

Python for simulation, animation and control

# Part 2: Tutorial for proper definition, computation and usage of reference trajectories

Demonstration using the model of a kinematic vehicle

Max Pritzkolet\*      Jan Winkler\*

January 7, 2022

## Contents

<b>1. Introduction</b>	<b>3</b>
<b>2. Trajectories for smooth point-to-point transitions</b>	<b>3</b>
2.1. Polynomials . . . . .	4
2.2. Polynomials using a prototype function . . . . .	6
2.3. Gevrey functions . . . . .	7
<b>3. Implementing trajectory generators in Python</b>	<b>7</b>
3.1. The trajectory generator base class . . . . .	7
3.2. The <i>PolynomialTrajGen</i> subclass . . . . .	9
3.3. The <i>PrototypeTrajGen</i> subclass . . . . .	12
<b>4. Feedforward control design</b>	<b>12</b>
4.1. Re-parameterization of the model . . . . .	12
4.2. Deriving feedforward control laws . . . . .	13

---

\*Institute of Control Theory, Faculty of Electrical and Computer Engineering, Technische Universität Dresden, Germany

---

4.3. Implementation . . . . .	15
4.3.1. Result . . . . .	17
<b>5. Feedback control design</b>	<b>20</b>
5.1. Deriving feedback control laws . . . . .	20
5.2. Implementation . . . . .	21
5.2.1. Result . . . . .	23
<b>Appendices</b>	<b>26</b>
<b>A. The prototype polynomial</b>	<b>26</b>
<b>B. Trajectory generator based on a Gevrey function</b>	<b>27</b>
B.1. Definition . . . . .	27
B.2. Efficient calculation of derivatives . . . . .	28
B.3. The <i>GevreyTrajGen</i> subclass . . . . .	29

## 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 different trajectory generators in a Python class hierarchy,
- flatness based feedforward control
- flatness based feedback control.

Additionally, some aspects of object-oriented programming are covered as required to solve the presented problems.

Later in this tutorial the trajectory generators are used to design control strategies for the car model.

Please refer to the first part of this tutorial series addressing basic concepts for the simulation of dynamic systems if you are not familiar with this [3]. It is available on GitHub:

<https://github.com/TUD-RST/pytutorials>.

The book [1] is also recommended (in German only).

## 2. Trajectories for smooth point-to-point transitions

In control theory, a common task is to transfer a system quantity from an initial value  $y^A$  at time  $t_0$  to a new value  $y^B$  at time  $t_f$ . The boundary conditions at  $t_0$  and  $t_f$  are a crucial part in the planning of such transfers. For example, if  $y$  denotes a position coordinate and a simple trapezoidal interpolation in time between the two points  $y^A$  and  $y^B$  is used, the value of the acceleration  $\ddot{y}$  at  $t_0$  and  $t_f$  would approach infinity. This cannot be fulfilled by any real system due to its inertia. That is why, when a point-to-point transition is planned, the derivative of the planned trajectory has to be smooth up to a certain degree. Figure 1 shows an example of a "smooth" trajectory.

Mathematically, the trajectory for a scalar quantity  $y$  can be described by a function  $f_y$  with  $f_y : \mathbb{R} \rightarrow \mathbb{R}$ . This function maps the domain of definition  $\mathbb{R}$  onto the co-domain  $\mathbb{R}$ . This means that the function  $f_y$  maps each time point  $t \in \mathbb{R}$  in a certain way to a value  $f_y(t) \in \mathbb{R}$ . Usually this is denoted by  $t \mapsto f_y(t)$ . It is quite annoying to denote trajectories as  $t \mapsto f_y(t)$  accompanied by the formal definition of a function  $f_y$ . Hence, as a shortcut, one simply writes  $y = f_y(t)$  although – literally speaking – this notion just denotes the value of  $f_y$  at time  $t$ . Especially in engineering science an even shorter

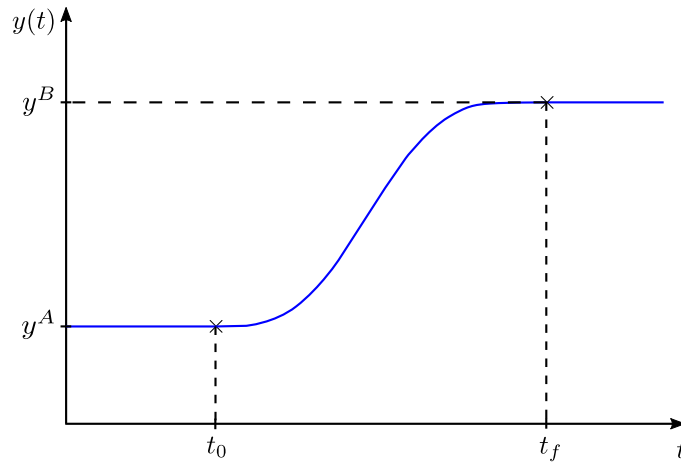


Figure 1: Smooth state transition of a quantity  $y$  from  $y^A$  to  $y^B$  on the time interval  $t_f - t_0$ .

notion is very common: One writes  $y(t)$  instead of  $t \mapsto f_y(t)$ . This notion will be used in this tutorial, too.

## 2.1. Polynomials

A simple way of defining a trajectory which realizes a smooth transfer between two values on the time interval  $t_f - t_0$  and which is  $d$  times continuously differentiable is the piecewise definition of a function  $y_d(t)$  as follows:

$$y_d(t) = \begin{cases} y^A & \text{if } t < t_0 \\ \sum_{i=0}^{2d+1} c_i \frac{t^i}{i!} & \text{if } t \in [t_0, t_f] \\ y^B & \text{if } t > t_f \end{cases} \quad (2.1)$$

with a polynomial

$$y_d(t) = \sum_{i=0}^{2d+1} c_i \frac{t^i}{i!} = c_0 + c_1 t + \frac{c_2}{2} t^2 + \frac{c_3}{6} t^3 + \dots + \frac{c_{2d+1}}{(2d+1)!} t^{2d+1} \quad (2.2)$$

of degree  $2d + 1$  which is evaluated on the interval  $[t_0, t_1]$  only. Here,  $2d + 2$  boundary conditions for the determination of the coefficients  $c_0, \dots, c_{2d+1}$  need to be fulfilled at  $t = t_0$  and  $t = t_f$  ( $d + 1$  at each one).

For  $t \in [t_0, t_f]$   $y_d(t)$  and its successive derivatives up to order  $d$  can be written down in matrix form:

$$\underbrace{\begin{pmatrix} y_d(t) \\ \dot{y}_d(t) \\ \vdots \\ y_d^{(d-1)}(t) \\ y_d^{(d)}(t) \end{pmatrix}}_{:=\mathbf{Y}_d(t) \in \mathbb{R}^{(d+1)}} = \underbrace{\begin{pmatrix} 1 & t & \frac{t^2}{2!} & \cdots & \frac{t^{2d+1}}{(2d+1)!} \\ 0 & 1 & t & \cdots & \frac{t^{2d}}{(2d)!} \\ 0 & 0 & 1 & \cdots & \frac{t^{2d-1}}{(2d-1)!} \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & 1 \end{pmatrix}}_{:=\mathbf{T}(t) \in \mathbb{R}^{(d+1) \times (2d+2)}} \underbrace{\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{2d-1} \\ c_{2d} \\ c_{2d+1} \end{pmatrix}}_{:=\mathbf{c} \in \mathbb{R}^{(2d+2)}}. \quad (2.3)$$

