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. The following topics are covered:

- Implementation of the model in *Python*,
- Simulation of the model,
- Presentation of the results.

Python source code file: `01_car_example_plotting.py`

Later in this tutorial we will extend our simulation by a visualization of the moving car and a full feedback control with trajectory planning.

Please refer to the Python List-Dictionary-Tuple tutorial and the NumPy Array tutorial if you are not familiar with the handling of containers and arrays in Python. If you are completely new to *Python* consult the very basic introduction on tutorialspoint.

2 Model of a car-like mobile robot

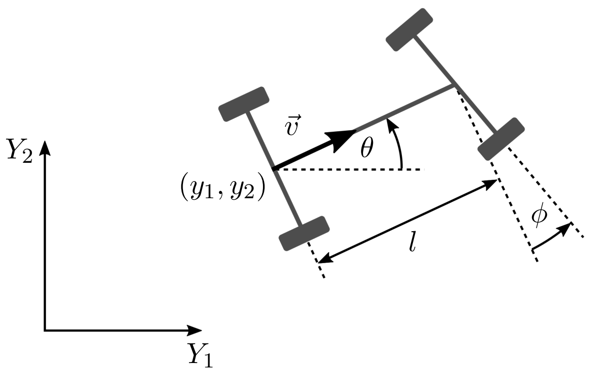

Figure 1: Car-like mobile robot

Given is a nonlinear kinematic model of a car-like mobile robot, cf. Figure 1, 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 $v = |\mathbf{v}|$:

$$\dot{y}_1(t) = v\cos(\theta(t)) \qquad y_1(0) = y_{10} \tag{1a}$$
$$\dot{y}_2(t) = v\sin(\theta(t)) \qquad y_2(0) = y_{20} \tag{1b}$$
$$\tan(\phi(t)) = \frac{l\dot{\theta}(t)}{v(t)} \qquad \theta(0) = \theta_0. \tag{1c}$$

The initial values are denoted by y_{10}, y_{20}, and θ_0, respectively. The velocity v and the steering angle ϕ can be considered as an input acting on the system.

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 $\mathbf{u} = (u_1, u_2)^{\mathrm{T}}$:

$$x_1 = y_1 \qquad u_1 = v \tag{2a}$$
$$x_2 = y_2 \qquad u_2 = \phi\,. \tag{2b}$$
$$x_3 = \theta \tag{2c}$$

Now we can express the IVP (1) in a general form $\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t))$, $\mathbf{x}(0) = \mathbf{x}_0$:

$$\underbrace{\begin{pmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \dot{x}_3(t) \end{pmatrix}}_{\dot{\mathbf{x}}(t)} = \underbrace{\begin{pmatrix} u_1(t)\cos(x_3(t)) \\ u_1(t)\sin(x_3(t)) \\ \frac{1}{l}u_1(t)\tan(u_2(t)) \end{pmatrix}}_{f(\mathbf{x}(t),\mathbf{u}(t))} \qquad \mathbf{x}(0) = \mathbf{x}_0. \tag{3}$$

This explicit formulation of the IVP is usually the basis for implementing a numerical integration needed for simulation of the system. In the following we will setup a simulation using the programming language *Python* which shows how the vehicle behaves if we drive with a continously decreasing velocity under a constant steering angle (of course we know the result in advance: the car will drive on a circle until it stops for $v = 0$). We will derive the *Python*-script for the simulation of the system line by line.

3 Libraries and Packages

Python itself does not offer any functions for the direct solution of the IVP (1) and for the presentation of the results. Therefore, we need to import certain packages which provide utilities for the mathematical calculations, array handling, numerical integration and plotting. Under *Python* such packages should be imported at the top of the executed script[1]. The packages which are most relevant for the simulation of control systems in this tutorial are NumPy for array handling and mathematical functions, *SciPy* for numerical integration of ordinary differential equations (and a lot of other stuff, of course) and MatplotLib for plotting.

[1]It is also possible to import them elsewhere in the code but this is not good style.

