







5839B's Programming Notebook

Table of Contents

Entries

2024/03/05		First Steps	1
2024/03/08		Library Structure	4
2024/03/08		Asset and TrackingPod Classes	6
2024/03/09		Odometry Class	7
2024/03/12		Mecanum Drivetrain Model	10
2024/03/14		Pure Pursuit	11

Appendix

Appendix A: Odometry Derivation	1
Glossary	3
Credits	4

Post Season Notes

With the end of the Over Under season, sadly we didn't make it to worlds, we evaluated our progress and decided that we need a better programming library. My job as the coder is to record and create a programming library that serves all our needs for the improved robot.

In the Over Under season we didn't get all of the systems we wanted running working in time for the competition, for the autonomous we used PROS with both the OkapiLib library and the LemLib library, however each of those had its own shortcomings, so we decided to create our own library.

But first, we had to decide which library to build off of, the following decision matrix shows our considerations and our decision

	Time to Develop	Extendable	Working Subsystems	Experience With Library	Future Proof	Code Readability	Total
PROS	2	10	1.5	4	3	0.5	21
OkapiLib	8	8	4.5	4	3	1.5	29
LemLib	4	6	6	2	0.75	0.5	19.25

Note

OkapiLib had the following Subsystems:

- Odometry (Centered Forward Pods)
- Solid Base Classes
- X-Drive Model
- PID

LemLib has the following Subsystems:

- Odometry (flexible placement)
- Pure Pursuit
- No other drivetrain functions worked
- PID not working
- Asset (*useful* utility)

Final Decision

We decided to build the library off of OkapiLib in the end, as it has more flexibility, and a solid codebase, while taking inspiration from LemLib

Necessary Components

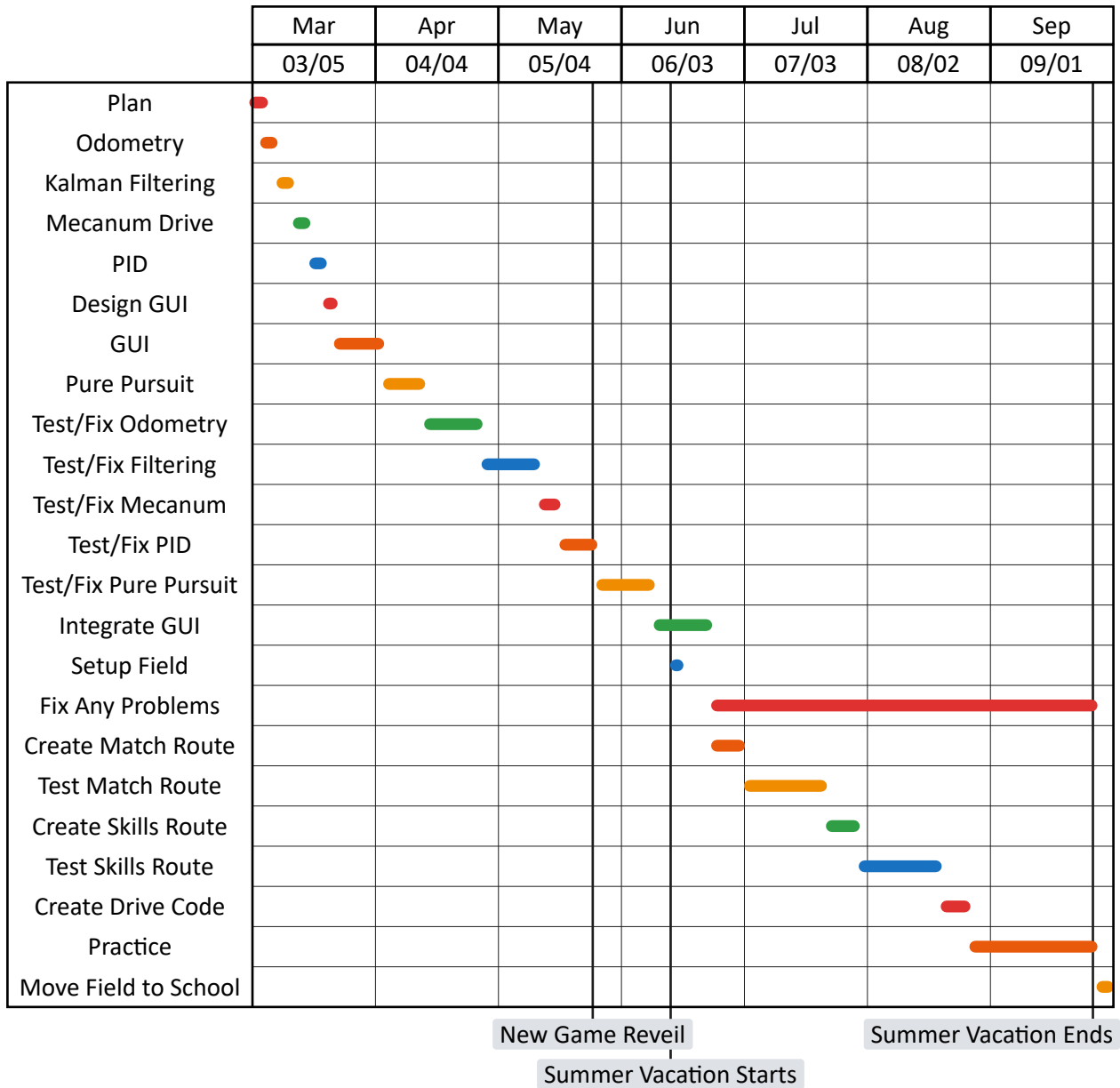
We have limited time to develop the library, as our schoolwork and jobs eat into the development time, so we have made a ongoing list, shown below, that shows all the features we need in the library along with a short explanation of the feature, a longer explanation will be given in a dedicated section of the notebook

