py
env = LocomotiveEnvironment(m=70, slope=0, dt=0.05, Tmax=15)
pid = PID(dt=0.05, Kp=40, Kd=0, Ki=0, target=0)
# Compute the first action using PID control:
print(f"When x_0 = 1 then the first action is u_0 = {pid.pi(x=1)} (and should be
↪  u_0 = -40.0)")
```

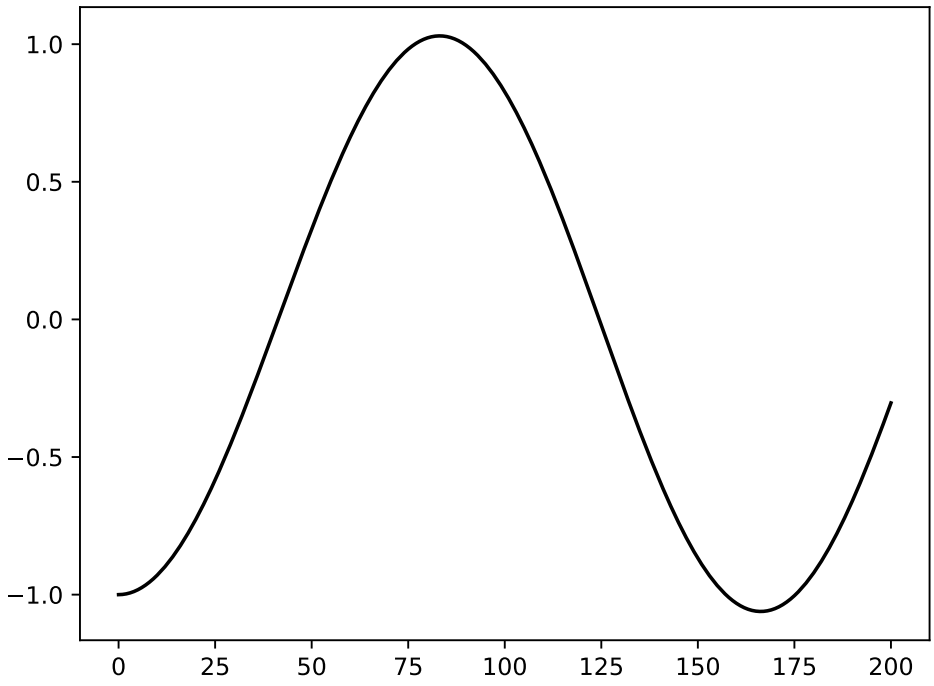
and this result in:

```
When x_0 = 1 then the first action is u_0 = -40.0 (and should be u_0 = -40.0)
```

Problem 3 *Bare-bones PID and locomotive*

Read the code and complete the implementation of the PID class and use it to compute the action. We clip the actions since the locomotive is not infinitely powerful.

Info:



Since this is a proportional controller, the locomotive will oscillate around the target at 0.

4.2 PID and the locomotive environment (`pid_locomotive_agent.py`)

Let's turn the basic PID class into a proper agent. This will allow us to re-produce the locomotive results

Problem 4 *PID agent and the locomotive environment*

Implement the PID agent. The computation of the actual action should be delegated to the `PID` class. When done, inspect the experiments and discuss the results.

Info: The code will also run an animation of a small train. You can turn these off by setting `render_mode=None`.

