EXERCISE 5
Direct methods and control by optimization

Tue Herlau
tuhe@dtu.dk

28 February, 2025

Objective: In this exercise, we will look at optimization-based methods for optimal control, specifically direct collocation. Direct collocation works by transforming the control problem into a single, large, non-linear constrained optimization problem which is then solved by a black-box solver. (33 lines of code)
Exercise code: `https://lab.compute.dtu.dk/02465material/02465students.git`
Online documentation: 02465material.pages.compute.dtu.dk/02465public/exercises/ex05.html

Contents

1 Applying direct methods to the Kuromoto problem

Note: An earlier version of this problem had a misprint affecting problems c, d, and e. The problem has been amended and simplified slightly. I am sorry for the inconvenience. We will once again consider the Kuromoto oscillator 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:

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

We will assume that the cost function is:

$$\{\text{cost}\} = \int_0^{t_F} \left(\left(2x(t) - \frac{\pi}{2}\right)^2 + \frac{\pi^2}{4}\left[2u(t) + 2\cos(x(t)) + 1\right]^2 \right) dt$$

We assume that the system is subject to the constraint that $t_0 = 0$, $t_F \leq 10$ and $x(t_0) = 0$, $x(t_F) = \frac{\pi}{2}$.

In the following, we apply Trapezoid collocation to this problem using $N = 3$. Hint: This means that the time between two time points is $h_k = \frac{t_F - t_0}{2}$.

(a.) What are the dimensions of the vectors $\boldsymbol{z}$, $\boldsymbol{z}_{\text{lb}}$ and $\boldsymbol{z}_{\text{ub}}$?

(b.) Define the relevant quantities. I.e., using the definitions of the Kuromoto problem, what are:

$$\boldsymbol{z} = \cdots, \quad \boldsymbol{z}_{\text{lb}} = \cdots, \quad \boldsymbol{z}_{\text{ub}} = \cdots$$

(c.) Show that there are two collocation constraints and they are equivalent to:

$$x_1 = \frac{t_F}{4}(u_1 + \cos(x_1) + u_0 + 1) \tag{2}$$

$$\frac{\pi}{2} - x_1 = \frac{t_F}{4}(u_1 + \cos(x_1) + u_2) \tag{3}$$

(d.) Assuming that $u_0 = u_2 = 0$, show that the discretized cost function is:

$$E(u_1, u_2, u_3) = \frac{c_0 + c_2}{2} + c_1 \tag{4}$$

$$c_0 = \frac{t_F}{2}\left(\frac{\pi^2}{4} + 9\frac{\pi^2}{4}\right) = \frac{5\pi^2 t_F}{4} \tag{5}$$

$$c_2 = \frac{t_F}{2}\left(\frac{\pi^2}{4} + \frac{\pi^2}{4}\right) = \frac{\pi^2 t_F}{4} \tag{6}$$

$$c_1 = \frac{t_F}{2}\left[(2x_1 - \frac{\pi}{2})^2 + \frac{\pi^2}{4}(2u_1 + 2\cos x_1 + 1)^2\right] \tag{7}$$

(e.) **(Challenge!)** Assume that $u_0 = u_2 = 0$. In other words, the only free variables are x_1, u_1, t_F and the cost function can be simplified.

Under these assumptions, solve the constrained collocation optimization problem to determine the final time t_F. Do this by writing the cost function as only a function of t_F by using the constraints, then minimize this function to find t_F.[1]

[1]Hint: This problem may seem daunting, but the cost has been chosen to simplify the exercise, and you will not need math outside ordinary highschool math. It may be useful to first solve for t_F^2

2 Direct methods and trapezoid collocation

In this section, we will implement the trapezoid collocation method as described in [Her24, section 15.3], specifically [Her24, algorithm 20]. Alternatively, the reference [Kel17] provides an excellent (and more in depth) description of the same material.

This will amount to transforming (transcribing) the continuous-time problem (in terms of the dynamics f and cost functions c_F and c as well as bounds) into a large optimization problem, which will be solved by an optimizer.

