RBE 502 — Robot Control

Instructor: Siavash Farzan

Spring 2023

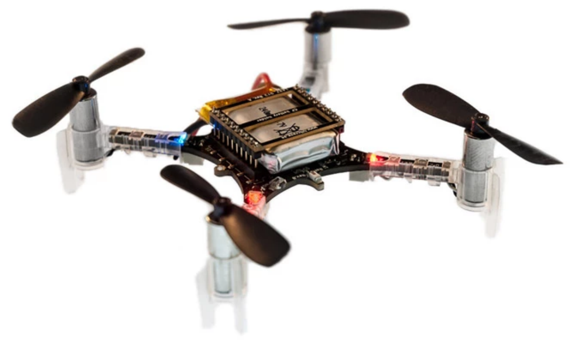
Final Project: Robust Trajectory Tracking for Quadrotor UAVs using Sliding Mode Control

1.1 Overview

The objective of this project is to develop a robust control scheme to enable a quadrotor to track desired trajectories in the presence of external disturbances.

The control design under study will be tested on the Crazyflie 2.0 platform. Crazyflie is a quadrotor that is classified as a micro air vehicle (MAV), as it only weighs 27 grams and can fit in your hand. The size makes it ideal for flying inside a lab without trashing half the interior. Even though the propellers spin at high RPMs, they are soft and the torque in the motors is very low when compared to a brushless motor, making it relatively crash tolerant. The Crazyflie 2.0 features four 7mm coreless DC-motors that give the Crazyflie a maximum takeoff weight of 42g.

The Crazyflie 2.0 is an open source project, with source code and hardware design both documented and available. For more information, see the link below:
`https://www.bitcraze.io/products/old-products/crazyflie-2-0/`


(a)

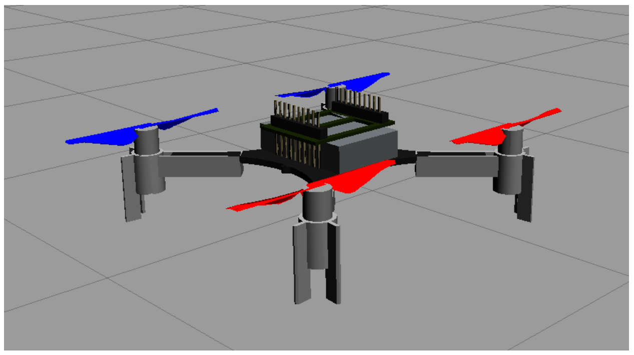

(b)

Figure 1: Crazyflie 2.0 Quadrotor (a) hardware, (b) Gazebo physics based simulation.

1.2 Crazyflie 2.0 Setup in Gazebo

For this project, we assume that you have already installed Ubuntu 20.04 and ROS Noetic by following the instructions provided in Programming Assignment 0.

To set up the Crazyflie 2.0 quadrotor in Gazebo, we need to install additional ROS dependencies for building packages as below:

```
sudo apt update
sudo apt install ros-noetic-joy ros-noetic-octomap-ros ros-noetic-mavlink
sudo apt install ros-noetic-octomap-mapping ros-noetic-control-toolbox
sudo apt install python3-vcstool python3-catkin-tools protobuf-compiler
   libgoogle-glog-dev
rosdep update
sudo apt-get install ros-noetic-ros libgoogle-glog-dev
```

We are now ready to create a new ROS workspace and download the ROS packages for the robot:

```
mkdir -p ~/rbe502_project/src
cd ~/rbe502_project/src
catkin_init_workspace  # initialize your catkin workspace
cd ~/rbe502_project
catkin init
cd ~/rbe502_project/src
git clone -b dev/ros-noetic https://github.com/gsilano/CrazyS.git
git clone -b med18_gazebo9 https://github.com/gsilano/mav_comm.git
```

Note: a new ROS workspace is needed for the project, because the `CrazyS` Gazebo package is built using the `catkin build` tool, instead of `catkin_make`.

