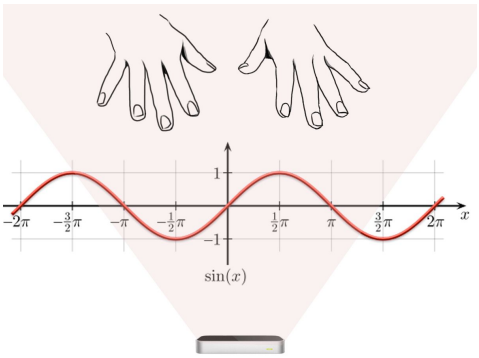

Mathemagica: Interacting with Math using Haptic Feedback

Carlos Henríquez, Erica Yuen
{carlosh, ejyuen}@mit.edu

Mathemagica is a system that allows users to feel mathematical functions using haptic feedback. By using the finger tracking capabilities of the Leap Motion, vibration motors integrated with an Arduino Micro, and a web app built with D3.js, this interface allows users to interact with the functions in virtual realm and receive tactile feedback based on finger position. The user can input a function into a textbox, hover his/her hand over a Leap Motion, visualize finger placement on the web app, and feel vibrations on the fingers when they are hovering over the graph of the function.

Users reported that the system did provide a unique and new experience with interacting with mathematical functions. However, no users were able to blindly detect general type of function based on this system from only haptic feedback they received. This poor performance was due to the fact that users have a hard time mapping out relative positions based solely on haptic feedback. In addition, the Leap Motion is extremely susceptible to noise caused by the form factor on the hand. Despite iterations of attempting to improve the form factor to reduce the noise, the performance of the Leap Motion still remains inconsistent.

Task and Motivation

With Mathemagica, users are able to interact with mathematical functions using haptic feedback based on their finger positions. Mathematical relations are often explained through their visual representation, usually 2D graphs on a page with no room for interaction. By adding multimodality to mathematical functions, users will expand their perception and understanding of mathematical functions beyond just physical lines on a page to other senses. This could potentially heighten the experience of mathematics for the visually impaired, students, children, or anyone wanting to get a better understanding of a mathematical relation by playing around with the graph in question.

System Description

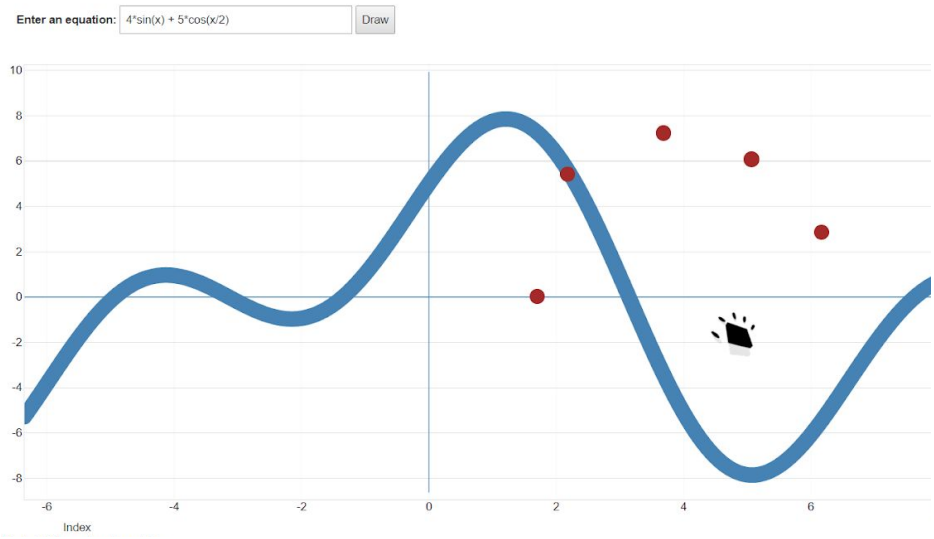

***Figure 1**: A screenshot of the Mathemagica web app. Each of the red dots corresponds to the position of a fingertip.*