It is good practice to connect the imported packages with a kind of namespace so we know in our code where which function comes from. In case of NumPy the following statement imports the packages NumPy and ensures that in the following every function from NumPy is addressed by the prefix `np.`:

```
import numpy as np
```

For "trivial" functions like $\cos(\cdot), \sin(\cdot)$ and $\tan(\cdot)$ it is annoying to prefix them like `np.cos(...)` each time. To avoid this we can directly import them as

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

To solve the IVP (2) we use the library *SciPy* and its sub-package *integrate*, which delivers different solvers for IVPs.

```
import scipy.integrate as sci
```

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
```

4 Storing parameters

In such simulations we usually have to deal with a lot of parameters describing the system or the simulation setup. It is a good idea to pack these parameters into one object so we do not have to deal with several individual variables holding the values of the parameters. There are several possibilities to do so. One easy way is to pack the parameters into a structure which basically is an instance of an empty class derived from `object` and susbsequently assign the members holding the parameter values:

```
class Parameters(object):
    pass


# Physical parameter
para = Parameters()  # instance of class Parameters
para.l = 0.3         # define car length
para.w = para.l*0.3  # define car width
```

The same is done with the simulation parameter:

```
# Simulation parameter
sim_para = Parameters()  # instance of class Parameters
sim_para.t0 = 0          # start time
sim_para.tend = 10       # end time
sim_para.dt = 0.04       # step-size
```

Alternatively, one could use a dictionary. However, the resulting keyword notation (e.g. `para["l"]` instead of `para.l`) in the formulas using the parameters becomes a bit annoying in that approach.

5 Simulation with *SciPy*'s integrate package

5.1 Implementation of the model

To simulate (2) using the numerical integrators offered by *SciPy*'s integrate package we need to implement a function which returns the right hand side of (2) evaluated for given values of $\mathbf{x}$, $\mathbf{u}$ and the parameters:

```python
def ode(x, t, p):
    """Function of the robots kinematics

    Args:
        x: state
        t: time
        p(object): parameter container class

    Returns:
        dxdt: state derivative
    """
    x1, x2, x3 = x  # state vector
    u1, u2 = control(x, t)  # control vector

    # dxdt = f(x, u):
    dxdt = np.array([u1 * cos(x3),
                     u1 * sin(x3),
                     1 / p.l * u1 * tan(u2)])

    # return state derivative
    return dxdt
```

The control law calculating values for v and ϕ depending on the state $\mathbf{x}$ and the time t is also implemented as a function:

```python
def control(x, t):
    """Function of the control law

    Args:
        x: state vector
        t: time

    Returns:
        u: control vector

    """
    u1 = np.maximum(0, 0.5 - 0.06*t)
    u2 = np.full(u1.shape, 0.25)
    return np.array([u1, u2]).T
```

As a first simple heuristic, we drive the car with a constant steering angle and continously reduce the speed starting from $0.5\,\mathrm{m\,s^{-1}}$ until it reaches zero. Later we can implement an arbitrary function, for expample a feedback law $\mathbf{u} = k(\mathbf{x})$. Note that the function needs to handle also time arrays as input in order to calculate the control for a bunch of values at once. That's why we use NumPy's array capable maximum function and set the shape of `u2` appropriately.

Note the way the two functions above are documented. The text within the `"""` is called *docstring*. Tools like Sphinx are able to build well formatted documentations out of them. There are several ways the docstrings can be written in the source code files. Here we use the so-called Google Style.

5.2 Solution of the IVP using *SciPy*

We are now ready to perform the numerical integration of system (2). At first, we define a vector `tt` specifying the time values at which we would like to obtain the computed values of $\mathbf{x}$. Then we define the initial vector $\mathbf{x}_0$ and call the odeint function of the *SciPy* integrate package to perform the simulation[2]. Note that `odeint` is a variable step solver although it outputs the result at equally spaced time steps in this case. The output is an array of shape `len(tt)`×`len(x0)`. Finally, the control input values are calculated from the obtained trajectory of $\mathbf{x}$ (we cannot directly save the values for $\mathbf{u}$ in the `ode` function because it is also repeatedly called between our specified time steps by the solver).