We need to build the project workspace using `python_catkin_tools`, therefore we need to configure it:

```
cd ~/rbe502_project
rosdep install --from-paths src -i
rosdep update
catkin config --cmake-args -DCMAKE_BUILD_TYPE=Release -DCATKIN_ENABLE_TESTING=
   False
catkin build
```

Do not forget to add sourcing to your `.bashrc` file:

```
echo "source ~/rbe502_project/devel/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

With all dependencies ready, we can build the ROS package by the following commands:

```
cd ~/rbe502_project
catkin build
```

To spawn the quadrotor in Gazebo, we can run the following launch file:

```
roslaunch rotors_gazebo crazyflie2_without_controller.launch
```

1.3 Dynamic Model

The quadrotor model is shown in Figure 2. Considering two coordinate frames — specifically the world coordinate frame O_W and the body coordinate frame O_B – the generalized coordinates for a quadrotor model are defined as:

$$q = \begin{bmatrix} x & y & z & \phi & \theta & \psi \end{bmatrix}^T$$

with the translational coordinates x, y, z with respect to the world frame, and the roll ϕ, pitch θ and yaw ψ angles with respect to the body frame.

The control inputs on the system can be considered simply as:

$$u = \begin{bmatrix} u_1 & u_2 & u_3 & u_4 \end{bmatrix}$$

where u_1 is the force from all the propellers, and u_2, u_3, and u_4 are the moments applied about the body frame axes by the propellers.

For a set of desired control inputs, the desired rotor speeds (i.e. ω_i for $i = 1, 2, 3, 4$) are obtained by using the "allocation matrix":

$$\begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix} = \begin{bmatrix} \frac{1}{4k_F} & -\frac{\sqrt{2}}{4k_F l} & -\frac{\sqrt{2}}{4k_F l} & -\frac{1}{4k_M k_F} \\ \frac{1}{4k_F} & -\frac{\sqrt{2}}{4k_F l} & \frac{\sqrt{2}}{4k_F l} & \frac{1}{4k_M k_F} \\ \frac{1}{4k_F} & \frac{\sqrt{2}}{4k_F l} & \frac{\sqrt{2}}{4k_F l} & -\frac{1}{4k_M k_F} \\ \frac{1}{4k_F} & \frac{\sqrt{2}}{4k_F l} & -\frac{\sqrt{2}}{4k_F l} & \frac{1}{4k_M k_F} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix}$$

where k_F and k_M denote the propeller thrust factor and moment factor, respectively.

Considering the generalized coordinates and the control inputs defined above, the simplified equations of motion (assuming small angles) for the translational accelerations and body frame angular accelerations are derived as:

$$\begin{aligned} \ddot{x} &= \frac{1}{m}\big(\cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi\big)u_1 \\ \ddot{y} &= \frac{1}{m}\big(\cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi\big)u_1 \\ \ddot{z} &= \frac{1}{m}\big(\cos\phi\cos\theta\big)\,u_1 - g \\ \ddot{\phi} &= \dot{\theta}\,\dot{\psi}\,\frac{I_y - I_z}{I_x} - \frac{I_p}{I_x}\,\Omega\,\dot{\theta} + \frac{1}{I_x}\,u_2 \\ \ddot{\theta} &= \dot{\phi}\,\dot{\psi}\,\frac{I_z - I_x}{I_y} + \frac{I_p}{I_y}\,\Omega\,\dot{\phi} + \frac{1}{I_y}\,u_3 \\ \ddot{\psi} &= \dot{\phi}\,\dot{\theta}\,\frac{I_x - I_y}{I_z} + \frac{1}{I_z}\,u_4 \end{aligned}$$

where m is the quadrotor mass, g is the gravitational acceleration, I_p is the propeller moment of inertia, and I_x, I_y, I_z indicate the quadrotor moment of inertia along the x, y and z axes, respectively. Moreover, the term Ω is expressed as: $\Omega = \omega_1 - \omega_2 + \omega_3 - \omega_4$.

The physical parameters for the Crazyflie 2.0 hardware are listed in Table 1.

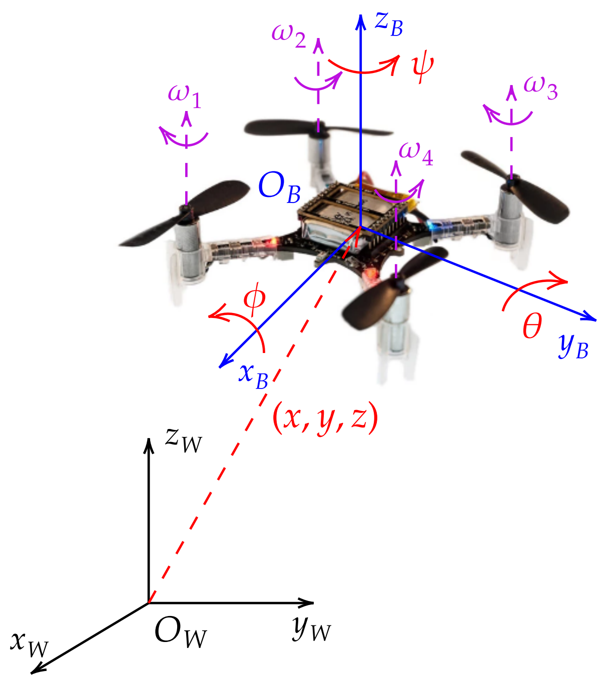


Figure 2: Crazyflie in the body (O_B) and the world (O_W) coordinate frames. The angular velocity ω_i of each rotor/propeller and the generalized coordinates x, y, z, ϕ, θ, ψ are depicted.

Table 1: Physical parameters of the Crazyflie 2.0 hardware platform.

Parameter	*Symbol*	*Value*
Quadrotor mass	m	27 g
Quadrotor arm length	l	46 mm
Quadrotor inertia along x-axis	I_x	16.571710×10^{-6} kg · m^2
Quadrotor inertia along y-axis	I_y	16.571710×10^{-6} kg · m^2
Quadrotor inertia along z-axis	I_z	29.261652×10^{-6} kg · m^2
