Project: Part 4

24-677 Special Topics: Modern Control - Theory and Design

Prof. D. Zhao

Due: Nov 23, 2021, 11:59 pm. Submit within the deadline.

- Your online version and its timestamp will be used for assessment.
- We will use Gradescope to grade. The link is on the panel of CANVAS. If you are confused about the tool, post your questions on Campuswire.
- Submit **your_controller.py** to Gradescope under **P4-code** and your solutions in **.pdf** format to **P4-writeup**. Insert the performance plot image in the **.pdf**. We will test **your_controller.py** and manually check all answers.
- We will make extensive use of Webots, an open-source robotics simulation software, for this project. Webots is available here for Windows, Mac, and Linux.
- For Python usage with Webots, please see the Webots page on Python. Note that you may have to reinstall libraries like `numpy`, `matplotlib`, `scipy`, etc. for the environment you use Webots in.
- Please familiarize yourself with Webots documentation, specifically their User Guide and their Webots for Automobiles section, if you encounter difficulties in setup or use. It will help to have a good understanding of the underlying tools that will be used in this assignment. To that end, completing at least Tutorial 1 in the user guide is highly recommended.
- If you have issues with Webots that are beyond the scope of the documentation (e.g. the software runs too slow, crashes, or has other odd behavior), please let the TAs know via Campuswire. We will do our best to help.
- We advise you to start with the assignment early. All the submissions are to be done before the respective deadlines of each assignment. For information about the late days and scale of your Final Grade, refer to the Syllabus in Canvas.

1 Introduction

In this project, you will complete the following goals:

1. Design a EKF SLAM to estimate the position and heading of the vehicle.

[Remember to submit the write-up, plots, and codes on Gradescope.]

2 P4: Problems

Exercise 1. In this final part of the project, you will design and implement an **Extended Kalman Filter Simultaneous Localization and Mapping (EKF SLAM)**. In previous parts, we assume that all state measurements are available. However, it is not always true in the real world. Localization information from GPS could be missing or inaccurate in the tunnel, or closed to tall infrastructures. In this case, we do not have direct access to the global position X, Y and heading ψ and have to estimates them from $\dot{x}$, $\dot{y}$, $\dot{\psi}$ on the vehicle frame and range and bearing measurements of map features.

Consider the discrete-time dynamics of the system:

$$\begin{aligned} X_{t+1} &= X_t + \delta t \dot{X}_t + \omega_t^x \\ Y_{t+1} &= Y_t + \delta t \dot{Y}_t + \omega_t^y \\ \psi_{t+1} &= \psi_t + \delta t \dot{\psi}_t + \omega_t^\psi \end{aligned} \tag{1}$$

Substitute $\dot{X}_t = \dot{x}_t \cos\psi_t - \dot{y}_t \sin\psi_t$ and $\dot{Y}_t = \dot{x}_t \sin\psi_t + \dot{y}_t \cos\psi_t$ in to (1),

$$\begin{aligned} X_{t+1} &= X_t + \delta t(\dot{x}_t \cos\psi_t - \dot{y}_t \sin\psi_t) + \omega_t^x \\ Y_{t+1} &= Y_t + \delta t(\dot{x}_t \sin\psi_t + \dot{y}_t \cos\psi_t) + \omega_t^y \\ \psi_{t+1} &= \psi_t + \delta t \dot{\psi}_t + \omega_t^\psi \end{aligned} \tag{2}$$

δt is the discrete time step. The input u_t is $[\dot{x}_t \quad \dot{y}_t \quad \dot{\psi}_t]^T$. Let $p_t = [X_t \quad Y_t]^T$. Suppose you have n map features at global position $m^j = [m_x^j \quad m_y^j]^T$ for $j = 1, ..., n$. The ground truth of these map feature positions are static but unknown, meaning they will not move but we do not know where they are exactly. However, the vehicle has both range and bearing measurements relative to these features. The range measurement is defined as the distance to each feature with the measurement equations $y_{t,distance}^j = \|m^j - p_t\| + v_{t,distance}^j$ for $j = 1, ..., n$. The bearing measure is defined as the angle between the vehicle's heading (yaw angle) and ray from the vehicle to the feature with the measurement equations $y_{t,bearing}^j = atan2(m_y^j - Y_t, m_x^j - X_t) - \psi_t + v_{t,bearing}^j$ for $j = 1, ..., n$, where the $v_{t,distance}^j$ and $v_{t,bearing}^j$ are the measurement noises.

