Control of Underwater Robots:

Implementation of an autonomous behavior on the BlueROV

Abhimanyu Bhowmik
Sergei Chashnikov
Madhushree Sannigrahi
Tayyab Tahir
Ishfaq Bhat

Under the supervision of:
Associate Prof. Cédric Anthierens and
Vincent Hugel (COSMER, University of Toulon).

Marine Mechatronics

1 Introduction

This project aims to create autonomous behaviour for the BlueROV underwater vehicle. The primary objectives of this project include maintaining a fixed heading while moving forward, detecting obstacles, halting at a safe distance from said obstacles, and subsequently navigating around them to continue forward motion.

Key components of this project include the trajectory-following PID controller for depth control, the P controller for heading and yaw control, and the Ping Sonar Echosounder for obstacle detection. The sensors used include an Inertial Measurement Unit (IMU) for obtaining yaw angle measurements and a frontal echosounder pinger for determining the distance to obstacles in the forward direction of the BlueROV. The obstacles comprise the tank wall, specific vertical wooden boards strategically placed within the tank environment, and other BlueROVs in the tank. This practical lab serves as an exploration of marine mechatronics, where we'll integrate theory with hands-on experience to develop autonomous capabilities in underwater vehicles.

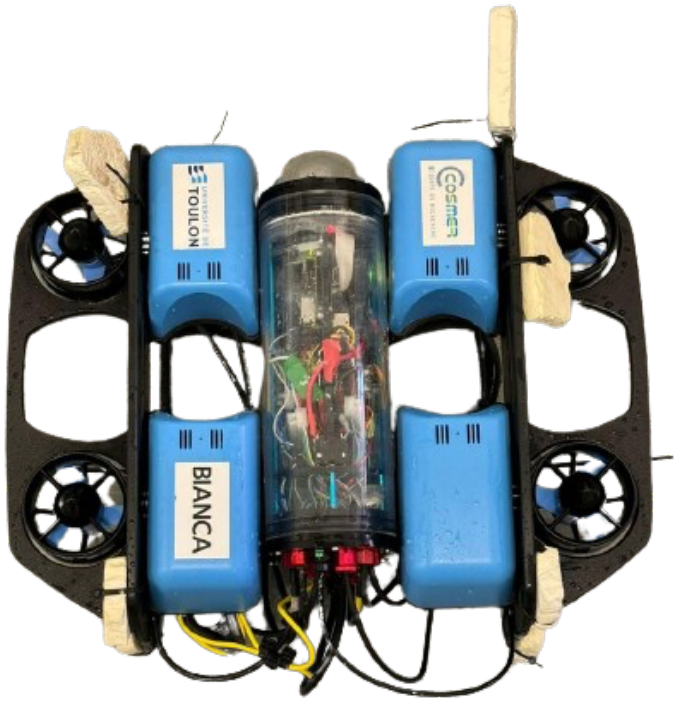

Figure 1: BlueROV BIANCA from COSMER lab, UTLN

2 Methodology:

The Python code was provided by the professor with callbacks for IMU and echosounder pinger, together with the conversion code for quaternions into Euler angles, to facilitate a smooth integration of the data. Our task was to implement controllers for depth and heading control, as well as obstacle detection and avoidance algorithms. The entire code is executed as Finite State Machine, specifying particular events as a change of space. The code structure was complex, but by using the Finite State Machine paradigm, we were able to break the overall behaviour of the robot into **Descent, Forward, Stop**, and **Search Left** states. The corresponding state machine diagram is given in Figure 2.

