

Control Theory Tutorial

Car-Like Mobile Robot

Python for simulation, animation and control

Contents

1	Introduction	2
2	Model of a car-like mobile robot	2
3	Storing parameters	3
4	Libraries and Packages	3
5	Simulation with SciPy's integrate package	3
5.1	Solution of the initial value problem (IVP) using <i>SciPy</i>	4
6	Plotting using <i>Matplotlib</i>	4
7	Animation using <i>Matplotlib</i>	6
8	Simulation with SciPy's new <i>solve_ivp</i> module and the <i>lambda</i> function	10
9	(Differential) flatness based tracking control	11
9.1	(Differential) flatness	11
9.2	State feedback via exact input-output linearization	12
9.2.1	Stabilizing the linearized system	12
9.2.2	Control law	13
9.3	Calculating a reference trajectory (path planner)	13
9.4	Implementation of the controller in <i>Python</i>	16

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

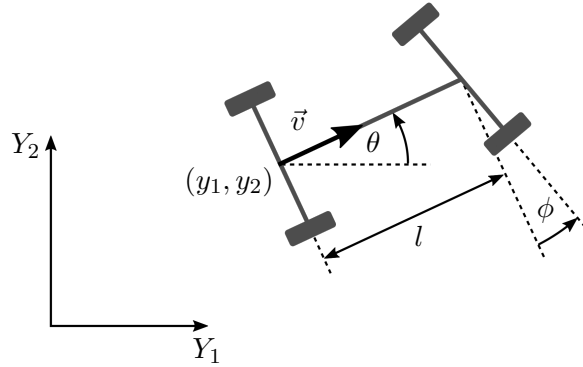


Figure 1: Car-like mobile robot

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 $v = |\mathbf{v}|$.

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

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

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

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

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

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

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

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
```

¹corresponding 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 example 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 $\text{length}(\text{tt}) \times \text{length}(\mathbf{x})$.

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     """
```

```

57 # creating a figure with 2 subplots, that share the x-axis
58 fig1, (ax1, ax2) = plt.subplots(2, sharex=True)
59
60 # set figure size to desired values
61 fig1.set_size_inches(fig_width / 2.54, fig_height / 2.54)
62
63 # plot y_1 in subplot 1
64 ax1.plot(t, x[:, 0], label='$y_1(t)$', lw=1, color='r')
65
66 # plot y_2 in subplot 1
67 ax1.plot(t, x[:, 1], label='$y_2(t)$', lw=1, color='b')
68
69 # plot theta in subplot 2
70 ax2.plot(t, x[:, 2], label=r'$\theta(t)$', lw=1, color='g')
71
72 ax1.grid(True)
73 ax2.grid(True)
74 # set the labels on the x and y axis in subplot 1
75 ax1.set_ylabel(r'm')
76 ax1.set_xlabel(r't in s')
77 ax2.set_ylabel(r'rad')
78 ax2.set_xlabel(r't in s')
79
80 # put a legend in the plot
81 ax1.legend()
82 ax2.legend()
83
84 # automatically adjusts subplot to fit in figure window
85 plt.tight_layout()
86
87 # save the figure in the working directory
88 if save:
89     plt.savefig('state_trajectory.pdf') # save output as pdf
90     plt.savefig('state_trajectory.pgf') # for easy export to LaTeX
91 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.

```

115 # plot
116 plot_data(x_traj, u_traj, tt, 12, 8, save=True)
117
118 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>

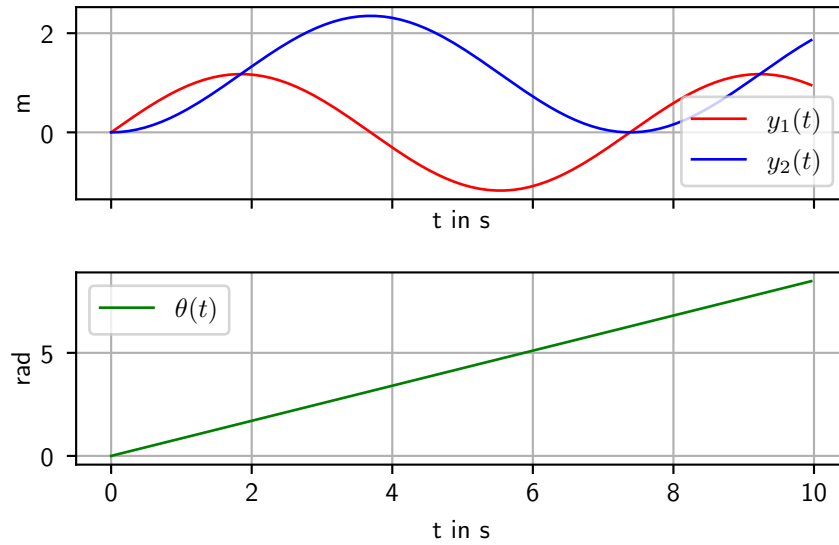


Figure 2: State trajectory plot created with *Matplotlib*

7 Animation using *Matplotlib*

² Plotting the state trajectory is often sufficient, but sometimes it can be helpful to have a visual representation 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),

```

²corresponding file: *car-like-mobile-robot-animation.py*

