SSH IPRO Report 2020 - Software

Our software development efforts this year can be broken up into 3 main categories Control, Autonomy, and Sensing by order of lines of code required.

Control

In the robot this entails control over all the moving parts and communication between the robot over the network for manual control and data logging during competition.

System/Hardware Design Considerations

With hindsight we helped to revise our system design. Previously we had several components that added an undesirable amount of complexity that we decided weren't needed or could be replaced with simpler solutions or software alternatives.

Diagram 1: 2019

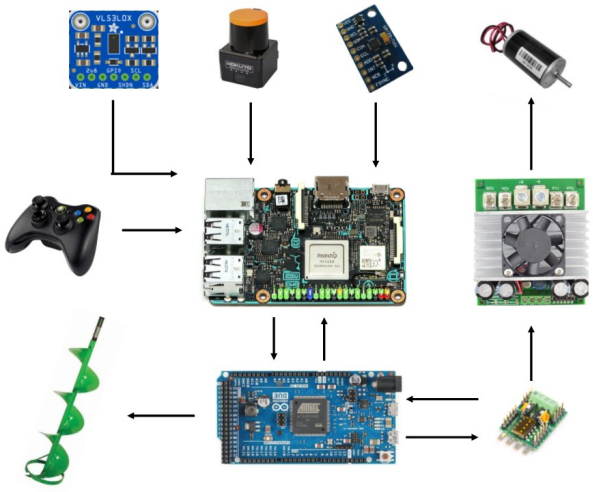
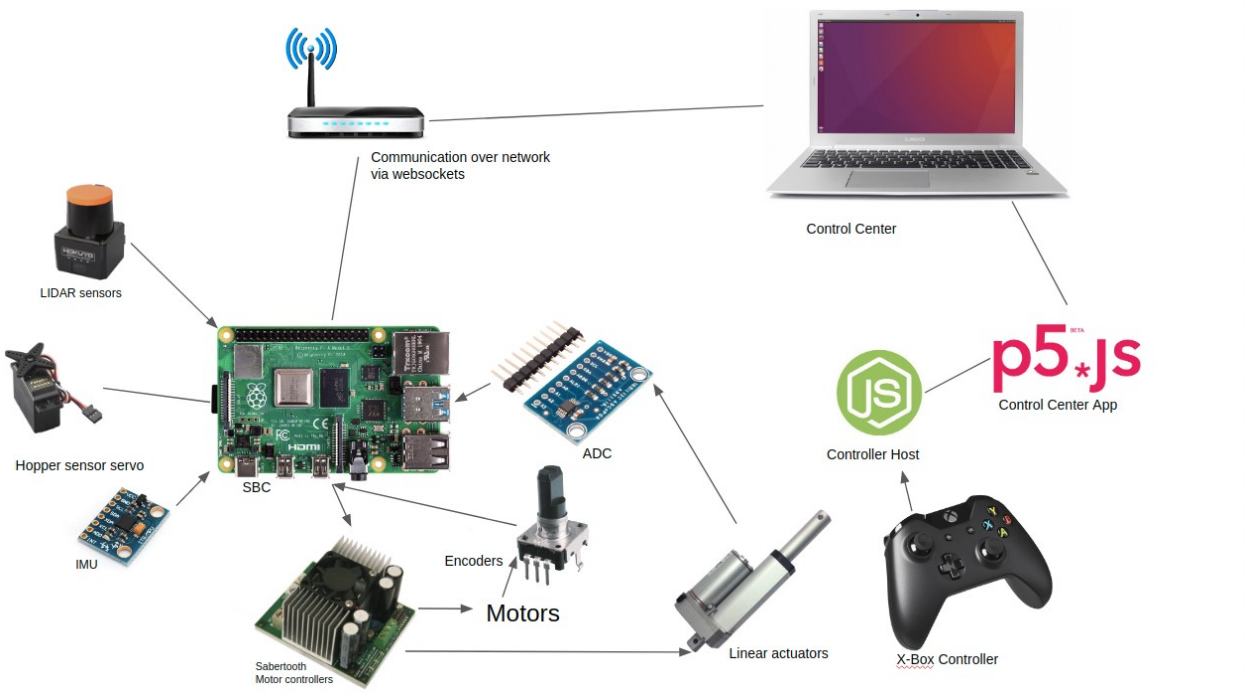


Diagram 2: 2020



Motors Subsystem

In previous years, we used an arduino running a custom communications protocol to interface with the kangaroo hardware PID controllers (Kangaroos) that would send commands to the sabertooth motor controllers. This system was, in addition to being needlessly complex and power-hungry, prone to error as the Kangaroos were poorly documented, had undesirable behaviors, and despite their high prices were quite delicate. This year we eliminated the motors subsystem by interacting directly with the motor controllers from the central SBC on the robot. Further there is no longer any confusion over which “brain” handles what functionality with the removal of the Arduino.

