Bothell Blue

Washington State Competition

Bothell High School

Detector Building

Alex Metzger and Samarth Venkatesh

C-05

Detector Building C

Specification Sheet – Version 1

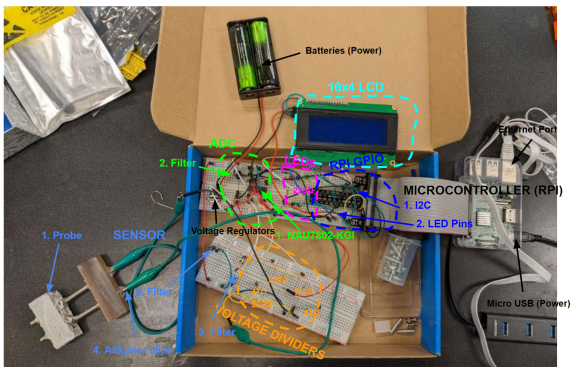
SCHOOL NAME	Bothell High School
TEAM NAME	Bothell Blue
TEAM NUMBER	C05
STUDENT NAME(S)	Alexander Metzger
	Samarth Venkatesh

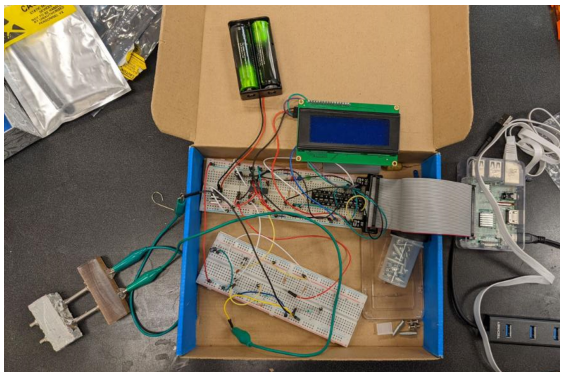
Instructions: In the following sections, follow the instructions for either inserting photographs or typing your responses. Please type your responses only in the white boxes. Any responses in gray boxes will not be read. To ensure any formatting, images, and/or figures (e.g. arrows, boxes) you add are preserved, please save and upload your completed specification sheet as a PDF with filename in the format *TeamNumber_DetectorBuilding_SpecSheet.pdf*.

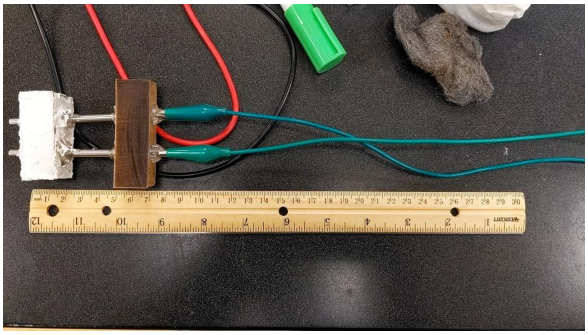
For the event **Detector Building C**, you will record your device testing video with the "test" in Scilympiad open, and you will document your device and results from your device testing in this specification sheet. You will submit this specification sheet with your device testing video via a Google Form.

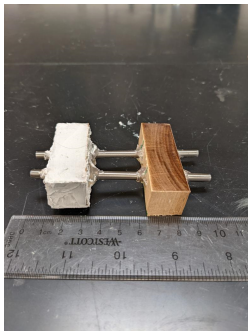
Device Photographs

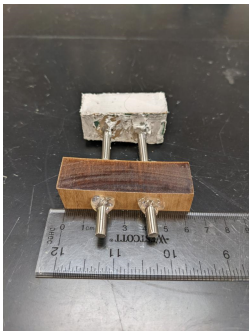
In the space below, insert photographs of your Device. You should include photographs that show the entire Device from multiple angles (top, each side, and isometric views) as well as ones that are close-up to show details so that the Event Supervisor can verify the compliance of your Device with the event's rules.

