EXERCISE 4
The continuous control problem and PID control

Tue Herlau
tuhe@dtu.dk

25 February, 2022

Objective: This exercise will be our first look at the control-software for the course. We will consider issues relating to discretization of continuous-time control models and then take a look at PID control which is a simple but widely-used model free control method. (42 lines of code)
Material: Obtain exercise material from our gitlab repository at
`https://gitlab.gbar.dtu.dk/02465material/02465students`

Contents

1 Models, simulation, discretization and environments (`kuramoto.py`)

The code in this section of the course will be a bit more complicated, because the models need to do more such as discretization and simulation. The good news is that the main functionality is still relatively simple.

The overall structure is similar to what you saw when you solved DP problems. Recall we had:

- a model (in the case of DP, a `DPModel` such as the inventory control model, with the `f` and `g` functions),
- an environment (with the `step` and `reset` functions)
- an agent that could use the model to solve tasks in the environment.

This structure will largely be the same. We will have:

- a model (which is now a differential equation, a cost function and some constraints),
- Possibly a discretized version of the original model,
- An environment (which simulate the model exactly),
- An agent, which can plan using the discretized or non-discretized model depend on the method we are using.

In addition to this, we need an easy way to compute derivatives because many control methods depend on that.

This exercise will illustrate these classes by creating a small toy environment, and then showing how the framework can be used to perform the various operations we are interested in. Along the way, you will also get to implement RK4 integration, which all subsequent code will depend on.

1.1 The Kuramoto toy problem

Assume that $x(t) \in \mathbb{R}$ and $u(t) \in \mathbb{R}$ are one-dimensional. The Kuramoto oscillator looks as follows: [1]

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

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

$$\dot{x} = f(x, u) = u + \cos(x). \tag{1}$$

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.

[1] This problem is an instance of a (simplified) Kuramoto oscillator `https://en.wikipedia.org/wiki/Kuramoto_model`, but that is not important. I have chosen it because it is a very simple, non-linear model.

1.2 Part 1: Computing derivatives using sympy

To allow us to compute derivatives, we have to specify the dynamics f using a sympy-expression, which is a really neat framework for symbolic manipulations. As an example, we are going to work with the function:

$$g(z) = e^{\cos(z)^2} \sin(z).$$

To input it, we create a symbolic variable corresponding to z, and input the special functions using `sympy` instead of `numpy`:

```
# kuramoto.py
    z = sym.symbols('z')     # Create a symbolic variable
    g = sym.exp( sym.cos(z) ** 2) * sym.sin(z) # Create a nasty symbolic expression.
    print("z is:", z, " and g is:", g)
```

(you can check the output below)

Problem 1 *Compute derivatives and evaluate the function*

Complete the two tasks outlined in the code (compute derivatives and turn the expression into a function). The lecture notes contain plenty of hints, but my guess is that if you use the debugger, pycharms autocomplete functionality may contain most of the answers.

Info: When done, the output you get should be as follows:

```
z is: z  and g is: exp(cos(z)**2)*sin(z)
The derivative of the nasty expression is dg/dz =
↪  -2*exp(cos(z)**2)*sin(z)**2*cos(z) + exp(cos(z)**2)*cos(z)
dg/dz (when z=0) = 2.718281828459045
dg/dz (when z=pi/2) = -6.123233995736766e-17
(Compare these results with the symbolic expression)
```

Inspect it, and verify the derivatives and your values are computed correctly.

1.3 Part 2: The first model

The model keeps track of the actual differential equation (i.e. f), constraints, cost function, and allows us to simulate it exactly using RK4. Excluding the parts you have to implement, the code for the model looks as follows:

```python
# kuramoto.py
class ContiniousKuramotoModel(ContiniousTimeSymbolicModel):
    def __init__(self):
        self.action_space = Box(low=np.array([-2]), high=np.array([2]), dtype=float)
        self.observation_space = Box(low=np.array([-np.inf]),
        ↪  high=np.array([np.inf]), dtype=float)
        cost = SymbolicQRCost(R=np.ones( (1,1) ), Q=np.zeros((1,1)))
        super().__init__(cost=cost, simple_bounds=None)

    def reset(self):
        """ Return the starting position as a numpy ndarray. In this x=0 """
        return np.asarray([0])

    def sym_f(self, x, u, t=None):
        return symbolic_f_expression
```

As you can tell, we set up an action and observation space (we now use openais `Box`-class, similar to how we used the `Discrete` class in the previous week), which serves the dual purpose of specifying the dimension of x and u and also allows us to specify any simple constraints.

