Loyola Marymount University
Department of Electrical Engineering and Computer Science
ELEC 402: Senior Project

Design of a Diode-Based Gas Chromatography System for Classroom Use

Matt McPartlan & Conor Green
mmcpart1@lion.lmu.edu cgreen18@lion.lmu.edu

Advisors:
Dr. Hossein Asghari
Dr. Robert Senter

Second version

Los Angeles, CA, April 2020

Abstract

Gas chromatography (**GC**) is an analytical separation technique in which a mixture is volatilized, pushed through a separation column by a carrier gas, and detected as each constituent component exits the column. Entry-level teaching instruments typically use thermal conductivity detectors (**TCD**s), which consist of a thin electrically heated filament suspended in the flowing gas. These TCDs provide good sensitivity but are fragile and require an uninterrupted flow of inert gas, usually helium, to operate without burning out the detector. This project developed an economic and sufficiently accurate gas chromatography instrument using a 1N4148 diode detector and ADS1115 analog to digital converter. A colorimetric array was also developed and tested but were not implemented in the final instrument due to COVID-19 related restrictions. The colorimetric detector, which makes use of a planar array of cross-responsive metalloporphyrin dyes, can produce identifying color patterns describing different types of ligating compounds. The diode detector was tested in both single-sided and differential pair configurations to assess its susceptibility to drift. After testing each potential detector, it was determined that a single-sided diode detector provided sufficient stability. In the future, this diode detector should be followed by colorimetric array analysis to provide identifying information about each separated compound. The collection of this additional compound-identifying information is a new and significant capability that could be of great use in a teaching lab, especially when attempting to determine if a desired product was synthesized. Notably, all of these detectors can function outside of an inert environment.

Table of Contents

1. Introduction

Gas chromatography (**GC**) is one of the most powerful tools currently available to identify components in an unknown sample and assess the purity of reagents. Separation of compounds with capillary or packed column gas chromatography utilizes differences in volatility, thermal conductivity, and polarity (amongst other properties) between the compounds being separated to split the mixture into its constituent components. As the sample is injected into the mobile gas phase, it is vaporized in the injector block and swept into the column by the carrier gas. As the sample moves through the column, interactions with the stationary phase column coating change the retention time of each of the unique compounds making up the sample. This change in retention time affects both the order and time of elution, with more volatile compounds generally having lower retention times and coming out of the column first.

As each compound elutes from the chromatography column, a detector measures physical differences between the pure carrier gas and the carrier/compound mix. There are countless types of detectors used in gas chromatography, but the most common are thermal conductivity detectors (**TCD**) and flame ionized detectors (**FID**). Flame ionized detectors work by passing the output of the column through a hydrogen-air flame and measuring the current produced by ions in the flame. The sensitivity and stability of FID detectors has led to their widespread use in research grade instruments, but they suffer from two key drawbacks that make them unsuitable for use in a teaching lab. The most significant issue is that the flame requires the use of compressed hydrogen to operate. The hazards and equipment associated with maintaining a safe hydrogen supply for multiple instruments in a lab makes FID detectors too expensive for entry level chromatography applications. In addition to the safety and cost considerations, FID detectors can only produce a strong signal when the compound passing through the flame becomes ionized. In practice, this means that an FID works reliably only when dealing with organic compounds.

Thermal conductivity detectors do not exhibit any of the problems faced by FID detectors, but they are less sensitive. This is an acceptable compromise for entry level applications. TCDs are commonly used for analysis of inert compounds and in teaching labs, where simplicity and reliability are more important than low detection limits. Additionally, TCDs measure the difference in thermal conductivity between the carrier gas and a carrier/sample mix. This is done by suspending an electrically heated filament in the flow of carrier gas as it comes off of the column. The carrier gas, traditionally helium in TCD applications, has a constant thermal conductivity that allows transfer of heat away from the filament. Sample compounds exiting the column will have thermal conductivities different than the carrier gas, causing a change in filament temperature that results in an imbalance in the Wheatstone bridge detector array as seen in Figure 1.1

The diode detector described in this report is a derivative of commonly used TCD detectors, but it measures heat capacity of the carrier/eluant mixture rather than thermal conductivity. True TCD detectors are glowing hot filaments that resemble the filament inside of an incandescent light bulb. These filaments are self-heating and rely on the carrier/eluant mixture to cool the detector. A sudden change in the thermal conductivity of the material exiting the separation column (such as a transition from pure helium, about 200 mW/mK, to ethanol and water vapor, about 30 mW/mK) will cause the filament temperature to rise dramatically and change the electrical properties of the wire. The diode detector described here is not self heating, and relies on residual heat from the eluant to heat the diode and induce a measurable

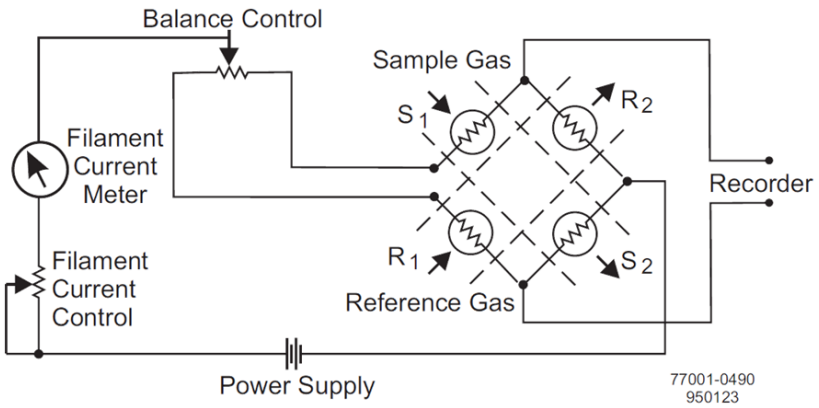


Figure 1.1: The classic Wheatstone balanced bridge detector circuit used in differential pair GC-TCD applications [1].

change.

Though gas chromatography is a powerful technique by itself, it only reports the time and order that compounds elute from the column. To use a pure gas chromatograph for identification of an unknown compound, the retention time must be matched to that of a standard solution run on the exact same machine and column. The difficulty associated with using a gas chromatograph in this capacity can be avoided with the addition of a second detector that can collect more information about each compound as it elutes. Mass spectrometers are one example of a commonly used secondary detector than can identify things as they exit the separation column. An important secondary goal of this project is to explore the use of metalloporphyrin dye arrays to colorimetrically identify compounds based on their ligating properties. This system is a completely novel approach to the problem of compound identification, and could lead to a unique and inexpensive instrument. Figure 1.2, taken from a recently published article in Nature [2], demonstrates what this could look like in practice.

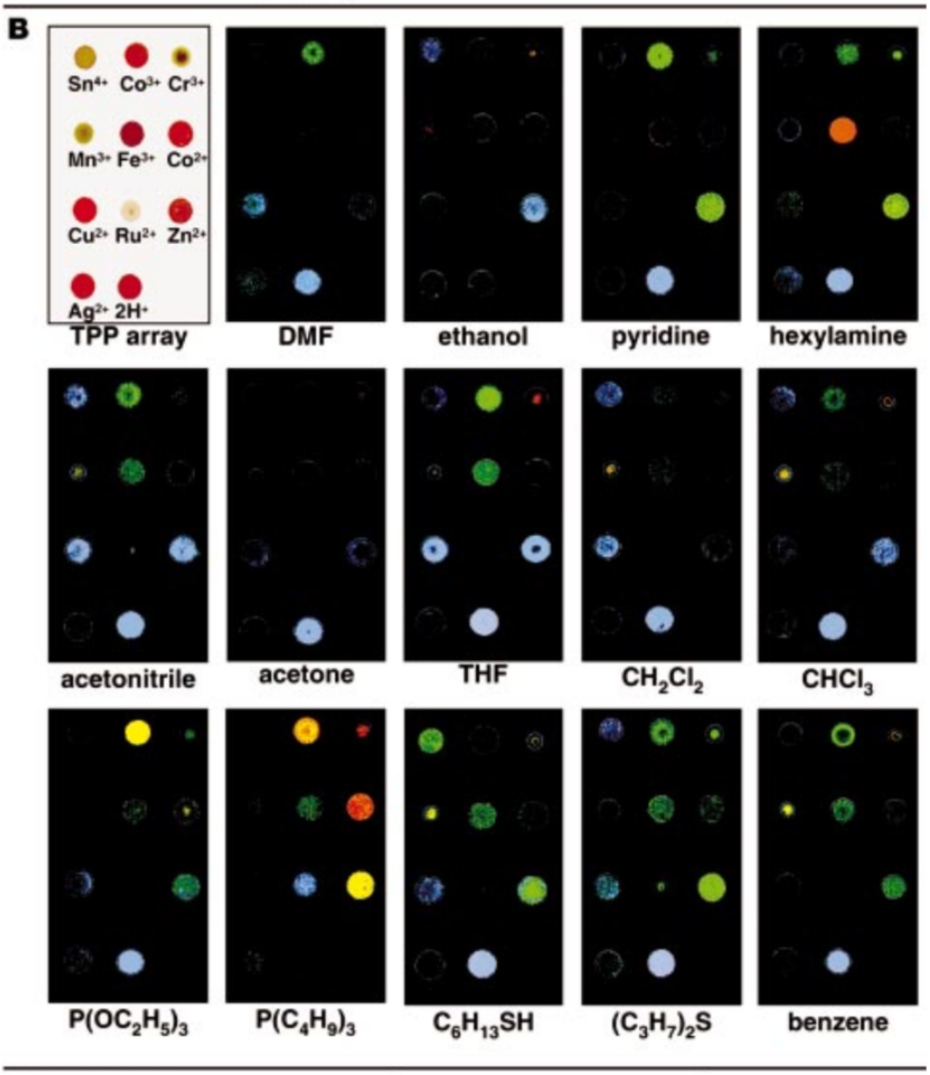


Figure 1.2: A graphic produced by [2] shows the type of data produced by a colorometric detector array of the type explored here.

2. Objectives

This project seeks to produce a GC that 'just works' using a rugged and inexpensive detector system that is easy to maintain. Though this is a simple goal, detector design is seriously limited by the harsh conditions inside of a GC instrument. The detector must handle sustained temperatures up to 200°C and must have an inert coating to protect the detector from oxidation and potentially harsh chemicals being examined within a sample.

There are five significant modules that must be complete before the end of the Spring 2020 semester: detector hardware, high-throughput columns, temperature control systems, chassis/cosmetic design, and a GUI interface. These are all necessary to create a final product that can be used by less technical students in a lab.

2.1 Collaboration with the LMU Department of Chemistry and Biochemistry

Design requirements, established in collaboration with the Loyola Marymount University (**LMU**) Department of Chemistry and Biochemistry, are oriented towards giving organic chemistry students the chance to interactively learn about gas chromatography through the analysis of compounds synthesized in lab. A specific list of features requested is shown below:

1. **Service Interval:** A service interval, the time between adjustment or repair during normal operation, of at least 2 weeks in a classroom setting.

2. **Carrier Gas:** Air, nitrogen, and helium are acceptable, but air is preferred.

3. **Maximum Operating Temperature:** Detectors must be able to withstand up to 150°C indefinitely, and up to 200°C for brief (up to 30 minute) intervals.

4. **Detection Limit:** The gas chromatograph should be able to detect impurities that make up more than 5% of the total analyte mixture.

5. **Procurement Cost:** The initial cost of each instrument should be less than $500, excluding the cost of the column.

6. **Instrument Size:** The instrument should be able to sit on a standard lab bench and me movable by the operator with no additional equipment.

7. **Scalability:** It should be possible to acquire more of the instruments if desired (this should not just be a one-off design).

8. **Documentation:** This instrument must be serviceable for many years, and will likely require maintenance at some point. There must be documentation describing how the instrument works, where to get replacement parts, and a detailed user guide oriented towards students.

9. **Ongoing Support:** Instrument maintenance must be possible for a reasonably skilled technician to do. Detectors, septa, and other components that wear out must be replaceable in the future.

10. **User Interaction:** The user must be able to perform important data analytics and output the measured values in convenient formats. The graphical user interface must provide functionality to control the heating elements, collect a sample, manipulate the data, save the state of the device for later dates, and output the results to convenient (image) formats.

2.2 Design features addressing each requirement

After collaborating with the LMU Department of Chemistry and Biochemistry to develop the design requirements described in section 2.1, it was important that those objectives were taken seriously. This section reviews the ability of the prototype developed here to address the needs and concerns of the Chemistry and Biochemistry department. Additionally, it should be noted that the COVID-19 shutdown significantly affected the ability of this team to collaborate on hardware, testing, and other parts of the design process. The prototype discussed in this report represents a good-faith attempt to meet the design requirements to the fullest extent allowed by current circumstances.

1. **Service Interval:** Though rigorous reliability testing could not be completed before conclusion of the Spring 2020 semester, the cold-detector design described previously allows the diode to operate at a low temperature of only about 50°C. This is well within the rated capacity of the detector and is not expected to lead to premature detector failure. Similarly, the hermetically sealed glass package of the selected small signal diode should not be susceptible to detector oxidation or degradation over time.

2. **Carrier Gas:** The prototype design was tested using nitrogen, compressed air, and helium. Due to the similar heat capacity of nitrogen (and compressed air, which is mostly nitrogen as well) to many of the organic compounds being measured, the diode detector described here is not able to reliably use nitrogen or compressed air as a carrier. Helium works very well, however.

3. **Maximum Operating Temperature:** The unique cold-detector design of this prototype allows the injector block and oven to be operated at temperatures that far exceed the temperature rating of the diode. Sustained oven temperatures up to 250°C are possible, and the injector was tested to maintain a temperature up to 300°C. Though these temperatures will very rarely be used in practice, it demonstrates the success and robust nature of this prototype design.

4. **Detection Limit:** Due to the COVID-19 shutdown, it was not possible to use the resources at LMU to measure the detection limit. A suitable test chemical of analytical grade purity was not available at home. This will need to be assessed in the future.

5. **Procurement Cost:** The material cost of the instrument (not including the column or any supporting hardware / tools purchased) was about $400, broken down as follows:

   (a) **Aluminum / chassis:** $100

   (b) **Heating elements:** $50

   (c) **Carrier gas control:** $150

(d) **Electronics:** $100

(e) **Insulation:** $50

(f) **Cables, connectors, etc.:** $50

This list does not include materials bought to support the production of the prototype unless they are physically attached or critical to the functionality of the prototype.

6. **Instrument Size:** The instrument is roughly the size of a desktop computer and weighs approximately 40 lbs fully assembled. It can easily be moved by a single person and the exterior of the prototype remains cool even after extended periods of testing.

7. **Scalability:** The scalability of this design is not yet known. There are no parts on the instrument that are difficult to produce or purchase, but assembly was very labor intensive and may be a problem in the future.

8. **Documentation:** Documentation, specifically regarding the software component, is written thoroughly and provided on GitHub. Hardware documentation is included throughout this report in sufficient detail to maintain the instrument.

9. **Ongoing Support:** Both the GitHub documentation and this report should include sufficient detail to maintain the instrument, but if an unanticipated problem arises it will be possible to contact the project managers for several years in the foreseeable future.

10. **User Interaction:** The Python based GUI allows for real time plotting, peak integration, peak picking, determination of retention time, chromatogram normalization, baseline detection and correction, exporting and loading of spectra as .gc files, and exporting / saving the current plot field as a .png or .jpeg file.

3. Detector Options

3.1 Background

Although sensitivity of the detector is the most obvious way to achieve a lower detection limit, there are several other relevant factors that can impact the performance of the instrument. Column design, carrier gas selection, and carrier gas flow rate can all significantly affect the separation and detection of compounds within the column. Additionally, the temperature of the injector, column, and detector can be set independently to provide the best separation for different types of compounds. The temperature of each of these components can have a major impact on the degree of separation and the efficacy of the detector.

While many of the aforementioned parameters, especially regarding flow rates and temperatures, can be optimized experimentally for different mixtures, the type of carrier gas is generally fixed. The design of thermal conductivity detectors that can be used with a nitrogen or air carrier gas is especially challenging due to the similarities in thermal conductivity between nitrogen and many of the organic compounds being separated. The difference between nitrogen and helium, the traditional choice, is significant. At 125°C, nitrogen has a thermal conductivity of 32.3 $mW/(m \cdot K)$ compared to helium's 190.6 $mW/(m \cdot K)$. The absolute number does not matter so long as the carrier is sufficiently different from the material being examined, but it can be seen looking at Table 3.1 that helium is extraordinarily thermally conductive compared to other common compounds. This is the primary reason that helium is the industry standard for use with TCD detectors.

Compound	25°C	125°C	225°C
Acetone	11.5	20.2	30.6
Methane	34.2	49.1	66.5
Methanol	-	26.2	38.6
Ethanol	14.4	25.8	38.4
Hexane	-	23.4	35.4

Table 3.1: Thermal conductivity ($mW/(m \cdot K)$) of several common organic compounds.

Though the use of nitrogen can be somewhat limiting, preliminary studies conducted as part of this project have shown that a combination of differential amplification, active filtering with hardware, and multi-pole recursive digital signal processing can be an effective way to extract a signal from a diode placed in a stream of nitrogen carrier gas. Only 2-component mixtures have been studied thus far and the limit of detection is relatively high, but ongoing experimentation with detector geometry and temperature has shown great promise. The detectors proposed here are a mix of TCD derivatives and detectors that make use of other physical properties to differentiate between carrier and carrier/compound mixtures.

3.2 Introduction to Detector Selections

The design of a new gas chromatograph for use in teaching labs depends almost entirely on the development of a new detector itself. This project originally intended to consider four types of detectors: diodes, carbon-film resistors, commercial methane sensors, and a colorimetric array assessed through the use of a camera and image processing software. The electrical detectors (diodes, resistors, and the methane sensor) were to be tested in both single-sided

and differential pair configurations to arrive at the most stable possible configuration. The colorimetric detector is fundamentally different from the electronic detectors because it makes use of organometallic ligand binding interactions to produce a signal that is different for many types of compounds. This type of cheap identifying detector is a completely novel approach, and would likely be coupled with a primary quantitative detector. Though information about all of the intended detector options is included here, only the resources to test the diode detector were available due to the COVID-19 shutdown.

3.2.1 Rhenium-Tungsten

The most common RTDs used in teaching instruments are electrically heated rhenium-tungsten filaments. These detect the difference in thermal conductivity between the carrier and sample gases. The Gow-Mac Series 400 gas chromatograph currently used in the LMU organic chemistry labs uses a rhenium tungsten filament. Because this type of filament is the dominant product that this project seeks to replace, all known mixtures used to test the alternate detector designs proposed here were also analyzed with a rhenium-tungsten filament. This was used to assess the relative sensitivity of all new detectors examined here and gauge how they will perform in lab.

3.2.2 1N4148 Diodes

The use 1N4148 small signal diodes configured to behave as a rugged TCDs is appealing primarily because of its the simplicity and low cost. Additionally, the hermetically sealed glass package insulation serves as an effective barrier that is relatively inert. The choice of this diode in particular was based on data from previous literature[3], but experimentation with surface-mount diodes is also being considered. The 1N4148 is also unique due to its high temperature rating for sustained operation at up to 160°C. In practice, a signal is extracted from the diode by holding the current constant and monitoring the change in forward voltage drop as a function of temperature.

3.2.3 Carbon-Film Resistors

Based on the success of the diode tests, a carbon-film resistor in an 1/8 W package is also being tested in all applications that the diode is. The use of a carbon-film resistor is analogous to a much less sensitive rhenium tungsten filament. The same constant current biasing, analog filtering, and software filtering applied to the diode detector was to be applied to this resistive detector as well.

3.2.4 Commercial Methane Detectors

The use of methane sensors as a GC detector is an unorthodox approach to solving the problem, but it is not without precedent. Generally the methane detectors used here, shown in Figure 3.1, are used in commercial building alarms and industrial applications. Though their use as GC detectors is worth investigating, there are two significant concerns that need to be addressed before deploying instruments based on these detectors. The first is the plastic housing that holds the detectors together, which is unlikely to be chemically inert. This could potentially reduce the life of the detector. The second issue is temperature tolerance. Though

Figure 3.1: An example image of the type of methane detectors being examined in this project [4].

the detector does not need to be as hot as the column or injector blocks (in most cases), it must still be hot enough to prevent analyte condensation. The temperature tolerance of these detectors has not yet been established.

3.2.5 Colorimetric Array

The colorimetric array of metalloporphyrin dyes is potentially the most interesting detector being investigated here, as it could have the ability to identify compounds as they elute from the column. This is a new capability not possible with either FID or TCD detectors, which only report that something is eluting from the column. Traditionally, if identification of compounds is required, the retention time of each is compared to a set of known standards that must be run in the same machine configured in exactly the same way. Figure 1.2, in comparison, shows the expected output from a detector of the type being investigated here. This sort of signal, which must be interpreted and processed differently (potentially with a camera), produces a unique signature for different molecules based on their ligating properties. Even weakly ligating compounds can be perceptibly detected. The primary concerns associated with this detector revolve around response time, particularly how long it takes for the array to reset, and durability.

4. Design

4.1 Hardware

Because a complete gas chromatograph is a relatively complex instrument with many subsystems, the design of the prototype hardware was broken down into five main categories: chassis design, carrier gas control, heating element control, detector operation, and separation column design. Each of these subsystems are critical to the operation of the instrument as a whole, and each was designed and tested independently before being assembled and incorporated into the instrument. This allowed for small errors to be caught and corrected early in the design process when minimal disassembly and reworking was required.

4.1.1 Chassis Design

Because there are parts of the instrument that can get as hot as 300°C, the chassis must both protect the user and insulate the heated components. To accomplish both of these goals while maintaining geometric simplicity for manufacturing, a system of two nested boxes was selected. The heated parts of the instrument, such as the injector and column oven, are inside or attached to the inner box. The inner box is wrapped with 2 inches of high temperature rockwool insulation to improve heat retention, and is supported by 2 inch long ceramic spacers on the bottom side of the assembly. The 2 inch insulation region greatly reduces the amount of heat lost to the environment and keeps dangerously hot parts away from places where a user could touch them.

The inner box is placed off-center in the large outer box to allow for a segmentation of systems inside the instrument. One half of the large outer box is taken up by the chromatography equipment inside the inner box, while the other half is taken up by carrier gas flow controllers, electronics, and the detector assembly. These components are temperature sensitive and rely on the insulation to remain cool and avoid thermal drift.

The entire chassis, down to the rivets that hold it together, is made up of 6061 aluminum stock. This aluminum alloy specifically was chosen because of its unique blend of malleability, weight, cost, and ability to be drilled and machined with basic equipment. Other materials like carbon steel are too hard to effectively be machined using hand tools and would produce a product that is unnecessarily heavy. Glass-reinforced thermoset plastic was also considered, but was determined to be too expensive compared to the metal options. When the instrument is plugged into the wall outlet, the all metal chassis is grounded to prevent it from ever becoming live during a malfunction. Figure 4.1 shows the final prototype with the top removed so that the nested box construction is visible.