Mathemagica allows a user to plot a function on a web app and receive haptic feedback by interacting with the graph using a Leap Motion, an Arduino, and some vibration motors. The user puts on the glove skeleton with vibration motors and places their hand above the Leap Motion's field of view, and an avatar and five red circles representing the user's hand and fingertips, respectively, appear on the screen. Users can then interact with the pre-loaded function (4*sin(x) + 5*cos(x/2), chosen arbitrarily), or they can type in their own function to be plotted.

The user's fingertips act as points of contact with the function. When the user's fingertips hover close to where the line is, they feel a slight vibration that gets stronger as their finger gets closer to touching the function being plotted. Figure 1 shows a screenshot of the web interface.

System Architecture

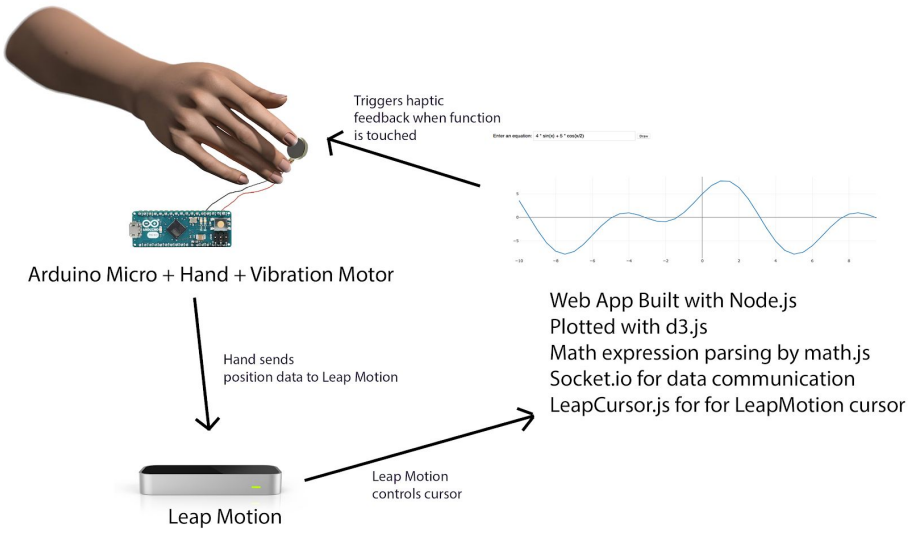

***Figure 2**: System architecture diagram demonstrating the communication loop. The Leap Motion generates position data for the hand, which is used to plot the cursor on the web app, which then sends signals to the Arduino to vibrate the appropriate motors.*

There are three main components to Mathemagica: the hardware, the client-side JavaScript, and the server-side JavaScript. Figure 2 shows the high-level architecture diagram of the system.

Hardware

The main input for the system is the Leap Motion[1], and its main output is the vibrations from the four vibration motors at the user's fingertips. The motors are small commodity vibration motors available from Amazon[2]. An Arduino Micro[3] is used to interface with the vibration motors. The Arduino is flashed with the Standard Firmata[4] library, which allows the Arduino to communicate with the Johnny-Five[5] JavaScript framework.

Although the Arduino Micro has several pins that can be used for output, only pins 3, 5, 6, 9, 10, 11, 13 are available for pulse-width modulation (PWM) output. PWM is used to provide the user with a vibration that increases in intensity the closer their fingertip is to the graph of the function. The Leap Motion and Arduino Micro are both connected via USB cables to a laptop, though the Leap Motion performs best when it is connected via a USB 3.0 connection. Figure 3 shows the pinouts for the Arduino.