2.1 Constrained optimization

The best way to get an idea of how this should proceed is to look at the code we have for solving a big constrained minimization task. You can find an example below, and a more in-depth discussion in the online documentation:

```python
# sample.py
ineq_cons = {'type': 'ineq',
             'fun': lambda x: np.array([1 - x[0] - 2 * x[1],
                                        1 - x[0] ** 2 - x[1],
                                        1 - x[0] ** 2 + x[1]]),
             'jac': lambda x: np.array([[-1.0, -2.0],
                                        [-2 * x[0], -1.0],
                                        [-2 * x[0], 1.0]])}
eq_cons = {'type': 'eq',
           'fun': lambda x: np.array([2 * x[0] + x[1] - 1]),
           'jac': lambda x: np.array([2.0, 1.0])}
from scipy.optimize import Bounds
z_lb, z_ub = [0, -0.5], [1.0, 2.0]
bounds = Bounds(z_lb, z_ub)  # Bounds(z_low, z_up)
z0 = np.array([0.5, 0])
res = minimize(J_fun, z0, method='SLSQP', jac=J_jac,
               constraints=[eq_cons, ineq_cons], bounds=bounds)
```

The code uses the `minimize` function to minimize the (python) function `J_fun` (corresponding to an optimization problem $J(\boldsymbol{z})$ subject to both equality and inequality constraints. The constraints will come in two forms: Simple constraints

$$\boldsymbol{z}_L \leq \boldsymbol{z} \leq \boldsymbol{z}_U$$

and complex, functional constraints of the form $h(\boldsymbol{z}) = 0$ and equality constraints. The simple constraints simply corresponds to two lists (`z_lb` and `z_ub`), whereas the complex constraints are implemented using dictionaries. Specifically what `scipy` wants is:

- The objective function `J_fun`
- An initial point `z0` (which should satisfy the constraints)
- `jac` refers to the Jacobian. I.e. this should be a *function* which returns the gradient of the objective with respect to all variables

- `bounds` is an instance of the `Bounds`, and represent simple upper/lower bounds on $\boldsymbol{z}$. In other words, we have to specify simple bounds from the complex ones

2.2 Overview

We will need to compute a lot of Jacobians. That is why we use the symbolic toolbox: We will simply specify symbolic expressions we need to use, and then let sympy turn these expressions into regular python functions and their Jacobians.

- First we will compute a python list of all symbolic variables corresponding to $\boldsymbol{z}$ (which we will simply call `z`); as well as lists of numbers corresponding to `z_lb, z_ub, z0` above.
- Recall that the order of the variables in $\boldsymbol{z}$ is described in the lectures. In other words, the last entry in $\boldsymbol{z}$, `z[-1]` will correspond to t_F.
- The main part of the program will therefore be concerned with creating lists of (symbolic) expressions corresponding to the inequality and equality constraints, as well as optimizer objective J, which should be a single symbolic expressions which uses only the variables in `z`. This correspond exactly to the procedure outlined in the slides and notes, and the variables will have similar names.
- The program then computes derivatives of all these symbolic expressions to obtain the various Jacobians of the objectives and so on. It also packs the code into dictionary objects matching `ineq_cons` above so we can pass it into the minimize-function
- The scipy optimizer is called and it returns a result datastructure `res`. The optimal $\boldsymbol{z}$ can be accessed as `z_star = res.x`. This can then be used to construct functions $\boldsymbol{x}(t)$ and control inputs $\boldsymbol{u}(t)$ by interpolating as in the slides/notes.

The whole thing is called iteratively on a finer and finer grid (i.e. higher N). That is, we first call the method with a crude guess and a low value of N, then the result (the last step) is used as a new guess, and we iteratively call the function with increasing values of N. This is required since the optimization problem is too hard to solve in one step.

2.3 Creating the simple bounds (`direct.py`)

First, we will compute the list of all symbolic variables, $\boldsymbol{z}$ (or `z` in the code), the simple upper/lower bounds on $\boldsymbol{z}$, and also the initial guess for $\boldsymbol{z}$ (`z0`).

The value of the bounds can be obtained using the methods defined in the `ControlModel`, i.e. `model.x_bound()` gives the bounds for $\boldsymbol{x}$.

