# Laboration 3: Differential Drive with WiFi control

Sensors and Sensing

Benny Frost, Tom Olsson

January 18, 2016

*All code for this project can be found at*
*https://github.com/tgolsson/ros-arduino-wifi*

## Contents



## Code listings



## List of Figures

# List of Tables



# 1 Motivation and theory

The goal of this project was to implement a differential drive robot based on the Robot Operation System [ROS] on Arduino, which can be controlled using a WiFi connection. While a differential drive is easy to implement and test using a wired connection, most realistic scenarios will require wireless operation. As ROS is very common in the scientific community, and Arduino is a cheap prototyping platform in comparison to commercial robots, this could allow a larger freedom in robot design.

The goals are:

⇒ To construct a robot with two powered wheels

⇒ Setup the two motors with a differential drive controller

⇒ Implement WiFi communication between PC and Arduino

⇒ Tune the odometry

## 1.1 PID with feedback

PID in the name PID-controller is short for *Proportional-Integral-Derivative*-controller. As this implies, the controlling signal is based on a proportion of the current error, the previous error, and the rate of change of the observed error. Sometimes, the error is augmented to include a part of the previous output value. In the case of velocity control it can be used to counteract overshoot if the observable value does not react instantly to output value. The mathematical formulation of this is shown in (1)-(2).

**Let:**

| | |
|---|---|
| $e(t)$ | be some error measurement between current state x(t) and preferred state r(t) |
| $K_\mathrm{p}$, $K_\mathrm{i}$, $K_\mathrm{d}$ | be the respective weights for the proportional, integral and derivative terms |
| $u(t)$ | be the output signal at time $t$ |

**Then:**

$$e(t) = r(t) - x(t) - u(t-1) \tag{1}$$

$$u(t) = K_\mathrm{p} \cdot e(t) + K_\mathrm{i} \cdot \int_0^t e(\tau) \cdot \mathrm{d}\tau + K_\mathrm{d} \cdot \tfrac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{2}$$

This formulation of the PID controller borrows from feed-forward open-loop control by partially ignoring the current measurements[1]. The controller is shown in fig. 1 on the next page.

Another modification of PID controllers to improve motor control is the addition of an output deadband or *stiction*[2]. Stiction is the static cohesion threshold that needs to be overcome in order to bring an object from rest when in contact with other objects. In the case of a motor, this can be seen as the minimum `PWM`-value where the shaft starts turning consistently.

[1] An open loop controller has no sensory information: it has only state and model.

[2] A portmanteau of *static friction*

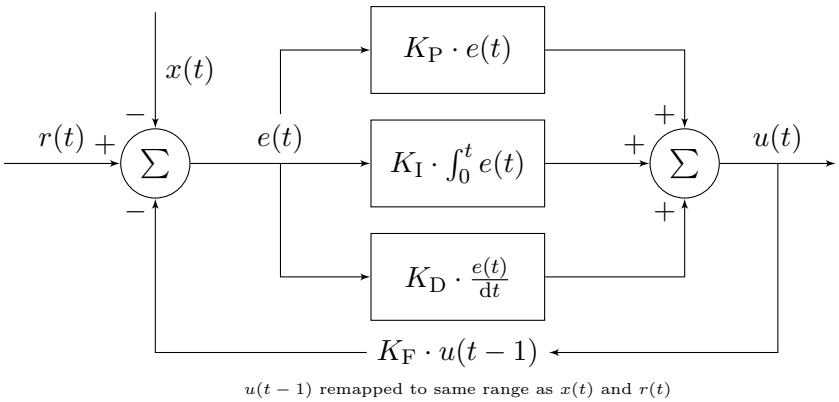


Figure 1: PID controller with feedback term.

## 1.2 Communication protocols

As the `Arduino ros_lib` implementation is based on the premise of serial communication, some things need to be taken into consideration. No matter whether operating over the USB port or a direct `TX,RX` connection a serial protocol is always synced, and has a more or less constant rate of information, and has predictable packet sizes.

TCP/IP on the other hand - which is the wireless protocol for a sustained server/client connection - is not synced, nor does it make any guarantee on any communication speeds. The only guarantee is that the data will arrive at some point, and that it will be possible to retrieve the packages in proper order. Compared to serial it also makes no guarantee on packet size; and may sometimes try to limit the number of packages sent by collating many small packets into larger ones using f.ex. Nagle's algorithm.