4.1.2 Carrier Gas Control

The carrier gas control hardware was selected to be compatible with the three carrier gasses that would likely be tested: nitrogen, helium, and compressed air. A Porter Instrument (Hatfield, PA) model 8311 inert gas pressure regulator was selected for this purpose. It can regulate accurately within the range of 5 to 100 PSI, allows for user servicing if anything ever needs to be replaced, and has an additional attachment point allowing a gauge to be connected to the column-pressure side of the regulator. Figure 4.2 shows this regulator.

Aside from the regulator, instrument grade stainless steel tubing and Swagelok stainless steel

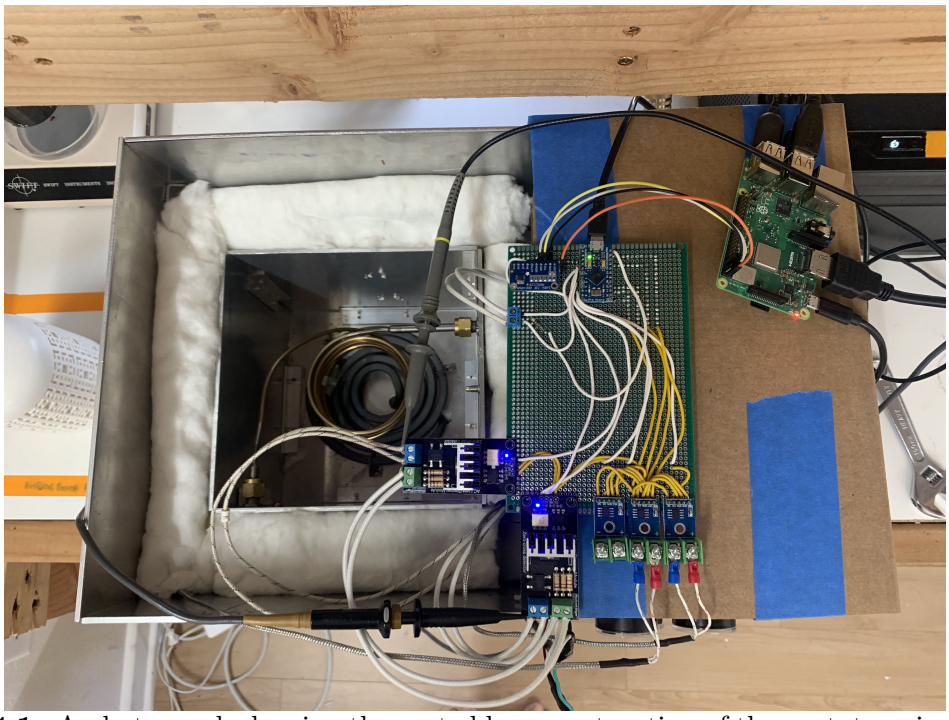


Figure 4.1: A photograph showing the nested box construction of the prototype instrument.

tubing connectors were used for all gas routing. This is standard equipment in high-end instrumentation and may have even been overkill for this application. It was chosen to eliminate any chance of fault due to a failure of basic hardware.

In practice, an input carrier gas pressure anywhere between 5 and 100 PSI could be connected to the carrier gas input port on the back of the instrument. Note that a high pressure gas cylinder can often contain pressures as high as 2000 PSI and that an external regulator must be used to connect a gas cylinder to the instrument safely. Once the carrier gas is connected, the pressure applied to the column can be set using the Ported pressure regulator shown in Figure 4.2. For convenience, pressure indicators are built into the instrument and can tell the operator both the pressure at the carrier gas input port and the pressure at the head of the separation column.

4.1.3 Heating Element Control

In addition to managing the carrier gas, the instrument described in this report must also be able to accurately measure and set the temperature of multiple parts of the instrument simultaneously. To do this, two thermocouples were connected to an Arduino Nano using a MAX6675 thermocouple controller and configured to communicate using a simple 4MHz SPI communication protocol. Every 5 seconds, the Arduino polls the two temperature sensors and records the current temperature at each one.

Based on the recorded temperature readings, the Arduino then manages the triggering of two TRIAC circuits to set how much power can be drawn by the heating element. This is done using a zero cross detector, which outputs a pulse when the 60 Hz AC from the wall outlet crosses zero. The Arduino detects this pulse using a state-change interrupt and records the

Figure 4.2: A photograph showing the Porter regulator selected for carrier gas control.

zero crossing time. In the main Arduino loop, the system clock is constantly checked and compared to the most recent zero crossing. After a delay, the Arduino triggers the TRIAC circuit and the TRIAC becomes conducting until the polarity of the line AC flips. This process is repeated twice for every full cycle of a 60 Hz AC voltage, once on the positive half of the sine wave and once on the negative half. The length of the delay between zero crossing and TRIAC triggering determines how much of the AC signal is conducted through the TRIAC. An example waveform can be seen in Figure 4.3. Because of the odd shape of the waveform, the power applied to the heating elements is recorded and discussed here in terms of RMS voltage. No delay between zero crossing and TRIAC triggering allows for practically the full 120 V RMS signal to be applied to a heating element. On the other hand, a full delay means that the TRIAC is never triggered and will not conduct at all (0 V RMS). During normal instrument operation, a simple PID loop is used to balance the measured temperature of each component and compare it to the RMS voltage that its heating element was receiving. This allowed for rapid instrument warm-up and "set and forget" temperature management.

4.1.4 Detector Operation

As previously mentioned in the section discussing detector options, the prototype instrument described in this report uses a diode detector biased with a constant current of approximately 50 mA. This current was chosen to ensure that the diode operates in the temperature range where its response to temperature changes is roughly linear. A relatively small current of 50 mA also makes the signal less prone to noise fluctuations and avoids self-heating of the detector. During normal operation, the detector dissipates approximately 35 mW continuously.

The voltage between the detector and ground, which fluctuates depending on mixture of

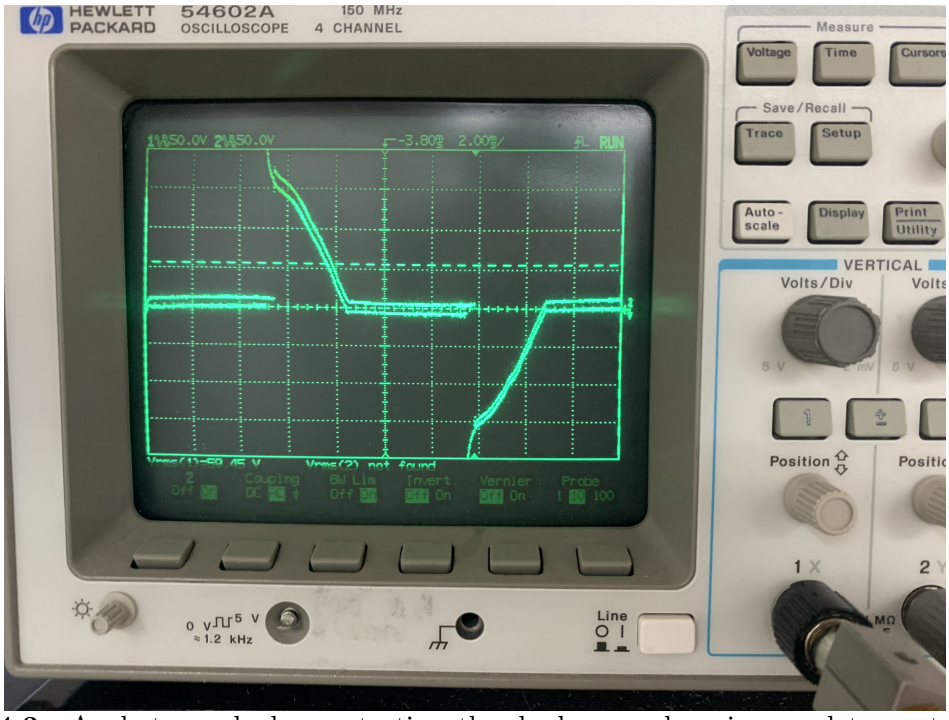


Figure 4.3: A photograph demonstrating the dual wave chopping used to control power applied to the heating elements.

gas/eluant exiting the column, is measured using a 12-bit analog to digital converter with an input resistance of over 1 megaohm. Connection of the ADC is not believed to significantly affect the detector circuit or cause any adverse loading effects.

4.2 Software

In order for students to use the GC, it needs to provide an intuitive and simple graphical user interface (**GUI**). This GUI will allow the user to initialize the GC, run samples, perform operations on the collected data, view previous tests, and save data in appropriate formats. Additionally, this GUI will run on a Linux based Raspberry Pi. A decision matrix was generated to consider the design requirements and is given in Figure 4.4. The GUI will be written in wxPython, compiled to a Python executable for speed, and run automatically upon start up via a bash script. Although wholly separate, the GUI will be written concurrently with the GC development.

Gas Chromatography Capstone	Decision Matrix		Graphical User Interface																	
Conor Green & Matt McPartlan																				
November 24, 2019																				
		*Scores are out of 5																		
		C				**Java**				**Python**										
Design Criteria	**Weight (%)**	C#		Visual Basic		AWT		Swing		Qt		TkInter		Pygame		PyGtk		wxPython		
Cross-platform compatability	40	1	0.4	1	0.4	3	1.2	3	1.2	5	2	5	2	1	0.4	4	1.6	5	2	
Compilable	5	5	0.25	5	0.25	5	0.25	5	0.25	2	0.1	2	0.1	1	0.05	5	0.25	3	0.15	
Speed	20	5	1	4	0.8	1	0.2	1	0.2	2	0.4	2	0.4	2	0.4	3	0.6	3	0.6	
Codability	10	3	0.3	3	0.3	2	0.2	2	0.2	4	0.4	4	0.4	5	0.5	3	0.3	3	0.3	
Range of abilities	25	5	1.25	5	1.25	4	1	4	1	4	1	4	1	3	0.75	5	1.25	5	1.25	
	Total:		3.2		3		2.85		2.85		3.9		3.9		2.1		4		4.3	

Figure 4.4: GUI Programming Language Decision Matrix

A software suite was written in Python and C++ (Arduino specific libraries) to accomplish the listed goals. The code integrates the TRIAC heating, thermocouple feedback, analog voltage measurement, and graphical user interface. A Raspberry Pi 3B+ running Rasbian Buster integrated the aforementioned components in various scripts. The TRIACs and thermocouples were treated as a subsystem, controlled by an Arduino Nano, that communicated to the Raspberry Pi through serial communication, using an original messaging system given in the *GCController* library. The analog voltage measurement was accomplished through the ADS1115 analog to digital conversion module that communicates over SPI and I^2C using TTL GPIO pins. The GUI given has sufficient functionality to analyze substances with gas chromatography. The software was configured to run automatically upon booting up the Raspberry Pi, providing a completely hands free, educational experience for the user. An authorized user (e.g. teacher's assistant) can modify the YAML file to change defaults without modifying source code. The source code is given on the Github[1]and Appendix C. The software written has/can

- Automatically start on boot up.
- Separate YAML configuration file for end users to change default settings without modifying source code.
- Threaded data collection and live plotting.
- Math operations
  - Integrate data (voltage)
  - Normalize data (voltage)
  - Clean the time axis
  - Apply a low pass filter
  - Calculate relative area of peaks and label on plot.
- Fill under data on plot.
- Threaded temperature feedback text box updating.
- Ability to set oven and injector temperatures.
- Save current session as ".gc". file type, containing JSON data structure.
- Save current plot as JPEG or PNG image.
- Open previous session files (".gc") into the current window.
- Pydocs documentation

An example of the graphical user interface after testing by measuring a voltage divider manually adjusted is given in Fig. 4.5. The demonstration shows data collection, applying filters, normalizing, determining the area of peaks, and filling the figure, while performing necessary background operations such as temperature adjustment.

A diagram of the scripts is given in Fig. 4.6, which shows the four important threads (discussed in subsequent sections). The four scripts *gas_chromatography.py*, *gc_gui.py*, *gc_class.py*, and *GCController.ino* run on two micro-controllers/processors that drive the ADS1115, TRIAC heating, and thermocouple hardware modules.

[1]https://github.com/cgreen18/Gas-Chromatography

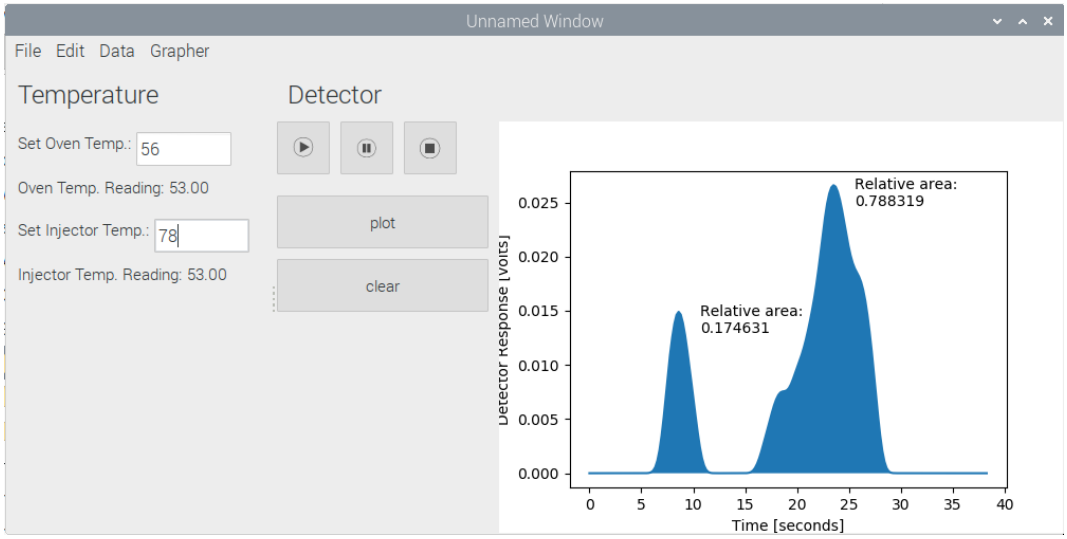


Figure 4.5: Example Operation of Software Suite

4.2.1 Arduino to Raspberry Pi Serial Libraries

The *USB_COMM* library was written to provide a common format for serial communication between the Raspberry Pi and Arduino. To transmit, both devices generate strings with a command code and three numbers and encode them as byte strings. To receive, the devices parse the byte string into the command and attached data and performs the indicated operation. The Arduino is responsible for measuring temperature, responding to temperature queries, and setting the TRIAC output power when sent commands. The Raspberry Pi periodically queries the temperature, updates the text boxes as necessary, and sends commands and values upon user input to the writable text boxes.

4.2.2 Analog Voltage Measurement

The Raspberry Pi must query the ADS1115 module for the measured voltage periodically (according to an internal sampling interval parameter) and plot these values concurrently. The script *gc_class.py* accomplishes measurement and data retention through the *GC* Python class. This class provides all the functionality necessary to measure voltages and perform operations on the measured data. The *GC* class is used by the script *gc_gui.py* by various classes to collect data, display it, and respond to user inputs. Concurrency was achieved through threading and separate (but synchronized) data structures. Both the *GC* and *GCFrame* classes maintain a copy of the current data that are updated to match periodically. The separation allows the software to measure voltages at a precise sampling frequency, plot when the CPU has available time, and allow the user to command program flow all at the same time.

4.2.3 Graphical User Interface

Considering the intended users are chemistry laboratory students, a simple, intuitive graphical user interface is required. As seen in the example in Fig. 4.5, there are two main panels: "Temperature" and "Detector". In the temperature panel, the TRIAC setting and feedback

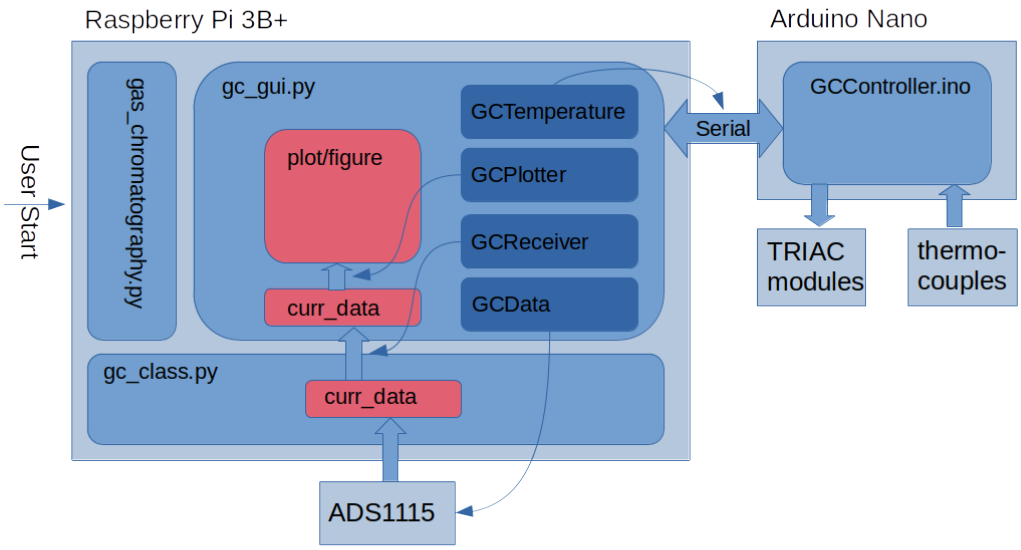


Figure 4.6: Diagram of the Software Developed with Threads Labeled.

is very clear to the user. The user can enter values in a text box (that automatically parses the value and communicates to serial). Automatically, the measured temperatures of the oven and injector are displayed. In the detector panel, the five available buttons are "play," "pause," "stop," plot, and clear, which begin/pause/end data collection and plot/clear the current data on the figure. Additional functionality beyond the most basic GC operation is provided in the top menu bar. The four categories of operations are: "File," "Edit," "Data," and "Grapher." The operations of the menu options are

- "File" menu allows the user to save and open .gc file types as well as exit the program gracefully.
- "Edit" is not yet connected but is intended to allow the user to manually adjust visual parameters in later versions.
- "Data" allows the user to manipulate the data by subtracting the initial time, normalizing, calculating the area, and apply a low pass filter. The user can also set the current data set to the most recent previous set.
- "Grapher" lets the user fill under the curve and label the area of peaks.

4.2.4 Threading

Considering the high CPU load of the features discussed, the program required threading to properly handle the load. Furthermore, graphical user interfaces basically require threading

to properly handle event and user interaction while performing background processes. Inheriting the *Thread* Python class, the four classes, *GCTemperature*, *GCPlotter*, *GCReceiver*, and *GCData* were written to handle temperature feedback value display, live plotting, data reception, and data collection, respectively. The thread classes and their important interactions with other parts of the code are given in Fig. 4.6. The most notable aspect of this design is how shared memory was handled. There were three objects/arrays shared between threads and/or the main process: the serial connection, current data array of the *GC*, and current data array of the *GCFrame*. The serial connection was protected through a threading *lock* shared between the temperature thread and main frame. The data array of *GCFrame* was protected through another *lock* shared between the receiver, plotter, and main frame. These locks are asynchronous, which lets the temperature text feedback and plotter wait until more important GUI events and data collection functions are handled. The data array of *GC* was protected through a threading *condition*, which is a lock with notification procedures to keep *GCData* and *GCReceiver* in sync, rapidly collecting data with appropriate time intervals between points.

4.2.5 Documentation

Throughout the development cycle of the software, a Github[1] was maintained. The Github contains all the source code as well as instructions, descriptions, previous versions, and deliverables for the capstone course.

Environment Libraries and Dependencies

The software written relies on the libraries: wxPython, ADS1x15, Virtual Environments, and various smaller modules. The exact versions used and installation procedures were documented in the "Installation" folder of the Github repository[1]. This documentation will be useful for future developers of this technology: the installation procedures describe all the steps necessary for reproduction. All the steps required to set up the environment were written in a bash script *install.sh* that a user can run on a new Raspberry Pi to get it ready to perform GC operations. Furthermore, a second bash script *config.sh* was written to configure the program to run on startup (great for lab environments). Within the Python code, Pydocs commenting style was followed. HTML files that describe every method and import are given on the Github.

Versions

In order to test the physical components while modifying the source code to potentially broken states, stable versions were saved as branches on the Github[1]. In this manner, the Github stays clear of extraneous files while providing stable versions to test hardware. Versions 2.0/3.0 and 4.0 are stable and are available in branches "contains-v3-of-GUI" and "Version_4.0", respectively.

[1] https://github.com/cgreen18/Gas-Chromatography

4.3 Design Impact

Though gas chromatography is one of the most well established and widely used techniques in analytical chemistry, there is a notable lack of instruments targeting the low-end market. A gas chromatograph is an inherently simple instrument that requires only a handful of specialized pieces: an injector block, a separation column, and a detector assembly. Almost everything else on a gas chromatograph can be assembled (or approximated) using standard gas routing lines and Swagelok connectors. The last major component, an electronic heating system, is a well documented and simple system as well.

Nearly all modern instruments have lost this simplicity, and affordability, as they compete with other instrument manufacturers for lower detection limits and more versatility. Features like cryo-traps, FID detectors, and other complicating factors are amazing innovations, but they are not needed to analyze the success or failure of classroom organic chemistry synthesis reactions. The instrument system outlined in this report is designed to address this need for simple and durable instruments that can meet the basic needs of an organic chemistry teaching lab. The low cost and durability of a diode detector makes it an ideal solution to this problem. The hermetically sealed glass package is resistant to oxidation and degradation that reduces the performance of a traditional tungsten filament over time. The diode detector is also uniquely capable of tolerating nearly any type of carrier gas, including unfiltered compressed air. Despite this versatility, it should be mentioned that a high quality carrier gas will greatly improve detector performance.

5. Development History

The proposed gas chromatography instrument was completed over the 2019-2020 academic year as part of the capstone project graduation requirement for the Electrical Engineering Program at LMU. An encompassing schedule for the project is given in Appendix A. As a team project, the roles and responsibilities of each partner is important in completing a high-quality product. These individual roles are given in Appendix B. Though great detail regarding the construction and operation of the final product was described in a previous section, that was not the first attempt that was made. A total of three prototypes were made, each addressing the shortcomings of the last, until the previously described product was finished. This section outlines the evolution of the three most significant development milestones that were reached *before* the finished product was built. These initial prototypes are labeled MkI - MkIII, in the order that they were completed. Each Mk prototype played a critical role in the development of the finished product, even if parts of them were not directly used.

5.1 Initial Prototype (MkI)

The first test platform (MkI), which had been in use for over a year, was a completely analog Gow-Mac Series 350 that had its original tungsten coil detectors replaced with the the new detectors being tested. No other changes were made to the instrument or column. While the MkI prototype was acceptable for proof of concept, it was limited by a few significant drawbacks:

- A lack of precision temperature control. Setting the temperature on the MkI and allowing it to reach thermal equilibrium can take between 1 and 5 hours.
- A lack of thermal protection. The detectors examined all have an absolute maximum temperature rating of about 150°C, and the solder in the detector prototype melts at about 180°C. It is very difficult to ensure that the maximum temperature is not exceeded using the analog triode for alternating current (**TRIAC**) controls installed on the instrument.
- Inaccurate temperature monitoring. Even if the TRIAC controls were set and forget, the analog thermocouple monitoring circuitry that displays the temperature to the user was not reliable. This compounds the previous problems.
- An excess of electronic and mechanical parts that do not reflect the vision of the project going forward. Extra regulators, gas routing equipment, and vestigial amplifier circuitry are all present within the MkI test platform, making it difficult to work on when changes need to be made. Additionally, the hazardous oven insulation is problematic and must be interchanged with a more friendly ceramic fiber option.

