Automatic Espresso Machine

https://pabstaaron.github.io/AutoCoffeeMaker/

Aaron Pabst, Ben Nagel, Nathan Donaldson

***Abstract*—Making espresso is difficult. It takes several pieces of bulky equipment and months, sometimes years, of practice to do well. However, most of the difficult tasks involved in the espresso making process can be automated in a way that is predictable and consistent. There are several machines on the market that do successfully automate the process, but none of these machines make espresso in the same way that skilled baristas do and the quality of the output reflects this. By creating a machine that mimics a human barista, these automatic espresso machines can eventually be superseded.**

I. Introduction

There are many espresso machines available for commercial and consumer applications. The most frequent of which an average consumer will encounter is the manual espresso machine used by their local coffee shop. These machines require a substantial amount of training and practice to wield effectively. The operator must master the skills of grinding the coffee, tamping grounds, and physically pulling the espresso; tasks that are out of reach for the average consumer to perform on their own.

However, there are an increasing number of espresso machines on the market that automate some of this process. Most of these machines grind beans, tamp the grounds, and pull the espresso with little intervention from the user. However, the quality of these machines often trails behind that produced by a human barista.

There is not presently an elegantly implemented solution for an automated espresso machine that mimics the way a human would make espresso. This is largely due to the fact that there are certain mechanisms and control systems that would have to be created for such a device that are non-trivial to design and implement.

The remainder of this document discusses the functionality of various modern coffee machines. We then compare our implementation of a fully automated, web connected espresso machine. This discussion will include a detailed description of how the device was designed and implemented.

A. How Espresso Machines Work

There are many different types of coffee makers available. Each of these systems is unique in its own way and each has its own set of pro's and con's. All coffee makers, however, have one thing in common: they all must push hot water through ground coffee beans in some way. In the case of espresso, hot water is heated to a near boiling temperature and pressurized in some way. This hot water is then forced through densely packed coffee grounds (known as a "puck") in order to produce thick, creamy coffee [1].

There are several different ways in which the water may be pressurized. One of the most common ways this is accomplished is to simply let the water pressurize as it heats in a sealed container and turns into steam. Once a suitable temperature is reached, the container is unsealed and the pressurized water passes through the puck. This approach is used in most low-end espresso machines as it requires few mechanical components and is generally inexpensive. It also tends to produce lower quality espresso as the water pressure is difficult to regulate and drops as the brewing cycle progresses.

Higher end machines used in most commercial applications use an electric pump to force the water through the grounds, allowing for tighter control of the water pressure as well as faster brewing times.

An optional, but important, component of an espresso machine is the frothing wand. The frothing wand is a hollow shaft of aluminum that is used to direct high pressure steam into milk (or a milk-like product), the effect of which is incorporating air into the milk that makes it light and foamy (or frothy, as the name suggests). Most modern espresso machines have a built-in frothing mechanism. On lower end machines, the frothing wand connects to the same boiler that produces the brewing water. This is undesirable due to the fact that frothing water needs to be heated to much higher temperatures than brewing water in order to produce the necessary high pressure steam. Machines that only have a single boiler therefore need to introduce a long delay between the brewing and frothing cycle while the boiler switches from one task to another.

Higher-end machines will generally introduce a second boiler for producing frothing steam. This boiler will operate at a much higher temperature than the brewing boiler in order to produce the necessary steam pressure.

II. Project Implementation

The project was completed in three main parts. The design and construction of an automatic dispensing/grinding/brewing/frothing system, the creation of embedded C++ to control this mechanical system, and an Android application to provide a user interface.

A. Design Decisions

Our implementation utilizes a single boiler to heat water for the brewing and frothing process. Water is pressurized and moved though this boiler using an alternating current vibratory pump, found in many home espresso machines. These vibratory pumps provide poor output pressure regulation

and tend to have short lifespans, but are cheap and easily salvageable from inexpensive home espresso machines.

B. The Brewing Process

The brewing mechanism consists of the three primary stages, a dispensing and tamping phase, a brewing phase, and a disposing phase. The mechanism is centered around a custom designed espresso puck that is mounted to a sliding rail. During the tamping phase grounds are dispensed from a receptacle sitting above a waterwheel type mechanism controlled by a stepper motor. As the wheel spins, ground coffee is dumped onto a chute that directs the coffee to spill into the puck. After a sufficent amount of grounds have been dispensed, an actuator presses down into the puck to compress the coffee grounds. At this point, the puck is slid over via a stepper motor to sit underneath a boiler salvaged from an existing espresso machine. The boiler is heated to a defined set temprature and then a pump is activated to begin moving water through the boiler. The amount of water moved is monitored using a hall-effect based flow meter. Once a suffiecent amount of water has been dispensed, the puck is moved to sit under another actuator, which punches the grounds out of the puck and into a waste container.

