Control Theory Tutorial

Car-Like Mobile Robot

***Python* for trajectory planning 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 different trajectory generators in a class hierarchy *Python*,
- Flatness based feedforward control
- Flatness based feedback control.

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

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. Trajectories for smooth point-to-point transitions

In control theory, a common task is to transfer a system from a starting state $y(t_0) = y^A$ at the start time t_0 to a new state $y(t_f) = y^B$ at time t_f. The objective of smooth point-to-point transition is, that the generated trajectory $y_d : t \mapsto y$ meets certain boundary conditions at t_0 and t_f. If y is for example a position coordinate and a simple trapezoidal interpolation in time between the two points y^A and y^B is used , the amount of the acceleration at t_0 and t_f approaches infinity which cannot be fullfilled by any system, due to 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.

***Python* source code file: `Planner.py`**

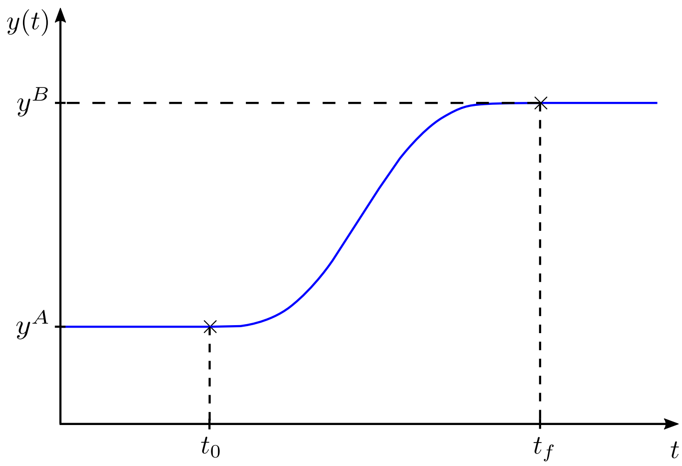


Figure 1: Smooth state transition from y^A to y^B

2.1. Polynomials

A simple way of defining a trajectory between two points in time is a polynomial $y_d(t) = \sum_{i=0}^{2d+1} c_i \frac{t^i}{i!}$ of degree $2d+1$, where $2d+2$ is the number of boundary conditions it has to fulfill. $y_d(t)$ and its successive derivatives up to order d can be written down in matrix form:

$$\underbrace{\begin{pmatrix} y_d(t) \\ \dot{y}_d(t) \\ \vdots \\ y_d^{(d-1)}(t) \\ y_d^{(d)}(t) \end{pmatrix}}_{=:\mathbf{Y}_d(t)\in\mathbb{R}^{(d+1)}} = \underbrace{\begin{pmatrix} 1 & t & \frac{t^2}{2!} & & \dots & \frac{t^{2d+1}}{(2d+1)!} \\ 0 & 1 & t & & \dots & \frac{t^{2d}}{(2d)!} \\ 0 & 0 & 1 & & \dots & \frac{t^{2d-1}}{(2d-1)!} \\ \vdots & & \ddots & \ddots & & \vdots \\ 0 & \dots & \dots\ 0 & 1 & \dots & \frac{t^d}{(d)!} \end{pmatrix}}_{=:\mathbf{T}(t)\in\mathbb{R}^{(d+1)\times(2d+2)}} \underbrace{\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{2d-1} \\ c_{2d} \\ c_{2d+1} \end{pmatrix}}_{=:\mathbf{c}\in\mathbb{R}^{(2d+2)}} \tag{2.1}$$

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

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

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:

$$\mathbf{c} = \begin{bmatrix} \mathbf{T}(t_0) \\ \mathbf{T}(t_f) \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{Y}^A \\ \mathbf{Y}^B \end{bmatrix} \tag{2.2}$$

Because the calculation of the invertible matrix is computationally expensive, in an implementation, it is more efficient to use a linear equation system solver, like `linalg.solve()` from *Numpy*, to solve for $\mathbf{c}$

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

$$\mathbf{Y}_d(t) = \mathbf{T}(t)\mathbf{c} \quad t \in [t_0, t_f] \tag{2.3}$$

The full trajectory can be defined as a piecewise-defined function:

$$y_d(t) = \begin{cases} y^A & \text{if } t < t_0 \\ \sum_{i=0}^{2d+1} c_i \frac{t^i}{i!} & \text{if } t \in [t_0, t_f] \\ y^B & \text{if } t > t_f \end{cases} \tag{2.4}$$

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:

$$y_d(t) = \begin{cases} y^A & \text{if } t < t_0 \\ y^A + (y^B - y^A)\varphi_\gamma\left(\frac{t-t_0}{t_f-t_0}\right) & \text{if } t \in [t_0, t_f] \\ y^B & \text{if } t > T \end{cases} \tag{2.5}$$

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

$$\varphi_\gamma(0) = 0 \quad \varphi_\gamma^{(j)}(0) = 0 \quad j = 1, ..., \gamma \tag{2.6a}$$

$$\varphi_\gamma(1) = 1 \quad \varphi_\gamma^{(j)}(1) = 0 \quad j = 1, ..., \gamma \tag{2.6b}$$

