ENEE408I-0101 Fall 2019 Group 2 Final Report

Names: Huy Do, Nathan Koenigsmark, Nikhil Uplekar

Professor: Gilmer Blankenship

Teaching Assistant: Xiaomin Lin

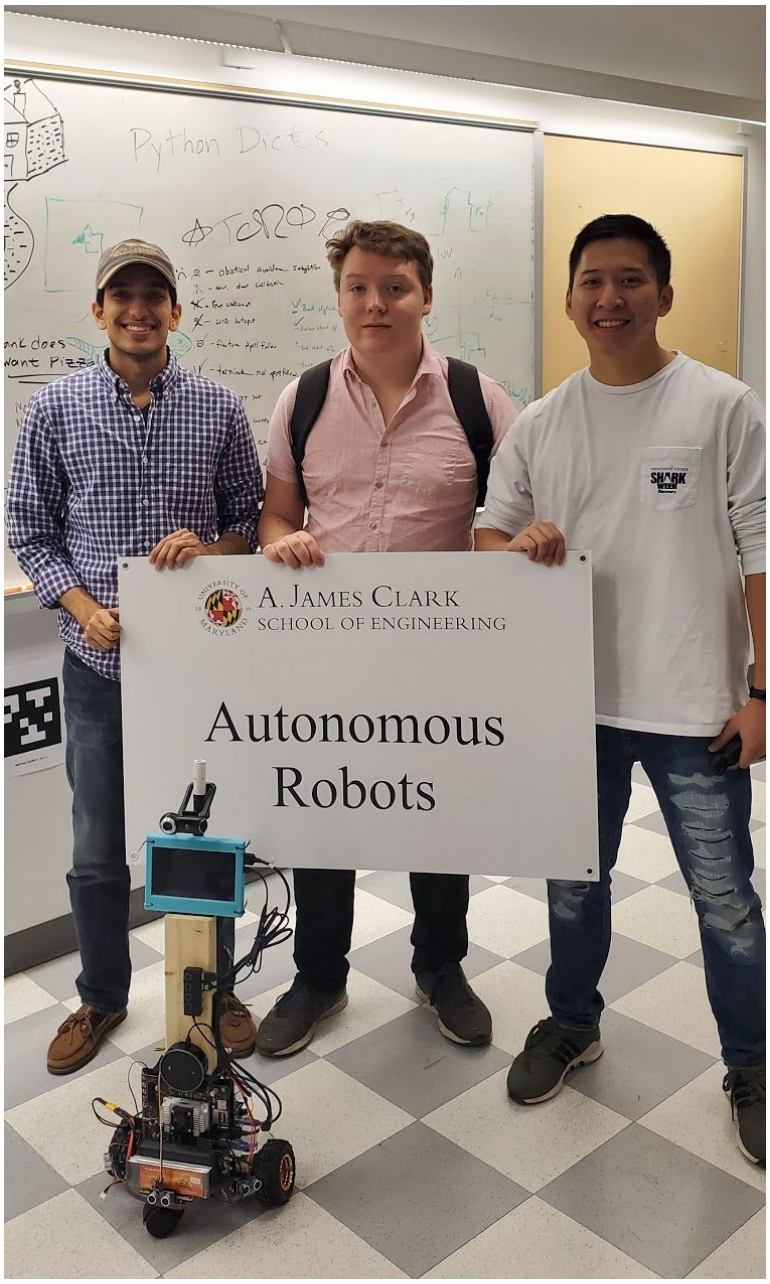

Table of Contents

1 Introduction

The purpose of this class is focused on the design and development of autonomous robotic features for a “Personal Companion Robot” - an autonomous mobile robot for person monitoring and interaction. The focus for this semester was the development of companion robots with enhanced communications (establishing a network of the 4 robots in the class and having them send messages to each other) and swarming capabilities (developing robots with the capability to meet at the location that one of the robots is sending a distress signal from—using some form of indoor location and mapping capability). As advised by Dr. Blankenship, we decided that our goal for the meetup task would be for the robots to navigate out of the lab room to a location in the hallway where the distress signal is originating from. For indoor location tracking, the class adopted an approach utilizing April tags assigned to known coordinates in the room to conduct pose estimation and create traveling/navigation paths.

Our group was happy with our final results—we were able to complete the tasks proposed for this semester along with additional features. We can communicate with our robot via the Alexa for various communications, it has the capability of monitoring a participants heartbeat via an external sensor connected to the robot via Bluetooth, it can communicate with other robots and receive distress signals via a chat server that we’ve generated, it can follow a participant (represented by a green folder), and it can quickly navigate to a distress signal location that is occurring in the hallway outside of the lab room. We’ve also included an onboard screen that generates eyeballs to follow a user (the folder) while it is being tracked. In addition, we’ve integrated a Raspberry Pi in our design to act as an access point for the Alexa because the Edu-roam wi-fi won’t let us connect Alexa to the Internet (this has been a persistent problem throughout the semester). We use the internet connection from the Jetson and bridge it to the Raspberry Pi, and then the Raspberry Pi distributes the network to Alexa. Lastly, we have shared our complete codebase with the other groups in the class to assist them in the process of developing their robot and to enable uniformity in action.

Overall, our group is proud of the final results of our robot, and we hope our code and lessons learned will be useful for future classes.

2 Primary Materials

- Jetson TX2
- Alexa Echo Dot
- USB webcam
- Arduino UNO
- Breadboard
- Motor drive Pololu MD03a
- 2 x Motors
- 3 x Ultrasound sensors
- 7 inch Raspberry Pi screen
- 12v Rechargeable Battery
- Jumper wires
- HDMI cable
- Micro USB cables
- Chassis frame
- 1 x Wood block

3 Code and Video Link(s)

We have uploaded all of our source code to a GitHub repository (we also used GitHub for version control and to simplify the development process while we were working on the robot). You can access the repository via the following link: https://github.com/nkoenigs/zGroup2.

Each file contains different major dependencies required for the system's operation. The following is a brief breakdown of the purpose of each core file:

- weaverv2.py - amazon json endpoint, and thread/processes integration
- openCVController.py - visual processing and serial outputs to the Arduino
- heartbeatController.py - bluetooth connectivity and API integration for the MiBand
- auth.py - authentication and connectivity for MiBand
- aprilTags.py - mapping and navigation instructions for arbitrary travel
- constants.py - useful constant for other files to reference
- chatController.py - client side processes for web based inter-robot communication

We have also posted the following two YouTube videos

1. **Title:** ENEE408I Robotic Capstone - Fall 2019 - Group 2
   Link: https://youtu.be/MfOe8kyghpU
   Description: This video is a collection of our team's progress during the semester. Our first out-of-lab meeting in the View apartment (which lasted for the majority of the day) really propelled us forward in the development and integration process. We also stayed in the lab before the Thanksgiving break from 11 AM to 9 PM to finish up our code integration/pipeline process and start working on our robot's navigation/distress-handling. Finally, the night before the final presentation, we were at the lab from approximately 2 PM to 3 AM in order to complete our robot's entire feature set (particularly our navigation). Snippets of our work on these days is present in the video. The last part of the video is our final demonstration of our robot's capabilities.

2. **Title:** ENEE408I Robotic Capstone - The Final Presentation Day - Fall 2019
   Link: https://youtu.be/UvEwEtQLJJs
   Description: This video covers demonstrations of the whole class on the final presentation day.

