iGEM 2016 Thesis

University of California, Santa Cruz

Henry Hinton

March 20, 2017

Contents

1 Background and Project Focus

Abstract

International Genetically Engineered Machine, or iGEM for short, is a student-oriented competition that allows students to bring a complex idea in bioengineering to fruition. Besides allowing participants to delve into synthetic biology, engineering, and other core sciences, iGEM gives students the opportunity to form cohesive teams with an emphasis on interpersonal communication, motivating friendship, and academic growth. The 2016 UC Santa Cruz iGEM team concentrated on the production of sweeteners using bacterial cultures. These cultures had to be sustained and monitored in a manner that yielded metabolic byproducts at relatively low cost through experimental setups replicable by those interested in DIY molecular biology or by other student groups. The team developed a chemostatic bioreactor to test the efficacy of continuous-throughput systems compared to conventional methods, and then optimized the resulting control schemes for maximum yield specific to the conditions required by the erythritol-producing bacteria.

The potential for change stemming from the engineering of these specific metabolic byproducts extends beyond the bounds of the competition. Although the event itself is organized in a way that spurs progress, the gathering at the end of the summer course is hardly the point of iGEM. By striving to develop a novel idea that impacts human practices in a meaningful way, students realize the importance of leading scientific progress oriented towards widespread social change. For instance, this particular project relies on the abundance of agricultural waste currently present in the agricultural industry to fuel the production of the end product–the alternative sweetener, erythritol. By promoting these types of experiments, the UCSC iGEM team also hopes to persuade the general public to consider that its ability to make full use of available resources is in its infancy, and to make aware the notion that many conventional systems are largely infeasible in the long run.

The starting point of the project lay in molecular biology. We decided to surpass the standard yeast-based production of erythritol–which uses the stock metabolic pathway of *Moniliella pollinis*–by using a non-sporulating *Bacillus subtilis* strain from UC Davis (our collaborating iGEM team). We used *subtilis* because of its ease of engineering, its metabolic simplicity (leading to rapid progress to accommodate the ten weeks of this course), and because of its GRAS designation. This designation was crucial in ensuring that our erythritol was food-safe, allowing it to compete with the standard production methods. The metabolic engineering team also used a phosphoketolase from

Lactobacillus plantarum because of its high rate of reaction (K_m). Without clear communication with metabolics, our other teams would have had no indication of the bacterial culture's constraints, preventing the functioning of the bioreactor altogether.

With the bacterial cultures sorted, we had to develop a method for purifying the erythritol from the expelled biomass. This was a two-step process, and was handled by our filtration team. The first step in purification was separating the biomass from the molecular products (i.e. the lysed cells and any particulate matter $> 1\mu$m). This initial filtration was accomplished by constructing a PVC coupling tube that was filled with diatomaceous earth and a glass filter. Initial testing yielded some transmittance of bacteria–namely *E. coli* used for basic characterization–which will require more thorough experimentation. The second purification step was extraction through an ion exchange resin. It was also necessary to relay the data from the bioreactor to a webserver, and in doing so, make the entire system user-friendly. The bioreactor team accomplished this by designing a two-part system that could run the feedback control loops, obtain sensor data for those controls on the bioreactor processor, and communicate with a browser-based user interface (provided by the software team) to set parameters and display the bioreactor sensor history. The intent of this development stage was to make the project accessible to a wide audience by offering an introduction to synthetic biology–thus fulfilling the educational goal of iGEM.

It was not possible, however, to fund these experiments without the operations team and our co-captains, who worked tirelessly to seek out funding and advertise our project to the general public. Because the final step of the competition was getting to the jamboree, the experiments were fairly moot without having the proper support structure. These different tasks meshed together under the structure iGEM provided, ensuring that, while each student knew a significant amount about their chosen task, he or she could interact with other students from different teams and contribute to their work, too. The aim was to prevent groups from being limited to their particular fields–for instance, to electronics alone. Instead we hoped to provide a catalyst for personal growth by enabling team members to participate in work outside of their comfort zone. The intended purpose of the 2016 UCSC iGEM team was thus threefold: 1) to challenge conventional production methods by increasing yield while lowering overall cost; 2) to provide a straightforward interface and transparent technical documentation to spur interest in bioelectronics; 3) to build a team of developing researchers with an interest in collaboration and a thirst for discovery.

2 Bioreactor Design

Abstract

My role in this process was designing and implementing a central bioreactor–which we coined Taris after the failed Star Wars ecumenopolis–in which to grow the culture, as well as programming the feedback controls and other required methods in order to relay the current state of the bioreactor and its contents to an online monitoring system. I also held a coordinating position as leader of the two-person bioreactor team, providing programming assistance to the software team as well as technical advice and workflow management.

2.1 Elementary Chemostasis

Before delving into the specifics of the bioreactor's design, it is important, first of all, to cover the processes that governs its output. Namely, the concept of chemostasis, in which fresh growth media consisting of glucose, fructose, and LB, are pumped through a sterilizable vessel (in this case, a 1-gallon glass kombucha container). The rate at which media are provided to the reaction chamber is governed by the specific growth rate μ of the culture. This is determined through the doubling time t_d of the specific bacterial strain:

$$\mu_{max} = \frac{\ln(2)}{t_d}$$

The goal of chemostasis is to achieve a steady state in which the dilution rate D, or the rate at which the bioreactor culture is being diluted with fresh media, equals the rate at which the bacteria are growing. The dilution rate can be calculated from the medium flow rate F, which is defined by the rate which liquid is flowing into and out of the reaction vessel, and the culture volume V:

$$D = \frac{F}{V}$$

In the case of our *Lactobacillus* strains, this $t_d \approx 100$ min. This means that

$$\mu_{max} = \frac{\ln(2)}{100\text{min}} \approx \boxed{0.007\ \text{min}^{-1}}$$

With a culture volume of 3.78L (1 US gallon), the medium flow rate can be determined:[1]

$$F = DV = \mu V = 3.789\text{L} \cdot 0.007\text{min}^{-1} = \boxed{0.0256\text{L/min}}$$

This calculation may vary according to the specific conditions provided with the strain, so optimization is required to determine the ideal temperature, pH, and oxygen concentration–the latter of which is currently a rough estimate due to sensor cost–in order to provide the culture with conditions for maximum growth. The result of keeping the culture in a constant state of growth is the avoidance of bacterial stagnation due to intrinsic density limiters (quorum sensing).

2.2 Peristaltic Pump Characterization

The 0.106L/min figure calculated above was achievable using the equipment we ordered. To keep the system closed-loop, peristaltic pumps were used in place of impellers. This meant that the system's tubing could be swapped when incompatible cultures were used in the same bioreactor in succession, limiting the amount of sterilization that would have otherwise been required with traditional impeller pumps. The specific peristaltic pumps used were analytical dosing pumps purchased for $12 a piece, a price point significantly lower than the standard $300 offered by most scientific vendors.