Problem 1 *Variables and simple bounds*

Implement the list corresponding to $\mathbf{z}$, the initial guess $\mathbf{z}_0$, and the upper and lower bounds $\mathbf{z}_{\text{lb}}$ and $\mathbf{z}_{\text{ub}}$. *Carefully check the output, both to ensure you have the right order, and to make sure you understand which variables are bounded or not in our notation.*

Info: When done, you should get the following output:

```
z=[x_0_0, x_0_1, u_0_0, x_1_0, x_1_1, u_1_0, x_2_0, x_2_1, u_2_0, x_3_0, x_3_1,
↪  u_3_0, x_4_0, x_4_1, u_4_0, t0, tF]
z0=[3.1, 0.0, 0.0, 2.4, 0.0, 0.0, 1.6, 0.0, 0.0, 0.8, 0.0, 0.0, 0.0, 0.0, 0.0,
↪  0.0, 4.0]
z_lb=[3.0999999046325684, 0.0, -inf, -6.300000190734863, -inf, -inf,
↪  -6.300000190734863, -inf, -inf, -6.300000190734863, -inf, -inf, 0.0, 0.0,
↪  -inf, 0.0, 0.5]
z_ub=[3.0999999046325684, 0.0, inf, 6.300000190734863, inf, inf,
↪  6.300000190734863, inf, inf, 6.300000190734863, inf, inf, 0.0, 0.0, inf, 0.0,
↪  4.0]
```

2.4 Symbolic collocation (`direct.py`)

Next, you must create all equality and inequality constraints in the problem, as well as computing the cost function as a symbolic expression. The code contains details, but crucially you must add code which computes the trapezoid collocation constraint as in [Her24, eq. (15.20)] (see [Her24, algorithm 20])

Note that the Mayer and Lagrange terms in the cost function are implemented in the symbolic environment as `cost.sym_cf` and `cost.sym_c`. You do not need to bother with how these are implemented in details[2].

Problem 2 *Collocation and constraints*

Implement the symbolic expressions for cost function and all constraint for trapezoid collocation. I.e. the lists of symbolic variables `Ieq` and the symbolic variable `J`.

[2]The implementation is similar to how the dynamics were treated

Info: When done, you should get the following output:

```
Ieq=[-x_0_0 + x_1_0 - (-0.125*t0 + 0.125*tF)*(x_0_1 + x_1_1), -x_0_1 + x_1_1 -
↪  (-0.125*t0 + 0.125*tF)*(1.25*u_0_0 + 1.25*u_1_0 + 9.82*sin(x_0_0) +
↪  9.82*sin(x_1_0)), -x_1_0 + x_2_0 - (-0.125*t0 + 0.125*tF)*(x_1_1 + x_2_1),
↪  -x_1_1 + x_2_1 - (-0.125*t0 + 0.125*tF)*(1.25*u_1_0 + 1.25*u_2_0 +
↪  9.82*sin(x_1_0) + 9.82*sin(x_2_0)), -x_2_0 + x_3_0 - (-0.125*t0 +
↪  0.125*tF)*(x_2_1 + x_3_1), -x_2_1 + x_3_1 - (-0.125*t0 +
↪  0.125*tF)*(1.25*u_2_0 + 1.25*u_3_0 + 9.82*sin(x_2_0) + 9.82*sin(x_3_0)),
↪  -x_3_0 + x_4_0 - (-0.125*t0 + 0.125*tF)*(x_3_1 + x_4_1), -x_3_1 + x_4_1 -
↪  (-0.125*t0 + 0.125*tF)*(1.25*u_3_0 + 1.25*u_4_0 + 9.82*sin(x_3_0) +
↪  9.82*sin(x_4_0))]
Iineq=[]
J=(-0.125*t0 + 0.125*tF)*(0.5*u_0_0**2 + 0.5*u_1_0**2 + 0.5*x_0_0**2 +
↪  0.5*x_0_1**2 + 0.5*x_1_0**2 + 0.5*x_1_1**2) + (-0.125*t0 +
↪  0.125*tF)*(0.5*u_1_0**2 + 0.5*u_2_0**2 + 0.5*x_1_0**2 + 0.5*x_1_1**2 +
↪  0.5*x_2_0**2 + 0.5*x_2_1**2) + (-0.125*t0 + 0.125*tF)*(0.5*u_2_0**2 +
↪  0.5*u_3_0**2 + 0.5*x_2_0**2 + 0.5*x_2_1**2 + 0.5*x_3_0**2 + 0.5*x_3_1**2) +
↪  (-0.125*t0 + 0.125*tF)*(0.5*u_3_0**2 + 0.5*u_4_0**2 + 0.5*x_3_0**2 +
↪  0.5*x_3_1**2 + 0.5*x_4_0**2 + 0.5*x_4_1**2)
```