Propeller moment of inertia	I_p	12.65625×10^{-8} kg · m^2
Propeller thrust factor	k_F	1.28192×10^{-8} N · s^2
Propeller moment factor	k_M	5.964552×10^{-3} m
Rotor maximum speed	ω_{max}	2618 rad/s
Rotor minimum speed	ω_{min}	0 rad/s

Remark 1: As shown in the equations of motion above, the quadrotor system has six DoF, with only four control inputs. As a result, the control of quadrotors is typically done by controlling only the altitude z and the roll-pitch-yaw angles ϕ, θ and ψ.

1.4 Problem Statement

Design a sliding mode controller for *altitude* and *attitude* control of the Crazyflie 2.0 to enable the quadrotor to track desired trajectories and visit a set of desired waypoints.

The main components of the project are described below.

Part 1. Write a MATLAB or Python script to generate quintic (fifth-order) trajectories (position, velocity and acceleration) for the translational coordinates (x, y, z) of Crazyflie. The quadrotor is supposed to start from the origin $p_0 = (0, 0, 0)$ and visit five waypoints in sequence. The waypoints to visit are

- $p_0 = (0, 0, 0)$ to $p_1 = (0, 0, 1)$ in 5 seconds
- $p_1 = (0, 0, 1)$ to $p_2 = (1, 0, 1)$ in 15 seconds
- $p_2 = (1, 0, 1)$ to $p_3 = (1, 1, 1)$ in 15 seconds
- $p_3 = (1, 1, 1)$ to $p_4 = (0, 1, 1)$ in 15 seconds
- $p_4 = (0, 1, 1)$ to $p_5 = (0, 0, 1)$ in 15 seconds

The sequence of visiting the waypoints does matter. The velocity and acceleration at each waypoint must be equal to zero.

Include a plot of the desired trajectories in your final report.

Part 2. Considering the equations of motion provided in Section 1.3, design boundary layer-based sliding mode control laws for the z, ϕ, θ, ψ coordinates of the quadrotor to track desired trajectories z_d, ϕ_d, θ_d, and ψ_d. Include the control formulations (as symbolic expressions) in your final report.

Remark 2: To convert the desired position trajectories (x_d, y_d, z_d) to desired roll and pitch angles (ϕ_d, θ_d), the desired forces in x and y direction can be calculated using PD control (according to Eq. (1) and (2)), and the resulting desired forces can be then converted to desired ϕ and θ according to Eq. (3) and Eq. (4):

$$F_x = m\big(-k_p\,(x - x_d) - k_d\,(\dot{x} - \dot{x}_d) + \ddot{x}_d\big), \tag{1}$$

$$F_y = m\big(-k_p\,(y - y_d) - k_d\,(\dot{y} - \dot{y}_d) + \ddot{y}_d\big), \tag{2}$$

$$\theta_d = \sin^{-1}\left(\frac{F_x}{u_1}\right) \tag{3}$$

$$\phi_d = \sin^{-1}\left(\frac{-F_y}{u_1}\right) \tag{4}$$

Remark 3: For the purpose of this assignment, the desired yaw angle ψ, and also the desired angular velocities $\dot{\phi}$, $\dot{\theta}$, $\dot{\psi}$ and the desired angular accelerations $\ddot{\phi}$, $\ddot{\theta}$, $\ddot{\psi}$ can be considered zero during the motion, i.e:

$$\psi_d = 0 \qquad \text{and} \qquad \dot{\phi}_d = \dot{\theta}_d = \dot{\psi}_d = 0 \qquad \text{and} \qquad \ddot{\phi}_d = \ddot{\theta}_d = \ddot{\psi}_d = 0$$

The resulting discrepancy can be considered as an *external disturbance* that is handled through the robust control design in this assignment.

Remark 4: When designing the sliding mode control laws, assume that all the model parameters are known. In fact, the objective of this assignment is to design a sliding mode controller to be robust under reasonable external disturbances, as discussed in class.

Part 3. Implement a ROS node in Python or MATLAB to evaluate the performance of the control design on the Crazyflie 2.0 quadrotor in Gazebo. You can create a new ROS package named `project` under the project workspace for this purpose. The script must implement the trajectories generated in Part 1 and the sliding mode control laws formulated in Part 2.

A Python sample code is provided as the starting point for your implementation. Please carefully review the provided code and understand its functionality. Feel free to define new functions and variables in your program if needed.
If using MATLAB ROS Toolbox, you will need to re-write the provided Python codes in MATLAB.

