ENG 8905 - Quadrature Encoders Part 1 The Knob

Kyle Chisholm

2022-05-01

Decoding a quadrature incremental encoder to measure position of a knob. A custom implementation of the decoding logic is compared against a published Python package.

Submission Deadline: May 01, 2022

Online version of this document: https://chishok.github.io/ENG8905/ENG8905-Lab7

Background

Incremental encoders are everywhere, used for high-precision motion control applications such as robotics or radar systems and also for simple interfaces like volume knobs. A rotary encoder repeats a signal sequentially "high" then "low" as it rotates. The signals are generated by various physical means, including magnetic, optical, and mechanical interfaces.

An **optical encoder** involves placing a light emitter and a received on either sides of an opaque disc, with open slots cut out at regular angle intervals. When the disc rotates, the receiver will register the light in a sequential on and off pattern as shown in Figure 1.

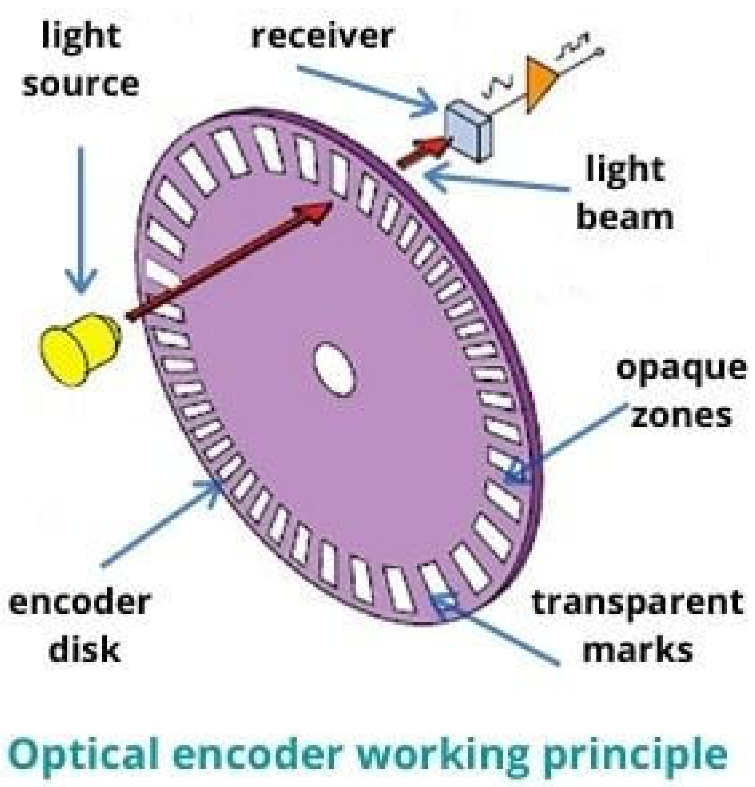


Figure 1: Optical Encoder

The rising and/or falling edges of the one/off square wave pattern are counted by a **decoder**, which can be implemented as a simple integrated circuit or part of microcontroller code. In this lab we will be using the Raspberry Pi GPIO and Python to count encoder signal edges.

A **mechanical encoder** uses physical conductive contacts that brush up one another to generate the on and off (high and low) signals as it rotates. This lab will utilize a mechanical encoder KY-040 shown in Figure 2 .