These shortcomings, particularly those related to temperature control, at one point caused the detector assembly to overheat and destroy the prototype. Because rebuilding the detector module in the MkI required a complete disassembly of the test platform, this was a significant setback and inspired the design of a second generation test platform.

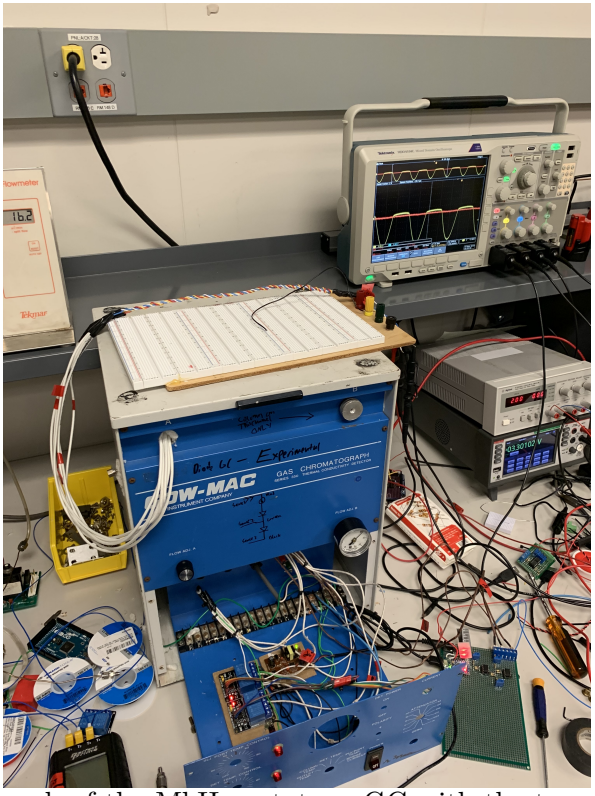


Figure 5.1: A photograph of the MkII prototype GC with the temperature control boards exposed.

5.2 Second/Third Prototypes (MkII/MkIII)

The MkII test platform, which was still based on the Gow-Mac Series 350 frame, addressed the temperature control problems by using a microprocessor to continuously monitor the instrument components and determine when the heating element needs to be turned on using mechanical relays. This system was used to automatically set the temperature of the injector, column oven, and detector. Additionally, the attachment point for prototype detector assemblies was moved to an easily accessible position to speed up the process of modifying the detector.

After verifying the functionality of the MkII prototype, development began on an original test frame that incorporates the features of the MkII into a significantly smaller frame with more powerful heaters and better electronics (MkIII). The MkIII prototype, which can be seen in Figure 5.2 and 5.3, is significant because, with the exception of the injector blocks, all components and parts were designed and machined independently. This represents a significant step forward towards the eventual goal of delivering custom-built finished products.

A comparison of the MkII (Figure 5.1) and MkIII (Figures 5.2 and 5.3) reveals the significant change between these two platforms. At the end of the Fall 2019 semester, the MkIII was the most advanced prototype developed as part of this project.

During the Spring 2020 semester, several problems became apparent during the testing of the MkIII prototype. Most significantly, the frame was too small effectively insulate the hot

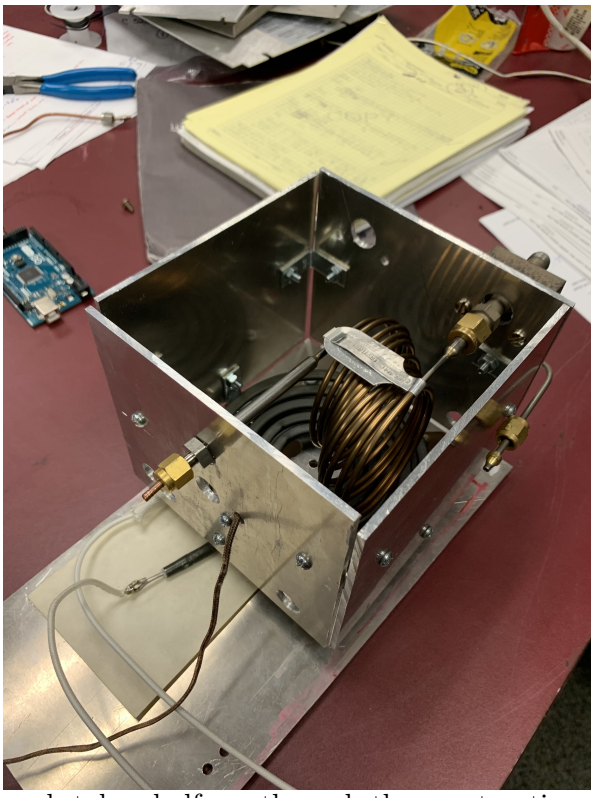


Figure 5.2: A photograph taken halfway through the construction of the MkIII prototype that shows a new column and more powerful heating element installed in the oven.

oven and injector from the cool electronics and flow control equipment. Additionally, it was much too cramped to swap basic pieces without completely disassembling the instrument. Size and design simplification are the primary differences between the final product and the MkIII prototype. The final product is roughly 4 times larger (in terms of volume displaced on a desk) than the MkIII prototype.

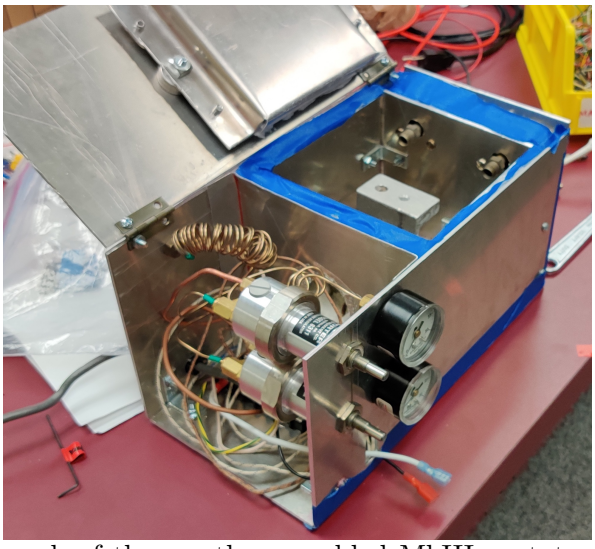


Figure 5.3: A photograph of the mostly assembled MkIII prototype. The blue tape is in place to temporarily contain the oven insulation and must be removed before any heating tests are performed.

5.3 Software

As discussed in Chapter 4 on Design, a multi-language software suite was developed to perform gas chromatography on the hardware. The software was tested extensively through data collection and manipulation in normal operations and at edge cases. With over 2,000 lines of code, the code is too complex to manually verify operation; however, through extensive testing, the current version has not failed to perform the intended operations. Largely, this is in part due to the slow version creation of the process. Versions were tested and saved until the current fourth version, which has minimal differences from the mostly stable third version. Any future bugs or errors will be addressed and patched live on the Github.

5.3.1 Summary

A gas chromatograph is a complicated system, and it is important to use good engineering practices to minimize the amount of wasted time. Following this test procedure is one potential way to individually verify the functionality of small design modules without putting the entire project together and discovering that it does not work. In addition to making problems easier to find and fix, there is a tremendous amount of time that can be saved if it is not necessary to disassemble the instrument completely whenever anything goes wrong. To help visualize the progress made throughout the semester, Figures 5.1 - 5.3 are included in this report.

6. Testing and Data Analysis

6.1 Test Setup

Parameter	Setting	Stabilization (min)	Verification
Injector Temperature ($°C$)	250	30	Thermocouple
Oven Temperature ($°C$)	150	30	Thermocouple
Detector Temperature ($°C$)	75	30	Thermocouple
Array Temperature ($°C$)	150	30	Thermocouple
Detector Drift (mV/min)	10	2	Tektronix DMM
Carrier Gas	Compressed Air	-	-
Carrier Flow (mL/min)	25.0	1	Tekmar Flowmeter
Column Packing	Carb	-	-
Column Length (m)	6	-	Tape Measure
Injection Volume (uL)	50	-	uL Syringe

Table 6.1: Instrument settings used for a comparison of different detector designs. All detector comparisons shown in this report were collected under these operating conditions.

6.2 Instrument Functionality Testing

6.2.1 Systems to be Tested

- **Oven heater test plan:** A large stove-top heating element driven by 120 VAC from a wall power supply will serve as the primary oven heater. This will be tested individually to verify that it can safely operate before being installed into the test unit. To test the heating element individually, power will be applied from the wall outlet and the heating element will be left on for 10 minutes while being constantly monitored. This will be repeated 10 times to ensure that the newly formed heating element is can be temperature cycled without issue.

- **Injector heaters test plan:** The injectors, due to their much smaller size and power requirements, can be heated with small 100W cartridge heaters. There are two injectors that will be tested independently before being installed into the test instrument. To test the heating element individually, power will be applied from the wall outlet and the heating element will be left on for 10 minutes while being constantly monitored. This will be repeated 10 times to ensure that the newly formed heating element is can be temperature cycled without issue.

- **Temperature control system test plan:** All of the aforementioned heating elements will be continuously monitored and controlled using a mix of thermocouples (for small spaces) and thermistors (for large spaces, like the oven). This temperature monitoring and feedback system must be thoroughly tested before being allowed to operate with the real components installed. The functionality of the temperature feedback and control system will be tested by applying a soldering iron of known temperature to the temperature sensors. The hot soldering iron, when applied to each temperature detector individually, should be registered as an over-temperature condition and lead to device shutdown.

6.2.2 Review of System Test Results

Trial	Rise Time (min)	Set Temp ($^{\circ}C$)	Stable Temp ($^{\circ}C$)
1	9.33	100	99
2	9.25	100	98
3	8.53	100	100
4	9.44	100	97
5	9.32	100	101
6	9.12	100	102
7	9.67	100	99
8	9.98	100	100
9	9.54	100	101
10	9.45	100	98

Table 6.2: Oven heater and temperature stabilization test. All trials began with the instrument resting at room temperature (20°C) and proceeded as described previously.

Trial	Rise Time (min)	Set Temp ($^{\circ}C$)	Stable Temp ($^{\circ}C$)
1	17.23	200	191
2	17.22	200	192
3	17.15	200	190
4	18.82	200	197
5	17.60	200	193
6	17.45	200	192
7	17.32	200	195
8	17.34	200	190
9	17.78	200	189
10	17.50	200	192

Table 6.3: Injector heater and temperature stabilization test. All trials began with the instrument resting at room temperature (20°C) and proceeded as described previously.

Trial	Shutdown Time (s)	Peak Recorded Temp ($^{\circ}C$)	Sensor
1	7.5	311	Oven
2	8.3	305	Oven
3	10.5	322	Oven
4	6.2	302	Oven
5	9.3	319	Oven
6	25.2	302	Injector
7	25.7	303	Injector
8	24.8	300	Injector
9	25.1	304	Injector
10	24.9	301	Injector

Table 6.4: Over-Temperature safety cutoff test results. All trials began with the instrument stabilized at a normal operating condition (250°C detector, 150°C oven) and proceeded as described previously.

6.3 Instrument Performance Testing

When comparing numerical data, such as chromatograms, it is important to use an objective technique to compare results and make a decision based on quantifiable results. Continuously using the conditions outlined in Table 6.1 will standardize the performance of each detector system and allow for meaningful comparisons to be made. All trials use the same mixture of 40% ethanol (V/V) and 60% water (V/V).

1. **Signal-to-Noise Ratio:** Signal to noise ratios (SNR) are one of the most fundamental and important metrics for checking the performance of an analytical system. Evaluation of the SNR under various operating conditions can allow determination of several critical instrument parameters. The limit of detection (LOD) is one example of such a parameter that can be defined in terms of the SNR. A high SNR indicates that the detector is responding well to changing analyte conditions and that there is little chance that the signal is occurring due to random chance. This project will use the MATLAB signal processing toolbox to numerically calculate the SNR for all samples once collection of detector comparison data begins in the Spring 2020 semester.

2. **Drift Rate:** All detectors, even those in expensive analytical instruments, have some degree of drift in the output of the detector. The detectors tested in this project must have a maximum drift of 10 mV/min to ensure a good quality baseline for integration and determination of peak areas. Baseline drift can be fixed somewhat in software, but the most reliable results come from developing good hardware from the beginning. To check the output drift from each type of detector, all amplifiers will be disconnected and the output of the sensor will be directly connected to the Tektronix benchtop DMM available in lab. The change in DC voltage for 2 minutes will be measured and must be less than the maximum of 10 mV/min.

3. **Limit of Detection:** The limit of detection (LOD) refers to the smallest amount of analyte that can elicit a measurable response from the detector. This can be difficult to determine qualitatively without the use a calculated SNR that can be compared to a threshold. In the Spring 2020 semester, the LOD for various common organic compounds will be determined for all detectors explored here. To do this, samples will be run with a constantly increasing concentration of analyte impurity until it is possible to detect the impurity.

To experimentally determine the signal to noise ratio (SNR) and the baseline drift, 50 uL of a test ethanol and water mixture was injected into the instrument when it was configured as shown in Table 6.1. This was repeated three times. The collected results are shown in Table 6.5 and an example chromatogram is shown in Figure 6.1.

Trial	Baseline Drift (mV/min)	SNR
1	-5.5	5.5
2	-7.3	5.3
3	-6.1	4.5

Table 6.5: The results of three SNR and baseline drift assessment experiments.

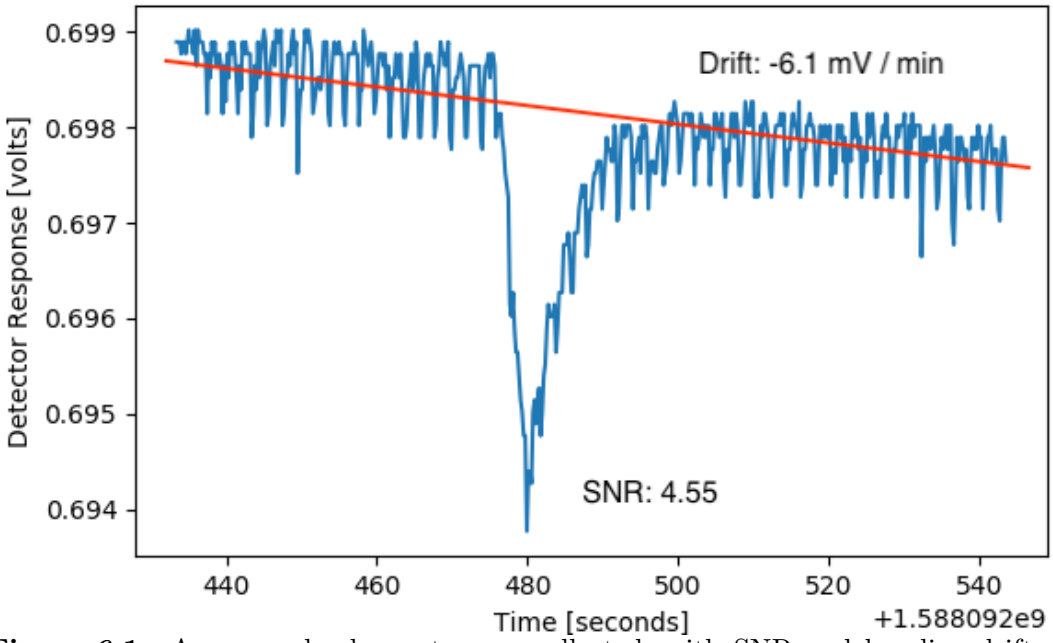


Figure 6.1: An example chromatogram collected, with SNR and baseline drift displayed over the graph.

6.4 Data Analysis

Though collection of chromatographic data is the primary focus of this project, the true value of the data can only be realized through further analysis. After collecting a sample chromatogram, the signal to noise ratio (SNR) must be calculated to determine the quality of the collected data. The signal to noise ratio for each detected peak must be checked to ensure that the peak is significant. Generally, a signal to noise ratio over 4 indicates that the peak is a real signal being picked up from the detector. For the chromatograms shown in this report, the signal to noise ratio was calculated graphically. Two baseline-drift corrected lines were drawn to mark the upper and lower limit of the background noise and the vertical distance between those two lines was recorded as the baseline noise amplitude. In the chromatogram shown in Figure 6.1, this amplitude is approximately 0.00100 mV. After determining the baseline noise amplitude, the difference between the average drift-compensated baseline noise and the magnitude of the peak was measured by marking another line at the most extreme inflection point of the peak. The vertical distance between the most extreme inflection point and the baseline noise represents the signal magnitude. In the example shown in Figure 6.1, this magnitude is approximately 0.00445 mV. The signal to noise ratio is calculated by dividing the two numbers, yielding a SNR of 4.55 in this example.

7. Ethical Considerations

As a technical instrument for chemistry, the ethical implications focus on copyright, conflicts of interest, and safety. The Institute of Electrical and Electronics Engineers (**IEEE**) was formed in 1963 and has since defined the moral responsibilities and considerations for electrical engineers world wide in the IEEE Code of Ethics. This code of ethics defines ten ethical guidelines, which apply to the GC project differently. The important ethical considerations are safety, disclosure of limitations and honest representation of results, and conflicts of interest.

7.1 Safety

The proposed gas chromatograph heats elements to excess of 200°C and expels unknown substances and gases freely, both of which affect public health. The high temperatures are necessary and as such, must be handled properly through insulation and a feedback control loop. The carrier gases, helium or nitrogen, are completely safe for human contact; however, any substance can be tested and introduced into the chromatograph. The high temperatures are necessary for operation and as such, the instrument must insulate the user from the heating elements and hot spaces. The current chromatograph uses fiberglass insulation, properly sealed, that assures temperatures outside the instrument do not reach unsafe levels. Assuming the temperature does not exceed expected levels, the insulation will make the instrument safe. In order to assure the temperature does not exceed the desired temperature, a digital feedback loop was implemented. Tested thoroughly, a microcontroller measures the temperature and applies power to the heating element to achieve the correct temperature.

7.2 Honest Disclosure of Limitations

Considering the Chemistry Department intends on buying three *accurate* gas chromatographs, bias must be addressed. The instrument is designed to be economic and educational and will sacrifice accuracy in order to achieve these goals. In sacrificing accuracy, the Chemistry Department must be honestly informed of limitations before sale. The project managers will not let bias interfere with good scientific product development.

7.3 Conflicts of Interest

There are possible conflicts of interest within the Seaver College of Science and Engineering. While this capstone project is rooted in the Electrical Engineering Department, it also draws funding from the Chemistry Department. Each department may desire separate goals. As electrical engineering students, the managers of this project must first serve the goals and deliverables of the Electrical Engineering Department and consider the goals of the Chemistry Department second.

7.4 Operating During the COVID-19 Pandemic

On 13 March 2020, the President of Loyola Marymount University, Dr. Snyder, extended online instruction to the entirety of the Spring semester. As such, the physical capstone project to construct a gas chromatography instrument was derailed. Fortunately for the

efficacy of the project, physical construction has continued from home; however, the ethical considerations of operating and engineering during a pandemic must be considered. The physical construction of the device is taxing on logistics that could otherwise combat the pandemic and required physical contact between the team members.

In order to construct the instrument, components (e.g. air compressor) needed to be ordered online, putting strain on critical supply and logistic lines of various health operations around the nation. Furthermore, sourcing components from online sources requires more hours from essential workers that are already overburdened. Despite these concerns, the continued development of this project is ethical because online shopping is the least taxing of available options and hopefully presents a minor influence. With a minor disturbance in supply lines comes the immense benefit of completing our educations and contributing to chemical sciences, education, and engineering. Earning a Bachelors of Science in an engineering field is a certification of engineering ability and to not complete the capstone project would be an infringement on that certification.

In continuing the development of a physical gas chromatography instrument, we met in person to work and test, which presents a potential source of infection for either one of us. Without the project requiring physical contact, we would not have otherwise seen each other in person. Despite concerns, again the educational benefit and minimal risk make physical contact worth the risk to ourselves. Both team members do not pose significant risks to others if infected.

8. Contribution to ABET

8.1 ABET

A non-profit, non-governmental agency, ABET has applied objective criteria to accredit college and university programs in the disciplines of applied and natural science, computing, and engineering at undergraduate and graduate levels. ABET draws experts from 35 member societies to apply industry knowledge to the accreditation process. In 1997, the Engineering Criteria of 2000 (**EC2000**) was adopted to focus on that students *learn* as opposed to what they are *taught*. Accrediting 4,144 programs in 32 countries, ABET focuses on pillars as given in the guidelines. ABET provides manuals for every academic year outlining the criteria used to assess and eventually accredit a program. The guidelines discussing student outcomes and institutional support are most pertinent to this capstone project [5].

8.1.1 Student Outcomes

ABET states that students enrolled in an accredited institution must learn the appropriate tools and skills necessary to solve engineering problems. The proposed project on gas chromatography poses a complete and challenging engineering design and management problem. The project is doing well both in terms of goals and schedule. As such, the students are demonstrating that they have *learned* adequate skills. The project covers topics from analog circuits, digital feedback systems, software, etc. in depth. These learning experiences will prove invaluable to a career in engineering.

8.1.2 Institutional Support

In Criterion 8 of the manual, ABET states that, "Institutional support and leadership must be adequate to ensure the quality and continuity of the program." LMU is providing adequate support through both the Electrical Engineering and Chemistry Departments in this cross-curricular project. The equipment and materials used in gas chromatography are expensive and as such, require institutional funding. LMU has properly funded the project, allowing uninhibited learning.

8.2 LMU Values

The greatest contribution of this project is three GCs specifically designed for educational use for the LMU Chemistry Department. After exploring the various GC designs, the most robust and simple design that still yields sufficiently accurate results will be created into three products. Not only will these three GCs reduce upfront and service costs but also allow chemistry students to operate the instruments themselves, greatly improving educational value.

Furthermore, the collaboration between electrical engineering and chemistry may facilitate further cross-departmental projects. Bridging the gap between the two allows this project to garner more funding and technical expertise. The budget provided by the Electrical Engineering Department can be spent on prototyping and exploring the best detector configuration. After experimenting, the chemistry department can provide the funding to create three complete GC products. Beyond the economic benefits of cross-departmental collaboration, the educational resources are increased as well. As expected, there are mentors available to help

manage a capstone project as well as for technical advice on electronics. On the chemistry side, there are professors available to help analyze chemical problems and offer technical advice. For example, Dr. Robert Senter, has been able to identify previously unknown errors in the GC system that were unknown to the students. The chemistry knowledge available in collaborations has proven invaluable. The benefits of collaboration cannot be denied and hopefully, future electrical engineering capstone projects can take advantage of these benefits, whether in collaboration with chemistry or another Seaver department.