To calculate the parameter vector  $\mathbf{c}$ , the boundary conditions of the trajectory at  $t_0$  and  $t_f$  have to be defined up to degree  $d$ :

$$\underbrace{\begin{pmatrix} y_d(t_0) \\ \dot{y}_d(t_0) \\ \vdots \\ y_d^{(d)}(t_0) \end{pmatrix}}_{:=\mathbf{Y}_d(t_0)} \stackrel{!}{=} \underbrace{\begin{pmatrix} y^A \\ \dot{y}^A \\ \vdots \\ y^{(d)A} \end{pmatrix}}_{:=\mathbf{Y}^A}, \quad \underbrace{\begin{pmatrix} y_d(t_f) \\ \dot{y}_d(t_f) \\ \vdots \\ y_d^{(d)}(t_f) \end{pmatrix}}_{:=\mathbf{Y}_d(t_f)} \stackrel{!}{=} \underbrace{\begin{pmatrix} y^B \\ \dot{y}^B \\ \vdots \\ y^{(d)B} \end{pmatrix}}_{:=\mathbf{Y}^B}.$$

This leads to a linear equation system:

$$\begin{bmatrix} \mathbf{Y}_d(t_0) \\ \mathbf{Y}_d(t_f) \end{bmatrix} = \begin{bmatrix} \mathbf{Y}^A \\ \mathbf{Y}^B \end{bmatrix} = \begin{bmatrix} \mathbf{T}(t_0) \\ \mathbf{T}(t_f) \end{bmatrix} \mathbf{c}.$$

Because  $\begin{bmatrix} \mathbf{T}(t_0) \\ \mathbf{T}(t_f) \end{bmatrix}$  is quadratic and not singular for  $t_0 \neq t_f$ , this linear equation system can be solved explicitly:

$$\mathbf{c} = \begin{bmatrix} \mathbf{T}(t_0) \\ \mathbf{T}(t_f) \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{Y}^A \\ \mathbf{Y}^B \end{bmatrix}. \quad (2.4)$$

Because the calculation of the inverse matrix is computationally expensive one is strongly advised to use a more efficient linear equation system solver, like `linalg.solve()` from NumPy, to solve for the vector  $\mathbf{c}$ .

$\mathbf{Y}_d(t)$  can be calculated in a closed form:

$$\mathbf{Y}_d(t) = \mathbf{T}(t)\mathbf{c} \quad t \in [t_0, t_f] \quad (2.5)$$

The full trajectory can be computed by evaluating [Equation 2.1](#).

**Remark:** Computational effort can be reduced if all derivatives at  $t_0$  and  $t_f$  are zero (steady state transfer). Then, one can formulate [Equation 2.1](#) as

$$y_d(t) = \begin{cases} y^A & \text{if } t < t_0 \\ \sum_{i=0}^{2d+1} c_i \frac{(t-t_0)^i}{i!} & \text{if } t \in [t_0, t_f] \\ y^B & \text{if } t > t_f \end{cases} \quad (2.6)$$

Evaluating the boundary conditions at  $t_0$  one observes  $c_0 = y_d(t_0)$  and  $c_1, \dots, c_{d+1} = 0$ . Then, the remaining coefficients  $c_{d+2}, \dots, c_{2d+2}$  can be computed from

$$\begin{pmatrix} c_{d+2} \\ \vdots \\ c_{2d+2} \end{pmatrix} = \mathbf{T}_{d+1 \times d+1}^{-1}(t_f) \mathbf{Y}^B \quad (2.7)$$

where  $\mathbf{T}_{d+1 \times d+1}$  is the quadratic lower right block matrix of [Equation 2.3](#) evaluated at  $t = t_f - t_0$ .

## 2.2. Polynomials using a prototype function

A slightly different approach for a polynomial reference trajectory  $y_d(t)$  is again a piecewise-defined function:

$$y_d(t) = \begin{cases} y^A & \text{if } t < t_0 \\ y^A + (y^B - y^A) \varphi_\gamma \left( \frac{t-t_0}{t_f-t_0} \right) & \text{if } t \in [t_0, t_f] \\ y^B & \text{if } t > t_f \end{cases} \quad (2.8)$$

with a prototype function  $\tau \rightarrow \varphi_\gamma(\tau)$ . The parameter  $\gamma$  denotes how often  $\varphi_\gamma(\tau)$  is continuously differentiable. The prototype function meets the following boundary conditions:

$$\varphi_\gamma(0) = 0 \quad \varphi_\gamma^{(j)}(0) = 0 \quad j = 1, \dots, \gamma, \quad (2.9)$$

$$\varphi_\gamma(1) = 1 \quad \varphi_\gamma^{(j)}(1) = 0 \quad j = 1, \dots, \gamma, \quad (2.10)$$

and is given by

$$\varphi_\gamma(\tau) = \frac{(2\gamma+1)!}{(\gamma!)^2} \sum_{k=0}^{\gamma} \binom{\gamma}{k} \frac{(-1)^k \tau^{\gamma+k+1}}{(\gamma+k+1)} \quad (2.11)$$

with its  $n$ th derivative as

$$\varphi_{\gamma}^{(n)}(\tau) = \frac{(2\gamma + 1)!}{(\gamma!)^2} \sum_{k=0}^{\gamma} \left( \binom{\gamma}{k} \frac{(-1)^k \tau^{\gamma+k-n+1}}{(\gamma + k + 1)} \prod_{i=1}^n (\gamma + k - i + 2) \right). \quad (2.12)$$

Note that the argument  $\frac{t-t_0}{t_f-t_0}$  in [Equation 2.8](#) runs from 0 to 1 on the definition interval, indeed!

The expression for the polynomial prototype is derived in [Appendix A](#).

### 2.3. Gevrey functions

Gevrey functions can come into play when you need to do trajectory planning for the control of infinite dimensional systems (e.g. systems with spatially distributed parameters). One type of such a function is provided by the trajectory generator class. Refer to [Appendix B](#) for more details.

## 3. Implementing trajectory generators in Python

In order to automate the process of trajectory planning at first a trajectory generator base class is implemented. Then a new subclass for each new planning algorithm is created.

**Python Source code file: `TrajGen.py`**

### 3.1. The trajectory generator base class

A trajectory generator class realizing the smooth transfer from  $y^A$  to  $y^B$  on the interval  $[t_0, t_f]$  should have the following attributes:

- YA - vector of  $y$  and it's derivatives up to order  $d$  at start time  $t_0$
- YB - vector of  $y$  and it's derivatives up to order  $d$  at final time  $t_f$
- $t_0$  - start time of the point-to-point transition
- $t_f$  - final time of the point-to-point transition
- $d$  - planned trajectory should be smooth up to the  $d$ -th derivative

The planned trajectory has to be evaluated at runtime, but the implementation of this feature should be defined in the specific subclass depending on the algorithm. Therefore, we define a base class `TrajGenBase` from which the several types of trajectory generator implementations need to be derived. By using an abstract base class method, we force a subclass of `TrajGenBase` to provide the methods `eval()` and `eval_vec()` for evaluation of the trajectory at a single point or for a series of points stored in an array-like container, respectively. For this purpose we need to import the `abc` module and need to derive the base class from the `ABC` class:

```

2 import numpy as np
3 import scipy.special as sps
4 import abc # abstract base class
5
6
7 class TrajGen(abc.ABC):
8     """ Base class for a trajectory generator.
9
10    Attributes:
11        y_a (int, float, ndarray): start value (size = d+1)
12        y_b (int, float, ndarray): final value (size = d+1)
13        t_0 (int, float): start time
14        t_f (int, float): final time
15        d (int): trajectory is smooth up at least to the d-th derivative
16    """
17
18    def __init__(self, y_a, y_b, t_0, t_f, d):
19        self.YA = y_a
20        self.YB = y_b
21        self.t0 = t_0
22        self.tf = t_f
23        self.d = d
24
25    @abc.abstractmethod
26    def eval(self, t):
27        pass
28
29    @abc.abstractmethod
30    def eval_vec(self, t):
31        pass

```

Note that any class derived from `TrajGenBase` needs to implement the methods decorated by `@abstractmethod`, otherwise a `TypeError` exception will be thrown when they are instantiated.

