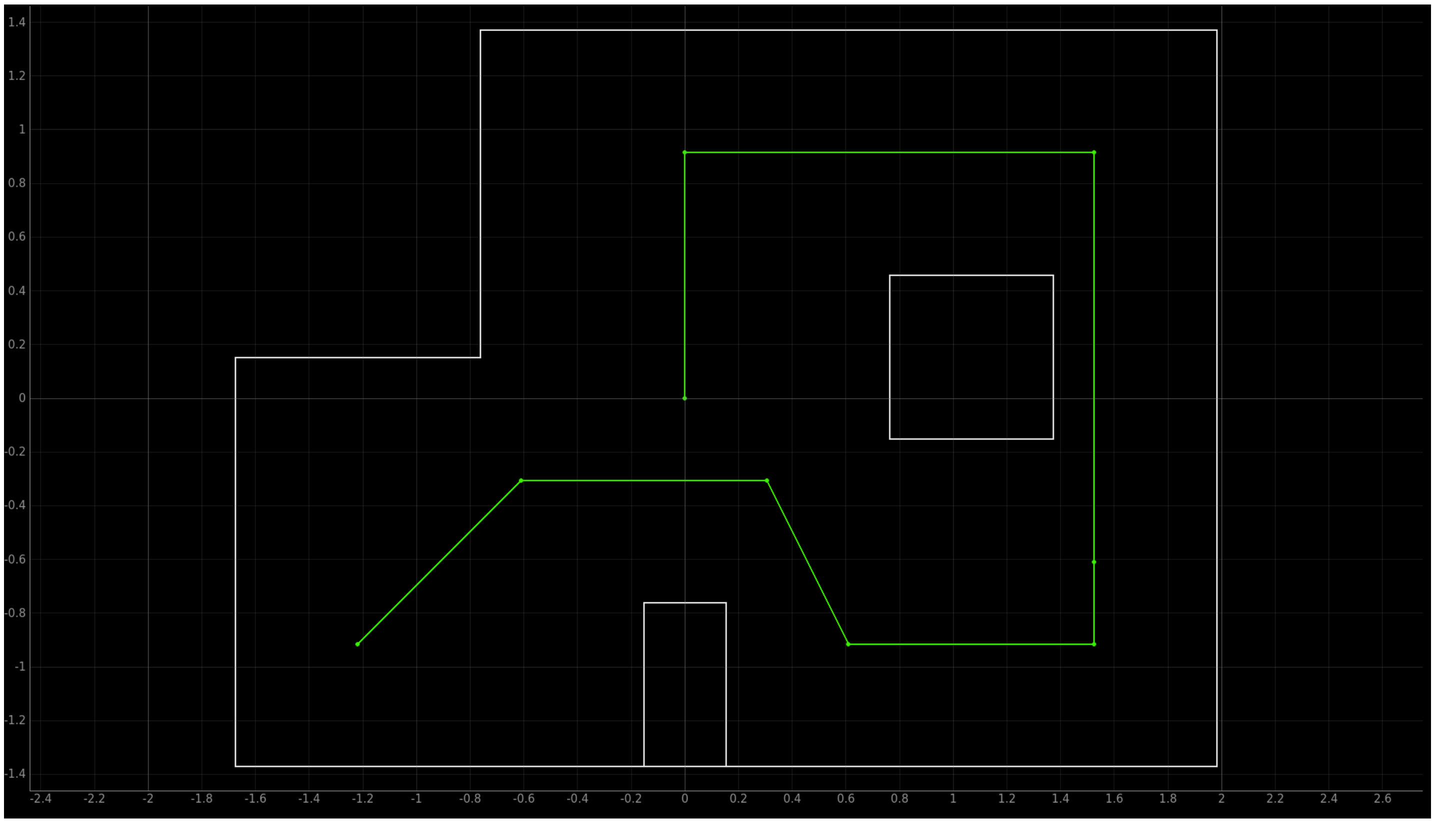
ECE4160 Lab 12

Akshati Vaishnav

Overview

The purpose of this lab was to combine everything we've learned from this class about PID, mapping, navigation, etc. and program our robot to traverse a map. The map along with the recommended points for the robot to hit are shown below:



```
Waypoints:
1. (-4, -3)    <--start
2. (-2, -1)
3. (1, -1)
4. (2, -3)
5. (5, -3)
6. (5, -2)
7. (5, 3)
8. (0, 3)
9. (0, 0)      <--end
```

My Approach

In order to traverse a map, I performed the following tasks:

