ECE 1896
Senior Design

Youngblood Photonics Lab Photodetector
Final Design Document

Team 10

Prepared By:

Nick Gismondi

Kevin McGoogan

Malik Rivers

Brendan Schuster

Jeffrey Socash

Table of Contents

Table of Figures

Table of Tables

1. Introduction

1.1 Problem Description

Typical photodetectors are quite expensive. A single-channel analog photodetector can run upwards of $1,000. Because the output of these photodetectors are typically analog signals, ADC or the use of an oscilloscope is still required to measure your signal. Our goal is to create a photodetection system capable of detecting optical signals within a given wavelength and optical intensity bandwidth at a fraction of the cost of conventional photodetectors with the same analog functionality. This system will include variable gain and DC offset capabilities, high precision ADC, and direct interfacing capabilities with a host computer in Dr. Youngblood's Photonics Lab. Creating an open-source photodetection system will reduce the cost of measurement equipment and save time with computer interfacing.

1.2 High-Level Description of Prototype

The device will include a photodiode for optical input and two slots for 9V batteries to supply the analog circuitry.To match the typical single channel photodetector capabilities, our device will also include an analog output from the signal amplification using a general SMA connector. The analog-to-digital conversion and digital circuitry will be powered with a 5V DC power supply through a micro-USB port. The output data will be sent to the computer through a high speed USB to UART cable. Variable gain will be accomplished through the user making different jumper connections on the RC networks attached to the op-amps.

The physical device shall interface with a computer over USB. A library for the photodetector, written in Python, will be installed on the host computer. This library will contain functionality for interfacing with and configuring the photodetector, with the primary functionality being the ability to sample measured light intensity data from the photodetector over an arbitrary period of time and return the sampled data as a list of ordered pairs representing intensity as a function of time. The library shall also support exporting sampled data in forms that can be processed by other applications, such as CSV.

In order to provide a user-facing means of controlling the photodetector without needing to write scripts, a basic data acquisition graphical user interface (GUI) will be used to control the photodetector on the lab station. This GUI will allow the user to start and stop sampling and will contain a graphical display which plots the calculated light intensity with respect to time as the sampling occurs.

2. Background

2.1 Youngblood Photonics Lab

The motivation for this project stems from Dr. Youngblood's optical computing research. He has helped to develop systems capable of matrix multiplication, convolution, and memory storage entirely in the optical domain. These circuits require multiple optical inputs and produce multiple optical outputs. Typical photodetection requires expensive measurement devices which take a lot of time to set up. When you need to measure multiple inputs and outputs of an optical computing system, the cost can quickly

skyrocket. With the goal to create a single channel photodetector, Dr. Youngblood recommended that the team develop a photodetector similar to the Newport 2011-FC.

In the lab, setting up the optical measurement system is quite tedious. Optical signals travel in silicon waveguides as computations are performed on them. These waveguides are extremely small, on the micrometer scale, so matching fiber optic cables to the optical circuit's waveguide requires tedious manual movement of the optical chip. The strongest connection from the optical circuit to a fiber will output a maximum voltage reading of the circuit, indicating maximum light transfer out of the optical circuit and into the fiber. Our photodetector will be used to optimize connectivity out of the optical chip, and the customizable gain and variable DC offset settings will make it easy to observe a wide range of optical signals in terms of power, bandwidth, and wavelength.

2.2 Optical Computing

Figure 1 below shows the setup of an optical computing system with multiple optical waveguide inputs and outputs for an experiment that Dr. Youngblood contributed towards. Figure 2 shows the physical realization of a photonic memory matrix with 16 x 16 memory cells. Such a device requires measuring 16 output channels.

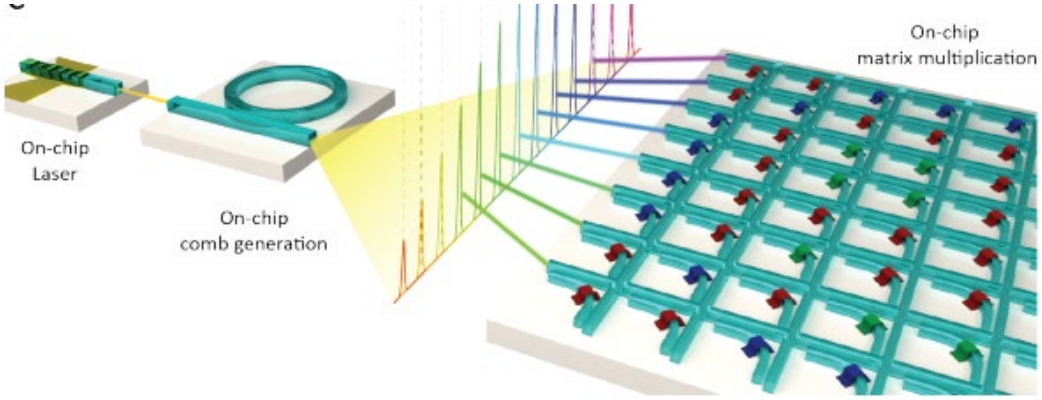


Figure 1: Optical Computing System Showing Multiple Input/Output Channels [1]

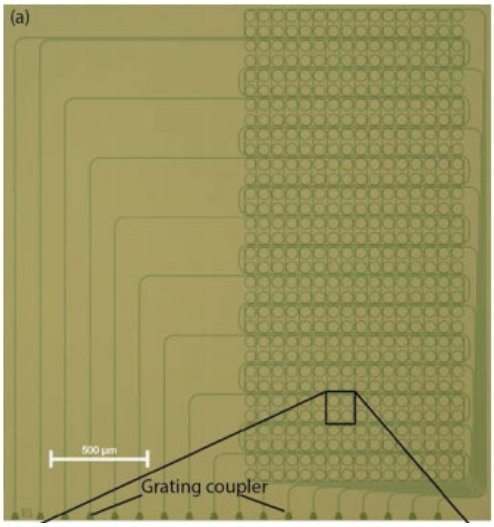


Figure 2: Optical Micrograph of a photonic matrix memory with 16 x 16 memory cells [2]

In such a system, all of these outputs need to be read simultaneously, similarly to a parallel bus on traditional memory devices such as DDR memory. Development of an open-source photodetector with the capability to directly interface with a computer for data acquisition will save time and money.

2.3 Evaluation of Existing Photodetector Solutions

Multi-channel photodetectors do exist, but they typically do not include customizable features, such as variable analog gain and DC offset capabilities. Furthermore, most photodetectors do not include any type of computer interfacing such as USB and only support analog waveform outputs that can be measured using an oscilloscope, which incurs additional costs for a lab setup.

Our goal is to create a single channel photodetector circuit that will easily interface with directly a lab computer. Within the current time constraints and the high data transfer speeds required to transmit the data to the computer, development of a multi-channel system photodetector is not feasible for the scope of this project. The proof of concept of a single-channel open-source photodetector with computer interfacing can easily be updated to include multi-channel functionality in the future.

This eliminates the need for a Data Acquisition system (DAQ), which would convert the analog output of the Newport photodetector to a digital system and send it to a computer. Including the cost of a DAQ to digitally obtain measurements, our open source photodetector would reduce the cost of obtaining measurements by about a factor of 10.

2.3.1 Newport 2011-FC Fiber-Optic Receiver

The Newport 2011-FC optical receiver features a low-noise InGaAS photodiode with variable gain. This is achieved using transimpedance amplifiers, which convert the photodiode current signal into a voltage signal. The Newport device's input wavelength spectrum is from 900 nm to 1700 nm, with a bandwidth of 200 kHz. The variable gain has a range of 90 dB in 10 dB increments, which allows for an input optical intensity range from 1pW to 10 mW [3]. The device is powered by two 9-V internal batteries. Below in Figure 3 the Newport photodetector is shown. The output SMA connector is the only output of this device.

Figure 3: Newport 2011-FC Photodetector

2.4 Photodiode Gain

The output voltage from a photodetector circuit does not tell the whole story. The electrical output does not indicate actual the incident light intensity, because the responsivity, noted R_λ, of the photodiode is wavelength dependent. The responsivity is defined as the current generated from the optical power of incident light, which gives us units of Amps/Watt. The wavelength of the incident laser light will always be known, and the responsivity curves of the photodiodes will be measured. The typical responsivity curve of the InGaAs-120L-FC is shown below in Figure 4.

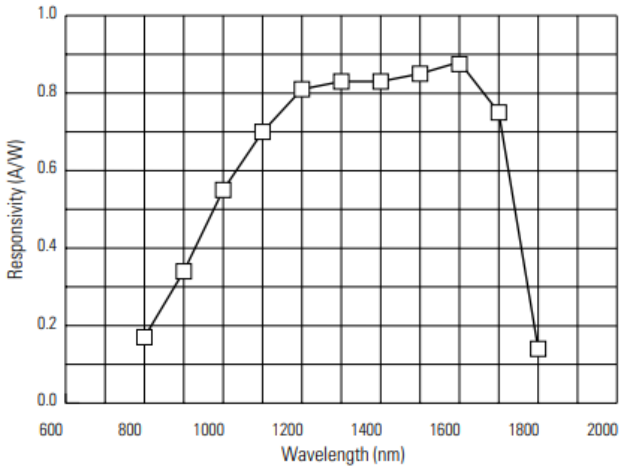


Figure 4: Typical Response curve of Newport 2011-FC Photodetector

Once a sufficient responsivity curve for a photodiode has been generated, calculating the input optical intensity can be completed. The gain of the circuitry is typically in Volts/Amps. This is because a photodiode is modeled as a current source, so our input source is represented by a current. The output of a photodetector is a voltage signal. Multiplying the two terms together, $R_\lambda * Gain$ results in units of Volts/Watt. By observing the analog output voltage waveform and knowing the incident laser wavelength, we can now determine the varying optical intensity of the incident signal on the photodiode.

3. System Requirements

The photodetector system consists of two user-facing components: the physical photodetector device and the data acquisition software that will control and read from the physical. Although the two components are closely related, they each contain their own individual needs

3.1 Hardware & Embedded System Requirements

3.1.1 The photodetector shall include a photodiode to take in an input optical signal.

The user will be able to take a laser in a laboratory setting and shoot it into the photodiode. The photodiode shall send a current into the system that will vary based on the input power of the laser device. The specific photodiode used is an InGaAs-120L-FC. This is the same photodiode found in the Newport 2011-FC device.

3.1.2 The device shall include a variable gain amplifiers

The user shall be able to send an optical input signal into the photodiode, which will generate a photocurrent. This current will travel into a transimpedance amplifier, where the user can adjust the gain via physical jumpers connected to various resistor-capacitor networks on the amplifier's feedback loop. This accomplishes two things; it converts the current signal into a voltage signal, and amplifies the signal. A second amplifier in the inverting configuration will allow the user to further alter the gain by a factor of 1 or 3. The user will adjust the gain by moving a physical jumper to add or remove an extra resistance into the feedback loop of the inverting amplifier.

3.1.3 The device shall have anti aliasing capabilities

Due to the nature of the input signals, aliasing is bound to occur, thus ruining the integrity of the signal output. To compensate, the device shall contain the ability to remove frequencies above the allotted threshold, thus allowing a more accurate output signal for the user to analyze.

3.1.5 The device shall have a 16-bit ADC conversion

The photodetector shall include a 16-bit ADC converter on the PCB at the request of Dr. Youngblood. The higher number of bits on the ADC will enable more accurate measurements.

3.1.6 The device shall be able to communicate with a computer

The device shall be able to transmit signal data to a computer over a USB connection. This interfacing will allow fast recording of optical inputs and outputs.

3.1.7 The device shall be able to run a program to process incoming signal data

The photodetector device shall be able to run a program to handle the processing and transmission of data to an external device.

3.1.8 The device shall operate with two separate power supplies

The device shall operate the analog circuitry using two 9V batteries. The digital circuitry shall operate on a separate 5V DC power supply.

3.1.9 The device shall include an analog output

The device shall include an output of the analog signal after amplification. This will match the black-box design functionality of the Newport 2011-FC photodetector and aid with testing the analog amplification.

3.1.10 The device shall have a linear operating range from 100 pW to 10 mW with a 200 kHz bandwidth

The device shall operate in a linear range to measure optical signals varying in intensity from 100pW to 10 mW. This is the same range that the Newport 2011-FC photodetector has.

3.2 Python Library and Data Acquisition Graphical User Interface (GUI)

3.2.1 The user shall be able to establish a connection with a photodetector plugged into the host computer

When the user opens the data acquisition GUI, they shall be prompted to specify the port which the photodetector is connected to. The prompt should allow the user to select the specified port from a drop-down menu of all ports with devices connected to them. This list of ports should be filtered so that only devices that are confirmed to be photodetectors are displayed.

3.2.2 The user shall be able to specify the input wavelength and gain configuration for a photodetector

Input wavelength (in nm) and gain configuration are two user-defined variables which are set physically outside of the GUI but affect the calculated intensity. The user must have a way of specifying these values in the user interface so that the intensity values calculated by the library are consistent with the physical settings of the device.

3.2.3 The user shall be able to control data sampling for the photodetector

The photodetector software should be able to initiate a sampling session which reads measurement data from the photodetector over an arbitrary period of time and creates a discrete mapping of sampling times to intensity values, starting at t=0 for the first sample collected. The user interface shall provide the user the capability to control when this sampling occurs and for how long. Samples shall be stored in a way where their units are unambiguous.

Due to discussed tradeoffs between sampling frequency and data processing speed in light of a request for real-time data sampling, two modes for sampling will be supported: asynchronous (i.e. “real-time”) sampling and synchronous (i.e. “precision”) sampling. Real-time sampling shall allow the user to sample indefinitely and asynchronously access sample data while sampling is occurring, with the tradeoff being a lower maximum sampling rate than precision sampling. Precision sampling will support higher sampling rates but requires the user to specify the sampling frequency and length of the sampling period

before the sampling occurs, and does not allow the user to access the samples until after this sampling period occurs. Dr. Youngblood's intended use of the photodetector aligns more with the needs provided by real-time sampling, but precision sampling is introduced for possible use cases where live-updating data is not necessary but a high sampling frequency is.

3.2.3.1 The user should be able to specify a desired sampling rate

Regardless of which sampling mode a user selects, the user should be allowed to enter a sampling rate for the device (in samples per second) while sampling is inactive. Updating this value should cause the software to send the new sampling rate to the device, and the device's embedded software should support changing sampling frequency. The user should be prompted if the sampling frequency they enter is not supported for the mode they wish to sample for.

3.2.3.2 The user shall be able to sample asynchronously (i.e. "real-time sampling")

The user shall be able to select an option to enable real-time sampling. The user shall be able to click a "Start Sampling" button on the GUI, which shall begin sampling data from the photodetector in real-time. The user shall only be able to initiate sampling if a sampling session is not currently active. If a previous sampling session has been stopped and there is sampling data present on the plot display, the plot should not be cleared. The latency from the user clicking the "Start Sampling" button to the first sample being measured by the photodetector shall be no greater than 1 second at any time.

The user shall be able to click a "Stop Sampling" button on the GUI, which shall stop sampling data from the photodetector. The user shall only be able to stop sampling if a sampling session is currently active. The latency from the user clicking the "Stop Sampling" button to the last sample being measured by the photodetector shall be no greater than 1 second at any time.

3.2.3.3 The user shall be able to sample synchronously (i.e. "precision sampling")

The user shall be able to select an option to enable precision sampling. The user shall be able to enter sampling frequency and sampling period for the precision sampling. The user should be able to set units for these values; otherwise they will default to samples per second and seconds respectively. After setting these values, the user shall be able to click a "Sample" button which begins precision sampling. The user should not be able to perform any other actions relating to controlling sampling or samples while precision sampling is occurring.

3.2.3.4 The user shall be able to reset the currently-collected samples

The user should be able to clear the device of all current samples if they wish to start a fresh plot. This should reset samples such that the next sample collected starts at t=0s.

3.2.4 The GUI shall plot calculated intensity with respect to time

The GUI shall contain a graphical plot of data acquired during sampling, with the x-axis measuring time displacement from the first sample and the y-axis measuring the calculated intensity data. A sample's data point should be connected by a straight line to the sample that came before it and the sample that comes after it. The user should be able to zoom in and out of the graph and drag the graph to display different

portions of the plot. The user should be able to choose the units that they wish to display time and intensity as.

3.2.4.1 The plot should support live updates as real-time sampling occurs

While real-time sampling is occurring, the graph should be redrawn with a frequency of no less than 30 Hz. The minimum refresh rate of 30 Hz was chosen as a compromise to satisfy a client request that the graph update in real time; 30 Hz was settled upon because it is reasonably attainable and it is half the refresh rate of a typical monitor, which should provide the appearance of real-time updating during measurement without appearing too slow.

3.2.4.2 The plot should update once after a precision sampling interval occurs

The plot shall not reset at a rate of 30 Hz when precision sampling is occurring; instead, the graph should only update once, after the precision sampling finishes. The graph should be redrawn such that the entire precision sampling interval is displayed.

3.2.4.3 The user shall be able to hover over a point on the plot and be shown the time displacement and intensity for the corresponding sample

The user shall be able to retrieve information for any sample by hovering their mouse over that sample's data point on the plot.

3.2.5 The user shall be able to retrieve the sample for which the maximum intensity occurred over the sampling period

The user shall be able to press a "Locate Max Intensity" button on the GUI which shall display the sample which contains the largest intensity for the collected samples. The user should be able to specify the interval over which they wish to search for this maximum intensity. This is a necessary feature in the context of the Youngblood photonics lab because the measurement of intensity with respect to time occurs simultaneously with the movement of a plate with respect to time, and finding the time at which the max intensity occurs allows the user to find the position of the plate when that intensity occurred.

3.2.6 The user shall be able to export the measured samples as a CSV file

The user shall be able to press an "Export to CSV" button on the GUI which shall take a subrange of the currently-plotted samples and generate a CSV file. The data shall be formatted such that each row represents an individual sample, with a sample containing the time since sampling started when that sample was taken, a comma, and the calculated intensity value. When the "Export to CSV" button is pressed, the user should be prompted to enter the desired save path for the file, the time interval of the samples to export, and the units for time and intensity that the user wishes to export the samples as. The time taken to write the measured sample data to the file should scale no worse than linearly with the amount of samples.

3.2.7 The code for controlling the photodetector shall be written in Python

The data acquisition GUI and all code to control the photodetector must be written using Python 3. Additionally, the data acquisition GUI must be built using components from the Dash and Dash DAQ libraries maintained by plotly for uniformity with other applications used in the lab [4].

4. Design Constraints

4.1 Budget

The total cost of components ordered to develop the prototype photodetector does not exceed the allocated ECE 1896 budget of $400. This includes components used to emulate hardware that are not part of the final prototype, such as Arduino boards.

4.2 Schedule

The scope of the project was limited by the deadline that a working prototype is expected by. The design for the printed circuit board was to be finalized by March 18, 2020 on account of the printing deadline. The prototype was expected to achieve the minimum standard for completion by April 13, 2020, when final presentations begin, and was originally intended to meet acceptance criteria for performance before the Swanson School of Engineering Senior Design Exposition on April 16, 2020.

4.3 Precision of Data

Due to the requirement of the device to measure intensity on the order of picowatts to milliwatts, precision of measured data is paramount. Calculations done by software will not be able to use floating-point numbers to store and calculate values such as time and intensity and will instead need to use data types that support arbitrary-precision decimal values, which will require additional optimization due to a larger data size.

Quantization error also arises when we send the analog waveform through the ADC. From the ADC specification sheet, the maximum quantization error is given by the following equation:

$$E_Q = \frac{V_{REFH} - V_{REFL}}{2^N}$$

V_{REFH}can have a minimum value of 1.13V to a the typical value of the supply voltage $V_{DDA} = 3.3\ V$. V_{REFL}has a typical value of 0V. This means we will have a maximum quantization error of 50.354 µV and a minimum error of 17.242 µV [5].

4.3.1 Analog-to-Digital Converter Precision

The request to support 16-bit ADC in our design limits our choices when designing the photodetector device, as many microcontrollers with on-chip ADC do not support 16 bits. Given an analog input window of 3.3 V of many ADCs, the target resolution is 50 µV. The accuracy of the analog circuitry

therefore should be about 25 μV, which is ½ of the least-significant bit's resolution. In order to accomplish this, extremely precise resistors need to be used and the op-amps must operate in their linear regions.

4.4 Serial Communication Speed

In order to meet the clients request of processing data in real time, communication between the photodetector device and the python module must occur at a minimum rate of 8 Megabits per second. The following equation outlines such logic:

(400 KHz Sample) * [(16 bits ADC) + (4 bits stop/start/parity)]
=
8 Megabits per Second

4.5 Through-Hole Design

After the campus closed, students had to construct the final PCB prototype needed to be constructed off-campus. All necessary tools for the first prototype construction were no longer available, and the components and board design needed to be changed to accommodate larger components that could be easily integrated into a system with a generic soldering iron. The first prototype included almost all surface mount components in smaller packages, but the second and final prototype switched the packages to through hole and large (1210) surface mount components.