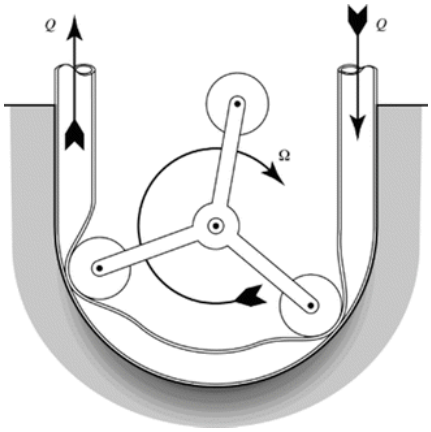

Figure 1: Peristaltic pump head, showing radial rollers attached to a main rotor that pushes fluid through a flexible silicone tube. Q refers to the pump's flow rate, and Ω its angular velocity.[2]

This setup came with the added "feature" of unoptimized pump movement, which was rectified by slightly modifying the pump head to reduce the amount of unwanted compression that might otherwise restricted the movement of the pump's compression heads. See Figure 1 for a diagram of the peristaltic mechanism.

With the pumps at hand, I breadboarded a simple nFET driver to test the output of each pump as a function of a PWM signal's duty cycle that was fed into the gate of the nFET (NXP PSMN022-30PL). The circuit was driven from an Agilent E3631A, with a maximum current of 200mA (which was initially incorrectly specified as 80mA in the datasheet) and a nominal voltage of 12V. This circuit is shown in Figure 2.

With assistance from Lon Blaumvelt, I varied the duty cycle of the PWM signal driving the motor switching and timed the fill rate for different volumes (depending on general flow rate) using a set of graduated cylinders. I also

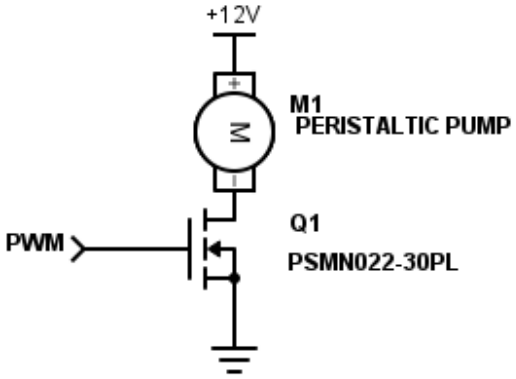

Figure 2: Circuit to aid in testing the flow rate of each peristaltic pump by using an NXP PSMN022-30PL nFET to drive the 12V, 200mA (nominal) DC motors. The Rigol function generators used did not have sufficient current supply to drive the small motors, of course.

tested several PWM frequencies, namely 50,75, and 100Hz, in order to find any correlation between motor performance and driving frequency. This data aggregation resulted in the plot shown in Figure 3.

One of the concerns with testing the pumps was the amount of random inconsistency with performance as a result of the pump heads' reliance on friction to rotate the primary spindle. This was most likely a manufacturing step to save build cost and lighten the overall BOM to a few plastic pieces, but the heads nevertheless could have used a small nut or other attaching bit– a collet, for instance–to keep the spindles freely rotating while maintaining constant angular velocity. It is wise to do this to maintain constant pump head rotation, and also to ensure that rotations are not being lost to poor contact and the resulting slippage.

I also tested the effect of the diatomaceous earth flow inhibitor that our filter team developed, comparing the uninhibited flow rate of the α pump to two types of earth. These two types were high- and low-flow, with the flow rate being dependent on the equivalent pore size of the diatomaceous earth. There seemed to be a linear flow impedance as a function of the granular size of the earth, and although the glass filter and tube most likely had some effect on the overall flow rate, the particulate filter had the largest effect. We noticed that it took some time for the filter tube to fill and fully saturate before the strained material could be extracted from the other side, and we factored this ramp-up into our tests. Because the bioreactor will have a startup sequence controlled by its accompanying hardware, it will probably be necessary to

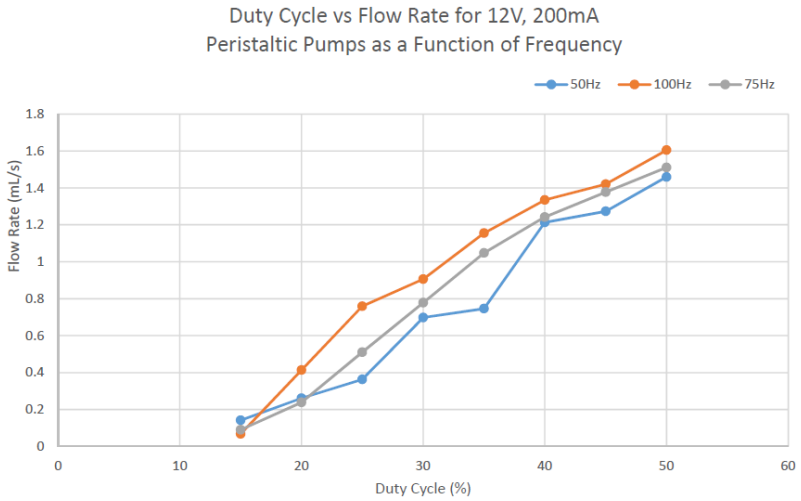

Figure 3: Results of testing the effects of duty cycle and PWM frequency for a single peristaltic pump. Performing an averaged linear fit yielded the relation $Q = 0.0431D - 0.5509$, where Q is the flow rate in mL/s and D is the duty cycle of the provided PWM signal.

have the pumps run on full for a short period of time to fill tubing, purge air bubbles, and saturate the filter. The filter-inhibited flow rate comparison for the two types of diatomaceous earth, as well as their associated linear fits, can be seen in Figure 4.

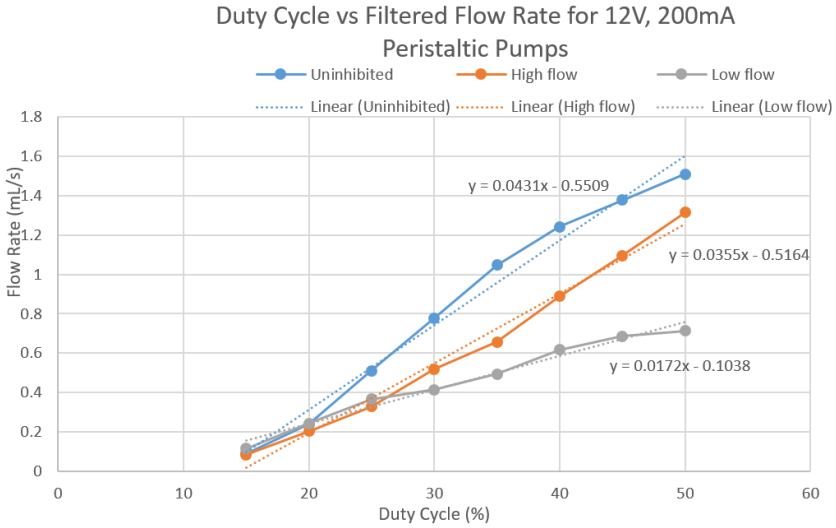

Figure 4: Peristaltic pump flow rate as a function of the type of diatomaceous earth present in the inline filter. The uninhibited flow rate is also shown as a control.

