

Python for simulation, animation and control

Part 1: Introductory tutorial for the simulation of dynamic systems

Demonstration using the model of a kinematic Vehicle

Max Pritzkoleit* Jan Winkler*

January 7, 2022

Contents

1	Introduction	2
2	Kinematic model of a vehicle	2
3	Libraries and Packages	3
4	Storing parameters	4
5	Simulation with SciPy's integrate package	5
5.1	Implementation of the model	5
5.2	Solution of the initial value problem using SciPy	6
6	Plotting using Matplotlib	7
7	Animation using Matplotlib	8
8	Time-Events	12

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

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 represented by nonlinear [ordinary differential equations \(ODEs\)](#). The following topics are covered:

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

Source code file: `01_car_example_plotting.py`

Later the simulation is extended by a visualization of the moving vehicle and some advanced methods for numerical integration of [ODEs](#).

Please refer to the [Python List-Dictionary-Tuple tutorial](#)¹ and the [NumPy Array tutorial](#)² 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](#)³. Additionally, the book [1] is recommended (in German only).

2 Kinematic model of a vehicle

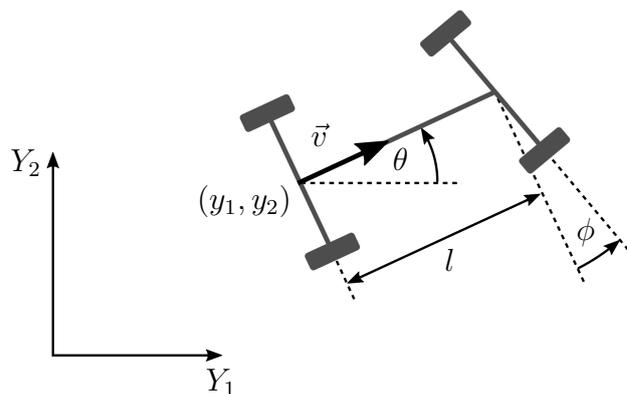


Figure 1: Car-like mobile robot

¹<http://cs231n.github.io/python-numpy-tutorial/#python-containers>

²<http://cs231n.github.io/python-numpy-tutorial/#numpy>

³<https://www.tutorialspoint.com/python/index.htm>

Given is a nonlinear kinematic model of a car-like mobile robot, cf. [Figure 1](#), with the following system variables: position (y_1, y_2) and orientation θ in the plane, the steering angle ϕ and the vehicle's lateral velocity $v = |\mathbf{v}|$:

$$\dot{y}_1(t) = v \cos(\theta(t)) \quad y_1(0) = y_{10} \quad (1a)$$

$$\dot{y}_2(t) = v \sin(\theta(t)) \quad y_2(0) = y_{20} \quad (1b)$$

$$\dot{\theta}(t) = \frac{1}{l} v(t) \tan(\phi(t)) \quad \theta(0) = \theta_0. \quad (1c)$$

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

To simulate this system (1) of first order [ODEs](#), one has to introduce a state vector $\mathbf{x} = (x_1, x_2, x_3)^T$ and a control vector $\mathbf{u} = (u_1, u_2)^T$ as follows:

$$x_1 := y_1 \quad u_1 := v \quad (2a)$$

$$x_2 := y_2 \quad u_2 := \phi. \quad (2b)$$

$$x_3 := \theta \quad (2c)$$

Now, the [initial value problem \(IVP\)](#) (1) can be expressed in the general form $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t))$ with $\mathbf{x}(0) = \mathbf{x}_0$:

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

Usually, this explicit formulation of the [IVP](#) is the basis for implementing a system simulation by numerical integration. In the following a simulation using Python is set up which shows the dynamic behavior of the vehicle when driving with a continuously decreasing velocity under a constant steering angle. Of course, in this simple case, the result is known in advance: The vehicle will drive on a circle until it stops for $v = 0$. In the following the Python-script for simulating the system will be derived step by step.

