02465 Project: Part 2

Tue Herlau
tuhe@dtu.dk

June 10, 2022

Formalities

- The deadline for this report is March 31st before 23:59.
- Submission of reports happen on DTU learn
- You can work in groups of 1, 2 or 3 students (but not 4)
- Collaboration policy: It is not allowed to collaborate with other groups on this project, except for discussing the text of the project with teachers and fellow students enrolled on the course in the same semester. Under no circumstances is it allowed to exchange, hand-over or in any other way communicate solutions or parts of solutions to the project to other people. It is not allowed to use solutions from previous years, or solutions found on the internet or elsewhere.
- You can (and probably should!) use code from the *exercises* when you solve the project, for instance the dynamical programming algorithm. The exercises may be solved with help from teachers or fellow students, and you can make use of the solutions I make available. However, you are not allowed to copy or share exercise code directly between groups or make solutions publicly available. This will ensure there is no accidental copying of projects.
- Your overall evaluation will be based on your written answers and your UNITGRADE score. They will be weighted based on an assessment of the required work.

Preparing the hand-in:

Hand in these three files (please do not hand in a `.zip` file as this confuse DTU learn):

A `.tex` file with your written answers: Prepared this by modifying the template in `irlc/project2/Latex/02465project2_handin.tex` . Simply write your answers where it says **YOUR SOLUTION HERE**. I recommend keeping the layout as it is.

A `.pdf` file corresponding to this `.tex` file

A `.token` file containing your python-solutions: Generate this file by running the script `irlc/project2/project2_grade.py` . It is very important you do not modify this file.

Contribution table

To make sure your final grade is properly individualized, each students contribution to the report must be clearly specified. Therefore, for each section or problem (or part of a problem), specify which student was responsible for it in the table in the template. A report must contain this documentation to be accepted. The responsibility assignment must be individualized. This means for reports made by 3 students: Each section must have a student who is 40% or more responsible. For reports made by 2 students: Each section must have a student who is 60% or more responsible. Please keep in mind this is an external requirement and it has to be that way. Ask me if you are in doubt about how to do this.

Code hand-in:

- Please keep the structure of the `irlc`-folder. All of your code which is specific to this report should be in the `irlc/project2/` directory. Solutions which use code outside the `irlc` folder cannot be verified and therefore cannot be evaluated. You can (of course) call, re-use or re-purpose any exercise code, including my solutions.
- If you wish to use additional third-party libraries please discuss them with me first to ensure you are on the right track, and make sure I can verify your solutions
- Breaking or tampering with the UNITGRADE framework, for instance by reporting a false number of points or making your solution unverifiable, is potentially cheating. Code build by reverse engineering specific tests and simply returning the values which makes them pass will not get credit and may also need to be treated as a cheating-attempt.
- That asides, this is not a programming course: Strange, long, undocumented and downright disturbing solutions will be evaluated simply based on whether they work or not.

Overall hints:

- You have free reins when it comes to solving the problems. However, they are often easier if you look at (and use) code from the exercises.
- You can put your code in multiple files if you feel like it.
- Although the script `irlc/project2/project2_grade.py` is used to generate the `.token`-file, I recommend running the normal version when you develop the code. You can run it by right-clicking on it from pycharm. I have tried to include tests which examine if you return the right data structures and so on. Use the debugger on the tests that fail, starting with the simplest tests.
- **I have added a video on `https://video.dtu.dk` on how to easily integrate the tests with the UI in Microsoft VSCode**.

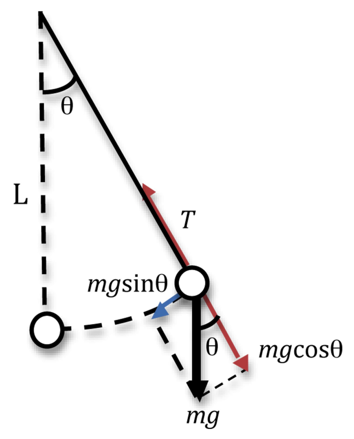

Figure 1: Master Yodas pendulum

1 Master Yodas pendulum (`yoda_part1.py`)

Yodas pendulum hang on a string of length L, at an angle to vertical of θ and with mass m. As part of Padawan training, Yoda will start the pendulum in position $\theta = 1$, release it, and you have to use THE FORCE u to stop the pendulum (see fig. 1).

