Microinjector

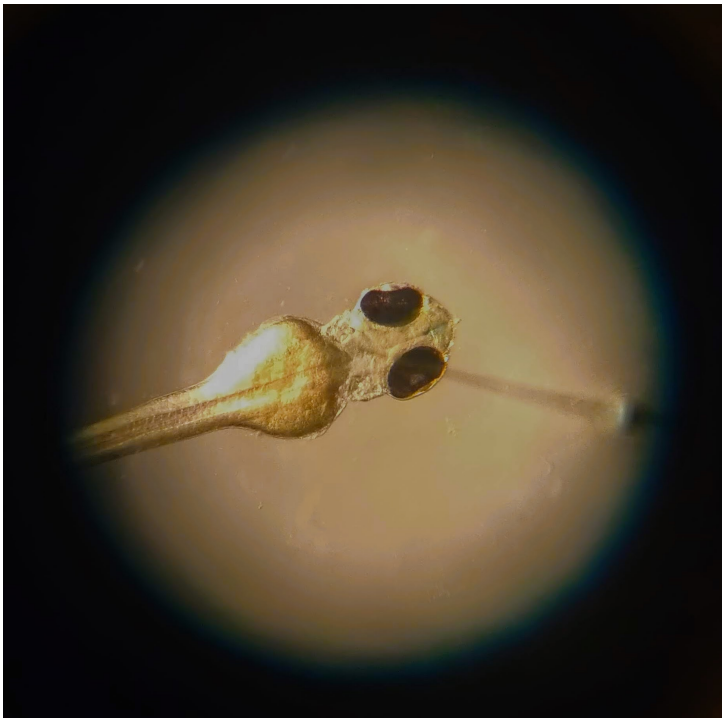
An OpenLabware Project

by

Ivo Smink

to obtain the degree of Bachelor of Science

at the Delft University of Technology



CPO Registration number:	TNDO 19-01	
Student number:	4352327	
Project duration:	Oct 1, 2018 – February 1, 2019	
Thesis committee:	Dr. E. Carroll,	TU Delft, supervisor
	Dr. J Hoogenboom,	TU Delft
	Dr. N. G. Ceffa,	TU Delft

.

TU Delft

Abstract

OpenLabware is an ongoing trend to create easy to use, robust, and open source lab equipment at a fraction of the cost of commercial alternatives. One such a project is known as the OpenSpritzer designed by Forman et al. [8]. The OpenSpritzer claims to be a direct replacement of a pressure injection system (Picospritzer) that can be used to transfect zebrafish embryos with DNA encoded fluorescent proteins. By making use of an Arduino, a fast-acting solenoid, and a 3D printer an adaptation of this system was designed, modified, and built to provide a simple and consistent method of producing nanoliter droplets needed for embryonic injection.

The new system proved reliable and showed a linear relationship between pressure, pulse duration, and bubble diameter. No precise volume calibration was achieved due to inconsistency in pipette tip size as well as other experimental errors. However, a graphic user interface (GUI) was produced to give greater control over the system. With the GUI a serial communication can be established between Python and the Arduino. This allows the user to precisely control pulse duration and program a sequence of pulses. When compared to the commercial alternative, the Picospritzer, this system is equal in performance and delivers greater control at a fraction of the cost.

Contents

1 Introduction

Open-Labware [3] is an ongoing trend to create easy to use, robust, and open source lab equipment at a fraction of the cost of commercial alternatives. Through the advent of open source technologies such as 3D printers and standardized microprocessors (e.g. Arduino), substantial reduction of time and cost of small scale engineering have been achieved; leading to a substantial increase of custom built devices. This research paper focuses primarily on the technology needed to reliably and easily inject microscopic volumes of fluid. The Carroll lab at the TU Delft require such a system for imaging research using zebrafish. Two of the lab's primary applications for nanoinjection are: injecting fertilized eggs with DNA/RNA for genetic manipulations and injecting pharmaceutical solutions (e.g., paralytics) intravenously in young zebrafish for imaging experiments.

Picoliter volumes were previously generated with the use of a pressure injection system [9]. Due to the limited consumer market, the Picospritzer has a substantially inflated cost. Therefore a open source alternative is desired; one such device is known as the OpenSpritzer[8]. Similarly to a Picospritzer, the OpenSpritzer claims to be able to reliably eject picoliter volumes. This would make the OpenSpritzer a valuable tool in many fields. A comparison of the two can be seen in table 1.1.

Specifications	**Picospritzer**	**OpenSpritzer**
Pulse Duration	2-999 ms	2ms - 2^{16}s
Pressure range	0.67 - 6.7 bar	0.6 - 8 bar
Reproducibility (S.D.)	0.048	0.059

Table 1.1: Both the Picospritzer and OpenSpitzer have very similar specifications. Pulse duration of the Openspritzer is only limited by the microcontroler. The reproducibility is given as a standard deviation form the norm value. All data was taken from Forman et al. [8] and Hannifin [9].

The OpenSpritzer technology, which this research is based upon, also uses a pressure injection system. Due to the open source nature of the project, additional features will be researched and added to increase ease of use.

This leads to the following research questions:

- Can a similar system to the OpenSpritzer be built and used to inject zebrafish embryos reliably with a picoliter volume of DNA/RNA?

- Could features be added to improve ease of use and accuracy?

The OpenSpritzer system contains a microcontroller and a microfluidic system. Due to the small size of young zebrafish microfluidics are often used to study motor and sensory responses of the fish[6]. Microfluidic systems are able to pump microliter to picoliter volumes efficiently and with great precision. Such a system could theoretically be used to stimulate the fish and observe the response. This could be achieved by using a slight adaptation of a similar system to the one used by the OpenSpritzer; which leads the the final research question:

- Could the Openspritzer design be adapted to give greater control and thus be of practical use for microfluidics?

In chapter 2 some background information is given, followed by the method in chapter 3. The results and conclusion are handled in chapters 4 and 6 respectively. There is also a short discussion with some additional recommendations for follow up research in chapter 5. An Appendix with all codes is also given.

2

Background Information

Zebrafish are a small variety of teleost fish that have been used in biomedical research since the 1970s. Due to their small size and similarities to other vertebrates, both physically and genetically, such as ourselves, zebrafish have been widely used in modern research [12]. Given their size, rapid development, and translucent skin, zebrafish are well-suited for optical imaging of a developing nervous system. Using a process called light sheet-microscopy the entire brain of a young zebrafish can be imaged every 1.3 seconds [1]. Whole brain imaging allows for analysis of neural cell interaction and other brain processes; thereby creating a better understanding of the neural activity within such a fish and thus also within ourselves.

Fluorescence imaging was revolutionized by the discovery, and subsequent bioengineering, of genetically encoded fluorescent proteins. By introducing modified DNA any cell can be made to produce a fluorescent protein (FP), thereby allowing the cell to produce a self-renewing label that can be easily visualized by light microscopy. However, wild zebrafish are not born with the genetic trait that allows them to produce the fluorescent indicator; thus the DNA needs to be injected manually. To ensure that all cells carry the desired genetics, the zebrafish embryo needs to be injected with the proper DNA at a very young age. This should preferably be done when the embryo is still in the single cell stage, thereby ensuring that all daughter cells have the desired genetic traits[4].

2.1. Picoinjector

Transfection of zebrafish embryos is most commonly done with a pressure microinjection system. [9]. Such a system works by ejecting a small pulse of compressed air that in turn forces a small amount of liquid out of a glass pipette and into the fish embryo. A picture of this can be seen in the figure 2.2.

Fish embryos undergo cell division every 20 to 30 minutes. Therefore, it is necessary to have a quick and reliable way to inject the cell with the desired genetic material. A well trained technician is able to inject a fish embryo every second. Emphasizing the need for a robust and easy to use system. However, due to the high cost of a commercial Picoinjector, alternatives have been desired.

Nonetheless, the Picoinjector is still the most reliable way of transfecting the embryos

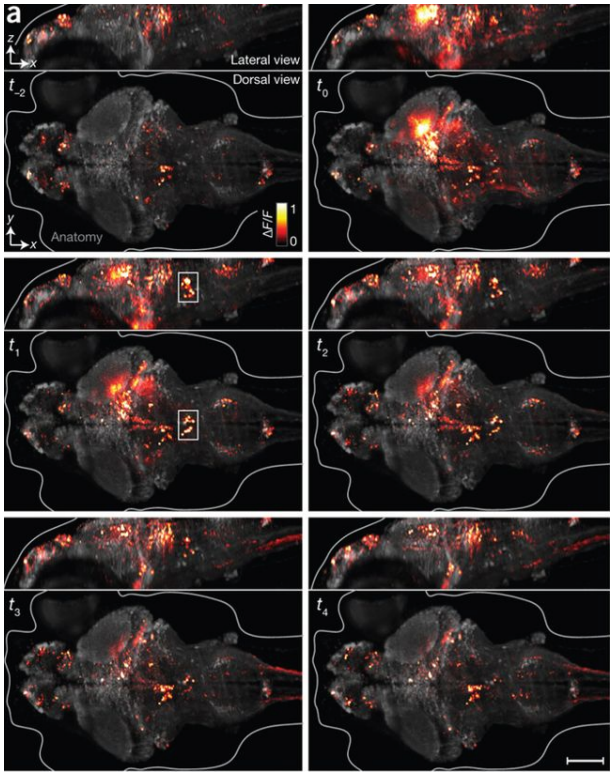


Figure 2.1: Light-sheet microscopy makes use of genetically encoded calcium indicator GCaMP5G. These indicators emit florescent light when the neural cell is activated, thereby allowing for imaging of neural cell groups when they are active [2].

with DNA. This is why OpenLabware provided a useful tool with their Openspritzer; a pressure injection system at a fraction of the cost.

The Openspritzer makes use of a fast acting solenoid in junction with an Arduino. The Arduino controls the length of time the solenoid is open and thereby provides a pulse of compressed air very similar to that of the Picoinjector. The Picoinjector is able to achieve droplet volumes in the picoliter range, nevertheless for injection into the fish embryos only several hundred picoliter is needed.

Measurement of the picoliter droplet can be achieved via a visual or digital measurement with a microscale tool. The volume of a spherical droplet,V, can be accurately calculated via the ball volume formula, seen in equation 2.1, with R the radius of the droplet and D the diameter:

$$V = \frac{4}{3}\pi R^3 = \frac{1}{6}\pi D^3 \tag{2.1}$$

2.2. Microfluidic Circuit

There is a distinct parallel between microfluidics and electronic circuits[5]. In a circuit the power supply applies current ”pressure” to the system which in turn flows through and is affected by all connected resistors, capacitors, and wiring. This is similar to how a pressure difference in a microfluidic system forces a flow through

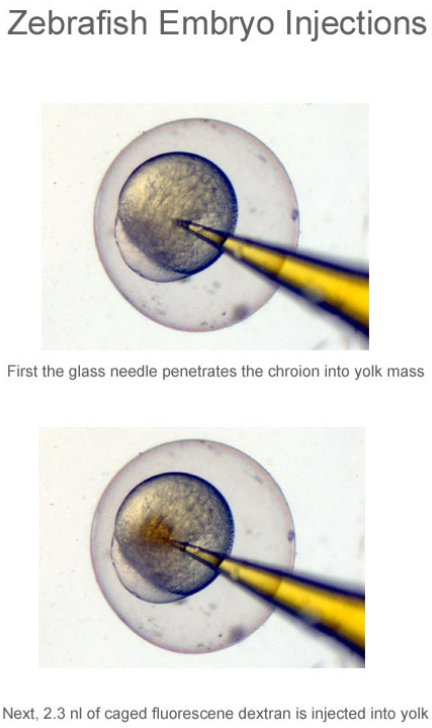


Figure 2.2: Fish embryos being injected with a pressure ejection system [4],

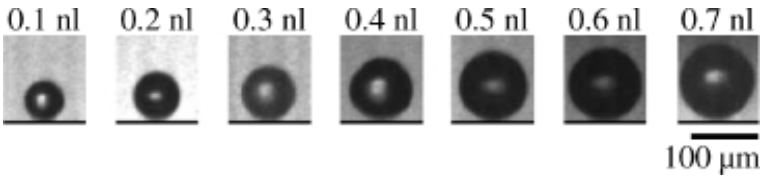


Figure 2.3: Droplets measured with the use of a digital strobe scope and calibrated microruler. [10]

tubing. The rigid channels of the microfluidic system act as a resistor, and the soft tubing as a capacitor due to the expansion when pressure is added.