This discrepancy means that one cannot rely on data arriving regularly, and must make sure that the implementation can handle the problems that can occur. These problems could be a message being broken up into multiple parts, or only receiving half the message, a connection dropout, etc.

## 1.3 Differential Drive

A differential drive is a kinematic model for two individually controlled wheels used to control both speed and rotation by altering the relative speed between the wheels. The controller implemented here is an inverse kinematic model; i.e., the desired output is given and then the parameters for the controller are calculated to give these output parameters.

**Let:**

| | |
|---|---|
| $v, \omega$ | be the desired linear $(m/s)$ and angular velocities $(rad/s)$ |
| $L$ | be the distance between the two wheels |
| $r$ | be the radius of the robots wheels |
| $k$ | be the number of encoder ticks per shaft rotation |
| $u_l, u_r$ | be the left and right encoder speeds |

**Then:**

$$u_l = \frac{2\pi r}{k} \cdot \begin{cases} \left(\frac{v}{\omega} - L\right) * \omega & if \omega \neq 0 \\ v & otherwise \end{cases} \tag{3}$$

$$u_r = \frac{2\pi r}{k} \cdot \begin{cases} \left(\frac{v}{\omega} + L\right) * \omega & if \omega \neq 0 \\ v & otherwise \end{cases} \tag{4}$$

### 1.4 Odometry

The kinematic model for differential drives can be used to estimate the robots current position relative to the starting position. The equations for updating the robots position are shown in eq. (5)-(13).

**Let:**

| | |
|---|---|
| $L$ | be the distance between the two wheels |
| $r$ | be the radius of the robots wheels |
| $k$ | be the number of encoder ticks per shaft rotation |
| $\hat{u}_l, \hat{u}_r$ | be the smoothed left and right measured encoder speeds (*ticks/second*) |
| $x$, $y$, $\theta$ | be the position and rotation of the robot in a 2D coordinate system |

**Then:**

$$l_{left} = \frac{2\pi r}{k} \cdot \hat{u}_l \tag{5}$$

$$l_{right} = \frac{2\pi r}{k} \cdot \hat{u}_r \tag{6}$$

$$\Delta = \frac{(l_{right} - l_{left})}{L} \tag{7}$$

$$d = \frac{l_{right} + l_{left}}{2} \tag{8}$$

$$\Delta_x = \begin{cases} d \cdot \cos\left(\frac{\Delta}{2}\right) & if\ |\Delta| < \epsilon \\ \frac{d}{\Delta} \cdot \sin(\Delta) & otherwise \end{cases} \tag{9}$$

$$\Delta_y = \begin{cases} d \cdot \sin\left(\frac{\Delta}{2}\right) & if\ |\Delta| < \epsilon \\ \frac{d}{\Delta} \cdot (1 - \cos(\Delta)) & otherwise \end{cases} \tag{10}$$

$$x_t = x_{t-1} + \Delta_x \cdot \cos(\theta_{t-1}) - \Delta_y \cdot \sin(\theta_{t-1}) \tag{11}$$

$$y_t = y_{t-1} + \Delta_y \cdot \sin(\theta_{t-1}) + \Delta_y \cdot \cos(\theta_{t-1}) \tag{12}$$

$$\theta_t = \theta_{t-1} + \Delta \tag{13}$$

## 2 Implementation

The purpose of this project was to build a robot with two drive wheels. This robot shall use an Arduino Due controller board with a differential drive controller for these wheels, and communicate to a master node using WiFi and TCP/IP. Part of the project is also measuring and tuning the accuracy for the robots odometry.

### 2.1 Hardware and environment

The project used an *Arduino Due* microcontroller [1], with the *Arduino Motor Shield R3* [2]. These are programmed using Serial-over-USB; with the dedicated IDE. The version of the IDE used is 1.6.5. The project also includes usage of the *Robot Operating System* [ROS], version *Indigo Igloo*. All ROS packages were installed directly from GitHub. Two WiFi modules were tested during the project, the official *WiFi Shield R3* [3] as well as the *ESP8266* [4].

The motors used are the *Micro Motors RHE158 75:1 12V DC* [5].