Note the special method `factorial` defined in the base class:

```

35 @classmethod
36 def factorial(cls, x):
37     """Calcualtes the faculty of x"""
38     result = 1
39     for i in range(2, x + 1):
40         result *= i
41     return result

```

It is used for calculation of the factorial  $x!$  of an integer  $x$ . Since it does not depend on any member variables it is defined as a class method. Class methods can be called



without instantiation of a class, and they have only access to class attributes (defined outside `__init__`)<sup>1</sup>. It is implemented here for didactic purposes only. Usually you would use `factorial` from the `math` or `scipy.special` module.

### 3.2. The *PolynomialTrajGen* subclass

To implement the planning algorithm that was developed in [subsection 2.1](#), a new class `PolynomialTrajGen` is created that inherits from the previously defined class `TrajGenBase`. All the attributes and methods of `TrajGenBase` are now also attributes and methods of `PolynomialTrajGen`. We define an additional attribute `c` holding the vector of coefficients of [Equation 2.1](#):

```

46 class PolynomialTrajGen(TrajGen):
47     """TrajGen subclass that uses a polynomial approach for trajectory generation
48
49     Attributes:
50         c (ndarray): parameter vector of polynomial
51
52     """
53     def __init__(self, y_a, y_b, t_0, t_f, d):
54         super().__init__(y_a, y_b, t_0, t_f, d)
55         self.c = self.coefficients()
```

The built-in function `super()` tells the interpreter to call the constructor of the parent class of `PolynomialTrajGen`. As the code in `__init__` shows the coefficients are calculated. This is explained in the following.

To solve for the parameter vector  $\mathbf{c}$ , the matrix  $\mathbf{T}(t)$  from [Equation 2.3](#) is calculated and a method `t_matrix()` is therefore created:

```

95 def t_matrix(self, t):
96     """Computes the T matrix at time t
97
98     Args:
99         t (int, float): time
100
101     Returns:
102         t_mat (ndarray): T matrix
103
104     """
105     d = self.d
106     n = d+1 # first dimension of T
107     m = 2*d+2 # second dimension of T
108
109     t_mat = np.zeros([n, m])
110
111     for i in range(0, m):
112         t_mat[0, i] = t ** i / self.factorial(i)
113     for j in range(1, n):
114         t_mat[j, j:m] = t_mat[0, 0:m-j]
115     return t_mat
116
```

<sup>1</sup>In C++ and Java such methods are named *static*.

Of course, this could be done more efficiently in a recursive way (try it!). Then, a method, that solves [Equation 2.4](#) and returns the parameter vector  $\mathbf{c}$  is needed:

```

120 def coefficients(self):
121     """Calculation of the polynomial parameter vector
122
123     Returns:
124         c (ndarray): parameter vector of the polynomial
125
126     """
127     t0 = self.t0
128     tf = self.tf
129
130     y = np.append(self.YA, self.YB)
131
132     t0_mat = self.t_matrix(t0)
133     tf_mat = self.t_matrix(tf)
134
135     t_mat = np.append(t0_mat, tf_mat, axis=0)
136
137     # solve the linear equation system for c
138     c = np.linalg.solve(t_mat, y)
139
140     return c

```

Finally a method `eval()` that implements [Equation 2.5](#) is defined:

```

59 def eval(self, t):
60     """Evaluates the planned trajectory at time t.
61
62     Args:
63         t (int, float): time
64
65     Returns:
66         y (ndarray): y and its derivatives at t
67
68     """
69     if t < self.t0:
70         y = self.YA
71     elif t > self.tf:
72         y = self.YB
73     else:
74         y = np.dot(self.t_matrix(t), self.c)
75     return y

```

Furthermore, a second method `eval_vec()` is implemented, that can handle a time array as an input:

```

78 def eval_vec(self, tt):
79     """Samples the planned trajectory
80
81     Args:
82         tt (ndarray): time vector
83
84     Returns:
85         y (ndarray): y and its derivatives at the sample points
86
87     """
88     y = np.zeros([len(tt), len(self.YA)])
89     for i in range(0, len(tt)):
90         y[i] = self.eval(tt[i])
91     return y

```

The polynomial trajectory generator is now successfully implemented and can be tested.

### Example:

#### Python source code file: **01\_trajectory\_planning.py**

Suppose a trajectory from  $y(t_0) = 0$  to  $y(t_f) = 1$  with  $t_0 = 1s$  and  $t_f = 2s$  has to be planned. The trajectory should be smoothly differentiable twice ( $d = 2$ ). Therefore, the boundary conditions for the first and second derivative of  $y$  have to be defined:

$$\begin{aligned} \dot{y}(t_0) &= 0 & \dot{y}(t_f) &= 0 \\ \ddot{y}(t_0) &= 0 & \ddot{y}(t_f) &= 0 \end{aligned}$$

The total time interval for the evaluation of the trajectory is  $t \in [0s, 3s]$ .

At first the boundary conditions for  $t = t_0$  and  $t = t_f$  are set:

```
8 YA = np.array([0, 0, 0]) # t = t0
9 YB = np.array([1, 0, 0]) # t = tf
```

After that the start and final time of the transition and the total time interval:

```
13 t0 = 0 # start time of transition
14 tf = 1 # final time of transition
15 tt = np.linspace(t0, tf, 100) # -1 to 4 in 500 steps
```

Then  $d$  is set and a `PolynomialTrajGen` instance `yd` with the defined parameters is created.

```
19 d = 2 # smooth derivative up to order d
20 yd = PolynomialTrajGen(YA, YB, t0, tf, d)
```

The calculated parameters can be displayed

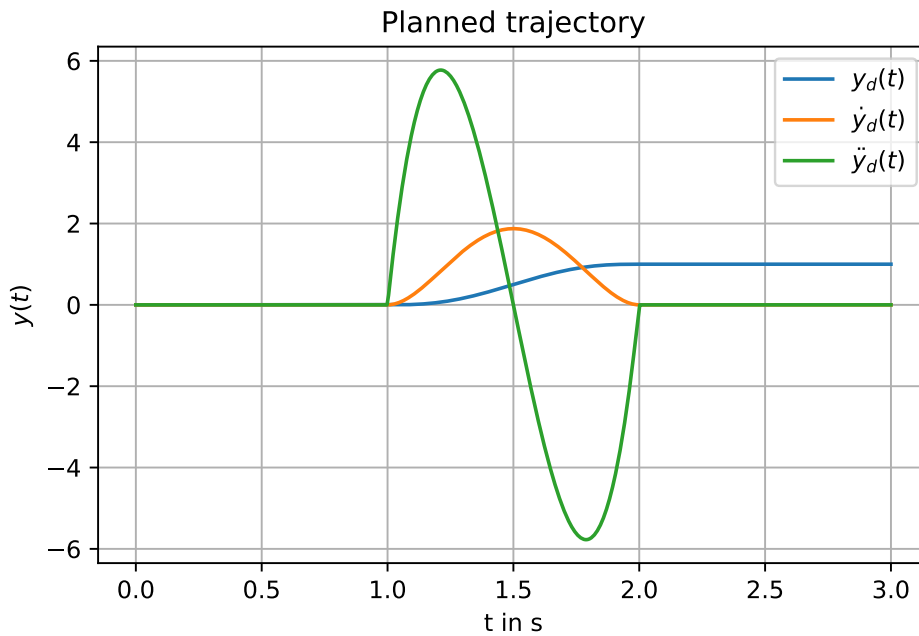
```
24 print("c = ", yd.c)
```

and the generated trajectory at the defined total time interval can be evaluated

```
28 Y = yd.eval_vec(tt)
```

At last, the results are plotted.

```
28 Y = yd.eval_vec(tt)
29
30 # plot the trajectory
31 plt.figure(1)
32 plt.plot(tt, Y)
33 plt.title('Planned trajectory')
34 plt.legend([r'$y_d(t)$', r'$\dot{y}_d(t)$', r'$\ddot{y}_d(t)$'])
35 plt.xlabel(r't in s')
36 plt.grid(True)
```



### 3.3. The *PrototypeTrajGen* subclass

Python Source code file: **TrajGen.py**

The implementation can be found in the source code file. It is not detailed here since it is similar to the *PolynomialTrajGen*.