Because of the through hole requirement, the team sacrificed precision for availability for components. This affects the exact gain of our analog conversion and amplification through voltage gain of our new resistor selection. The reference voltages created for the photodiode and the ADC also were switched to through hole zener diode voltage references. The team did not have the ability to test the performance difference between the surface mount and new through hole reference chips. Op-amp availability is affected the most by switching to through-hole packaging, but luckily the team had already purchased and constructed surface mount-to-through hole breakouts for the initially selected operational amplifiers.

4.6 Runtime Performance of Python and Dash

The software design was constrained due to the requirement that the photodetector software on the host computer be written in Python. Unlike compiled languages which are known for performance such as Rust and C++, Python is largely an interpreted language and does not have standard facilities for compilation and optimization to machine code. Python's interpretation and the runtime required to support it makes it difficult to write Python code with good performance, particularly in the context of the photodetector and the large amount of data it samples per second. Although Python modules can be partially composed in compilable C code, doing so requires a high level of sophistication with Python and C; does not allow the full leveraging of Python features such as dynamic typing, list comprehensions, and many Python libraries which are of use; and generally results in code that is difficult to read and write. In the interest of remaining within the scope of a senior design course and keeping the photodetector library maintainable, the decision was made to keep the library and GUI code pure Python for the sake of

producing a working prototype, prioritizing the manageable implementation of desired functionality over premature optimization.

Additionally, the request of Dr. Youngblood to produce the GUI using Dash components provided another layer of performance concerns which would persist throughout the design and implementation process. Rather than just being a GUI component library (which the team assumed when agreeing on the requirement to use Dash), Dash is a framework used to build server-side apps which can be hosted on a web server and viewed in a web browser. This means that to run a Dash app locally, a local server instance must be run and furthermore wrapped as a desktop application using a third-party library, which carries its own performance concerns. Furthermore, the Dash framework uses Python code to generate HTML, CSS, and JavaScript code automatically for the app, which means that developing a Dash app doesn't afford many opportunities for manual optimization. Despite this, Dash provides a very attractive set of components for managing data acquisition and plotting, so the decision was again made to prioritize working functionality before performance optimization. Thus the host computer software developed in this project constitutes a compromise on performance for manageable implementation; this report shall later discuss alternatives considered after development finished which may have improved performance.

4.7 Inability to Leverage Lab Station Performance

The software was originally planned to be tested on the lab station in the Swanson School of Engineering's Photonics Lab. At the time of measuring lab station specs, this lab station ran a 64-bit Windows operating system, used an Intel Core i7-9700 CPU @ 3.00GHz, and had 32GB of physical memory installed. Due to the closure of campus facilities due to COVID-19 before the data acquisition GUI was in a suitable testing state, the team was unable to test the software with the intended lab station. Instead, testing for the software was primarily done on a team member's personal computer. This personal computer ran a 64-bit Windows 10 Home operating system, used an Intel Core i5-4690K CPU @ 3.50GHz, and had 8GB of physical memory installed at the time testing was done. That testing had to be done on a personal computer with inferior processing power and available memory is significant; there is no definitive way to know if the same issues with trading off performance and ease of implementation as described in 4.6 would have been an issue on a lab station powerful enough to run similarly-demanding lab software. At best, educated guesses can be made as to whether or not the final design choices made would have allowed the system to meet performance criteria on the lab station.

5. Evaluation of Design Concepts

The initial design for the analog amplification circuitry was developed from researching the Newport 2011-FC photodetector. The user manual for these devices specifies the use of op-amps for variable gain amplification and filtration. Dr. Youngblood recommended using the OPA192 operational amplifier, which has very low drift current, rail-to-rail I/O, and small temperature voltage fluctuations. After selecting the operational amplifiers, the architecture for the variable gain was developed.

Physical circuit reconfiguration seemed to be the best way to manually change the gain of the circuit. Jumpers on the board provided the cheapest and easiest way to realize such a system. This allowed the user to alter the op-amp feedback paths to include various RC networks. The of the first operational amplifiers resistors were chosen to create transimpedance gains of 10, 100, 1000, and 10000,

and the feedback capacitors were chosen through guess-and-test SPICE simulations. Each circuit's frequency response was simulated, and the capacitors were chosen to ensure a 3-dB bandwidth of 200 kHz.

The anti-aliasing filter was developed using an online tool that allowed us to create a circuit given the parameters we fed into the simulator. This tool also allowed us to adjust the bode plot in order to create the most efficient topology. With a circuit design in place, we moved to the prototyping stage of the filter. From the original schematic provided to us by the online tool, many of the component values had to be altered and adjusted given the original values were of very specific component values that did not exist for us in the lab environment. Once the filter was constructed, a frequency sweep was performed in order to test the filter's capabilities. These tests show that while the filter was able to remove frequencies past 200kHz, there was a large spike in the frequency response of the filter. This revealed the circuit contained a high quality factor, most likely caused by some of the inaccuracy of the components. The filter prototype was later integrated with the variable gain circuit and frequency response data showed a similar response with the same spike of the quality factor.

The digital layout was the most challenging part of the design, because of the high precision ADC reduced component availability and increased data communication speed requirements. A serial ADC was selected in lieu of a parallel CMOS ADC to simplify IC integration and because the software team had experience working with serial communication before. Dr. Dickerson recommended using a Teensy development board to handle the ADC and digital data manipulation. Teensy Microcontrollers offer powerful microcontrollers, arduino bootloader compatibility, and on-board ADCs. The Teensy 3.2 was selected because it includes an on-chip 16 bit ADC and microprocessor with sufficient sampling and clock frequency speeds.

The data acquisition software on the host computer was developed in Python at the request of Dr. Youngblood, so the software team made use of the serial python library. This library can instantiate driver software to incoming read serial data from the microcontroller. In order to send the data from the microcontroller to the host computer and receive the data as serial, the team decided to use a CS232HD USB 2.0 High-Speed to UART cable. This takes the serial data from the microcontroller, and mimics serial protocol through a host computer's USB 2.0 port. This eliminated the development of our own driver software and allowed the software team to utilize the serial python library for data reception, while ensuring that the system could still achieve the required bit rate for 16 bit - serial data transmission.

6. Final Design

6.1. Overall System Design

Figure 5 shows a system level design of the data conversion. The optical signal is converted to an analog electrical signal and amplified by the PCB, where it is sent into an ADC. The analog amplification also includes anti-aliasing capabilities to ensure that a bandwidth of 200kHz is realized. The digital signal is then sent to the Desktop computer using a high-speed UART-USB converter and eventually displayed on the host computer. Software on the host computer is divided into two subsystems: the Photodetector Library, which is strictly concerned with reading and storing samples on the device, and the Data Acquisition GUI, a proof-of-concept application which utilizes the Photodetector Library's functionality to visualize the data read from the device and provide methods for exporting this data.

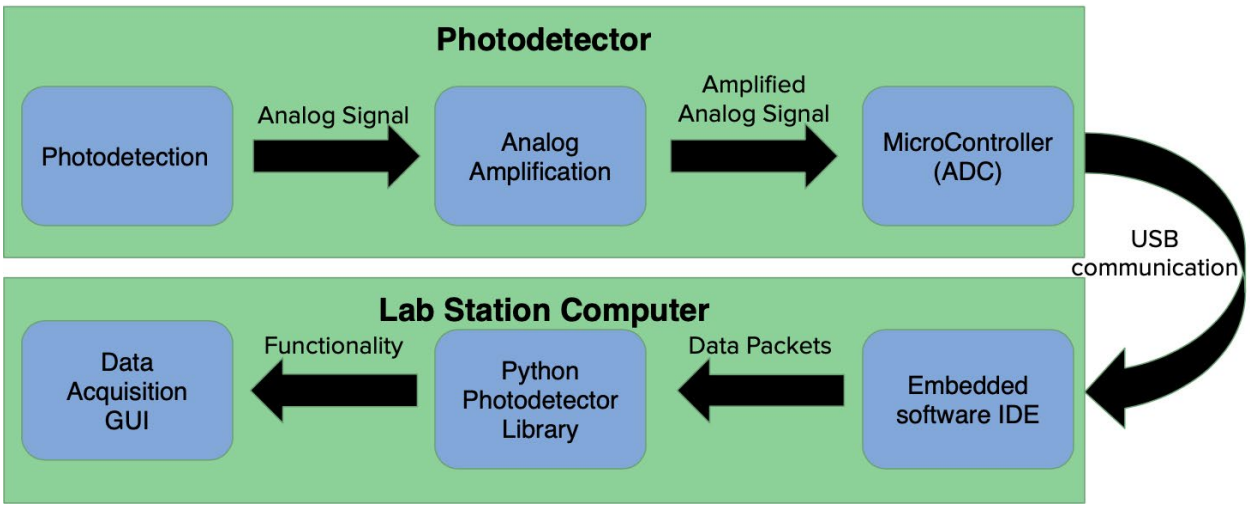


Figure 5: System Level Design

6.1.1 System Architecture

Figure 6 below shows the top level design of our hardware. It has four inputs: optical input, analog power supply input, digital power supply input, and a serial data interface. There will be two outputs of our circuit: an analog output SMA connector, and a serial data output interface cable that will plug into the host computer. The analog powering input has two sockets for standard 9V batteries, and the digital power supply is realized with a 5V DC power supply input through micro-USB.

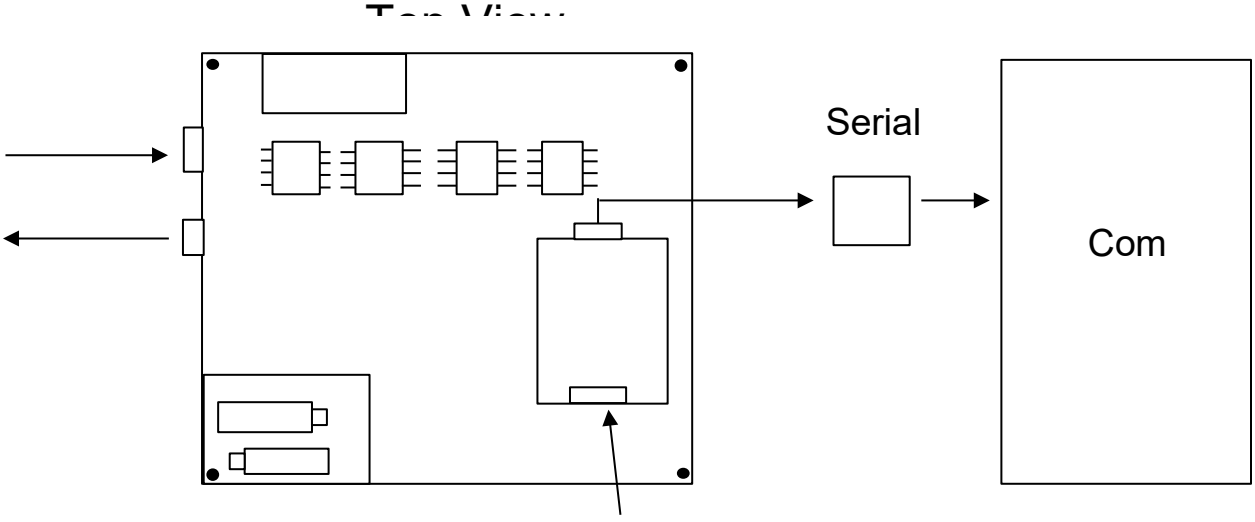


Figure 6: Top Level Physical Design

Figure 7 below shows a side view of the same circuit. Because the photodetector circuit will be used on an optical measurement table, four standoffs are used to secure the circuit to the optical table. These standoffs are compatible with ¼-20” screws and sit 5 x 5 inches apart to match the optical table’s threaded holes.

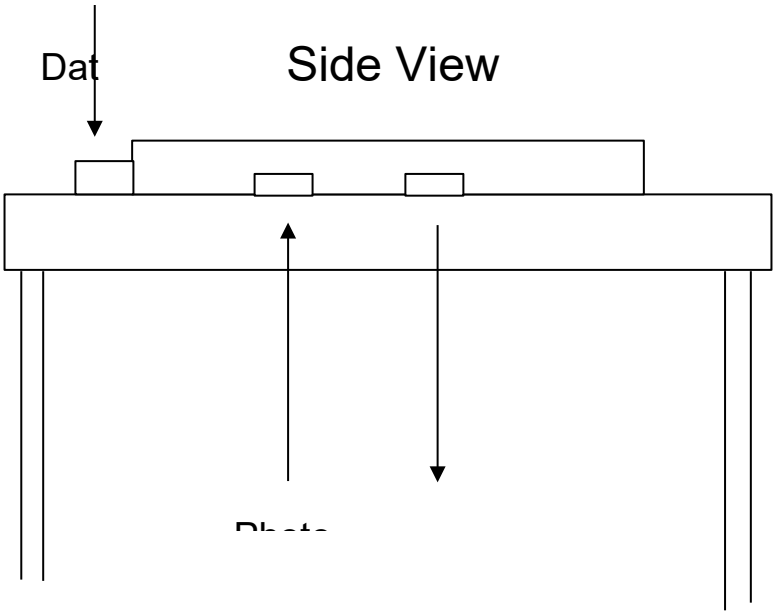


Figure 7: Side View Showing Standoff Configuration

6.1 Hardware Design

6.1.1 Analog Layout

Figure 8 below shows how the gain will be achieved. A transimpedance amplifier configuration will be used in series with an inverting op-amp. The gain of this circuit is in terms of Volts/Amp. The voltage source and R4 arbitrarily represent a current source. Because of the virtual ground on the inverting pin of the first op-amp, the input current can be represented by: $I_{in} = V_{cc}/R_4$. The overall gain of this can now be calculated.

$$Gain = R_1 * (-\frac{R_3}{R_2})\, V/A$$

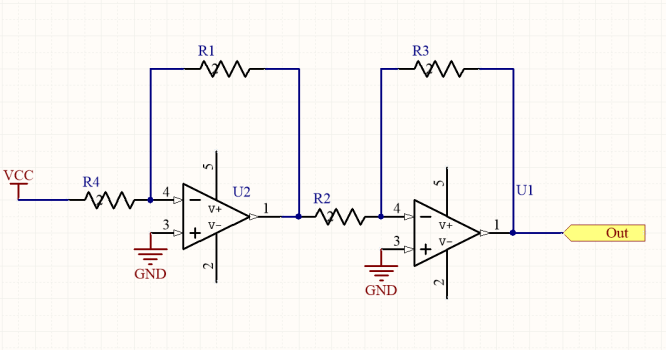


Figure 8: Transimpedance Op-Amp Gain

The OPA192 was chosen as our primary op-amp for it's rail-to-rail Input/Output, Low offset voltage, and low input bias current. These characteristics make it perfect for detecting small currents (from an active photodiode), but the GPB of 10 MHz means that the highest gain achieved at 200 kHz is 5

V/V. This means a gain of 10,000 V/V would be impossible to achieve at 200 kHz, but a frequency sweep on our initial breadboard prototype was able to achieve a 200 kHz bandwidth at most gain settings. Dr. Youngblood said this is the op-amp he used when he was originally trying to design his own photodetector, so the hardware team decided to stick with it.

Instead of just connecting an op-amp (OPA-192) to only a single resistor-capacitor pair, it will be attached to four different pairs, each of which will induce a different gain factor (see Figure 9). All of these resistor-capacitor pairs will be fed into a header, which will allow the user to choose which gain setting to apply. See Table 1 below for the specific values of both the resistors and capacitor as well as the gain which they induce. The output from this is then fed into another op-amp in the inverting configuration which will give the user the option of amplifying their signal by a factor of 3 or 1. This is achieved by setting R9 = R5 = 1 kΩ and R7 = 2kΩ.

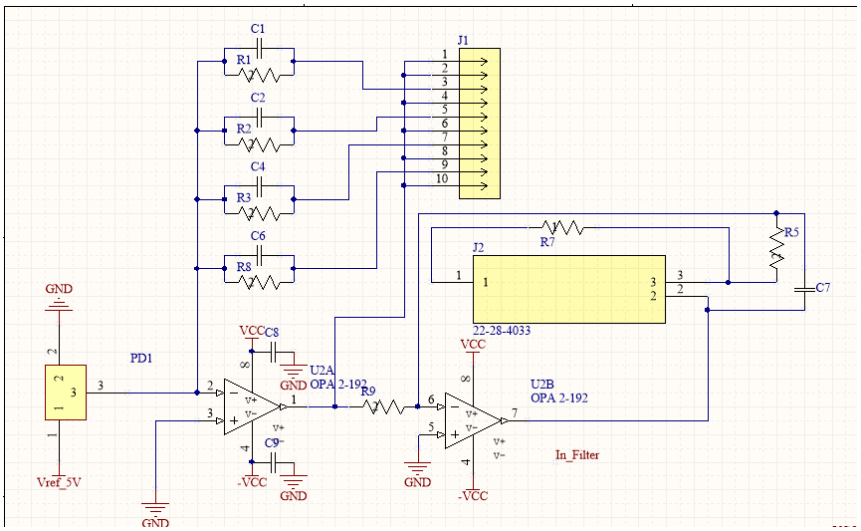


Figure 9: Circuitry to produce specified gain and DC offset

Capacitor	Resistor	Gain
C1 = 0.1 μF	R1 = 10 Ω	10
C2 = 1 nF	R2 = 100 Ω	100
C4 = 10 pF	R3 = 1 kΩ	1000
C6 = 5 pF	R8 = 10 kΩ	10000

Table 1: Resistors and capacitors which control gain settings. Resistor tolerances are 1% and capacitor tolerances are 20% If U1 connects pin 1 to pin 3, each of these gain values is multiplied by 3.

The output from the transimpedance amplifier will then be fed into the Sallen-Key low-pass filter which can be seen in Figure 10. This filter will attenuate any frequencies above the threshold of 200 kHz, which is the maximum frequency of the optical signal our photodetector will support with linear measurement. This filter will act as an anti-aliasing filter, so that signal integrity can be maintained

throughout the analog-to-digital conversion. Table 2 shows the values that were chosen for resistors and capacitors.

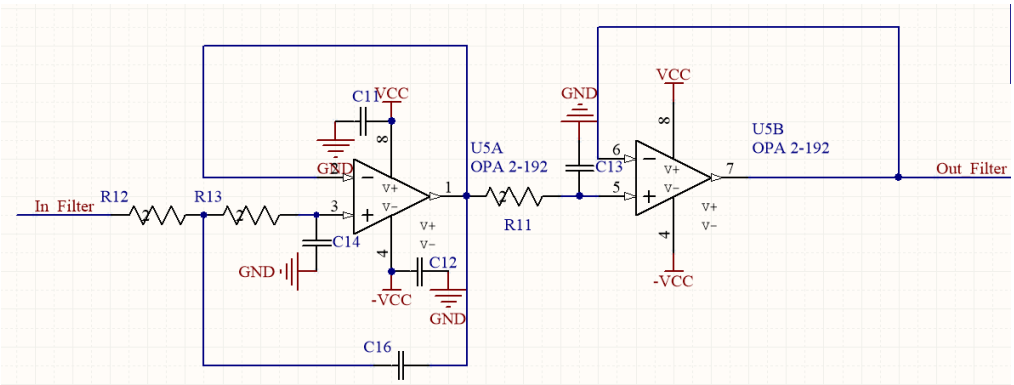


Figure 10: Sallen-Key Anti-aliasing filter

Name	Value
R11	1 kΩ
R12	1.4 kΩ
R13	4.99 kΩ
C11	0.1 µF
C12	0.1 µF
C13	1 nF
C14	100 pF
C16	1 nF

Table 2: Component Values in Sallen-Key Filter

As can be seen in Figure 11, after the anti-aliasing filter, the signal will follow two different paths. One of these paths will lead to an SMA output connector which can be fed directly into an oscilloscope to more closely match the design of the Newport 2011-FC photodetector. The other path will drop the voltage from a 9V to a 3.3V analog signal through resistor division, then feed into a voltage follower. This will ensure that there is no output impedance going into the ADC on the microcontroller, and the resistor division will ensure that no voltage above 3.3V will be placed on the analog pins on the Teensy 3.2, which can permanently damage the circuit. In the software, this means that the actual gain going to the ADC is the large gain value (G) times the voltage division, which is 3.3 / 9.

$$Real\ Gain\ =\ Analog\ Gain\ (G)\ *\ \frac{3.3}{9}$$

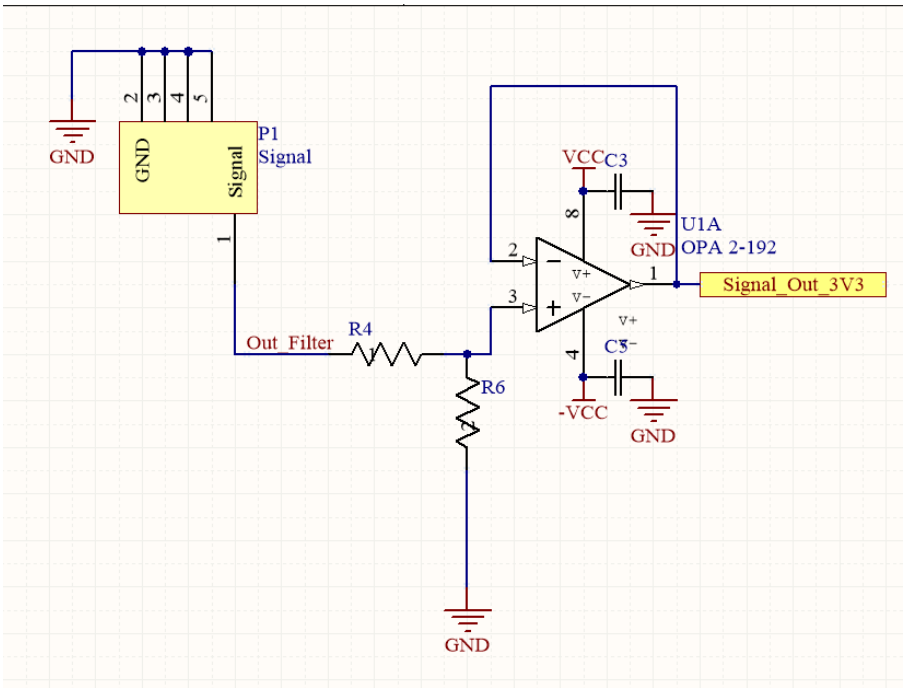


Figure 11: Voltage follower with SMA connector

This voltage division is realized with R4 = 172.72 kΩ and R6=100kΩ. The output from the voltage follower (see Figure 10) will be sent to the A0 pin on the Teensy 3.2. The analog ground return path will be sent to the AGND pin on the same Teensy, and the analog 3V3 signal will be our voltage reference for the ADC conversion.