2.5 Computing the derivatives (`direct.py`)

The SLQP optimizer is greatly helped if it is passed the derivative of the objective and of the constraints. Since these are symbolic variables it is reasonably easy to implement, but you might want to look around in the code for inspiration.

Problem 3 *Gradients*

Add code to compute the gradient of the objective $\nabla J(\boldsymbol{z})$ as a numpy function. This will be passed on to `minimize`.

Once this is completed, the trapezoid collocation procedure is completed and we can call minimize! Assuming all checks turned out well, you now have a solution expressed as an (optimal) value of $\boldsymbol{z}$.

2.6 Trapezoidal interpolation (`direct.py`)

Now that we have the optimal value of $\boldsymbol{z}$, we need to unpack it to obtain the solutions paths as described in [Her24, section 15.3.3].

Problem 4 *Interpolation*

Implement the interpolation rule that takes the solution $\boldsymbol{z}$, extract the relevant variables, and re-construct the predicted trajectory $\boldsymbol{x}(t)$ from this information using the method in [Her24, section 15.3.3].

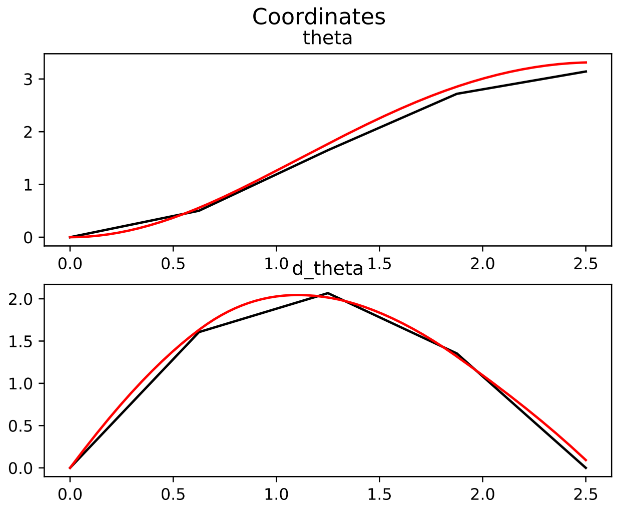
Info: When done, you should get the followin g output:

```
[[ 3.14159274e+00  1.84310455e+00  3.54438028e-01  6.99947725e-08]
 [ 0.00000000e+00 -2.84705443e+00 -9.97919763e-01 -3.08682250e-07]]
```

as well as the figure shown next

2.7 Output

The output of running the above code will be the following figure:



It shows how well the solution of the path $\boldsymbol{x}(t)$ found by the direct method you just implemented matches the solution obtained by taking the control signal you found from the solver, $\boldsymbol{u}(t)$, and simulating it in the real system using the RK4 scheme. We see the solution actually works: it brings the pendulum to the right place. However, the mesh is so coarse the quality of the solution is not great. Try to increase $N = 20$ to see how that solves the problem.

2.8 Iteratively calling the method (`direct_pendulum.py`)

We need one last component before we have have a complete implementation: Iteration. Non-linear optimization will in general be very dependent on initialization. This goes for neural networks, but even more so for constrained optimizers as considered here. The way we will accomplish this is to first run the direct optimizer on a very coarse grid, for instance $N = 10$, giving an approximate solution $\boldsymbol{x}^{N=10}(t), \boldsymbol{u}^{N=10}(t)$. Then we use that solution to initialize the guess for the next grid. I.e. given a new value of N, for instance $N = 30$, we compute grid-points $t_0, \dots, t_{N=40}$, and initialize a guess as