- **Mecanum Drivetrain Model** - Although OkapiLib has an X-drive model, it doesn't have a Mecanum drive model, which has slightly different kinematics
- **Boosted Mecanum Wheel Model** - Since our robot not only has a 4 motor mecanum drivetrain, the drivetrain also includes 2 separately powered omni-wheels to help boost forward torque
- **Custom Odometry** - Although OkapiLib already has odometry, our design places all of the odometry pods in a straight line with the 2 forward facing wheels not being centered, OkapiLib doesn't support this so we need to create a custom odometry class for this, however this should be fairly straightforward
- **PID and other Control Loops** - this will allow us to have fast and accurate movements
- **Filtering Algorithms** - this will allow us to read more accurate readings by using previous values to validate and estimate a more accurate sensor reading
- **Pure Pursuit** - Pure Pursuit is an algorithm that allows the robot to follow paths smoothly, even with disturbances, which is necessary for a fast and accurate autonomous period
- **Asset** - this is a helpful utility from LemLib that would be very helpful in creating Pure Pursuit paths, as it allows the coder to not have to remove and reinsert the SD card every time a path is updated
- **GUI** - this is necessary as it allows us to live tune variables, for example, we can create a PID tuner so we don't have to recompile after changing a single variable, this will speed up tuning by a significant amount as half the time it takes to tune PID is spent waiting for code to compile. This will also help our robot look cleaner as a build

Goal

Our goal is to finish most (> 95%) of the library before **September 10, 2024**

This will allow us to have a reasonable amount of time to actually create the autonomous and any other functions that come up while doing so



Using proper C++ coding styles is a must in a project of this size, so we need a consistent and readable file structure which the following diagram will show

```
include/
├─ lib5839/
│  ├─ robot/
│  │  ├─ flywheel.hpp
│  │  ├─ catapult.hpp
│  │  ├─ lift.hpp
│  │  ├─ wings.hpp
│  │  └─ PT0.hpp
│  └─ chassis/
│     ├─ controller/
│     │  ├─ purePursuitController.hpp
│     │  └─ purePursuitControllerPID.hpp
│     └─ model/
│        ├─ mecanumDriveModel.hpp
│        ├─ boostedMecanumDriveModel.hpp
│        ├─ threeEncoderMecanumDriveModel.hpp
│        └─ threeEncoderBoostedMecanumDriveModel.hpp
├─ odometry/
│  ├─ trackingPod.hpp
│  └─ threeWheelOdometry.hpp
├─ GUI/
│  ├─ odomDebugGUI.hpp
│  └─ PIDTunerGUI.hpp
├─ utils/
│  ├─ filtering.hpp
│  ├─ odomMath.hpp
│  └─ asset.hpp
├─ api.hpp
├─ liblvgl/
├─ ...
├─ okapi/
├─ ...
├─ pros/
├─ ...
├─ api.h
├─ main.hx
├─ globals.hpp
├─ gui.hpp
src/
├─ lib5839/
│  ├─ robot/
│  │  ├─ flywheel.cpp
│  │  ├─ catapult.cpp
│  │  ├─ lift.cpp
│  │  ├─ wings.cpp
│  │  └─ PT0.cpp
│  └─ chassis/
│     └─ controller/
```

```
| | | | └─ purePursuitController.cpp
| | | | └─ purePursuitControllerPID.cpp
| | | | └─ model/
| | | |   └─ mecanumDriveModel.cpp
| | | |   └─ boostedMecanumDriveModel.cpp
| | | └─ odometry/
| | |   └─ trackingPod.cpp
| | |   └─ threeWheelOdometry.cpp
| | └─ GUI/
| |   └─ odomDebugGUI.cpp
| |   └─ PIDTunerGUI.cpp
| └─ utils/
|   └─ odomMath.cpp
└─ main.cpp
└─ globals.cpp
└─ gui.cpp
static/
  └─ path.txt
```



