Injection Lock Stabilizer with Arduino

Caleb Heuvel-Horwitz

July 2021

This device is capable of stabilizing a standard laser injection lock indefinitely. It is easy to build and use, and it requires minimal power supply and computing power to run.

1 Instructions For Use

An injection lock needs to first be established manually before the program can run. Once a lock is established and the stabilizer is engaged, it behaves according to the following procedure:

1. The Arduino samples readings from the cavity scanner, stores them in an array, and determines the maximum of the array, which is the height of the peaks from the injection lock. This is done a fixed number of times, after which an average is taken. A threshold peak height is set equal to 0.95 $\times$ the average peak height.
2. Readings from the cavity scanner are continuously sampled at a fixed rate, which can be adjusted. These readings are stored in a separate array and the average peak height (maximum of the array) is determined and compared with the threshold value. This process continues indefinitely as long as the sample peak height is always found to be greater than the threshold.
3. If the sample peak height is found to be less than the threshold, the program enters recovery mode:
    i. The current controller increases the current by 1 mA, which should place the current above the injection lock range for most injection locks. Depending on the size of your injection lock range, you may wish to adjust this parameter.
    ii. The current is decreased by 0.1 mA, then sample readings from the cavity scanner are stored in an array and the new peak height is determined. This step is repeated until the sample peak height is found to be greater than the threshold.
    iii. If the sample peak height is found to be above the threshold, the injection lock has been recovered. The program exits recovery mode and returns to 2.

Before running the program, read the section about Arduino code and make the appropriate changes based on your hardware.

2 Injection Lock Setup

Some of the light from your injection locked laser setup should be sent through an optical cavity. Aside from this, the actual setup for the injection lock is unimportant, as long as there is a current controller for the slave laser that allows for external voltage input. The setup used to test this stabilizer is included in figure 1.

The master laser diode was kept at around a current 80 mA and the slave laser diode was kept around 100 mA. The slave light was sent into a scanning optical cavity with a free spectral range of 1.5 GHz.