The next part sets up the cost function. The cost function is handled similar to the dynamics (as a symbolic expression), but it is a bit tedious, and generally something I will have already specified in the code; simply note you can specify the various matrices and vectors in the constructor of the `SymbolicQRCost` suitable for a quadratic control problem and the framework will take care of discretization etc.).

The final part, where the constructor in the superclass is called using `super().__init__`, is very important, as this is how the class turns the symbolic expression into a python function using the same ideas you just worked with.

To specify the dynamics, you have to complete the `def sym_f` function. Insert a breakpoint and check what u and x is. We use the conventions that vectors are list, that is, `x` and `u` are in this case singleton lists, and you have to return a singleton list with a symbolic expression eq. (1).

Problem 2 *Implement the continuous-time version of the Kuramoto model*

Implement the symbolic expression in the Kuromoto model. Then complete the function `def f(x,u)` which computes the dynamics using the symbolic model. This function should create a `ContiniousKuramotoModel` object and then call the `def f(x,u)`-function defined in the model to compute the answer.

Info: Use the previous problem to see how symbolic expressions are specified in sympy. Check the output below.

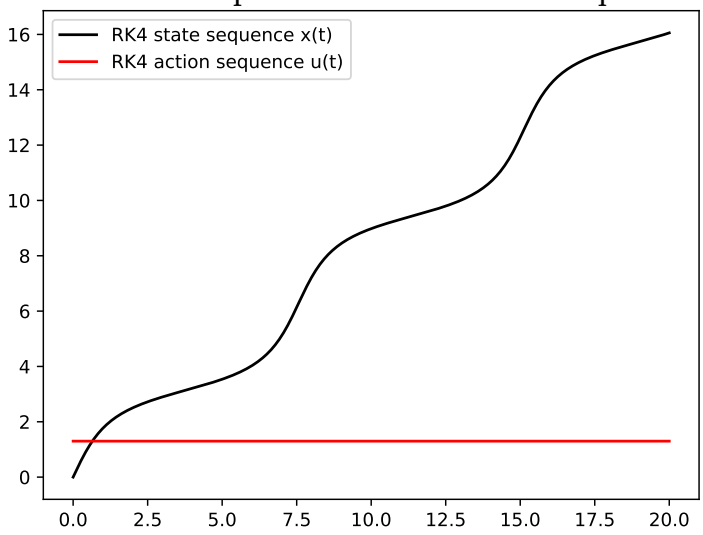
```
Value of f(x,u) in x=2, u=0.3 [-0.11614683654714242]
Value of f(x,u) in x=0, u=1 [2.0]
```

The next line in the code will try to simulate the environment using a highly-accurate RK4 simulation. Unfortunately, RK4 is not yet implemented in the framework:

Problem 3 *RK4 simulation*

Insert a breakpoint and implement RK4 as described in [Her21, algorithm 18] in the file `continuous_time_model.py`. When done, read the code that produces the plot and check that the plot of the trajectory (obtained by using a constant action) is correct.

Info: When done a plot of action and state sequences looks as follows:



I have uploaded the solution to gitlab to avoid this problem being a sticking point.

1.4 Part 3: Discretization

Our next task will be to discretize the environment. The main work occurs in `continuous_time_discretize` in the `DiscretizedModel` class. Given a continuous model, this class will automatically discretize the environment and create the function f_k such that $x_{k+1} = f_k(x_k, u_k)$. In practice this can be as simple as:

```python
# kuramoto.py
class DiscreteKuramotoModel(DiscretizedModel):
```

```
    """ Create a discrete version of the Kuramoto environment.
    The superclass will automatically Euler discretize the continious model (time constant 0.5) and set up useful functionality.
    Note many classes overwrite part of the functionality here to suit the needs of the environment. """
    def __init__(self, dt=0.5):
        model = ContiniousKuramotoModel()
        super().__init__(model=model, dt=dt)
```

Problem 4 *Discrete model*

Use the `DiscreteKuramotoModel` to compute the Euler-discretized step function and derivatives. I.e., when you compute these quantities, create an instance of the discrete model class and simply use functions defined in the class.