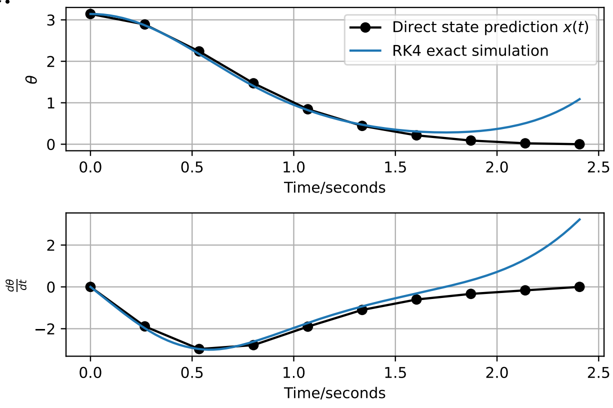
$$\boldsymbol{x}^{\text{guess}}(t_k) = \boldsymbol{x}^{N=10}(t_k). \tag{8}$$

Code-wise this is easy to do. The guess is supplied to the collocation method already, and therefore all you need to do is to set the guess (at a given iteration $k > 0$ of the method) equal to the solution at iteration $k - 1$.

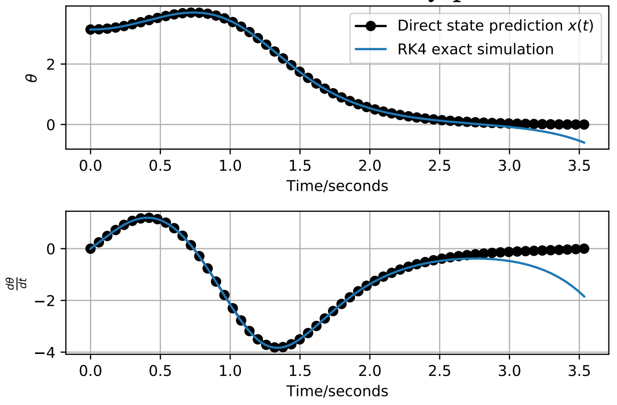
Problem 5 *Guess and iteration*

Update the code so that, when called iteratively, the guess is initialized based on the output from the previous iteration. The code can then solve the pendulum swingup task.

Info: The output will both show the defects, actions and states. The states for $N = 10$ are:



which can be seen to be a fairly poor solution. For $N = 60$ they become:



which is sufficient to solve the task.

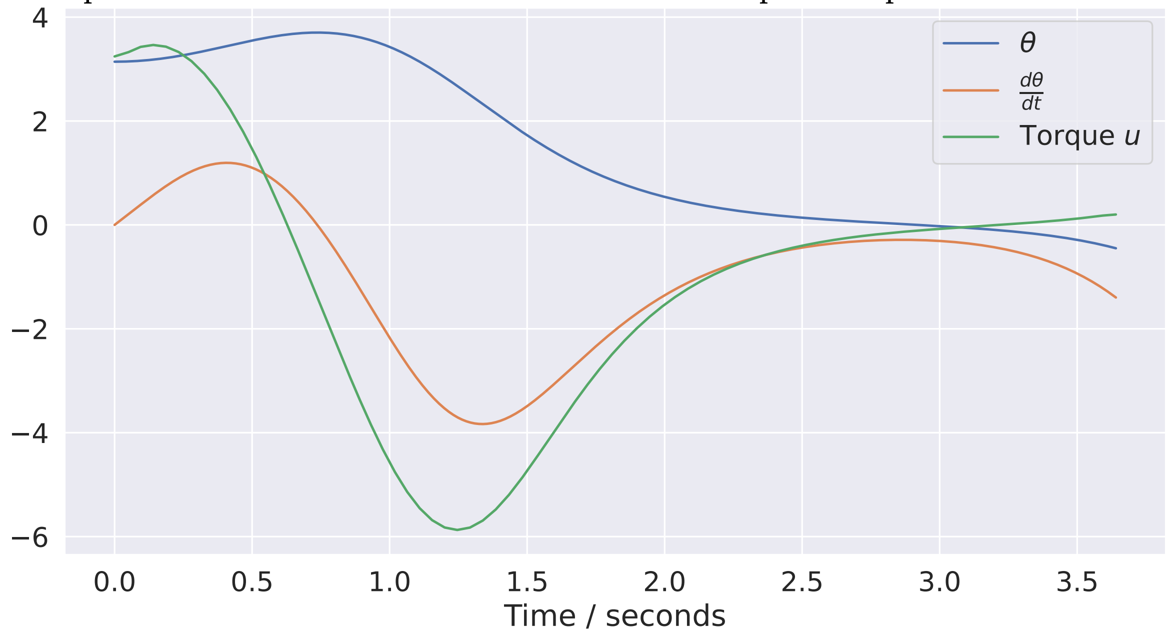
3 A direct-solver agent (`direct_agent.py`)