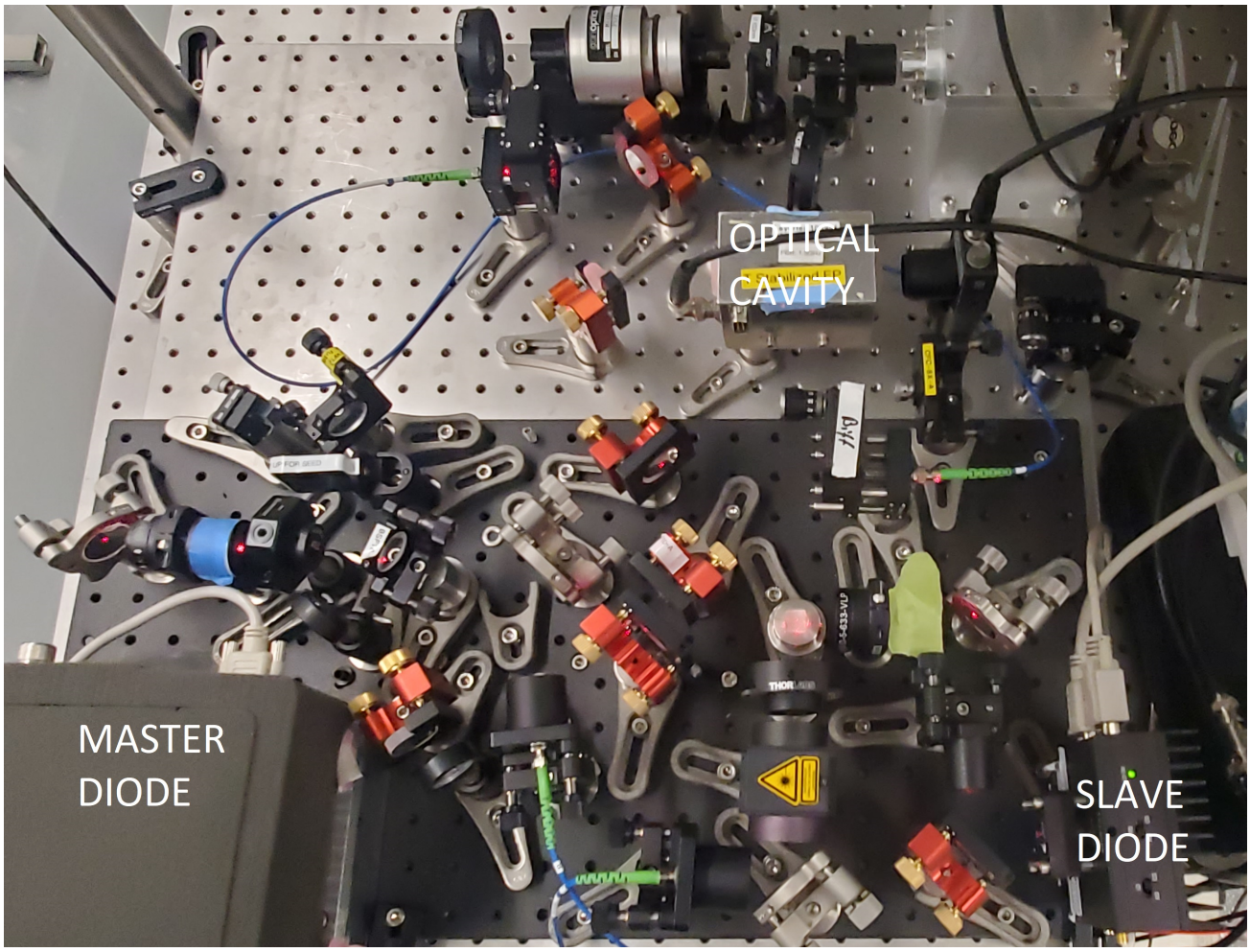

Figure 1: The injection lock setup used in testing.

3 Arduino Code

The program is thoroughly annotated for those who wish to make edits directly to the source code, but it can usually be run without any changes. However, it is important that you know the hardware specifications of your Arduino before running the program. With your hardware in mind, you may wish to adjust the following parameters (listed by location of line in the code):

- Line 3: 'avgnum'. This gives the number of peak heights that will be averaged over to find the lock threshold and to continuously test against it. The greater the number of sample peak heights, the greater the accuracy of the recovery program will be at detecting a lock recovery, but recovery will take longer. It is set to 30 by default.
- Line 4: 'avgnumtwo'. This gives the number of peak heights that will be averaged over to test against the threshold during recovery mode. typically this number should be greater than 1, but smaller than 'avgnum', so that less time is spent in recovery mode and more time is spent with the lock stable. It is set to 5 by default.
- Line 5: 'scantime'. This gives the amount of time, in milliseconds, that the program waits between collecting each sample peak height. It effectively controls the sampling rate of the program. Most Arduinos, including the MKRZERO which was used to test this program, have a sampling rate of 9600 on their analog pins (or about 10 kHz). You can adjust the sampling rate of the program to be up to 10 kHz, but generally the sampling rate of the Arduino should be greater than the sampling rate of the cavity scanner for your injection lock. The code tells the Arduino to sample readings from the cavity at a default of 10 kHz.
- Line 6: 'conv'. This gives the ratio of maximum digital values to maximum analog values that the DAC can handle. If you are using an Arduino that has a DAC with a resolution that can be set to 10 bits (sush as one from the DUE, ZERO, or MKR families), this line can be left unchanged. This device must requires an Arduino with a DAC to work. In theory and with some modifications a low-pass PWM could be used instead, but this program was not tested using a PWM, so proceed at your own risk. The default resolution of most DAC pins is 8 bits, but this program sets the analog write resolution to 10 explicitly. In principal it can be changed to an arbitrary value, and should correspond to your Arduino's hardware capabilities. Since the MKRZERO has a 10 bit DAC with a maximum output of 3.3 V, 'conv' is set to 1023/3.3. If for example your Arduino instead had an 8 bit DAC with a maximum output of 5 V, 'conv' would need to be changed to 255/5.
- Lines 7-8: 'a' and 'b'. The Arduino adjusts the current via the current controller by converting the desired current to a voltage before writing it to the DAC via the formula voltage $= a \times$ current $+ b$. By default, the parameters are set to $a = 0.33$ and $b = 1.65$. It maps $\pm$5mA to 0V$-$3.3V (this voltage is later mapped back to $\pm$5mA by the circuit before being sent to the current controller). This is so that the Arduino isn't being told to output a negative voltage, which it cannot do. These parameters can be changed to something arbitrary depending on the magnitude of the anticipated drift of the injection lock, but they should work as-is for most locks.
- Line 9: 'currentjump'. This tells the current controller to increase the current by a certain amount at the start of recovery mode, with the aim of getting the current above the injection lock range. It is set to 1 mA by default, which should apply to most injection locks, but change it as you see fit.
- Line 10: 'currentrange'. This gives the range of current that the Arduino can tell the current controller to adjust by. It is set to 5 mA by default, which should be enough in most cases. If you do change this parameter, you will also have to adjust the second resistor divider (see the circuit section).
- Line 11: 'res'. This gives the resolution of the DAC. Set it to what your Arduino can handle (most are fine with 10), but if you change it you will also have to adjust line 5, as explained above.
- Line 12: 'threshb'. This gives the proportion of the threshold that the average peak height needs to overcome to remain stable. By default it is set to 0.95, but you may wish to lower it as far as 0.8 depending on the natural stability of your lock.