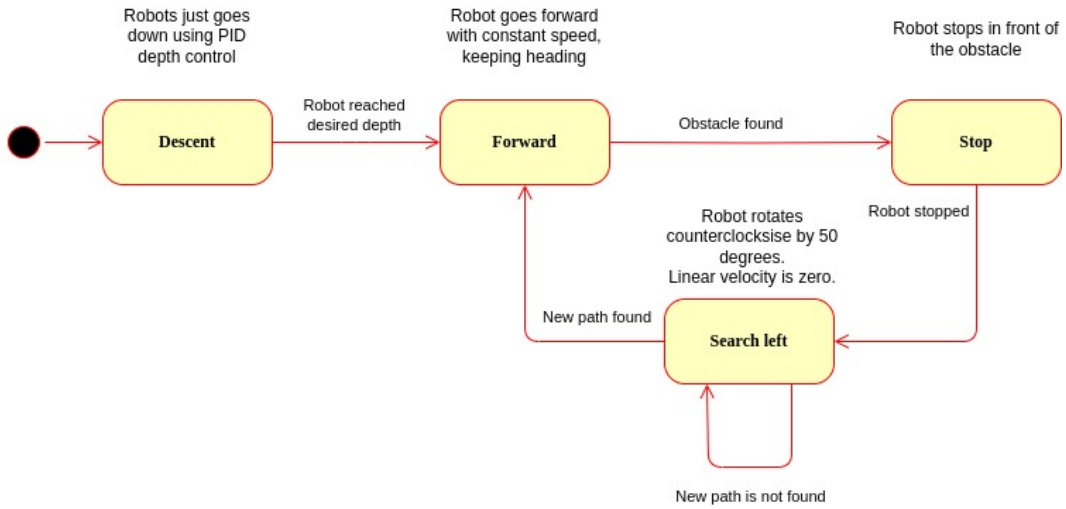

Figure 2: Overall algorithm of the system as Finite State Machine.

3 Description of the states and implementation

3.1 States and initial position

The current state of the State Machine is saved in `state` variable. The initial value of the variable, according to the state diagram (Figure 2) is **"descent"**.

```
# State of the robot
state = "descent"
print("Initial state: ", state)
```

Then, when a corresponding event happens, this variable changes to other states. It will be shown in more detail in sections about particular states (3.2 - 3.5). At the beginning, BlueRov is located on the surface of the water.

3.2 Descent

When the robot enters this state, it starts going down to the desired depth. In our case, we set the depth to 0.5 meters. We implemented a PID controller to reach a desired depth

along a fixed trajectory and maintain it. The control output (τ_{PID}) is computed by incorporating proportional, integral, and derivative terms, alongside floatability compensation

$$\tau_{\text{PID}} = K_p\tilde{z} + K_i \int \tilde{z}(t)dt + K_d\dot{\tilde{z}} + \text{floatability compensation}$$

Here,
$\tilde{z} = z_{\text{des}} - z$ represents the depth error, and
$\dot{\tilde{z}} = \dot{z}_{\text{desired}} - \dot{z}$ represents the error in heave speed.
K_p, K_i, and K_d are the respective proportional, integral, and derivative gains.

After adjusting, we used the following values for gains: $K_p = 120, K_i = 10, K_d = 2$.

The integral term accumulates error over time to handle steady-state errors, while the derivative term accounts for the error's rate of change, enhancing quick response and stability.

Since we don't have sensors on the ROV to measure the speed vertical speed (heave) of the robot, we implemented the alpha-beta filter (simpler version of the Kalman filter) to compute the speed.

$$x_k = x_{k-1} + (v_{k-1} \cdot dt)$$

$$r_k = x_m - x_k$$

$$x_k += \alpha \cdot r_k$$

$$v_k += \frac{\beta \cdot r_k}{dt}$$

By adjusting the values of the α, β and updating the values of x_k, r_k, we can find the values of heave speed. In our implementation we used default values: $\alpha = 0.1, \beta = 0.005$.

Depth control is implemented in the `PressureCallback` function. When this function is called (new data from the pressure sensor is received), the following steps are performed:

- We compute what time of the trajectory movement is now (variable `current_t`)

```
    # Motion according to the trajectory
    a2 = 3*(z_final - 0)/t_final_int**2
    a3 = -2*(z_final - 0)/t_final_int**3
    current_t = rospy.Time.now() - t_init

    current_sampling = rospy.Time.now()
    sampeling_period = (current_sampling - t_previous)
    sampeling_period = sampeling_period.to_sec()
    t_previous = current_sampling

```

- We compute desired depth with respect to the trajectory and to the current time (variable `z_des`)

```
    if current_t < t_final:
        z_des = 0 + a2*current_t.to_sec()**2 + a3*current_t.to_sec()**3
        z_dot_des = 0 + 2*a2*current_t.to_sec() + 3*a3*current_t.to_sec()**2
    elif current_t >= t_final:
        z_des = z_final
        z_dot_des = 0

