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

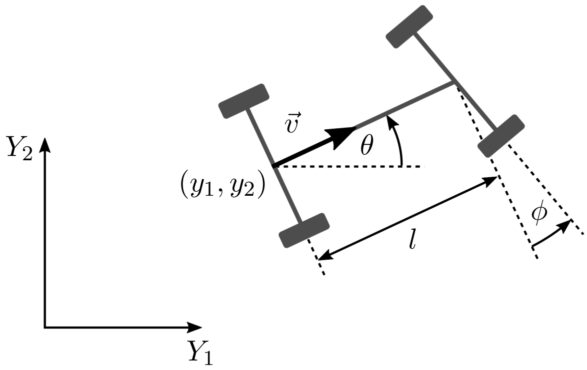

Figure 1: Car-like mobile robot

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* .

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

    Args:
        x: state
        t: time
        prmtrs(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 / prmtrs.l * u1 * tan(u2)])

    # return state derivative
    return dxdt
```

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

The control law is also implemented as function.

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

    Args:
        x: state vector
        t: time

    Returns:
        u: control vector

    """
    u = [1, 0.25]
    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.

```
prmtrs.l = 0.3  # define car length
prmtrs.w = prmtrs.l*0.3  # define car width

t0 = 0  # start time
tend = 10  # end time
dt = 0.04  # step-size

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

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

```
# initial state
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.

```
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

    """
```

```python
    # 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.

```python
# 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

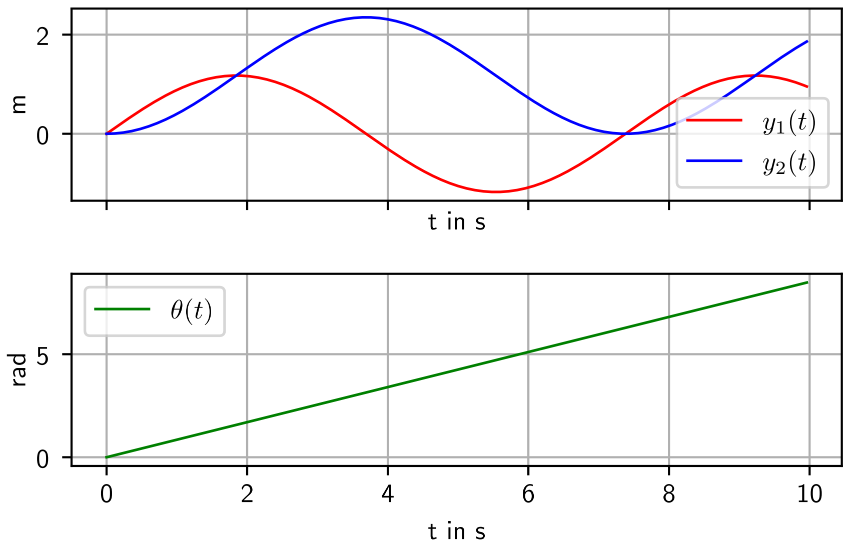

Figure 2: State trajectory plot created with *Matplotlib*

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 subpackage *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.

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

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

    Returns: None

    """
    dx = 1.5 * prmtrs.l
    dy = 1.5 * prmtrs.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),
```

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

```
                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 **x** and control signal **u**. This means we need to define a function inside *car_animation()* that maps from **x** and **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.

```python
    def init():
        """Initialize plot objects that change during animation.
           Only required for blitting to give a clean slate.

        Returns:

        """
        x_traj_plot.set_data([], [])
        car.set_data([], [])
        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.

```python
    def animate(i):
        """Defines what should be animated

        Args:
            i: frame number

        Returns:

        """
        k = i % len(t)
        ax.set_title('Time (s): ' + str(t[k]), loc='left')
        x_traj_plot.set_xdata(x[0:k, 0])
        x_traj_plot.set_ydata(x[0:k, 1])
        car_plot(x[k, :], control(x[k, :], t[k]))
        return x_traj_plot, car
```

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

```python
    ani = animation.FuncAnimation(fig2, animate, init_func=init,
                                  frames=len(t) + 1,
                                  interval=(t[1] - t[0]) * 1000,
                                  blit=False)

    ani.save('animation.mp4', writer='ffmpeg', fps=1 / (t[1] - t[0]))
    plt.show()
    return None
```

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

```python
# animation
car_animation(x_traj, u_traj, tt, prmtrs)

plt.show()
```

