RSS Team 5 Challenge Design Document

Eric Wieser[1,2], Ernie Ho[1], Shi-ke Xue[1], Steven Homberg[1], Winter Guerra[1]

***Abstract*—The project is a culmination of the work and learning this semester about the provided racecar and ROS. The racecar must autonomously navigate through MIT's basement tunnels and complete an obstacle course in the fastest time possible. This is a race against other teams in two parts - the robot is timed running the course on its own and also timed while concurrently running the course with other teams' racecars.**

I. Problem Statement

The goal of the challenge is to enable a mobile robot platform to efficiently traverse a static environment with known map containing several unknown obstacles and targets.

The course the robot must traverse is in a portion of the MIT tunnels below MIT, with flat ground and walls. This provides a reasonably uniform environment for the robot to traverse. Beyond the baseline of the tunnel, there are several additional obstacles presented by the environment. People will be standing at the side in various portions of the course, which can alter the observed environment compared to that which is expected based on the map. There are also cone-like obstacles in the middle of the course in several places which the robot must avoid.

Positive Goals

Reaching the end of the course through the tunnels as fast as possible is the ultimate goal of the challenge. In addition to completion of the course, there are colored targets on the ground which, if touched by the robot, will increase the robot's score.

Negative Goals

Colliding with one of the cone-like obstacles placed on the course results in a penalty to the robot's score.

Although not explicitly penalized, other crashes, for example with the walls, decrease the robot's score by postponing the completion of the goal.

II. Assumptions

- We have "instantaneous" control of the car's angular velocity using the steering. This simplifies the RRT motion update steps and simplifies the generation of valid paths without greatly adversely affecting performance.
- We assume that every path generated by the RRT is traversable regardless of the speed at which we are traveling. This helps simplify our path generation as the RRT can ignore speed as a variable. This is accomplished by constraining our motion update to the lower bound of our steering capabilities at speed.
- We assume from previous experience that our ZED camera will give us data that is outdated by half a second or more and also has a slow update time of 5HZ or so. Therefore, we assume that we will have to use tf's time travel features to allow our sensor processing suite to work correctly.
- We also assume that our high level planner has a reasonably high probability of latency. Therefore, lower level path following algorithms should be able to "fast-forward" or ignore path data that is outdated or already traversed.

III. Software Architecture

Our software architecture divides units of functionality into separate ROS nodes, maintaining reasonable computational efficiency while respecting division of labor.

At the lowest level, ROS wrappers run the drivers to interact with the hardware, including the laser scanner, VESC, and ZED stereo camera. These wrappers were developed by the hardware providers or the staff.

On top of this, AMCL runs to provide higher level nodes information about the robot's pose. Most of the abstract behaviors require this, as they are coordinating the robot's interaction with the world.

The next level of functionality includes nodes which pass messages to and from the driver nodes and access world transformation information to provide an interface for more advanced behavior to the higher level abstract nodes. The path follower node converts a sequence of pose goals in the environment and information about the robot's location to the requisite sequence of inputs to the ackermann command mux to allow the robot to achieve these poses. To convert ad-hoc

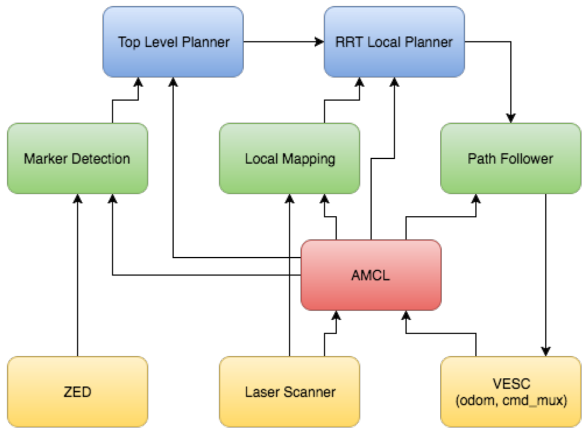


Fig. 1: Information flow in the ROS architecture

[1]Massachusetts Institute of Technology, EECS
[2]University of Cambridge

obstacle information to a usable form, the local mapping node converts the laser scan and information about the robot's location to occupancy information in the world frame. The marker detection node uses the ZED camera's input to calculate the relative location of markers.