```
# time vector
tt = np.arange(sim_para.t0, sim_para.tend + sim_para.dt, sim_para.dt)

# initial state
x0 = [0, 0, 0]

# simulation
x_traj = sci.odeint(ode, x0, tt, args=(para, ))
u_traj = control(x_traj, tt)
```

Note that the interval specified by `np.arange` is open on the right hand side. Hence, we add `dt` to include also `tend` in the simulation.

6 Plotting using *Matplotlib*

Usually you will want to publish your results with underlying illustrations. 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.

```
def plot_data(x, u, t, fig_width, fig_height, save=False):
    """Plotting function of simulated state and actions

    Args:
        x(ndarray): state-vector trajectory
        u(ndarray): control vector trajectory
        t(ndarray): time vector
        fig_width: figure width in cm
        fig_height: figure height in cm
        save (bool) : save figure (default: False)
    Returns: None

```

[2]Consider using the more advanced integrators offered by *SciPy*, see Section 8.

```python
    """
    # creating a figure with 3 subplots, that share the x-axis
    fig1, (ax1, ax2, ax3) = plt.subplots(3, 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, np.rad2deg(x[:, 2]), label=r'$\theta(t)$', lw=1, color='g')

    # plot control in subplot 3, left axis red, right blue
    ax3.plot(t, np.rad2deg(u[:, 0]), label=r'$v(t)$', lw=1, color='r')
    ax3.tick_params(axis='y', colors='r')
    ax33 = ax3.twinx()
    ax33.plot(t, np.rad2deg(u[:, 1]), label=r'$\phi(t)$', lw=1, color='b')
    ax33.spines["left"].set_color('r')
    ax33.spines["right"].set_color('b')
    ax33.tick_params(axis='y', colors='b')

    # Grids
    ax1.grid(True)
    ax2.grid(True)
    ax3.grid(True)

    # set the labels on the x and y axis and the titles
    ax1.set_title('Position coordinates')
    ax1.set_ylabel(r'm')
    ax1.set_xlabel(r't in s')
    ax2.set_title('Orientation')
    ax2.set_ylabel(r'deg')
    ax2.set_xlabel(r't in s')
    ax2.set_title('Velocity / steering angle')
    ax3.set_ylabel(r'm/s')
    ax33.set_ylabel(r'deg')
    ax33.set_xlabel(r't in s')

    # put a legend in the plot
    ax1.legend()
    ax2.legend()
    ax3.legend()
    li3, lab3 = ax3.get_legend_handles_labels()
    li33, lab33 = ax33.get_legend_handles_labels()
    ax3.legend(li3 + li33, lab3 + lab33, loc=0)

    # 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, 16, save=True)

plt.show()
```

The result can be found in Figure 6. 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* or consult the exhaustive *Matplotlib* example gallery.

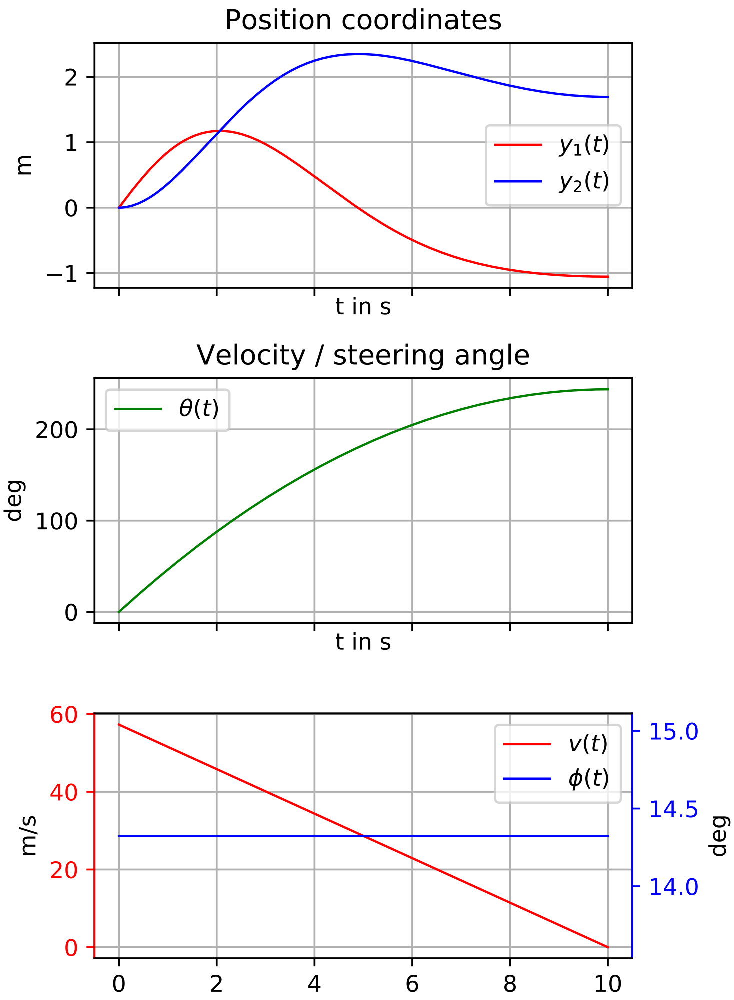

Figure 2: State and control trajectory plot created with *Matplotlib*.

7 Animation using *Matplotlib*

Python source code file: `02_car_example_animation.py`

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