Asset Class

The asset class is a useful utility that originated from the LemLib library. It is a compile time function and accesability class that compiles any files in the static folder and sends them to the brain while still allowing c++ functions to have access to the contents of the file, this allows us to not have to take out the SD card that would have stored the static files, find and use a SD card reader to upload files to the SD card, and then insert the SD back into the brain every time we change or add a static file. Static files being files that aren't evalutated during runtime, they are read by the program to get large blocks of information.

This is especially helpful when creating the GUI and Pure Pursuit Subsystems as both of those require static files. The Pure Pursuit requires a text file containing the path, which will be explained more in a future entry, and the GUI requiring images to display.

Implimentation

Stores:

- the whole file as a string
- the length of the file

Can Return:

- the file converted to many file types
- the file split by a deliminitor

TrackingPod Class

The TrackingPod class is part of the Odometry system, it stores information on where it is located relative to the tracking center of the robot, and have helpful functions that convert encoder ticks to distance traveled.

Note

This class is not comaptible with the chassisScales class used in the OkapiLib's odometry base class, so there will be some challanges with this.

Implimentation

Stores:

- diameter of tracking wheel
- distance to tracking center
- reference to encoder (to get the absolute position of the sensor)
- gear ratio (default: 1)

Can Return:

- distance traveled since last reset



Explanation

Odometry, also known as Dead Reckoning, is a way for a robot to know its global position, by using, at minimum, 2 encoders parallel to each other, a third encoder perpendicular to the other two can allow the robot to also track horizontal movement.

Knowing the global position of the robot is useful as it allows the robot to self correct if it is disturbed by a game element or other robot. It also allows the robot to be able to use the pure pursuit algorithm, which will be explained in a later entry

This is done by taking the amount each encoder changed by each frame and using some math to convert that into how much the robot has moved and turned in the x and y direction.

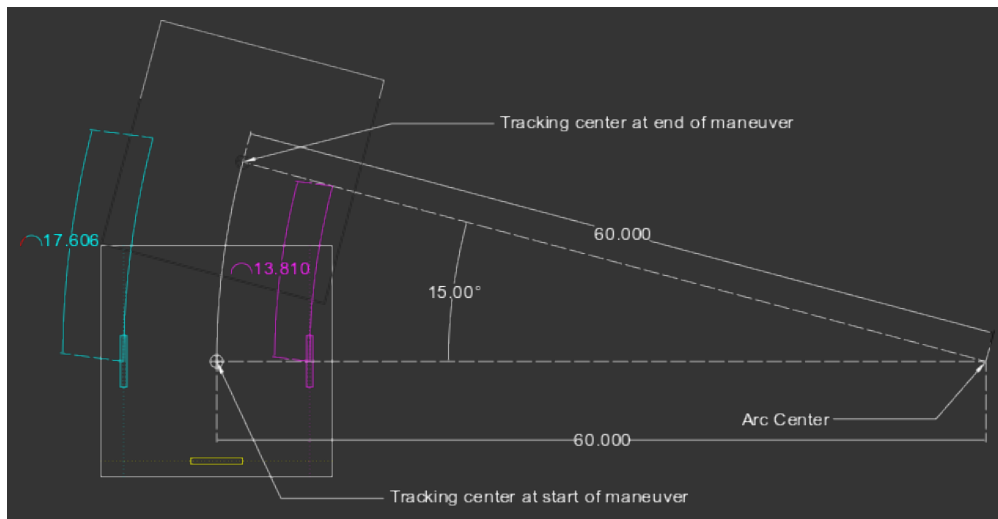


Figure 1: A simple case of the robot turning while moving forward

Figure 1 shows how when the robot moves, the wheels make 2 arcs that can then be used to calculate the radius of the turn and the distance traveled in each direction with the equations below. See Appendix A for a derivation of the equations



fx Equation

ΔL : the distance the left encoder wheel moved since the last tick (inches)