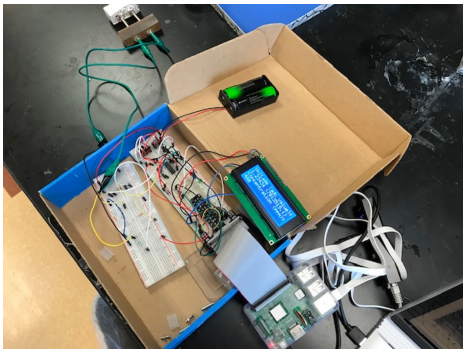


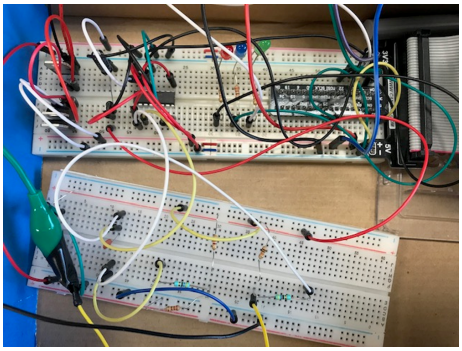


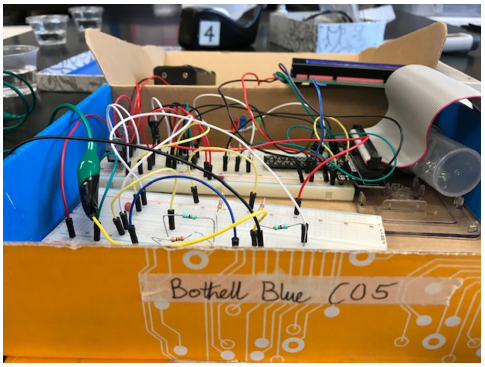










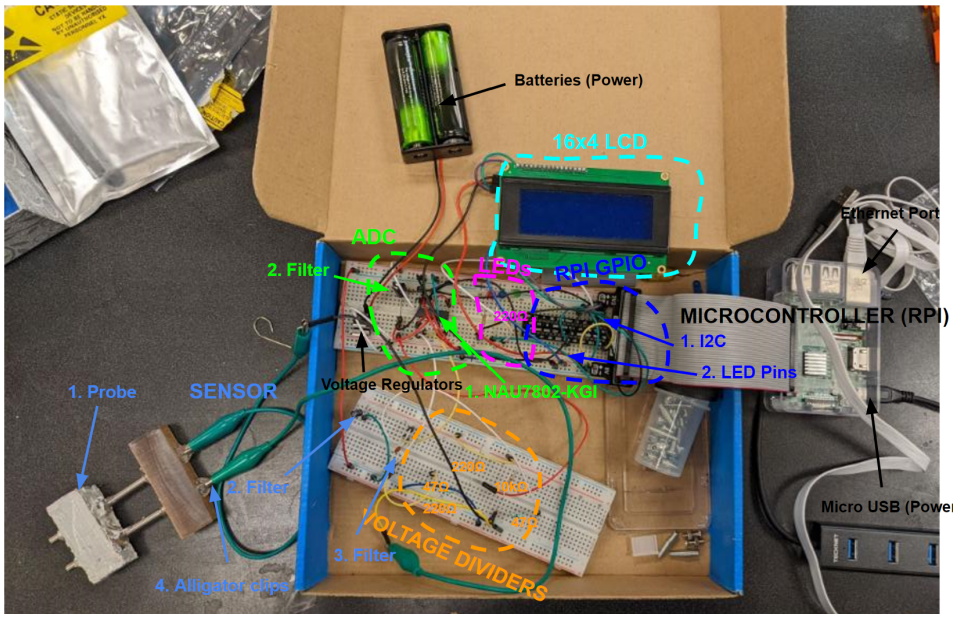


Design Log

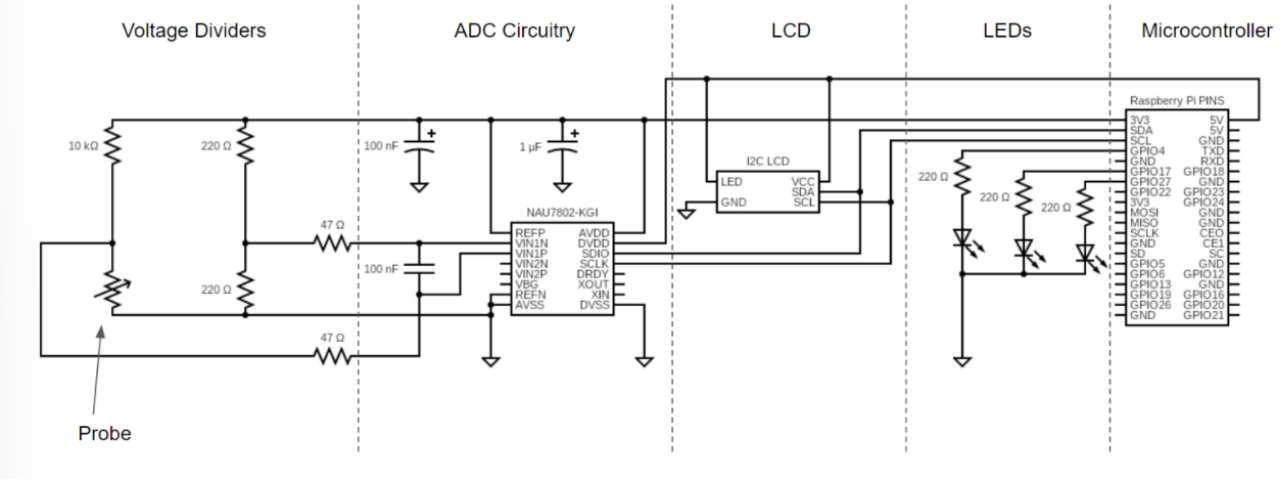
In the space below, insert images (e.g. scans, photographs, screenshots) of your entire Design Log.

Labeled Diagrams and Working Principle

Labeled Photograph



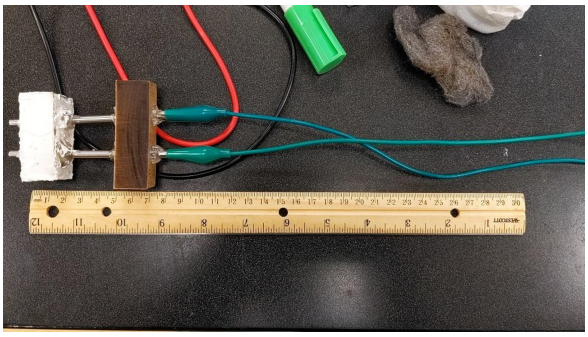
Schematic

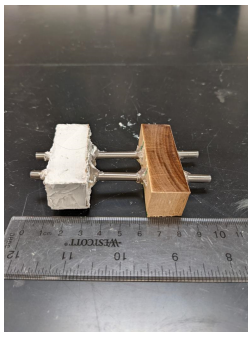


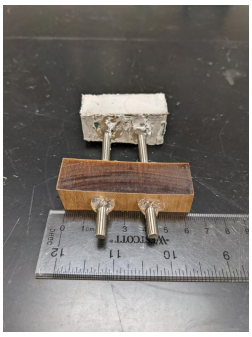
Working Principle

The SENSOR consists of a probe, some filters and two VOLTAGE DIVIDERS. The probe is a primitive 2 electrode conductivity probe which is submerged in the solution. The resistance across the probe is then measured by the rest of the circuit and the conductivity and concentration calculated. The filters are simple capacitor filters which remove noise by letting higher frequency signals go to ground and preventing lower frequency signals from going through due to the high impedance. The VOLTAGE DIVIDERS act like a Wheatstone Bridge, producing a potential difference between the two legs (dependent on the variable resistance of the probe) which is then measured by the differential ADC (the NAU7802 chip) by feeding the voltage divider containing the probe to the positive input of the ADC (VIN1P) and the other voltage divider to the negative input (VIN1N). The Microcontroller, a Raspberry Pi 3b (RPI), then communicates with the NAU7802 via the I2C bus on the RPI and calculates the conductivity and concentration based on the voltage reading from the NAU7802. Finally, the RPI updates the LEDs and LCD appropriately. The LEDs are connected via GPIO pins and the LCD is connected to the I2C bus.

Within Spec (30+ cm long, Less than 7 cm wide, 5+ cm submersible)







How it was constructed

The probe was made by drilling 2 holes in 2 chunks of wood (4 total holes) and putting 2 steel rods through the holes so they are kept at a constant distance. The steel rods were then glued in place using a hot glue gun and the wood was caulked in critical places to keep water/solution from seeping in. Lastly the alligator clips were attached to the ends of the steel rod to connect the probe with the sensor circuitry.

The rest of the circuit components are placed on two breadboards (one for the sensor circuit plus one for the ADC and display circuitry). The RPI was programmed using python. The LCD was bought with the I2C protocol implemented.