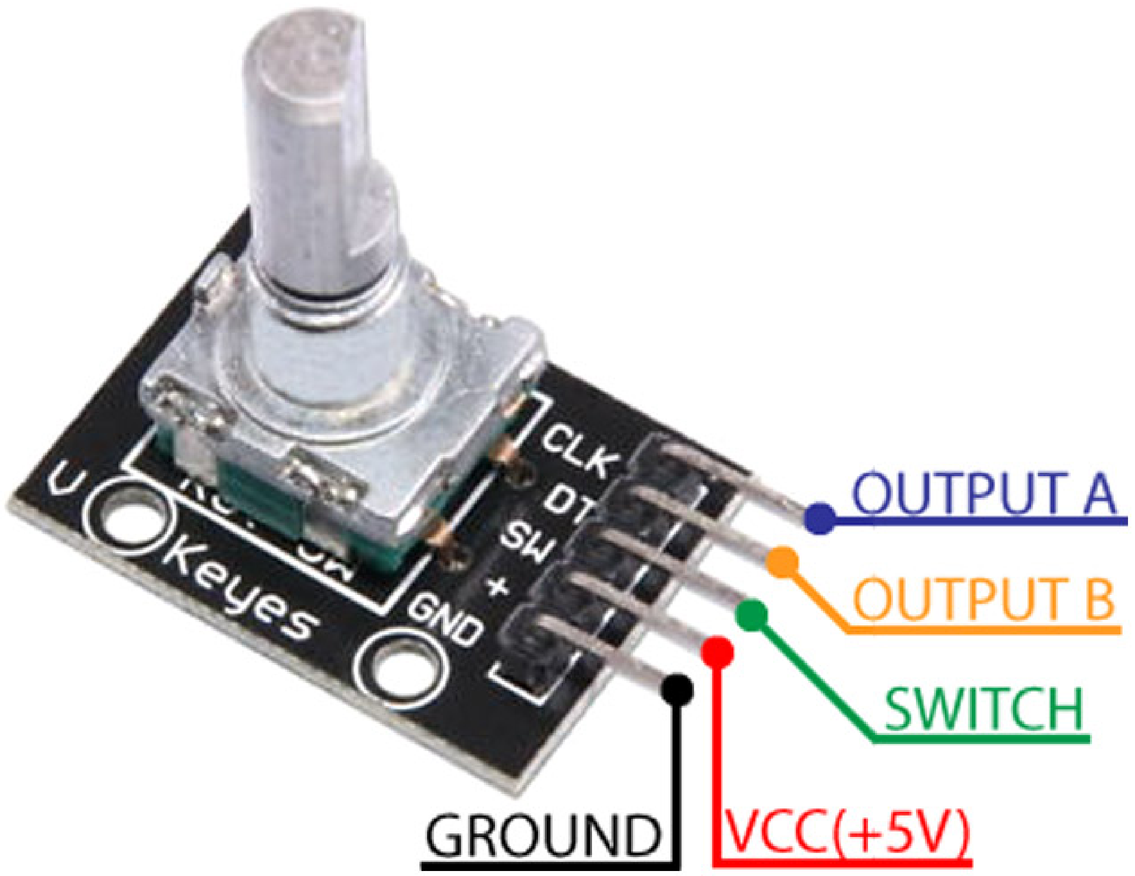


Figure 2: Mechanical Encoder KY-040

Compared to optical encoders, mechanical encoders are much cheaper but also more prone to noise associated with the mechanical interface. We will see the effect of mechanical noise on the ability to capture the knob position as it's rotated.

Magnetometers or hall effect sensors and rotating magnetic fields are also used for rotational sensor encoding, especially in 3-phase DC motor commutation. Encoded magnetic strips are also used in linear sensing applications. This lab will be focused on a low-cost mechanical rotary encoder and the next part will use a hall effect rotary encoder on a DC motor.

Quadrature and Absolute Encoders

The disadvantage of using an incremental encoder is that the pulsing provided by the sensor in one direction is indistinguishable from motion in the opposite direction. Single channel incremental encoders are more useful for applications with one direction of motion.

For position feedback and determining the direction of rotation, a **quadrature encoder** solves the problem by adding a row of on/off signals in the physical interface with an offset such that the direction can be determined. Refer to Figure 3 which shows the operation of a quadrature encoder and the resulting two signals A and B that follow from a clockwise rotation. Figure 4 shows the resulting signal from rotation in clockwise and counter-clockwise directions.

