EXERCISE 5
Direct methods and control by optimization

Tue Herlau
tuhe@dtu.dk

4 March, 2022

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. (30 lines of code)
Material: Obtain exercise material from our gitlab repository at
`https://gitlab.gbar.dtu.dk/02465material/02465students`

Contents

1 Direct methods and trapezoid collocation

In this section, we will implement the trapezoid collocation method as described in [Her21, section 13.3], specifically [Her21, algorithm 20]. Alternatively, the reference

[Kel17] provides an excellent (and more in depth) description of the same material, and I would recommend it if you are going to expand this exercise into a project.

At any rate, what this exercise will amount to is that given a description of the continuous-time problem (in terms of the dynamics f and cost functions g_F and g as well as constraint functions and bounds), we will turn this into a large optimization problem. Let's therefore begin with the optimization.

1.1 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. The following is taken from `https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html#sequential-least-squares-program` and solves a 2-dimensional optimization problem with three inequality constraints and one equality constraint:

```
# 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)
```

I recommend looking at this code carefully as well as the above link.

What the code does is using 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 (see above link for the function which is minimized). 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 $h(\boldsymbol{z}) \leq 0$ (as well as equality constraints). The simple constraints are implemented using a `Bounds` object, but the complex constraints are implemented using dictionaries as given above. Specifically what we need 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 z. In other words, we have to specify simple bounds from the complex ones

The inequality and equality constraints should be specified in two different dictionary data structures. Once more they contain a function (to evaluate the inequality constraint at a given point) as well as a different function to evaluate the Jacobian at a given point. When the constraints are correctly specified, this should work:

```
# sample.py
ineq_cons['fun'][z0]  # return a 3 x 1 array
ineq_cons['jac'][z0]  # return a 3 x 2 matrix (the Jacobian of fun) because we have 2
↪  input variables
ineq_cons['fun'][z0] <= 0 # this will be true of the inequality constraint for a
↪  solution
```

Note that as the code indicates, we can pass an arbitrary number of inequality/equality constraints to the minimizer as the list `constraints=[...]`; therefore, we can just specify the constraints one at a time and collect them in this list at the end.

1.2 Overview

The observant reader will notice that the function requires the Jacobian of all quantities. That is why we use the symbolic toolbox: We will simply specify symbolic expressions to compute

```
ineq_cons['fun']
J_fun
```

and so on and then sympy can later be used to turn these expressions into regular python functions and their Jacobian.

What we need is therefore the following:

- First we will compute a python list of all symbolic variables corresponding to z (which we will simply call `z`); as well as lists of numbers corresponding to `z_lb, z_ub, z0` above. These are fed into the objective function just as in the example. The order of these variables in e.q. `z` and `z_lb` must obviously match each other, and since it will also be the order the optimizer returns things in we have to take care so we can re-construct the solution later.
- Recall that z (in the optimizer) is the collection of all relevant variables in the optimization problem as described in the lectures. In other words, the last entry in z, `z[-1]` will correspond to t_F.

- The main part of the program will therefore be concerned with taking an symbolic environment, as well as the number of discretization steps N, and turning it all into relevant inequality and equality constraints, as well as optimizer target J, consisting of symbolic expressions all expressed using the variables in `z`. This correspond exactly to the procedure outlined in the slides and you can assume variables with similar names play similar roles.

- 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 optimizer is then called and returns a results structure. The optimal $\boldsymbol{z}$ can be accessed as `res.x`. This then needs to be unpacked into the different knot-point variables and we have to construct functions $\boldsymbol{x}(t)$ and control inputs $\boldsymbol{u}(t)$ by interpolating

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 and possible other (stricter) optimizer options. I found this step to be important and it is also in the code by [Kel17]

1.3 Creating the 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 order of the variables will be important, as we will later use it to recover $\boldsymbol{u}_k$ and $\boldsymbol{x}_k$ from the optimizers solution.

The bounds are obtained from two places: Firstly, and the only thing important until the last exercise, simple inequality constraints which are stored in the variable `self.simple_bounds` as a dictionary (the notation should be self-explanatory. Note we use the `Bounds` object from before for simplicity).

Secondly, we will also allow for dynamical path-constraints of the form $h(\boldsymbol{z}) \leq 0$, which are implemented in the `def sym_h` function in the environment, but you do not need to bother with this for now.