Remote C&C

During competition, the robot must be able to be controlled remotely over the competition network. In some ways this was overlooked in previous years, with our final version simply plugging the controller into the robot. Although there were some rough prototype GUI applications written in C++ with the QT framework the pace of development was too slow and there were no plans for communication protocols. This year, we used javascript for our frontend in order to finish on time and have a more flexible system.

Hardware Interface

This involved creating and using libraries that interact with the hardware we’re using to control the robot and creating higher level abstractions that we can use elsewhere in the software.

Motor Controls

We created modules for controlling all the moving parts of the robot with the highest level of abstraction being contained in motors.py. To control The CIM motors and linear actuators we used the pysabertooth library that interacts with the motor controllers over serial. We are currently not using encoders, but adding them should be trivial. We read the potentiometer that gives us the position of the linear actuators using an i2c analog to digital converter

LIDAR

The Hokuyo UST-10 LX 2d LIDAR uses an ethernet connection to connect to the Raspberry Pi SBC, however because the Raspberry Pi only has one ethernet port, one is connected via a USB-to-ethernet adapter. We are using the python libraries provided by the LIDAR manufacturer.

Control Center

In previous years we tried to make a monolithic desktop application using a GUI framework like QT in a compiled language like C++. However the development process was too slow among other things. This year we took advantage of modern JavaScript and web frameworks to make a two part solution.

Frontend/GUI

The majority of the code is a static HTML, CSS and JS webpage that interacts with the robot and other systems via websocket connections. We used a web framework called P5.js to simplify creation of visuals. The frontend displays a variety of sensor readings and processed values, for example it provides us with cartography data and localization so that we know where the robot is in the arena and where it has detected obstacles without using the camera which (improves our score). The frontend also gives us some control over the robot and the ability to “supervise” its autonomous operations and make adjustments on the fly if needed. There is also some limited logic in the frontend that handles things like scaling values and converting to a simple instruction set for transmission over the websocket. Finally the control center facilitates remote manual control over the competitions network infrastructure.
Additional Documentation:
https://github.com/SpaceHawks/robot2020/blob/master/docs/specs/control_center_spec.md

Xbox server

Because our frontend GUI is running within the browser, it doesn’t have direct hardware access. To get around this, we created a simple server that posts data from a usb connected xbox controller to a websocket. The frontend can read this information as it comes in or it can make api requests to endpoints.

Robot Main Program

The robot main program reads instructions coming in from the frontend over a websocket and reacts accordingly by changing control systems, responding with sensor readings, or performing specified actions. The instruction set is very simple and easy to interpret. In addition to receiving instructions the robot also sends data back to the frontend, for example debug messages and position updates. The main program is designed to be a modular system so that different autonomous instruction sets can be built quickly and easily. This allows for easy testing, as more complex functions like pathing and fine motor and actuator control are already built into main commands. The commands in the main program are designed to receive data from sensors on the robot so that different autonomous protocols can be adapted to the environment with the need for a change in the base code. Below is a list of some autonomous and nonautonomous instructions and their arguments.

Name	Data	Example
Message	M:message	M:Hello There
Obstacle Point	O:x,y, ...	O:30,23,45,60
Robot Point	R:x,y,θ	R:3,2,45
Arcade Drive	AD:throttle,turn	AD:85,40
Tank Drive	TD:left,right	TD:80,30
Autonomous	AI	AI
STOP	STOP	STOP
Read Encoder	ENC	ENC
Set Servo Angle	SER:angle	SER:90
Deploy Trench Digger	DEPLOY:position,speed	DEPLOY:1,0.5
Dump Hopper	DUMP:position1,position2	DUMP:1,0

Sensing and Autonomy

Of course our final goal was to have a fully or nearly fully autonomous robot in order to achieve the highest score in competition. In order to do this we needed to create control systems that utilize sensors for autonomy.