Data Collection and Analysis

Data Tables

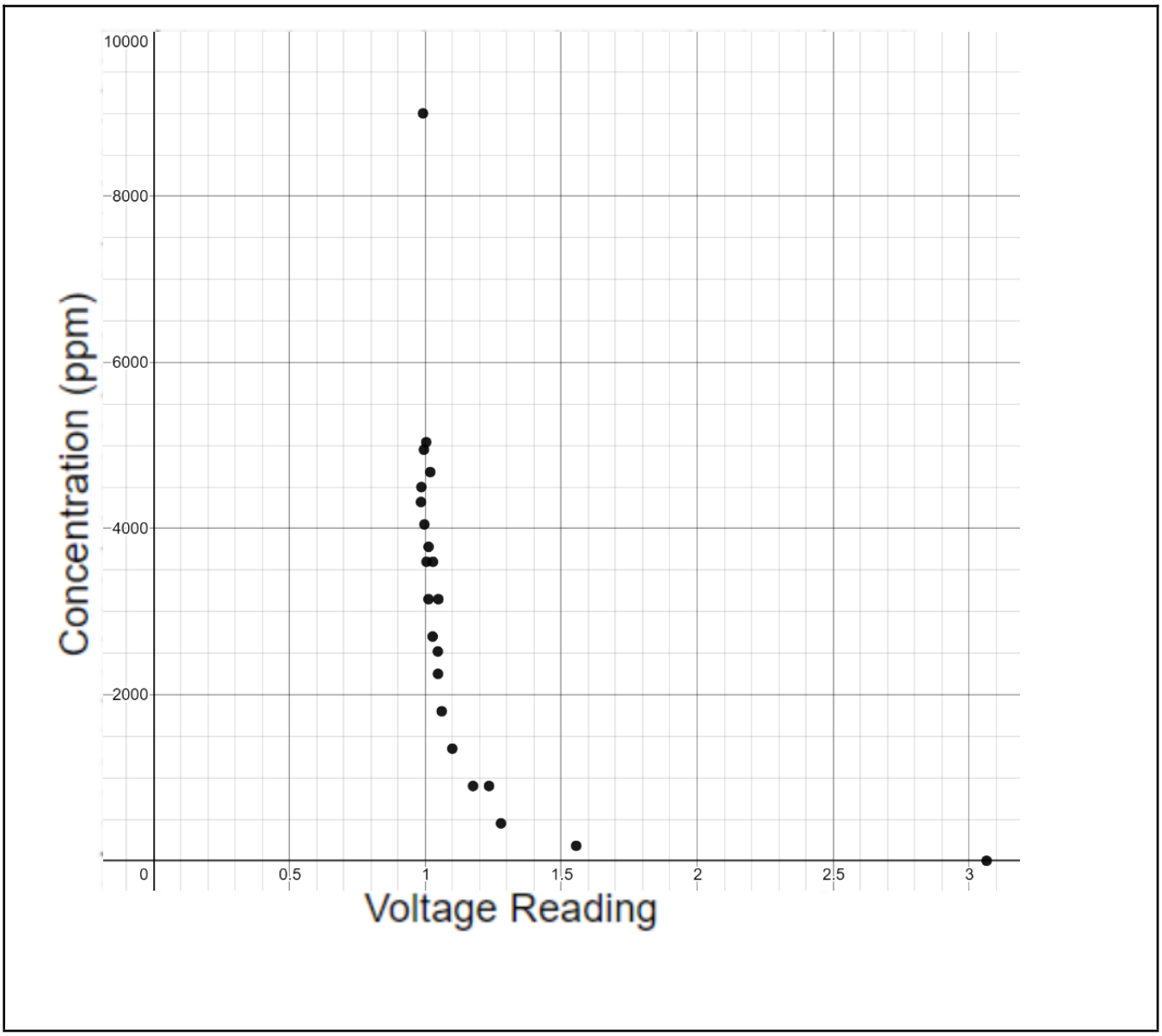
Voltage Reading	Concentration (ppm)
3.06487	0 (distilled water)
1.55496	180
1.27825	450
1.17553	900
1.09922	1350
1.0602	1800
1.04669	2250
1.04613	2520
1.02689	2700
1.01173	3150
1.00434	3600
1.01207	3780
0.99651	4050
1.028	3600
1.04793	3150
0.98412	4320
0.98531	4500
0.99483	4950
1.00316	5040
1.01816	4680
0.99145	9000

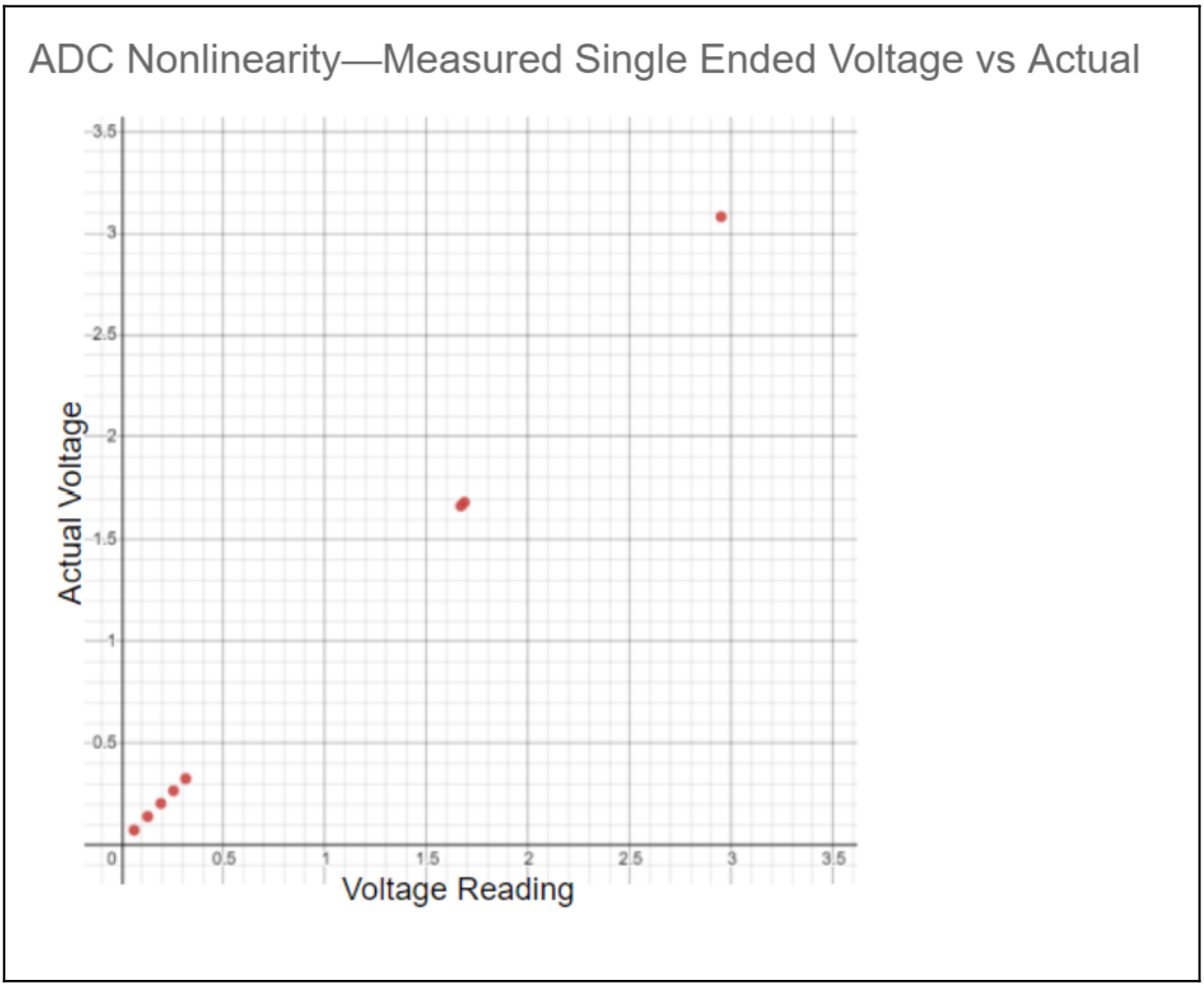
Voltage Reading (non-linearized)	Actual Voltage*
0.61250	0.0725
0.12788	0.1399
0.19300	0.2041
0.25500	0.2661
0.31500	0.3235
1.66900	1.666
1.68650	1.684
2.95000	3.081

