Robot Control Documentation Version 1.1

Keegan Kelly

July 18, 2022

Contents

1 Introduction

This document will cover the basic model of the robots used, hardware on the robots relating to movement and position estimation, an overview of the code used to control robots, and an explanation of how to utilize the Robot class for movement. The code for this project is available on my GitHub. If you require information on the localization system or wireless networking, please refer to the other documentation files.

2 Wiring and Connections

2.1 Ardumoto and Motors

Place the Ardumoto shield on top of the Arduino Uno. Connect the two screw terminals for motor A to the left motor and connect the motor B terminals to the right motor. Use the drive function and send an angular velocity of 0 and any positive linear velocity. If the wheels both turn in the right direction everything is good. If either motor is turning in the wrong direction, flip the positive and negative wires going to the motor to reverse its direction. For more information and alternative pins to use for motor control reference this Ardumoto setup guide.

The Ardumoto supports alternate pinouts which this project will take advantage of. Look on the bottom of the Ardumoto and there should be several sets of 3 solder pads for toggle options. To switch pins, cut the piece of copper adjoining the two pads for PWMA, PWMB, DIRA, and DIRB. Next, solder a connection between the two pads that weren't previously connected (center to right if the text is the correct orientation). Make sure not to join all three pads together.

2.2 Encoders

Connect the hall effect sensors to power and ground. The sensors can accept 3-24V. Information on the encoder kit is available here. The output from the left sensor should be connected to pin D2 and connect the right sensor output to pin D3. These are the only interrupt pins on the Arduino Uno and if more are needed, a different Arduino with more interrupt pins is required. For more information on setting up the encoders in software, refer to this guide.

2.3 Adafruit LSM9DS0

If the LSM9DS0 IMU is needed it can easily be connected to the Arduino. Follow this table to connect the sensor. The sensor will also require the installation of the Adafruit LSM9DS0 library. For information on how to use this library, reference the example sketches and this guide.

LSM9DS0 Pin	**Arduino Pin**
VIN	3.3V
GND	GND
SCL	A5
SDA	A4

3 Kinematic Model of the Robot

The robot is a two wheeled differential drive robot with a simple kinematic model. The state of the robot can be described by a 3 component vector containing the position of the robot and its heading $\begin{bmatrix} x, & y, & \theta \end{bmatrix}$. The

movement of the robot in space will be modeled by a linear velocity V parallel to the wheels and an angular velocity ω caused by a rotation about a point centered between the two wheels. The distance from the centre line of the robot to each wheel is R and the radius of each wheel is r. A diagram of these parameters can be seen below.