Problem 1 *Variables and simple bounds*

Implement the list of symbolic variables, the initial guess, and upper and lower bounds in the code. *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*. Talk to each other why each variable in the bound look the way it does in relationship to the pendulum problem.

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, -6.0, 2.4, 0.0, -3.0, 1.6, 0.0, 0.0, 0.8, 0.0, 3.0, 0.0, 0.0, 6.0,
↪  0.0, 4.0]
z_lb=[3.1, 0.0, -6.0, -6.3, -inf, -6.0, -6.3, -inf, -6.0, -6.3, -inf, -6.0, 0.0,
↪  0.0, -6.0, 0.0, 0.5]
z_ub=[3.1, 0.0, 6.0, 6.3, inf, 6.0, 6.3, inf, 6.0, 6.3, inf, 6.0, 0.0, 0.0, 6.0,
↪  0.0, 4.0]
```

1.4 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 [Her21, eq. (13.20)] (see [Her21, algorithm 20])

Note that the Mayer and Lagrange terms in the cost function are implemented in the symbolic environment as `def sym_cF` and `def sym_c`. You do not need to bother with how these are implemented in details[1].

Problem 2 *Collocation and constraints*

Implement the symbolic expressions for cost function and all constraint for trapezoid collocation. I.e. the lists and variable `ineqC, eqC, J`.

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

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

```
eqC=[-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))]
ineqC=[]
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)
```

1.5 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 $\boldsymbol{z}$.

1.6 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 [Her21, section 13.3.3].

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 [Her21, section 13.3.3].

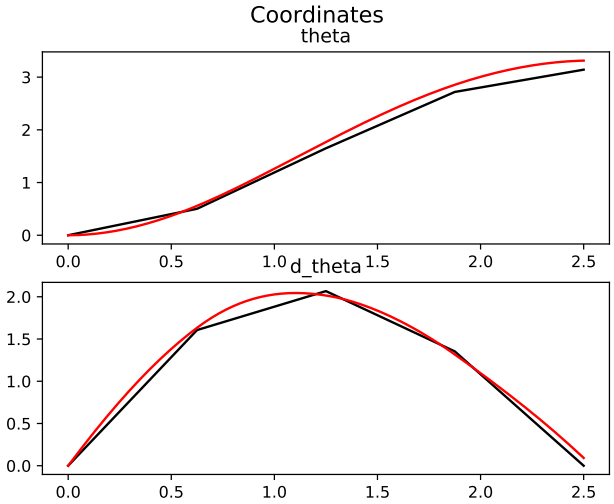
Info: When done, you should get the followin g output:

```
[[ 3.14159265e+00  1.83743989e+00  3.58576757e-01  8.36493697e-09]
 [ 0.00000000e+00 -2.51416359e+00 -9.30895320e-01 -5.01240982e-08]]
```

as well as the figure shown next

1.7 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.

1.8 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 $t_0, \ldots, t_{N=40}$, and initialize a guess as

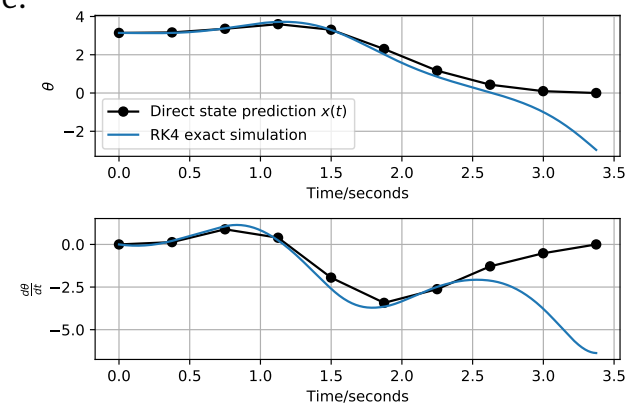
$$\boldsymbol{x}^{\text{guess}}(t_k) = \boldsymbol{x}^{N=10}(t_k). \tag{1}$$

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 $k - 1$.

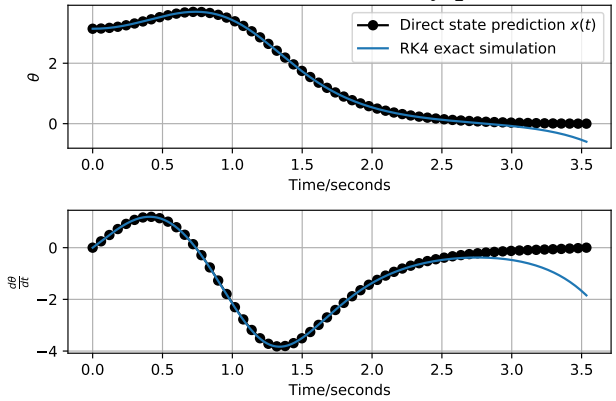
Problem 5 *Guess and iteration*

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.

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

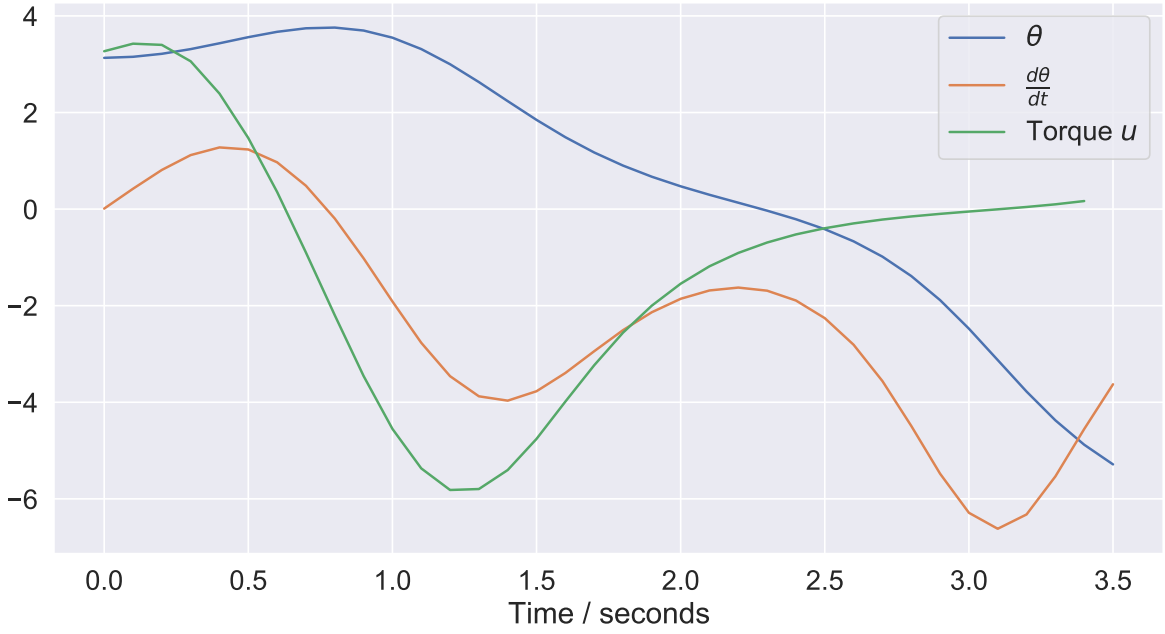
Direct optimization is perhaps the method which is the least well-suited for the agent-environment interface we use in this course, however, we can certainly still build an

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 1.8 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.

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.



3 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 from the paper, but this might be due to us using different dynamics (as discussed in the code we are using physically more correct dynamics). 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 $20N$.

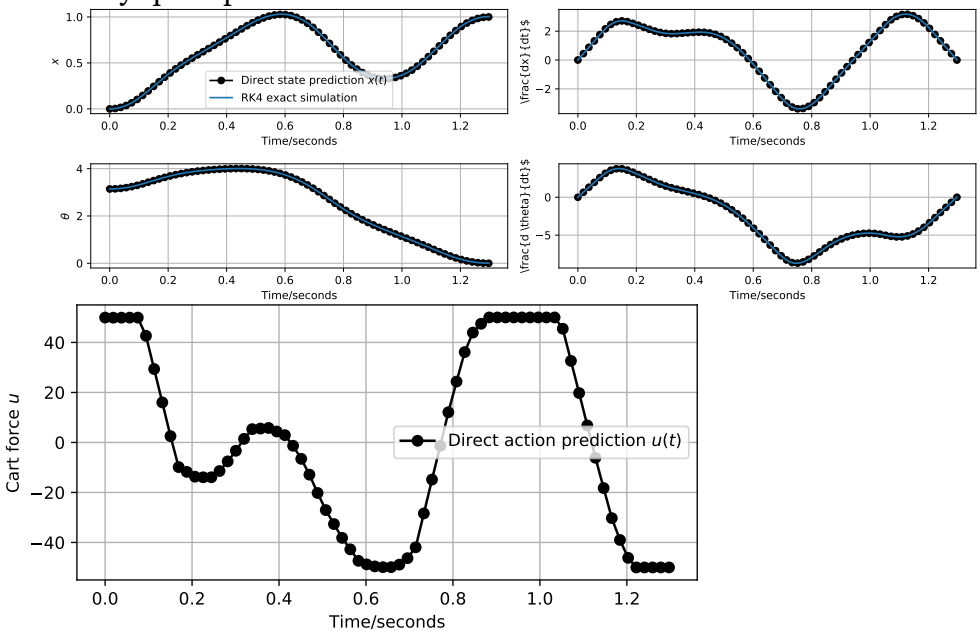
The constraints you have to implement is that $x(t)$ is between `-2*dist` and `2*dist`, $u(t)$ is between the maximum/minimum force, x starts up hanging downwards, i.e. that `simple_bounds['x0']` corresponds to $x = \ldots x = \dot{\theta} = 0$ and $\theta = \pi$ and ends up standing upright at $x =$ `dist`, i.e. that `simple_bounds['xF']` corresponds to $x =$ `dist` and $\ldots x = \theta = \dot{\theta} = 0$.