```python
#!/usr/bin/env python3
from math import pi, sqrt, atan2, cos, sin
from turtle import position
import numpy as np
from numpy import NaN
import rospy
import tf
from std_msgs.msg import Empty, Float32
from nav_msgs.msg import Odometry
from mav_msgs.msg import Actuators
from geometry_msgs.msg import Twist, Pose2D
import pickle
import os

class Quadrotor():
    def __init__(self):
        # publisher for rotor speeds
        self.motor_speed_pub = rospy.Publisher("/crazyflie2/command/
           motor_speed", Actuators, queue_size=10)

        # subscribe to Odometry topic
        self.odom_sub = rospy.Subscriber("/crazyflie2/ground_truth/odometry",
           Odometry, self.odom_callback)

        self.t0 = None
        self.t = None
        self.t_series = []
        self.x_series = []
        self.y_series = []
        self.z_series = []
        self.mutex_lock_on = False
        rospy.on_shutdown(self.save_data)

        # TODO: include initialization codes if needed

    def traj_evaluate(self):
        # TODO: evaluating the corresponding trajectories designed in Part 1
           to return the desired positions, velocities and accelerations

    def smc_control(self, xyz, xyz_dot, rpy, rpy_dot):
        # obtain the desired values by evaluating the corresponding
           trajectories
        self.traj_evaluate()
        # TODO: implement the Sliding Mode Control laws designed in Part 2 to
            calculate the control inputs "u"

        # REMARK: wrap the roll-pitch-yaw angle errors to [-pi to pi]

        # TODO: convert the desired control inputs "u" to desired rotor
            velocities "motor_vel" by using the "allocation matrix"

        # TODO: maintain the rotor velocities within the valid range of [0 to
            2618]

        # publish the motor velocities to the associated ROS topic
        motor_speed = Actuators()
        motor_speed.angular_velocities = [motor_vel[0,0], motor_vel[1,0],
            motor_vel[2,0], motor_vel[3,0]]
        self.motor_speed_pub.publish(motor_speed)

    # odometry callback function (DO NOT MODIFY)
    def odom_callback(self, msg):
        if self.t0 == None:
            self.t0 = msg.header.stamp.to_sec()
        self.t = msg.header.stamp.to_sec() - self.t0

        # convert odometry data to xyz, xyz_dot, rpy, and rpy_dot
        w_b = np.asarray([[msg.twist.twist.angular.x], [msg.twist.twist.
            angular.y], [msg.twist.twist.angular.z]])
        v_b = np.asarray([[msg.twist.twist.linear.x], [msg.twist.twist.linear.
            y], [msg.twist.twist.linear.z]])
        xyz = np.asarray([[msg.pose.pose.position.x], [msg.pose.pose.position.
            y], [msg.pose.pose.position.z]])
        q = msg.pose.pose.orientation
        T = tf.transformations.quaternion_matrix([q.x, q.y, q.z, q.w])
        T[0:3, 3] = xyz[0:3, 0]
        R = T[0:3, 0:3]
        xyz_dot = np.dot(R, v_b)
        rpy = tf.transformations.euler_from_matrix(R, 'sxyz')
        rpy_dot = np.dot(np.asarray([
            [1, np.sin(rpy[0])*np.tan(rpy[1]), np.cos(rpy[0])*np.tan(rpy[1])],
            [0, np.cos(rpy[0]), -np.sin(rpy[0])],
            [0, np.sin(rpy[0])/np.cos(rpy[1]), np.cos(rpy[0])/np.cos(rpy[1])]
            ]), w_b)
        rpy = np.expand_dims(rpy, axis=1)