4 Robot Functionalities

The following is a list of our robot's demonstration features:

- Navigate to the location of a distress signal via pre-placed april tags
- Send out a distress signal and location via a publicly available chat server
- Take a photo of the robots user
- Connect to a MiBand3 (worn by a group member) to measure the user's heart rate
- Color based user tracking—with eyeball graphics for user feedback
- Alexa based voice commands for all aforementioned features
- Always available halting of current tasks
- Always available terminations of current tasks and future tasks

5 Code Descriptions/Breakdown

5.1 Arduino Code

The core purpose of the Arduino code is to listen for serial commands passed down from the Jetson TX2 and respond to them accordingly (the Arduino is connected to the motor controller and is responsible for movement). At the same time, we want to utilize ping/ultrasonic sensors to detect obstacles and respond accordingly to prevent collisions. In order to combine these two purposes and to ensure that they function together seamlessly (respond to commands from the jetson while having to ignore those commands and act independently to get around obstacles when we are presented with them) we utilized FreeRTOS (a multi-tasking Arduino IDE that serves as an operating system for real-time applications which allows the system to do multiple tasks at the same time). With this, we could have our ultrasound sensors checking for obstacles while also checking and responding to the serial commands that we established.

Ultimately, the ping sensors were not used for our final presentation. Obstacle avoidance was a capability we possessed, but integrating it to work seamlessly with our ability to navigate to the distress signal and follow a person around the room was something we could have done but ultimately didn't have the right amount of time to complete.

All of the Arduino code we created and used for this project is in the github linked above (under the Arduino subfolder). We've added detailed comments to our Arduino codebase, so descriptions of the purpose of each block of code can be found there. In summary, we used FreeRTOS to break our system into three tasks: updating the obstacle avoidance flags using the ping sensors, updating our orders from the Jetson by checking the serial port, and operating on the serial port commands and the current obstacle observations to decide how to drive the motors.

To simplify the commands coming from the Jetson, we opted for single characters (with a size of 1 byte) to represent different commands, as follows:

Character (char)	Command
f	Forward
l	Left
r	Right
b	Backwards
h	Halt
o	Obstacle Avoidance ON
z	Obstacle Avoidance OFF
p	Person Following
a	April <Tag> Following

We made an effort to lower the frequency of commands/bytes sent on the Jetson's end to reduce serial traffic (this is covered in the OpenCV section/code) and we also have code in the Arduino codebase so that we only act on a command if we aren't already running the same operation.

5.2 Heartbeat Integration

Heart rate data (beats per minute) is collected via an activity tracker MiBand 3. The MiBand 3 uses Bluetooth Low Energy (LE) to send and receive data to the associated phone application. Utilizing this capability, we can use the Jetson TX2 as a host to conduct communication in Bluetooth LE and send/capture data from the band/tracker.

We accomplished this with the help of the BluePy library for Python and a MiBand 3 demo library from user "yogeshojha" which can be found via the following GitHub link: https://github.com/yogeshojha/MiBand3.

For our purposes, we only desired real-time heart rate monitor functionality, so we

extracted that specific function from the library, modularized it into a class, and integrated it into our heartbeat controller code file.

Our heartbeat controller runs as a thread and tries to connect to the MiBand 3 on Jetson boot-up. After a successful connection is established, it will keep polling the heart rate data from the MiBand 3 continuously. When the Alexa command is requested, it will return the latest value that was provided from the sensor.

The polling function needs its own thread because if we do a single measurement at a time, it will take a long time for the heart rate sensor on MiBand 3 to activate. We worked around this by providing it with a thread and ensuring that it always measured the heart rate (this way it wouldn't take a long time to restart).

5.3 Alexa Skill

Alexa Skill development was accomplished with three key pieces of support software. First, Amazon's Skill Development Console was used to create a voice interaction model to transform a users audio input into useful json(s). With this, we had the functionality that, in addition to handling amazon default intents, the model included 9 different custom functionalities for the robot and about 5 keywords/phrases for each function. These functions/capabilities are listed below:

- Test amazons communications with the robot
- Track the user
- Look at the user while tracking
- Stop moving the robot
- Terminate all active scripts
- Get the user's current heart rate
- Take the user's photo
- Send a distress signal with the robots current position

These generated jsons were then sent to a web based endpoint publicly available via the ngrok application. This software allowed us to forward incoming packets through our firewall, so they could be handled securely on the jetson.

We then used the flask-ask python library to handle these jsons. This was convenient as flask-ask handled the signed certificates from Amazon in the background for us, and

it let us treat incoming jsons as events in Python. These Python events then placed the commands into the relevant queue(s) before returning a statement to flask-ask which pushed it through ngrok to Amazon and finally to the user as audio from the Alexa.

One thing to note was that we opted not to use reprompts in our Alexa Skill. The down side to this was that you had to provide Alexa with the launch command before every new custom command you wanted her to perform. The upsides were that the 15 second timeout on open skills didn't matter now and that the Alexa would take a lot less traffic (as we need constant messages to keep the event active mid operations).

5.4 OpenCV

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library—built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Our work with the OpenCV library in Python serves as the core of our person following and april-tag/distress-signal/swarming functionality.

Extensive comments have been added to our code for OpenCV-related tasks (openCVController.py—which can be found in the aforementioned GitHub repository). These comments are meant to provide justification for our design decisions and hopefully accelerate OpenCV-related development for future classes that use our work as a reference. Please refer to the openCVController.py file and its comments to understand the details of our approach.

The openCVController.py file contains two classes and a separate function, "run", that serves as the main driver of our code. The "run" function is called externally by weaver2.py and listens to the queue passed in as an argument to initiate appropriate actions (please refer to our description on code integration for the purpose of queues in our program). The first class, Direction, is essentially a set of enumerations (Enums) to simplify sending commands down to the Arduino via serial communication and keeping track of some of our last movements/directions sent.

The other class, OpenCVController, is the primary class and centerpiece of this file. The following table is a breakdown of the functions that make up this class:

OpenCVController Function	**Description**
__init__(self)	Initializer for this class—look at the code for this constructor for descriptions of each element/variable used in the class
internet_on()	Static method that tries to open a specific URL to confirm that we are connected to the internet - this is never used in the codebase because some of our startup steps for our code already depended on internet connection.
com_connect()	Static method that is used to connect to Arduino for serial communication. This method will hang/wait until it can make the connection, so make sure not to call this method if you aren't testing with an Arduino connected (this can be changed in the __init__ function).
send_serial_command(self, direction_enum, dataToSend):	Send a serial command to Arduino. If the last command that we've sent is the same as the one we're trying to send now, then ignore it since the Arduino already has the up-to-date command. Note that there's a commented out section of this code that made it so that even if the command was a duplicate of the last send one, it would still send the command as long as a certain time period had passed. Depending on how the Arduino code worked this may have been necessary, but we did not need it.
cleanup_resources(self):	Call this when closing this openCV process. It will stop the WebcamVideoStream thread, close all openCV windows, and close the SerialPort as long as it exists (if we're connected to an Arduino).
send_mail(send_from: str, subject: str, text: str, send_to: list, files=None):	Send an email with optional attachments specified (see usage in this file)
take_photo(self, cvQueue: Queue):	Countdown timer and then take a photo using a frame from the WebcamVideoStream. The photo is sent to the emails of the group members. Note that the quality of the image can likely be increased by tuning the webcam settings (we did not because it was not necessary).
april_following(self, desiredTag, desiredDistance, cvQueue: Queue, isFirstUse, isLastUse):	Utilizes the person/eyeball_follow algorithms to move to the requested april tag and reach a desired distance from the tag. This function is called multiple times (once for each april tag we want to go to). The variables isFirstUse and isLastUse are used to avoid setting and resetting the webcam's gain and exposure rapidly back and forth (this creates a problem). By using these variables we set it only on the first april tag and reset it back to the original

	specifications on the last tag.
calc_weight(p1, p2):	Static method that is used to prioritize the coordinate information provided by closer tags as opposed to the farther away ones (which would likely have less reliable information).
get_coordinates(self, cvQueue: Queue):	Get an estimate of our current x and z coordinates and return them so they can be passed through to the chat server and disseminated to the other teams (for the distress signal). To get the coordinate, we rotate on our axis some X number of times to form images that compose a complete 360 degree view of our surroundings. We use each image (as long as there are april tags in it) to get a (x, z) coordinate value, and then we choose which (x,z) coordinate to return based off of which we deem the most correct/reliable (this decision is shown in the code). We specify time to turn and time to wait after each semi-turn because we want a stable photo/shot.
person_following(self, run_py_eyes, cvQueue: Queue):	Follow a person (represented by a green folder). If the second argument (run_py_eyes) is set to True then we'll generate eyeballs using PyGame that follow the user (folder) as it moves around.