3 Libraries and Packages

Neither the numerical solution of the [IVP](#) (1) nor the presentation of the results can be done comfortably in pure Python. To overcome this limitation separate packages for

array handling, numerical integration, and plotting are provided. Under Python such packages should be imported at the top of the executed script⁴.

The most relevant packages for the simulation of control systems are

- [NumPy](#) for array handling and mathematical functions,
- [SciPy](#) for numerical integration of [ODEs](#) (and a lot of other stuff, of course),
- [Matplotlib](#) for plotting.

It is good practice to connect the imported packages with a namespace, so it can be easily seen in the code which function comes from where. For example, in case of NumPy the following statement imports the package NumPy and ensures that every function from NumPy is addressed by the prefix `np.:`

```
2 import numpy as np
```

For frequently used functions like `cos(...)`, `sin(...)`, and `tan(...)` it is annoying to prefix them like `np.cos(...)` each time. To avoid this one can directly import them as

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

To solve the [IVP \(2\)](#) the library SciPy with its sub-package *integrate* offers different solvers:

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

For plotting the output of the simulation results the library Matplotlib with its sub-package *pyplot* introduces a user experience similar to *MATLAB* into Python:

```
5 import matplotlib.pyplot as plt
```

4 Storing parameters

In simulations usually a lot of parameters describing the system or the simulation setup have to be handled. It is a good idea to store these parameters as attributes in a structure, so it is not necessary to deal with several individual variables holding the values of the parameters. In Python, such a structure can be a so-called dataclass class, a Python class with the decorator `@dataclass`. Data classes are available after putting

```
7 from dataclasses import dataclass
```

⁴It is also possible to import them elsewhere in the code but following the official style guide PEP8 “imports are always put at the top of the file, just after any comments and docstrings, and before globals and constants”.

at the beginning of the file. Then the structures holding the required data can be defined as follows (with type annotations for the members):

```

11 # Physical parameter
12 @dataclass
13 class Para:
14     l: float = 0.3 # define car length
15     w: float = l*0.3 # define car width

```

Similarly this can be done with the simulation parameters:

```

20 # Simulation parameter
21 @dataclass
22 class SimPara:
23     t0: float = 0 # start time
24     tf: float = 10 # final time
25     dt: float = 0.04 # step-size

```

Alternatively, one could use the datatype *dictionary*. However, the resulting keyword notation (e.g., `Para["l"]` instead of `Para.l`) in the code using the parameters is quite annoying. Furthermore, an IDE providing static code analysis or IntelliSense features might warn you about missing attributes if you use data classes.

5 Simulation with SciPy's integrate package

5.1 Implementation of the model

In order to simulate the IVP (3) using the numerical integrators offered by SciPy's integrate package a function returning the right-hand side of (3) evaluated for given values of \mathbf{x} , \mathbf{u} and the parameters has to implemented:

```

30 def ode(t, x, p: Type[Para]):
31     """Function of the robots kinematics
32
33     Args:
34         x      : state
35         t      : time
36         p(object): parameter container class
37
38     Returns:
39         dxdt: state derivative
40     """
41     x1, x2, x3 = x # state vector
42     u1, u2 = control(t) # control vector
43
44     # dxdt = f(x, u):
45     dxdt = np.array([u1 * cos(x3),
46                    u1 * sin(x3),
47                    1 / p.l * u1 * tan(u2)])
48
49     # return state derivative
50     return dxdt

```

Note that we added a type annotation here for the parameter argument `p`. This is not necessary for the code to execute. It just tells the static code analysis/ IntelliSense mechanism of your IDE which type the argument has, so it can correctly identify any errors in the code. This approach is recommended in order to speed up code developing and to avoid errors. You need to import the [typing package](#) for this to work:

```
6 from typing import Type
```