ΔR : the distance the right encoder wheel moved since the last tick (inches)

ΔS : the distance the back encoder wheel moved since the last tick (inches)

s_L : the distance from the tracking center to the left encoder wheel (inches)

s_R : the distance from the tracking center to the right encoder wheel (inches)

s_S : the distance from the tracking center to the back encoder wheel (inches)

$\Delta\theta$: the change in heading (radians)

$\vec{\Delta d}_l$: the translation vector in the local coordinate system (inches)

$$\Delta\theta = \frac{\Delta L - \Delta R}{s_L - s_R}$$

$$\vec{\Delta d}_l = 2 \sin\left(\frac{\theta}{2}\right) * \begin{pmatrix} \frac{\Delta S}{\Delta\theta} + s_S \\ \frac{\Delta R}{\Delta\theta} + s_R \end{pmatrix}$$

We can use those equations to create an algorithm that runs every 10 milliseconds, shown below

```

1  void lib5839::threeWheelOdometry::step() {
2      // get the distances each encoder moved the last tick
3      QLength deltaV1 = sensors.vertical1->getDistanceTraveled() - prevV1;
4      QLength deltaV2 = sensors.vertical2->getDistanceTraveled() - prevV2;
5      QLength deltaH1 = sensors.horizontal1->getDistanceTraveled() - prevH1;
6
7      prevV1 = sensors.vertical1->getDistanceTraveled();
8      prevV2 = sensors.vertical2->getDistanceTraveled();
9      prevH1 = sensors.horizontal1->getDistanceTraveled();
10
11     // get change in heading
12     deltaTheta = (deltaV1 - deltaV2) / (sensors.vertical1.getOffset() -
sensors.vertical2.getOffset());
13     avgHeading = currentPose.theta + deltaTheta / 2.0;
14
15     // prevent divide by zero when finding local translation
16     if (deltaTheta == 0) {
17         localX = deltaH1;
18         localY = deltaV1;
19     } else {
20         localX = 2 * sin(deltaTheta / 2) * (deltaH1 / deltaTheta +
sensors.horizontal1.getOffset());
21         localY = 2 * sin(deltaTheta / 2) * (deltaV1 / deltaTheta +
sensors.vertical1.getOffset());
22     }

```



```
23
24 // convert local translation to global position
25 currentPose.x += localY * sin(avgHeading);
26 currentPose.y += localY * cos(avgHeading);
27 currentPose.x += localX * -cos(avgHeading);
28 currentPose.y += localX * sin(avgHeading);
29 currentPose.theta += deltaTheta;
30 }
```

the odometry class also inherits from OkapiLib's base odometry class so it needs some other functions:

- getState() - gets the current pose of the robot as calculated by the odometry
- setState(Pose pose) - sets the current pose of the robot
- getModel() - gets the chassis model the odometry is using
- getScales() - get information about the position of the sensors, however since the positioning on our robot is incompatible with this function it will return an empty class

This odometry class will be updated to also include the imu on the robot after we work on kalman filtering, which will help improve the accuracy of the odometry



tldr;

Odometry is a way for a robot to know its global position on the field by using 2-3 free spinning wheels, and using arcs to approximate the path the robot takes.

The global position can be used to allow the robot to self correct even though major disturbances to the path.



A Mecanum drivetrain is a type of holonomic drivetrain, which means it can move in all directions without having to face in the direction it needs to move, basically, it can move sideways.

This is useful in both driver control and autonomous, in driver control, this would help the driver maneuver around the field, avoiding conflicts with other teams, and better allowing the scoring of points around the field. And in autonomous this can allow the robot to not have to waste time turning before and after moving.

