TSFS12 HAND-IN EXERCISE 3
Vehicle motion control

August 26, 2022

1 OBJECTIVE

The objective of this hand-in exercise is to explore and implement two approaches to control of a ground vehicle, pure pursuit control and state-feedback control, with the aim to follow a planned path. It will also illustrate the effects of introducing non-linear feedback laws. A simulation environment is provided and you are expected to explore properties of your solutions and summarize in appropriate plots.

2 PREPARATIONS BEFORE THE EXERCISE

Before doing this exercise, make sure you have read and understood the material covered in

- Lecture notes on Control of autonomous vehicles I: Ground vehicles
- Paden, Brian, et al. "*A survey of motion planning and control techniques for self-driving urban vehicles*". IEEE Transactions on intelligent vehicles 1.1 (2016): 33-55. A good but slightly advanced text and therefore many details are included in the lecture notes.
- Coulter, R. Craig. "*Implementation of the Pure Pursuit Path Tracking Algorithm*" (1992)[1].
- This exercise description, including the appendices.

To get the background code for the exercise, get the latest files from `https://gitlab.liu.se/vehsys/tsfs12` and familiarize yourself with the provided code. Start by running the code in the lab template file `main`.

You are free to choose in which language to implement your planners, there are skeleton code available in Matlab and Python. The Python skeleton files will be available in both Notebook format (`.ipynb`) (`https://jupyterlab.readthedocs.io/`) or as python script files (`.py`) and you can choose to work with the format you prefer. This document is written with reference to the Matlab files, but pointers towards corresponding Python equivalents are provided in the text. In the student labs, all Python packages needed are pre-installed in the virtual environment activated by

```
source /courses/tsfs12/env/bin/activate
```

1 But don't look at figure 1, it is badly scaled which makes it difficult to understand; use figures in the lecture slides instead

3 REQUIREMENTS

The objective of the exercise is to implement vehicle controllers tracking a specified planned path, evaluate their properties, and understand when and how they are appropriate. In particular, to explore linear design techniques and be aware of their limitations, and how non-linear techniques are useful.

To complete this exercise you should implement the following controllers

- Pure-pursuit controller
- Linear state-feedback controller with feed-forward term
- Non-linear state feedback

The exercise is examined by submitting the following on the Lisam course page.

1. Runnable code. If your code consist of several files, submit a zip-archive.
2. A short document, that does *not* have to be formatted as a self contained report:
   - answers to the questions, including relevant plots, in Section 5, and other questions that you have encountered during solving this exercise.
   - a concluding discussion.

   Submit the document in PDF format.

It is allowed to discuss the exercises on a general level with other course participants. However, code sharing outside groups is not allowed. Moreover, both students in the group should be fully involved in performing all exercises, and thereby be prepared to answer questions on all subtasks.

See Appendix D for the extra assignment needed for higher grades. The extra assignment is done individually and is submitted by a document answering the questions, including suitable figures, plots, and tables. The document is to be submitted in PDF format. The code should also be submitted as a zip-archive. There is only pass/fail on the extra exercise and the document can not be revised or extended after first submission. It is not required to get everything correct to pass the assignment, we assess the whole submission. You are of course welcome to ask us if you have questions or want us to clarify the exercise.

> Since these exercises examines this course, we would like to ask you not to distribute or make your solutions publicly available, e.g., by putting them on GitHub. A private repository on gitlab@liu is of course fine.
>
> Thanks for your assistance!

4 CONTROLLER IMPLEMENTATION ENVIRONMENT

There are pre-implemented code to help with implementation. One important class is the path that the vehicle should follow. Read Appendix A carefully to understand how the class `SplinePath` works. You define your controllers in a class, which is described in more detail in Appendix B. There are skeleton files available for both `PurePursuitController` and `StateFeedbackController`.

For example, a `PurePursuitController` object is defined as

```
1 controller = PurePursuitController(l, L, ref_path);
```

where `l` is the look-ahead horizon, `L` the car wheel base, and `ref_path` the path the controller should follow. Now that the controller object is defined, define your car object and assign your controller to the car as

```
car = SingleTrackModel();
car.controller = controller;
```

Now, you can simulate your car and controller with the commands

```
Tend = 80;
Ts = 0.1;
w0 = [0, 1, pi/2*0.9, 2]; % (x, y, theta, v)
[t, w, u] = car.simulate(w0, Tend, Ts);
```

where `w0` is the initial state, `Tend` the end time (or until the controller says stop), and `Ts` the sampling time for the controller and also the integration step. This means that the `controller.u()`-method will be called every `Ts` seconds[2].