Above these nodes, the RRT local planner takes occupancy information from the global and local maps as well as the robot's location and produces a sequence of pose goals for the path follower to use to move the robot to the target location. The top-level planner takes information about the robot's location and marker location to produce a sequence of local goals for the RRT local planner to achieve.

IV. Technical approach

A. Vision (Ernie)

The vision task is using camera to recognize the orange triangle cone and green rectangle markers in front of the robot. Its optional to recognize people and other robots as well. There are several problems while attacking this problem. One of the main problems is that the color of obstacles might change depending on different environments. For example, the light green might turn into dark green if the light source is not enough. Therefore, instead of setting the RGB range value manually, we collect many obstacles' pixel color combinations in real life by recording the obstacles under different brightness. Fig. 2 shows the set of pixel colors recorded from the marker. In fact, it was what we did in lab4- extract the obstacles pixel color under different brightness and look it up while the new image comes. Here are the details about how we implement this program. First, we record the bag of obstacles. Second, we extract the obstacles pixels color into arrays. Third, when new image is captured, we scan the image and look up whether there is any pixel with same color in the arrays. If so, we return where the pixels are and locate them as obstacles. After knowing the position of obstacles in the images, we calculate the real world relative position from the car to the obstacles base on the information from zed camerainfo, which includes information P matrix ,the multiplication of intrinsic and extrinsic matrix. After multipying the P matrix with the x, y obstacles' pixel coordinates in the images, we can get the x(rightward), y(upward), z(forward) distance value from the car to the obstacles. Then we can wrap these position information into messages, and publish them into driver node. Moreover, we also can use NVIDIA visionworks to accelerate the obstacle recognition or learning approach to recognize obstacles.

B. Localization and Mapping (Steven)

In order to understand the environment, the robot must both know where it is in the environment, and the occupancy of the relevant parts of the environment. The laser scanner data in conjunction with the odometry estimates computed by the motor controller should provide the required information to infer these.

In the environment of the challenge, most of the surroundings is known prior to the run, while there are some obstacles

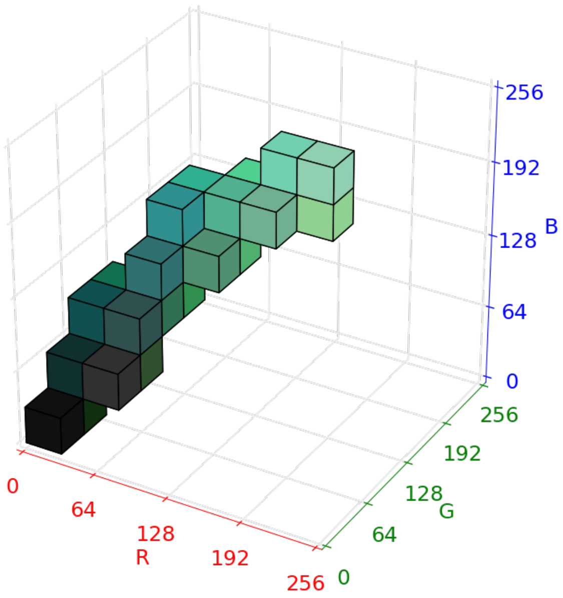
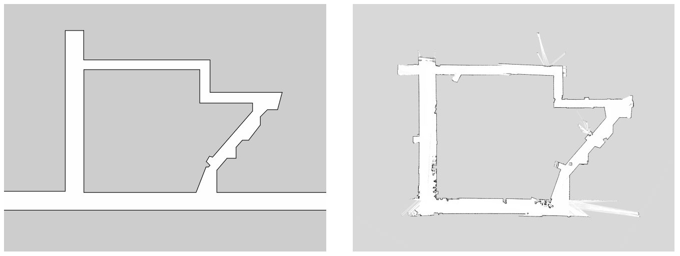


Fig. 2: A plot showing the regions of RGB color-space which are considered marker-like



(a) From simulation. Notice how few features this map provides for AMCL scan matching.

(b) From `hector_slam` using added environment features. Notice the noise and slight loop closure issues.

Fig. 3: Maps of the tunnels

which are not known until detected online. Thus, the task of understanding the environment contains some elements of simple localization, and some elements of mapping. This means that a localization algorithm is not sufficient for the task, while a SLAM algorithm is unnecessarily powerful.