Problem 7 *Cartpole minimum time swingup task*

Complete the problem specification in the code (i.e. insert the correct boundary constraints). 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 using the Bounds object, but recall that equality constraints can be obtained by letting the upper/lower bounds agree.

The script will use 4-steps of 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.

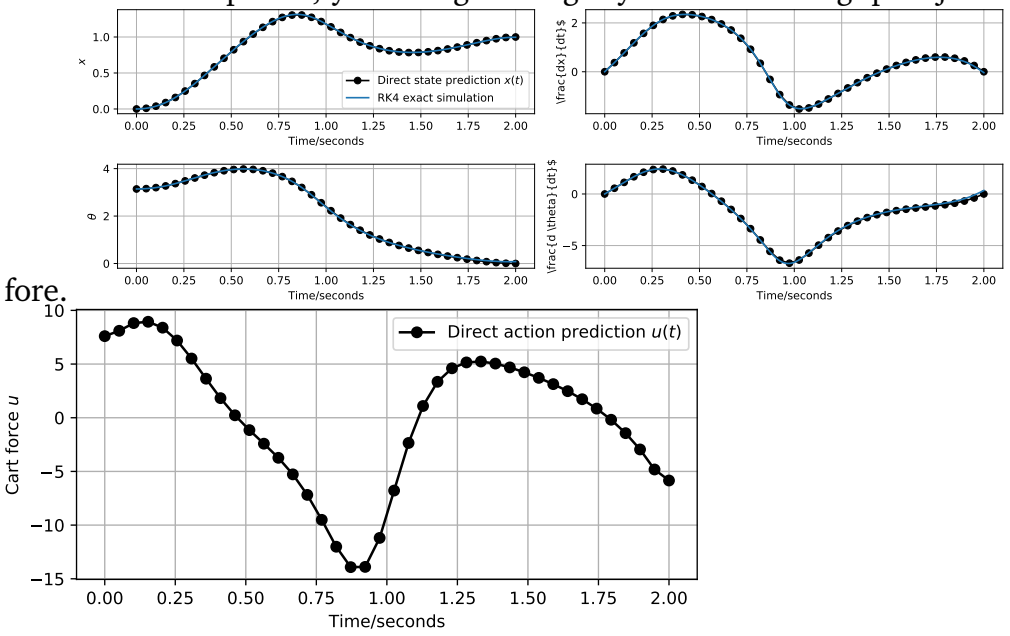
3.1 The Kelly-swingup task ✸

As an additional example, we will consider the (alternative) cartpole swingup task from [**?**] (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 [**?**, section 6] and [**?**, Appendix E, table 3]. The task is similar to the one in section 3, 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. It should only be a matter of supplying the right parameters to the environment and you should be all set. Note the problem appears relatively easier than the previous one numerically.

Info: Once completed, you will get a slightly different swingup trajectory from before.



4 Brachiostochrone problem (`direct_brachistochrone.py`) ✸

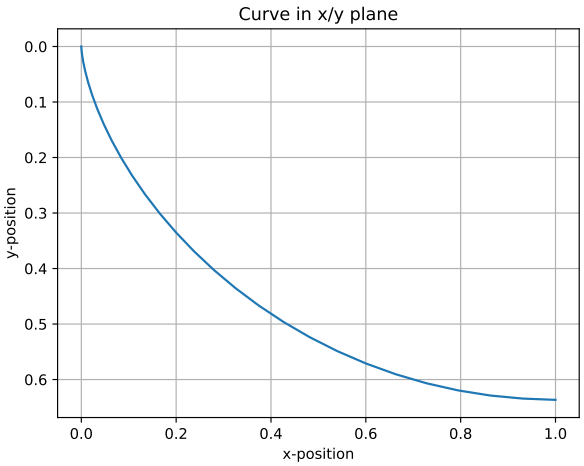
We will now consider the Brachiostochrone problem as discussed in the lectures. To complete this exercise you have to implement the dynamical system corresponding to the problem as described in the lecture slides.

Problem 9 *Brachiostochrone problem*