Below are some important tools/features that were incorporated into our work that are worth highlighting:

Overshooting the desired target/destination was a common problem in the class. For that reason, it's important to design your target's bounds appropriately in reference to the dimensions of your frame (webcam image). After resizing, our frame was 600x400 pixels, and we went through a series of trials-and-errors to tune our target-at-center-of-frame and target-in-range bounds (which were specified in pixels) to work well with the build and speed of our robot and field of view of our image. You can see the usage of these bounds in driving the robot's direction in the person_following(...) function (see variables "radiusInRangeLowerBound", "radiusInRangeUpperBound", "centerRightBound", "centerLeftBound", and "radiusTooCloseLowerLimit").

For april tag recognition, we used the python apriltag library. Do note that this library has compatibility issues with Windows, so, for simplicity, development on Linux is recommended (either with a personal laptop running linux, a virtual machine, or with the Jetson as we did). Getting the library to work in Windows does seem to be possible—but the effort was taking too much time. Please refer to this library's usage in our code for examples of the different tuning options that are available for the apriltag

detector, and there are several resources available online for more information on this subject.

Lastly, a critical issue we encountered was the influence of motion blur on our webcam stream and its hindrance on our ability to detect apriltags while our robot was in motion. The problem and solution to this issue is covered in the OpenCV portion of the Problems section of this report.

5.5 Chat Server

Our class sourced the integration of chat client and server from the following link: https://pythonprogramming.net/client-chatroom-sockets-tutorial-python-3/.

In order to connect to the server (on a Windows machine), we needed to do port forwarding and set our default port for the server to 1234 (we followed the following tutorial from Tom's hardware to do the port forwarding on our Windows 10 laptop to host the server: https://www.tomshardware.com/news/how-to-open-firewall-ports-in-windows-10,36451.html). When the server is run, it will provide the IP address of the server that clients need to connect to (this would be displayed at the beginning of the terminal). For the client, we need to configure the port and IP address, and ensure that the client and server are on the same network.

We integrated the client into our chat controller file (accessible via the previously linked GitHub). In our chat controller file, we have a send function that takes two arguments: the (x,z) coordinates of the distress signal.

Because we are using a multi-threading model, we have a run function that is always listening to the response from the server to capture the distress signal. After capturing a distress signal, we parse it into (x,z) coordinates and push these coordinates to the queue for the weaver (our threads controller, weaver2.py in the GitHub) to process.

Initially, we ran into an issue with the chat and client—these files were originally written in Python 3.6 and our Jetson TX2 runs on Python 3.5 (at this point we could not afford to upgrade to 3.6 and risk any compatibility issues). In Python 3.5, the f-strings library wasn't available, and this broke the header calculation for sending/receiving between

server and client. Fortunately, we fixed this issue after figuring out how to calculate the header and then writing our own function to calculate it. After this fix, we shared the modified chat/server code to the rest of the class.

5.6 Navigation Methods

The algorithm we used for navigation to points in the test-space was straightforward. We assumed that the robot would start in the lab, in-view of the front wall, and that the final tag was in the hallway. First, we would look in 360 degrees to find our current position in the room, and then we would drive to the tag closest to that position on the wall. We then follow a preset path of 3 tags to navigate out of the door and into the hallway. Following this, we follow a zig-zag path of tags down the hallway—stopping when we get to the tag closest to the desired end location. This was a reliable, proof-of-concept, method. We discussed several, more advanced, alternatives, which we would have adopted if we had additional time to experiment.

5.7 Integration of Pieces

To get all of these subsystems working together, we called them as threads in our main file (weaverv2.py). Each of these threads is given a joinable queue as an argument which it uses to communicate with weaver. Weaver also calls flask-ask to run as a thread which creates event that trigger for each command. These events put keywords into the queue for the relevant thread, and in the threads these keywords are parsed so that the correct action is taken.

Most of the separate threads also call a thread of their own for reading commands/data from their queue parameter. This was advantageous as it allows us to push updates without waiting for the process to complete its current task (something that proved extremely valuable when creating commands for terminate and halt).

But, do note that we did not call the thread for our open CV code as a thread—rather, we created a separate process. We did this because of the python global interpreter lock—it forces all threads onto a signal processor core, but processes are not subject to this restriction. Within the CV process we also did not create a thread for checking the queue, as each switch of the active thread forces a pipeline flush and thus hurting

runtimes. If you look at our OpenCV code, you'll see that an operation (such as person_following) is ended when new additions are made to the queue (we end our operation and return to the run() method to analyze what the queue message is—thus avoiding the use of threads entirely in the code).

6 Challenges & Solutions

6.1 Initial building/wiring

Our group had a rough start in terms of wiring/construction. When we first received our robot chassis and Jetson, we were told that our Jetson was not functional based on the previous owner's description (and inability to get the Jetson working for their project). It took us a while to reflash the "broken" Jetson, but we ultimately succeeded in turning the Jetson into a workable state. Additionally, before we finished reflashing process, we received a new Jetson as a replacement. The other Jetson became a spare for us to develop on, and this proved to provide a huge benefit as we had two pieces of hardware to test and develop with.

We also had problems with our chassis—the previous group left their chassis in a horrible state their robot did not use a Jetson and relied on a laptop). The pole in the middle was loosely held by 1 small screw, and there was nothing for the Jetson to mount onto. All the wheels are bent and the tail drag pushed up the chassis which made it unbalanced. We had no idea how the previous owners could work with this construction, so we scrapped everything and started from scratch. Luckily, we received spare parts from Murphy's team which we used to construct our robot.

After fixing the chassis, we looked at the wiring. With no color-coordination or uniformity, we felt it necessary to redo the wiring for the Arduino, motor controller, and ping sensors.

6.2 Alexa Configuration Issues

We ran into a variety of time wasting problems when setting up the alexa.

First was the wifi network. As no member of our group lived on campus we didn't have access to the umd-iot network for devices like this. We solved this by hosting a wifi hotspot off of a group members laptop.

We then attempted to use Amazon's traditional method of backend hosting, lambda. This had two problems—(1) it was very hard to test code hosted on lambda making progress slow and unintuitive, and (2) it made it hard to get the needed information of what script was called onto the jetson. We tried sending emails with this information but this was a slow and ineffective solution. So we switched to using flask-ask and ngrok for our backend. This also had problems as the version of cryptography pip install called by flask-asks wheel pulled a newer version of cryptography that flask-ask hadn't been updated to support. Downgrading the cryptography then asking pip to ignore flask-asks wheels fixed this problem.

6.3 OpenCV

Our group encountered and resolved several issues while working with OpenCV in Python. A notable issue was with motion blur in the webcam stream when the robot was moving and trying to detect apriltags—this was resolved by modifying the webcam's exposure and gain on-the-fly to settings that worked best for the lighting in the lab.

