Theory and Mathematical Background

December 10, 2022

1 Outline

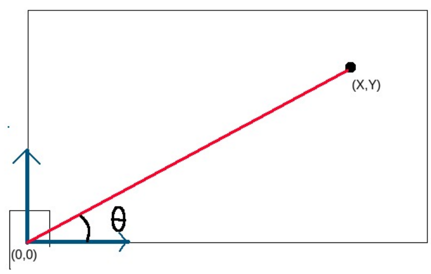
In this chapter of the report, we have discussed the approach that has been used by the bot to navigate the 2D workspace while avoiding the obstacles coming in between its path. The first section of this chapter will lay down the methodology and maths used in writing the code logic for navigation. This would be done in chronological order of real-time occurrence with the snippets of corresponding Arduino code. The second section of this chapter will lay down the underlying concepts for the obstacle avoidance algorithm used in the bot.

2 2D Navigation

At the initial stage of the project, the idea of giving different velocities to the left and right servo of the bot to navigate in the 2d workspace was taken into consideration. But keeping track of the state while doing so was tough due to the timely slippage of the tires. A 6-axis gyroscope and acceleration measurement unit sensor was initially used to track down orientation and position in the 2D space. It turned out that the position estimates were too inaccurate due to the drift caused by the noisy sensor measurements, which were considerable in amount after integrating the acceleration values. Due to these reasons, two types of motions were performed by the bot to reach any point in the 2D space. These are rotational and rectilinear motions. In this way, it was very less likely to have tire slippage affect the state estimate of the bot. A bot's state in 2d workspace comprises of (x,y,θ)

$$X = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{1}$$

Figure 1: Bot and 2D workspace: black point being any desired point in the XY plane



For this approach to work, the need for tracking position estimates was still present. It was later decided to keep the bot's velocity constant when moving in a straight line. This way it was easy to compute the distance by just using the below formula:

$$distance = speed \times time \tag{2}$$

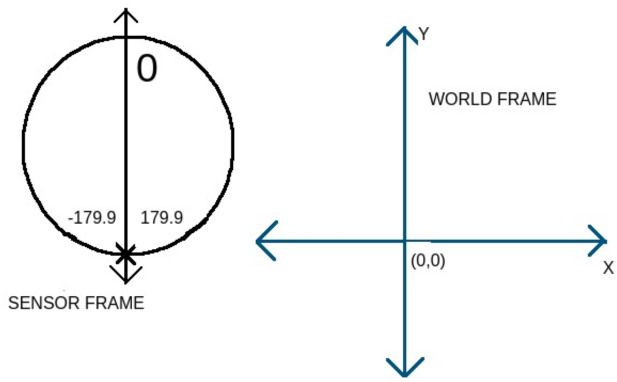
where time was tracked using the millis() function in the Arduino. For tracking orientation (yaw in this case), it was found that the yaw measurement value from the gyroscope was pretty accurate and could be directly used. The gyroscope sensor provides 28-byte-sized FIFO packets to Arduino at a particular frequency. These 28 bytes contain the four values of quaternion parameters in its first 13 bytes. Considering quaternion to be q, then yaw can be calculated using the following formula:

$$q =< w, x, y, z > \tag{3}$$

$$yaw = atan2\begin{pmatrix} 2xy - 2wz \\ 2w^2 + 2x^2 - 1 \end{pmatrix} \tag{4}$$

This yaw angle is computed in the Sensor's frame, which is different than the world frame that is shown in Fig 1. Both the sensor frame and world frame are shown in Figure 2.

Figure 2: Sensor and world frame: they both share the same origin



In Figure 3, ***Desired_state*** function is shown. The idea of attaining any desired state in the 2d workspace is broken down into the following steps:
1) First compute the desired heading angle θ_d, as shown in Figure 1, by using the below formula:

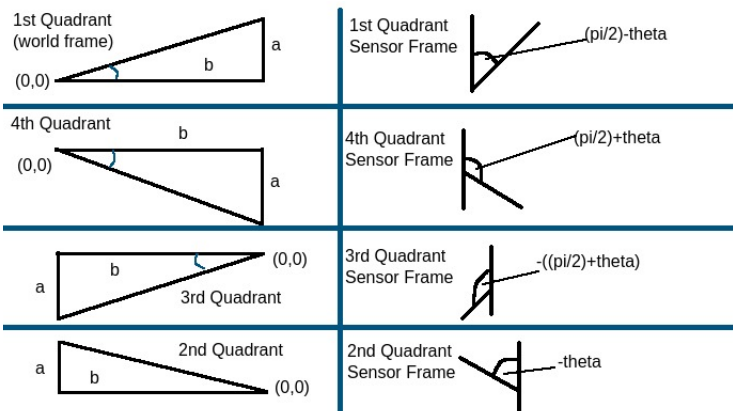
$$\theta_d = atan2\begin{pmatrix} abs(y_d - y \\ abs(x_d - x) \end{pmatrix} \tag{5}$$