C. The Frothing Process

The frothing system consists of two stages, a milk dispensing phase and a steaming phase. First, milk is pumped out of a small tube situated near the steam outlet via a peristaltic pump. As the milk is dispensed, the wand is raised at roughly the same rate that the milk is being pumped in order to get an approximate location of the top of the liquid. After the milk has been dispensed, the wand lowers itself to a position dictated by the user in order to obtain the desired froth intensity.

The machine will then begin heating the boiler until it reaches 250 degrees Celsius, at which point it will actuate a valve on the boiler that allows steam to pass through the frothing wand. Steam will be dispensed for a fixed period of 30 seconds. The machine will then reset the boiler for the next brewing cycle.

D. Mechanical Design and Implementation

All of the design work for the machine was completed in the Fusion 360 CAD package. This design was then fabricated using a combination of 3D printing and laser cutting.

Initially, the brewing mechanism was designed by drawing all components as floating objects with no housing or support. Once the mechanism was designed and simulating properly, shelves were drawn under and behind the pieces to support the components. All mechanical pieces that couldn't be readily purchased were 3D printed in PETG, which has good mechanical properties and is generally considered food safe (although fused deposition printing is not an FDA approved process). The housing/support components were then fabricated out of quarter inch acrylic using a laser cutter.

The custom mechanical components consist of a puck, tampers, a ground dispensing wheel, a chute for directing the grounds, motor mounts, and acutator mounts. The purchased componets included NEMA-17 stepper motors, linear servos (actuators), linear bearings, rails, rail mounts, and a timing belt.

Once all pieces were fabricated, the acrylic housing was assembled using five minute epoxy. All mechanical pieces were then bolted into their approprate place and the timing belt was installed and tensioned between the stepper motor and the puck.

At this point, not all of the components lined up with each other as initally designed due to the kerf of the laser cutter not being taken into account as well as assembly errors. Some mounting holes needed to be redone with a drill. Additionally, some of the pieces were altered after the acrylic housing was cut, so some mounting positions had to be manually drilled.

E. Electronics and Control

The electronics for the system were initially designed as a custom PCB that would be responsible for all low-level control tasks. This PCB would communicate over UART with a Raspberry Pi 3 for WiFi control.

Due to outstanding issues with the motor control circuitry, the system was redesigned to replace the custom PCB with an Adafruit Feather board along with accessories that provided equivalent functionality to what the original PCB was designed for. In many cases, these accessories use components identical to those used in the original PCB.

The electrical system consists for the following major blocks: stepper motor controllers, PWM servo controllers, a temperature sensor, relays, a liquid flow meter, and several homing switches.

The stepper motor controller is an Adafruit Motor Controller Featherwing. This controller uses a TB6613 H-bridge chip along with the Arduino stepper motor library to control the stepper. We additionally added a library (AccelStepper [2]) between the stepper motor library and the control chip for moving the stepper motor with acceleration and deceleration. We needed acceleration on the stepper movements for the puck in order to achieve an accurate positioning.

Mechanisms involving stepper motors generally have a certain point that is considered to be a starting, or home, position. However, the starting position of the mechanism may not necessarily be known. Therefore, the mechanism must be "homed",that is, the starting position must be found. We accomplish this task in the tamp/brew/dispose mechanism using two microswitches on either end of the device. On startup, the puck is moved slowly to the tamper side of the device until it runs into one of these homing switches. This position is noted and the motor moves to the disposer side of the machine until a switch on that end is hit, this position is also noted. It is important that we have accurate positions on both ends of the mechanism in order to ensure that the tamper and disposer are accurately positioned above the puck.

The temperature sensor is used to regulate the water temperature in the boiler during the brewing process. The sensor consists of thermocouple with an AD8494 temperature sensing

