Introduction

For my UC Berkeley Master of Engineering capstone, I worked as the controls/ hardware/ software/ mechanical engineer for Squishy Robotics' centrally actuated payload team. My most notable achievement, perhaps beside building Squishy's first centrally actuated tensegrity mobile robot, was implementing face-to-face, semi-autonomous control using Dijkstra's Shortest Path First algorithm and an IMU.

Squishy Robotics is a startup based in Berkeley, California. Squishy's mission is to develop mobile robots which can be used to assist emergency responders. Squishy robots are tensegrities in structure, meaning that they are composed of members in pure tension and compression. Tensegrity structures are resilient to high impact (such as from being air-dropped via drone) and have many degrees of freedom.
Squishy Robotics Website: https://squishy-robotics.com/

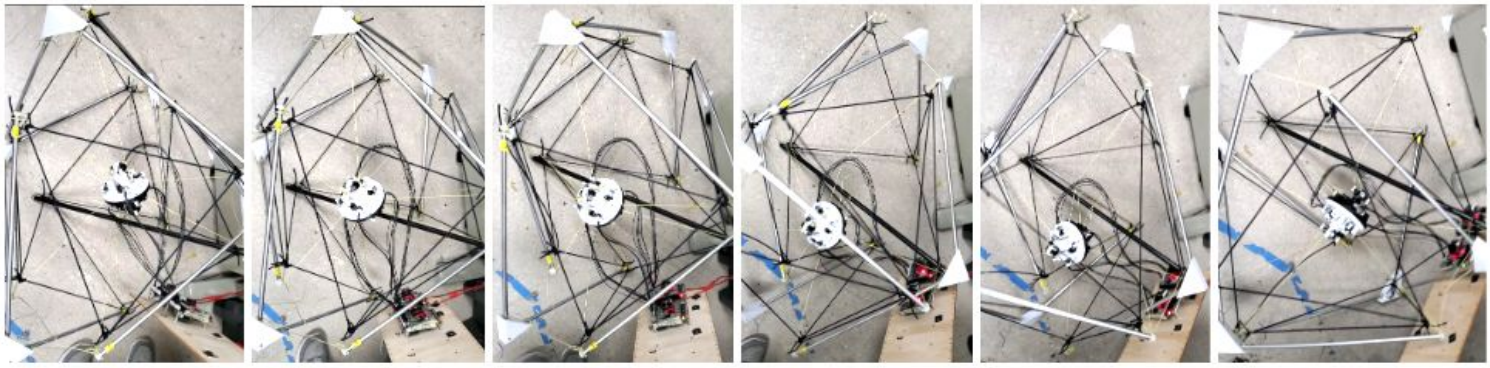
The goal of the centrally actuated payload mobile robot team is to demonstrate the feasibility of a mobile robot using a quarter the amount of motors used in the current mobile robot prototype, Mobile Robot 2 (MR2) as seen in **Figure 1**.

Figure 1: MR2, Squishy's Flagship Mobile Robot

Flounder 1.0: Punctuated Motion with 12 Motors

After designing a center payload, assembling hardware, building the structure, and troubleshooting, Flounder (**Figure 2**), Squishy's first centrally actuated mobile robot, achieved punctuated motion (**Figure 3**). See a video here. Flounder has bent rods to allow free motion of the center payload, bungee cords as the members in tension, and a center payload with 12 mounted motors. For motion, the motors would plug into the board, which would be held by an operator. The board would be incorporated into the center payload in a later iteration of Flounder. An operator would type in encoder values into a terminal and that signal would be wirelessly sent to the board, which would then operate the motors.

Figure 2: Flounder 1.0 - 12 Motors and Original Bent Rods



0.0 s 1.5 s 3.0 s 3.4 s 3.7 s 4.2 s

Figure 3: Flounder 1.0 Punctuated Motion

Flounder was iterated upon and became Flounder 1.1 (**Figure 4**), where the optimal angle for bent rods was found. It was found that in Flounder 1.0, the bent rods would get in the way of full punctuated motion.

