Control Theory Tutorial

Car-Like Mobile Robot

***Python* for simulation, animation and control**

Contents

1 Introduction

The goal of this tutorial is to teach the usage of the programming language *Python* as a tool for developing and simulating control systems.

2 Model of a car-like mobile robot

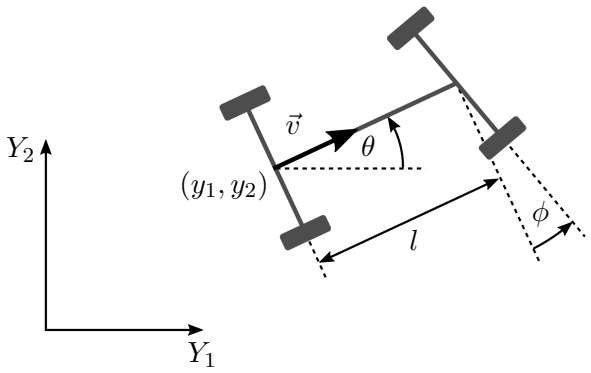

Figure 1: Car-like mobile robot

Given is a nonlinear kinematic model of a car-like mobile robot, with the following system variables: position (y_1, y_2) and orientation θ in the plane, the steering angle ϕ and the robots lateral velocity $v = |\mathbf{v}|$.

$$\dot{y}_1 = v\cos(\theta) \tag{1a}$$

$$\dot{y}_2 = v\sin(\theta) \tag{1b}$$

$$\tan(\phi) = \frac{l\dot{\theta}}{v} \tag{1c}$$

To simulate this system of 1st order ordinary differential equations (ODEs), we define a state vector $\mathbf{x} = (x_1, x_2, x_3)^{\mathrm{T}}$ and a control vector $\mathbf{u} = (u_1, u_2)^{\mathrm{T}}$:

$$\begin{aligned} x_1 &= y_1 & \qquad u_1 &= v \\ x_2 &= y_2 & \qquad u_2 &= \phi \\ x_3 &= \theta \end{aligned}$$

Now we can express (1) in a general form $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$:

$$\underbrace{\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{pmatrix}}_{\dot{\mathbf{x}}} = \underbrace{\begin{pmatrix} u_1\cos(x_3) \\ u_1\sin(x_3) \\ \frac{1}{l}u_1\tan(u_2) \end{pmatrix}}_{f(\mathbf{x},\mathbf{u})} \tag{2}$$

3 Storing parameters

We store the parameters of our system in a class *Parameters()*.

```
class Parameters(object):
    pass
```

We therefore create an entity of *Parameters()* and assign attributes.

```
prmtrs = Parameters() # entity of class Parameters
prmtrs.l = 0.3 # define car length
prmtrs.w = prmtrs.l*0.3 # define car width
```

4 Libraries and Packages

In order to use $\cos(\cdot), \sin(\cdot)$ and $\tan(\cdot)$ we need to import these functions at the beginning of our code from the *numpy* library.

```
import numpy as np
from numpy import cos, sin, tan
```

To simulate (2) we need to solve an initial value problem (IVP). In *Python* we can use the library *SciPy* and its sub-package *integrate*, which delivers different solvers for IVPs.

```
from scipy.integrate import odeint
```

For plotting the output of our simulation, we use the library *Matplotlib* and its sub-package *pyplot*, which delivers a user experience similar to *MATLAB*.

```
import matplotlib.pyplot as plt
```

5 Simulation with SciPy's integrate package

[1] To simulate (2) we need to implement the ODE system as a function in *Python*.

```
7  def ode(x, t, prmtrs):
8      """Function of the robots kinematics
9
10     Args:
11         x: state
12         t: time
13         prmtrs(object): parameter container class
14
15     Returns:
16         dxdt: state derivative
17     """
18     x1, x2, x3 = x  # state vector
19     u1, u2 = control(x, t)  # control vector
20     # dxdt = f(x, u)
21     dxdt = np.array([u1 * cos(x3),
22                      u1 * sin(x3),
23                      1 / prmtrs.l * u1 * tan(u2)])
24
25     # return state derivative
26     return dxdt
```

[1] corresponding file: *car-like_mobile_robot_plotting.py*

The control law is also implemented as function.

```
29 def control(x, t):
30     """Function of the control law
31
32     Args:
33         x: state vector
34         t: time
35
36     Returns:
37         u: control vector
38
39     """
40     u = [1, 0.25]
41     return u
```

As a first simple heuristic, we set (u_1, u_2) equal to constant values. Later we can implement an arbitrary function, for expample a feedback law $\mathbf{u} = k(\mathbf{x})$.