---

[1] https://www.leapmotion.com/
[2] https://www.amazon.com/0-05A-10x2-7mm-Mobile-Phone-Vibration/dp/B00PZYMCT8/
[3] https://store.arduino.cc/usa/arduino-micro
[4] https://github.com/firmata/arduino
[5] http://johnny-five.io/

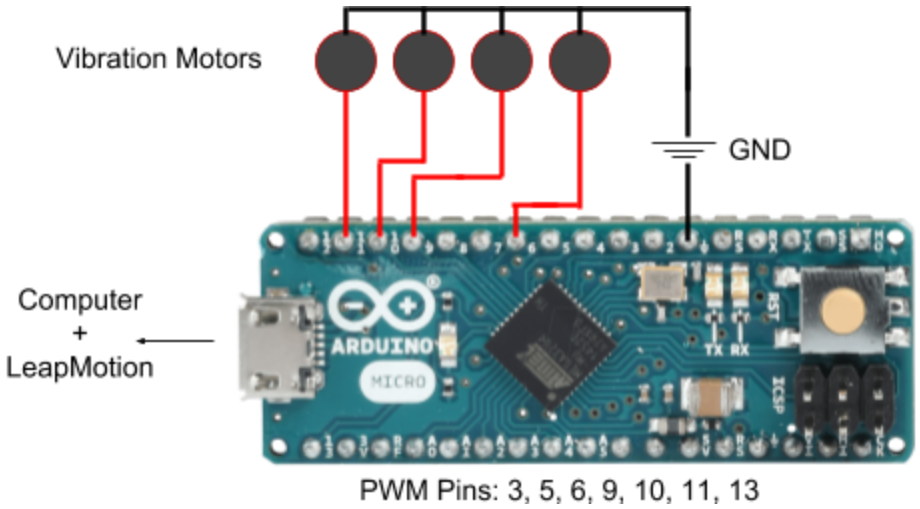

***Figure 3**: The pinouts for the Arduino Micro and the vibration motors. The red lines represent positive connections, and the black line represent connections to ground. The positive motors are connected to pins 11, 10, 9, and 6, which correspond to the pinky, index, middle, and ring fingers, respectively. Pin 11 was originally used for the thumb but user testing found the pinky to be more comfortable to use than the thumb for interacting with the function.*

Server-side JavaScript

The server runs a Node.js[6]-based web app that communicates with the client using Sockets[7]. When the server receives a message from the client, it sends a PWM signal (an integer from 0-255) to the Arduino that corresponds to how strongly to vibrate the appropriate motors. To aid in debugging, the server includes a mode that accepts the messages from the client but that does not connect to the Arduino. The port that the server uses to connect to the Arduino varies across operating systems, and one must make sure that the server can read and

[6] https://nodejs.org/
[7] https://socket.io

write to the ports. The server is run with Node Version 8.0.0 with NPM 5.0.0. We used the Node Version Manager[8] to control which version of Node was used to run the server.

Client-side JavaScript

The client interface is a simple web app that plots a mathematical function, takes hand position data from the Leap Motion, and displays a representation of the user's hand on the graph. The Leap Motion input is processed with LeapCursor.js[9] which creates a virtual hand for the user. Because the user can use multiple fingers to interact with the graph, we use jQuery and other JavaScript code to create five circles representing the user's fingertips to act as cursors and fire custom events, which then send messages to the server using Sockets.

Performance Results

The system is able to successfully take the input from the Leap Motion and cause the motors on the glove to vibrate when the user touches the line on the graph. However, identifying a function based solely on the vibrations felt proved too difficult; none of our users during our user testing were able to identify anything more complicated than $y = x^2$. Although using more than one finger for tracking generally increased the level of interaction with the system, it made it harder for users to reason about the function question as opposed to using a single finger.

