5839B's Programming Notebook

Table of Contents

Entries

Appendix

Post Season Notes

With the end of the Over Under season, sadly we didn't make it to worlds, we evalutated 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 shortcommings, 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 of of OkapiLib in the end, as it has more flexibility, and a solid codebase, while taking inspiration from LemLib

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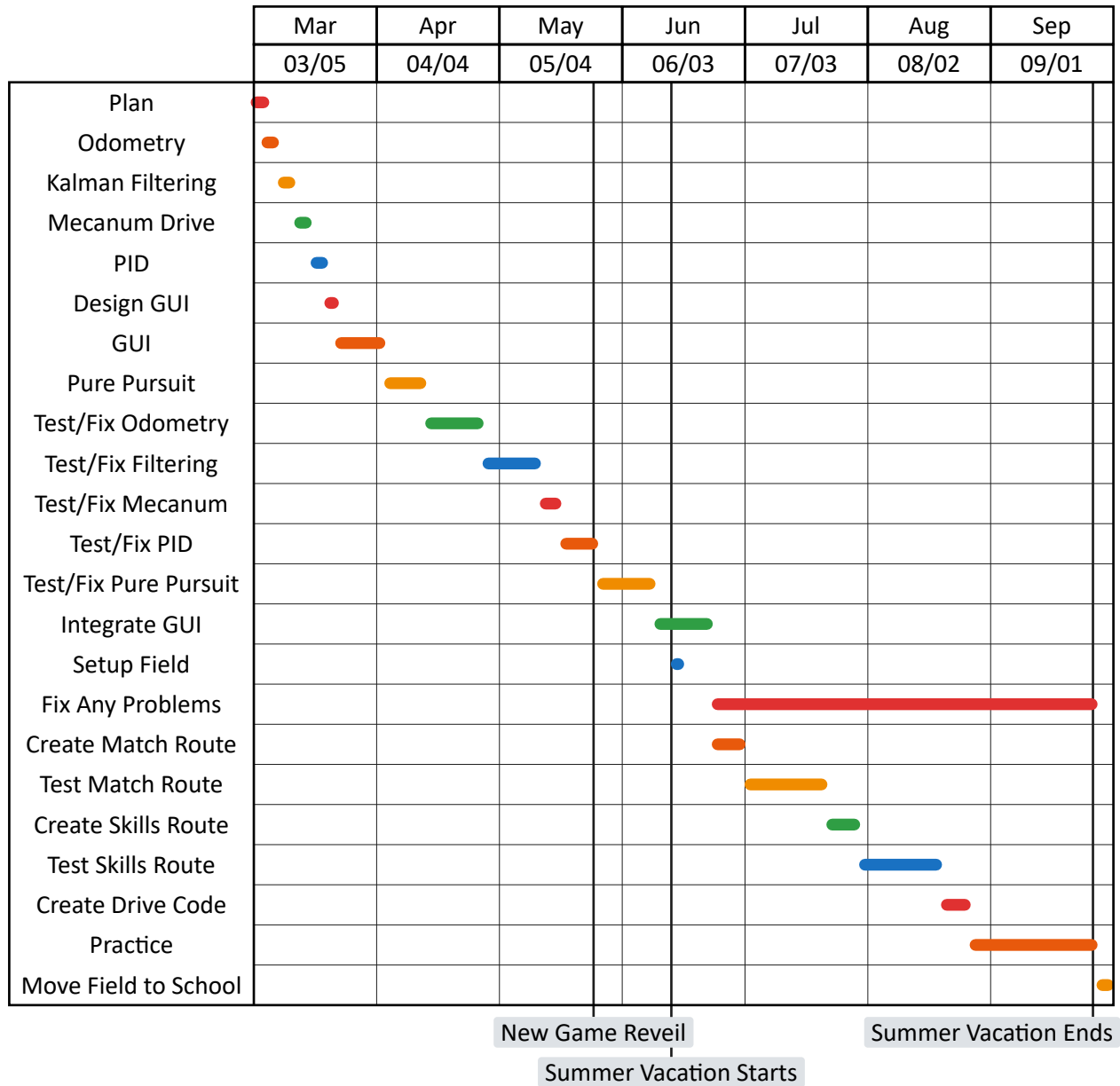
We have limited time to develop the library, as our schoolwork and jobs eat into the development time, so we have made a ongoing list, shown below, that shows all the features we need in the library along with a short explaination of the feature, a longer explaination will be given in a dedicated section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