Using Newtons laws, it is a stable of high-school physics to derive the equations of motion to be[1]

$$u - mg\sin(\theta(t))L = mL^2 \frac{d\theta(t)}{dt^2} \tag{1}$$

If we use that $\sin(\theta) \approx \theta$ when θ is small we can write this as

$$\ddot{\theta} = -\frac{g}{L}\theta + \frac{1}{mL^2}u \tag{2}$$

This equation will be our starting point and can from now on be taken as the true equation of motion for θ. In the following exercises, we will use the following parameters

```python
# yoda_part1.py
g = 9.82       # Gravitational force in m/s^2
m = 0.1        # Mass of pendulum in kg
Tmax = 8       # Planning horizon in seconds
Delta = 0.04   # Time discretization constant
```

The string length is by default $L = 0.4$, however Yoda can extend/shorten it and so it should be considered variable.

[1]See `https://www.acs.psu.edu/drussell/Demos/Pendulum/Pendulum.html`

Problem 1 *Formulate Yodas pendulum as a linear problem*

In the absence of a cost-function, Yodas pendulum can be written as a linear system of the form

$$\dot{\boldsymbol{x}} = A\boldsymbol{x} + B\boldsymbol{u} \tag{3}$$

Where $\boldsymbol{x} = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}$.

- Define A and B below in terms of g, L and m mathematically
- Implement the function `get_A_B(g, L, m)` which return the two matrices as two numpy `ndarray`

Info:

- The two should be consistent, allowing you to verify your result

Answer:

$$A = \begin{bmatrix} \cdots \\ \cdots \end{bmatrix} \tag{4}$$

$$B = \ldots \tag{5}$$

YOUR SOLUTION HERE

In the rest of the questions in this section, it is assumed the pendulum model has been discretized using (exact) **exponential integration**, and simulated with a near-exact RK4 simulator.

Problem 2 *Build a PID controller to stop Master Yodas pendulum*

Implement a PID controller which can stop master Yodas pendulum when released from an angle $\theta = 1$. Describe your value of the target x^* as well as K_P, K_I and K_D below, and insert a picture containing trace plots of $\boldsymbol{u}(t)$ and $\boldsymbol{x}(t)$. Briefly describe/discuss your choices.

Info:

- According to Yoda, when you can stop a pendulum with THE FORCE, it is in fact the same as stopping something much heavier, like a speeder or a **locomotive**. This is supposed to be some kind of hint.
- The plot (here and elsewhere) should have labels and be readable. Look at the plots produced by `plot_trajectory` during the exercises. Using `plot_trajectory` is obviously fine.
- There are many possible choices; as long as you bring the pendulum to rest within the time period your choice is fine.
- The linear models we look at in the exercises are all (automatically) discretized using exponential integration by the framework; tl;dr, simply building a small linear model and re-use parts of the framework can probably save a lot of hassle. Good places to start is with the locomotive, or with the harmonic oscillator model.

Answer:

$$x^* = \cdot, \quad K_d = \cdot, \quad K_p = \cdot, \quad K_I = \cdot$$

Insert your figure here

YOUR SOLUTION HERE

Using LQR as a way to set parameters in a PID controller: The rest of the exercise will look at a classical result in control theory, namely a connection between PID and

LQR control.

To do this, we will now treat the discretized problem as a LQR problem with diagonal cost-matrices $Q_k = 0.1I$ and $R_k = 100I$ (and no terminal cost term, i.e. $Q_N = \mathbf{0}$).

Problem 3 *Optimal LQR to stop the pendulum*

- Implement the `cost_discrete(x_k, u_k)` function. It should implement the discrete linear-quadratic cost term $c_k(x_k, u_k)$.
- Then implement the function `part1(L)`, which takes the string length L as an input, and returns the control law $L_0, \boldsymbol{l}_0$, as well as the action-sequence obtained by applying **just** the (L_0, $\boldsymbol{l}_0$) control law to the pendulum at **each step** over the full time horizon.

Info:

- If you use the framework, the first problem will allow you to test if you specified the cost-function correctly (see hints in code).
- Remember the control law is not time dependent in this problem. I.e. only use L_0 and $\boldsymbol{l}_0$ at all time steps. But you can still let yourself be inspired by the LQR agent, as this would be a very small change to the existing implementation. The Boing-example is perhaps a good place to start.
- To simulate the action-sequence, you can use the framework with the `train`-function. The action sequence is part of the trajectory.