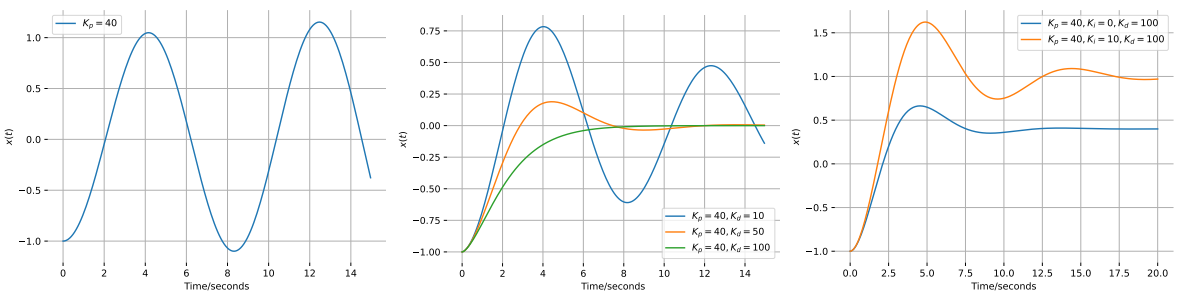


Figure 2: Illustration of pendulum system

4.3 Pendulum balancer (`pid_pendulum.py`) ✸

Our final example relates to the pendulum problem (see fig. 2). Our goal is to create a PID agent which can stabilize the pendulum in both an upright position as well as an angle. This is accomplished by applying torque.

To make the problem more interesting we start the problem in an imbalanced position, obtained by initializing the pendulum as standing upright and applying a small force $u > 0$ for a few steps. The more steps, the more difficult the subsequent balancing task.

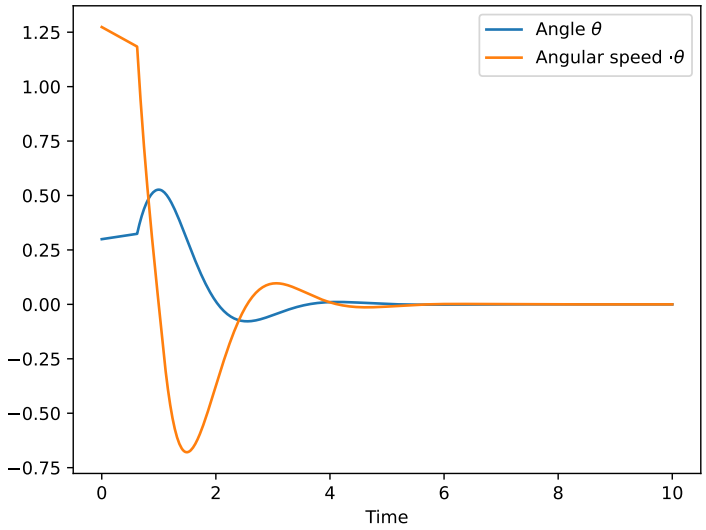
- `x[0]` : θ, angle. $\theta = 0$ is upright.
- `x[1]` : $\dot{\theta}$, angular velocity.

I have split the problem into three progressively more difficult tasks, but you should only implement a single agent and call it with the same set of parameters. As for how to find the right parameters and inputs to the PID Agent: You simply have to guess and use your intuition (that was what I did), and look forward to next week when we will look at model-based controllers which will require less guesswork.

Problem 5 *Implement the pendulum balancer in the simplest version*

Implement the pendulum balancing agent where the goal is simply to bring the pendulum angle to $\theta = 0$.

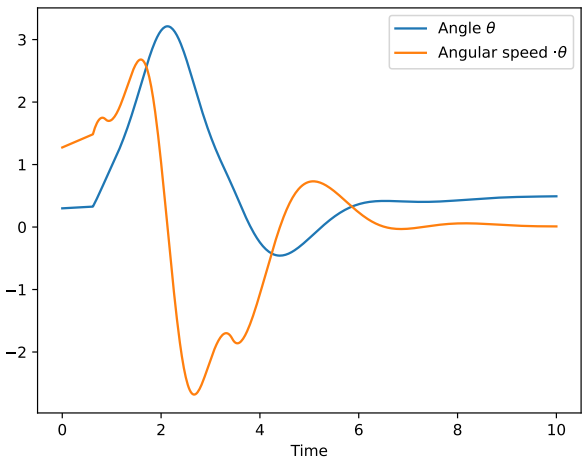
Info: As a hint, for this question, it was sufficient to use `x[0]` as input for this part of the problem. My implementation used K_d and K_p but not K_i. Remember to apply action clipping. `u = np.clip(u, min_val, max_val)` You can get the minimum and maximum values from the action space (see online documentation).