9. Demonstration

Though much has been said about the importance of the carrier gas and a properly designed column, the best way to demonstrate this is through a demonstration of a bad chromatogram and an analysis of how to improve it. Figure 9.3 shows a poor quality chromatogram collected from a two-component mixture of 2-propanol and ethyl butyrate. Though two peaks corresponding to the two constituent components are present, they are poorly resolved and extremely noisy. The noise problem can likely be fixed electronically, but the non-Gaussian shape of each peak indicates that too much sample is being fed through the column. Gaussian peaks indicate that separation is occurring properly and that coelution, when multiple components leave the column at the same time, is not a problem. Quantitative analysis of peak area, a critical tool used to determine the concentration of each compound within the mixture, is not possible if coelution has occurred. Ordinarily, the amount of sample being injected would be reduced, but this is not possible when using relatively low-sensitivity diodes and a nitrogen carrier gas. To get around the problem and allow for the collection of properly resolved chromatograms like those shown in Figure 9.4, it was decided that a larger high-throughput column was required. This column, which will likely be made of 1/4" stainless steel tubing packed with silica beads, will be designed to handle injection volumes up to 100uL. This column should provide a significantly higher capacity than the 1/8" column used to collect the images in Figures 9.3 and 9.4. The construction and testing of the high-throughput columns described here will be conducted early in the Spring 2020 semester.

9.1 Physical Prototype

A physical prototype was constructed using riveted aluminum sheets with perforations for necessary seals/connections and wires and an oven heating element as seen in Fig. 9.1. Though there is currently insulation separating the oven chamber from the exterior shell and empty compartment, the electronics were not put into the shell to avoid unresolved thermal issues in the untested version of the oven.

The instrument and software was tested in the lab setup given in 9.2 by using dessicated air for the carrier gas and introducing an alcohol/water solution through the injector. Before data collection, the temperatures of the injector and oven were set to 80 C°through the GUI. The GUI indicates the measured temperature of the oven and injector and when both reached the desired temperatures, a sample was taken. The

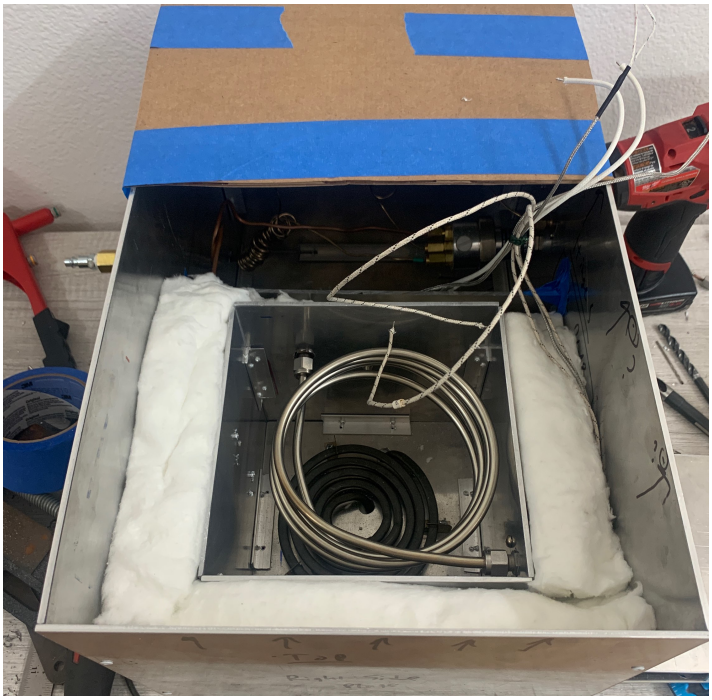


Figure 9.1: Disassembled Gas Chromatography Instrument

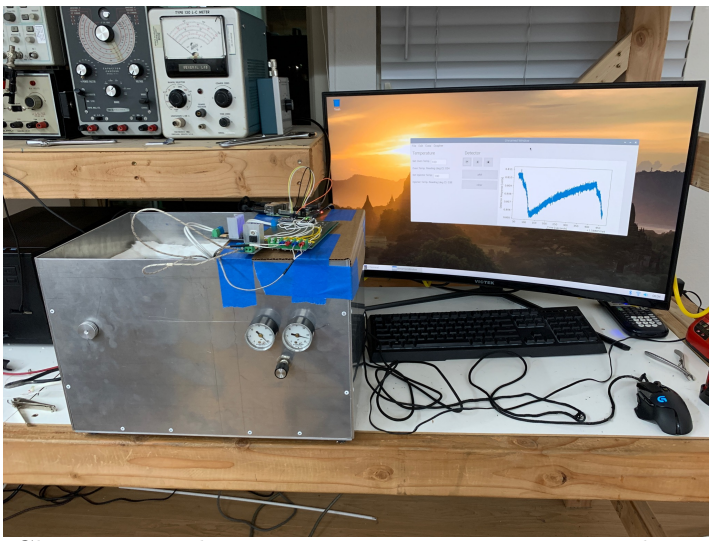


Figure 9.2: Gas Chromatography Instrument Measuring Data and Interfacing with User

Figure 9.3: An unamplified and unfiltered chromatogram (10uL injection) collected using a single-sided 1N4148 diode detector module with a nitrogen carrier. Note the peak distortion and misshapen area under each peak compared to the proper chromatogram shown in Figure 9.4.

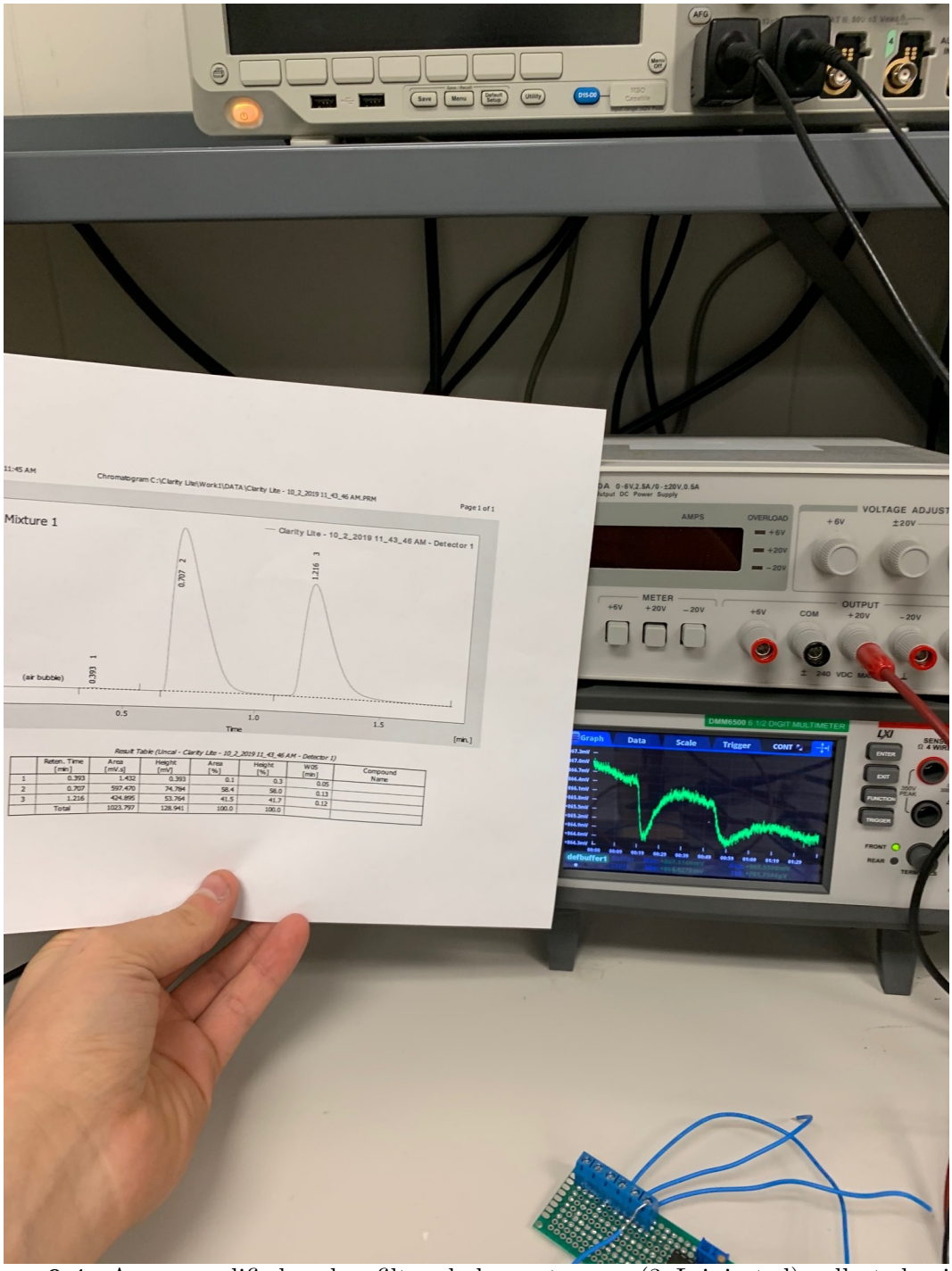


Figure 9.4: An unamplified and unfiltered chromatogram (3uL injected) collected using a single-sided 1N4148 diode detector module with a helium carrier. Note the roughly Gaussian peak shape and relative areas under each peak compared to the true chromatogram shown on the printed paper.

10. Conclusion

As part of the degree requirements of the Electrical Engineering Department at LMU, the proposed GC capstone projects aims to scientifically explore detectors and develop finished products for educational use in the Chemistry Department. As a multi-disciplinary subject, this GC instrument will provide tremendous educational benefit students. The proposed project will design and test detectors based on 1N4148 diodes, hydrocarbon sensors, carbon-film resistive elements, and colorimetric arrays and compare them to a purchased rhenium-tungsten TCD. There are five elements that must be complete before the end of the Spring 2020 semester: detector hardware, high-throughput columns, temperature control systems, chassis/cosmetic design, and a GUI interface. The project will comply with all ethical standards and fulfill program and accrediting requirements as set by the university. By the end of the Fall 2019 semester, a working prototype has been produced and will be presented to faculty. Far along in the design process, the project is on schedule to produce finished products by the end of next semester.

11. Future Works

The project was designed and developed with the intention of future development. Considering the diverse problem space of the topic and variety of solutions, this project can be modified and further developed by the original contributors as well as any person around the world to produce a refined product. Currently, the product is unfinished: the physical prototype is a riveted aluminum shell with external electronics and the software leaves the user with few inputs/actions available. In order to facilitate this future work, the hardware and software designs and source codes are available on the Github for perpetuity. The original contributors, future students, and interested people can communicate and develop concurrently over the platform. Furthermore, the original contributors will always be available for communication.

11.1 Hardware

A list of possible improvements to the physical prototype include, in no particular order,

- Weld together walls of instrument shell.
- Better gas flow rate control.
- Variety of carrier gases to determine cheapest that yields sufficient signal-to-noise ratio.
- Place electronics inside the shell for portability and application to educational setting.
- Add an additional heating block to allow independent setting of the detector temperature. Currently the detector is heated by its proximity to the main oven, but this does not provide the necessary flexibility.
- Paint to make it look pretty (Rustoleum high temperature black enamel)
- Configure to run using helium to provide a cleaner signal.
- Explore the use of other detectors that do not rely on measuring differences in the thermal conductivity of the eluant. Perhaps a capacitive or arc discharge ionization detector would be worth exploring.

11.2 Software

A list of possible improvements to the software/code include, in no particular order,

- Compile to executable to simplify start up and maintain source code security. For the reader's information, the YAML file type for defaults was chosen because it compiles well. Currently, there is no security on the code because it is raw Python.
- More complex data structure
  - More variables/information on data collection
  - Possibly a better structure than 2D numpy array; although, numpy was specifically chosen for computational speed and functions.
- Ability for user to modify settings such as graph color and markers. This functionality was began with the $GCConfigPopup$ class but was left unconnected in the final product (version 4).

- Cutting/removing sections of data via a GUI tool.
- GUI tool for zooming and moving around the plot.
- Window/menu functionality to modify settings pertaining to the hardware (e.g. gas flow rate).
- Proper error handling through custom exception classes: the code should throw informational exceptions for the programmer/user during new code testing or hardware set up. For example, currently nothing is done when no open serial ports are found (most likely unconnected cord) and this causes errors further down execution.
- Printing to printer.

Bibliography

[1] Benjamin S. Valle. Wheatstone bridge of the thermal conductivity detector, 1999. [Online; accessed December 3, 2019]. 2

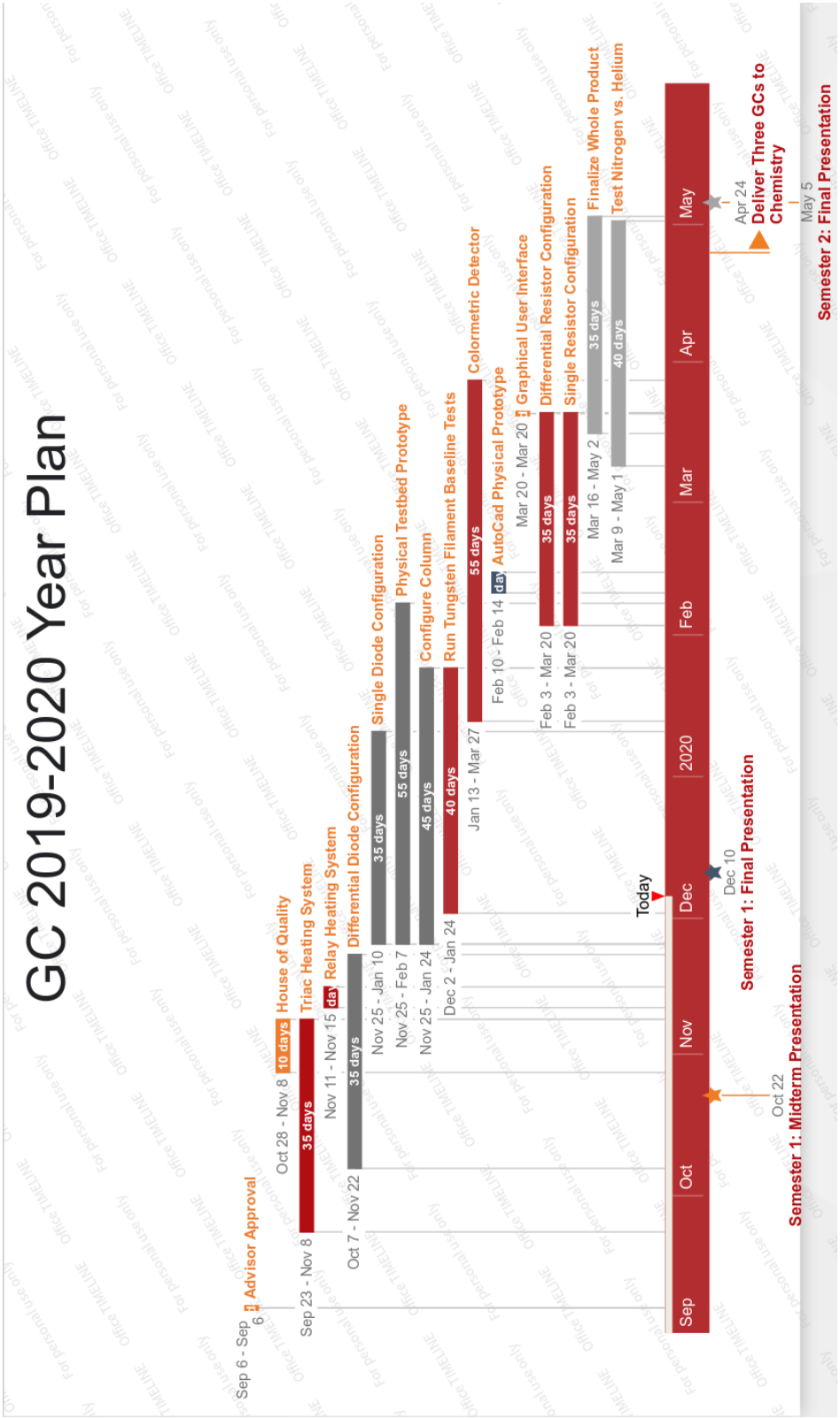
[2] K. Rakow, N. & Suslick. A colorimetric sensor array for odour visualization. *Nature*, 406:710–713, August 2000. 2, 3

[3] M. Jones. A simple-to-build thermal-conductivity gc detector. *Journal of Chemical Education*, 71:995–996, November 1994. 8

[4] Japan figaro tgs813 combustion gas sensor methan butane, 2017. [Online; accessed December 3, 2019]. 9

[5] ABET Engineering Accreditation Commission. Criteria for accrediting engineering programs, October 2017. 29

A. Detailed Schedule



B. Teammate Roles and Responsibilities

B.1 Conor Green

Conor has expertise in academic writing and digital systems. As such, Conor will document and manage the projects timeline and goals, particularly in the context of the capstone course. Conor will be responsible for software and systems such as the Python GUI and the Arduino heating system. Of course, the analog and instrumental obstacles will be addressed as well to the best of his abilities.

- Documentation
  - Github
  - Weekly Reports
  - Deliverables for Capstone Course
  - Abstract
  - Areas of Expertise Within this Report:
    - Ethical Considerations
    - ABET Contributions
    - Latex Formatting
    - GUI Software Development
- Software
  - Arduino Heating Feedback System
  - Python suite
  - User Interface
- Circuit Design and Building
  - Thermocouple Amplifier
  - Differential Configuration

B.2 Matthew McPartlan

As a Electrical Engineering and Chemistry double major, Matt has the required knowledge to deal with the chemical side of the project, specifically regarding design of the physical chromatograph itself. Matt has experience determining the optimal temperatures, compounds, injection volumes, flow rates, and expected results in order to test and compare the various detectors. Beyond the application of Chemistry knowledge, an understanding of chemistry and how electrical characteristics change is critical to designing the best detector.

- Circuit Design and Building
  - Single Component Configuration
  - Differential Configuration

- Performing GC Tests
  - Determining Target Temperature
  - Controlling Gas Flow Rate
  - Injecting Test Substance
- Documentation
  - Weekly Reports
  - Deliverables for Capstone Course
  - Abstract
  - Areas of Expertise Within this Report:
    * Introduction
    * Objectives
    * Detectors
    * Development
- Software
  - Arduino software, especially the USB_COMM library
  - TRIAC heating element control

C. Source Code

C.1 GCController.ino

```
1
2  #include <SPI.h>
3  #include "USB_COMM.h"
4
5  #define THERM1 9
6  #define THERM2 7
7  #define THERM3 8
8  #define TRIAC1 5
9  #define TRIAC2 3
10 #define INTERRUPT_PIN 2
11 uint16_t blankSend = 0;
12 uint16_t digitMask = 0x1FFE;
13
14 // Delays are in microseconds
15 unsigned long maxDelay = 8500;
16 unsigned long ISRDelay1 = maxDelay;
17 unsigned long ISRDelay2 = maxDelay;
18 unsigned long ZCPeakOffset = 0;
19 volatile unsigned long lastZC = 0;
20 volatile int interruptCounter = 0;
21
22 int setTemp1;
23 int setTemp2;
24
25 double tempDelayTable[101];
26
27 USB_COMM thermalCont(9600);
28
29 void recordZC() {
30   lastZC = micros();
31 }
32
33 void setup() {
34   // put your setup code here, to run once:
35   pinMode(THERM1, OUTPUT);
36   pinMode(THERM2, OUTPUT);
37   pinMode(THERM3, OUTPUT);
38   pinMode(TRIAC1, OUTPUT);
39   pinMode(TRIAC2, OUTPUT);
40   pinMode(INTERRUPT_PIN, INPUT);
41   digitalWrite(THERM1, HIGH);
42   digitalWrite(THERM2, HIGH);
43   digitalWrite(THERM3, HIGH);
44
45   Serial.begin(115200);
46   SPI.begin();
47   SPI.beginTransaction(SPISettings(400000, MSBFIRST, SPI_MODE1));
48   delay(1000);
49
50   for (int i = 0; i < 101; i++) {
51     if (i < 50) {
52       tempDelayTable[i] = 0;
53     } else if (i < 100) {
54       tempDelayTable[i] = 0.5 + ((double)i / 100 - 0.8);
```

```
55      } else {
56        tempDelayTable[i] = 1;
57      }
58    }
59
60    attachInterrupt(digitalPinToInterrupt(INTERRUPT_PIN), recordZC, RISING);
61  }
62
63  void loop() {
64    // Tell comm manager to look for message and update set temps if found
65    thermalCont.checkForMsg();
66
67    if (lastZC + ISRDelay1 + ZCPeakOffset ≤ micros()) {
68      digitalWrite(TRIAC1, HIGH);
69    }
70    if (lastZC + ISRDelay2 + ZCPeakOffset ≤ micros()) {
71      digitalWrite(TRIAC2, HIGH);
72    }
73    digitalWrite(TRIAC1, LOW);
74    digitalWrite(TRIAC2, LOW);
75
76    interruptCounter += 1;
77    if (interruptCounter ≥ 20000) {
78      digitalWrite(THERM1, LOW);
79      int thermalSPI1 = ((uint16_t)SPI.transfer16(blankSend) & digitMask) >> 5;
80      digitalWrite(THERM1, HIGH);
81      digitalWrite(THERM2, LOW);
82      int thermalSPI2 = ((uint16_t)SPI.transfer16(blankSend) & digitMask) >> 5;
83      digitalWrite(THERM2, HIGH);
84      digitalWrite(THERM3, LOW);
85      int thermalSPI3 = ((uint16_t)SPI.transfer16(blankSend) & digitMask) >> 5;
86      digitalWrite(THERM3, HIGH);
87
88      // Tell the communication manager what the temps are...
89      thermalCont.setRealTemps(thermalSPI1, thermalSPI2, thermalSPI3);
90
91      // Refresh temperatures set by raspberry pi
92      setTemp1 = thermalCont.getSetTemp1();
93      setTemp2 = thermalCont.getSetTemp2();
94
95      int lookupTableIndex1 = round(((double)thermalSPI1 / (double)setTemp1) ...
            * 100);
96      int lookupTableIndex2 = round(((double)thermalSPI2 / (double)setTemp2) ...
            * 100);
97
98      if (lookupTableIndex1 > 99) {
99        ISRDelay1 = maxDelay * tempDelayTable[100];
100     } else {
101       ISRDelay1 = maxDelay * tempDelayTable[lookupTableIndex1];
102     }
103
104     if (lookupTableIndex2 > 99) {
105       ISRDelay2 = maxDelay * tempDelayTable[100];
106     } else {
107       ISRDelay2 = maxDelay * tempDelayTable[lookupTableIndex2];
108     }
109     interruptCounter = 0;
```

```
110     }
111 }
```