## 4. Feedforward control design

Python Source code file: **02\_car\_feedforward\_control.py**

Recapture the model of the car from tutorial 1 [3], parameterized in time  $t$ :

$$\dot{y}_1 = v \cos(\theta) \quad (4.1a)$$

$$\dot{y}_2 = v \sin(\theta) \quad (4.1b)$$

$$\dot{\theta} = \frac{v}{l} \tan(\varphi). \quad (4.1c)$$

### 4.1. Re-parameterization of the model

The model of the car has to be parameterized in arc length  $s$  to take care of singularities that would appear in steady-state ( $v = 0$ ).

The following can be assumed:

$$\frac{d}{dt} = \frac{d}{dt} \frac{ds}{ds} = \frac{d}{ds} \frac{ds}{dt} = \frac{d}{ds} \dot{s}.$$

Replacing  $\frac{d}{dt}$  in the model equations leads to:

$$\frac{d}{ds} \dot{s} y_1 = v \cos(\theta) \quad (4.2a)$$

$$\frac{d}{ds} \dot{s} y_2 = v \sin(\theta) \quad (4.2b)$$

$$\frac{d}{ds} \dot{s} \theta = \frac{v}{l} \tan(\varphi) \quad (4.2c)$$

$$v = |\dot{\mathbf{y}}| = \sqrt{\dot{y}_1^2 + \dot{y}_2^2}. \quad (4.3)$$

This equation is parameterized in  $s$ :<sup>2</sup>

$$v = \sqrt{\left(\frac{d}{ds} \dot{s} y_1\right)^2 + \left(\frac{d}{ds} \dot{s} y_2\right)^2} = \dot{s} \sqrt{(y_1')^2 + (y_2')^2}. \quad (4.4)$$

If  $s$  is the arc length, the Pythagorean theorem  $ds^2 = dy_1^2 + dy_2^2$  leads to:

$$1 = \left(\frac{dy_1}{ds}\right)^2 + \left(\frac{dy_2}{ds}\right)^2 \quad (4.5a)$$

$$\Leftrightarrow 1 = \sqrt{(y_1')^2 + (y_2')^2}. \quad (4.5b)$$

Therefore,  $v = \dot{s}$ . The system parameterized in  $s$  is given by:

$$y_1' = \cos(\theta) \quad (4.6a)$$

$$y_2' = \sin(\theta) \quad (4.6b)$$

$$\theta' = \frac{1}{l} \tan(\varphi). \quad (4.6c)$$

## 4.2. Deriving feedforward control laws

Goal: Drive the car in the  $y_1$ - $y_2$ -plane from a point  $(y_{1A}, y_{2A})$  to a point  $(y_{1B}, y_{2B})$  in time  $T = t_f - t_0$ . The car should be in rest at the beginning and at the end of the process and the trajectory is defined by a sufficiently smooth function  $f : \mathbb{R} \rightarrow \mathbb{R}$  with  $y_2 = f(y_1)$ . Note that  $(y_1, y_2)$  is a flat output of the system.

---

<sup>2</sup>Assuming  $\dot{s} > 0$

**Step 1:** Calculate the dependency of the remaining system variables  $\theta$  and  $\varphi$  of the length parameterized system on  $(y_1, y_2)$ :

$$\begin{aligned}\tan(\theta) &= \frac{y_2'}{y_1'} = \frac{dy_2}{dy_1} = f'(y_1) \\ (1 + \tan^2(\theta)) \frac{d\theta}{dy_1} &= f''(y_1) \\ \Leftrightarrow \frac{d\theta}{dy_1} &= \frac{f''(y_1)}{1 + (f'(y_1))^2} = \frac{\theta'}{y_1'}.\end{aligned}\tag{4.7}$$

With  $(y_1')^2 + (y_2')^2 = 1 \Leftrightarrow y_1' = 1/\sqrt{1 + (f'(y_1))^2}$  one obtains:

$$\Leftrightarrow \theta' = \frac{f''(y_1)}{(1 + (f'(y_1))^2)^{3/2}}\tag{4.8}$$

$$\tan(\varphi) = l\theta' = \frac{l f''(y_1)}{(1 + (f'(y_1))^2)^{3/2}}.\tag{4.9}$$

Result: Depending on the planning  $y_2 = f(y_1)$  the required steering angle can be calculated solely from  $y_1$  and derivatives of  $f$  w.r.t.  $y_1$  up to order 2. The planned trajectory has to fulfill the following boundary conditions:

$$\begin{aligned}f(y_{1A}) &= y_{2A} & f(y_{1B}) &= y_{2B} \\ f'(y_{1A}) &= \tan(\theta_A) & f'(y_{1B}) &= \tan(\theta_B) \\ f''(y_{1A}) &= (1 + \tan^2(\theta_A)) \left( \frac{\frac{1}{l} \tan(\varphi_A)}{\cos(\theta_A)} \right) & f''(y_{1B}) &= (1 + \tan^2(\theta_B)) \left( \frac{\frac{1}{l} \tan(\varphi_B)}{\cos(\theta_B)} \right).\end{aligned}$$

By always setting  $\varphi_A = \varphi_B = 0$ , these conditions simplify to:

$$f''(y_{1A}) = 0 \qquad f''(y_{1B}) = 0$$

**Step 2:** Calculation of the required velocity  $v$ . Another function  $g : \mathbb{R} \rightarrow \mathbb{R}$  is defined, with  $y_1 = g(t)$  and  $g(t_0) = y_{1A}$ ,  $\dot{g}(t_0) = 0$ ,  $g(t_f) = y_{1B}$ ,  $\dot{g}(t_f) = 0$ .

$$v = \sqrt{\dot{y}_1^2 + \dot{y}_2^2} = \dot{y}_1 \sqrt{1 + (f'(y_1))^2} = \dot{g}(t) \sqrt{1 + (f'(g(t)))^2}\tag{4.10}$$

Hence, the overall, time parameterized feedforward control reads:

$$v(t) = \dot{g}(t) \sqrt{1 + (f'(g(t)))^2}\tag{4.11a}$$

$$\varphi(t) = \arctan \left( \frac{l f''(g(t))}{(1 + (f'(g(t)))^2)^{3/2}} \right)\tag{4.11b}$$

Or expressed in  $s$ :

$$v(s) = \dot{s} \sqrt{y_2'^2 + y_1'^2} \quad (4.12a)$$

$$\varphi(s) = \arctan \left( l \frac{y_2'' y_1' - y_1'' y_2'}{(y_1'^2 + y_2'^2)^{\frac{3}{2}}} \right) = \arctan (l(y_2'' y_1' - y_1'' y_2')) \quad (4.12b)$$

If polynomials are chosen for the two functions  $f$  and  $g$  it has to be ensured that  $f$  is of order 3 and  $g$  of order 2 to make sure the control law is smooth. The resulting  $f(g)$  is of order 5.