2.3 I²C Sensors & ADC

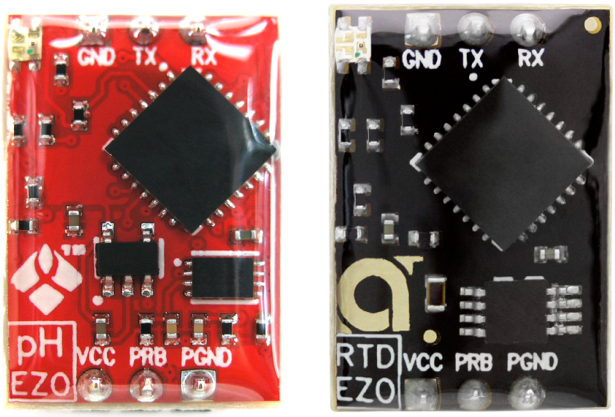

Figure 5: Atlas Scientific EZO pH and RTD sensor boards. These communicate over I²C and are each about the size of a quarter (note the 603 package SMD resistors).[3]

In order to function as a proper bioreactor, our design had to include sensors that allowed for conditional control. The two main parameters we wanted to measure were temperature and pH. These were detected using Atlas Scientific EZO boards (see Figure 5). Each of the boards deployed a similar control scheme, using the I²C communication protocol with a provided series of addressing commands to relay information over GPIO to a Raspberry Pi. I²C was chosen because of its compactness, with up to 1008 devices supported on only 2 data lines (compared to SPI's $N + 3$ for N slaves). The Atlas Scientific EZO sensors were decided upon because the pH sensor offered a competitive price point with relatively easy integration with the Rapsberry Pi; the RTD sensor followed in suit. Although temperature could have been measured using a simple voltage divider, it was stylistically beneficial to put an additional BNC probe on the PCB, and the communication library was already in place because of the pH sensor. The basic commands for the EZO boards are as follows:

CAL Performs calibration–3-step using 4.0, 7.0, and 10.0 pH buffers for pH sensor, 1-point calibration for RTD sensor.

R Reads value of probe and sends result over I²C in 7-byte package.

PLOCK Locks sensors from further system changes.

T Writes temperature value to pH sensor for additional compensation.

Although there were more instructions, these were the only ones used in the bioreactor control programs. To communicate with the sensors, I wrote a class, TarisSensor, to allow integration with the main control loop. Each sensor was set to I^2C mode by jumping the TX and PGND pins and power cycling, and locked to the mode by writing a PLOCK command to SDA.

To make sure the motors were running within their current limits, I measured the amperage flowing through them using current-sense resistors (described in depth in *Schematic Design*) attached in parallel with single-ended input pins of a Texas Instruments ADS1115 analog-to-digital converter. I chose this particular component because it offers native I^2C, and could be easily connected to the SDA/SCL lines already present for the EZO sensor boards. I also wrote a TarisADC class to take instantaneous readings of any of the motor channels using the Adafruit ADS1x15 library as a backend.[4] Figure 6 shows the ADS1115 pinout.

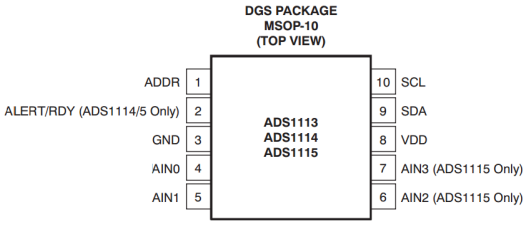

Figure 6: Texas Instruments ADS1115 pinout. Note the I^2C data lines that provide automatic conversion to the desired communication standard. The default address for the ADS1115 is `0x48` when `ADDR` is jumped to `GND`.[5]

More on the specific integration of these sensor modules and the ADC is discussed in *PCB Implementation*, and the Raspberry PI usage is detailed in *Hardware Controller*.

2.4 PID Control

The control scheme we used for stabilizing the internal conditions of the bioreactor was PID, which stands for Proportional-Integral-Derivative. This is a common feedback mechanism that is widely applicable because of its simplicity and scalability. The PID control consists of three main transfer functions, each providing a particular gain effect on the system's output. Figure 7 shows a basic PID block diagram.

The first parameter in PID control is the Proportional signal, $P(t)$, where the error $e(t)$ is the difference between the last system output value and the

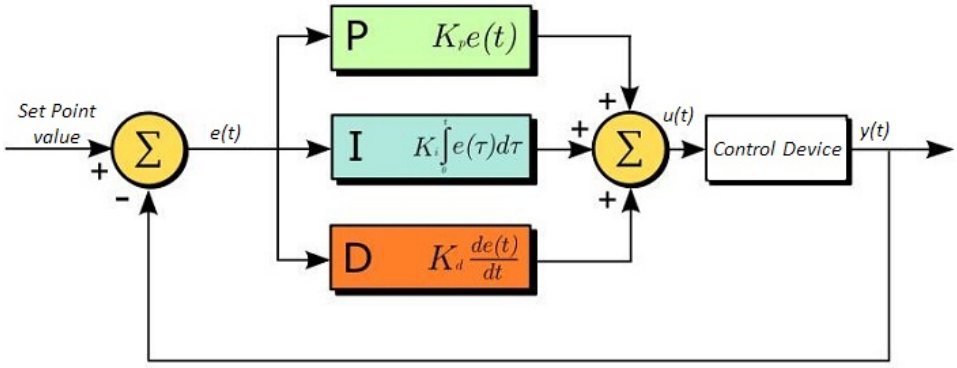

Figure 7: PID block diagram. In this case, the input to the system $x(t)$ is data from the EZO sensors, and the output $y(t)$ is the motor PWM frequency–the conversion from flow rate is done automatically.[6]

system setpoint. In other words, $e(t) = x(t) - y_{sp}$. This proportional signal is completely based on the current error signal, scaled by some gain factor K_p:

$$P(t) = K_p e(t)$$

The second parameter is the Integral signal, which takes into account the speed at which an error has occurred. This means that the controller will limit the rate at which it responds to large error in order to decrease over-response to a brief oscillation that would otherwise keep the system average fairly constant. This means that, given the system error $e(t)$,

$$I(t) = K_i \int_0^t e(\tau)d\tau$$

where K_i is another scaling factor for the integral signal. The integral signal also has the opposite behavior for an error that has not been rectified for a long time. By integrating over the history of the error within a given time frame, the system's output continuously increases if the error is not rectified. It is evident that if $e(t) = 0$ (the system has reached its desired value), the integral signal will reset to 0 and will begin integrating again from that new time origin.