Let the state vector be

$$\boldsymbol{x_t} = \begin{bmatrix} X_t \\ Y_t \\ \psi_t \\ m_x^1 \\ m_y^1 \\ m_x^2 \\ m_y^2 \\ \vdots \\ m_x^n \\ m_y^n \end{bmatrix} \tag{3}$$

The measurement system be

$$\boldsymbol{y_t} = \begin{bmatrix} \|m^1 - p_t\| \\ \vdots \\ \|m^n - p_t\| \\ atan2(m^1_y - Y_t, m^1_x - X_t) - \psi_t \\ \vdots \\ atan2(m^n_y - Y_t, m^n_x - X_t) - \psi_t \end{bmatrix} + \begin{bmatrix} v^1_{t,distance} \\ \vdots \\ v^n_{t,distance} \\ v^1_{t,bearing} \\ \vdots \\ v^n_{t,bearing} \end{bmatrix} \tag{4}$$

Derive F_t and H_t for EKF SLAM to estimate the vehicles state X, Y, ψ and feature positions $m^j = [m^j_x \quad m^j_y]^T$ simultaneously.

Hints:

- write out the complete dynamical system with the state in (3) and then follow the standard procedure of deriving EKF. Think what is the dynamic for static landmarks?

Solution:

By the definition of EKF,

$$\begin{aligned} F_t &= \left.\frac{\partial f(\boldsymbol{x_t}, \boldsymbol{u_t})}{\partial \boldsymbol{x_t}}\right|_{\boldsymbol{x_t}=\boldsymbol{\mu_{t-1|t-1}}} \\ &= \left[\begin{array}{ccc|c} 1 & 0 & -\delta t\left(\dot{x}_t sin(\psi_t) + \dot{y}_t cos(\psi_t)\right) & \\ 0 & 1 & \delta t\left(\dot{x}_t cos(\psi_t) - \dot{y}_t sin(\psi_t)\right) & O_{3\times 2n} \\ 0 & 0 & 1 & \\ \hline & O_{2n\times 3} & & I_{2n} \end{array}\right] \end{aligned} \tag{5}$$

$$\begin{aligned} H_t &= \left.\frac{\partial h(\boldsymbol{x_t}, \boldsymbol{u_t})}{\partial \boldsymbol{x_t}}\right|_{\boldsymbol{x_t}=\boldsymbol{\mu_{t|t-1}}} \\ &= \begin{bmatrix} -\frac{m^1_x - X_t}{\|m^1-p_t\|} & -\frac{m^1_y - Y_t}{\|m^1-p_t\|} & 0 & \frac{m^1_x - X_t}{\|m^1-p_t\|} & \frac{m^1_y - Y_t}{\|m^1-p_t\|} & 0 & 0 & \cdots & \cdots & 0 \\ -\frac{m^2_x - X_t}{\|m^2-p_t\|} & -\frac{m^2_y - Y_t}{\|m^2-p_t\|} & 0 & 0 & 0 & \frac{m^2_x - X_t}{\|m^2-p_t\|} & \frac{m^2_y - Y_t}{\|m^2-p_t\|} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{m^1_y - Y_t}{\|m^1-p_t\|^2} & -\frac{m^1_x - X_t}{\|m^1-p_t\|^2} & -1 & -\frac{m^1_y - Y_t}{\|m^1-p_t\|^2} & \frac{m^1_x - X_t}{\|m^1-p_t\|^2} & 0 & 0 & \cdots & \cdots & 0 \\ \frac{m^2_y - Y_t}{\|m^2-p_t\|^2} & -\frac{m^2_x - X_t}{\|m^2-p_t\|^2} & -1 & 0 & 0 & -\frac{m^2_y - Y_t}{\|m^2-p_t\|^2} & \frac{m^2_x - X_t}{\|m^2-p_t\|^2} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \end{aligned} \tag{6}$$

Then follow the standard predict and correct step of EKF.

