Python for simulation, animation and control

Introductory tutorial for the simulation of dynamic systems

Demonstration using the model of a kinematic Vehicle

Max Pritzkoleit* Jan Winkler*

February 2, 2020

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.

***Python* 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].

2 Kinematic model of a vehicle

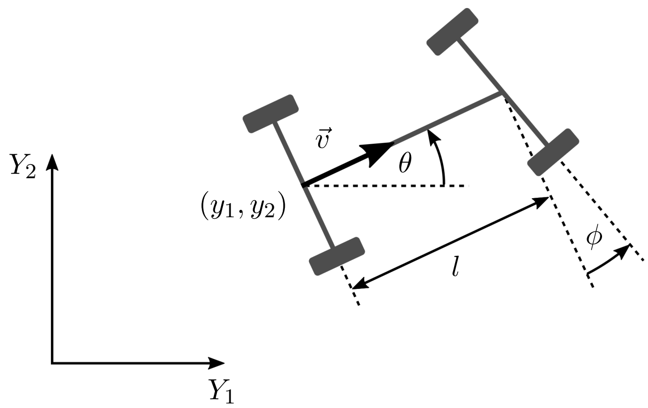

Figure 1: Car-like mobile robot

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

[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`

vehicle's lateral velocity $v = |\mathbf{v}|$:

$$\dot{y}_1(t) = v\cos(\theta(t)) \qquad y_1(0) = y_{10} \tag{1a}$$
$$\dot{y}_2(t) = v\sin(\theta(t)) \qquad y_2(0) = y_{20} \tag{1b}$$
$$\dot{\theta}(t) = \frac{1}{l}v(t)\tan(\phi(t)) \qquad \theta(0) = \theta_0. \tag{1c}$$

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 $\mathbf{x} = (x_1, x_2, x_3)^{\mathrm{T}}$ and a control vector $\mathbf{u} = (u_1, u_2)^{\mathrm{T}}$ as follows:

$$x_1 := y_1 \qquad u_1 := v \tag{2a}$$
$$x_2 := y_2 \qquad u_2 := \phi\,. \tag{2b}$$
$$x_3 := \theta \tag{2c}$$

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 $\mathbf{x}(0) = \mathbf{x}_0$:

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

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 setup 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 script[4].

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),

[4]It 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".

- *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.`:

```
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

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

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

```
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*:

```
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. Basically, such a structure can be an instance of an empty class derived from `object` to which members holding the parameter values are subsequently assigned:

```
class Parameters(object):
    pass


# Physical parameter
para = Parameters()  # instance of class Parameters
para.l = 0.3         # define car length
para.w = para.l*0.3  # define car width
```

Similarly this can be done with the simulation parameters:

```
# Simulation parameter
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
```

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.

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:

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

    Args:
        x        : state
        t        : time
        p(object): parameter container class

    Returns:
        dxdt: state derivative
    """
    x1, x2, x3 = x  # state vector
    u1, u2 = control(t, x)  # control vector

    # dxdt = f(x, u):
    dxdt = np.array([u1 * cos(x3),
                     u1 * sin(x3),
                     1 / p.l * u1 * tan(u2)])

    # return state derivative
    return dxdt
```

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

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

    Args:
        x: state vector
        t: time

    Returns:
        u: control vector

    """
    u1 = np.maximum(0, 1.0 - 0.1 * t)
    u2 = np.full(u1.shape, 0.25)
    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 $\mathbf{x}$ (the values for $\mathbf{u}$ in the `ode` function cannot be directly saved because the function is also repeatedly called between the specified time steps by the solver).

```
# time vector
tt = np.arange(sim_para.t0, sim_para.tf + sim_para.dt, sim_para.dt)

# initial state
x0 = [0, 0, 0]

# simulation
sol = sci.solve_ivp(lambda t, x: ode(t, x, para), (sim_para.t0, sim_para.tf), x0, t_eval=tt)
x_traj = sol.y.T
u_traj = control(tt, x_traj)
```