The addition of the integral signal $I(t)$, however, does not come without its consequences. As briefly noted, because any error that exists for a significant multiple of the sampling time t_s will quickly throw the output of the system to a very large value–and ruin the purpose of PID. This sharp rise in integral error then necessitates the additional Derivative signal, which analyzes the rate

of change of the signal (and thus its future values) and accordingly dampens the behavior of the integral signal. This prevents the system output from oscillating due to high gain on the first two transfer functions, and allows it to narrow in on the desired final output value. As with the previous transfer functions, the derivative control also has a gain factor, K_d:

$$D(t) = K_d \frac{d}{dt} e(t)$$

Combining all of these transfer functions yields the full continuous form of the PID control:

$$y(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

Table 1 describes the effects of modifying the gain factors K_p, K_i, and K_d for a PID controller:

Parameter Increase	Rise Time	Overshoot	Settling Time	Steady-State Error
K_p	Decrease	Increase	$\pm\delta$	Decrease
K_i	Decrease	Increase	Increase	Greatly Reduce
K_d	$\pm\delta$	Decrease	Decrease	$\pm\delta$

Table 1: Effects of increasing parameter gain for each of the transfer function coefficients of a standard PID controller.[7]

Determining the gain coefficients is another matter. Aside from using expensive computational tools, most gain coefficients are found experimentally. Although manual tuning is possible, it requires a trained hand and a fair amount of time to do numerical analysis. We decided on a rough baseline tuning, although we will improve the system once we get enough data to accurately characterize the system. The basic tuning is as follows:

- Set all gain coefficients to zero.
- Increase K_p until system oscillates
- Increase K_d until system is critically damped
- Repeat last two steps until K_d does not damp oscillations, then increase K_i until system stabilizes to the desired oscillation time T_i.

We are still working on determining the correct gain coefficients for the system, although we are very close (we finally assembled the bioreactor in the

last week of class, and can now begin testing pH and temperature modulation). We have also decided to use a PI control instead of PID, since the system's response is too slow to warrant a predictive, differential component. This is because heating or cooling a gallon of water and completing on a neutralization reaction (in the same volume) can take many minutes to stabilize.

2.5 Hardware Controller

For the hardware control, we used a Raspberry Pi 3 for several reasons:

- It has a 1.2GHz ARM processor and 1GB RAM that provide sufficient computational power to run multiple control loops for the Atlas Scientific EZO sensors and the outputs of the ADC (from the motors' current-sense resistors).
- It has integrated I^2C on dedicated GPIO pins that allows for easy software access through the /dev/I2C IO path.
- It has processor-independent PWM that provides hardware-level duty cycle adjustments without having to do timing interrupts.
- It can run Unix-based operating systems and has an integrated WiFi adapter. This allows for easy communication with an external webserver without need for a middleman.

The hardware controller has several purposes, the first of which is to collect data from the pH and temperature sensors into software object instances to be used in the Taris PID loops. Although the motor currents are also recorded, they are more for monitoring purposes and not are used in the control loops (except by software failsafes to ensure that components are not being overdriven). The second purpose of the controller is to run the feedback loops and temporary offline data aggregation that allows for recording and storage even if the network link is inactive or has been randomly cut. Its third purpose is to relay that information to the webserver, which runs the main database and user interface for monitoring reactor response and editing parameters. These user-defined limits are used as the SP (set point) value in the PID control.

Besides the I^2C interface, the Raspberry Pi also uses the `PiBlaster`[8] library to write out PWM signals to the motor-driving nFETs. Although the Raspberry Pi has two dedicated, precision PWM modules, all of the GPIO pins can be driven in hardware at up to approximately 800Hz. Since we used 75Hz PWM frequencies to drive the peristaltic pumps, this was sufficient. Figure 8 shows the flow of information through the Taris system.

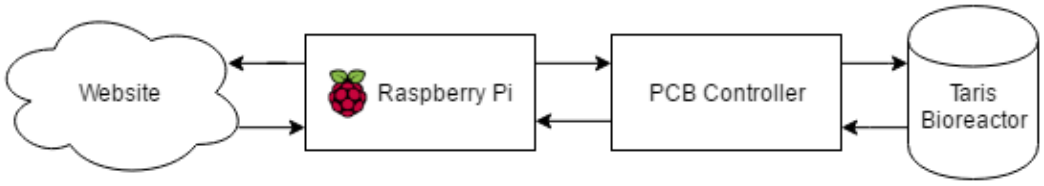

Figure 8: PID block diagram. In this case, the input to the system $x(t)$ is data from the EZO sensors, and the output $y(t)$ is the motor PWM frequency–the conversion from flow rate is done automatically.

2.6 Software

This section describes the software running on the Raspberry Pi. Most of this has been completed during the summer course, although there is still a fair amount of refactoring and documentation to finish. Our work is fairly thorough, nonetheless, and we have started packaging the Taris API in the form of two github repositories: one for the controller code and one for the server and its backend. These are freely available as downloadable binaries at

`https://github.com/UCSC-iGEM-2016/`

2.6.1 Server Communication

The Raspberry Pi communicates with the webserver–which runs a combination of Flask, D3, Bokeh, and SQL–by sending and receiving JSON packets through GET and POST commands. The HTTP communication was done using the `Requests` Python library, which allows for rapid HTTP development and integration with existing Flask architecture. In the case of our bioreactor, we opted for a packet format consisting of actions to control website UI and Pi-side feedback controls, error messages to allow easy debugging, and a payload containing sensor and motor data. We also included several other packet types with the backend API to better communicate with the Raspberry Pi. A remaining task is to refactor the codebase to include a `setup.py` script for installing all necessary packages and setting bioreactor parameters automatically.

2.6.2 Main Methods

The software is organized as a representation of the hardware layout. That is, each component on the bioreactor has its own class object that handles a specific function. Figure 2.6.2 shows an overview of the software design.

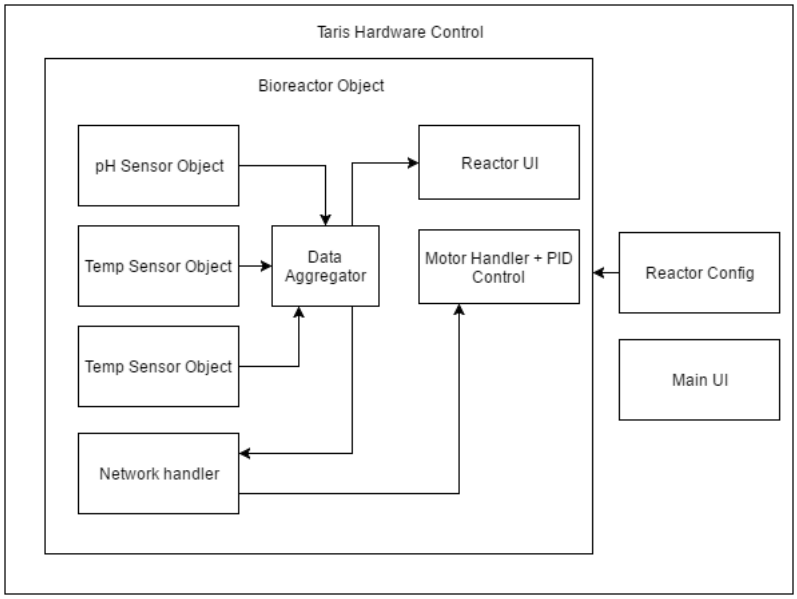

Figure 9: Overview of the software organization. This was designed to match how the control board is organized, giving a nicely-linear flow as far as data is concerned.