```
import matplotlib.animation as mpla
```

to the top of our code used in the previous sections. Under Windows it might be necessary to explicitely specify the path to the ffmpg library, e.g.:

```
plt.rcParams['animation.ffmpeg_path'] = 'C:\\Progs\\ffmpeg\\bin\\ffmpeg.exe'
```

FFMPG can be downloaded from https://www.ffmpeg.org/download.html.

We encapsulate all functions for the animation in a function called `car_animation()`. At first we create a figure with two empty plots into which we will later draw the car and the curve of the trajectory dependung on the state `x`, the control input `u` and the parameters:

```
def car_animation(x, u, t, p):
    """Animation function of the car-like mobile robot

    Args:
        x(ndarray): state-vector trajectory
        u(ndarray): control vector trajectory
        t(ndarray): time vector
        p(object): parameters

    Returns: None

    """
    # Setup two empty axes with enough space around the trajectory so the car
    # can always be completely plotted. One plot holds the sketch of the car,
    # the other the curve
    dx = 1.5 * p.l
    dy = 1.5 * p.l
    fig2, ax = plt.subplots()
    ax.set_xlim([min(min(x_traj[:, 0] - dx), -dx),
                 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'$y_2$')

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

The handles `h_x_traj_plot` and `h_car` are used later to draw onto the axes.

During animation we want to display a representation of our car in this 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*:

```python
    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:

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

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

        # define lines for the front and rear axle
        rear_ax_y1 = [y1 + p.w * sin(theta), y1 - p.w * sin(theta)]
        rear_ax_y2 = [y2 - p.w * cos(theta), y2 + p.w * cos(theta)]
        front_ax_y1 = [chassis_y1[1] + p.w * sin(theta + phi),
                       chassis_y1[1] - p.w * sin(theta + phi)]
        front_ax_y2 = [chassis_y2[1] - p.w * cos(theta + phi),
                       chassis_y2[1] + p.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),
                        rear_ax_y2[1] - wheel_length * sin(theta)]
        rear_r_wl_y1 = [rear_ax_y1[0] + wheel_length * cos(theta),
                        rear_ax_y1[0] - wheel_length * cos(theta)]
        rear_r_wl_y2 = [rear_ax_y2[0] + wheel_length * sin(theta),
                        rear_ax_y2[0] - wheel_length * sin(theta)]
        front_l_wl_y1 = [front_ax_y1[1] + wheel_length * cos(theta + phi),
                         front_ax_y1[1] - wheel_length * cos(theta + phi)]
        front_l_wl_y2 = [front_ax_y2[1] + wheel_length * sin(theta + phi),
                         front_ax_y2[1] - wheel_length * sin(theta + phi)]
        front_r_wl_y1 = [front_ax_y1[0] + wheel_length * cos(theta + phi),
                         front_ax_y1[0] - wheel_length * cos(theta + phi)]
        front_r_wl_y2 = [front_ax_y2[0] + wheel_length * sin(theta + phi),
                         front_ax_y2[0] - wheel_length * sin(theta + phi)]

        # empty value (to disconnect points, define where no line should be plotted)
        empty = [np.nan, np.nan]

        # 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]
```

```
221
222         # set data
223         h_car.set_data(data_y1, data_y2)
224
225     def init():
226         """Initialize plot objects that change during animation.
```

Note that we are in the scope of the `car_animation` function and have full acess to the handle `h_car` here.