6.1.1.1 Analog Powering

Two 9 volt batteries will power all of the analog circuitry. All of the op-amps power supply pins are connected in parallel close by to 0.1 µF ceramic capacitors.

The resistors for correct biasing are selected by determining the minimum and maximum current and voltages going into and out of the reference ICs. Because both reference chips utilize zener diodes in the breakdown region, we must ensure that enough current is flowing through the diodes at all times to produce the desired reference voltage. Figure 12 from the datasheet of the ZXRE330ASA-7 shows the setup and calculation of the correct resistor to place between the 9V power supply and the voltage reference [6].

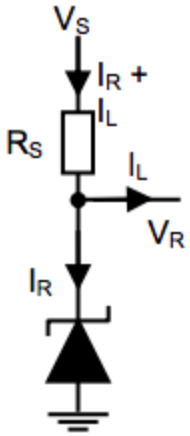


Figure 12: Analog Reference Biasing

$R_s$for each the 5V biasing circuit is calculated using the supply voltage ($V_s = 9V$), the biasing voltage ($V_r = 5V$). The sum of the load and operating current was estimated to be small, because the photodiode creates current and does not draw any. Therefore, an estimation of 1 mA for both was used. The calculation follows:

$$R_s = \frac{V_s - V_r}{I_L - I_R}$$

$$R_s = R_{10} = \frac{9 - 5}{0.001} = 4\ k\Omega$$

A resistor value of $2.7k\Omega$was chosen as a conservative value in the case more current was drawn.

$R_s$for the 3.3V analog ADC reference was chosen in the same manner. The reference voltage this time was changed to ($V_r = 3.3V$). The load and reference current were also chosen to be 1 mA. The maximum current of the analog voltage reference (AREF) pin is 1.7 mA [7], but a total current input estimate of 1mA was once again used. This estimation is based on the fact that external analog references on an ADC do not draw much current. The calculation for this resistor is shown below:

$$R_s = \frac{V_s - V_r}{I_L - I_R}$$

$$R_s = R_{14} = \frac{9 - 3.3}{0.001} = 5.7\ k\Omega$$

$R_s$was chosen to be $5k\Omega$ to compensate for any additional current that would be drawn into the ADC.

The analog circuitry will also include the LM404AIZ-5.0 VREF shunt zener diode reference chip. This will be used to create a 5 V bias to place the photodiode in photoconductive mode. This increases the depletion region in p-n the junction, and decreases the junction capacitance of the photodiode, improving the response time and linearity of the overall system [3]. Figure 13 below shows how we will create the 5V reference voltage to bias the photodiode and Figure 14 shows how the analog 3.3 V reference for the ADC will be created. The voltage reference for the ADC is accomplished through the ZXRE330ASA-7 VREF shunt zener diode chip.

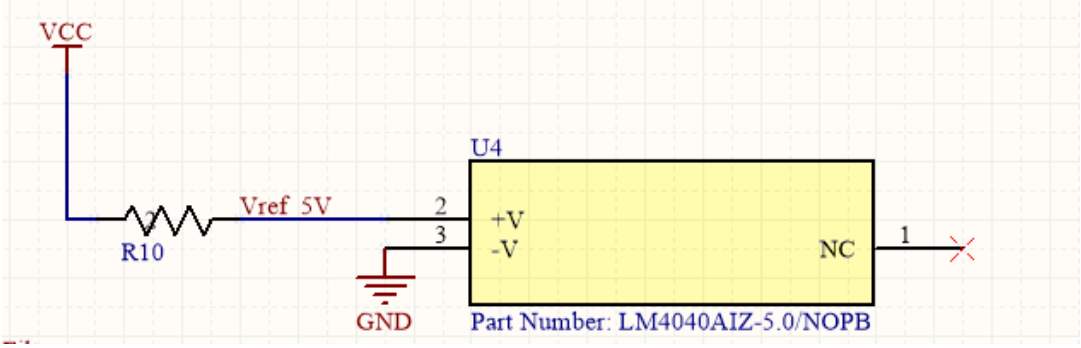


Figure 13: Circuit to create reference voltage

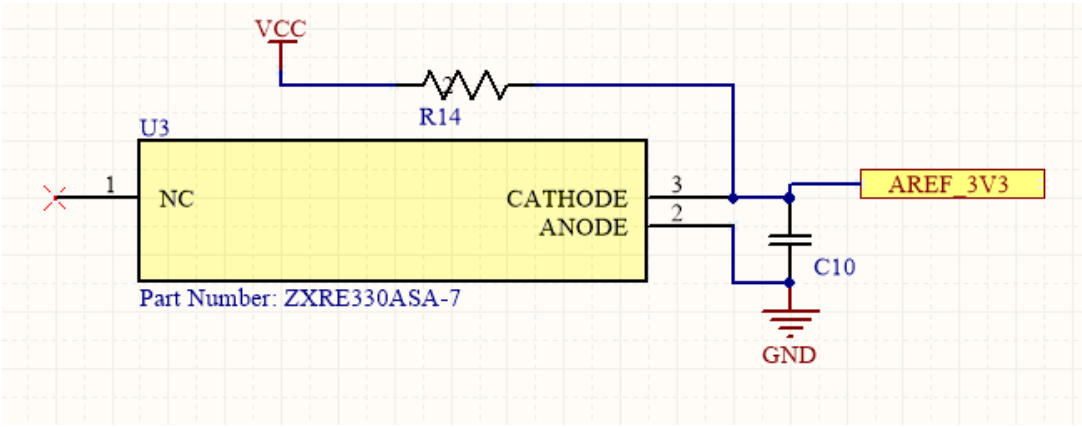


Figure 14: Circuit to produce Analog Reference

6.1.2 Digital Layout

The digital layout will be accomplished using the Teensy 3.2 USB development board to handle both the ADC and microcontroller data processing. This board will be clipped into the circuit after male DIP pins have been soldered onto the Teensy, and female DIP sockets are soldered onto our main PCB. This allows the user to manually program the Teensy and then place it into the PCB, as well as easily remove it from the PCB if the user desires to use just the analog output.

Interfacing with a computer will be accomplished using the C232HD USB 2.0 Hi-Speed to UART Cable. This component uses USB 2.0 protocol on the chip to handle USB specific firmware programming. This eliminates the need to write our own driver software. The chip mimics serial communication on the computer side, allowing the utilization of existing serial python libraries to finish processing the data before displaying the resulting waveform on the GUI. The output data from the microcontroller will be sent out on the TX1 pin of the Teensy 3.2 and into the TX pin of the CS232. The RX pin of this chip will connect to the RX1 pin of the Teensy 3.2. It will bring data into the microcontroller, enabling remote data acquisition from the host computer. Figure 14 below shows the top level functionality of the CS232HD component.

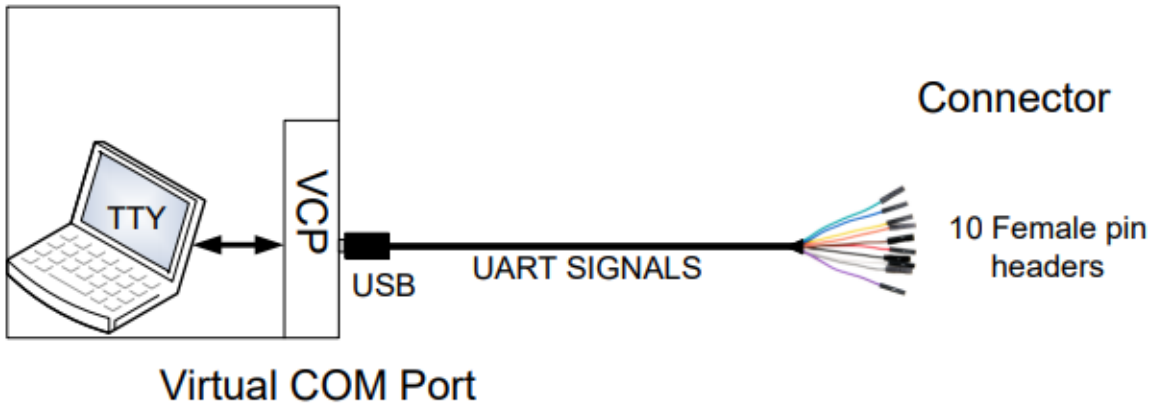


Figure 15: CS232HD Black Box Diagram

6.1.2.1 Digital Powering

In order to maintain longer battery life and more stable analog power supply with the 9V batteries, a separate digital power supply will be included in the design. This will include a 5V DC power supply that connects directly to the micro-USB connection on the Teensy 3.2.

6.2 Embedded System Design

6.2.1 High Level Description

The embedded system responsible for capturing all signal data is shown in Figure 16. The role of this system is to digitize signal data and relay it to the python module in real time using serial communication protocol. The embedded system polls a serial port and waits to receive a command from the host computer. The embedded system can receive two commands from the host computer: start sampling and stop sampling. When the start sampling command is received, the embedded system will continuously write 16-bit digital voltage values to the host computer at a maximum of 60,000 samples per second (30 KHz). When the stop sampling command is received, the embedded system will stop sending values over serial.

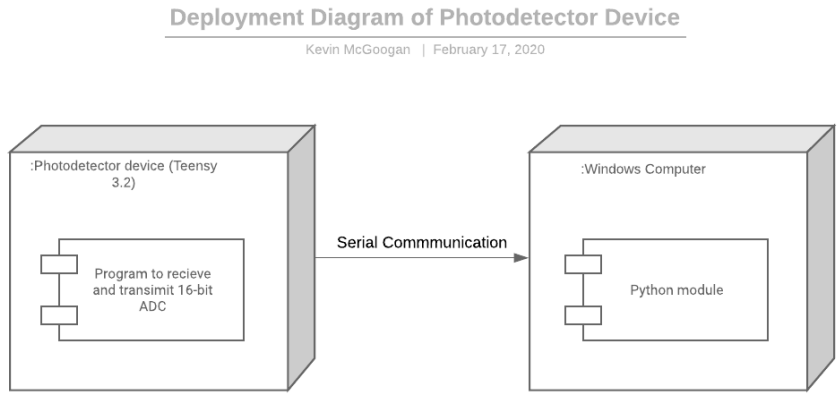


Figure 16: Deployment of Photodetector Device

6.2.2 Data Processing

All on-board device processing is handled using the Teensy 3.2 development board. An Arduino bootloader is used to service the device's operating system

6.2.3 Data Retrieval

The 16-ADC is accomplished on the Teensy 3.2 Microcontroller Development board. The analog output (Out_Signal) is fed into the ADC of the Teensy Development board. The conversion and data processing occurs on the microcontroller by running firmware on top of an arduino bootloader.

6.2.4 Data Transmission

After the Teensy development board receives a 16-bit signal, the firmware immediately outputs the signal using the Teensy's UART interfaces. Data transmission is facilitated by the C232HD USB 2.0 Hi-Speed to UART cable. The cable is able to reach speeds close to 12 Mbauds, appropriate for 30 KHz sample speeds. This cable relays incoming data to a windows computer where a python module is able to handle further processing.

6.3 Photodetector Library

The software to be used on the lab computer to interface with the device was designed with a consideration for modularity and loose coupling between functionality for interacting with the device and functionality for operating on data from the device. Thus, host computer software development was split into two subsystems: an application-independent Photodetector library, which seeks to provide an interface to control the device using Python while not restricting possible future usage of the interface, and a Data Acquisition GUI, which utilizes this Photodetector library to provide the data acquisition functionality and interface requested by Dr. Youngblood for this project. This section is concerned with describing the Photodetector Library, which was developed as the first of the two subsystems.

6.3.1 High Level Description

The Photodetector Library consists of a Python 3 module called youngblood_photodetector. It provides a software interface for communicating with the photodetector's embedded system over a Serial communication protocol by sending commands and receiving data in response. The photodetector library is concerned specifically with functionality related to controlling the serial device; it retrieves, stores, and provides a way to access the data sampled by the device, but leaves it to applications and modules (such as the Data Acquisition GUI, which is discussed in 6.4) to perform operations on this data. The youngblood_photodetector library has the following responsibilities: verifying that a serial device is a photodetector, providing a means to configure the device's responsivity and gain measurements, allowing the configuration of input parameters for the device (i.e. input wavelength, gain setting, and sampling rate), sending commands to start and stop asynchronous or synchronous sampling on the device, collecting 16-bit digital voltage data from the device while sampling is active, converting digital voltage

data to intensity samples with time displacement measurements, and providing this sample data in a form that is unambiguous with respect to units. In short, youngblood_photodetector provides a means of interfacing with the photodetector over serial to configure the device, collect data from the device, and provide access to that data in a form that can be utilized by any applications that need it.

6.3.2 Identifying Core Behavior of Library

The youngblood_photodetector library was developed and verified before the development of the data acquisition GUI, but much of the set of necessary functionality for the youngblood_photodetector library was identified by considering the immediate context of the data acquisition GUI, as the GUI was the desired product which necessitated the development of a photodetector device library. One way this was done was through the creation of sequence diagrams to trace the interactions between the user, the GUI, the library, and the serial device. The primary sequence diagram which was used to establish the functionality of the youngblood_photodetector library as a facilitator between user applications and the serial device is shown in Figure 17. It encompasses the following procedure for asynchronous sampling, the original functionality that was considered when designing the photodetector interface:

1. The Data Acquisition GUI is launched
2. The user selects the port that the photodetector is connected to
3. The user sets the input light wavelength and the gain of the photodetector
4. The user begins sampling
5. The user stops sampling after an indefinite amount of time

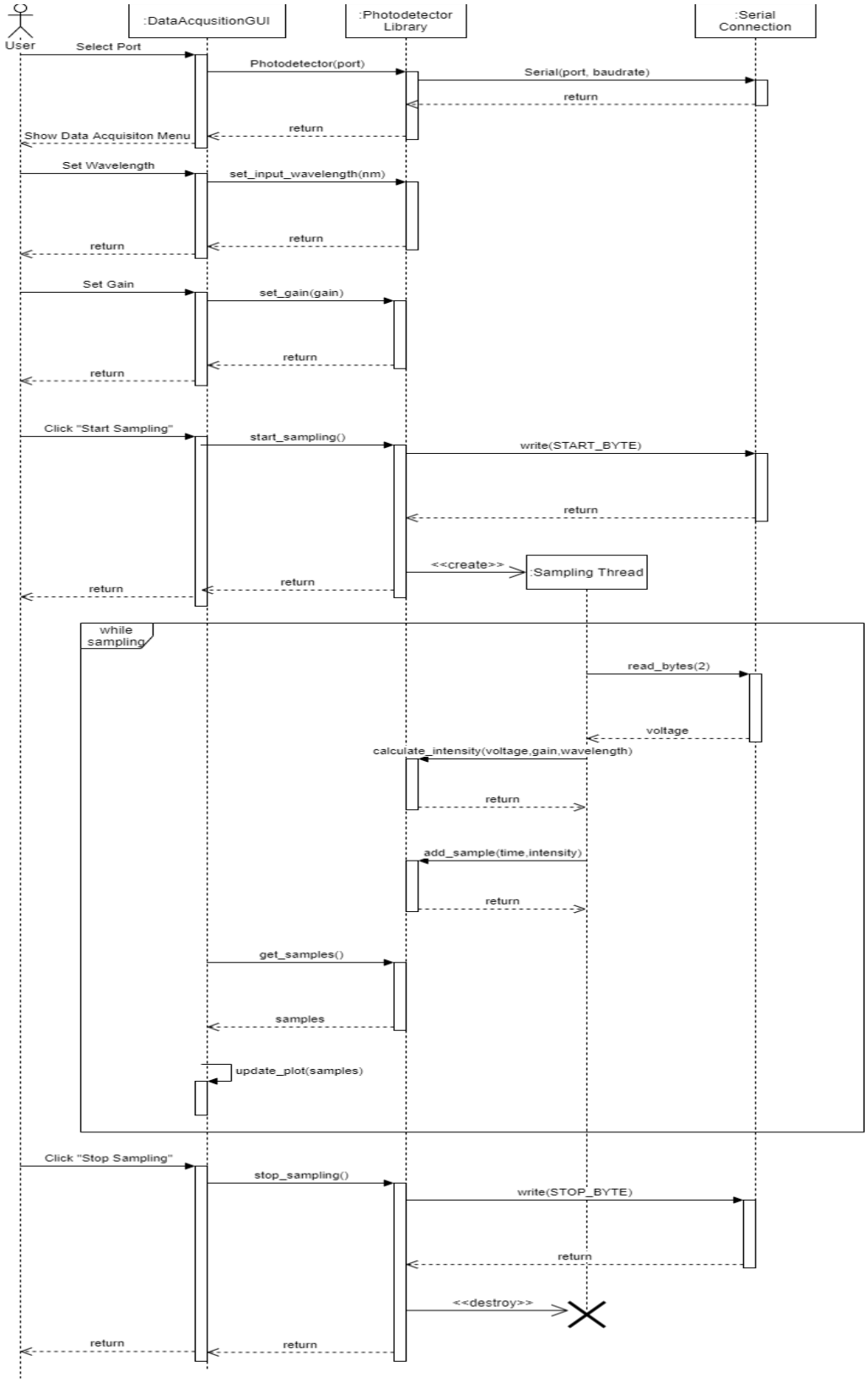


Figure 17: Sequence Diagram for typical asynchronous data sampling procedure

Note the frame that occurs between the command to start sampling and the command to stop sampling. This is a loop that occurs indefinitely while the start sampling command is issued before the stop sampling command is issued. When the start sampling command is issued, the Photodetector library creates a thread which does the following procedure:

1. Checks the serial connection to see if the device has written a 16-bit voltage value
2. If there is a 16-bit value available
    a. Read the available 2 bytes
    b. Call the method in the photodetector library to calculate intensity from the voltage value and the preconfigured wavelength and gain values
    c. Assign the calculated intensity value a time value

Thus the following core set of behavior was determined to be needed for the library, as distinguished from the data acquisition app:

- Connect to a photodetector device over serial
- Verify that a serial device is a photodetector
- Start, stop, and resume sampling both asynchronously and synchronously
- Convert digital data read from the device into physical measurements which are understandable by humans and provide access to these measurements
- Reset the current set of measured samples when the client wishes to start a new sampling session

6.3.3 Implementing the youngblood_photodetector Library

6.3.3.1 Disambiguating Units With the Pint Library

The issue of storing units in a disambiguous way that allows for scientific calculations and simple unit conversions was addressed using the pint Python library [13]. Pint is a Python library which allows for the use of units in Python code in a mathematical way. The library is used by creating a Unit Registry object, deriving Unit objects from this object, and dividing or multiplying them with numeric quantities or other Units to create dimensioned values. Pint's website provides the following example:

```
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> 3 * ureg.meter + 4 * ureg.cm
<Quantity(3.04, 'meter')>
```

The youngblood_photodetector library uses pint in every case where it uses a dimensioned value; this includes time, intensity, wavelengths, responsivity, gain, and voltage values. What should be noted is that Unit values are not global, and Quantity values computed with Units from two different UnitRegistry objects are never compatible. Thus, to ensure external applications can interface with the samples provided by the youngblood_photodetector library, the youngblood_photodetector library provides a UNIT_REGISTRY attribute which contains the UnitRegistry object used for the entire library. This allows any library to naturally interface with the dimensioned quantities provided by the library and take advantage of unit conversion facilities which are built directly into the pint library.