An approach for the derivative of $\varphi_\gamma(\tau)$, which meets the conditions (2.6) is:

$$\frac{\mathrm{d}\varphi_\gamma(\tau)}{\mathrm{d}\tau} = \alpha \frac{\tau^\gamma}{\gamma!} \frac{(1-\tau)^\gamma}{\gamma!} \tag{2.7}$$

Integration leads to:

$$\varphi_\gamma(\tau) = \alpha \int_0^\tau \frac{\tilde{\tau}^\gamma}{\gamma!} \frac{(1-\tilde{\tau})^\gamma}{\gamma!} \mathrm{d}\tilde{\tau} \tag{2.8}$$

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

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

$$\Leftrightarrow \quad \alpha = (2\gamma+1)!$$

Finally the prototype function is defined as:

$$\varphi_\gamma(\tau) = \frac{(2\gamma+1)!}{(\gamma!)^2} \sum_{k=0}^{\gamma} \binom{\gamma}{k} \frac{(-1)^k \tau^{\gamma+k+1}}{(\gamma+k+1)} \tag{2.9}$$

and it's n-th derivative:

$$\varphi_\gamma^{(n)}(\tau) = \frac{(2\gamma+1)!}{(\gamma!)^2} \sum_{k=0}^{\gamma} \left(\binom{\gamma}{k} \frac{(-1)^k \tau^{\gamma+k-n+1}}{(\gamma+k+1)} \prod_{i=1}^{n} (\gamma+k-i+2) \right) \tag{2.10}$$

In the last step the n-th derivative of (2.5) ($n = 1, ..., \gamma$) is derived.

$$y_d^{(n)}(t) = \begin{cases} 0 & \text{if } t < t_0 \\ \frac{(y^B-y^A)}{(t_f-t_0)^n} \varphi_\gamma^{(n)}\left(\frac{t-t_0}{t_f-t_0}\right) & \text{if } t \in [t_0, t_f] \\ 0 & \text{if } t > t_f \end{cases} \tag{2.11}$$

2.3. Implementation in *Python*

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

***Python* source code file:** `Planner.py`

2.3.1. The *Planner* base class

A *Planner* should have the following attributes:

`YA` - vector of y and it's derivatives up to order `d` at start time `t0`

`YB` - vector of y and it's derivatives up to order `d` at final time `tf`

`t0` - start time of the point-to-point transition

`tf` - 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 how the this functionality should be implemented, should be defined in the specific subclass. By using in abstract base class method, we force a subclass of *Planner* to have a method `eval()`.

```python
import numpy as np
import abc # abstract base class
import math
import scipy as sp
from scipy import special

class Planner(object):
    """ Base class for a trajectory planner.

    Attributes:
        YA (int, float, ndarray): start value (size = d+1)
        YB (int, float, ndarray): final value (size = d+1)
        t0 (int, float): start time
        tf (int, float): final time
        d (int): trajectory is smooth up at least to the d-th derivative
    """

    def __init__(self, YA, YB, t0, tf, d):
        self.YA = YA
        self.YB = YB
        self.t0 = t0
        self.tf = tf
        self.d = d

    @abc.abstractmethod
    def eval(self):
        return
```

2.3.2. The *PolynomialPlanner* subclass

To implement the planning algorithm that was developed in subsection 2.1, a new class `PolynomialPlanner` is created that inherits from the previously defined class `Planner`. All the attributes and methods of `Planner` are now also attributes and methods of `PolynomialPlanner`.

```
class PolynomialPlanner(Planner):
    """Planner subclass that uses a polynomial approach for trajectory generation

    Attributes:
        c (ndarray): parameter vector of polynomial

    """
```

To solve for the parameter vector $\mathbf{c}$, the matrix $\mathbf{T}(t)$ from (2.1) is calculated and a method `TMatrix()` is therefore created:

```
    def TMatrix(self, t):
        """Computes the T matrix at time t

        Args:
            t (int, float): time

        Returns:
            T (ndarray): T matrix

        """

        d = self.d
        n = d+1 # first dimension of T
        m = 2*d+2 # second dimension of T

        T = np.zeros([n, m])

        for i in range(0, m):
            T[0, i] = t ** i / math.factorial(i)
        for j in range(1, n):
            T[j, j:m] = T[0, 0:m-j]
        return T
```

Then a method, that solves (2.2) and returns the parameter vector $\mathbf{c}$ is needed:

```
    def coefficients(self):
        """Calculation of the polynomial parameter vector

        Returns:
            c (ndarray): parameter vector of the polynomial

        """
        t0 = self.t0
        tf = self.tf

        Y = np.append(self.YA, self.YB)

        T0 = self.TMatrix(t0)
        Tf = self.TMatrix(tf)

        T = np.append(T0, Tf, axis=0)

        # solve the linear equation system for c
        c = np.linalg.solve(T, Y)
        return c
```

Because the parameters don't change, ones they are calculated, a new attribute `c` is created:

```
    def __init__(self, YA, YB, t0, tf, d):
        super(PolynomialPlanner, self).__init__(YA, YB, t0, tf, d)
        self.c = self.coefficients()
```