```

112         max(max(x_traj[:, 0] + dx), dx)])
113     ax.set_ylim([min(min(x_traj[:, 1] - dy), -dy),
114                 max(max(x_traj[:, 1] + dy), dy)])
115     ax.set_aspect('equal')
116     ax.set_xlabel(r'$y_1$')
117     ax.set_ylabel(r'$y_2$')
118
119     x_traj_plot, = ax.plot([], [], 'b') # state trajectory in the y1-y2-plane
120     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*.

```

122     def car_plot(x, u):
123         """Mapping from state x and action u to the position of the car elements
124
125         Args:
126             x: state vector
127             u: action vector
128
129         Returns:
130             car:
131
132         """
133         wheel_length = 0.1 * prmtrs.l
134         y1, y2, theta = x
135         v, phi = u
136
137         # define chassis lines
138         chassis_y1 = [y1, y1 + prmtrs.l * cos(theta)]
139         chassis_y2 = [y2, y2 + prmtrs.l * sin(theta)]
140
141         # define lines for the front and rear axle
142         rear_ax_y1 = [y1 + prmtrs.w * sin(theta), y1 - prmtrs.w * sin(theta)]
143         rear_ax_y2 = [y2 - prmtrs.w * cos(theta), y2 + prmtrs.w * cos(theta)]
144         front_ax_y1 = [chassis_y1[1] + prmtrs.w * sin(theta + phi),
145                       chassis_y1[1] - prmtrs.w * sin(theta + phi)]
146         front_ax_y2 = [chassis_y2[1] - prmtrs.w * cos(theta + phi),
147                       chassis_y2[1] + prmtrs.w * cos(theta + phi)]
148
149         # define wheel lines
150         rear_l_wl_y1 = [rear_ax_y1[1] + wheel_length * cos(theta),
151                       rear_ax_y1[1] - wheel_length * cos(theta)]
152         rear_r_wl_y1 = [rear_ax_y1[1] - wheel_length * cos(theta),
153                       rear_ax_y1[1] + wheel_length * cos(theta)]
154         front_l_wl_y1 = [front_ax_y1[1] + wheel_length * sin(theta + phi),
155                          front_ax_y1[1] - wheel_length * sin(theta + phi)]
156         front_r_wl_y1 = [front_ax_y1[1] - wheel_length * sin(theta + phi),
157                          front_ax_y1[1] + wheel_length * sin(theta + phi)]
158         rear_l_wl_y2 = [rear_ax_y2[1] + wheel_length * sin(theta),
159                       rear_ax_y2[1] - wheel_length * sin(theta)]
160         rear_r_wl_y2 = [rear_ax_y2[1] - wheel_length * sin(theta),
161                       rear_ax_y2[1] + wheel_length * sin(theta)]
162         front_l_wl_y2 = [front_ax_y2[1] - wheel_length * cos(theta + phi),
163                          front_ax_y2[1] + wheel_length * cos(theta + phi)]
164         front_r_wl_y2 = [front_ax_y2[1] + wheel_length * cos(theta + phi),
165                          front_ax_y2[1] - wheel_length * cos(theta + phi)]
166
167         # concatenate set of coordinates
168         data_y1 = [rear_ax_y1, empty, front_ax_y1, empty, chassis_y1,
169                   empty, rear_l_wl_y1, empty, rear_r_wl_y1,
170                   empty, front_l_wl_y1, empty, front_r_wl_y1]
171         data_y2 = [rear_ax_y2, empty, front_ax_y2, empty, chassis_y2,
172                   empty, rear_l_wl_y2, empty, rear_r_wl_y2,
173                   empty, front_l_wl_y2, empty, front_r_wl_y2]
174
175         # set data
176         car.set_data(data_y1, data_y2)
177         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()

