47510W Phoenix Horizon Programming

2024-2025

Contents

Chapter 1

Programming Environments

This section details the way in which we are managing our code for this robotics year.

1.1 Version Control: Git

For our version control system this year we have chosen the popular Git version control system. Git keeps track of each version of our code and also allows us to revert if needed. We can also create multiple branches. Which allows us to work on new code while maintaining the functional code. Also, through the use of GitHub we can easily create a copy of our code and version history which lives on the web. Git also facilitates collaboration between team members working on the drive code. It is for these reasons that we have chosen Git for our version control for the 2024-2025 VRC competition.

1.2 Robot API: Purdue PROS

We have decided to use the Purdue PROS API to interface with our robot. We have chosen this API for its superior performance and documentation over the standard VEX API, we also have more experience with PROS than VEXCODE. PROS also has great support for libraries. PROS does have one big drawback and that is the lack of support for newer Vex equipment such as the AI Vision Sensor. Being a third party software, the PROS development team does not have access to the VEX Software Development Kit(SDK).

Figure 1.1: API Comparison Table

PROS API	VEX API
Superior Performance	Can Be Sluggish
Used Last Year	Used Two Years Ago(First Year)
Readable Documentation	Some Docs Are Poorly Organized
Third Party	Official VEX
Stable	New Features Earlier

1.3 Programming Libraries

For this year we have decided to use the library LemLib. We have chosen this library because it is a popular library for the PROS API. LemLib gives us essay to use PID controllers and odometry algorithms such as point to point and boomerang. In accordance with the REC Code of Conduct, we have made sure that we understand the algorithm behind LemLib. The notebook details each of these algorithms.

Chapter 2

Driver Control

2.1 Drivetrain

On our drivetrain we have decided to use tank drive. With tank drive, each joystick controls one side of the drivetrain. We choose this control schema because it is familiar to our driver. To implement tank drive, we get the joysticks position and feed it to the motors. We use LemLib for controlling the motors, which is included here with an explanation.

```
void Chassis::tank(int left, int right,
    bool disableDriveCurve) {
if (disableDriveCurve) {
    drivetrain.leftMotors->move(left);
    drivetrain.rightMotors->move(right);
}
}
```

This excerpt from LemLib shows the important parts to our code. We choose to disable LemLib's scaling curve, so we can implement our own (Discussed in the subsection). After the if statement we see that the inputs to left and right sides of the drivetrain are passed in and fed to motors. To get the values to pass we use the following lines:

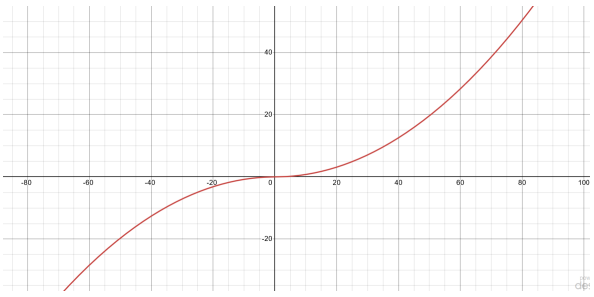
```
master.get_analog(pros::E_CONTROLLER_ANALOG_LEFT_Y)
master.get_analog(pros::E_CONTROLLER_ANALOG_RIGHT_Y)
```

This gives us the y position of the joysticks which we can then feed to the motors.

2.1.1 Input Scaling

Under normal operation the power to the motors will follow a linear curve. The linear curve allows us to reach full power, but it can be difficult to control for fine movements. We can not simply reduce input sensitivity because that would decrease the top speed. Instead, we need to make a scaling curve that has low sensitivity at low values but still reaches the top speed.

We chose the curve shown and defined in Figure **??**. We can see that the function is flatter at lower values and steeper at higher values.



(a) Curve

$$f(x) = \begin{cases} \frac{x^2}{127} & x > 0 \\ 0 & x = 0 \\ -\frac{x^2}{127} & x < 0 \end{cases}$$

(b) Definition of Curve

Figure 2.1: Scaling Curve and Definition

The code for this function is as follows.

```
double inputCurve(double x)
{
    return ((x * x) / 127) * helpers::math::sign(x);
}
```

The sign function returns 1 if greater than 0, and -1 if less than 0. To avoid an error, the sign function is equal to 0 at 0. The first term, `(x*x)/127`, is equivalent to $\frac{x^2}{127}$. Together these form the piecewise function above.