Info: When done, you can compute derivatives and step-function values as follows:

```
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.]]
```

1.5 Part 4: 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.

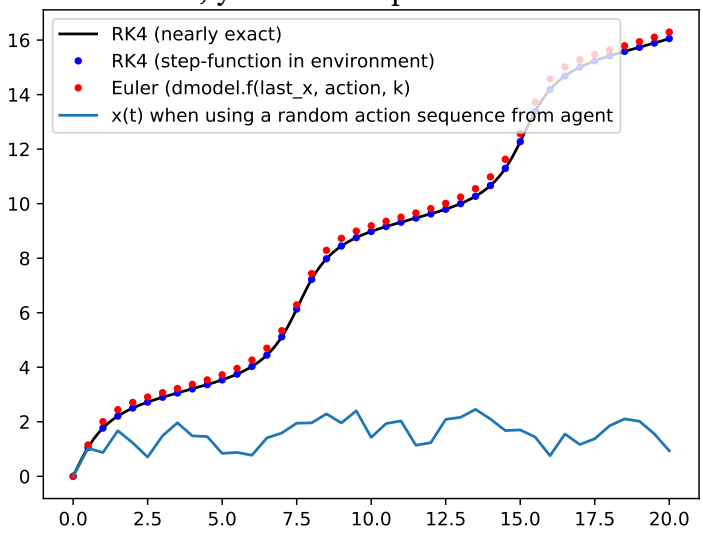
```
# kuramoto.py
class KuramotoEnvironment(ContiniousTimeEnvironment):
    """ Turn the whole thing into an environment. The step()-function in the environment will use *exact* RK4 simulation.
    and automaticcally compute the cost using the cost-function.
    """
    def __init__(self, Tmax=5, dt=0.5):
        discrete_model = DiscreteKuramotoModel(dt)
        super().__init__(discrete_model, Tmax=Tmax)
```

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 5 *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 the `f`-function in the discrete model class (Euler integration). The two will nearly agree but not quite. Note the step-function automatically tells you when the environment has terminated.

Info: When done, you can compute derivatives and step-function values as follows:



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:

```
# kuramoto.py
    env = KuramotoEnvironment(Tmax=20)
    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')
```

2 PID Control

This section will consider PID control. The basic PID method in algorithm 19 accept a single number as input and outputs a single number. Since e.g. the racecar-example involves using multiple pid-instances control multiple numbers (i.e. steering/throttle) we will implement this functionality as a class so it can later be reused.

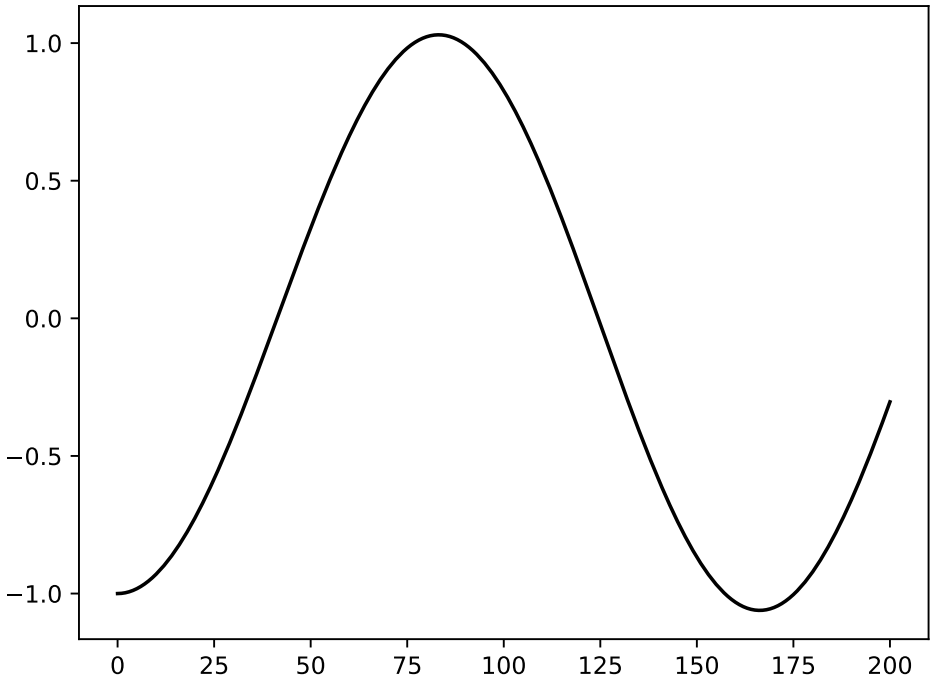
2.1 PID and the locomotive environment (`pid.py`)

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.