This organization is also beneficial in that instantiation is handled within each holding object, so that the only communication necessary between different tiers is the data passed through the system to the server. This means that the same data aggregated from the sensors used to control the PI feedback is simultaneously sent to a network handler object, which relays information in the form of JSON packets to the server. These packets were organized by the software team, and all the network handler has to do is fill in the internal dictionary to be JSONified with current system parameters:

```
self.data = {
  'comment': 'This JSON is automatically sent with the most recent
    sensor/motor information to be read into the server side
   database.',
  'header': {
    'fromPItoServer_getSensorData': True,
    'date': pi_time
    },
  'payload': {
    'pH': pH,
    'temp': temp,
    'inMotor': {
      'PWM': inPWM,
```

```
        'current': inCURRENT
    },
    'outMotor': {
        'PWM': outPWM,
        'current': outCURRENT
    },
    'naohMotor': {
        'PWM': naohPWM,
        'current': naohCURRENT
    },
    'filterMotor': {
        'PWM': filtPWM,
        'current': filtCURRENT
    },
    'heater': {
        'PWM': heaterPWM
    },
    'parameters': {
        'des_pH': des_pH,
        'des_temp': des_temp
    }
  }
}
```

As can be seen, this organizational scheme allows for a sensible data hierarchy, which in turn enables easy sending and receiving of JSON data to and from the server. For instance, getting new parameters can be done in two lines:

```
r = requests.pull(pull_path, json=self.paramdata).json()
return r['parameters']['des_pH'], r['parameters']['des_temp']
```

2.7 Schematic Design

The schematic can be broken into two main sections: the 5V logic and the 12V power. These two sections of the board had to be separated from each other as much as possible to avoid interference by the inductive elements in the motors, and to prevent shorts from the 12V, 10A power supply to the delicate sensor boards.

2.7.1 Motor Drivers

I started with the layout of the 12V power section because I wanted to get to pump characterization as early as possible. Using the PSMN022-30PL nFETs from before, I formed a scalable motor driver with noise filtering and EMI

suppression. This filtering was necessary because of the inductive spikes due to the rotation of the DC motors that powered the peristaltic pumps. With each turn of a motor, back EMF (electromotive force) $\mathcal{E}$ is generated. This EMF appears as a voltage spike with a reverse polarity to the power supply, and can occur multiple times per rotation depending on the number of poles in the motor. Because the coils of wire constituting the motor windings have low resistance (the average off-state resistance of our peristaltic pumps was $\sim 1.2\Omega$), EMF also generates large current spikes–relative to steady state operation–to the ground plane. For instance, although the operating current of the peristaltic pump motors was normally 200mA, tests using 0.1Ω sense resistors yielded momentary spikes of up to 1.6A. By putting a bulk capacitor bank in parallel with each motor, noise due to EMF could be effectively attenuated. This bank was comprised of a 470μF, 16V aluminum polymer capacitor and a 10μF, 50V ceramic capacitor. This range was used to optimize noise suppression across a range of frequencies–covering the high-frequency spikes from EMF as well as the accompanying lower-frequency "plateaus" due to the motor charging with reverse polarity.

In addition to the capacitors, Vishay Semiconductors VS-STPS20L15DPBF[9] Schottky Rectifier diodes were used for flyback suppression. In the same line as EMF suppression, the diodes were used to form a conductive loop to starve the motor of residual charge. These diodes are placed with a reverse bias relative to the power supply so that they only conduct–becoming forward-biased–when the motor charges with reverse polarity to the power supply. Energy is then dissipated by the diode as heat, instead of being transferred to the rest of the circuit. These Schottky diodes were chosen because of their low forward voltage (0.2V compared to 0.7V for standard diodes), their fast response (<500ns), and their high power tolerances–up to 1.5°C/W thermal resistance from junction to case and 125 °C T_{JMAX}, with transient spikes from the motors of 0.3W. These power ratings may be excessive, but they will allow us to scale the board to large motors without requiring significant PCB redesign. Figure 10 shows a motor driver circuit, complete with bulk capacitors, a flyback diode, and a current-sense resistor used as a failsafe.

2.7.2 Logic-Level Circuitry

The logic-level circuits were fairly simple to design. Because all of the ICs used for data acquisition–that is, the EZO sensors and the ADS1115–use the I^2C communication standard, each device has its own address. Moreover, each default address was unique, so no additional configuration other than setting the EZO boards to I^2C was required. Unlike SPI, I^2C does not have dedicated

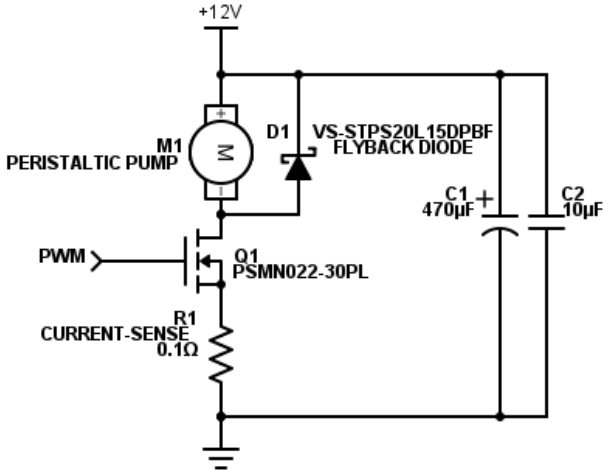

Figure 10: Motor driver for peristaltic pump control, complete with current-sense resistor and back EMF suppression.

slave lines. Rather, it uses a 2-line bus consisting of `SDA` data and `SCL` master clock lines. We could then string all of the 5V boards with two lines, rather than 4. One "feature" I did not initially realize was that the I^2C lines were open-drain, meaning that they could pull the `SDA` and `SCL` lines to `GND` but not to `VCC`. I later rectified this constraint by soldering 1.8kΩ pullup resistors to the PCB traces. In addition, parallel 0.1μF and 10μF bypass capacitors were placed between the `VCC` and `GND` pins of each IC in order to prevent noise leakage from the power supply into their signal lines. The generalized I^2C path is shown in Figure 11.