*Determined through comparison with multimeter and known resistors

Scatterplots

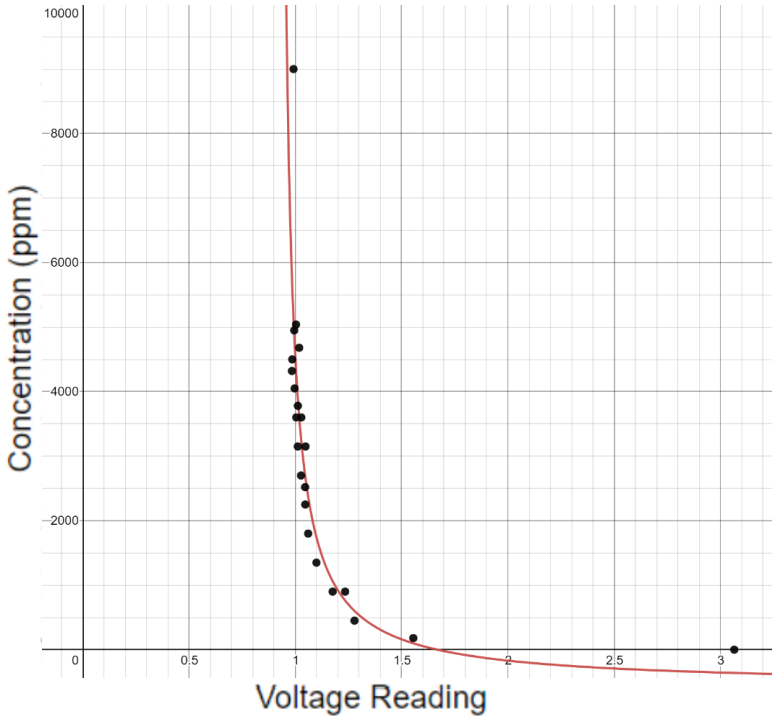
Voltage Reading (V) vs Concentration (ppm)


10000
8000
6000
4000
2000
0
0.5
1
1.5
2
2.5
3
Concentration (ppm)
Voltage Reading

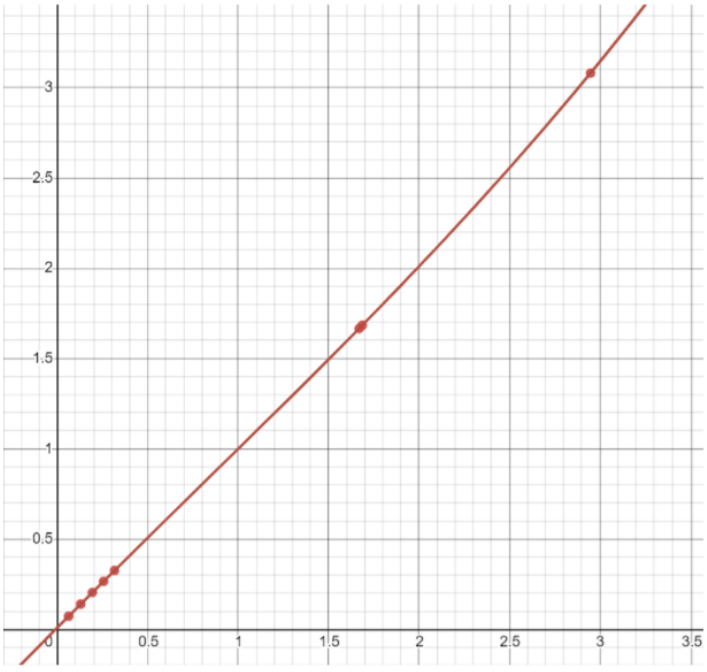

ADC Nonlinearity—Measured Single Ended Voltage vs Actual
Actual Voltage
Voltage Reading
0
0.5
1
1.5
2
2.5
3
3.5

Function Graphs (Regression)

Voltage Reading (V) vs Concentration (ppm)



ADC Nonlinearity—Measured Single Ended Voltage vs Actual



Regression Equations

Voltage Reading (V) vs Concentration (ppm)

$$c_1 \sim \frac{h}{d + v_1} + p$$

STATISTICS

$R^2 = 0.7441$

RESIDUALS

e_1 plot

PARAMETERS

$h = 416.125$ $d = -0.918042$

$p = -554.131$

Where c_1 is the known concentration and v_1 is the measured voltage
Then, for the actual predictions, the predicted concentration c is limited to nonnegative values and becomes a function of measured voltage v as follows:

$$c = \max\left(\frac{h}{v + d} + p, 0\right)$$

ADC Nonlinearity—Measured Single Ended Voltage vs Actual

$$y_1 \sim ax_1^3 + bx_1^2 + cx_1 + d$$

STATISTICS

$R^2 = 1$

RESIDUALS

e_1 plot

PARAMETERS

$a = 0.0166036$ $b = -0.0384329$

$c = 1.00958$ $d = 0.010942$

Code Printout

Three python files shown below. Detector.py is the main file where program execution starts and the magic happens. NAU7802_I2C.py and LCD_I2C.py contain classes for communicating with the NAU7802 and LCD respectively. The following colors have been used to highlight relevant code in Detector.py:

ADC VALUE TO VOLTAGE READING CODE HIGHLIGHTED IN ORANGE
VOLTAGE TO PPM CALCULATION HIGHLIGHTED IN GREEN
LED UPDATE CODE HIGHLIGHTED IN CYAN

Detector.py

```
import RPi.GPIO as GPIO
import LCD_I2C
import NAU7802_I2C
import smbus2
import time

# Create the bus
bus = smbus2.SMBus(1)

# initialize the NAU7802 ADC
adc = NAU7802_I2C.NAU7802()
if adc.begin(bus):
    print("Connected ADC!\n")
    print("Calibrated ADC!\n")
else:
    print("Can't find the adc, exiting ...\n")
    exit()

# initialize LCD display + verify it works
lcd = LCD_I2C.lcd(bus)
lcd.lcd_display_string("Calibrating ADC", 1)

# Constants
R1 = 214.3
R2 = 215.8
R3 = 9770
VIN = 3.29
VREF = VIN / 2
VDEVIATION = (0.0166036203409, -0.0384329033931, 1.00958174262,
0.010941954783)
# unknown voltage divider seems slightly off but it varies with voltage?
EPSILON = 1e-19

ADC_MAX = 2**23
SAMPLES = 8 # how many adc readings are averaged
```

```
H = 416.125
P = -554.131
D = -0.918042
CONVERSION = lambda volt: min(max(0, H / (volt + D) + P),5040)

def main():
    # LEDs
    GPIO.setmode(GPIO.BCM)
    RedLED = LED(22, 0, 1600)
    BlueLED = LED(27, 1600, 3300)
    GreenLED = LED(17, 3300, float('Inf'))
    LEDs = [RedLED, BlueLED, GreenLED]

    # cool lightshow to verify LEDs work
    letThereBeLight(LEDs)

    minVout = float('Inf')
    try:
        print('Reading detector values')
        while True:
            # Read the ADC for SAMPLES and return the average reading
            value = adc.getAverage(SAMPLES)

            # convert ADC value to the desired values (voltage, resistance,
conductivity)
            vout, resistance, conductance = interpretAdc(value)
            if vout < minVout: minVout = vout
            else: minVout = vout
            conductivity = CONVERSION(vout)

            # print values
            printValues(value, vout, resistance, conductivity)

            # show values on LCD
            lcd.lcd_clear()
            lcd.lcd_display_string("Voltage (adc value):", 1)
            lcd.lcd_display_string(str(round(vout,5)) + " (" + str(value) +
")", 2)
            lcd.lcd_display_string("Concentration (ppm):", 3)
            lcd.lcd_display_string(str(round(conductivity,3)), 4)
```

```
            # activate LEDs
            for l in LEDs:
                l.update(conductivity)

            # Pause for a second
            time.sleep(1)
    finally:
        print("Clean Exit \n")
        GPIO.cleanup() # ensure clean exit

def printValues(value, vout, resistance, conductivity):

    print('ADC: ' + str(value) + #' ' + format(int(value), 'b') +
            ' | Vol: ' + str(round(vout,5)) +
            ' | Res: ' + str(round(resistance)) +
            ' | Con: ' + str(round(conductivity,4))
    )

def interpretAdc(value):
    return interpretWheatstone(value)

def interpretWheatstone(value):
    vout = VREF * value / ADC_MAX
    v12 = VIN * R2 / (R1 + R2) # voltage out at known voltage divider
    vout += VDEVIATION[0] * (vout + v12) ** 3 + VDEVIATION[1] * (vout + v12)
** 2 + VDEVIATION[2] * (vout + v12) + VDEVIATION[3] - vout - v12
    # experimentally determined adjustment factor lol

    print('known voltage divider: ' + str(v12) +
          ' vout: ' + str(vout)
    )
    v3x = vout + v12 # voltage out at unknown voltage divider
    resistance = R3 * (R2 * VIN + (R1 + R2) * vout) / (R1 * VIN - (R1 + R2) *
vout + EPSILON)
    conductance = (R1 * VIN - (R1 + R2) * vout) / (R3 * (R2 * VIN + (R1 + R2)
* vout) + EPSILON)
    return v3x, resistance, conductance

def letThereBeLight(LEDs):
    for l in LEDs:
```

```
        l.on()
        time.sleep(0.3)
        l.off()
    for l in LEDs:
        l.on()
    time.sleep(0.5)

class LED():
    def __init__(self, pin, lower, upper):
        self.pin = pin
        self.lower = lower
        self.upper = upper
        GPIO.setup(pin, GPIO.OUT, initial=GPIO.LOW)

    def on(self):
        GPIO.output(self.pin, GPIO.HIGH)

    def off(self):
        GPIO.output(self.pin, GPIO.LOW)

    def update(self, value):
        if value >= self.lower and value < self.upper:
            self.on()
        else:
            self.off()

# runs main() when program is run from terminal
if __name__ == '__main__':
    main()
```