6.3.3.1 Verifying Serial Devices as Photodetectors

The youngblood_photodetector library contains a device_is_photodetector(port) method which takes a string representation of a serial port and tests to see if the device connected to that port is a

photodetector device. This is done by agreeing with the photodetector's embedded software on two values: a byte sent by the library to the device representing a verify device command (for our prototype, the UTF-8 encoding of 'v') and a string sent from the device to the library representing the device ID (for our prototype, the string "Youngblood Photodetector"). The device is programmed to send this ID back when it receives the verify command byte. Thus, the verification process works as follows:

1. The library sends the device command byte over serial
2. The device attempts to read back the device ID
3. If the read times out or a string is read back which does not match "Youngblood Photodetector", the device is not a photodetector. Otherwise, it is a photodetector
   This method allows for applications to selectively filter serial ports to include only those that are actually connected to a photodetector, decreasing the chance of an erroneous connection.

6.3.3.2 Creating a Photodetector Wrapper Class

Because the Python library does not actually control the photodetector device and is limited to serial communication, a Photodetector class was created for the youngblood_photodetetector class which essentially wraps a pyserial Serial object that is used to communicate with the device, provides methods for interfacing with the device without directly calling the Serial object, and automatically converts digital samples to real intensity values and provides them to client code. The Photodetector class has one required parameter: the serial port that the device is connected to. It can additionally take the baud rate of the photodetector device (set to 9600 by default), a csv file containing device responsivity measurements (set by default to a csv file included with the youngblood_photodetector package), a csv file containing device gain configuration measurements (set by default to a csv file included in the package), an initial gain setting (set by default to 1), and an initial input wavelength (not defined by default but required to be set before sampling can occur).

6.3.3.3 Configuring Device Measurements

6.3.3.3.1 Responsivity

The photodetector's responsivity curve is initialized for the Photodetector class instance using a CSV file containing responsivity measurements that is passed as a parameter. In this CSV file, the first column represents wavelengths (in nm) and the second column represents responsivity (in A/W) that was measured for the device. This responsivity curve is stored in a class in youngblood_photodetector called Responsivity; it takes a responsivity measurement CSV and stores them as a list of tuples where the first element of the tuple is the wavelength (stored in nm by default) and the second element is the responsivity (stored in A/W by default). In addition to the measurements, the Reponsivity class has an operating_range attribute which is a tuple containing the minimum wavelength in the measurements and the maximum wavelength in the measurements, constituting the effective operating range of the device.

Additionally, there is a Responsivity method at(wavelength) which, given a wavelength in the operating range, returns its corresponding responsivity. If there is a direct match in the device measurements, that measurement is returned; otherwise, it is linearly interpolated according to the following procedure:

1. Select from the list of measured responsivities tuples $(\lambda_{low}, R_{\lambda,low})$and $(\lambda_{high}, R_{\lambda,high})$, where $\lambda_{low}$is the discrete wavelength closest to λ that is less than λ, and $\lambda_{high}$is the discrete wavelength closest to λthat is greater than λ.
2. Linearly approximate $R_\lambda$by using the following equation:

$$R_\lambda(\lambda) = R_{\lambda,low} + \frac{R_{\lambda,high} - R_{\lambda,low}}{\lambda_{high} - \lambda_{low}}(\lambda - \lambda_{low})$$

When the Photodetector class is initialized, it takes a responsivity CSV file as a parameter. It uses this file to create a member variable of the Responsivity class which is used in internal calculations. Keeping the responsivity curve as a member variable of the Photodetector class allows for the possibility of supporting many photodetectors with different responsivity curves without binding the library to a specific set of measurements.

6.3.3.3.2 Gain

Just as the device responsivity measurements are represented by a Responsivity class which takes a responsivity measurements CSV as a parameter, gain configuration information for the device is stored in a GainConfig class. This GainConfig class takes a gain configuration CSV as a parameter. This CSV has three colums: gain setting as listed on the device, the actual gain in V/A, and the maximum device voltage obtainable for that gain setting. The GainConfig class reads these values from the CSV file and provides get_real_gain(gain_setting) and get_max_voltage(gain_setting) methods to access them trivially. Like the responsivity measurements, the Photodetector class can also take a gain config CSV file as a parameter, and it creates a gain_config member variable which can be accessed by client code to get gain configuration info for the device.

6.3.3.4 Setting Device Input Parameters

The photodetector has three parameters which affect the quantity and values of the sample data it collects: the sampling rate, the input wavelength, and the gain configuration.

The sampling rate is set using the Photodetector's set_sampling_rate(frequency) method, where frequency is a pint Quantity class with units that are some order of 1/second. In order to send this sampling rate to the device, the Photodetector library sends a byte indicating a command to change frequency (for this prototype, a UTF-8 encoding of 'f'). When the device receives this, it then waits until the Photodetector instance sends over an unsigned 32-bit sampling rate value as a python bytes() object.

The input wavelength and the gain configuration are not set on the device by software; rather, these values are set physically on the device. However, there needs to be some way for the Photodetector class and the actual device to agree on gain and input wavelength, so the Photodetector class provides set_gain_setting(gain_setting) and set_input_wavelength(wavelength) methods respectively for these values to be set on the library end. If an invalid gain setting or wavelength is passed to their respective methods, a ValueError is raised by the library.

6.3.3.5 Collecting Digital Samples Over Serial

The host computer sends two commands to the photodetector device over serial related to sampling: start sampling, and stop sampling. The Photodetector class thus has a start_sampling() and stop_sampling() method to send these commands respectively. These commands are sent by writing a corresponding byte of information to the photodetector over serial.

In the period between a start sampling command being sent and a stop sampling command being sent, the photodetector device will write a series of 16-bit values at a rate of *n* values per second specified by the user. Because we cannot know how long the user will wish to sample for and client applications may wish to perform other tasks while sampling occurs, start_sampling() supports the option to start a thread to asynchronously poll the serial port for samples by passing a parameter "async" set to True. After a voltage value is read in using a call to Serial.read(2), a corresponding intensity value is calculated using the procedure in 6.3.3.6 and a tuple containing the sample time and the sample intensity is added to the device's current sample data. The first sample in the sampling session is assumed to happen at time 0, so the sampling time for the *i*th sample, where i = 1,2,3,..., is calculated using the equation

$$t_i = \frac{i-1}{n}$$

When the stop_sampling() command is invoked, the library terminates the thread that is polling for voltage values (if sampling was done asynchronously) and send the command to the photodetector to stop sampling. The calculated samples can then be accessed by client code using Photodetector's current_samples member variable. At any time during asynchronous sampling, client code can check whether or not the device is in sampling mode by invoking the is_sampling() method (although this does not actually check the device itself; since the Photodetector instance is the only code directly communicating over Serial, it can keep track of whether or not the device is sampling by simply setting a bool variable True or False)

In addition to supporting asynchronous sampling using the start_sampling() and stop_sampling() methods, the Photodetector class support synchronous sampling using the sample_n_times(n_samples) method. This method works by invoking the start_sampling() method with the async parameter set to False and iterating in a loop for i in range(n_samples) which reads one sample from the device on each synchronous iteration. It then returns the last n_samples samples in the list, representing a synchronously-collected list of n samples. This method is used to support precision sampling on the device.

6.3.3.6 Converting Digital Samples to Intensity Samples

Intensity will be calculated by the library using three parameters: output voltage from the photodetector (transmitted as a 16-bit integer value which corresponds to a voltage of 0V - 3.3V), wavelength of the light input (in nm), and gain (in V/A).

Given a 16-bit digital voltage value *n*, the library linearly interpolates its real voltage value in volts using the following equation:

$$V_{out}(n) = \frac{3.3V}{2^{16}} n$$

Digital-to-analog conversion in the youngblood_photodetector library is done using a class called VoltageDAC. The VoltageDAC object is initialized with a minimum voltage (in volts), a maximum voltage (in

The real gain G for the device at the current gain setting is obtained using the GainConfig's get_real_gain(gain_setting) method as described in 6.3.3.3.2, and the Responsivity object's at(wavelength) method is used to obtain the responsivity R_λ

After calculating $V_{out}$and R_λ, the measured intensity is calculated using the following equation:

$$P_{in}\left(V_{out}, G, R_{\lambda}\right) = \frac{V_{out}}{G R_{\lambda}} * \frac{3.3}{9}$$

where $P_{in}$is in units of Watts. This equation is implemented in the Photodetector class in its calculate_intensity(voltage, gain_setting, input_wavelength) method.

Sample data for the current sampling session is stored in the Photodetector library as a member variable called current_samples. Sample data is stored in a class called SampleData, which provides the following methods: add_sample(time, intensity), which appends a sample to the current set of samples; clear_samples(), which clears all samples from the sample data; and asdict(), which returns the sample data as a dictionary of the following format: {'time': [ordered list of all times in samples], 'intensity': [ordered list of all intensities in samples]}. This class was created to provide an easier way of serializing sample data as JSON. Furthermore, the SampleData class can be iterated using a for loop or list comprehension and supports indexing subranges with the bracket operators, which will produce a new SampleData object with the expressed subrange of samples.

6.3.4 Final Package and Class Structure

The final version of the classes and the youngblood_photodetector package, as described in the previous sections, can be found in a class diagram in Figure 18. All publicly-exposed classes, fields, and functions in youngblood_photodetector are shown, as well as class methods and fields. Additionally, third-party library dependencies are charted with the <<uses>> relationship.

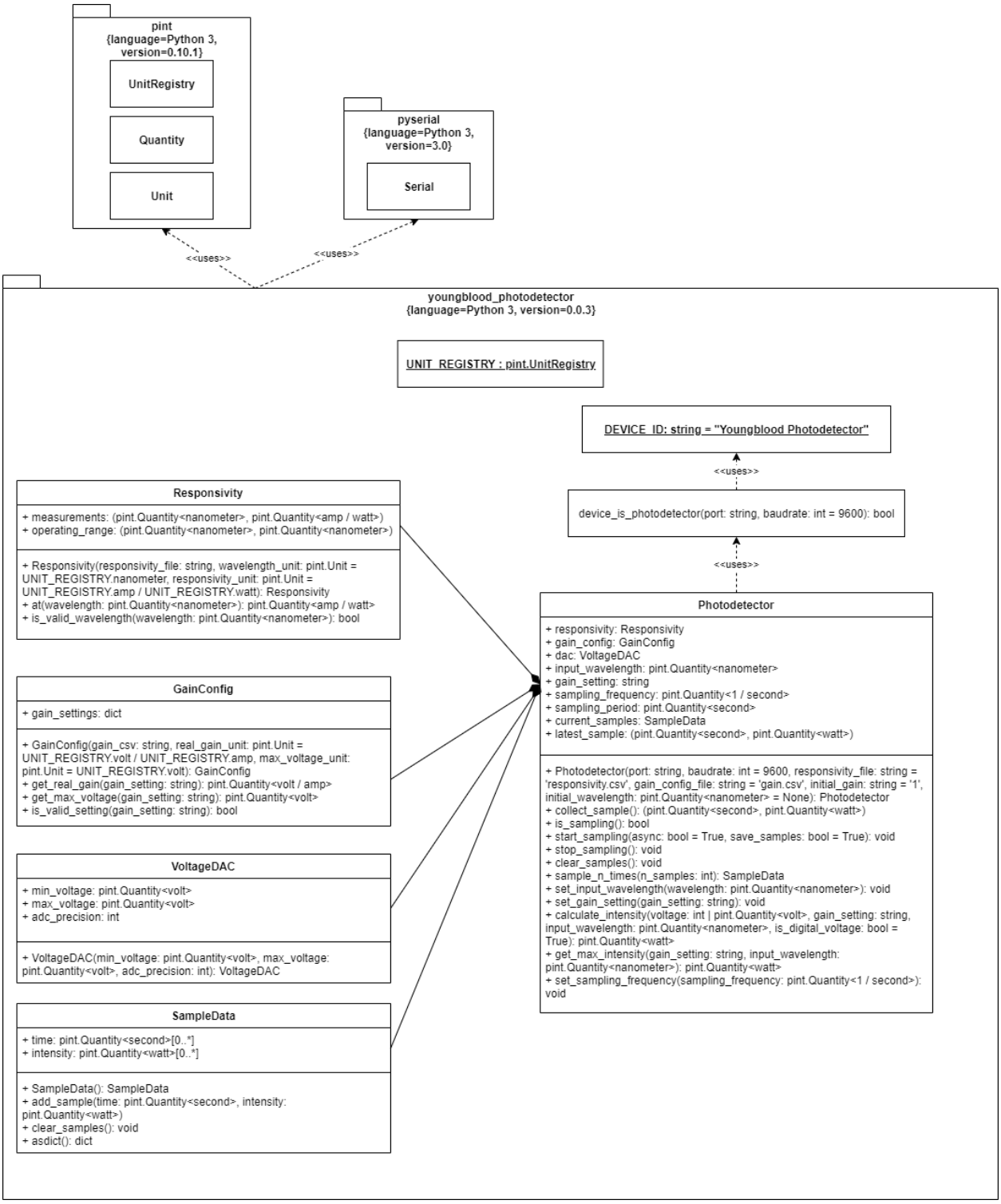


Figure 18: Class Diagram for youngblood_photodetector library.

6.3.5 Deploying the youngblood_photodetector Library

After the library was finished being developed, it was packaged as a Python wheel library. This resulted in a .whl file which contained the source code for youngblood_photodetector, the default

responsivity and gain configuration CSV files, and instructions to install third party dependency libraries (i.e. pyserial and pint). Packaging youngblood_photodetector as a wheel archive provides the following benefits. First, the entire package can be installed using pip as long as the person has access to the .whl file, which is easily hostable on package services such as PyPI. Installing the package through pip treats the youngblood_photodetector library as a properly-installed package, meaning youngblood_photodetector and its contents can be included from a program anywhere on the machine that used pip to install it, meaning that the youngblood_photodetector package does not need to be copied into the directory of each application that uses it.

6.4 Data Acquisition GUI

6.4.1 High Level Description

As mentioned in 6.3, the Data Acquisition GUI is the second software subsystem developed for the lab station. It constitutes the actual "product" which was requested by Dr. Youngblood in that it provides a graphic user interface for controlling sampling on the device, visualizing the samples collected from the device, and exporting this data for further use. The code for the Data Acquisition GUI is bundled in a Python module called data_acquisition. The user interface is constructed using the Dash application framework and its corresponding component libraries. On the back end of the application is an instance of the youngblood_photodetector library's Photodetector class. When a user selects an option or sets a configuration value in the GUI, the GUI's code invokes the proper operations on the device through the Photodetector instance. Likewise, when the Photodetector instance returns values such as sample data, the GUI is programmed to update its display to reflect this new information. In this way, the software stack on the lab station constitutes a form of Model-View-Controller architecture: youngblood_photodetector provides the model, while data_acquisition provides the view and the controller. Additionally, the Data Acquisition GUI provides functionality related to the manipulation of sample data which is outside the scope of youngblood_photodetector: namely, locating samples with max intensity and exporting samples to CSV.

6.4.2 Application Wireframe

Figure 20 shows the wireframe for the preliminary data acquisition application. The window on the left represents the application after finishing a sampling session. Red arrows signify windows that open after their corresponding buttons are pressed.

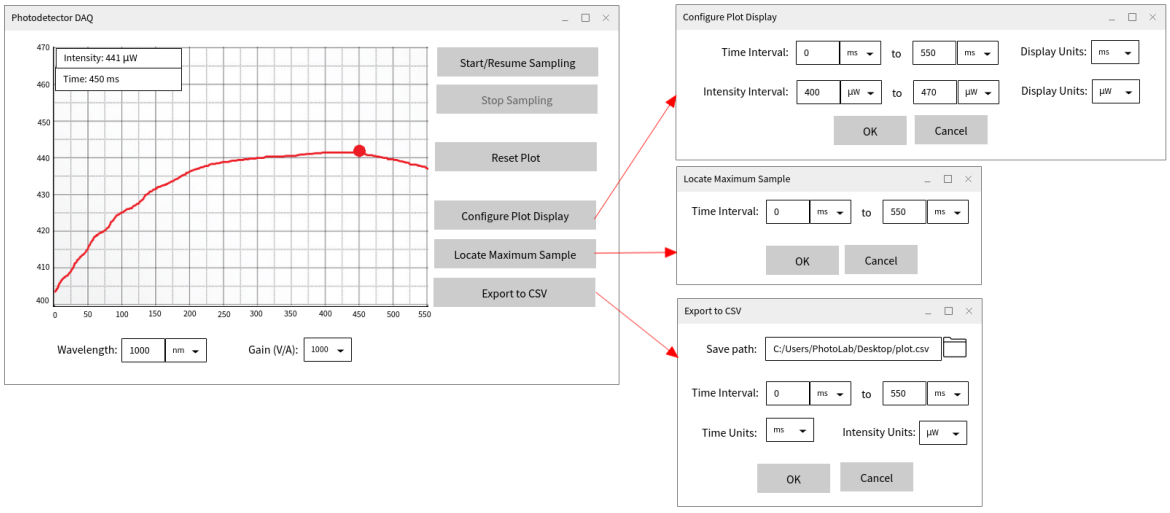


Figure 19: Preliminary Wireframe Application

The user is able to specify the wavelength using a Dash Input component [9] and select its units using a Dash Dropdown component [10]. Whenever the value in the Input box or the Dropdown value changes, the GUI calculates the wavelength in nm using pint and invokes the Photodetector object's set_input_wavelength() method. In a similar manner, the user is able to specify the gain using a Dash Dropdown component and the Photodetector's set_gain() method is called whenever the Dropdown value changes.

The graph is a Dash Graph component [11]. The graph is updated by a Dash callback which is invoked when sampling is active and each 0.1 seconds passes This graph-drawing method was planned to be invoked every 1/30th second while sampling is active, but testing on the hardware that was available only managed a refresh rate of 10 samples/s. When sampling is not occurring, the user is able to click on a point and see the value of its corresponding sample by hovering their mouse over the graph; the Dash Graph has built-in support for this as well as resizing the graph and zooming into certain points.

Clicking the "Start/Resume Sampling" button calls the Photodetector's start_sampling() method, and the button is disabled while sampling is active so that the start sampling command is not issued twice. While sampling is active, all buttons except "Stop Sampling" should be disabled.

Clicking the "Stop Sampling" button calls the Photodetector's stop_sampling() method, and the button is disabled while sampling is inactive so that the command is not issued twice.

Clicking the "Reset Plot" button deletes all of the sample data currently being stored by the Photodetector instance by invoking Photodetector.reset_samples() and clears all points on the graph.

The GUI code is capable of converting the Photodetector's representation of sample data to the user-requested units without modifying the data in the Photodetector object by using pint's data conversion facilities. This is wrapped in a Dash callback that updates the data sent to the Dash Graph every time a different unit is selected from the unit dropdowns.

Clicking the "Locate Maximum Sample" button would open a menu which allows the user to set an interval for which they want to find its maximum intensity value. The menu's default interval should be the entire sampling period; the user is given the option to specify a sub-interval in the event that they want to find a local maximum rather than an absolute maximum. After "OK" is clicked in the menu, the

Graph should be redrawn such that the portion of the graph containing the located maximum is displayed and the maximum sample is selected such as it had been clicked.

Clicking the "Export to CSV" button would open a menu which allows the user to set a time interval for the samples they wish to export. By default, this interval should encompass all of the collected samples. Clicking the file folder icon on the "Save path" line should open up a file select menu which allows the user to navigate to a save location and specify a filename, after which the "Save path" field should be populated with the file path generated from the navigated path and the filename. After "OK" is clicked, the method data_acquisition.daq_utils. should be called which creates the CSV file and writes the samples within the interval to this file in the units specified by the user.

Due to time constraints involved with creating a working prototype for the project, these sub-windows are not opened as separate windows; rather, they are fully visible at all times when the sampling plot is visible.

6.4.3 Implementing Auxiliary Utilities of GUI

In addition to the Dash app and component code, there are auxiliary methods in the data_acquisition library which are not tied explicitly to a Dash callback but are nonetheless used in the data acquisition process. They are stored in the subpackage data_acquisition.daq_utils.

6.4.3.1 Locate Local Maximum Sample

The process of locating the maximum intensity sample over a certain interval is accomplished using the data_acquisition.daq_utils.get_local_maximum() method. This method takes the following parameters:

- sample_data : a SampleData object containing all of the samples on the device
- start_time : a pint Quantity of the order of seconds which represents the start of the interval of samples in which to locate max intensity sample
- end_time : a pint Quantity of the order of seconds which represents the end of the interval of samples in which to locate max intensity sample

  Locating the maximum intensity sample employs a trivial linear-time max algorithm: the sample at the beginning of the interval is initially determined as the max sample and then the samples which are between start_time and end_time are iterated over, resetting the maximum any time a sample with a larger intensity than the current maximum is found. This sample is then returned as a tuple of the form (time, intensity).