C.2 USB_COMM.cpp

```
1  //
2  //  BLE_SP.cpp
3  //  BLE_SP
4  //
5  //  Created by Matt Mcpartlan.
6  //  Copyright    2020 Matt Mcpartlan. All rights reserved.
7  //
8  // This library goes with the GCController main logic board.
9  // It adds shift 16-bit register functionality (implemented in the Shift ...
       class)
10 // and
11
12 #include "Arduino.h"
13 #include "USB_COMM.h"
14 #include <math.h>
15
16 // Params are PIN NUMBERS (SCLK and MISO for temp comms) (THERM1, THERM2, ...
       THERM3 for temp chip select)
17 USB_COMM::USB_COMM(int baud) {
18   Serial.begin(baud);
19 };
20
21 bool USB_COMM::checkForMsg() {
22   String incommingSignal = "";
23   int stringIndex = 0;
24   while (Serial.available() > 0) {
25     incommingSignal = incommingSignal + (char) Serial.read();
26     stringIndex += 1;
27     if (stringIndex > 14) {
28       // Clear the register after collecting data
29       while (Serial.available()) {
30         Serial.read();
31       }
32     }
33   }
34   return inputHandler(incommingSignal);
35 }
36
37 bool USB_COMM::inputHandler(String strObjInput) {
38     char inputString[16];
39     strObjInput.toCharArray(inputString, 16);
40     if (strlen(inputString) != 15) {
41         _exitCode = 2;
42         return false;
43     } else {
44         int spaceCounter = 0;
45         for (int i = 0; i < 15; i++) {
46           // Look for spaces (ASCII #32)
47             if ((int) inputString[i] == 32) {
48                 spaceCounter += 1;
```

```
            }
            if (((int) inputString[i] < (int) '0' || ((int) inputString[i] ...
                > (int) '9')) && (int) inputString[i] > (int) ' ') {
                // SPACES ARE NOT INTS
                _exitCode = 1;
                Serial.println(generateTransmissionString(2));
                _exitCode = 0;
                return false;
            }
        }
        if (spaceCounter != 3) {
            _exitCode = 2;
            Serial.println(generateTransmissionString(2));
            _exitCode = 0;
            return false;
        } else {
            int convert[3] = { 0, 0, 0 };
            int tempInt;

            // Read operation code
            convert[0] = inputString[0] - '0';
            convert[1] = inputString[1] - '0';
            convert[2] = inputString[2] - '0';
            _opCode = convert[0] * 100 + convert[1] * 10 + convert[2];

            // Read temp arguments
            int tempArrayIndex = 0;
            for (int i = 4; i < 15; i += 4) {
                convert[0] = inputString[i] - '0';
                convert[1] = inputString[i + 1] - '0';
                convert[2] = inputString[i + 2] - '0';
                tempInt = convert[0] * 100 + convert[1] * 10 + convert[2];
                if (_opCode == 0) {
                  // Set new desired temperatures
                  switch (tempArrayIndex) {
                    case 0:
                      _setTemp1 = tempInt;
                      break;
                      case 1:
                      _setTemp2 = tempInt;
                      break;
                      case 2:
                      // For now, this should always be 0
                      _setTemp3 = tempInt;
                      if (_setTemp3 != 0) {
                        _exitCode = 3;
                        Serial.println(generateTransmissionString(2));
                        _exitCode = 0;
                        return false;
                      }
                    }
                }
                tempArrayIndex++;
            }
        }
    }
    Serial.println(generateTransmissionString(_opCode));
```

```
105     return true;
106 }
107
108 int USB_COMM::getTemp1() {
109     return _realTemp1;
110 }
111
112 int USB_COMM::getTemp2() {
113     return _realTemp2;
114 }
115
116 int USB_COMM::getTemp3() {
117     return _realTemp3;
118 }
119
120 int USB_COMM::getSetTemp1() {
121     return _setTemp1;
122 }
123
124 int USB_COMM::getSetTemp2() {
125     return _setTemp2;
126 }
127
128 int USB_COMM::getSetTemp3() {
129     return _setTemp3;
130 }
131
132 void USB_COMM::setDesiredTemps(int set1, int set2, int set3) {
133   _setTemp1 = set1;
134   _setTemp2 = set2;
135   _setTemp3 = set3;
136 }
137
138 void USB_COMM::setRealTemps(int set1, int set2, int set3) {
139   _realTemp1 = set1;
140   _realTemp2 = set2;
141   _realTemp3 = set3;
142 }
143
144 int USB_COMM::getOpCode() {
145     return _opCode;
146 }
147
148 int USB_COMM::getExitCode() {
149     return _exitCode;
150 }
151
152 bool USB_COMM::setTemps(int temp1, int temp2, int temp3) {
153   _setTemp1 = temp1;
154   _setTemp2 = temp2;
155   _setTemp3 = temp3;
156   return true;
157 }
158
159 String USB_COMM::to3String(double number) {
160   char hundreds = char((floor(number / 100.0) + 48));
161   char tens = char((floor(number / 10.0) - ((hundreds - 48) * 10) + 48));
```

```
162   char ones = char((number - ((hundreds - 48) * 100) - ((tens - 48) * 10) ...
          + 48));
163   String output = "";
164   output = output + hundreds + tens + ones;
165   return output;
166 }
167
168 String USB_COMM::generateTransmissionString(int opCode) {
169   String opCodeS;
170   String arg1S;
171   String arg2S;
172   String arg3S;
173     switch (opCode) {
174       case 0:
175         opCodeS = "000";
176         arg1S = to3String(_setTemp1);
177         arg2S = to3String(_setTemp2);
178         arg3S = to3String(_setTemp3);
179         break;
180       case 1:
181         opCodeS = "001";
182         arg1S = to3String(_realTemp1);
183         arg2S = to3String(_realTemp2);
184         arg3S = to3String(_realTemp3);
185         break;
186       case 2:
187         opCodeS = "002";
188         arg1S = to3String(_exitCode);
189         arg2S = "***";
190         arg3S = "***";
191     }
192     String retStr = opCodeS + " " + arg1S + " " + arg2S + " " + arg3S;
193     return retStr;
194 }
```

C.3 USB_COMM.h

```
1
2  //
3  //  BLE_SP.h
4  //
5  //  Created by Matt Mcpartlan on 4/25/18.
6  //  Copyright    2018 Matt Mcpartlan. All rights reserved.
7  //
8  // Example transmission: "000 000 333 100"
9  // Transmission length: 15
10
11 #ifndef USB_COMM_h
12 #define USB_COMM_h
13
14 #include "Arduino.h"
15
16 class USB_COMM {
17 public:
18     USB_COMM(int baud);
```

```
    bool inputHandler(String strObjInput);
    bool setTemps(int temp1, int temp2, int temp3);
    bool checkForMsg();
    int getTemp1();
    int getTemp2();
    int getTemp3();
    int getSetTemp1();
    int getSetTemp2();
    int getSetTemp3();
    int getOpCode();
    int getExitCode();
    String to3String(double number);
    String generateTransmissionString(int opCode);
    void setRealTemps(int set1, int set2, int set3);
    void setDesiredTemps(int set1, int set2, int set3);
private:
    // Temperature storage
    int _setTemp1 = 0;
    int _setTemp2 = 0;
    // Do NOT use setTemp3, the hardware does not support it (yet)
    int _setTemp3 = 0;
    int _realTemp1 = 0;
    int _realTemp2 = 0;
    int _realTemp3 = 0;

    // Arduino command codes (recieved by Arduino):
      // 000 - Verify set temperature
      //       Raspberry pi should call this once to make that the Arduino ...
          understood the command
      // 001 - Read temperature
      //         Raspberry pi should call this to ask for the current ...
          sensor readings
      // 002 - Error
      //         Arduino will return error if unable to execute raspberry ...
          pi commands (usually overtemperature protection)
    int _opCode = 0;
    int _exitCode = 0;
};

#endif
```

C.4 gas_chromatography.py

```

'''
Name: gas_chromatography.py
Authors: Conor Green and Matt McPartlan
Description: Highest abstraction of GC GUI interface. Will be initially ...
    called from bash upon startup of Raspberry Pi.
Usage: Call as main
Version:
1.0 - 26 January 2020 - Initial creation. Skeleton to outline work for later
1.1 - 20 February 2020 - Import and create gc_class Gas_Chrom object.
1.2 - 30 March 2020 - Complete overhaul. Utilizes frame and panel classes ...
    created in gc_gui and smashes them together using the SplitterWindow.
```

```
1.3 - 30 March 2020 - Added Save As window that navigates directories and ...
    updates listbox.
1.4 - 31 March 2020 - Added Open window, similar to Save As. Both inherit ...
    new window class.
2.0 - 21 April 2020 - Final version. Moved all code to gc_gui.py...
                          and made this a simple App running script.
3.0 - 24 April 2020 - Same as 2.0. Updated version to match gc_class and ...
    gc_gui
4.0 - 27 April 2020 - Manually configure serial string constants. Serial ...
    integer constants are set normally through yaml file.
'''

import wx
import yaml
import os

import gc_gui

'''
self - MainApp object
param: self.parent - the parent wxPython object, in this case None (main ...
    call of MainApp(None))
attribute: self.options - the UI defeaults and current settings loaded ...
    from YAML config file
'''
class MainApp(gc_gui.GCFrame):
    def __init__(self, parent):
        self.__version__ = '3.0'
        self.__authors__ = 'Conor Green and Matt McPartlan'

        print('\n\n\n\n----------------------------------------------------------------------------------')
        print('Running gas_chromatography.py')
        print('Press ctrl+c to terminate.')
        print('Messages and errors printed below.')

        with open('config.yaml') as f:
            options = yaml.load(f, Loader=yaml.FullLoader)
            self.options = options

        # dont have time to figure out in yaml
        _dict = {'READ_TMP_CMD_STR' :'001 000 000 000' , ...
            'SET_TMP_CMD_STR': '000 XXX XXX 000'}
        self.options.update(_dict)

        gc_gui.GCFrame.__init__(self, parent, self.options)

def main():
    app = wx.App()
    window = MainApp(None) #GCFrame
    window.Show(True)
    app.MainLoop()

if __name__ == '__main__':
    main()
```

C.5 config.yaml

```
1
2  #Name: config.yaml
3  #Usage: Import from gas_chromatography.py and use values to create GUI.
4
5  BODY_FONT_SIZE: 11
6  HEADER_FONT_SIZE: 18
7  EXTRA_SPACE: 10
8  BORDER: 10
9
10 DEFAULT_FRAME_SIZE: !!python/tuple [1200,500]
11 DEFAULT_SASH_SIZE: 300
12
13 BAUDRATE: 9600
14 OVEN_INDEX: 1
15 INJ_INDEX: 2
16 SER_DELAY: .01
```

C.6 gc_class.py

```
1
2  '''
3  Name: gc_class.py
4  Authors: Conor Green and Matt McPartlan
5  Description: Mid-level abstraction that will do most of the work. User ...
       interaction as well as data processing.
6  Usage: Call as main
7
8  Version:
9  1.0 - 26 January 20 - Initial creation. Skeleton to outline work for later
10 1.1 - 18 February 20 - Initialized and created some methods for ADS1115 ...
       and numpy
11 1.2 - 20 February 20 - Added methods for data collection and organizing ...
       self variables
12 1.3 - 21 February 20 - Debugged and works in preliminary testing!
13
14 2.0 - 21 April 2020 - Final version. Sufficiently supports gc_gui.py in ...
       data collection with simple methods. ...
15                          Old methods, that can plot, print, etc. are left ...
                                at the end for future debugging/testing.
16 2.1 - 22 April 2020 - Integrate and normalize voltage methods.
17 2.2 - 22 April 2020 - Huge security upgrade. Moved lock to here (gc_class) ...
       and protected curr_data through getters and setters.
18 2.3 - 24 April 2020 - Modified lock structure. Added re_init_data and ...
       curr_to_prev to move current data to prev_data list.
19
20 3.0 - 24 April 2020 - Clean version that works great with gui script. ...
       Version 3 is very pydocs friendly and has accompanying html, gc_class.html
21 3.1 - 25 April 2020 - Functions to split into peaks, calculate the area of ...
       each region, and get the maximum.
22 3.2 - 27 April 2020 - Moving mean works. Fixed indices.
23 4.0 - 27 April 2020 - Finalized for capstone project submission. Pydocs ...
       final. Moved to main.py (=> .exe/bin)
```

```
'''


# GPIO imports
import board
import busio
import adafruit_ads1x15.ads1115 as ADS
from adafruit_ads1x15.analog_in import AnalogIn

# Extra
import time

# Numpy and Matplotlib
import numpy as np
import matplotlib.pyplot as plt

from threading import Lock

class GC:
    '''
    Modification functions: clean_time_, normalize_volt_, mov_mean_, ...
        curr_to_prev_
    Math functions: integrate_volt, integrate_volt_direct, ...
        break_into_peaks, break_into_peaks_ret_volt_copy,
                        integrate_peaks, get_peak_local_maximas, ...
                            calc_cumsum_into_area_
    helper functions: define_peaks, reint_curr_data_, inc_run_num_, integrate
    ADS configuration functions: reint_ADS, set_gain, set_mode
    Lock functions: get_lock, is_locked
    Getters: get_curr_data, get_volt, get_time
    Setters: set_curr_data_, set_time_w_ref_, set_volt_, set_time_, ...
        set_time_w_ref_, set_area_
    Printer functions: print_voltage, print_value
    Measurement functions: measure_voltage, measure_value
    '''

    #@param: single_ended = True if single ended ADC and vice-versa
    def __init__(self, single_ended):
        self.__version__ = '4.0'
        self.__authors__ = 'Conor Green and Matt McPartlan'

        #ADS1115
        self.i2c = busio.I2C(board.SCL , board.SDA)
        self.ads = ADS.ADS1115(self.i2c)

        self.single_ended = single_ended # T/F
        self.port0 = ADS.P0 # Later allow user input
        self.port1 = ADS.P1

        if single_ended:
            self.chan = AnalogIn(self.ads , self.port0)
        else:
            self.chan = AnalogIn(self.ads, self.port0 , self.port1)

        #Numpy/data
        self.dims = 4 #voltage, dt, t
        self.indices = {'v':0,'a':1,'t':2,'dt':3}
```

```
77
78          self.curr_data = np.zeros((self.dims, 0))
79          self.curr_data_lock = Lock()
80
81          self.prev_data = []
82          self.run_num = 0
83
84          self.time_out = 1 #sec
85
86          self.epsilon = 0.01
87
88          self.pk_volt_min_after_norm = 0.005
89          self.pk_time_min = 10
90          self.peaks = []
91
92      #@description: Subtracts initial time from all time points => t[0] = 0
93      def clean_time_(self):
94          if self.run_num > 0:
95              to = self.time_out
96              _e = self.curr_data_lock.acquire(to)
97              t = self.get_time()
98              _e = self.curr_data_lock.release()
99
100             t = t - t[0]
101
102             _e = self.curr_data_lock.acquire(to)
103             self.set_time_(t)
104             _e = self.curr_data_lock.release()
105
106     #@description: Calculates the cumulative sum of the voltage at each ...
            point and stores the result in area.
107     def calc_cumsum_into_area_(self):
108         to = self.time_out
109         #ignore err for now
110         _e = self.curr_data_lock.acquire(to)
111         voltage = self.get_volt()
112         _e = self.curr_data_lock.release()
113
114         cs = np.cumsum(voltage)
115
116         _e = self.curr_data_lock.acquire(to)
117         self.set_area_(cs)
118         _e = self.curr_data_lock.release()
119
120     #@description: Normalizes (integral[voltage] = 1 & min[voltage] = 0) ...
            the voltage in place.
121     def normalize_volt_(self):
122         to = self.time_out
123         #ignore error for now
124         _e = self.curr_data_lock.acquire(to)
125         volt = self.get_volt()
126         _e = self.curr_data_lock.release()
127
128         _min = volt.min()
129         volt = volt - _min
130
131         _area = self.integrate_volt_direct(volt)
```

```
132         volt = volt / _area
133
134         _e = self.curr_data_lock.acquire(to)
135         self.set_volt_(volt)
136         _e = self.curr_data_lock.release()
137
138     #@description: Applies moving mean of window size given to voltage via ...
            convolution
139     #@param: window size
140     def mov_mean_(self, window):
141         to = self.time_out
142         _e = self.curr_data_lock.acquire(to)
143         volt = self.get_volt()
144         _e = self.curr_data_lock.release()
145
146         volt = np.convolve(volt, np.ones((window,))/window, mode='same')
147
148         _e = self.curr_data_lock.acquire(to)
149         self.set_volt_(volt)
150         _e = self.curr_data_lock.release()
151
152     #@description: Appends current data copy to previoius data list and ...
            sets current data back to zero vector.
153     def curr_to_prev_(self):
154         _l = self.curr_data_lock
155         with _l:
156             _d = self.get_curr_data()
157
158         self.prev_data.append(_d)
159         self.reint_curr_data_()
160
161     #@returns: int or float that is the area of voltage, including negatives
162     def integrate_volt(self):
163         to = self.time_out
164         #ignore err for now
165         _e = self.curr_data_lock.acquire(to)
166         voltage = self.get_volt()
167         _e = self.curr_data_lock.release()
168
169         if len(voltage) != 0:
170             _l = 0
171             _h = -1
172             _a = self.integrate(voltage, _l, _h)
173             return _a
174         else:
175             print("No data")
176
177     #@param: directly given voltage vector
178     #@returns: int or float that is the area of voltage, including negatives
179     def integrate_volt_direct(self, voltage):
180         if len(voltage) != 0:
181             _l = 0
182             _h = -1
183             _a = self.integrate(voltage, _l, _h)
184             return _a
185         else:
186             print("No data")
```

```
187
188     #@description: Breaks the voltage vector into potential peaks. Simple ...
            and untested algorithm.
189     #@returns: List of tuples of start and end index (both inclusive) of peaks
190     def break_into_peaks(self):
191         ep = self.epsilon
192         if abs(self.integrate_volt() - 1.0 ) > ep:
193             self.normalize_volt_()
194         to = self.time_out
195         #ignore err for now
196         _e = self.curr_data_lock.acquire(to)
197         volt = self.get_volt()
198         _e = self.curr_data_lock.release()
199
200         volt_thresh = self.pk_volt_min_after_norm
201         time_thresh = self.pk_time_min
202
203         peaks = []
204
205         num_pts = len(volt)
206         on_peak = False
207         low = 0
208         for i in range(0, num_pts):
209             if not on_peak:
210                 if volt[i] ≥ volt_thresh:
211                     on_peak = True
212                     low = i
213             else:
214                 if volt[i] ≤ volt_thresh:
215                     on_peak = False
216                     if i - low ≥ time_thresh:
217                         peaks.append((low, i))
218
219         return peaks
220
221     #@description: Breaks the voltage vector into potential peaks.
222     #@returns: List of tuples of start and end index (both inclusive) of peaks
223     #@returns: Copy of voltage vector to save a little time in parent method.
224     def break_into_peaks_ret_volt_copy(self):
225         ep = self.epsilon
226         if abs(self.integrate_volt() - 1.0 ) > ep:
227             self.normalize_volt_()
228         to = self.time_out
229         #ignore err for now
230         _e = self.curr_data_lock.acquire(to)
231         volt = self.get_volt()
232         _e = self.curr_data_lock.release()
233
234         volt_thresh = self.pk_volt_min_after_norm
235         time_thresh = self.pk_time_min
236
237         peaks = []
238
239         num_pts = len(volt)
240         on_peak = False
241         low = 0
242         for i in range(0, num_pts):
```

```
243             if not on_peak:
244                 if volt[i] ≥ volt_thresh:
245                     on_peak = True
246                     low = i
247             else:
248                 if volt[i] ≤ volt_thresh:
249                     on_peak = False
250                     if i − low ≥ time_thresh:
251                         peaks.append((low, i))
252 
253         return [peaks , volt ]
254 
255     #@description: Calculates the area of peaks (calls within) of voltage.
256     #@returns: List of areas corresponding to peaks in order
257     def integrate_peaks(self):
258         [peaks , volt] = self.break_into_peaks_ret_volt_copy()
259 
260         areas = []
261 
262         for low, high in peaks:
263             _a = self.integrate(volt, low, high)
264             areas.append(_a)
265 
266         return areas
267 
268     #@description: Calculates the maximum values and indices of those ...
            maximas of voltage.
269     #@returns: List of (max_index, max_val) pairs
270     def get_peak_local_maximas(self):
271         [peaks,volt] = self.break_into_peaks_ret_volt_copy()
272 
273         maximas = []
274 
275         for low, high in peaks:
276             arr = volt[low:high]
277             _m = np.max(arr)
278             _mi = arr.argmax() + low
279             maximas.append((_mi, _m))
280 
281         return maximas
282 
283     '''
284     helper functions: reint_curr_data_, inc_run_num_, integrate
285     '''
286     def integrate(self, arr, low, high):
287         if low < len(arr) −1:
288             arr = arr[low:−1]
289             high = high − low
290 
291         tot_area = np.cumsum(arr)
292         if high == −1:
293             area = tot_area[−1]
294         elif high < len(arr) −1:
295             area = tot_area[high]
296         else:
297             print("High index in integration out of bounds")
298             area = tot_area[−1]
```

```
299
300             return area
301
302         def reint_curr_data_(self):
303             _dims = self.dims
304
305             _l = self.curr_data_lock
306             with _l:
307                 _z = np.zeros((_dims, 0))
308                 self.set_curr_data_w_ref_(_z)
309
310         def inc_run_num_(self):
311             self.run_num += 1
312
313         '''
314         ADS configuration functions: reint_ADS, set_gain, set_mode
315         '''
316         def reinit_ADS(self):
317             self.i2c = busio.I2C(board.SCL , board.SDA)
318             self.ads = ADS.ADS1115(self.i2c)
319
320             self.chan = AnalogIn(ads , self.port0) if self.single_ended else ...
                    AnalogIn(ads, self.port0 , self.port1)
321
322         def set_gain(self, g):
323             self.ads.gain = g
324
325         def set_mode(self, se):
326             self.single_ended = se
327             self.reinit_ADS()
328
329         '''
330         Lock functions: get_lock, is_locked
331         '''
332         def get_lock(self):
333             return self.curr_data_lock
334
335         def is_locked(self):
336             _il = self.curr_data_lock.locked()
337             return _il
338
339         '''
340         Getters: get_curr_data, get_volt, get_time
341         '''
342         def get_curr_data(self):
343             _il = self.is_locked()
344             if _il:
345                 _d = np.copy(self.curr_data)
346                 return _d
347             else:
348                 print('get_curr_data: no access')
349
350         def get_volt(self):
351             is_locked = self.is_locked()
352             _vi = self.indices['v']
353             if is_locked:
354                 _v = np.copy(self.curr_data[_vi])
```

```
355             return _v
356         else:
357             print('get_volt: no access')
358 
359     def get_time(self):
360         _il = self.is_locked()
361         _ti = self.indices['t']
362         if _il:
363             _t = self.curr_data[_ti]
364             return _t
365         else:
366             print('get_time: no access')
367 
368     def get_dims(self):
369         return self.dims
370 
371     '''
372     Setters: set_curr_data_, set_time_w_ref_, set_volt_, set_time_, ...
            set_time_w_ref_, set_area_
373     '''
374     def set_curr_data_(self, d):
375         _il = self.is_locked()
376         if _il:
377             d = np.copy(d)
378             self.curr_data = d
379         else:
380             print('set_curr_data: no access')
381 
382     def set_curr_data_w_ref_(self, d):
383         _il = self.is_locked()
384         if _il:
385             self.curr_data = d
386         else:
387             print('set_curr_data_w_ref: no access')
388 
389     def set_volt_(self, d):
390         is_locked = self.is_locked()
391         _vi= self.indices['v']
392         if is_locked:
393             d = np.copy(d)
394             self.curr_data[_vi] = d
395         else:
396             print('set_volt: no access')
397 
398     def set_time_(self, d):
399         _il = self.is_locked()
400         _ti = self.indices['t']
401         if _il:
402             d = np.copy(d)
403             self.curr_data[_ti] = d
404         else:
405             print('set_time: no access')
406 
407     def set_time_w_ref_(self, d):
408         _il = self.is_locked()
409         _ti = self.indices['t']
410         if _il:
```

```
411             self.curr_data[_ti] = d
412         else:
413             print('set_time_w_ref: no access')
414 
415     def set_area_(self, d):
416         _il = self.is_locked()
417         _ai = self.indices['a']
418         if _il:
419             d = np.copy(d)
420             self.curr_data[_ai] = d
421         else:
422             print('set_area_w_ref: no access')
423 
424     '''
425     Printer functions: print_voltage, print_value
426     '''
427     def print_voltage(self):
428         print(self.chan.voltage)
429 
430     def print_value(self):
431         print(self.chan.value)
432 
433     '''
434     Measurement functions: measure_voltage, measure_value
435     '''
436     def measure_voltage(self):
437         return self.chan.voltage
438 
439     def measure_value(self):
440         return self.chan.value
441 
442     '''
443     Temporary/old methods
444     '''
445     # old, dont trust
446     def graph_curr_data_on_popup(self):
447         plt.figure()
448         to = self.time_out
449         _e = self.curr_data_lock.acquire(to)
450         t = self.get_time()
451         v = self.get_volt()
452         _e = self.curr_data_lock.release(to)
453         plt.scatter( t, v )
454         plt.show()
455 
456     # old, dont trust
457     def coll_volt_const_pts(self , num_pts):
458         _vi = self.indices['v']
459         _ai = self.indices['a']
460         _dti = self.indices['dt']
461         _ti = self.indices['i']
462 
463         temp_data_arr =  np.zeros((self.dims , num_pts ) )
464 
465         t_start = time.time()
466         for i in range(0,num_pts):
467             time.sleep(.01)
```

```
468             temp_data_arr[_vi][i] = self.measure_voltage()
469             temp_data_arr[_ai] = None
470             t_curr = time.time()
471             temp_data_arr[_dti][i] = t_curr - t_start
472             temp_data_arr[_ti][i] = t_curr
473 
474         return voltage_an_dtime
475 
476     # old, dont trust
477     def coll_volt_const_pts_self(self, num_pts):
478         self.run_num += 1
479         self.prev_data.append(self.curr_data)
480         self.curr_data = self.coll_volt_const_pts(num_pts)
481         print(self.curr_data)
482 
483 if __name__ == '__main__':
484     pass
485 
486     # Remove pass above to run test script: single-ended 1000 point data...
487     #       colleciton and graphing
488     gc = GC(True)
489 
490     print("Collecting 1000 data points...")
491     gc.coll_volt_const_pts_self(1000)
492     print("\nGraphing...")
493     gc.graph_curr_data()
```