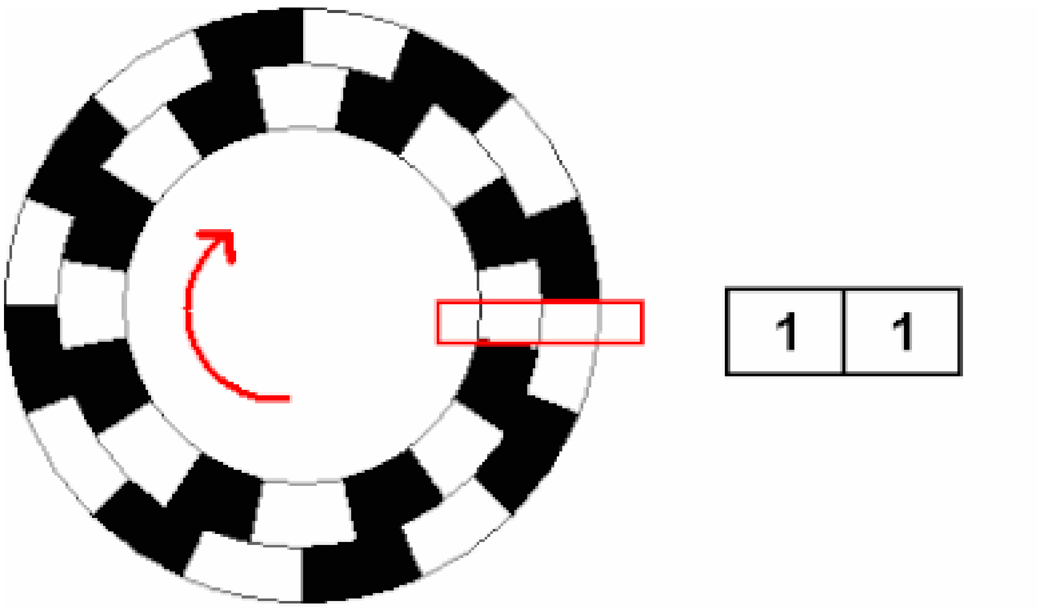


Figure 3: Quadrature code change in clockwise direction

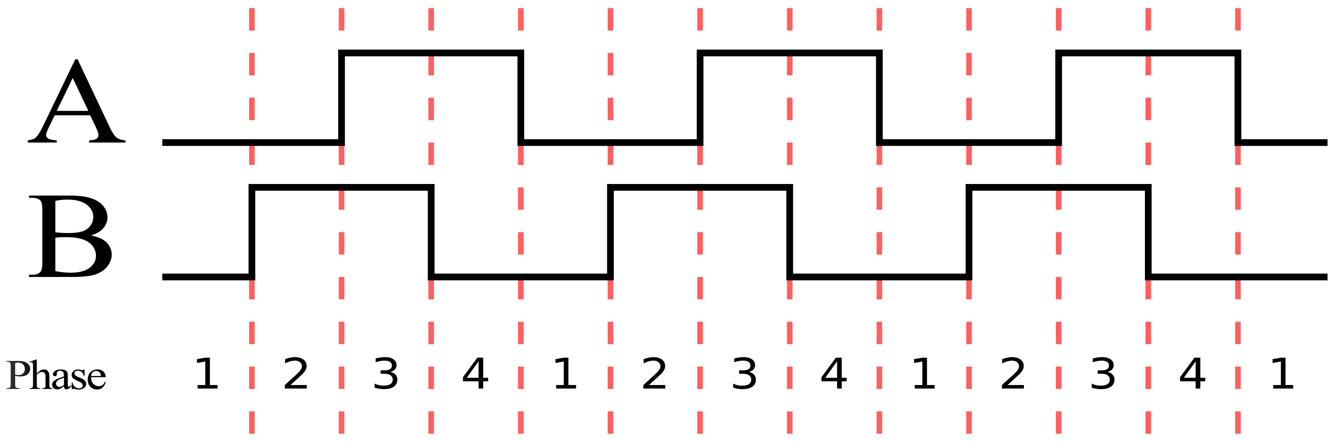


Figure 4: Quadrature A/B signal pulses

Gray-Code Decoding

When the A and B signals arrive from a quadrature encoder, we need to interpret the sequence of high and low signals from both lines to generate a change in value. This is the decoding process.

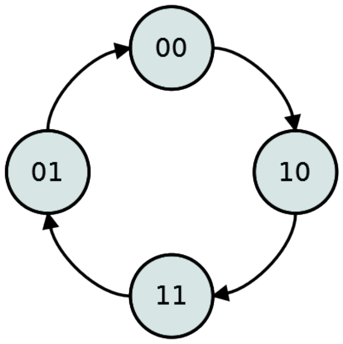
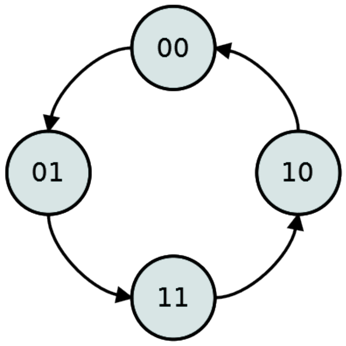
It is convenient to interpret the two signals in an incremental quadrature encoder as a 2-bit code. The first least significant bit (LSB: the right-most bit in most programming languages) signifies the B value as either ON (1) or OFF (0). The next bit contains the value of the A signal. We can therefore assign a change in value according to the sequence of bits that are received from the encoder.

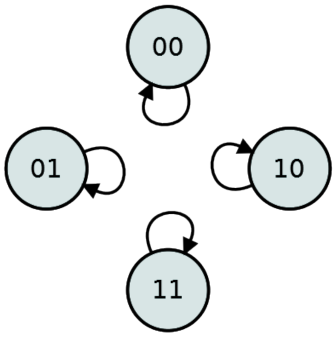
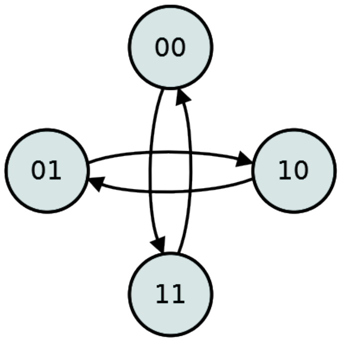
The progression of the 2-bit code is as follows:

Direction	Sequence
CW	00 → 01 → 11 → 10
CCW	00 → 10 → 11 → 01

From these state transitions, given the current state of the A and B inputs, and the next state when either signal changes, we can compute the increment of pulse count in positive or negative direction. With only one bit changing at a time, this sequence is referred to as a 2-bit **Gray Code**.

However, this sequence is only valid if one bit changes. If two bits change at the same time, it is an invalid sequence where there is no way to tell if there was a clockwise or counter-clockwise turn to reach this state. The following Table describes the possible state transitions. For the invalid transitions, instead of aborting the process, we can make an assumption that the direction is the same as the previous transition but advances by 2 steps in the direction of travel instead of just one.

Transitions	Diagram
Forward Movement (+1)	
Reverse Movement (-1)	

Transitions	Diagram
No Movement (0)	
Invalid Motion (±2)	

The number of edges counted (state transitions) per each revolution of the rotary encoder provides the resolution of the encoder in **Counts Per Revolution (CPR)**. A higher resolution may provide better precision of an angle measurement, but if the encoder is spinning too fast, the frequency of the pulses that need to be counted per second increases and may exceed the capabilities of the decoder (counter) and counts can be missed.