### 2.2 Physical design

The main body of the robot is built from aluminum plating and profiles to get a robust robot. The circuit boards are attached to Plexiglas which then are attached to the main body. This is done to get a good insulation from the main body. The Plexiglas pieces are attached with Velcro so it will be easy to detach them if necessary. At front the two motors with the driving wheels was mounted and two passive wheels where mounted at the back of the robot. The passive

wheels do not swivel and there was a concern if this would work. Testing showed that the two electrical motors were too weak to be able to turn the robot with this setup. This is probably because the friction against the floor is too high. A centered swivel wheel was therefore mounted at the rear of the robot. The robot is shown in fig. 2.

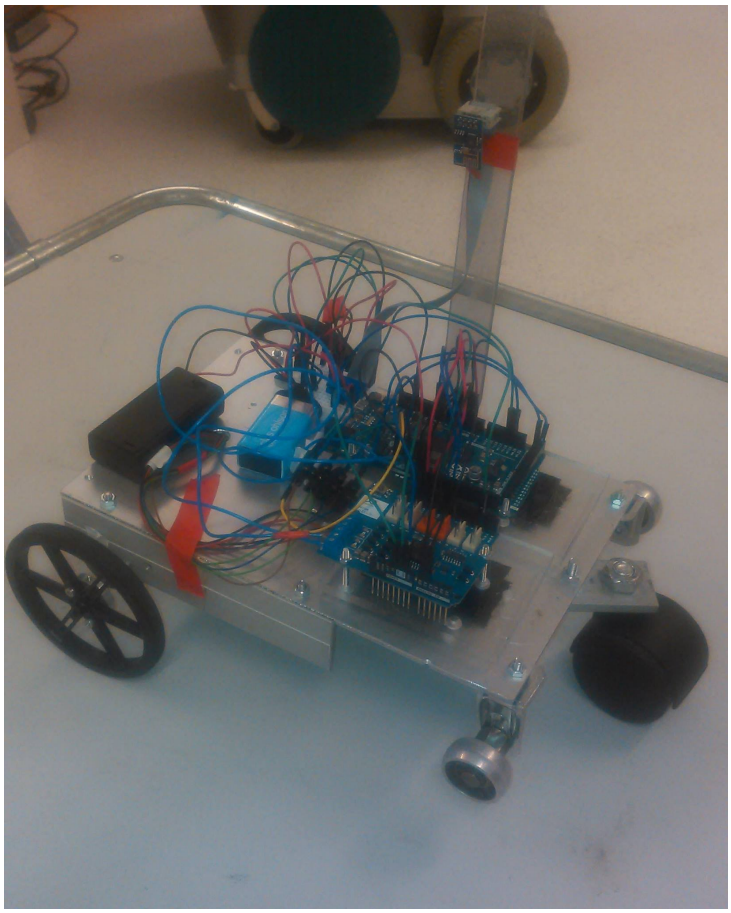

Figure 2: Physical design of the robot. The centered swivel wheel is at the back of the robot. Motor battery pack not mounted for better viewing.

For controlling the robot, an Arduino Due microcontroller is used with an Arduino Motor Shield, and for communication an Arduino WiFi shield is used. These two shields are made to sit on top of the Arduino Due. However, this means that the WiFi shield and Motor shield will also sit on top of each other. Since the WiFi shield and the Motor shield use the same I/O pins of the Arduino Due it was decided that only the WiFi shield should be mounted on top of the Arduino Due and the Motor shield should be mounted next to the main board and be connected to the Arduino Due with wires, and by doing this I/O pins could be rerouted to be used for the Motor shield. The WiFi shield was eventually replaced with an EP8266 as described in the next section. Fig. 3 on the following page shows how the Arduino Due, Arduino Motor shield and ESP8266 are connected.

### 2.3 WiFi communication

During setup and configuration of the Arduino WiFi shield it was obvious that the communication was slow, taking almost two seconds from sending data to reaching the destination, which was documented by others online [6, 7]. Similarly, it was discovered that the largest package that could be sent to the Arduino WiFi card is 92 bytes. If the package is larger there will be a silent failure. Because of this it was decided to use another WiFi card. We decided to use a popular combined microcontroller/WiFi card, ESP8266.