The way a PID controller select actions, and the way an infinite-horizon optimal LQR controller does so is in fact very similar if you just write them up as function $u_k = \cdots$. In other words, the control matrices L_0, $\boldsymbol{l}_0$ from LQR control can be used to specify the constants K_P, K_I and K_D in a PID controller, so that they are *nearly* the same. The following exercise will examine this:

Problem 4 *PID corresponding to optimal controller*

Implement the function `part2(L)`. Using the above observation, specify the parameters K_P, K_I, K_D and x^* in a PID controller as suggested by L_0 and $\boldsymbol{l}_0$. Your function should also return the action sequence obtained when simulating your PID controller on the actual pendulum (same format as the previous exercise).

Info:

- Your code does not have to be fully general – Just make it work for the pendulum when the string length L is varied. Some of the parameters you return will just be 0.
- The result is simple
- To get started, write up the control law $u_k = \ldots$ in the two cases, and use this to match the constants. Think about what the derivative-term in the PID controller might correspond to in the pendulum case.
- One constant is easy, the other requires a little work. Unitgrade will let you know if you are on the right track. Look at the numbers if you are stuck.

Problem 5 *Reflect on PID controller*

Insert the plot of the two action sequences as generated by the script. They should look similar with a single notable difference. Comment on why this difference occur, what the practical importance is, and what you, as an engineer, would (practically) do in a real-world setting if you had to deploy this method.

Info:

- Think practically. What tends to damage a mechanical system?

Answer:

Insert your figure here

The two action sequences differs because... **YOUR SOLUTION HERE**

2 Yodas pendulum revisited `yoda_part2.py`

In this problem, we will investigate what happens with the pendulum, described in eq. (3), when no external force is applied to it ($B = \boldsymbol{u} = 0$) and different types of discretization schemes, Euler and Exponential Integration (EI), are used. The point of the exercise will be to show mathematically that one discretization scheme is numerically stable and the other is not.

Problem 6 *State at a later time*

Assume the system has been discretized using a time step of Δ to give states $\boldsymbol{x}_0, \boldsymbol{x}_1, \dots, \boldsymbol{x}_N$.

1. Prove that when Euler discretization or exponential discretization is used to discretize the system the final state can be written as:

$$\text{(Euler discretization):} \quad \boldsymbol{x}_N = \tilde{A}_0^N \boldsymbol{x}_0 \tag{6}$$

$$\text{(Exponential discretization):} \quad \boldsymbol{x}_N = A_0^N \boldsymbol{x}_0 \tag{7}$$

2. then give an (analytical) expression for the two matrices $\tilde{A}_0$ and A_0 (i.e., express them in terms of known quantities such as L, g and Δ)
3. implement the functions `A_euler` and `A_ei` to compute both matrices in python.

Info:

- As to the first part, start with $N = 1$ or $N = 2$ to get the general idea. Lecture 5 actually contained a very relevant hint.
- As to the form of A_0 and $\tilde{A}_0$, look in the lecture notes. For the exponential integration part, you need to use the matrix exponential. Code to compute it is imported and described at the top of the file.

Answer:
To solve the first part, we can write $\boldsymbol{x}_N = \dots$

As for the second part we get:

$$\tilde{A}_0 = \cdots, \quad A_0 = \cdots \tag{8}$$

YOUR SOLUTION HERE

Problem 7 *State at a later time II*

According to the previous problem, we can in fact write

$$\text{(Euler discretization):} \quad \boldsymbol{x}_N = \tilde{M}\boldsymbol{x}_0 \tag{9}$$
$$\text{(Exponential discretization):} \quad \boldsymbol{x}_N = M\boldsymbol{x}_0 \tag{10}$$

for two matrices M and $\tilde{M}$. Implement the functions to compute both matrices in python as `M_euler` and `M_ei`

Info:

- The intention with a problem such as this is to show you can test derivations numerically, i.e. the matrices M and $\tilde{M}$ can in fact be computed explicitly. When in the following you are asked to compute e.g. their Eigenvalues, you can test your result numerically to know you are on the right track. This kind of self-check is in my experience something that can help you speed up derivations by a great deal as it will let you know if you made simple mistakes.
- The imports at the top of the file can simplify the solution a bit.