Problem 6 *Bare-bones PID and locomotive*

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.

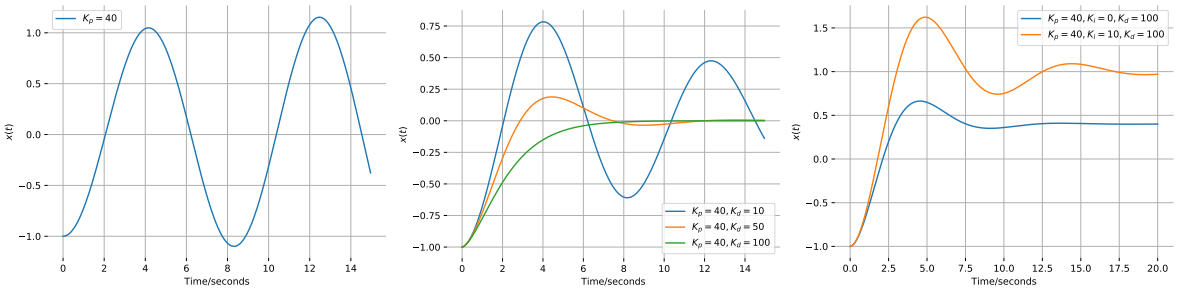
2.2 PID agent and the locomotive environment (`pid_locomotive_agent.py`)

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

Problem 7 *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 trian. You can turn these off by commenting out the `VideoMonitor` lines.



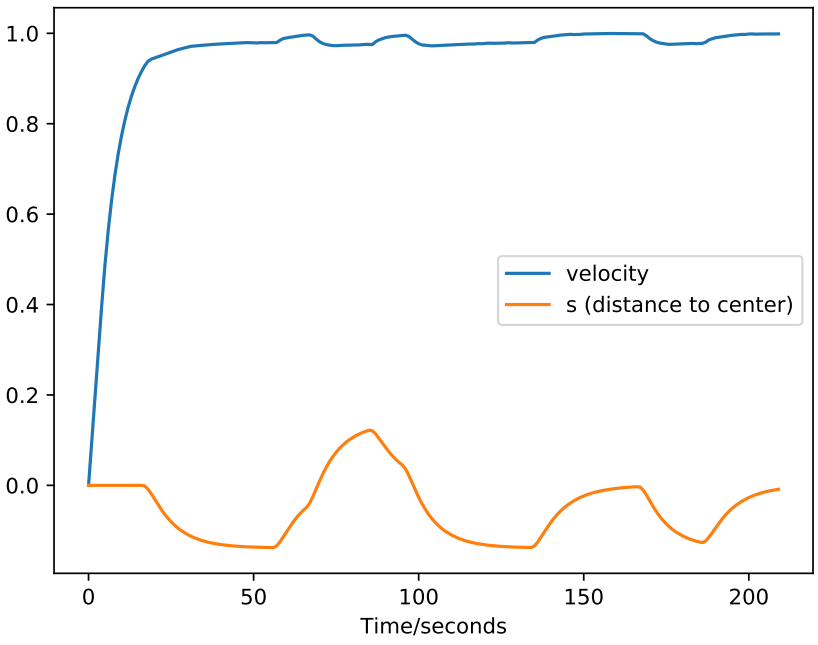
2.3 PID Racecar controller (`pid_car.py`)

Implement the PID racecar example from the lecture notes. 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 8 *Implement*

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.



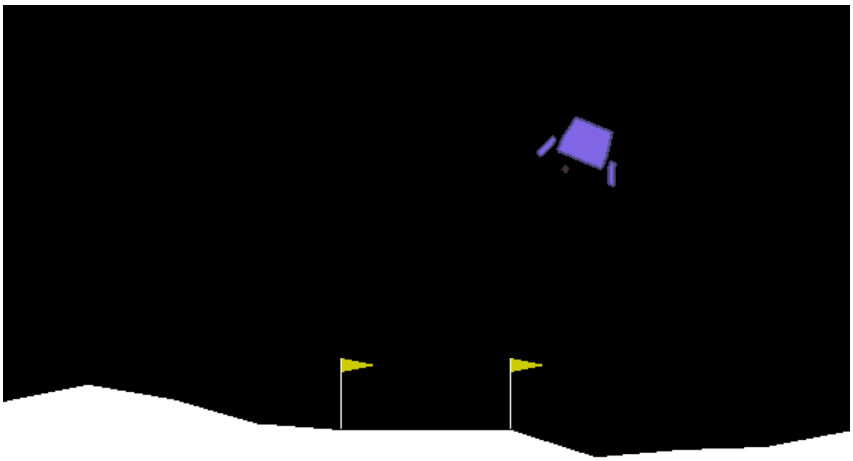


Figure 1: Gyms lunar-lander module.