The Leap Motion's hand tracking was also not as precise as we had expected and struggled at a few edge cases. For example, the Leap Motion sometimes struggles to properly recognize the position of the fingers when the glove is worn by users, likely due to the black electrical tape used to anchor the glove to the user's hand and the wires extending from the

[8] https://github.com/creationix/nvm
[9] https://github.com/roboleary/LeapCursor.js

fingers to the back of the hand. Further, the Leap Motion spazzes out and reports erratic positions when the hand is tilted sideways at certain angles, likely due to the occlusion of parts of the hand making it difficult for the Leap Motion to identify the hand and estimate its position.

Another interesting failure of the system is that the Leap Motion sometimes thinks that faces are hands, which in turn interferes with accurately tracking the actual hand the system should be using. We were unable to determine the cause of this issue, but worked around it by trying to keep our faces as far as comfortably possible from the Leap Motion's field of view.

All in all, we learned that while the Leap Motion is decent at capturing general hand position data at a fast enough rate, the granularity with which it reports this data leaves something to be desired. Further, the Leap Motion does not make a great mouse; hands floating in the air generally do not remain in a stable position and even the smallest of twitches will be interpreted as a mouse move, contrary to the stationary mice users are accustomed to when using computers.

Challenges

On the hardware side, actually building the glove ended up being harder than expected. What is conceptually a simple circuit came with complications like cold solder joints, motor wires being hard to strip without cutting, and creating a comfortable right form factor. In addition, we did not realize how much noise the form factor would add to the Leap Motion. Our initial understanding was that the Leap Motion's cameras were infrared, so we did not think that the material obscuring the hand partially would cause the magnitude of problems that we had. When testing our system, it was very difficult to see if bugs were from the hardware side, the

software side, or both. We had to iterate on the hardware and form factor designs several times to make sure it would cause as little noise to the Leap Motion as possible.

Plotting Libraries

On the software side, getting the right plotting library that could properly handle the custom events we needed to fire took a lot of time. At first, we started out using Plot.ly[10] as our plotting library, which worked fine in terms of plotting arbitrary graphs. From there, we used Math.js[11] to convert simple text input (like “x^2”) into an object that could take in values and evaluate the function at those values. When it came time to creating a representation for a cursor, such as that for a single finger, we began running into some issues. LeapCursor.js worked great in that it was open-source and well-documented such that we could extend it for our purposes. LeapCursor.js fires `'mouseenter'` and `'mouseleave'` events whenever it receives a frame from the Leap Motion, which most HTML elements could listen for.

However, when we wanted to be a bit more specific as to which element of the graph would listen for the events (e.g. the line being plotted and not the entire canvas), we discovered that Plot.ly did not support many mouse events, let alone custom ones. We then stumbled upon C3.js[12], which is built on D3.js[13], but after several hours of trying to understand poorly-documented code discovered that it too did not support listeners for custom events. Trying to use D3.js by itself entailed having to manually handle a lot of common functionality, like panning or zooming, but luckily we discovered Function Plot[14], a plugin for D3 that abstracted away the logic for creating a plot and also supported custom event listeners.

---

[10] https://plot.ly/
[11] http://mathjs.org/
[12] http://c3js.org/
[13] https://d3js.org
[14] https://mauriciopoppe.github.io/function-plot/