Pathing

The key component to a fully autonomous robot is the ability for the robot to navigate from one destination to another. During the competition, the robot will have to navigate around at least 5 boulders, ranging in diameter from 30 to 50 centimeters. Due to this, the robot needs obstacle avoidance capabilities.

The first concept we explored was just that– pure obstacle avoidance. This means that the robot would not have to keep track of any kind of internal state (as it would with pathfinding, discussed later), and instead would rely only on the current sensor inputs to determine in which direction to move. Due to the LIDAR's capability of scanning 270 degrees, it seemed natural to use this ability to not only detect an obstacle in our immediate path, but also to choose the best detour available.

The algorithm is very simple:

```
if can_move_at(0): # If we can move at 0°, do
    turn_to_angle(0)
    go_forward()

elif can_move_at(current_angle): # If we can move in the same direction, do
    go_forward()

else: # Otherwise find the closest available angle
    a = find_available_angle()
    turn_to_angle(a)
    go_forward()
```

The algorithm prioritizes going straight forward – that is, moving in a direction perpendicular to the wall. This acts as a sort of heuristic to ensure the robot reaches the other side as fast as possible. If this fails, the robot prioritizes turning as little as possible, as turning often would not be optimal behavior.

This turned out to work very well in simulations for most randomly generated possible maps. However, there were a few caveats to this system:

1. **The algorithm does not take into account the collisions while turning.**
   While the robot will never move forward into a wall, it is possible that it will *turn* into an obstacle. This is because the algorithm does not consider whether the back half will collide while turning to the target angle, as demonstrated in the figure below:

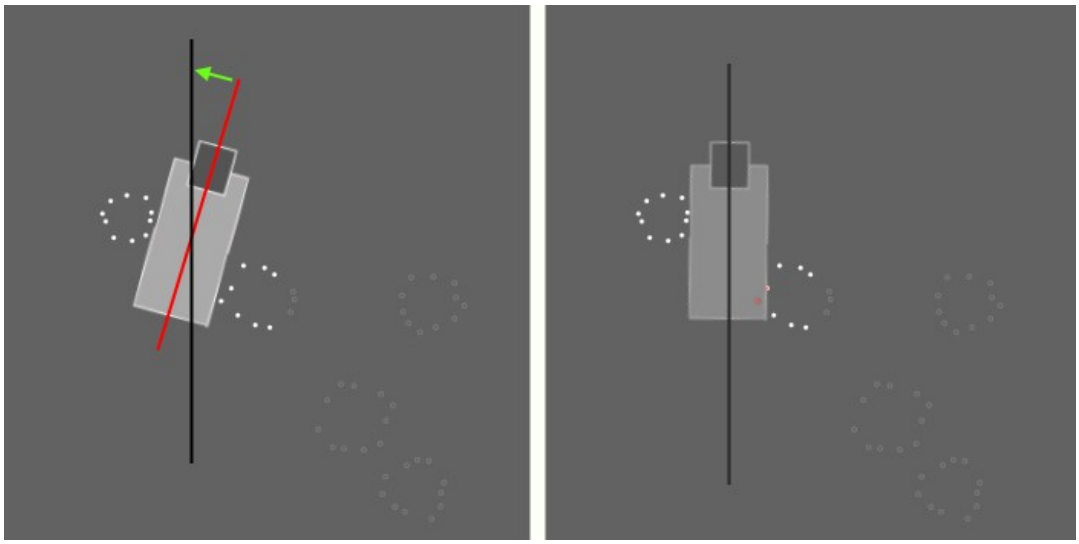


Figure ???
Example of robot erroneously turning into obstacle

The robot will turn straight ahead, as it believes the path to be clear. However, it has lost sight of the obstacle it passed, causing its back half to collide with the obstacle. In our simulation, this occurs fairly often, and a sufficient fix has not been found.

2. **The robot can become trapped in a loop**
   This can occur if there is a "pocket" of obstacles. The robot will reach a dead end, turn around, and begin to navigate out. However, eventually it will come far enough out that it once again tries to move at 0° again. This, however, is incredibly rare as the pocket must be very deep in order to fully trap the robot. Most of the time, the robot manages to navigate out of the pocket without entering a loop.