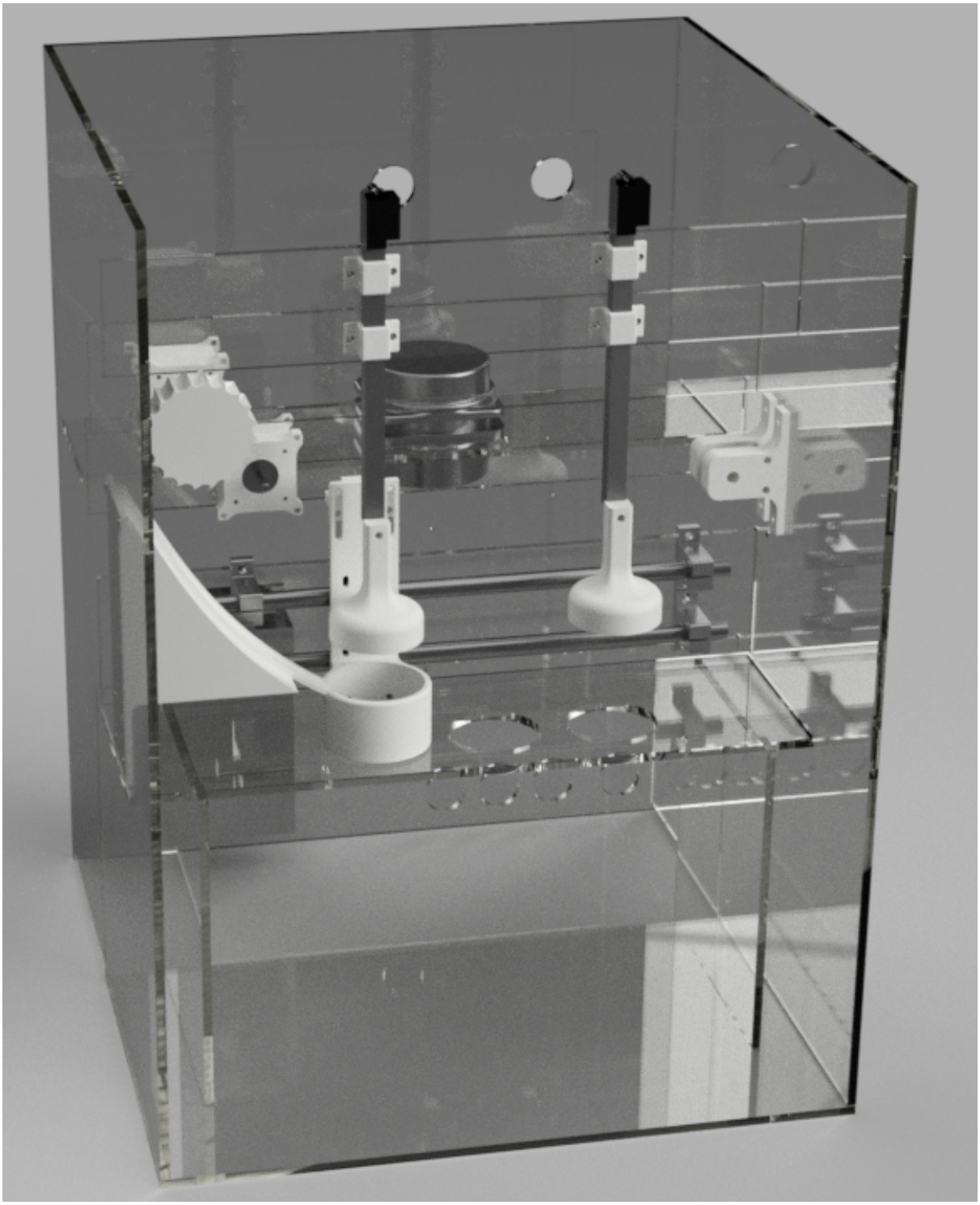

Fig. 1. A render of the phyical system design.

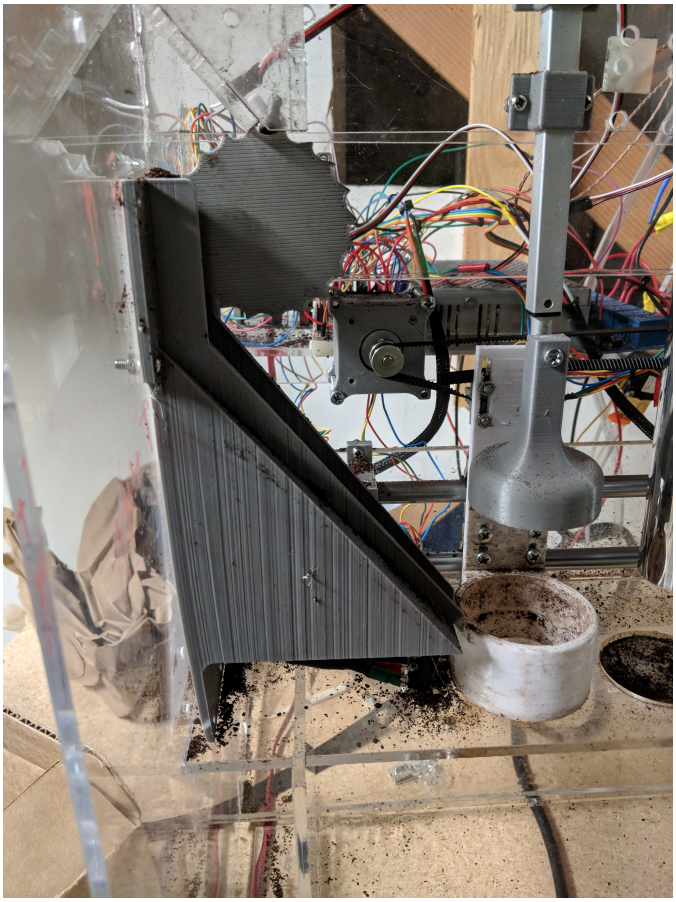

Fig. 2. An image of the ground dispenser and tamper.

amplifer. A thermocouple is a device that produces a small voltage depending on the temperature it is exposed to. This voltage is too small for most microcontrollers to detect, so it must be amplified. The AD8494 takes the thermocouple voltage and amplifies it such that the output of the device is 5mV per degree Celsius [3].