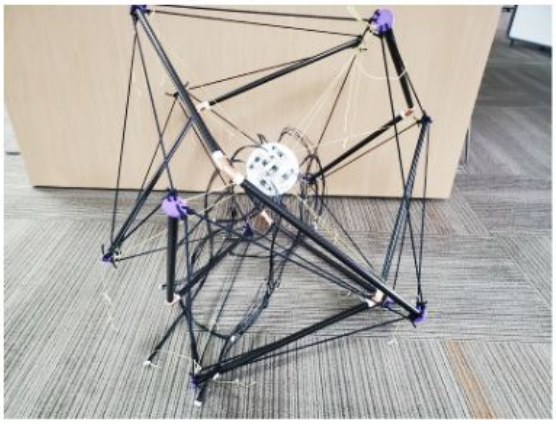
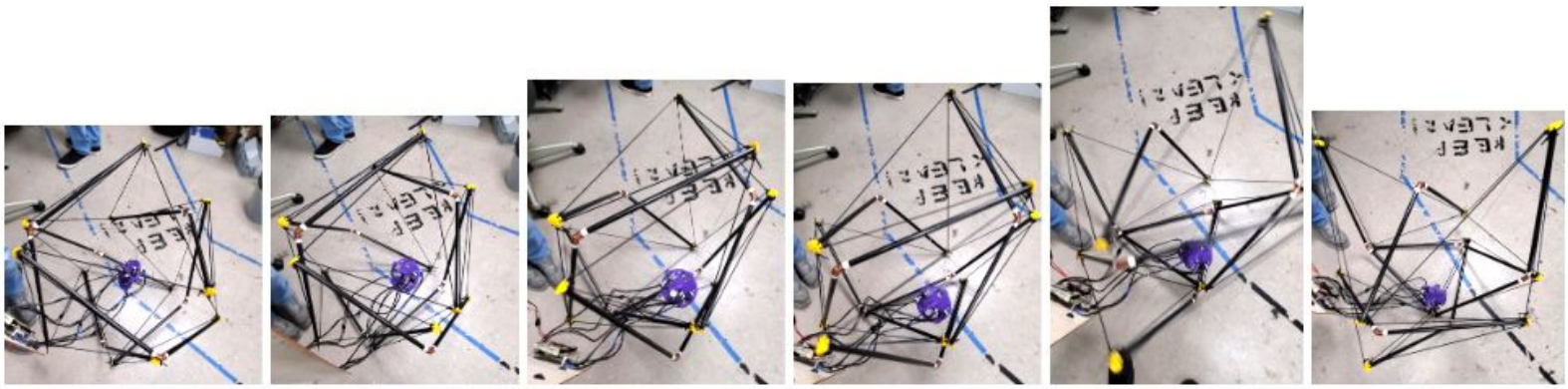


Figure 4: Flounder 1,1 - 12 Motors and Improved Bent Rods

Flounder 2.0: Punctuated Motion with 6 Motors

After the 12-motor Flounder was able to demonstrate punctuated motion in multiple directions, the second main iteration of Flounder was built, Flounder 2.0 (**Figure 5**). Flounder 2.0 incorporated a newly designed center payload for housing only 6 newly specced motors, and custom designed and machined double spools made of Aluminum. Flounder 2.0 achieved punctuated motion (**Figure 6**). See a video here. Flounder 2 achieved several instances of punctuated motion in all directions.

Figure 5: Flounder 2.0 - 6 Motors



0.0 s 8.7 s 14.0 s 14.2 s 14.3 s 14.5 s

Figure 6: Flounder 2.0 Punctuated Motion Before PID Fix

Flounder 2 experienced motor gear slippage, which was partially solved with a higher communication rate. The gear slippage caused inconsistencies is the needed encoder counts to achieve punctuated motion, and thus deemed the robot unreliable. It is estimated that with the new code running, Flounder 2 achieved punctuated motion in less than 14 seconds.

Flounder 3.0: Autonomous Punctuated Motion with 6 Motors

Despite Berkeley's shelter-in-place, which meant that Flounder could only be worked on by one person at a time, significant progress was made. Flounder 3 (**Figure 7, 8**) was developed, which most notably includes the control board within the payload itself, unlike previous iterations of Flounder.

2 s 7 s 11 s 13 s

Figure 7: Flounder 3.0 Punctuated Motion

1 s 7 s 15 s

Figure 8: Flounder 3.0 Punctuated Motion, Recentering

Flounder 3.1 (**Figure 9**) was then quickly developed so that the center payload would take on a spherical shape rather than an asymmetric oblong shape. With this iteration of Flounder, Adafruit's BNO055 IMU was used, along with software programmed in Python using Dijkstra's Shortest Path First algorithm to implement face-to-face motor control. Note that in previous iterations of Flounder, the operator would have to tediously input desired encoder counts into the Python user interface; control was manual. Initial tests with Flounder 3 showed that the software I built was working. Full testing was not implemented due to imperfections of the center payload, i.e. fraying pulleys.