Complete the dynamical system specification for the Brachiostochrone problem as shown in lecture 4 eq. (**??**) in the file `ct_brachistochrone.py` and run the main simulation script. Note you do not need to modify the direct method for this part of the problem.

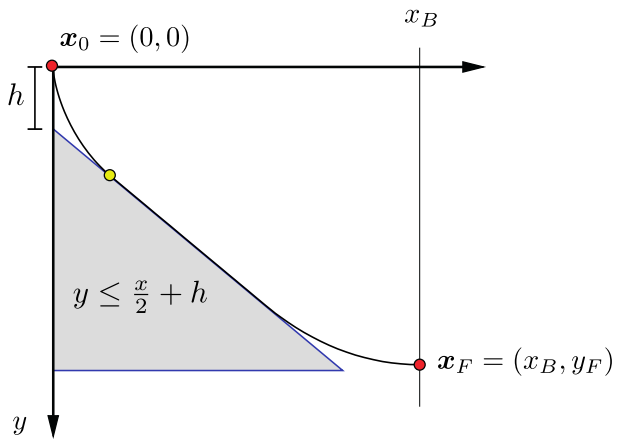
Info: Note you need to implement a cost function corresponding to minimum time, but this was just what you did for the carpole problem.

When done you should get the classical solution to the brachistochrone problem shown below:



4.1 Brachiostochrone problem with constraints (`direct_brachistochrone.py`)

The Brachiostochrone problem has a solution which can be found using variational methods, however if we add a constraint this is no longer the case. The constraint we will consider will be that the bead cannot pass through a slanted wedge indicated by the gray color in the following figure:

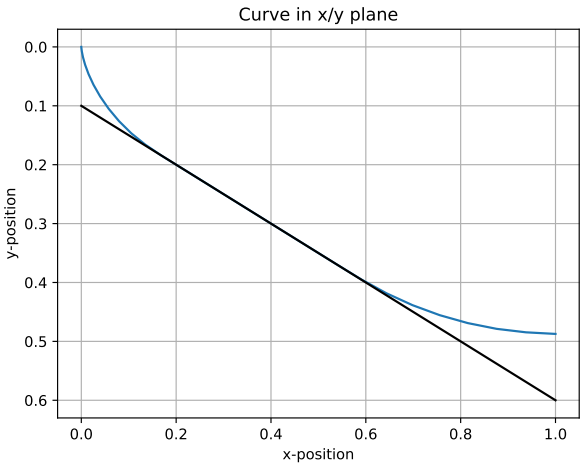


This constraint in itself may not seem very interesting, but dynamical constraints can be expected to arise in many places in robotics to encode that e.g. a robots limbs cannot intersect, that cars should stay on roads, etc.

Problem 10 *Brachiostochrone problem with a dynamical constraint*

This is a one-line addition to the script which implements the brachistochrone problem (see `def sym_h`), but you will also have to modify the main simulation script `direct.py` to add this inequality constraint; once more, it should be an one-line addition.

Info: When done you should get the following solution to the modified problem:



References

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

[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**).