The `ode` functions calls the control law function `control` calculating values for v and ϕ depending on the time t . As a first heuristic approach, the vehicle is driven with a constant steering angle while continuously reducing the speed from 0.5 m s^{-1} to zero. Later, an arbitrary function, for example a feedback law $\mathbf{u} = k(\mathbf{x})$, can be implemented.

```
55 def control(t):
56     """Function of the control law
57
58     Args:
59         t: time
60
61     Returns:
62         u: control vector
63
64     """
65     u1 = np.maximum(0, 1.0 - 0.1 * t)
66     u2 = np.full(u1.shape, 0.25)
67     return np.array([u1, u2]).T
```

It is important that the function needs to handle also time arrays as input in order to calculate the control for a bunch of values at once (not during the numerical integration but later for analysis purposes). That's why NumPy's array capable [maximum function](#) is used here with appropriately adjusted shape of `u2`.

Furthermore, attention has to be paid how the two functions above are documented. The text within the `"""` is called *docstring*. Tools like [Sphinx](#) are able to convert these into well formatted documentations. Docstrings can be written in several ways. Here the so-called [Google Style](#) is used.

5.2 Solution of the initial value problem using SciPy

Having implemented the system dynamics the numerical integration of system (3) can be performed. At first, a vector `tt` specifying the time values at which one would like to obtain the computed values of \mathbf{x} has to be defined. Then the initial vector \mathbf{x}_0 is defined and the `solve_ivp` function of the SciPy *integrate* package is called to perform the simulation. The function `solve_ivp` takes a function of the type `func(t, x)` calculating the value of the right-hand side of (3). Further parameters are not allowed. In order to be able to use the previously defined ode-function `ode(t, x, p)` which additionally takes the parameter structure `p`, a so-called *lambda-function* is used. The solver is called as follows:

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

This way the `ode` function is encapsulated in an anonymous function, that has just `(t, x)` as arguments (as required by `solve_ivp`) but evaluates as `ode(t, x, para)`⁵. Additionally, the following arguments are passed to `solve_ivp`: A tuple `(t0, tf)` which defines the simulation interval and the initial value `x0`. Furthermore, the optional arguments' `method` (the integration method used, default: Runge-Kutta 45), and `t_eval` (defining the values at which the solution should be sampled) can be passed.

The return value `sol` is a `Bunch` object. To extract the simulated state trajectory, one has to execute:

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

Finally, the control input values are calculated from the obtained trajectory of `x` (the values for `u` in the `ode` function cannot be directly saved because the function is also repeatedly called between the specified time steps by the solver).

```
146 # time vector
147 tt = np.arange(SimPara.t0, SimPara.tf + SimPara.dt, SimPara.dt)
148
149 # initial state
150 x0 = [0, 0, 0]
151
152 # simulation
153 sol = sci.solve_ivp(lambda t, x: ode(t, x, Para), (SimPara.t0, SimPara.tf), x0, t_eval=tt)
154 x_traj = sol.y.T
155 u_traj = control(tt)
```

Note that the interval specified by `np.arange` is open on the right-hand side. Hence, `dt` is added to obtain also values for `x` at `tf`.

6 Plotting using Matplotlib

Usually one wants to publish the results with descriptive illustrations. For this purpose the required plotting instructions are encapsulated in a function. This way, one can easily modify parameters of the plot, for example figure width, or if the figure should be saved on the hard drive.

```
72 def plot_data(x, u, t, fig_width, fig_height, save=False):
73     """Plotting function of simulated state and actions
74
75     Args:
76         x(ndarray) : state-vector trajectory
77         u(ndarray) : control vector trajectory
78         t(ndarray) : time vector
79         fig_width  : figure width in cm
```