To expand on the above, modifying the exposure and gain settings of our webcam helped enormously with improving apriltag recognition. During our initial testing, we noticed that our robot failed to detect the desired apriltag while it was moving/rotating. The issue was that the motion of the robot was impairing the OpenCV/image-processing capabilities (the motion blur was making the apriltags in the frame undetectable). We got around this by introducing a turn-stop-turn mechanism so that we could let the camera settle down and analyze frames that did not have motion blur. This worked, but it was a solution that slowed down our speed-of-task-completion. Further, as the robot moved toward the target/apriltag, it constantly jittered (sending forward and stop commands in a quick succession) because the motion blur from the forward movement of the robot was causing the same blurring issue.

Our solution was to reduce the exposure of our webcam and increase the webcam's gain to counteract the now dimmer image. We only want to do this when we're running the april_following(...) code, so we save the original exposure and gain of the webcam for restoring later. We'd like to restore the camera to its original settings when it's not needed to specialize/adjust them because that enables the camera to choose the best settings for the environment and change its exposure automatically, and so that our HSV color space settings for person_following remain correct). These exposure and gain changes can be seen in the april_following(...) code. With this solution, our robot could recognize apriltags across the room even while rotating at a fast speed. Do note that not all webcams have the capability of tuning these parameters.

Lastly, our group spent a fair amount of time trying to utilize the full capability of the Jetson TX2 to facilitate snappy OpenCV performance and high frame rates. It became apparent that, from our understanding, to easily utilize the full capability of OpenCV and the Jetson TX2 we would need to develop our work in C++ as opposed to in Python. Ultimately, dedicating 1 CPU core entirely to our OpenCV functionality (that was written in Python) was a sufficient solution for our purposes. We also opted not to split these functionalities into separate threads on that core to improve cycle rates on image processing. This change created a number of smaller problems and forced us to move data with queue instead of using global variables, as each process is a separate instance of python. While this took a while for us to implement it reduces overall project complexity while pushing our frame rates. This proved even more valuable as higher frame rates allowed us to remove pauses in navigation and speed up the motor speed without creating overshoot errors in our controls (1, 2, 3, 4, 5, 6, 7, 8, 9).

In addition, building OpenCV from source would be a good decision to ensure that it is optimized to work with Jetson TX2 (1, 2, 3, 4).

6.4 Bluetooth disconnects

As we mentioned above, the Bluetooth protocol that MiBand 3 is using is the Bluetooth Low Energy instead of the normal Bluetooth connection. The most common problem we encountered with the Bluetooth Low Energy is that sometimes it can connect and sometimes it cannot. This happened when we stood a bit far away from the robot. If the connection between Jetson and MiBand3 couldn't be established, the library would throw a large amount of traceback to the terminal.

To work around this, every time we run the weaver controller (which will run the heartbeat controller in the beginning), we keep the MiBand 3 right next to the Jetson in order for it to connect. After it connects, we don't have to stand close to the robot to measure the heart rate (this problem only happens at the connecting phase of Bluetooth Low Energy).

With additional time, we would have tested using try-except clauses to keep retrying to connect to the MiBand 3 until the connection is established.

6.5 World Coordinates inaccurate and extremely poor Yaw

Before developing our current navigation method the first plan was to use a map of the april tags and the view of them around the robot to triangulate our position. We would then use that position to navigate to the door and through the hallway.

Unfortunately, this method proved to not be nearly accurate enough to accomplish this. We could find our X and Z position within about a foot of error but the yaw of our robot was very inaccurate with up to 90 degree error in the worst case. This led to sporadic driving and a lot of navigation failure.

To fix this we switched to a system where we drove straight at an individual tag and ignored our own position. To prevent collision, we used the size of the tag to judge our distance from walls. This proved effective and solved the problem. After our success, other groups seemed to adopt the same approach.

6.6 Raspberry Pi 3+ WiFi failure

Near the end of this semester, we had new ideas in our heads that we wanted to try out to improve the quality of our system. One such idea was the following:

Since the school changed the WiFi this semester into "edu-roam" and to connect to it, we needed to put our login id and password in. Alexa didn't support this type of verification on the network, so we had to use our phone or laptop to make a hotspot for Alexa to connect to. Our idea was using a Raspberry Pi 3+ with built-in WiFi as an access point to distribute wifi to the Alexa. Our Raspberry Pi will get the network

connection from Jetson's ethernet port. This is called bridging in network.

We tested everything and it worked fine at home, but when we brought it to the lab and tried it. It only connected for about 3 to 5 minutes and then the connection was dropped for some unknown reason—this made our Alexa unreliable when connected to the Raspberry Pi. We suspected that the reason either could be the Raspberry Pi was under voltage or the DHCP from the Jetson or the Raspberry Pi messed up. In the end, we decided to scrap this and used our laptop or phone to host the hotspot for Alexa.

6.7 Low power supply for Alexa + screen

As we added more components to our robot we ran into power supply issues. The combination of an extra screen and the alexa resulted in low currents from the Jetson's USB ports. As a result, the Alexa Skill would close and the device would need to reboot while we were trying to give it commands.

We fixed this problem by adding an additional battery which powered the Alexa exclusively. We tried to add an additional Raspberry Pi later (see Section 6.6, above), but this proved to be a power issue as well so it was scrapped as we didn't have a third battery to strap onto the robot and the weight was starting to cause issues with our locomotion.

7 Feedback

Overall, the class was a great experience—we appreciated that we pretty much had to uncover and solve all our problems on our own (both within our group and between all class members). The creative freedom and sense of autonomy provided us with a valuable experience with several takeaways (as mentioned in the conclusion).

In terms of the general approach to the class—I do not much feedback regarding necessary changes. Xiaomin was a great TA that pushed us to meet our goals while leaving the otis on us to actually follow through and be responsible for the work.

From our discussion with other class members, many of us have agreed that we hope our write-ups regarding challenges faced and their solutions will prove useful for future

classes. If future students take the feedback and lessons learned from these papers, subsequent semesters should hopefully be able to create a more advanced and sophisticated robots. Given that next semesters project seems to have been largely decided at this point, I would consider the recommendation that goals from this semester be incorporated into, or as an addition to, the goals of next semester. This would provide a strong incentive for future students to look back at these papers and build off of the knowledge and lessons learned from the past.

-Nikhil Uplekar

The class gives us opportunities to learn and explore different skill sets and different perspectives. Honestly, I'm happy because I have two awesome teammates. We have good communication between us which is rare and needs to be built over time. They helped me share my workload and I really appreciate that. Xiaomin helped us and gave us advice on how to manage the project and which part was expected to be done in a certain period. To me, he's a good TA and a good friend.

About the class's pace, I feel like the class was very slow in the beginning. Honestly, the goal for swarming is hard to achieve in a short period of time and we only started doing it in the last half of the semester. I saw this created huge pressure on the whole class during the last half of this semester. Also, it would be really helpful if we have more than 1 lab a week. For our group, we had to meet outside lab time 3 times. All 3 times, we spent like 8-12 hours straight to get things right and working.

Over all, I love this class. It's been a good and fun experience for me.

-Huy Do

I really enjoyed this class. I felt the TA did a good job with help us over problems without trying to direct to much, and I felt the challenge level was appropriate.

I think in the future the class would be better with less network heavy challenges. Making the teams robots communicate proved to be a big problem that sapped resources from more interesting parts of the project. While it's reasonable to get a 3 man group together overtime to solve problems getting the whole class to work on bug ridden server code was impractical.

I also would have appreciated more direction on alexa development. Personally I

wasted a lot of time doing stupid thing and trying to follow internet guides on alexa developement that did not pertain to the way we wanted to use her. Having better resources on this part of the project (like guides or just documentation) would have been appreciated.

That being said this is one of my favorite course I've taken in ECE, and im glad I got to work with such excellent hardworking teammates.