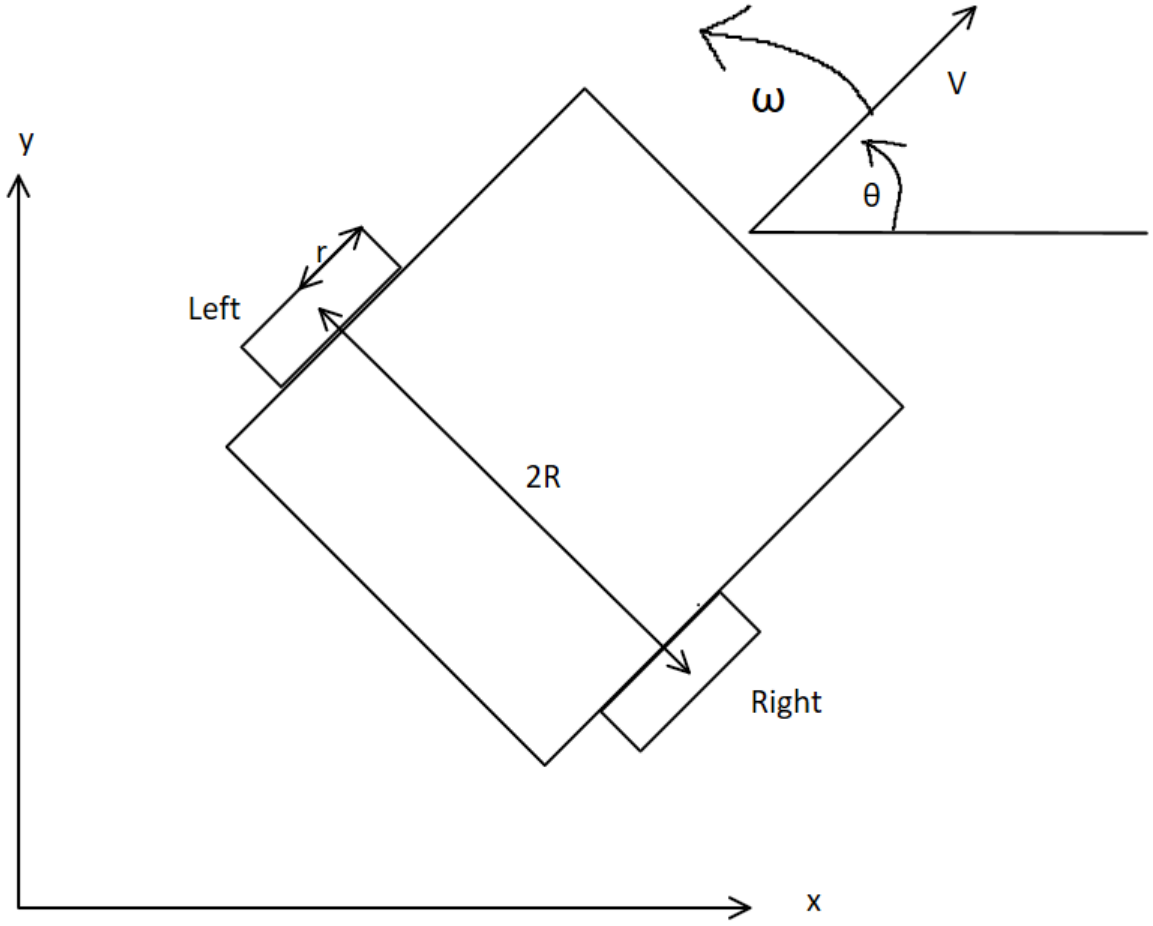

Figure 1: Simplified diagram of the robot

With the left wheel turning at an angular velocity ω_L and the right wheel turning at an angular velocity ω_R, the velocity V of the robot is determined by

$$V = \frac{1}{2}r(\omega_L + \omega_R)$$

and the angular velocity of the robot about the intersection of the robot's and wheels' centrelines, ω, is

determined by

$$\omega = \frac{r}{2R}(\omega_r - \omega_L)$$

These equations will serve as the foundation for updating the robot's position using odometry and driving the robot at a given linear and angular velocity.

4 Odometry

Currently the robot is equipped with disk magnets with 4 sets of poles and hall effect sensors for wheel encoders. The hall effect sensors output a high signal whenever it is near a North pole and a low signal whenever it is near a South pole. If the interrupts attached to the hall effect sensor outputs are set to increment the number of ticks every time the output of the sensor changes, there will be 192 ticks/ π rad. For more information on wheel encoders, reference this guide. With this information and the known values of r, Wheelbase (2R), and the direction each wheel is turning(1 for forward and -1 for reverse), the linear distance traveled and change in heading can be calculated by

$$\Delta L = (LeftTicks * LeftDirection + RightTicks * RightDirection)\frac{\pi}{192}\frac{r}{2}$$

$$\Delta\theta = (RightTicks * RightDirection - LeftTicks * LeftDirection)\frac{\pi}{192}\frac{r}{Wheelbase}$$

To simplify the calibration process and reduce the compute time the factor $\frac{\pi}{192}\frac{r}{2}$ will be replaced with CL and CR ($\frac{\pi}{192}\frac{r}{2} \times CalibrationFactor$). The equations then will be

$$\Delta L = (LeftTicks * LeftDirection * CL + RightTicks * RightDirection * CR)$$

$$\Delta\theta = (RightTicks * RightDirection * CR - LeftTicks * LeftDirection * CL)\frac{2}{Wheelbase}$$

The new state of the robot can then be calculated by assuming that the direction of linear travel was the previous heading plus half of the change in heading.

$$x+ = \Delta L \cos(\theta + \frac{\Delta\theta}{2})$$

$$y+ = \Delta L \sin(\theta + \frac{\Delta\theta}{2})$$

$$\theta+ = \Delta\theta$$

This method on its own is fairly accurate, but the model does not account for wheels slipping and the wheels are simplified to be infinitesimally thin. For this reason the values of r and the wheelbase should not be the exact measurements of wheel radius and the wheelbase. Instead, the values should be experimented with to try and get better results. The starting point for r should be 33.5 mm and R should start around 80 mm. Alternatively, a multiplier could be added to $\Delta\theta$ and ΔL to tune the odometry performance.