5 DISCUSSION TOPICS, QUESTIONS

This section summarizes a number of discussion topics and questions you should implement, investigate, and reflect upon to pass this exercise.

5.1 Pure-pursuit controller

Exercise 5.1. In pseudo-code, write down an approach to find the pursuit-point.

For efficiency, make sure you have an simple way of finding the pursuit-point. It is not crucial hat the pursuit point is *exactly* at distance l.

Hint 1: Try to avoid using the `project`-function from Section A since this is a computationally expensive operation. Expect to spend no more than 40 msek (depending on your computer and implementation) in the controller for the provided scenario. If you spend significantly more time than that, reconsider your approach.

Hint 2: It is typically not a good idea to represent the look-ahead horizon circle and check with intersection points of the path.

Exercise 5.2. In psudo-code, write down how to find the x-coordinate, x, of the pursuit-point in the vehicle-local coordinate system. In particular, explain why the first of the vector from the car to the pursuit-point is *not* equal to x.

Exercise 5.3. Implement a pure-pursuit controller to follow the path `ref_path` that is given by the `MiniController`. Experiment with different look-ahead horizons and initial states. Think about what happens if the vehicle is further away from the path than the look-ahead horizon. Comment on how the look-ahead horizon affects control performance and what considerations should be taken when choosing the horizon.

Skeleton files are given by `main.m` and `PurePursuitController.m`. Discuss the results. Figure 1 shows sample controller performance that you should achieve. If you have significantly worse performance, have a closer look at your solution.

2 You can choose sampling interval as you like, but $T_s = 0.1$ is a recommended start.

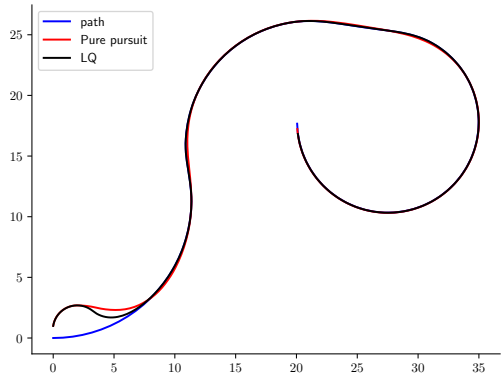

Figure 1: Sample controller performance for pure-pursuit and state-feedback (LQ) controller.

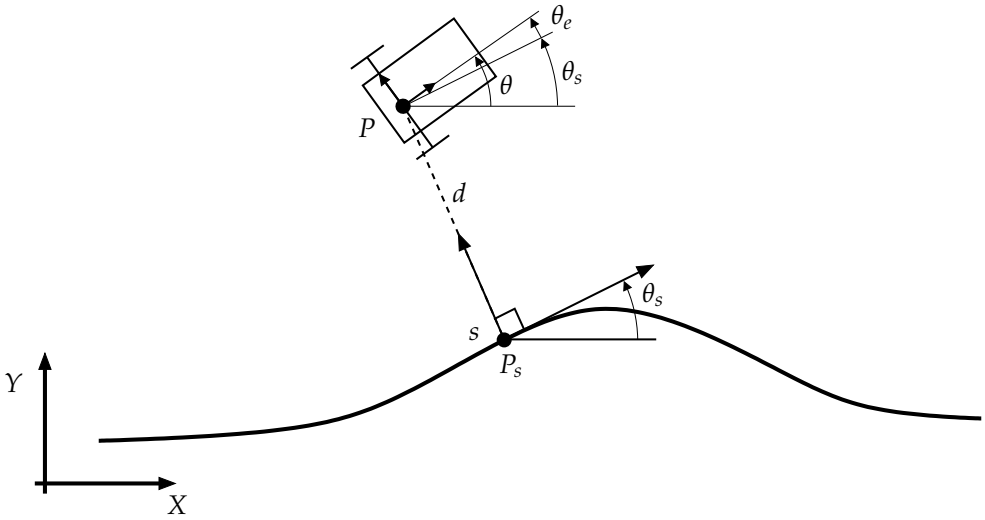

Figure 2: Frenet frame.