here x_d, y_d refers to the desired position coordinates and x, y are the current state position coordinates. As mentioned earlier, the sensor frame is different from the world frame, it is required that the θ_d is expressed in the sensor frame. Code for doing so is given in Figure 3.(code lines from 255 to 265.)Transformations have been shown in Figure 4.

Figure 3: Code snippet of Desired_State function

```
void Desire_state(float xd, float yd) {
RE:
  float desired_yaw = atan2(abs(yd - y), abs(xd - x)); //computing desired yaw ,x and y are current states
  // change desired_yaw according to imu frame
  if (yd - y >= 0 && xd - x >= 0) {
    desired_yaw = M_PI / 2 - desired_yaw;// 1st quadrant
  } else if (yd - y < 0 && xd - x > 0) {
    desired_yaw = M_PI / 2 + desired_yaw;// 4rd quadrant
  } else if (yd - y <= 0 && xd - x <= 0) {
    desired_yaw = -(M_PI / 2 + desired_yaw);// 3rd quadrant
  } else {
    desired_yaw = -desired_yaw; // 2nd quadrant
  }
  desired_yaw = desired_yaw * (180 / M_PI);  // in degrees
  //Now first rotating the robot in desired yaw direction
  Rotate(desired_yaw);
  float distance = sqrt((xd - x) * (xd - x) + (yd - y) * (yd - y));// // distance and time calculate in cm
  time = (distance / speed);// in seconds
  int d = Straight(time, desired_yaw); // function to move in straight line
  if (d == 0) {
    /* this means desired state has not been reached, obstacle came in between run obstacle avoidance
     routine untill a state arrives where feedback from all the three sensors infers no obstacles around*/
    obstacle_avoidance_routine();
    // at this point obstacle has been avoided, but destination has not yet been reached.
    // therefore going back to start of this function.
    goto RE;
  }
}
```

Figure 4: Transformation formulae: World frame to Sensor Frame; a=$y_d - y$, b=$x_d - x$



2) Then implement ***Rotate*** function to align the robot's heading angle in the direction of the desired position. Here, a PID law is implemented to provide the error-proportional signal to the right and left servo of the bot. The code snippet of the function is shown in Figure 5.

Figure 5: PID implementation code for Rotating bot at the desired yaw angle

```
void Rotate(float desired_yaw) {
  int Kp = 1.9; //PID gain constants
  int Ki = 0.02;
  int Kd = .5;
  float error = 0;// error trackers varibles
  float errSum = 0;
  float lastErr = 0;
  while (1) {
    if (mpu.dmpGetCurrentFIFOPacket(fifoBuffer)) {  // Get the Latest packet
      mpu.dmpGetQuaternion(&q, fifoBuffer);
      mpu.dmpGetGravity(&gravity, &q);
      mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
      actual_yaw = ypr[0] * (180 / M_PI);
      error = actual_yaw - desired_yaw;
      errSum = errSum + error;
      if (error < -1 || error > 1) {
        servol.attach(7);
        servol.write(90 - (Kp * (error) + Ki * errSum + Kd * (error - lastErr)));  //   90 is stop
        servor.attach(8);
        servor.write(90 - (Kp * (error) + Ki * errSum + Kd * (error - lastErr)));  //   90 is stop
      } else {
        MotorBrake();
        break;
      }
      lastErr = error;
    }
  }
}
```

3) After the bot has attained desired yaw orientation, it calculates the distance between the current point and the desired point by using the distance formula(code line:336). After calculating the distance, time is being calculated by using Eq2.

$$distance = \sqrt{(x_d - x)^2 + (y_d - y)^2} \tag{6}$$

4)Now,***Straight*** function is implemented for the calculated time. While moving in straight line bot may deviate from the desired yaw angle computed in the step number 1, Eq(5). To correct for this kind of deviation a PID control law is coded inside the Straight function. This function is shown in Figure 6 and 7. If by chance, an object comes in robot's path the while loop terminates before the allotted time and sends back the output 0, which indicates the ***Desired_state*** (Figure 3)function to run the ***obstacle_avoidance_routine*** . If no obstacle comes in the robot's path then straight function's while loop runs for the allotted time and returns the value 1.