To upload the code as noninvasively as possible, unscrew the back end of the box and the USB port of the board will be easily accessible.

4 Circuit

Details about constructing the circuit for this device are shown in figure 2.

The signal inverter (outlined in red) was used because during testing of the program it was found that the signal coming from the cavity scanner output was negative, and Arduino cannot interpret negative voltages. You should check that your signal is negative coming out of the cavity scanner, and if it's not, this part of the circuit can be ignored.

The resistors R_1 and R_2 in the first resistor divider (outlined in orange) should have values such that $R_2/(R_1 + R_2) = (V/2)/5$, where V is the maximum voltage that your DAC can output. For MKRZERO, this is $(3.3/2)/5 = 0.33 \longrightarrow R_1 = 2.0303R_2$. The purpose of this voltage divider is to output a voltage of $V/2$ which will serve as one of the inputs for the differential amplifier (outlined in green). Since it will likely be dificult to find resistors such that R_1 is exactly equal to $2.0303R_2$, you may find it easier to use a potentiometer for the orange section of the circuit. The voltage going into this resistor divider comes from the 5V pin on the Arduino. If your Arduino does not have a 5V pin, use a +5V external power supply instead.

The second resistor divider (outlined in blue) serves to shrink the outgoing signal to the DAC such that telling the Arduino to output the maximum voltage results in the current controller adjusting the current by +5mA (and −5mA for an output of 0V). Instead of finding explicit values for the resistors R_3 and R_4, it is easiest to use another potentiometer to make a pair of variable resistors for this section of the circuit. The potentiometer can then be adjusted manually to achieve the ±5mA range on the current controller. The current controller used in testing had a maximum current of 200mA for ±10V, so you will have to adjust the potentiometer depending on your current controller.

Observe that capacitors connect the positive and negative power supplies for the operational amplifiers in both of the buffer amplifiers (yellow and blue) to the ground. This is to reduce signal noise before being sent to the current controller, which was found to be significantly high in testing. The buffer amplifiers themselves exist to stabilize the voltage coming out of the voltage dividers, as it was found in testing that the voltage would shift when the entire circuit was turned on, possibly due to inductance in the wiring of the setup.

This circuit setup requires two power supplies: one of +5V (to power the Arduino) and one of −5V (to power the negative node of the op amps). Note that the positive node of the op amps is powered by the 5V pin on the Arduino. Most Arduino boards have a 5V pin, but if yours does not, you can also use the +5V power supply for this purpose. NOTE: always connect the +5V power supply before connecting the −5V supply, and disconnect them in reverse order, otherwise you risk damaging the op amps.

Figure 3 shows the testing setup, with approximate locations of circuit sections in corresponding colors.

5 Demonstration of stabiliser in action

To demonstrate the effectiveness of this program, we plot the average peak height, as well as the current being output by the program.