6.4.3.2 Export to CSV

The process of exporting samples over a certain interval to CSV is accomplished using the data_acquisiton.daq_utils.export_to_csv() method. This method takes the following parameters:

- filepath : a string containing the save path of the CSV file to export
- sample_data : a SampleData object containing all samples on the device at the time of the export
- start_time : a pint Quantity of the order of seconds which represents the start of the interval of samples to export to CSV
- end_time : a pint Quantity of the order of seconds which represents the end of the interval of samples to export to CSV

- time_unit : a pint Unit which represents what seconds-based unit to export time data as
- intensity_unit : a pint Unit which represents what watt-based unit to export intensity data as

This function simply opens a new CSV file at the location given by filepath and iterates through the samples in sample_data. For each sample in sample_data whose time is greater than or equal to start_time and less than or equal to end_time, the time of the sample is converted using pint's conversion functionality to the unit specified by time unit, the intensity of the sample is converted using pint's conversion functionality to the unit specified by intensity_unit, and the samples are added as a row to the CSV. The final CSV takes on the following format:

Sample Time (unit), Intensity (unit)
time 1,intensity 1
time 2,intensity 2
....
time n,intensity n

6.4.4 Implementing the GUI components

The GUI was created using Dash. Dash apps consist of three major types of components: the Dash app instance (of type dash.Dash()), components, and callbacks. The Dash app instance is the first entity to be defined in a Dash app, and it invokes the framework that is used to build the app. The Dash app object has a configurable layout field which contains the layout of the app. The layout of the app is composed of Dash components; these components are specified in Python and support both Dash's library of HTML component wrappers (dash_html_components) and Dash's library of their own custom components (dash_core_components). Each Dash component has two basic properties, id (the device's configurable, unique HTML ID) and children (all Dash components which are HTML DOM children of the component in question), in addition to other properties that can be set based on the value of the component. Thus, each component in this app, which can be seen in Figure 20, can be represented as a hierarchy of Dash components. Finally, there are callbacks. These are functions which can be configured to execute any time certain properties of elements with certain IDs change (for instance, a Dash callback can be triggered when an Input field has its values changed). These functions can then return an indefinite amount of elements, which can be used to change the properties of elements in the Dash app. Thus, the logic for the data acquisition app is implemented as a series of callbacks which are configured to modify the Dash components that make up the app.

The data_acquisition package has the following file Dash app file hierarchy. In the root of the data_acquisition package are two files, app.py and index.py. The app.py file contains the two elements which are shared amongst all Dash components: the Dash app instance for the data acquisition application (called "app") and the variable used to hold the youngblood_photodetector.Photodetector instance wrapping the serial device. The file index.py contains the main layout of the app and is used as the basis for launching the application.

In addition to these two core files, additional components are contained within the data_acquisition.components subpackage. These components are used by index.py to specify the layout of the application in a modular form. Each component submodule has a layout variable which is used by index.py in its own layout, and each submodule defines a set of Dash callbacks to operate on the

components contained within this module. This allows for the components to be semi-independent of each other and provides modularity by not requiring all components and callbacks to be in the same file.

6.4.5 Class and Package Diagram for Data Acquisition GUI

The class diagram for the data_acquisiton package is shown in Figure 20. Note that the data_acquisition package has a dependency on youngblood_photodetector but youngblood_photodetector does not have a dependency on data_acquisition; this represents a successful decoupling of the serial interfacing functionality of the photodetector and the data acquisition requirements established by Dr. Youngblood, promoting reuse of the youngblood_photodetector library for other uses. Note that the GUI component hierarchy represented in 6.4.4 is present in the composition relationships between the components in data_acquisition and data_acquisition.components.

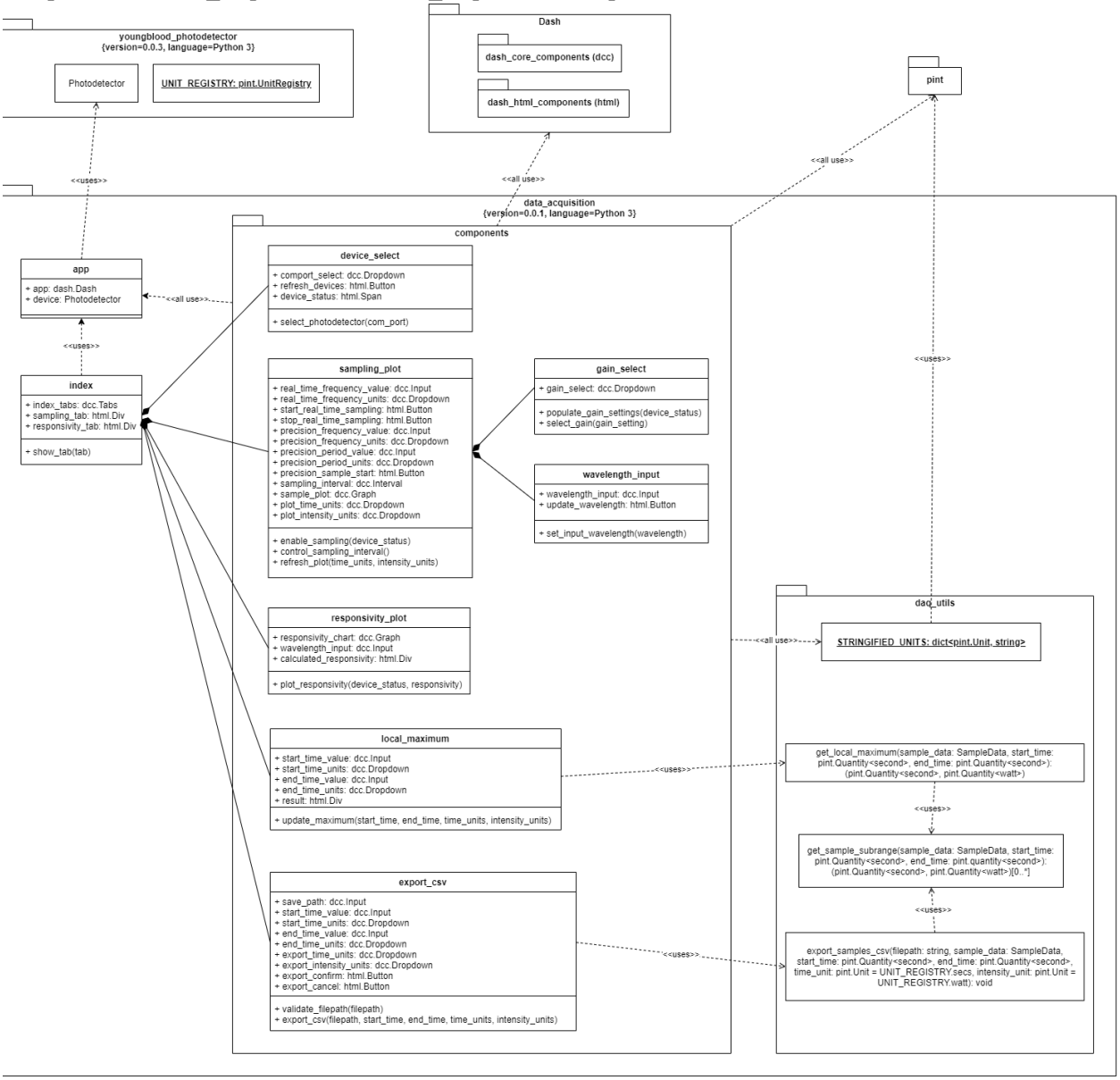


Figure 20: Class Diagram for the data_acquisition package

6.5 Data Mapping

Data mapping is the process of data fields from one database to another. With any precision conversion system, it is crucial to determine the capabilities of your system, and data mapping the PCB's analog output helps the team to determine the linear regions of operation for each gain setting. The SPICE simulations gave the hardware team the data necessary to determine the maximum inputs before the op-amps saturated or saw overcurrent voltage attenuation.

Figure 21 below shows the linear output voltage range over the input current. The variable input current here represents varying optical intensity (higher intensity corresponds to higher current). The corner corresponds to the maximum voltage output and the maximum current that would produce this voltage. By dividing the current by average responsivity, one can estimate the maximum power input that would cause saturation of the op-amps and the maximum 3.3V output.

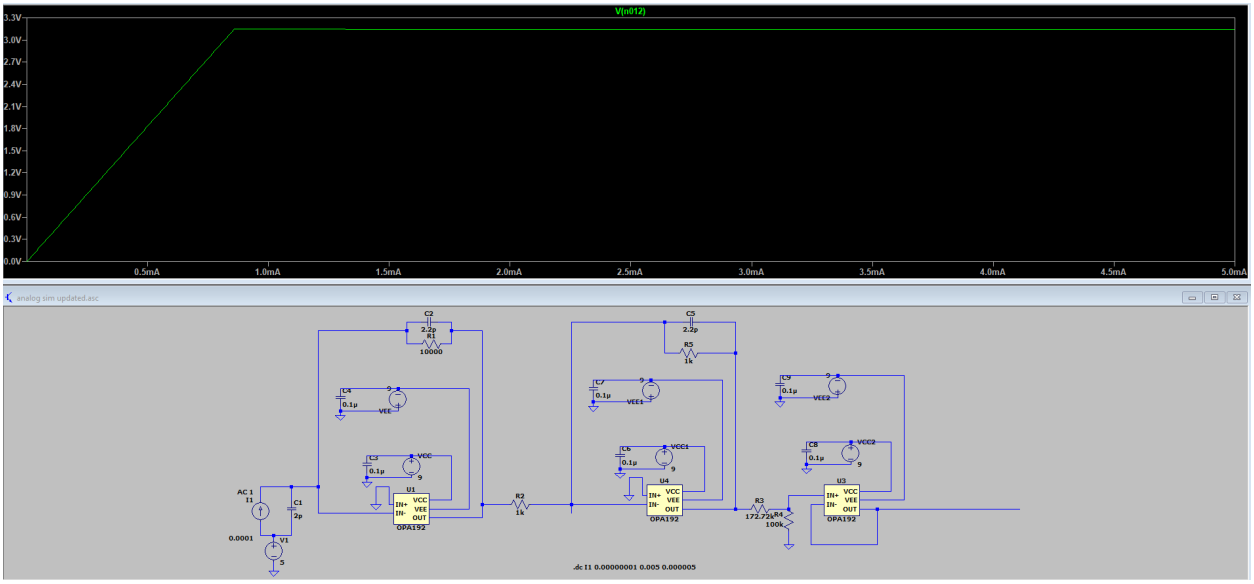


Figure 21: Output analog Voltage vs Input Current Simulation at gain of 10,000

Simulations for lower gain settings showed that a higher current is needed to cause op-amp saturation. Some higher currents cause problems after saturation because the overcurrent produced on the output of the op-amps reduces the voltage on the output, and causes non-linearity. Figure [x2] below shows one such simulation of too much current.

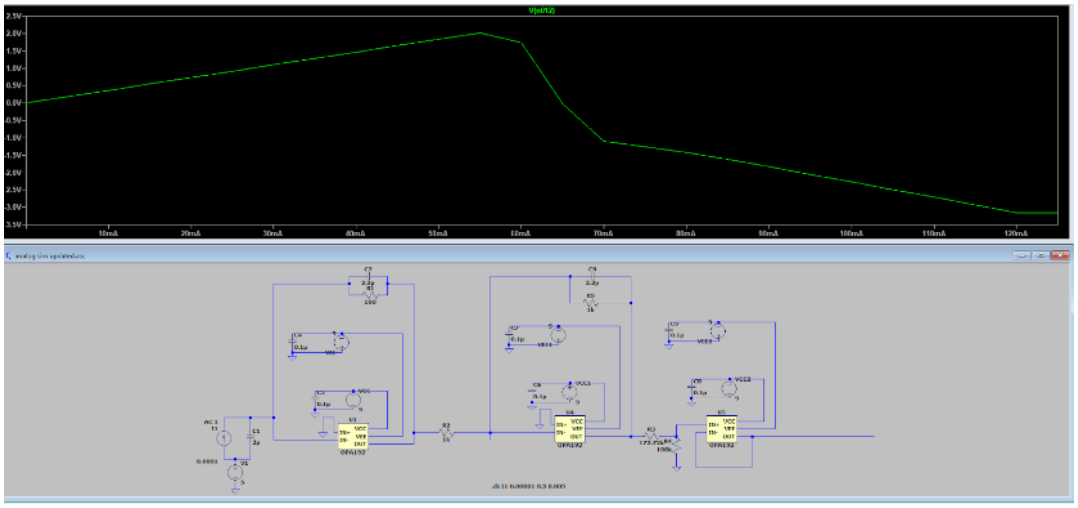


Figure 22: Output Analog Graph Showing Overcurrent

This simulation also shows another flaw of the analog design. The analog amplification operates in the linear region from 0mA to about 55mA in the simulation above. The photodiode can only take a maximum optical input of 10 mW, which corresponds to a maximum current produced of about 8.8 mA. This value was calculated by dividing the maximum optical input power (from the datasheet) by the maximum responsivity on the datasheet. A worst-case scenario of 8.8 mA means that some optical amplification settings have an extremely small operating range. Table 3 below shows the Maximum simulated output voltages, input currents, and power resolutions of each gain setting.

Gain Setting	Max Voltage (V)	Max Input Current	Max Input Power (W)
3 x 10,000	3.245 V	295 uA	335. 227 uW
1 x 10,000	3.15 V	860 uA	977.72 uW
3 x 1,000	3.245 V	2.95 mA	3.3523 mW
1 x 1,000	3.0355 V	8.3 mA	9.43 mW
3 x 100	0.968 V	8.8 mA	10 mW
1 x 100	0.3227 V	8.8 mA	10 mW
3 x 10	0.0968 V	8.8 mA	10 mW
1 x 10	0.03227 V	8.8 mA	10 mW

Table 3: Analog Minimum Input and Output Voltages and Currents

This table shows that the gain settings past 1 x 1000 are unable to realize a linear region as large at the higher gain settings. In order to increase this range, a photodiode with a larger optical input range would be required. This would likely decrease the sensitivity of the photodiode and increase the dark current if we the team is planning to spend about the same money on a different photodiode model.

This table also shows that the OPA192 is not truly rail-to-rail when being used. The theoretical output voltage struggles to truly reach 3.3V. Another op-amp could be considered here that can source more current and has better truly rail-to-rail performance.

6.6 Physical Realization

Figure 23 and 24 below shows the finished PCB Prototype. The board was designed to be 5.5x5.5 inches, with mounting holes set 5x5 inches apart. The mounting holes were made compatible with ¼ x 20 in screws that could screw into the honeycomb optical table in Dr. Youngblood's lab.

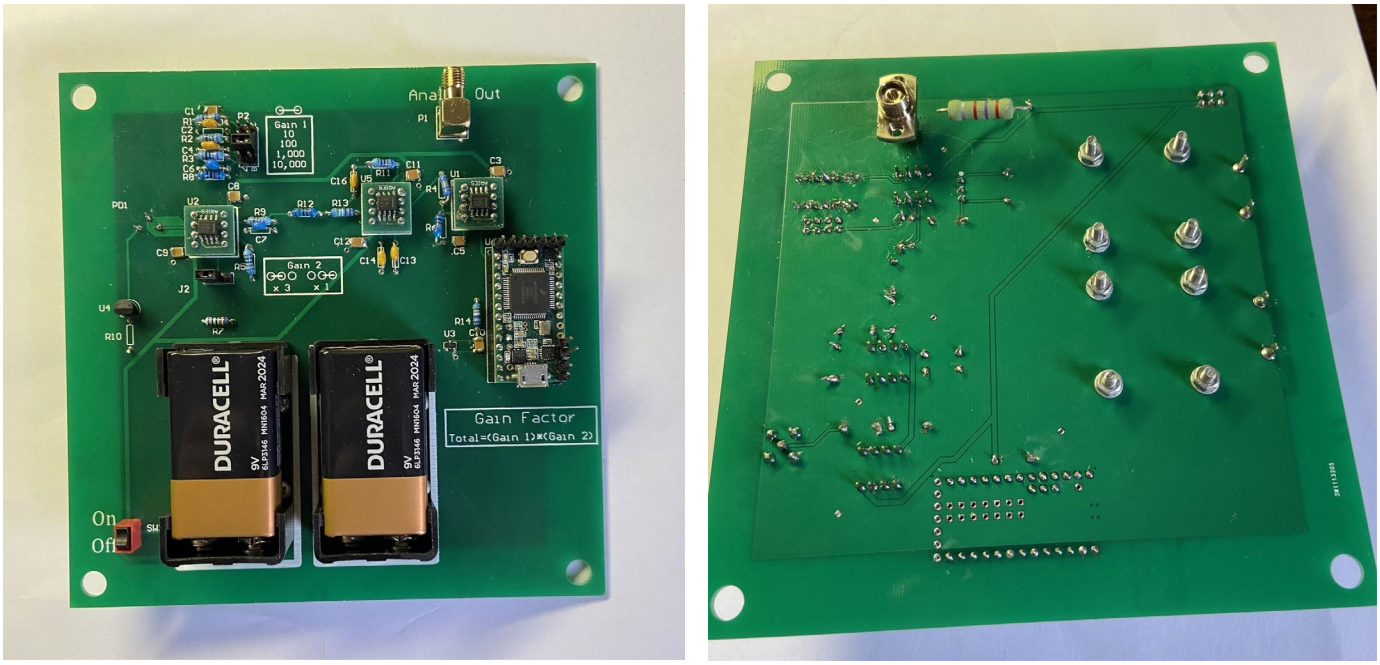


Figure 23 & 24: Final PCB Prototype, Top View (Left) and Bottom View (Right)

The Photodiode input was placed on the bottom so optical fibers resting on the optical table can screw in from the bottom. These fiber optical cables are extremely sensitive and will be damaged if they get folded or kinked- this can stop optical transmission. Screwing the cable in from the bottom will reduce the amount of turns and twists the cable would have to make. A power switch can be found to the bottom left of the batteries, which connects/opens the +9 and -9 V power buses. An analog SMA output is found in the top right corner. The variable gain jumpers are found next to their corresponding connection settings. The Teensy 3.2 is also installed in the board, and can be pulled out of the socket for necessary reprogramming.

7. Testing and Verification

7.1 Software Systems

7.1.1 Photodetector Library

The youngblood_photodetector library was periodically manually tested by running a test Python script which instantiated an instance of Photodetector class and performed various device configuration and sampling operations. In addition to the Python script, these tests used a Teensy 3.2 programmed with Teensyduino to serve as a mock device.

7.1.1.1 Mock Device

In lieu of being able to test the library with an actual prototype and lab equipment, the functionality of the library and GUI were tested using a Teensy 3.2 programmed with an Arduino script to serve as a mock device. The Teensy was chosen as the hardware to run the mock program because it was the same microcontroller that was planned to be used on the prototype, allowing the library to be developed under similar hardware constraints as the planned prototype. The mock device operated at 9600 baud, communicated over serial, and ran a program functionally equivalent to the following pseudocode:

```
setup() {
      Serial.begin(9600);
      uint16_t sampleData = 0x0000;
      bool sampling = false;
      uint32_t samplingFrequency = 400;
}
loop() {
      if (sampling) {
            Serial.write(sampleData);
            Serial.send_now();
            sampleData++;
      }
      if (command sent over serial) {
            if (command is VERIFY) {
                  Serial.println("Youngblood Photodetector");
            } else if (command is START_SAMPLING) {
                  sampling = true;
            } else if (command is STOP_SAMPLING) {
                  sampling = false;
            } else if (command is SET_FREQUENCY) {
                  delay until 4 bytes on serial
```

```
                samplingFrequency = (uint32_t)Serial.readBytes(4)
            }
        }
        delayMicroseconds(1000000 / samplingFrequency);
}
```

Thus, the mock device waits for commands to start or stop sampling, and sends a linearly-increasing series of 16-bit values over serial when sampling is active, which makes it simple to verify data is being correctly sent . Serial.send_now() is invoked to prevent the default behavior of sending USB packets by Teensy which would make real-time sampling impossible. The mock device can receive a 32-bit unsigned integer value representing sampling frequency and adjust the rate it sends samples accordingly. Finally, a verify command can be sent which returns the device ID string expected by the youngblood_photodetector library to distinguish it from non-photodetector serial devices.

7.1.1.2 Photodetector Library Test Script

Much of the youngblood_photodetector library functionality is tested when testing the Data Acquisition GUI. However, for ease of development and ability to test the youngblood_photodetector library independently of the data_acquisition GUI, a test script was developed which exclusively tests features of the library in a general sampling use case. The script was run with a mock device as described above connected to the testing computer over serial. Pseudocode for the youngblood_photodetector test script can be found below:

```
import youngblood_photodetector
from youngblood_photodetector import UNIT_REGISTRY as ureg
from time import sleep

# test that serial device can be instantiated as the photodetector
# device_port is determined by examining Device Manager and manually
changing value
# to match the COM port of the mock device (e.g. 'COM4')
assert youngblood_photodetector.device_is_photodetector(device_port)
photodetector = youngblood_photodetector.Photodetector(device_port)

# test setting sampling frequency and asynchronous sampling (i.e. real-time
mode)
async_sampling_frequency = 10000 / ureg.secs
photodetector.set_sampling_frequency(async_sampling_frequency)
photodetector.start_sampling(async=True, save_samples=True)
sleep(1)
print(len(list(photodetector.current_samples))) # print number of samples
# at this point, manually verify that 1 second sampling at 10000 samples
per second
# produces a number of samples close to 10000 (not guaranteed to be exact
```

```
due to asynchronicity)

# test that clearing samples results in 0 samples being on device
photodetector.clear_samples()
assert len(list(photodetector.current_samples)) == 0

# test sampling 100000 times synchronously at 40000 samples/s (i.e.
precision mode)
# should produce a list of exactly 40000 samples from t=0s to t=2.499975s
n_synchronous_samples = 100000
synchronous_sampling_rate = 40000 / ureg.secs
photodetector.set_sampling_rate(synchronous_sampling_rate)
collected_samples = photodetector.sample_n_times(n_synchronous_samples)
# verify that exactly 100000 samples collected
assert len(list(collected_samples)) == 100000
# verify that time boundaries are as expected
assert list(collected_samples)[0][0] == 0 * ureg.s
assert list(collected_samples)[-1][0] == 2.499975 * ureg.s
```

The asynchronous sampling frequency of 10000 samples/s and synchronous sampling frequency of 40000 samples/s were chosen because these were established to be the minimum sampling benchmarks for real-time and precision sampling respectively for the final deliverable. They do not meet the initial set of requirements established for the device, which can be owed to an inability to test on the Photonics Lab lab station itself and instead being constrained to a less-powerful personal laptop to test with.