5.1 Solution of the initial value problem (IVP) using *SciPy*

We then define the simulation time and the initial state value.

```
98  prmtrs.l = 0.3  # define car length
99  prmtrs.w = prmtrs.l*0.3  # define car width
100
101 t0 = 0  # start time
102 tend = 10  # end time
103 dt = 0.04  # step-size
104
105 # time vector
106 tt = np.arange(t0, tend, dt)
```

Now we can parse these parameters and our ODE function to the solver.

```
108 # initial state
109 x0 = [0, 0, 0]
```

The output is an array of size length(tt)$\times$length($\mathbf{x}$).

6 Plotting using *Matplotlib*

We encase our plotting instructions in a function. This way, we can define parameters of our plot, which we would like to change easily, for example figure width, or if the figure should be saved on the hard drive.

```
44 def plot_data(x, u, t, fig_width, fig_height, save=False):
45     """Plotting function of simulated state and actions
46
47     Args:
48         x(ndarray): state-vector trajectory
49         u(ndarray): control vector trajectory
50         t(ndarray): time vector
51         fig_width: figure width in cm
52         fig_height: figure height in cm
53         save (bool) : save figure (default: False)
54     Returns: None
55
56     """
```

```
    # creating a figure with 2 subplots, that share the x-axis
    fig1, (ax1, ax2) = plt.subplots(2, sharex=True)

    # set figure size to desired values
    fig1.set_size_inches(fig_width / 2.54, fig_height / 2.54)

    # plot y_1 in subplot 1
    ax1.plot(t, x[:, 0], label='$y_1(t)$', lw=1, color='r')

    # plot y_2 in subplot 1
    ax1.plot(t, x[:, 1], label='$y_2(t)$', lw=1, color='b')

    # plot theta in subplot 2
    ax2.plot(t, x[:, 2], label=r'$\theta(t)$', lw=1, color='g')

    ax1.grid(True)
    ax2.grid(True)
    # set the labels on the x and y axis in subplot 1
    ax1.set_ylabel(r'm')
    ax1.set_xlabel(r't in s')
    ax2.set_ylabel(r'rad')
    ax2.set_xlabel(r't in s')

    # put a legend in the plot
    ax1.legend()
    ax2.legend()

    # automatically adjusts subplot to fit in figure window
    plt.tight_layout()

    # save the figure in the working directory
    if save:
        plt.savefig('state_trajectory.pdf')  # save output as pdf
        plt.savefig('state_trajectory.pgf')  # for easy export to LaTex
    return None
```

Now that we have defined our plotting function, we can execute it with the calculated trajectories and our desired values for the functions parameters.

```
# plot
plot_data(x_traj,u_traj,tt,12, 8, save=True)

plt.show()
```

If your not satisfied with the result, you can change other properties of the plot, like linewidth or -color and many others easily. Just look up the documentation of *Matplotlib* : https://matplotlib.org/index.html