Finally a method `eval()` that implements (2.3) is defined:

```
    def eval(self, t):
        """Evaluates the planned trajectory at time t.

        Args:
            t (int, float): time

        Returns:
            Y (ndarray): y and its derivatives at t
        """
        if t < self.t0:
            Y = self.YA
        elif t > self.tf:
            Y = self.YB
        else:
            Y = np.dot(self.TMatrix(t), self.c)
        return Y
```

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

```
    def eval_vec(self, tt):
        """Samples the planned trajectory

        Args:
            tt (ndarray): time vector

        Returns:
            Y (ndarray): y and its derivatives at the sample points

        """
        Y = np.zeros([len(tt), len(self.YA)])
        for i in range(0, len(tt)):
            Y[i] = self.eval(tt[i])
        return Y
```

The polynomial trajectory planner 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:

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

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:

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

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

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

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

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

The calculated parameters can be displayed

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

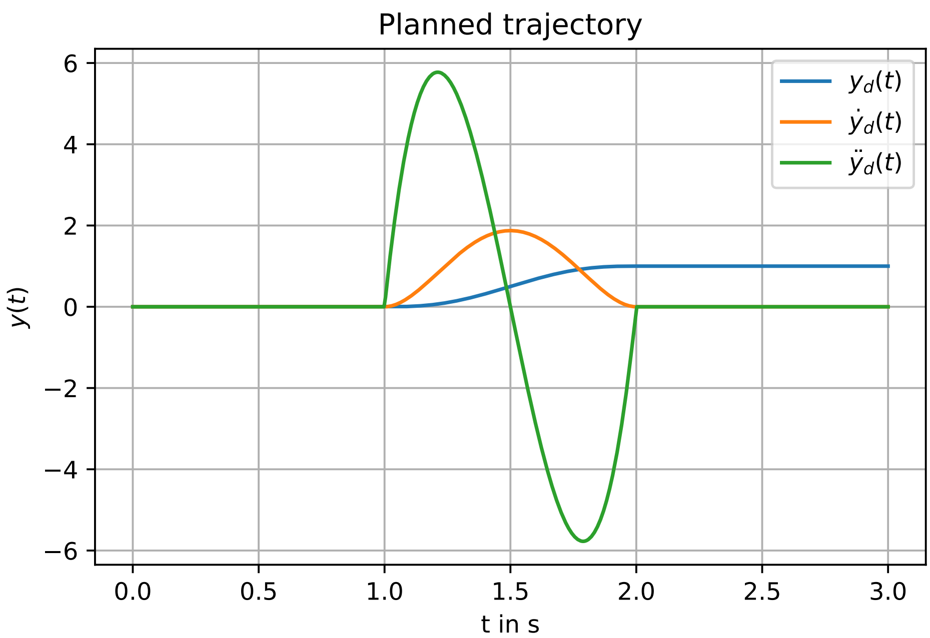
and the generated trajectory at the defined total time interval can be evaluated

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

At last, the results are plotted.

```
Y = yd.eval_vec(tt)

#plot the trajectory
plt.figure(1)
plt.plot(tt, Y)
plt.title('Planned trajectory')
plt.legend([r'$y_d(t)$', r'$\dot{y}_d(t)$',r'$\ddot{y}_d(t)$'])
plt.xlabel(r't in s')
plt.grid(True)
```



2.3.3. The *PrototypePlanner* subclass

***Python* source code file: `Planner.py`**

Implementation can be found in the file.

3. Feedforward control design

***Python* source code file: `02_car_feedforward_control.py`**

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

$$\begin{aligned} \dot{y}_1 &= v\cos(\theta) && (3.1a)\\ \dot{y}_2 &= v\sin(\theta) && (3.1b)\\ \dot{\theta} &= \frac{v}{l}\tan(\varphi). && (3.1c) \end{aligned}$$

3.1. Reparameterization 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:

$$\frac{\mathrm{d}}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\frac{\mathrm{d}s}{\mathrm{d}s} = \frac{\mathrm{d}}{\mathrm{d}s}\frac{\mathrm{d}s}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}s}\dot{s}$$

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

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}s}\dot{s}y_1 &= v\cos(\theta) && (3.2a)\\ \frac{\mathrm{d}}{\mathrm{d}s}\dot{s}y_2 &= v\sin(\theta) && (3.2b)\\ \frac{\mathrm{d}}{\mathrm{d}s}\dot{s}\theta &= \frac{v}{l}\tan(\varphi). && (3.2c) \end{aligned}$$

$$v = |\dot{\boldsymbol{y}}| = \sqrt{\dot{y}_1^2 + \dot{y}_2^2} \tag{3.3}$$

This equation is parameterized in s:[2]

$$v = \sqrt{(\frac{\mathrm{d}}{\mathrm{d}s}\dot{s}y_1)^2 + (\frac{\mathrm{d}}{\mathrm{d}s}\dot{s}y_2)^2} = \dot{s}\sqrt{(y_1')^2 + (y_2')^2} \tag{3.4}$$