4.1 Calibrating the Odometry

For useful operation of the robot, the odometry will need to be calibrated. For calibration, CL, CR, the wheel base and wheel radius should be adjusted. Start by setting a straight forward path of 1-2m with increments of 25cm to reduce speed. Check to see how straight the robot is driving. If it is veering to either side, reduce the corresponding calibration factor by a small amount ($1 - 2\%$) and repeat (if it veers to the right reduce CR and reduce CL if it veers left). Then use the same path and measure how far forward the

robot actually moves. If it travels too far, increase the wheel radius and if it comes short decrease the wheel radius (remember that the final position will only be within the absolute error threshold for PID control). Next, setup a path where the robot will drive along a square path. In this test the goal is to see how close to 90° the robot gets on each turn. If it turns too far, reduce the wheelbase and if it doesn't turn far enough, increase the wheel base.

Please note that these calibration parameters are not isolated and changing any of them will likely require the other parameters to change. If the turning performance is adequate but the robot is not driving the correct distance, multiplying the wheel radius by a factor X will require that wheelbase is also multiplied by X. This is simple enough, however, making other changes can be more complicated so it is recommended that these tests be performed in the above order. Also note that the wheelbase will likely be the most complicated to tune as it is not easy to directly measure and small changes will severely affect the path after multiple turns. After doing a first pass at calibration, the results will likely be good and a second iteration will require less change. There are more technical calibration methods like UMBmark however all published papers on the method did not show how to adjust the parameters after calculating the α and β values so concepts were adapted from this calibration method.

5 IMU

The Adafruit LSM9DS0 inertial measurement unit can be added to each robot to help with position and heading estimation. The most effective way to use the unit is to use an algorithm that corrects the accelerometer, gyroscope, and magnetometer readings with the other sensors. However, these calculations are computationally expensive and the results are dependant on the quality of the sensors. Following this guide and using the Adafruit AHRS library to compute the Euler angles for the orientation of the robot proved unsuccessful. With calibration there was still about 15°/s of constant drift on the heading which would make the output of the computations useless. The examples sensor fusion estimate also takes up 75% of the Arduino Uno's storage so a higher end Arduino would be needed to use a more accurate sensor fusion algorithm.

Additionally, from research it was determined that using any consumer grade IMU to determine x and y position would not produce usable results.

The magnetometer could be used by itself for heading estimation where the heading is computed with

$$\theta = -atan2(Mag_Y, Mag_X)$$

if the y axis on the IMU is aligned parallel to the long side of the robot. However, this method does not work particularly well with the current sensors and the estimate it produces sometimes shows an angle change of π through a rotation of $\frac{\pi}{2}$ and a change of π through the remaining $\frac{3\pi}{2}$ of the rotation. For this reason, the magnetometer or a sensor fusion algorithm is not recommended for state estimation.

Using a Kalman filter to determine pitch, roll, and yaw was also attempted, but a library or implementation capable of working on an Arduino Uno and the LSM9DS0 IMU was not found and the disappointing results from the Adafruit AHRS library discouraged further exploration.

6 PID Control

PID control is a method of controlling a parameter with the error of another parameter. PID stands for proportional, integral, and derivative as the output is determined by a term proportional to the error, in-

tegral of the error, and derivative of the error. Each term is multiplied by a positive gain. K_p is the proportional gain, K_i is the integral gain, and K_d is the derivative gain. For the robots, the linear velocity is controlled using the absolute position error and the angular velocity is controlled using the heading error. In the Robot class, there are private attributes for the two sets for gains for angular and linear velocity.

For the PID control for linear velocity, the absolute error times the cosine of the heading error is used for the error. This will ensure that the robot does not drive fast when forward or reverse is not a productive direction. It also provides the correct logic for when the robot should drive in reverse, ensuring that the robot never has to turn more than $\pi/4$rad. This also allows the PID control to work properly as the cosine of the heading angle allows the error to be negative and positive. Only using the absolute error would not work properly as the integral could only increase in value and if the robot overshot its position it would continue to accelerate away from the target position. Incorporating the cosine of the heading angle in this way also requires that the heading error is offset by π if the cosine of the heading error is negative as the robot will be driving in reverse and thus is effectively pointing πrad away from the current heading. These calculations should be done before updating the integrals and derivatives.