⁵The lambda function corresponds to `@` in *MATLAB*

```

80     fig_height : figure height in cm
81     save (bool): save figure (default: False)
82     Returns: None
83
84     """
85     # creating a figure with 3 subplots, that share the x-axis
86     fig1, (ax1, ax2, ax3) = plt.subplots(3)
87
88     # set figure size to desired values
89     fig1.set_size_inches(fig_width / 2.54, fig_height / 2.54)
90
91     # plot y_1 in subplot 1
92     ax1.plot(t, x[:, 0], label='$y_1(t)$', lw=1, color='r')
93
94     # plot y_2 in subplot 1
95     ax1.plot(t, x[:, 1], label='$y_2(t)$', lw=1, color='b')
96
97     # plot theta in subplot 2
98     ax2.plot(t, np.rad2deg(x[:, 2]), label=r'$\theta(t)$', lw=1, color='g')
99
100    # plot control in subplot 3, left axis red, right blue
101    ax3.plot(t, np.rad2deg(u[:, 0]), label=r'$v(t)$', lw=1, color='r')
102    ax3.tick_params(axis='y', colors='r')
103    ax33 = ax3.twinx()
104    ax33.plot(t, np.rad2deg(u[:, 1]), label=r'$\phi(t)$', lw=1, color='b')
105    ax33.spines["left"].set_color('r')
106    ax33.spines["right"].set_color('b')
107    ax33.tick_params(axis='y', colors='b')
108
109    # Grids
110    ax1.grid(True)
111    ax2.grid(True)
112    ax3.grid(True)
113
114    # set the labels on the x and y axis and the titles
115    ax1.set_title('Position coordinates')
116    ax1.set_ylabel(r'm')
117    ax1.set_xlabel(r't in s')
118    ax2.set_title('Orientation')
119    ax2.set_ylabel(r'deg')
120    ax2.set_xlabel(r't in s')
121    ax3.set_title('Velocity / steering angle')
122    ax3.set_ylabel(r'm/s')
123    ax33.set_ylabel(r'deg')
124    ax3.set_xlabel(r't in s')
125
126    # put a legend in the plot
127    ax1.legend()
128    ax2.legend()
129    ax3.legend()
130    li3, lab3 = ax3.get_legend_handles_labels()
131    li33, lab33 = ax33.get_legend_handles_labels()
132    ax3.legend(li3 + li33, lab3 + lab33, loc=0)
133
134    # automatically adjusts subplot to fit in figure window
135    plt.tight_layout()
136
137    # save the figure in the working directory
138    if save:
139        plt.savefig('state_trajectory.pdf') # save output as pdf
140        # plt.savefig('state_trajectory.pgf') # for easy export to LaTeX, needs a lot of
        # extra packages
141    return None

```

Having defined the plotting function, one can execute it passing the calculated trajectories.

```
159 # plot
160 plot_data(x_traj, u_traj, tt, 12, 16, save=True)
161 plt.show()
```

The result can be found in Figure 6. Other properties of the plot, like line width or line color and many others, can be easily changed. One may refer to the [documentation of Matplotlib](#) or consult the exhaustive [Matplotlib example gallery](#).