-Nathan Koenigsmark

8 Conclusion

As mentioned in our introduction, our group is happy with the final results of our project. We are proud of our work, and we hope our documentation and code will be of use in future semesters. We believe that our class—as a whole—encountered, solved, and documented problems that could be considered significant "time-sinks" for students. We hope that our work and documentation (along with the work and documentation from the other groups of our class) will help future students navigate past the issues faster so that they may focus on the primary task at hand. The three members of our group have come away from this class with valuable experience in teamwork, time management, technical specification and requirement analysis, and engineering prototyping.We wish the best of luck to the students taking the course in the future, and to Dr. Blankenship, Xiaomin Lin, and Jay Renner for making this class possible.

9 Appendix

Figure 1: Back view of robot's wiring

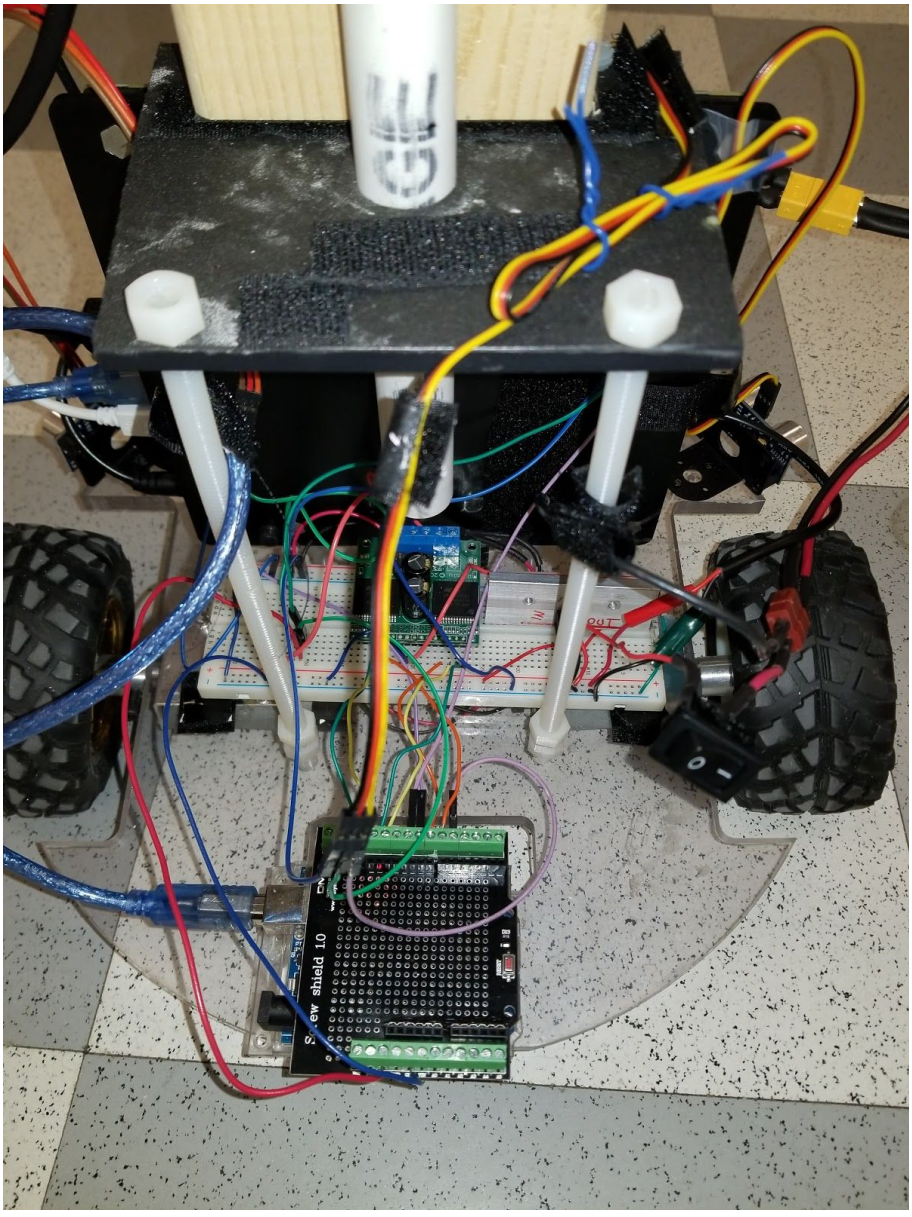

Figure 2: Front view of robot

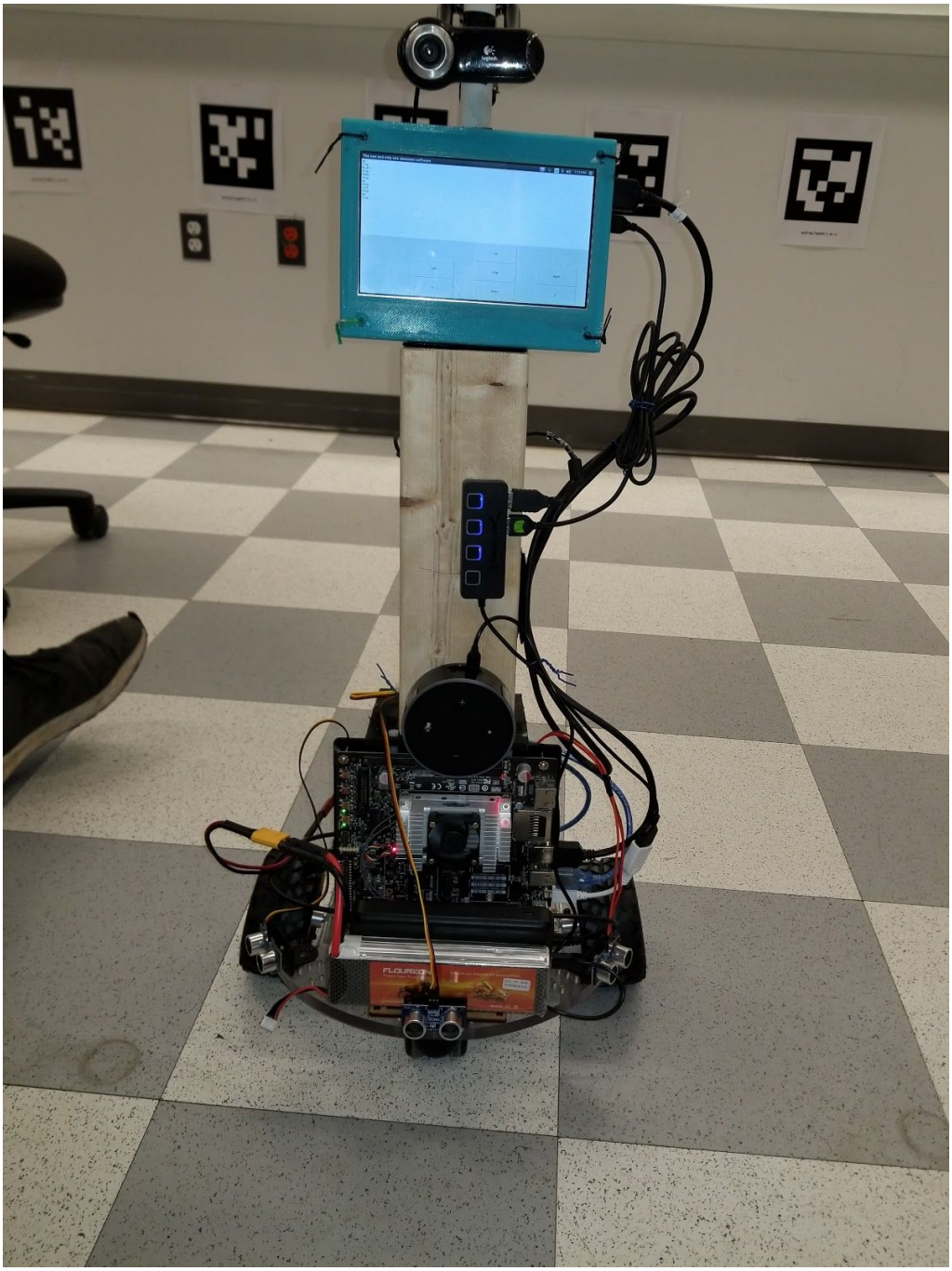

Figure 3: Front view of robot (2)