A liquid flow meter was used to monitor the amount of fluid being moved through the pump. This sensor has a pinwheel with a magnet and a hall effect sensor. As fluid flow through the sensor the pinwheel spins, causing the magnet to move over the hall effect sensor, generating a digital pulse. Each pulse equates to roughly 2.2mL of fluid. This signal was setup to trigger an interrupt in the microcontroller. Each time this interrupt is generated, a counter keeping track of how much fluid has been dispensed is incremented.

When the system is started, the machine runs through a startup check in which all motors and actuators are twitched in order to verify correct functionality. The homing procedure defined above is then performed. The machine will then move the puck back to the tamping position and wait until a brew command is received over UART. When this command is received, the machine drives the motors and actuators to perform the brewing process defined above. During the actual brewing phase of the process, relays are used to turn on and off the boiler and pump as described in the mechanical design section. When the process is complete, the puck is returned to the home position and the cycle repeats.

F. Raspberry Pi Setup and Initialization

G. Flask

Flask is a web framework that provides tools to allow you to build a web application. Flask is a micro-framework, so it requires no outside dependencies or external libraries. This means that Flask is lightweight. Flask is designed for creating web applications, with a database backend that operates through the browser. We will be using it to open a port on the local WiFi network that will act as a REST server. We will use this server to send commands through the Raspberry Pi to the microcontroller. Thus, Flask is a middleman from the android application and the components inside the coffee machine [4].

H. Mobile Application

The UI was implemented as an Android application written in Java. Upon opening the application, it will try to reconnect to the previous device it was connected to and login the most recent user, if either fields have been entered in the past. If it successfully connects to the last known device, it will take them straight to main user control screen. If the tablet could