```

- We compute estimated vertical speed (variable `z_dot`)

```
    #alpha-beta filter
    z_eval = z_prev + sampeling_period * vel_prev
    r = depth_wrt_startup - z_eval
    z_eval = z_eval + alpha*r

    vel_eval = vel_prev + beta*r/sampeling_period
    vel_prev = vel_eval

    z_dot = vel_eval
    z_prev = z_eval

```

- We compute error for proportional and integral part(variables `error` and `sum_error_depth`)

```
    error = (z_des-depth_wrt_startup)
    sum_error_depth = sum_error_depth + (error * sampeling_period)

```

- We compute the vertical thrust of the robot using PID controller (variable `f`)

```
    #PID Controller
    f = (kp * error + ki * sum_error_depth + floatibility + kd * (z_dot_des -
                                        z_dot))/ num_thrusters

```

- If ROV is descending, we check if the planned time for the descend has ended and the desired depth is reached with the provided threshold (we used value 0.05 m). If both conditions are met, it means that BlueRov has ended descent and state changes to ”forward”.

```
    # Change state if the robot is descending and reached the desired depth
    current_t = rospy.Time.now() - t_init
    if current_t >= t_final:
        if state == "descent" and error <= error_threshold:
            state = "forward"
            print("state: forward")

```

- At the end of the callback function, we compute the PWM value for the given thrust and send it to the low-level control function.

```
    # update Correction_depth
    Correction_depth = computePWM(f)
    setOverrideRCIN(1500, 1500, Correction_depth, Correction_yaw,
                                        Correction_surge, 1500)

```

In our program, we have three callback functions working simultaneously (`PressureCallback`, `OdoCallback`, `pingerCallback`) and they all use one function (`setOverrideRCIN`) to send PWM commands. To synchronise all commands we use global variables (`Correction_depth`, `Correction_yaw`, `Correction_surge`) that are translated through these three functions. For example, the value of `Correction_depth`, calculated in `PressureCallback`, then used in `OdoCallback` and `pingerCallback` functions.

```
    global Correction_depth
    global Correction_yaw
    global Correction_surge

```

3.3 Forward

In the **Forward** state robot keeps the constant surge speed and constant heading. Also, we control the distance to the obstacle in front of the robot.

3.3.1 Forward speed

Constant speed for the surge is set in `pingerCallback` function. We used value 0.00001 for `forward_speed` variable.

```
    f = forward_speed
    Correction_surge = computePWM(f*num_thrusters)
    setOverrideRCIN(1500, 1500, Correction_depth, Correction_yaw, Correction_surge
                                            , 1500)

```

3.3.2 Yaw control

Constant heading is implemented in `OdoCallback` function. When this function is called (new data from IMU is received), following steps are done:

- We start by converting the orientation feedback of the robot's IMU (in quaternions) into Euler angles. We get our roll, pitch and yaw angles from here.

```
    # extraction of yaw angle
    q = [orientation.x, orientation.y, orientation.z, orientation.w]
    euler = tf.transformations.euler_from_quaternion(q)
    angle_roll = euler[0]
    angle_pitch = euler[1]
    angle_yaw = euler[2]

```

- The initial yaw angle is saved when the ROV is turned to autonomous mode. After that, for each callback, we check the difference between the initial_yaw and current_yaw angle.

```
    if (init_a0):
        # at 1st execution, init
        angle_roll_a0 = angle_roll
        angle_pitch_a0 = angle_pitch
        angle_yaw_a0 = angle_yaw
        angle_yaw_final = 0
        init_a0 = False
        sum_error_yaw = 0

    angle_wrt_startup[0] = ((angle_roll - angle_roll_a0 + 3.0*math.pi)%(2.0*
                                    math.pi) - math.pi) * 180/math.pi
    angle_wrt_startup[1] = ((angle_pitch - angle_pitch_a0 + 3.0*math.pi)%(2.0
                                    *math.pi) - math.pi) * 180/math.pi
    angle_wrt_startup[2] = ((angle_yaw - angle_yaw_a0 + 3.0*math.pi)%(2.0*
                                    math.pi) - math.pi) * 180/math.pi
    angle_yaw_current = angle_wrt_startup[2]