Using a tank drivetrain, to move to a point, the robot needs to first face that point, by turning to $\arctan\left(\frac{\Delta y}{\Delta x}\right)$ radians where Δy is the difference between the robot's y position and the y position of the goal point, and similarly for Δx we can then use the distance formula $\sqrt{(\Delta x)^2 + (\Delta y)^2}$ to move forward until we reach the goal point, then finally we have to turn to the desired angle. This takes 3 steps, but using a holonomic drivetrain we can just do that in one step

To go from the robot's current position to a desired goal point, we first need to find the velocity and travel distance of each wheel separately, we found 2 different methods of calculating velocity, one just added up the turn, forward and strafe velocities with differing signs for each wheel, while the other algorithm uses the sine and cosine functions to compute the wheel velocity for the forward and strafe directions but then it adds the turn velocity separately. Both algorithms then need to normalize the values after adding the velocities together. However the first algorithm we mentioned is easier and gives a very similar result to the second algorithm, so we decided to use that algorithm.

To find the wheel velocities V_{FL} , V_{FR} , V_{BL} , V_{BR} we add the turn (V_T), forward (V_F) and strafe (V_S) velocities in the following ways

Equation

$$V_{FL} = V_F + V_S + V_T$$

$$V_{FR} = V_F - V_S - V_T$$

$$V_{BL} = V_F + V_S - V_T$$

$$V_{BR} = V_F - V_S + V_T$$

we then have to normalize the values: if $\max(V_{FL}, V_{FR}, V_{BL}, V_{BR}) > V_{\max}$ we need to multiply all wheel velocities by $\frac{V_{\max}}{V_{\text{biggest}}}$ where $V_{\text{biggest}} = \max(V_{FL}, V_{FR}, V_{BL}, V_{BR})$

We first implemented this in python VexCode so we can test if the equations work, the results can be seen at <https://youtu.be/eSuvUhx34>

After the successful results of the VexCode prototype, we then started working on the implementation in the library, the OkapiLib code style guide mandates that we build off of the ChassisModel base class, we did this and used many of the same functions as the X-drive class as the drivetrains are almost the same, just some small differences in the kinematic equations.

The next step was to work on finding the travel distance for each wheel, this was not implemented in the X-drive class from OkapiLib so we decided to delay the development until after the Kalman Filtering.



Pure Pursuit is, in our opinion, the most useful algorithm that is used in the vex autonomous period. Pure Pursuit is, in essence, a algorithm to make the robot smoothly follow a predetermined path.

To learn the algorithm, i used Purdue Sigbots's [Resource](#) as a base and added on the features I wanted. Their website used a donkey chasing a carrot, which always stays in front of the donkey, as a helpful metaphor on how the algorithm actually worked



An image showing the metaphor used by the Purdue Sigbots Website on the Pure Pursuit Algorithm

So speaking more technically, the algorithm is given a set of points on a path, these points are linearly interpolated between so the point density has to be high for the path to be accurate, the algorithm is also given the robot's position and a lookahead distance, the distance the robot should look ahead to find the point it's heading towards.

We first need to find the point the robot is heading towards, we'll call this the goal point. The goal point needs to meet certain criteria,

- it needs to be on the path
- it cannot be further away from the robot than the lookahead distance
- it needs to be ahead of the nearest point to the robot on the path
- it needs to be the furthest point on the path that satisfies the conditions above

The criteria above do simplify the algorithm, however it creates some limitations:

- the path cannot cross over itself, as the algorithm would skip part of the path
- the lookahead distance needs to be tuned properly, high lookahead distance leads to path smoothness, but low leads to accuracy

To find the goal point we draw a line between each point on the path, we then draw a circle, centered on the robot with a radius equal to the lookahead distance, and list any intersections, we then find the furthest point on the path and check if it is ahead of the robot, if the robot is ahead of the furthest point, we know the path is ending so we set the lookahead point to the final point in the path, otherwise it is the furthest point on the path. We can use the following code to find the goal point.

.