### 4.3. Implementation

For the implementation of the controller, the polynomial trajectory generator from [subsection 3.2](#) is used. At first all necessary simulation parameters are defined:

```

30 @dataclass
31 class SimPara:
32     t0: float = 0          # start time
33     tf: float = 10         # final time
34     dt: float = 0.04       # step-size
35     tt = np.arange(0, tf + dt, dt) # time vector
36     x0 = [0, 0, 0]         # initial state at t0
37     xf = [5, 5, 0]        # final state at tf

```

We define a function which sets up the trajectory generator objects for trajectory generation:

```

42 def setup_trajectories(sp: Type[SimPara]) -> List[PolynomialTrajGen]:
43     """Setup the trajectory objects
44
45     Args:
46         sp: Object of the SimPara
47
48     Returns:
49         A list holding two objects of type PolynomialTrajGen. The first
50         one is f, the second g.
51
52     """
53
54     # Start and final time of transistion
55     t0: float = sp.t0 + 1
56     tf: float = sp.tf - 1
57
58     # boundary conditions for y1
59     y1_a = np.array([sp.x0[0], 0])
60     y1_b = np.array([sp.xf[0], 0])
61
62     # boundary conditions for y2
63     y2_a = np.array([sp.x0[1], tan(sp.x0[2]), 0])
64     y2_b = np.array([sp.xf[1], tan(sp.xf[2]), 0])
65
66     # From Y2A to Y2B on the interval [Y1A[0], Y1B[0]]
67     f_traj = PolynomialTrajGen(y2_a, y2_b, y1_a[0], y1_b[0], 2)

```

```

68
69     # from Y1A to Y1B on the interval [t0, tf]
70     g_traj = PolynomialTrajGen(y1_a, y1_b, t0, tf, 1)
71
72     return [f_traj, g_traj]

```

Note, that we added type annotations here for the argument `sp` and the return value. This is optional. However, it is recommended since it improves static code analysis and IntelliSense features in your IDE. You need to import the typing module to use it for user defined types, lists and other complex data types. [More information about type hints...](#)

Within the right-hand side of the system differential equation the feedforward control law (4.11) needs to be evaluated. Hence, it needs to be implemented:

```

100 def control(t, p: Type[Para]):
101     """Function of the control law
102
103     Args:
104         t (int): time
105         p (object): parameter container class
106
107     Returns:
108         u (ndarray): control vector
109
110     """
111
112     # evaluate the planned trajectories at time t
113     g_t = g_traj_gen.eval(t)      # y1 = g(t)
114     f_y1 = f_traj_gen.eval(g_t[0]) # y2 = f(y1) = f(g(t))
115
116     # setting control laws
117     u1 = g_t[1]*np.sqrt(1 + f_y1[1]**2)
118     u2 = arctan2(p.l*f_y1[2], (1 + f_y1[1]**2)**(3/2))
119
120     return np.array([u1, u2]).T

```

Now, the global trajectory objects can be instantiated and simulation can be executed as usual:

```

326 # Setup the trajectories, we hold it as global objects here
327 [f_traj_gen, g_traj_gen] = setup_trajectories(SimPara)
328
329 # simulation
330 sol = sci.solve_ivp(lambda t, x: ode(x, t, Para), (SimPara.t0, SimPara.tf), SimPara.x0,
331                    method='RK45', t_eval=SimPara.tt)

```

The results can be extracted by:

```

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

```

To get the control vector one has to evaluate `control()` with the simulated trajectory again:

```

336 u_traj = np.zeros([len(SimPara.tt), 2])
337 for i in range(0, len(SimPara.tt)):
338     u_traj[i] = control(SimPara.tt[i], Para)

```



Because `control()` works only for scalar time values, this has to be done in a **for**-loop. This is a bit annoying, but there is no alternative. It is impossible to store the control values when they are calculated within the right-hand side of the differential equation. The reason is that it is usually called by a variable step solver which evaluates it multiple times between the required time steps.

Plotting the simulation results and the reference trajectories:

```

343 y1D = g_traj_gen.eval_vec(SimPara.tt)
344 y2D = f_traj_gen.eval_vec(y1D[:, 0])
345
346 x_ref = np.zeros_like(x_traj)
347 x_ref[:, 0] = y1D[:, 0]
348 x_ref[:, 1] = y2D[:, 0]
349 x_ref[:, 2] = arctan(y2D[:, 1])
350
351 # Plot results
352 plot_data(x_traj, x_ref, u_traj, SimPara.tt, 12, 16, save=True)
353
354 # animation
355 car_animation(x_traj, u_traj, SimPara.tt, Para)
356
357 plt.show()

```

Compared to the previous example `plot_data()` was adopted to also plot `x_ref`.

**Exercise:** Modify the parameter  $l$  in the control function, e.g. to 80 % of its real value. You will see that the feedforward control will drive the car to the wrong final position – as expected, since no information about the real behavior of the car is fed back.

#### 4.3.1. Result

As an example, the transition from  $(0,0,0)$  to  $(5,5,0)$ , starting at  $t = 1s$  ending at  $t = 9s$  is shown in [Figure 3](#). The whole simulation time interval goes from  $t = 0s$  to  $t = 10s$ . The animation shows the behavior of the car in the plane:

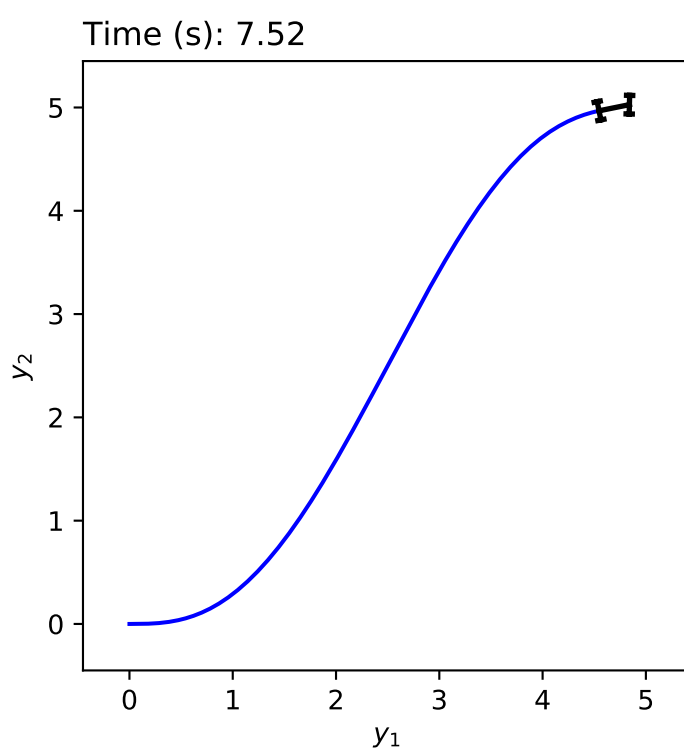


Figure 2: Smooth state transition from  $y^A$  to  $y^B$  in the plane

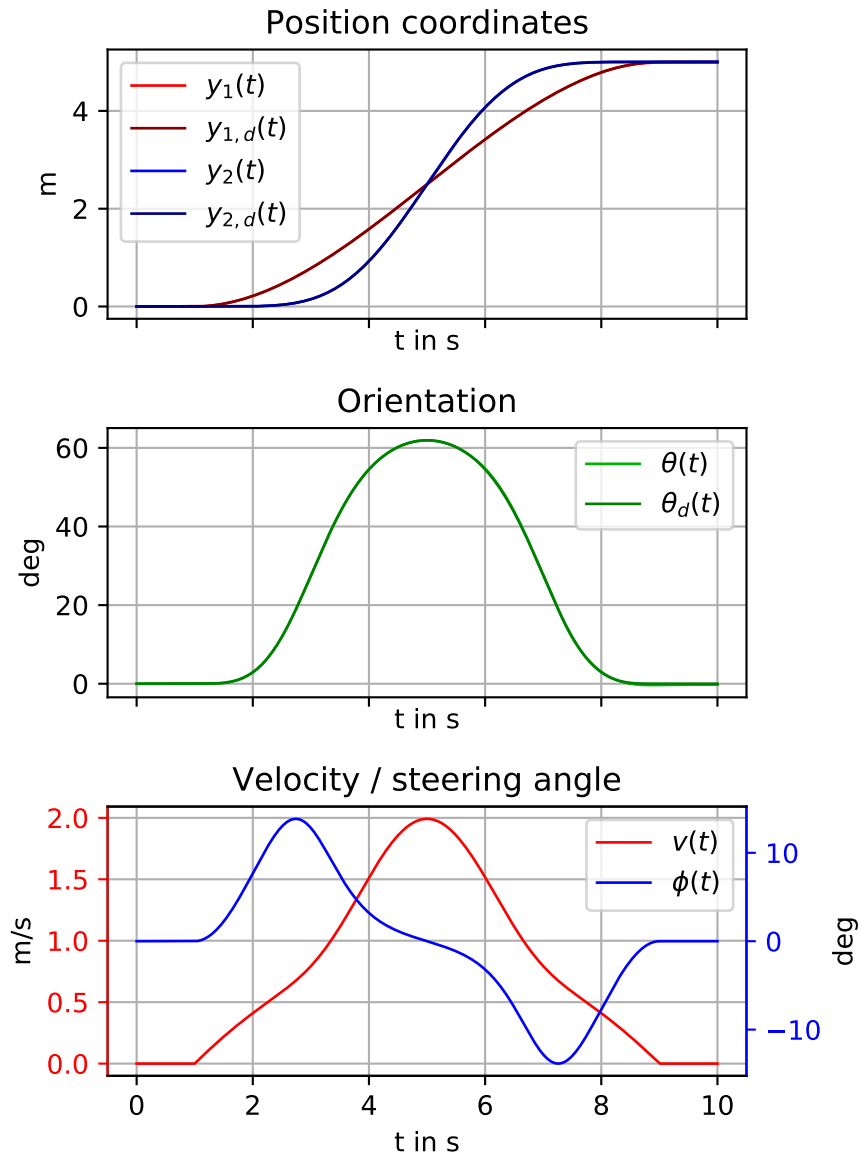


Figure 3: Feedforward control without model errors

## 5. Feedback control design

Source code file: `03_car_feedback_control.py`

In [section 4](#) the controller acts on the exact same system as it was designed for, but in the real world, model errors are inevitable, and a feedforward control is not sufficient. Assuming the length of the car in the controller  $\tilde{l}$  differs from the real car length  $l$  by a factor of 0.9, the feedforward control of [subsubsection 4.3.1](#) will show a bad performance.

### 5.1. Deriving feedback control laws

To account for model errors, a feedback controller has to be designed to fulfill the objective. This is done by a feedback linearization. The linearization is done by introducing new inputs  $w_1$  and  $w_2$ :

$$w_1 = y_1' \quad w_2 = y_2''. \quad (5.1)$$

This leads the linear system shown in [Figure 4](#). The tracking error  $e$  is defined as:

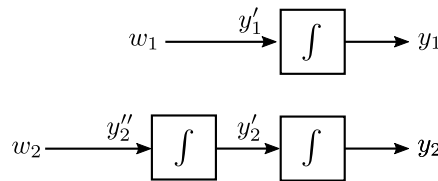


Figure 4: Block diagram of the linearized system

$$e_i = y_i - y_{i,d} \quad i = 1, 2. \quad (5.2)$$

A differential equation for the error term can be defined:

$$0 = e_i'' + k_{1i}e_i' + k_{0i}e_i \quad i = 1, 2 \quad k_{0i}, k_{1i} \in \mathbb{R}^+. \quad (5.3)$$

Substituting [Equation 5.1](#) and [Equation 5.2](#) in [Equation 5.3](#) leads to:

$$w_1 = y_{1,d}' - k_{01}(y_1 - y_{1,d}) \quad (5.4a)$$

$$w_2 = y_{2,d}'' - k_{02}(y_2' - y_{2,d}') - k_{02}(y_2 - y_{2,d}). \quad (5.4b)$$

These equations are substituted into [Equation 4.12](#) to obtain the feedback control law:

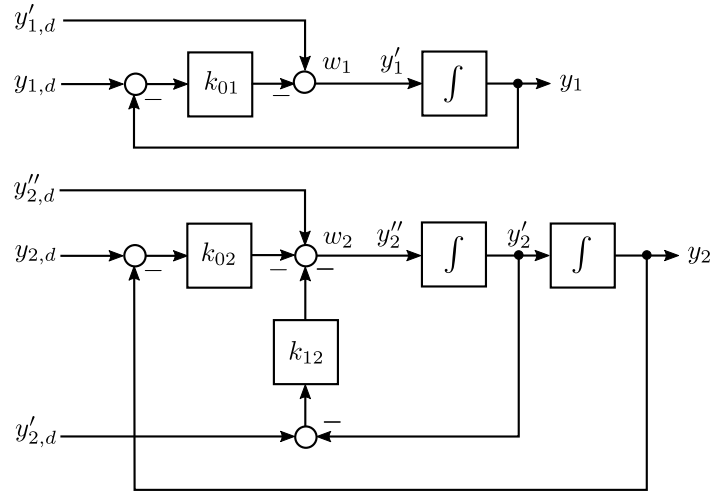


Figure 5: Block diagram of the feedback system

$$v(s) = \dot{s}_d \sqrt{w_1^2 + y_2'^2} \quad (5.5a)$$

$$\varphi(s) = \arctan(l(w_2 w_1 - y_1'' y_2')) \quad (5.5b)$$

where  $\dot{s}_d$  is the desired velocity and  $y_1'' = 0$ . To re-parametrize these control laws in time, the desired trajectories are expressed in  $f$  and  $g$ :

$$\begin{aligned} y_{1,d} &= g(t) & y_{2,d} &= f(g(t)) \\ y'_{1,d} &= \frac{1}{\sqrt{1 + (f'(g(t)))^2}} & y'_{2,d} &= \frac{f'(g(t))}{\sqrt{1 + (f'(g(t)))^2}} \\ \dot{s}_d = v_d(t) &= \dot{g}(t) \sqrt{1 + (f'(g(t)))^2} & y''_{2,d} &= \frac{f''(g(t))}{1 + (f'(g(t)))^2}. \end{aligned}$$

## 5.2. Implementation

To implement the controller, at first the controller parameters are defined:

```

29 @dataclass
30 class ControllerPara:
31     k01: float = 3
32     k02: float = 2
33     k12: float = 10
34     l: float = 0.5 * PhysPara.l

```

The controller parameters have to be hand tuned and must be at least  $> 0$  for the system to be stable. In order to test the performance of the controller with respect to parameter uncertainties the real car length can be scaled, so a disturbed value is used in the control law.

The simulation parameter data class is extended by an attribute defining the real initial state of the car which might be different from what is planned:

```

40 @dataclass
41 class SimPara:
42     t0: float = 0           # start time
43     tf: float = 10          # final time
44     dt: float = 0.04        # step-size
45     tt = np.arange(0, tf + dt, dt) # time vector
46     x0d = [0, 0, 0]         # reference initial state at t0
47     xfd = [5, 5, 0]         # reference final state at tf
48     x0real = [0.25, 0.5, 0.25] # real initial state at t0

```

Within the control law

```

110 def control(x, t, p: Type[ControllerPara]):
111     """Function of the control law
112
113     Args:
114         x (ndarray, int): state vector
115         t (int): time
116         p (ControllerPara): Parameter
117
118     Returns:
119         u (ndarray): control vector
120
121     """

```

the desired trajectories  $f$  and  $g$  are given in the objects `f_traj_gen` and `g_traj_gen` which are evaluated at the given time  $t$  and then further processed:

```

132 # reference trajectories yd, yd', yd''
133 # evaluate the planned trajectories at time t
134 g_t = g_traj_gen.eval(t) # y1 = g(t)
135 f_y1 = f_traj_gen.eval(g_t[0]) # y2 = f(y1) = f(g(t))
136
137 y1d = g_t[0]
138 dy1d = 1/(np.sqrt(1 + f_y1[1] ** 2))
139
140 y2d = f_y1[0]
141 dy2d = f_y1[1]/(np.sqrt(1 + f_y1[1] ** 2))
142 ddy2d = f_y1[2]/(1 + f_y1[1] ** 2)

```

Afterwards  $w_1$  and  $w_2$  are set:

```

146 # stabilizing inputs
147 w1 = dy1d - p.k01 * (y1 - y1d)
148 w2 = ddy2d - p.k12 * (dy2 - dy2d) - p.k02 * (y2 - y2d)

```

In the final step, the control laws are calculated and returned from the function:

```
152 # control laws
153 ds = g_t[1] * np.sqrt(1 + (f_y1[1]) ** 2) # desired velocity
154 u1 = ds*np.sqrt(w1**2+dy2**2)
155 u2 = arctan2(p.l * (w2 * w1), 1)
156
157 return np.array([u1, u2]).T
```

### 5.2.1. Result

The exercise from [subsubsection 4.3.1](#) is repeated, but now using the feedback controller instead. As it can be seen in [Figure 7](#) the control objective of following the planned trajectory succeeded, even with model errors.

