Overview

Entach is a complex and multi-faceted system. This guide is meant to detail new developments in Entach that have occurred this past summer (2014). Topics covered are: the Entach API, the Entach Field Mapper, updatePosition, goToPoint program flow, drift correction, obstacle detection, and as a conclusion, I will address the future of Entach.

The Entach API

API stands for application programming interface. It's how a user interfaces with your program. So, like other APIs, there is a very specific way of interfacing with Entach_Common.h, the heart of the Entach API. In RobotC, you can create APIs by use of header files or ".h" files. A user interfaces with these header files by "including" them into the program by use of the command: `#include "C:\Users\Woz4tetra\Google Drive\2014 – 2015\Ben's Hit-by-a-Bus Folder\Entach API\Entach_Common.h”` (The file path is an example).

You would use header files in RobotC if multiple programs use the same methods and you don't want to keep constantly updating all files by coping and pasting. Before this, there were only two files, an autonomous and a driver program. Each had their own set of drive methods (`driveForwards()`, `driveBackwards()`, etc.) and their own set of arm control methods. Autonomous had Entach, and driver did not. During 2013's season, I found it was very tedious to keep copying and updating these two programs' methods. So, I created two header files: one for Entach specific methods and the other for driver specific methods. Entach_Common.h has no driving methods. and Robot_Common.h has no Entach methods.

However, Entach does need some of the methods in Robot_Common.h. Therefore, at the beginning of Entach_Common.h, there are method stubs indicating those methods, which should be implemented in Robot_Common.h. If you find during the season that some methods currently in Entach_Common.h are in fact robot specific (like the later explained drift correction methods), then create a method stub in Entach_Common.h and create their implementations in Robot_Common.h.

Another important note is, because every robot is different and requires different methods to function, there should only be one Entach_Common.h file per computer (in the google drive application folder) and one Robot_Common.h for each robot. So all Robot_Common.h's should include that one Entach_Common.h file.

Based on the explanations above, here's a checklist to fill out every time you create a new set of programs for your robot:

1. *Create a project folder and name after your robot*
2. *Copy Robot_Common.h into your project folder*
3. *Replace "Robot" with your robot's name*
4. *Find the directory of Entach_Common.h and put it under the "// Entach_Common.h #include" comment: #include filepath\Entach_Common.h*
5. *Copy Autonomous Template.c and Manual Control Template.c into your project folder*
6. *Rename them by the convention "mm-dd-yy [Robot Name] Autonomous.c" and "mm-dd-yy [Robot Name] Manual Control.c"*
7. *In both ".c" files, replace the path under the comment "Robot #includes" with the file path of your Robot_Common.h file*
8. *Define your pragmas in the Autonomous and Manual Control files according to your robot's configuration*
9. *Implement all methods in Robot_Common.h*

The Entach Field Mapper

So, I've gone over this program before but I haven't detailed how to use it, the context in which you should use it, or how it works. If you recall, the point of the field mapper was to take out the tedious and time consuming method of measuring every stinking goToPoint with a measuring tape. Currently, the only way to install and run it is to create a new project in Eclipse and copy in the "src" folder. I have yet to figure out how to export eclipse projects as their own applications. If someone is motivated, he/ she can work on that on his/ her own time. But for now, the mapper is contained within Eclipse.

The mapper is a single window application. Pretty much WYSIWYG (what you see is what you get). Since most of the features can be discovered just by clicking things, I'll jump to the less obvious. Arrow keys change the angle of the robot. Holding shift will fixate all new angle adjustments and new points to 45° increments. Delete removes the most reset point. Escape removes all points. Enter outputs all the points to a text file with a screenshot in a folder called "Outputs" found in the project folder. To change the default field and robot images replace the fieldImageFilePath and robotImageFilePath file paths in "Runner.java".

The purpose of the checkboxes is to set whether the robot should face a specific angle after it has arrived at the destination. This also lets Entach know if you want to strafe to a point as well. If your endAngles are the same at two consecutive points, the robot will know to strafe along that path.

To change the default robot size, scroll to the constructor of Runner() until you see:

```
robotImageWidth = robotImage.getWidth(); // ## / Coordinate.getPixelsToCM()
robotImageHeight = robotImage.getHeight(); // ## / Coordinate.getPixelsToCM()
originalRobotImageWidth = robotImageWidth;
originalRobotImageHeight = robotImageHeight;
fieldImageWidth = FTCFieldImage.getWidth();
fieldImageHeight = FTCFieldImage.getHeight();
```

You can comment out "robot.getWidth()" and "getHeight()" and replace it with the commented out code. The process to change the default field size is similar. Though I don't think you'll need to change that. Keep in mind, all the variables above are in pixels.