```

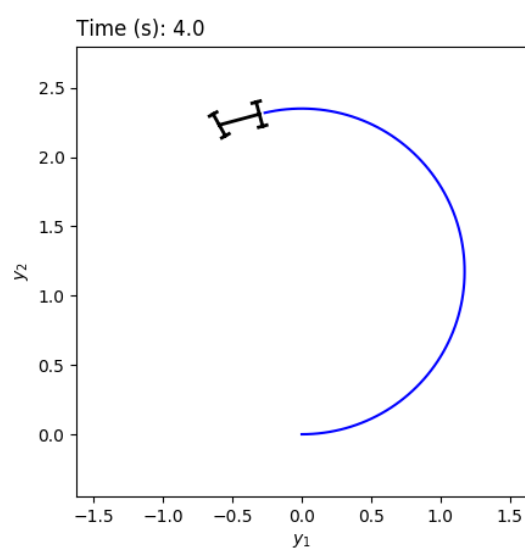



Figure 3: Car animation

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 (t_0, t_{end}) which defines the simulation interval, the initial value x_0 . 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)
```

³corresponding file: *car-like_mobile_robot_lambda.py*

⁴the 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 $\mathbf{z} = \psi(\mathbf{y}, \dot{\mathbf{y}}, \dots, \mathbf{y}^{(\alpha)})$ for $\mathbf{y} = (y_1, y_2)$ exists.

Recapture system (1) from section 2:

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

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

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

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

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

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

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

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

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

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

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

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

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

$\mathbf{y} = (y_1, y_2)$ 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.

$$y_1 = x_1 \quad (7a)$$

$$y_2 = x_2 \quad (7b)$$

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

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

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

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

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

9.2.1 Stabilizing the linearized system

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

$$0 = \ddot{\mathbf{e}} + \mathbf{K}_1 \dot{\mathbf{e}} + \mathbf{K}_0 \mathbf{e} \quad (9)$$

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:

$$w_1 = \dot{y}_2 = \dot{y}_{2,d} - k_{0,2}(y_2 - y_{2,d}) \quad (10a)$$

$$w_2 = \ddot{y}_1 = \ddot{y}_{1,d} - k_{1,1}(\dot{y}_1 - \dot{y}_{1,d}) - k_{0,1}(y_1 - y_{1,d}) \quad (10b)$$

9.2.2 Control law

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

$$u_1 = \frac{w_1}{\sin x_3} \quad (11a)$$

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

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

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

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

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

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

$$\dot{w}_1 = \ddot{y}_2 = \frac{dw_1}{dt} = \frac{d}{dt}(\dot{y}_{2,d} - k_{0,2}(y_2 - y_{2,d})) \quad (13a)$$

$$= \ddot{y}_{2,d} - k_{0,2} \underbrace{(\dot{y}_2 - \dot{y}_{2,d})}_{=w_1} \quad (13b)$$

$$= \ddot{y}_{2,d} - k_{0,2}(\dot{y}_{2,d} - k_{0,2}(y_2 - y_{2,d}) - \dot{y}_{2,d}) \quad (13c)$$

$$\Leftrightarrow \dot{w}_1 = \ddot{y}_{2,d} - k_{0,2}^2(y_2 - y_{2,d}) \quad (13d)$$

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

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

$\tau \rightarrow \varphi_\gamma(\tau)$ is a prototype 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:

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

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

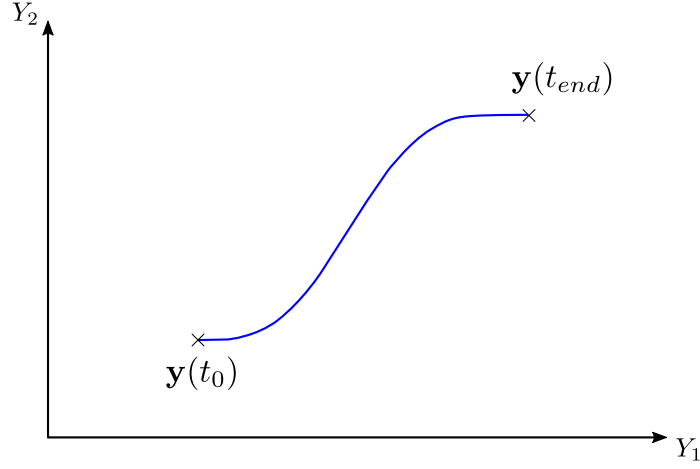


Figure 4: A feasible state transition from $\mathbf{y}(t_0)$ to $\mathbf{y}(t_{end})$

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

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

Integration leads to:

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

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:

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

and it's n -th derivative:

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

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

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