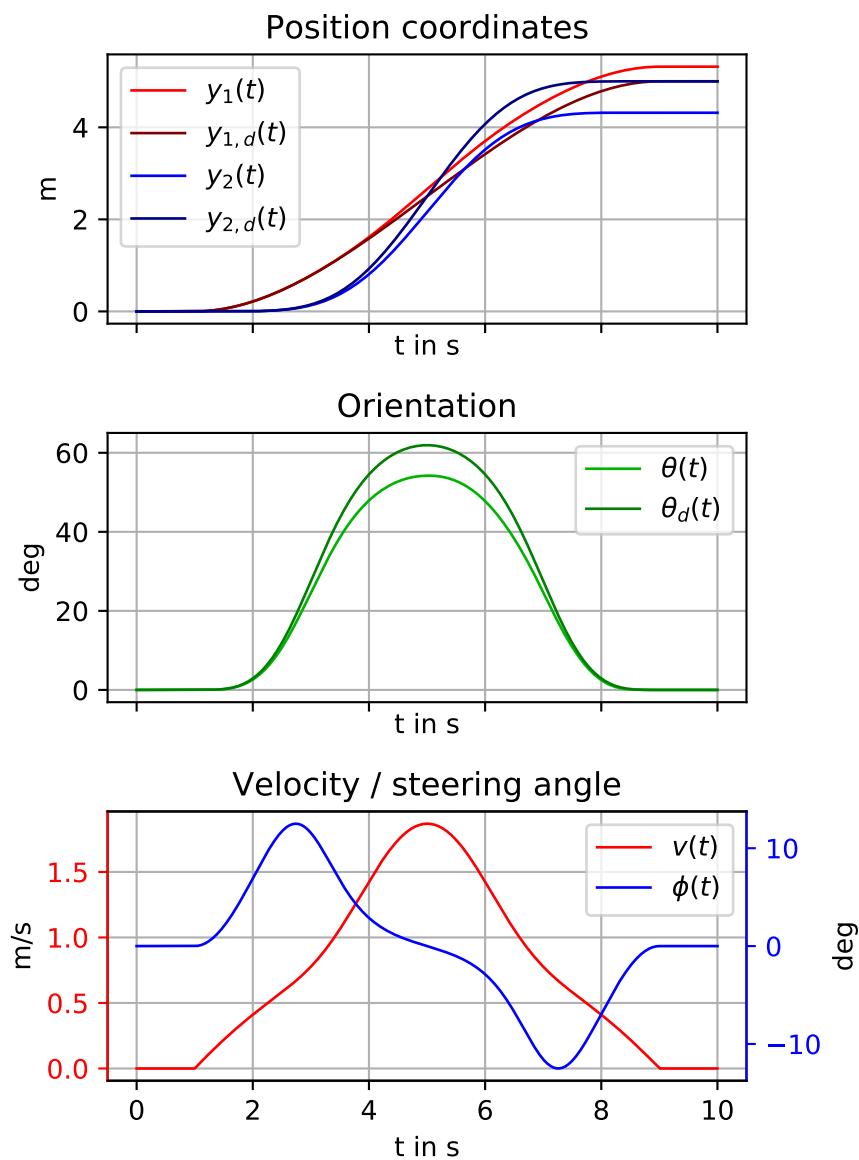


Figure 6: Feedforward control for  $\tilde{l} = 0.9l$ . The car does not end up in the correct position.



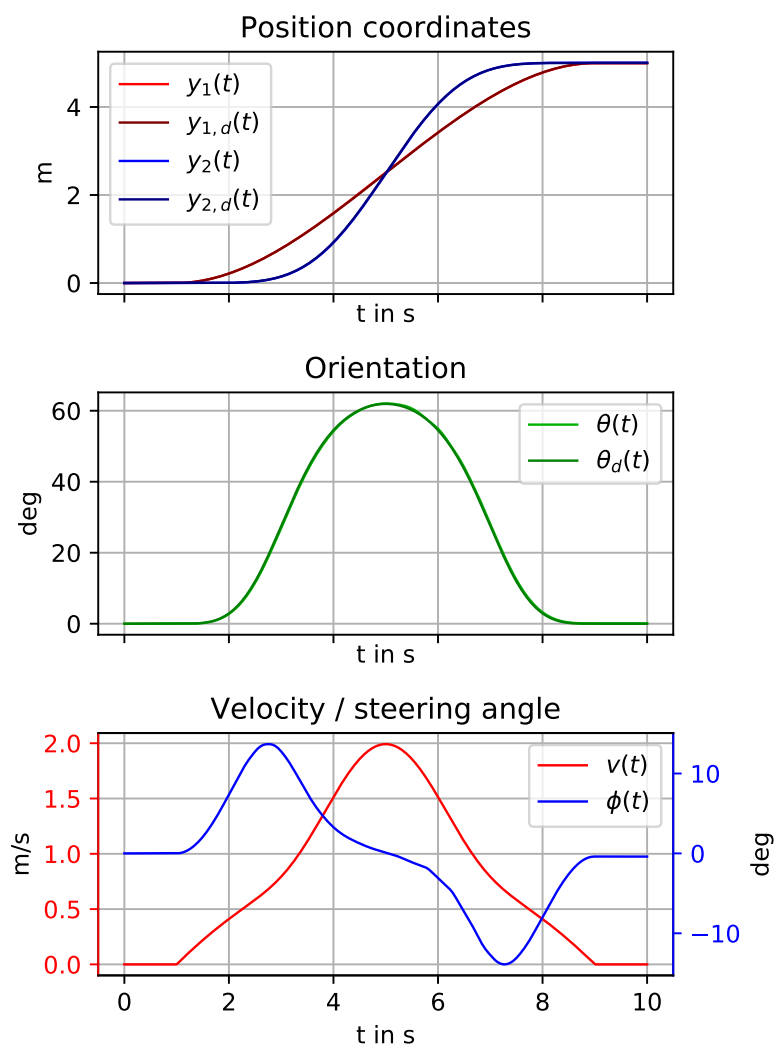


Figure 7: Feedback control for  $\tilde{l} = 0.9l$ . The car successfully reaches the planned final position.

# Appendices

## A. The prototype polynomial

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

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

An approach for the derivative of  $\varphi_\gamma(\tau)$ , which meets the conditions [Equation A.1](#) is:

$$\frac{d\varphi_\gamma(\tau)}{d\tau} = \alpha \frac{\tau^\gamma (1-\tau)^\gamma}{\gamma!} \quad (\text{A.2})$$

Integration leads to:

$$\varphi_\gamma(\tau) = \alpha \int_0^\tau \frac{\tilde{\tau}^\gamma (1-\tilde{\tau})^\gamma}{\gamma!} d\tilde{\tau} \quad (\text{A.3})$$

After  $\gamma$  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  $\alpha$ , the condition  $\varphi_\gamma(1) \stackrel{!}{=} 1$  is used:

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

Finally the prototype function is defined as:

$$\varphi_\gamma(\tau) = \frac{(2\gamma+1)!}{(\gamma!)^2} \sum_{k=0}^{\gamma} \binom{\gamma}{k} \frac{(-1)^k \tau^{\gamma+k+1}}{(\gamma+k+1)} \quad (\text{A.4})$$

and it's  $n$ -th derivative:

$$\varphi_\gamma^{(n)}(\tau) = \frac{(2\gamma+1)!}{(\gamma!)^2} \sum_{k=0}^{\gamma} \left( \binom{\gamma}{k} \frac{(-1)^k \tau^{\gamma+k-n+1}}{(\gamma+k+1)} \prod_{i=1}^n (\gamma+k-i+2) \right) \quad (\text{A.5})$$

In the last step the  $n$ -th derivative of [Equation 2.8](#) ( $n = 1, \dots, \gamma$ ) is derived.