Exercise 2. For this exercise, you will implement EKF SLAM in the `ekf_slam.py` file by completing four functions (marked with TODO in the script). You can use the same controller from your previous parts or write a new one. Before integrating this module with Webots, you can run the script by `$ python ekf_slam.py` to test your implementation. Feel free to write your own unit-testing scripts. You should not use any existing Python package that implements EKF. [Hints: remember to wrap heading angles to $[-\pi, \pi]$]

Check the performance of your controller by running the Webots simulation. You can press the play button in the top menu to start the simulation in real-time, the fast-forward button to run the simulation as quickly as possible, and the triple fast-forward to run the simulation without rendering (any of these options is acceptable, and the faster options may be better for quick tests). If you complete the track, the scripts will generate a performance plot via `matplotlib`. This plot contains a visualization of the car's trajectory, and also shows the variation of states with respect to time.

Submit `your_controller_ekf_slam.py`, `ekf_slam.py`, and the final completion plot as described on the title page. You do not need to submit the plot generated by running the test script (by running `$ python ekf_slam.py`). Your controller is **required** to achieve the following performance criteria to receive full points:

1. Time to complete the loop = 250 s
2. Maximum deviation from the reference trajectory = 10.0 m
3. Average deviation from the reference trajectory = 5 m

Debugging tips:

- Do not hardcode the number of map features. Instead, use n in your code.
- Check all the signs carefully.
- Check all the numpy array indexing.
- Use wrap_to_pi function smartly. Only wrap the angle terms but not the distance terms.
- When you run `$ python ekf_slam.py`, it is normal if the estimation diverge gradually, but you should have some reasonable tracking performance.