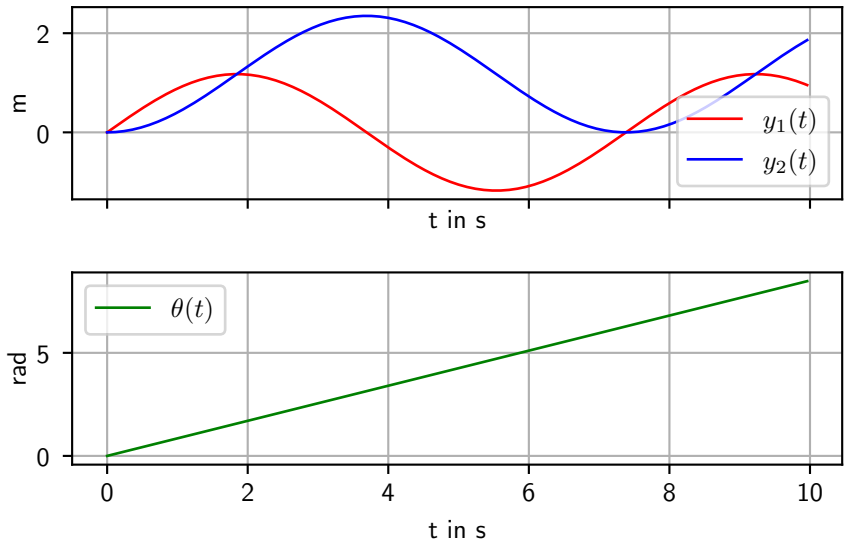

Figure 2: State trajectory plot created with *Matplotlib*

7 Animation using *Matplotlib*

[2] Plotting the state trajectory is often sufficient, but sometimes it can be helpful to have a visiual represantation of the system to get a better understanding of what is actually happening. This applies especially for mechanical systems. *Matplotlib* provides the sub-package *animation*, which can be used for such a purpose. We therefore need to add

```
from matplotlib import animation
```

to the top of our code. We encapsulate all functions for the animation in a function *car_animation()*. At first we create a figure like we did in 6.

```
96  def car_animation(x, u, t, prmtrs):
97      """Animation function of the car-like mobile robot
98
99      Args:
100         x(ndarray): state-vector trajectory
101         u(ndarray): control vector trajectory
102         t(ndarray): time vector
103         prmtrs(object): parameters
104
105     Returns: None
106
107     """
108     dx = 1.5 * prmtrs.l
109     dy = 1.5 * prmtrs.l
110     fig2, ax = plt.subplots()
111     ax.set_xlim([min(min(x_traj[:, 0] - dx), -dx),
```

[2] corresponding file: *car-like_mobile_robot_animation.py*

```
                max(max(x_traj[:, 0] + dx), dx)])
    ax.set_ylim([min(min(x_traj[:, 1] - dy), -dy),
                max(max(x_traj[:, 1] + dy), dy)])
    ax.set_aspect('equal')
    ax.set_xlabel(r'$y_1$')
    ax.set_ylabel(r'$y_2$')

    x_traj_plot, = ax.plot([], [], 'b')  # state trajectory in the y1-y2-plane
    car, = ax.plot([], [], 'k', lw=2)  # car
```

Now we want to display a representation of our car in the figure. We do this by plotting lines. All lines that represent the car are defined by points, which depend on the current state $\mathbf{x}$ and control signal $\mathbf{u}$. This means we need to define a function inside *car_animation()* that maps from $\mathbf{x}$ and $\mathbf{u}$ to a set of points in the (Y_1, Y_2)-plane using geometry and passes these to the plot instance *car*.

```
    def car_plot(x, u):
        """Mapping from state x and action u to the position of the car elements

        Args:
            x: state vector
            u: action vector

        Returns:
            car:

        """
        wheel_length = 0.1 * prmtrs.l
        y1, y2, theta = x
        v, phi = u

        # define chassis lines
        chassis_y1 = [y1, y1 + prmtrs.l * cos(theta)]
        chassis_y2 = [y2, y2 + prmtrs.l * sin(theta)]

        # define lines for the front and rear axle
        rear_ax_y1 = [y1 + prmtrs.w * sin(theta), y1 - prmtrs.w * sin(theta)]
        rear_ax_y2 = [y2 - prmtrs.w * cos(theta), y2 + prmtrs.w * cos(theta)]
        front_ax_y1 = [chassis_y1[1] + prmtrs.w * sin(theta + phi),
                       chassis_y1[1] - prmtrs.w * sin(theta + phi)]
        front_ax_y2 = [chassis_y2[1] - prmtrs.w * cos(theta + phi),
                       chassis_y2[1] + prmtrs.w * cos(theta + phi)]

        # define wheel lines
        rear_l_wl_y1 = [rear_ax_y1[1] + wheel_length * cos(theta),
                        rear_ax_y1[1] - wheel_length * cos(theta)]
        rear_l_wl_y2 = [rear_ax_y2[1] + wheel_length * sin(theta),
```