The Openspritzer design includes a simple version of a microfluidic system. The pressure value can be adjusted as well as the length of time the system is open. The different tubing in addition to the transition points from one tubing to the next both impact the final pressure on the liquid. The soft tubing can expand given a pressure increase, thereby acting as a capacitance (i.e. stores pressure) as can be seen in figure 2.4.

In addition to tubing properties, liquid properties also play a role in the fluid flow through the system. Reynolds number is a way of expressing the relationship between inertial forces and viscous forces, thereby giving us flow properties (i.e. turbulent or laminar flow). Equation 2.2 gives us the Reynolds number with the velocity (U) given in terms of the the area-average velocity (Q the flow rate and R the radius) equation 2.3. With values $D = 10$ µm (diameter), $\rho = 10^3$ kg m^{-3}(density), $\eta = 10^{-3}$ Pa s (viscosity), and $Q = \frac{1nL}{50ms}$. This leads to a laminar flow with a Reynolds number of 1.82.

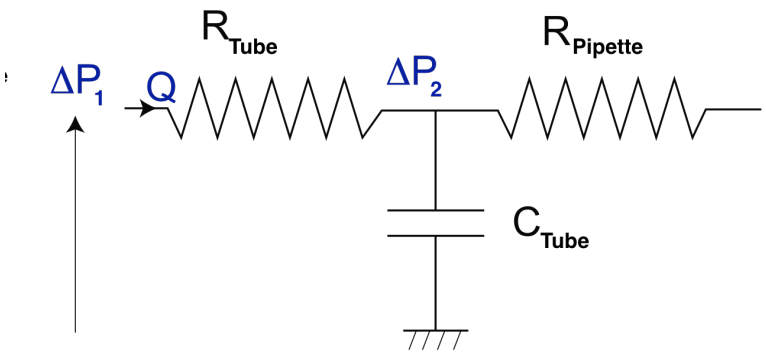


Figure 2.4: A circuit representation of the microinjector. With R_{tube} and $R_{Pipette}$ the hydraulic resistance, C_{tube} hydraulic capacitance from expansion of tubing, and p_1 the input pressure.

$$Re = \frac{\rho U D}{\eta} \tag{2.2}$$

$$U = \frac{Q}{\pi R^2} \tag{2.3}$$

Given a laminar flow in a smooth channel, hydraulic resistance (R_{Hydro}) can be calculated using equation 2.4 with $C_{Geometry} = 8\pi$ [11]. Due to the conical shape of the microinjector pipette tip the exact resistance can not easily be calculated; this does play a very large role and acts as a hydraulic resistor. For a microfluid system with constant diameter channels a more accurate estimate of the hydraulic resistance can be given.

$$R_{Hydro} = C_{Geometry} \eta \frac{L}{A^2} \tag{2.4}$$

Although it is known what factors play a role in the setup, hydraulic resistance and hydraulic capacitance, the exact values are not easily determined. Therefore, the system acts as a black box. A given pressure is applied to the input and an output is observed. To determine the correlation between these two factors, experimentation is needed. It is expected that a linear correlation between volume out (V_{out}), puff duration ($T_{duration}$), and an unknown constant (A) dependent on the pressure (P) and pipette tip diameter (D) as seen in equation 2.5[9].

$$V_{out} = A(D, P) T_{duration} \tag{2.5}$$

Given the core of the system, the Arduino controlled solenoid, a more complex ”circuit” can be created. As mentioned in the introduction, one such circuit could be used to create stimulus response from the fish[6]. However, with a different design a new calculation would be needed to redetermine the relationship between input and output.

3

Method

In the following chapter an overview of the parts of the microinjector will be given. Additionally, the testing method and calibration will also be touched upon.

3.1. OpenSpritzer

The microinjector was based heavily on the OpenSpritzer mentioned in chapter 1. However, several aspects of the microinjector were modified to better fit the use-case. To achieve this, a thorough understanding of the OpenSpritzer design and code was needed. In brief, the whole system is centered around a fast switching solenoid, which is controlled via a 5V transistor–transistor logic (TTL) pulse. The source of the TTL pulse is an encoded Arduino (micro-controller). In the following subsections a further explanation to the design and the features added will be given.

3.1.1. Solenoid, Power Supply, and Potentiometer

In this system a fast switching solenoid is used. The manufacturer claims a pulse of 3.7 milliseconds could be achieved [7]. The accuracy of this claim will be tested by comparing expected volume to measured volume, thereby giving a clear lower limit to the puff duration. However, for the application of transfecting the fish eggs with DNA, puffs of this duration should not be necessary.

The solenoid is controlled via a single digital port of the Arduino(digital port 5). To open the solenoid a TTL pulse of a given length is sent from the digital port to the base input of a NPN transistor. The base current allows for current to flow through the transistor and thereby opening the solenoid. If there is no base current, the transistor is not conductive and thereby blocking current flowing through the solenoid.

The solenoid requires a current of 114 mA. This is significantly more than the Arduino can supply (max 40 mA). Therefore a power supply is connected to deliver the required current. Two diodes are also located in the circuit, one directly before the Arduino output and the other in parallel to the solenoid, to prevent back EMF (electromotive force) from damaging the Arduino.

The solenoid has two inputs and a single output. One input is connected to compressed air and the other is left open. The output is connected directly to the syringe.

When a pulse is given, the input switches to the compressed air input and thereby applying pressure to the syringe fluid. This gives a direct correlation between puff length and the amount of fluid excreted through the syringe.

The system is fitted with a single potentiometer. When the value of the potentiometer changes the Arduino calculates a new value for the pulse duration. This is done by comparing the measured value to the maximum possible value. Within the code a given number of possible steps between a zero measurement and the maximum measurement is given. This value can be adjusted to allow for a large range of pulse duration values. This allows for a quick adjustment of the pulse duration and thereby the volume ejected from the syringe.

3.1.2. Arduino

The solenoid valve is controlled via a 5V TTL pulse. In the original design the pulse was generated either via a TTL pulse generator or by means of an Arduino. The source of the TTL pulse was to be determined by a switch, however, it was later determined to be unnecessary due to always using the Arduino to control the solenoid. Therefore in the final version the switch was modified to choose between sequence mode (multiple puffs of predetermined length) or single puff mode.

An Arduino consists of 13 digital pins and 5 analog pins. The digital pins can be set to either read or write mode. This allows for the measurement or sending of digital information ranging from 0 to 5 volts. This allows the Arduino to control the solenoid as well as a few other key features: the LCD display, monitoring switch value, and monitoring button clicks. A single analog pin is also used to monitor the value of a variable resistor. This value is then used to determine puff length. Lastly, the Arduino is also fitted with a USB connection; allowing for quick reprogramming as well as communication with an external computer.

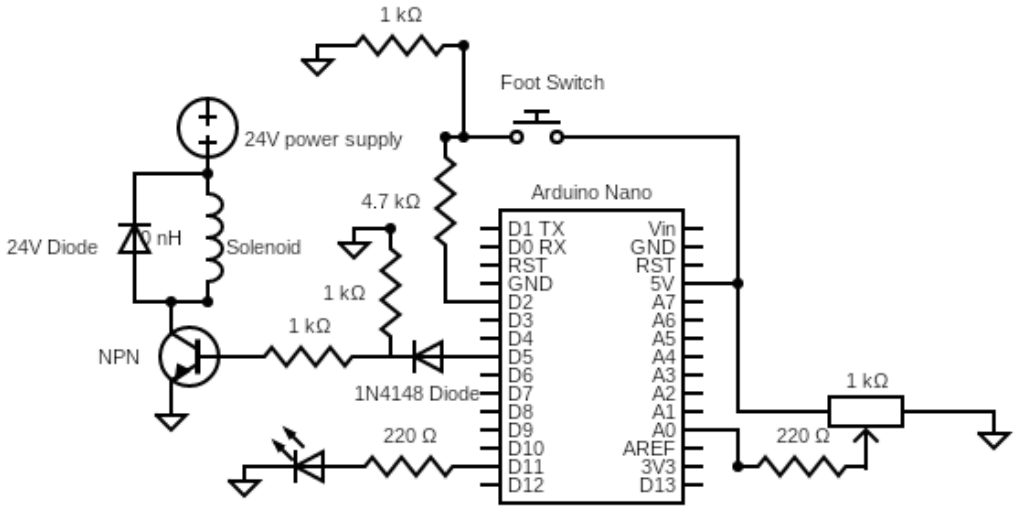


Figure 3.1: Diagram of how circuit is connected to the Arduino.

3.2. Test Method

A test method similar to the calibration method used in Laboratory Animal Course by Bakker [4] will be used. For this, a set of droplets will be created at constant pressure with varying time. This will give a correlation between puff duration and droplet volume; calculated with equation (2.1). This measurement will be repeated using constant puff duration and varying input pressure as well.

Droplet size is also dependent on the gauge of the pipette. To ensure a constant variable, the pipette will be pulled under the same conditions. Once pulled pipettes need to be broken at the time so that they are open. This can be done using tweezers, however, this is not very accurate and leads to tips of varying size. In addition, the precise diameter of the pipette will be measured using a digital microscope. The same microscope will also be used to measure the diameter of the bubbles.

To ensure the bubbles do not evaporate or merge during transport to the microscope, the bubbles will be injected into oil. This also suspends the bubbles and thus allows them to retain a spherical form; giving a more accurate volume measurement.

The bubbles themselves are comprised of a low mixture of agarose, water and ink to ensure a sharp contrast with the oil. The mixture will be tested to determine a combination that retains its shape well after being ejected by the microinjector.

4

Results

4.1. Building Microinjector

A brief explanation of features added and changed will be given in the following subsections.

4.1.1. Switches, LCD, and LED

A footswitch is constantly monitored by the Arduino to determine when the puff is to be delivered. Originally the number of clicks would determine whether a sequence of pulses were to be deliver or just a single one. Throughout the experiments this proved to be quite inconvenient as it would lead to improper puff lengths due to occasional miss clicks. Therefore, the toggle switch was adjusted to allow for a simpler method of choosing the pulse type. The switch value, HIGH or LOW, was monitored by a digital input from the Arduino. The HIGH state is given when the switch is connected to the 5V connection from the Arduino, and LOW when the switch is in the ground state.

In the original design a LED was used to indicate a few things: when the potentiometer was being measured, the length of the pulse duration (given by a number of blinks), and was activated simultaneously with the solenoid. Counting the number of LED blinks proved to be a inconvenient way of conveying the pulse duration. It would require the user to know what the hard-coded pulse duration steps are. This could easily be improved by connecting the Arduino to an LCD screen. This also proved to be useful to display additional information as well as the pulse duration.

The LCD screen required the use of several digital outputs as well as connection to the 5V output and ground. Additionally, the Arduino code needed to be adjusted.

4.1.2. Arduino Code

The full code can be seen in the Appendix. The Arduino is written using C++. A majority of the program was taken from the Openspritzer code [8]. Additionally, code was added to send information to the LCD. The information that is displayed is the following:

- Puff duration
- Moment the puff is executed
- If sequence is begun which sequence is currently being executed
- When sequence is broken or ends
- Error code when too many clicks are registered

The error code for too many clicks is executed when the foot switch depressed more than once within 500 milliseconds. This was done to overcome the execution of multiple puffs within a second due to accidental clicks. Furthermore, code was added to communicate with an external Python script. This portion of the code has no impact on the workings of the Picospritzer and thus only adds additional features if wanted. This portion of the code will be explained under the section Microfluidics.

4.1.3. 3D Printing

An Ultimaker 2+ was used to make several things: a box, a lid, and a knob. An initial design for the box and lid was given by Forman et al. [8]. However, due to changes such as the LCD and use of an external pressure gauge, the box was made larger and a few wholes were changed. A low fill percentage was used to speed up the printing of the box and lid, however, this may lead to slightly weaker walls and pillars. Lastly, a knob was also printed for the potentiometer to further reduce the cost of the entire unit. All the printing codes used can be found in appendix B.