In the lab with the motion control application (part 2), we will investigate the relationship between the motor RPM and pulse frequency of the encoder. They are important factors in designing a feedback control system using rotary encoders. For this lab, we not as concerned with keeping an accurate absolute position count and missed steps are tolerable.

Materials

The two modules connected to the GPIO for this lab are a rotary encoder input and a simple readout display. The list of materials are provided in Table 1.

> **NOTE**: Many Adafruit or Sparkfun branded modules have generic versions that may be more affordable and/or have faster shipping from alternative suppliers.

Wire up the components as shown in Figure 5. The encoder A and B wires are connected to GPIO 6 and 13, respectively. The encoder push-button switch is connected to GPIO 5 but is unused in this lab. The OLED display is an I2C device and is connected to SDA and SCL pins on the GPIO. Power is only 3.3V, be careful to never use the 5V power unless it's for an external device not connected to the GPIO (such as a fan).

Table 1: List of materials.

Item	Description
Raspberry Pi	Raspberry Pi 4, 3B, or Zero W2
SSD1306 I2C OLED 128x32 Display	A simple display for readout
KY-040 Quadrature Encoder	A knob for manual incremental encoder counting
Breadboard	For prototyping
Jumper Wire	For prototyping
USB Power Supply	Power for the Raspberry Pi

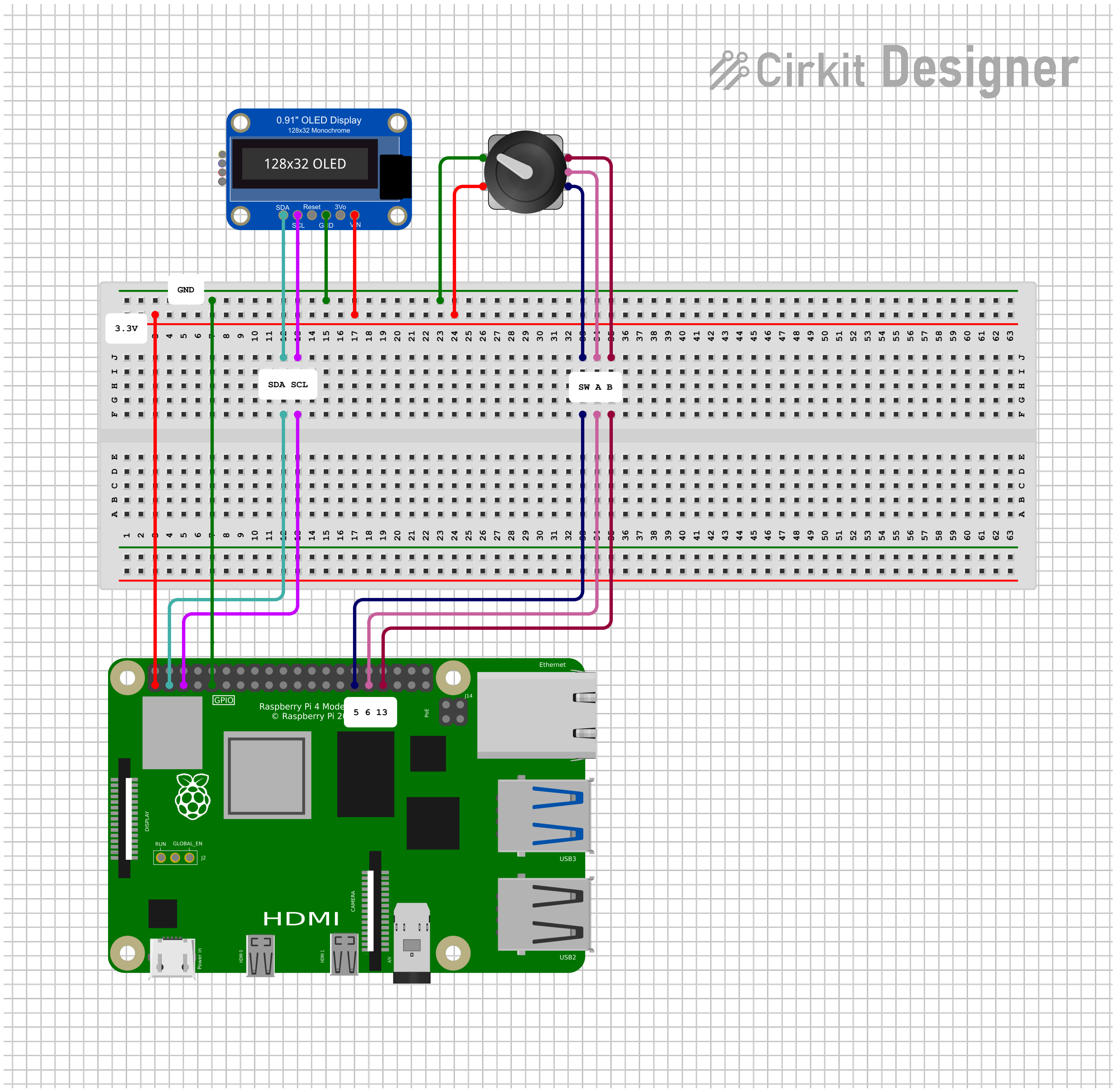


Figure 5: Wiring Diagram

Methods

Raspberry Pi System Software Prerequisites

1. Install python and git

   ```
   sudo apt install -y \
       git \
       python3 \
       python3-pip \
       python3-venv \
       python3-setuptools \
       python3-flake8 \
       python3-flake8-docstrings \
       python3-autopep8 \
       python3-rpi.gpio
   ```