This script was executed every time a major change to the photodetector library was made to verify no major functionality was compromised. In a way it served as a smoke test before testing the GUI; identifying a problem in the library using this script indicated that the same problem would be present in the GUI without involving as much manual testing as testing the GUI. Thus this test script was always run before testing or demonstrating the GUI.

7.1.2 Functional Demonstration of Data Acquisition GUI

The following section shows the look of the Data Acquisition GUI implementation and its finished functionality as of the final deliverable demo.

7.1.2.1 Initial State

Figure 25 shows the initial state of the GUI upon launching the program. The program does not select a device by default and options to sample are disabled until a device is collected. By default, the sampling tab is shown and the graph is configured to display samples with time units of seconds and intensity units of nanowatts. Units of nanometers are enforced for wavelength but the user is allowed to select from microseconds to seconds for time-based units and picowatts to watts for intensity-based units.

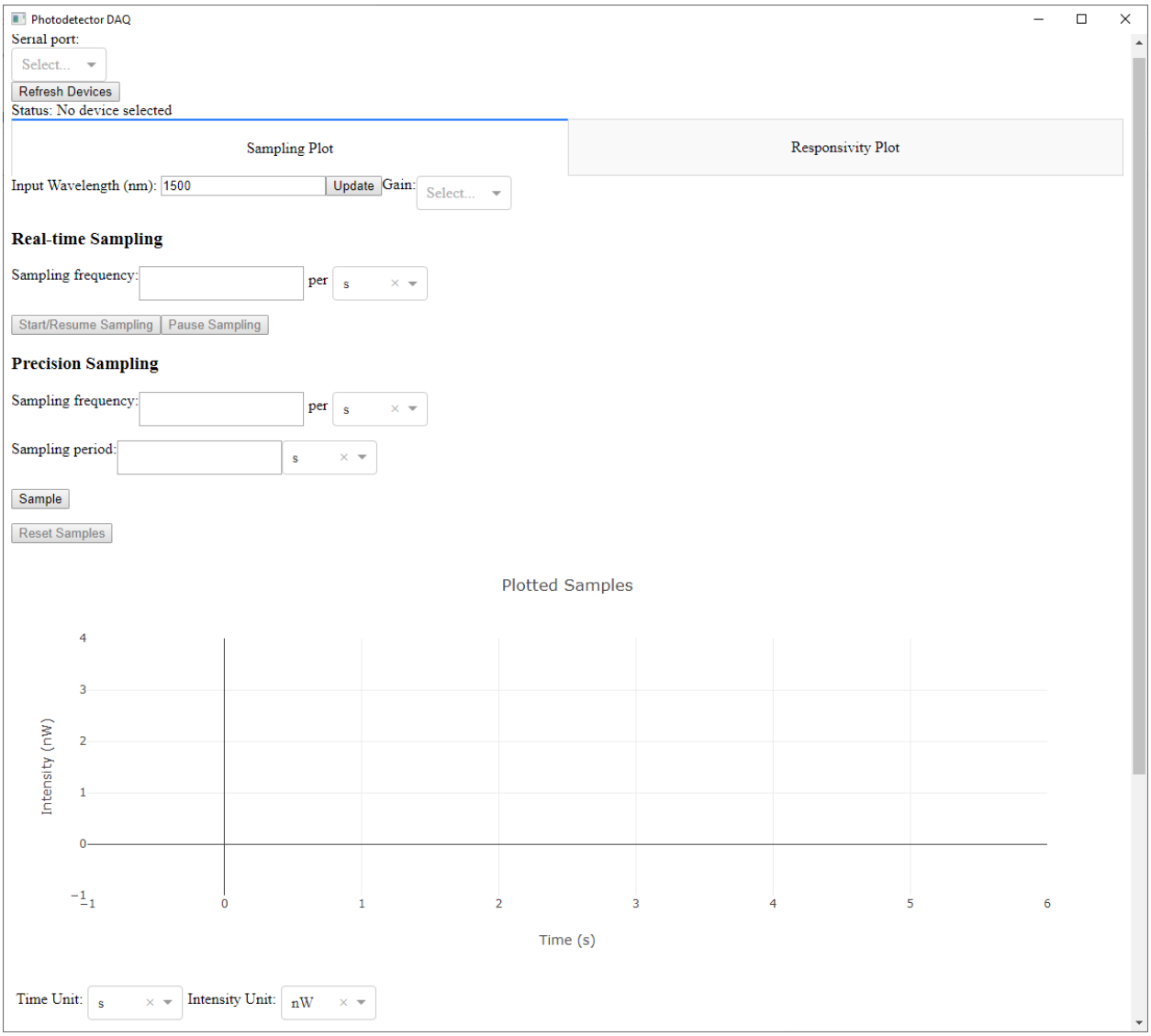


Figure 25: Data Acquisition GUI upon immediately launching the program.

7.1.2.2 Real-Time Sampling Mode

Figure 26 shows the GUI while real-time sampling is active. For real-time sampling, the GUI enables the user to set the COM port of the device, set an input wavelength and a gain setting, set a real-time sampling frequency, and sample asynchronously. Sampling begins when the "Start/Resume Sampling" button is pressed. While sampling is active, the last 100 measured samples are shown to the user, which we can verify is the case here due to the sampling rate of 400 samples/s and the ~0.25s interval currently being displayed. The sampling will continue indefinitely until the "Pause Sampling" button is pressed.

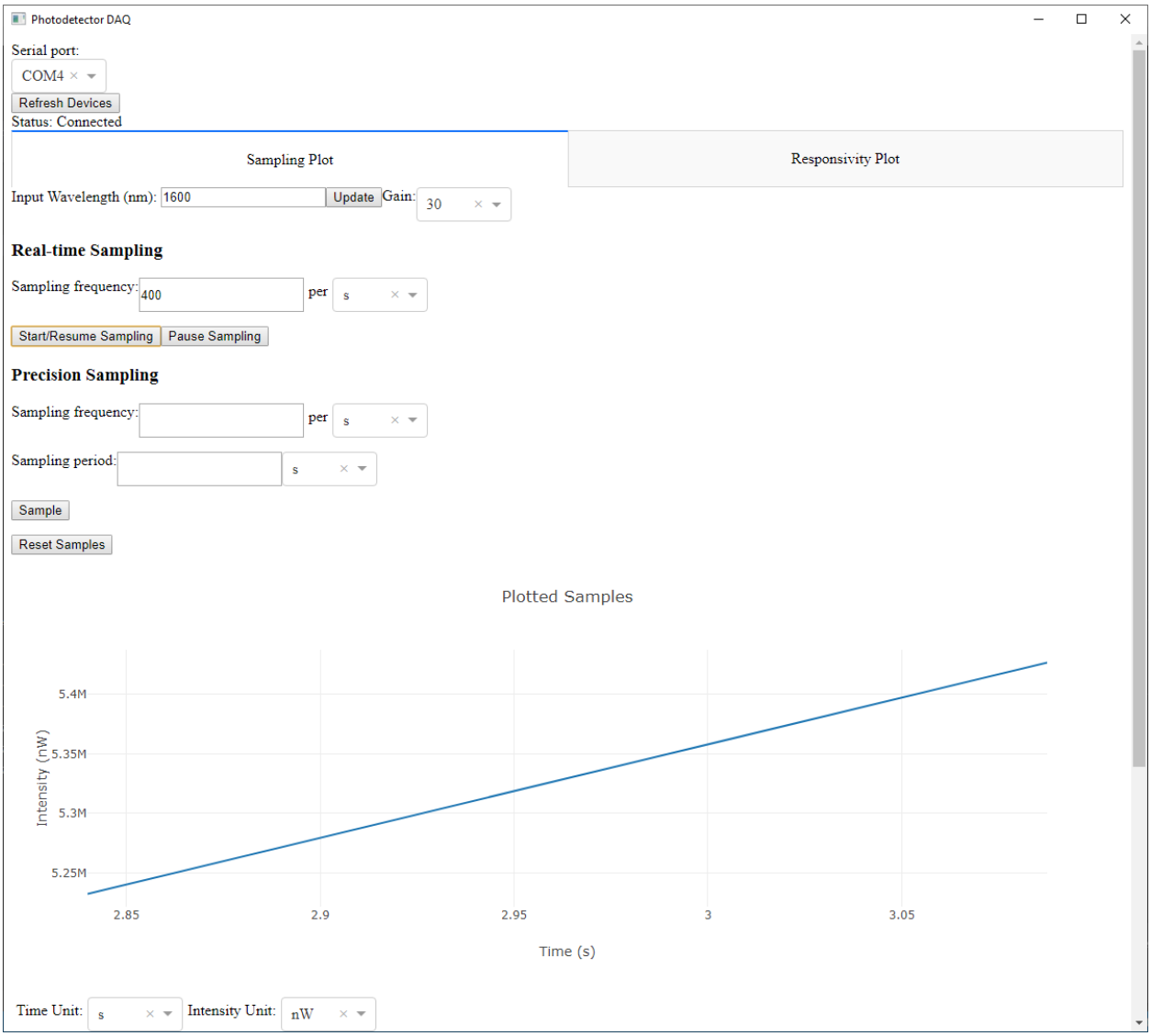


Figure 26: Demonstration of the Data Acquisition GUI's real-time sampling mode while sampling is active.

Figure 27 shows the GUI immediately after data sampling is paused. Note that the sample plot automatically adjusts to show the entire sampling interval. Additionally, one can hover the cursor over a point on the graph to see the time and intensity for that particular sample.

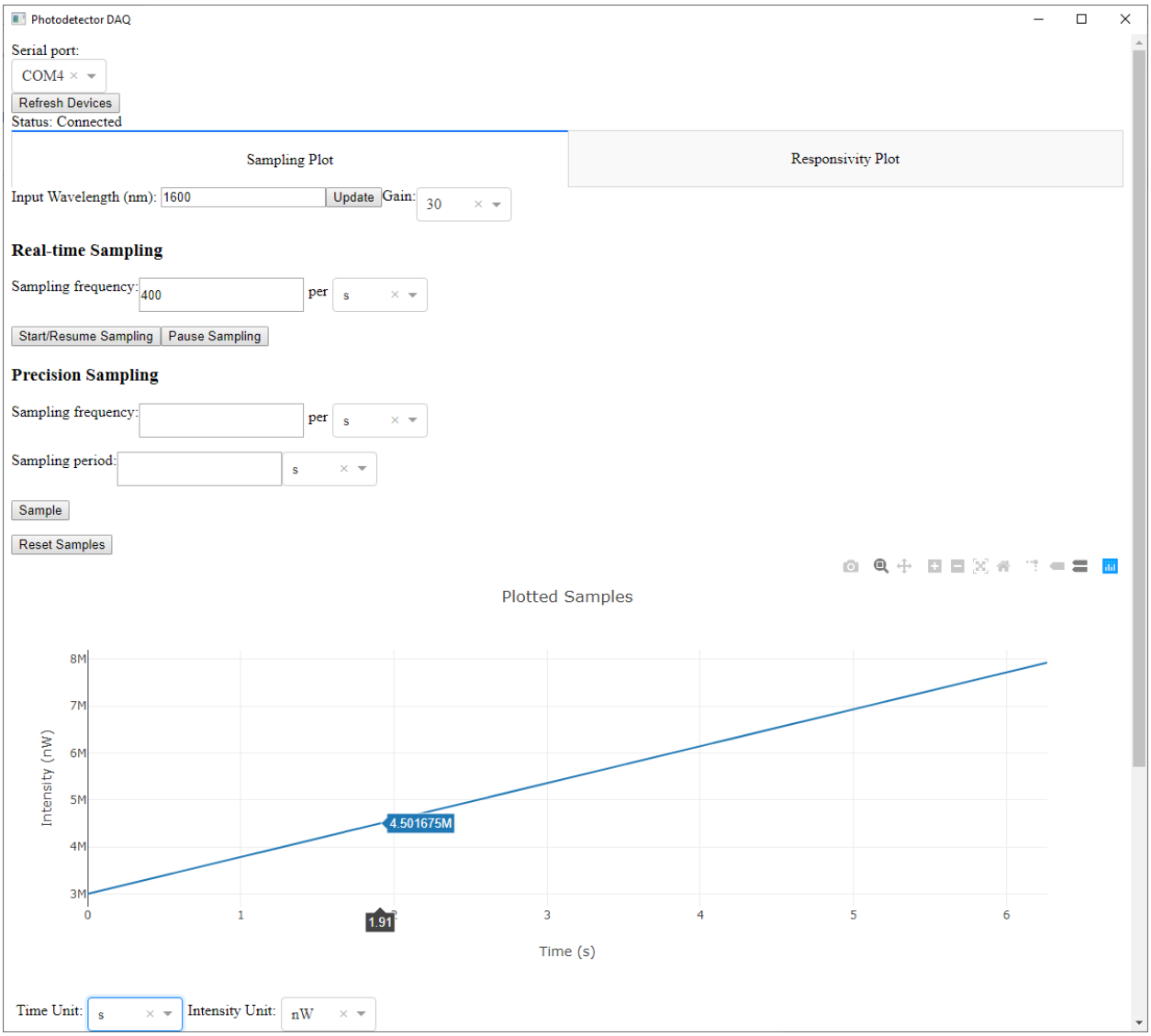


Figure 27: Data Acquisition GUI immediately after real-time sampling is paused.

7.1.2.3 Precision Sampling

Figure 28 shows the demonstration of the precision sampling mode. The user is allowed to set the sampling frequency and the length of time for which to sample. In precision sampling mode the graph does not update while the sampling occurs; instead, it updates the graph once, after the synchronous sampling is completed, and the graph is updated to contain only those samples collected over the interval. This behavior can be verified by observing that the sampling period is set to 500ms and the time interval spans from 0 to 500ms. Like in asynchronous mode, the plot's scale and units can be modified when sampling is inactive.

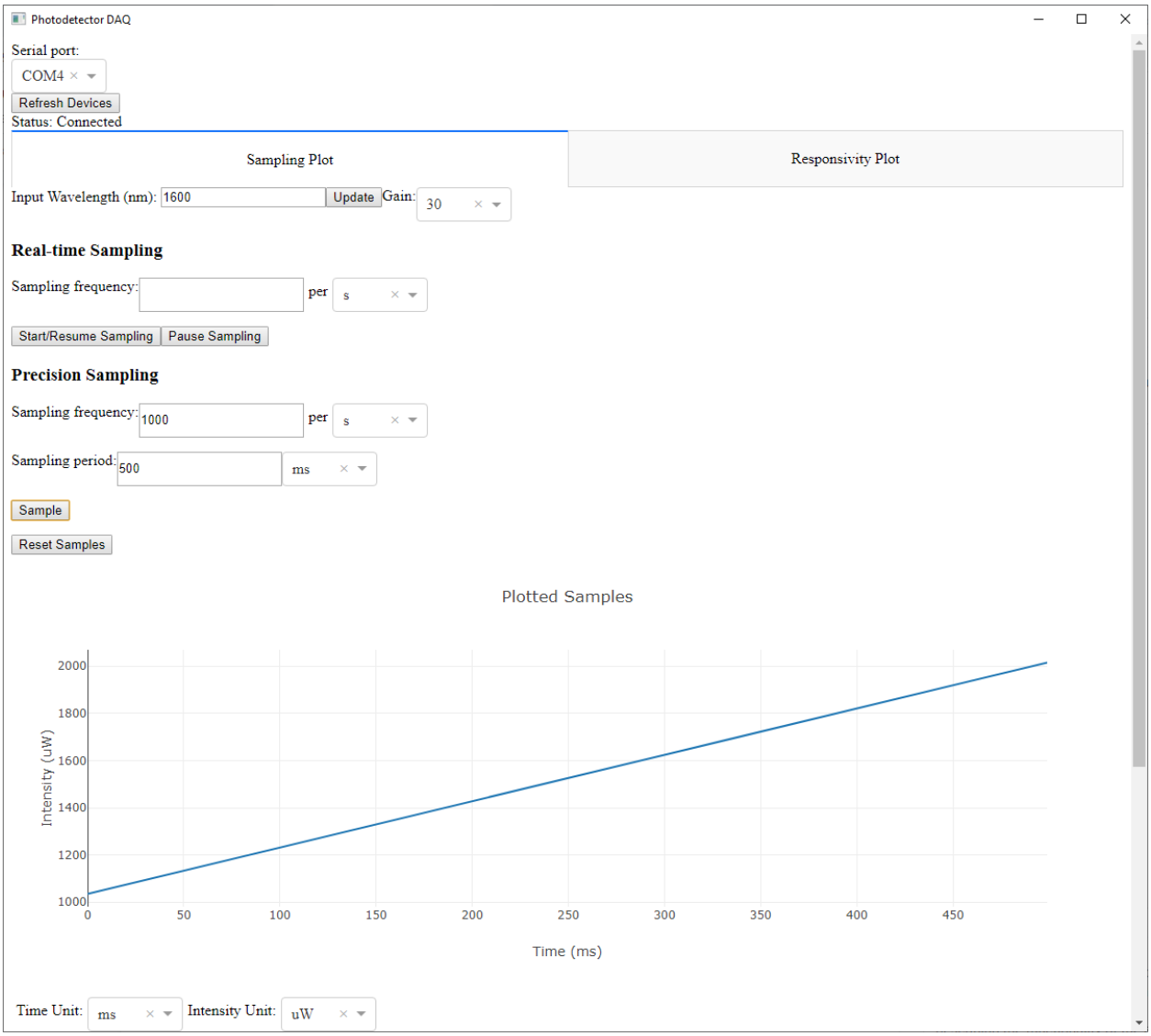


Figure 28: Precision sampling mode.

7.1.2.4 Configuring the Graph's Units

Figure 29 shows the configuration of the graph to display in units besides seconds and nanowatts. The device and samples are in the same state as Figure 23 and no additional sampling was performed. Note that as expected, the conversion from s to ms increases the time values on the axis by 1000, as expected. Likewise, changing intensity from nW to uW decreases the intensity values on the interval by 1000, as expected. Likewise, the values that are shown when the mouse is hovered over a sample are in the corresponding units.

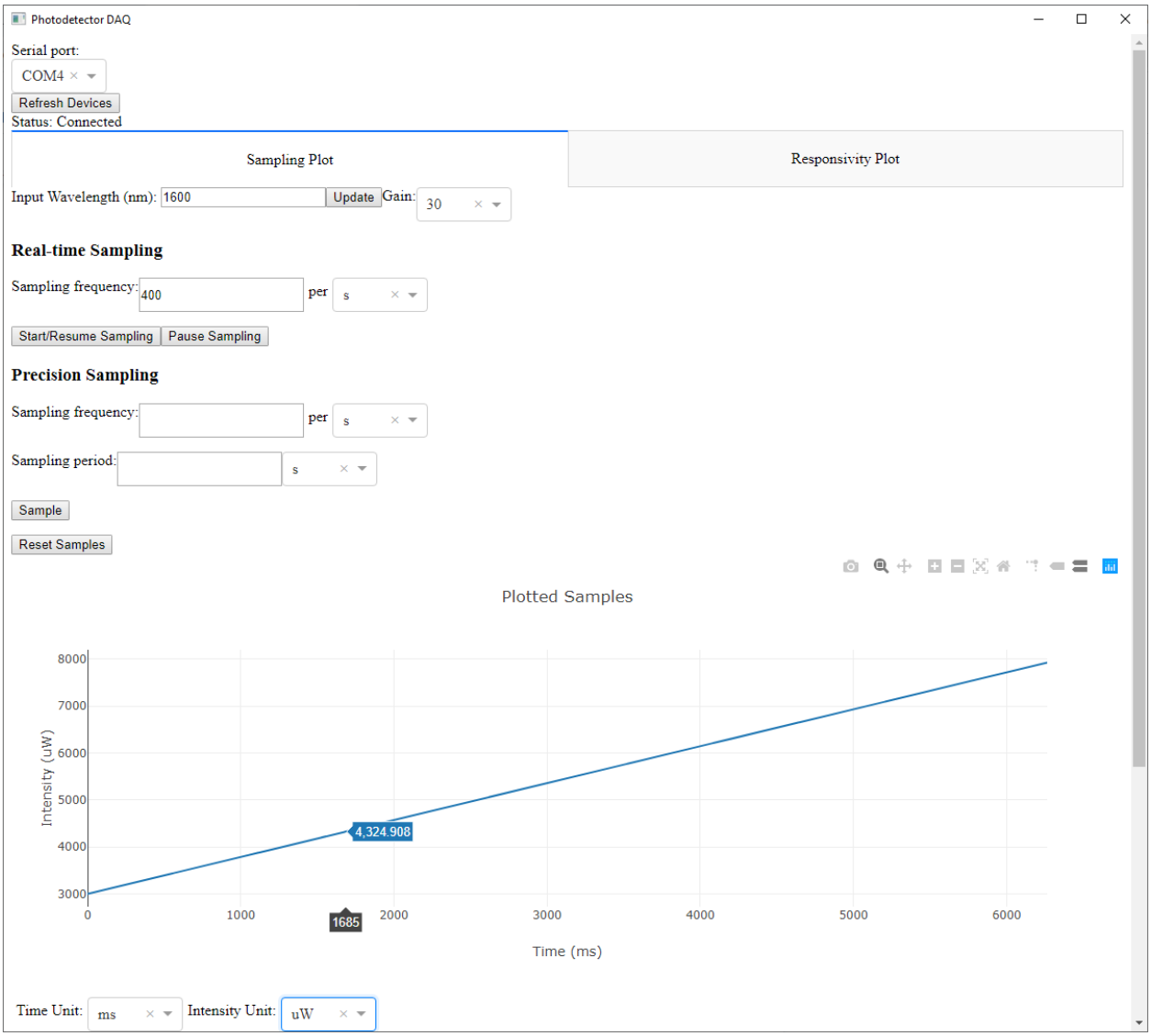


Figure 29: The Data Acquisition GUI set to display time in ms and intensity in uW.