Problem 6 *Balance at an off-center angle*

Expand your implementation above to balance the pendulum at an angle different than zero. $\theta^* \neq 0$.

Info: Solve this by tuning your previous implementation. The target should be θ^*, but at this position, gravity will pull the pendulum down – this is similar to a problem we saw with the train. How did we fix it? You may need to tune the PID parameters a bit to make the pendulum work.



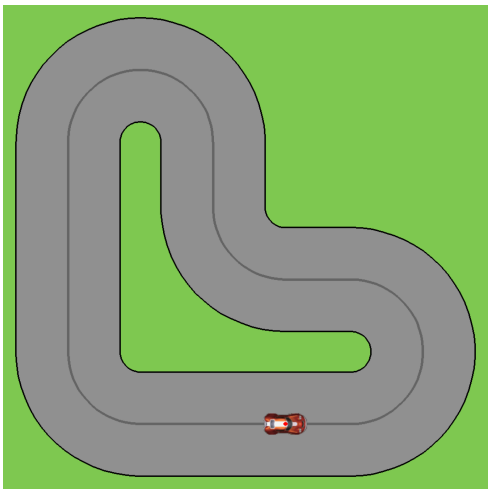


Figure 3: Screenshot of the car environment. Recall the cars action space consist of the steering angle and throttle

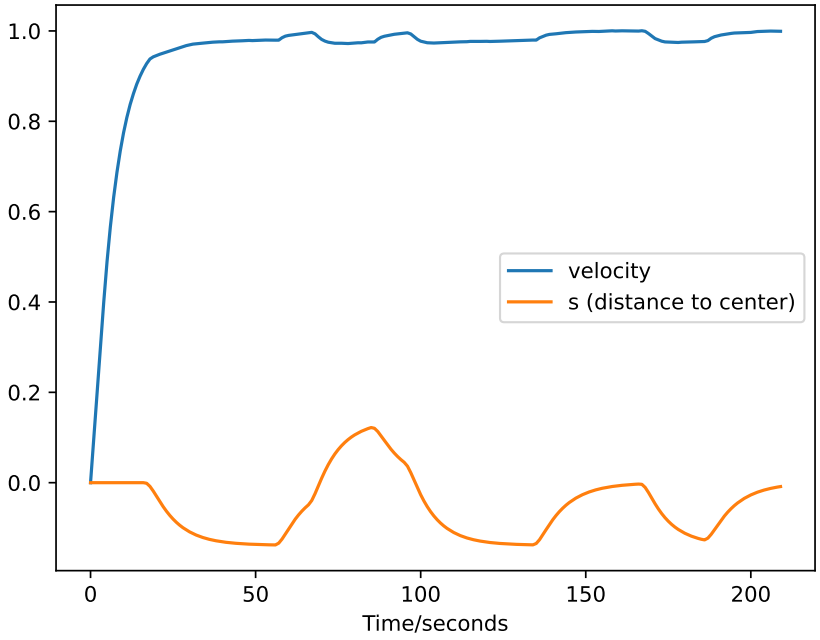
5 PID Racecar controller (`pid_car.py`)★

Implement the PID racecar example from the lecture notes (see also fig. 3). You need to set up two pid controllers and tune their values; viewing what the car does will tell you if you are on the right track.

Problem 7 *Implement racecar controller*

Implement the PID car agent. Nearly all functionality is delegated to the two PID controllers (one for velocity, one for car angle) that you set up, but you need to tune their parameters.

Info: Look at the animation to see if the behavior is approximately correct (too fast, too slow, over-steering, etc.). When you run the code, the car will explain what the coordinates of the state and action vectors mean.



5.1 Moon landing (`pid_lunar.py`) ★★

This is a more open-ended (and challenging!) problem which is partly inspired by the apollo lander, which used a custom-build controller that used techniques similar to PID, see `https://eli40.com/lander/02-debrief/`.