Direct optimization is perhaps the method which is the least well-suited for the agent-environment interface we use in this course, however, we can certainly still build an agent which plan using a direct solver and then executes the actions, and thereby allow us to re-use the visualization and plotting methods we already know. The example for doing so will build on the pendulum-problem section 2.8 and should return the same solution. Note we need to turn the continuous model into an environment, and we do so using the standard methods we discussed last week.

Problem 6 *Agent code*

Implement the agent code. the missing functionality is code which saves the action trajectory $\boldsymbol{u}(t)$ from the solution, and code which then call the action trajectory whenever the agent needs to execute the policy. The result is not perfect. Can ´you explain why?

Info: The output will show the state and actions trajectories, and a small animation which repeats itself. The result should be identical to the previous problem.



4 Swingup task (`direct_cartpole_time.py`)

We should now be able to run the cartpole swingup task from [Kel17, Section 6]. The cartpole swingup task consists of swinging up the cartpole in the minimum amount of time.

I had problems making the swingup task succeed using the parameters in [Kel17], but this might be due to us using different dynamics (we are using physically correct dynamics in this course). We are therefore going to consider a slightly easier version where the parameters are taken from `https://github.com/MatthewPeterKelly/OptimTraj/`

`blob/master/demo/cartPole/MAIN_minTime.m` where the cart-movement constraint is the same as in [Kel17] but the actuator force is now $50N$ rather than $20N$.

The code is already complete, and the task is to run the existing code and discuss what the input arguments are. This is relevant for project 2, where you have to apply the direct method for an optimal planning problem.

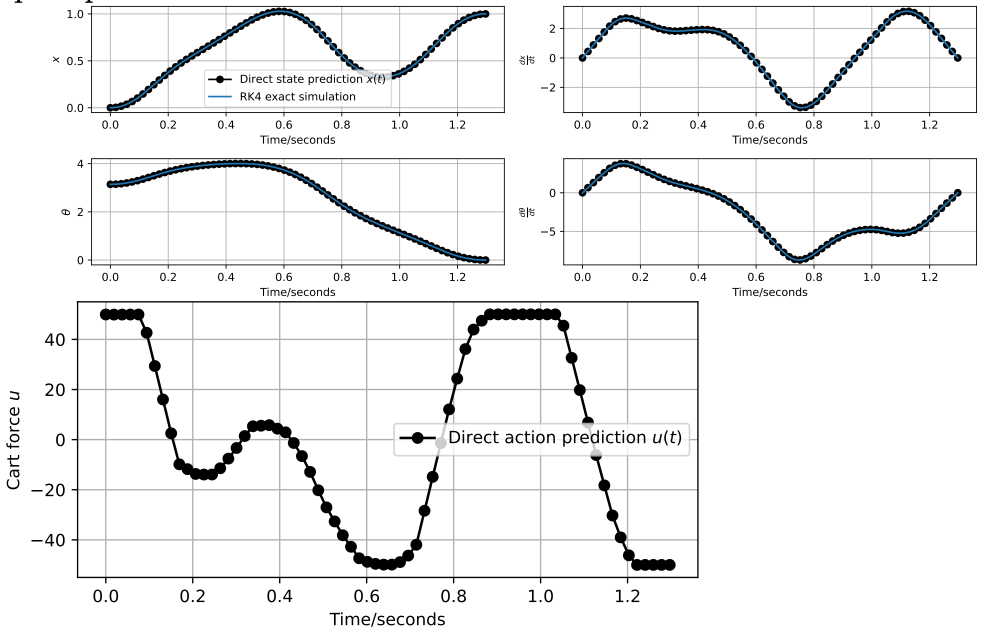
Problem 7 *Cartpole minimum time swingup task*

Investigate and discuss what bounds are applied to the Cartpole problem. Then discuss how the grid is refined. Try to change the grid-refinement procedure and see if the solution still works.