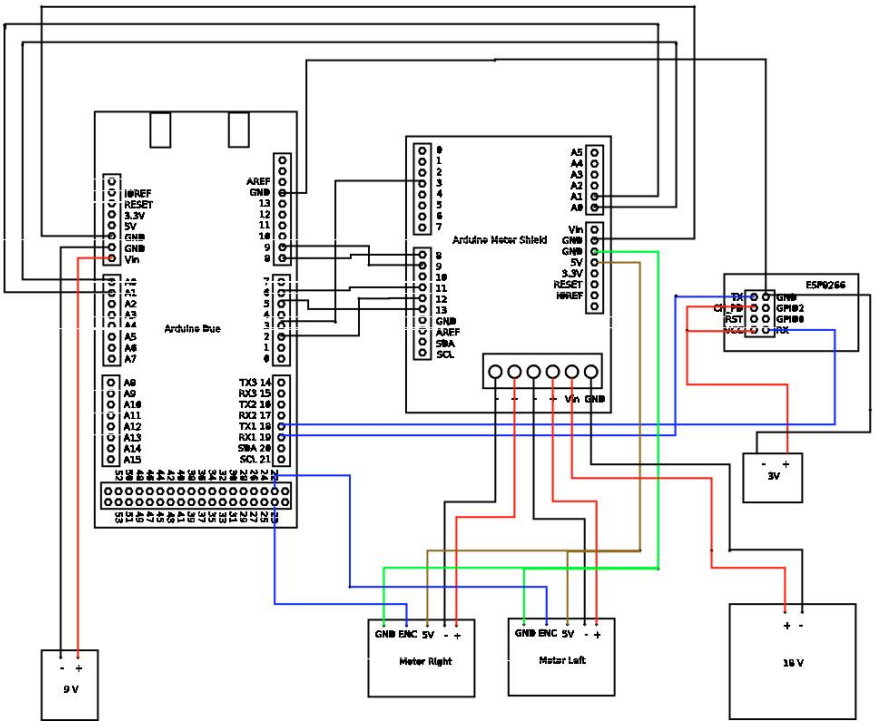


Figure 3: The wiring diagram for the robot's electronics

The ESP8266 is a standalone microcontroller with built-in WiFi support [4]. This card is often recommended instead of the official WiFi card both because of its cost as well as superior performance, which is the reason we use it. Similarly to the normal Arduino boards it comes with its own toolchain, and runs the code on its own in a separate process from the main Arduino board. This allows the card to be setup to handle I/O asynchronously from the main control loop which is a big performance gain. The final communication setup is shown in fig. 4.

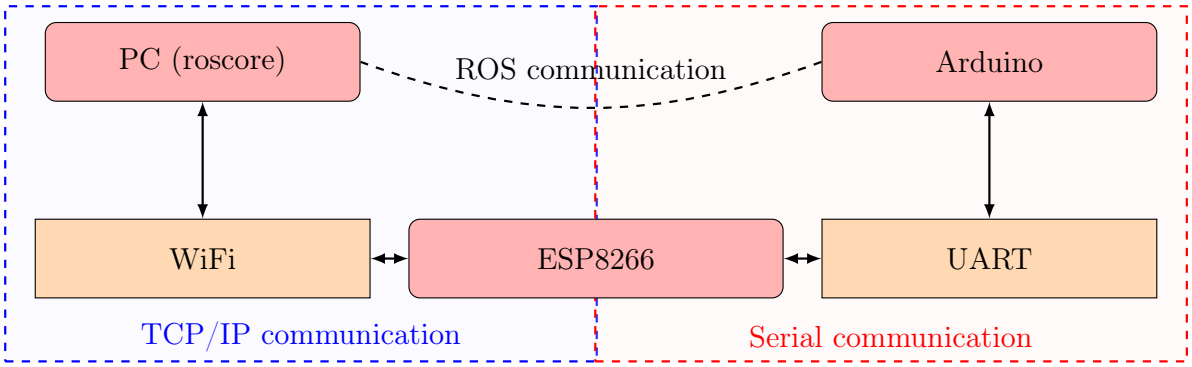


Figure 4: System view of communication

The PC and the Arduino both run `rosserial` to handle all communication and serialisation. To solve the issues raised in section 1.2 on page 3 the baud rate between the ESP8266 and the Arduino Due was reduced by 33 % compared to what `ros_lib` uses normally at; to 38.4k from 57.6k. To further reduce the risk of dropping messages, the `NodeHandle` class was also modified to add a retry clause; where it would retry a second time whenever a message was at risk of being discarded. The code for this is shown in listing 1 on the next page.