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

[5]The lambda function corresponds to @ in *MATLAB*

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.

```python
def plot_data(x, u, t, fig_width, fig_height, save=False):
    """Plotting function of simulated state and actions

    Args:
        x(ndarray) : state-vector trajectory
        u(ndarray) : control vector trajectory
        t(ndarray) : time vector
        fig_width  : figure width in cm
        fig_height : figure height in cm
        save (bool): save figure (default: False)
    Returns: None

    """
    # creating a figure with 3 subplots, that share the x-axis
    fig1, (ax1, ax2, ax3) = plt.subplots(3)

    # set figure size to desired values
    fig1.set_size_inches(fig_width / 2.54, fig_height / 2.54)

    # plot y_1 in subplot 1
    ax1.plot(t, x[:, 0], label='$y_1(t)$', lw=1, color='r')

    # plot y_2 in subplot 1
    ax1.plot(t, x[:, 1], label='$y_2(t)$', lw=1, color='b')

    # plot theta in subplot 2
    ax2.plot(t, np.rad2deg(x[:, 2]), label=r'$\theta(t)$', lw=1, color='g')

    # plot control in subplot 3, left axis red, right blue
    ax3.plot(t, np.rad2deg(u[:, 0]), label=r'$v(t)$', lw=1, color='r')
    ax3.tick_params(axis='y', colors='r')
    ax33 = ax3.twinx()
    ax33.plot(t, np.rad2deg(u[:, 1]), label=r'$\phi(t)$', lw=1, color='b')
    ax33.spines["left"].set_color('r')
    ax33.spines["right"].set_color('b')
    ax33.tick_params(axis='y', colors='b')

    # Grids
    ax1.grid(True)
    ax2.grid(True)
    ax3.grid(True)

    # set the labels on the x and y axis and the titles
    ax1.set_title('Position coordinates')
    ax1.set_ylabel(r'm')
    ax1.set_xlabel(r't in s')
    ax2.set_title('Orientation')
    ax2.set_ylabel(r'deg')
    ax2.set_xlabel(r't in s')
    ax3.set_title('Velocity / steering angle')
    ax3.set_ylabel(r'm/s')
    ax33.set_ylabel(r'deg')
    ax3.set_xlabel(r't in s')

    # put a legend in the plot
    ax1.legend()
```

```python
    ax2.legend()
    ax3.legend()
    li3, lab3 = ax3.get_legend_handles_labels()
    li33, lab33 = ax33.get_legend_handles_labels()
    ax3.legend(li3 + li33, lab3 + lab33, loc=0)

    # automatically adjusts subplot to fit in figure window
    plt.tight_layout()

    # save the figure in the working directory
    if save:
        plt.savefig('state_trajectory.pdf')  # save output as pdf
        # plt.savefig('state_trajectory.pgf')  # for easy export to LaTeX, needs a lot of extra
            packages
    return None
```

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

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

The result can be found in 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.

7 Animation using *Matplotlib*

Python 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

```python
import matplotlib.pyplot as plt
```

at the top of the code used in the previous sections. Under Windows it might be necessary to explicitely specify the path to the *FFMPG* library, e.g.:

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

FFMPG can be downloaded from `https://www.ffmpeg.org/download.html`.

All processing steps required for the animation are encapsulated in a function called `car_animation()`. At first this functions creates a figure with two empty plots into which the car and the curve of the trajectory dependung on the state $\mathbf{x}$, the control input $\mathbf{u}$ and the parameters are plotted later:

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

    Args:
        x(ndarray): state-vector trajectory
        u(ndarray): control vector trajectory
        t(ndarray): time vector
```

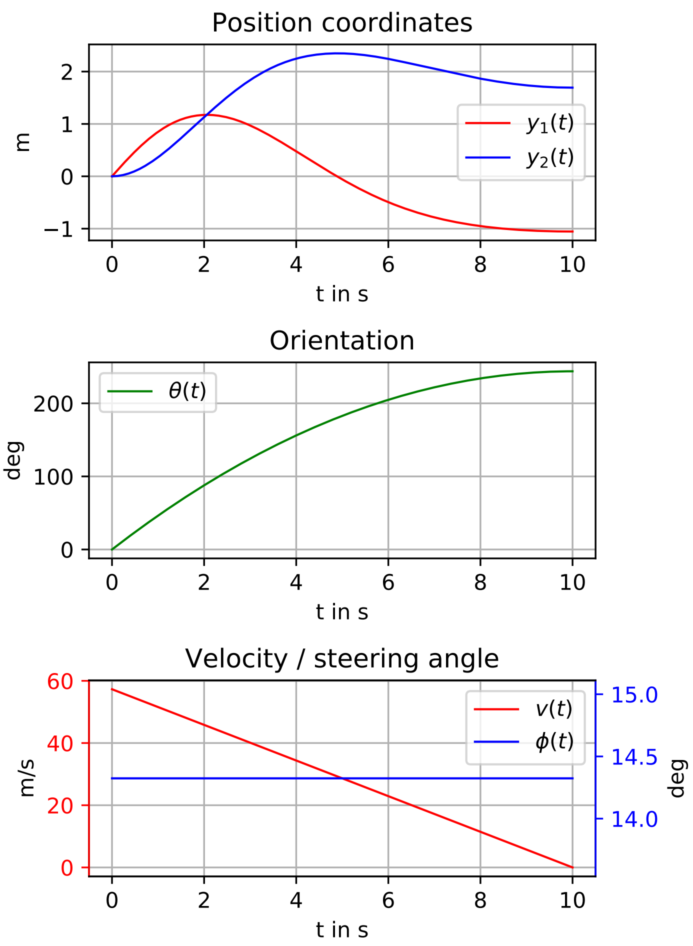

Figure 2: State and control trajectory plot created with *Matplotlib*.