Remember the pole is pointing up at an angle of $\theta = 0$ and down at $\theta = \pi$ and look at `https://github.com/MatthewPeterKelly/OptimTraj/blob/master/demo/cartPole/MAIN_minTime.m` for further details.

Info: All the constraints are implemented as inequality constraints. Use `print(model)` to see details about a model (see the online documentation). Recall that equality constraints can be obtained by letting the upper/lower bounds agree.

The script uses gradual grid refinement and this appears necessary for this particular problem. Note the first solutions will be quite poor. You should check the constraint violations (defects) are small. The following plots show the trajectory, for the finest mesh, as well as the action path, however the script will also generate outputs for the less-refined action paths which have the same shape but are obviously quite poor.



The curves show the direct methods predicted trajectory (i.e. what the optimizer think will happen) as well as the true trajectory as obtained by RK4 integration (there are two curves, they just agree!). Other plots, not shown here, will show the defects, i.e. how much the equality constraints in the collocation are violated. The defects should be small, otherwise the optimization has failed.

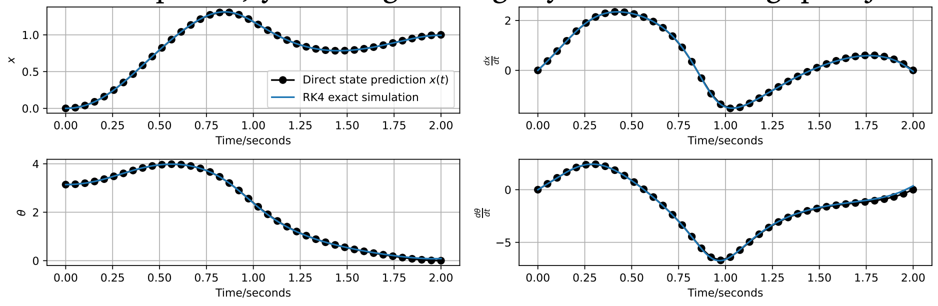
4.1 The Kelly-swingup task (`direct_cartpole_kelly.py`)

As an additional example, we will consider the (alternative) cartpole swingup task from [Kel17] (we denote this the *Kelly swingup task* in the following). Since the direct solver should not require any more work, the only challenge is to set up the problem correctly. To do so, look at the hints in the code as well as [Kel17, section 6] and [Kel17, Appendix E, table 3]. The task is similar to the one in section 4, except we fix $t_F = 2$ and use a different cost-function and physical parameters. From a mechanical point of view, the swingup is much more gentle.

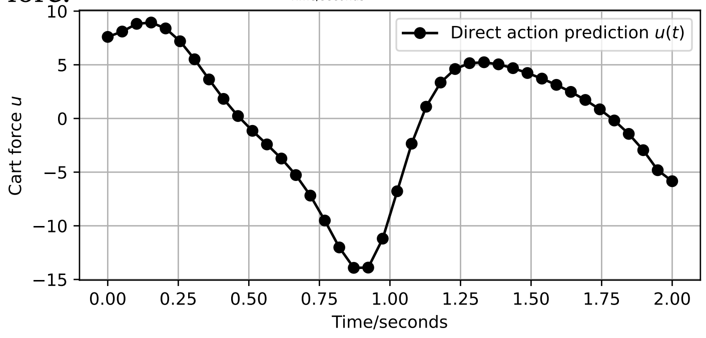
Problem 8 *Kelly swingup task*

Complete the problem specification in the code. Complete the `KellyCartpoleModel` to implement (i) a constraint so that $t_F = 2$ and (ii) A new quadratic cost-function with matrices $Q = 0$, $R = I$. Note the problem appears relatively easier than the previous one numerically.

Info: Once completed, you will get a slightly different swingup trajectory from be-



fore.



References

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

[Kel17] Matthew Kelly. An introduction to trajectory optimization: How to do your own direct collocation. *SIAM Review*, 59(4):849–904, 2017. (See **kelly2017.pdf**).