Designed by: Praful Adiga
Witnessed by: Brandon Lewis

- **Mecanum Drivetrain Model** - Although OkapiLib has an X-drive model, it doesn't have a Mechanum drive model, which has slightly different kinematics
- **Boosted Mecanum Wheel Model** - Since out robot not only has a 4 motor mechanum drivetrain, the drivetrain also includes 2 seperately 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 nessasary 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 nessasary as it allows us to live tune variables, for exmaple, 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 to tune PID is spent waiting for code to compile. This will also help out 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

---

Designed by: Praful Adiga
Witnessed by: Brandon Lewis

	Mar	Apr	May	Jun	Jul	Aug	Sep
	03/05	04/04	05/04	06/03	07/03	08/02	09/01
Plan							
Odometry							
Kalman Filtering							
Mecanum Drive							
PID							
Design GUI							
GUI							
Pure Pursuit							
Test/Fix Odometry							
Test/Fix Filtering							
Test/Fix Mecanum							
Test/Fix PID							
Test/Fix Pure Pursuit							
Integrate GUI							
Setup Field							
Fix Any Problems							
Create Match Route							
Test Match Route							
Create Skills Route							
Test Skills Route							
Create Drive Code							
Practice							
Move Field to School							

New Game Reveil

Summer Vacation Starts

Summer Vacation Ends

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

```
include/
├── lib5839/
│   ├── robot/
│   │   ├── flywheel.hpp
│   │   ├── catapult.hpp
│   │   ├── lift.hpp
│   │   ├── wings.hpp
│   │   ├── PTO.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
│   │   ├── PTO.cpp
│   ├── chassis/
│   │   ├── controller/
```

Designed by: Praful Adiga
Witnessed by: Brandon Lewis

```
|   |   |   ├── 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.

> ### >_ Implimintation
>
> 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 deliminator

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.

> ### >_ Implimintation
>
> 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

Explaination

Odometry, also known as Dead Reckoning, is a way for a robot to know it's 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 horizonal movement.

Knowing the global position of the robot is usful 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.
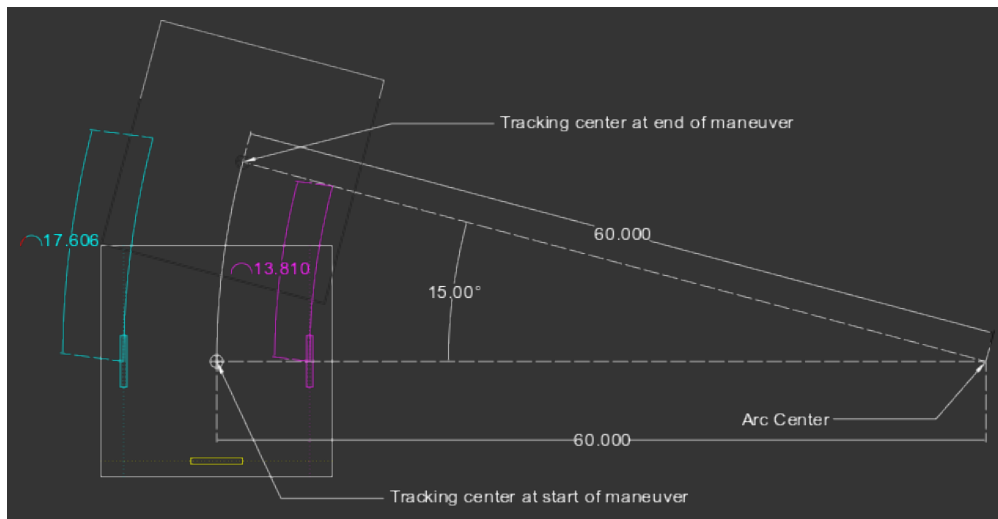

Figure 1: A simple case of the robot turning while moving forward

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 Apendix A for a derivation of the equations

---

Designed by: Praful Adiga
Witnessed by: Brandon Lewis

𝆑x 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)

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

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

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

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