Solution:

```python
import numpy as np

class EKF_SLAM():
    def __init__(self, init_mu, init_P, dt, W, V, n):
        """Initialize EKF SLAM
        Create and initialize an EKF SLAM to estimate the robot's pose and
        the location of map features

        Args:
            init_mu: A numpy array of size (3+2*n, ). Initial guess of the
↪   mean
            of state.
            init_P: A numpy array of size (3+2*n, 3+2*n). Initial guess of
            the covariance of state.
            dt: A double. The time step.
            W: A numpy array of size (3+2*n, 3+2*n). Process noise
            V: A numpy array of size (2*n, 2*n). Observation noise
            n: A int. Number of map features


        Returns:
            An EKF SLAM object.
        """
        self.mu = init_mu  # initial guess of state mean
        self.P = init_P  # initial guess of state covariance
        self.dt = dt  # time step
        self.W = W  # process noise
        self.V = V  # observation noise
        self.n = n  # number of map features


    def _f(self, x, u):
        """Non-linear dynamic function.

        Compute the state at next time step according to the nonlinear
↪   dynamics f.

        Args:
            x: A numpy array of size (3+2*n, ). State at current time step.
            u: A numpy array of size (3, ). The control input [\dot{x},
↪   \dot{y}, \dot{\psi}]

        Returns:
            x_next: A numpy array of size (3+2*n, ). The state at next time
↪   step
        """
        x_dot = u[0]
        y_dot = u[1]
        psi_dot = u[2]
        X = x[0]
        Y = x[1]
        psi = x[2]

        X_next = X + self.dt*(x_dot*np.cos(psi) - y_dot*np.sin(psi))
        Y_next = Y + self.dt*(x_dot*np.sin(psi) + y_dot*np.cos(psi))
        # psi_next = psi + self.dt*psi_dot
        psi_next = self._wrap_to_pi(psi + self.dt*psi_dot)

        x_next = np.array([X_next, Y_next, psi_next])
        x_next = np.hstack((x_next, x[3:]))
        return x_next


    def _h(self, x):
        """Non-linear measurement function.

        Compute the sensor measurement according to the nonlinear function
↪   h.

        Args:
            x: A numpy array of size (3+2*n, ). State at current time step.

        Returns:
            y: A numpy array of size (2*n, ). The sensor measurement.
        """
        p = x[0:2]
        psi = x[2]
        m = x[3:].reshape((-1,2))

        y = np.zeros(2*self.n)

        for i in range(self.n):
            y[i] = np.linalg.norm(m[i, :] - p)
            y[self.n+i] = self._wrap_to_pi(np.arctan2(m[i,1]-p[1],
             ↪  m[i,0]-p[0]) - psi)
            # y[self.n+i] = np.arctan2(m[i,1]-p[1], m[i,0]-p[0]) - psi

        return y


    def _compute_F(self, x, u):
        """Compute Jacobian of f

        You will use self.mu in this function.
        Args:
            x: A numpy array of size (3+2*n, ). The state vector.
            u: A numpy array of size (3, ). The control input [\dot{x},
↪   \dot{y}, \dot{\psi}]

        Returns:
            F: A numpy array of size (3+2*n, 3+2*n). The jacobian of f
↪   evaluated at x_k.
        """
        psi_est = x[2]

        x_dot = u[0]
        y_dot = u[1]

        F = np.array([[1, 0, -self.dt*(x_dot*np.sin(psi_est) +
         ↪  y_dot*np.cos(psi_est))],
                      [0, 1,  self.dt*(x_dot*np.cos(psi_est) -
                       ↪  y_dot*np.sin(psi_est))],
                      [0, 0, 1]])
        F = np.hstack((F, np.zeros((3, 2*self.n))))
        F = np.vstack((F, np.hstack((np.zeros((2*self.n, 3)),
         ↪  np.eye(2*self.n)))))

        return F


    def _compute_H(self, x):
        """Compute Jacobian of h

        You will use self.mu in this function.

        Args:
            x: A numpy array of size (3+2*n, ). The state vector.

        Returns:
            H: A numpy array of size (2*n, 3+2*n). The jacobian of h
↪   evaluated at x_k.
        """

        X = x[0]
        Y = x[1]
        p = x[0:2]
        m = x[3:].reshape((-1,2))
        H = np.zeros((2*self.n, 3+2*self.n))
        for i in range(self.n):
            # distance sensor
            H[i, 0] = -(m[i, 0] - X)/np.linalg.norm(m[i, :]-p)
            H[i, 1] = -(m[i, 1] - X)/np.linalg.norm(m[i, :]-p)
            H[i, 3+i*2] = (m[i, 0] - X)/np.linalg.norm(m[i, :]-p)
            H[i, 4+i*2] = (m[i, 1] - Y)/np.linalg.norm(m[i, :]-p)