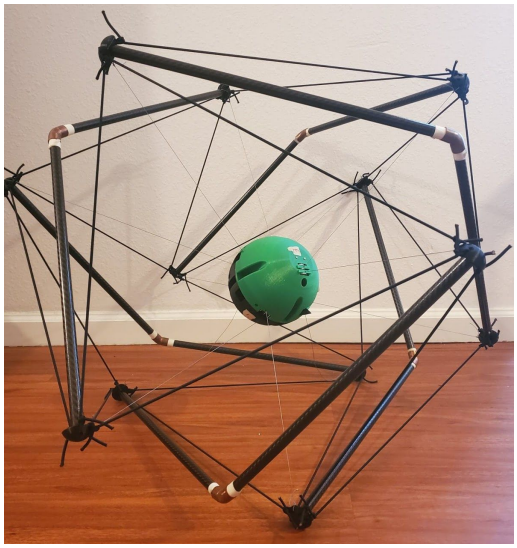


Figure 9: Flounder 3.1 - 6 Motors, Integrated Microcontroller

Face-to-Face Control of Flounder using Dijkstra

Flounder 3 was an immense benchmark for the team, and was quickly iterated upon and marked the final iteration of Flounder built during the duration of the school year. Flounder 3.2, or more appropriately, Flounder (**Figure 10**) was able to demonstrate successful face-to-face motion using Dijkstra (**Figure 11**).

Figure 10: Physical Mapping of Faces for Flounder

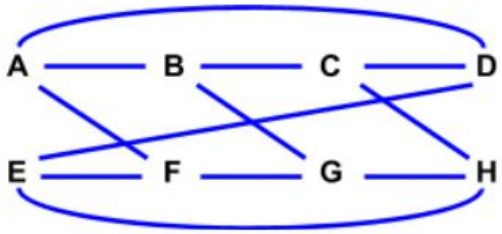


Figure 11: Dijkstra Mapping of Flounder

To set up Dijkstra's algorithm, nodes and edges were defined - with each node representing a closed face (denoted using an alphabetic letter) and each edge (shown with blue line) in Figure 11. This mapping represents the physical adjacency between faces and shows the potential paths to get between any two faces. Figure 10 shows the physical mapping of these faces on Flounder.

In implementing Dijkstra's algorithm, edges between adjacent faces are given a weight of 1, representing that only 1 instance of punctuated motion is needed to travel from one face to the other (note that this implementation of Dijkstra is bidirectional). The cost of each path is determined by the total weight between the two nodes. For example, the cost of going from A to B is 1, while the cost to go from A to C is 2, with the optimal path defined by Dijkstra as A → B → C. The greatest possible weight of the most optimal path between the furthest two nodes, for example from A to H, is 3, with the optimal path defined as A → B → C → H. That said, there are multiple paths of the same weight that can be used to get between these two nodes - which represents the different directional movements in the ground plane. This principle leads to the idea of directional continuous motion, which will likely be implemented in future iterations of Flounder.

When implemented on the actual robot, Dijkstra defines the optimal path with assistance from the IMU to ensure the robot is actually travelling along the intended path. For example, if Flounder is currently resting on face A, the user interface informs the user that Flounder is resting on face A. The user may declare a desired face for Flounder to go to, for example, face C. Dijkstra's algorithm runs and determines the next face, B, and the robot actuates to move to that position. Once the new position is reached, the IMU performs a check to ensure the robot has actually reached face B and, once it has correctly reached that position, actuates again to follow Dijikstra's algorithm from face B to face C. If the IMU detects that a different face than intended has been reached, Dijkstra's algorithm will be rerun to determine the optimal path to go to reach the desired face. Once the IMU determines that Flounder is on the desired face, all motor actuation ceases. **Figure 12** below shows pseudo-code for the software built.

```
while True:
    ask user if they'd like to induce face-to-face motion
    if yes:
        Output current face to user
        Ask user the desired face to reach
        while current face != desired face:
            Use Dijkstra to calculate next face
            Actuate motors to get to next face
            Output current face
```

Figure 12: Pseudo-Code for Face-to-Face Motor Control of Flounder

Centrally Actuated Mobile Robot Simulation

Our team achieved a specific face-to-face version of autonomous control, but the idea can be extended further to more practical applications, including directionality based control. More generalized autonomous control requires a greater range of sensors, such as cameras, to physically interpret and manipulate the entire system. To achieve this, higher level controls could be implemented, based on minimizing a cost function, while constraining states and inputs. An example of one such higher level controller is a Model Predictive Controller, or MPC. Unlike the algorithmic approach used for controlling face-to-face motion, MPC has the potential to allow Flounder to travel greater distances with minimized energy losses.

It was found that a 0.01s simulated timestep is significantly better than a 0.1s simulated timestep, likely due to errors in the linearization of the dynamics (see links for videos). Note that all other variables between simulations were kept constant. **Figure 13** below shows the progression of the simulation when set at a 0.01s timestep and **Table 1** shows the parameters for this particular simulation.