4.2. Microfluidics

The microfluidics system is an additional use case for the same Arduino solenoid design. In this case the solenoid would be used to stimulate a fish by pulsing a small amount of water against the lateral side of the fish. This would affect the fish and allow for the observation of a stimulus response in the fish brain.

The system as described in the previous sections needs no additional parts to be used for microfluidics. However, the code controlling the Arduino needs to be adapted slightly. Additionally, a Python script was created to give greater control over the Arduino. In the following subsections the Arduino code as well as the Python GUI code will be touched upon. In appendix B the entire code can be found.

The GUI was designed to help with the stimulation of fish. To integrate this properly into the experimental setup a much greater control of the system was needed as well as communication between the Arduino and the computer software controlling the microscope. This could be done by communication over the serial port of the Arduino in addition to a Python script.

4.2.1. Python GUI

From Python 2.7 and 3.1 Tkinter was included as a built in graphic user interface (GUI). This allows the user to create a so called “front panel” with programmable buttons, labels, and text boxes. The interface can be used to display variable values and/or set in action a desired script.

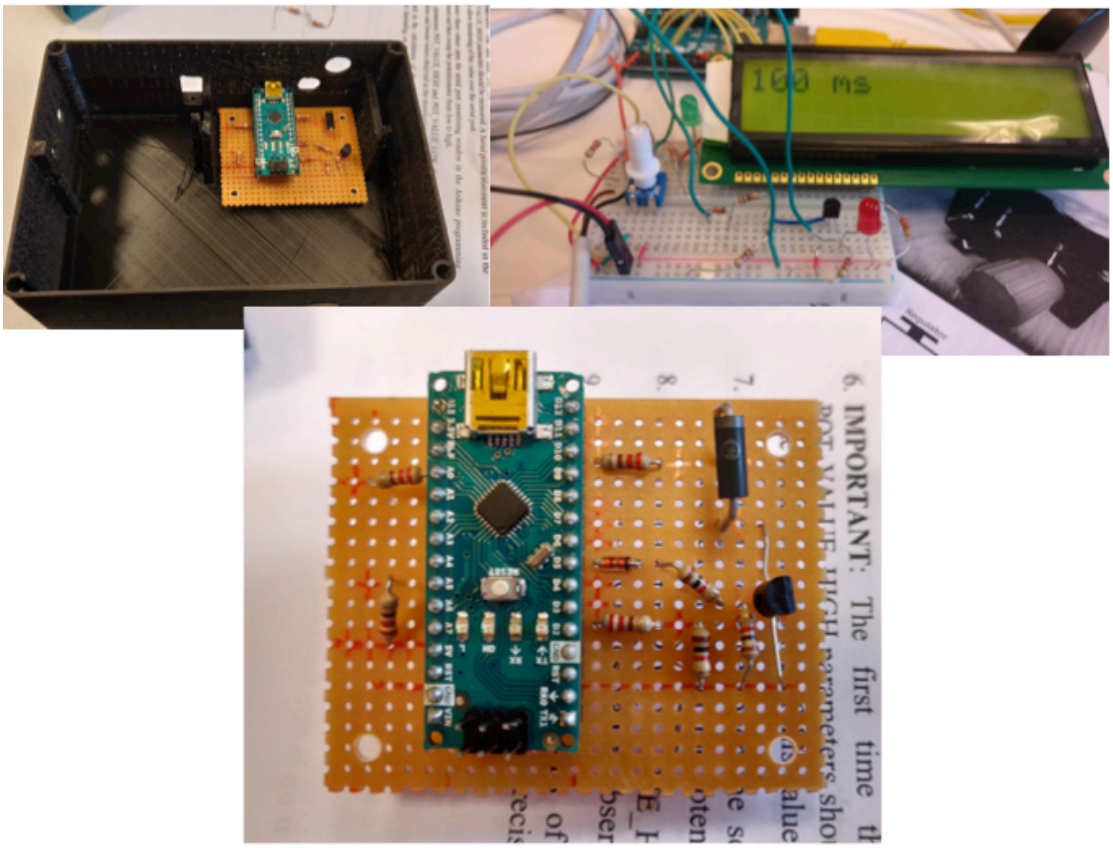


Figure 4.1: Arduino circuit test fitted in 3D printed box, Circuit and LCD being tested, all components soldered onto final strip board.

The main goal was to give greater control over the Arduino and thereby the microfluid system. Without the Python script the Arduino can only be controlled via the variable resistor located on the side of the box. This allows for a range of puff lengths to be selected but gives no fine control over the system.

The python GUI allows for better control of a few key features:

- Serial port selection
- Puff duration
- Sequence duration

This can be seen in the figure 4.2. As well as giving control of the above features, the GUI also displays information such as total puff duration, current puff duration, and total number of puffs.

The port selection box was necessary to ensure communication with the correct port. This also allows for the controlling of multiple Arduino systems. With a simple drop down menu all available ports are displayed and can be selected. The port name is also displayed in the command window when information is sent.

Below the port selection box are two buttons labeled Single Puff and Sequence Mode. Dependent on the button that is selected the Arduino changes a given value(s). For example, if single puff mode is selected then the Puff Duration will be changed. This

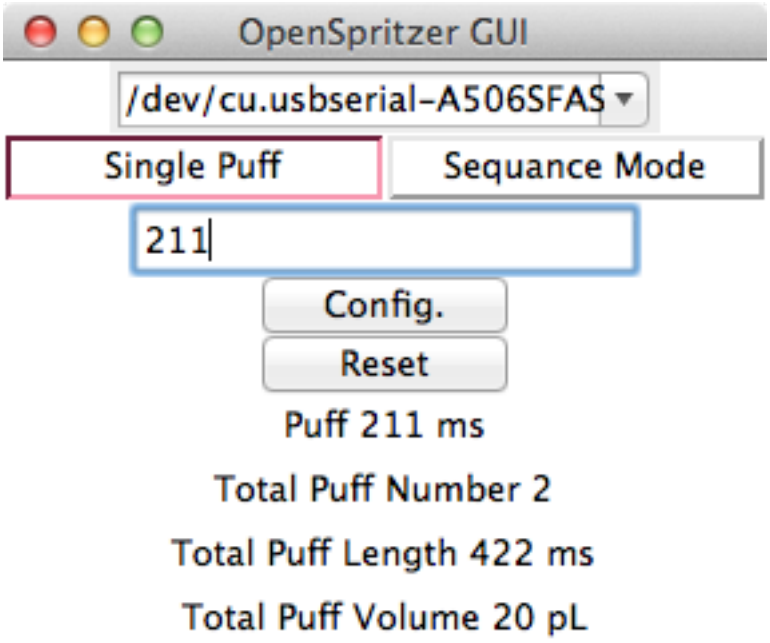


Figure 4.2: GUI set to single puff mode and two puffs executed. Volume shown is not accurate. Also displays total number of puffs and total puff length. Values can be reset with reset button.

has no impact on the reprogrammed sequence list. Once sequence mode is activated the Arduino fills the sequence list with the given values. All values not given will be filled with zero. There is a maximum of 28 possible entries in sequence mode. How this communication with the Arduino is exactly done will be explained in the following subsection.

The entry box allows for the input of integers as well as a list of integers if the GUI is in sequence mode. Below that are two buttons Confirm and Reset. Confirm sends the entered value(s) through the selected serial port. The reset button allows for the reset of the listed values: Total Puff Length, Total Puff Number, and Total Puff Volume.

4.2.2. Serial Communication

Serial communication between Python and the Arduino is needed to properly utilize this system. The Arduino needs to communicate with the computer at which moment the stimulus is given to the fish as well as receive information from the computer. To achieve this a serial communication between the Arduino and python needs to be established.

As mentioned in the previous section the following things need to be communicated between the Arduino and the computer: Puff Duration, Sequence mode activation, and Puff volume. To communicate these values a protocol needed to be established to ensure that the values are attributed to the correct variables.

The Arduino has a single serial port which allows for the sending and receiving of data. However, the Arduino's processor can only receive a single byte at a time. Therefore it was required to send an array of digits of the same length each time. The first integer of this array is used to indicate what needs to be done with the following

digits in the array. An example can be seen in the table 4.1.

Sent from Python	Sequence or Single puff	Puff length (ms)
[1,25]	Single puff	25
[1,234]	Single puff	234
[2,12,43,534]	Sequence	12, 43, 534

Table 4.1: Arrays sent from python to Arduino via serial port. First value (either 1 or 2) determines what Arduino does with additional number in the array.

The Python GUI changes the first digit depending on which button is selected. This same process is also used to send information back to the computer. An array of length 3 is used to send Puff duration as well as Puff volume. The fist integer sent is a place holder that can also be used to activate the camera.

The camera used to image the fish brain needs an indicator to start recording information once a stimulus is given. This is why the Arduino sends information over the serial port every time it is activated. The camera could thus be programmed to start recording every time information is received over the serial port for the given length that is also sent as the second value in the array. A new array is built every time 3 integers are received by the computer.

4.3. Calibration

4.3.1. New Test Method

Due to several unanticipated factors the calibration of the microinjection system went differently than explained in the previous chapter. Mainly, it proved very difficult to suspend the bubbles in oil consistently and without large movement during transport to the microscope. Additionally, due to the use of an agarose solution to form the bubbles; the shape was asymmetrical and inconsistent. Therefor a new test method was required.

As stated in the previous chapter, the microinjector should be able to reliably and consistently create bubbles of similar size. To test this, a spherical bubble will be created and measured at a microscope in another room. Therefore, the bubbles need to be stable on or in their test medium. This was achieved by placing the bubbles on a dry Petri dish. If the dish is wet or dusty the bubbles move and become asymmetrical; thus cannot be measured accurately.

The bubbles are very stable once ejected onto the Petri dish, and allow for numerous bubbles to be placed very close together and measured at once. With this method, consistency could be measured by measuring the diameter of the ejected bubbles, however, volume measurements could not be achieved with this method. This is in large part due to the unknown and changing contact angle between the bubbles and Petri dish. Yet a correlation between pressure, pulse duration, and pipette gauge could be measured.

To give a volume estimate per bubble a series of bubbles will be ejected and weighed. This will lead to a weight per bubble and could then be converted to a volume per bubble given equation 4.1 with M mass and ρ density of water.

$$V = \frac{M}{\rho} \tag{4.1}$$

4.3.2. Experimental Data

To demonstrate the linear correlation between pressure and duration bubbles of varying pulse length (50, 100, 150, 200, 250 milliseconds) were produced at varying pressures (0.6, 0.8, 1.0, 1.2, 1.4 bar). This created a three-dimensional surface that should show a linear correlation along the x and y-axis at every pressure and time duration. This can be seen in figure 4.3.

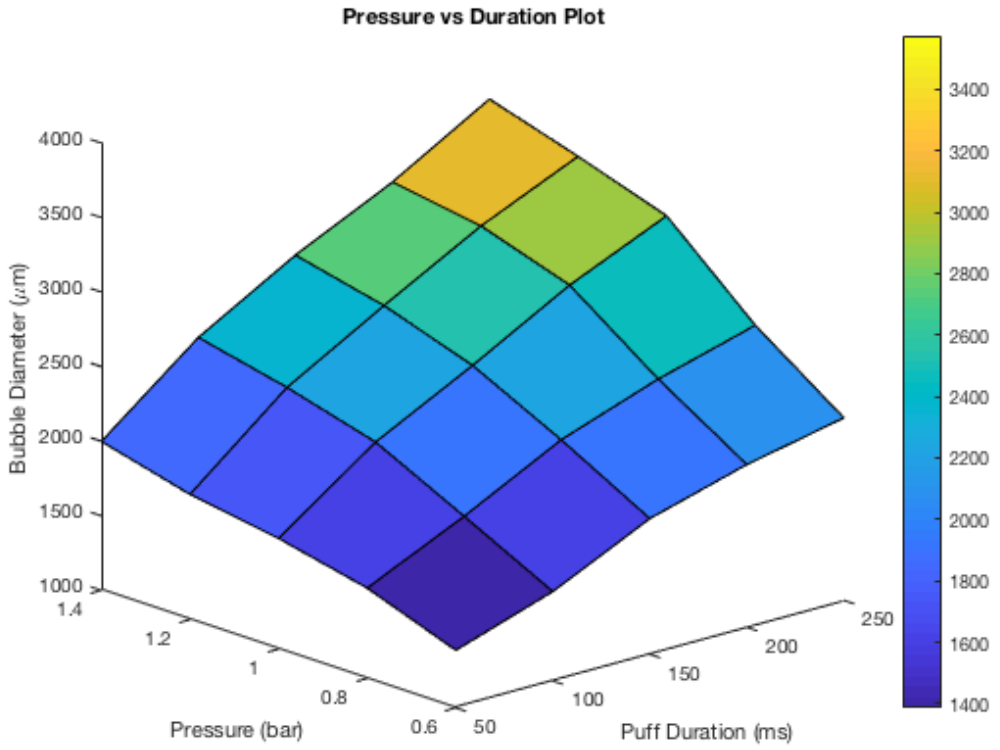


Figure 4.3: Linear relationship between puff duration and pressure projected onto 3D surface.