7.1.2.5 Locating Local Maximum Intensity Sample

Figure 30 shows the functionality for the GUI to locate the sample with the maximum intensity over the user-given interval. The user is able to specify values and units for the time interval over which to locate the sample within that interval which has the relative maximum intensity. The max intensity sample data is displayed in the same units that are set for displaying the plot. As demonstrated, the maximum over the interval 3s to 4s is correctly identified as the point where the increase stops right before dropping.

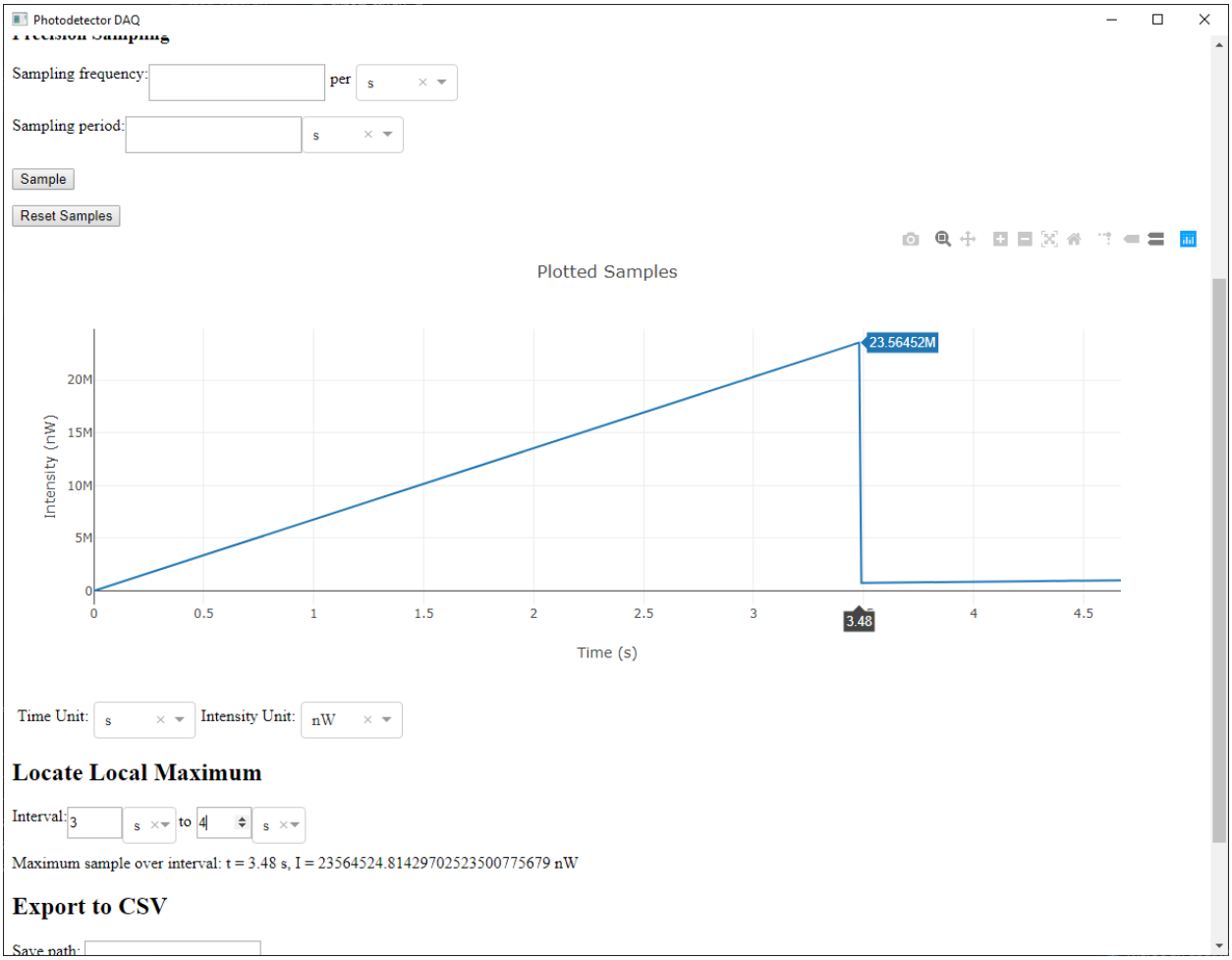


Figure 30: Demonstration of local maximum functionality

7.1.2.6 Export to CSV

Figure 31 demonstrates the functionality of the GUI to export samples to CSV. Like the local maximum, the user is able to specify the values and units for the interval they wish to export. Additionally, they are able to set the units to export the samples as. Examining the file on the command line allows us to verify that the units, start time, and end time are correctly established in the CSV.

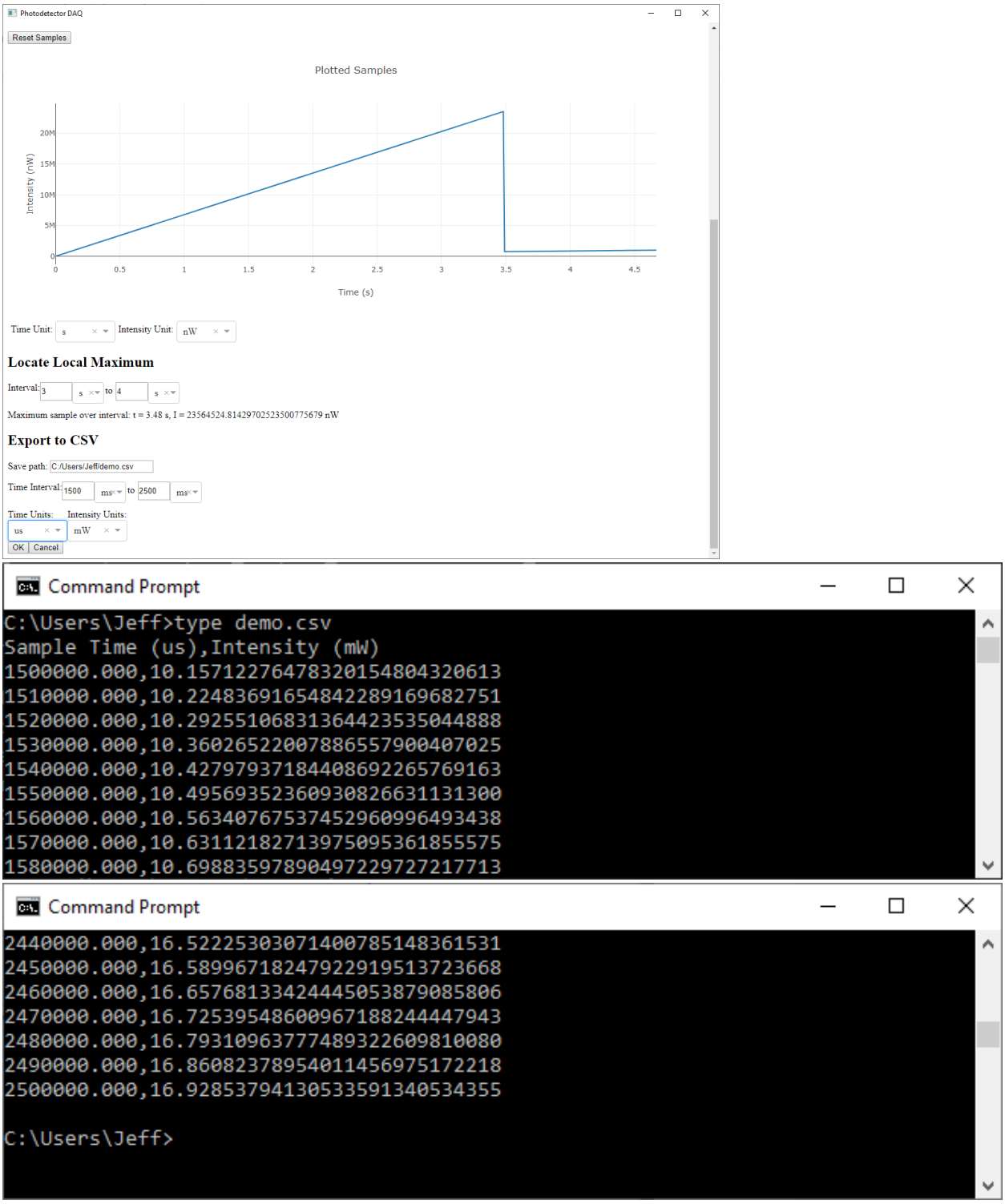


Figure 31: Demonstration of export to CSV functionality.

7.1.2.7 Evaluating Responsivity Curve

Figure 32 demonstrates the GUI's responsivity plot. This data is plotted once upon specifying the device and its units are fixed. Users are able to enter any wavelength within the operating range of the device and will be given either the exact responsivity measurement that corresponds with that wavelength, if a measurement for that wavelength exists, or a linear interpolation using the two nearest measurements.

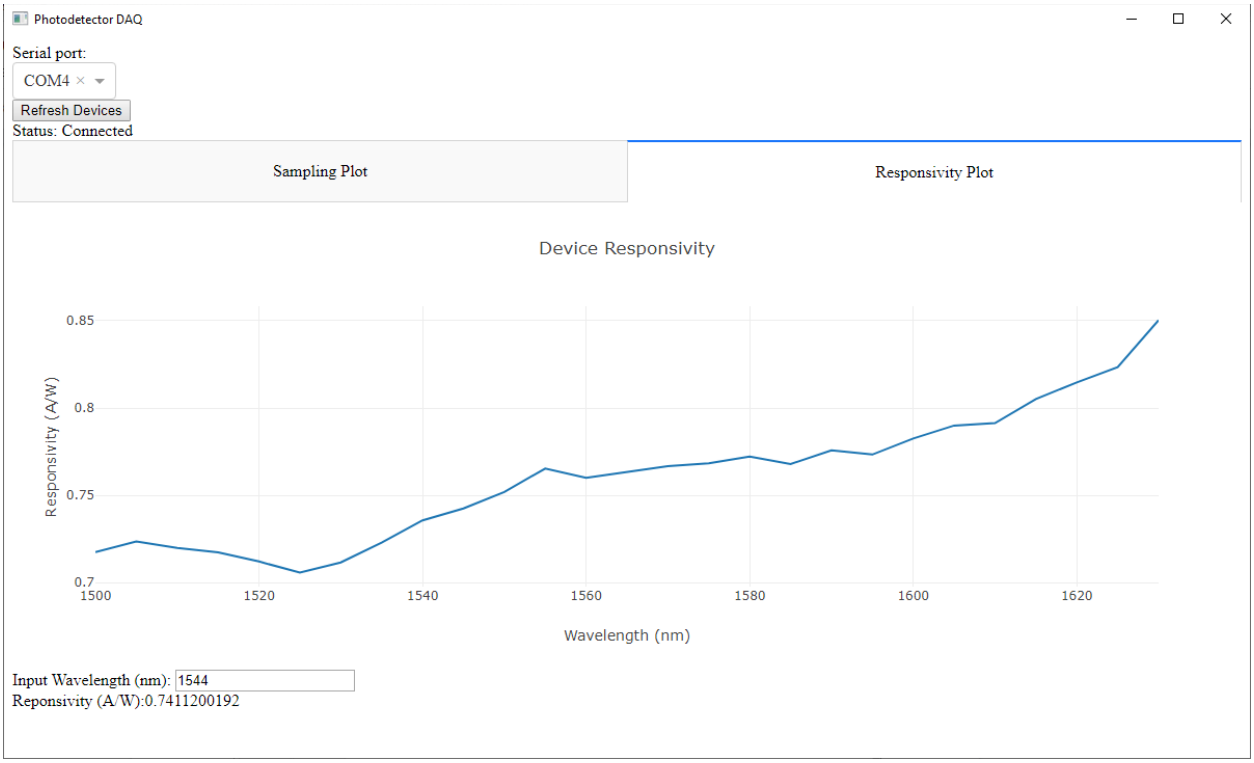


Figure 32: Demonstration of the Data Acquisition GUI's responsivity plot.

7.2 Hardware Systems

To test the hardware of the photodetector system, we planned to measure the following parameter of the circuit: the Analog Gain of the amplifier circuit compared to the Newport 2011-FC photodetector through the 200kHz frequency bandwidth. This requires producing a frequency response graph for each gain setting to ensure linear optical-to-electrical conversion. The photodiode responsivity testing and characterization was successfully completed before the campus closed for one of the two photodiodes purchased.

7.2.1 Completed Hardware Testing

7.2.1.1 Responsivity of the Photodiode

One of the two photodiodes was characterized before the campus shut down. The responsivitity measures the photodiode's current created from an input optical signal. Each photodiode's responsivity is

different and based on the physical characteristics of the device, and it is important to characterize each photodiode used in a measurement system (much like characterizing a Bipolar Junction transistor).

In order to characterize the photodiode, it was biased at 5V using a digital power supply and placed in series with a 1 kΩ resistor, which was measured to be 994Ω with a digital multimeter. An optical input from a laser was split using an optical signal splitter. One of these signals went into the photodiode being characterized, and the second light signal was sent to the Newport 2011-FC photodetector. This allows the current created from the photodiode to be directly compared to the voltage output of the Newport photodetector. By dividing the voltage output of the Newport by the Response Factor in terms of (V/Watt), we are able to calculate the actual optical power that is input to the photodiode being characterized. The Response Factor is used as a calibration tool for analyzing unknown optical properties of a device. The final calculation gives us the current (amps), divided by the input optical power (watts), which is the Responsivity of the photodiode. The setup of the photodiode biasing is shown below in Figure 33.

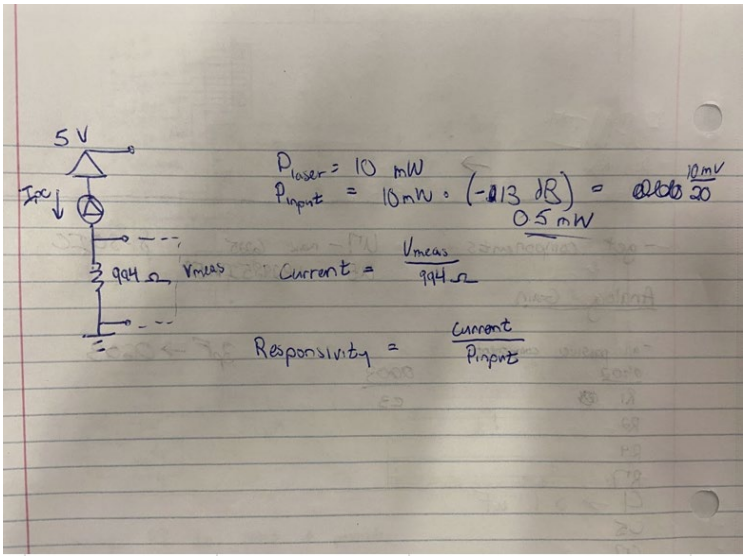


Figure 33: Photodiode Characterization setup

The Laser used in Dr. Youngblood's optical lab only has a wavelength bandwidth from 1500nm to 1630 nm. Measurements in 5 nm wavelength intervals were taken over the entire spectrum, and the calculations were automatically completed using Windows excel. The final responsivity curve of the photodiode is shown below in Figure 34.

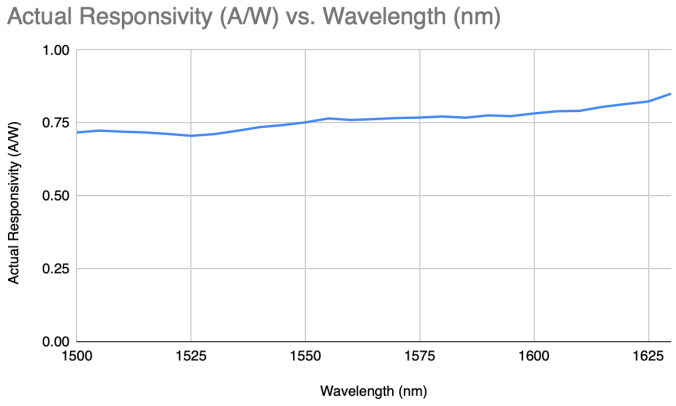


Figure 34: Final Photodiode Responsivity Graph

The raw data (wavelength vs. responsibility) was compiled into a .csv file that could be used by the software algorithm to correctly display the actual optical power on the host computer. In the future, the second photodiode also needs to be characterized, because it is used in the final design.

7.2.2.2 DC Analog Gain of the Circuit

Measuring the DC analog gain of the circuit requires a similar setup to the photodiode characterization. Because the goal of the project is to match the functionality compared to the Newport photodetector, the 9V analog output of the PCB was compared to the Newport photodetector. The same method for splitting the optical signal was employed here when testing the first prototype. One optical signal was sent to the photodiode installed into the PCB, while the other signal was sent to the Newport. The PCB and Newport were set to the same gain setting, and their DC voltage outputs were measured using the National Instrument's DAQexpress program. The DC gain was observed to be within 9% of the Newport device at a gain setting of 1000. This was not recorded because the team was under time constraints to verify AC operation of the PCB, but it showed proof of concept of accurate DC optical-to-electrical conversion.

7.2.2.3 Frequency Response of the Circuit

Creating an optical signal that with varying power over time required the use of a function generator and digital multimeter from the 12th floor labs. In order to create a non-DC optical signal, the function generator sends a sinusoidal voltage into the TSL-550 Low-frequency optical attenuation input. Inputting a sinusoidal 0 to -2V signal into the TSL-550 creates an AC optical signal of the same frequency, but does not change the set optical DC average optical power of the output laser light signal. Labview 10 runtime was then downloaded onto the host computer that both the digital multimeter and function generator were connected with. The Sweep demo could then be used to measure the analog electrical output with respect to the input AC signal from the function generator.

Figure 35 below shows the Sweep Demo output of the initial prototype, showing a flat response up to 300kHz. This output shows proof of concept of optical-to-electrical conversion with AC signals, but it did not include the 200 kHz bandwidth, because there is still a flat band up to 300 kHz.

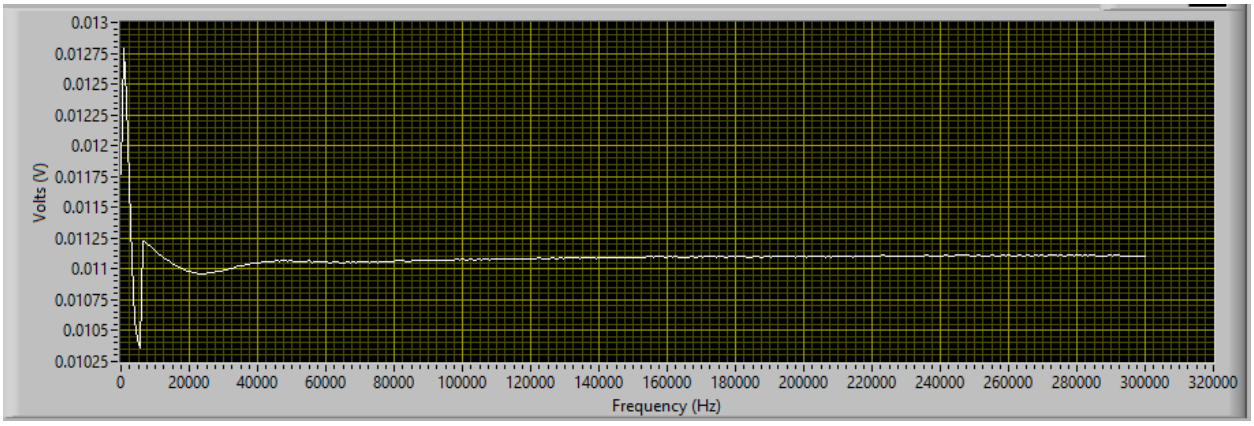


Figure 35: Prototype 1 Frequency Response at 1 x 1000 Gain Setting

This flatband error caused the team to review the anti-aliasing circuitry, where it was discovered that the anti-aliasing filter's C14 capacitor value was 0.1 µF when it should have been 100 pF in Altium. This was completed by comparing the Altium schematic to the anti-aliasing filter that was generated

online and simulated with SPICE. The change was made for the design of the final PCB, but due to campus being closed, we were unable to show proof-of-concept for the 200 kHz bandwidth on a PCB. Because the initial breadboard design showed successful filtering, the team believes correcting this mistake will translate into successful PCB operation once testing can be completed.