For the animation to work we need to define another two functions, *init()* and *animate(i)*. They will be later called by *Matplotlib* to initialize and perform the animation. The *init()*-function defines which objects change during the animation, in our case the two axes the handles of which are returned:

```
225     def init():
226         """Initialize plot objects that change during animation.
227            Only required for blitting to give a clean slate.
228
229         Returns:
230
231         """
232         h_x_traj_plot.set_data([], [])
233         h_car.set_data([], [])
234         return h_x_traj_plot, h_car
```

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

```
236     def animate(i):
237         """Defines what should be animated
238
239         Args:
240             i: frame number
241
242         Returns:
243
244         """
245         k = i % len(t)
246         ax.set_title('Time (s): ' + str(t[k]), loc='left')
247         h_x_traj_plot.set_xdata(x[0:k, 0])
248         h_x_traj_plot.set_ydata(x[0:k, 1])
249         car_plot(x[k, :], control(x[k, :], t[k]))
250         return h_x_traj_plot, h_car
```

Finally we instanciate an object of `FuncAnimation` of the animation subpackage of *Matplotlib*. There, we pass the `animate` and `ini` to the constructor:

```
252     ani = mpla.FuncAnimation(fig2, animate, init_func=init, frames=len(t) + 1,
253                              interval=(t[1] - t[0]) * 1000,
254                              blit=False)
255
256     ani.save('animation.mp4', writer='ffmpg', fps=1 / (t[1] - t[0]))
257     plt.show()
258     return None
```

Note that all lines from 138 to 257 belong to the function `car_animation`!

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

```
# animation
car_animation(x_traj, u_traj, tt, para)

plt.show()
```

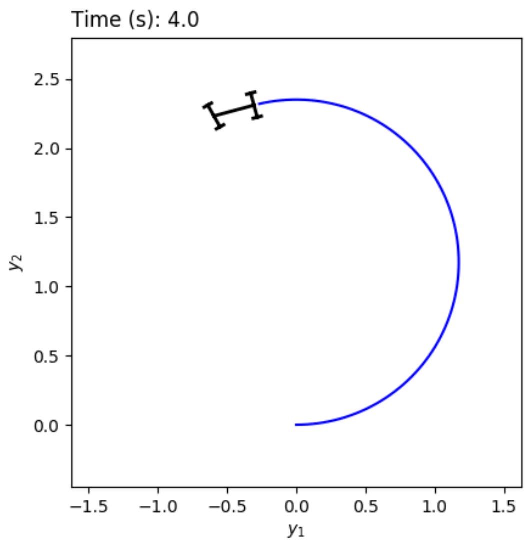

Figure 3: Car animation

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

Python source code file: `03_car_example_scipy_solve_ivp.py`

In addition to the solution in section 5 using *odeint*, *SciPy*'s integrate package contains some newer and more powerful solver functions. One of them is the function `solve_ivp`. The function `solve_ivp` takes a function of the type `func(t, x)` calculating the value of the right hand side of (2). Further parameters are not allowed. In order to be able to use our previously defined ode-function `ode(x, t, p)` which additionally takes the parameter structure `p` and has a different order for `t` and `x`, we make use of a so-called lambda-function. We call the solver as follows:

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

This way we encapsulate our `ode` function in an anonymous function, that has just (t, x) as arguments (as required by `solve_ivp`) but evaluates as `ode(x, t, para)`[3].