In figures 4 and 5, the average peak height and adjusted current are shown for 200 seconds of operation. Whenever a peak is found to be below the threshold, the recovery program adjusts the current until all peaks are measured above the threshold again.

In figures 6 and 7, this process can be seen for a much longer time of operation.

As can be seen from the figures, the test injection lock setup was already very stable on its own and required little recovery. Regardless, the robustness of the program is evident from these results. Although the lock was usually stable, the peak heights fluctuated much more later on as compared with the first 200 seconds of operation. If you don't want the program to be constantly correcting the current for some reason, you can make the threshold less sensitive (ie change variable 'threshb' to 0.8 instead of 0.95). This would certainly accommodate for the natural stability of the testing lock, by having the blue threshold line lie almost completely below the bulk of the peak height measurements. Even in this case, though, we can see in figure 6 that the lock does occasionally fail, and is soon corrected.

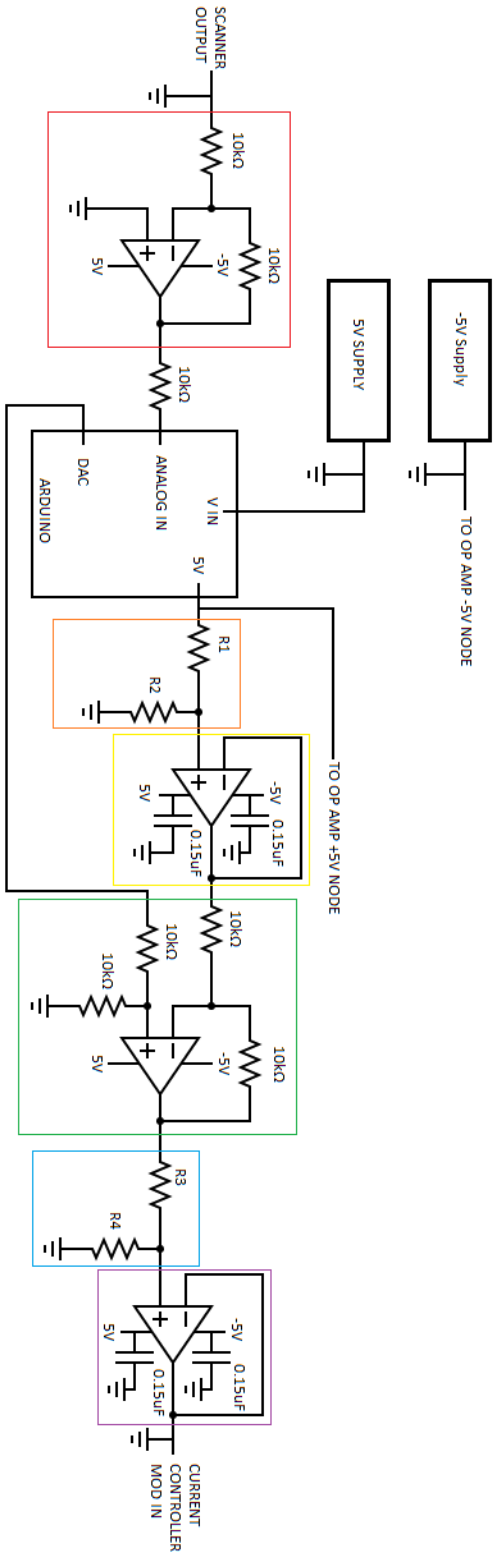

Figure 2: Circuit construction. (Red) Inverts the signal coming from the scanner output. (Orange) Resistor divider that shrinks 5V to 1.65V. (Yellow) Buffer amplifier. (Green) Differential amplifier that shifts the voltage range of the Arduino from 0V-3.3V to ±1.65V. (Blue) Resistor divider. (Purple) Buffer amplifier.