```

- We use a P-controller to minimize this error and adjust the yaw angle of the ROV accordingly. This ensures a steady heading angle when the ROV is in the **"forward"** state and helps maintain stability during navigation.

```
    # setup depth servo control here sum_error_yaw
    error = (angle_yaw_des - angle_yaw_current)
    error = computeYawError(error)
    f = (Kp * error) / num_thrusters

    Correction_yaw = computePWM(f)
    setOverrideRCIN(1500, 1500, Correction_depth, Correction_yaw,
                                    Correction_surge, 1500)

```

The yaw angle, that is, obtained from IMU measurements, is in range $[0, 360]$ degrees. This can lead to the following problems:

- Inadequate control value when the robot is stabilising around zero-degree angle.
- Overly large rotations. For example, when robot should rotate from 10° to 270°. From common sense it is obvious, that the correct way to fulfill this rotation is to go 100° counter-clockwise, but not 260° clockwise.

To challenge these issues we implemented `computeYawError` function:

```
    def computeYawError(error):
        if abs(error) > 180:
            sign = 1 if error >= 0 else -1
            error = - sign * (360 - abs(error))

        return error
```

3.3.3 Distance control

Distance is measured by an ultrasound sensor mounted on the robot. When the distance is lower than the given threshold (variable `desired_distance`), the state changes to **"stop"**. Control of the distance is implemented in `pingerCallback` function:

```
    elif state == "forward":
    if pinger_distance <= desired_distance:
        state = "stop"
        print("state: stop")
        f = 0
        sum_error_distance = 0
        time_pinger_prev = rospy.Time.now()
        dist_prev = pinger_distance
        vel_prev = 0
    else:
        f = forward_speed
```

For practical implementation, to attain a distance of 70 cm, we had to realistically set the `desired_distance` at 1.2 m, accounting for the BlueROV's forward drift caused by inertia. A distance of less than 50cm is a "dead zone" for the pinger, beyond which it produces absurd results. If the ROV is stopped precisely at 70cm from the obstruction, it will inevitably drift forward and crash into the obstacle or the pool walls.

3.4 Stop

The implementation of this state is done in `OdoCallback` function. The robot just switches to the **"search_left"** state and saves the moment of time corresponding to the start of the turn. The reason for this will be explained further (3.5.2).

```
    elif state == "stop":
        f = 0
        state = "search_left"
        print("state: search_left")
        start_search = True
        t_search_left_start = rospy.Time.now()
```

3.5 Search left

In this state robot turns counter-clockwise by 50° and after this check is there still an obstacle in front of him.

3.5.1 Rotation

Turn by 50° is implemented in `OdoCallback` function.

```
    if state == "search_left":
    if start_search:
        angle_yaw_des = angle_yaw_des + 50
        start_search = False
```

After this robot uses K-controller (3.3.2) to set a new yaw angle.

3.5.2 Exploration of the free path

This feature is done in `pingerCallback` function. The robot checks that turn started 10 seconds ago and then checks the presence of the obstacle. If there is no obstacle, it starts moving forward (**"forward"**), otherwise, it turns again by 50°. 10 10-second pause is added to prevent obtaining inadequate measurements from the distance sensor.

```
    elif state == "search_left":
        # Check that 10 seconds have passed
        time_passed = rospy.Time.now() - t_search_left_start
        if time_passed.to_sec() >= 10:
            if pinger_distance > desired_distance:
                state = "forward"
                print("state: forward")
                init_a0 = True
                f = forward_speed
            else:
                f = 0
                state = "search_left"
                print("state: search_left")
                start_search = True
                t_search_left_start = rospy.Time.now()
```