Pressure \ Duration	50 ms	100 ms	150 ms	200 ms	250 ms
0.6 bar	1386.29	1600.37	1909.72	2092.11	2225.10
0.8 bar	1606.49	1907.57	2240.84	2468.41	2649.23
1.0 bar	1742.47	2208.60	2542.55	2901.00	3186.02
1.2 bar	1840.22	2377.52	2747.02	3100.95	3387.62
1.4 bar	1998.51	2516.03	2888.05	3198.12	3574.67

Table 4.2: Surface data, with 0.6 and 0.8 bar data sets taken with a pipette tip of 76.9 µm and the rest with a tip size of 102.4 µm.

A linear fit of the same data can be seen in figure 4.5. Here, every pressure is given a different color and symbol. Unfortunately, not all data was achieved with the same

pipette tip, therefore, the slope of the lines vary between tip sizes. At pressures 0.6 and 0.8 bar, a tip of 76.9 µm was used. A second tip of side 102.4 µm was used for the remaining lines.

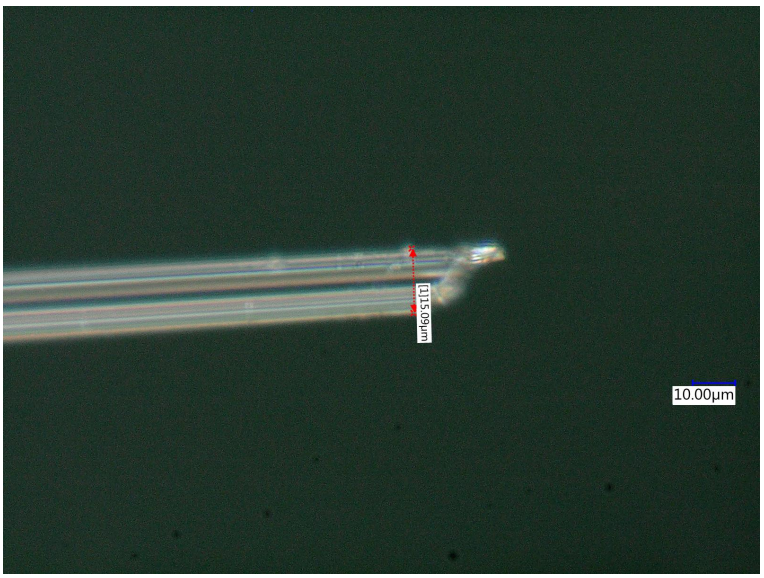


Figure 4.4: Example of how tip outer diameter was measured. Tip shown is smallest pipette achieved; tip of this size were used to inject fish as can be seen in figure 4.9.

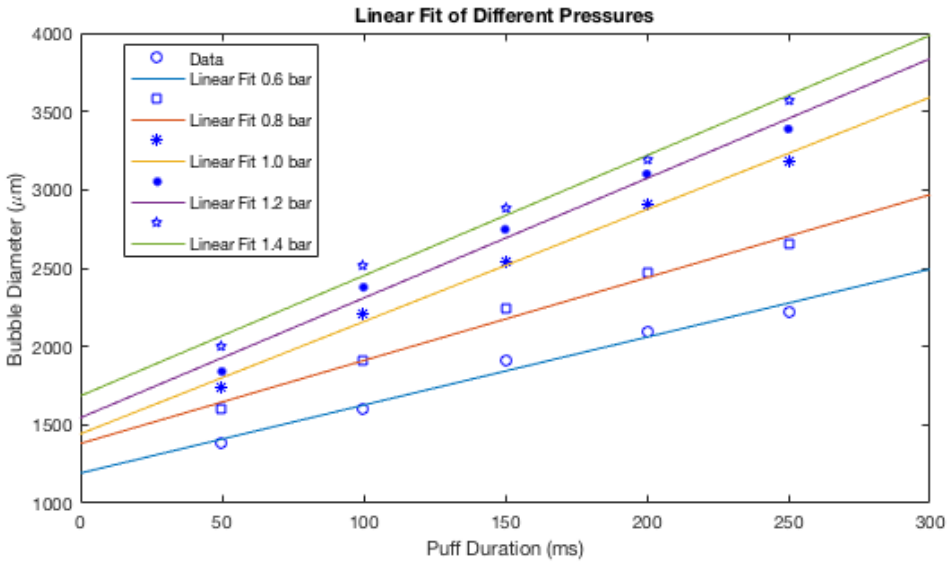


Figure 4.5: Linear relationship between puff duration and pressure.

Pressure (bar)	Slope (µm/ms)	y-intercept
0.6	4.33	1191.9
0.8	5.29	1380.0
1.0	7.15	1442.3
1.2	7.63	1545.2
1.4	7.69	1684.8

Table 4.3: Analysis of slope and y intercepts from figure 4.5

To determine the consistency of bubble formation, 16 bubbles were produced at the same pressure and time duration, 1.0 bar and 100 milliseconds respectively. This produced a standard deviation of 20.059 µm, with an average bubble size of 1522.3 ± 20.059 µm. A linear fit of the data points can been seen in figure 4.6. This leads to a normalized standard deviation of 0.0132.

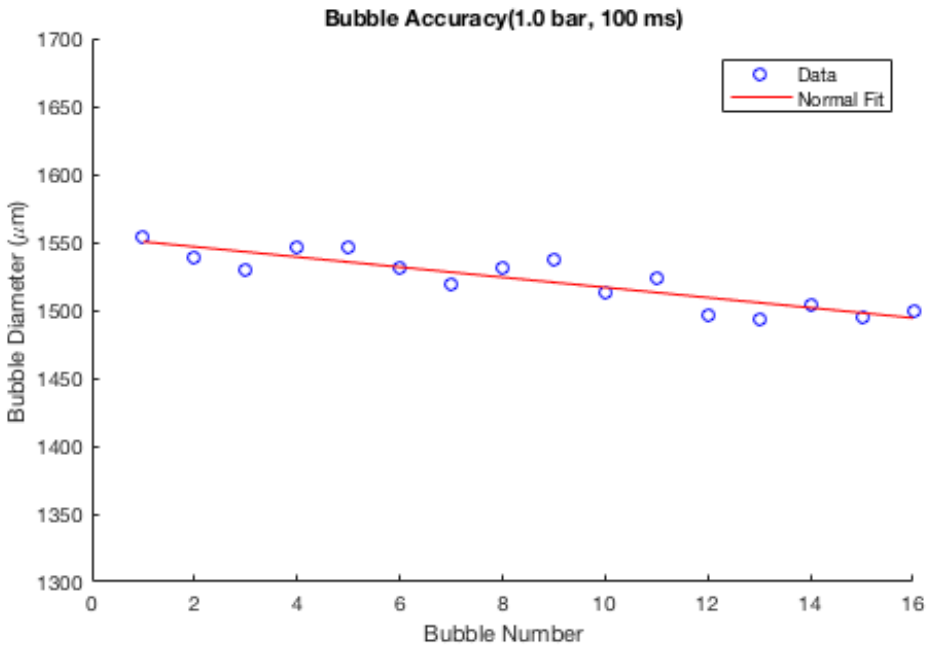


Figure 4.6: Accuracy measurement of 16 bubbles at 1.0 bar and 100 ms. Higher bubble number were measured slightly later then earlier bubbles. A standard deviation of 20.059 µm and average of 1522.3 ± 20.059 µm was calculated. Produced with pipette of size 76.9 µm

Due to the exposure to air, the bubbles are allowed to evaporate. This is possibly why a slight negative slope can be seen in the bubble measurements in figure 4.6. During measurements presented in figure 4.3, the time between bubble placement and measurement varies more than in the accuracy test seen in figure 4.6.

To generate a volume estimate a series of water puffs at 1 bar and 50 ms was ejected and weighed. This was repeated several times to generate the linear relationship seen in figure 4.8. A simple conversion from weight to volume gives a volume of 2.3375µL. This experiment was again produced using a tip of size 123.6 µm.

Due to the large correlation between pipette tip size and bubble size, a new tip was pulled and used to inject a zebrafish with dye. This tip can be seen in figure 4.4. Although no volume measurement was done with this tip, a zebrafish was injected with the proper volume of liquid as can be seen in figure 4.9. This demonstrates the possibilities when a pipette of proper size is used.

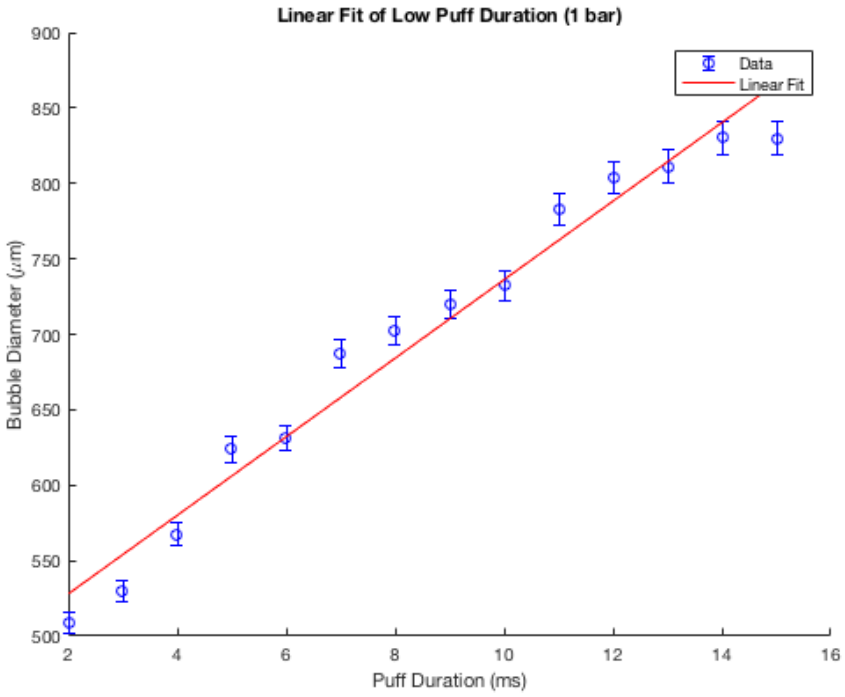


Figure 4.7: Low end measurements ranging from 2-15 milliseconds with normalized error from accuracy measurements. Produced with pipette of size 76.9 μm

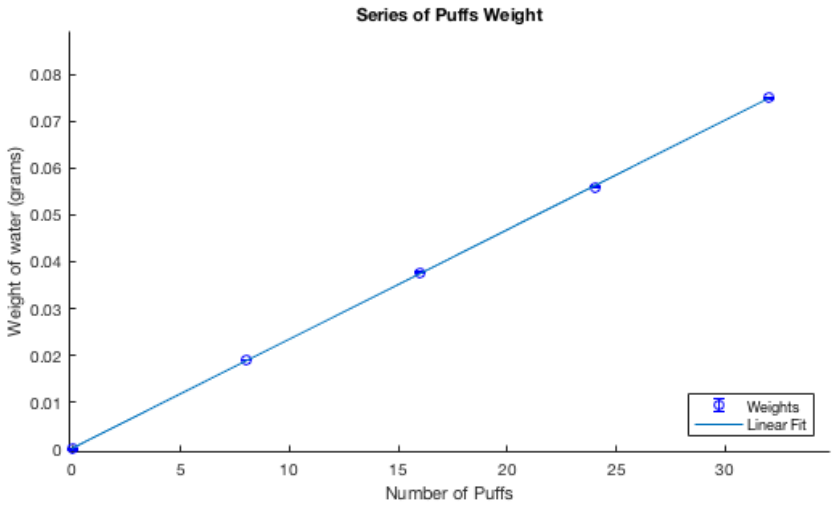


Figure 4.8: Weights of a series of 8 puffs at 1 bar and 50 ms. A clear linear relationship is also displayed, with measurement error of $\pm 10^{-4}$ gram.