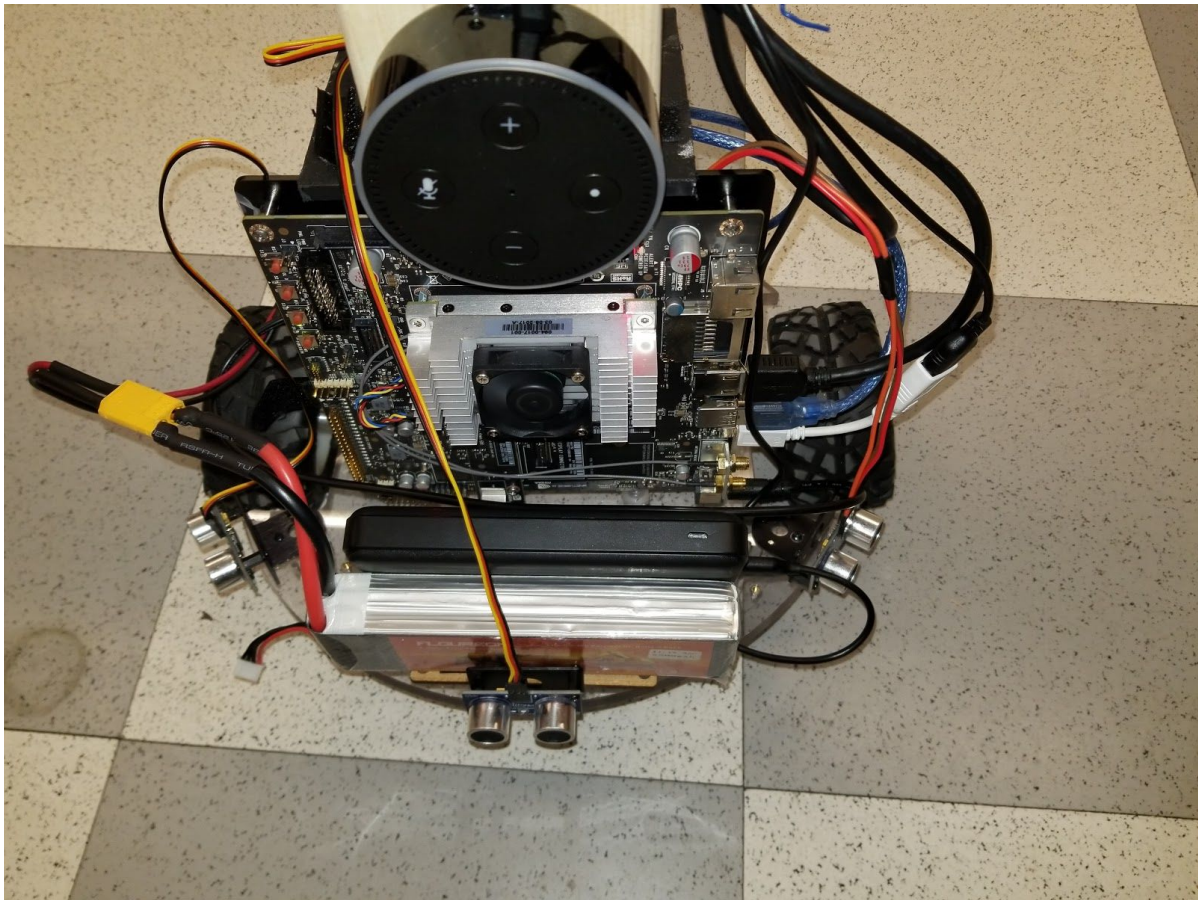

Figure 4: Side view of robot

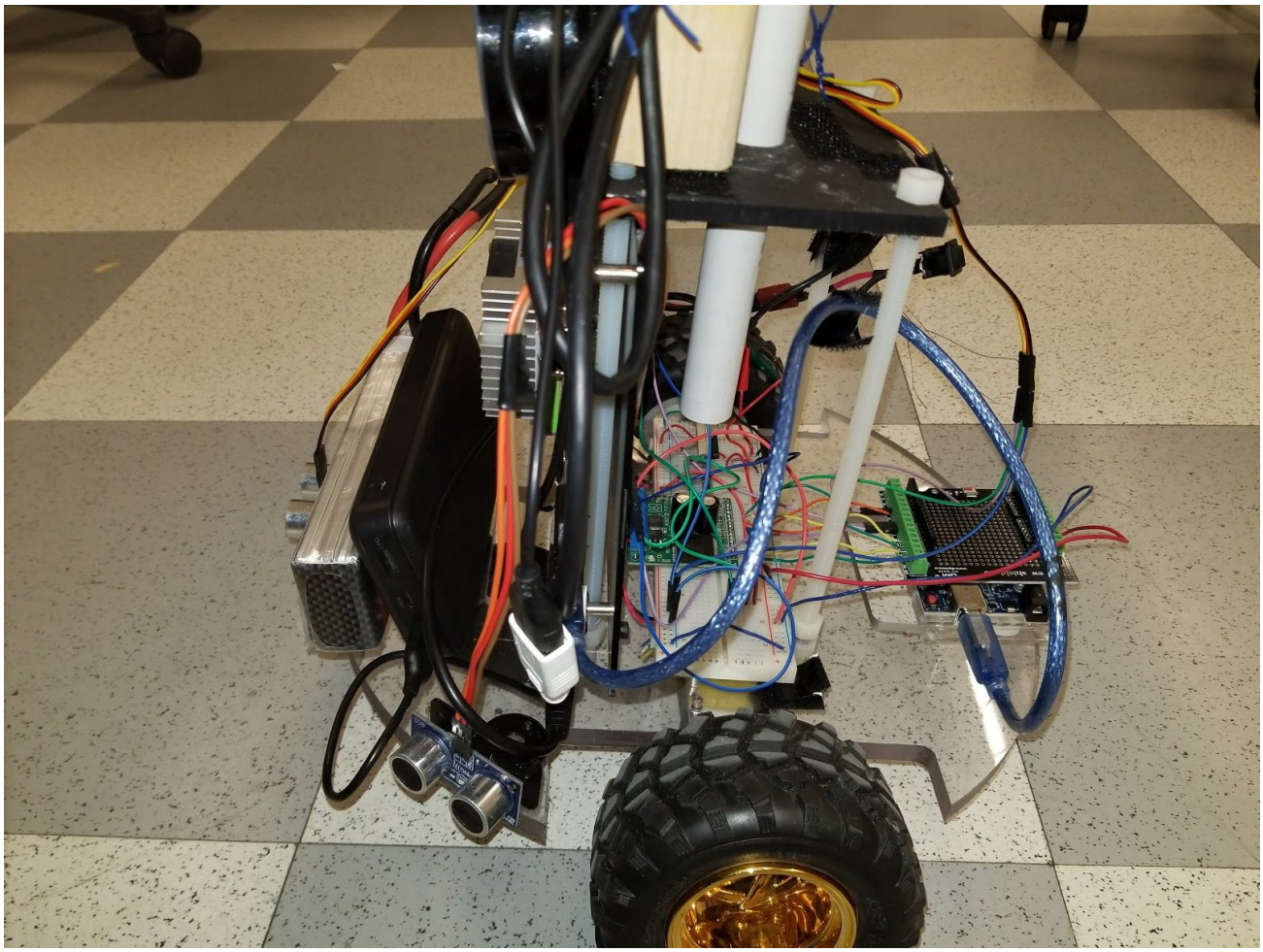

Figure 5: Close up of wiring

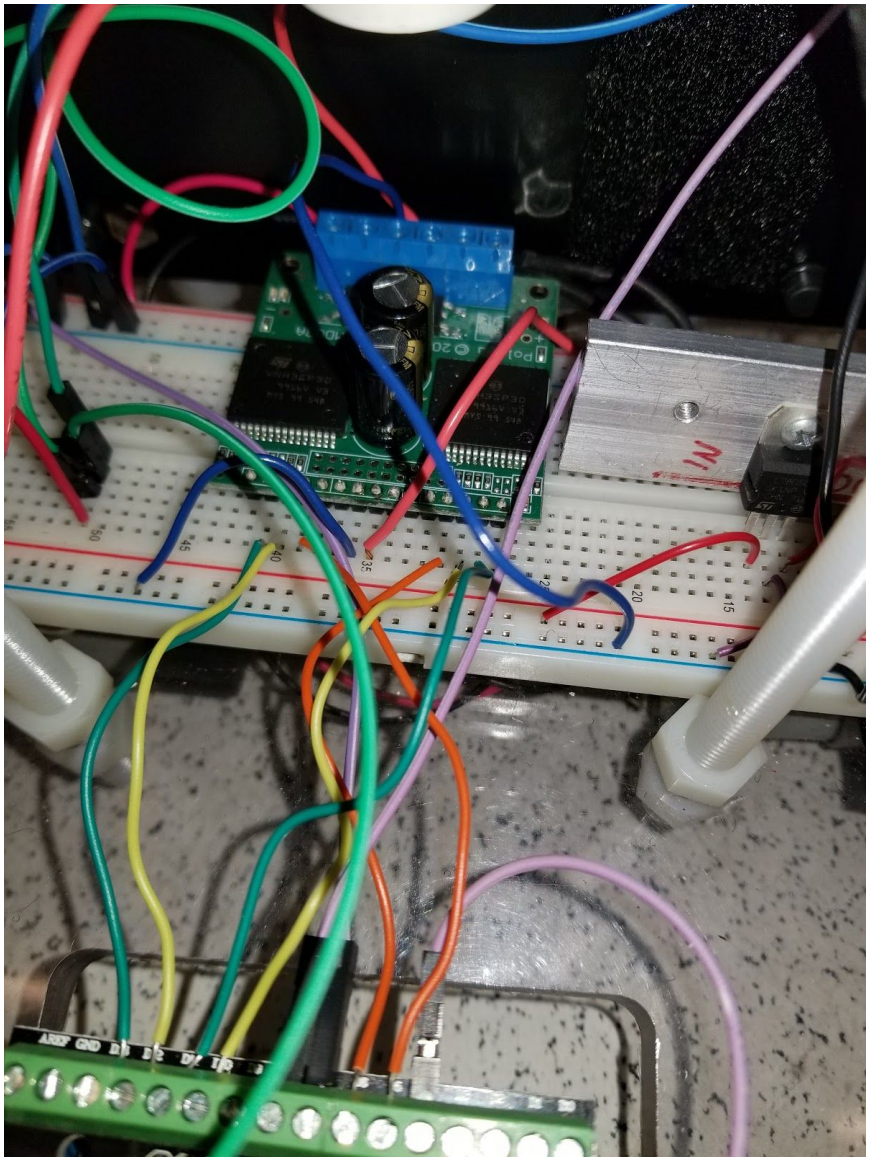

Figure 6: Back view of robot's wiring (2)