2. Enable i2C in `raspi-config`

   ```
   sudo raspi-config
   ```

   Set `3 Interface Options -> I5 I2C` to `Yes`.

   Check that i2c is available

   ```
   ls -l /dev/i2c*
   ```

   Show connected devices

   ```
   i2cdetect -y 1
   ```

3. Update i2C bitrate

   ```
   echo "dtparam=i2c_baudrate=1000000" | sudo tee -a /boot/config.txt
   ```

   Reboot the module.

4. Update bash config with local directory `~/.local` added to `PATH` environment variable

   ```
   mkdir -p "${HOME}/.local/bin"
   echo "source ${HOME}/.bashrc" >> ${HOME}/.bash_profile
   echo "export PYTHONPATH=\$PYTHONPATH:/usr/local/lib/python3.9/site-packages" >> ${HOME}/.bash_profile
   echo "export PATH=\$PATH:${HOME}/.local/bin" >> ${HOME}/.bash_profile
   ```

Lab Software Requirements

On the Raspberry Pi, install matplotlib for plotting, pillow library for LED screen imaging, and scipy with numpy for numerical processing.

```
sudo apt install -y \
    python3-scipy \
    python3-numpy \
    python3-matplotlib \
    python3-pil
```

Install Python packages Encoder and adafruit-circuitpython-ssd1306 for the OLED display hardware interface.

```
python3 -m pip install --user \
    Encoder \
    adafruit-circuitpython-ssd1306
```

Use VS Code with Python extension for development.

> **NOTE**: If you do not want to use the Raspberry Pi desktop environment, use VS Code on your personal computer with remote development extension pack to develop on the pi over a local SSH network connection.

Test The Hardware and Run Python Scripts

Check out the git repo https://github.com/chishok/ENG8905-Lab7 and open the folder in VS Code.

1. Get familiar with how to **use the `Encoder` python package and the GPIO**. Read and run the script scripts/DatalogPlotSample.py:

   ```
   python3 scripts/DatalogPlotSample.py
   ```

   A printout in the terminal will show the current value of the encoder at a regular interval (10Hz refresh rate). Rotate the encoder knob and the updated count will be displayed. The package which was installed to count the encoder is hosted on GitHub here: https://github.com/mivallion/Encoder. The next task in this lab is to implement our own decoding logic and compare it against this package.

2. **Use the display**. Read and run the script scripts/DisplaySample.py:

   ```
   python3 scripts/DisplaySample.py
   ```

   You should see the text "Hello World!" displayed on the small OLED display with a border surrounding the text. Inside the script, change the line

   ```
   text = "Hello World!"
   ```

   to another message and re-run. Do you see the updated text?

3. Generate **display output of running encoder in a timed loop**. Read and run the script scripts/EncoderAndDisplaySample.py:

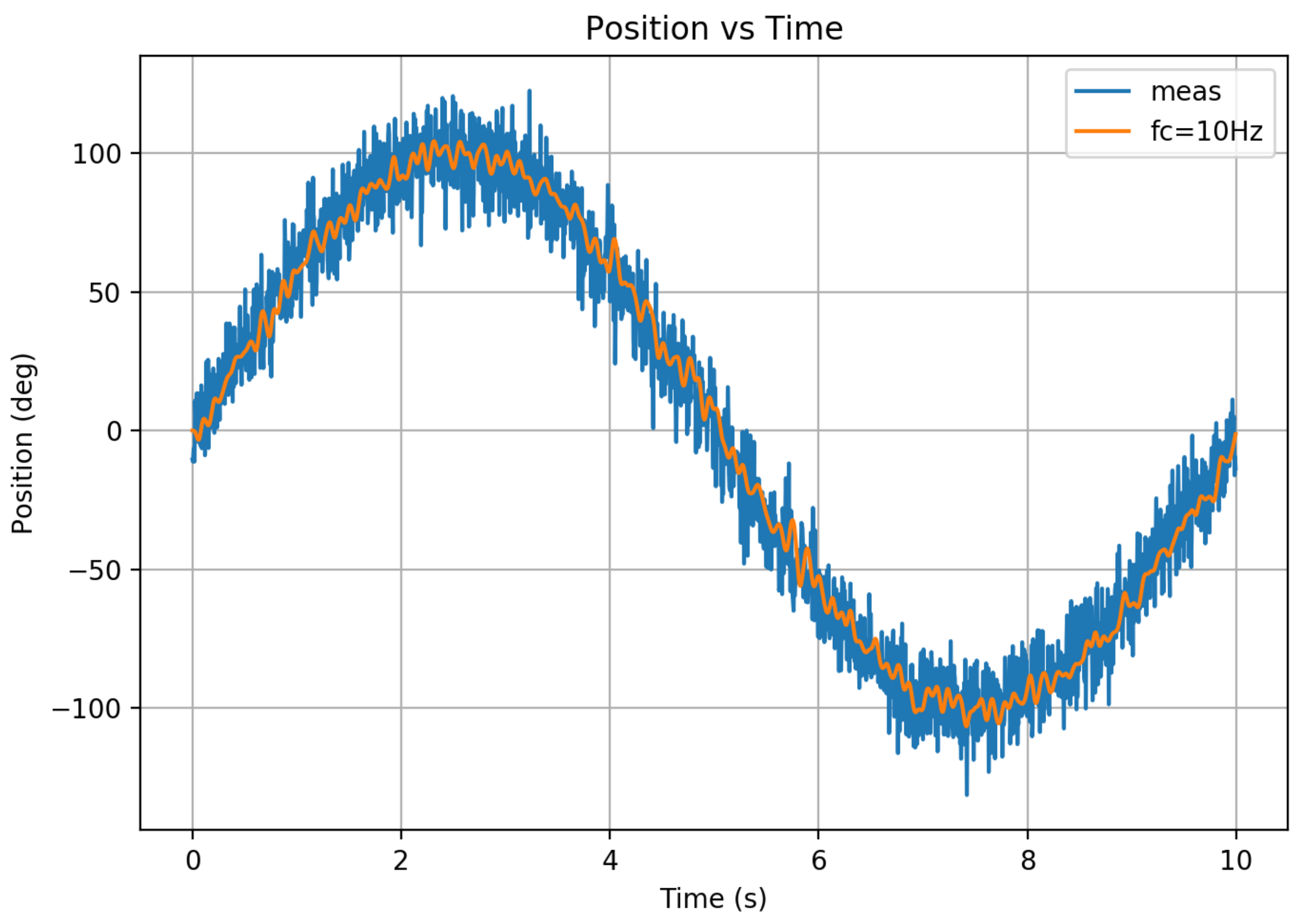
   ```
   python3 scripts/EncoderAndDisplaySample.py
   ```

   Rotate the knob and you should see the values updated on the OLED display.

4. Save, load, and plot data. Read and run the script scripts/DatalogPlotSample.py:

   ```
   python3 scripts/DatalogPlotSample.py
   ```

   This will generate an image file of the plot called `plot_output.png` with simulated noisy sinusoidal and filtered data. The plot should look like:



Sample plot of filtered sinusoidal

The code samples in the scripts folder have been repurposed in a folder named encoder which contains multiple python files and classes. You are encouraged to read through the entire source code, but the focus of the lab will be to modify a portion of the encoder/quadrature.py file which handles the quadrature encoder counting.

Coding Task: Decode Quadrature Input

Run the encoder module. Make sure the current directory is the root of the workspace (project folder) and create a `data` subfolder then run the module

```
mkdir -p data
PYTHONPATH=. python3 encoder --output "data/dataLogEncoder"
```

> **NOTE**: The environment variable `PYTHONPATH=.` before running python ensure the module folder `encoder` is on the python's search path.

When the command is run, the encoder and display are active for 15 seconds. While it's running, rotate the knob and watch the output angle values. Figure 6 shows a sample of what the readout looks like. Do the readout degrees correspond to the rotation of the knob? At the end of the duration, the program writes out data and exits. The data folder should contain two plots and a data file:

```
data/dataLogEncoder_position.png
data/dataLogEncoder_timing.png
data/dataLogEncoder.npz
```