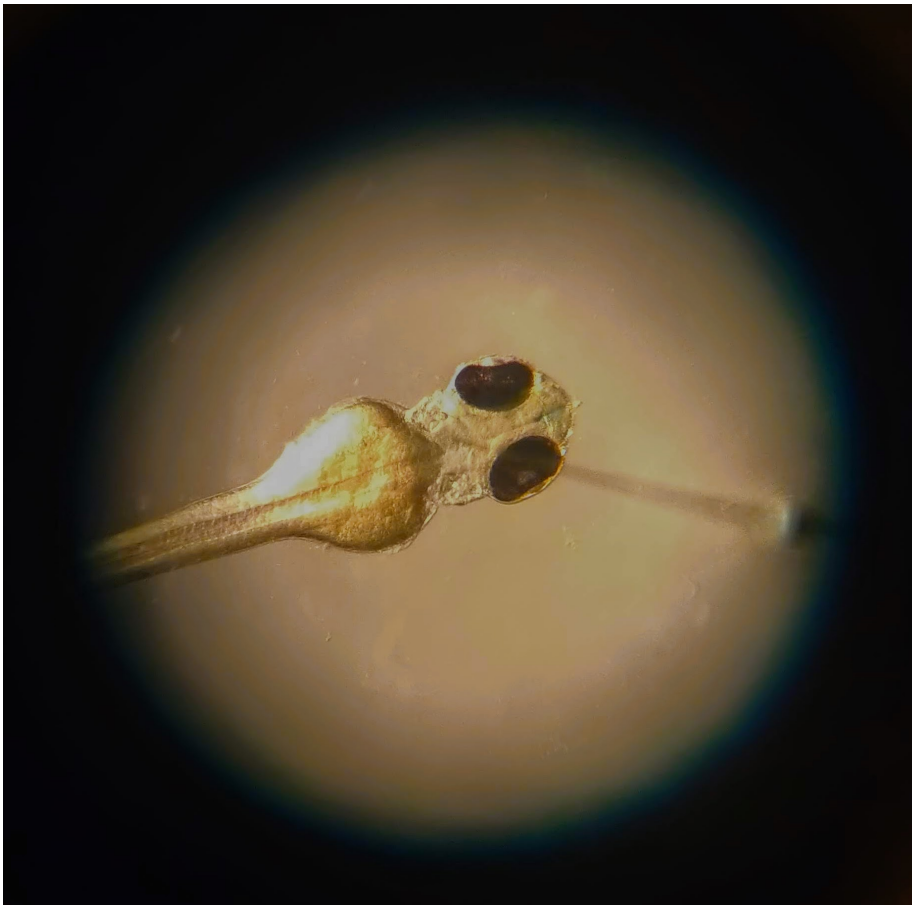


Figure 4.9: A Zebrafish was injected with a very small volume of dye. This is difficult to see, but there is a blue tint in the frontal portion of the fish head.

5

Discussion and Recommendations

5.1. Hardware and Software

The system works as expected and seems to benefit from the added LCD as well as the new switch. However, occasionally the LCD doesn't eliminate its bottom row. This problem occurs when the Arduino is slightly out of place, therefore, care needs to be taken when plugging in the USB port to ensure the Arduino stays in place. This has no further impact on the rest of the system; only the LCD is effected.

According to the spec sheet given by FESTO the solenoid should have a limit to how quickly it can open and close, however, this was not evident when testing the lower end of the scale (figure 4.7). FESTO claims open time of 1.7 ms and close time of 2 ms [7]. This would lead to a lower bound of the solenoid; yet there is a clear difference between 2 and 3 ms bubble as can be seen in figure 4.7. This could be due to the solenoid not opening fully and thus allowing a smaller amount of air through. It is worth noting that the linearity, even given the large uncertainty,was observed in every experiment. Further testing to the limits of the solenoid could be done to get a better understanding of the limits of the system hardware.

The Arduino software works well and without failure. In combination to the Python GUI a new level of control is added. It often proved easier to use the Python code to adjust the pulse duration instead of the knob; as well as adjustment of the programmed sequence. This was used to create bubbles of the same size and allow the user to focus fully on bubble placement.

The system was used to inject a young zebrafish with a colored dye. This can be seen in in figure 4.9. The pipette tip used was also much smaller than the tips used during other measurements. This in combination with higher higher pressure used, 1.4 bar, produced bubbles of proper size and volume. The success of this test validated the second research question and proves the usefulness of the system.

5.2. Linearity and Errors

Throughout the testing there arose a clear correlation between tip diameter, pressure, pulse duration, and bubble volume. To better understand this correlation a method to consistently produce pipette tips of the same gauge is needed. This could

be achieved with a better understanding of the pipette puller, thereby producing a consistent variable. Thus allowing a new calibration method can be used to determine the necessary pressure and puff length to produce desired volumes.

Throughout testing, a linear correlation was observed numerous times. This can be seen in figures 4.3, 4.7, and 4.8. A slope of $2^{1/3}$ was expected (i.e. doubling in time = doubling in volume) when measuring the diameter, however, this was only truly achieved when measuring the weights of multiple bubbles, seen in figure 4.8. There is an inconsistency with this behavior in figure 4.3; mainly with 0.6 and 0.8 data set. These two data sets were produced using a smaller tip size when compared to the remaining lines, however, it was not expected to lead to such a different slope for the linear fit. A cause for this inconsistency could be due to a higher hydraulic pressure caused by the smaller tip size. This can be overcome by using a higher pressure. Thereby supporting the recommendation of higher pressures by Bakker [4] and Forman et al. [8]. Another factor that could influence the bubble formation is capillary action due to water adhesion to the pipette tube. Capillary action is also increased when the pipette diameter is smaller.

Due to the consistency of the bubbles produced by the microinjector, weighing a series of bubbles could be an accurate and easy way to determine a volume of the bubbles produced. However, with very small bubbles a larger sequence of bubbles would be needed and thus could lead to inaccuracies in calculated volume due to slight variance per bubble produced. Evaporation will still play a role with this method as well.

Lastly, bubble evaporation is an effect that should not be underestimated. Evaporation accounted for a larger error than first estimated. This, in combination with asymmetry in bubble shape, accounts for the unexpected large variance when measuring the first data set as can be seen by figures 2.5 and 4.3. Asymmetry in bubble shape could be overcome by placing the bubbles on a hydrophobic surface. Even when bubbles are produced relatively quickly after each other, as was the case in figure 4.6, a downward trend could still be observed. With smaller bubbles this effect will even play a larger role. Due to this fact, measurements of picoliter volumes could prove to be quite the challenge. The normalized standard deviation is not comparable to that stated in the introduction. This is due to the diameter of a non spherical bubble being measured instead of the volume of a spherical bubble. However, the deviation is quite low and well within a acceptable range.

5.3. Microfluidics

As mentioned before, a variance of the system described in this paper can be used to stimulate the lateral sides of the fish. To achieve this either an additional solenoid needs to be added, or a valve to allow for multiple channels to be controlled by the single solenoid. These channels would need to be attached to a reservoir that could store enough water to stimulate the fish multiple times. One such a reservoir could be a simple syringe, however, attaching this to the air tubing might prove difficult.

If an entirely new system was to be built for this purpose, replacement of the solenoid should also be researched. The Arduino is able to send out 5V TTL pulses, with such a pulse an servo replacing the solenoid can also easily be controlled to depress a syringe. Given the Arduino has additional digital output ports multiple servos

could be controlled in parallel. The syringe would then eject a small amount of water against the lateral side of the fish and thus provide stimulation.

6

Conclusion

In this research, a reliable way of producing bubble of size ranging from nanoliter to microliter volumes was established. By modifying the OpenSpritzer design an additional range of freedom was given to the system as well as an increase in ease of use. An LCD displays prudent information such as pulse duration and error codes if they occur. In addition, a Python GUI allows the user to determine a specific pulse duration as well as reprogram the pulse sequence. Although no exact volume calibration was achieved, the system does consistently produce bubbles of similar diameter. Thereby proving its ability to reliably inject DNA/RNA into zebrafish embryos.

Furthermore, a modification of the system described in this paper can be used to stimulate fish response. Additional research is needed for proper implementation, yet the first steps were taken. When compared to the commercial alternative, the Picospritzer, this system is equal in performance and delivers greater control at a fraction of the cost.

Bibliography

[1] Misha B Ahrens, Michael B Orger, Drew N Robson, Jennifer M Li, and Philipp J Keller. Whole-brain functional imaging at cellular resolution using light-sheet microscopy. *Nature methods*, 10(5):413, 2013.

[2] Jasper Akerboom, Tsai-Wen Chen, Trevor J Wardill, Lin Tian, Jonathan S Marvin, Sevinç Mutlu, Nicole Carreras Calderón, Federico Esposti, Bart G Borghuis, Xiaonan Richard Sun, et al. Optimization of a gcamp calcium indicator for neural activity imaging. *Journal of Neuroscience*, 32(40):13819–13840, 2012.

[3] Tom Baden, Andre Maia Chagas, Greg Gage, Timothy Marzullo, Lucia L Prieto-Godino, and Thomas Euler. Open labware: 3-d printing your own lab equipment. *PLoS biology*, 13(3):e1002086, 2015.

[4] Petra Bakker. Injection zebrafish embryos/ micro injections. November 2016.

[5] Henrik Bruus. *Theoretical microfluidics*, volume 18. Oxford university press Oxford, 2008.

[6] Raphaël Candelier, Meena Sriti Murmu, Sebastián Alejo Romano, Adrien Jouary, Georges Debrégeas, and Germán Sumbre. A microfluidic device to study neuronal and motor responses to acute chemical stimuli in zebrafish. *Scientific reports*, 5:12196, 2015.

[7] *solenoid valve MHE2-MS1H-3/2G-M7-K*. FESTO, 1 2019.

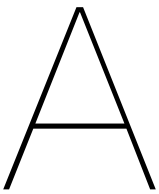
[8] Chris J Forman, Hayley Tomes, Buchule Mbobo, RJ Burman, M Jacobs, T Baden, and JV Raimondo. Openspritzer: an open hardware pressure ejection system for reliably delivering picolitre volumes. *Scientific reports*, 7(1):2188, 2017.

[9] Parker Hannifin. *PICOSPRITZER III Manual FAQ's*. Parker Precision Fluidics Division, 45 Route 46 East Pine Brook, NJ 07058.

[10] Dongqing Li. Picoliter flow calibration. In *Encyclopedia of microfluidics and nanofluidics*, page 1650. Springer Science & Business Media, 2008.

[11] Kwang W Oh, Kangsun Lee, Byungwook Ahn, and Edward P Furlani. Design of pressure-driven microfluidic networks using electric circuit analogy. *Lab on a Chip*, 12(3):515–545, 2012.

[12] Leonard I Zon and Randall T Peterson. In vivo drug discovery in the zebrafish. *Nature reviews Drug discovery*, 4(1):35, 2005.