3. **The algorithm performs suboptimally**
   Due to the sole goal of the algorithm being to not hit an obstacle, the robot will not arrive at the mining area as quickly as possible. However, we believe the efficiency is very reasonable for the simplicity and overall reliability of the algorithm.

It should be noted that the first two issues can definitely be fixed. Both cases could be solved by keeping track of the obstacles and adding more control logic to the algorithm. If these were fixed, this algorithm would be a very strong contender, as it takes very little power computationally, allowing computing power to be used for the other functionalities the robot needs to perform. Although we could not test it this season, it would be well worth it to implement this algorithm and test it on the actual robot once it is finished. For a web demo of this algorithm, visit this webpage.

We also decided to attempt to implement a pathfinding algorithm, which should provide closer to optimal performance.

In the simplest case, we could implement a pathfinding algorithm which simply operates on a 2D plane. This would, of course, only take into account the X and Y position of the robot. However, an issue arises when you consider the assumptions that most pathfinding algorithms make about the navigator – that it is a single point. At 1 meter wide and 1.5 meters tall, our robot is most certainly not a point, so this assumption fails. How can we implement a pathfinding algorithm, such as A*, if the navigator is not a point?

The important connection to make to allow us to still use pathfinding algorithms is to notice that while the robot has to be considered as a point, obstacles do *not*. Thus, we can expand the obstacles by the dimensions of the robot, effectively shrinking our robot to a point. In the figure below, our robot can be represented in the algorithm as the green square, a single point:

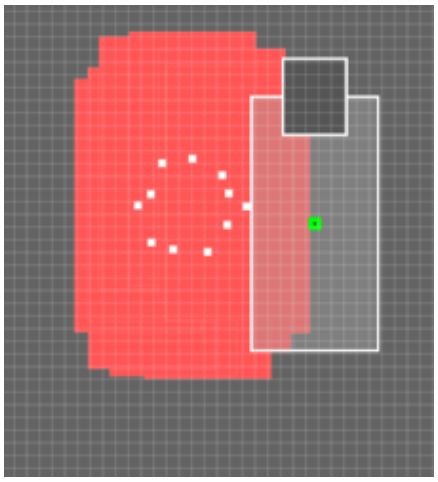


Figure ???
Obstacle expanded at 0°

However, this still is insufficient in modeling the map. Yet another issue is encountered due to the rotation of the robot. Consider a robot that is moving at a 90° angle. The obstacles would now need to be expanded like this:

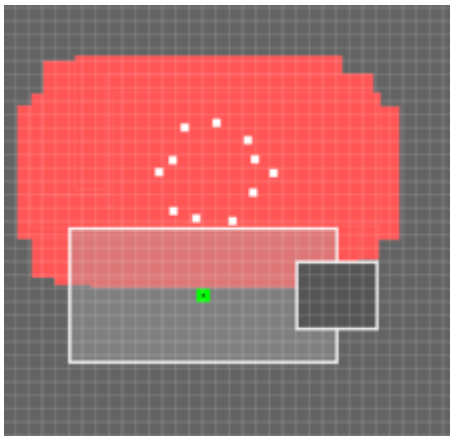


Figure ???
Obstacle expanded at 90°

This is just the robot tilted 90°, centered at each obstacle point. In fact, this is the general relationship between an obstacle and the robot – simply move the robot's center to the obstacle (maintaining the orientation of the robot), and that will be the new, expanded obstacle.

Thus, the X and Y locations of the expanded obstacle points will change depending on the rotation of the robot. This means that in order to effectively model the map, we must introduce a third dimension, θ. If you were to view this third dimension, it would look like this:

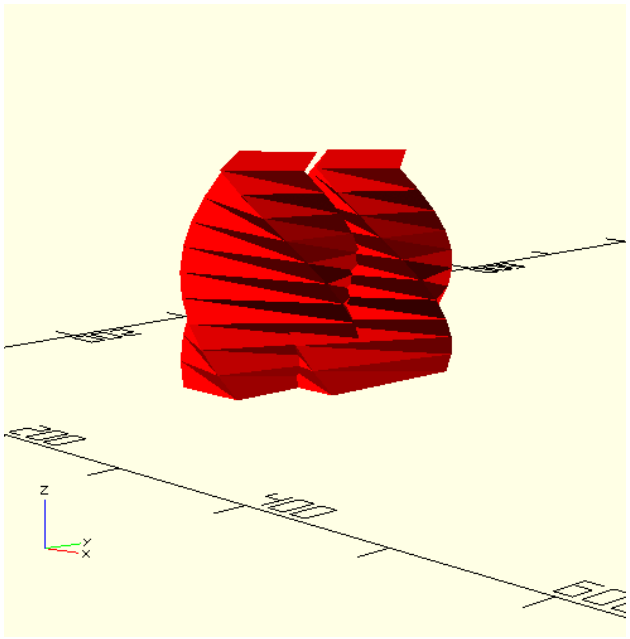


Figure ???
Two obstacles expanded in 3 dimensions

One can see that at 0° (on the XY plane) passing through the obstacles is impossible. However, at the terminating angle, there is a small gap, implying the robot can pass through the objects at this angle. To obtain a better understanding of this 3 dimensional mapping, take a look at this web demo. When you press the right or left arrow key, you move through the 3rd dimension.

We now have a goal for our pathfinding algorithm: Find a continuous, three-dimensional path such that the robot arrives at the destination in an optimal amount of time, without intersecting any obstacles. The last caveat to this is finding the right heuristic. The heuristic assigns a numerical value to each node it could travel down, with a lower score being better. An inefficient heuristic can result in very, very slow processing times – our first heuristic took 40 seconds to find a path! Our current heuristic now only takes about a tenth of a second, on average, to find a path.

The current heuristic assigns a value of the Manhattan distance from (x, y, θ) to (goalX, goalY, 0). This ensures that the algorithm prioritizes paths that are straight forward toward the goal, and also that it stays away from the side walls as much as possible.
We chose A* as the pathfinding algorithm of choice, as it is widely documented and easy to implement. However, there is a huge factor that has been left out in the discussion of pathfinding thus far – the robot has no prior knowledge of obstacle location. The implication of

this is that when the robot encounters a new obstacle, it must reconsider its path. This *could* require too much computing power, as the robot will continuously have to repath.

There's no way to know if this will be an issue without testing, which we unfortunately were able to perform this year. However, once this is tested and efficiency is determined, we can judge whether or not a different pathfinding algorithm will be required. There are available algorithms that replan by adjusting previously calculated values, instead of restarting the path calculation like A* does. This would lead to much more efficiency, if implemented correctly. These are known as incremental heuristic search algorithms, and the two most popular ones that came up in our research were D* Lite and Lifelong Planning A*. Unlike regular A*, these have very little documentation and thus would be harder to implement correctly.

Finally, a demo of our A* pathfinding algorithm is available by cloning our repo and running the test_astar.py file.

Next year, both of these equally viable approaches to pathing will need to be considered, along with any new ideas, to determine the most appropriate, efficient, and reliable way to navigate through the terrain autonomously.

Localization Via LIDAR

Although we were able to make changes to our localization design to improve accuracy, some pieces of localization methodology remained unchanged between the Spring 2019 and 2020 IPROs. Namely, our target design (including both concept of stripes and materials used for those stripes) remains unchanged as well as our method for choosing our data points relative to the target and calculating robot position and orientation. Figure ??? shows a summary of our old method for calculating position and orientation. For more information, see the Spring 2019 IPRO Programming Report.