NAU7802_I2C.py

```
'''
RPI code to read the NAU7802 ADC values.
The NAU7802 is a 24-Bit Dual-Channel ADC
that uses the I2C protocol. Datasheet:
https://www.nuvoton.com/resource-files/NAU7802%20Data%20Sheet%20V1.7.pdf

Based on similar (arduino) code for a scale:
https://github.com/sparkfun/SparkFun_Qwiic_Scale_NAU7802_Arduino_Library

'''
```

```
import smbus2
import time

#########################################
# Constants
#########################################
""" Address """
DEVICE_ADDRESS = 0x2A

""" Register Map """
NAU7802_PU_CTRL = 0x00
NAU7802_CTRL1 = 0x01
NAU7802_CTRL2 = 0x02
NAU7802_OCAL1_B2 = 0x03 # CH1 OFFSET Calibration[23:16]
NAU7802_OCAL1_B1 = 0x04 # CH1 OFFSET Calibration[15:8]
NAU7802_OCAL1_B0 = 0x05 # CH1 OFFSET Calibration[7:0]
NAU7802_GCAL1_B3 = 0x06 # CH1 GAIN Calibration[31:24]
NAU7802_GCAL1_B2 = 0x07 # CH1 GAIN Calibration[23:16]
NAU7802_GCAL1_B1 = 0x08 # CH1 GAIN Calibration[15:8]
NAU7802_GCAL1_B0 = 0x09 # CH1 GAIN Calibration[7:0]
NAU7802_OCAL2_B2 = 0x0A # CH2 OFFSET Calibration[23:16]
NAU7802_OCAL2_B1 = 0x0B # CH2 OFFSET Calibration[15:8]
NAU7802_OCAL2_B0 = 0x0C # CH2 OFFSET Calibration[7:0]
NAU7802_GCAL2_B3 = 0x0D # CH2 GAIN Calibration[31:24]
NAU7802_GCAL2_B2 = 0x0E # CH2 GAIN Calibration[23:16]
NAU7802_GCAL2_B1 = 0x0F # CH2 GAIN Calibration[15:8]
NAU7802_GCAL2_B0 = 0x10 # CH2 GAIN Calibration[7:0
NAU7802_I2C_CONTROL = 0x11
NAU7802_ADCO_B2 = 0x12 # ADC_OUT[23:16]
NAU7802_ADCO_B1 = 0x13 # ADC_OUT[15:8]
NAU7802_ADCO_B0 = 0x14 # ADC_OUT[7:0]

# No idea why this is like this lol:
NAU7802_ADC = 0x15  # Shared ADC and OTP 32:24
NAU7802_OTP_B1 = 0x16 # OTP 23:16 or 7:0?
NAU7802_OTP_B0 = 0x17 # OTP 15:8
NAU7802_PGA = 0x1B # ?????
NAU7802_PGA_PWR = 0x1C # ?????

NAU7802_DEVICE_REV = 0x1F # Device Revision Code
```

```
""" Bits within the PU_CTRL register """
NAU7802_PU_CTRL_RR = 0
NAU7802_PU_CTRL_PUD = 1
NAU7802_PU_CTRL_PUA = 2
NAU7802_PU_CTRL_PUR = 3
NAU7802_PU_CTRL_CS = 4
NAU7802_PU_CTRL_CR = 5
NAU7802_PU_CTRL_OSCS = 6
NAU7802_PU_CTRL_AVDDS = 7

""" Bits within the CTRL1 register """
NAU7802_CTRL1_GAIN = 2
NAU7802_CTRL1_VLDO = 5
NAU7802_CTRL1_DRDY_SEL = 6
NAU7802_CTRL1_CRP = 7

""" Bits within the CTRL2 register """
NAU7802_CTRL2_CALMOD = 0
NAU7802_CTRL2_CALS = 2
NAU7802_CTRL2_CAL_ERROR = 3
NAU7802_CTRL2_CRS = 4
NAU7802_CTRL2_CHS = 7

""" Bits within the PGA register """
NAU7802_PGA_CHP_DIS = 0
NAU7802_PGA_INV = 3
NAU7802_PGA_BYPASS_EN = 4
NAU7802_PGA_OUT_EN = 5
NAU7802_PGA_LDOMODE = 6
NAU7802_PGA_RD_OTP_SEL = 7

""" Bits within the PGA PWR register """
NAU7802_PGA_PWR_PGA_CURR = 0
NAU7802_PGA_PWR_ADC_CURR = 2
NAU7802_PGA_PWR_MSTR_BIAS_CURR = 4
NAU7802_PGA_PWR_PGA_CAP_EN = 7

""" Allowed Low dropout regulator voltages """
NAU7802_LDO_2V4 = 0b111
NAU7802_LDO_2V7 = 0b110
```

```
NAU7802_LDO_3V0 = 0b101
NAU7802_LDO_3V3 = 0b100
NAU7802_LDO_3V6 = 0b011
NAU7802_LDO_3V9 = 0b010
NAU7802_LDO_4V2 = 0b001
NAU7802_LDO_4V5 = 0b000

""" Allowed gains """
NAU7802_GAIN_128 = 0b111
NAU7802_GAIN_64 = 0b110
NAU7802_GAIN_32 = 0b101
NAU7802_GAIN_16 = 0b100
NAU7802_GAIN_8 = 0b011
NAU7802_GAIN_4 = 0b010
NAU7802_GAIN_2 = 0b001
NAU7802_GAIN_1 = 0b000

""" Allowed samples per second """
NAU7802_SPS_320 = 0b111
NAU7802_SPS_80 = 0b011
NAU7802_SPS_40 = 0b010
NAU7802_SPS_20 = 0b001
NAU7802_SPS_10 = 0b000

""" Select between channel values """
NAU7802_CHANNEL_1 = 0
NAU7802_CHANNEL_2 = 1

""" Calibration state """
NAU7802_CAL_SUCCESS = 0
NAU7802_CAL_IN_PROGRESS = 1
NAU7802_CAL_FAILURE = 2


##########################################
# Classes
##########################################
class NAU7802:
    """ Class to communicate with the NAU7802 """
    _i2cPort: smbus2.SMBus = None
```

```
    # Sets up the NAU7802 for basic function
    # If initialize is true (or not specified), default init and calibration
is performed
    # If initialize is false, then it's up to the caller to initalize and
calibrate
    # Returns true upon completion
    def begin(self, wire_port: smbus2.SMBus = smbus2.SMBus(1), initialize:
bool = True) -> bool:
        """ Check communication and initialize sensor """
        # Get user's options
        self._i2cPort = wire_port

        # Check if the device ACK's over I2C
        if not self.isConnected():
            # There are rare times when the sensor is occupied and doesn't
ACK. A 2nd try resolves this.
            if not self.isConnected():
                return False

        result = True  # Accumulate a result as we do the setup

        if initialize:
            result &= self.reset()  # Reset all registers
            result &= self.powerUp()  # Power on analog and digital sections
            result &= self.setGain(NAU7802_GAIN_1)  # Set gain to 1
            result &= self.setSampleRate(NAU7802_SPS_10)  # Set samples per
second to 10
            result &= self.setRegister(NAU7802_ADC, 0x30)  # Turn off CLK_CHP.
From 9.1 power on sequencing.
            result &= self.calibrateAFE()  # Re - cal analog frontend when we
change gain, sample rate, or channel

        return result

    def isConnected(self) -> bool:
        """ Returns true if device ACK's at the I2C address """
        try:
            self._i2cPort.read_byte(DEVICE_ADDRESS)
            return True  # All good
        except OSError:
            return False  # Sensor did not ACK
```

```
    def available(self) -> bool:
        """ Returns true if Cycle Ready bit is set (conversion is complete)
"""
        return self.getBit(NAU7802_PU_CTRL_CR, NAU7802_PU_CTRL)

    def getReading(self) -> int:
        """ Returns 24 bit reading. Assumes CR Cycle Ready bit
        (ADC conversion complete) has been checked by .available() """
        try:
            value_list = self._i2cPort.read_i2c_block_data(DEVICE_ADDRESS,
NAU7802_ADCO_B2, 3)
        except OSError:
            return False  # Sensor did not ACK
        value = int.from_bytes(value_list, byteorder='big', signed=True)

        return value

    def getAverage(self, average_amount: int) -> int:
        """ Return the average of a given number of readings """
        total = 0
        samples_acquired = 0

        start_time = time.time()

        while samples_acquired < average_amount:
            if self.available():
                total += self.getReading()
                samples_acquired += 1

            if time.time() - start_time > 1.0:
                return 0  # Timeout - Bail with error

            time.sleep(0.001)

        total /= average_amount

        return total

    def setGain(self, gain_value: int) -> bool:
        """ Set the gain.x1, 2, 4, 8, 16, 32, 64, 128 are available """
```

```
        if gain_value > 0b111:
            gain_value = 0b111  # Error check

        value = self.getRegister(NAU7802_CTRL1)
        value &= 0b11111000  # Clear gain bits
        value |= gain_value  # Mask in new bits

        return self.setRegister(NAU7802_CTRL1, value)

    def setLDO(self, ldo_value: int) -> bool:
        """ Set the on-board Low - Drop - Out voltage regulator to a given
value.
        2.4, 2.7, 3.0, 3.3, 3.6, 3.9, 4.2, 4.5 V are available """
        if ldo_value > 0b111:
            ldo_value = 0b111  # Error check

        # Set the value of the LDO
        value = self.getRegister(NAU7802_CTRL1)
        value &= 0b11000111  # Clear LDO bits
        value |= ldo_value << 3  # Mask in new LDO bits
        self.setRegister(NAU7802_CTRL1, value)

        return self.setBit(NAU7802_PU_CTRL_AVDDS, NAU7802_PU_CTRL)  # Enable
the internal LDO

    def setSampleRate(self, rate: int) -> bool:
        """ Set the readings per second. 10, 20, 40, 80, and 320 samples per
second is available """
        if rate > 0b111:
            rate = 0b111  # Error check

        value = self.getRegister(NAU7802_CTRL2)
        value &= 0b10001111  # Clear CRS bits
        value |= rate << 4  # Mask in new CRS bits

        return self.setRegister(NAU7802_CTRL2, value)

    def setChannel(self, channel_number: int) -> bool:
        """ Select between 1 and 2 """
        if channel_number == NAU7802_CHANNEL_1:
```

```
            return self.clearBit(NAU7802_CTRL2_CHS, NAU7802_CTRL2)  # Channel
1 (default)
        else:
            return self.setBit(NAU7802_CTRL2_CHS, NAU7802_CTRL2)  # Channel 2

    def calibrateAFE(self) -> bool:
        """ Synchronous calibration of the analog front end of the NAU7802.
        Returns true if CAL_ERR bit is 0 (no error) """
        self.beginCalibrateAFE()
        return self.waitForCalibrateAFE(1000)

    def beginCalibrateAFE(self) -> None:
        """ Begin asynchronous calibration of the analog front end of the
NAU7802.
        Poll for completion with calAFEStatus() or wait with
waitForCalibrateAFE(). """
        self.setBit(NAU7802_CTRL2_CALS, NAU7802_CTRL2)

    def waitForCalibrateAFE(self, timeout_ms: int = 0) -> bool:
        """ Wait for asynchronous AFE calibration to complete with optional
timeout. """
        timeout_s = timeout_ms/1000
        begin = time.time()
        cal_ready = self.calAFEStatus()

        while cal_ready == NAU7802_CAL_IN_PROGRESS:
            if (timeout_ms > 0) & ((time.time() - begin) > timeout_s):
                break
            time.sleep(0.001)
            cal_ready = self.calAFEStatus()

        if cal_ready == NAU7802_CAL_SUCCESS:
            return True
        else:
            return False

    def calAFEStatus(self) -> int:
        """ Check calibration status. """
        if self.getBit(NAU7802_CTRL2_CALS, NAU7802_CTRL2):
            return NAU7802_CAL_IN_PROGRESS
```

```
        if self.getBit(NAU7802_CTRL2_CAL_ERROR, NAU7802_CTRL2):
            return NAU7802_CAL_FAILURE

        # Calibration passed
        return NAU7802_CAL_SUCCESS

    def reset(self) -> bool:
        """ Resets all registers to Power Of Defaults """
        self.setBit(NAU7802_PU_CTRL_RR, NAU7802_PU_CTRL)  # Set RR
        time.sleep(0.001)
        return self.clearBit(NAU7802_PU_CTRL_RR, NAU7802_PU_CTRL)  # Clear RR
to leave reset state

    def powerUp(self) -> bool:
        """ Power up digital and analog sections, ~2 mA """
        self.setBit(NAU7802_PU_CTRL_PUD, NAU7802_PU_CTRL)
        self.setBit(NAU7802_PU_CTRL_PUA, NAU7802_PU_CTRL)

        # Wait for Power Up bit to be set - takes approximately 200us
        counter = 0
        while not self.getBit(NAU7802_PU_CTRL_PUR, NAU7802_PU_CTRL):
            time.sleep(0.001)
            if counter > 100:
                return False  # Error
            counter += 1

        return True

    def powerDown(self) -> bool:
        """ Set low power 200 nA mode """
        self.clearBit(NAU7802_PU_CTRL_PUD, NAU7802_PU_CTRL)
        return self.clearBit(NAU7802_PU_CTRL_PUA, NAU7802_PU_CTRL)

    def setIntPolarityHigh(self) -> bool:
        """ Set Int pin to be high when data is ready(default) """
        return self.clearBit(NAU7802_CTRL1_CRP, NAU7802_CTRL1)  # 0 = CRDY pin
is high active (ready when 1)

    def setIntPolarityLow(self) -> bool:
        """ Set Int pin to be low when data is ready """
```

```
        return self.setBit(NAU7802_CTRL1_CRP, NAU7802_CTRL1)  # 1 = CRDY pin
is low active (ready when 0)

    def getRevisionCode(self) -> int:
        """ Get the revision code of this IC.Always 0x0F. """
        revisionCode = self.getRegister(NAU7802_DEVICE_REV)
        return revisionCode & 0x0F

    def setBit(self, bit_number: int, register_address: int) -> bool:
        """ Mask & set a given bit within a register """
        value = self.getRegister(register_address)
        value |= (1 << bit_number)  # Set this bit
        return self.setRegister(register_address, value)

    def clearBit(self, bit_number: int, register_address: int) -> bool:
        """ Mask & clear a given bit within a register """
        value = self.getRegister(register_address)
        value &= ~(1 << bit_number)  # Set this bit
        return self.setRegister(register_address, value)

    def getBit(self, bit_number: int, register_address: int) -> bool:
        """ Return a given bit within a register """
        value = self.getRegister(register_address)
        value &= (1 << bit_number)  # Clear all but this bit
        return bool(value)

    def getRegister(self, register_address: int) -> int:
        """ Get contents of a register """
        try:
            return self._i2cPort.read_byte_data(DEVICE_ADDRESS,
register_address)

        except OSError:
            return -1  # Sensor did not ACK

    def setRegister(self, register_address: int, value: int) -> bool:
        """ Send a given value to be written to given address.Return true if
successful """
        try:
            self._i2cPort.write_byte_data(DEVICE_ADDRESS, register_address,
value)
```

```
            return True

        except OSError:
            return False
```