The purpose of the Landmark boxes is to give you a size of setting or field setting. I would put things like "window" or "table," just to keep your sense of orientation. Keep in mind these values won't be remembered when the program closes (that functionality hasn't been added).

For the images, I would follow the the images sizes in src/Images. When the new game field image is released, I would get someone handy with photoshop to find and edit a field image to the dimensions I have set (inserting gridlines would be good too).

Regarding the setting in which you'd use this, when you get to the stage of writing the autonomous code and have decided to use Entach, you use this program to layout where the robot should go. It's also very handy for quickly rerouting paths when talking to autonomous team members during competitions if there's a conflict or if we're just showing the path we follow.

If you want to make changes, you can try. I can't easily explain how everything in the code works without taking up three more pages of explanation and I don't think it's worth spending the time. If anyone is motivated and feels the mapper could be improved, I would start a new project.

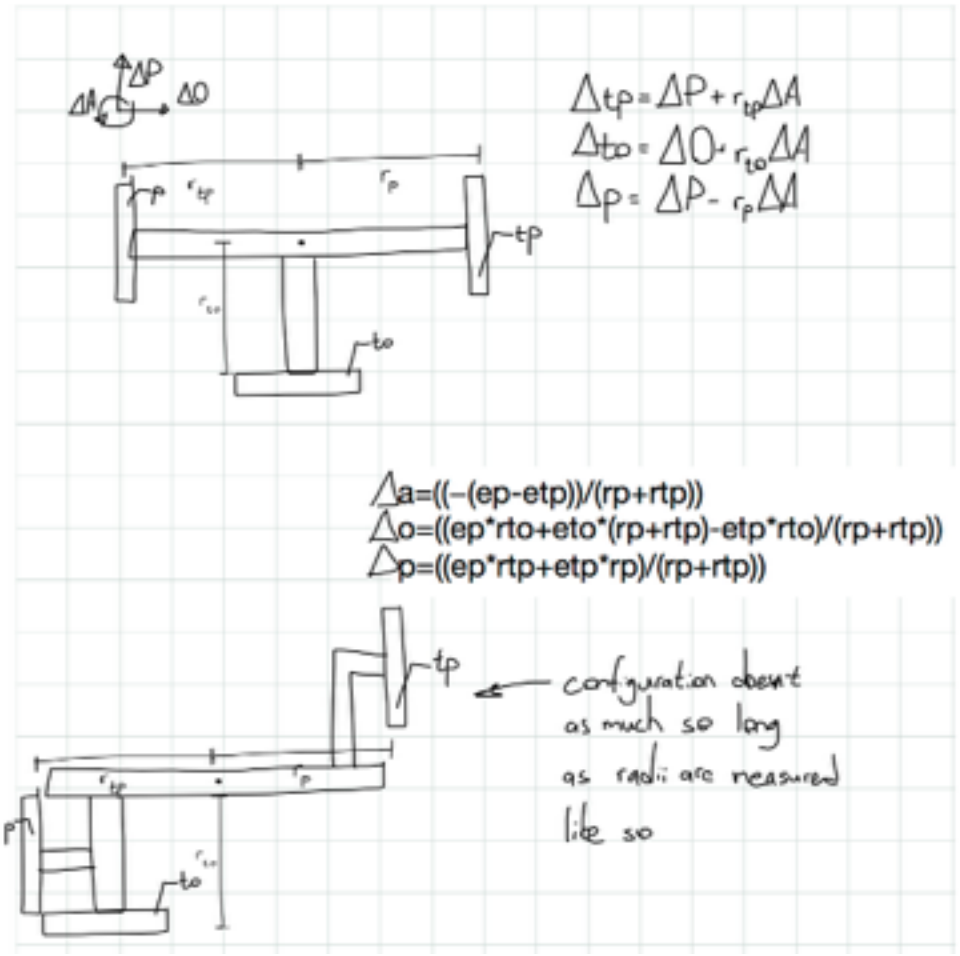
If you do start a new project, here's some of the important elements I've included into this program. I use Java's Swing to draw things on. I have an array list to store every point, every ghost image location, and checkbox. I create and set all buttons and their actions in the runner. ListOverseer handles things like text file writing and screenshots. For the most part, it does its one task pretty well and I think it'll last you through the season.

updatePosition

The latest and most advanced addition to Entach is the inclusion of the task updatePosition. This paragraph explains what a task is. If you know what they do already, skip to the next paragraph. Tasks (or threads in other programming languages) allow for multitasking. Its primary use is the ability to run multiple "while" loops at once. `task main` acts as the "super task." When this task ends, all other tasks are shut off and the program ends. updatePosition is "subtask" with its own while loop.