5.2 Linear state feedback controller

A next step is to explore a more classical control-oriented approach with state feedback where the path following problem is transformed into a stability problem. A first step is to re-write the vehicle motion equations in the error frame, the Frenet frame, shown in Figure 2.

Exercise 5.4. Write the motion equations for the single-track model in the Frenet frame from the lecture notes, i.e., write down the dynamics of the error dynamics d and θ_e. Linearize the error dynamics around the reference path and around a straight path, i.e., with $c(s) = 0$. This linearized model will be used in the state-feedback design.

Exercise 5.5. Implement a linear state-feedback controller

$$u = u_0 - K \begin{pmatrix} d \\ \theta_e \end{pmatrix} \tag{1}$$

where u_0 is a feed-forward term, K the feedback gain, d the path error (distance from path), and θ_e the orientation error. Important, implement such that u is the desired *curvature*, and then transform into a steering angle for a single-track model using

$$\tan \delta = Lu$$

Use the Frenet frame description in the last exercise to design the feedback-gain. Implement the controller in the skeleton file `StateFeedbackController.m`. If you choose to do an LQR-design, see Appendix C for a quick summary, be sure to consult the lecture slides for comments on implementation. Do note that the equations in Appendix C is in discrete time and the linearization expression in the lecture slides is in continuous time so you need to discretize the linearization. A simple Euler-forward is sufficient here.

Think about a suitable choice of feed-forward term u_0, or stated differently what is a suitable choice of control signal if you are on the path, i.e., when the error terms $d = 0$ and $\theta_e = 0$.

Experiment with your controller, similar to the pure-pursuit controller, and report your findings.

The projection operation of the vehicle position on the path is crucial in this controller, see the `SplinePath` class described in Appendix A.

Exercise 5.6. Describe why the projection operation is difficult for a general path. Also, find an example where the projection is non-unique. Also, read trough the code for the `SplinePath` and get an understanding of how this implementation works.

Exercise 5.7. Describe and explain what happens when you vary the feedback gains or, alternatively, the weights in the LQ-controller.

5.3 Non-linear state feedback controller

The linear feedback controller should perform well in a close vicinity of the path but may experience problems when at a significant distance from the path. The exercise below is intended to illustrate such problems and introduce a non-linear feedback law that mitigates such situations.

Exercise 5.8. To illustrate the problems with a linear feedback law, start your simulation far away from the reference path, for example in the point:

$$(x,\ y,\ \theta,\ a) = (-5,\ 10,\ 0.9\frac{\pi}{2},\ 2)$$

You should observe non-desirable behavior of your vehicle. If not, try to make your controller more aggressive, i.e., make it approach the path faster. Discuss and try to explain the observed behavior.

Exercise 5.9. Now, copy your implementation `StateFeedbackController.m` to a new file `NonlinearStateFeedbackController.m` and modify the feedback controller (1) to

$$u = u_0 - K \begin{pmatrix} \frac{\sin(\theta_e)}{\theta_e} d \\ \theta_e \end{pmatrix} \tag{2}$$

The factor $\sin(\theta_e)/\theta_e$ has a removable singularity at $\theta_e = 0$ and can be approximated using its series expansion

$$\frac{\sin(\theta)}{\theta} = 1 - \frac{\theta^2}{6} + \frac{\theta^4}{120} - \frac{\theta^6}{5040} + \mathcal{O}(\theta^8)$$

Implement and explore the effects of this non-linear controller. Again, remember that u here is curvature that should be transformed into a steering angle.

Comment on the properties of the control signal in the first transient part of the path and how this is related to controller tuning.

Exercise 5.10. Compare paths and control signals for the three different controllers and discuss the results.

It is required to include a plot where control signals for all controllers are plotted in the same plot for the same starting state, for example $w_0 = (0,\ 1,\ 0.9 \cdot \pi/2,\ 2)$. This will highlight properties of the controllers. Use the `path_error` method to plot the distance error as a function of time.