In the next question, we will just focus on the Euler discretization matrices $\tilde{A}_0$ and $\tilde{M}$ and the relationship between the four Eigenvalues of these matrices. To do so, we will use the following special case of the Eigendecomposition: Let A be a 2×2 matrix with two linearly independent unit-length Eigenvectors satisfying $A\boldsymbol{v}_1 = \lambda_1\boldsymbol{v}_1$ and $A\boldsymbol{v}_2 = \lambda_2\boldsymbol{v}_2$. Assume we form the matrices $Q = \begin{bmatrix}\boldsymbol{v}_1 & \boldsymbol{v}_2\end{bmatrix}$ and $\Sigma = \begin{bmatrix}\lambda_1 & 0 \\ 0 & \lambda_2\end{bmatrix}$. It then holds that

$$A = Q\Sigma Q^{-1}. \tag{11}$$

The reverse is also true: If a Matrix has this representation, the Eigenvalues are the diagonal of Σ.

Problem 8 *Eigenvalues and powers*

Use the Eigendecomposition to show there exists a simple relationship between the eigenvalues of $\tilde{A}_0$ and the eigenvalues of $\tilde{M}$ involving N.

Info:

- If stuck, always start with $N = 1$ and $N = 2$.
- You need to use something about eigenvectors, specifically Q, you learned in Mat1, and which was repeated in 02450 in the context of PCA
- The matrix power is simple $A^n = AAA \cdots A$ (matrix product of n terms)
- Test your analytical answer numerically.
- The result turns out to be very simple.

Answer:
Assume λ_1, λ_2 are the eigenvalues ... then the Eigenvalues of M is ... similarly for $\tilde{M}$ ... **YOUR SOLUTION HERE**

Problem 9 *Analytical expression of Eigenvalues using Euler discretization*

Derive an analytical expression of the Eigenvalues λ_1 and λ_2 of $\tilde{A}_0$.

Info:

- Don't be scared if the result looks **complex**.
- This is really just a Mat1 problem.

Answer:
... we get a characteristic polynomial of ... and therefore it follows from Mat1 that the two Eigenvalues are ... **YOUR SOLUTION HERE**

Let A be a matrix. The matrix norm is defined as $\|A\| = \max_{\|\boldsymbol{x}\|=1} \|A\boldsymbol{x}\|$. It holds that if λ_1, λ_2 are the two Eigenvalues of A that

$$\|A\| = \max\{|\lambda_1|, |\lambda_2|\}. \tag{12}$$

where $|\lambda|$ is the absolute value and λ may be a complex number.

Problem 10 *Bound using Euler discretization*

Use the Matrix norm, and the previous problem to derive a tight upper bound for $\|\boldsymbol{x}_N\|$ assuming $\|\boldsymbol{x}_0\| = 1$ and the problem has been discretized using Euler integration. Implement the bound as the function `xN_bound_euler(g, L, Delta, N)`.

Info:

- Upper-bound means you have to find a function F such that $\|\boldsymbol{x}_N\| \leq F(g, \Delta, L, N)$ for all $\boldsymbol{x}_0$ so that $\|\boldsymbol{x}_0\| = 1$. That the bound is tight means you cannot find a function with smaller values.
- You should use the property of the Matrix norm listed above in conjunction with your answers to the previous questions
- The solution can be written as a single line; start with the given expression and try to use the things you have seen so far one at a time.

Answer:
Using Euler discretization we get the upper bound:

$$\|\boldsymbol{x}_N\| \leq \cdots$$

YOUR SOLUTION HERE

Problem 11 *Matrix norm of Exponential discretization (harder)*

Repeat the same steps you did before to derive a tight upper-bound of $\|\boldsymbol{x}_N\|$ assuming $\|\boldsymbol{x}_0\| = 1$ and that we are using exponential integration. Use the hints below. When done, implement the function as `xN_bound_ei(g, L, Delta, N)` and check your result

Answer:
Using exponential discretization we get an upper bound of:

$$\|\boldsymbol{x}_N\| \leq \cdots$$

YOUR SOLUTION HERE

Info:

- Most of the problem is similar to the previous one. The part which is different has to do with computing eigenvalues. Remember that you can compute eigenvalues numerically to get ideas and check you are on the right track.
- If you follow the same steps as before, you will see that your solution way back in eq. (7) means you will have a matrix of the form e^A to deal with, where A is *some* 2×2 matrix you *can* compute the Eigenvalues of. Compute these eigenvalues first.
- Note you can always write $A = Q\Sigma Q^{-1}$ using the Eigendecomposition; you don't need to know what Q is, but you can find Σ
- Then you probably need the eigenvalues of e^A (in a situation where you know the eigenvalues of A). This may seem impossible, but keep the definition of the Matrix exponential and the following in mind:
- **Taylor** Swift has a **series of** top hits that caused **exponential** growth of her career
- Everything will involve powers of the form A^n; you have dealt with this situation before. Be inspired by eq. (11) and re-write these
- Use linear algebra to simplify things into an expression of the form $Q(\text{sum-of-matrices})Q^{-1}$. Use the Taylor-series trick in reverse on the sum to simplify the sum of matrices to something neat. Then use the result to argue you know the Eigenvalues.
- The end-result will be simple; in fact, it will be simpler than before, and with the sufficient hind-sight it might even seem kind of obvious.
- This is one case where a hard-coded solution is fine.

Problem 12 *Stability*

What do the bounds on $\|\boldsymbol{x}_N\|$ (using Euler discretization) and $\|\boldsymbol{x}_N\|$ (using exponential discretization) tell you about the stability of Euler discretization and exponential discretization?

Answer:
YOUR SOLUTION HERE

3 R2D2 and control (`r2d2.py`)

In this problem, we will consider control of r2d2, who can be seen as an example of a primitive car model. R2D2 is characterized by an (x, y) location and the angle his

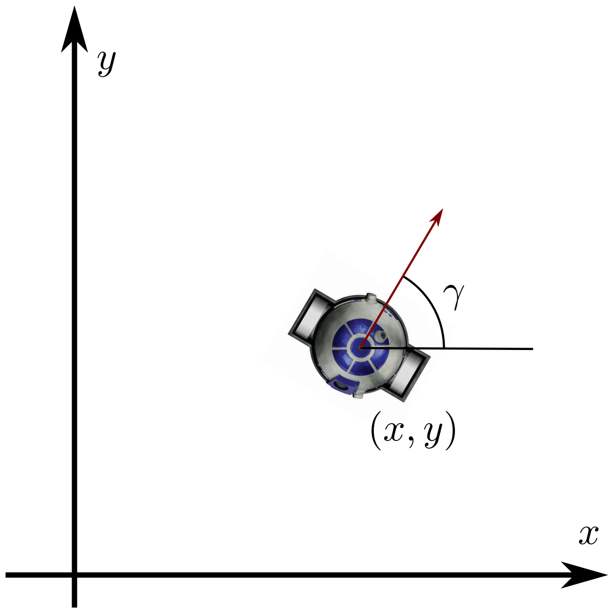

Figure 2: R2D2, as seen from above. The red arrow indicates the direction of motion.

direction of motion makes with the x-axis (this is also called the yaw) denoted by γ, see fig. 2.

R2D2 can travel forward and spin around on a dime in place. In other words, the available controls are the linear velocity v, in direction of the red arrow, as well as the angular velocity ω, which controls how fast R2D2 spins around in place (i.e., the rate at which the yaw γ changes). Taken together, we can describe R2D2 using state and control vectors

$$\boldsymbol{x}(t) = \begin{bmatrix} x(t) \\ y(t) \\ \gamma(t) \end{bmatrix}, \quad \boldsymbol{u}(t) = \begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix},$$

and the equations of motion are:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t)) = \begin{bmatrix} v(t)\cos(\gamma(t)) \\ v(t)\sin(\gamma(t)) \\ \omega(t) \end{bmatrix} \tag{13}$$

The starting position will always be $\boldsymbol{x}(0) = \boldsymbol{x}_0 = \boldsymbol{0} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$. Your task is to help R2D2 carry out a couple of control-related tasks, all of which involve driving from the starting position $\boldsymbol{x}_0$ to a target position $\boldsymbol{x}^*$.

Problem 13 *Discretization*

We apply Euler discretization to the system with time constant Δ to get $\boldsymbol{x}_{k+1} = \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{u}_k)$. Give an explicit expression for $\boldsymbol{f}_k$ below.

