[ELE00118M] MSc Practical Robotics (PRAR)

[ELE00103M] Year 4 Robotics module

# Lab Session 5 Handout

## Lab 5: Wheel Odometry

Lecturer: Mark A Post (mark.post@york.ac.uk)

Technician: Mike Angus

## 1. Aims and Objectives

In this laboratory session, you will add the use of wheel encoders on your robot and use them to improve movement control.

## 2. Learning outcomes

- Understand wheel encoders and how they can be used for robot control
- Measuring distance in a program using GPIO pin feedback
- Control of a robot in prescribed patterns using feedback from wheel encoders

## 3. Software and hardware

You will improve your C or Python programs using the Raspberry Pi with robot kit parts and exercises from previous lab sessions.

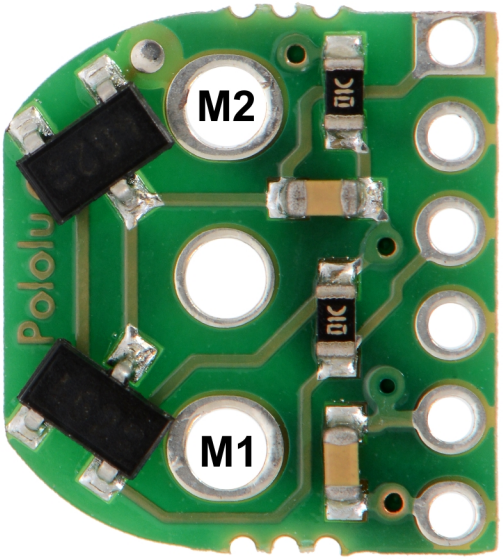
Your motor has a small rotary quadrature encoder on the back shaft opposite the gearbox. The pinout of the encoder is shown below and the datasheets and information on using this encoder are online here:

https://www.pololu.com/product/3081/resources

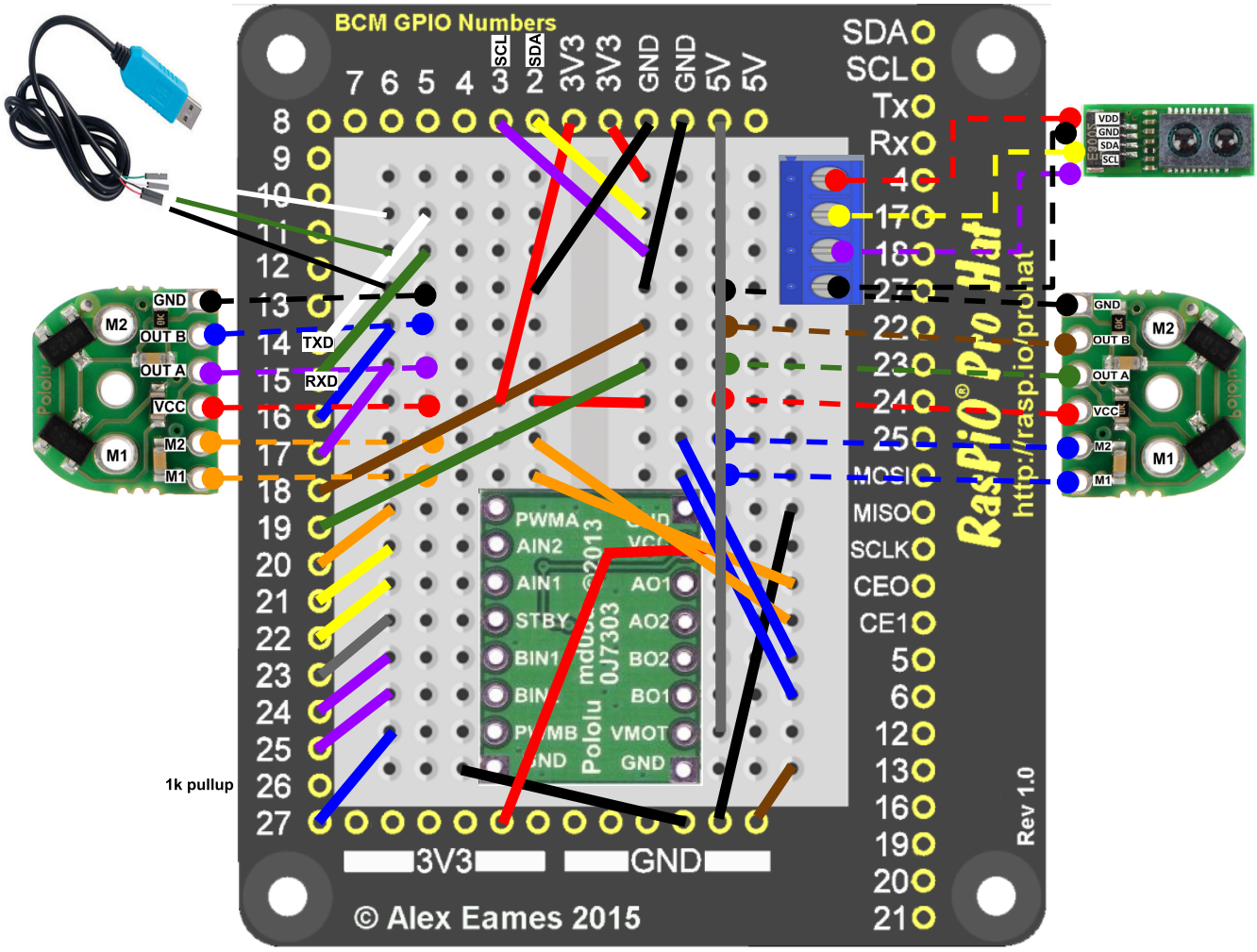
The encoder board with pins specified is below along with a suggested wiring layout. While assembling parts on your robot please refer to the “Wiring Instructions” section starting on page 50 of the document “**Home Robotics Kit 2020**” for guidance and tips on how to build a good quality robot.