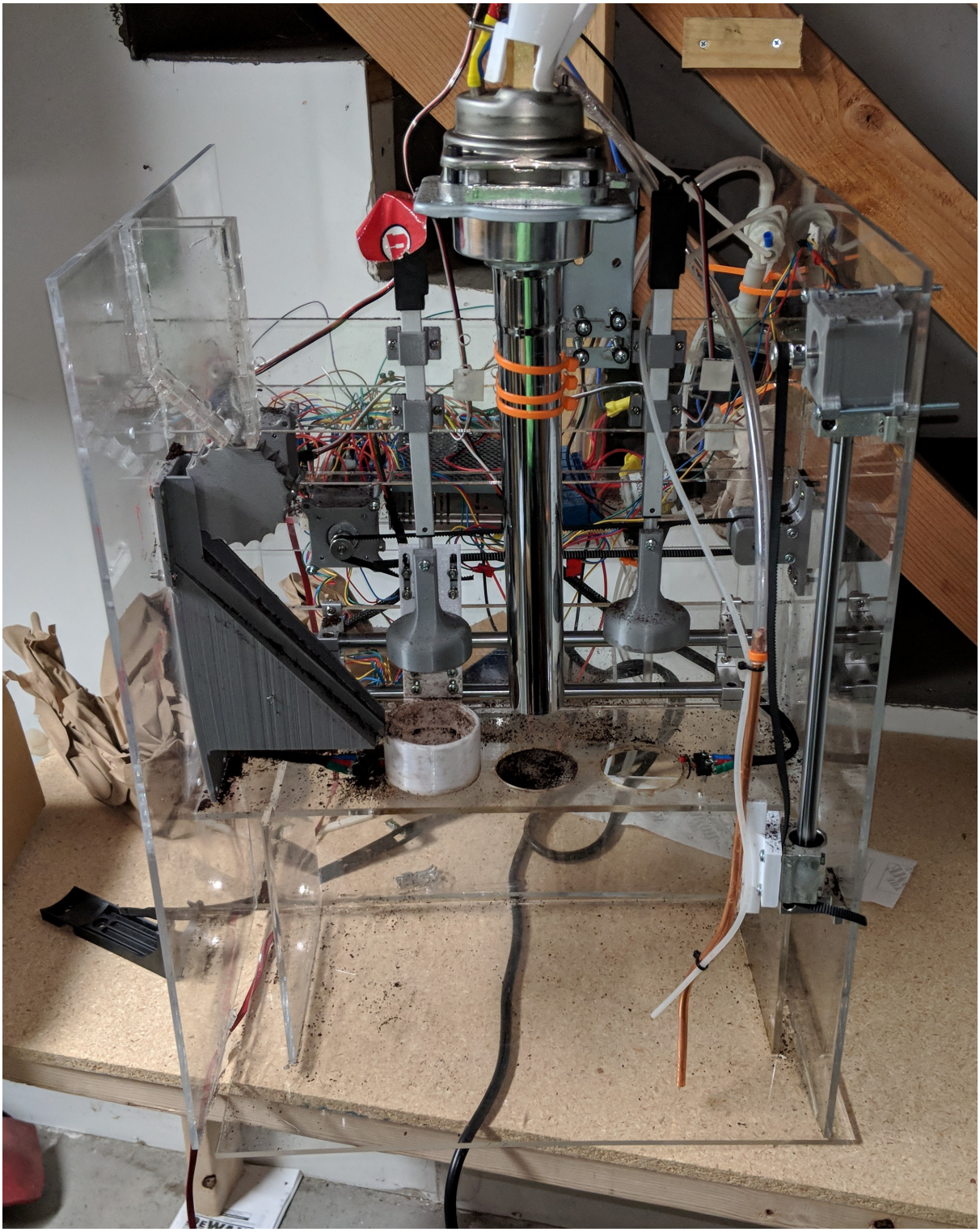

Fig. 3. An image of the assembled system.

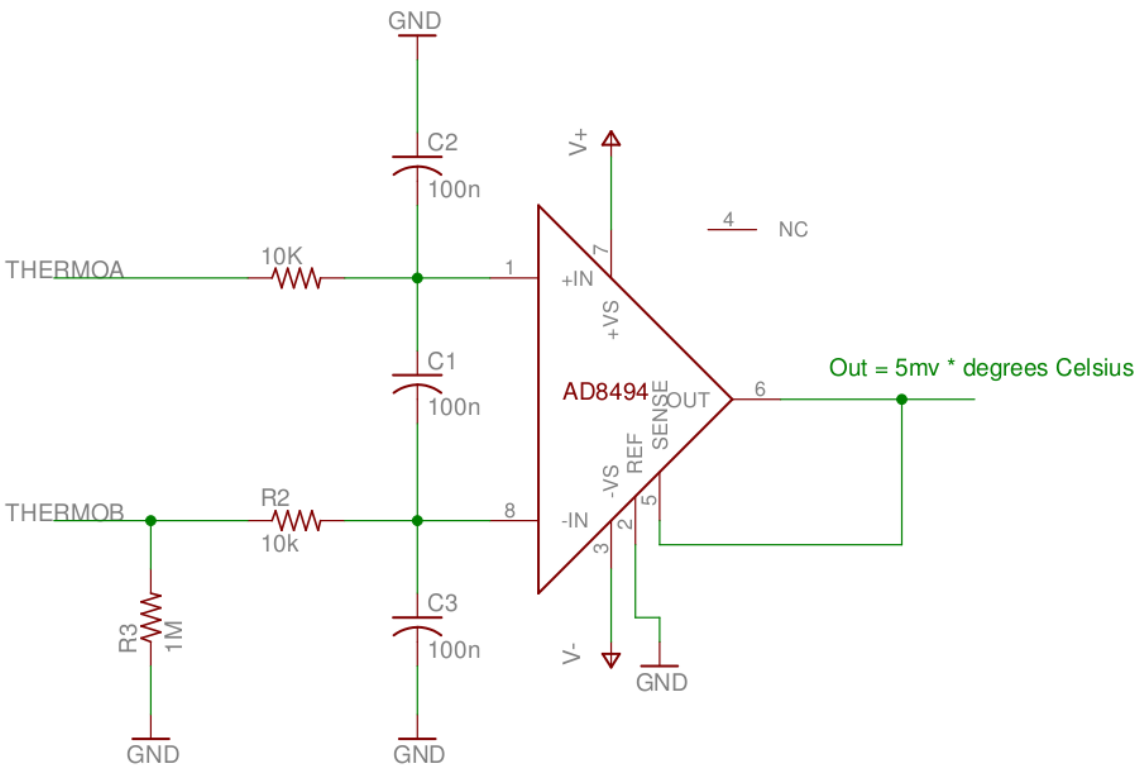

Fig. 4. Thermocouple amplifier with 100Hz low-pass filter.

not connect to the device, the user has the option to start the connection process, or decide to change the active profile.

Connecting to a device is simple. A screen for connecting to a machine opens up, it shows a list of all of the coffee machines that support the application in a recycler view. Once one is selected, the user may try to connect. If the device had previously connected properly then it will not prompt the user for a password and attempt connection, otherwise the user must input a password that is associated with the device first.

The brewing page has both basic and advanced options. The basic options are the default active options and have just a few settings for brewing the coffee: choosing how much, how strong, and how frothy. The advanced settings have much more to offer; selecting pressure, amount, and temperature for milk, water, froth, syrup, and coffee. Not all of those will have the same available selections to change, but there is much higher customization in advanced. We may be scrapping the ability to control pressure, but temperature and amount are staying. From the brewing page we are also allowed to change active users. With a user actively logged in, the option to use the favourites feature comes into play for advanced settings. This allows the user to load or delete favourites they have previously saved and also save new ones. Once all settings are where the user would like them, the brew button may be pressed and the process shall be sent to the device. If the app recognizes that it is a brew that was not loaded from favourites and a user is logged in, it will ask the user if they would like to save it as one and leave a description.