Listing 1: `Code for error recovery in NodeHandle`

```
217                 if( data < 0 )
218                 {
219                     if (mode_ != MODE_FIRST_FF)
220                     {
221                         delay(10);
222                         data = hardware_.read();
223                         if (data < 0)
224                         {
225                             Serial.println("Dropped message due to early end.");
226                             break;
227                         }
228                     }
229                     else
230                         break;
231                 }
```

## 2.4 Controller node

The node used to steer the robot was developed by Husqvarna for their lawn mower *Auto Mower 320* and used for research at Örebro University. This node fulfilled the requirements that was needed for this project. It is a Python script that reads from the keyboard and publishes a `geometry_msgs::Twist` message on the ROS bus. The message uses `linear.x` for forward speed and `angular.z` for angular speed.

The PC also runs the `roscore` as well as `rosserial_python` in TCP mode.

## 2.5 Passthrough node

The WiFi node is also simple. During startup it connects to a dedicated WiFi router, and then connects to a hard-coded server IP. In the main `loop` function it reads from the serial and writes to the WiFi; as well as from the WiFi to the serial as was shown in fig. 4 on the preceding page. To make it more reliable the node checks both network and server connection each iteration, and attempts to reconnect if any issues are detected.

## 2.6 Arduino node

This node has two PID controllers with their own $K_\mathrm{P}$, $K_\mathrm{I}$ and $K_\mathrm{D}$ values, to set the velocity of the two wheels. It is necessary to have two PID controllers since the motors appear to have slightly different performance.

The PID controller uses the minimum jerk equation to control the setpoint value, to ensure that the acceleration and deceleration is smooth. The implementation used here is based on the derivation in [8] for estimating the rate of change; and otherwise uses the canonical equation.

One major problem when implementing PID controllers to control the electrical motors is the deadband of the motors. The value of the output from the PID controller is clamped to [0,4095] and between 0-700 nothing happens. To solve this; if the output is between 0-10 the output is set to 0, and if the output is between 11-700 the output is set to 700, else the output is clamped to be in between 700-4095.

The velocity of the motors is calculated by counting the number of interrupts from the motors during one cycle of the controller. This input signal from the encoders is then smoothed with a sliding window filter (eq. (14) on the next page), which means to get a velocity change to be fully visible it takes around 330 ms with $\alpha = 0.1$ and the controller running at 30 Hz. This also means there is a delay in the signal that the PID controller receives, which skews the error value and by extension causes the controller to overreact. As this mostly affects the integral term, it can cause the controller to overshoot by a large margin. To counteract this, a fraction of the

output is fed back into the input as a promised velocity that will happen, so the controller will not ramp up to high output and therefore not overshoot.

$$x_f(t) = (1 - \alpha) \cdot x_f(t - 1) + \alpha \cdot x(t) \tag{14}$$

For the communication with ESP8266 the Serial1 (`TX1,RX1`) is used instead of the usual Serial/USB (`TX0,RX0`). With this setup it is still possible to send debug data to the Arduino IDE with the USB connection.

The Arduino sketch uses `rosserial_arduino` to handle all communication through the `NodeHandle`. The node subscribes to `/robot_0/velocity_commands` (`geometry_msgs::Twist`) to get the commands. It advertises its pose on `/robot_0/odometry` (`geometry_msgs::Pose2D`). The Arduino `loop` function runs at 30 Hz and the advertised topics are published every third iteration.

# 3 Results

Overall the system performs well programmatically, with no bugs or crashes inside the Arduino or the ESP8266. The device seems to run at a consistent speed, and reacts quickly to new commands. There are some issues related to network connectivity, with a dropped message once every two minutes on average, as well as one recoverable connection loss in 30 minutes of runtime. However, during execution these have no noticeable impact.