We will use AMCL to localize with respect to the global map (see Fig. 3 for an example map). In order to understand the ad-hoc obstacles, a local occupancy grid relative to the well-localized robot is created. Laser scan ranges are projected into the global frame and the resulting occupancy grid is published, and the planner efficiently integrates the local and global maps to use for planning. To avoid the computational cost of ray-tracing and to ensure that the known walls are in the correct place, the tips of the scans only are treated as occupied and overlaid in the map.

AMCL is a ROS c implementation of particle filter local-

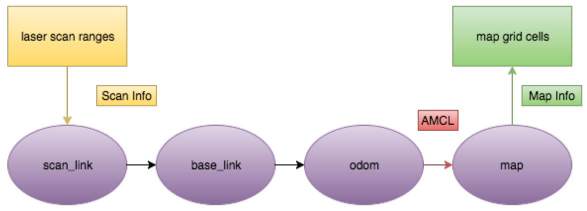
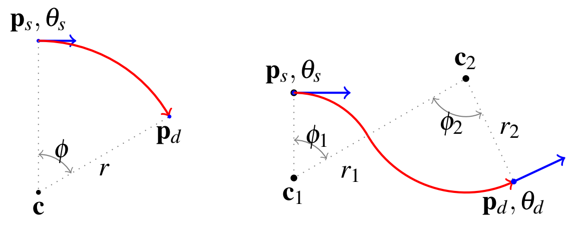


Fig. 4: The transformations along the tree from scan to map indices are precomputed and efficiently applied using numpy arrays.



(a) Circular arcs (b) S-shaped curves

Fig. 5: Possible choices for RRT edges (red) between two states (blue), and their parameterizations.

ization. With input from the odometry published by the VESC as well as the laser scanner, iterative updates are applied using scan-matching on the pre-computed map of the tunnels to estimate the robot's pose. The node publishes the estimate as a tf transform from the map frame to the odometry frame, thus acting as a periodically updated correction to the imperfect odometry. The tf tree then allows the local mapping node to calculate the transformation from the scanner frame to the map frame through odometry, and behaves reasonably even when the odometry updates more frequently than the particle filter.

In order to efficiently implement local mapping, the transformation from the laser frame to the global map indices is precomputed with information from tf and the map's resolution information. This transformation is then applied to the laser scan points with numpy vectorization. The occupancy grid is limited to the points which lie in a small square around the robot's location in order to reduce the cost of publishing the map, and the local map grid squares are aligned to those of the global map for rapid integration to the global map on the planner's side.

C. *Low-level Path planning (Eric)*

To find obstacle-free paths between the car's location, and its goal, we use a Rapidly-expanding Random Tree (RRT) algorithm. As this algorithm is probabilistically complete, it is guaranteed to (eventually) find a path if one exists. It works by iteratively builds a tree of possible paths, on each iteration sampling a random point and connecting it to the nearest point on the tree.

It needs to be ensured that all the paths generated by the RRT are feasible, i.e. fall within the constraints imposed by an Ackermann-steered car. This is equivalent to stating that any path through the tree must be decomposable into a series of tangentially-smooth circular arcs, all of which have a radius that must be greater than the minimum turning radius of the car[1].

With this in mind, there are two good candidates for edge representations in the RRT: circular arcs, and pairs of arcs joined into S-shaped curves, as in Fig. 5. In both cases, our tree nodes are poses of x, y, θ. Circular arcs are able to uniquely connect a pose $\mathbf{p}_s, \theta_s$ to a point $\mathbf{p}_d$, whereas an s-curve is able to connect a pair of poses, $\mathbf{p}_s, \theta_s$ and $\mathbf{p}_d, \theta_d$. Note however, that even after this constraint, the S-curve has a remaining degree of freedom.

[1]determined experimentally to be roughly $r_{\min} = 0.6$m

Due to time constraints, we chose to use the simpler circular arcs as our edges. It's worth noting this is not suitable for an RRT* algorithm, since that requires us to find paths between two poses when rerouting, not just a pose and a point. This choice also removes our ability to specify a goal orientation from the RRT.