            # bearing sensor
            H[self.n+i, 0] =  (m[i, 1] - Y)/(np.linalg.norm(m[i, :]-p)**2)
            H[self.n+i, 1] = -(m[i, 0] - X)/(np.linalg.norm(m[i, :]-p)**2)
            H[self.n+i, 2] = -1
            H[self.n+i, 3+i*2] = -(m[i, 1] - Y)/(np.linalg.norm(m[i,
             ↪  :]-p)**2)
            H[self.n+i, 4+i*2] =  (m[i, 0] - X)/(np.linalg.norm(m[i,
             ↪  :]-p)**2)

        return H


    def predict_and_correct(self, y, u):
        """Predice and correct step of EKF

        You will use self.mu in this function. You must update self.mu in
↪   this function.

        Args:
            y: A numpy array of size (2*n, ). The measurements according to
↪   the project description.
            u: A numpy array of size (3, ). The control input [\dot{x},
↪   \dot{y}, \dot{\psi}]

        Returns:
            self.mu: A numpy array of size (3+2*n, ). The corrected state
↪   estimation
            self.P: A numpy array of size (3+2*n, 3+2*n). The corrected
↪   state covariance
        """

        # compute F
        F = self._compute_F(self.mu, u)

        #***************** Predict step *****************#
        # predict the state
        self.mu = self._f(self.mu, u)
        self.mu[2] =  self._wrap_to_pi(self.mu[2])
        # predict the error covariance
        self.P = F @ self.P @ F.T + self.W

        #***************** Correct step *****************#
        # compute H matrix
        H = self._compute_H(self.mu)

        # compute the Kalman gain
        L = self.P @ H.T @ np.linalg.inv(H @ self.P @ H.T + self.V)

        # update estimation with new measurement
        diff = y - self._h(self.mu)
        diff[self.n:] = self._wrap_to_pi(diff[self.n:])
        self.mu = self.mu + L @ diff
        self.mu[2] =  self._wrap_to_pi(self.mu[2])

        # update the error covariance
        self.P = (np.eye(3+2*self.n) - L @ H) @ self.P

        return self.mu, self.P


    def _wrap_to_pi(self, angle):
        angle = angle - 2*np.pi*np.floor((angle+np.pi )/(2*np.pi))
        return angle


if __name__ == '__main__':
    import matplotlib.pyplot as plt

    m = np.array([[0.,  0.],
                  [0.,  20.],
                  [20., 0.],
                  [20., 20.],
                  [0,  -20],
                  [-20, 0],
                  [-20, -20],
                  [-50, -50]]).reshape(-1)

    dt = 0.01
    T = np.arange(0, 20, dt)
    n = int(len(m)/2)
    W = np.zeros((3+2*n, 3+2*n))
    W[0:3, 0:3] = dt**2 * 1 * np.eye(3)
```

```python
V = 0.1*np.eye(2*n)
V[n:,n:] = 0.01*np.eye(n)

# EKF estimation
mu_ekf = np.zeros((3+2*n, len(T)))
mu_ekf[0:3,0] = np.array([2.2, 1.8, 0.])
# mu_ekf[3:,0] = m + 0.1
mu_ekf[3:,0] = m + np.random.multivariate_normal(np.zeros(2*n),
↪  0.5*np.eye(2*n))
init_P = 1*np.eye(3+2*n)

# initialize EKF SLAM
slam = EKF_SLAM(mu_ekf[:,0], init_P, dt, W, V, n)

# real state
mu = np.zeros((3+2*n, len(T)))
mu[0:3,0] = np.array([2, 2, 0.])
mu[3:,0] = m

y_hist = np.zeros((2*n, len(T)))
for i, t in enumerate(T):
    if i > 0:
        # real dynamics
        u = [-5, 2*np.sin(t*0.5), 1*np.sin(t*3)]
        # u = [0.5, 0.5*np.sin(t*0.5), 0]
        # u = [0.5, 0.5, 0]
        mu[:,i] = slam._f(mu[:,i-1], u) + \
            np.random.multivariate_normal(np.zeros(3+2*n), W)

        # measurements
        y = slam._h(mu[:,i]) +
        ↪  np.random.multivariate_normal(np.zeros(2*n), V)
        y_hist[:,i] = (y-slam._h(slam.mu))
        # apply EKF SLAM
        mu_est, _ = slam.predict_and_correct(y, u)
        mu_ekf[:,i] = mu_est


plt.figure(1, figsize=(10,6))
ax1 = plt.subplot(121, aspect='equal')
ax1.plot(mu[0,:], mu[1,:], 'b')
ax1.plot(mu_ekf[0,:], mu_ekf[1,:], 'r--')
mf = m.reshape((-1,2))
ax1.scatter(mf[:,0], mf[:,1])
ax1.set_xlabel('X')
```

```
ax1.set_ylabel('Y')

ax2 = plt.subplot(322)
ax2.plot(T, mu[0,:], 'b')
ax2.plot(T, mu_ekf[0,:], 'r--')
ax2.set_xlabel('t')
ax2.set_ylabel('X')

ax3 = plt.subplot(324)
ax3.plot(T, mu[1,:], 'b')
ax3.plot(T, mu_ekf[1,:], 'r--')
ax3.set_xlabel('t')
ax3.set_ylabel('Y')

ax4 = plt.subplot(326)
ax4.plot(T, mu[2,:], 'b')
ax4.plot(T, mu_ekf[2,:], 'r--')
ax4.set_xlabel('t')
ax4.set_ylabel('psi')

plt.figure(2)
ax1 = plt.subplot(211)
ax1.plot(T, y_hist[0:n, :].T)
ax2 = plt.subplot(212)
ax2.plot(T, y_hist[n:, :].T)

plt.show()
```