...

```
        # concatenate set of coordinates
        data_y1 = [rear_ax_y1, empty, front_ax_y1, empty, chassis_y1,
                   empty, rear_l_wl_y1, empty, rear_r_wl_y1,
                   empty, front_l_wl_y1, empty, front_r_wl_y1]
        data_y2 = [rear_ax_y2, empty, front_ax_y2, empty, chassis_y2,
                   empty, rear_l_wl_y2, empty, rear_r_wl_y2,
                   empty, front_l_wl_y2, empty, front_r_wl_y2]

        # set data
        car.set_data(data_y1, data_y2)
        return car,
```

For the animation to work we need to define another two functions, *init()* and *animate(i)*. The *init()*-function defines which objects change during the animation.

```
182     def init():
183         """Initialize plot objects that change during animation.
184            Only required for blitting to give a clean slate.
185 
186         Returns:
187 
188         """
189         x_traj_plot.set_data([], [])
190         car.set_data([], [])
191         return x_traj_plot, car
```

The *animate(i)*-function assigns data to the changing objects, in our case the car and trajectory plots and the simulation time.

```
193     def animate(i):
194         """Defines what should be animated
195 
196         Args:
197             i: frame number
198 
199         Returns:
200 
201         """
202         k = i % len(t)
203         ax.set_title('Time (s): ' + str(t[k]), loc='left')
204         x_traj_plot.set_xdata(x[0:k, 0])
205         x_traj_plot.set_ydata(x[0:k, 1])
206         car_plot(x[k, :], control(x[k, :], t[k]))
207         return x_traj_plot, car
```

Finally we have to pass these functions and the figure we created to *animation.FuncAnimation()*.

```
209     ani = animation.FuncAnimation(fig2, animate, init_func=init,
210                                   frames=len(t) + 1,
211                                   interval=(t[1] - t[0]) * 1000,
212                                   blit=False)
213 
214     ani.save('animation.mp4', writer='ffmpeg', fps=1 / (t[1] - t[0]))
215     plt.show()
216     return None
```

Now we have all things set up to simulate our system and animate it.

```
244 # animation
245 car_animation(x_traj, u_traj, tt, prmtrs)
246 
247 plt.show()
```

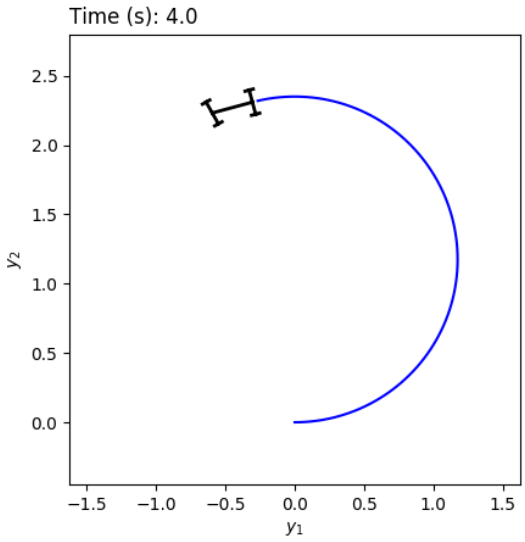

Figure 3: Car animation

8 Simulation with SciPy's new *solve_ivp* module and the *lambda* function

[3] In addition to the solution in section 5 using *odeint*, SciPy's integrate package contains some newer solver classes. To use the general *solve_ivp* class, we need to import the new package.

```
from scipy.integrate import solve_ivp
```

Then we have to switch the arguments of our *ode*-function, because in *solve_ivp* the desired order of function arguments in the ODE is different. We therefore replace

```
def ode(x, t, prmtrs):
```

with

```
def ode(t, x, prmtrs):
```

Now we can call the solver.

```
sol = solve_ivp(lambda t, x: ode(t, x, prmtrs),
                (t0, tend), x0, method='RK45',t_eval=tt)
```