The purpose of updatePosition is to interpret three encoder readings into an x, y, and angle reading. updatePosition() is a task because it's constantly updating the robot's position as fast at it can. Every 5 milliseconds the difference between the current encoder values and the previous encoder values are calculated. For those of you taking integral calculus, what we're doing is taking very small Riemann sums of the encoder values. In this case it would be a right Riemann sum with a rectangular width of 5 milliseconds.

We use these three delta values (three encoders) to calculate our three unknowns: x, y and θ. The equations were derived from thinking "how can encoder movements be represented in parallel, orthogonal, and angular movement?"



ΔP
ΔA
ΔO
$\Delta tp = \Delta P + r_{tp}\Delta A$
$\Delta to = \Delta O - r_{to}\Delta A$
$\Delta p = \Delta P - r_p\Delta A$
p
r_{tp}
r_p
tp
r_{to}
to
Δa=((-(ep-etp))/(rp+rtp))
Δo=((ep*rto+eto*(rp+rtp)-etp*rto)/(rp+rtp))
Δp=((ep*rtp+etp*rp)/(rp+rtp))
tp
configuration doesn't
as much so long
as radii are measured
like so
r_{tp}
r_p
p
r_{to}
to

We then solved in terms of parallel, orthogonal, and angular movement and got the three equations currently in the program. But let's take a few steps back. If the encoders are mounted like they are in the diagram and the robot is traveling forward, then Δtp (delta trackball parallel) and Δp encoders will measure positive parallel movement and Δto will measure nothing. If the robot is traveling right, Δto will measure positive orthogonal movement and Δtp and Δp encoders will measure nothing.

Things get more complicated when angular movement is included. If the robot is just rotating, the encoder is representing the robot's arc (arc = angle • radius). However, in reality, the robot isn't perfectly spinning or moving in straight lines. It's usually spinning a little and moving at the same time (i.e. drifting). Since we're measuring the robot's movement in small increments, imagine the robot has just turned a fraction of a degree to the left (on the unit circle, that's a positive increase in angle) and moved forward a tiny amount. What's the net result in the encoders? Δtp will pick up on the positive parallel movement and at the same time pick up on the angular movement. In other words, this combined movement results in the parallel movement adding to the angular movement.

It's the same with the other encoders with some minor changes. With Δto, it picks up on the orthogonal movement combined (added) with the angular movement. With Δp it's the same as Δtp except when the robot is spinning in a positive direction, the angular movement is subtracted from the parallel movement because Δp encoder is spinning the opposite direction from the Δtp encoder.

So after we solved for parallel, orthogonal, and angular movement these equations, we now have a means of getting our change in parallel, orthogonal, and angular movement. We still aren't quite at x, y and θ yet. Parallel and orthogonal movement is relative to where the robot was previously facing. We need to make it relative to where the robot first started. That means we use trigonometry. You can think of our change in movement variables as hypotenuses. If you multiply the cosine of your angle by the hypotenuse, you get an x value. In the case of orthogonal movement though, x and y are flipped because it is perpendicular to parallel movement. This results in the three equations below:

```
currentAngle += changeInAngularMovement;
currentY +=  changeInOrthogonalMovement * cos(currentAngle) +
             changeInParallelMovement * sin(currentAngle);
currentX += -changeInOrthogonalMovement * sin(currentAngle) +
             changeInParallelMovement * cos(currentAngle);
```

Lastly it's very important to set the previous encoder instance (instance 1 in this case) to the current instance (instance 2) and *not* to the actual encoder reading.

One last thing to mention is based on testing. The configuration of the encoders doesn't matter as much and radius is measured from the center of rotation to **the first point of contact of the encoder wheel** (so if you have a very thick wheel, the point nearest the center is the point you measure to). So long as the basic layout in the diagram above is followed and the radii are measured properly, the equations should still hold. From the data found under "Entach Config Test Data," I found that changing the encoders' positions around didn't have much effect on the encoder readings. If it does, its effect is negligible compared to other errors.