$$y_d^{(n)}(t) = \begin{cases} 0 & \text{if } t < t_0 \\ \frac{(y^B - y^A)}{(t_f - t_0)^n} \varphi_\gamma^{(n)}\left(\frac{t-t_0}{t_f-t_0}\right) & \text{if } t \in [t_0, t_f] \\ 0 & \text{if } t > t_f \end{cases} \quad (\text{A.6})$$

## B. Trajectory generator based on a Gevrey function

It is sometimes necessary, that a planned trajectory is infinitely differentiable<sup>3</sup>. A polynomial approach cannot be used in this case, because an infinite number of parameters is needed to construct such a polynomial. One approach to deal with this problem is to use Gevrey functions instead [2].

$$y_d(t) = \begin{cases} y^A & \text{if } t < t_0 \\ y^A + (y^B - y^A)\varphi_\sigma\left(\frac{t-t_0}{t_f-t_0}\right) & \text{if } t \in [t_0, t_f] \\ y^B & \text{if } t > t_f \end{cases}$$

### B.1. Definition

A function  $\varphi : [0, T] \rightarrow \mathbb{R}$  the derivatives of which are bounded on the interval  $[0, T]$  by

$$\sup_{t \in [0, T]} |\varphi^{(k)}(t)| \leq m \frac{(k!)^\alpha}{\gamma^k}, \text{ with } \alpha, \gamma, m, t \in \mathbb{R}, \quad k \geq 0 \quad (\text{B.1})$$

is called a Gevrey function of order  $\alpha$  on  $[0, T]$ . Here we deal with the Gevrey function

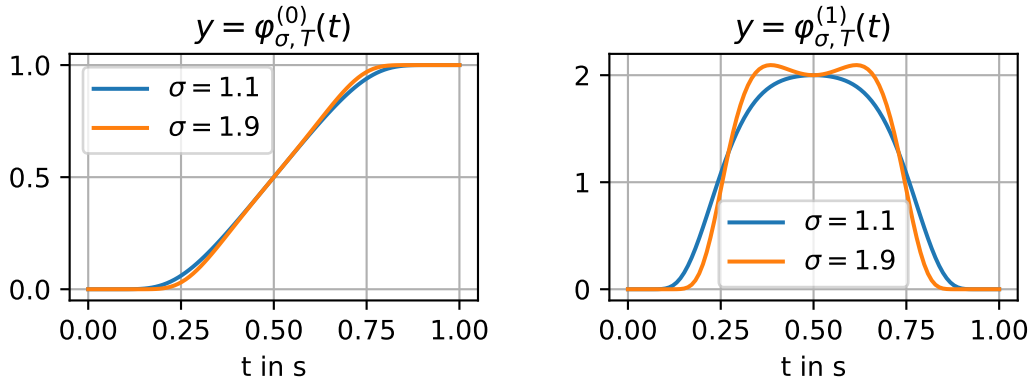


Figure 8: Plot of function  $\varphi_{\sigma,T}$  and its first derivative for different parameters.

$$\varphi_\sigma(\tau) = \frac{1}{2} \left( 1 + \tanh \left( \frac{2(2\tau - 1)}{(4\tau(1 - \tau))^\sigma} \right) \right) \quad (\text{B.2})$$

<sup>3</sup>For example in infinite dimensional systems control.

which is based on the hyperbolic tangent. Some example plots of this function and its derivatives are given in [Figure 8](#). The parameter  $\sigma$  influences the steepness of the transition, for  $\tau = \frac{t}{T}$ ,  $T$  defines the length of the interval where the transition takes place. The order  $\alpha$  is given by  $\alpha = 1 + 1/\sigma$ .

The function is not analytic in  $t = 0$  ( $\tau = 0$ ) and  $t = T$  ( $\tau = 1$ ), all of its derivatives are zero in these points.

## B.2. Efficient calculation of derivatives

**Problem:** Find an algorithm which calculates all derivatives of

$$y := \tanh\left(\frac{2(2\tau - 1)}{(4\tau(1 - \tau))^\sigma}\right) \quad (\text{B.3})$$

efficiently.

[Equation B.3](#) can be written as

$$y = \tanh(a), \quad a = \frac{(4\tau(1 - \tau))^{1-\sigma}}{2(\sigma - 1)}. \quad (\text{B.4})$$

At first, we assume that all derivatives  $a^{(n)}, n \geq 0$  are known, and we show that an iteration formula can be given for  $y^{(n)}$ .

Differentiating [Equation B.4](#) leads to

$$\dot{y} = \ddot{a}(1 - \tanh^2(a)) = \ddot{a}(1 - y^2). \quad (\text{B.5})$$

Introducing the new variable

$$z := (1 - y^2), \quad (\text{B.6})$$

and differentiating [Equation B.5](#)  $(n - 1)$  times gives

$$y^{(n)} = \sum_{k=0}^{n-1} \binom{n-1}{k} a^{(k+2)} z^{(n-1-k)}. \quad (\text{B.7})$$

**Problem:** In [Equation B.7](#) derivatives of  $z$  up to order  $(n - 1)$  are needed. These can be obtained by differentiating [Equation B.6](#)  $(n - 1)$  times:

$$z^{(n-1)} = - \sum_{k=0}^{n-1} \binom{n-1}{k} y^{(k)} y^{(n-1-k)}.$$

Inspecting [Equation B.7](#) one finds that an iteration formula for the derivatives of  $a$  is missing. Using [Equation B.4](#) one gets

$$\dot{a} = \frac{2(2\tau - 1)}{(4\tau(1 - \tau))^{-\sigma}} = \frac{(2\tau - 1)(\sigma - 1)}{\tau(1 - \tau)} a.$$

Multiply this with  $\tau(1 - \tau)$  and differentiate it  $(n - 1)$  times:

$$\sum_{k=0}^{n-1} \binom{n-1}{k} a^{(n-k)} \frac{d^k}{dt^k} (\tau(1 - \tau)) = (\sigma - 1) \sum_{k=0}^{n-1} \binom{n-1}{k} a^{(n-k-1)} \frac{d^k}{dt^k} (2\tau - 1).$$

Solving for  $a^{(n)}$  one gets

$$a^{(n)} = \frac{1}{\tau(1 - \tau)} \left( (\sigma - 1) \sum_{k=0}^{n-1} \binom{n-1}{k} a^{(n-k-1)} \frac{d^k}{dt^k} (2\tau - 1) + \sum_{k=0}^{n-2} \binom{n-1}{k+1} a^{(n-k-1)} \frac{d^k}{dt^k} (2\tau - 1) \right).$$

Note: The sums in the preceding equation have to be evaluated up to the second order only because higher derivatives of  $(2\tau - 1)$  vanish. The result reads

$$a^{(n)} = \frac{1}{\tau(1 - \tau)} \left( (\sigma - 2 + n)(2\tau - 1)a^{(n-1)} + (n - 1)(2\sigma - 4 + n)a^{(n-2)} \right), n \geq 2.$$

### B.3. The *GevreyTrajGen* subclass

The implementation can be found in the **Python source code file: `TrajGen.py`**.

## References

- [1] Carsten Knoll and Robert Heedt. *Python für Ingenieure für Dummies: Mit vielen Programmbeispielen zu Numpy, Matplotlib und mehr*. Weinheim: Wiley-VCH, 2021.
- [2] Joachim Rudolph, Jan Winkler, and Frank Woittennek. *Flatness based control of distributed parameter systems : examples and computer exercises from various technological domains*. Aachen : Shaker, 2003.
- [3] Jan Winkler and Max Pritzkolet. *Python for simulation, animation and control Part 1: Introductory tutorial for the simulation of dynamic systems – Demonstration using the model of a kinematic Vehicle*. Tech. rep. Institute of Control Theory, Faculty of Electrical Engineering and Computer Science, Technische Universität Dresden, Germany, 2021. URL: <https://github.com/TUD-RST/pytutorials/tree/master/01-System-Simulation-ODE>.