        # store the actual trajectory to be visualized later
        if (self.mutex_lock_on is not True):
            self.t_series.append(self.t)
            self.x_series.append(xyz[0, 0])
            self.y_series.append(xyz[1, 0])
            self.z_series.append(xyz[2, 0])

        # call the controller with the current states
        self.smc_control(xyz, xyz_dot, rpy, rpy_dot)
    # save the actual trajectory data
    def save_data(self):
        # TODO: update the path below with the correct path
        with open("/home/sfarzan/rbe502_project/src/project/scripts/log.pkl",
            "wb") as fp:
            self.mutex_lock_on = True
            pickle.dump([self.t_series,self.x_series,self.y_series,self.
                z_series], fp)

if __name__ == '__main__':
    rospy.init_node("quadrotor_control")
    rospy.loginfo("Press Ctrl + C to terminate")
    whatever = Quadrotor()
    try:
        rospy.spin()
    except KeyboardInterrupt:
        rospy.loginfo("Shutting down")
```

Part 4. Once the program is shut down, the actual trajectory is saved into a `log.pkl` file under the `scripts` directory. We provide a separate Python script to help visualize the trajectory from the saved `log.pkl` file. Copy and paste the following code into a new script named `visualize.py`:

```
#!/usr/bin/env python3
import matplotlib.pyplot as plt
import pickle
from mpl_toolkits.mplot3d import Axes3D

def visualization(x_series, y_series, z_series):
    # load csv file and plot trajectory
    fig = plt.figure()
    ax = plt.axes(projection='3d')

    # Data for a three-dimensional line
    ax.plot3D(x_series, y_series, z_series, 'blue')
    ax.plot3D([0, 0, 1, 1, 0, 0], [0, 0, 0, 1, 1, 0], [0, 1, 1, 1, 1, 1], '
        green')
    plt.xlim(-0.5, 1.5)
    plt.ylim(-0.5, 1.5)
    plt.minorticks_on()
    plt.grid(which='both')
    plt.xlabel('x (m)')
    plt.ylabel('y (m)')
    plt.savefig('trajectory.png', dpi = 300)
    plt.show()

if __name__ == '__main__':
    file = open("log.pkl",'rb')
    t_series, x_series, y_series, z_series = pickle.load(file)
    file.close()
    visualization(x_series, y_series, z_series)
```

You can now execute the `visualize.py` script to generate a plot of the actual vs. desired trajectories. Remember to save the resulting plot as a figure to be included in your final report.

1.5 Performance Testing in Gazebo

i) Open a terminal in Ubuntu, and spawn the Crazyflie 2.0 quadrotor on the Gazebo simulator :

```
roslaunch rotors_gazebo crazyflie2_without_controller.launch
```

Note that the Gazebo environment starts in the *paused* mode, so make sure that you start the simulation by clicking on the play button before proceed.

ii) We can now test the control script developed in Part 3 by running the script in a new terminal. The quadrotor must be controlled smoothly (no overshoot or oscillations) to track the trajectories generated in Part 1 and reach the five desired waypoints.

iii) While the control script is running, make sure to record your Gazebo simulation using a screen recorder software to be submitted along with your report.

iv) After the simulation is over, run the visualization script to plot the actual trajectories on top of the reference trajectories in 3D to ensure a satisfactory control performance. Remember to save the resulting plot as a figure to be included in your report.

1.6 Submission

- Submission: group submission via Gradescope. Submit the Python and/or MATLAB files you have written as a zip folder, a video of the performance of your program, and a report. In your report, please include:
  - a brief discussion on trajectory generation, along with the plots of the reference trajectories generated in Part 1;
  - complete derivation of the sliding mode control laws formulated in Part 2 (correct control laws, unsupported by calculations, derivations and explanations will receive no credit);
  - please do not include screenshots of MATLAB or Python codes, instead, write down mathematical formulations similar to a research paper;
  - a list of all design and tuning parameters used in the control values, with sufficient discussion on the effect of each of the design parameters;
  - an explanation of the code developed in Part 3;
  - 3D plot(s) of the actual trajectory over the reference trajectory, along with sufficient discussion on the performance of the controller.
- Due: as specified on Canvas.
- Files to submit: (please use the exact file name, and do NOT include the PDF file in the zip folder)
  - `lastname1_lastname2_project_report.pdf`
  - `lastname1_lastname2_project_codes.zip`
  - `lastname1_lastname2_project_video.mov` (or `.mp4` or `.mpg`)