Figure 6: Straight function part 1 snippet

```
int Straight(float td, float dyaw) {
  // this function is used to move in staright line
  state_update_lasttime = millis() / 1000;  // in seconds, to note the initial time
  //PID gain constants
  int Kp = 1;
  int Ki = 0.02;
  int Kd = .9;
  // error trackers varibles
  float error = 0;
  float errSum = 0;
  float lastErr = 0;
  float current_millis = millis() / 1000;
  float checker = (millis() / 1000) - current_millis;
  while (checker <= td) {
    int checkObstacle = obstacle_present();
    if (checkObstacle == 2 || checkObstacle == 3 || checkObstacle == 6) {
      MotorBrake();// stops both the motor
      State_Update_1((float)(millis() / 1000), dyaw);// updates the state(x,y)
      return 0;
    }
    if (mpu.dmpGetCurrentFIFOPacket(fifoBuffer)) {  // Get the Latest packet
      mpu.dmpGetQuaternion(&q, fifoBuffer);
      mpu.dmpGetGravity(&gravity, &q);
      mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
      actual_yaw = ypr[0] * (180 / M_PI);
      error = actual_yaw - dyaw;// error used for PID
      errSum = errSum + error;
      if (error < -1.5 || error > 1.5) {
```

Figure 7: Straight function part 2 snippet

```
    if (mpu.dmpGetCurrentFIFOPacket(fifoBuffer)) {  // Get the Latest packet
      mpu.dmpGetQuaternion(&q, fifoBuffer);
      mpu.dmpGetGravity(&gravity, &q);
      mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
      actual_yaw = ypr[0] * (180 / M_PI);
      error = actual_yaw - dyaw;// error used for PID
      errSum = errSum + error;
      if (error < -1.5 || error > 1.5) {
        servol.attach(7);
        servol.write(90 - (Kp * (error) + Ki * errSum + Kd * (error - lastErr)));  // backward  90 is stop
        server.attach(8);
        server.write(90 - (Kp * (error) + Ki * errSum + Kd * (error - lastErr)));  // forward   90 is stop
      } else {
        Forward_Drive();// this functions make the two motors run at a constant speed
      }
      lastErr = error;
    }
    checker = (millis() / 1000) - current_millis;
  }
  MotorBrake();
  State_Update_1((float)(millis() / 1000), dyaw);// updates the state(x,y)
  return 1;
}
```

5) After 4th step there are two possible scenarios; If obstacle present then the code control is taken to the staring point of the ***Desired_State*** function. If obstacle is not present then ***Desired_State*** function terminates.(desire state reached)

All these 5 steps are coded in Figure 3 code snippet; ***Desired_State*** function.

3 Obstacle Avoidance

Obstacle present check is coded inside the straight function(Figure 6 and figure 7). This function checks for the obstacle from all the three sensor inputs. Its code is shown in Figure 8.

Figure 8: Function for checking Obstacle presence

```
int obstacle_present() {
  //000  :  0 no obstacle, leave  obs //001, : 1 straight
  //010, :  2 left turn              //011  :  3 left
  //100  :  4 straight               //101  : 5 straight
  //110  :  6  right turn           //111  : wont arrive in this situation
  int resultU = ultrasonic_value();// in cm
  int resultIR = IR_Module();// digital
  int mid = 0; left = 0;right =0;
  if (resultU < 15) {
    mid = 1;
  }
  if (resultIR == 0) {
    left = 0;
    right = 0;
  } else if (resultIR == 1) {
    left = 1;
    right = 0;
  } else if (resultIR == 2) {
    left = 0;
    right = 1;
  } else {
    left = 1;
    right = 1;
  }

  return (4 * left + 2 * mid + 1 * right);
}
```

Whenever bot faces an obstacle, its code flow switches its control into the obstacle_avoidance routine function. This function's while loop keeps on running until a state is attained where there is no obstacle around the bot. The code snippet for this function is shown in Figure 9. This function is always called from the Desired_state function.

