EXERCISE 3
DP reformulations and introduction to Control

Tue Herlau
tuhe@dtu.dk

16 February, 2024

Objective: The goal of this exercise is to introduce the control model that will be used in the remainder of the course. You will see how you can specify the dynamics, cost and constraints, and how the control model allows you to simulate the effect of a policy. We will also continue working with the dynamical programming algorithm. (29 lines of code)
Exercise code: `https://lab.compute.dtu.dk/02465material/02465students.git`
Online documentation: 02465material.pages.compute.dtu.dk/02465public/exercises/ex03.html

Contents

1 Pen-and-paper control **1**

2 Kuramoto osscilator (`kuramoto.py`) **2**
2.1 The Kuramoto toy problem 2

3 Implementing as a model **2**

4 Pen-and-paper dynamical programming **4**

5 Exam Question: Exact evaluation (`inventory_evaluation.py`) **5**

6 A 2d toy control problem (`toy_2d_control.py`) **6**

1 Pen-and-paper control

During the lecture, I probably said we were not going to solve differential questions during this course. Well guess what that turned out to not be completely true :-).

In this problem, we will consider a system with a 1-dimensional state $x(t)$ and one-dimensional control signal $u(t)$. Assume that in our standard notation

$$\dot{x}(t) = f(x(t), u(t))$$

where $f(x, u) = e^{-x}u$.

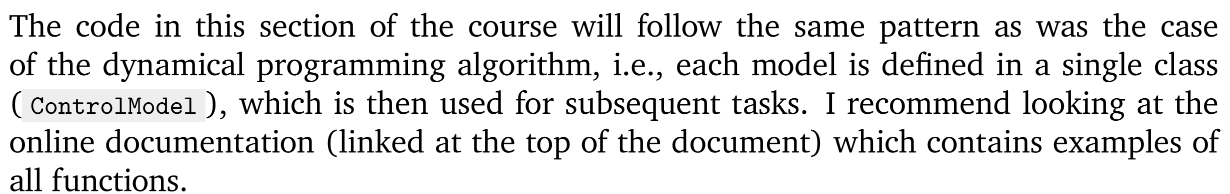
(a.) Assume the initial condition $x(0) = 1$ and that we apply a control signal $u(t) = 2t$. Determine $x(t)$ for $t > 0$. What is $x(1)$? [1]

(b.) Continuing the problem, suppose the cost function is:

$$\text{cost} = \int_0^{t_F} e^{x(t)} dt$$

Determine the total cost as a function of t_F when the control signal in the previous problem is applied.

2 Kuramoto osscilator (`kuramoto.py`)



The code in this section of the course will follow the same pattern as was the case of the dynamical programming algorithm, i.e., each model is defined in a single class (`ControlModel`), which is then used for subsequent tasks. I recommend looking at the online documentation (linked at the top of the document) which contains examples of all functions.

2.1 The Kuramoto toy problem

Assume that $x(t) \in \mathbb{R}$ and $u(t) \in \mathbb{R}$ are one-dimensional. The so-called Kuramoto oscillator is defined by the following differential equation: [2]

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

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.

3 Implementing as a 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:

---

[1] *Hint: This is a problem of plugging in what you know in the first equation. You will end up with a differential equation. Although you can ask Maple to solve it, I recommend giving it a try yourself first.*

[2] 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.

```python
# kuramoto.py
class KuramotoModel(ControlModel):
    def u_bound(self) -> Box:
        return Box(-2, 2, shape=(1,))

    def x0_bound(self) -> Box:
        return Box(0, 0, shape=(1,))

    def get_cost(self) -> SymbolicQRCost:
        return SymbolicQRCost(Q=np.zeros((1, 1)), R=np.ones((1,1)))


    def sym_f(self, x: list, u: list, t=None):
        # define the symbolic expression
        return symbolic_f_list
```

The code sets up the boundary conditions on x and u and a cost function (see the online documentation).

The cost function is handled similar to the dynamics (as a symbolic expression). You can specify the matrices and vectors in the constructor of the `SymbolicQRCost` – more on this next week.