- Perform angular PID at every step to ensure that the robot faces the right direction to move in
- Perform open-loop distance control at the first three line segments
- Perform distance PID for the last four line segments
- Perform localization for analysis purposes only at (-4, -3), (5, 3), and (0, 0)

Reasoning

1. Localization: I wanted to traverse the map initially by performing localization at every step and using that to determine the distance to traverse and the angle to turn (taking the difference between current belief and next expected belief). However, I realized that the belief was consistently different in terms of both distance and angle. So, localizing back-to-back at various points resulted in significant errors because the robot sometimes believed at it was almost 2 meters away than where it actually was. Even if localization works decently well on 6-7 points, if it fails on the remaining, it can cause the robot to not hit the waypoints. Instead, I decided to perform localization for analytics as the robot traverses the map. I collected localization data at (-4, -3), (5, 3), and (0, 0), and hitting these points would imply that the robot did relatively well localizing and traversing the map.
2. Open loop control: For the first three paths, I initially tried performing PID control. However, the effectiveness of distance PID depended on the distance sensor readings which changed drastically based on how well the previous angular PID worked (which depended on the initial orientation of the robot that was set by me and therefore was prone to error). These first few line segments had multiple layers of walls so detecting a wall behind the actual wall to detect would cause the robot to go further than necessary. Thus, doing open-loop control was best to ensure that the results were reproduceable.
3. Distance PID: For the latter half of the map, the turns were 90 degrees, and there were fewer walls behind walls, which is why I used distance PID to control the robot.
4. Angular PID: Since we were given a suggested path to follow, I calculated the angle required to go from one point to the next. Then, I sent an angular PID command to ensure that the robot pointed in the correct direction. One important part of this was to ensure that the robot points in the right direction when uploading the code to it (or powering it). This is important for DMP initialization, since the 0° angle is always in the direction of power-up.

Code

Arduino

On the Arduino side, I wrote four commands which made the robot programming significantly easier. `GET_USER_MOVE` is a command that takes in five inputs: four PWM values for all four pins and one delay input which determines how long to spin the wheels for. This is the command which I use to perform open-loop distance control.

```
//OPEN-LOOP MOTOR CONTROL
case GET_USER_MOVE: {
    int a, b, c, d, e;

    // Extract the next value from the command string as a int
    success = robot_cmd.get_next_value(a);
    if (!success)
        return;

    // Extract the next value from the command string as a int
    success = robot_cmd.get_next_value(b);
    if (!success)
        return;

    // Extract the last value from the command string as a int
    success = robot_cmd.get_next_value(c);
    if (!success)
        return;

    success = robot_cmd.get_next_value(d);
    if (!success)
        return;

    success = robot_cmd.get_next_value(e);
    if (!success)
        return;

    float start = millis();
    while (millis() - start < e) {
        run_motors(a, b, c, d, e);
    }
    run_motors(0, 0, 0, 0, 0);
    break;
}
```

In order to perform distance PID, I used my `DIST_PID` command that takes in the PID parameters K_p, K_i, K_d as inputs, along with the final distance I want the robot to be from the wall. As I did in Lab 5, I re-scale my PWM values such that the final range is from `min_pwm` to `max_pwm` (which are values determined by the robot's circuitry and battery levels) rather than from 0 to 255.

```
//DISTANCE PID
case DIST_PID: {
    float exp_dist;
    //get Kp, Ki, Kd values from user input
    success = robot_cmd.get_next_value(Kp);
    if (!success)
        return;

    success = robot_cmd.get_next_value(Ki);
    if (!success)
        return;

    success = robot_cmd.get_next_value(Kd);
    if (!success)
        return;

    success = robot_cmd.get_next_value(exp_dist);
    if (!success)
        return;

    distanceSensor.startRanging();
    float distance_tof = 0.0;
    double initialYaw = 0;
    double curr_ang;
    float error = 0, prev_error = 0, errorSum = 0;
    float dt = 0;
    float pwm = 0, Pterm, Iterm, Dterm;
    int max_pwm = 130;
    int min_pwm = 50;
    success = robot_cmd.get_next_value(max_pwm);
    if (!success)
        return;
    success = robot_cmd.get_next_value(min_pwm);
    if (!success)
        return;
    int pwm_range = max_pwm - min_pwm;
    float last_time = millis();
    float start_time = millis();


    while (1) {
        //get distance
        if (distanceSensor.checkForDataReady()) {
            distance_tof = distanceSensor.getDistance();
            distanceSensor.clearInterrupt();
            distanceSensor.stopRanging();
            distanceSensor.startRanging();
            Serial.println(distance_tof);
        }

        //PID control
        dt = (millis() - last_time) / 1000.0;
        last_time = millis();
        error = distance_tof - exp_dist;

        errorSum += error * dt;

        Pterm = Kp * error;
        Iterm = Ki * errorSum;
        Dterm = Kd * (error - prev_error) / dt;
        pwm = Pterm + Iterm + Dterm;

        //Cap PWM
        if(pwm>255) {
            pwm = 255;
        }
        if(pwm<-255) {
            pwm = -255;
        }

        prev_error = error;

        if (abs(error) <= 0.5 || millis()-start_time>15000) {
            // Stop motors
            analogWrite(PIN0, 0); analogWrite(PIN1, 0);
            analogWrite(PIN3, 0); analogWrite(PIN2, 0);

            break;
        }

        // Determine motor direction
        bool go_back = (pwm < 0);
        float abs_pwm = fabs(pwm);

        // Scale PWM value so that PWM goes from min_pwm to max_pwm instead of 0 to 255
        int scaled_pwm = (int)((abs_pwm / 255.0f) * pwm_range) + min_pwm;

        if (go_back) {
            // MOVE BACK
            analogWrite(PIN0, 0); analogWrite(PIN1, scaled_pwm);
            analogWrite(PIN2, 0); analogWrite(PIN3, scaled_pwm);
        } else {
            // MOVE FORWARD
            analogWrite(PIN0, scaled_pwm); analogWrite(PIN1, 0);
            analogWrite(PIN2, scaled_pwm); analogWrite(PIN3, 0);
        }

        Serial.print(", ");
        Serial.println(pwm);
    }

    break;
}
```