Answer:

$$\boldsymbol{x}_{k+1} = \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{u}_k) = \begin{bmatrix} \cdots \\ \cdots \\ \cdots \end{bmatrix}$$

YOUR SOLUTION HERE

Info:

- This is the same operation we have used to discretize all systems so far, i.e. what the framework does by default.

Problem 14 *Linearization*

The next task is to linearize the system around a particular, fixed point $\bar{\boldsymbol{x}}$, $\bar{\boldsymbol{u}}$. Give an explicit expression for the linearized system which takes the form

$$\boldsymbol{x}_{k+1} \approx A\boldsymbol{x}_k + B\boldsymbol{u}_k + \boldsymbol{d}$$

when the system is linearized around $\bar{\boldsymbol{x}} = \begin{bmatrix} 0 \\ 0 \\ \theta \end{bmatrix}$ and $\bar{\boldsymbol{u}} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$.

Answer:

$$\boldsymbol{x}_{k+1} \approx \begin{bmatrix} \cdots \\ \cdots \\ \cdots \end{bmatrix} \boldsymbol{x}_k + \begin{bmatrix} \cdots \\ \cdots \\ \cdots \end{bmatrix} \boldsymbol{u}_k + \begin{bmatrix} \vdots \end{bmatrix}$$

YOUR SOLUTION HERE

Info:

- Remember to include Δ, and that $\bar{\boldsymbol{u}}$ is not zero!.
- Linearization was discussed in lecture 6; use the general expression and compute the derivatives of $\boldsymbol{f}_k$ using what you have learned in Mat1.

Implementation: General comments You are free to use the framework or not. If you choose to use the framework, I have included a bit of boiler-plate code to set up a discrete model/environment and all you need is to complete the continuous model. If you use the code, take care you understand the role of `x_target` and `Q0`! You need to set their values in the following exercises.

Problem 15 *Unitgrade self-check*

Implement the two functions to compute the Euler discretization from problem 13 (`f_euler`) and linearization matrices from problem 14 (`linearize`). The functions should return numpy arrays.

Info:

- The recommended way of doing this exercise is to specify R2D2 as a `ContiniousTimeSymbolicMod` where you specify the dynamics as you have seen examples of in the exercises, and from that define a `DiscretizedModel` and finally a `ContiniousTimeEnvironment`. When you have done that, you can use function from the `DiscretizedModel` to solve the exercise very simply, and you will get a self-check you have implemented the model correctly.
- Use unitgrade to make sure you get the result right. If you don't, check which matrices/entries in those matrices gives you problems
- We have actually worked with the linearization procedure during one of the exercises.
- Once you are done with this method, you can check (by inserting specific values and checking the result agrees) that your two previous expressions are 100% right. This is something I always try to do when I have to calculate something non-trivial.

We will now consider the problem of actually reaching the target location $\boldsymbol{x}^*$ at a minimal effort. To do so, we construct the following quadratic cost-function which depends on two parameters Q_0 and $\boldsymbol{x}^*$:

$$\int_0^{t_F} \left(\frac{1}{2} Q_0 \|\boldsymbol{x}(t) - \boldsymbol{x}^*\|^2 + \frac{1}{2} \|\boldsymbol{u}(t)\|^2 \right) dt \tag{14}$$

To simplify things, we will always consider a planning horizon of $t_F = 5$ seconds.

3.1 Optimal planning

Our first approach will be based on optimal planning using direct collocation. In this approach, we will handle the problem of reaching the destination as an equality constraint $\boldsymbol{x}(t_F) = \boldsymbol{x}^*$, and therefore set $Q_0 = 0$. The cost function is in other words simply a quadratic cost function with $R = I$:

$$\int_0^{T_F} \frac{1}{2}\|\boldsymbol{u}(t)\|^2 dt. \tag{15}$$

Problem 16 *Optimal planning*

Implement the method `drive_to_direct(x_target, plot)`. The function should plan a path to the end-point $\boldsymbol{x}^* = \begin{bmatrix} 2 & 2 & \frac{\pi}{2} \end{bmatrix}^\top$ using direct collocation (see above). The boolean variable `plot` controls if the method plot the resulting state trajectory or not.