The arguments of *solve_ivp* differ from *odeint*. The ODE must have the form $f(t, x)$. In order to use *ode(t, x, prmtrs)*, which takes 3 arguments, we need to use a *lambda* function. This way we encapsulate the ODE in an anonymous function, that has just (t, x) as arguments and can be evaluated by *solve_ivp*.[4] After the ODE is passed the solver takes the following arguments: a tuple $(t0, tend)$ which defines the simulation interval, the initial value $x0$. Additionally we pass the optional arguments *method*, in this case a Runge-Kutta method and *t_eval*, which defines the values at which the solution should be sampled. The return value *sol* is an *OdeResult* object. To extract the simulated state trajectory, we execute:

```
x_traj = sol.y.T # size=len(x)*len(tt) (.T -> transpose)
```

[3] corresponding file: *car-like_mobile_robot_lambda.py*
[4] the lambda function corresponds to @ in *MATLAB*

9 (Differential) flatness based tracking control

For controlling a nonlinear system like (2), linear control methods are not sufficient. We therefore use an advanced control method called (differential) flatness based tracking control, where we design a model based feedforward control and stabilize the system along a planned state trajectory.

9.1 (Differential) flatness

A system $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$ is called (differntially) flat, if a tuple of differential independent variables exists, from which we can derive all other system variables $\mathbf{z} = (\mathbf{x}, \mathbf{u})$, without solving an ODE. Such a tuple is called a flat output $\mathbf{y} = h(\mathbf{x})$. The flat output has $m = s - q$ components, where s is the number of system variables and q is the number of equations. In system (1), we have 5 system variables $(y_1, y_2, \theta, v, \phi)$ and 3 equations, a flat output must therefore have 2 components. A flat output is $\mathbf{y} = (y_1, y_2)$. We now want to show, that a function $\mathbf{z} = \psi(\mathbf{y}, \dot{\mathbf{y}}, ..., \mathbf{y}^{(\alpha)})$ for $\mathbf{y} = (y_1, y_2)$ exists.

Recapture system (1) from section 2:

$$\dot{y}_1 = v\cos(\theta) \tag{1a}$$

$$\dot{y}_2 = v\sin(\theta) \tag{1b}$$

$$\tan(\phi) = \frac{l\dot{\theta}}{v} \tag{1c}$$

Dividing (1b) by (1a) leads to:

$$\frac{\dot{y}_2}{\dot{y}_1} = \tan(\theta) \tag{3a}$$

$$\Leftrightarrow \theta = \arctan\left(\frac{\dot{y}_2}{\dot{y}_1}\right) \tag{3b}$$

The velocity v can be derived from the time derivative of the position vector $\mathbf{y}$.

$$v = |\mathbf{v}| = |\dot{\mathbf{y}}| = \sqrt{\dot{y}_1^2 + \dot{y}_2^2} \tag{4}$$

We take the derivative of (3b) and (4) insert the result in (1c):

$$\tan(\phi) = \underbrace{\frac{l}{\sqrt{\dot{y}_1^2 + \dot{y}_2^2}}}_{v} \underbrace{\frac{\ddot{y}_1\ddot{y}_1 - \dot{y}_1\ddot{y}_2}{(\dot{y}_1^2 + \dot{y}_2^2)}}_{\dot{\theta}} \tag{5a}$$

$$\Leftrightarrow \phi = \arctan\left(l\frac{\ddot{y}_1\ddot{y}_1 - \dot{y}_1\ddot{y}_2}{(\dot{y}_1^2 + \dot{y}_2^2)^{\frac{3}{2}}}\right) \tag{5b}$$

Now we have found $\mathbf{z} = \psi(\mathbf{y}, \dot{\mathbf{y}}, ..., \mathbf{y}^{(\alpha)})$:

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \theta \\ v \\ \phi \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \arctan\left(\frac{\dot{y}_2}{\dot{y}_1}\right) \\ \sqrt{\dot{y}_1^2 + \dot{y}_2^2} \\ \arctan\left(l \frac{\ddot{y}_1 \dot{y}_1 - \dot{y}_1 \ddot{y}_2}{(\dot{y}_1^2 + \dot{y}_2^2)^{\frac{3}{2}}}\right) \end{pmatrix} \tag{6}$$

$\mathbf{y} = (y_1, y_2)$ is indeed a flat output.