Next, I perform angular PID using my `ANGLE_PID` function. In order to help with quicker parameter tuning, this command takes the K_p, K_i, K_d parameters as inputs. Furthermore, it takes in another input for the expected angle which I want the robot to rotate to. This command uses the IMU's DMP to perform angular control, which I also used in Lab 6.

```
//ANGULAR PID
case ANGLE_PID: {
    int exp_ang;
    //get Kp, Ki, Kd values from user input
    success = robot_cmd.get_next_value(Kp);
    if (!success)
        return;

    success = robot_cmd.get_next_value(Ki);
    if (!success)
        return;

    success = robot_cmd.get_next_value(Kd);
    if (!success)
        return;
    success = robot_cmd.get_next_value(exp_ang);
    if (!success)
        return;

    get_dmp_data();
    double initialYaw = 0;
    double curr_ang;
    float error = 0, prev_error = 0, errorSum = 0;
    float dt = 0;
    float pwm = 0, Pterm, Iterm, Dterm;
    int max_pwm = 130;
    int min_pwm = 100;
    int pwm_range = max_pwm - min_pwm;
    float last_time = millis();
    float start_time = millis();


    while (1) {
        //get angle
        get_dmp_data();
        float norm_yaw_dmp = yaw_dmp;
        if (norm_yaw_dmp < 0) norm_yaw_dmp += 360;

        // Calculate relative yaw
        curr_ang = norm_yaw_dmp - initialYaw;
        if (curr_ang < 0) curr_ang += 360;
        if (curr_ang >= 360) curr_ang -= 360;

        //PID control
        dt = (millis() - last_time) / 1000.0;
        last_time = millis();
        Serial.print(curr_ang);
        Serial.print(", ");
        Serial.print(exp_ang);
        error = curr_ang - exp_ang;
        if(error>180) error -=360;
        if(error<-180) error +=360;

        errorSum += error * dt;

        Pterm = Kp * error;
        Iterm = Ki * errorSum;
        Dterm = Kd * (error - prev_error) / dt;
        pwm = Pterm + Iterm + Dterm;

        //Cap PWM to be +/-255
        if(pwm>255) {
            pwm = 255;
        }
        if(pwm<-255) {
            pwm = -255;
        }

        prev_error = error;

        if (abs(error) <= 0.5 || millis()-start_time>10000) {
            analogWrite(PIN0, 0); analogWrite(PIN1, 0);
            analogWrite(PIN3, 0); analogWrite(PIN2, 0);
            break;
        }
        // Determine motor direction
        bool rotate_ccw = (pwm < 0);
        float abs_pwm = fabs(pwm);

        // Scale PWM value so that PWM goes from min_pwm to max_pwm instead of 0 to 255
        int scaled_pwm = (int)((abs_pwm / 255.0f) * pwm_range) + min_pwm;

        if (rotate_ccw) {
            // Rotate counter-clockwise
            analogWrite(PIN0, scaled_pwm); analogWrite(PIN1, 0);
            analogWrite(PIN3, 1.2*scaled_pwm); analogWrite(PIN2, 0);
        } else {
            // Rotate clockwise
            analogWrite(PIN0, 0); analogWrite(PIN1, scaled_pwm);
            analogWrite(PIN3, 0); analogWrite(PIN2, scaled_pwm);

        }
    }
    break;
}
```