Scalable Vector Graphics

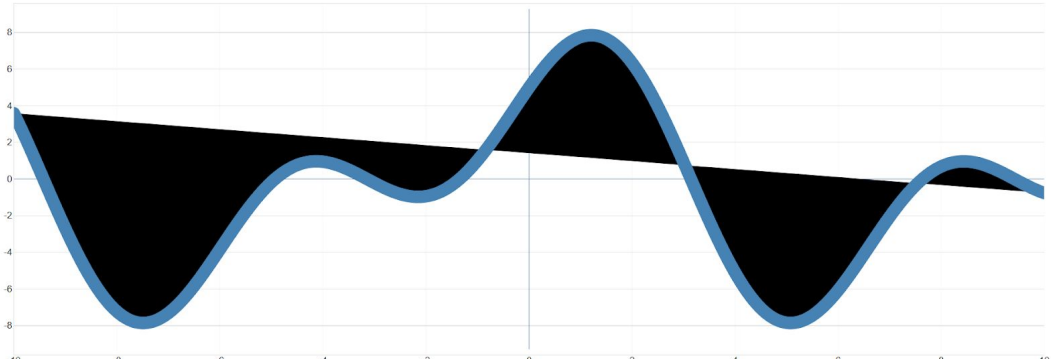

***Figure 4**: The SVG fill bug. SVGs implicitly connect the last point to the first point of the data, which causes the filled region (black) to count as an interaction area. Setting the CSS properties `'fill: none'` and `'pointer-events: stroke'` both hide the filled area and make the interaction area for the SVG only the drawn line (blue).*

The root of the issue with the plotting libraries was our lack of knowledge of how Scalable Vector Graphics (SVGs) worked. We ran into both interaction and cosmetic issues with SVGs. The first interaction issue was being unable to have just the line for the graph that was plotted listen to custom mouse events without having the entire canvas also listen for the same events. This was resolved when we switched to using Function Plot, though with it came more issues.

The first was that, when creating SVGs, after all the points and lines are drawn, the SVG implicitly also draws one last line connecting the first point with the last point such that what was previously an open polyline became a closed polygon. Although this last line was not drawn, it was still present in the underlying object which determined the points of interaction for the SVG. This was troublesome because the area of interaction now included the space the polygon enclosed and not just the line that we were plotting, as we had intended. Figure 4 demonstrates

what the underlying SVG looked like when we removed some of the CSS that was hiding the fill. It took several days of Googling and reading SVG and CSS specifications to discover that there is a CSS attribute called `'pointer-events'` for SVGs that determines which parts of the SVG can listen for pointer events. Using D3.js to programmatically set this property to `'stroke'` allowed us to attach custom event listeners to just the line on the graph without interacting with the invisible filled area.

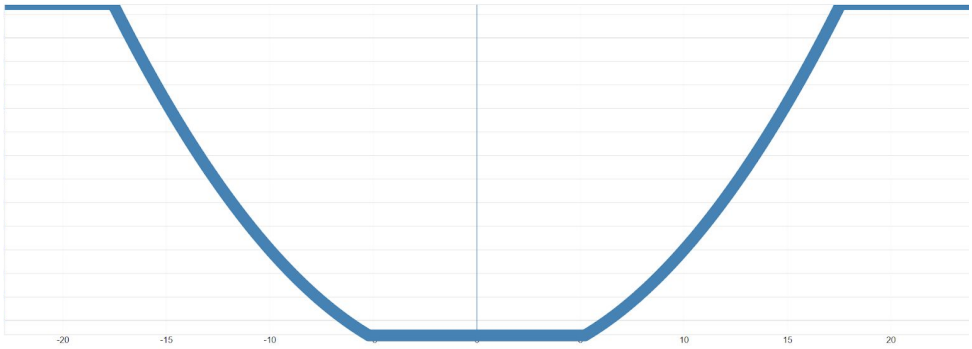

***Figure 5**: The Function Plot taper bug for the function y=x^2. Without an SVG `clip` path, horizontal lines appear at places where the graph would otherwise go off the canvas.*

The second issue was more of a slight cosmetic one that arose from using Function Plot. Because of the way the library dynamically generates the SVG as a user pans and zooms the canvas, there are parts of the function being plotted that would naturally extend beyond the canvas. Because of the way Function Plot draws the line for the function, the parts that should have been cut off were instead drawn as horizontal lines that would expand either to the edge of the canvas or to another point on the SVG. Figure 5 demonstrates what the SVG drawn looked like before adding an SVG `clip` path to hide the parts of the graph that should have been tapered. This solution also involved much time dedicated to reading the SVG documentation.