Having chosen our edge type, we want to be able compute all of the properties shown in Fig. 5a. We start off by finding a direction vector pointing left, which allows the center of the circle to be expressed:

$$\begin{aligned} \text{let } \mathbf{v}_{\text{left}} &= \begin{bmatrix} -\sin\theta_p \\ \cos\theta_p \end{bmatrix} \\ \implies \mathbf{c} &= \mathbf{p}_s + r\mathbf{v}_{\text{left}} \end{aligned}$$

Noting that $\mathbf{p}_s$ and $\mathbf{p}_d$ lie on the same circle,

$$\begin{aligned} \|\mathbf{p}_s - \mathbf{c}\| &= \|\mathbf{p}_d - \mathbf{c}\| \\ \implies r^2 &= \|\mathbf{p}_d - \mathbf{p}_s - r\mathbf{v}_{\text{left}}\|^2 \\ &= \|\mathbf{p}_d - \mathbf{p}_s\|^2 - 2r\mathbf{v}_{\text{left}} \cdot (\mathbf{p}_d - \mathbf{p}_s) + r^2 \\ \implies r &= \frac{1}{2}\frac{\|\mathbf{p}_d - \mathbf{p}_s\|^2}{(\mathbf{p}_d - \mathbf{p}_s)\cdot\mathbf{v}_{\text{left}}} \end{aligned}$$

where it is worth noting that r is signed, with positive values indicating that the circle's center is to the left. This allows us to find $\mathbf{c}$, from which the signed angle ϕ can be found,

$$\phi = \operatorname{arctan2}\left((\mathbf{p}_s - \mathbf{c}) \times (\mathbf{p}_d - \mathbf{c}), (\mathbf{p}_s - \mathbf{c}) \cdot (\mathbf{p}_d - \mathbf{c})\right)$$

Where "$\times$" here is the 2D generalization of the vector cross product, yielding a scalar. Finally, this allows us to calculate two important properties:

$$\begin{aligned} \texttt{distance}(\mathbf{p}_s, \theta_s, \mathbf{p}_d) &= \begin{cases} |\phi r| & \text{if } |r| > r_{\min} \\ \infty & \text{otherwise} \end{cases} \\ \theta_d &= \theta_s + \phi \end{aligned}$$

The first of these is the path length of the arc, our distance metric for use in the RRT. The second is the resulting robot orientation after navigating the arc, which we need to compute to convert our destination point into a pose before appending it to the tree.

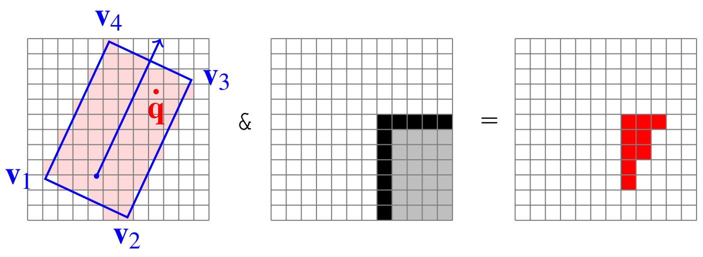


Fig. 7: Rasterizing the car footprint for collision detection

Another task that the RRT must perform is collision detections. One approach would be to transform the map from world space to task space. This approach suffers from discretization errors, and processing time, if it must be recomputed every time the map updates. Instead, we do collision detection on a rectangular model of the robot as in Fig. 7, by cropping the map to the bounding box of the robot, and creating a mask wherea a pixel $\mathbf{q}$ is shaded iff $(\mathbf{q}-\mathbf{v}_i)\times(\mathbf{v}_{i+1}-\mathbf{v}_i)\leq 0\ \forall i$.

This mask can then be compared against the underlying map, and any collision detected.

One final concern with the RRT is efficiency. We used the builtin python `cProfile` module along with a third-party visualizer `snakeviz`[2] to find bottlenecks in our implementation, and then used `numpy`'s vectorization to speed up these sections of code. To store the tree, we required a custom implementation of an arraylist, a sequence data structure that reallocates memory whenever it comes close to filling up, backed by a `np.ndarray`. This required very careful memory management, so as not to leave pointers to old data dangling when the buffer is reallocated.

Fig. 6 shows some visualizations of paths found by the RRT. For testing, we used the navigation tools built into `rviz` to feed the RRT with start and end points.

D. High-level Path planning (Shi-Ke)

At a higher level, path selection is determined not just by distance but by the speed it takes to traverse the path. Although cones and markers each penalize or reward time, it is not necessarily the correct strategy to always path to avoid cones or to chase markers. A higher level strategy needs to take into consideration the trade-offs for avoiding or approaching the two targets in terms of time. With the updated race design, plowing through cones is no longer a correct strategy - however, the same consideration still needs to be applied to markers.