2.2 Controlling Pneumatics

The pneumatic solenoid is controlled by the 3 wire ports on the robot. This means that the brain does not know what device is on the other end, and we must tell it which signal to send. Solenoids are digital out devices meaning they accept an on or off value from the brain. We use the following function

to control solenoids: `<solenoid>.set_value(<state-boolean>);`. When we want to control the solenoid we define a boolean value to hold the current state (on or off) of the solenoid. The most common way to utilize pneumatics in driver control is by toggling them, which is how we control the mobile goal clamp.

2.2.1 Toggling Pneumatics

To create a toggle we change the state of the boolean and send an update to the solenoid. The code for the mobile goal clamp toggle is as follows:

```
void highstakes::Robot::clampToggle()
{
    this->clampState = !this->clampState;
    this->clamp.set_value(this->clampState);
}
```

The first line changes the state of the boolean. The exclamation mark is a boolean NOT, meaning that we want the opposite truth value. This second line sends an update to the solenoid with the new state.

This will work as intended if triggered once, however when a button is pressed it often sends multiple inputs. The multiple inputs by the button trigger the solenoid multiple times, and often results in a failed toggle. We can remedy this issues by detecting when the button is pressed, and not checking for more inputs until the button is released.

We first create a boolean to keep track if the button is pressed. If the button is pressed and the boolean is false, we set the toggle and set the boolean to true. Due to the fact that the boolean is now true, the previous condition will not trigger, and the toggle will not trigger again.

```
if(master.get_digital(highstakes::config::CLAMP_BUTTON)
    && !clamp_latch)
{
    robot.clampToggle();
    clamp_latch = true;
}
// Where clamp_latch is the boolean
```

When the button is released the boolean is set to false, and the toggle can be triggered again.

```
else if(!master.get_digital(
        highstakes::config::CLAMP_BUTTON
        ))
{
    clamp_latch = false;
}
```

Together this code ensures that the toggle works consistently on every button press.

2.3 Controlling the Intake

The intake allows us to pick up rings and move them to the conveyor, which is controlled by the same motor. We need to be able to run the intake in both directions(intake and extake). When the set buttons are held (drivers preference) the code will set the motor voltage to either positive 12 volts or negative 12 volts, respective to intake or extake. When the button is released, the motor in stopped.

Chapter 3

The PID Controller

This section of the notebook will detail the Algorithms that we are using for robot control.

3.1 PID Controller Basics

A PID Controller (Proportional–Integral–Derivative Controller) is a control system that brings the system softly to a target, preventing overshoots and making for a more accurate program. PID controllers are used within our program to make more accurate turns and allow fine control over the amount the robot moves.

3.1.1 How Does a PID Controller Work?

Each term, P I D, each serve a specific role in creating and controlling the input which is used to reach the target. By adding up the values of the terms we get the input. We will analyze the function of each term individually.

P Term The P (Proportional) increases the input proportionally to the distance to the target. In other words, the farther you are from your target the larger the P Term is. P is the error, or distance from the target. We calculate the value of P with the following formula.

$$P = T - M$$

Where T is target and M is the measured current value

The P Term is the primary way of tuning the PID and is always the first step. The specifics of tuning and how we tune each term will be discussed in a later section.

I Term The I (Integral) is the cumulative error. The I term increases as the program runs and increases faster if farther away from the target. Which allows the controller to power over an obstacle, most often friction close to the target. We calculate the value of I with the following formula.

$$I = I_0 + P$$

Where I_0 is the I term from the previous cycle, and P is the current error(The P term)

D Term The D (Derivative) is how fast the error is changing. The D term helps bring the controller out of an oscillating state caused by repeated overshoots, as are often induced by the P term. We calculate the D term with the following formula.

$$D = P - P_0$$

Where P is the error(P term) and P_0 is the previous error (last P Term)

Putting It Together To the get the input that we feed to our system, usually a motor for our purposes, we add together our three terms. Each of which multiplied by its tuning information. Tuning specifics will be covered in the next section. Our input formula is as follows.

$$R = P \cdot K_P + I \cdot K_I + D \cdot K_D$$

Where R is the output(or return), P I and D are the calculated values, and K_P K_I and K_D are the tuning values

3.1.2 Tuning A PID

For a PID to properly reach the target it must be tuned. Inadequate tuning can lead to failure to reach the target or aggressive oscillations. We tune a PID by changing tuning parameters, these parameters are called K_P K_I

and K_D. The first step is to set all tuning parameters to 0. By setting a parameter to zero we can cut out their influence for the time being. As in the previous section we will look at the tuning of each term individually.

Tuning the K_P Term The first step to tuning a PID is to set the K_P term. The first step to setting K_P is to increase it until we get an overshoot and a settle either on the target or settles into an under-shoot. If it settles on the target we are done with tuning and PID should be ready to use. If we settle into an under-shoot we must move onto setting the K_I term. If we overshoot or enter oscillations without settling we need to decrease the K_P term.