Hall Sensor B

Hall Sensor A




GND
OUT B
OUT A
VCC
M2
M1



# 4. Pre-Lab Preparation

- Look up the datasheets for the encoders at https://www.pololu.com/product/3081/resources and become familiar with how they should be connected up to a microcontroller
- Review the theory of quadrature encoders to understand how to use the digital information from a quadrature encoder to determine both speed and direction.

# 5. Tasks

### *TASK 1: WIRE UP THE WHEEL ENCODERS TO THE RASPBERRY PI*

A suggested wiring layout that includes the new connections for the encoders is given above. Using the datasheets and your knowledge about rotary quadrature encoders, connect the *OUTA* and *OUTB* outputs from the encoder to your Raspberry Pi. Power the encoder using the *Vcc* and *GND* terminals from the 3.3V rails on the RasPiO Pro Hat.

*NOTE: be especially careful not to mix up the motor drive voltage pins M1 and M2 with the encoder outputs OUTA and OUTB as the 5V drive voltage is dangerous to GPIO pins.*

### *TASK 2: READ IN THE ENCODER COUNTS*

Building on your motor control code in C or Python from Lab 4, you will now need to add the *OUTA* and *OUTB* connected pins from the encoders as GPIO inputs. As before, some C examples are provided that can be easily replicated in Python if desired. It is suggested to map the pins *OUTA* and *OUTB* use to *AENC1* and *AENC2* for motor A and to *BENC1* and *BENC2* for motor B, to make programming less confusing. The following defines do this assuming the pins from the recommended wiring diagram above are used:

```
#define AENC1 18
#define AENC2 19
#define BENC1 17
#define BENC2 16
```

You will need global variables (defined before any program loops) to keep track of the current encoder counts for each motor, such as the following *countA* and *countB*:

```
int countA = 0;
int countB = 0;
```

If your program were running continuously and sufficiently fast, you could read in the encoder counts by *polling* them, which means checking periodically to see if the state has changed, using a simple GPIO read within your program loop like below and checking if the state has changed (indicating an encoder count) since the last loop:

```
lastState = state;
state = gpioRead(AENC1);
if(state != lastState)
    countA = countA + 1;
```

However, polling in real-time systems such as robots is highly **not recommended**, Not only is it unnecessarily inefficient to have to check the state of the pin on each program loop, it will also not detect more than one encoder pulse per loop and may even ignore one if a full pulse has occurred in the time that a program loop has taken. Motor shaft encoders running at thousands of RPM often produce pulses much faster than program loops can catch them. We will therefore use an **asynchronous** approach to reading the encoders using *interrupt handlers* (functions that are instantly called by the program usually using hardware to detect pulses and without waiting to reach a read statement in the program loop)

Interrupt handlers in pigpio are called *Alert Functions* in C, for details and examples see http://abyz.me.uk/rpi/pigpio/cif.html#gpioSetAlertFunc. In Python they are more conventionally called *callbacks*, see http://abyz.me.uk/rpi/pigpio/python.html#callback for details and examples. We need to create two callback functions that are named for example *pulseAcallback()* and *pulseBcallback()*. We set these functions to be called as soon as the state of the *AENC1* or *BENC1* pins changes as follows:

```
gpioSetAlertFunc(AENC1, pulseAcallback);
gpioSetAlertFunc(BENC1, pulseBcallback);
```

The functions themselves do not need to read the state of the pin causing the alert – when pulseAcallback is called we already know that a pulse edge has occurred on *AENC1* and can update *countA* accordingly. We only want to count single pulses so we will not update *countA* unless a rising edge (*level = 1*) has occurred. Also, in order to determine the direction of the encoder we need to check the value of *AENC2* to see if it leads (high) or lags (low) the value of *AENC1* as in the these example C functions:

```
void pulseA(int gpio, int level, uint32_t tick)
{
    if(level == 1)
```

```
        if(gpioRead(AENC2))
            countA++;
        else
            countA--;
}
void pulseB(int gpio, int level, uint32_t tick)
{
    if(level == 1)
        if(gpioRead(BENC2))
            countB++;
        else
            countB--;
}
```

Add reading of both encoders to your program and have it print out the value of both motor encoder A and motor encoder B while rotating the encoder wheels forward and backward with your finger. Then try measuring the number of counts produced when the motors are running at low, medium, and high duty cycles. Are you catching each encoder count and are the number of counts reasonable for each speed?