4 Results:

4.1 Depth Control using PID:

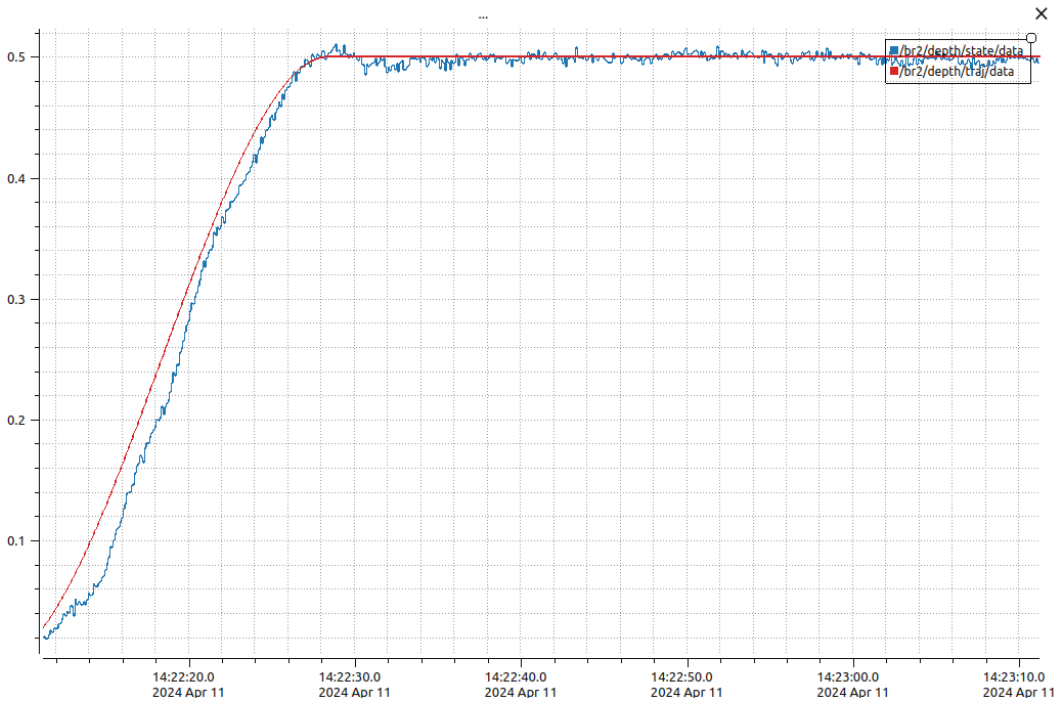

Figure 3: Depth along with the Trajectory

We achieved good heave control using a PID controller along a fixed trajectory. We got our best results with K_p = 120, K_i = 10, and K_d = 2, maintaining a constant depth of 0.5m. These parameters allowed us to minimize overshoot and achieve a smooth response during our testing. The depth pressure graph of BlueROV along with the trajectory is presented in Figure 3.

4.2 Acoustic servoing in the Surface:

After heave, we attempted to implement the acoustic servoing on the surface. The objective was to detect the walls of the pool or any other obstacle in front of the ROV and stop the ROV at an approximately 70cm distance. However, due to inertial drift, we kept the desired distance at 120cm and the thrusters were kept at its lowest power. A constant oscillation in yaw is observed as the ROV tries to maintain the correct heading angle. For the heading control, a simple P-controller was implemented using K_p = 0.000001. The