[3]The lambda function corresponds to @ in *MATLAB*

Additionally, the following arguments are passed to `solve_ivp`: A tuple $(t0, tend)$ which defines the simulation interval and the initial value $x0$. Furthermore, 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)
```

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):

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

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

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

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

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

9.2 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.

$$y_1 = x_1 \tag{7a}$$

$$y_2 = x_2 \tag{7b}$$

$$\dot{y}_1 = \dot{x}_1 = u_1 \cos(x_3) \tag{7c}$$

$$\dot{y}_2 = \dot{x}_2 = u_1 \sin(x_3) \overset{!}{=} w_1 \tag{7d}$$

$$\ddot{y}_1 = \frac{\mathrm{d}}{\mathrm{d}t}(u_1 \cos(x_3)) = \dot{u}_1 \cos(x_3) - \frac{1}{l} u_1^2 \sin(x_3) \tan(u_2) \overset{!}{=} w_2 \tag{7e}$$

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 the control laws, we first substitute (10) into (7d) and solve for u_1.

$$u_1 = \frac{w_1}{\sin x_3} \tag{11a}$$
$$\Leftrightarrow \quad u_1 = \frac{\dot{y}_{2,d} - k_{0,2}(y_2 - y_{2,d})}{\sin x_3} \tag{11b}$$
$$\Leftrightarrow \quad u_1 = \frac{\dot{y}_{2,d} - k_{0,2}(y_2 - y_{2,d})}{\sin\left(\arctan\left(\frac{\dot{y}_2}{\dot{y}_1}\right)\right)} \tag{11c}$$

Then substitute $\mathbf{w}$ in (5b):

$$u_2 = \arctan\left(l\frac{\ddot{y}_1\dot{y}_2 - \dot{y}_1\ddot{y}_2}{(\dot{y}_1^2 + \dot{y}_2^2)^{\frac{3}{2}}}\right) \tag{12a}$$
$$= \arctan\left(l\frac{w_2 w_1 - \dot{y}_1\dot{w}_1}{(\dot{y}_1^2 + w_1^2)^{\frac{3}{2}}}\right) \tag{12b}$$

The derivative of w_1 appears in the feedback law. We therefore have to derive the equation for it:

$$\dot{w}_1 = \ddot{y}_2 = \frac{\mathrm{d}w_1}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}(\dot{y}_{2,d} - k_{0,2}(y_2 - y_{2,d})) \tag{13a}$$
$$= \ddot{y}_{2,d} - k_{0,2}(\underbrace{\dot{y}_2}_{=w_1} - \dot{y}_{2,d}) \tag{13b}$$
$$= \ddot{y}_{2,d} - k_{0,2}(\dot{y}_{2,d} - k_{0,2}(y_2 - y_{2,d}) - \dot{y}_{2,d}) \tag{13c}$$
$$\Leftrightarrow \quad \dot{w}_1 = \ddot{y}_{2,d} - k_{0,2}^2(y_2 - y_{2,d}) \tag{13d}$$

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 derivatives (to the second order) for a given state transition as shown in Figure 4. A simple, but straight forward

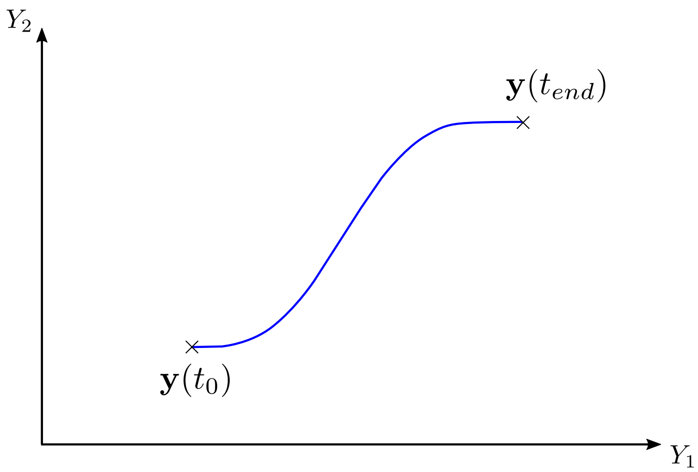

Figure 4: A feasible state transition from $\mathbf{y}(t_0)$ to $\mathbf{y}(t_{end})$

approach for a reference trajectory $\mathbf{y}_d(t)$ is a piecewise-defined function:

$$\mathbf{y}_d(t) = \begin{cases} \mathbf{y}(t_0) & \text{if } t < t_0 \\ \mathbf{y}(t_0) + (\mathbf{y}(t_{end}) - \mathbf{y}(t_0))\varphi_\gamma\left(\frac{t-t_0}{t_{end}-t_0}\right) & \text{if } t \in [t_0, t_{end}] \\ \mathbf{y}(t_{end}) & \text{if } t > T \end{cases} \tag{14}$$

$\tau \to \varphi_\gamma(\tau)$ is a protoype function, where γ indicates how often $\varphi_\gamma(\tau)$ is continuously differentiable. The function has to meet the following conditions, such that the reference trajectory is feasible:

$$\varphi_\gamma(0) = 0 \quad \varphi_\gamma^{(j)}(0) = 0 \quad j = 1, ..., \gamma \tag{15a}$$

$$\varphi_\gamma(1) = 1 \quad \varphi_\gamma^{(j)}(1) = 0 \quad j = 1, ..., \gamma \tag{15b}$$

An approach for the derivative of $\varphi_\gamma(\tau)$, which meets the conditions (15) is:

$$\frac{\mathrm{d}\varphi_\gamma(\tau)}{\mathrm{d}\tau} = \alpha \frac{\tau^\gamma}{\gamma!} \frac{(1-\tau)^\gamma}{\gamma!} \tag{16}$$

Integration leads to:

$$\varphi_\gamma(\tau) = \alpha \int_0^\tau \frac{\tilde{\tau}^\gamma}{\gamma!} \frac{(1-\tilde{\tau})^\gamma}{\gamma!} \mathrm{d}\tilde{\tau} \tag{17}$$

After γ partial integrations we get:

$$\varphi_\gamma(\tau) = \frac{\alpha}{(\gamma!)^2} \sum_{k=0}^{\gamma} \binom{\gamma}{k} \frac{(-1)^k \tau^{\gamma+k+1}}{(\gamma+k+1)}$$

To solve for the unknown α, we use the condition $\varphi_\gamma(1) \overset{!}{=} 1$:

$$\begin{aligned} \varphi_\gamma(1) =& \frac{\alpha}{(\gamma!)^2} \sum_{k=0}^{\gamma} \binom{\gamma}{k} \frac{(-1)^k}{(\gamma+k+1)} \overset{!}{=} 1 \\ \Leftrightarrow \quad & \alpha = (2\gamma+1)! \end{aligned}$$

Finally we can define the prototype function:

$$\varphi_\gamma(\tau) = \frac{(2\gamma+1)!}{(\gamma!)^2} \sum_{k=0}^{\gamma} \binom{\gamma}{k} \frac{(-1)^k \tau^{\gamma+k+1}}{(\gamma+k+1)} \tag{18}$$

and it's n-th derivative:

$$\frac{\mathrm{d}^n}{\mathrm{d}\tau^n} \varphi_\gamma(\tau) = \varphi_\gamma^{(n)}(\tau) = \frac{(2\gamma+1)!}{(\gamma!)^2} \sum_{k=0}^{\gamma} \left(\binom{\gamma}{k} \frac{(-1)^k \tau^{\gamma+k-n+1}}{(\gamma+k+1)} \prod_{i=1}^{n} (\gamma+k-i+2) \right) \tag{19}$$

In the last step we can derive the n-th derivative of (14) ($n = 1, ..., \gamma$).

$$\frac{\mathrm{d}^n}{\mathrm{d}t^n} \mathbf{y}_d(t) = \begin{cases} \mathbf{0} & \text{if } t < t_0 \\ (\mathbf{y}(t_{end}) - \mathbf{y}(t_0)) \left(\frac{1}{t_{end}-t_0}\right)^n \varphi_\gamma^{(n)}\left(\frac{t-t_0}{t_{end}-t_0}\right) & \text{if } t \in [t_0, t_{end}] \\ \mathbf{0} & \text{if } t > T \end{cases} \tag{20}$$

Glossary

IVP initial value problem. 1, 3, 5