2.4 Moon landing (`pid_lunar.py`) ★★

This example 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.

Note: I think this is a fairly fun exercise, but a slight drawback is that it requires openais extended 2d-game environments for the physical simulation. The starting point in installing these are:

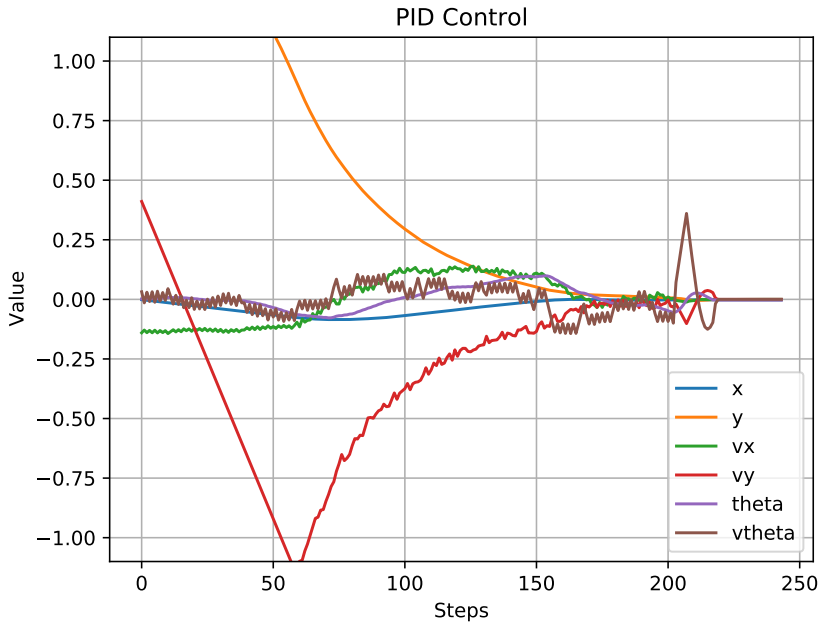
```
pip install gym[box2d]
```

However, on some platforms you may need to install additional software such as `swig`. The good news is that openai gym is such a widely used platform you should be able to find detailed instructions for your system. I have curated the known installation options on our gitlab page.

Problem 9 *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.

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.

2.5 Cartpole balancer (`pid_cartpole.py`)

Our final example relates to the cartpole problem (see fig. 2). Our goal is to create a PID agent which can both balance the pole upright as well as bring the car to $x = 0$ (recall the cartpole can be controlled by applying a force to the cart).

To make the problem more interesting we start the problem in an imbalanced position, obtained by initializing the cart as standing upright and applying a force of $u = 1$ for a few steps. The more steps, the more difficult the subsequent balancing task. As a hint, recall the cartpole is parameterized as

- `x[0]` : x-position. $x = 0$ is center.
- `x[1]` : $\dot{x}$, velocity
- `x[2]` : Pendulum (stick) angle θ. When $\theta = 0$ the pendulum is upright
- `x[3]` : Pendulum angular velocity $\dot{\theta}$

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.