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

$$\begin{aligned} 1 &= \left(\frac{dy_1}{ds}\right)^2 + \left(\frac{dy_2}{ds}\right)^2 && (3.5a)\\ \Leftrightarrow \quad 1 &= \sqrt{(y_1')^2 + (y_2')^2} && (3.5b) \end{aligned}$$

[1] https://github.com/TUD-RST/pytutorials/tree/master/01-System-Simulation-ODE

[2] assuming $\dot{s} > 0$

Therfore $v = \dot{s}$. The system paramterized in s is given by:

$$y_1' = \cos(\theta) \tag{3.6a}$$
$$y_2' = \sin(\theta) \tag{3.6b}$$
$$\theta' = \frac{1}{l}\tan(\varphi). \tag{3.6c}$$

3.2. Deriving feedforward control laws

Goal: Drive the car in the y_1-y_2-plane frome 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 $f : \mathbb{R} \to \mathbb{R}$ with $y_2 = f(y_1)$. Note that (y_1, y_2) is a flat output of the system.

Step 1: Calculate the dependency of the remaining system variables θ and φ of the length parameterized system on (y_1, y_2):

$$\tan(\theta) = \frac{y_2'}{y_1'} = \frac{\mathrm{d}y_2}{\mathrm{d}y_1} = f'(y_1) \tag{3.7}$$
$$(1+\tan^2(\theta))\frac{\mathrm{d}\theta}{\mathrm{d}y_1} = f''(y_1)$$
$$\Leftrightarrow \quad \frac{\mathrm{d}\theta}{\mathrm{d}y_1} = \frac{f''(y_1)}{1+(f'(y_1))^2} = \frac{\theta'}{y_1'}$$

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

$$\Leftrightarrow \quad \theta' = \frac{f''(y_1)}{(1+(f'(y_1))^2)^{3/2}} \tag{3.8}$$
$$\tan(\varphi) = l\theta' = \frac{lf''(y_1)}{(1+(f'(y_1))^2)^{3/2}} \tag{3.9}$$

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:

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

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

$$v = \sqrt{\dot{y}_1^2 + \dot{y}_2^2} = \dot{y}_1\sqrt{1 + (f'(y_1))^2} = \dot{g}(t)\sqrt{1 + (f'(g(t)))^2} \tag{3.10}$$

Hence, the overall, time parameterized feedforward control reads:

$$v(t) = \dot{g}(t)\sqrt{1 + (f'(g(t)))^2} \tag{3.11a}$$

$$\varphi(t) = \arctan\left(\frac{l f''(g(t))}{(1 + (f'(g(t)))^2)^{3/2}}\right) \tag{3.11b}$$

Or expressed in s:

$$v(s) = \dot{s}\sqrt{y_2'^2 + y_1'^2} \tag{3.12a}$$

$$\varphi(s) = \arctan\left(l\frac{y_2''y_1' - y_1''y_2'}{(y_1'^2 + y_2'^2)^{\frac{3}{2}}}\right) = \arctan\left(l(y_2''y_1' - y_1''y_2')\right) \tag{3.12b}$$

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.

3.3. Implementation

For the implementation of the controller, a the polynomial planner from 2.3.2 is used. At first all necessary simulation parameters are defined:

```
sim_para = Parameters()  # instance of class Parameters
sim_para.t0 = 0          # start time
sim_para.tf = 10         # final time
sim_para.dt = 0.04       # step-size
sim_para.tt = np.arange(sim_para.t0, sim_para.tf + sim_para.dt, sim_para.dt) # time vector
sim_para.x0 = [0, 0, 0]  # inital state at t0
sim_para.xf = [5, 5, 0]  # final state at tf
```

Initialization of the trajectory planners:

```
# Trajectory parameters
traj_para = Parameters() # instance of class Parameters
traj_para.t0 = sim_para.t0 + 1 # start time of transition
traj_para.tf = sim_para.tf - 1 # final time of transition

# boundary conditions for y1
traj_para.Y1A = np.array([sim_para.x0[0], 0])
traj_para.Y1B = np.array([sim_para.xf[0], 0])

# boundary conditions for y2
traj_para.Y2A = np.array([sim_para.x0[1], tan(sim_para.x0[2]), 0])
traj_para.Y2B = np.array([sim_para.xf[1], tan(sim_para.xf[2]), 0])

# ininitialize the planners
traj_para.f = PolynomialPlanner(traj_para.Y2A, traj_para.Y2B, traj_para.Y1A[0], traj_para.Y1B
    [0], 2)
traj_para.g = PolynomialPlanner(traj_para.Y1A, traj_para.Y1B, traj_para.t0, traj_para.tf, 1)
```

Implementation of the the control law:

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

    Args:
        x (ndarray, int): state vector
        t (int): time
        p (object): parameter container class

    Returns:
        u (ndarry): control vector

    """

    # get planners from traj_para
    f = traj_para.f
    g = traj_para.g

    # evaluate the planned trajectories at time t
    g_t = g.eval(t) # y1 = g(t)
    f_y1 = f.eval(g_t[0]) # y2 = f(y1) = f(g(t))

    # setting control laws
    u1 = g_t[1]*np.sqrt(1 + f_y1[1]**2)
    u2 = arctan2(p.l*f_y1[2], (1 + f_y1[1]**2)**(3/2))

    return np.array([u1, u2]).T
```

Running the simulation:

```python
sol = sci.solve_ivp(lambda t, x: ode(x, t, para), (sim_para.t0, sim_para.tf), sim_para.x0,
    method='RK45',t_eval=sim_para.tt)
```

The results can be extracted by:

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

To get the control vector can be done by evaluating `control()` with the simulated trajectory and recalculate the values that were applied to the system.