C.7 gc_gui.py

```
1  '''
2  Title: gc_gui.py
3  Author: Conor Green & Matt McPartlan
4  Description: Mid-level abstraction of GUI application to perform data ...
       acquisition and display. Defines GUI and threading classes and ...
       instantiates GC object.
5  Usage: Instantiate GCFrame object from gas_chromatography.py
6  Version (mainly for final report later):
7  1.0 - November 24 2019 - Initial creation. All dependencies left in the ...
       script: will later be split into various scripts that are imported.
8  1.1 - November 24 2019 - Implements numpy and plotting to window. Uses ...
       random numbers
9  1.2 - 31 March 2020 - Old gas_chromatography.py -> gc_gui.py. This script ...
       defines the frame and panel classes that are put together in ...
       gas_chromatography.py. As of currently, it plots an example sin curve ...
       in the plotter but interfacing with the ADS1115 will be implemented ...
       when this is tested on a Raspberry Pi.
10 1.3 - 31 March 2020 - Added images to buttons. Added more menu options.
11 1.4 - 31 March 2020 - Save current figure as .png or .jpg
12 1.5 - 21 April 2020 - FINALLY got threading working.
13 2.0 - 21 April 2020 - Version 2.0 successfully graphs live data with play ...
       button, ...
14                              graphs it live, saves images and .gc files, ...
                                    and clears.
15                              Basic, minimum GC supporting software.
```

```
2.1 - 22 April 2020 - Serial connection with Arduino through temperature ...
    thread, which updates the text feedback.
                        Commands not yet implemented.
2.2 - 22 April 2020 - Works with updated gc_class and lock.
2.3 - 22 April 2020 - Added functions on voltage in menu bar.
2.4 - 24 April 2020 - Stable. Correctly increments run_number and stores ...
    data in prev_data.
                        Menu data functions: normalize, integrate, and ...
                            clean time all work as intended.
                        Menu grapher function: fill almost works
3.0 - 24 April 2020 - Arduino serial interface. Temperature control and ...
    feedback thread. Normalize, integrate, and clean time menu functions.
3.1 - 25 April 2020 - Open .gc file will set current and previous data to ...
    the loaded file.
3.2 - 25 April 2020 - Opening .gc file works.
3.3 - 25 April 2020 - Label peaks works.
3.4 - 27 April 2020 - Low pass filter works. Run numbers added to .gc json ...
    file type.
4.0 - 27 April 2020 - Finalized to submit for capstone course. Will move ...
    to main.py (=>.exe/bin)
'''

import numpy as np

import matplotlib
matplotlib.use('WXAgg')
from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as ...
    FigureCanvas
from matplotlib.backends.backend_wx import NavigationToolbar2Wx
from matplotlib.figure import Figure
from matplotlib import pyplot as plt

import wx
import wx.lib.plot as plot

import os
import codecs, json
import time

from gc_class import GC

import threading
from threading import Thread

import serial

imdir = '.images'

# Frames
class GCFrame(wx.Frame):
    # self is MainApp(GCFrame)
    # parent is None
    def __init__(self, parent, user_options):
        self.__version__ = '3.3'
        self.__authors__ = 'Conor Green and Matt McPartlan'

        self.establish_options_(user_options)
```

```
65          self.parent = parent
66
67          wx.Frame.__init__ ( self, parent, id = wx.ID_ANY, title = ...
                wx.EmptyString, pos = wx.DefaultPosition, size = ...
                self.options['frame_size'], style = ...
                wx.DEFAULT_FRAME_STYLE|wx.TAB_TRAVERSAL )
68          self.SetSizeHints( wx.DefaultSize, wx.DefaultSize )
69
70          self.build_figure_()
71
72          self.establish_serial_conn_()
73          self.ser_lock = threading.Lock()
74
75          # Requires serial cofnnection self.ser_conn
76          self.temp_thread_running = False
77          self.establish_temperature_thread_()
78
79          # including curr_data_frame and curr_data_frame_lock
80          self.curr_data_frame_lock = threading.Lock()
81          self.establish_GC_()
82
83          _dims = self.gc.get_dims()
84          self.curr_data_frame = np.zeros((_dims, 0))
85          self.update_curr_data_()
86
87          self.prev_data = []
88          self.run_number = 0
89
90          self.data_running = False
91          self.data_paused = False
92
93
94      '''
95      Building functions
96      '''
97      def get_arduino_port(self):
98          possible = [x for x in os.listdir('/dev/') if 'ACM' in x]
99
100         if len(possible) == 0:
101             print("Error: No ACM ports found to connect to Arduino.")
102             print("Attempting on default ttyACM0 for now.")
103             possible.append('ttyACM0')
104         elif len(possible) != 1:
105             print("Error: Multiple possible ACM ports to connect Arduino.")
106             print("Defaulting to '/dev/ttyACM0'")
107             possible[0] = 'ttyACM0'
108
109         local = possible[0]
110         string = '/dev/' + possible[0]
111
112         return string
113
114     def establish_serial_conn_(self):
115         bd = self.options['BAUDRATE']
116         to = self.options['time_out']
117
118         port = self.get_arduino_port()
```

```
119         try:
120             ser = serial.Serial(port, baudrate = bd, timeout = to)
121         except:
122             print("Error: Cannot create serial connection with Arduino.")
123 
124         self.ser_conn = ser
125 
126     def establish_temperature_thread_(self):
127         sp = 1 / self.options['temp_refresh_rate']
128         ep = self.options['epsilon_time']
129         conn = self.ser_conn
130         lock = self.ser_lock
131 
132         self.temperature_thread = GCTemperature( self, conn, lock, args = ...
                (sp, ep) )
133         self.temperature_thread.start()
134         self.temp_thread_running = True
135 
136     def establish_GC_(self):
137         se = self.options['single_ended']
138         self.gc = GC(se)
139         self.gc_lock = self.gc.get_lock()
140 
141     def establish_options_(self, uo):
142         self.options = {'frame_size':(1200,600), 'sash_size':400, ...
                'data_samp_rate':5.0,
143                         'time_out':3, 'epsilon_time':0.001, ...
                            'plot_refresh_rate':2.0, 'temp_refresh_rate':1.0,
144                         'single_ended':True, ...
                            'indices':{'v':0,'a':1,'t':2,'dt':3}, ...
                            'area_accuracy': 8,
145                         'units_str':{'x-axis':'Time [seconds]' , ...
                            'y-axis':'Detector Response [volts]'},
146                         'gc_file_indices': {'cd':'Current Data', ...
                            'pd':'Previous Data', 'rn':'Run Number'},
147                         'window':3}
148 
149         _constants = {'BODY_FONT_SIZE': 11, ...
                'HEADER_FONT_SIZE':18,'EXTRA_SPACE':10, 'BORDER':10}
150         self.options.update(_constants)
151         self.options.update(uo)
152         self.options['frame_size'] = self.options['DEFAULT_FRAME_SIZE']
153         self.options['sash_size'] = self.options['DEFAULT_SASH_SIZE']
154 
155 
156     '''
157     Lock function
158     '''
159     def is_frame_data_locked(self):
160         _il = self.curr_data_frame_lock.locked()
161         return _il
162 
163     '''
164     Frame data functions
165     '''
166     '''
167         Setters
```

```
168     '''
169     def set_frame_from_session_(self, filename):
170         cd , pd, rn = self.parse_session(filename)
171 
172         _l = self.curr_data_frame_lock
173         with _l:
174             self.set_curr_data_w_ref_(cd)
175 
176         _gcl = self.gc_lock
177         with _gcl:
178             self.gc.set_curr_data_w_ref_(cd)
179 
180         self.prev_data = pd
181         self.run_number = rn
182 
183     def parse_session(self, name):
184         # # Format of JSON .gc filetype
185         # curr_session = {
186         # 'Date' : date_str ,
187         # 'Time' : time_str ,
188         # 'Current Data' : curr_data ,
189         #  'Previous Data': prev_data
190         # }
191 
192         _djson = self.load_json_file(name)
193 
194         data_dict_numpy = self.reverse_jsonify(_djson)
195 
196         ind = self.options['gc_file_indices']
197 
198         _cd = ind['cd']
199         _pd = ind['pd']
200         _rn = ind['rn']
201 
202         cd = data_dict_numpy[_cd]
203 
204         pd = data_dict_numpy[_pd]
205 
206         rn = data_dict_numpy[_rn]
207 
208         return (cd , pd, rn)
209 
210     def load_json_file(self, n):
211         with open(n, encoding ='utf-8') as json_file:
212             _d = json.load(json_file)
213         return _d
214 
215     '''
216     dict(lists)->dict(numpys)
217     '''
218     def reverse_jsonify(self, json_dict):
219         numpy_dict = json_dict
220         ind = self.options['gc_file_indices']
221 
222         _cd = ind['cd']
223         _curr_data = [val for key, val in numpy_dict[_cd].items()]
224         numpy_dict.update({_cd : np.array(_curr_data)})
```

```
225
226             _pd = ind['pd']
227             _prev_data = []
228             for data_slice in numpy_dict[_pd]:
229                  _arr = [val for key,val in data_slice.items() ]
230                  _np_arr = np.array(_arr)
231                  _prev_data.append(_np_arr)
232
233             numpy_dict.update({_pd:_prev_data})
234
235             return numpy_dict
236
237         def update_curr_data_(self):
238             _gcl = self.gc.get_lock()
239             with _gcl:
240                 _d = self.gc.get_curr_data()
241
242             _l = self.curr_data_frame_lock
243             with _l:
244                 self.set_curr_data_(_d)
245
246         def curr_to_prev_(self):
247             _l = self.curr_data_frame_lock
248             with _l:
249                 _d = self.get_curr_data_copy()
250
251             self.prev_data.append(_d)
252             self.re_int_curr_data()
253
254         def prev_to_curr_(self):
255             d = self.prev_data.pop()
256
257             _l = self.curr_data_frame_lock
258             with _l:
259                 self.set_curr_data_(d)
260
261             self.run_number -= 1
262
263         def re_int_curr_data(self):
264             _dims = self.gc.get_dims()
265
266             _l = self.curr_data_frame_lock
267             with _l:
268                 _z = np.zeros((_dims, 0))
269                 self.set_curr_data_w_ref_(_z)
270
271         def set_curr_data_(self, d):
272             _il = self.is_frame_data_locked()
273             if _il:
274                 d = np.copy(d)
275                 self.curr_data_frame = d
276
277         def set_curr_data_w_ref_(self, d):
278             _il = self.is_frame_data_locked()
279             if _il:
280                 self.curr_data_frame = d
281
```

```
282     '''
283         Getters
284     '''
285     def get_curr_data(self):
286         _il = self.is_frame_data_locked()
287         if _il:
288             _d = self.curr_data_frame
289 
290         return _d
291 
292     def get_curr_data_copy(self):
293         _il = self.is_frame_data_locked()
294         if _il:
295             _d = self.curr_data_frame
296             _c = np.copy(_d)
297 
298         return _c
299 
300     def get_prev_data_copy(self):
301         _pd = self.prev_data
302         _pdc = [ np.copy(_item) for _item in _pd ]
303         return _pdc
304 
305     def get_figure(self):
306         return self.panel_detector.get_figure()
307 
308     '''
309     Button Events
310     '''
311     def on_temp_txt_ctrl(self, ov_str, inj_str):
312         self.set_temp_ser_cmd(ov_str, inj_str)
313 
314     def set_temp_ser_cmd(self, ov_str_val, inj_str_val):
315         base_str = self.options['SET_TMP_CMD_STR']
316         ov_ind = self.options['OVEN_INDEX']
317         inj_ind = self.options['INJ_INDEX']
318         ser_delay = self.options['SER_DELAY']
319 
320         ov_str_val = ov_str_val[:3]
321         while len(ov_str_val) < 3:
322             ov_str_val = '0' + ov_str_val
323 
324         inj_str_val = inj_str_val[:3]
325         while len(inj_str_val) < 3:
326             inj_str_val = '0' + inj_str_val
327 
328         print("Setting oven temperature to: {:s}".format(ov_str_val))
329         print("Setting injector temperature to: {:s}".format( inj_str_val))
330 
331         base_str = base_str[:4*ov_ind] + ov_str_val + base_str[4*(ov_ind + ...
                1) -1:]
332         base_str = base_str[:4*inj_ind] + inj_str_val + ...
                base_str[4*(inj_ind + 1) -1:]
333 
334         b_str = base_str.encode()
335 
336         ser = self.ser_conn
```

```
337             lock = self.ser_lock
338
339             with lock:
340                 ser.flushInput()
341                 ser.flushOutput()
342
343                 time.sleep(ser_delay)
344
345                 _ = ser.write(b_str)
346
347     def on_play_btn(self):
348         if self.data_paused:
349             self.data_rover_thread.un_pause()
350             self.data_paused = False
351             return
352         elif self.data_running:
353             return
354
355         rr = self.options['data_samp_rate']
356         sp = 1 / rr
357         ep = self.options['epsilon_time']
358
359         self.data_running= True
360         self.run_number += 1
361         self.gc.inc_run_num_()
362
363         self.curr_to_prev_()
364         self.gc.curr_to_prev_()
365
366         gcl = self.gc_lock
367         self.gc_cond = threading.Condition(gcl)
368
369         gc = self.gc
370         condition = self.gc_cond
371         _ind = self.options['indices']
372         self.data_rover_thread = GCData(gc, condition, _ind, args = ( sp, ...
                ep ) )
373
374         rsp = self.options['plot_refresh_rate']
375         lock = self.curr_data_frame_lock
376         self.receiver_thread = GCReceiver(self, lock, gc, condition, args ...
                = ( rsp, ep ))
377
378         self.data_rover_thread.start()
379         self.receiver_thread.start()
380
381         self.plotter_thread = GCPlotter(self, args=(rsp,ep))
382         self.plotter_thread.start()
383
384     def on_paus_btn(self):
385         gc_thread = self.data_rover_thread
386
387         if not self.data_paused:
388             gc_thread.pause()
389             self.data_paused = True
390
391     def on_stop_btn(self):
```

```
392         if self.data_running:
393             self.stop_data_coll_()
394 
395     def stop_data_coll_(self):
396         self.plotter_thread.stop()
397         self.plotter_thread.join()
398 
399         self.receiver_thread.stop()
400         self.receiver_thread.join()
401 
402         self.data_rover_thread.stop()
403         self.data_rover_thread.join()
404 
405         self.data_running = False
406 
407         if self.curr_data_frame_lock.locked():
408             self.curr_data_frame_lock.release()
409 
410         print('Message: Stopped data collection threads.')
411 
412     def on_plot_btn(self):
413         if self.data_running:
414             self.stop_data_coll_()
415 
416         self.panel_detector.update_curr_data_()
417         self.panel_detector.draw()
418 
419     def on_clr_btn(self):
420         if self.data_running:
421             self.stop_data_coll()
422 
423         if self.run_number > 0:
424             with self.curr_data_frame_lock:
425                 self.update_curr_data_()
426                 self.prev_data.append(self.curr_data)
427 
428         _dims = self.gc.get_dims()
429         with self.curr_data_frame_lock:
430             self.curr_data = np.zeros((_dims,0))
431 
432     '''
433     Formatting/building functions
434     '''
435     def build_figure_(self):
436         self.split_vert_()
437         self.set_up_menu_bar_()
438 
439     def split_vert_(self):
440         splitter = GCSplitter(self)
441 
442         self.panel_detector = DetectorPanel(splitter)
443         self.panel_config = ControlPanel(splitter)
444 
445         splitter.SplitVertically(self.panel_config , self.panel_detector, ...
                self.options['sash_size'])
446 
447         self.SetSizer(wx.BoxSizer(wx.HORIZONTAL))
```

```
448         self.GetSizer().Add(splitter, 1, wx.EXPAND)
449
450     def set_up_menu_bar_(self):
451         menubar = GCMenuBar(self)
452         self.SetMenuBar(menubar)
453
454     '''
455     Menu events
456     '''
457     def on_quit(self , err):
458         if self.data_running:
459             self.stop_data_coll_()
460
461         self.Close()
462
463     def on_saveas(self, err):
464         _saveas_gc_window = SaveasGC( self, self.options)
465
466     def on_open(self, err):
467         _open_window = OpenWindow(self, self.options)
468
469     def on_png_save(self, err):
470         _saveas_png_window = SaveasPNG(self, self.options)
471
472     def on_jpg_save(self, err):
473         _saveas_jpg_window = SaveasJPG(self, self.options)
474
475     def on_previous_set(self, err):
476         self.prev_to_curr_()
477
478     def on_data_integrate(self, err):
479         if not self.data_running:
480             ans = self.gc.integrate_volt()
481             print("The integral of voltage over the sampling period is: ")
482             print(ans)
483             self.gc.calc_cumsum_into_area_()
484             self.update_curr_data_()
485
486     def on_data_normalize(self, err):
487         if not self.data_running:
488             self.gc.normalize_volt_()
489             self.update_curr_data_()
490
491             self.panel_detector.update_curr_data_()
492             self.panel_detector.draw()
493
494     def on_clean_time(self, err):
495         if not self.data_running:
496             self.gc.clean_time_()
497             self.update_curr_data_()
498
499             self.panel_detector.update_curr_data_()
500             self.panel_detector.draw()
501
502     def on_fill(self, err):
503         if not self.data_running:
504             self.panel_detector.fill_under_()
```

```
505
506     def on_label_peaks(self, err):
507         if not self.data_running:
508             areas = self.gc.integrate_peaks()
509             # list of (x,y) pairs
510             maximas = self.gc.get_peak_local_maximas()
511
512             self.panel_detector.update_curr_data_()
513             self.panel_detector.label_peaks_(areas, maximas)
514
515     def on_mov_mean(self, err):
516         if not self.data_running:
517             _w = self.options['window']
518             self.gc.mov_mean_(_w)
519             self.update_curr_data_()
520
521             self.panel_detector.update_curr_data_()
522             self.panel_detector.draw()
523
524     '''
525     Defaults overridden
526     '''
527     def __del__(self):
528         pass
529
530     def main(self):
531         pass
532
533 # SplitterWindow
534 class GCSplitter(wx.SplitterWindow):
535     '''
536     parent is GCFrame
537     '''
538     def __init__(self, parent):
539         ops = parent.options
540         fs = ops['frame_size']
541         wx.SplitterWindow.__init__(self, parent, ...
                id=wx.ID_ANY,pos=wx.DefaultPosition , size=fs, style = ...
                wx.SP_BORDER, name='Diode Based Gas Chromatography' )
542         self.options = ops
543
544         self.parent = parent
545
546     def create_fonts(self):
547         font = wx.SystemSettings.GetFont(wx.SYS_SYSTEM_FONT)
548         font.SetPointSize(self.options['BODY_FONT_SIZE'])
549         header_font = wx.SystemSettings.GetFont(wx.SYS_SYSTEM_FONT)
550         header_font.SetPointSize(self.options['HEADER_FONT_SIZE'])
551
552         return {'font':font,'header_font':header_font}
553
554 # Popup Configuration Window
555 # Maybe wx.PopupWindow later?
556 class GCConfigPopup(wx.Frame):
557     def __init__(self, parent, text, type, variable_in_focus):
558         self.parent = parent
559         self.options = self.parent.options
```

```
560
561         self.prompt = text
562
563         self.default = variable_in_focus
564         self.type = type
565
566         self.fonts = self.create_fonts()
567
568         _p = parent
569         _id = wx.ID_ANY
570         _t = 'Configuration Popup Window'
571         _pos = wx.DefaultPosition
572         _s = (250, 100)
573         _sty = wx.DEFAULT_FRAME_STYLE | wx.TAB_TRAVERSAL
574         wx.Frame.__init__(self, _p, _id, _t, _pos, _s, _sty)
575
576     def create_fonts(self):
577         font = wx.SystemSettings.GetFont(wx.SYS_SYSTEM_FONT)
578         font.SetPointSize(self.options['BODY_FONT_SIZE'])
579         header_font = wx.SystemSettings.GetFont(wx.SYS_SYSTEM_FONT)
580         header_font.SetPointSize(self.options['HEADER_FONT_SIZE'])
581
582         return {'font':font,'header_font':header_font}
583
584     def create_frame_(self):
585         hf = self.fonts['header_font']
586         f = self.fonts['font']
587         b = self.options['BORDER']
588
589         hbox = wx.BoxSizer(wx.HORIZONTAL)
590
591         _t = self.prompt
592         _s = (100,50)
593         txt_prompt = wx.StaticText(self, label = _t, size= _s)
594
595         txt_prompt.SetFont(hf)
596         hbox.Add(txt_prompt, proportion=1)
597
598         _dv = self.default
599         _p = wx.DefaultPosition
600         _s = (50,50)
601         self.tc_val = wx.TextCtrl(self, value = _dv, pos = _p, size = _s)
602         self.tc_val.SetFont(hf)
603
604         hbox.Add(self.tc_val, proportion=1, border= b)
605
606         _id = wx.ID_ANY
607         _t = 'Enter'
608         _s = (50, 50)
609         self.btn_entr = wx.Button(self, id=_id, label=_t, size = _s)
610         self.Bind(wx.EVT_BUTTON, self.entrbtn_click_evt, self.btn_entr)
611
612         hbox.Add(self.btn_entr, proportion=1)
613
614         self.SetSizer(hbox)
615         self.Centre()
616         self.Show()
```

```
617
618     def entrbtn_click_evt(self, event):
619         pass
620
621 #DirectoryWindow(s)
622 class DirectoryWindow(wx.Frame):
623     def __init__(self, parent, ops):
624         self.parent = parent
625         self.cwd = os.getcwd()
626
627         self.options = ops
628         self.fonts = self.create_fonts()
629
630         _p = parent
631         _id = wx.ID_ANY
632         _t = wx.EmptyString
633         _pos = wx.DefaultPosition
634         _s = (600, 600)
635         _sty = wx.DEFAULT_FRAME_STYLE | wx.TAB_TRAVERSAL
636         wx.Frame.__init__(self, _p, _id, _t, _pos, _s, _sty)
637
638         self.create_frame_()
639         self.update_cwd()
640
641     def create_fonts(self):
642         font = wx.SystemSettings.GetFont(wx.SYS_SYSTEM_FONT)
643         font.SetPointSize(self.options['BODY_FONT_SIZE'])
644         header_font = wx.SystemSettings.GetFont(wx.SYS_SYSTEM_FONT)
645         header_font.SetPointSize(self.options['HEADER_FONT_SIZE'])
646
647         return {'font':font,'header_font':header_font}
648
649     def create_frame_(self):
650         vbox = wx.BoxSizer(wx.VERTICAL)
651         vbox.Add((-1,self.options['EXTRA_SPACE']))
652
653         nav_hbox = self.create_nav_hbox()
654
655         vbox.Add(nav_hbox, border = self.options['BORDER'])
656
657         self.list_box = self.create_cwd_listbox()
658
659         vbox.Add(self.list_box, border = self.options['BORDER'])
660
661         name_hbox = self.create_name_and_enter()
662         vbox.Add(name_hbox)
663
664         self.SetSizer(vbox)
665         self.Centre()
666         self.Show()
667
668     def create_nav_hbox(self):
669         nav_menu_hbox = wx.BoxSizer(wx.HORIZONTAL)
670         nav_menu_hbox.Add((self.options['EXTRA_SPACE'],-1))
671
672         global imdir
673         bmp = wx.Bitmap(imdir + '/btn_back_im_20p.png', wx.BITMAP_TYPE_ANY)
```

```
674
675         self.btn_bck = wx.BitmapButton(self, id=wx.ID_ANY,bitmap=bmp, size ...
                = (45,40))
676         self.Bind(wx.EVT_BUTTON, self.bckbtn_click_evt, self.btn_bck)
677
678         nav_menu_hbox.Add(self.btn_bck)
679         #nav_menu_hbox.Add((EXTRA_SPACE, -1))
680         self.tc_cwd = wx.TextCtrl(self, value = self.cwd, ...
                pos=wx.DefaultPosition, size=(500,40))
681         self.tc_cwd.SetFont(self.fonts['font'])
682
683         nav_menu_hbox.Add(self.tc_cwd, proportion=1, border = ...
                self.options['BORDER'])
684
685         return nav_menu_hbox
686
687     def create_cwd_listbox(self):
688         self.cwd_list = os.listdir(self.cwd)
689
690         self.list_box = wx.ListBox(self, size = (550,400), choices = ...
                self.cwd_list, style=wx.LB_SINGLE)
691
692         self.Bind(wx.EVT_LISTBOX_DCLICK, self.basic_cwdlist_dclick_evt, ...
                self.list_box)
693
694         return self.list_box
695
696     def create_name_and_enter(self):
697         font = wx.SystemSettings.GetFont(wx.SYS_SYSTEM_FONT)
698         font.SetPointSize(self.parent.options['BODY_FONT_SIZE'])
699
700         hbox = wx.BoxSizer(wx.HORIZONTAL)
701
702         self.tc_name = wx.TextCtrl(self, value = wx.EmptyString, pos = ...
                wx.DefaultPosition, size = (400,40))
703         self.tc_name.SetFont(font)
704
705         hbox.Add(self.tc_name, proportion=1, border= ...
                self.parent.options['BORDER'])
706
707
708         self.btn_entr = wx.Button(self, id=wx.ID_ANY, ...
                label=wx.EmptyString, size = (150,40))
709         self.Bind(wx.EVT_BUTTON, self.entrbtn_click_evt, self.btn_entr)
710
711         hbox.Add(self.btn_entr)
712
713         return hbox
714
715     def entrbtn_click_evt(self, event):
716         pass
717
718     def basic_cwdlist_dclick_evt(self, event):
719         index = event.GetSelection()
720         choice = self.cwd_list[index]
721
722         is_dir = os.path.isdir(choice)
```

```
723         if is_dir:
724             try:
725                 os.chdir(choice)
726                 self.cwd = os.getcwd()
727                 self.cwd_list = os.listdir(self.cwd)
728                 self.update_cwd()
729                 return
730 
731             except Exception:
732                 pass
733 
734         filename, extension = os.path.splitext(choice)
735 
736         self.update_namectl_to_dclick(choice)
737 
738         self.spec_cwdlist_dclick_evt(choice, filename, extension)
739 
740     def spec_cwdlist_dclick_evt(self,  choice, filename, extension):
741         pass
742 
743     def bckbtn_click_evt(self, event):
744         os.chdir('..')
745         self.cwd = os.getcwd()
746         self.cwd_list = os.listdir(self.cwd)
747         self.update_cwd()
748 
749     def update_cwd(self):
750         self.list_box.Clear()
751         self.list_box.Append(self.cwd_list)
752 
753         self.tc_cwd.Clear()
754         self.tc_cwd.write(self.cwd)
755 
756         self.Show()
757 
758     def update_namectl_to_dclick(self, choice):
759         self.tc_name.Clear()
760         self.tc_name.write(choice)
761 
762 class SaveasWindow(DirectoryWindow):
763     def __init__(self, parent, ops):
764         str = 'Save As'
765         DirectoryWindow.__init__(self,parent, ops)
766         self.SetTitle(str)
767         self.btn_entr.SetLabel(str)
768 
769     def entrbtn_click_evt(self, event):
770         pass
771 
772 class SaveasGC(SaveasWindow):
773     def __init__(self, parent, ops):
774         SaveasWindow.__init__(self, parent, ops)
775 
776         self.SetTitle('Save Session As GC')
777         self.btn_entr.SetLabel('Save as .gc')
778         #Get important parameters
779         self.parent = parent
```

```
780         self.options = ops
781
782     def entrbtn_click_evt(self, event):
783         name = self.tc_name.GetValue()
784         self.save_gc(name)
785
786     def save_gc(self, name):
787         _date = time.strftime('%d %m %Y (MM/DD/YYYY)', time.localtime())
788         date_str = 'Current date: ' + _date + ' '
789         _time = time.strftime('%H:%M:%S',time.localtime())
790         time_str = 'Time at save: ' + _time
791
792         _l = self.parent.curr_data_frame_lock
793         with _l:
794             data = self.parent.get_curr_data_copy()
795
796         _ind = self.options['indices']
797         _vi = _ind['v']
798         _ai = _ind['a']
799         _ti = _ind['t']
800         _dti = _ind['dt']
801
802         data_dict = ...
                {'v':data[_vi],'a':data[_ai],'t':data[_ti],'dt':data[_dti]}
803
804         curr_data = self.jsonify_data(data_dict)
805
806         prev_data = []
807
808         list_data = self.parent.get_prev_data_copy()
809         for data in list_data:
810             data_dict = ...
                    {'v':data[_vi],'a':data[_ai],'t':data[_ti],'dt':data[_dti]}
811             _jsond  = self.jsonify_data(data_dict)
812             prev_data.append(_jsond)
813
814         if name[-3:] != '.gc':
815             name = name + '.gc'
816
817         run_num = self.parent.run_number
818
819         # Format of JSON .gc filetype
820         curr_session = {
821         'Date' : date_str ,
822         'Time' : time_str ,
823         'Current Data' : curr_data ,
824         'Run Number' : run_num,
825          'Previous Data': prev_data
826         }
827
828         with codecs.open(name , 'w', encoding='utf-8') as json_file:
829             json.dump(curr_session, json_file, separators =(',',':'),indent=4)
830
831         self.Close()
832     '''
833     dict(numpys)->dict(lists)
834     '''
```

```
835     def jsonify_data(self, numpy_dict):
836         json_dict = {}
837         json_dict.update((key,val.tolist()) for key,val in ...
                numpy_dict.items()  )
838 
839         return json_dict
840 
841     '''
842     dict(lists)->dict(numpys)
843     '''
844     def reverse_jsonify(self, json_dict):
845         numpy_dict = {}
846         numpy_dict.update((key, np.array(val)) for key,val in ...
                json_dict.items() )
847         return numpy_dict
848 
849 class SaveasPNG(SaveasWindow):
850     def __init__(self, parent, ops):
851         SaveasWindow.__init__(self, parent, ops)
852 
853         self.SetTitle('Save Current Figure As PNG')
854         self.btn_entr.SetLabel('Save as .png')
855         self.figure = self.parent.get_figure()
856 
857     def entrbtn_click_evt(self, event):
858         name = self.tc_name.GetValue()
859         self.save_png(name)
860         self.Close()
861 
862     def save_png(self, name):
863         if name[-4:] == '.png':
864             self.figure.savefig(name )
865         else:
866             self.figure.savefig(name + '.png')
867 
868 class SaveasJPG(SaveasWindow):
869     def __init__(self, parent, ops):
870         SaveasWindow.__init__(self, parent, ops)
871 
872         self.SetTitle('Save Current Figure As JPEG')
873         self.btn_entr.SetLabel('Save as .jpg')
874         self.figure = self.parent.get_figure()
875 
876     def entrbtn_click_evt(self, event):
877         name = self.tc_name.GetValue()
878         self.save_jpg(name)
879         self.Close()
880 
881     def save_jpg(self, name):
882         if self.is_jpg(name):
883             self.figure.savefig(name)
884         else:
885             self.figure.savefig(name + '.jpg')
886 
887     def is_jpg(self, n):
888         if n[-4:] == '.jpg':
889             return True
```

```
890         return False
891
892 class OpenWindow(DirectoryWindow):
893     def __init__(self, parent, ops):
894         str = 'Open GC File'
895         super().__init__( parent, ops)
896         self.SetTitle(str)
897         self.btn_entr.SetLabel('Open as .gc')
898
899     def spec_cwdlist_dclick_evt(self,  choice, filename, extension):
900         name = self.tc_name.GetValue()
901         if self.is_gc(name):
902             self.open_gc(name)
903             self.Close()
904
905     def open_gc(self, name):
906         self.parent.set_frame_from_session_(name)
907
908     def is_gc(self, name):
909         if name[-3:] == '.gc':
910             return True
911         return False
912
913 # Threads
914 class GCTemperature(Thread):
915     def __init__(self, frame, serial_connection, serial_lock, *args, ...
        **kwargs):
916         super(GCTemperature, self).__init__()
917
918         self.frame = frame
919
920         self.sp = kwargs['args'][0] #sampling_period
921         self.ep = kwargs['args'][1] #epsilon
922
923         self._stop_event = threading.Event()
924
925         self.ser_conn = serial_connection
926         self.ser_lock = serial_lock
927
928         self.oven_temp = None
929         self.last_oven_temp = None
930         self.oven_val_change = True
931         self.oven_stc_txt = self.frame.panel_config.str_ov_fdbk_val
932
933         self.inj_temp = None
934         self.last_inj_temp = None
935         self.oven_val_change = True
936         self.inj_stc_txt = self.frame.panel_config.str_inj_fdbk_val
937
938         self.last_update_time_raw = None
939
940         self.ov_location = self.frame.options['OVEN_INDEX']
941         self.inj_location = self.frame.options['INJ_INDEX']
942
943
944     def stop(self):
945         self._stop_event.set()
```

```
946
947     def stopped(self):
948         return self._stop_event.is_set()
949
950     def run(self):
951         sampling_period = self.sp
952         epsilon = self.ep
953
954         o_l = self.ov_location
955         d_l = self.inj_location
956
957         t_last = time.time()
958
959         while not self.stopped():
960             t_curr= time.time()
961             while (t_curr - epsilon -t_last < sampling_period):
962                 time.sleep(.1)
963                 t_curr = time.time()
964
965             t_last = t_curr
966
967             bit_response = self.query_temp()
968
969             temperatures = self.parse_response(bit_response)
970
971             self.last_oven_temp = self.oven_temp
972             self.last_inj_temp = self.inj_temp
973
974             self.oven_temp = float(temperatures[0])
975             self.inj_temp = float(temperatures[1])
976
977             if self.last_oven_temp == self.oven_temp:
978                 self.oven_val_change = False
979             else:
980                 self.oven_val_change = True
981
982             if self.last_inj_temp == self.inj_temp:
983                 self.inj_val_change = False
984             else:
985                 self.inj_val_change = True
986
987             self.last_update_time_raw = time.time()
988
989             if self.inj_val_change and self.oven_val_change:
990                 func = self.set_both_txt_ctrls
991                 args = temperatures
992                 wx.CallAfter(func, args)
993
994             elif self.oven_val_change:
995                 ov_str = temperatures[0]
996
997                 func = self.oven_stc_txt.SetLabel
998                 args = ov_str
999
1000                wx.CallAfter(func, args)
1001
1002            elif self.inj_val_change:
```

```
1003                inj_str = temperatures[1]
1004
1005                func = self.inj_stc_txt.SetLabel
1006                args = inj_str
1007                wx.CallAfter(func, args)
1008
1009    def query_temp(self):
1010        ser_delay = self.frame.options['SER_DELAY']
1011        # From c library. Defined in c_constants.py
1012        #READ_TMP_CMD_STR = '000 000 000 000'
1013        _str = self.frame.options['READ_TMP_CMD_STR']
1014        b_str = _str.encode()
1015
1016        bit_response = []
1017
1018        ser = self.ser_conn
1019
1020        lock = self.ser_lock
1021        with lock:
1022            ser.flushInput()
1023            ser.flushOutput()
1024
1025            time.sleep(ser_delay)
1026
1027            _ = ser.write(b_str)
1028            while ser.in_waiting == 0:
1029                time.sleep(ser_delay)
1030
1031            while ser.in_waiting > 0:
1032                line = ser.readline()
1033                bit_response.append(line)
1034
1035        return bit_response
1036
1037
1038    def parse_response(self, resp):
1039        o_l = self.ov_location
1040        i_l = self.inj_location
1041
1042        str_resp = [item.decode() for item in resp]
1043        str_resp = [item.strip('\r\n') for item in str_resp]
1044
1045        str_resp = str_resp[0]
1046
1047        _ind = self.ov_location
1048        ov_tmp = str_resp[4*_ind: 4*(_ind + 1) - 1]
1049
1050        _ind = self.inj_location
1051        inj_tmp = str_resp[4*_ind: 4*(_ind + 1) - 1]
1052
1053        temps = [ov_tmp, inj_tmp]
1054
1055        return temps
1056
1057    def set_both_txt_ctrls(self, temps):
1058        ov_str = temps[0]
1059        inj_str = temps[1]
```

```
1060          self.oven_stc_txt.SetLabel(ov_str)
1061          self.inj_stc_txt.SetLabel(inj_str)
1062
1063  class GCPlotter(Thread):
1064      def __init__(self, frame, *args, **kwargs):
1065          super(GCPlotter, self).__init__()
1066
1067          self.frame = frame
1068          #self.curr_data_lock = lock
1069
1070          self.sp = kwargs['args'][0]
1071          self.ep = kwargs['args'][1]
1072
1073          self._stop_event = threading.Event()
1074
1075      def stop(self):
1076          self._stop_event.set()
1077
1078      def stopped(self):
1079          return self._stop_event.is_set()
1080
1081      def run(self):
1082          sampling_period = self.sp
1083          epsilon = self.ep
1084
1085          t_last = time.time()
1086          while not self.stopped():
1087              t_curr= time.time()
1088              while (t_curr - epsilon -t_last < sampling_period):
1089                  time.sleep(.1)
1090                  t_curr = time.time()
1091
1092              t_last = t_curr
1093
1094              self.frame.panel_detector.update_curr_data_()
1095              func = self.frame.panel_detector.draw
1096              wx.CallAfter(func)
1097
1098  class GCReceiver(Thread):
1099      def __init__(self, frame, lock, gc, condition, *args, **kwargs):
1100          super(GCReceiver, self).__init__()
1101
1102          self.sp = kwargs['args'][0]
1103          self.ep = kwargs['args'][1]
1104          self.time_out = 1
1105
1106          self._stop_event = threading.Event()
1107
1108          self.frame = frame
1109          self.gc = gc
1110          self.data_lock = lock
1111
1112          self.gc_cond = condition
1113
1114      def stop(self):
1115          self._stop_event.set()
1116
```

```
1117      def stopped(self):
1118          return self._stop_event.is_set()
1119
1120      def run(self):
1121          sampling_period = self.sp
1122          epsilon = self.ep
1123          to = self.time_out
1124
1125          t_last = time.time()
1126          while not self.stopped():
1127              t_curr= time.time()
1128              while (t_curr - epsilon -t_last < sampling_period):
1129                  time.sleep(.01)
1130                  t_curr = time.time()
1131
1132              t_last = t_curr
1133              with self.gc_cond:
1134                  val = self.gc_cond.wait(to)
1135
1136              if val:
1137                  with self.gc_cond:
1138                      gc_d = self.gc.get_curr_data()
1139                  with self.data_lock:
1140                      self.frame.set_curr_data_(gc_d)
1141
1142              else:
1143                print("Error: Timeout on data reception reached.")
1144
1145  class GCData(Thread):
1146      def __init__(self, gc, condition, indices, *args, **kwargs):
1147          super(GCData, self).__init__()
1148
1149          self.sp = kwargs['args'][0]
1150          self.ep = kwargs['args'][1]
1151          self.indices = indices
1152          self.gc = gc
1153          self._stop_event = threading.Event()
1154          self._pause_event = threading.Event()
1155
1156          self.condition = condition
1157          #
1158          self.avail = False
1159
1160      def stop(self):
1161          self._stop_event.set()
1162
1163      def stopped(self):
1164          return self._stop_event.is_set()
1165
1166      def pause(self):
1167          self._pause_event.set()
1168
1169      def un_pause(self):
1170          self._pause_event.clear()
1171
1172      def paused(self):
1173          return self._pause_event.is_set()
```

```
1174
1175      def is_avail(self):
1176          return self.avail
1177
1178      def run(self):
1179          t_last = time.time()
1180
1181          sampling_period = self.sp
1182          epsilon = self.ep
1183          dims = self.gc.get_dims()
1184
1185          while not self.stopped():
1186              while self.paused():
1187                  time.sleep(.01)
1188                  if self.stopped():
1189                      return
1190
1191              t_curr= time.time()
1192              while (t_curr - epsilon -t_last < sampling_period):
1193                  time.sleep(.01)
1194                  t_curr = time.time()
1195
1196              v = self.gc.measure_voltage()
1197
1198              dt = t_curr - t_last
1199              t = t_curr
1200              t_last = t_curr
1201
1202              new = np.zeros((dims, 1))
1203
1204              ind = self.indices
1205              _vi = ind['v']
1206              _dti = ind['dt']
1207              _ti = ind['t']
1208
1209              new[_vi] = v
1210              new[_dti] = dt
1211              new[_ti] = t
1212              # a default zero
1213
1214              with self.condition:
1215                  old = self.gc.get_curr_data()
1216
1217              new = np.append(old, new, axis=1)
1218
1219              with self.condition:
1220                  self.gc.set_curr_data_(new)
1221                  self.condition.notify_all()
1222
1223  #Panels
1224  class DetectorPanel(wx.Panel):
1225      def __init__(self, parent):
1226          wx.Panel.__init__(self, parent, id = wx.ID_ANY, pos = ...
                wx.DefaultPosition, style = wx.TAB_TRAVERSAL)
1227
1228          self.parent = parent
1229          self.gcframe = parent.parent
```

```
1230            self.options = self.parent.options
1231            self.indices = self.options['indices']
1232            self.units_str = self.options['units_str']
1233
1234            self.fonts = parent.create_fonts()
1235
1236            self.create_panel()
1237
1238        def create_panel(self):
1239            f = self.fonts['font']
1240            hf = self.fonts['header_font']
1241            b = self.options['BORDER']
1242            es = self.options['EXTRA_SPACE']
1243            bfs = self.options['BODY_FONT_SIZE']
1244
1245            vbox = wx.BoxSizer(wx.VERTICAL)
1246
1247            str_det_panel = wx.StaticText(self, label = 'Detector')
1248            str_det_panel.SetFont(hf)
1249
1250            vbox.Add(str_det_panel, flag=wx.EXPAND|wx.LEFT|wx.RIGHT|wx.TOP, ...
                    border = b)
1251            vbox.Add((-1,es))
1252
1253            hbox = wx.BoxSizer(wx.HORIZONTAL)
1254
1255            self.vbox2 = wx.BoxSizer(wx.VERTICAL)
1256
1257            hbox2 = self.create_control_box()
1258
1259            self.vbox2.Add(hbox2, border=b)
1260            self.vbox2.Add((-1,20))
1261
1262            self.create_plot_btn_()
1263            self.create_clr_btn_()
1264
1265            hbox.Add(self.vbox2, border = b)
1266            hbox.Add((es, -1))
1267
1268            self.create_figure_()
1269
1270            hbox.Add(self.canvas)
1271
1272            vbox.Add(hbox, border = b)
1273            self.SetSizer(vbox)
1274            self.Fit()
1275
1276        def create_plot_btn_(self):
1277            f = self.fonts['font']
1278            b = self.options['BORDER']
1279            es = self.options['EXTRA_SPACE']
1280
1281            self.btn_plot = wx.Button(self, label = 'plot', size = (200,50))
1282            self.btn_plot.SetFont(f)
1283            self.btn_plot.SetCursor(wx.Cursor(wx.CURSOR_DEFAULT))
1284
1285            self.Bind(wx.EVT_BUTTON, self.plot_btn_evt,self.btn_plot)
```

```
1286
1287            self.vbox2.Add(self.btn_plot, border= b)
1288            self.vbox2.Add((-1,es))
1289
1290        def create_clr_btn_(self):
1291            f = self.fonts['font']
1292            b = self.options['BORDER']
1293            es = self.options['EXTRA_SPACE']
1294
1295            self.btn_clr = wx.Button(self, label = 'clear', size = (200,50))
1296            self.btn_clr.SetFont(f)
1297            self.btn_clr.SetCursor(wx.Cursor(wx.CURSOR_DEFAULT))
1298
1299            self.Bind(wx.EVT_BUTTON,self.clear_plot_btn_evt,self.btn_clr)
1300
1301            self.vbox2.Add(self.btn_clr, border= b)
1302            self.vbox2.Add((-1,es))
1303
1304        def create_figure_(self):
1305            self.figure = Figure()
1306            self.axes = self.figure.add_subplot(111)
1307            _xstr = self.units_str['x-axis']
1308            _ystr = self.units_str['y-axis']
1309            self.axes.set_xlabel(_xstr)
1310            self.axes.set_ylabel(_ystr)
1311
1312            self.canvas = FigureCanvas(self, -1, self.figure)
1313
1314        def create_control_box(self):
1315            b = self.options['BORDER']
1316            es = self.options['EXTRA_SPACE']
1317
1318            hbox = wx.BoxSizer(wx.HORIZONTAL)
1319
1320            bmp = wx.Bitmap(imdir + '/play_btn_20p.png',wx.BITMAP_TYPE_ANY)
1321            self.btn_ply = wx.BitmapButton(self, id=wx.ID_ANY, bitmap=bmp, ...
                    size = (50,50))
1322            self.Bind(wx.EVT_BUTTON, self.ply_btn_evt, self.btn_ply)
1323
1324            bmp = wx.Bitmap(imdir + '/paus_btn_20p.png',wx.BITMAP_TYPE_ANY)
1325            self.btn_paus = wx.BitmapButton(self, id=wx.ID_ANY, bitmap=bmp, ...
                    size = (50,50))
1326            self.Bind(wx.EVT_BUTTON, self.paus_btn_evt, self.btn_paus)
1327
1328            bmp = wx.Bitmap(imdir + '/stop_btn_20p.png',wx.BITMAP_TYPE_ANY)
1329            self.btn_stp = wx.BitmapButton(self, id=wx.ID_ANY, bitmap=bmp, ...
                    size = (50,50))
1330            self.Bind(wx.EVT_BUTTON, self.stp_btn_evt, self.btn_stp)
1331
1332            hbox.Add(self.btn_ply, border = b)
1333            hbox.Add((es,-1))
1334            hbox.Add(self.btn_paus, border =b)
1335            hbox.Add((es,-1))
1336            hbox.Add(self.btn_stp, border =b)
1337            hbox.Add((es,-1))
1338
1339            return hbox
```

```
1340
1341      def label_peaks_(self, areas, maximas):
1342          num_pts = len(areas)
1343
1344          cd = self.get_curr_data()
1345          ind = self.gcframe.options['indices']
1346          _vi = ind['v']
1347          _ti = ind['t']
1348
1349          v = cd[_vi]
1350          t = cd[_ti]
1351
1352          accuracy = self.options['area_accuracy']
1353
1354          for i in range(0,num_pts):
1355              _astr = str(areas[i])
1356              _text = 'Relative area:\n' + _astr[:accuracy]
1357              max_index , max_val  = maximas[i]
1358              _x = t[max_index]
1359              _y = v[max_index]
1360              self.axes.annotate(_text, xy= (_x,_y), xytext=(20,-20), ...
                      xycoords='data', textcoords = 'offset pixels')
1361
1362          _xstr = self.units_str['x-axis']
1363          _ystr = self.units_str['y-axis']
1364          self.axes.set_xlabel(_xstr)
1365          self.axes.set_ylabel(_ystr)
1366
1367          func = self.canvas.draw
1368          wx.CallAfter(func)
1369
1370      def fill_under_(self):
1371          cd = self.get_curr_data()
1372          _vi = self.indices['v']
1373          _ti = self.indices['t']
1374          v = cd[_vi]
1375          t = cd[_ti]
1376          self.axes.fill_between(t,v)
1377          func = self.canvas.draw
1378          wx.CallAfter(func)
1379
1380      def draw(self):
1381          _vi = self.indices['v']
1382          _ti = self.indices['t']
1383
1384          if self.curr_data.size != 0:
1385              self.axes.cla()
1386              self.axes.plot(self.curr_data[_ti], self.curr_data[_vi])
1387
1388              _xstr = self.units_str['x-axis']
1389              _ystr = self.units_str['y-axis']
1390              self.axes.set_xlabel(_xstr)
1391              self.axes.set_ylabel(_ystr)
1392
1393              func = self.canvas.draw
1394              wx.CallAfter(func)
1395          else:
```

```
1396             print("Error: Empty curr_data")
1397
1398     def ply_btn_evt(self, event):
1399         self.gcframe.on_play_btn()
1400
1401     def paus_btn_evt(self, event):
1402         self.gcframe.on_paus_btn()
1403
1404     def stp_btn_evt(self, event):
1405         self.gcframe.on_stop_btn()
1406
1407     def plot_btn_evt(self, event):
1408         self.gcframe.on_plot_btn()
1409
1410     def update_curr_data_(self):
1411         with self.gcframe.curr_data_frame_lock:
1412             self.curr_data = self.gcframe.get_curr_data_copy()
1413
1414     def clear_plot_btn_evt(self, event):
1415         self.gcframe.on_clr_btn()
1416         self.axes.cla()
1417         self.canvas.draw()
1418
1419     def get_figure(self):
1420         return self.figure
1421
1422     def get_curr_data(self):
1423         return self.curr_data
1424
1425     def __del__(self):
1426         pass
1427
1428 class ControlPanel( wx.Panel ):
1429     def __init__( self, parent ):
1430         wx.Panel.__init__ (self, parent, id = wx.ID_ANY, pos = ...
                 wx.DefaultPosition, style = wx.TAB_TRAVERSAL )
1431
1432         self.parent = parent
1433         self.gcframe = parent.parent
1434         self.options = self.parent.options
1435
1436         self.fonts = parent.create_fonts()
1437
1438         self.create_panel()
1439
1440     def create_panel(self):
1441         f = self.fonts['font']
1442         hf = self.fonts['header_font']
1443         b = self.options['BORDER']
1444         es = self.options['EXTRA_SPACE']
1445         bfs = self.options['BODY_FONT_SIZE']
1446
1447         self.vbox = wx.BoxSizer(wx.VERTICAL)
1448
1449         # Temperature header
1450         self.build_static_header_()
1451
```

```
1452            #Oven temp
1453            hbox_ov_set = self.build_oven_static_text_one()
1454
1455            self.tc_ov_set = wx.TextCtrl(self)
1456            hbox_ov_set.Add(self.tc_ov_set, proportion=1)
1457            self.Bind(wx.EVT_TEXT , self.oven_set , self.tc_ov_set)
1458
1459            self.vbox.Add(hbox_ov_set, flag=wx.LEFT|wx.TOP,border =b)
1460
1461            hbox_ov_fdbk = self.build_oven_static_text_two()
1462
1463            self.str_ov_fdbk_val = wx.StaticText(self, label = 'N/A')
1464            self.str_ov_fdbk_val.SetFont(f)
1465            hbox_ov_fdbk.Add(self.str_ov_fdbk_val)
1466
1467            self.vbox.Add(hbox_ov_fdbk, flag=wx.LEFT|wx.TOP,border =b)
1468
1469            self.vbox.Add((-1,es))
1470
1471            #injector temp
1472            hbox_inj_set = self.build_inj_static_text_one()
1473
1474            self.tc_inj_set = wx.TextCtrl(self)
1475            hbox_inj_set.Add(self.tc_inj_set, proportion = 1)
1476            self.Bind(wx.EVT_TEXT, self.inj_set , self.tc_inj_set)
1477
1478            self.vbox.Add(hbox_inj_set, flag=wx.LEFT|wx.TOP,border =b)
1479
1480            hbox_inj_fdbk = self.build_inj_static_text_two()
1481
1482            self.str_inj_fdbk_val = wx.StaticText(self, label = 'N/A')
1483            self.str_inj_fdbk_val.SetFont(f)
1484            hbox_inj_fdbk.Add(self.str_inj_fdbk_val)
1485
1486            self.vbox.Add(hbox_inj_fdbk, flag=wx.LEFT|wx.TOP,border =b)
1487
1488            self.SetSizer(self.vbox)
1489
1490
1491        def oven_set(self, event):
1492            ov_str = self.tc_ov_set.GetValue()
1493
1494            inj_str = self.tc_inj_set.GetValue()
1495
1496            if len(ov_str) != 0 and len(inj_str) != 0:
1497                self.gcframe.on_temp_txt_ctrl(ov_str, inj_str)
1498
1499        def inj_set(self, event):
1500            ov_str = self.tc_ov_set.GetValue()
1501            inj_str = self.tc_inj_set.GetValue()
1502            if len(ov_str) != 0 and len(inj_str) != 0:
1503                self.gcframe.on_temp_txt_ctrl(ov_str, inj_str)
1504
1505        def build_inj_static_text_two(self):
1506            hbox_inj_fdbk = wx.BoxSizer(wx.HORIZONTAL)
1507            str_inj_fdbk = wx.StaticText(self, label = 'Injector Temp. Reading ...
                (deg C): ')
```

```
1508            f = self.fonts['font']
1509            str_inj_fdbk.SetFont(f)
1510            hbox_inj_fdbk.Add(str_inj_fdbk)
1511
1512            return hbox_inj_fdbk
1513
1514        def build_inj_static_text_one(self):
1515            hbox_inj_set = wx.BoxSizer(wx.HORIZONTAL)
1516            str_inj_set = wx.StaticText(self, label = 'Set Injector Temp.: ')
1517            f = self.fonts['font']
1518            str_inj_set.SetFont(f)
1519            hbox_inj_set.Add(str_inj_set)
1520
1521            return hbox_inj_set
1522
1523        def build_oven_static_text_one(self):
1524            hbox_ov_set = wx.BoxSizer(wx.HORIZONTAL)
1525            str_ov_set = wx.StaticText(self,label='Set Oven Temp.: ')
1526            f = self.fonts['font']
1527            str_ov_set.SetFont(f)
1528            hbox_ov_set.Add(str_ov_set)
1529
1530            return hbox_ov_set
1531
1532        def build_oven_static_text_two(self):
1533            hbox_ov_fdbk = wx.BoxSizer(wx.HORIZONTAL)
1534            str_ov_fdbk = wx.StaticText(self, label = 'Oven Temp. Reading (deg ...
                C): ')
1535            f = self.fonts['font']
1536            str_ov_fdbk.SetFont(f)
1537            hbox_ov_fdbk.Add(str_ov_fdbk)
1538
1539            return hbox_ov_fdbk
1540
1541        def build_static_header_(self):
1542            #Temperature
1543            str_temp = wx.StaticText(self, label = 'Temperature')
1544            hf = self.fonts['header_font']
1545            str_temp.SetFont(hf)
1546            b = self.options['BORDER']
1547            self.vbox.Add(str_temp, ...
                    flag=wx.EXPAND|wx.LEFT|wx.RIGHT|wx.TOP,border =b)
1548
1549            es = self.options['EXTRA_SPACE']
1550            self.vbox.Add((-1,es))
1551
1552        def __del__( self ):
1553            pass
1554            # Virtual event handlers, overide them in your derived class
1555        def changeIntroPanel( self, event ):
1556            event.Skip()
1557
1558    # Menus
1559    class GCMenuBar(wx.MenuBar):
1560
1561        def __init__(self, parent):
1562            wx.MenuBar.__init__(self)
```

```
1563            self.parent = parent
1564
1565
1566            file_menu = self.create_file_menu()
1567            self.Append(file_menu,'&File')
1568
1569            edit_menu = self.create_edit_menu()
1570            self.Append(edit_menu,'&Edit')
1571
1572            data_menu = self.create_data_menu()
1573            self.Append(data_menu,'&Data')
1574
1575            grapher_menu = self.create_grapher_menu()
1576            self.Append(grapher_menu, '&Grapher')
1577
1578        def create_grapher_menu(self):
1579            grapher_menu = wx.Menu()
1580
1581            # grapher_menu.Append(wx.ID_ANY, '&Edit Axes')
1582            #
1583            # grapher_menu.AppendSeparator()
1584
1585            grapher_menu.Append(wx.ID_ANY, '&Save Image')
1586
1587            graph_menu_saveim = wx.Menu()
1588            _png_save = graph_menu_saveim.Append(wx.ID_ANY, '&.png')
1589            self.parent.Bind(wx.EVT_MENU, self.parent.on_png_save, _png_save)
1590
1591            _jpg_save = graph_menu_saveim.Append(wx.ID_ANY,'&.jpg')
1592            self.parent.Bind(wx.EVT_MENU, self.parent.on_jpg_save, _jpg_save)
1593
1594            grapher_menu.Append(wx.ID_ANY, '&Save Image As...', graph_menu_saveim)
1595
1596            _fill = grapher_menu.Append(wx.ID_ANY, '&Fill')
1597            self.parent.Bind(wx.EVT_MENU, self.parent.on_fill, _fill)
1598
1599            _label_peaks = grapher_menu.Append(wx.ID_ANY, '&Label Peaks')
1600            self.parent.Bind(wx.EVT_MENU, self.parent.on_label_peaks, ...
                    _label_peaks)
1601
1602            return grapher_menu
1603
1604        def create_data_menu(self):
1605            data_menu = wx.Menu()
1606
1607            prev_set = data_menu.Append(wx.ID_ANY, '&Previous Set')
1608            self.parent.Bind(wx.EVT_MENU, self.parent.on_previous_set, prev_set)
1609
1610            data_menu_ops = wx.Menu()
1611            integ = data_menu_ops.Append(wx.ID_ANY, '&Integrate')
1612            self.parent.Bind(wx.EVT_MENU, self.parent.on_data_integrate, integ )
1613
1614            norm = data_menu_ops.Append(wx.ID_ANY, '&Normalize')
1615            self.parent.Bind(wx.EVT_MENU, self.parent.on_data_normalize, norm)
1616
1617            clean_time = data_menu_ops.Append(wx.ID_ANY, '&Clean Time Axis')
1618            self.parent.Bind(wx.EVT_MENU, self.parent.on_clean_time, clean_time)
```

```
1619
1620            lpf = data_menu_ops.Append(wx.ID_ANY, '&Apply Low Pass Filter')
1621            self.parent.Bind(wx.EVT_MENU, self.parent.on_mov_mean, lpf)
1622
1623            data_menu.Append(wx.ID_ANY, '&Operations...', data_menu_ops)
1624
1625            return data_menu
1626
1627        def create_edit_menu(self):
1628            edit_menu = wx.Menu()
1629            edit_menu.Append(wx.ID_PAGE_SETUP, '&Settings')
1630
1631            return edit_menu
1632
1633        def create_file_menu(self):
1634            file_menu = wx.Menu()
1635            file_menu.Append(wx.ID_NEW, '&New')
1636            file_menu.Append(wx.ID_CLEAR, '&Clear')
1637
1638            item_open = file_menu.Append(wx.ID_OPEN, '&Open')
1639            self.parent.Bind(wx.EVT_MENU, self.parent.on_open, item_open)
1640
1641            file_menu.AppendSeparator()
1642
1643            file_menu.Append(wx.ID_SAVE, '&Save')
1644
1645            item_saveas =file_menu.Append( wx.ID_SAVEAS, '&Save as' )
1646
1647            self.parent.Bind(wx.EVT_MENU, self.parent.on_saveas, item_saveas)
1648
1649            file_menu.AppendSeparator()
1650
1651            item_quit = file_menu.Append(wx.ID_EXIT, '&Quit' , 'Quit application')
1652            self.parent.Bind(wx.EVT_MENU, self.parent.on_quit,item_quit)
1653
1654            return file_menu
1655
1656    if __name__ == '__main__':
1657        pass
```