One major factor in network functionality seems to be the voltage of the ESP8266 battery pack. It is intended to operate with 3.3 V input voltage, and if it drops down to 2.7-2.8 V it becomes significantly harder to connect and maintain the connection. There is also a noticeable pulldown effect if the Arduino Due battery pack drops in voltage, where the shared ground will pull down the ESP8266 as well, therefore breaking the connection. Both the Arduino Due and the ESP8266 seem to consume a lot of power as well; needing replacement batteries after 40 minutes to an hour. Though 40 minutes is ample time for working inside a lab, it imposes limits on practical usage of the robot; especially as rechargeable batteries generally have lower energy density.

Lastly, there is significant work to be done on the design of the robot. Though the robot works well in situations with few velocity changes, it suffers from large inaccuracies during *stop-go* scenarios. This is described in more detail in section 3.2 on the following page.

## 3.1 PID tuning

The PID parameters were tuned by first setting up the $K_\mathrm{P}$ parameter to achieve a good tracking when plotting the setpoint and the measured velocity. When this tracking was smooth the $K_\mathrm{I}$ term was added until overshoot appeared, and then both parameters were reduced in proportion until overshoot stopped. This was required because of the smoothing applied to the measured velocity. Once the behaviour was stable with different setpoints, the $K_\mathrm{D}$ term was added to further reduce overshoot, as well as to stabilise the value.

After adding and doing final tuning on the whole set of PID-values the feedback term was added to the controller. The feedback term allowed us to reduce the minimum-jerk rise-time while keeping overshoot low on large velocity changes. It does so by acting as a promised velocity. Since the output of the PID - and by extension the PWM value - maps directly to velocity, an output value is instantly actuated as a change in speed. However due to the sliding window smoothing, only 10 % of this change will be visible in the next iteration, and then 19 %, and so on. By feeding back the PWM value (converted to the internal velocity measurement) we can reduce the error in proportion to our output PWM value, which allows the error to closely track the real error without smoothing applied.

Our final PID values are shown in table 1. These values are tuned for movement on the ground with all battery packs on board. We found that the PID values were different depending on whether the wheels were under load or not. The values chosen provide a good trade-off between decent tracking, reliable movement and quick reactions to commands.

| Left motor | Parameter | Right motor |
|---|---|---|
| 23 | $K_\mathrm{P}$ | 24 |
| 36 | $K_\mathrm{I}$ | 35 |
| 0.1 | $K_\mathrm{D}$ | 0.1 |
| 0.05 | $Feedback$ | 0.05 |

Table 1: The final values used for the left respectively right motor controller.

### 3.2 Odometry accuracy

The position estimation of the robot (as described in section 1.4 on page 4) is accurate given a few constraints, but inaccurate if these are not met. During straight line or long smooth movements the robot has a relative error of less than 1 %. However when executing movements with multiple changes in angular velocity the odometry does not keep up with the turning; and becomes inaccurate. In this case, the errors can accumulate up to 3-4 % of the distance traveled.

One large part of this is the specific swivel wheel we use at the rear. When starting with the wheel properly aligned with the direction of travel the errors are small. However, after turning the wheel may be orthogonal to the vehicles direction, and acts as a lever to twist the robot. However, this wheel is not intended for usage in robots, and could therefore skew the results. The property we believe influences the performance is the distance between the wheel attachment rotation point, and the wheel friction point. In the case of our wheel, this is almost as large as the wheel's radius. This means that to align with the vehicles direction, the wheel needs to start rotating. During this period of time, the wheel travels more easily in a direction orthogonal to the vehicles movement, i.e., it rotates while at the same time applying friction in the direction of travel causing slippage.

The effect described above primarily causes overturning, where a turning movement continues for longer than intended while the swivel wheel corrects itself. When using the original two static wheels however, the issues was the opposite. In that case, the friction was orthogonal to the direction of travel at all times, and therefore caused slippage during fast turning instead, causing underturning. When testing the setup we found that a 90° turn could result in as little as 30-45° of actual turning, which is obviously worse than the performance of the swivel wheel.