Figure 9: obstacle_avoidance_routine,this function runs until bot has no obstacles around it

```
void obstacle_avoidance_routine() {
  float aux1;int aux2;
  int ostate = obstacle_present();
  //000  : 0 no obstacle, leave  obs //001,  : 1 straight //010, : 2 left turn //011  :  3 left
  //100  : 4 straight //101  : 5 straight //110  : 6  right turn //111  : wont arrive in this situation
  while (ostate != 0) {
    ostate = obstacle_present();
    switch (ostate) {
      case 1:
        aux2 = Straight(2, aux1);break;
      case 2:
        MotorBrake();// stops both the motor
        aux1 = Left_Drive(60);aux2 = Straight(2, aux1);break;
      case 3:
        MotorBrake();
        aux1 = Left_Drive(60);aux2 = Straight(2, aux1);break;
      case 4:
        aux2 = Straight(2, aux1);break;
      case 5:
        aux2 = Straight(2, aux1);break;
      case 6:
        MotorBrake();
        aux1 = Right_Drive(60);aux2 = Straight(2, aux1);break;
      default:  // for 0
        MotorBrake();break;
    }
  }
}
```

It can be seen in Figure 8, the usage of IR_Module function and Ultrasonic_value function to extract the output from ultrasonic and IR sensors. Figure 10 and Figure 11 shows the code of the above said functions. Functions are well commented and self explanatory.

Figure 10: Processing of IR Sensors input

```
int IR_Module() {
  //function output
  // if left sensor senses, output : 1
  // if right sensor senses, output : 2
  // if both sensor senses, output : 3
  // if none of the sensor senses, output : 0
  int statusSensorL = digitalRead(IRSensorL);
  int statusSensorR = digitalRead(IRSensorR);

  if (statusSensorL == 1 && statusSensorL == 1) {
    return 3;
  } else if (statusSensorL == 1 && statusSensorL == 0) {
    return 1;
  } else if (statusSensorL == 0 && statusSensorL == 1) {
    return 2;
  } else {
    return 0;
  }
}
```

Figure 11: Function for Computing distance from Ultrasonic sensor

```
int ultrasonic_value() {
  // establish variables for duration of the ping, and the distance result
  // in inches and centimeters:
  long duration, inches, cm;

  // The PING))) is triggered by a HIGH pulse of 2 or more microseconds.
  // Give a short LOW pulse beforehand to ensure a clean HIGH pulse:
  pinMode(pingPin, OUTPUT);
  digitalWrite(pingPin, LOW);
  delayMicroseconds(2);
  digitalWrite(pingPin, HIGH);
  delayMicroseconds(5);
  digitalWrite(pingPin, LOW);
  // The same pin is used to read the signal from the PING))): a HIGH pulse
  // whose duration is the time (in microseconds) from the sending of the ping
  // to the reception of its echo off of an object.
  pinMode(pingPin, INPUT);
  duration = pulseIn(pingPin, HIGH);
  // convert the time into a distance
  cm = microsecondsToCentimeters(duration);
  return cm;
}
```

Figure 12 and 13 shows the code of Left Drive and Right Drive functions. These functions comes in handy when bot needs to avoid obstacle inside the obstacle avoidance routine. Left drive function rotates the robot to an angle which is at the left of the angle at which obstacle has been detected. Same thing goes with the Right Drive function. Both of the functions outputs the final state angle at which bot finally stops at the end of the function.

Figure 12: Function to rotate the bot left by certain amount of angle; used in obstacle avoidance routine

```
float Left_Drive(float angle) {
  // rotate by any degrees  anticlockwise
  float current_yaw;
  float aux;
  int i = 1;
  while (i) {
    if (mpu.dmpGetCurrentFIFOPacket(fifoBuffer)) {  // Get the Latest packet
      mpu.dmpGetQuaternion(&q, fifoBuffer);
      mpu.dmpGetGravity(&gravity, &q);
      mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
      current_yaw = ypr[0] * (180 / M_PI);  // in degrees
      i = 0;
      aux = current_yaw - angle;  // in imu frame
      if (aux < -179.9) {
        aux = 360 + current_yaw - angle;// in imu frame
      }
      Rotate(aux);
    }
  }
  return aux;
}
```

Figure 13: Function to rotate the bot right by certain amount of angle; used in obstacle avoidance routine

```
float Right_Drive(float angle) {
  // rotate by any desired  degrees  anticlockwise
  float current_yaw;
  int aux;i = 1;
  while (i) {
    if (mpu.dmpGetCurrentFIFOPacket(fifoBuffer)) {  // Get the Latest packet
      mpu.dmpGetQuaternion(&q, fifoBuffer);
      mpu.dmpGetGravity(&gravity, &q);
      mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
      current_yaw = ypr[0] * (180 / M_PI);  // in degrees
      i = 0;
      aux = current_yaw + angle;//converting angle in imu frame
      if (aux > 179) {
        aux = -(360 - current_yaw - angle);// converting angle in imu frame
      }
      Rotate(aux);
    }
  }
  return aux;
}
```

Whenever the control flow is switched out of the Straight function, State update function runs. This function updates the x and y coordinates in the world frame. This function is shown in Figure 14.