Lastly, I have my localization command `LOCALIZATION`. Again, the inputs to this command are K_p, K_i, K_d which are the PID parameters. Essentially, this command makes the robot spin on-axis while collecting ToF data that can be used to localize the robot.

```
//LOCALIZATION
case LOCALIZATION: {
    // Extract the next value from the command string as a int
    success = robot_cmd.get_next_value(Kp);
    if (!success)
        return;
    // Extract the next value from the command string as a int
    success = robot_cmd.get_next_value(Ki);
    if (!success)
        return;
    // Extract the next value from the command string as a int
    success = robot_cmd.get_next_value(Kd);
    if (!success)
        return;
    distanceSensor.startRanging();

    for(int targetYaw = 360; targetYaw > 0; targetYaw-=10) {
        bool targetReached = false;
        if(targetYaw == 360) {
            get_dmp_data();
            initialYaw = yaw_dmp;
        }

        // PID Turn Loop
        while (1) {
            get_dmp_data();
            float norm_yaw_dmp = yaw_dmp;
            if (norm_yaw_dmp < 0) norm_yaw_dmp += 360;

            // Calculate relative yaw
            currentYaw = norm_yaw_dmp - initialYaw;
            if (currentYaw < 0) currentYaw += 360;
            if (currentYaw >= 360) currentYaw -= 360;

            dt = (millis() - last_time) / 1000.0;
            last_time = millis();
            error = shortest_angular_distance(currentYaw, targetYaw);
            if(error>180) error -=360;
            if(error<-180) error +=360;


            errorSum += error * dt;

            Pterm = Kp * error;
            Iterm = Ki * errorSum;
            Dterm = Kd * (error - prev_error) / dt;
            pwm = Pterm + Iterm + Dterm;

            if(pwm>255) {
                pwm = 255;
            }
            if(pwm<-255) {
                pwm = -255;
            }

            prev_error = error;

            if (abs(error) <= 0.5) {
                // Stop motors
                analogWrite(PIN0, 0); analogWrite(PIN1, 0);
                analogWrite(PIN3, 0); analogWrite(PIN2, 0);

                // Get distance
                int distance_tof = 0;
                while (!distanceSensor.checkForDataReady()) {
                    delay(1);
                }
                distance_tof = distanceSensor.getDistance();
                distanceSensor.clearInterrupt();
                distanceSensor.stopRanging();
                distanceSensor.startRanging();
                Serial.println(distance_tof);

                // Save data to arrays
                expected_ang_arr[step] = targetYaw;
                ang_arr[step] = currentYaw;
                dist_time[step] = millis();
                dist[step] = distance_tof;
                step++;
                break;
            }
            // Determine motor direction
            bool rotate_ccw = (pwm < 0);
            float abs_pwm = fabs(pwm);

            // Scale PWM value to desired range
            int scaled_pwm = (int)((abs_pwm / 255.0f) * pwm_range) + min_pwm;

            // Apply scaled PWM to motors
            if (rotate_ccw) {
                // Counter-clockwise
                analogWrite(PIN0, scaled_pwm); analogWrite(PIN1, 0);
                analogWrite(PIN3, 1.2*scaled_pwm); analogWrite(PIN2, 0);
            } else {
                // Clockwise
                analogWrite(PIN0, 0); analogWrite(PIN1, scaled_pwm);
                analogWrite(PIN3, 0); analogWrite(PIN2, scaled_pwm);

            }
        }
    }
    for (int j = 0; j < steps; j++) {
        tx_estring_value.clear();
        tx_estring_value.append(expected_ang_arr[j]);
        tx_estring_value.append("|");
        tx_estring_value.append(ang_arr[j]);
        tx_estring_value.append("|");
        tx_estring_value.append(dist_time[j]);
        tx_estring_value.append("|");
        tx_estring_value.append(dist[j]);
        tx_characteristic_string.writeValue(tx_estring_value.c_str());
    }
    break;
}
```

Python

In order to perform localization, I created the `local_update()` function, which resets the plotter and runs the update step of the algorithm accordingly. Note that the get_observation_data function and notification handler was the same as Lab 11.

```
async def local_update():
    robot = RealRobot(cmdr, ble)

    # Reset Plots
    cmdr.reset_plotter()

    # Init Uniform Belief
    loc.init_grid_beliefs()

    # Get Observation Data by executing a 360 degree rotation motion
    await loc.get_observation_data()

    # Run Update Step
    loc.update_step()
    loc.plot_update_step_data(plot_data=True)
```