We measured the odometry with four different movement patterns. In all cases, each movement action was discrete, i.e., each movement was fully completed before the next one. As is common when measuring odometry we started by moving in square patterns both counterclockwise (table 2 on the next page) and clockwise (table 3 on the following page). The square used was a 2x2 m in size, and counter-measured along the diagonals to make sure it was perfectly square. The movement was controlled using PC node, and the error in odometry was measured as the difference between odometry position and actual position. As can be seen in tables 2 and 3 there is a high error during square movements, especially when moving clockwise. The error is measured as absolute value to make comparison easier, and to prevent opposite signs from skewing the score. As can be seen there is also better performance when doing counterclockwise turns instead of clockwise turns, though the sample is small in this case.

| Iteration | $\lvert\mathrm{Error}_x\rvert$ [cm] | $\lvert\mathrm{Error}_y\rvert$ [cm] |
|---|---|---|
| 1 | 26 | 0 |
| 2 | 28 | 3 |
| 3 | 36 | 6 |
| Average | 30 | 3 |

Table 2: Odometry accuracy during counterclockwise movement of a 2 m square.

| Iteration | $\lvert\text{Error}_x\rvert$ [cm] | $\lvert\text{Error}_y\rvert$ [cm] |
|---|---|---|
| 1 | 41 | 53 |
| 2 | 45 | 26 |
| 3 | 35 | 12 |
| Average | 40 | 30 |

Table 3: Odometry accuracy during clockwise movement of a 2 m square.

In order to prove the hypothesis that the number of turns has a correlation with the actual performance we also measured the errors for a straight line as well as with a single turn as shown in tables 4 and 5. As it was impossible to accurately measure the error in $x$ and $y$ during the straight line while still moving a reasonable distance, we calculated the hypotenuse of the position reported by the odometry and compared this with the distance traveled in the real world. Obviously, if the odometry is accurate these should be equal to each other. As can be seen in table 4 the error over almost four meters is less than 2 centimeters; compared to 40 centimeters over 8 meters in the case of the clockwise square above.

To further prove this, we also measured the error when moving the full line and coming back again; as this is the same method as with the square but with a single 180° rotation. As shown in table 5 the error is still small compared to above, though slightly bigger than in the case of no turns. This, in addition to our visual observation of the robots movement and the swivel wheel specifically shows that the error in measurements is directly proportional to the number of direction changes in the path.

| Iteration | Measured [m] | Odometry [m] | \|Error\| [cm] |
|---|---|---|---|
| 1 | 3.84 | 3.81 | 2.97 |
| 2 | 3.52 | 3.51 | 0.64 |
| 3 | 3.47 | 3.45 | 1.96 |
| 4 | 3.96 | 3.94 | 1.79 |
| 5 | 3.95 | 3.93 | 1.84 |
| Average | | | 1.84 |

Table 4: Odometry accuracy during straight line movement.

| Iteration | $\lvert\text{Error}_x\rvert$ [cm] | $\lvert\text{Error}_y\rvert$ [cm] |
|---|---|---|
| 1 | 0 | 7 |
| 2 | 2 | 15 |
| 3 | 6 | 0 |
| Average | 3 | 7 |

Table 5: Odometry accuracy during two-way movement along straight line.

## References


[1] Arduino LLC, "Arduino - arduino due." https://www.arduino.cc/en/Main/ArduinoBoardDue. (Last Visited on 01/10/2016).

[2] Arduino LLC, "Arduino - arduino motor shield." https://www.arduino.cc/en/Main/ArduinoMotorShieldR3. (Last Visited on 01/10/2016).

[3] Arduino LLC, "Arduino - arduino wifi shield." https://www.arduino.cc/en/Main/ArduinoWiFiShield. (Last Visited on 01/10/2016).

[4] NURDSpace, "Esp8266 - nurdspace." http://nurdspace.nl/ESP8266, November 2014. (Last visited on 01/10/2016).

[5] Reductor Motor, "Rh158.12 , rh158.24 micromotors." http://www.reductor-motor.com/eng-micRH158.htm. (Last Visited on 01/17/2016).

[6] MattS-UK, "Arduino wificlient (tcp)." http://mssystems.emscom.net/helpdesk/knowledgebase.php?article=51, November 2013. (Last visited on 01/10/2016).

[7] Group discussion, "Wifi shield tcp to webserver is very slow." http://forum.arduino.cc/index.php?topic=123824.0, September 2012. (Last visited on 01/10/2016).

[8] M. Floßmann and T. Olsson, "Laboration 1: PID Controls," Nov 2015. unpublished.