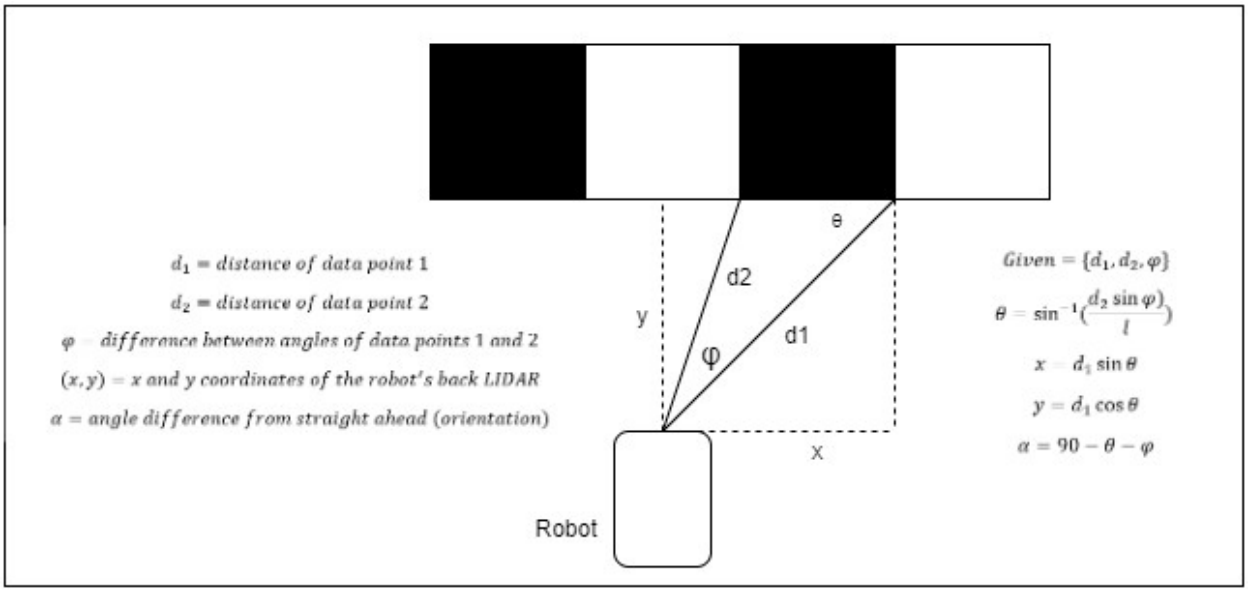


Figure ???
An overview of last year's localization solution.

During the Spring 2020 IPRO, the software sub-team made changes to LIDAR localization to increase the accuracy of position calculation by using more data points in our calculations and taking the mean of those results. This is achieved by adding more stripes to our target, as the number of alternating stripes is directly proportional to the number of data-points our target will give. With n alternating stripes, we have n-1 data points. By doing calculations on using several of these points, we will have more results for position, which we can then average for a more accurate and precise measurement.

Figure ??? shows the difference between 2019 and 2020 target design.

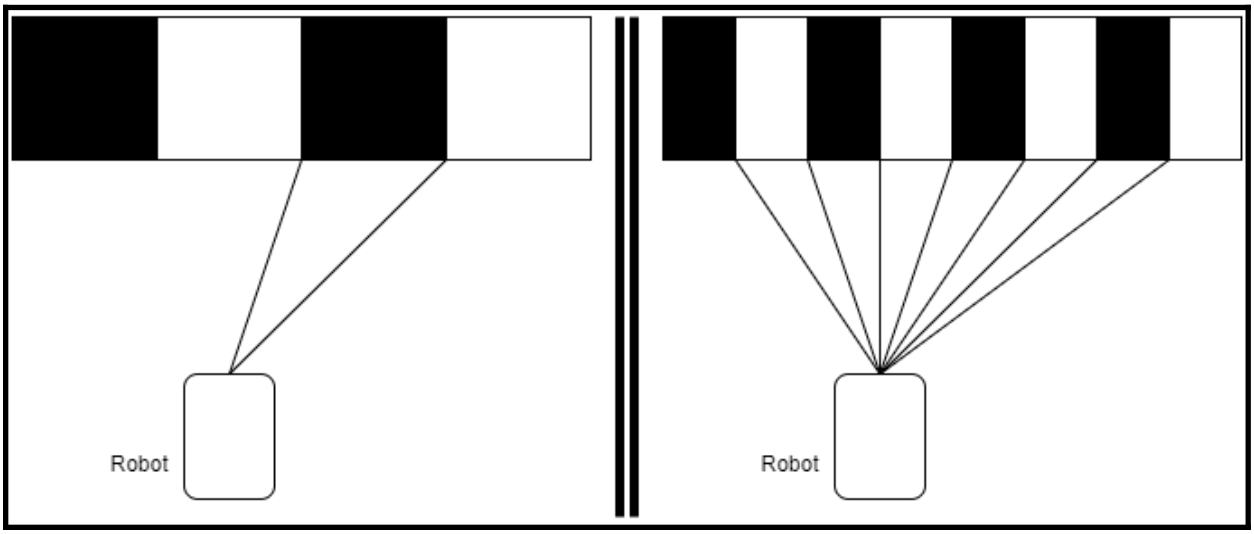


Figure ???
On the left is the data point selection for Spring 2019 IPRO Software. Only one pair of data points is used, leaving us entirely at the mercy of the sensor's accuracy. On the right, Spring 2020 IPRO software design uses many points and averages their results, allowing us to combat hardware limitations of accuracy.