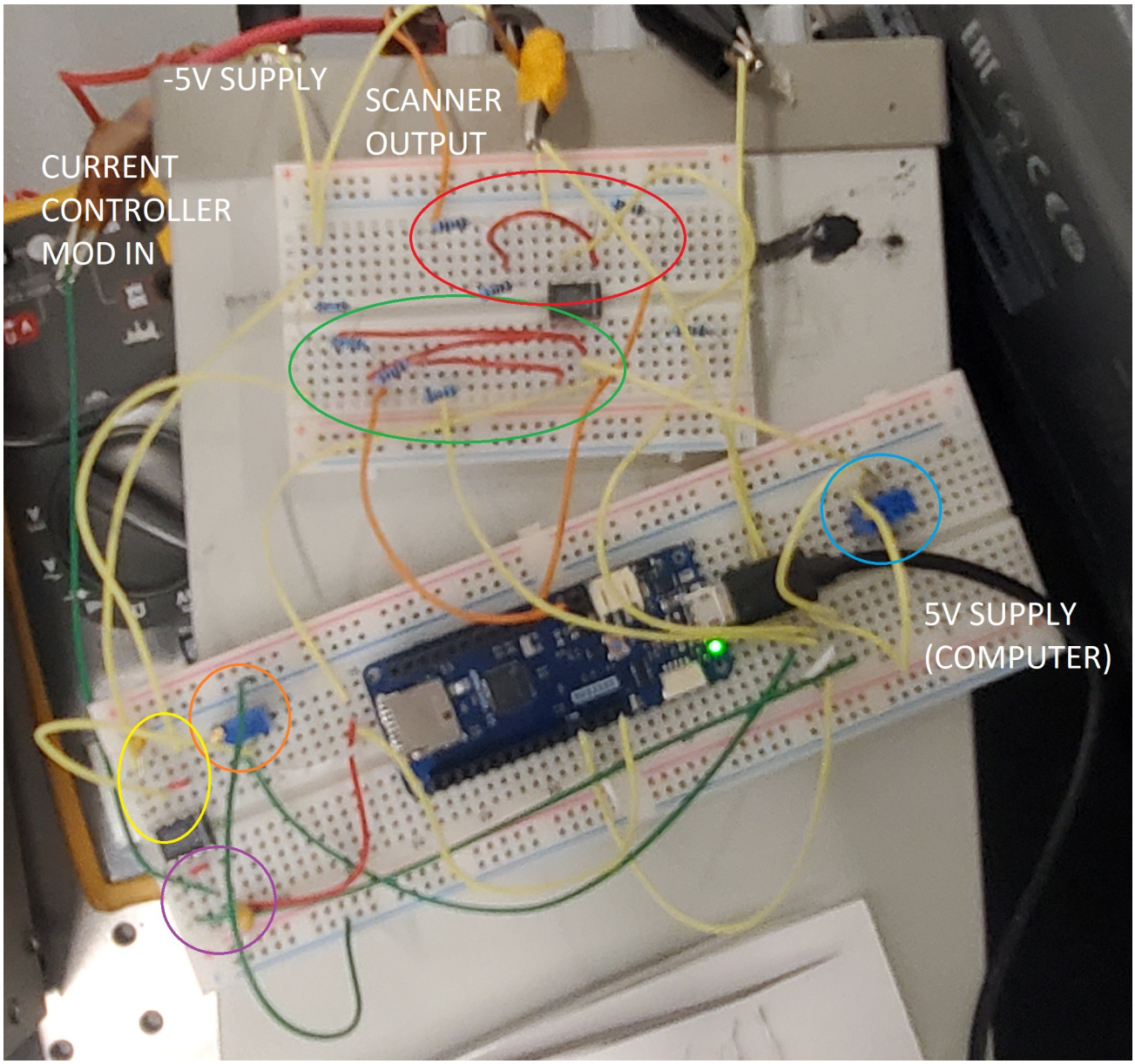

Figure 3: The circuit used in testing. Approximate locations of circuit sections are circled in corresponding colors.