A SPLINEPATH

For this exercise, a helper class, `SplinePath`, is provided that represent a 2-dimensional path using spline functions. The class implements a parametric representation $p(s)$ where the path parameter $s \in [0, \text{length}]$. With this class, important operations on a path are pre-implemented, such as

- Get coordinates for an arbitrary point on the path
- Compute the curvature for any point on the path
- Compute the heading for any point on the path
- Perform an orthogonal projection on the path
- Compute the path error for a given trajectory

To create and plot a path, let the $N \times 2$ matrix `p` represent N points on the path. Then the path object `ref_path` is created by

```
ref_path = SplinePath(p);
```

The path is parameterized by a parameter $s \in [0, L]$ where L is the length of the path. The coordinate of the path at position s is given by $p(s)$.

A position parameter vector `s`, with 200 points distributed equidistantly over the path length, is created below and then used to plot the path where class methods `length()`, `x()`, and `y()` are used as below to produce Figure 3(a).

```
s = linspace(0, ref_path.length, 200);
figure(10)
plot(ref_path.x(s), ref_path.y(s), 'b', 'linewidth', 2)
xlabel('x [m]');
ylabel('y [m]');
```

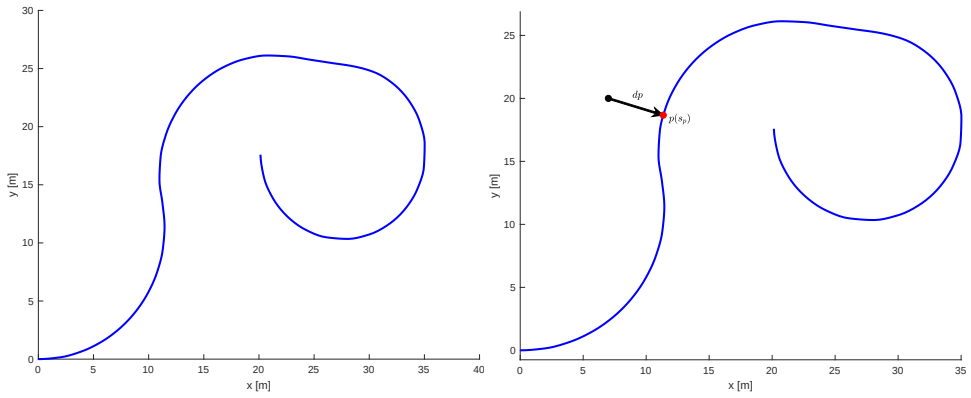

Figure 3: Figure (a) shows a sample path and figure (b) the orthogonal projection of point $(7, 20)$ onto the path.

In the control algorithms, for example when determining how large the path error is, it is important to project a given point onto the path. In general, this is a difficult problem and for the projection to be well defined the point need to be sufficiently close to the path. Here, a numerical approximation is implemented and to project a point p_{car} call the `project`-method as

```
[s_p, dp] = ref_path.project(p_car, so);
```

where `s0` is an approximate path parameter, i.e., an approximate position on the path and the location where the search for projection point is initiated. The projection method has additional parameters you can use to tune operation, see the help text for the method for more details. The function has default values that is appropriate for this exercise so you should not have to modify them here. The function call returns `s_p` which is the path parameter for the projection, i.e., $p_{\text{projection}} = p(s_p)$, and `dp` is the distance between `p_car` and `p_projection`. Figure 3(b) illustrates the orthogonal projection and also shows dp and $p(s_p)$. To illustrate the role of the initial guess `s0`, consider Figure 4 where the black point should be projected onto the path. Since it is in between to segments of the path it is unclear where to project. The red crosses indicate two different starting points for the search, i.e., approximate positions on the path. The blue points indicate the corresponding projection points on the path. This illustration shows, in case of convoluted paths, that

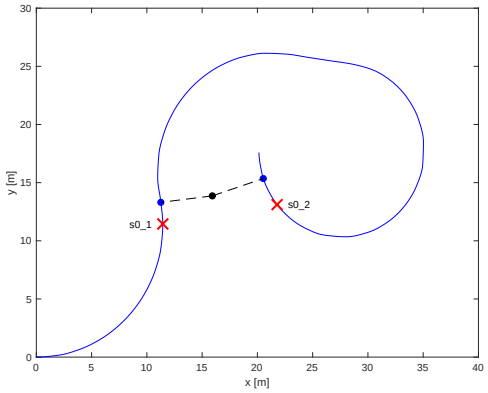

Figure 4: Influence of the initial guess of the path position when projecting.

it is important to keep track of approximate positions on the path to get consistent projections.