Figure 14: Code to update the X and Y coordinates; used in Straight function

```
void State_Update_1(float time, float yaw) {
  float timed = time - state_update_lasttime;
  // here yaw comes in imu frame
  // it needs to be transferred to world frame.
  yaw = 90 - yaw;
  yaw = yaw * (M_PI / 180);
  x = x + ((speed * timed) * cos(yaw));
  y = y + ((speed * timed) * sin(yaw));
}
```

Primitive functions which makes the bot move forward and remain at still position are shown in Figure 15.

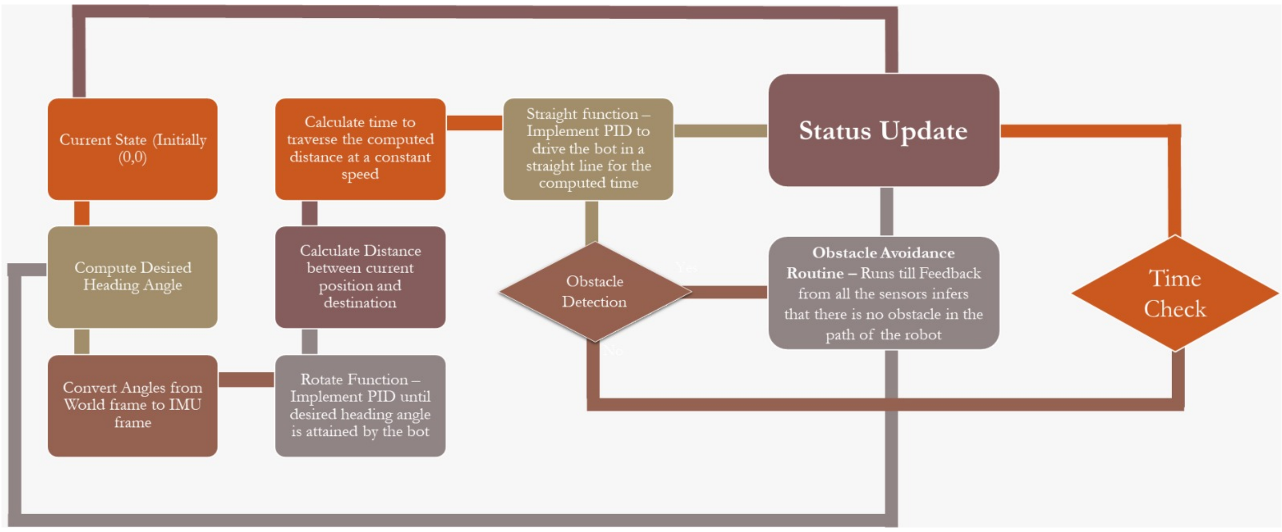
Figure 15: Functions for making the bot go forward and stop

```
void Forward_Drive() {
  servol.attach(7);
  servol.write(180);  // forward  90 is stop

  servor.attach(8);
  servor.write(0);  // forward   90 is stop
}

void MotorBrake() {
  servol.attach(7);
  servol.write(90);  //  90 is stop

  servor.attach(8);
  servor.write(90);  //  90 is stop
}
```

4 Overview

All the functions that have been discussed till now are all inter related in the fashion shown in Figure 16.

Figure 16: FlowChart of all the processes



Ultimately what one needs to call in the Void Loop function of the Arduino code is the function ***Desired_state*** with the input as the desired coordinates received from the ESP32 module Via Wifi. An example of the same is shown in the Figure 17.

Figure 17: Usage of Desired_state function to implement an use-case

```
Desire_state(0, 100);// go to service station
yini = 2;
// code to check for button press
while (digitalRead(buttonPin) == 0) {
  delay(1);
}
Desire_state(100, 40);// go to table
// check for button press
while (digitalRead(buttonPin) == 0) {
  delay(1);
}
Desire_state(0, 0);// go to home
Rotate(0);// straight orient in home
```

The use case shown in Figure 17 shows that bot first goes to the service station which is located at the (100,0) and then waits for the button press which is the indication of the item kept. After the button is pressed, bot goes to the table located at the (100,40) coordinate. There it again waits for the button press which is the indication of item delivered. After the button is pressed, bot again goes to the home position(0,0) and orients itself straight at 0 degree.