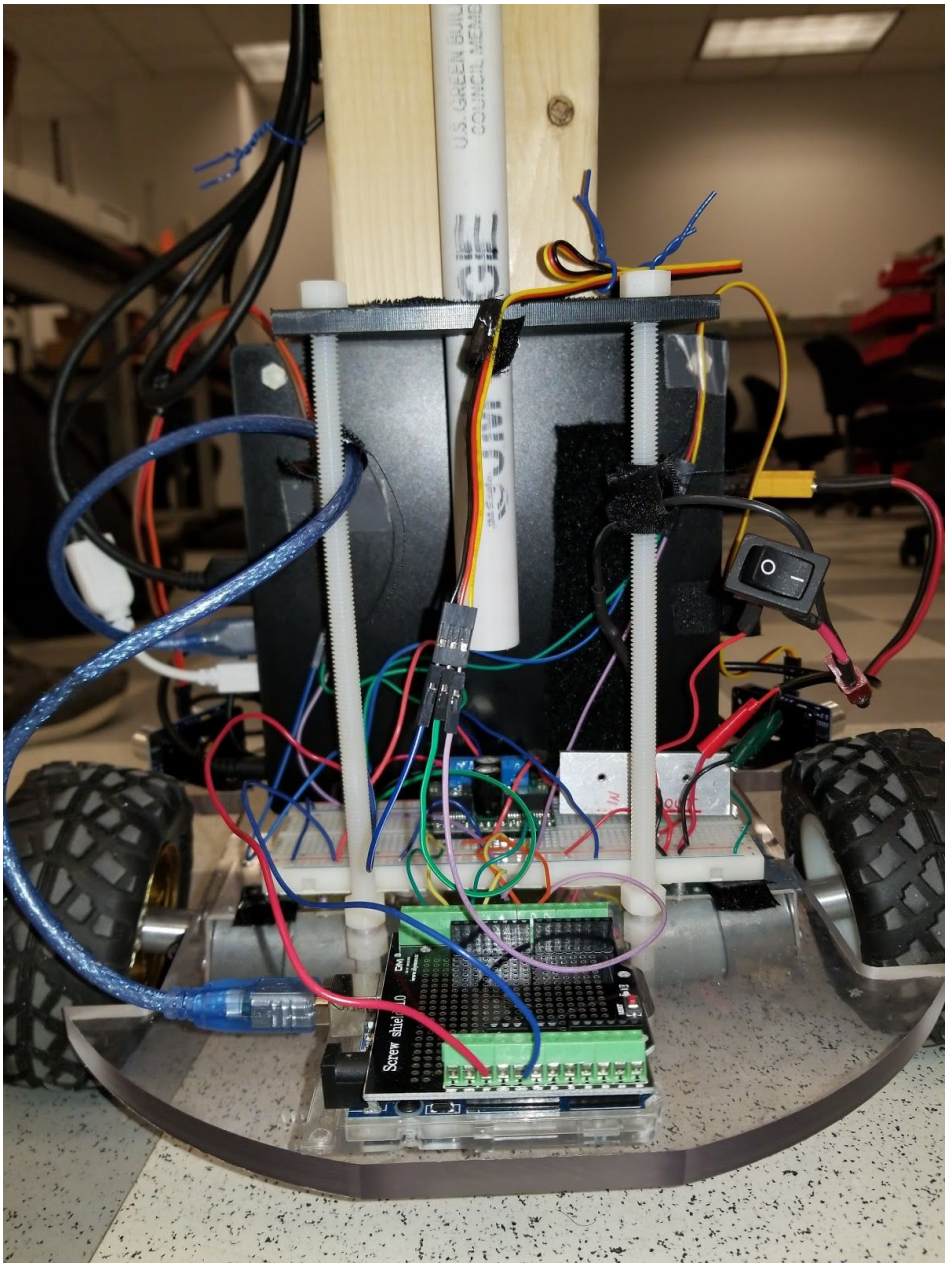

Figure 7: Back view of robot

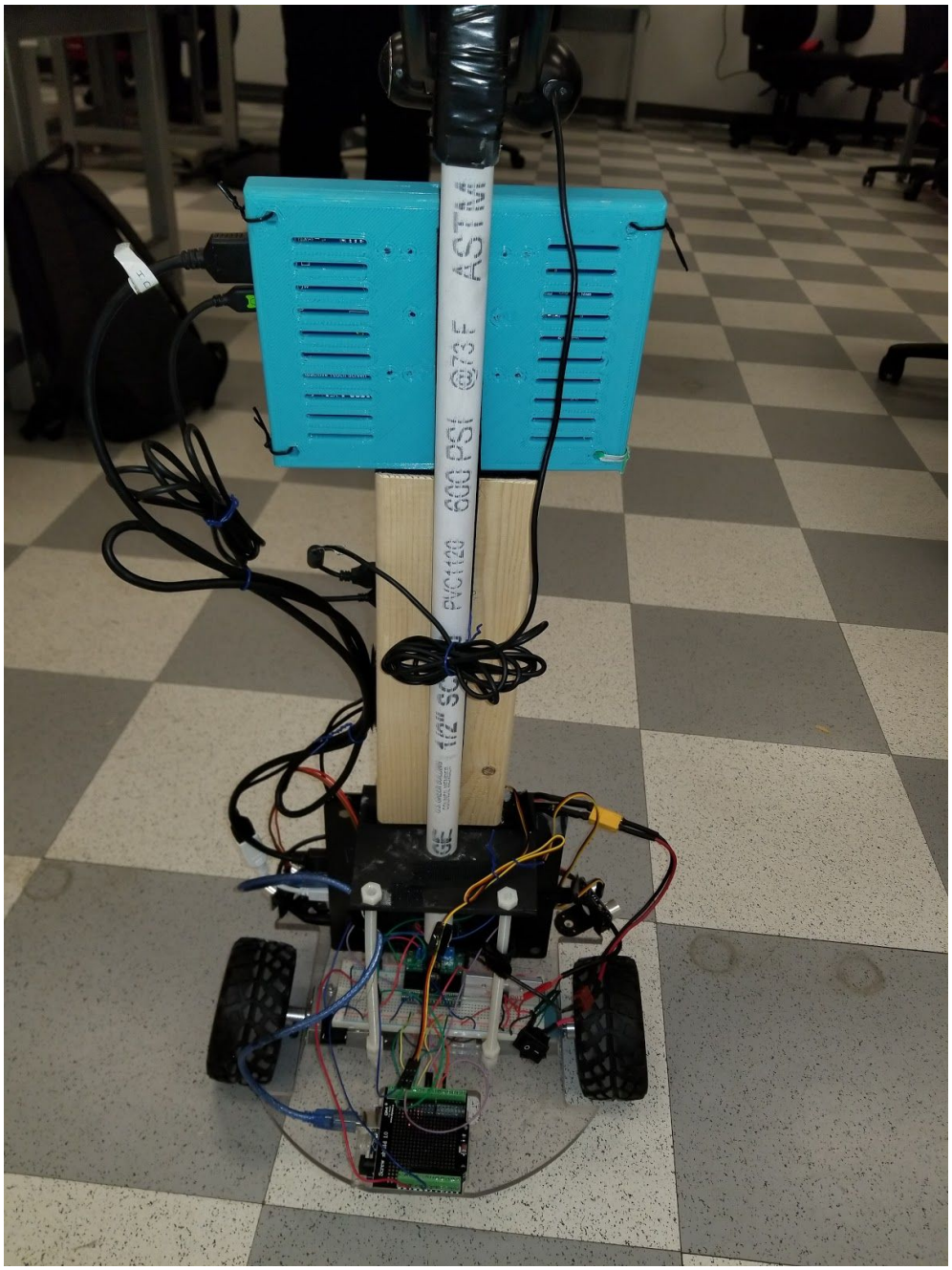

Figure 8: Side view of robot (2)

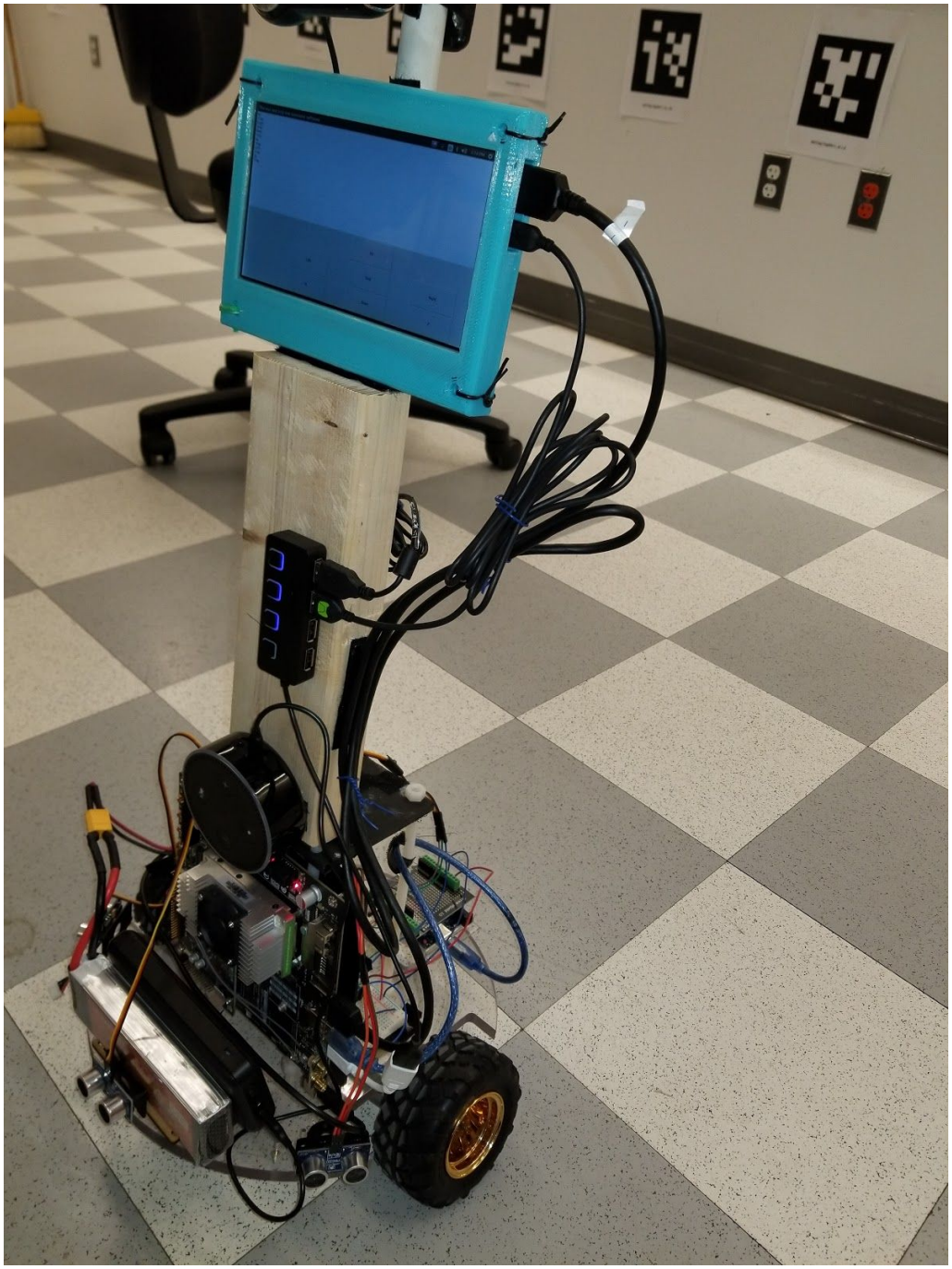

Figure 9: Robot after navigating to distress signal location (outside lab)

Figure 10: Photo taken from robot during final demonstration (sent to group members via email)