Rotor speed measurements

**Summary:**Put reflective tape (we have some) on the motor casing. Get aQRE1113 optical sensor to count each time the tape passes by. Send that digital signal to a GPIO pin on the RPi. Use the sample python code to measure rotor speed. This may cause too much latency for actual flight, but for a single motor test, it should be sufficient.

There are several options for how to measure, estimate, or extrapolate rotor speeds. Matt Brown (similar to Pat but in BME) is a good resource for this. The options are (roughly from worst to best prognosis):

1. Back-EMF
   1. Still unclear how exactly you do this, and it doesn’t appear to be very direct or precise by most accounts. It was recommended in Mahoney, but not elsewhere.
   2. ChatGPT says you would use a current sensor, and we would have to do some computations to correlate the spikes in current with motor rotation to indirectly estimate motor speed. It may be less accurate and require filtering.
2. Bidirectional ESC
   1. This was suggested by Matt Brown, but I suspect it is inaccurate. The information I’ve found on bidirectional ESCs suggests that it’s an ESC that can operate the motor forwards and backwards, but is otherwise identical to a normal ESC, so provides no feedback information about rotor speed.
3. Hall effect sensor
   1. The motor is composed of several (probably 8) magnets, which rotate. A hall effect sensor mounted next to the motor could, in theory, register a pulse every time a magnet passes by.
   2. Matt Brown tried this with one of his spare hall sensors, and it failed. A different hall sensor may work, but unclear what parameters would need to change.
4. Encoder
   1. We could swap the motors for a motor with a built-in encoder. The downside is that these tend to be heavier and larger, and we would have to redo the motor parameter calibrations. Those calibrations were time-consuming the first time, but they should be quicker now that the procedure is developed.
5. ESCs with feedback
   1. ChatGPT seems to think some ESCs have built-in feedback either directly of RPM or of hall effect. I haven’t found any of these so far.
6. RPM sensor
   1. I’ve seen something that looks like an ESC but it’s an RPM sensor, [here](https://www.amainhobbies.com/hobbywing-rpm-sensor-hwa86060041/p453341?gad_source=1&gclid=CjwKCAiAudG5BhAREiwAWMlSjEF_wBF7MdS3Pok57UJvcvGmU0ONkwzUObOhpZO9lE81yxkUKAUyDBoCXFkQAvD_BwE). It doesn’t have much documentation, but it’s cheap, so it may be worth getting just to see if it’s an easy solution.
7. Drive current frequency
   1. Matt Brown suggests attaching a sensor (voltage? Current?) to one of the output wires of the ESC to measure the frequency that’s driving the motor. This should be directly proportional to the rotor speed, except for the slip angle.
   2. It’s unclear what sensor to use and whether the slip angle will cause significant deviations between this measurement and the actual rotor speed.
8. Optical sensor
   1. We could put a strip of reflective tape (we already have some) on the motor, and mount an optical sensor facing the motor, so that it registers a blip every time the rotor completes a revolution.

Optical Sensor Options

* 1. Infrared Reflective Sensors (e.g. TCRT5000 or QRE1113)
     1. Compact, affordable, easy interface with RPi. May be sensitive to ambient light.
  2. Photoreflector Modules (e.g. KY-033 Module)
     1. Built specifically for detecting reflective surfaces. Could be oversensitive in fluctuating light conditions (which may include shadows of rotor blades). Digital output for each pulse.
  3. Laser Reflective Sensors (e.g. VL53L0X Time-of-Flight Sensor)
     1. Precise, but complex, expensive, sensitive

Suppose we use the [QRE1113](https://www.sparkfun.com/products/9454) (other sensors would have similar usage). Some sample python code from ChatGPT:

*import RPi.GPIO as GPIO*

*import time*

*# Set up the GPIO mode*

*GPIO.setmode(GPIO.BCM)*

*GPIO.setwarnings(False)*

*# Define the GPIO pin connected to the OUT pin of QRE1113*

*SENSOR\_PIN = 17*

*GPIO.setup(SENSOR\_PIN, GPIO.IN)*

*# Variables for measuring RPM*

*pulse\_count = 0*

*last\_time = time.time()*

*rpm = 0*

*# Callback function to count pulses*

*def pulse\_detected(channel):*

*global pulse\_count*

*pulse\_count += 1*

*# Attach an event to detect rising edges*

*GPIO.add\_event\_detect(SENSOR\_PIN, GPIO.RISING, callback=pulse\_detected)*

*try:*

*# Run indefinitely*

*while True:*

*# Calculate RPM every second*

*current\_time = time.time()*

*elapsed\_time = current\_time - last\_time*

*if elapsed\_time >= 1.0: # Every 1 second*

*# RPM calculation (for 1 pulse per rotation)*

*rpm = (pulse\_count / elapsed\_time) \* 60 # Convert to RPM*

*print(f"RPM: {rpm}")*

*pulse\_count = 0*

*last\_time = current\_time*

*# Small delay to avoid excessive CPU usage*

*time.sleep(0.01)*

*except KeyboardInterrupt:*

*# Clean up on CTRL+C*

*GPIO.cleanup()*

To adjust this to our purposes, in every iteration of our control loop, we would read the pulse count and then reset the pulse count to 0. So we would only estimate the pulses every time we need to update the control action.

Now, this code uses an interrupt flag every time the pulse is detected, which is several thousand times a second. During those interrupts, the RPi has to stop what it’s doing to update the pulse count. This might significantly slow down the control loop, causing unacceptable latency. If that’s the case, we would want to add in an integrated circuit that could take the digital signal from the QRE1113, count the pulses continuously, and periodically update its output, which would be the rpm in I2C communication.

The simplest solution to this appears to be to use a *digital frequency-to-voltage converter with digital readout,* like the[AD7740](https://www.analog.com/en/products/ad7740.html#part-details) (careful with this link, you’ll want to find a version that’s soldered to a breakout board so you can solder it easily). This should measure the frequency of pulses and converts it to a hexadecimal value, which can be communicated over UART or I2C.

**However**. Our first goal is just to get data from a single motor, not necessarily connected to the drone. For that, we don’t need the feedback loop to work. We just need to take open-loop measurements. So for this first experiment, ignore the AD7740, and just get it working with the QRE1113 and the sample code. When we actually need to fly, if the lag is too much, then we can add in the AD7740.