Interacting with D3.js using Custom Mouse Events

One last challenge that we had to work around was getting our cursor events to fire properly. Unfortunately, unlike with the Battleship project where we had used the Famous Engine[15] to create our own canvas on which we then drew a grid and the corresponding ships, because we were attempting to use more than one mouse and because the object we were interacting with (the D3.js plot) was external to the Famous library, we had to forego using much of the Leap Motion code that we had used for the Battleship project. It took us a while to understand how to properly fire custom DOM events and attach the appropriate listeners to them, but thankfully LeapCursor.js had a way of creating such events that we could extend.

Creating our own events piggybacked well with Socket.io in that it allowed us to create a separate set of `'mouseenter'` and `'mouseleave'` events for each finger (e.g. `'indexenter'` and `'indexleave'`). The listeners for each of these events would send distinct messages to the server over the websocket, which we could then route to the appropriate pin on the Arduino to make the corresponding motor vibrate. Using Socket.io was very simple and allowed us to expand to using multiple fingers.

Project Roadblocks and Pivots

We started out with very lofty and ambitious goals from the project, including using wireless hardware, interacting with 3D virtual surfaces, and using electrical muscle stimulation (EMS) as a form of output. As the term progressed and as we encountered challenges, both in

[15] http://staging.famous.org/

the form of software roadblocks and hardware difficulties, we had to aggressively downscope the project in order to make it feasible within the term.

We started out with a Python backend[16] that was the original proof-of-concept for connecting the Leap Motion with an Arduino, but we then pivoted to using Node.js as it would give us access to libraries like D3.js and Socket.io while keeping both the backend and frontend in the same language. We had initially also wanted to use the WolframAlpha/Mathematica API to process natural language queries for functions, but after learning about how limited the API was we were forced to rely on typing out a function for input.

Although the research scientist in the Media Lab we initially reached out to had a tutorial on how to build the EMS system we had planned to use, we had assumed that we would be able to use the one he had already built. It turns out that he had already left the country and that building our own EMS system would introduce a lot more hardware complexity that we did not want to deal with until we had a working MVP.

As we encountered limitations with how faithfully the Leap Motion tracked the fingertips and how difficult it was to get multiple cursors set up, we aimed for getting the 2D case working before we attempted 3D. We initially also wanted to include voice commands to add an extra modality with which to operate the system, but because we struggled to get an MVP working with just simple graphing libraries, we decided to cut that feature as well. It was until after the Implementation Studio that we were able to resolve many of the graph interaction issues alluded to in the Challenges section.

[16] https://github.com/shensleymit/HapticGlove/tree/master/Haptic%20Feedback%20Code

***Figure 6**: Initial prototype of the haptic feedback glove (with LEDs instead of motors). Users generally found the breadboard to be a little heavy and the glove uncomfortable.*

User Study Design and Results

For our user study, we helped users put on our haptic feedback glove, had them try out interacting with the graph of `'y = x'` to familiarize themselves with how the glove felt, and then had them try to guess what a certain set of functions were without looking. This task was significantly harder than we had expected, with none of our users being able to identify any function more complex than $y = x^2$. We also had our users try different configurations for the motors, specifically using one motor vs multiple motors, and having a binary on/off signal vs a signal that scales with how far away the user's fingers are from the plotted function. Users liked being able to use multiple motors, but found it harder to identify functions blindly when using multiple motors. One user said that she "felt a bunch of vibrations on one side, then felt a bunch of vibrations on another side, then [she] tried to figure out what was between the two but had no idea." Our users unanimously liked having a variable vibration that scaled inversely with distance as opposed to a binary one. One user stated that "it was hard to know like how close

[he] was, especially if [he] moved [his] hand quickly when it was just on and off, but like with the scaled vibrations [he] could get a better sense of where [he] was relative to the parabola".