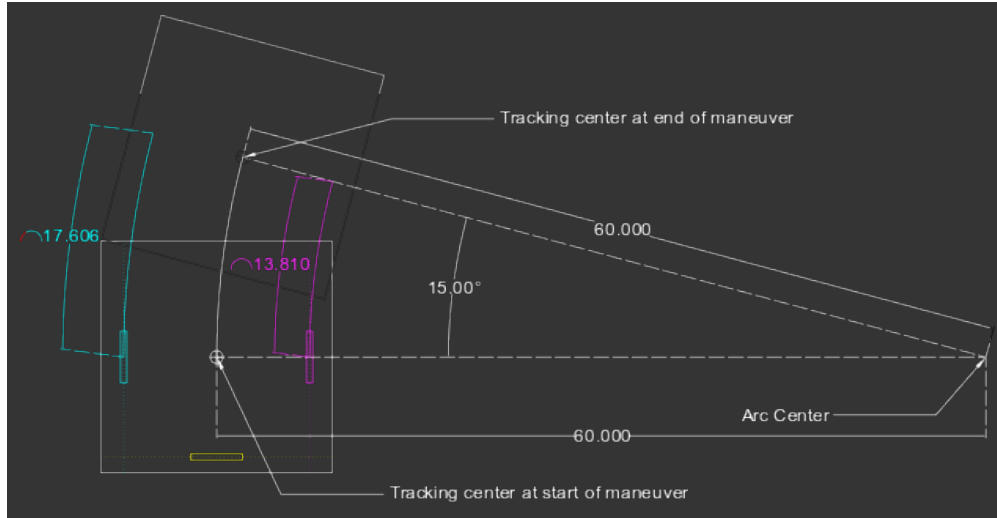
[illegible]



```
47     goalPt = path[path.length];  
48     else  
49         goalPt = path[std::max(points)];  
50  
51     return goalPt  
52 }
```

we can then use the functions created previously to go towards the goal point and update it as we do so, finishing the pure pursuit algorithm

Appendix A: Odometry Derivation



We first start with the arc length formula and solve for r_A :

$$\begin{aligned}\Delta L &= r_L \Delta \theta \\ \Delta L &= (r_A + s_L) \Delta \theta \\ \frac{\Delta L}{\Delta \theta} &= r_A + s_L \\ r_A &= \frac{\Delta L}{\Delta \theta} - s_L\end{aligned}$$

$$\begin{aligned}\Delta R &= r_R \Delta \theta \\ \Delta R &= (r_A + s_R) \Delta \theta \\ \frac{\Delta R}{\Delta \theta} &= r_A + s_R \\ r_A &= \frac{\Delta R}{\Delta \theta} - s_R\end{aligned}$$

The two equations can then be combined to eliminate the radius (the only unknown) and solved for $\Delta \theta$:

$$\begin{aligned}\frac{\Delta L}{\Delta \theta} - s_L &= \frac{\Delta R}{\Delta \theta} - s_R \\ \Delta L - s_L \Delta \theta &= \Delta R - s_R \Delta \theta \\ \Delta L - \Delta R &= \Delta \theta (s_L - s_R) \\ \Delta \theta &= \frac{\Delta L - \Delta R}{s_L - s_R}\end{aligned}$$

To find the local translation, we first need to set the local coordinate system, the easiest one being one where the y axis coincides with the endpoints of the arc. This will allow us to calculate the y translation using the equation for chord length:

$$2 \sin\left(\frac{\Delta \theta}{2}\right) * \left(\frac{\Delta R}{\Delta \theta} + s_R\right)$$

And now we can use the back wheel to find out how much the robot strays from the arc, by approximateing another arc, and this arc's chord is perpendicular to the chord of the main arc so we can use the same formula for chord length and just add it to the y translation to make a translation vector:

$$2 \sin\left(\frac{\Delta \theta}{2}\right) * \left(\begin{pmatrix} \frac{\Delta S}{\Delta \theta} + s_S \\ \frac{\Delta R}{\Delta \theta} + s_R \end{pmatrix}\right)$$

Appendix A: Odometry Derivation

And to rotate the local translation vector back to the global coordinate system we rotate it back by $\theta_0 + \frac{\Delta\theta}{2}$ which is the angle of the main chord, using the following equation:

$$\begin{pmatrix} Y_l \sin\left(\theta_0 + \frac{\Delta\theta}{2}\right) - X_l \cos\left(\theta_0 + \frac{\Delta\theta}{2}\right) \\ Y_l \cos\left(\theta_0 + \frac{\Delta\theta}{2}\right) + X_l \sin\left(\theta_0 + \frac{\Delta\theta}{2}\right) \end{pmatrix}$$

GUI

Graphical User Interface - A way to display information on the robot brain in a clean and presentable manner

PID

Proportional, Integral, Derivative - A type of control loop that takes in error and returns new motor value

PTO

Power Take Off - A mechanism where a single motor can switch between powering 2 different mechanism using pneumatics

delimiter

A character or set of characters that separate parts of a string

pose

A structure containing the x, y, and heading of the robot

Credits

- Purdue Sigbots
- The creators and contributors to PROS
- The creators and contributors to OkapiLib
- The creators and contributors to LemLib
- Theo from Team 7842F/B
- Felix from 53E
-

