Control Theory Tutorial - Car-Like Mobile Robot

1 Introduction

The goal of this tutorial is to teach the usage of *Python* as a tool for developing and simulating control systems.

2 Model of a car-like mobile robot

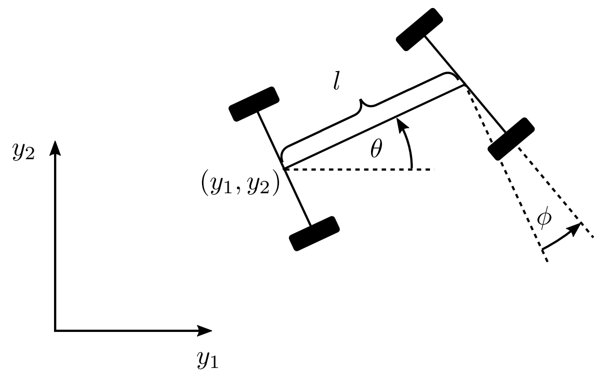

Figure 1: Car-like mobile robot

Given is a nonlinear kinematic model of a car-like mobile robot, with the following system variables: position (y_1, y_2) and orientation θ in the plane, the steering angle ϕ and the robots lateral velocity v.

$$\dot{y_1} = v\cos(\theta) \tag{1a}$$
$$\dot{y_2} = v\sin(\theta) \tag{1b}$$
$$\tan(\phi) = \frac{l\dot{\theta}}{v} \tag{1c}$$

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

$$x_1 = y_1 \qquad u_1 = v$$
$$x_2 = y_2 \qquad u_2 = \phi$$
$$x_3 = \theta$$

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

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

3 Storing parameters

We store the parameters of our system in a class *Parameters()*.

```
class Parameters(object):
    pass
```

We therefore create an entity of *Parameters()* and assign attributes.

```
prmtrs = Parameters() # entity of class Parameters
prmtrs.l = 0.3 # define car length
prmtrs.w = prmtrs.l*0.3 # define car width
```

4 Simulation with SciPy's integrate package

To simulate (2) we need to implement the ODE system as a function in *Python* .

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

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

    Returns:
        dxdt: state derivative
    """
    x1, x2, x3 = x # state vector
    u1, u2 = control(x, t) # control vector
    # dxdt = f(x, u)
    dxdt = np.array([u1 * cos(x3),
                     u1 * sin(x3),
                     1 / prmtrs.l * u1 * tan(u2)])

    # return state derivative
    return dxdt
```

In order to use $\cos(\cdot), \sin(\cdot)$ and $\tan(\cdot)$ we need to import these functions at the beginning of our code.

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

The control law is also implemented as function.

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

    Args:
        x: state vector
        t: time

    Returns:
        u: control vector

    """
    u = [1, 0.25] # v, phi

    return u
```

As a first simple heuristic, we set (u_1, u_2) equal to constant values. Later we can implement an arbitrary function, for expample a feedback law $\mathbf{u} = k(\mathbf{x})$.

4.1 Solution of the initial value problem (IVP) using *SciPy*

To simulate (2) we need to solve an IVP. In *Python* we can use the library *SciPy* and its sub-package *integrate*, which delivers different solvers for IVPs.

```
from scipy.integrate import odeint
```

We then define the simulation time and the initial state value.

```
t0 = 0 # start
tend = 10 # end
dt = 0.01 # stepsize (not of the solver, just evaluation points)
tt = np.arange(t0, tend, dt) # simulation interval

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

Now we can parse these parameters and our ODE function to the solver.

```
x_traj = odeint(ode, x0, tt, args=(prmtrs, )) # solution of the IVP
```

The output is an array of size length(tt)$\times$lenght($\mathbf{x}$).

5 Plotting using *Matplotlib*

For plotting the output of our simulation, we use the library *Matplotlib* and its sub-package *pyplot*, which delivers a user experience similar to *MATLAB*.

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

We encase our plotting instructions in a function. This way, we can define parameters of our plot, which we would like to change easily, for example figure width, or if the figure should be saved on the hard drive.

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

    Args:
        fig_width: figure width in cm
        fig_height: figure height in cm
        save (bool) : save figure (default: False)

    Returns: None

    """
    # creating a figure with 2 subplots, that share the x-axis
    fig1, (ax1, ax2) = plt.subplots(2, sharex=True)

    # 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(tt, x_traj[:, 0], label='$y_1(t)$', lw=1, color='r')

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

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

    ax1.grid(True)
    ax2.grid(True)
    # set the labels on the x and y axis in subplot 1
    ax1.set_ylabel(r'm')
    ax1.set_xlabel(r't in s')
    ax2.set_ylabel(r'rad')
    ax2.set_xlabel(r't in s')

    # put a legend in the plot
    ax1.legend()
    ax2.legend()

    #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
    return None
```

Finally, we have to execute

```python
plt.show()
```

to display the results. If your not satisfied with the result, you can change other properties of the plot, like linewidth or -color and many others easily. Just look up the documentation of *Matplotlib*: https://matplotlib.org/index.html

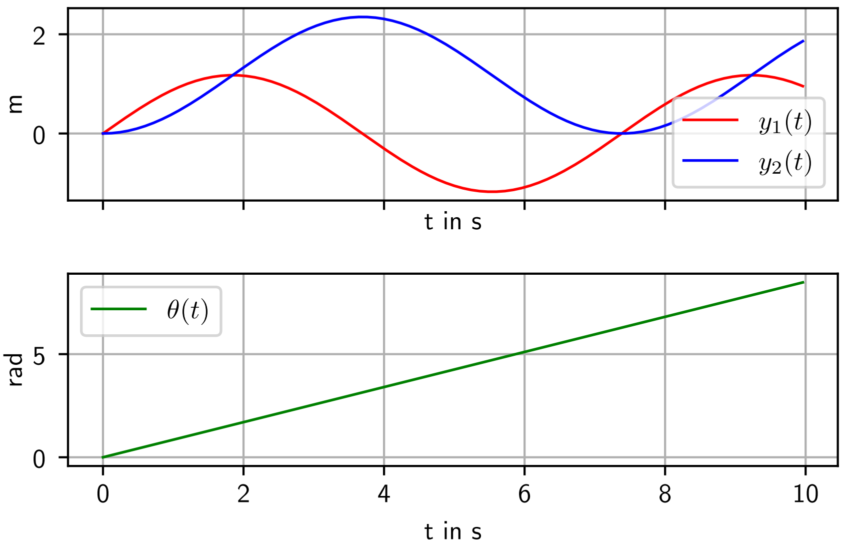

Figure 2: State trajectory plot created with *Matplotlib*