3 Appendix

(Already covered in P1)

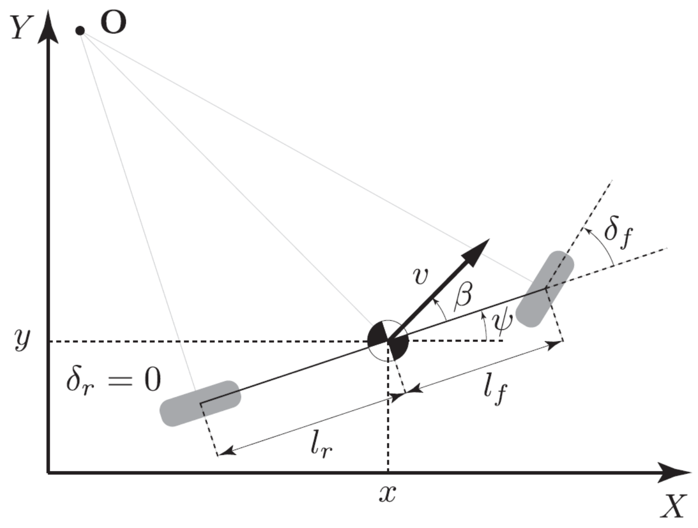

Figure 1: Bicycle model[2]

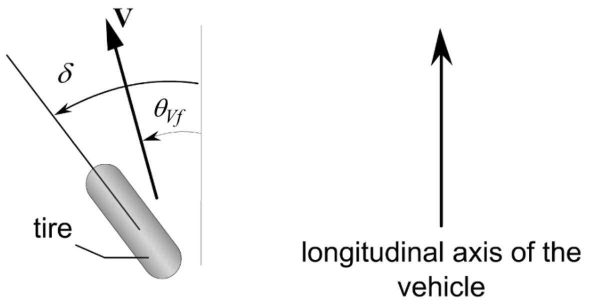

Figure 2: Tire slip-angle[2]

We will make use of a bicycle model for the vehicle, which is a popular model in the study of vehicle dynamics. Shown in Figure 1, the car is modeled as a two-wheel vehicle with two degrees of freedom, described separately in longitudinal and lateral dynamics. The model parameters are defined in Table 2.

3.1 Lateral dynamics

Ignoring road bank angle and applying Newton's second law of motion along the y-axis:

$$ma_y = F_{yf} \cos \delta_f + F_{yr}$$

where $a_y = \left(\dfrac{d^2y}{dt^2}\right)_{inertial}$ is the inertial acceleration of the vehicle at the center of geometry in the direction of the y axis, F_{yf} and F_{yr} are the lateral tire forces of the front and rear wheels, respectively, and δ_f is the front wheel angle, which will be denoted as δ later. Two terms contribute to a_y: the acceleration $\ddot{y}$, which is due to motion along the y-axis, and the centripetal acceleration. Hence:

$$a_y = \ddot{y} + \dot{\psi}\dot{x}$$

Combining the two equations, the equation for the lateral translational motion of the vehicle is obtained as:

$$\ddot{y} = -\dot{\psi}\dot{x} + \frac{1}{m}(F_{yf}\cos\delta + F_{yr})$$

Moment balance about the axis yields the equation for the yaw dynamics as

$$\ddot{\psi}I_z = l_f F_{yf} - l_r F_{yr}$$

The next step is to model the lateral tire forces F_{yf} and F_{yr}. Experimental results show that the lateral tire force of a tire is proportional to the "slip-angle" for small slip-angles when vehicle's speed is large enough - i.e. when $\dot{x} \geq 0.5$ m/s. The slip angle of a tire is defined as the angle between the orientation of the tire and the orientation of the velocity vector of the vehicle. The slip angle of the front and rear wheel is

$$\alpha_f = \delta - \theta_{Vf}$$
$$\alpha_r = -\theta_{Vr}$$

where θ_{Vp} is the angle between the velocity vector and the longitudinal axis of the vehicle, for $p \in \{f, r\}$. A linear approximation of the tire forces are given by

$$F_{yf} = 2C_\alpha\left(\delta - \frac{\dot{y} + l_f\dot{\psi}}{\dot{x}}\right)$$
$$F_{yr} = 2C_\alpha\left(-\frac{\dot{y} - l_r\dot{\psi}}{\dot{x}}\right)$$

where C_α is called the cornering stiffness of the tires. If $\dot{x} < 0.5$ m/s, we just set F_{yf} and F_{yr} both to zeros.

3.2 Longitudinal dynamics

Similarly, a force balance along the vehicle longitudinal axis yields:

$$\ddot{x} = \dot{\psi}\dot{y} + a_x$$
$$ma_x = F - F_f$$
$$F_f = fmg$$

where F is the total tire force along the x-axis, and F_f is the force due to rolling resistance at the tires, and f is the friction coefficient.