Table 1: Parameters of Central Payload Mobile Robot Simulation

Parameter	**Value or Method**
Controller	MPC
Simulation Timestep	0.01 s
Total Simulated Timesteps	1000 steps
Controller Prediction Horizon	10 steps
Incline	3 degrees
Simulated Robot Apx. Diameter	0.5 m

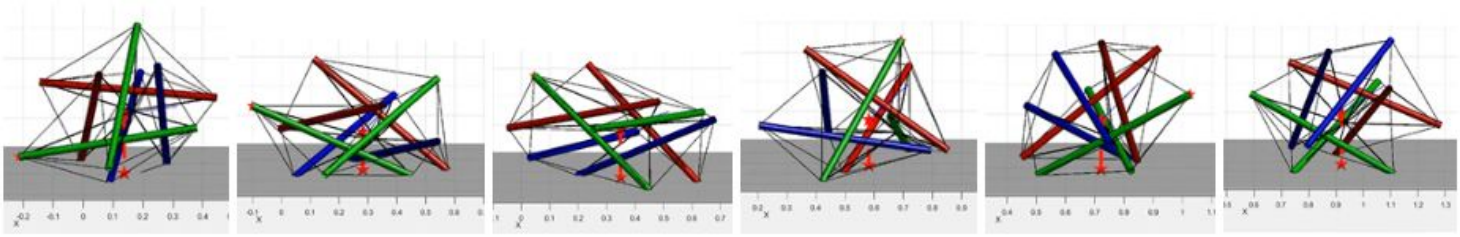


Figure 13: Central Payload Simulation with 0.01 s timestep

Traveling at an average velocity of about 0.11 m/s, **Figure 14** shows the progression of the robot's center of mass (COM) from a bird's eye view - represented by the solid blue line. The red star shows the starting point of the robot, and the dotted triangles show how the robot's external structure interacted with the plane along which it traversed.

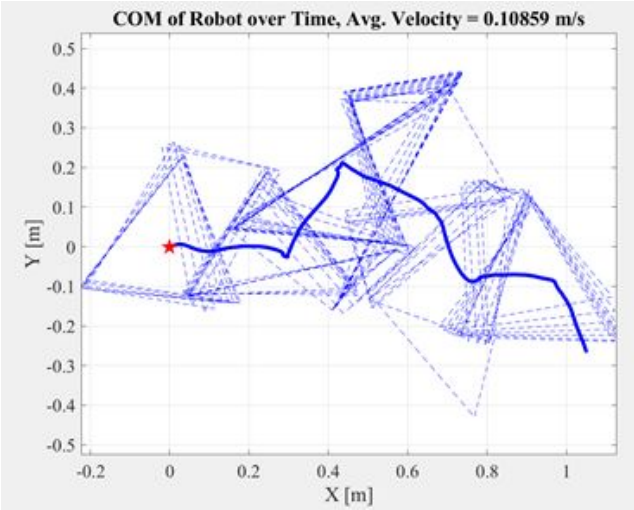


Figure 14: Progression of a Mobile Robot's Center of Mass during a 10s Simulation

Presently, physical motion on Flounder is implemented via a system of tensioning and loosening paired cables to induce motion. Only one pair of motors, and thus, only four cables, are actuated at a time. Using MPC, there is the possibility of actuating multiple motor pairs simultaneously in order to optimize maneuverability. **Figure 15** below shows how cables are tensioned and loosened throughout the 10 second simulation. The top graph shows how individual cables acted, and the bottom graph shows the cumulative trend of tensioning and loosening of cables over time.

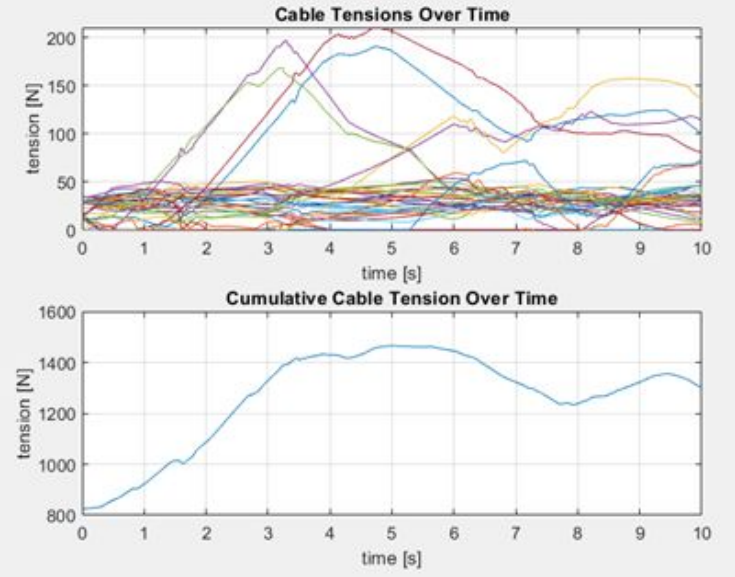


Figure 15: Individual and Cumulative Cable Tension Over Time