I. Communication between FLASK and App

The Raspberry PI hosts it's own web server that the user communicates with. It has a REST API that will allow us to control the board, and in turn the machine as well. Within the android application itself, we use Java.net. Within that class we are able to use things such as HttpURLConnection to send HTTP requests such as GET and POST requests. For Java HTTP GET requests we can get all the info we need in a browser URL. We can use static functions or functions with parameters within that URL. An example of a GET request would be something like:

http://localhost:8080/CoffeeMaker/login?userName=Jim

which could be a request for the user information on a user named "Jim". We could also send out information in a POST request that would be received on the local server and

then interpret the information and perform an action. There are also other actions that are useful in REST API's such as PUT, PATCH, and DELETE. Steps below are ones that would be used to send HTTP requests to the server using the HttpURLConnection class:

- Create a URL object with a GET/POST URL string like one mentioned above
- Open a connection with that URL that creates an instance of an HttpURLConnection
- Set the request method in the HttpURLConnection
- Call setRequestProperty() on HttpURLConnection
- Call getResponseCode() to see if the request was processed successfully or if there were errors
- For GET, use a Reader and InputStream and process the response
- For POST, before reading the response, get an OutputStream from HttpURLConnection and write POST parameters into it.

The Flask API exposed calls

- The first API call will be to verify that it is connected, this call will be a GET call to verify connectivity. If the call succeeds a 200 response will be posted as well as the session number, UTC timestamp, and model of the machine in a json formatted payload.
- The second API call will be to initiate a brewing cycle. This will be a POST call with machine settings embedded in the url or in a payload sent. If the call succeeds a 201 response will be posted as well as session number, utc timestamp, model and the data that was interpreted. If the machine is currently making a cup of coffee, it will respond with a 200 response that indicates that the machine is busy and the order will either be queued or discarded.

The REST API will be exposed to anyone on the network as it will run on host '0.0.0.0'. '0.0.0.0' is a non-routable address that allows all IPV4 addresses on the network to be listened to on the Raspberry Pi operating system. However, each request will not be accepted by the machine unless it has a serialized key that matches the machine data. Each machine will come with a unique serial number that must be included in each request in order for the request to be accepted. This will validate that a request is intentional and legitimate. The flask application will control and log every request made, as well as push every coffee call up to an external database, making it the only device that is pushing up to the database.

In the case that we cannot complete the project the database element will be scratched. This would affect the Flask part of the project such that the expectation of data pushing and pulling will not be implemented. If time does not permit us to complete the database section in whole, then the database loses all of its importance to this project and thus should not be attempted.

In the case that we cannot demonstrate our project at the University of Utah due to network protections, we would buy / bring a popup wifi-network that we could connect to.

1) UI Skeleton: The mobile application consists of 6 activities, 2 fragments, a connection state machine, some data classes, adapters, and customized widgets, etc. Quite a few animations, layouts, and drawables were used in the application, with the help of some 3rd party libraries as well. All device info is saved in EXTERNAL STORAGE and all user information is stored in Shared Preferences.

Most of the application is controlled by the WifiRunner which has these main states: CONNECTED, WAITING FOR USER, SEARCHING, CONNECT TO LAST, WAITING FOR RESPONSE, NO WIFI, UNKNOWN. Each activity behaves differently depending on the state of the this thread. It also houses alot of the data communication for the entire application. Any time a user does something, it sends the WifiRunner into a different state and updates the current activity with its results. When connected to a device, the WifiRunner sends out a connection check requests to the flask server on the RaspberryPi every second.

The MainMenu activity (shown in top half of Fig. 2) consists of a two buttons, an animated background, and some custom imageViews for backgrounds, logos, and buttons. A 3rd party library was used for a loading progress bar called NewtonCradleLoading. This is used while trying to connect to the previous device on this page. The "Connect Device" button on this page allows the user to continue to the DeviceSelectioon activity. The Login button opens a fragment (shown in the bottom half of Fig. 2) to login a user or opens dialogs, depending on the current active user state, and updates to the label to the current user when when is logged in. There is a logo in the top right labeled Beanster for the application and there is also a wifi indicator in the top left that displays whether or not the application is already successfully connected to a coffee machine currently. When the application starts, the WifiRunner tries to CONNECT TO LAST and disables everything until it confirms whether or not it has CONNECTED to the previous device or not. If it goes into a CONNECTED state it kicks the application into the brewing activity, otherwise it enables all of the buttons and displays a disconnected symbol and is WAITING FOR USER.