3.3 Global coordinates

In the global frame we have:

$$\begin{aligned}\dot{X} &= \dot{x}\cos\psi - \dot{y}\sin\psi\\ \dot{Y} &= \dot{x}\sin\psi + \dot{y}\cos\psi\end{aligned}$$

3.4 System equation

Gathering all of the equations, if $\dot{x} \geq 0.5$ m/s, we have:

$$\begin{aligned}\ddot{y} &= -\dot{\psi}\dot{x} + \frac{2C_\alpha}{m}(\cos\delta\left(\delta - \frac{\dot{y}+l_f\dot{\psi}}{\dot{x}}\right) - \frac{\dot{y}-l_r\dot{\psi}}{\dot{x}})\\ \ddot{x} &= \dot{\psi}\dot{y} + \frac{1}{m}(F - fmg)\\ \ddot{\psi} &= \frac{2l_fC_\alpha}{I_z}\left(\delta - \frac{\dot{y}+l_f\dot{\psi}}{\dot{x}}\right) - \frac{2l_rC_\alpha}{I_z}\left(-\frac{\dot{y}-l_r\dot{\psi}}{\dot{x}}\right)\\ \dot{X} &= \dot{x}\cos\psi - \dot{y}\sin\psi\\ \dot{Y} &= \dot{x}\sin\psi + \dot{y}\cos\psi\end{aligned}$$

otherwise, since the lateral tire forces are zeros, we only consider the longitudinal model.

3.5 Measurements

The observable states are:

$$y = \begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\psi}\\ X\\ Y\\ \psi\end{bmatrix}$$

3.6 Physical constraints

The system satisfies the constraints that:

$$\begin{array}{l}|\delta| \leqslant \frac{\pi}{6}\ rad\\ F \geqslant 0 \text{ and } F \leqslant 15736\ N\\ \dot{x} \geqslant 10^{-5}\ m/s\end{array}$$

Table 1: Model parameters.

Name	Description	Unit	Value
$(\dot{x}, \dot{y})$	Vehicle's velocity along the direction of vehicle frame	m/s	State
(X, Y)	Vehicle's coordinates in the world frame	m	State
ψ, $\dot{\psi}$	Body yaw angle, angular speed	rad, rad/s	State
δ or δ_f	Front wheel angle	rad	State
F	Total input force	N	Input
m	Vehicle mass	kg	1888.6
l_r	Length from rear tire to the center of mass	m	1.39
l_f	Length from front tire to the center of mass	m	1.55
C_α	Cornering stiffness of each tire	N	20000
I_z	Yaw intertia	kg m^2	25854
F_{pq}	Tire force, $p \in \{x, y\}$,$q \in \{f, r\}$	N	Depends on input force
f	Rolling resistance coefficient	N/A	0.019
`delT`	Simulation timestep	sec	0.032

3.7 Simulation

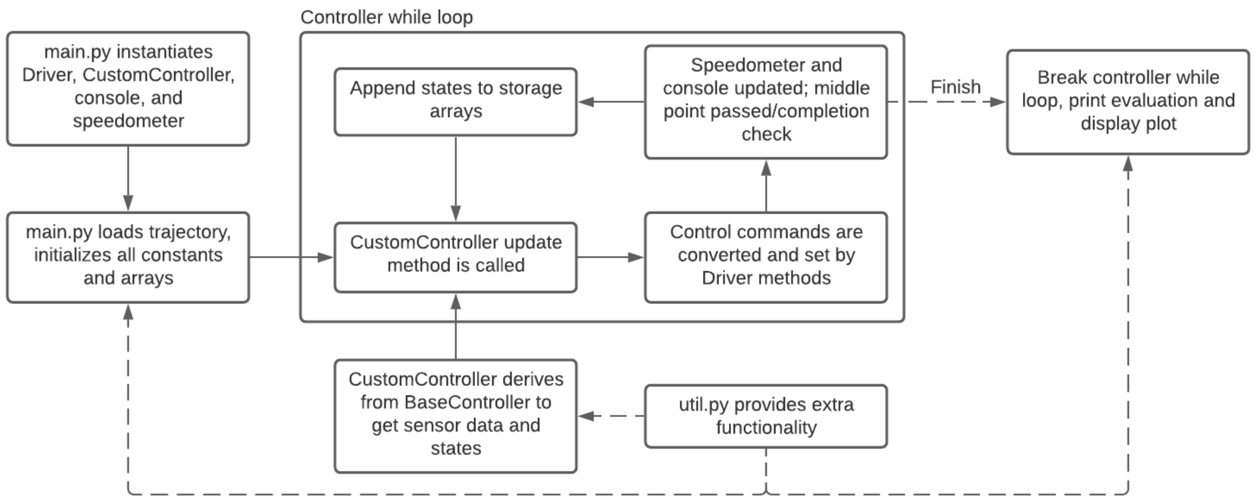
Controller while loop
main.py instantiates Driver, CustomController, console, and speedometer
main.py loads trajectory, initializes all constants and arrays
Append states to storage arrays
Speedometer and console updated; middle point passed/completion check
Finish
Break controller while loop, print evaluation and display plot
CustomController update method is called
Control commands are converted and set by Driver methods
CustomController derives from BaseController to get sensor data and states
util.py provides extra functionality