LCD_I2C.py

```
# Original code found at:
# https://gist.github.com/DenisFromHR/cc863375a6e19dce359d

"""
Compiled, mashed and generally mutilated 2014-2015 by Denis Pleic
Made available under GNU GENERAL PUBLIC LICENSE

# Modified Python I2C library for Raspberry Pi
# as found on http://www.recantha.co.uk/blog/?p=4849
# Joined existing 'i2c_lib.py' and 'lcddriver.py' into a single library
# added bits and pieces from various sources
# By DenisFromHR (Denis Pleic)
# 2015-02-10, ver 0.1

"""

'''
Further Mutilated by Alex Metzger

'''

# i2c bus (0 -- original Pi, 1 -- Rev 2 Pi)
I2CBUS = 1

# LCD Address
ADDRESS = 0x27

import smbus2 as smbus
from time import sleep

class i2c_device:
   def __init__(self, addr, bus=smbus.SMBus(I2CBUS)):
      self.addr = addr
      self.bus = bus
    #   self.bus = smbus.SMBus(port)
```

```
# Write a single command
   def write_cmd(self, cmd):
      self.bus.write_byte(self.addr, cmd)
      sleep(0.0001)

# Write a command and argument
   def write_cmd_arg(self, cmd, data):
      self.bus.write_byte_data(self.addr, cmd, data)
      sleep(0.0001)

# Write a block of data
   def write_block_data(self, cmd, data):
      self.bus.write_block_data(self.addr, cmd, data)
      sleep(0.0001)

# Read a single byte
   def read(self):
      return self.bus.read_byte(self.addr)

# Read
   def read_data(self, cmd):
      return self.bus.read_byte_data(self.addr, cmd)

# Read a block of data
   def read_block_data(self, cmd):
      return self.bus.read_block_data(self.addr, cmd)


# commands
LCD_CLEARDISPLAY = 0x01
LCD_RETURNHOME = 0x02
LCD_ENTRYMODESET = 0x04
LCD_DISPLAYCONTROL = 0x08
LCD_CURSORSHIFT = 0x10
LCD_FUNCTIONSET = 0x20
LCD_SETCGRAMADDR = 0x40
LCD_SETDDRAMADDR = 0x80

# flags for display entry mode
LCD_ENTRYRIGHT = 0x00
```

```
LCD_ENTRYLEFT = 0x02
LCD_ENTRYSHIFTINCREMENT = 0x01
LCD_ENTRYSHIFTDECREMENT = 0x00

# flags for display on/off control
LCD_DISPLAYON = 0x04
LCD_DISPLAYOFF = 0x00
LCD_CURSORON = 0x02
LCD_CURSOROFF = 0x00
LCD_BLINKON = 0x01
LCD_BLINKOFF = 0x00

# flags for display/cursor shift
LCD_DISPLAYMOVE = 0x08
LCD_CURSORMOVE = 0x00
LCD_MOVERIGHT = 0x04
LCD_MOVELEFT = 0x00

# flags for function set
LCD_8BITMODE = 0x10
LCD_4BITMODE = 0x00
LCD_2LINE = 0x08
LCD_1LINE = 0x00
LCD_5x10DOTS = 0x04
LCD_5x8DOTS = 0x00

# flags for backlight control
LCD_BACKLIGHT = 0x08
LCD_NOBACKLIGHT = 0x00

En = 0b00000100 # Enable bit
Rw = 0b00000010 # Read/Write bit
Rs = 0b00000001 # Register select bit

class lcd:
   #initializes objects and lcd
   def __init__(self, i2cbus = smbus.SMBus(I2CBUS)):
      self.lcd_device = i2c_device(ADDRESS, i2cbus)

      self.lcd_write(0x03)
      self.lcd_write(0x03)
```

```
        self.lcd_write(0x03)
        self.lcd_write(0x02)

        self.lcd_write(LCD_FUNCTIONSET | LCD_2LINE | LCD_5x8DOTS | LCD_4BITMODE)
        self.lcd_write(LCD_DISPLAYCONTROL | LCD_DISPLAYON)
        self.lcd_write(LCD_CLEARDISPLAY)
        self.lcd_write(LCD_ENTRYMODESET | LCD_ENTRYLEFT)
        sleep(0.2)


    # clocks EN to latch command
    def lcd_strobe(self, data):
        self.lcd_device.write_cmd(data | En | LCD_BACKLIGHT)
        sleep(.0005)
        self.lcd_device.write_cmd(((data & ~En) | LCD_BACKLIGHT))
        sleep(.0001)

    def lcd_write_four_bits(self, data):
        self.lcd_device.write_cmd(data | LCD_BACKLIGHT)
        self.lcd_strobe(data)

    # write a command to lcd
    def lcd_write(self, cmd, mode=0):
        self.lcd_write_four_bits(mode | (cmd & 0xF0))
        self.lcd_write_four_bits(mode | ((cmd << 4) & 0xF0))

    # write a character to lcd (or character rom) 0x09: backlight | RS=DR<
    # works!
    def lcd_write_char(self, charvalue, mode=1):
        self.lcd_write_four_bits(mode | (charvalue & 0xF0))
        self.lcd_write_four_bits(mode | ((charvalue << 4) & 0xF0))

    # put string function with optional char positioning
    def lcd_display_string(self, string, line=1, pos=0):
     if line == 1:
       pos_new = pos
     elif line == 2:
       pos_new = 0x40 + pos
     elif line == 3:
       pos_new = 0x14 + pos
     elif line == 4:
```