```
        p(object): parameters

    Returns: None

    """
    # Setup two empty axes with enough space around the trajectory so the car
    # can always be completely plotted. One plot holds the sketch of the car,
    # the other the curve
    dx = 1.5 * p.l
    dy = 1.5 * p.l
    fig2, ax = plt.subplots()
    ax.set_xlim([min(min(x_traj[:, 0] - dx), -dx),
                 max(max(x_traj[:, 0] + dx), dx)])
    ax.set_ylim([min(min(x_traj[:, 1] - dy), -dy),
                 max(max(x_traj[:, 1] + dy), dy)])
    ax.set_aspect('equal')
    ax.set_xlabel(r'$y_1$')
    ax.set_ylabel(r'$y_2$')

    # Axis handles
    h_x_traj_plot, = ax.plot([], [], 'b')  # state trajectory in the y1-y2-plane
```

```
    h_car, = ax.plot([], [], 'k', lw=2)    # car
```

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

In the animation a representation of the vehicle has to be subsequently drawn. 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 `car`:

```
    def car_plot(x, u):
        """Mapping from state x and action u to the position of the car elements

        Args:
            x: state vector
            u: action vector

        Returns:

        """
        wheel_length = 0.1 * p.l
        y1, y2, theta = x
        v, phi = u

        # define chassis lines
        chassis_y1 = [y1, y1 + p.l * cos(theta)]
        chassis_y2 = [y2, y2 + p.l * sin(theta)]

        # define lines for the front and rear axle
        rear_ax_y1 = [y1 + p.w * sin(theta), y1 - p.w * sin(theta)]
        rear_ax_y2 = [y2 - p.w * cos(theta), y2 + p.w * cos(theta)]
        front_ax_y1 = [chassis_y1[1] + p.w * sin(theta + phi),
                       chassis_y1[1] - p.w * sin(theta + phi)]
        front_ax_y2 = [chassis_y2[1] - p.w * cos(theta + phi),
                       chassis_y2[1] + p.w * cos(theta + phi)]

        # define wheel lines
        rear_l_wl_y1 = [rear_ax_y1[1] + wheel_length * cos(theta),
                        rear_ax_y1[1] - wheel_length * cos(theta)]
        rear_l_wl_y2 = [rear_ax_y2[1] + wheel_length * sin(theta),
                        rear_ax_y2[1] - wheel_length * sin(theta)]
        rear_r_wl_y1 = [rear_ax_y1[0] + wheel_length * cos(theta),
                        rear_ax_y1[0] - wheel_length * cos(theta)]
        rear_r_wl_y2 = [rear_ax_y2[0] + wheel_length * sin(theta),
                        rear_ax_y2[0] - wheel_length * sin(theta)]
        front_l_wl_y1 = [front_ax_y1[1] + wheel_length * cos(theta + phi),
                         front_ax_y1[1] - wheel_length * cos(theta + phi)]
        front_l_wl_y2 = [front_ax_y2[1] + wheel_length * sin(theta + phi),
                         front_ax_y2[1] - wheel_length * sin(theta + phi)]
        front_r_wl_y1 = [front_ax_y1[0] + wheel_length * cos(theta + phi),
                         front_ax_y1[0] - wheel_length * cos(theta + phi)]
        front_r_wl_y2 = [front_ax_y2[0] + wheel_length * sin(theta + phi),
                         front_ax_y2[0] - wheel_length * sin(theta + phi)]