By using several pairs of data points instead of just one, this year's software team overcame the problem of accuracy. Hardware limitations cause the LIDAR to give inconsistent readings. By averaging the results of several points' position calculations, our results for position and orientation are now much more accurate.

IMU

We created a library that interfaces with the IMU hardware based on the documentation provided by the manufacturer. We discovered that although the data from the sensor is easy to access, it's not very accurate and needs to be combined with a kalman filter to reduce the error rate.

Kalman Filter

Although we plan to continue to improve the LIDAR and IMU to obtain as much accuracy, speed, and reliability as possible, there will always be some associated error and noise in our readings, as well as delays between new available readings, especially from the LIDAR.

To combat this, we implemented a Kalman Filter to produce a more reliable output from our sensors. The full inner-workings of the Kalman Filter and the complete implementation are not detailed in this report, but instead can be found here:
https://github.com/SpaceHawks/robot2020/blob/master/docs/documentation/kalman.md

However, the overall concept of a Kalman Filter isn't too difficult to grasp, and a simplified explanation will be presented below.

We are interested in finding the most accurate **x**, **y**, and **θ** values as possible. The LIDAR gives us these, but due to the angular resolution and distance reading fluctuation, this comes with inherent error. This leads to the need to average multiple runs, as discussed in the **Localization Via LIDAR** section. Thus, the LIDAR is relatively accurate, but slow to collect readings from. Each scan takes 25ms, so even with only 5 scans to average, this already allows us only 8 readings/second.

On the other hand, the IMU is very cheap to access, but only provides acceleration and angular velocity data. One could use this directly to obtain a rough idea of the change in position of the robot, but that comes with a huge amount of error and thus wouldn't be very helpful on its own.

This is where the Kalman Filter comes in.

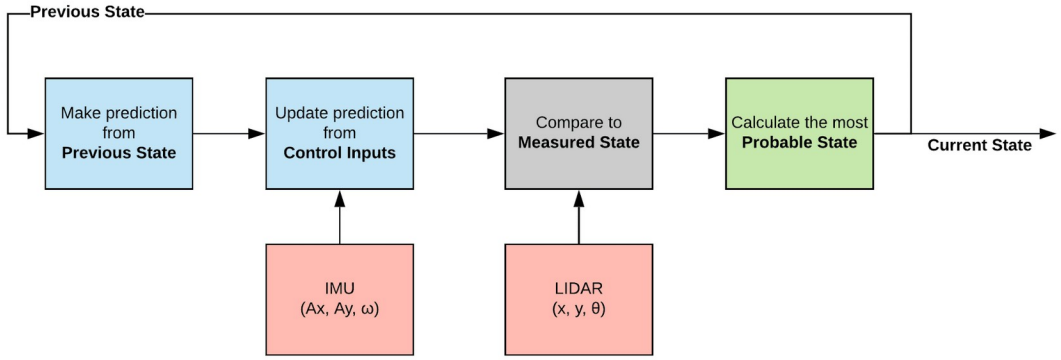


Figure ???
Basic process of our Kalman Filter

Stepping through this process one by one:

1. **Make prediction from Previous State**

This is purely mathematical. We simply take the previously calculated velocity and position to update our new position over the time step.

2. **Update prediction from Control Inputs**
This is where we use our IMU data. Although the IMU is not itself controlling anything, it describes any sudden changes in direction (due to manual/autonomous control decisions) that the filter should consider. This updates the prediction from the previous step to provide a final mathematical prediction for our location.

3. **Compare to Measured State**
We can now compare the mathematical prediction of our "state" (our position and direction) with the LIDAR's measurements. We are interested in the difference between the two (as they should be very close if our model and the LIDAR are accurate).

4. **Calculate the most Probable State**
This is where the true power of the Kalman Filter comes into play. The filter corrects itself over time, updating hidden variables to help decide what combination of the mathematical prediction and the LIDAR readings will produce the most reliable positional and directional data.