We received a lot of feedback on the form factor of our glove prototype. Initially, we had a fabric glove with the Arduino harnessed to the back of the hand. Figure 6 shows the initial prototype of the glove (with LEDs instead of motors, initially used for debugging), which users found to be more uncomfortable than we had anticipated. Specifically, users commented on the glove's weight feeling unevenly distributed and on the fabric being a bit too tight for the comfort of some of our users. Users also found using their thumbs to be more uncomfortable than using their other four fingers. To our surprise, our users did not have an issue with the system being wired to the computer.

Given this feedback, we significantly changed our glove a more lightweight and adjustable design. Figure 7 shows the new design of the glove, which uses adjustable straps made from electrical tape to accommodate different hand and finger sizes. We also wired the motors such that users can interact with their four fingers and not their thumb, which users found to be much more comfortable.

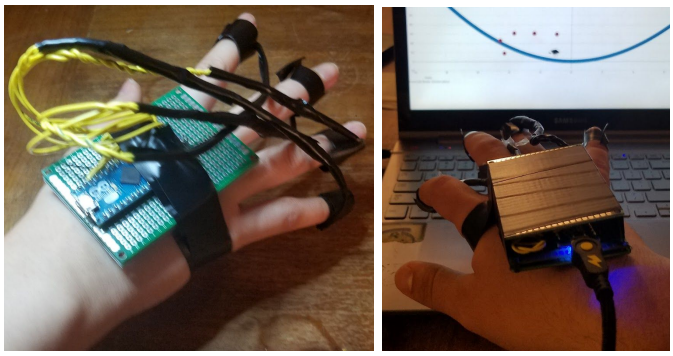

Figure 7*: Versions 2 (left) and 3 (right) of the haptic feedback "glove". The fabric glove was removed entirely and replaced with adjustable straps made from electrical tape. The Leap Motion sometimes struggled when it saw extraneous wires, so for version 3 we made the glove even more compact.*

What We Learned

We learned a lot of both technical and non-technical lessons from working on Mathemagica. The first, in terms of project management, is to not be afraid to downscope aggressively, especially in the early stages of a project. It it important to have a vision and stretch goals, but it is equally important to have more manageable and attainable goals that allow the project to progress forward, be tested by users, and be iterated upon. This iteration is important before huge development efforts are made on a feature that users might not receive as well as one had originally hoped.

A wise professor once told us to take everything we see, especially from product demos, with a large grain of salt. The Leap Motion showed a lot of promise in its hand tracking, but when we began to deviate from the happy path of the product we started to discover more and more corner cases and realize the limitations of the system. It is crucial to be wary of what these corner cases may be and realize that with almost any product your mileage may vary.

Dealing with undocumented or not-well-maintained libraries involves a lot more detective work for what sometimes is just a single line of code. We have a newfound appreciation for good documentation as a result of several hours of browsing thin documentation and StackOverflow posts just to find that our issues could be resolved in one line of code.

From the technical side, neither of us had much experience with how SVGs were created in a browser and with how D3.js worked, so it was great getting to develop those skills. Further, learning about custom DOM events was really interesting, especially when paired with the flexibility of Socket.io for passing arbitrary messages between a client and a server as a result of these custom events. Lastly, the Johnny-five framework is something we will turn to first

when considering other projects that integrate a web app with an Arduino or other Internet of Things device.

Conclusion and Next Steps

A relatively straightforward next step for improving the performance of the system would be to include the use of voice commands like `'zoom in'`, `'origin'` or `'pan left'` using the speech API from the Battleship project. Alternatively, gestures from the Leap Motion could be used for the same functionality, for example, drawing a circle to re-center the graph back the origin, using a closed fist to "grab" the canvas and pan around, or using a grab plus a clockwise or counterclockwise rotation to zoom in and out, respectively. Alternatively, our system currently only handles one hand, so while one is used for direct interaction with the plotted function, the other hand could be used for the navigation controls.