```python
u_traj = np.zeros([len(sim_para.tt),2])
for i in range(0, len(sim_para.tt)):
    u_traj[i] = control(x_traj[i], sim_para.tt[i], para)
```

Because `control()` works only for scalar time values, this has to be done in a `for`-loop. Plotting the simulation results and the reference trajectories:

```python
y1D = traj_para.g.eval_vec(sim_para.tt)
y2D = traj_para.f.eval_vec(y1D[:,0])

x_ref = np.zeros_like(x_traj)
x_ref[:,0] = y1D[:,0]
x_ref[:,1] = y2D[:,0]
x_ref[:,2] = arctan(y2D[:,1])

plot_data(x_traj, x_ref, u_traj, sim_para.tt, 12, 16, save=True)
plt.show()
```

`plot_data()` was adopted to also plot `x_ref`. The changes can be found in the corrosponding file.

3.3.1. 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 behaviour of the car in the plane:

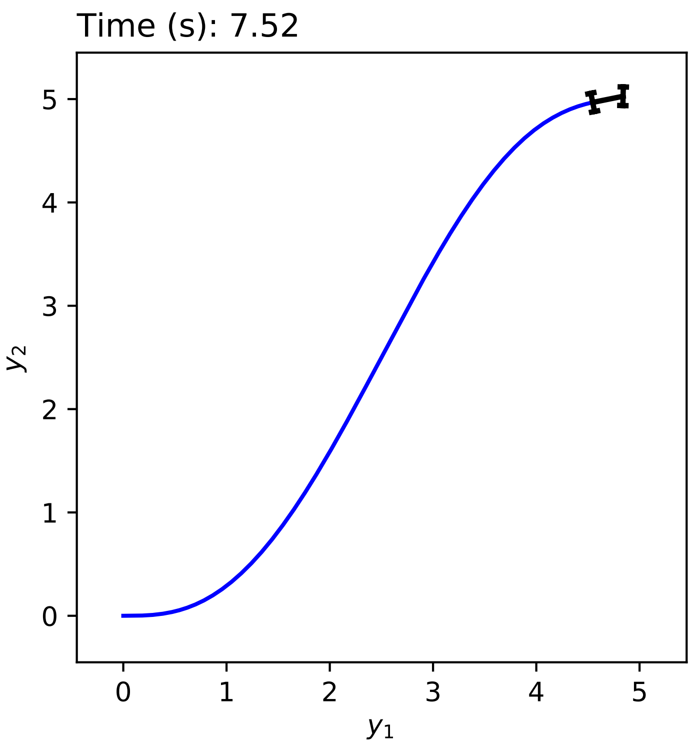


Figure 2: Smooth state transition from y^A to y^B in the plane

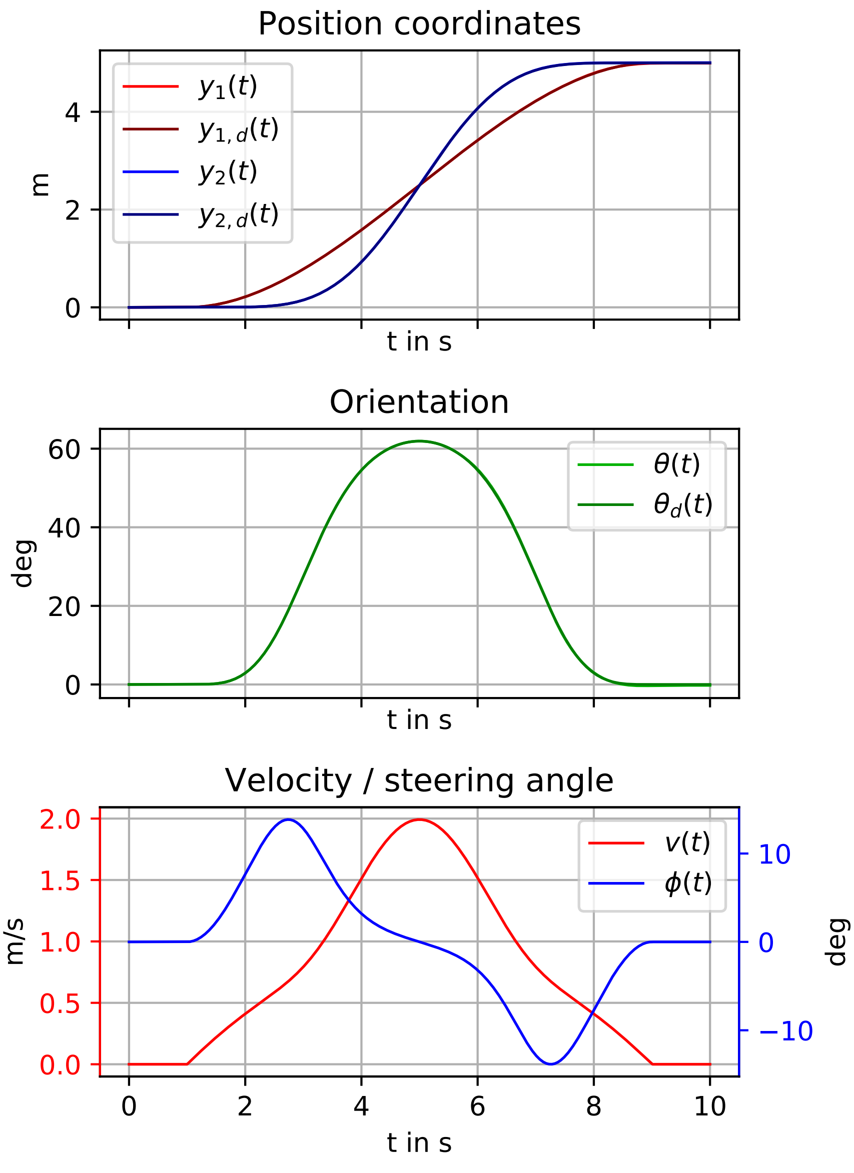


Figure 3: Feedforward control without model errors

4. Feedback control design

***Python* source code file: `03_car_feedback_control.py`**

In section 3 the controller acts on the exact same system as it was designed for, but in the real world, model errors are inevidable 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 feedfoward control of 3.3.1 shows a bad performance, as can be seen in Figure 6.

4.1. 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:

$$w_1 = y_1' \qquad w_2 = y_2'' \tag{4.1}$$

This leads the linear system shown in Figure 4. The tracking error e is defined as:

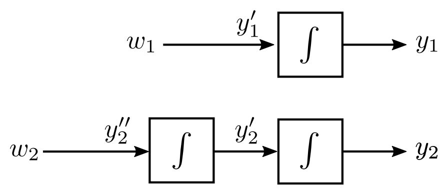


Figure 4: Block diagram of the linearized system

$$e_i = y_i - y_{i,d} \qquad i = 1,2 \tag{4.2}$$

A differential equation for the error term can be defined:

$$0 = e_i'' + k_{1i}e_i' + k_{0i}e_i \qquad i = 1,2 \quad k_{0i}, k_{1i} \in \mathbb{R}^+ \tag{4.3}$$

Substituting (4.1) and (4.2) in (4.3) leads to:

$$w_1 = y_{1,d}' - k_{01}(y_1 - y_{1,d}) \tag{4.4a}$$
$$w_2 = y_{2,d}'' - k_{02}(y_2' - y_{2,d}') - k_{02}(y_2 - y_{2,d}) \tag{4.4b}$$

These equations are substituted into (3.12) to obtain the feedback control law:

$$v(s) = \dot{s}_d \sqrt{w_1^2 + y_2'^2} \tag{4.5a}$$
$$\varphi(s) = \arctan\left(l(w_2 w_1 - y_1'' y_2')\right) \tag{4.5b}$$

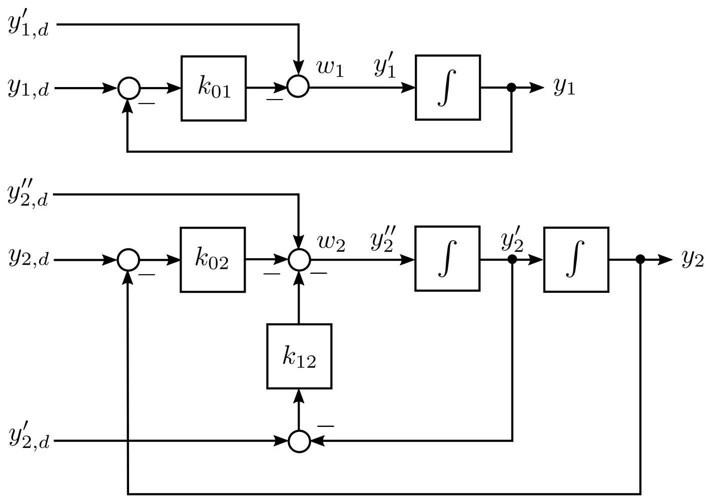


Figure 5: Block diagram of the feedback system

where $\dot{s}_d$ is the desired velocity and $y''_1 = 0$. To reparametrize these control laws in time, the desired trajectories are expressed in f and g:

$$y_{1,d} = g(t) \qquad y_{2,d} = f(g(t))$$

$$y'_{1,d} = \frac{1}{\sqrt{1 + (f'(g(t)))^2}} \qquad y'_{2,d} = \frac{f'(g(t))}{\sqrt{1 + (f'(g(t)))^2}}$$

$$\dot{s}_d = v_d(t) = \dot{g}(t)\sqrt{1 + (f'(g(t)))^2} \qquad y''_{2,d} = \frac{f''(g(t))}{1 + (f'(g(t)))^2}$$

4.2. Implementation

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