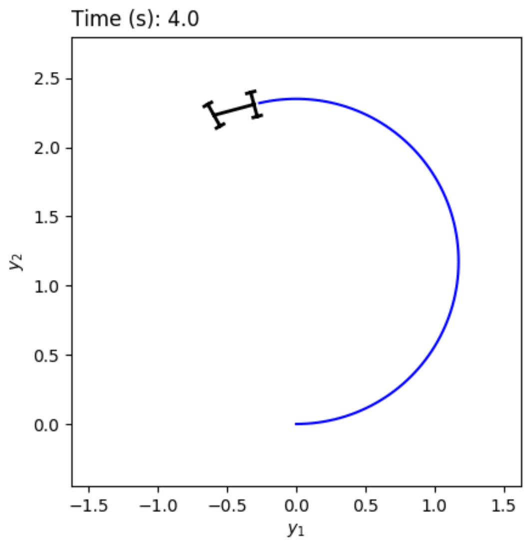

Figure 3: Car animation

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 (differentially) 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):

$$\tan(\phi) = \frac{l}{\underbrace{\sqrt{\dot{y}_1^2 + \dot{y}_2^2}}_{v}} \underbrace{\frac{\ddot{y}_1\dot{y}_2 - \dot{y}_1\ddot{y}_2}{(\dot{y}_1^2 + \dot{y}_2^2)}}_{\dot{\theta}} \tag{5a}$$
$$\Leftrightarrow \phi = \arctan\left(l\frac{\ddot{y}_1\dot{y}_2 - \dot{y}_1\ddot{y}_2}{(\dot{y}_1^2 + \dot{y}_2^2)^{\frac{3}{2}}}\right) \tag{5b}$$

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 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 r-th derivative of the flat output $\mathbf{y}$, until the input $\mathbf{u}$ shows up explicitly, then we set $\mathbf{y}^{(r)} = \mathbf{w}$.

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

We get a second order system for $\mathbf{y}$ that is linear to the input $\mathbf{w}$, this is called exact input-output linearization.

$$\ddot{\mathbf{y}} = \mathbf{w} \tag{8}$$

9.2.1 Stabilizing the linearized system

To stabilize system (8), we design a stabilizing feedback law for the input $\mathbf{w}$. We therefore define a differential equation for the tracking error $\mathbf{e}(t) = \mathbf{y}(t) - \mathbf{y}_d(t)$, where $\mathbf{y}_d$ is a reference trajectory:

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

We choose the matrices $\mathbf{K}_0$ and $\mathbf{K}_1$ such that the ODE is stable [5]. If we solve (9) for $\mathbf{w}$ we get:

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

[5] $\mathbf{K}_0$ and $\mathbf{K}_1$ could be diagonal matrices with positive entries

9.2.2 Determining control laws

To determine the control laws,we first substitute $\mathbf{w}$ in (5b) to solve for u_2:

$$u_2 = \arctan\left(l\frac{\ddot{y}_1\dot{y}_2 - \dot{y}_1\ddot{y}_2}{(\dot{y}_1^2 + \dot{y}_2^2)^{\frac{3}{2}}}\right) \tag{11a}$$

$$= \arctan\left(l\frac{w_1\dot{y}_2 - \dot{y}_1 w_2}{(\dot{y}_1^2 + \dot{y}_2^2)^{\frac{3}{2}}}\right) \tag{11b}$$

Then we take (7e) to solve for u_1, but what we find is a differential equation for u_1:

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

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

This ODE has to be solved by the controller to determine u_1. This kind of feedback is called endogenous dynamic state feedback. To do this, we add a 4-th state $x_4 = u_1$ to system (2) and replace u_2 with (11):

$$\underbrace{\begin{pmatrix}\dot{x}_1\\ \dot{x}_2\\ \dot{x}_3\\ \dot{x}_4\end{pmatrix}}_{\dot{\mathbf{x}}} = \underbrace{\begin{pmatrix} x_4\cos(x_3)\\ x_4\sin(x_3)\\ x_4\frac{w_1x_2 - x_1w_2}{(x_1^2+x_2^2)^{\frac{3}{2}}}\\ \frac{w_2}{\cos(x_3)} + x_4^2\tan(x_3)\frac{w_1x_2-x_1w_2}{(x_1^2+x_2^2)^{\frac{3}{2}}}\end{pmatrix}}_{f(\mathbf{x},\mathbf{w})} \tag{13}$$

The result is a new system $\dot{x} = f(\mathbf{x}, \mathbf{w})$.

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 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:

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

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

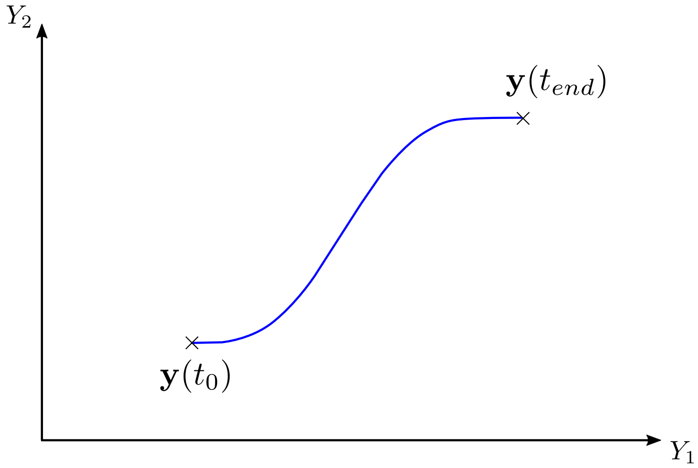

Figure 4: A feasible state transition from $\mathbf{y}(t_0)$ to $\mathbf{y}(t_{end})$

An approach for the derivative of $\varphi_\gamma(\tau)$, which meets the conditions (19) 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 $\varphi_\gamma(1) \stackrel{!}{=} 1$:

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

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 (18) ($n = 1, ..., \gamma$).

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