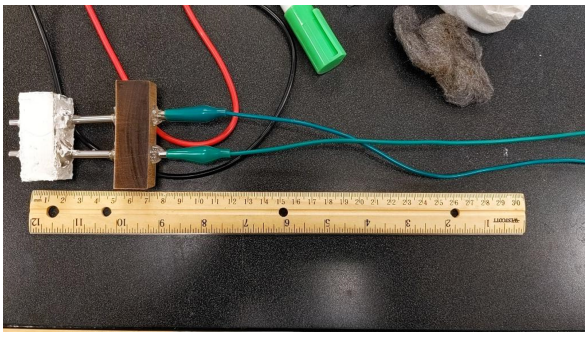
```
    pos_new = 0x54 + pos

   self.lcd_write(0x80 + pos_new)

   for char in string:
      self.lcd_write(ord(char), Rs)

# clear lcd and set to home
def lcd_clear(self):
   self.lcd_write(LCD_CLEARDISPLAY)
   self.lcd_write(LCD_RETURNHOME)

# define backlight on/off (lcd.backlight(1); off= lcd.backlight(0)
def backlight(self, state): # for state, 1 = on, 0 = off
   if state == 1:
      self.lcd_device.write_cmd(LCD_BACKLIGHT)
   elif state == 0:
      self.lcd_device.write_cmd(LCD_NOBACKLIGHT)

# add custom characters (0 - 7)
def lcd_load_custom_chars(self, fontdata):
   self.lcd_write(0x40)
   for char in fontdata:
      for line in char:
         self.lcd_write_char(line)
```