Codes

A.1. Arduino Code

This code is an adaptation of the code provided by Openlabware. [8].

```
1 const int TIMEUNIT = 25;     // Base Time Unit in milliseconds.
2 const int POT_VALUE_LOW = 0; // Lowest value available from the pot.
3 const int POT_VALUE_HIGH = 1023; // Highest value available from the
      pot (measured value). Can make this higher by reducing resistor
      value in series with pot, but go below 125 Ohms and current will
      exceed 40 mA max on Analog pin.
4 const int SENSITIVITY = 20; // Number Of allowed settings of
      Potentiometer is (Sensitivity) (1 based...) make sure that Pot
      range = (High-low)/sensitivity isn't smaller than potnoise
5 const int POT_NOISE = 5; // Amplitude of fluctuations in the Pot
      Value when it is static. Measured value.some power supplies are
      noisier than others.
6 const int STABILITY_TIME = 500; // number of milliseconds for which
      stability must be achieved when adjusting pot to accept new value
      .
7 const int MONITOR_DELAY = 100; // delay between loops for analog
      monitoring cycles - allows user time to adjust pot a significant
      amount between loops.
8 const int PAUSE_BEFORE_BLINK = 500; // number of milliseconds to
      pause before starting blinks.
9 const int BLINK_ON_DURATION = 100; // this combo seems to work
      nicely.
10 const int BLINK_OFF_DURATION = 250;
11 const int NUM_PROGRAMMED_PULSES = 28; // number of pulses to
      implement in hard coded sequence
12 int PROGRAMMED_SEQUENCE[NUM_PROGRAMMED_PULSES] = {1, 2, 3, 4, 5, 6,
      7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400,
      500, 600, 700, 800, 900, 1000}; // sequence of pulses in ms
13 int Python_Array[28] = {0}; // Empty array to process Python Data
14 const int PROGRAMMED_PULSE_DELAY = 1000; // delay between pulses.
```

```
15 const int MONITOR_FOOTSWITCH_PERIOD = 500; // seems long enough for
     a double click. Can get a triple in there too.
16 const int FOOTBUTTONDELAY = 1000; // liberally dispersed throughout
     the code to give time for people to release the button.
17 char serialData;
18 int SwithcValue;
19 // Set up the pins
20 const int D_FootButtonPin = 2;
21 const int D_LEDPin = 10;
22 const int A_PotPin = 0;
23 const int D_SwitchPin = 8;
24 const int D_SolenoidPin = 5;
25
26 // Declare global variables - These are designed to be adjusted
     simultaneously using the SetPulseDuration function.
27 long PulseDuration = 0;
28 long SerialInput = 0;
29 int PotSetting = 0;
30 int PotValue = 0;
31 int numButtonClicks=0;
32
33 #include <LiquidCrystal.h>
34 LiquidCrystal lcd(12, 11, 7, 6, 4, 3);
35
36 void setup() {
37
38   //initialiase serial port for output messages
39   Serial.begin(9600);
40
41   // initialize the pins
42   pinMode(A_PotPin, INPUT);
43   pinMode(D_SwitchPin, INPUT);
44   pinMode(D_FootButtonPin, INPUT);
45   pinMode(D_LEDPin, OUTPUT);
46   pinMode(D_SolenoidPin, OUTPUT);
47   lcd.begin(16,2);
48
49   // set initial conditions on pins.
50   digitalWrite(D_SolenoidPin, LOW);
51   digitalWrite(D_LEDPin, LOW);
52
53   // monitor the pot to make sure it is stable, includes lighting
       the LED to show aliveness...
54   MonitorAnalog();
55   SetPulseDuration(); // set the initial pulse duration and report
       new values over serial link.
56   LEDBlinks(); // blink the LED to indicate the pot Setting.
57 }
58
```

```
59 // function reads the potentiometer and populates the three global
       variables PotValue, PotSetting and PulseDuration
60 // with mutually consistent values. These global values are not
       intended to be set using any other function.
61 void SetPulseDuration() {
62   PotValue = analogRead(A_PotPin);
63
64   // compute the Potentiometer Setting as an integer between 1 and
       Sensitivity
65   PotSetting = int(float(SENSITIVITY - 1) * float(PotValue - (
       POT_VALUE_LOW + 20))/float( (POT_VALUE_HIGH - 20) - (
       POT_VALUE_LOW + 20))) + 1;
66
67   //Compute pulse Duration in milliseconds
68   PulseDuration = PotSetting * TIMEUNIT;
69 //   Serial.println("Computing Pulse Duration in Milliseconds.");
70 //   Serial.println("Time Unit");
71 //   Serial.println(TIMEUNIT);
72 //   Serial.println("PotValue");
73 //   Serial.println(PotValue);
74 //   Serial.println("PotSetting");
75 //   Serial.println(PotSetting);
76 //   Serial.println("PulseDuration");
77 //   Serial.println(PulseDuration);
78 }
79
80 // function flashes the LED PotSetting number of times
81 void LEDBlinks() {
82   int i=0;
83
84   // Switch LED off for PAUSE_BEFORE_BLINK milliseconds prior to the
       blinks starting.
85   digitalWrite(D_LEDPin, LOW);
86   delay(PAUSE_BEFORE_BLINK);
87
88   for (i = 0; i< PotSetting; i++) {
89     digitalWrite(D_LEDPin, HIGH);
90     delay(BLINK_ON_DURATION );
91     digitalWrite(D_LEDPin, LOW);
92     delay(BLINK_OFF_DURATION );
93   }
94 }
95
96 // monitors the foot switch and counts the pulses
97 void MonitorFootSwitch() {
98   unsigned long StartTime = 0;
99   int previousValue = 1;  //assume we get here with the first switch
       still down.
100  int currentValue = 1;
```

```
101   int changesDetected = 1;  // the very first pulse to get us here
          from the main loop.
102
103   // get the start time
104   StartTime = millis();
105
106   while ( (abs(millis() - StartTime)) < MONITOR_FOOTSWITCH_PERIOD)
107   {
108     previousValue = currentValue;
109     currentValue = digitalRead(D_FootButtonPin);
110     if (previousValue != currentValue) {
111       changesDetected = changesDetected + 1;
112     }
113   }
114
115   if (currentValue==1) {
116     changesDetected +=1; //assume that if it was high when we left
          the loop it will go low - means we always get an even number
          of changes.
117   }
118   numButtonClicks = floor((float)changesDetected/2.0);
119 }
120
121 // continuously monitors the value on the Analog pin until PotValue
      is stable (i.e. no changes bigger than POT_NOISE)
122 // for STABILITY_TIME milliseconds.
123 void MonitorAnalog() {
124   int PreviousPotValue = 0;
125   int CurrentPotValue = 0;
126   int KeepLooping = 1;
127   unsigned long StartTime = 0;
128   unsigned long CurrentTime = 0;
129
130   // Set LED Pin high, so user knows that system has detected
        potentiometer change.
131   digitalWrite(D_LEDPin, HIGH);
132   // get the start time
133   StartTime = millis();
134
135   // start looping
136   while (KeepLooping) {
137     // Store the PotValue from the previous loop.
138     PreviousPotValue = CurrentPotValue;
139     //LCD shows current Pot value
140     lcd.clear();
141     lcd.setCursor(0,0);
142     lcd.print("Calibrating");
143     // read in the new PotValue.
144     CurrentPotValue = analogRead(A_PotPin);
```

```
145     // Serial.println(CurrentPotValue);
146     // if the current PotValue has changed from the previous
            PotValue by more than POT_NOISE
147     // then restart timer.
148     if (abs(PreviousPotValue - CurrentPotValue)> POT_NOISE) {
149       StartTime = millis();
150     }
151
152     // get the current time
153     CurrentTime = millis();
154
155     // Check for an overflow condition for the timer. The timer hits
            an overflow every fifty days.
156     // If the system is left active for that long it is conceivable
            that the user adjusts the potentiometer
157     // at precisely the moment the overflow occurs thus potentially
            getting locked into a 50 day wait.
158     // If CurrentTime is lower than the StartTime then we must have
            hit the overflow condition.
159     // Solution is simply to restart the timer.
160     // The user will just have to wait an additional STABILITY_TIME
            milliseconds for the stability condition to emerge.
161     // Better than waiting an additional 50 days for the loop exit
            conditions to emerge.
162     if (CurrentTime <= StartTime){
163       StartTime = millis();
164     }
165
166     // if the time elapsed since the StartTime exceeds
           STABILITY_TIME then stop looping
167     if ((CurrentTime - StartTime) > STABILITY_TIME) {
168       KeepLooping = 0;
169
170     }
171
172     delay(MONITOR_DELAY); // put in delay to ensure that user has
            time to change the pot a significant amount between loops.
173   }
174
175   // switch off LED.
176   digitalWrite(D_LEDPin, LOW);
177
178 }
179
180 void rapidFlash(int numPulses, int delayOn, int delayOff){
181   int currentPulse = 0;
182     for (currentPulse = 0; currentPulse < numPulses; currentPulse
          +=1) {
183       digitalWrite(D_LEDPin, HIGH); // light the LED
```

```
184         delay(delayOn); // wait for the pulse duration.
185         digitalWrite(D_LEDPin, LOW);
186         delay(delayOff); // wait for the pulse duration.
187       }
188 }
189
190 void loop() {
191   int pulsed = 0; // Set to zero at the beginning of each loop.
192   int PotValueNew = 0; //initialise the new pot value
193   int currentPulse = 0; // counter for number of pulses
194   unsigned long startTime = 0; // timer
195   int SwitchValue=0;
196   digitalWrite(D_SolenoidPin, LOW);  //ensure output pins are set to
          low at the start of each loop.
197   digitalWrite(D_LEDPin, LOW);
198   // Read a new pot value.
199   PotValueNew = analogRead(A_PotPin);
200   SwitchValue = digitalRead(D_SwitchPin);
201
202
203   if(Serial.available() > 0){ //read python GUI for Pulse duration
204     for (int i = 0; i < 28; i++){
205       Python_Array[i] = Serial.parseInt(); //fills entire python
            array with values in serial buffer
206       if (Python_Array[i] == 0){// if next value is already zero all
            other values will be zero and thus skip next step
207        for (int j = i; j < 28; j++)
208         {
209           Python_Array[j] = 0;//fill rest of array with zeros
210         }
211         break;
212       }
213      }
214
215
216     if (Python_Array[0] = 1){
217      PulseDuration = Python_Array[1]; //first value of new python
           array determines what needs to be done
218       }
219     if (Python_Array[0] = 2){ // if value is two python sequance
          gets coppied to programmed sequance
220      memcpy(PROGRAMMED_SEQUENCE, Python_Array+1,sizeof(
           PROGRAMMED_SEQUENCE));
221
222     }
223   }
224
225   // if the pot value has changed by more than POT_NOISE from the
       currently set value then
```

```
226   // jump to a pot monitoring loop which exits when the pot is
          stable for STABILITY_TIME milliseconds.
227   if (abs(PotValueNew - PotValue) > POT_NOISE)
228   {
229     MonitorAnalog(); // continuously reads the analog until it is
          stable for more than STABILITY_TIME;
230     SetPulseDuration(); // Read the latest value and set the new
          pulse duration.
231     lcd.clear();
232     lcd.print(PulseDuration);
233     lcd.setCursor(4,0);
234     lcd.print("ms");
235     Serial.println(0); //place holder to indicate given information
236     Serial.println(PulseDuration);
237     Serial.println(0);  //place holder
238     LEDBlinks();// blink the LED the appropriate number of times
          with the latest values.
239   }
240
241   // check for button presses
242   numButtonClicks = 0; // assume no button clicks have happened/
          reset after last loop.
243   numButtonClicks = digitalRead(D_FootButtonPin); // read the button
          pin.
244   if (numButtonClicks == 1) {
245     MonitorFootSwitch(); // as soon as button is pressed we monitor
          it for a some time to see if it is pressed repeatedly.
246     // function sets the global variable numButtonClicks.
247   }
248   if (SwitchValue==HIGH) // dispay if switch in in sequance mode
249   {
250     lcd.clear();
251     lcd.print("Sequance Mode On");
252   }
253   //do behaviour based on global variable numButtonClicks. One is a
          single pulse
254   if (numButtonClicks==1 && SwitchValue== LOW) {
255         digitalWrite(D_SolenoidPin, HIGH);
256         digitalWrite(D_LEDPin, HIGH);
257         Serial.println(POT_NOISE); //will change to different
              information such as Volume
258         Serial.println(PulseDuration);
259         Serial.println(D_LEDPin);//will change to different
              information such as total time
260         lcd.setCursor(0,1);// light the LED
261         lcd.print("Single Puff");
262         delay(PulseDuration); // wait for the pulse duration.
263         digitalWrite(D_SolenoidPin, LOW);  // close the solenoid and
              switch off the LED.
```

```
264           digitalWrite(D_LEDPin, LOW);
265
266           }
267
268   // switch to perform the pulse sequence.  Click to interrupt.
269   if (numButtonClicks==1 && SwitchValue==HIGH)
270   {
271     numButtonClicks = 0;
272     delay(FOOTBUTTONDELAY);
273     for (currentPulse = 0; currentPulse < NUM_PROGRAMMED_PULSES;
          currentPulse +=1)
274     {
275
276       lcd.clear();
277       lcd.setCursor(10,0);
278       lcd.print(currentPulse+1); // display pulse number
279       lcd.setCursor(0,0);
280       lcd.print("Sequence #");
281       lcd.setCursor(0,1);
282       lcd.print(PROGRAMMED_SEQUENCE[currentPulse]); // display pulse
            value
283       lcd.setCursor(5,1);
284       lcd.print("ms");
285       delay(750); // have time between the next value is displayed
          and the puff is exicuted
286       Serial.println(currentPulse+1); //will change to different
          information such as Volume
287       Serial.println(PROGRAMMED_SEQUENCE[currentPulse]);
288       Serial.println(D_LEDPin);
289       digitalWrite(D_SolenoidPin, HIGH);
290       digitalWrite(D_LEDPin, HIGH); // light the LED
291       delay(PROGRAMMED_SEQUENCE[currentPulse]); // wait for the
          pulse duration of the current time.
292       digitalWrite(D_SolenoidPin, LOW);  // close the solenoid and
          switch off the LED.
293       digitalWrite(D_LEDPin, LOW);
294
295
296
297
298       startTime = millis();
299       while ((numButtonClicks == 0)&&((millis() - startTime) <
          PROGRAMMED_PULSE_DELAY))
300       {
301         numButtonClicks = digitalRead(D_FootButtonPin);
302       }
303
304       if (numButtonClicks>0)
305       {
```

```
306             rapidFlash(5,100,100);
307             lcd.clear();
308             lcd.print("Sequence Broken");
309             break;          //break sequance by pushing button again,
                    also display sequance is broken
310         }
311         if (PROGRAMMED_SEQUENCE[currentPulse+1]==0) //sequance ends if
                next puff value is 0
312                                                     //nessassary due
                                                          to how python
                                                          array is formed
313         {
314           lcd.clear();
315           lcd.print("Sequence Ended");
316           break;
317         }
318       }
319       numButtonClicks = 0;
320       delay(FOOTBUTTONDELAY);
321     }
322
323     if (numButtonClicks >2){ // an error alert
324        rapidFlash(5, 100, 100);
325        delay(FOOTBUTTONDELAY);
326        lcd.clear();
327        lcd.print("Error:too many");
328        lcd.setCursor(0,1);
329        lcd.print("clicks");
330        Serial.println(404); //send number to indicate error occured
331        Serial.println(PulseDuration);
332        Serial.println(0); //
333        delay(2000);
334
335     }
336         delay(150);
337         lcd.clear();
338         lcd.print(PulseDuration);
339         lcd.setCursor(4,0);
340         lcd.print("ms");
341 }
```