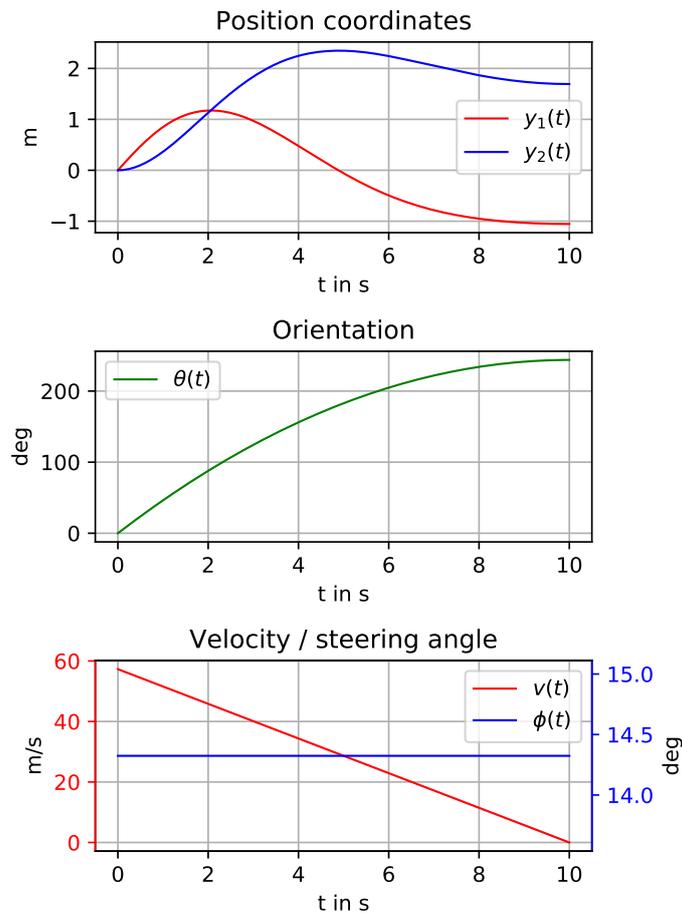


Figure 2: State and control trajectory plot created with Matplotlib.

7 Animation using Matplotlib

Source code file: `02_car_example_animation.py`

Plotting the state trajectory is often sufficient, but sometimes it can be helpful to have a visual representation of the system dynamics in order 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. One has to add

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

at the top of the code used in the previous sections. You need to install the ffmpeg library. For Windows it can be downloaded from <https://www.ffmpeg.org/download.html>. On Ubuntu based Linux systems it might be installed via `sudo apt install ffmpeg`.

Under Windows, it might be additionally necessary to explicitly specify the path to the `FFMPEG` library, e.g.:

```
plt.rcParams['animation.ffmpeg_path'] = 'C:\\path\\to\\ffmpeg\\ffmpeg.exe'
```

The Matplotlib animation package provides the class `FuncAnimation`. Objects of this class can be used to realize an animation. Three items need to be handed over to an object of this class when it is instantiated:

1. A handle to a figure into which the animation is rendered,
2. an animation function responsible for drawing *a single frame* of the animation. It must have the signature `def animate(i)` where `i` denotes the i th frame to be drawn,
3. an initialization function which is called before the animation starts. It cleans up the content of the figure.

In this tutorial all this is encapsulated in a function called `car_animation()`.

```
146 def car_animation(x, t, p: Type[Para]):
147     """Animation function of the car-like mobile robot
148
149     Args:
150         x(ndarray): state-vector trajectory
151         t(ndarray): time vector
152         p(object): parameters
153
154     Returns: None
155
156     """
```