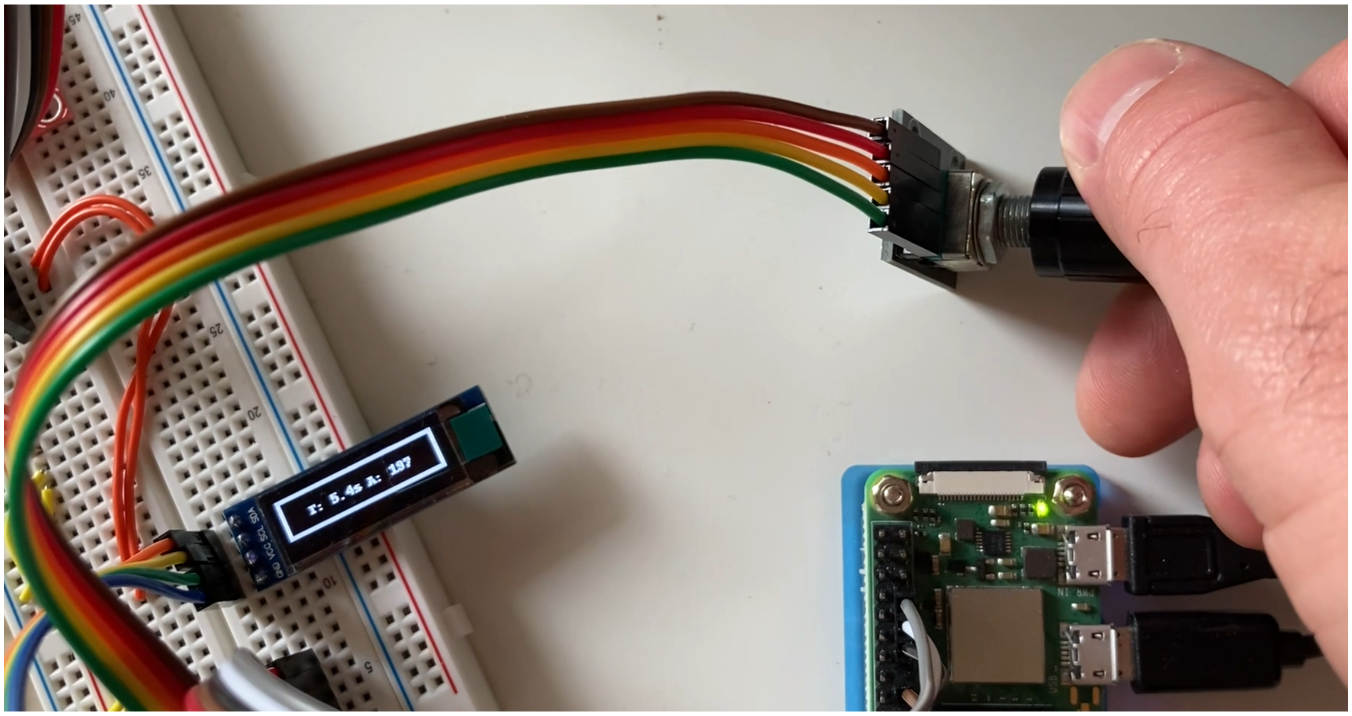


Figure 6: Data acquisition readout.

Inspect the file encoder/configuration.py to see what the default configured values are for the encoder nad display. Some command line arguments override the configured default values. Inspect the file encoder/__main__.py to see what arguments can be passed when running `python3 encoder`.

Modify the source file encoder/quadrature.py to implement the quadrature decoding logic in between the comments

```
# STUDENT WORK SECTION BEGIN
# ==========================
```

and

```
# STUDENT WORK SECTION END
# ========================
```

Re-run the program with the argument `--custom-decoder` to test your implementation:

```
PYTHONPATH=. python3 encoder --output "data/dataLogCustom" --custom-decoder
```

Data Collection

Run the data collection script

```
./scripts/Lab7/scripts/run_data_collection.sh
```

Follow the prompts in the terminal. You will be prompted to turn the encoder knob to +360 then reverse to -360 and back to 0 over 15 seconds a total of 4 times. Don't start to turn the knob until the printout The following files are generated when the script is completed:

```
data/dataLogEncoder_position.png
data/dataLogEncoder_timing.png
data/dataLogEncoder.npz
data/dataLogCustom_position.png
data/dataLogCustom_timing.png
data/dataLogCustom.npz
data/dataLogEncoder200Hz_position.png
data/dataLogEncoder200Hz_timing.png
data/dataLogEncoder200Hz.npz
data/dataLogCustom200Hz_position.png
data/dataLogCustom200Hz_timing.png
data/dataLogCustom200Hz.npz
```