A.2. Python Code

A.2.1. Python GUI classes

Code was run on version 3.7.1 tkinter is a included package with python 3 or above. serial will need to be imported via pip install or a different method.

```
from tkinter import *
from tkinter import ttk
```

```
import serial.tools.list_ports
import serial
from time import sleep
class MainWindow:
    def __init__(self,master):

        #Setup
        self.master = master
        master.title("OpenSpritzer GUI")

        #Get nessasary Port info
        ports = serial.tools.list_ports.comports() #make list of ports where
        available_ports = []
        for p in ports:
            available_ports.append(p.device)
        print(available_ports) #just print to command window

        #Place port info into combo box
        self.cb = ttk.Combobox(master,values = available_ports)
        self.cb.pack()
        self.cb.bind('<<ComboboxSelected>>')
        self.cb.current(0) #random initial port
        self.arduinoData = serial.Serial(self.cb.get(),9600)
        # selects port where data will be sent
        MainWindow.Arduinodata = serial.Serial(self.cb.get(),9600)

        #Switch Button

        self.switch_btn = Frame(master)
        self.switch_btn.pack()

        self.switch_var = IntVar()
        Single_button = Radiobutton(self.switch_btn, text="Single
        Puff", variable =self.switch_var,
                        indicatoron=0, value=1, width=15)
        Sequance_button = Radiobutton(self.switch_btn,
        text="Sequance Mode", variable=self.switch_var,
                            indicatoron=0, value=2, width=15)
        Single_button.pack(side="left")
        Sequance_button.pack(side="left")

        # Entry Box
        self.e = Entry(master)
        self.e.pack()
        entry = self.e.get()
        self.e.focus_set()

        # Buttons
        self.config_btn = Button(master, text="Config.",
```

```
        width=10, command=self.callback)
        self.config_btn.pack()

        self.reset_btn = Button(master, text="Reset",
        width=10, command=self.reset)
        self.reset_btn.pack()

        #Label

        MainWindow.v = StringVar()
        self.lab = Label(master, textvariable=MainWindow.v)
        self.lab.pack()

        MainWindow.PuffNum = StringVar()
        self.lab2 = Label(master, textvariable=MainWindow.PuffNum)
        self.lab2.pack()

        MainWindow.PuffTim = StringVar()
        self.lab3 = Label(master, textvariable=MainWindow.PuffTim)
        self.lab3.pack()

        MainWindow.PuffVol = StringVar()
        self.lab4 = Label(master, textvariable=MainWindow.PuffVol)
        self.lab4.pack()



    def callback(self): #function to send input data to arduino
        sleep(0.5)
        print ("Port:", self.cb.get(), self.e.get())
        MainWindow.v.set("Puff␣"+ self.e.get() +"␣ms")
        if self.switch_var.get() == 1:
        #dependent on button value. fist value is changed
            datatowrite = "1,"+self.e.get()
            self.arduinoData.write(datatowrite.encode())

        else:
            datatowrite = "2,"+self.e.get()
            # 2 as first value will tell Arduino to write list to
            Sequance list
            self.arduinoData.write(datatowrite.encode())
            print(len(datatowrite))


    def reset(self):
        MainWindow.reset = StringVar()
        MainWindow.reset = 1
```

```
if __name__ == '__main__':
        obj = MainWindow()
```

A.2.2. Python GUI command

```
import serial.tools.list_ports
import serial
from tkinter import *
from tkinter import ttk
import tkinter
import threading
from time import sleep
from GUI_Classes import MainWindow
master = Tk()
MainWindow(master)
arduinoData = MainWindow.Arduinodata
Plist = [] #empty list
TotPuffDuration = 0
TotPuffNum = 0
TotPuffVol = 0
def read_from_port(ser):
    global TotPuffDuration
    global TotPuffNum
    global TotPuffVol
    while True:
        global Plist
        if MainWindow.reset == 1:#check if rest button was pressed
            TotPuffNum = 0
            TotPuffVol = 0
            TotPuffDuration = 0
            print("reset")
            MainWindow.reset = 0
        if len(Plist) >= 3:
        #once 3 values have been writen to list it will reset to empty
            Plist = []
        Arduinoread = arduinoData.readlines(2)#read incoming data
        Arduinodecode = Arduinoread[0].decode()
        Puff_length = Arduinodecode.rstrip()
        Plist.append(Puff_length)#create list

        if len(Plist) == 3:
        #only read list once it has been filled to 3
            if Plist[0] == '404':
            #dependent on 1st value different information is
            communicated
                MainWindow.v.set("Error: Too many clicks")
                sleep(5)
                MainWindow.v.set("Puff "+ Plist[1] +" ms")
```

```
        elif Plist[0] == '0':#updated puff duration
            MainWindow.v.set("Puff␣"+ Plist[1] +"␣ms")

        else:
            print(Plist)#labels change once puff is exicuted
            MainWindow.v.set("Puff␣"+ Plist[1] +"␣ms")
            TotPuffNum = TotPuffNum + 1
            TotPuffVol = TotPuffVol + int(Plist[2])
            TotPuffDuration = TotPuffDuration + int(Plist[1])
            MainWindow.PuffTim.set("Total␣Puff␣Length␣"+
            str(TotPuffDuration)+"␣ms")
            MainWindow.PuffNum.set("Total␣Puff␣Number␣"+
            str(TotPuffNum))
            MainWindow.PuffVol.set("Total␣Puff␣Volume␣"+
            str(TotPuffVol)+"␣pL")


thread = threading.Thread(target=read_from_port, args=(arduinoData,))
thread.start()
master.mainloop()
```

A.3. 3D Code

use <GoodBox.scad>; tol = 0.2;

//Regulator- these are measurements taken from the regulator RegHeight = 50.1; RegWidth = 50.1; RegLength = 52.75; RegKnobDiam = 35.8 + tol;//35.7; AirHoseInDiameter = 13.3 + tol; AirHoseInYOffset = (36.95 + 48.05)/2 + .4; AirHoseOutDiameter = 10.3 + tol; //AirHoseOutYOffset = (15.78 + 5.72)/2; // (this one is port 1) AirHoseOutYOffset = (13.49 + 19.18)/2; // (this one is port 2) GaugeDiameter = 9.8 + tol; GaugeYOffset = (13.2 + 22.72)/2 +.3; CollarHeight = 9;

//solenoid - measurements taken from the solenoid. SolLength=88.6; SolBodyLength = 73.2; SolWidth = 32.5; SolHeight = 10.1; SolRetWidth = 4; SolRetOffsetX = 6.5;//5; SolShelfDepth = 4;

// basic box params - size of the box and screw holes. Smoothness of corners etc. Thickness = 1.5; CornerR = 4; XLen = 100; //external sizes YLen = SolLength + 2*Thickness + 2*CornerR+40; ZLen = RegHeight + Thickness; Smoothness = 60; HeadR = 2; HeadD = Thickness; NutD = ZLen/1.05; NutR = 7/2; //6.5/2 too small; //5 wayyy too big. ScrewR = 2.5; ScrewD = ZLen;

// control where lid appears relative to box. Probably best to leave as is because // holes might not move in same way as lid. LidOffset = 50; SideBySide= false; Centering= true;

// switch on lid and box independently. Lid = true; Box = true;

// sol derived quantities SolYPosition = -YLen/2 + 2*CornerR; SolShelfHeight = ZLen - Thickness - 2 * SolHeight; SolRetHeight = SolShelfHeight + SolHeight;

//Circuit board position and screw stand offs. BoardYPosition = SolYPosition + Sol-

ShelfDepth + 2; BoardEdgeX = 6.5; BoardEdgeY = 4.0; MountPointWidth = 38;//(36.34 + 38.5)/2;//(37.29 + 39.10)/2; MountPointLength = (48.93 + 51.2)/2;//(54.82 + 56.78)/2; MountPointHeight = 5; MountPointInnerDiameter = 2.6; MountPointOuterDiameter = 6;

// spacing betweeen holes HoleSpacing = 7; // side panel HoleSpacingFront = (XLen - Thickness - RegWidth/2 - RegKnobDiam/2)/3; // front panel

// Define cable hole sizes USBHeight = 3.8 + tol; USBWidth = 7.6 + tol; USBWidth2 = 6.5 + tol; BNCDiam = 8.3 + tol; BNCFlatDiam = 8 + tol; BNCFlatbitOffset = BNCDiam - BNCFlatDiam; PotDiam = 6.2 + tol; PotFlatDiam = 8.72 + tol; PotFlatbitOffset = BNCDiam - BNCFlatDiam; ToggleSwitchDiam = 6.2 + tol; TogSwitchWidth = 11.5 + tol; TogSwitchHeight = 12.8 + tol;

PowerJackWidth = 8.9 + tol; //9 PowerJackHeight = 11.15 + tol; //11.25 PowerJackSupportDepth = 8.3 - Thickness; PowerJackSupportHoleDiam = 2.1 + tol; FootSwitchHoleDiam = 5.8 + tol; LEDDiameter = 4.4 + tol;

// determine the position of the cables on the size panel.