This function provides the `init` and `animate` functions required by the `FuncAnimation` object as sub-functions. A third sub-function `draw_the_car` is used to draw the car in a certain state. This is done by plotting lines. All lines that represent the vehicle are defined by points, which depend on the current state \mathbf{x} and the control input \mathbf{u} . Hence, one needs 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 geometric relations and passes these to the plot instance `h_car`:

```

160 def draw_the_car(cur_x, cur_y):
161     """Mapping from state x and action cur_y to the position of the car elements
162
163     Args:
164         cur_x: The current state vector
165         cur_y: The current action vector
166
167     Returns:
168
169     """
170     wheel_length = 0.1 * p.l
171     y1, y2, theta = cur_x
172     v, phi = cur_y
173
174     # define chassis lines
175     chassis_y1 = [y1, y1 + p.l * cos(theta)]
176     chassis_y2 = [y2, y2 + p.l * sin(theta)]
177
178     # define lines for the front and rear axle
179     rear_ax_y1 = [y1 + p.w * sin(theta), y1 - p.w * sin(theta)]
180     rear_ax_y2 = [y2 - p.w * cos(theta), y2 + p.w * cos(theta)]
181     front_ax_y1 = [chassis_y1[1] + p.w * sin(theta + phi),
182                  chassis_y1[1] - p.w * sin(theta + phi)]
183     front_ax_y2 = [chassis_y2[1] - p.w * cos(theta + phi),
184                  chassis_y2[1] + p.w * cos(theta + phi)]
185
186     # define wheel lines
187     rear_l_wl_y1 = [rear_ax_y1[1] + wheel_length * cos(theta),
188                  rear_ax_y1[1] - wheel_length * cos(theta)]
189     rear_l_wl_y2 = [rear_ax_y2[1] + wheel_length * sin(theta),
190                  rear_ax_y2[1] - wheel_length * sin(theta)]
191     rear_r_wl_y1 = [rear_ax_y1[0] + wheel_length * cos(theta),
192                  rear_ax_y1[0] - wheel_length * cos(theta)]
193     rear_r_wl_y2 = [rear_ax_y2[0] + wheel_length * sin(theta),
194                  rear_ax_y2[0] - wheel_length * sin(theta)]
195     front_l_wl_y1 = [front_ax_y1[1] + wheel_length * cos(theta + phi),
196                  front_ax_y1[1] - wheel_length * cos(theta + phi)]
197     front_l_wl_y2 = [front_ax_y2[1] + wheel_length * sin(theta + phi),
198                  front_ax_y2[1] - wheel_length * sin(theta + phi)]
199     front_r_wl_y1 = [front_ax_y1[0] + wheel_length * cos(theta + phi),
200                  front_ax_y1[0] - wheel_length * cos(theta + phi)]
201     front_r_wl_y2 = [front_ax_y2[0] + wheel_length * sin(theta + phi),
202                  front_ax_y2[0] - wheel_length * sin(theta + phi)]
203
204     # empty value (to disconnect points, define where no line should be plotted)
205     empty = [np.nan, np.nan]
206
207     # concatenate set of coordinates
208     data_y1 = [rear_ax_y1, empty, front_ax_y1, empty, chassis_y1,
209              empty, rear_l_wl_y1, empty, rear_r_wl_y1,
210              empty, front_l_wl_y1, empty, front_r_wl_y1]
211     data_y2 = [rear_ax_y2, empty, front_ax_y2, empty, chassis_y2,
212              empty, rear_l_wl_y2, empty, rear_r_wl_y2,
213              empty, front_l_wl_y2, empty, front_r_wl_y2]
214
215     # set data
216     h_car.set_data(data_y1, data_y2)

```

Note that `draw_the_car` is in the scope of the `car_animation` function and, hence, has full access to the handle `h_car` defined there.

The `init()`-function defines which objects change during the animation, in this case the

two axes the handles of which are returned:

```

220 def init():
221     """Initialize plot objects that change during animation.
222     Only required for blitting to give a clean slate.
223
224     Returns:
225
226     """
227     h_x_traj_plot.set_data([], [])
228     h_car.set_data([], [])
229     return h_x_traj_plot, h_car

```

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

```

233 def animate(i):
234     """Defines what should be animated
235
236     Args:
237         i: frame number
238
239     Returns:
240
241     """
242     k = i % len(t)
243     ax.set_title('Time (s): ' + '%.2f' % t[k], loc='left')
244     h_x_traj_plot.set_xdata(x[0:k, 0])
245     h_x_traj_plot.set_ydata(x[0:k, 1])
246     draw_the_car(x[k, :], control(t[k]))
247     return h_x_traj_plot, h_car

```

The main function creates a figure with two empty plots into which the car and the curve of the trajectory depending on the state \mathbf{x} , the control input \mathbf{u} and the parameters are plotted later:

```

251 # Setup two empty axes with enough space around the trajectory so the car
252 # can always be completely plotted. One plot holds the sketch of the car,
253 # the other the curve
254 dx = 1.5 * p.l
255 dy = 1.5 * p.l
256 fig2, ax = plt.subplots()
257 ax.set_xlim([min(min(x_traj[:, 0] - dx), -dx),
258             max(max(x_traj[:, 0] + dx), dx)])
259 ax.set_ylim([min(min(x_traj[:, 1] - dy), -dy),
260             max(max(x_traj[:, 1] + dy), dy)])
261 ax.set_aspect('equal')
262 ax.set_xlabel(r'$y_1$')
263 ax.set_ylabel(r'$y_2$')
264
265 # Axis handles
266 h_x_traj_plot, = ax.plot([], [], 'b') # state trajectory in the y1-y2-plane
267 h_car, = ax.plot([], [], 'k', lw=2) # car