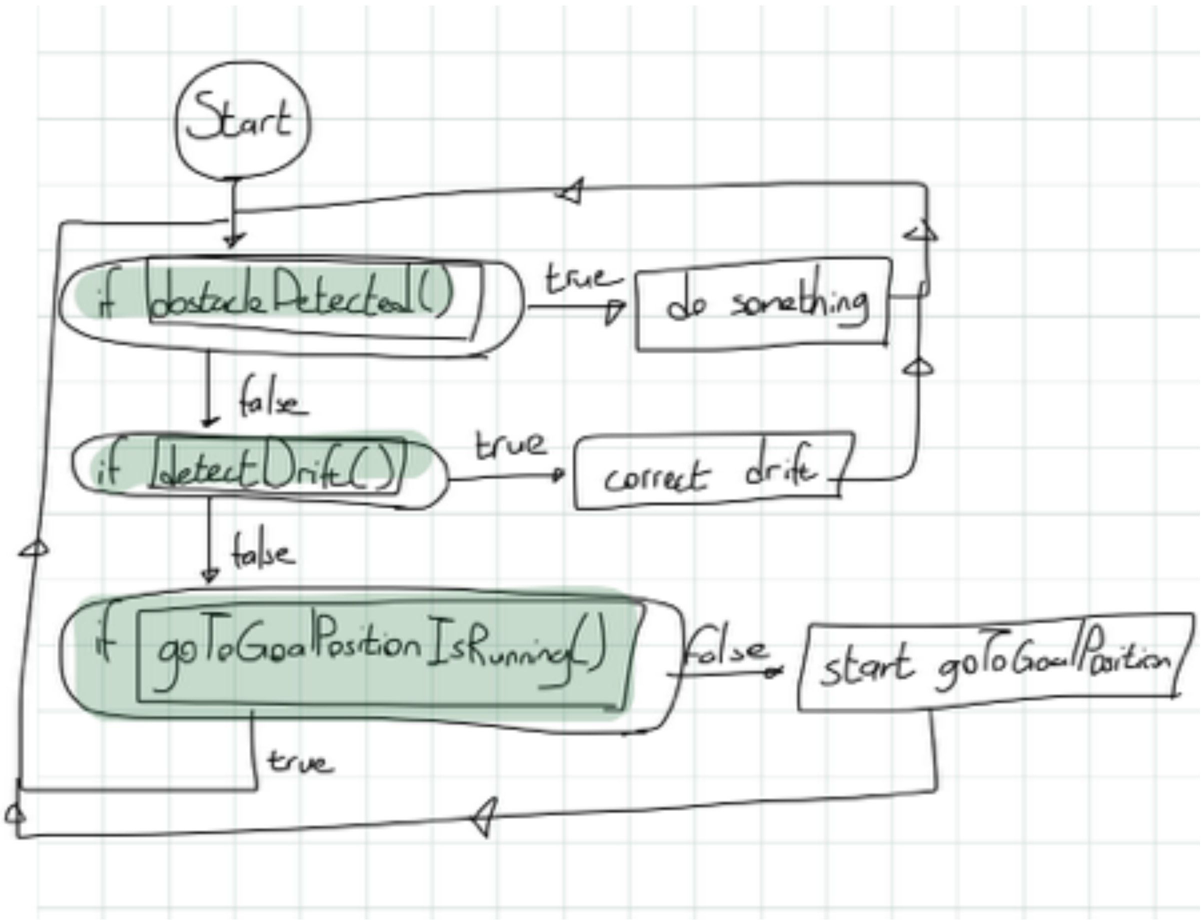
As a recap, updatePosition takes in three encoders and their radii as input and outputs the robot's current x, y, and θ.

goToPoint Program Flow

Before the creation of updatePosition(), our current position was only updated when the robot was told to move to a point (i.e. in goToDistance() and goToAngle()). However, now that our current position is being updated outside of our drive methods, the goToPoint program flow had to change. In addition, now that we have the ability to get our position at any time, we have now been given the opportunity to introduce drift correction and obstacle detection! Those will be reviewed in detail in the next sections.

So, in order to include our correcting methods, there must be some condition somewhere in the program that tells the robot to stop going to the goal position if drift or an obstacle is detected. This reminded me of another useful feature of tasks. They can be started and stopped at anytime; you can create a while(true) loop that can start and stop at will. So, goToPoint has its own subtask called "goToGoalPosition()." In this task, the flow is pretty much the same as it was in the old program: face the goal position, drive to it, face endAngle if there is one. Outside of this task, however, everything is new.