Our particular example will use the much simpler Gym environment, and in fact the implementation is a re-worked version of `https://github.com/wfleshman/PID_Control/blob/master/pid.py`. In other words, this link contains the correct PID controller, and your job is simply to translate the control rules in the code to a standard formulation of PID you can implement.

Problem 8 *Implement the lunar-lander module*

Implement the PID lunar lander. I have selected a different set of parameters from the reference implementation but asides this change the two implementations should be equivalent.

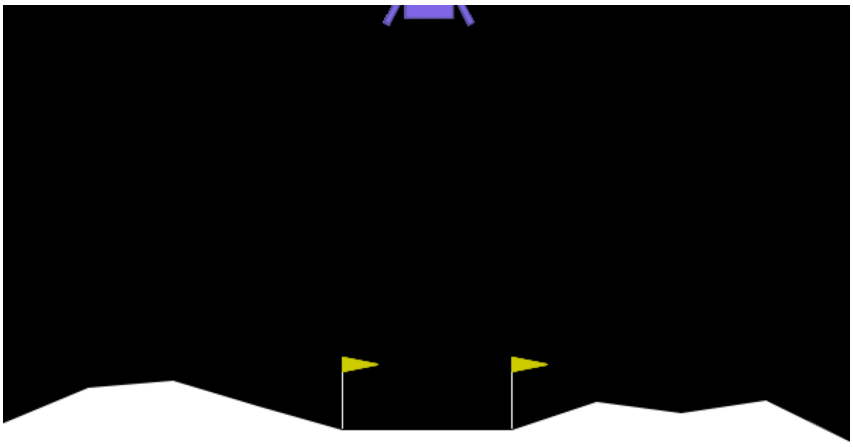
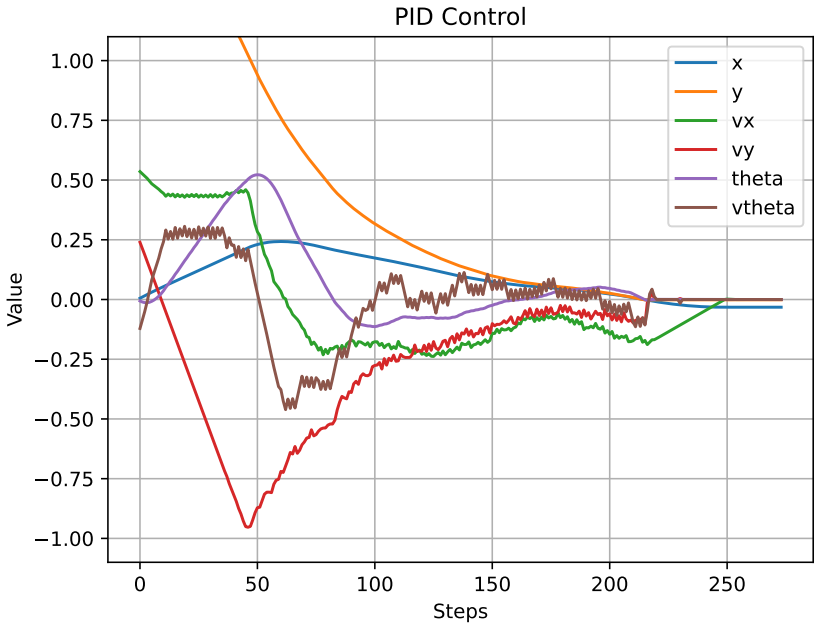


Figure 4: Gyms lunar-lander module.

Info: Pay particular attention to `https://github.com/wfleshman/PID_Control/blob/master/pid.py#L37` and the following lines of code. Your result will depend on the random seed and the parameters I have found works about 95% of the time.



Note the Lunar-lander environment requires the extended openai environments for the physical simulations. You can check the gitlab page for installation instructions suitable for your system if these cause a problem.

References