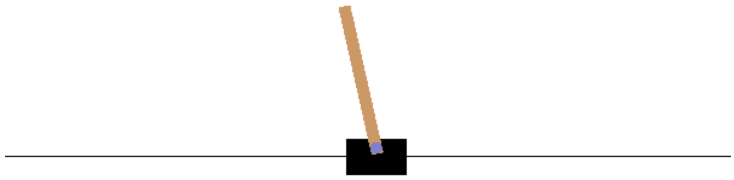
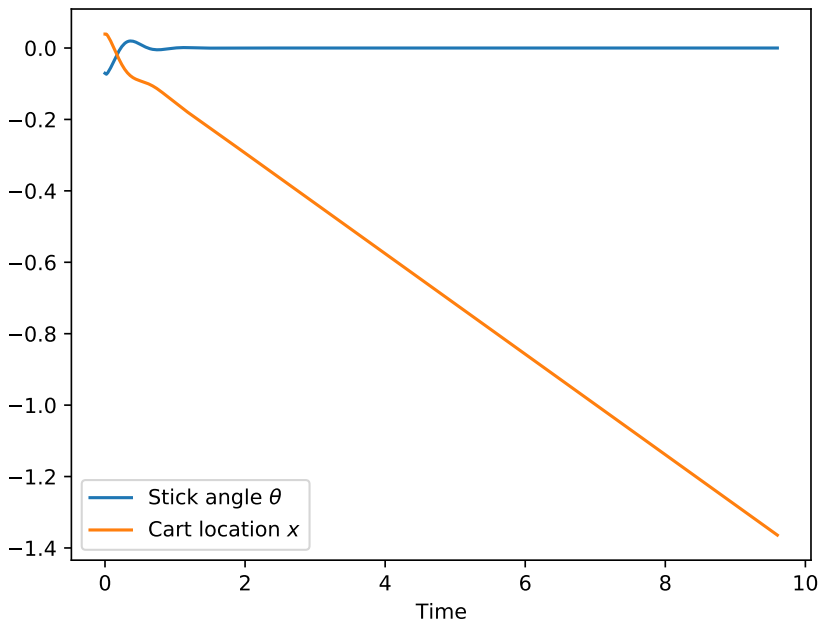


Figure 2: Illustration of cartpole system

Problem 10 *Implement the cartpole balancer in the simplest version*

Implement the cartpole balancing agent where the goal is simply to bring to get the pendulum angle to $\theta = 0$. In the first implementation, we only care about the angle θ.

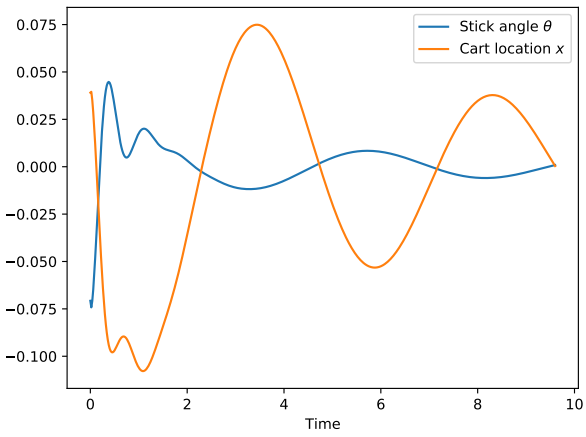
Info: As a hint, for this question, it was sufficient to use `x[2]` as input for this part of the problem. My implementation used K_d and K_p but not K_i (this is also true for the rest of the question). Remember to apply action clipping: `u = np.clip(u, -env.max_force, env.max_fo`



Problem 11 *Bring the cart to the center*

Expand your implementation above so the car is also brought to center, i.e. $\theta = x = 0$.

Info: Solve this by tuning your previous implementation. You need to use additional coordinates from the vector; try to guess a bit and apply scaling before calling the PID controller.

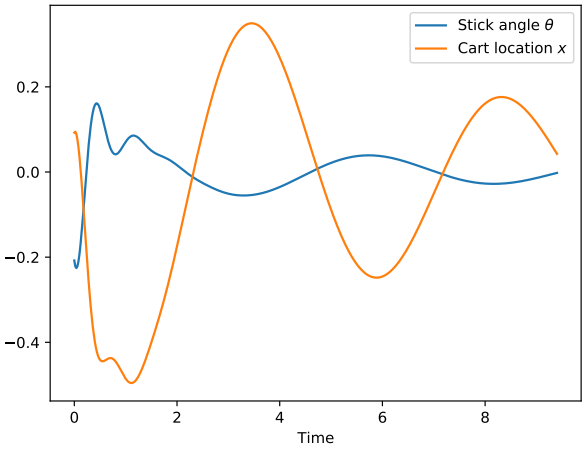


Problem 12 *A more challenging system*

The final question is similar to the previous, but we start from a more challenging position.

Info: Try a few parameters and observe how the controller fails. I found it easier to turn of the animation (and then blindly tune the values) and see what seemed to improve the trajectories. It also makes the problem easier by first tuning K_p with $K_d = 0$; it's value needs to be fairly large.

My implementation is able to handle 40 waiting steps. What is your record?



References

[Her21] Tue Herlau. Sequential decision making. (See **02465_Notes.pdf**), 2021.