Tuning the K_I Term The purpose of changing the K_I term is to fix steady state error after the control settles. As a result of the behavior of the I term to increases over time, it can add input to make the final push to the target even after the distance is not great enough for the P term to have an effect. The K_I does not need to be changed in all application. The K_I term should be increased with caution because increasing it even by small amounts can introduce instability and overshoot.

Tuning the K_D Term The purpose of the K_D term is to control oscillations in the system. We should change K_D when the system has small but unacceptable oscillations. If the oscillations are large, it points to error in the K_P term. Changing K_D is often unnecessary in the context of robotics. We should change K_D to refine out small errors when working with more precise systems such as driving and tuning. However, systems which do not need to be super precise, like the speed of a flywheel, we can forgo tuning this term. Tuning of K_D is a final refinement step to add extra stability or to counteract instability added by tuning K_I.

3.2 PID in Code

When using a PID for an actual application we can use the information described above to create a foundation, but we also need to add some extra parts to help with the imperfections of the real world. The change that must be made is to tell the program when it is "close enough" and it can stop. There is also the problem of feeding the input to the application. First we will look at the parts which are similar to the PID theory above.

3.2.1 Initializing Variables and Setting Up

The first step to creating any new code system is to create an object (See the "Overview of Object-Oriented Programming" to get information on this concept), and create variables we need. For now, we will only create variables related to the theory and not the modifications required for a code implementation. This is all done in the header file (Also see the "Overview of Programming Concepts" for more information). So far, our code looks like the following.

```
in pid.hpp:

class PID_OBJ
{
    public:
        double p;
        double i;
        double d;
        double kp;
        double ki;
        double kd;

        double lastErr;

        PID_OBJ(double kp, double ki, double kd);

        double pidCalc(double target, double current);
        void reset(double target);
};
```

Let's break down that chunk of code a piece at a time. The first thing we see is the class declaration and the public keyword. These are explained in depth in the "Overview of Object-Oriented Programming". For our purposes of explaining PID, they are unimportant. Then we see a list of variables all declared as floating point(decimal) values with double precision(hence the name 'double'). After that we declare 3 functions which we will look at in a moment. First, however, let us analyze the variables.

Variables. We can see that we have many familiar variables which are reminiscent of the equations in the previous section. We also have a new variable `double lastErr` which will keep track of what the error was during the last cycle of the control loop. If you recall from the section of the D term, having access to the last error is important to the calculation of the term.

Functions. Now we will look at the functions we have declared. The first function `PID_OBJ` is a constructor and is called on the creation of a new PID. It takes three arguments,`kp ki kd`, which should be familiar from both the section on tuning and our variables above. These arguments are how we will tune our system. The `pidCalc` function does the operations described in section 3.1.1. Finally, the reset function allows us to reset the terms without creating a new PID. This is important between tasks to prevent interference. First we will look at the constructor and reset functions, then we will look at the `pidCalc` function.

3.2.2 Constructor and Reset Functions

The constructor and reset functions serve to initialize the PID before an external control loop. We will first look at the constructor.

Constructor. The constructor serves to build a new PID object and set tuning information. The code within the constructor is as follows.

```
in pid.cpp:

    this->kp = kp;
    this->ki = ki;
    this->kd = kd;
```

We are simply setting the passed values we saw in the header to internal values in the object which we can access later to do calculations with. This allows us to tune once. However, to use the PID multiple times we need to look at the rest function.

Reset. The reset function sets the values of the PID to be ready for operation. It is absolutely necessary that this function is called before attempting to use the PID. We write the reset function like so.

in pid.cpp:

```
this->p = 0;
this->i = 0;
this->d = 0;
this->lastErr = target;
```

We set each term to zero, so the calculation has a clean slate. Then, we set `lastErr` to our target value because before the robot has moved, we can assume that the error(distance to the target) is equal to the target. This code could also be placed in the constructor, but this creates a "Disposable PID" which can only be used once. By moving the reset code into its own function, we can reuse our PID.

3.2.3 pidCalc Function

The `pidCalc` function is the heart of our PID code. It calculates the P I and D terms and adds them together. Later we will also add code to this function to deal with the inconsistency of the robot. First we will look at calculating our terms. It is worth noting that target and current are passed in from the control outside the function, which will discuss later.

```
this->p = target - current;
this->i += this->p;
this->d = this->p - this->lastErr;
this->lastErr = this->p;
```

We see that we calculate the terms according to the formulas above just with different names. The symbol "+=" after 'i' tells us that we should add

the value on the right to the value on the left. We set `lastErr` after we calculate 'd' because it is no longer relevant to this cycle, and it will be equal to 'p' on the next cycle.