Commands

1. (-4, -3)
    a. Perform localization
    b. Sleep for 15 seconds
    c. Turn to 35 degrees using angular PID
    d. Sleep for 15 seconds
    e. Move forwards at a PWM value of 80 for 800 ms using open-loop control
    f. Sleep for 15 seconds
2. (-2, -1)
    a. Turn to 90 degrees using angular PID
    b. Sleep for 15 seconds
    c. Move forwards at a PWM value of 90 for 1200 ms using open-loop control
    d. Sleep for 15 seconds
3. (1, -1)
    a. Turn to 150 degrees using angular PID
    b. Sleep for 15 seconds
    c. Move forwards at a PWM value of 90 for 1000 ms using open-loop control
    d. Sleep for 15 seconds
4. (2, -3)
    a. Turn to 90 degrees using angular PID
    b. Sleep for 15 seconds
    c. Move forwards using closed-loop control until about 457mm from wall
    d. Sleep for 15 seconds
5. (5, -3)
    a. Turn to 0 degrees using angular PID
    b. Sleep for 15 seconds
    c. Move forwards using closed-loop PID control until about 457mm from wall
    - Passes through (5, -2)
    d. Sleep for 15 seconds
6. (5, 3)
    a. Perform localization
    b. Turn to -90 degrees using angular PID

   c. Sleep for 15 seconds
   d. Move forwards using closed-loop PID control until about 762mm from wall
   e. Sleep for 15 seconds
7. (0, 3)
   a. Turn to 180 degrees using angular PID
   b. Sleep for 15 seconds
   c. Move forwards using closed-loop PID control until about 762mm from wall
   d. Sleep for 15 seconds
8. (0, 0)
   a. Final destination reached

For angular PID, $K_p = 1$, $K_i = 0.000001$, $K_d = 1$. For distance PID, $K_p = 1$, $K_i = 0.000001$, $K_d = 0$.

```
#sample Angular PID command
ble.send_command(CMD.ANGLE_PID, f"1|0.000001|1|180") #Kp, Ki, Kd, angle

#sample sleep command
asyncio.run(asyncio.sleep(15))

#sample distance PID command
ble.send_command(CMD.DIST_PID, f"1.5|0.00001|0|762|60|130") #Kp, Ki, Kd, dist from wall, min_pwm, max_pwm

#sample open loop control
ble.send_command(CMD.GET_USER_MOVE, "80|0|80|0|900") #PWM|Time (ms)

#sample localization code
reset_localization_arrays()
asyncio.run(asyncio.sleep(15))
await local_update()
argmax_bel = get_max(loc.bel)
current_belief = loc.mapper.from_map(*argmax_bel[0])
current_odom, current_gt = (0,0,0), (0,0,0)
cmdr.plot_gt(current_gt[0], current_gt[1])
cmdr.plot_odom(current_odom[0], current_odom[1])
```

15-second Sleep

The purpose of the sleep between the commands is to ensure that the motor drivers don't become too hot while operating for long. I noticed that Becky's robot's motor drivers would significantly heat up after each step therefore causing unexpected behavior in the steps after. In order to decrease this uncertainty, I give the motor drivers a 15-second cooldown time.

Review

I think that the robot works decently well in the map. Sometimes I had to move the robot slightly around the middle of the path, as the battery drains throughout the run causing the open-loop components to not work as well and affecting the PID parameters. But, other than this slight issue, the robot seemed to perform decently well with the task it was given. The total time to execute the complete navigation was around 6 minutes (360 seconds), but there were 16 of the 15-second sleeps. If I didn't have to wait for cooldown, it would then hypothetically take the robot $360 - (16 * 15) = 120$ seconds, or 2 minutes, which I think is quite well.
One thing I liked about my program was its design. I performed most of this lab off-board, while using my on-board code from previous labs. This allowed me to make quick changes to the commands sent to the robot, without having to re-flash every time.
Another interesting thing was that my distance PID often resulted in slipping but I needed the tape on the robot for better localization results. So, even though there was some slipping, I decided to keep the tape on the robot.

Videos

Videos of my attempts are shown on the website.

References

A huge shoutout to all of the TAs for the course!! Also, huge thanks to Becky for letting me borrow her robot since mine was not so great with doing localization. I discussed high-level ideas on how to get started on the lab with Becky and Annabel as well.