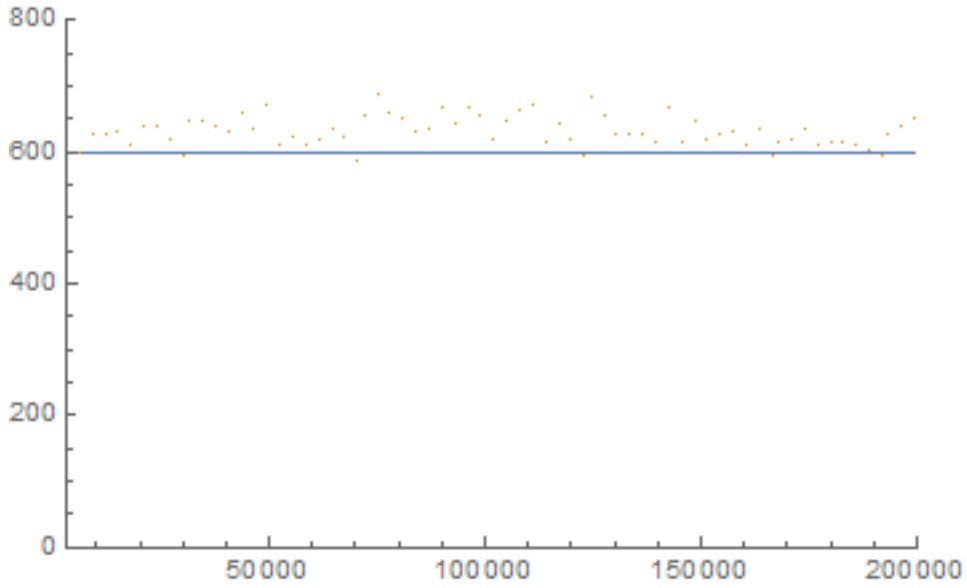

Figure 4: Average max peak height (mV) versus time (ms) for 200 seconds of operation. The threshold is also shown in blue.

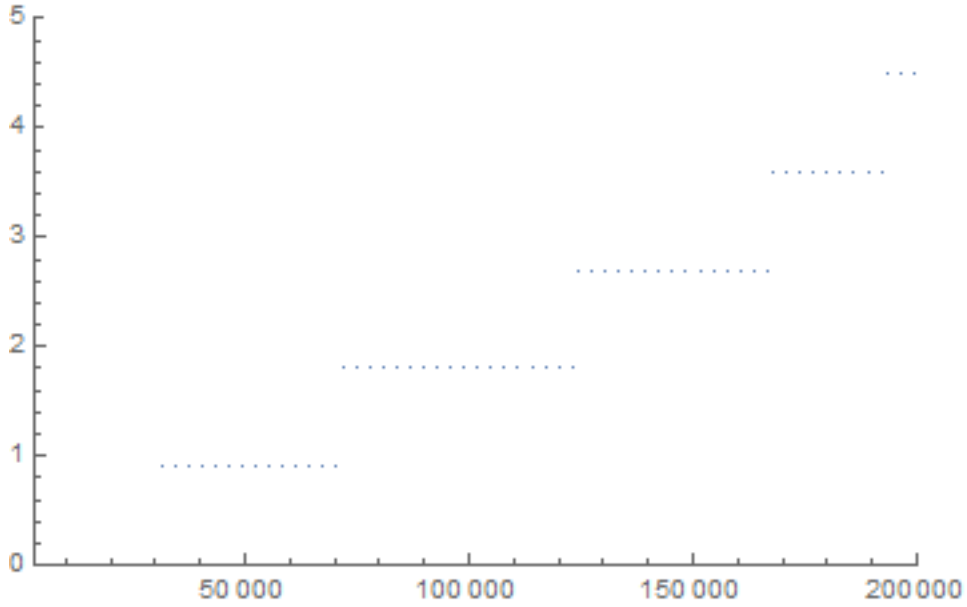

Figure 5: Current adjustment (mA) versus time for 200 seconds of operation.