After this we have a line to put the parts together.

```
return (this->p*this->kp)
   +(this->i*this->ki)+(this->d*this->kd);
```

This line instructs the function to add the terms together after multiplying with their tuning information and return the sum as a result. We now have the foundations of a PID. Before we can implement it, we need to add code to deal with the real world.

3.2.4 Dealing With Real World Inaccuracy

Sadly, the robot is not perfect and is not able to make zero tolerance turns Our sensors also have some inaccuracy that makes setting exact values impossible. Due to this fact we need to add code to call close enough when we are within a satisfactory zone. The first step is to add the following variables to our PID object.

```
double acceptableErr;
bool isSettled;
int acceptableCycles;
```

Let's look at what each of these variables will do. The `acceptableErr` variable will be set in the constructor and tells the PID how close is close enough. Like the tuning values, this needs to be set on a case by case basis for each application, however as a general rule values closer to zero will have more accurate results but require more precision. Values to close to zero may never settle. The `isSettled` is a true or false value that will tell the control loop that we have settled and to move on. In a drive train context, the motors will be given stop commands after isSettled is set to true. Finally, the `acceptableCycles` tracks how many cycles we have been in range for.

We start with checking if we are in an acceptable range.

```
if(fabs(p) <= acceptableErr)
```

In this line we are saying that if the absolute value of our error is within the acceptable range we can move on. Otherwise, we go to the else condition which sets the acceptableCycles to zero. This else condition means that we need to be in an acceptable range for a continuous span of time as opposed to passing through it a set number of times. The next line is the next step when we are within acceptable range.

```
if(this->acceptableCycles++ == 15)
  {
      this->isSettled = 1;
  }
```

These lines add one to the acceptableCycles variable (reflecting the fact we just passed the acceptableCycles check) and then checks if we have at least 15 acceptable cycles. If we have reached this target we set is settled to True (represented by 1), otherwise we do nothing. Note that there is nothing special about 15. Any number will work there but larger numbers will be more accurate but take longer to settle.

By putting these parts together we get code that is ready to handle the imperfection of the real world. Our full pidCalc function is as follows. (on next page)

```
in pid.cpp:

this->p = target - current;
this->i += this->p;
this->d = this->p - this->lastErr;
this->lastErr = this->p;

if(fabs(p) <= acceptableErr)
{
    if(this->acceptableCycles++ == 15)
    {
        this->isSettled = 1;
    }
} else {
    acceptableCycles = 0;
```

```
}
return (this->p*this->kp)
    +(this->i*this->ki)+(this->d*this->kd);
```

To use this function to run a system we need to create a control loop. We will use the autonomous tuning code as an example of this.

3.2.5 PID Control Loop Example

What does the control loop do? By its self the PID object is not able to affect the robot in any way. To put our code to work we need to add a control loop. This will take the outputs from the PID and give commands to the robot. We will show an example using autonomous turning code as this is a common use for PID.

Turning Example We should first establish how we will track our current value. On the robot we have a sensor called the inertial sensor. This sensor tracks a number of values but the one we care about is rotation. Rotation tracks how far we have rotated left or right in an angle. We will pass a value `deg` to our turn function which will instruct the robot on how far to turn.

Our first step is to calculate our starting value and reset the PID. We calculate our target with the following formula.

$$T = d + m$$

Where T is target, d is the value deg that we pass, and m is the measured current value

After this we reset our PID, passing the target to the reset function. Our first two lines look as follows.

```
double target = deg + this->readInertialDeg();
this->turPID.reset(target);
```

Note that turPID is the PID that is set up with the tuning information regarding turning. Structure of the autonomous program will be discussed later.

Next we create a loop which will calculate the current PID value and feed it to the motors. The code for that looks as follows.

```
while(true) {
    double calcVal = this->turPID.pidCalc(target,
        this->readInertialDeg());

    this->driveRight(calcVal);
    this->driveLeft(-calcVal);
```

Note that the value we feed to the left motors is the opposite of that fed to the right motors. This is done so the robot will actually turn as opposed to going straight. We pass the value from the inertial sensor as the current measured value to the pidCalc function of the turPID (turn PID).

Next we check if the PID has settled with a quick if statement, which is the following.

```
if(this->turPID.isSettled)
   {
       this->brake();
       return;
   }
```

If the PID has settled the program tells the robot to brake and stop turning. After braking the loop ends and the robot has turned to the correct position. This concludes the section on the PID, and we will now move onto optometry.