Analysis

Example output

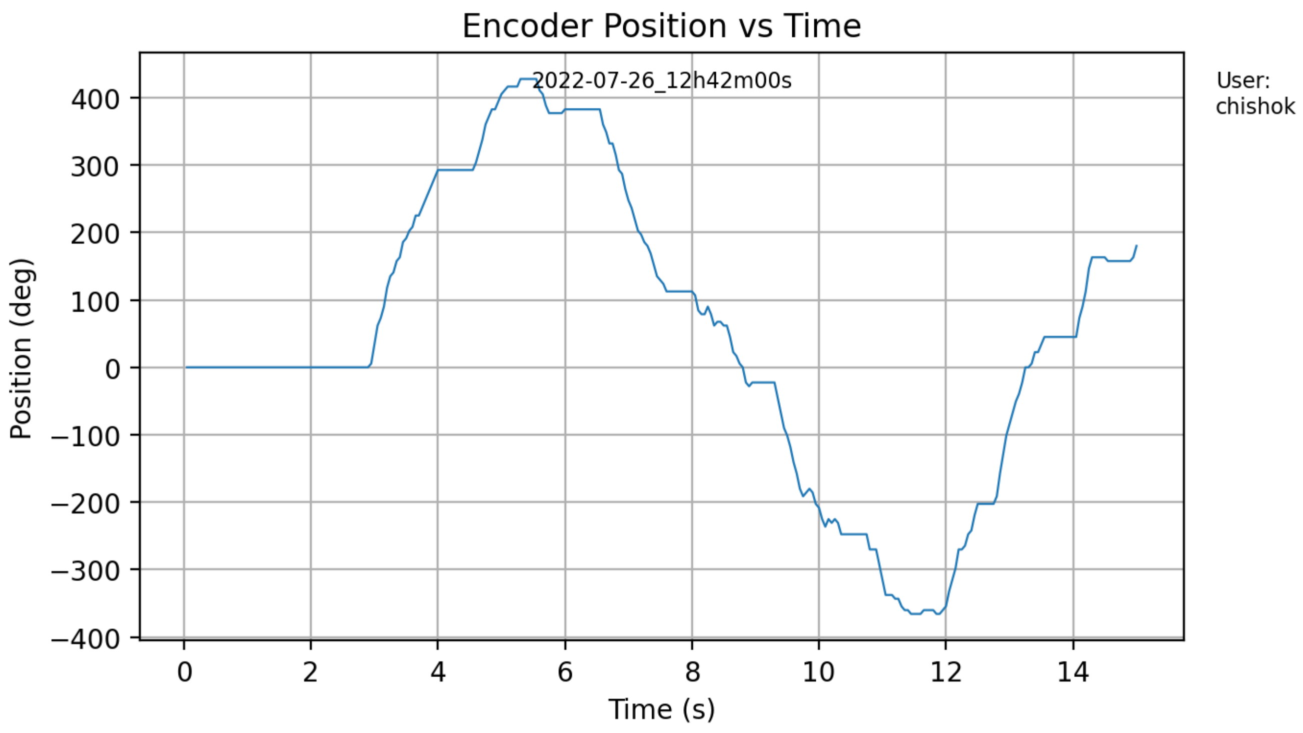


Figure 7: Example of position data output at 50Hz sampling rate.

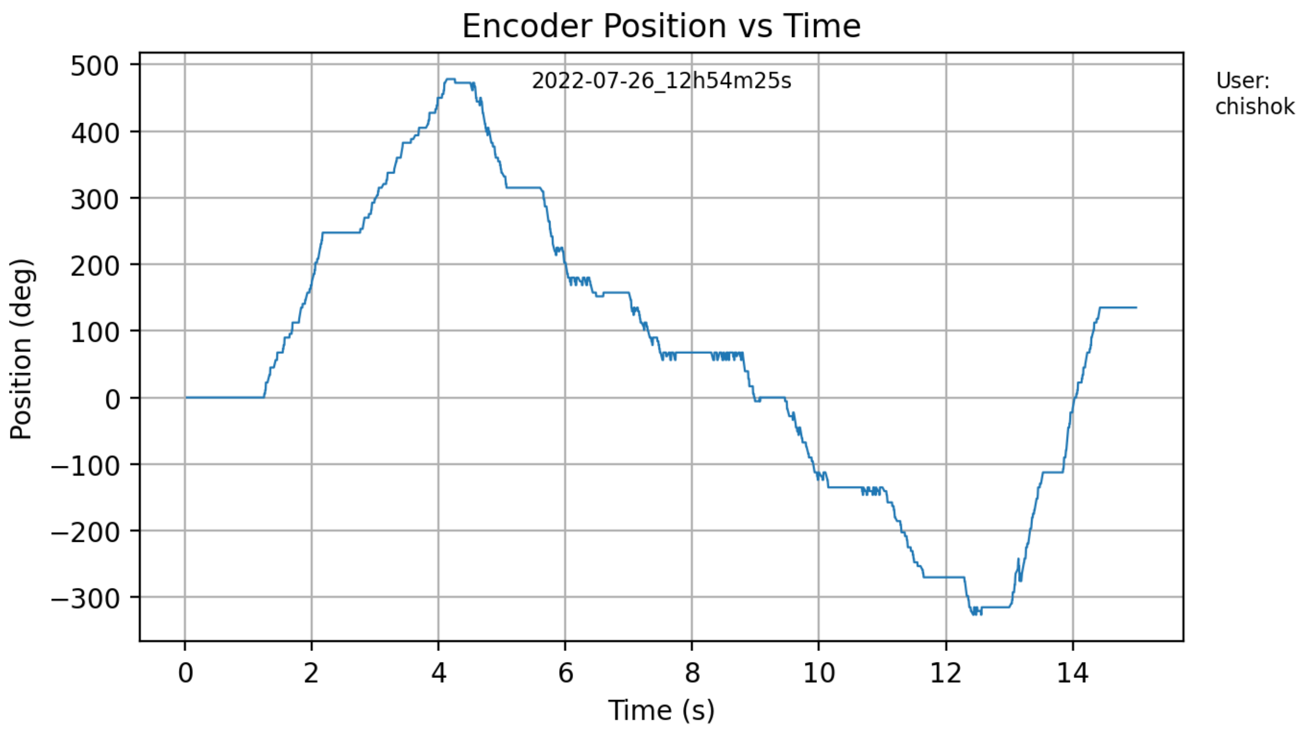


Figure 8: Example of position data output at 200Hz sampling rate.

Submission Guidelines