        # empty value (to disconnect points, define where no line should be plotted)
        empty = [np.nan, np.nan]

        # concatenate set of coordinates
        data_y1 = [rear_ax_y1, empty, front_ax_y1, empty, chassis_y1,
                   empty, rear_l_wl_y1, empty, rear_r_wl_y1,
                   empty, front_l_wl_y1, empty, front_r_wl_y1]
        data_y2 = [rear_ax_y2, empty, front_ax_y2, empty, chassis_y2,
                   empty, rear_l_wl_y2, empty, rear_r_wl_y2,
```

```
223                     empty, front_l_wl_y2, empty, front_r_wl_y2]
224
225         # set data
226         h_car.set_data(data_y1, data_y2)
```

Note that `car_plot` is in the scope of the `car_animation` function and, hence, has full acess to the handle `h_car` here.

Two further functions are required: `init()` and `animate(i)`. They will be called later by *Matplotlib* to initialize and perform the animation. The `init()`-function defines which objects change during the animation, in this case the two axes the handles of which are returned:

```
230     def init():
231         """Initialize plot objects that change during animation.
232            Only required for blitting to give a clean slate.
233
234         Returns:
235
236         """
237         h_x_traj_plot.set_data([], [])
238         h_car.set_data([], [])
239         return h_x_traj_plot, h_car
```

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

```
243     def animate(i):
244         """Defines what should be animated
245
246         Args:
247             i: frame number
248
249         Returns:
250
251         """
252         k = i % len(t)
253         ax.set_title('Time (s): ' + '%.2f' % t[k], loc='left')
254         h_x_traj_plot.set_xdata(x[0:k, 0])
255         h_x_traj_plot.set_ydata(x[0:k, 1])
256         car_plot(x[k, :], control(t[k], x[k, :]))
257         return h_x_traj_plot, h_car
```

Finally an object of type `FuncAnimation` is instancciated. It is provided by the animation subpackage of *Matplotlib*. It takes `animate()` and `init()` as argumets in the constructor:

```
261     ani = mpla.FuncAnimation(fig2, animate, init_func=init, frames=len(t) + 1,
262                              interval=(t[1] - t[0]) * 1000,
263                              blit=False)
264
265     file_format = 'mp4'
266     ani.save('animation.'+file_format, writer='ffmpeg', fps=1 / (t[1] - t[0]))
```

Note that all lines from 138 to 258 belong to the function `car_animation`!

Now the system can be simulated with animated results.

```
274 # time vector
275 tt = np.arange(sim_para.t0, sim_para.tf + sim_para.dt, sim_para.dt)
276
277 # initial state
278 x0 = [0, 0, 0]
```

```

# simulation
sol = sci.solve_ivp(lambda t, x: ode(t, x, para), (sim_para.t0, sim_para.tf), x0, t_eval=tt)
x_traj = sol.y.T
u_traj = control(tt, x_traj)

# plot
plot_data(x_traj, u_traj, tt, 12, 16, save=True)

# animation
car_animation(x_traj, u_traj, tt, para)

plt.show()
```

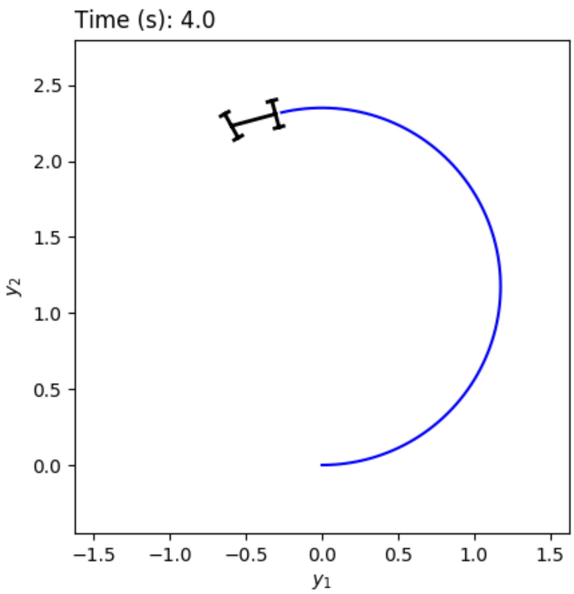

Figure 3: Car animation

8 Time-Events

Python 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
```

```python
# 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)
```