The DeviceSelection activity (shown in Fig. 3) uses most of the same components of the first activity, but introduces a recyclerView and a couple of other buttons as well. The recycler view is populated by information from the WifiRunner. The WifiRunner scans the network and looks for devices compatible with the application and then sends all of the information to the activity when it opens. The search button within this activity allows the user to refresh that information inside of the recyclerview. The back button returns to MainMenu activity and the connect button sets the WifiRunner into WAITING FOR RESPONSE to try and communicate with the flask server hosted on the Raspberry PI. Depending on the information on the device trying to be connected to, a dialog might popup, prompting the user for a password for the device. Depending

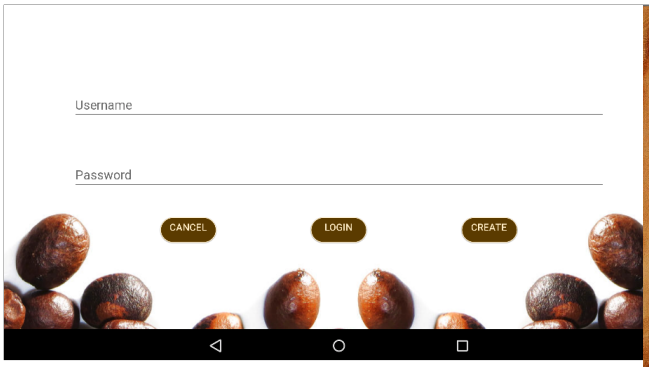

Fig. 5. Opening screen for application and login fragment

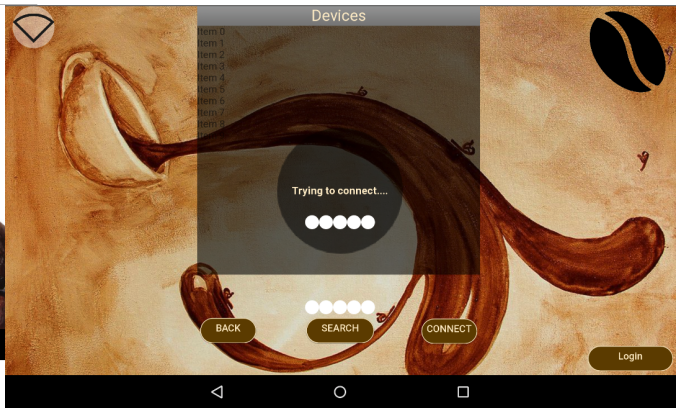

Fig. 6. Device selection page

on the response from the WifiRunner, the activity warns that it couldn't connect or continues to the brewing activity.

The CoffeeBrew activity (shown in top half of Fig. 4) uses a lot of the same widgets, but a lot more of them. This page keeps all info internally as it keeps track of the RequestData information to be sent via an HTTP POST to the Raspberry Pi. The top two buttons change the rest of the center screen and give different options. In the basic state, a layout of 3 rows of buttons appears that are given specific values for RequestData and are set when pressed. When Advanced is clicked, more tab options open with sliders available for each tab. These are used to more finely adjust the RequestData before the POST request. If a user is logged in and the Advanced button is pressed, a favourites button is available, which opens a Favourites fragment (shown in bottom of Fig. 4), allowing loading and deletion of RequestData saves with labels that are shown in a recycler view like the one previously in the DeviceSelection activity.

In the brewing activity there will be many different widgets that control options on the coffee machine. There are Spinners, Buttons, Sliders, Seekbars, etc. to control the different mechanisms within the coffee machine. The different mechanisms that will need control are temperatures of water/milk, PSI values for water/frothing, amount of milk(ounces), amount of froth(percentage), what kind of syrup and how much, the amount of coffee, and the fineness of coffee grounds. These will all be represented as integer values. Below is a mockup of the brewing activity:

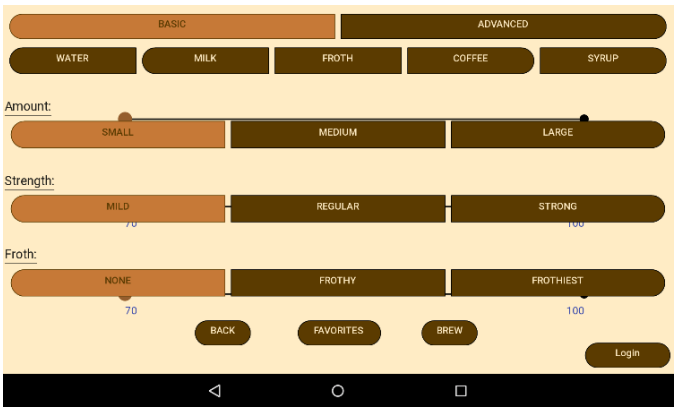

Fig. 7. Opening screen for application and login fragment

III. Evaluation

When we started this project we set out to make a perfect coffee machine that could do it all. Brew, froth, grind, dispense, and even more. We quickly found, however, that creating just an automatic tamper/brewer was months of work in and of itself. We had to start scaling back our expectations pretty quickly into the project. There were also some unexpected pitfalls throughout the course of the project. Our custom PCB not working as expected set us back by two weeks, at least. We had to scramble fast to come up with an alternate solution.

Ultimately however, the result of our work is pretty functional and something that we can really be proud of. We successfully designed and implemented a frightenly complex machine with a complicated embedded system and an Android app, and it all works toghether to actually make espresso, just like we wanted.