```
    # controller parameters
    k01 = 1
    k02 = 1
    k12 = 5
```

The controller parameters have to be hand tuned and must be > 0 for the system to be stable.

Then the desired trajectories are expressed in the planner trajectories f and g:

```
    # reference trajectories yd, yd', yd''
    y1d = g_t[0]
    dy1d = 1/(np.sqrt(1 + f_y1[1] ** 2))

    y2d = f_y1[0]
    dy2d = f_y1[1]/(np.sqrt(1 + f_y1[1] ** 2))
    ddy2d = f_y1[2]/(1 + f_y1[1] ** 2)
```

Afterwards w_1 and w_2 are set:

```
    # stabilizing inputs
    w1 = dy1d - k01 * (y1 - y1d)
    w2 = ddy2d - k12 * (dy2 - dy2d) - k02 * (y2 - y2d)
```

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

```
    # control laws
    ds = g_t[1] * np.sqrt(1 + (f_y1[1]) ** 2) #desired velocity
    u1 = ds*np.sqrt(w1**2+dy2**2)
    u2 = arctan2(0.9*p.l * (w2 * w1), 1)

    return np.array([u1, u2]).T
```

4.2.1. Result

The experiment from 3.3.1 is repeated with the same model error as in Figure 6, 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.

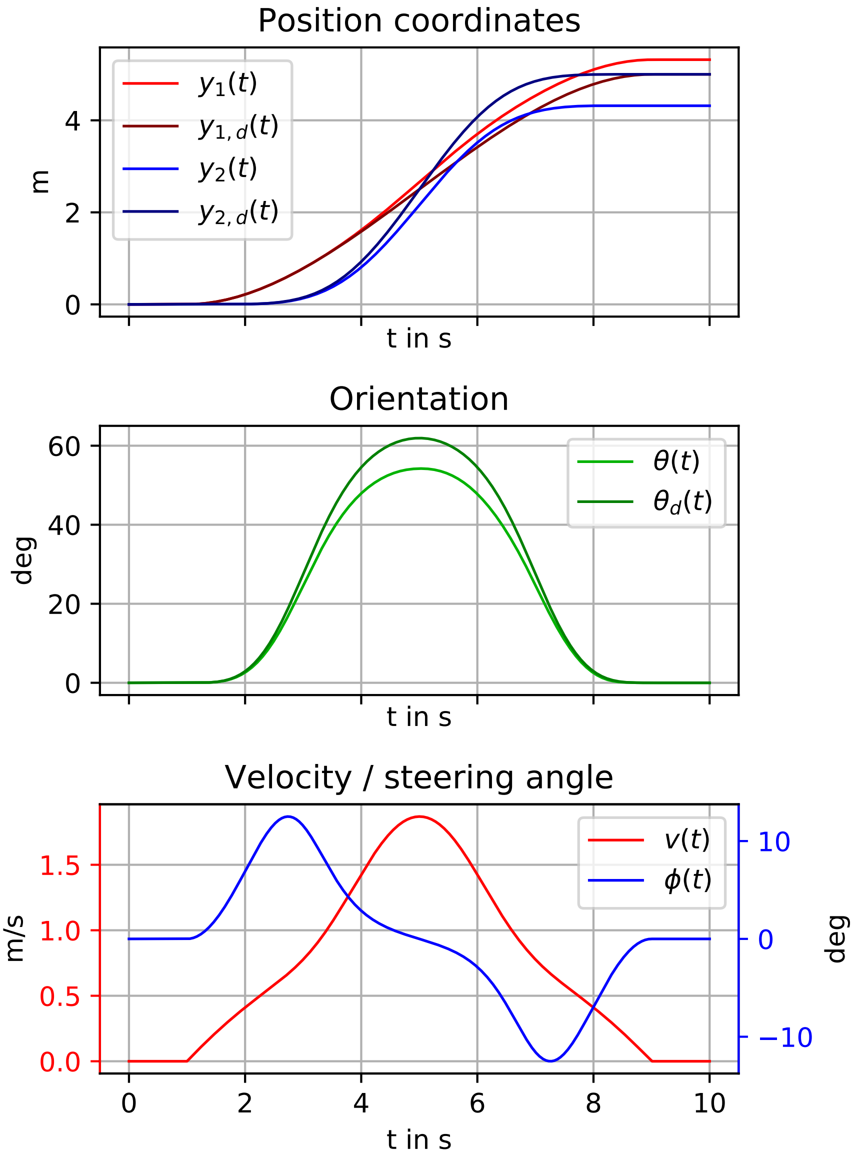


Figure 6: Feedfoward control for $\tilde{l} = 0.9l$

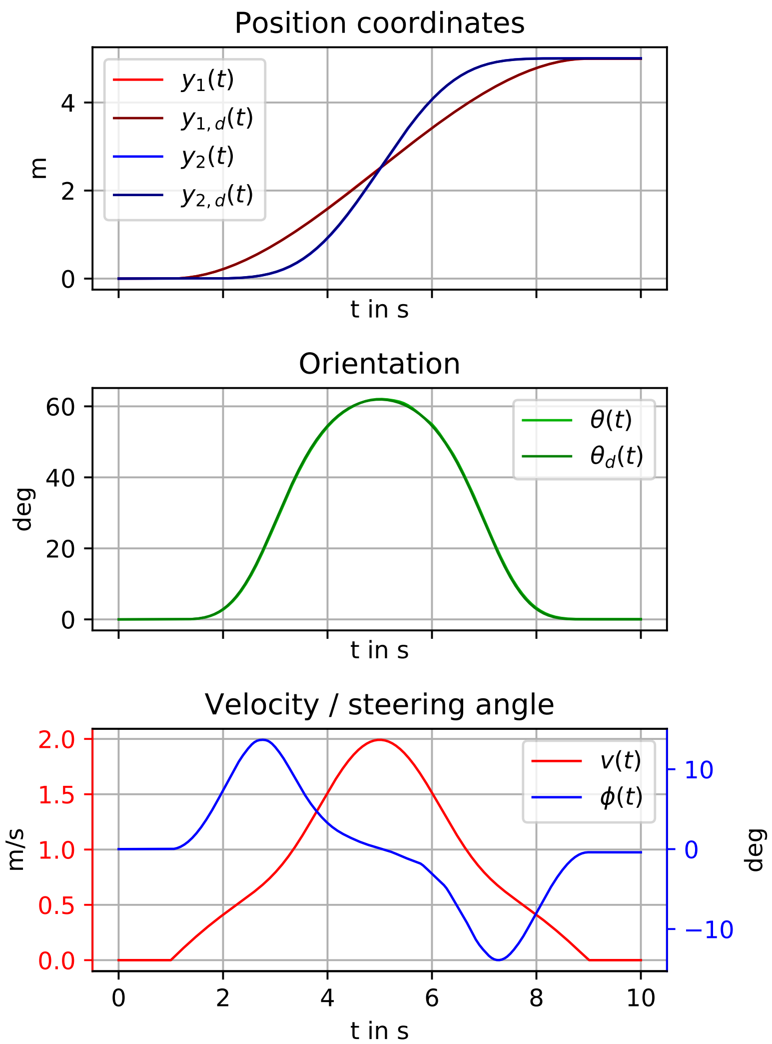


Figure 7: Feedback control for $\tilde{l} = 0.9l$

Appendices

A. Gevrey function planner

Reference:

Rud03

It is sometimes necessary, that a planned trajectory is infinitely differentiable [3]. A polynomial approach can't 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-funtions instead.

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

A.1. Definition

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

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

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