2.7.3 I^2C Considerations

The Raspberry Pi came with the additional recommendation of buffering its I^2C inputs when using multiple devices. I used a Texas-Instruments TCA9517DGKR Level-Shifting I^2C Bus Repeater because of its low cost compared to other similar buffers. Although both sides used 5V (so it did not shift levels), it allowed for the line current drawn by the sensors that the Raspberry Pi alone could not provide. Figure 12 shows the pinout for the TCA9517.

I was encountering issues with the EZO RTD and pH sensors working on breadboard, but not on the PCB. I initially encountered this when using the `i2cdetect` command on the Raspberry Pi to list all connected devices; when running the command with the breadboarded sensors, the RTD and pH sensors showed up without delay as `0x66` and `0x63`, respectively. When run connected

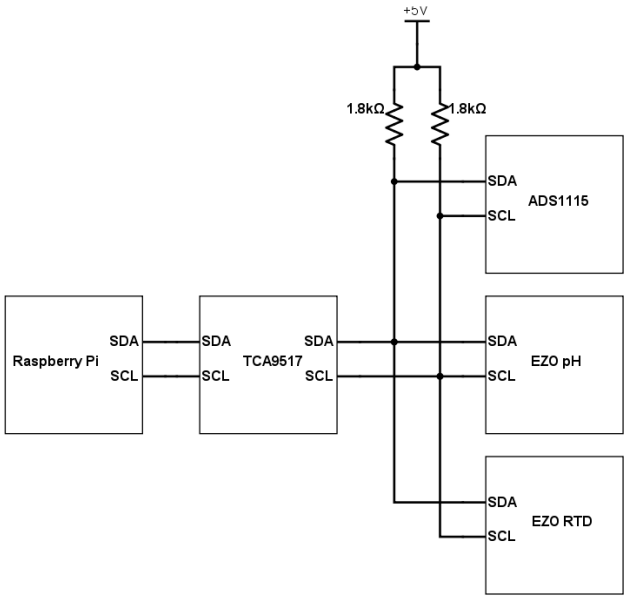

Figure 11: Bus wiring for the Taris' I^{2}C circuits. Note that this does not include power connectors for the parts' pins; the 1.8kΩ pullups are shown for emphasis.

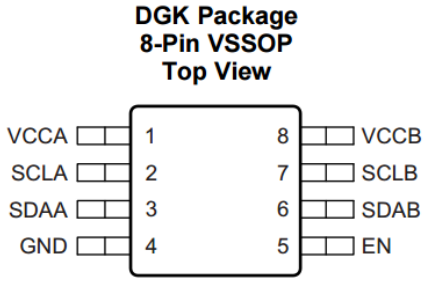

Figure 12: Texas Instruments TCA9517 pinout. In hindsight, it was probably better opting for the SOIC package, which offered a bit more pin pitch than the VSOP model. The `EN` (Enable, pin 5) required an additional 0.1μF capacitor to delay startup of the I^{2}C bus to prevent erroneous message handling. This ∼0.5ms delay–calculated using standard on-time described in the datasheet and a given 1kΩ internal pullup resistor–was factored into the Taris startup sequence.[10]

to the Taris PCB, however, `i2cdetect` took approximately 500ms per address scan to complete. I initially figured this was due to clock stretching because an issue with the buffer's transmittance of data (I^{2}C requires a response veri-