7.2.2.4 Sample Rate Capabilities of The Teensy

A sample rate testing experiment was established to evaluate the performance of the Teensey 3.2 Development board. The procedure included using a signal generator to evaluate the performance of data transmission at various sample rates. The set-up included attaching the signal generator output to the 16-bit ADC pin of the teensy. Firmware was written for the Teensey to retrieve and transmit incoming data in real-time. A serial port on a windows computer was used to receive this transmission. The incoming values were displayed using a putty session monitoring the serial port the USB cable was attached to. Performance was evaluated by reviewing the continuity of values displayed in the putty session. If values displayed characteristics of continuity then the Teensy was relaying the incoming signal in real time. After this experiment it was concluded that the Teensey was capable of handling sample rates of up to 30 KHz.

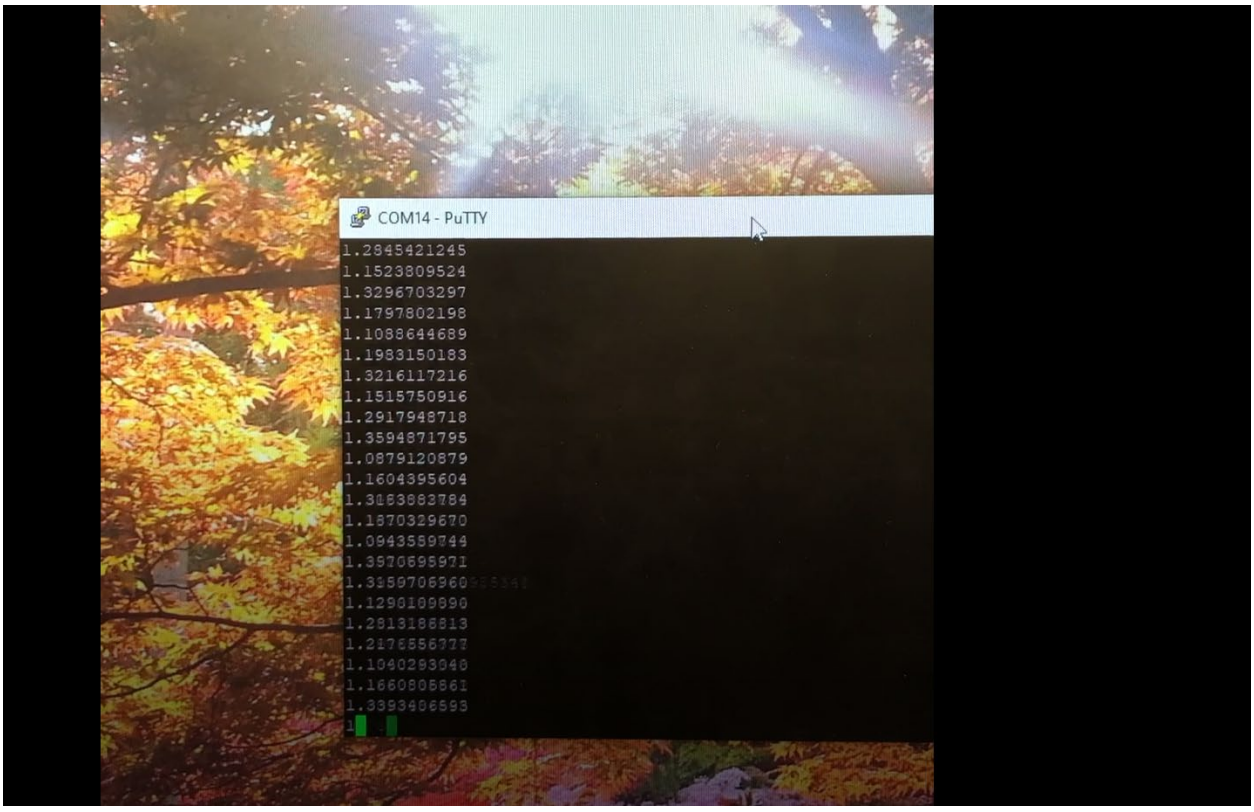


Figure 36: Sample Rate Testing

7.2.2. Future Hardware Testing & Verification Plan

Future testing of the project will require redesign of the entire board. Almost all of the components of the initial prototype are surface mount components. Given all of the tools available in

Benedum, the final design was going to utilize the same through surface mount packaging. Surface mount components offer better signal integrity and allow components to be placed closer together. Placing capacitors closer to the powering pins of the operational amplifiers typically improves performance, and our final design does not allow this. All of the surface mount op-amps were soldered to breakouts then through-hole soldered to the board. Such a roundabout method was only used because of through hole soldering requirements and the immediate availability of these op-amps from earlier in the semester. All other resistors, capacitors, and other ICs would be switched back to surface mount to increase availability, increase board density, and decrease cost.

In order to meet the initial design requirements, more testing and verification is going to be necessary. From a hardware perspective, the analog gain must match the Newport device's output as closely as possible. The gain of the circuit is affected mainly by the values of the feedback resistors of the variable gain op-amps. In order to match the Newport's analog output, the four transimpedance feedback resistor values would be fine tuned using a decade box at DC. The two resistors of the second gain stage would also then be tweaked with a decade box to ensure that output voltages are closer than the originally observed values. This testing procedure would follow section 7.2.2.2.

Once the DC values agreed as closely as possible, an AC optical signal would be input into the PCB and the Newport to determine how similar their AC characteristics are. This would follow the testing procedure of section 7.2.2.3, and might require development of another anti-aliasing filter. Because the initial SPICE simulations showed OPA192 op-amps struggling to achieve a 200kHz bandwidth at higher gains, consideration of op-amps with a higher GBP might be considered if the design cannot meet the AC design requirements.

Linearity will be tested by selecting 5 input intensities in the linear operating region of each gain setting at DC, using the same setup as section 7.2.2.1. The R^2 shall not be below 0.6 when we apply linear regression to the 5 points.

7.3 Project Box

After campus closed, we realized that physical integration in the lab would not be possible. The hardware team decided to build a project box for the PCB. See Figure 36. The box includes ports for the optical signal, the SMA connector, and the USB which connects to the teensy. In this way, we can ensure that the device is still easy to use, as well as allowing for the possibility that if someday the project is expanded on and another PCB is added to the project, there will be an easy way to keep them all organized and ensure that connections made from board-to-board are kept intact.

Figure 37: Project Box for PCB

8. Conclusions and Future Work

While the project was not fully realized and tested, a great amount of progress has been completed on the development of an open-source photodetector. The original design on the breadboard created proof-of-concept of the analog circuitry. Dr. Youngblood's original wishlist was to create an open source photodetector to easily measure multiple optical signals in the lab. Performing some testing and verification in his lab made it clear what would make this project ideal for the lab. Because his lab (and likely most optical labs) already has an NI DAQ system, creating a GUI and independent digital communication system between the PCB and host computer is slightly redundant. The software development for this project was mainly for learning purposes.

For someone that does not have access to an NI DAQ or oscilloscope, analog measurement of the signal would be quite difficult. The ADC and digital interfacing is still quite valuable and has been shown to work in proof-of-concept, but the data transmission and manipulation is well below our original goal of displaying a 200 kHz 16-bit digital signal on a host computer.

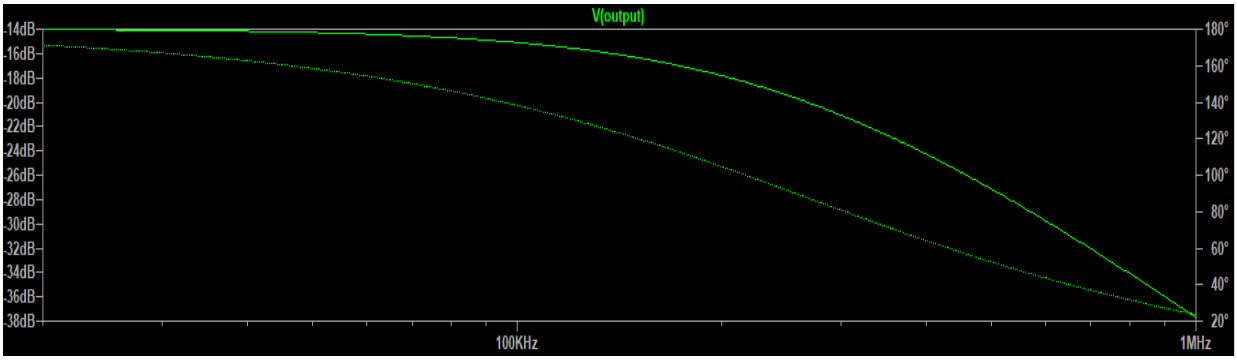
8.1. Hardware Considerations

Multiple optical inputs and multiple analog outputs would offer the most use in his lab, and would allow an Electrical Engineer to fine tune and develop a solid analog design. If the target market for this product is labs that already have NI DAQ systems, simple optical-to-analog conversion would be all that is needed. Eliminating the ADC and computer communication system would save board space and reduce power consumption.

The focus for improving this design would include op-amp testing with higher GPB. The initial breadboard design struggled to achieve the 200 kHz bandwidth at two higher gain settings, and better op-

amps might be able to compensate for this, creating a flatter passband. Fine tuning resistors and capacitors would further improve the system's gain and move its output closer to the original Newport 2011-FC device.

8.1.1. Filter Considerations



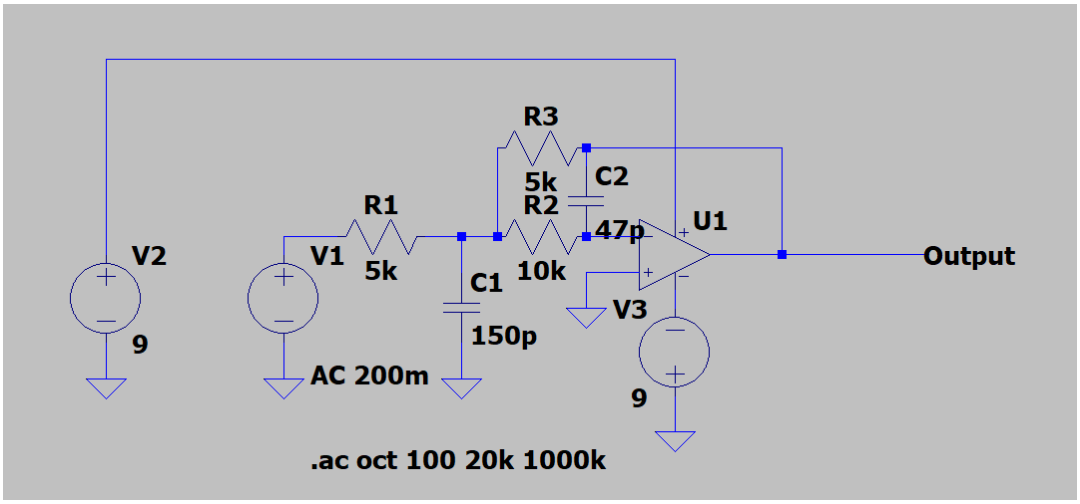


Figure 38: Multiple Feedback Anti-Aliasing Filter with SPICE response

An alternate filter design was experimented with to act as an improvement to the previous filter design. This design utilized a multiple feedback configuration in order to reduce the number of op amps needed, thus simplifying the design. This low pass filter allows us to perform the same operation without having to worry about an additional drop in signal that comes from sending the input through multiple op amps. Initial SPICE simulations show this new design is very much capable of filtering out the same frequencies as the original design. Future testing would have to be performed to test for the quality factor in the frequency response.

8.1.2 Embedded System Considerations

To continue with the current design, a memory buffer would be needed to account for the differential between retrieval and transmission speeds posed by the limitations of the Teensey 3.2 Development board. However, the device does not have enough Flash memory to prevent overlap. There

is a 4Mbaud gap between ADC intake and UART transmission. This would require at least 500,000 Bytes of memory to perform a circular buffer without overlap. The Teensy is only capable of 256,000 Bytes. Using these facts, the only alternative course of action to solve this problem is to identify a new embedded system capable of processing our data requirements.

Feature	Teensy 2.0	Teensy++ 2.0	Teensy LC	Teensy 3.2	Teensy 3.5	Teensy 3.6	Units
Price	$16.00	$24.00	$11.65	$19.80	$24.25	$29.25	US Dollars
Processor	ATMEGA32U4	AT90USB1286	MKL26Z64VFT4	MK20DX256VLH7	MK64FX512VMD12	MK66FX1M0VMD18	
Core	AVR	AVR	Cortex-M0+	Cortex-M4	Cortex-M4F	Cortex-M4F	
Rated Speed	16	16	48	72	120	180	MHz
Overclockable	-	-	-	96	-	240	MHz
Flash Memory	31.5	127	62	256	512	1024	kbytes
Bandwidth	32	32	96	192	192	411	Mbytes/sec
Cache	-	-	64	256	256	8192	Bytes
RAM	2.5	8	8	64	256	256	kbytes
EEPROM	1024	4096	128 (emu)	2048	4096	4096	bytes
Direct Memory Access	-	-	4	16	16	32	Channels
Digital I/O	25	46	27	34	58	58	Pins
Breadboard I/O	22	36	24	24	40+2	40+2	Pins
Voltage Output	5V	5V	3.3V / 5V	3.3V	3.3V	3.3V	Volts
Current Output	20mA	20mA	5mA / 20mA	10mA	10mA	10mA	milliAmps
Voltage Input	5V	5V	3.3V Only	5V Tolerant	5V Tolerant	3.3V Only	Volts
Interrupts	4	8	18	34	58	58	Pins

Figure 39: Teensey Develpoment Board Specifications

An alternative approach could be to use a Native Instruments Data Acquisition System (DAQ). DAQ's are very powerful data acquisition tools that have an existing track record within the ECE department. Using a DAQ would simplify the current system's architecture a great deal since DAQs already have professional grade firmware and software capable of handling the device's data processing requirements. In addition they offer the flexibility of scaling the device's functionality by having the ability to process multiple signals simultaneously; A requirement we had to let go during the design phase.

8.2. Software Considerations

Were this project to be redone, we would strongly consider using a more performant and optimizable choice of language and frameworks than standard Python and Dash. The agreement to use Dash was made before the team fully understood exactly how a Dash app was made, which led to a "square peg in a round hole" syndrome where Dash, best suited for web applications which query and visualize large datasets, was being used to create a desktop app that needed to be quite fast relative to the typical usage of Dash applications. The inability to optimize standard Python 3 and the lack of optimization of the front-end code autogenerated by Dash led to an unoptimized app which failed to meet the performance criteria despite meeting all of the functional requirements.

Assuming that the requirement to use Python still had to be met and the scope of the project was not restricted by scope of the class, strong consideration would be given to implementing the youngblood_photodetector library in Cython, a superset of Python 3 which allows for the annotation of standard Python code to be compiled into C/C++ and therefore be statically optimizable, use a less intensive runtime, and ideally be more performant overall.

9. Team

Our team is composed of five members; three (Nick Gismondi, Malik Rivers, and Brendan Schuster) are Electrical Engineering students and two (Kevin McGoogan and Jeffrey Socash) are Computer Engineering students. The following five sections will each describe the skills and responsibilities of each team member.

9.1 Nick Gismondi

Nick Gismondi is currently an Electrical Engineering student. He has experience in digital circuit design as well as PCB design. His primary responsibilities were designing the PCB and then thoroughly testing the board. Then once the course went remote, he was responsible for constructing the project box and ensuring that the PCB could be seamlessly integrated into it.

9.2 Kevin McGoogan

Kevin McGoogan is a Computer Engineering student. He has experience in embedded system design as well as python development experience. His primary responsibilities were designing and programming the on-board embedded system which converts analog input from the photodetector circuit to digital output for the host computer and handles commands from the host computer to start and stop sampling.

9.3 Malik Rivers

Malik Rivers is an Electrical Engineering student. He has experience in analog circuit design and PCB design. He was primarily responsible for the design and implementation of the low pass anti aliasing filter circuit. This responsibility included the initial design of all circuit schematics and calculations and the early prototyping and testing of the circuit on breadboard. Other responsibilities included the implementation and testing of all analog prototype systems for the analog gain amplifier with the low pass filter circuit stage.

9.4 Brendan Schuster

Brendan Schuster is an electrical engineering student. He has experience with analog and digital circuit design and layout methods and SPICE simulations. He is the main line of communication with Dr. Youngblood for the hardware design. His responsibilities included analog design, parts ordering / budgeting, and PCB testing.

9.5 Jeffrey Socash

Jeffrey Socash is a Computer Engineering student. He has experience with Python development and user experience design. His responsibilities were the development of all Python code in the Photodetector Library

and Data Acquisition GUI. He created, implemented, and packaged the youngblood_photodetector library and tested its functionality. He also created the data_acquisition package and the Dash app contained within it.

References

[1] J. Feldmann, N. Youngblood, M. Karpov, H. Gehring, and X. Li, et. al. “Parallel convolution processing using an integrated photonic tensor core,” 2020.

[2] J. Feldmann, N. Youngblood, X. Li, C. D. Wright, H. Bhaskaran and W. H. P. Pernice, "Integrated 256 Cell Photonic Phase-Change Memory With 512-Bit Capacity," in *IEEE Journal of Selected Topics in Quantum Electronics*, vol. 26, no. 2, pp. 1-7, March-April 2020, Art no. 8301807.

[3] Newport Electronics, “Models 2001 and 2011 User Manual. Front-End Optical Receivers.” https://www.newport.com/medias/sys_master/images/images/hc5/hb2/8796990341150/2001-2011-User-Manual-RevB.pdf.

[3] Osio Optoelectronics, “Equations and General Information.”http://www.osioptoelectronics.com/application-notes/an-photodiode-parameters-characteristics.pdf .

[4] “Dash DAQ,” *Dash DAQ*, 08-Jun-2018. [Online]. Available: https://www.dashdaq.io/. [Accessed: 18-Feb-2020].

[5] “Teensy 3.2 & 3.1 - New Features,” *PJRC*. [Online]. Available: https://www.pjrc.com/teensy/teensy31.html. [Accessed: 18-Feb-2020].

[6] Texas Instruments, “Precision Micropower Shunt Voltage Reference,” LM4040-AIZ-5.0 datasheet, Oct. 2000 [Revised Jun 2016].

[7] Freescale Semiconductor, “K20 Sub Family K20P64M72SF1,” K20P64M72SF1 datasheet, [Rev Nov 2012]

[8] “pySerial 3.0 Documentation,” *Welcome to pySerial's documentation - pySerial 3.0 documentation*. [Online]. Available: https://pythonhosted.org/pyserial/. [Accessed: 18-Feb-2020].

[9] N. Thomas, “How To Use The System Usability Scale (SUS) To Evaluate The Usability Of Your Website,” *Usability Geek*, 13-Sep-2019. [Online]. Available: https://usabilitygeek.com/how-to-use-the-system-usability-scale-sus-to-evaluate-the-usability-of-your-website/. [Accessed: 18-Feb-2020].

[10] “Input Examples and References,” *dcc.Input*. [Online]. Available: https://dash.plot.ly/dash-core-components/input. [Accessed: 18-Feb-2020].

[11] “Dropdown Examples and Reference,” *dcc.Dropdown*. [Online]. Available: https://dash.plot.ly/dash-core-components/dropdown. [Accessed: 18-Feb-2020].

[12] “Graph Examples and Reference,” *dcc.Graph*. [Online]. Available: https://dash.plot.ly/dash-core-components/graph. [Accessed: 18-Feb-2020].

[13] “Pint: makes units easy -- pint 0.10.1 documentation” [Online]. Available: https://pint.readthedocs.io/en/0.10.1/ [Accessed: 23-Apr-2020]

[14] “Cython: C-Extensions for Python” [Online]. Available: https://cython.org/ [Accessed 23-Apr-2020]

Acknowledgements

The overall idea for this project came from the hassle Dr. Youngblood experiences in measuring and recording optical data. Much of the background section of the proposal highlights the need for this technology. Our time speaking with Dr. Youngblood was instrumental in developing the background for our final design.