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 1 *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 `KuramotoModel` 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 [-0.11614684]
Value of f(x,u) in x=0, u=1 [2.]
```

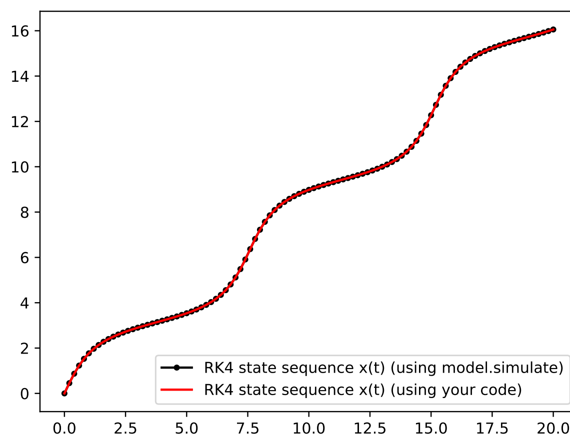
The next parts of the code will try to simulate the function `def f` you just implemented using a the highly-accurate RK4 simulation and while applying a constant action

$u(t) = u_0$. Your job is to implement RK4 and compare to the version already implemented in the framework; when you are done, your result should obviously agree:

Problem 2 *RK4 simulation*

Implement RK4 as described in [Her24, algorithm 18]. Note that in your version, `def f(x,u)` does not depend on `t` and that $u(t)$ is a constant, which simplifies RK4. The function should return lists of all simulated states and times x_k and t_k. Note how the model, defined in `continuous_time_model.py`, contains a working implementation you can glance at if you are stuck.

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



4 Pen-and-paper dynamical programming

Consider a dynamical programming problem where it is known that:

$$f_k(x_k, u_k, w_k) = 2e^{(x_k+w_k-u_k)^2}$$
$$g_k(x_k, u_k, w_k) = (x_k + w_k - u_k)^2$$
$$g_N(x_N) = \log x_N$$

We also assume that $w_k \sim \mathcal{N}(0, 1)$ (i.e. a normal distribution with unit variance).

(a.) We wish to apply the DP algorithm to this problem, and we consider the first step where $k = N - 1$. Why is the expression:

$$Q(x, u) = \mathbb{E}_w \left[g_{N-1}(x, u, w) + J_N(f_{N-1}(x, u, w)) \right]$$

of particular relevancy? Make sure you understand how it arises from the DP algorithm.

(b.) Evaluate the expectation and determine a closed-form expression for $Q(x, u)$ (Hint: Don't use maple. The integral is not supposed to be scary).

(c.) Determine the optimal policy $\mu_{N-1}(x_{N-1})$ *(i.e., only for time $k = N-1$)* various

(d.) Determine the optimal cost function $J_{N-1}(x_{N-1})$

(e.) Assume that we used a noise distribution with standard deviation σ, i.e. that

$$w_k \sim \mathcal{N}(0, \sigma^2).$$

How does this choice change μ_{N-1}?. How would this affect $J_{N-1}(x_{N-1})$?.

(f.) The change to μ_{N-1} is particularly interesting and perhaps somewhat paradoxical – discuss what it means in practical terms in the limit where σ is very large (Hint: We will see something very similar when we discuss LQR in a few weeks).

When you solve these problems I recommend that you avoid maple: If you end up with an expression maple would genuinely help you with, you are probably doing it wrong!

5 Exam Question: Exact evaluation (`inventory_evaluation.py`)

This problem exclusively focuses on the inventory control problem discussed in [Her24, section 5.1.2]. The inventory control model and the dynamical programming algorithm is included in the exam folder.

The dynamical programming algorithm determines the optimal cost-to-go function J_k^* for a dynamical programming problem as a list of dictionaries. We are interested in building a variant of the DP algorithm which computes the expected tail cost of a policy $J_{\pi,k}(x_k)$, also represented as a list of dictionaries.

- The problem itself will be represented as a `DPModel` instance
- The policy $\pi = (\mu_0, \mu_1, \ldots, \mu_{N-1})$ is represented in the usual way as a list of length $N-1$. Each element of this list corresponds to a μ_k and is represented as a dictionary which maps states to actions

(a.) Complete `def a_expected_items_next_day(x : int, u : int)`: This function is given the starting state x_0 and the first action u_0, and computes the expected value of the next state x_1:

$$\mathbb{E}\left[x_1 | x_0, u_0\right].$$

I.e., the function computes the expected amount of goods in the warehouse on day 1 given information about how much was in the warehouse on day 0 and how much we ordered u_0. *Hint: Recall that $x_1 = f_0(x_0, u_0, w_0)$ where w_0 is the random noise disturbance at time step $k = 0$.*

(b.) Complete `def b_evaluate_policy(pi : list, x0 : int)`: This function is given a policy π in the beforementioned format and a starting state x_0 and compute the expected tail cost $J_{\pi,0}(x_0)$ when starting in state x_0 at time $k = 0$ and subsequently taking actions according to π. *Hint: You may find [Her24, section 6.3.1] to be of help.*

6 A 2d toy control problem (`toy_2d_control.py`)

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:

$$\ddot{w} = \cos(u + w) \tag{2}$$

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:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), u(t)) \tag{3}$$

(a.) Determine the function $\boldsymbol{f}$ above:

$$\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}) = \dots$$

(b.) Suppose our goal is to bring the system to an angle $w = \frac{\pi}{2}$ (where it should remain), corresponding to the goal state $\boldsymbol{x}^* = \begin{bmatrix} \frac{\pi}{2} \\ 0 \end{bmatrix}$.

If we succeed at bringing the system to this target state $\boldsymbol{x}^*$ at time t', how much control $u(t')$ do we subsequently need to apply to keep the system at $w(t) = \frac{\pi}{2}$?

(c.) Consider the system written as eq. (3). Suppose we initialize the system in state $\boldsymbol{x}^*$ and apply a constant control signal $u(t) = u_0$ for T seconds. Complete the function `toy_simulation(u0, T)` which should return the angle the system will be in, $w(T)$, after T seconds of RK4 simulation.

Hint: You must have solved problem (a). Similar to the Kuramoto-example, implement a `ControlModel` and use the `model.simulate`-function. When done, you will get the state $\boldsymbol{x}(T)$ as a numpy `ndarray`. The first coordinate will be $w(T)$.

Info: When done, your program should produce this output:

```
1 Starting in x0=[pi/2, 0], after T=5 seconds the system is an an angle
  ↪ wT=1.265264700821797 (should be 1.265)
```

References

[Her24] Tue Herlau. Sequential decision making. (Freely available online), 2024.