Another important class method is `heading` that computes a tangent and normal vector of the path at a given position. The call

```
[h, nc] = ref_path.heading(s_o);
```

returns *normalized* tangent and normal vectors.

Other useful class methods are `c(s)` that returns the curvature at point s and `path_error` method can be used to compute the path error given a vehicle trajectory.

Type `help SplinePath.method_name` at the Matlab prompt for more detailed documentation.

Useful class methods: `x()`, `y()`, `length()`, `c()`, `project()`, `heading()`, `path_error()`.

B CONTROLLER OBJECTS

To implement a controller, you have to implement two functions;

- `u(t, w)` - The main controller function called at time t in state $w = (x, y, \theta, v)$. Returns control signal (δ, a) where δ is the steering angle and a the acceleration.
- `run(t, w)` - Returns `true` if the controller should run and `false` if the controller should stop, e.g., when reaching the goal state.

Consider a simple open-loop controller that only gives a steering angle as a function of time as

$$u = \begin{pmatrix} \delta \\ a \end{pmatrix} = \begin{cases} (10°, 0) & 0 \leq t < 10 \\ (-11°, 0) & 10 \leq t < 20 \\ (0°, 0) & 20 \leq t < 23 \\ (-15°, 0) & 23 \leq t < 40 \\ (0°, 0) & t \geq 40 \end{cases}$$

The implementation, in a file `MiniController.m`, is then

```
classdef MiniController < ControllerBase
    methods
        function obj = MiniController()
            obj = obj@ControllerBase();
        end

        function r = u(obj, t, w)
            a = 0.0;
            if t < 10
                r = [pi/180*10, a];
            elseif t >= 10 && t < 20
                r = [-pi/180*11, a];
            elseif t >= 20 && t < 23
                r = [0, a];
            elseif t >= 23 && t < 40
                r = [-pi/180*15, a];
            else
                r = [0, a];
            end
        end

        function r = run(obj, t, w)
            r = true;
        end
    end
end
```

The first method, named equal to the class name, here `MiniController`, is the constructor method. The constructor need to call the constructor of the super-class `ControllerBase()`. The constructor method does not need to be changed and can be used to pass parameters to the controller. For example the feedback gain K in the state-feedback controller or the look-ahead horizon for the pure-pursuit controller. In the templates provided, you do not need to change the constructor methods (but you may if you want to add additional functionality of course).

Note that you can add any states, properties, or methods that you like to your controller as long as methods `u` and `run` are implemented. There are skeleton files provided for both `PurePursuitController` and `StateFeedbackController`.

For the `PurePursuitController`, an extended base class `PurePursuitControllerBase` that provides some visualization functionality to help debug pursuit point selection.

C LINEAR QUADRATIC REGULATOR – LQR

LQR control is an optimal control technique for linear systems. Consider a time-discrete system

$$x_{t+1} = Ax_t + Bu_t \tag{3}$$

and we want to find the control signal u_t that minimizes the cost function

$$J = \sum_{t=0}^{\infty} x_t^T Q x_t + u_t^T R u_t$$

where Q and R are weighting matrices penalizing the error in x and the size of the control input u respectively. The matrices Q and R are used as tuning parameters where it can be specified what to prioritize, fast convergence in the states or small control signals.

In this case there exists an explicit solution to the optimization problem. The optimal control signal is

$$u = -Kx$$

where

$$K = (R + B^T P B)^{-1} B^T P A$$

and P is the unique symmetric solution to the discrete-time algebraic Ricatti equation (DARE)

$$P = A^T P A - A^T P B (R + B^T P B)^{-1} B^T P A + Q$$

The Ricatti equation has a unique solution under general conditions, e.g., if (3) is controllable. The Ricatti equation can be solved in Matlab using the `idare` command. In python a solver `solve_discrete_are` is available in module `scipy.linalg`.

D EXTRA ASSIGNMENT

In this assignment, you will implement a Model Predictive Controller (MPC) for doing path tracking. Using a linearization strategy, you will formulate a linear MPC problem with a quadratic cost-function for which there exists efficient solvers. In this exercise, the CasADi tool for nonlinear optimization and algorithmic differentiation will be used. See Section D.1 for instructions how to run CasADi. For more information on MPC controllers, see lecture "*Model Predictive Control for Autonomous Vehicles*" and the recommended reading literature.