The method goToPoint itself houses the conditions that check for drift and obstacles. If nothing is detected, the robot tries to get to its goal position. The following diagram describes the program flow:

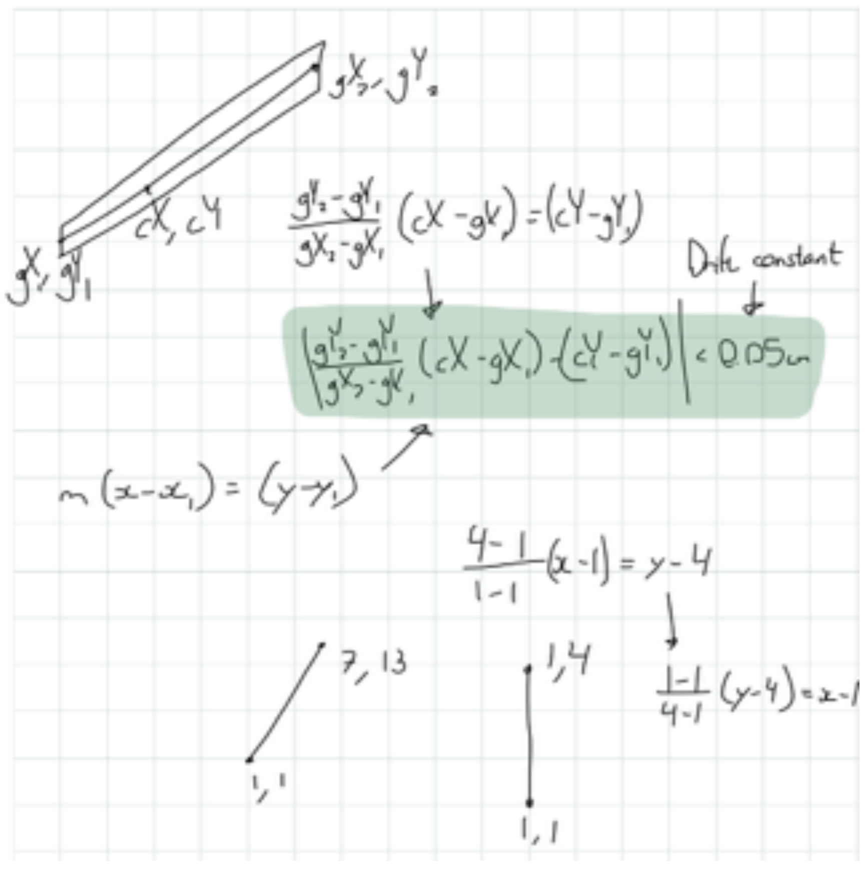


In summary, the actual action of moving to a goal position has been moved to a task called goToGoalPosition() in order to accommodate updatePosition(), drift correction, and obstacle detection.

Drift Correction

As of right now, neither drift correction nor obstacle detection has been completed or fully explored. During the last meeting at my house on July 31st 2014, we made some progress in these areas. Drift correction as of now is more developed than obstacle detection because we at least have a means of detecting drift. The way we check for drift is by checking our expected vs. actual position. The way we calculate expected position is by using a form of *m(x - x₁) = (y - y₁)* or point slope form. Expected angle is very easy to find because we set our goal angle at the start of goToPoint().

I'm using point slope because the means of traveling we've selected is straight lines. We set where we want to go at the beginning of goToPoint() in the form of goalX (x2) and goalY (y2) and we store where we were in previousGoalX (x1) and previousGoalY (x2). If we adapt point slope to these variables, you get the following:


gX₂, gY₂
cX, cY
gX₁, gY₁
$\frac{gY_2 - gY_1}{gX_2 - gX_1}(cX - gX_1) = (cY - gY_1)$
Drift constant
$\left|\frac{gY_2 - gY_1}{gX_2 - gX_1}(cX - gX_1) - (cY - gY_1)\right| < 0.05cm$
$m(x - x_1) = (y - y_1)$
$\frac{4-1}{1-1}(x-1) = y-4$
$\frac{1-1}{4-1}(y-4) = x-1$
7, 13
1, 1
1, 4
1, 1

This equation gives you a "drift constant" in centimeters. This is not enough however. This will only give you your vertical drift (or y drift constant) which is useless if you're traveling along the y axis (your slope is infinity). However, if you take the inverse function, then you get your horizontal drift. So now if we get the hypotenuse of y drift constant and x drift constant using getRelativeRadius or the Pythagorean theorem (which includes negatives for left and right drift), we now have how much we've drifted left or right from our expected position.