We also included one more add-on to the Kalman Filter: adaptive filtering. Our robot will often be stopping and turning; this is something that the mathematical model cannot possibly predict.

Consider a robot moving in a straight line. The LIDAR data will be noisy – bouncing around the line due to small errors. Because of this, the Kalman Filter will most likely heavily favor the mathematical model over the LIDAR data, as it can easily and accurately predict straight-line motion.

If the robot now suddenly turns, it will still, at first, heavily favor the mathematical prediction. This prediction is now very wrong, as it will predict it to continue along the same path. It will take some time for the Kalman Filter output to again fully converge on the correct output, a behavior we want to avoid. This behavior is demonstrated below:

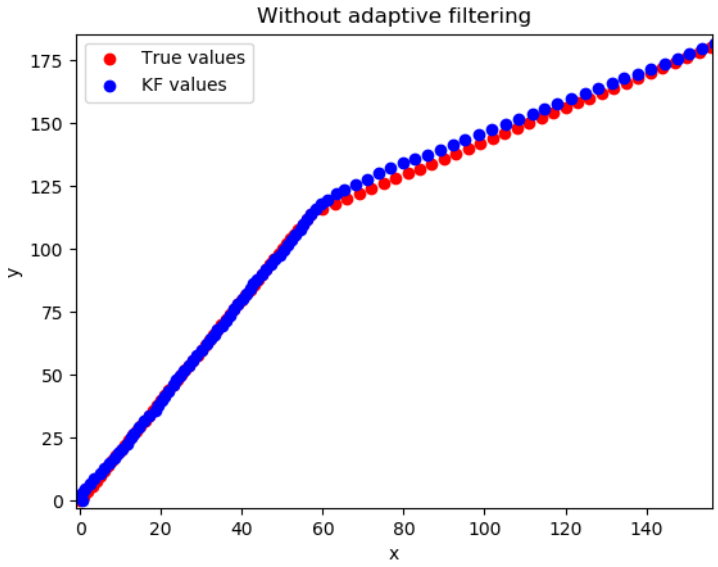


Figure ???
Kalman Filter without adaptive filtering

We can remedy this by adding one simple rule: if the mathematical prediction and LIDAR readings differ *greatly*, put less trust into the mathematical prediction. We always expect some

difference in the prediction and sensor readings, so the threshold for this must have a very low probability of occurring.

Our current implementation of an adaptive filter yields the following:

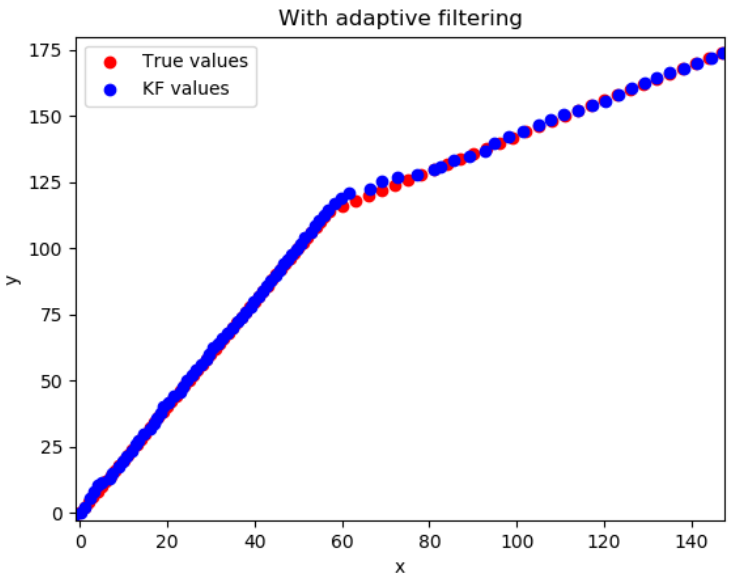


Figure ???
Kalman Filter with adaptive filtering

This is not yet a perfect implementation, as you can see a great amount of noise around the turning point. However, you can see that it converged to the correct path at around x=100, compared to x=140 without the adaptive filter.

A lot of work will need to be added on top of this to fully utilize the potential of the Kalman Filter, but hopefully this will serve as a good foundation for future years to build on.