For more information on PID control check out this MATLAB tech talk on PID control systems.

7 Code Overview

All functions pertaining to controlling the robot are found in the file *RobotControl.h* in the *RobotLocal* folder. There should be a copy of the file in the subfolder pertaining to each robot. At the top of the file there are pin definitions, functions for the encoder interrupts, and a function $float\ fixAngle(float\ Angle)$ that returns the argument angle to the range $[-\pi, \pi]$. There is also the value of π in the variable $float\ pi$. All other code is contained in the class Robot. Within the Robot class there are also some variables and functions related to wireless networking and localization. These are covered in the localization documentation. For a more detailed explanation on how the methods work, check the comments in the code.

7.1 Public Attributes and Methods

The Robot class has three public attributes, $float\ x$, $float\ y$, and $float\ theta$. These variables hold the current (x, y) position of the robot and its heading in radians. In localization updating these variables will override the position estimate given by odometry.

7.1.1 Constructor: Robot(float X, float Y, float THETA, int id, String address)

When declaring an object of the Robot class, one must use the constructor function and pass the initial (x, y) position and heading of the robot. If you are using localization these variables are not important and can be assigned to any value you want. The id and address are used for localization and should be the id (1, 2, or 3) of the robot and the address of the server. The constructor function also initializes the motor controller and sets up the interrupts for the encoders.

7.1.2 void drive(float v, float w)

This method uses the kinematic model of the robot and a linear map to set the PWM speeds for each motor based on the given linear velocity v and angular velocity w. It also sets the direction of each motor and adds a cutoff for the motor speeds to keep the value in the acceptable range and sets them to 0 if the desired angular velocity of the wheel is below a threshold of $0.25 rad/s$.

7.1.3 void moveTo(float X, float Y)

This method accepts a target (x, y) position and moves the robot to that position through use of PID control for the linear and angular velocity. The gains and absolute error threshold for the PID control are private attributes. This function also updates the position of the robot with odometry on each iteration of the while loop and uses the drive function to set velocities. Once the absolute error is less than the specified acceptable error, this method will set the linear and angular velocity of the robot to 0 to stop it from moving.

7.2 Private Attributes and Methods

The private attributes for the Robot class include:

- float r: radius of the wheels
- float WB: effective wheelbase (measured value multiplied by calibration factor)
- float CL: calibration factor that converts left encoder ticks to distance
- float CR: calibration factor that converts right encoder ticks to distance
- `uint8_t` DirWL: direction of the left wheel (0 is reverse, 1 is forward)
- `uint8_t`: direction of the right wheel (0 is reverse, 1 is forward)
- float Kp: proportional gain for linear velocity
- float Ki: integral gain for linear velocity
- float Kd: derivative gain for linear velocity
- float KpTheta: proportional gain for angular velocity
- float KiTheta: integral gain for angular velocity
- float KdTheta: derivative gain for angular velocity
- float Err: absolute positional error tolerance in m

7.2.1 void updatePosition(void)

This method uses odometry to update the x, y, and theta attributes.

7.2.2 void fixTheta()

This method will return the robots theta attribute to the range $[-\pi, \pi]$.

7.2.3 void setupArdumoto()

This method initializes the pins for the Ardumoto motor controller.

8 Usage Example

Here is a basic example showing how to initialize a Robot object and have it move from $(0, 0)$, to $(0, 1)$, to $(0.5, 0.5)$ in a loop. It is important not to declare the object robotA inside the setup function as this would restrict its scope to the setup function. As well, this example does not incorporate any localization so the position and heading will drift over time due to inaccuracies in the assumptions made in the kinematic model used for odometry (wheels slipping is not accounted for, the wheels are not aligned perfectly, imperfections in the value of r and R, ...). This will serve as a good test to see how far the position drifts over time. From my experimentation, most of the drift is a result of the heading drifting as an error in heading will drastically change where the robot moes to. As the error in the heading accumulates from each turn, the robot can end up $\pi/2$rad off after 4 or 5 cycles of this test.

```
#include "RobotControl.h"
void setup(void)
{
    Serial.begin(9600);
}

Robot robotA(0, 0, pi / 2);

void loop(void)
{
    robotA.moveTo(0, 1);
    robotA.moveTo(0.5, 0.5);
    robotA.moveTo(0, 0);
}
```