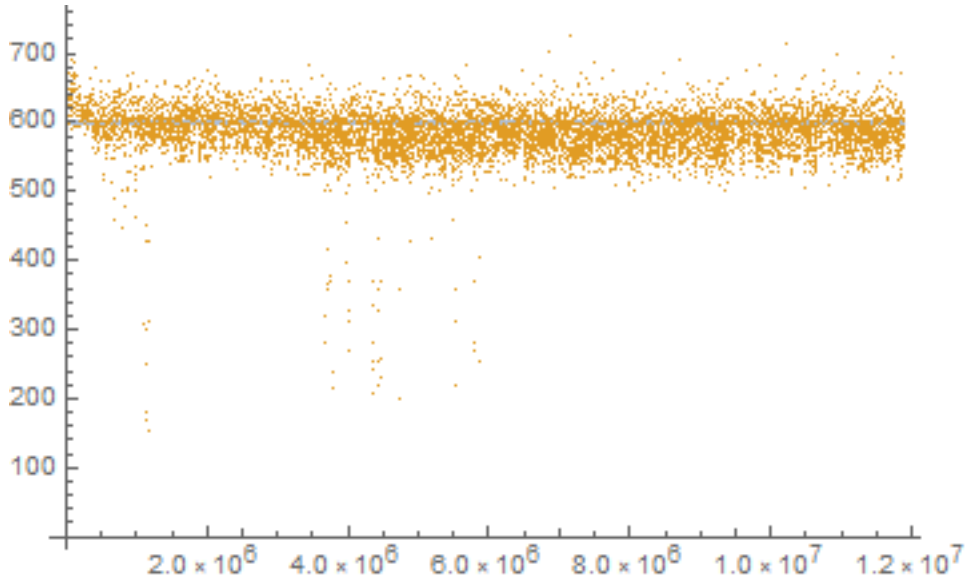

Figure 6: Average max peak height (mV) versus time (ms) for roughly 3 hours of operation. The threshold is also shown in blue.

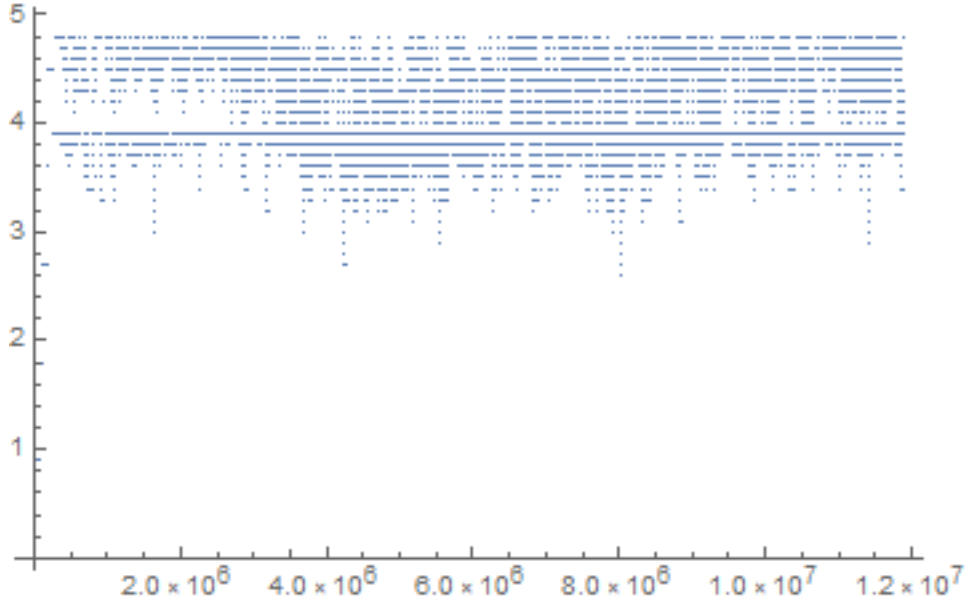

Figure 7: Current adjustment (mA) versus time for roughly 3 hours of operation.

6 Bluetooth connecting to serial monitor

Traditionally, if one wants to connect wirelessly to the serial monitor of their Arduino via Bluetooth, they would use an Arduino Bluetooth module HC-05 or HC-06, which connects directly to the board's RX and TX pins. Unfortunately, the most popular way to do this is through the Softwareserial library, which is incompatible with SAMD boards, including the MKR Zero. There appear to be some posts on the Arduino forums where someone has successfully implemented Bluetooth onto their MKR Zero (conveniently with no actual code posted), but as far as I can tell there are no existing instructions for how to do it. For now, a decent alternative may be to run a long wire from the box containing the board to a computer USB port and reading the serial monitor from there. Of course, having access to the serial monitor isn't necessary to run the program, but it is useful for debugging.