CONSTRUCTION PARAMETERS		
Respond to the following statements for your Device by typing either "TRUE" or "FALSE" in the space to the right of each statement.		
3.a.	Our Device is built using a microcontroller or microcontroller board, a display, LED lights, and a participant-built sensor/probe. The sensor produces a voltage which varies according to the concentration of the water. WiFi/Internet connection is not used at any time during competition.	TRUE
3.b.	Our sensor is student-constructed from fundamental electronic components such as resistors, capacitors, wire, and DIP package integrated circuits. All circuits are assembled on a breadboard. No preassembled integrated circuit PCB boards are used. The sensor and wires/cables, together, are ≥ 30.0 cm in length, and narrow enough to fit through an opening of 7.0 cm and the end is immersible up to 5.0 cm in water.	TRUE
3.d.	Our Device has a digital display that clearly shows voltage, and the salt concentration in ppm to the nearest unit value. If a laptop is used for display purposes, it is not used for the Written Test portion.	TRUE
3.e.	Our Device is able to indicate the specific concentration zone using three separate LEDs – one red, one green, and one blue. RGB LEDs, if used, is wired for only one color.	TRUE

3.f.	We did not use electrical outlets at any time during the competition. If our Device is not powered by a connected laptop or calculator, our Device is powered by commercially available batteries.	TRUE
3.g.	Our Device is clearly labeled with our team's name and team number.	TRUE
5.Part I.a.	Once my partner and I entered the event area, we did not leave or receive outside assistance, materials, or communication.	TRUE

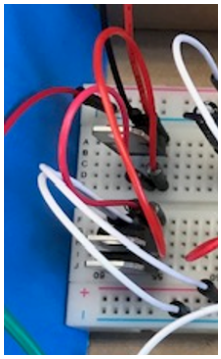
Device Dimensions

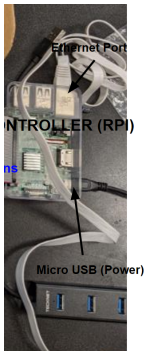
2.e.	**Device length:** In the space below, insert a photograph of a ruler held up to your Device, showing the measurement of the length of the sensor and wires/cables together.



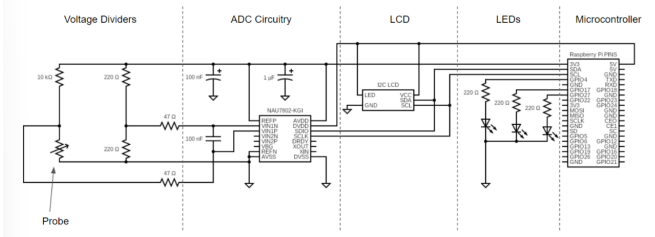
2.e.	Is the length, in centimeters, of your Device's sensor and wires/cables together ≥ 30.0 cm?	YES

2.e.	**Device power:** In the space below, insert a photograph or schematic showing how your Device is powered. Specify input voltages and the output voltage of your probe.





Schematic



The primary power supply of the device is two commercial 3.7 V batteries in series. They are fed through three 5 V voltage regulators before reaching the device so the power supplied never exceeds 5 V. This 5 V potential difference is applied to the 5 V rail of the Raspberry Pi (microcontroller) and powers the circuitry.

The secondary power supply is from a laptop through a Micro USB to facilitate screen sharing (i.e. using the laptop as a secondary display). As per USB standards, this power never exceeds 5 V so total power to the device never exceeds 5 + 5 = 10 V.

The probe is connected to the 3.3 V rail on the Raspberry Pi which is voltage regulated so the input and output voltage of the probe never exceeds 3.3 V.		
2.e.	What is the total input voltage, in volts, supplied to your Device by your laptop or batteries?	10 V
2.e.	What is the expected output voltage, in volts, of your Device's probe?	3.3 V

Device Testing

You will complete this section when you test your device during the tournament.	
Station 1: What was the voltage, to the nearest whole number, displayed by your Device?	1
Station 1: What was the concentration (ppm), to the nearest whole number, displayed by your Device?	3229
Station 1: What LED colors were displayed by your Device?	Blue
Station 2: What was the voltage, to the nearest whole number, displayed by your Device?	1
Station 2: What was the concentration (ppm), to the nearest whole number, displayed by your Device?	2689
Station 2: What LED colors were displayed by your Device?	Blue, Red
Station 3: What was the voltage, to the nearest whole number, displayed by your Device?	1
Station 3: What was the concentration (ppm), to the nearest whole number, displayed by your Device?	1103
Station 3: What LED colors were displayed by your Device?	Blue, Red
Station 4 (State Only): What was the voltage, to the nearest whole number, displayed by your Device?	1
Station 4 (State Only):: What was the concentration (ppm), to the nearest whole number, displayed by your Device?	1834
Station 4 (State Only): What LED colors were displayed by your Device?	Blue, Green, Red

You have reached the end of this document. Please do not add anything beyond this point.