The method should return a $N \times 3$ `ndarray` of the state-trajectory. When you call the direct collocation method, first use a grid of size $N = 10$, then $N = 20$, and finally $N = 40$. Below, insert a plot of both the states trajectory, i.e. time along the x-axis and the coordinates of $\boldsymbol{x}$ along the y-axis, as well as a plot of R2D2s (x, y) position.

A

Answer:

Insert your figure here

Insert your figure here

Info:

- Easiest approach is to use the method we already implemented. I used the `DirectAgent`. Look at the examples.
- The trace plot can (here and elsewhere) be made using a call such as `plot_trajectory(traj[0],` but you are free to write your own plot code.
- The code for the second plot is already included in the script.
- If you use the framework, remember to set the `simple_bounds` variable. Check the various examples to see how it is used. Once implemented, you can use the `guess` =function to get a reasonable initial guess (see examples from week 5).

3.2 Simple Linearization

Although direct methods should solve the problem optimally they are brittle. Our first attempt at solving this problem will involve simple linearization using [Her21, algorithm 23]. The dynamics is discretized as before, and the cost function is discretized in the usual manner to be:

$$c_k = \Delta\left(\frac{1}{2}Q_0\|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2 + \frac{1}{2}\|\boldsymbol{u}_k\|^2\right) \tag{16}$$

When we apply simple linearization, the system is linearized around $\bar{\boldsymbol{x}} = \boldsymbol{0}$ and $\bar{\boldsymbol{u}} = \boldsymbol{0}$. Use this to derive an LQR controller for the linearized problem (by planning on a horizon of $N = 50$ steps using the linearized dynamics and cost-matrices), and use this to plan the future actions.

Problem 17 *Control using simple linearization*

Implement the simple linaerization procedure as the function `drive_to_linearization(x_target, plot)`. The `plot`-variable should control whether we are doing plotting or not. The function should plan using the simple linarization method to reach `x_target` and return the obtained states when that plan was carried out in a simulation of the environment (i.e. using RK4 on a fine grid).

When done, test the method using first a simple problem consisting of driving forward for two meters, and then the problem we are interested in, where R2D2 drive to $(2, 2)$ and face north:

- $\boldsymbol{x}^* = \begin{bmatrix} 2 & 0 & 0 \end{bmatrix}^\top$
- $\boldsymbol{x}^* = \begin{bmatrix} 2 & 2 & \frac{\pi}{2} \end{bmatrix}^\top$

Insert figures below of the obtained state trajectories. The first case should work perfectly, the second case will fail. Explain the result.

Answer:

Insert your figure here

Insert your figure here

Intuitively, the second case fails because... **YOUR SOLUTION HERE**

Info:

- We solved a problem very similar to this in the exercises. You can re-use the solution without writing a new agent.
- You can turn on rendering of R2D2 by using the supplied rendering function in the `utils.py` file to see what happens.

3.3 Model-predictive control and Iterative linearization

Simple linearization fails to solve the problem, and full iterative LQR is difficult to implement and may fail to converge. In this problem, we will see if we can get away with something which is a lot simpler.

What we are going to do is, in each step, to apply the simple linearization procedure we tested in problem 17 (but where we expand around the current state) and then replan in the next step. Specifically, the method is as follows:

- Initialize $\bar{\boldsymbol{u}} = \boldsymbol{0}$
- In step k, linearize round $\bar{\boldsymbol{x}} = \boldsymbol{x}_k$ and $\bar{\boldsymbol{u}}$ to get controller output $\boldsymbol{u}_k = L_0 \boldsymbol{x}_k + \boldsymbol{l}_0$ by planning on a $N = 50$ horizon using the linearized dynamics (i.e., similar to the simple linearized agent)
- Set $\bar{\boldsymbol{u}} = \boldsymbol{u}_k$ and repeat

Your task is to implement this method and check the outcome.

Problem 18 *MPC*

Apply iterative LQR to solve the problem by implementing the function `drive_to_iterative_linearization(x_target, plot)`. Insert the plot of the trajectory the script generates below, and comment on why you think it performs better.

Answer:

Insert your figure here

Iterative linearization solves the problem because... **YOUR SOLUTION HERE**

Info:

- I solved this by writing a new `Agent`. Since you are re-using the iterative linearization approach, you can re-use existing functionality.

References

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