7 Full script

```
int analogPin = A2; // analog input
int outPin = A0; // DAC
int avgnum = 30; // number of times to compute peak max height before averaging
int avgnumtwo = 5;
float scantime = 0.1; // how long to wait before recording each peak height
float conv = 1023/3.3; // digtal to analog ratio
float a = 0.33; // used to convert current to voltage
float b = 1.65; // used to convert current to voltage
float currentjump = 1; //initial current increase in recovery mode
float currentrange = 5; //range of current arduino can tell current controller to do
int res = 10;
float threshb = 0.95;
/* do not change following code */
float sampleArray[1000]; // arrays to hold scanner data
float testArray[1000];
float threshold = 0; //variables used to compare peak height
float test = 0;
float amps = 0;
float maxsum;
void setup() {
  Serial.begin(9600);
  pinMode(analogPin,INPUT); // set analog pin mode to input
  analogWriteResolution(res); // set resolution of DAC
  analogWrite(outPin, (a * amps + b) * conv);
  maxsum = 0;
  //Serial.println("acquiring threshold...");
  for (int i = 0; i < avgnum; i++) { // fill array with scanner data
    for (int i = 0; i < 1000; i++) {
      sampleArray[i] = analogRead(analogPin);
      delay(scantime);
    }
    float max_v = sampleArray[0]; // determine max peak height and set as threshold
    for ( int i = 1; i < 1000; i++ ) {
      if ( sampleArray[i] > max_v ) {
        max_v = sampleArray[i];
      }
    }
    maxsum += max_v;
  }
  threshold = threshb * (maxsum/avgnum);
```

```
  //Serial.println("threshold acquired:");
  //Serial.println(threshold);
}
void loop() {
  analogWrite(outPin, (a * amps + b) * conv);
  test = maxsum/avgnum; // continuously test new peak data against threshold
  //Serial.println("printing test values:");
  while (test > threshold){
    float maxsumm = 0;
    for (int i = 0; i < avgnum; i++) {
      for (int i = 0; i < 1000; i++) {
        testArray[i] = analogRead(analogPin);
        delay(scantime);
      }
      float max_s = testArray[0];
      for ( int i = 0; i < 1000; i++ ) {
        if ( testArray[i] > max_s ) {
          max_s = testArray[i];
        }
      }
      maxsumm += max_s;
    }
    test =  maxsumm/avgnum;
    //Serial.println(test);
  }
  //Serial.println("lock lost!");
  //Serial.println("relocking...");
  delay(1000);
  while (test < threshold) { //recovery mode
    if (amps > currentrange){ // a safety mechanism to make sure
    // arduino isn't being told to output >5 volts
      amps = (currentrange - 1);
    }
    if (amps < -currentrange){
      amps = 0;
    }
    if (amps < currentrange && amps > -currentrange) { // initial current
    // increase to above lock range
      amps += currentjump;
    }
    analogWrite(outPin, (a * amps + b) * conv);
    //Serial.println(amps);
    while (test < threshold) { // run downwards in current by 0.1 until
    // lock is reestablished
      if (amps > currentrange){ // a safety mechanism to make sure arduino
      // isn't being told to output >5 volts
        amps = (currentrange - 1);
      }
      if (amps < -currentrange){
        amps = 0;
      }
      if (amps < currentrange && amps > -currentrange) {
        amps -= 0.1;
      }
```

```
      analogWrite(outPin, (a * amps + b) * conv);
      //Serial.println(amps);
      float maxsumm = 0;
      float minsumm = 0;
      for (int i = 0; i < avgnumtwo; i++) {
        for (int i = 0; i < 1000; i++) {
          testArray[i] = analogRead(analogPin);
          delay(scantime);
        }
        float max_s = testArray[0];
        for ( int i = 0; i < 1000; i++ ) {
          if ( testArray[i] > max_s ) {
            max_s = testArray[i];
          }
        }
        maxsumm += max_s;
      }
      test =  maxsumm/avgnumtwo;
      //Serial.println(test);
    }
  }
  //Serial.println("relocked!");
  //Serial.println("threshold:");
  //Serial.println(threshold);
}
```