9.2 Dynamic state feedback via exact input-output linearization

In order to determine a control law we linearize the system by defining a new input $\mathbf{w} = (w_1, w_2)$. To do this we have to take the derivative of the flat output $\mathbf{y}$, until the input $\mathbf{u}$ shows up explicitly.

$$y_1 = x_1 \tag{7a}$$
$$y_2 = x_2 \tag{7b}$$
$$\dot{y}_1 = \dot{x}_1 = u_1 \cos(x_3) \tag{7c}$$
$$\dot{y}_2 = \dot{x}_2 = u_1 \sin(x_3) \stackrel{!}{=} w_1 \tag{7d}$$
$$\ddot{y}_1 = \frac{\mathrm{d}}{\mathrm{d}t}(u_1 \cos(x_3)) = \dot{u}_1 \cos(x_3) - \frac{1}{l} u_1^2 \sin(x_3) \tan(u_2) \stackrel{!}{=} w_2 \tag{7e}$$

With a generalized state vector $\mathbf{q} = (q_1, q_2, q_3)^{\mathrm{T}} = (y_1, \dot{y}_1, y_2)^{\mathrm{T}}$ we now get a new linear state space model $\dot{\mathbf{q}} = g(\mathbf{q}, \mathbf{w})$:

$$\begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{pmatrix} = \begin{pmatrix} q_2 \\ w_2 \\ w_1 \end{pmatrix} \tag{8}$$

9.2.1 Stabilizing the linearized system

To stabilize system (8), we define a differential equation for the tracking error $\mathbf{e} = \mathbf{y} - \mathbf{y}_d$:

$$0 = \ddot{\mathbf{e}} + \mathbf{K}_1 \dot{\mathbf{e}} + \mathbf{K}_0 \mathbf{e} \tag{9}$$

We choose the matrices $\mathbf{K}_0$ and $\mathbf{K}_1$ such that the ODE is stable. If we solve (9) for $\mathbf{w}$ we get:

$$w_1 = \dot{y}_2 = \dot{y}_{2,d} - k_{0,2}(y_2 - y_{2,d}) \tag{10a}$$
$$w_2 = \ddot{y}_1 = \ddot{y}_{1,d} - k_{1,1}(\dot{y}_1 - \dot{y}_{1,d}) - k_{0,1}(y_1 - y_{1,d}) \tag{10b}$$

9.2.2 Control law

To determine a control law, we substitute (10) into (7d) and (7e) and solve for $\mathbf{u}$. What we'll find is a differential equation for the input, that's why the control law is called dynamic state feedback.

$$u_1 = \frac{w_1}{\sin x_3} \tag{11a}$$

$$\Leftrightarrow \quad u_1 = \frac{\dot{y}_{2,d} - k_{0,2}(y_2 - y_{2,d})}{\sin x_3} \tag{11b}$$

$$\Leftrightarrow \quad u_1 = \frac{\dot{y}_{2,d} - k_{0,2}(y_2 - y_{2,d})}{\sin\left(\arctan\left(\frac{\dot{y}_2}{\dot{y}_1}\right)\right)} \tag{11c}$$

$$\dot{u}_1 \cos(x_3) - \frac{1}{l} u_1^2 \sin(x_3) \tan(u_2) = w_2 \tag{12a}$$

$$\Leftrightarrow \quad \frac{1}{l} u_1^2 \sin(x_3) \tan(u_2) = \dot{u}_1 \cos(x_3) - w_2 \tag{12b}$$

$$\Leftrightarrow \quad \tan(u_2) = \frac{l}{u_1^2 \sin(x_3)} \Big(\dot{u}_1 \cos(x_3) - w_2 \Big) \tag{12c}$$

$$\Leftrightarrow \quad u_2 = \arctan\left(\frac{l}{u_1^2 \sin(x_3)} \Big(\dot{u}_1 \cos(x_3) - w_2 \Big) \right) \tag{12d}$$

9.3 Calculating a reference trajectory (path planner)

Now that we have defined the control law we need to develop a path planner, that calculates a feasible trajectory of the flat output and its derivitives for a given state transition.

figure: showing a state transition

9.4 Implementation of the controller in *Python*

work in progress