C.8 install.sh

```
1       #!/bin/bash
2   #install.sh
3
4   # Root privileges check
5   if (( $EUID != 0 )); then
6       printf "\n""Please run as root (i.e. sudo ./install.sh)"
7       exit
8   fi
9
10  printf "%-120s\n" ...
        "-------------------------------------------------------------------------------"
11  printf "%-120s\n" "| Installation script to install the required ...
        dependencies for Gas-Chromatography on a new Raspberry Pi."
```

```
12 printf "%-120s\n" "| Instructions given at ...
       https://github.com/cgreen18/Gas-Chromatography/tree/master/Installation"
13 printf "%-120s\n" "| Tested on Raspberry Pi Model 3B+ w/ Raspbian 10 Buster"
14 printf "%-120s\n" "| Functions as intended as of 10 April 2020"
15 printf "%-120s\n" "| Conor Green and Matt McPartlan"
16 printf "%-120s\n" ...
       "-------------------------------------------------------------------------------------------------------------------"
17
18 sleep 10s
19
20 printf "\n\n\n\n\n\n\n\n\n\n"
21
22
23 echo -e "\x1B[31Errors if /home/pi/ does not exist\e[0m"
24 echo "Theres a general issue with \$USER becoming 'root' as opposed to 'pi'."
25 echo "This was fixed by hardcoding 'pi' user."
26 echo "There will be errors if the user is not 'pi' and there is no ...
       /home/pi/ directory."
27 echo "If you can fix this, please merge request or email me."
28
29 sleep 5s
30
31 printf "\n\n\n\n\n\n\n"
32
33 echo -e "\e[4mUpdating repos\e[0m"
34
35 cd ~
36 sudo apt-get update
37
38 echo -e "\e[4mInstalling and configuring virtual environments\e[0m"
39 # Installing virtualenv
40 sudo apt-get install python3-venv
41 sudo pip install virtualenv
42 echo -e '\nexport PATH="/home/$USER/.local/bin:$PATH"' >> /home/pi/.bashrc
43 echo -e '\nexport PATH="/home/pi/.local/bin:$PATH"' >> /home/pi/.bashrc
44 source "/home/pi/.bashrc"
45
46
47 echo -e "\e[4mCreating virtual environment: .gc_venv\e[0m"
48 # Creating gc Py3 virtualenv
49 cd /home/pi/
50 python3 -m venv --system-site-packages .gc_venv
51 source "/home/pi/.gc_venv/bin/activate"
52
53 echo -e "\e[4mInstalling dependencies\e[0m"
54 # Upgrade pip
55 pip install -U pip
56 # Install dependencies
57 pip install -U six wheel setuptools
58 apt-get -y install build-essential tk-dev libncurses5-dev libncursesw5-dev ...
       libreadline6-dev libdb5.3-dev libgdbm-dev libsqlite3-dev libssl-dev ...
       libbz2-dev libexpat1-dev liblzma-dev zlib1g-dev
59 sleep 10s
60 apt-get -y install dpkg-dev build-essential libjpeg-dev libtiff-dev ...
       libsdl1.2-dev libgstreamer-plugins-base0.10-dev libnotify-dev freeglut3 ...
       freeglut3-dev libwebkitgtk-dev libghc-gtk3-dev libwxgtk3.0-gtk3-dev
61 sleep 10s
```

```
apt-get -y install python3.7-dev
sleep 10s

echo -e "\e[4mAcquiring wxPython4.0.6 (requires internet connection)\e[0m"
# Get wxPython 4.0.6 to /home/pi
wget ...
    https://files.pythonhosted.org/packages/b9/8b/31267dd6d026a082faed35ec8d97522c0236f2e083bf15

cd /home/pi

echo -e "\e[4mUnpacking wxPython tar\e[0m"
tar -xf wxPython-4.0.7.post2.tar.gz

echo -e "\e[4mInstalling requirements\e[0m"
cd /home/pi/wxPython-4.0.7.post2
pip3 install -r requirements.txt

echo -e "\e[4mBuilding wxPython. Will take a long time ~4 hrs\e[0m"
sleep 10s
sleep 10s

python3 build.py build bdist_wheel
sleep 10s

echo -e "\e[4mInstalling final libraries: atlas, matplotlib, PyYAML\e[0m"
echo -e "\e[4mInstalling libatlas\e[0m"
apt-get install libatlas-base-dev

echo -e "\e[4mInstalling matplotlib\e[0m"
pip3 install matplotlib

echo -e "\e[4mInstalling PyYAML\e[0m"
pip3 install pyyaml
```