The tamping and brewing mechanism worked as designed after only two iterations of the 3D printed parts. We sucessfully wrote code to acuratly move and home stepper motors, and the machine continues to maintain its accuracy after repeated runs. That is, the machine doesn't get knocked out of alignement after a use or two. It will function indefinetly.

Comparing our machine to existing automatic espresso machines, our implementation falls a little flat in comparision due to the complexity and costs invloved with our machine. A standard home automatic espresso machine costs somewhere between $1000 and $2000. If our machine were to be produced and sold as is, it would likely cost twice that amount. Although we set out with a final product in mind, this machine is a proof of concept. Additional work by a balanced engineering team with substantial funding would be needed to take the machine to a point where it could be a competive product.

Our machine does, however, demonstrate that the espresso making process can be automated in a way that mimics the way in which a human would manually produce espresso.

A. Demo Day Evaluation

The night before demo day, the pump being used to push water through the boiler failed (the exact cause of the failure is unknown, but it's likely due to overuse). Because of this, we were not able to do produce espresso with the machine. We were, however, able to effectively demonstrate all of the novel mechanisms, hardware, and software that were developed for the machine. We did this by doing dry runs of the machine in which all of the steps for making espresso described above were performed when evoked by our Android application.

Those viewing the demonstration seemed to be impressed with the machine, and we got several compliments for sucessfully putting together all of the mechanisms and for the fluidity with which the whole setup ran.

The feedback we received on demo day validated the project and the months of hard work we put into our machine.

IV. Appendix A: Discoveries and Pitfalls

A. Never Start with a PCB

When we first started the project, the very first thing we did was design a large, complicated PCB that had all of the circuitry that we thought we would need.

After we got the PCB back and assembled it we discovered that while the majority of the board functioned as expected, our motor controller circuitry did not function.

Designing and building this board was about a month long process, and that time was largely wasted. After a few nights of frantically and pointlessly trying to get the board to work, we decided it would be better to switch over to using off the shelf Arduino boards and accessories.

Had we started with the Arduino setup, we would have had substantially more time to iron out any issues with the mechanical side of the project. If there was time, we could have then based a PCB on our prototype setup, which would have been much more likely to work as expected.

B. Inductive Loads

We drove our vibratory espresso pump via a relay connected to 120V mains. Ocassionally when we opened up the relay driving the pump, the machine reset. The pump and the digital side of the system were electrically isolated, but were in a fairly close physical proximity and some digital control lines ran fairly close to the pump.

It turns out that when the pump was being disconnected from power, the leftover energy stored in the pump's coil caused a large voltage spike across the pump terminals. The spike was sufficent to create a large burst of electromagnetic interference that induced a voltage on the digital side of the system that was sufficent to cause a reset in the control board.

To solve this issue, we placed a large 2mF capacitor with a bleeder resistor across the pump terminals in order to supress the inductive spiking.

C. Have a Spare of Everything

We went through the whole project having, for the most part, exact quantities of the components we needed. For most of the project this wasn't an issue. When our only brewing pump failed the night before demo day however, it became clear that we should have been keeping spares around.

We salvaged our pump from a consumer Delonghi espresso machine that we picked up in a "for parts" state on Ebay. The type of vibratory pump that we were using is known for its short lifespan and in hindsight, we should have thought to find another one.

It turns out we could have picked up an idential, never used, pump online for $50, a price that would have been well worth saving our demo.

V. Appendix B: Code Archive

- Arduino Code - https://github.com/pabstaaron/AutoCoffeeMaker/tree/master/Arduino%20Code
- Android Code - https://github.com/pabstaaron/AutoCoffeeMaker/tree/master/Beanster
- Web Page Code - https://github.com/pabstaaron/AutoCoffeeMaker/tree/master/docs
- Flask Code - https://github.com/pabstaaron/AutoCoffeeMaker/tree/master/flask_server

VI. Appendix B: Bill of Materials

- Raspberry Pi 3
- Espresso Boiler
- Vibratory Pump
- Adafruit Feather
- Adafruit Motor Control
- Copper pipe
- Silicone Tubing
- PETG Filament
- Stepper motors
- Peristaltic pumps
- Thermocouples
- AD8494 Thermocouple Amplifiers
- Acrylic
- NEMA-17 Stepper Motors
- 12mm Steel Rod
- 12mm Pillow Block
- Bearings
- 12mm Rail Mounts
- Flow Meter

References

[1] J. Smith. Wikipedia - espresso machine. [Online]. Available: https://en.wikipedia.org/wiki/Espresso_machine

[2] M. McCauley. Accelstepper. [Online]. Available: https://www.airspayce.com/mikem/arduino/AccelStepper/

[3] *Precision Thermocouple Amplifiers with Cold Junction Compensation*, Analog Devices, 4 2018, rev. D.

[4] A. Ronacher. Flask. [Online]. Available: http://flask.pocoo.org/