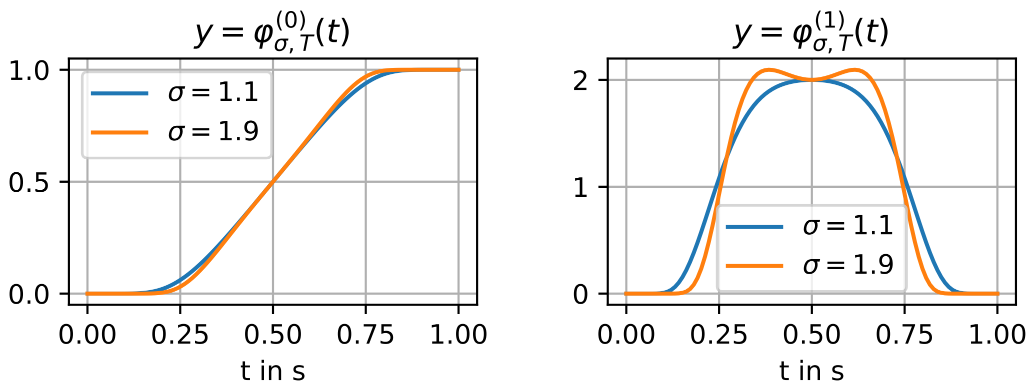


Figure 8: Plot of function $\varphi_{\sigma,T}$ and its first derivative for different parameters.

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

[3]For example in infinite dimensional systems control.

which is based on the tangens hyperbolicus. Some example plots of this function and its derivatives are given in Fig. 8. The parameter σ 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 α 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.

A.2. Efficient calculation of derivatives

Problem: Find an algorithm which calculates all derivatives of

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

in an efficient way.

Eq. (A.3) can be written as

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

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 eq. (A.4) leads to

$$\dot{y} = \ddot{a}(1 - \tanh^2(\dot{a})) = \ddot{a}(1 - y^2). \tag{A.5}$$

Introducing the new variable

$$z := (1 - y^2) \tag{A.6}$$

and differentiating (A.5) $(n-1)$ times gives

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

Problem: In (A.7) derivatives of z up to order $(n-1)$ are needed. These can be obtained by differentiating (A.6) $(n-1)$ times:

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

Inspecting (A.7) one finds that an iteration formula for the derivatives of a is missing. Using (A.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 preceeding equation have to be evaluated up to the second order only because higher derivatives of $(2\tau - 1)$ vanish. The result reads

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

A.3. The *GevreyPlanner* subclass

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