It's possible that a cone is detected too late to route around without significant time loss, such as if one is spotted right as the racecar turns a corner. Similarly, it's possible for markers to be detected too late to feasibly change route to cross over them. As a result, the high level path planner must weigh the costs of missing a marker or hitting a cone against the cost of decelerating to pass over a marker or avoiding a cone. Of course, this cannot be applied to all obstacles - the car should avoid all other obstacles whenever possible.

Other than this consideration, the high level planner just needs to generally keep track of the overall path that needs to be followed by the local planner and ensure that the local planner continues as expected. If the local planner becomes lost, then information is gathered from the LaserScan and the robot attempts to continue towards open space in the same general direciton.

E. Path following (Winter)

To optimize the speed at which our car runs the course, we would like our path follower control system to be tightly connected to our higher level RRT path planner. To do this, it should try its best to follow the exact path the RRT outputs and also output signals describing the robot execution state.

Since our RRT will only output paths that are feasible for our control dynamics, the technical specifications of the path follower are clear. Our path follower must be able to follow provided paths with the highest degree of accuracy possible, while also intelligently dealing with localization error and upstream path-planning latency.

We've already proven in simulation tests that our path follower tracks low resolution tracks to an acceptable degree of accuracy. We have confirmed that the previous iteration of the path follower algorithm can guide a robot around a simulation track. However, our current outstanding concerns are with the handling of real-world errors.

Since we are using AMCL for our localization in the real-world, we expect that our robot pose belief to not be continuous. That is, we expect that AMCL will update our

[2]https://jiffyclub.github.io/snakeviz/

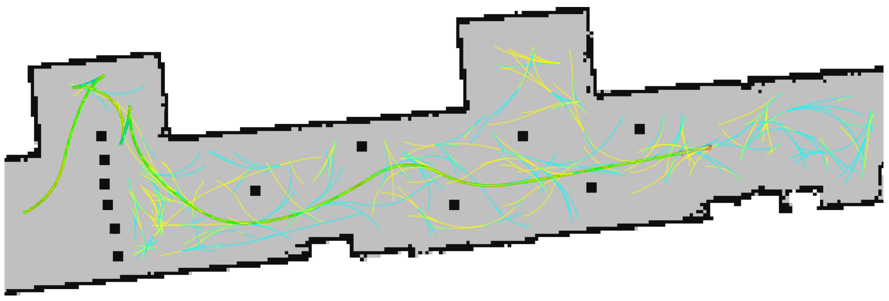
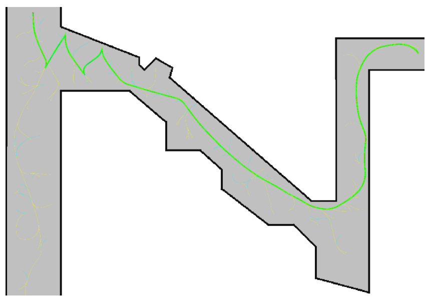



Fig. 6: Test runs from the RRT. Green is the final path, yellow edges explored forwards, and cyan the edges explored reversing. Note the suboptimal reversing path found on the right - a basic RRT makes no optimality guarantees.

robot pose based off of laser scan matching in an unpredictable way. This means that our path follower should be able to handle the robot pose unexpectedly jumping around the map in an intelligent manner.

Put simply, there are 2 cases we need to handle: either the robot has jumped off the path, or has jumped back on the track. In the case where the robot has jumped off the track, the path follower should keep the target waypoint constant until the robot converges on the path. This triggers the second condition.

In the second condition, the robot has reentered the path, but may be further behind where we previously expected it to be. To fix the issue, we "fast-forward" our playback of the path until we find a waypoint that the robot has not yet moved to. Then playback of the path resumes in the usual way.

As an added benefit of our "fast-forwarding" error correction method, we can handle latency upstream in a graceful manner. For example, let the robot be in motion executing a path. Then, a new path (perhaps to hit a newly detected marker) comes in from the RRT. Since the robot is in motion, the robot is well past the start point of the path. Therefore, to correct the situation, we simply "fast-forward" path playback until we find a waypoint that is in an unexecuted portion of the path. This method allows us to dynamically handle path changes while also considering for real-world factors such as planning latency and localization error.