Figure 3: Simulation code flow

Several files are provided to you within the `controllers/main` folder. The `main.py` script initializes and instantiates necessary objects, and also contains the controller loop. This loop runs once each simulation timestep. `main.py` calls `your_controller.py`'s `update` method on each loop to get new control commands (the desired steering angle, δ, and longitudinal force, F). The longitudinal force is converted to a throttle input, and then both control commands are set by Webots internal functions. The additional script `util.py` contains functions to help you design and execute the controller. The full codeflow is pictured in Figure 3.

Please design your controller in the `your_controller.py` file provided for the project part you're working on. Specifically, you should be writing code in the `update` method. Please **do not** attempt to change code in other functions or files, as we will only grade the relevant `your_controller.py` for the programming portion. However, you are free to add to the `CustomController` class's `__init__` method (which is executed once when the `CustomController` object is instantiated).

3.8 BaseController Background

The `CustomController` class within each `your_controller.py` file derives from the `BaseController` class in the `base_controller.py` file. The vehicle itself is equipped with a Webots-generated GPS, gyroscope, and compass that have no noise or error. These sensors are started in the `BaseController` class, and are used to derive the various states of the vehicle. An explanation on the derivation of each can be found in the table below.

Table 2: State Derivation.

Name	Explanation
(X, Y)	From GPS readings
$(\dot{x}, \dot{y})$	From the derivative of GPS readings
ψ	From the compass readings
$\dot{\psi}$	From the gyroscope readings

3.9 Trajectory Data

The trajectory is given in `buggyTrace.csv`. It contains the coordinates of the trajectory as (x, y). The satellite map of the track is shown in Figure 4.

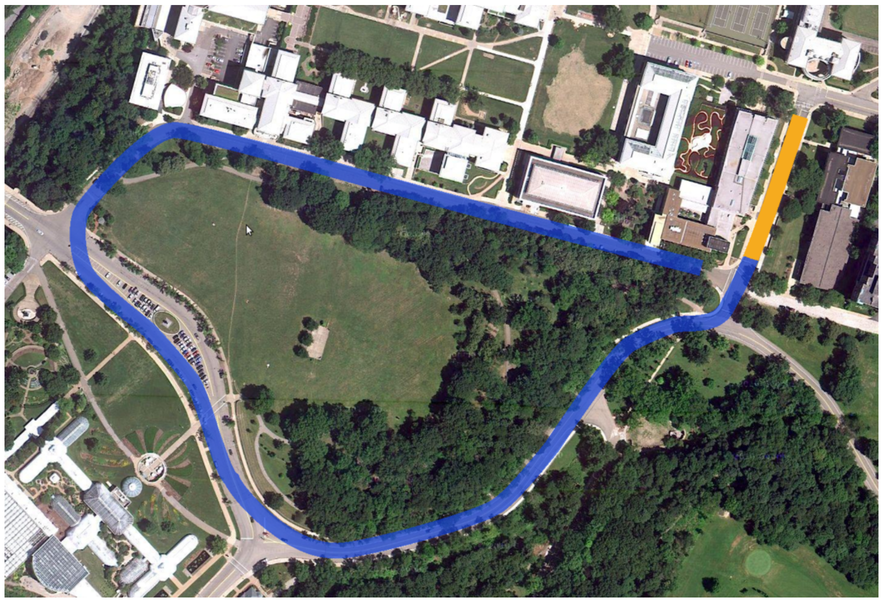

Figure 4: Buggy track[3]

4 Reference

1. Rajamani Rajesh. Vehicle Dynamics and Control. Springer Science & Business Media, 2011.

2. Kong Jason, et al. "Kinematic and dynamic vehicle models for autonomous driving control design." Intelligent Vehicles Symposium, 2015.

3. cmubuggy.org, https://cmubuggy.org/reference/File:Course_hill1.png

4. "PID Controller - Manual Tuning." *Wikipedia*, Wikimedia Foundation, August 30th, 2020. https://en.wikipedia.org/wiki/PID_controller#Manual_tuning