**NOTE: Depending on your callback functions you may find a problem in C or python that adding direction detection (reading pin 2) takes too long and the callbacks/interrupts can't keep up with the number of counts produced at high duty cycles (because the callbacks don't have time to finish before the next one).**

This problem can be solved by minimizing the size of your callback function. Just **increment the counter and do nothing else** in the callback/ISR and see if you get more accurate encoder counts. To determine direction, you can assume that the last commanded direction to the motor is the direction the wheels are currently spinning (as the motors are almost impossible to back-drive while powered without breaking them)

### *TASK 3: MOTION ESTIMATION BASED ON ODOMETRY*

Wheeled robots often sense their distance and speed of movement by measuring the revolutions of their wheels using rotary encoders. Although this is rarely accurate over long distances, it is useful as a reference when fused with other sensors over short distances. Now that you are obtaining the number of counts from the A and B encoder outputs using rising edge-triggered interrupts, you need to estimate the overall speed and distance covered of the motors.

First, calculate how many ticks per centimetre your robot will move at. Each encoder pulse represents one-sixth of a full rotation of the motor shaft, for *p* encoder pulses. The gearbox on the motor has a ratio of 298:1. The wheels on the robot attached to the gearbox have a diameter of *d* (measure your own wheels to be sure), and the encoder wheels produce ***6 pulses per revolution on each Hall sensor*** (12 pulses total per revolution, but you don't need/want to record all 12 pulses on both channels) Therefore, *298 * 6* encoder pulses represent one full rotation of the wheel. Make an equation that represents this simple relationship and create global variables *distanceA* and *distanceB* in your program to represent the total distance travelled by the two wheels. You will need to reset these distances later on when estimating movement as the error steadily increases with movement and will quickly become inaccurate. Periodically update these variables based on the number of encoder counts.

**NOTE: make sure to experimentally check that your number of counts per motor shaft rotation, and per wheel rotation are correct, and if not adjust your function so that you can accurately measure number of rotations from the encoder**

Also *speed = distance/time* so you can calculate speed by taking the total distance travelled and dividing by the elapsed time. Create two more global variables *speedA* and *speedB* in your program and update them at the same. Don't do this in the encoder callback function as the length of time will be very short between counts and you need to make your callback function as short as possible – make global variables for the previous time and the current time and then subtract them to calculate elapsed time.

There are several ways to obtain the system time on the Raspberry Pi:

- the pigpio callback functions above pass in a "*tick*" variable which is the number of microseconds ($10^{-6}$ s) since boot, and you can keep track of the previous value and subtract it from each new value to calculate the elapsed time since the last encoder update.
- the pigpio library includes, in addition to time_sleep (which is an unreliable way of measuring time), the gpioSetTimerFunc function in C (http://abyz.me.uk/rpi/pigpio/cif.html#gpioSetTimerFunc) that allows you to set a callback that will be called at regular intervals. This function could be used to periodically update the encoders in a C program.
- the python "time" library provides easy functions to get elapsed time in floating-point seconds, see https://www.tutorialspoint.com/How-to-measure-elapsed-time-in-python

- the low-level C functions for getting time in Linux can be used as well, there are many tutorials on this such as https://www.tutorialspoint.com/how-to-measure-time-taken-by-a-function-in-c

Use a timer to count the number of encoder counts in a fixed period of time. and use a test to determine the direction that the motor is turning by checking whether A is leading B, or B is leading A. Output this speed and direction information to the PC through the serial terminal as you did with the range sensor.

Make your program run the motors forward on command for a given distance from the encoders, start at maximum speed, and then try subsequent runs at lower speeds. Use this program to make a table of measurements of motor speed level versus actual physical speed of the robot as measured by the encoders. Make sure you record enough measurements to answer the following:

- How does the actual speed vary with the commanded speed to the motor?
- How reliable is the motor speed? How much does it vary between runs and why?
- Using the encoders, how accurately can you make the robot travel a 10cm distance? How can you improve this accuracy?

Estimate the transformation between commanded speed and actual speed, and make a function that will reliably make the robot move at a desired speed in centimeters per second.

Now, modify your program with additional code to turn the robot by varying the difference between left and right wheel speeds. Use this program to make a table of measurements of set difference in speed between the two motors versus physical angular turning speed of the robot. Record enough measurements to answer the following:

- How does the turning speed vary with the difference in motor speeds. Is it linear, or if not what is the mathematical relationship between them?
- How consistent is the turning speed?
- Using the timer, how accurately can you make the robot turn $90^o$ (or $\pi/2$ radians)? How can you improve this accuracy?

Based on your recorded data, implement a move(speed, turn) function in your program to allow you to set a desired speed and turning rate based on the “best fit” to the data you have gathered. Test and retain this program with both linear movement and turning functionality for the next tasks.