V. Capability milestones

May 4	Dry run - Navigate the course with the RRT path planner to move toward open space. [3]
May 7	Navigate the course using localization and the RRT planner; detect markers.
May 9	Navigate the course with obstacles, using high-level planner to pass over markers.

VI. Decision making process

The general decision making process for both how to divide work and for our implementation for the final project has been to communicate important decisions during team meetings. General consensus or at least agreement can be reached on major decisions - minor issues can be talked about online through Slack.

Technical decisions were motivated by a balance between effectiveness and ease of implementation. Given the goal that the robot should be robust even to difficult obstacle configurations which would likely not appear in the challenge, we decided to focus on giving the robot a strong ability to plan and designed the rest of the infrastructure to provide the necessary abstractions for this as efficiently and simply as possible.

[3]Partially completed. On May 4th, we were able to use the RRT planner to traverse short paths while avoiding obstacles. However, work still needs to be done to the path-follower node since execution of tough-to-traverse paths would sometimes 'hang'.

Division of labor

In order to keep track of which tasks are assigned to which people, we will use github issues. This allows us to label tasks by the technical topic they pertain to, so we can see which groups of tasks are falling behind and need more people.

The current mapping of tasks to people is:

Eric:

- Improve performance of the existing RRT code.
- Augment the RRT code to allow replanning from an existing tree.

Ernie:

- Detect the cones, markers.

Shi-ke:

- Implementation of RRT*.

Steven:

- Optimize AMCL performance on-robot.
- Implement local-grid mapping.

Winter:

- Upgrade path follower code to be robust in the face of real-world errors and latency.
- Implement a high-level course planner such that emits a series of high-level goals for the RRT to traverse towards. This node is higher level than the cone detector node.

VII. Self-assessments

Steven

Technical: Over the course of the labs, I gained an appreciation for the computational constraints that arise when implementing things. Thinking about algorithms is something I have a lot of experience with, but figuring out how to speed up constants when actually writing code to execute them was a new experience. I also learned more about managing my code while working in a team.

Communication and Collaboration: The most impactful thing I learned in terms of communication was how to effectively convey information with slides in a presentation. I didn't have much experience with genuinely trying to get people to understand something technical through a presentation, so the tips about formatting slides were quite informative.

Winter

Technical: During 6.141, I learned how to work on complex software architectures as part of a team. Although working as a team to develop software architectures sounds easy, it involves a lot of intercommunication, specifications, and system design something that I did not have a lot of experience in. Additionally, I also learned how to use ROS for the first time.

Communication and Collaboration: Building a robot with a team requires a lot of team communication about system design, tradeoffs, timelines, and feasibility. In 6.141, I learned how to improve my communication such that I could integrate needs from my teammates into my work while also communicating my needs to the team.

Shi-Ke

Technical: The bulk of the technical difficulty has been adapting to new architectures such as ROS. On top of learning new workflows for working with ROS, the algorithms taught in class have been a great experience both to learn and to implement in the labs.

Communication and Collaboration: Having taken 6.UAT, I felt pretty comfortable with individual presentations but I have not had much experience with group presentations, which are very different. 6.141 contributed greatly to my experience doing these types of presentations. Additionally, this was the first group that I had used Slack with, which was a valuable collaboration experience as well.

Ernie

Technical: 6.141 gave me a chance to integrate the knowledge of hardware and software and build the complex system cooperatively which I seldom experienced before. I realize how important to document the code and organize the issues to work as a team.

Communication and Collaboration: I learnt some basic concept of project management before but didn't find a good tool before. In 6.141, we use slack and github as communication tool and I think they are the best combination communication tools which help me keep track not only the conversation but also the progress.

Eric

Technical: 6.141 has introduced me to the ROS infrastructure, and taught me alternate ways of structuring complex systems. It's provided me with great opportunities to work on the implementation of algorithms I've learnt about in previous classes. Having to profile and optimize code is a learning experience for me, as it's something I've not had to think about before. It's also allowed me to contribute to open-source projects at all depths of the stack, from the zed drivers, through the ROS codebase, and even up to numpy itself. On the flipside, it's reduce my confidence in open source software working perfectly all the time!

Communication and Collaboration: It's been a while since I've had to do a group presentation, so this has been valuable to me. 6.141 has also acted as a very effective trial run of Slack, a tool I previously dismissed as not solving a problem that I had. I now consider it part of my toolbox, and will aim to use it in future group projects!