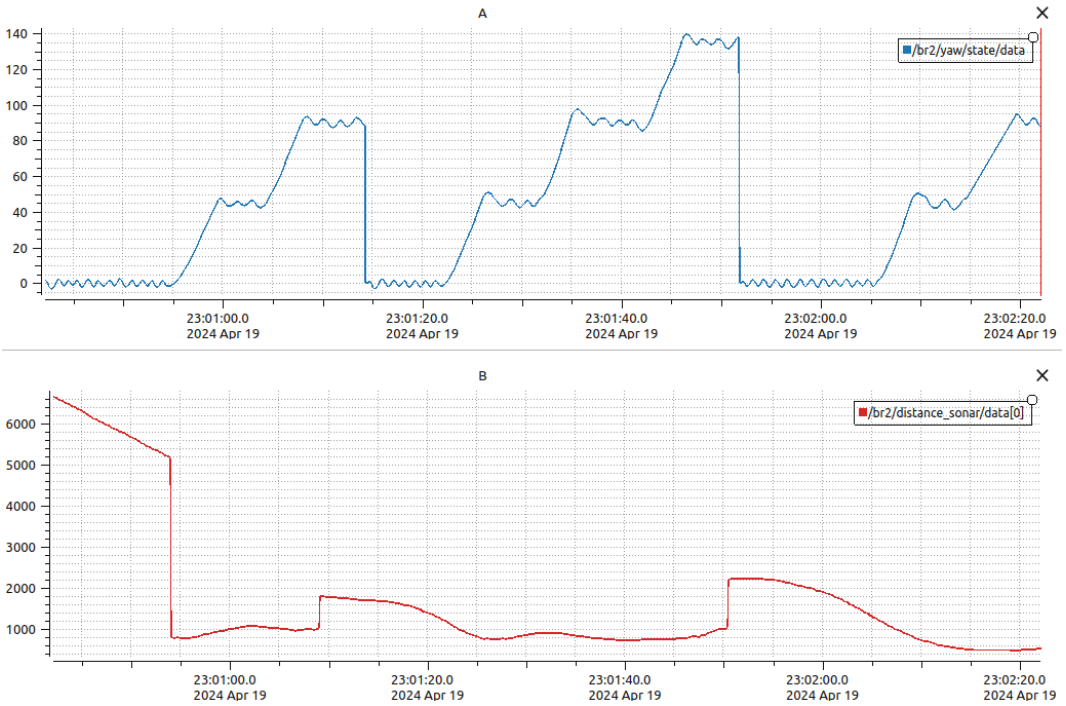

Figure 4: (A) The heading angle of the ROV (B) Distance of the obstacle from the ROV

graph portraying the distance from the obstacle and the yaw angle of the ROV is given in Figure 4.

4.3 Servoing with depth control:

We also managed to merge the heave and heading control on the ROV. Here, 2 different PIDs were working together. There was some drift in the heading when the state changed from ”Forward” to ”Stop” due to inertia. The ROV overall succeeded in acoustics servoing

and obstacle avoidance while maintaining a constant depth. The depth pressure graph along with yaw control is portrayed in Figure 5.

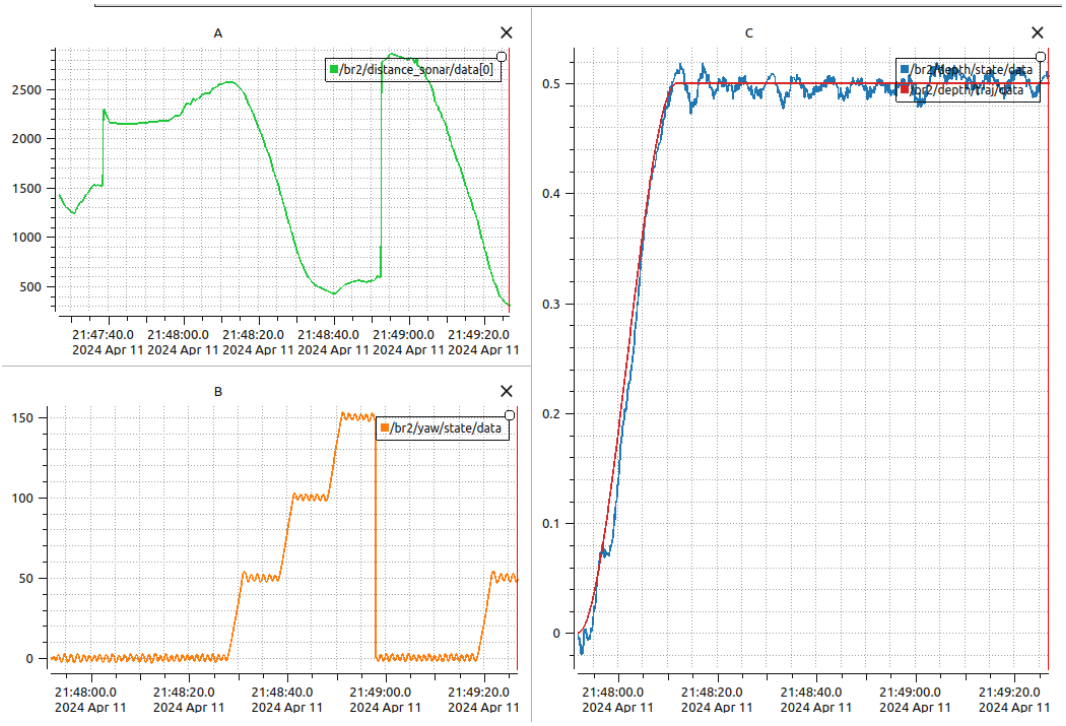

Figure 5: (A) Distance measured by pinger. (b) Yaw angle from IMU sensor. (C) Depth of BlueROV with trajectory.

The bag files, code and video are attached with the mail.