fication, in a similar manner to TCP). After several days of testing, including bouts of bypassing the buffer in different areas of the PCB, I found myself without hope. I took the board down to the Baskin Engineering basement and thoroughly examined all of the pins connected to the I^2C lines under a Leica 45x optic. I noticed a small solder bridge underneath the TCA9157's VSSOP package connected its `GND` and `SDA` lines together. This made sense– the data line was being held at 0V. I was, nonetheless, puzzled by one thing: if I had shorted the `EN` (enable) pin to ground on the buffer and bypassed it with carefully-placed jumper wires, why would the output-side solder bridge still affect the input-side lines? The answer lay in the internal design of the buffer; it was only acting as a current-providing repeater, not a full-on optoisolator. Because the lines are physically connected without isolation, an output-side bridge would still pull the entire data line to ground. Shown below is the output of `i2cdetect` for the Taris' connected sensors:

```
# sudo i2cdetect -y 1
0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:         -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- 48 -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- 63 -- -- 66 -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

One might determine through process of elimination that the device connected at the `0x48` address is the ADS1115. The job of the ADS1115 is to measure the voltage across a 3W, 0.1Ω current-sense resistor connected in series between the source of the nFET driving the peristaltic pump and `GND` to provide overcurrent protection for the pumps' motors. The current through each of drivers can easily because calculated using Ohm's Law.

2.8 PCB implementation

The PCB design was challenging considering I had not done it for such a complex circuit; although I had designed simple Arduino shields, the PCB for the Taris controller had 168 pins and many more design rules. To start, I began by dividing the board into two sections and drawing up a rough placement for all of the components. Aided by Diptrace's connectivity rules and automatic

schematic-to-PCB associations (ratlines, component pads, etc.), I came up with an iterative plan:

- Place all of the terminal blocks and header pins at the edges of the board in their respective regions
- Place components in rough areas of the board, and begin finalizing part placement
- Route out areas and divide pours to minimize trace impedance
- Lay out power traces, keeping in mind ground return paths
- Place bypass capacitors on logic-level ICs and motor headers
- Place MOSFETS and snubber diodes and route control traces to Raspberry Pi header pins
- Finalize component placement, add mounting posts, check via and traces sizes, optimize board for space

After milling the board, I encountered several problems. The first and most obvious was that the through-holes were not connected together by copper, as they normally are on poured PCBs. To rectify this, I had to bore out the through-holes with a hand drill, then feed 30 or 22 gauge wire (depending on the pin type and its current limit) through the hole, then flow solder between both sides of the board. I found that for most components this was a non-issue, but it was a concern when it came to those that sat flush against the board (like screw terminals or the aluminum-polymer capacitors).

The next issue I encountered was with solder not flowing onto the pads. After cleaning the board and coating the pads in flux, the solder flowed on evenly. Lastly, I did not realize the size–or lack thereof–of some of the surface mount components; soldering 603 SMD resistors under a microscope was quite time-consuming. In the second version, I will include spoked thermals on vias and pads connecting to the ground plane, to ensure neatness and ease of soldering (having the spokes separate a dedicated pad from the plane means less heat is lost to planar conduction). Thankfully, I did not notice any trace delamination due to excess heat dissipation on the motor driver lines. I suspect heat will not be a concern until we swap the current peristaltic pumps in favor of more powerful ones (that draw in the 1-3A range). Figure 13 shows the topside of the PCB with the accompanying silkscreen.

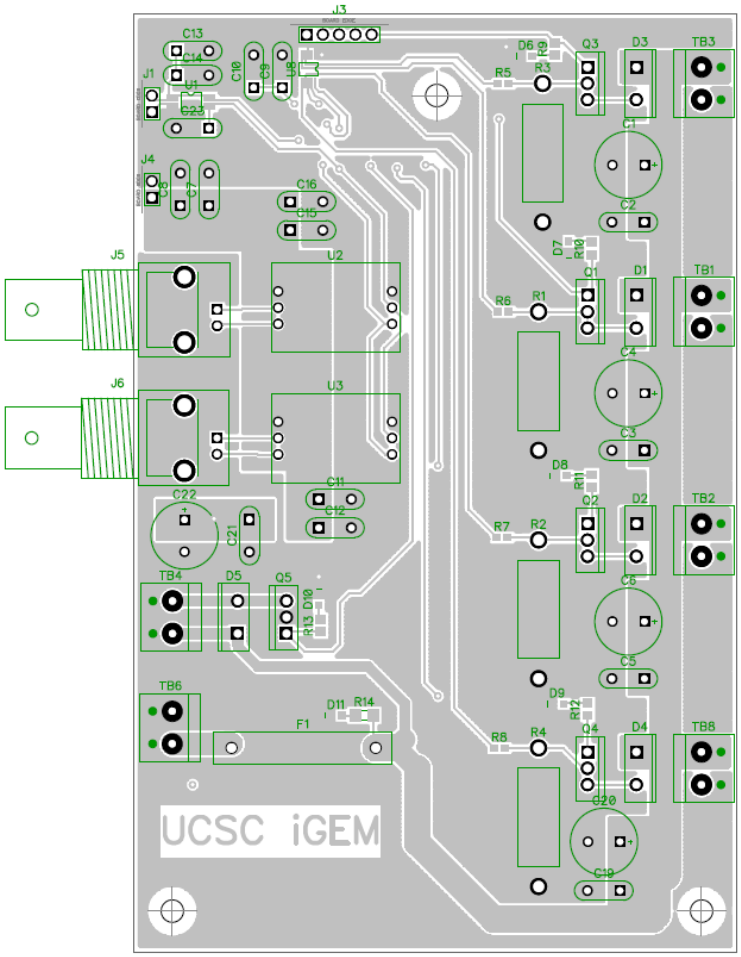

Figure 13: Taris control board PCB. Notice that the connections to the ground plane don't have any separation. This made it annoying to solder because heat would conduct to the plane instead of to the solder. Flux made it easier to ensure adhering, but the end result looked sloppy. Adding in a 2-4 spoked thermal in later designs will ensure a good connection that looks more professional. This was especially an issue with the SMD components.

Other than these minor concerns, there were not any problems with the PCB that were not easily rectified. Although milling was a good experience, I suspect that any subsequent revisions will be ordered in the form of a prefabricated PCB.

3 Results

The biomolecular team cultivated some *Bacillus subtilis* to test in the bioreactor, which we sterilized and filled with media. We set the pumps up with the PID controller, which implemented the duty cycle-to-PWM function as found in Section 2.2. Controlling the temperature, however, proved to be a bit trickier; because the 150W nichrome heating pad could draw up to 10A, we had to revert back to an on-off control through a 15A relay. This resulted in a loss of precise thermal control, as can be seen in Figure 14. The overall swing, however, was still within a nominal 0.2-0.3C. This was acceptable for our trial run, and would not pose a significant threat to the bacterial culture.

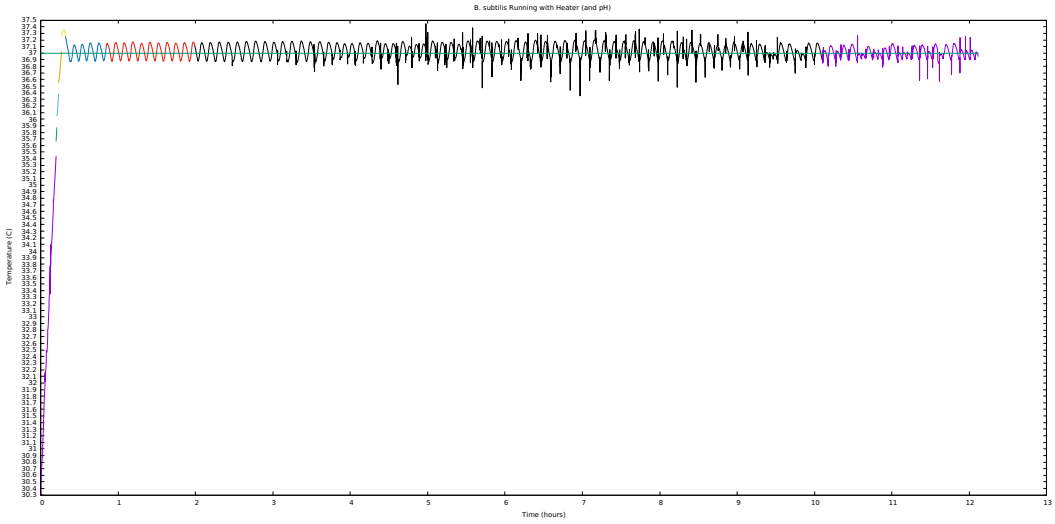

Figure 14: On-Off temperature control using a 15A relay and a control signal from the Raspberry Pi. Rise time is on the order of 30 minutes, which was as expected for such a small power input (the nominal delivered power was around 80W for the initial rise). Because of the basic control scheme, the temperature in the bioreactor exhibited an oscillatory motion. This could be fixed by implementing PID with a paired solid-state relay or a power transistor.

The pH, on the other hand, stayed constant for most of the trial run. As can be seen in Figure 15, the reactor was able to maintain a constant acidity for around 7 hours. We did, however, notice a large overshoot in the acidity level near the end of the run. We think this was a problem with sterility, as another bacterial strain may have been present from previous testing (9 hours would be plenty for noticeable culture contamination). This could be fixed by autoclaving all components before running a test, but that would also require changing out the distributive air stone and the reactor vessel itself.

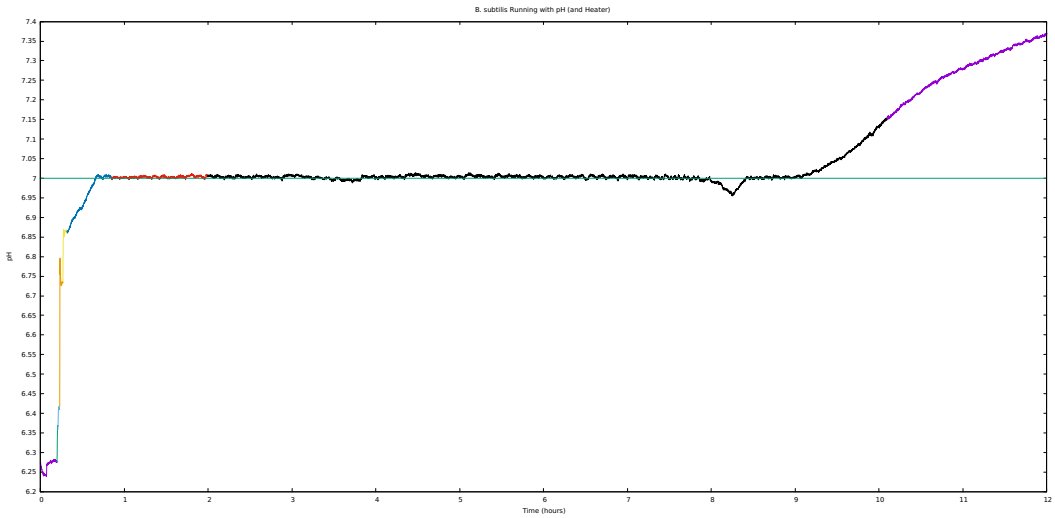

Figure 15: pH control implemented through PID. The initial acidity is due to the media; this was corrected within about 20 minutes of startup. Titrating any faster would result in poor control and a massive overshoot with any small addition of dilute NaOh (which was already in the mM range). The sharp curve at the end is most likely due to bacterial contamination from cultures present at the start of the run (which could not be corrected by the base-adding pump).