This is only part 1 of drift correction, however. As of right now all I've told the robot to do when drift is detected is to stop going to the goal position and drive left or right depending on the drift constant until we're at our expected position. This may not be the best solution (it's certainly not the most time efficient). Spencer has suggested we modify the motor speeds to curve us back onto the path and not interrupt goToGoalPosition at all. This would save us time instead of stopping in our tracks every time drift is detected. My point is that this area is very much open for development.

In summary, to detect drift we have a means of detecting drift (our drift constant). What we do with the drift constant is still up for debate. I've had ideas, Spencer's had ideas, but it's up to you to finish it off.

Obstacle Detection

Obstacle detection is in a less developed stage than drift correction is as I've said before. We have no tested or thoroughly thought out way of even detecting obstacles. Laura and Ben P. worked on this problem last time and narrowed down the problem. They concluded we only need to worry about other robots and stray field elements, or in other words, things we can't predict as we map out our goToPoints() (walls and fixed game elements are excluded).

Sensors aren't very practical here because we need to detect obstacles coming in on all sides. Touch sensor bumpers would work, but they take a lot of sensors and the bumpers are difficult to build correctly (as demoed by my engineering robot that Hewitt borrowed for the summer). Ultrasonic sensors are the same. You need a lot for them to be effective.

So, that's left us with the encoders again. I've had the idea detecting speed differences in the hopes of detecting collisions. A sudden drop in speed would indicate we've hit something. Although efficient in hardware resources, it will be difficult to interpret speed differences as impact. If someone hits us from behind or the side, the speed won't change as much as if the robot came to a dead stop. Ben P. pointed out that any lip we climb onto (like the ramp from last year) may decrease speed and could be confused for an obstacle.

In summary, obstacle detection is the least explored frontier, but what we have is a narrowed down problem and some ideas of speed delta detection.

What's Next?

I've sort of insinuated what I think should happen with Entach next, but there's a few more things I'd like to suggest: should we still build a trackball, drift and obstacle detection, awards for Entach, and the Entach sub-team.

The newest physical feature to the robot is the sort of "skeletal" trackball (a track wheel) I've attached to the side of the robot. I've talked to Roger about adding a fully fledged mouse ball trackball, but what you must decide for yourselves to decide whether it is still worth building a 3D printed trackball. The biggest problem with trackballs and the reason they were replaced with optical mice is dust getting jammed in the system. The track wheel doesn't seem to have this problem. The track wheel has worked great for the past month and now the equations have arrived at a point where you can put the encoders pretty much anywhere and they'll do the right thing. However, you'll probably want to make this decision after you've settled on a base layout because Entach was designed to be adaptable to *your* setup. Though, because

Entach does have strafing methods in it, I would recommend you select a base that has strafing capabilities. It may also make drift correction easier.

For drift and obstacle detection, I've already said what needs to be done, but to summarize, we have a means of drift detection, now we need to come up with a method of correcting the detected drift. We have no means of obstacle detection, though we have a few ideas. You may find you don't need to work on these problems but past experience tells me you will have to face these challenges in some form or another if Entach is to work. Unless Rivers gets an industrial grade machine shop and staff, I doubt we'll be producing a "driftless" base. In other words, our equipment isn't adequate enough to machine out all the mechanical imperfections. Also, robots in terms of movement behavior usually stays consistent from year to year meaning there will inevitably be collisions on the field. So, regardless of the challenge next season, I would highly recommend working on these two frontiers.

I would also recommend creating an sub-team for Entach or at least select a few dedicated robotics members who are willing to put in many after school, weekend hours, and who know trigonometry (calculus would be nice too). Entach thrives on dedication and its downfall is lack of hours. It took me all summer and last year's post season to figure out everything in this document (I'd roughly estimate 30 – 40 work hours). So, if you find Entach to be relevant next year, a lot of time and dedication will need to be put in if it is to be finished.

I would also look for awards for Entach and how to submit our code. Mr. Ganderson mentioned there was one and I made the huge mistake of overlooking this award last year. I'm not sure how much you can and should submit, but I'd think everything Entach related in "Ben's Hit-by-a-Bus Folder" should be submitted along with a report explaining everything (this document should be of help).

I'd like to close by thanking you all. If you've read everything in this document and understood it all, you are a champion and I have full confidence Entach is in good hands. Don't be nervous about filling the senior shoes or anything like that. So long as you are dedicated to your goals, persistent through the challenges, and take the time once and a while to think, you'll do just fine.