```

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

Finally, an object of type `FuncAnimation` is instantiated. It takes the `animate()` and `init()` functions as well as the figure handle as arguments in the constructor:

```

271 ani = mpla.FuncAnimation(fig2, animate, init_func=init, frames=len(t) + 1,
272                          interval=(t[1] - t[0]) * 1000,
273                          blit=False)
274
275 ani.save('animation.mp4', writer='ffmpeg', fps=1 / (t[1] - t[0]))

```

Now the system can be simulated with animated results.

```

283 # time vector
284 tt = np.arange(SimPara.t0, SimPara.tf + SimPara.dt, SimPara.dt)
285
286 # initial state
287 x0 = [0, 0, 0]
288
289 # simulation
290 sol = sci.solve_ivp(lambda t, x: ode(t, x, Para), (SimPara.t0, SimPara.tf), x0, t_eval=tt)
291 x_traj = sol.y.T
292 u_traj = control(tt)
293
294 # plot
295 plot_data(x_traj, u_traj, tt, 12, 16, save=True)
296
297 # animation
298 car_animation(x_traj, tt, Para)
299
300 plt.show()

```

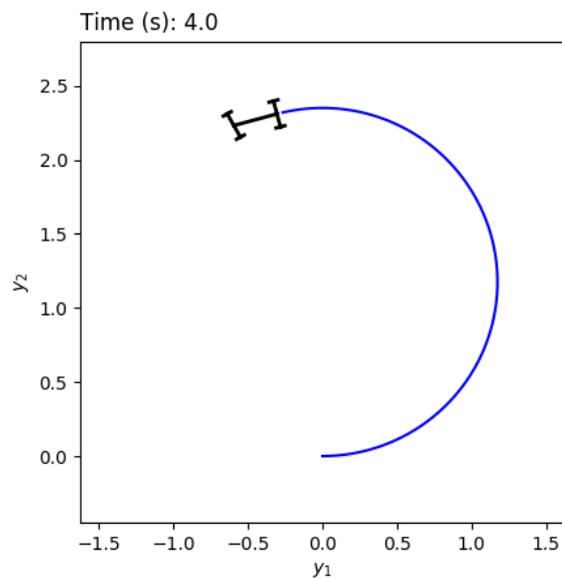


Figure 3: Car animation

8 Time-Events

Source code file: `03_events.py`

It is sometimes necessary to cancel the simulation, for example if the system is unstable and the state gets very large in a short period of time. A function `event(t, x)` is defined, that returns 0, if a certain condition is met. This is called a zero-crossing detection. The solver detects the sign switch of `event(t, x)` while calculating the solution of the [ODE](#).

```
def event(t, x):
    """Returns 0, if simulation should be terminated"""

    x_max = 5 # bound of the state variable x
    return np.abs(x)-x_max

# set the attribute 'terminal' of event, to stop the simulation, when zero-crossing is
# detected.
event.terminal = True

# simulate the system with event detection
sol = solve_ivp(lambda t, x: ode(x, t, para),
                (t0, tf), x0, method='RK45', t_eval=tt, events=event)
```

Glossary

IVP initial value problem. [3–5](#)

ODE ordinary differential equation. [2–4, 12](#)

References

- [1] Carsten Knoll and Robert Heedt. *Python für Ingenieure für Dummies: Mit vielen Programmbeispielen zu Numpy, Matplotlib und mehr*. Weinheim: Wiley-VCH, 2021.