4 Reflection on the iGEM Experience

4.1 The First Five Weeks

Overall, I feel as though the first session of the iGEM course has partially fulfilled the goals stated in the background of this thesis. Because we have not completed the bioreactor yet, it is not possible to do price point analysis, nor can we determine whether the yield of the reactor exceeds conventional methods. The following weeks of microbial testing and bioreactor characterization will give us the necessary information to confirm our proposed methods. The main interface for the website has been completed, and the backend for handling data is under construction. The third and perhaps most crucial point, "to build a team of developing researchers with an interest in collaboration and a thirst for discovery," has been met with a fervor I did not forsee. Everyone on the team has been passionate about the project since its outset, and has latched on to their respective roles despite initial academic instability. By this, I do not mean that those chosen for the iGEM team were by any means underprepared. Rather, I applaud our members' propensity to learn in the

face of new exposure.

Academically, this has been a pivotal part of my college experience. I was able to build something and call it my own design, without being told what to do or, for the most part, how to do it. The research process required more than my own limited knowledge of electrical engineering–it necessitated communication with the biomolecular team to understand the culture parameters, and the software team to know how to communicate with their server. Moreover, I was put into a place of academic discomfort, and was able to come out the other side having succeeded in my intents. Although I wrote a majority of the backend for the controller, my role on the software team was more administrative. Because I had a good idea of the design path that the codebase required, I decided I would be more useful coordinating, rather than coding. This was, in a way, just as rewarding as the electronics; I felt comfortable with the other members of the team, seeing them more as friends than as coworkers (despite the team's structure), and was able to contribute my knowledge to our workflow from a hands-off standpoint. When it came down to directing the team, they gravitated to my offered help not because I told them what to do, but rather because they trusted me to lead them in the right direction. I found this incredibly gratifying, and found that it was the first time I actually felt included in the team's efforts.

4.2 Addendum: The Second Five Weeks

The second half of this course has flown by. I did not anticipate the length of time that debugging would require, although I would still argue that the schedule we adhered to has been extremely fruitful. I have had some health concerns that have impeded my progress during the second session, but I would not say that at any particular point I had fallen behind; I was still able to coordinate with the rest of my team, as well as the software team, and provide valuable feedback and design considerations even when I was not physically present. Because the board was mostly done, I was able to spend afternoons that I was in analyzing any remaining hardware constraints and staying out of the rest of the team's hair.

Unfortunately, the social environment I encountered during the second half–when I had to communicate more closely with the other member on my team–was not entirely stellar. I found a lot of resistance when trying to delegate certain tasks and attempting to keep the team on track. I did not feel as though I was able to rectify this situation on my own, and approached the co-captains for help. I was not, unfortunately, provided with any useful assistance in diplomatically handling the matter, and I am still encountering that

resistance in the last week of the project. It is getting better, though, and I am learning. It is a careful mix of subtlety and assertiveness, and finding what works for each person.

This experience has shown me a side of teamwork that I had not previously experienced. I did not anticipate that working so closely for only ten weeks would put such tension on a group of people and their relationships with one another. I do not say this regretfully, however, and I think the process of learning how to deal with these instances of miscommunication have only made me more adept at leading people and assisting them with their perspectives in mind. It has been interesting watching what makes people tick, how they function in different scenarios, and how, when they ultimately mesh with the rest of an equally unique group of people, their abilities flourish.

Reflecting on my own actions, I feel as though I have grown immensely over the course of this summer. I have developed programs to fit others' design constraints, and not just my own. I have designed a PCB from the ground up, starting only with a fundamental understanding of electronic circuits. I have set aside my personal feelings to listen to others, and brought them back when necessary. I have learned to stand up for my thoughts and to concede in the face of academic stubbornness. In this way, I have fulfilled my requirements for iGEM and what I wanted to get out of it. Although we have not yet produced erythritol, I do not think the first stipulation, "to challenge conventional production methods by increasing yield while lowering overall cost," was in any way the most important.

5 Acknowledgments

Many thanks to everyone who has helped along the way–there are too many to list. I'd especially like to give thanks to our PI, David Bernick, to our co-captains, Hannah Meyers and Pavle Jeremic, and to Kevin Karplus, my mentor and the team's secondary PI. I also want to thank Stephen Petersen, the undergraduate director of the Electrical Engineering department, and Russell Evans and Christian Monnet of Baskin Engineering Lab Support for their knowledge in PCB design and their help in implementing the Taris control board.

References

[1] K. Dane Wittrup. Course materials for 10.37 chemical and biological reaction engineering, spring 2007. MIT OpenCourseWare (http://ocw.mit.edu), Massachusetts Institute of Technology. Downloaded on 13/7/2016.

[2] Jordan M. Berg and Dongqing Dallas, Tim. Li. *Encyclopedia of Microfluidics and Nanofluidics: Peristaltic Pumps*, pages 1–12. Springer US, Boston, MA, 2013. ISBN 978-3-642-27758-0.

[3] Atlas Scientific. EZO Sensor Circuits. `http://www.atlas-scientific.com`.

[4] Adafruit, ADS1x15 ADC Python Library. GitHub Repository, 2013. `https://github.com/adafruit/Adafruit_ADS1X15`.

[5] Texas Instruments. ADS1115. *Ultra-Small, Low-Power, 16-Bit Analog-to-Digital Converter with Internal Reference*, 2009.

[6] Pid basics. Thor Labs. `https://www.thorlabs.com/tutorials.cfm?tabid=5dfca308-d07e-46c9-baa0-4defc5c40c3e`.

[7] Bruce R Land. Pid controller, N/A. Cornell University. `https://people.ece.cornell.edu/land/`.

[8] Thomas Sarlandie. PiBlaster Raspberry Pi PWM Library. GitHub Repository, 2016. `https://github.com/sarfata/pi-blaster`.

[9] Vishay Semiconductor. VS-STPS20L15DPbF. *Schottky Rectifier, 20A*, 2016.

[10] Texas Instruments. TCA9517. *Level-Shifting I2C Bus Repeater*, 2015.