The MPC controller will be formulated in the Frenet frame, similar to the state-feedback controller from the basic exercise. In Exercise 5.4, you derived the error dynamics in the Frenet frame with input $u = \delta$ and the state is $w = (d(t), \theta_e(t))$ where $d(t)$ is the distance error and $\theta_e(t)$ the heading error. Let d_0 and $\theta_{e,0}$ be the distance and heading error at $t = t_k$, then the optimization problem to be solved in the MPC controller is in this exercise formulated as

$$\begin{aligned} \min_{u(t)} \quad & \int_{t_k}^{t_k+h_p} \gamma_d d(t)^2 + \gamma_\theta \theta_e(t)^2 + \gamma_u u(t)^2 \, dt \\ \text{s.t.} \quad & \dot{w} = g(w, u) \\ & d(t_k) = d_0 \\ & \theta_e(t_k) = \theta_{e,0} \\ & |u| \leq \delta_{\text{lim}} \end{aligned} \tag{4}$$

where h_p is the prediction horizon, u the steer angle, and γ_d, γ_θ, and γ_u are weighting factors for the distance error, heading error, and actuation (steer angle) size respectively. This problem is solved repeatedly and the first control output is applied at each sample instant.

There is a skeleton file `extra.m/ipynb` for Matlab and Python respectively, where you can implement and simulate your MPC path following controller.

Before you start: Read through the code in class `ModelPredictiveController` and ensure that you understand how it is supposed to work.

Exercise D.1. Start by implementing the cost function in method `construct_problem`, i.e., implement

$$J = \sum_{i=0}^{N} \gamma_d d_i^2 + \gamma_\theta \theta_{e,i}^2 + \gamma_u u_i^2$$

which is a discrete version of the loss-function in (4).

Use the CasADi function `sumsqr` to compute the sum-of-squares needed. Also, use the `heading_error` function from the state-feedback controller in the basic exercise.

Exercise D.2. Write down the Frenet equations, i.e., function g in (4). Then, linearize the equations and implement them in the method `error_model`(w, v, u, κ) in the `ModelPredictiveController` class where $w = (d, \theta_e)$ is the error state, v current velocity, and κ the current curvature. Note that the controller is written in continuous time, the controller is discretized in the controller using a Runge-Kutta integration method.

Hint: Important that you ensure that the error model is linear. Therefore, assume constant velocity, constant curvature, small θ_e ($\sin\theta_e \approx \theta_e$), and small u ($\tan\delta \approx \delta$),

and path velocity equal to car velocity ($\dot{s} \approx v$). For more information about the linearization procedure, see details in lecture "*Ground vehicle motion control*".

Exercise D.3. Now, you should have everything you need to run the MPC path following controller. Make a first experiment and see that it works. When building the optimization problem, you get a printout similar to

```
This is casadi::QRQP
Number of variables:                       302
Number of constraints:                     302
```

Note that the number 302 may be different for you since it depends on the parameterization of your controller.

Explain how the number of variables and constraints in the optimization problem corresponds to controller parameters such as sampling-period and prediction horizon. Also determine how many of the constraints are equality constraints and how many are inequality constraints and what does that mean for the degree of freedom in the optimization problem.

Exercise D.4. Now, experiment with different initial states, values for controller parameters (as defined above); the sampling rate (`dt`), prediction horizon (`h_p`), steer limitations (`steer_limit`) as well as the different weights (`gamma_d, gamma_theta, gamma_u`).

Find illustrative experiments and discuss the results, in particular related to tracking performance, computational time, and quality of the performed maneuver.

Exercise D.5. Discuss how properties of the MPC controller compare with the state-feedback controller from the basic exercise, advantages/disadvantages.

D.1 CasADi installation

CasADi is pre-installed in the student labs, you only need to add the installation directory to the Matlab path. You do this by including the line below in your scripts

```
addpath /courses/tsfs12/casadi
```

For Python users, CasADi is installed in the provided virtual environment that is activated by

```
% source /courses/tsfs12/env/bin/activate
```

To run on your own computer, download and install CasADi according to the instructions on https://web.casadi.org/.

In Python, you install the CasADi package by running

```
pip install casadi
```

after you activated your virtual environment.