We initially envisioned Mathemagica being used for interacting with 3D surfaces, but we recognize that a web app might not be the best medium for that. Future work would include creating a Unity app where a user could plot a surface on a 3D grid and interact with the surface in that space. Unity is one of the standards for creating 3D applications, so we hypothesize that 3D interaction in a Unity environment would be more natural than 3D interaction in a browser.

One the hardware side, the first improvement would be to use a Wi-Fi or Bluetooth enabled microcontroller to make the haptic feedback glove wireless. This would also improve the form factor of the glove. Other forms of providing haptic feedback could also be explored, such as using actuators or small servos to move the user's fingers.

Libraries and Tools

The following is a list of all of the libraries and tools used in the creation of Mathemagica. They have been referenced throughout the report with the appropriate links as footnotes, but are duplicated here for the sake of completeness:

- Leap Motion[17] - Our main source of input that collected hand position. While it was good at collecting general hand position data, there were certain edge cases, such as using having visible wires or tilting the hand to much that would cause it to spaz out.
- Arduino Micro[18] - Main source of output and integration with the vibration motors. When flashed with the Standard Firmata, worked great as an interface for the hardware. There are other IoT devices that could have been used, but this was the most easily available for us. Because of its limitations, it must remain connected to the host laptop, whereas other microcontrollers could connect to the host laptop via Bluetooth or WiFi.
- Vibration Coin Motors[19] - Main source of output and feedback for the user. The coin motors we used had extremely thin wires that made stripping the insulation and trying to solder jumper cables a huge pain. The vibration they provided was nice to the touch, and weka vibrations could be differentiated well enough from the strongest vibrations.
- LeapCursor.js[20] - The main controller for the web app and without which we would have struggled a lot more. Draws the avatars representing the hand and fingers. This library was very well documented and easily extensible for multiple cursors (in our case, multiple fingers).

[17] https://www.leapmotion.com/
[18] https://store.arduino.cc/usa/arduino-micro
[19] https://www.amazon.com/12000RPM-Mobile-Phone-Vibrating-Vibration/dp/B00GN67918
[20] https://github.com/roboleary/LeapCursor.js

- Math.js[21] - Used for turning strings like `'x^2'` into functions that could be evaluated. There were some issues when paired with Function Plot, such as plotting constant values like `'y = 5'`, but otherwise worked well enough. Provided us with what we wished WolframAlpha could have provided us.
- Function Plot[22] and D3.js[23] - The main libraries used for plotting. Function Plot is built on top of D3.js and provided us with enough freedom to add event listeners to specific parts of the drawn SVG that the other libraries we tried did not allow us to do. It also abstracted away much of the plot-drawing logic, such as zooming, panning, and drawing the axes.
- Johnny-five.io[24] and Standard Firmata[25] - The JavaScript framework used to communicate with the Arduino and the firmware loaded onto the Arduino, respectively. Johnny-five has support for several microcontrollers, so extending Mathemagica to use wireless hardware would still be possible using Johnny-five.
- Socket.io[26] - The main method of communication between the server and the client. Socket.io is a simple yet incredibly powerful framework that allowed us to send arbitrary messages between the client web app and the server that interfaced with the hardware.
- Node.js[27] and NVM[28] - The main JavaScript framework for the server. The Node Version Manager (NVM) was crucial in making sure that we ran the project with the right version of Node given that Johnny-five needed to be compiled using Node 8.0.0, otherwise it would throw an error.

---

[21] http://mathjs.org/
[22] https://mauriciopoppe.github.io/function-plot/
[23] https://d3js.org/
[24] http://johnny-five.io/
[25] https://github.com/firmata/arduino
[26] https://socket.io/
[27] https://nodejs.org/
[28] https://github.com/creationix/nvm

Contribution Breakdown and Thanks

Erica - Hardware design and integration, user study logistics, initial setup of web app

Carlos - Integration with d3.js, software debugging, integration with socket.io, PWM, creating multiple cursors for LeapCursor.js

Special thanks to 6.835 staff for the resources and support throughout this project.