```cpp
void lib5839::threeWheelOdometry::step() {
  // get the distances each encoder moved the last tick
  QLength deltaV1 = sensors.vertical1->getDistanceTraveled() - prevV1;
  QLength deltaV2 = sensors.vertical2->getDistanceTraveled() - prevV2;
  QLength deltaH1 = sensors.horizontal1->getDistanceTraveled() - prevH1;

  prevV1 = sensors.vertical1->getDistanceTraveled();
  prevV2 = sensors.vertical2->getDistanceTraveled();
  prevH1 = sensors.horizontal1->getDistanceTraveled();

  // get change in heading
  deltaTheta = (deltaV1 - deltaV2) / (sensors.vertical1.getOffset() -
sensors.vertical2.getOffset());
  avgHeading = currentPose.theta + deltaTheta / 2.0;

  // prevent divide by zero when finding local translation
  if (deltaTheta == 0) {
    localX = deltaH1;
    localY = deltaV1;
  } else {
    localX = 2 * sin(deltaTheta / 2) * (deltaH1 / deltaTheta +
sensors.horizontal1.getOffset());
    localY = 2 * sin(deltaTheta / 2) * (deltaV1 / deltaTheta +
sensors.vertical1.getOffset());
  }
```

```
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 incompatable with this function it will return an empty class

This odometry class will be updated to also include the imu on the robot after I work on kalman filtering, which will help improve the accuracy of the odometry
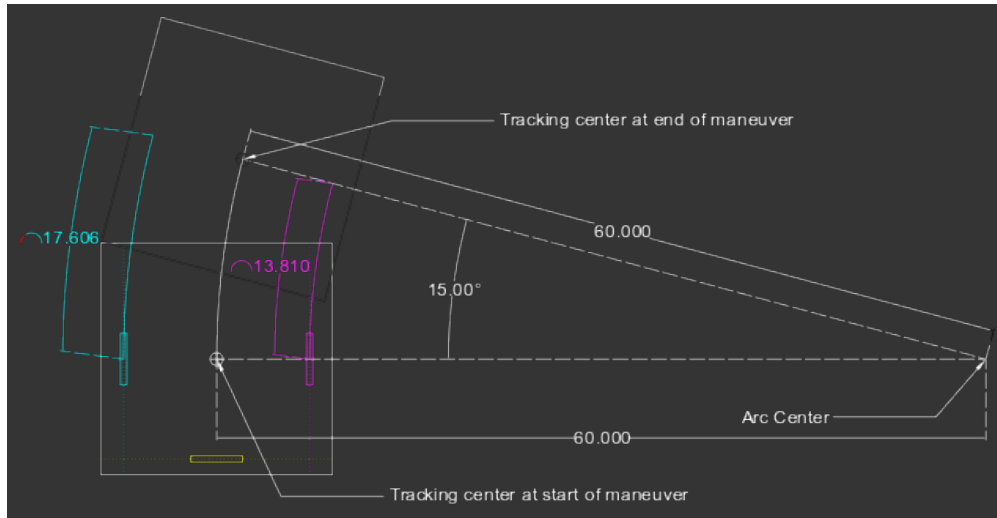
> 💡 **tldr;**
>
> Odometry is a way for a robot to know its global position on the field by using 2-3 free spinning wheels, and using arcs to approximate the path the robot takes.
>
> The global position can be used to allow the robot to self correct even though major disturbances to the path.

Appendix A: Odometry Derivation

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

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

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

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

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

$$\Delta L - s_L\Delta\theta = \Delta R - s_R\Delta\theta$$

$$\Delta L - \Delta R = \Delta\theta(s_L - s_R)$$

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

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 chrod length and just add it to the y translation to make a translation vector:

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

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:

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

Glossary

GUI

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

PID

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

PTO

Honostly I dont know what it stands for, i just know its used to switch which gear a motor is powering with pneumatics

deliminator

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

pose

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

Credits

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