C.9 config.sh

```
    # !/bin/bash

printf "%-120s\n" ...
    "-----------------------------------------------------------------------------------------------------------------------"
printf "%-120s\n" "| Configuration script to install the required ...
    dependencies for Gas-Chromatography on a new Raspberry Pi."
printf "%-120s\n" "| Instructions given at ...
    https://github.com/cgreen18/Gas-Chromatography/blob/master/Installation/Configuration.md"
printf "%-120s\n" "| Tested on Raspberry Pi Model 3B+ w/ Raspbian 10 Buster"
printf "%-120s\n" "| Functions as intended as of 10 April 2020"
printf "%-120s\n" "| Conor Green and Matt McPartlan"
printf "%-120s\n" ...
    "-----------------------------------------------------------------------------------------------------------------------"

sleep 10s

printf "\n\n\n\n\n\n\n\n\n\n"

# Root privileges check
```

```
16 if (( $EUID != 0 )); then
17     printf "Please run as root (i.e. sudo ./config.sh)\n"
18     exit
19 fi
20 # to configure auto-start programs and allow SPI/SCI
21 printf "\n ""Enabling Serial in raspi-config"
22 # Enable SPI and SCI from raspi-config
23 raspi-config nonint do_serial 0
24 cat /boot/cmdline.txt
25
26 printf "Enabling I2C and SPI in raspi-config\n"
27 # Enable SPI and SCI from raspi-config
28 raspi-config nonint do_i2c 0
29 raspi-config nonint do_spi 0
30
31 printf "\n""Configuring 'gas_chromatograph.py' to autorun at startup via ...
       .bashrc\n"
32 # Default gc_venv
33 echo '\n#Probably bad style to have 5 lines in .bashrc but this RPi is ...
       dedicated to GC, so oh well.' >> /home/pi/.bashrc
34 echo '\nsource "/home/pi/gtest/bin/activate"' >> /home/pi/.bashrc
35 echo '\ncd /home/pi/Gas-Chromatography' >> /home/pi/.bashrc
36 echo '\ncat README.md' >> /home/pi/.bashrc
37 echo '\ncd /home/pi/Gas-Chromatography/GUI/current_version' >> ...
       /home/pi/.bashrc
38 echo '\npython3 gas_chromatography.py' >> /home/pi/.bashrc
```