// Z-Y offset of the USB hole from it's position relative to the circuit board. USBYOffset = 22.11;//(11.62 + 13.74 + 26.87 + 29.01)/4// (18.99 + 9.64)/2;//(18.99 + 9.64)/2; //measured from the mount point nearest arduino. controls the Y position of the four cable holes on the side panel. USBZOffset = 17.2; // was (18.01 + 14.01)/2 // controls Z position of four cable holes on the side panel

// Y position of the center of the USB socket including its offset and position relative to circuit board USBYPosition = BoardYPosition + BoardEdgeY + MountPointLength - USBYOffset + USBWidth/2;

// the position of the other cable holes in the side panel relative to the USB socket BNCYPosition = USBYPosition + USBWidth/2 + HoleSpacing + BNCDiam/2; PowerJackYPosition = SolYPosition + SolBodyLength + PowerJackWidth/2; FootswitchYPosition = USBYPosition - USBWidth/2 - HoleSpacing - BNCDiam/2;

//Regulator Y Position RegYPosition = YLen/2 - 6.8; RegSpacerDepth = YLen/2 - RegYPosition; RegSpacerWidth = 6;

// call the function build the box PicoSpritz();

// first the bits to add, then the bits to subtract. module PicoSpritz() difference() PicoSpritz$_add$(); $PicoSpritz_sub$();

// all the components to add module PicoSpritz$_add$()

// Set up the basic good box GoodBox(XLen, YLen, ZLen, Thickness, CornerR, Smoothness, HeadR, HeadD, NutR, NutD, ScrewR, ScrewD, LidOffset, SideBySide, Centering, Lid, Box);

if (Box==true) // Regulator Spacer translate([-XLen/2 + Thickness + RegWidth, RegYPosition + RegSpacerDepth/2, Thickness/2]) cube([RegSpacerWidth, RegSpacerDepth, ZLen - Thickness], true);

// Solenoid Shelf Back translate([XLen/2 - Thickness - SolWidth/2, SolYPosition + SolShelfDepth/2, -ZLen/2 + Thickness + SolShelfHeight/2]) cube([SolWidth, SolShelfDepth, SolShelfHeight], true);

// Solenoid rear Retainer translate([XLen/2 - Thickness - SolWidth - SolRetWidth/2, SolYPosition + SolShelfDepth/2, -ZLen/2 + Thickness + SolRetHeight/2]) cube([SolRetWidth, SolShelfDepth, SolRetHeight], true);

// Solenoid Shelf Front translate([XLen/2 - Thickness - SolWidth/2, SolYPosition + SolBodyLength - SolShelfDepth/2, -ZLen/2 + Thickness + SolShelfHeight/2]) cube([SolWidth, SolShelfDepth, SolShelfHeight], true);

// Solenoid Front Retainer1 translate([XLen/2 - Thickness - SolWidth - SolRetWidth/2 + SolRetOffsetX, SolYPosition + SolBodyLength + SolRetWidth/2, -ZLen/2 + Thickness + SolRetHeight/2]) cube([SolRetWidth , SolRetWidth, SolRetHeight], true);

// Solenoid Front Retainer2 translate([XLen/2 - Thickness - SolWidth - SolRetWidth/2, SolYPosition + SolBodyLength - SolShelfDepth/2, -ZLen/2 + Thickness + SolRetHeight/2]) cube([SolRetWidth, SolShelfDepth, SolRetHeight], true);

// MountingPoints // Rear Right translate([XLen/2 - Thickness - BoardEdgeX, BoardYPosition + BoardEdgeY, -ZLen/2 + Thickness + MountPointHeight/2]) cylinder(MountPointHeight, MountPointOuterDiameter/2, MountPointOuterDiameter/2, $fn = Smoothness, true$);

// Rear Left translate([XLen/2 - Thickness - BoardEdgeX - MountPointWidth, BoardYPosition + BoardEdgeY, -ZLen/2 + Thickness + MountPointHeight/2]) cylinder(MountPointHeight, MountPointOuterDiameter/2, MountPointOuterDiameter/2, $fn = Smoothness, true$);

// Front Right translate([XLen/2 - Thickness - BoardEdgeX, BoardYPosition + BoardEdgeY + 2 + MountPointLength, -ZLen/2 + Thickness + MountPointHeight/2]) cylinder(MountPointHeight, MountPointOuterDiameter/2, MountPointOuterDiameter/2, $fn = Smoothness, true$);

// Front Left translate([XLen/2 - Thickness - BoardEdgeX - MountPointWidth, BoardYPosition +2 + BoardEdgeY + MountPointLength, -ZLen/2 + Thickness + MountPointHeight/2]) cylinder(MountPointHeight, MountPointOuterDiameter/2, MountPointOuterDiameter/2, $fn = Smoothness, true$);

// Power Jack Support vertical translate([XLen/2 - Thickness/2 - PowerJackSupportDepth/2, PowerJackYPosition - PowerJackWidth/2 - Thickness/2, -ZLen/2 + Thickness + MountPointHeight + USBZOffset]) cube([PowerJackSupportDepth, Thickness, PowerJackHeight], true);

// Power Jack Support Horiz translate([XLen/2 - Thickness/2 - PowerJackSupportDepth/2, PowerJackYPosition, -ZLen/2 + Thickness + MountPointHeight + USBZOffset - PowerJackHeight/2 - Thickness/2]) cube([PowerJackSupportDepth, PowerJackWidth, Thickness], true);

if (Lid==true) // Lid Blocks for solenoid translate([XLen/2 - Thickness - SolWidth/2, SolYPosition + SolShelfDepth/2, LidOffset - SolHeight/2 - Thickness/2]) cube([SolWidth, SolShelfDepth, SolHeight], true);

translate([XLen/2 - Thickness - SolWidth/2, SolYPosition + 3*SolBodyLength/4 - SolShelfDepth/2, LidOffset - SolHeight/2 - Thickness/2]) cube([SolWidth, SolShelfDepth, SolHeight], true);

// all the holes module PicoSpritz$_sub()//SolenoidLidBlocktrimmingtoallowSolRetainer//translate([XLe$

// LED Hole translate([+XLen/2 - Thickness - RegWidth/2, RegYPosition - GaugeY-Offset, LidOffset]) cylinder(Thickness, LEDDiameter/2, LEDDiameter/2, $fn = Smoothness, true$);

//LCD Hole translate([-XLen/5, 0, LidOffset])rotate(90,[0,0,1]) cube([107, 37, 2*Thickness,],$fn = Smoothness, true$);

//LCD Mounting top left translate([-XLen/5+37/2+3, 107/2+5, LidOffset]) cylinder(Thickness, LEDDiameter/2, LEDDiameter/2, $fn = Smoothness, true$);

//LCD Mounting top right translate([-XLen/5+37/2+3, -107/2-5, LidOffset]) cylinder(Thickness, LEDDiameter/2, LEDDiameter/2, $fn = Smoothness, true$); $//LCDMountingbottom$ $37/2+1.5, 107/2+5, LidOffset])cylinder(Thickness, LEDDiameter/2, LEDDiameter/2$,fn=Smoothnes true); //LCD Mounting bottom right translate([-XLen/5-37/2+1.5, -107/2-5, LidOffset]) cylinder(Thickness, LEDDiameter/2, LEDDiameter/2, $fn = Smoothness, true$);

// Input Airhose Hole translate([-XLen/2 + Thickness/2,SolYPosition + AirHoseOutYOffset-5, -ZLen/2 + Thickness + SolShelfHeight + SolHeight/2]) rotate(90,[0,1,0]) cylinder(2*Thickness, AirHoseInDiameter/2, AirHoseInDiameter/2, $fn = Smoothness, true$);

// Potentiometer difference() // POT Main Hole translate([-XLen/2 + Thickness + RegWidth/2 + RegKnobDiam/2 + HoleSpacingFront, YLen/2 - Thickness/2, -ZLen/2 + Thickness + RegHeight/2]) rotate(90, [1, 0, 0]) cylinder(Thickness+1, PotDiam/2, PotDiam/2, $fn = Smoothness, true$);

// toggle switch translate([-XLen/2 + Thickness + RegWidth/2 + RegKnobDiam/2 + HoleSpacingFront + HoleSpacingFront, YLen/2 - Thickness/2, -ZLen/2 + Thickness + RegHeight/2]) rotate(90, [1, 0, 0]) cylinder(Thickness+1, ToggleSwitchDiam/2, ToggleSwitchDiam/2, $fn = Smoothness, true$);

// cube to allow switch to be installed. Y position solenoid shelf front. translate([-XLen/2 + Thickness + RegWidth/2 + RegKnobDiam/2 + HoleSpacingFront + HoleSpacingFront - TogSwitchWidth/2, SolYPosition + SolBodyLength - SolShelfDepth, -ZLen/2 + Thickness + RegHeight/2 - TogSwitchHeight/2]) cube(TogSwitchWidth, SolShelfDepth, TogSwitchHeight, $fn = Smoothness, true$);

// powerJackHole translate([XLen/2 - Thickness/2, PowerJackYPosition, -ZLen/2 + Thickness + MountPointHeight + USBZOffset]) cube([2*Thickness, PowerJackWidth, PowerJackHeight], true);

// miniUSB Hole translate([XLen/2 - Thickness/2, USBYPosition, -ZLen/2 + Thickness + MountPointHeight + USBZOffset + USBHeight/4]) cube([Thickness, USBWidth, USBHeight/2], true); translate([XLen/2 - Thickness/2, USBYPosition, -ZLen/2 + Thickness + MountPointHeight + USBZOffset - USBHeight/4]) cube([Thickness+1, USBWidth2, USBHeight/2], true);

// powerJackHole translate([XLen/2 - Thickness/2, PowerJackYPosition, -ZLen/2 + Thickness + MountPointHeight + USBZOffset]) cube([2*Thickness, PowerJackWidth, PowerJackHeight], true);

// Power Jack Support Horiz Hole translate([XLen/2 - Thickness/2 - PowerJackSupportDepth/2, PowerJackYPosition, -ZLen/2 + Thickness + MountPointHeight + USBZOffset - PowerJackHeight/2 - Thickness/2]) cylinder(Thickness, PowerJackSupportHoleDiam/2, PowerJackSupportHoleDiam/2, $fn = Smoothness, true$);

// Output Airhose hole translate([XLen/2 - Thickness/2, SolYPosition + AirHoseOutYOffset, -ZLen/2 + Thickness + SolShelfHeight + SolHeight/2]) rotate(90,[0,1,0]) cylinder(2*Thickness, AirHoseOutDiameter/2, AirHoseOutDiameter/2, $fn = Smoothness, true$);

//Footswitch BNC

// BNC switch hole translate([XLen/2 - Thickness/2, FootswitchYPosition, -ZLen/2 + Thickness + MountPointHeight + USBZOffset]) rotate(90, [0, 1, 0]) cylinder(Thickness + 1, 2.5, 2.5, $fn = Smoothness, true$);

// MountingPoints // Rear Right translate([XLen/2 - Thickness - BoardEdgeX, BoardYPosition + BoardEdgeY, -ZLen/2 + Thickness + MountPointHeight/2]) cylinder(MountPointHeight, MountPointInnerDiameter/2, MountPointInnerDiameter/2, $fn = Smoothness, true$);

// Rear Left translate([XLen/2 - Thickness - BoardEdgeX - MountPointWidth, BoardYPosition + BoardEdgeY, -ZLen/2 + Thickness + MountPointHeight/2]) cylinder(MountPointHeight, MountPointInnerDiameter/2, MountPointInnerDiameter/2, $fn = Smoothness, true$);

// Front Right translate([XLen/2 - Thickness - BoardEdgeX, BoardYPosition + BoardEdgeY +2 + MountPointLength, -ZLen/2 + Thickness + MountPointHeight/2]) cylinder(MountPointHeight, MountPointInnerDiameter/2, MountPointInnerDiameter/2, $fn = Smoothness, true$);

// Front Left translate([XLen/2 - Thickness - BoardEdgeX - MountPointWidth, BoardYPosition +2 + BoardEdgeY + MountPointLength, -ZLen/2 + Thickness + MountPointHeight/2]) cylinder(MountPointHeight, MountPointInnerDiameter/2, MountPointInnerDiameter/2, fn=Smoothness, true);