Chapter 4

Odometry Theory

4.1 Basics of Odometry

4.1.1 What is Odometry

Odometry allows the robot to keep track of where it is on the field. We use odometry to reduce imperfections in the autonomous routine and robot construction. Odometry works by tracking the rotation of specific wheels and the heading of the robot.

With Odometry we are able to tell the robot to go to a point as opposed to manually setting turns and movement. This allows for more precise programming that can remove errors in real applications.

4.1.2 Odometry Sensors

Working with Odometry requires two kinds sensors. One to track the robot heading, the inertial sensor, and one to track the position of the tracking wheels, the rotation sensor. We could also use a motor encoder, a comparison table of the two follows this.

Rotation Sensor	Motor Encoder
One Smart Motor Point	Two 3 wire ports
More Precise	Less Precise
Less Structure	More structure
Center Screw Pivot	Must have makeshift pivot

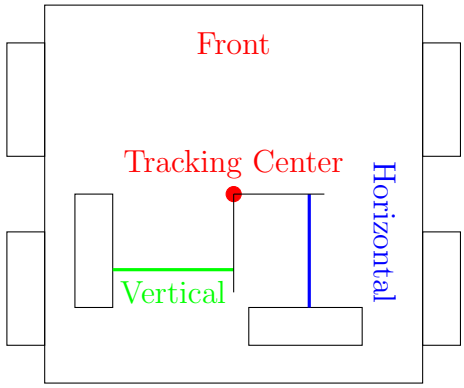
We decided to use the rotation sensor because we benefit from the increased precision and better pivot.

4.1.3 Tracking Wheels

We have decided to use 2.75in wheels for odometry. We chose this size because it is the smallest size we have. It is beneficial to use the smallest wheels available because the take of the least space. We also must measure the distance from the tracking center of the robot to the face of each tracking wheels. See Figure 4.1 below for how we measure tracking wheel offsets.

One final note about the sign of the distance. Based on the position of the wheel relative to the tracking center we make the distance positive or negative. Front or left is positive and back or right is negative.

Figure 4.1: Measuring Odom Offsets



4.1.4 Determining Field Position

Finding Δx, Δy, and $\Delta\theta$

The first step in determining the change in the field position is determining how much the robot heading changed. We use a continuous heading approach meaning that the heading does not reset after completing a rotation. Turing clockwise is repeated by a negative number and turning counterclockwise is represented by a positive number. We find the change in heading with the following formula. We choose the turning center of the robot for the tracking center because reduces potential error.

$$\Delta\theta = \theta_1 - \theta_0$$

Where $theta_1$ is the current heading, θ_0 was the last heading, and $\Delta\theta$ is the change in heading

Next we must get the change on both tracking wheels by using our rotation sensors. We can use the following formulas.

Let $degToIn(\theta)$ is a function which converts a value in degrees to a value in inches using the wheel diameter. (Wheel diameter is assumed in these formulas so is not listed.)

See A.1 for more information about degToIn including a formal derivation.

$$\Delta x = degToIn(R_{x_1} - R_{x_0})$$

Where Δx is the change in the x direction. R_{x_1} is the current position and R_{x_0} is the last position of the wheel

$$\Delta y = degToIn(R_{y_1} - R_{y_0})$$

Where Δy is the change in the y direction. R_{y_1} is the current position and R_{y_0} is the last position of the wheel

Combining Δx, Δy, and $\Delta\theta$ in the current plane

For this section we define a variable robotOdomPos to store the odom position relative to the robot. This variable stores a (X,Y) coordinate.

When $\Delta\theta = 0$ We don't have to do any trigonometry at this step. We can simply store the x, y change into the variable. We can shorthand this with the following equation.

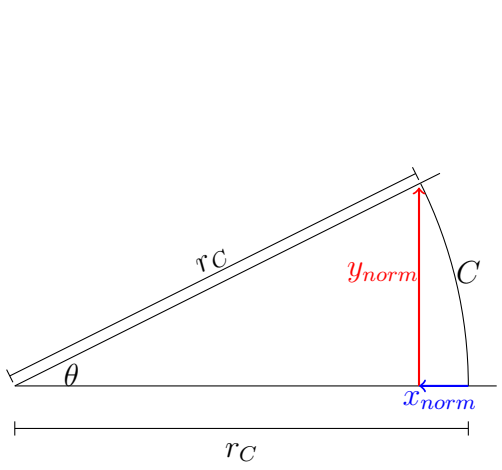
$$(\Delta x, \Delta y) = (X, Y)_{robotOdomPos}$$

When $\Delta\theta \neq 0$ Our job becomes much more difficult if the robot heading changes during the movement. We must use the values from the tracking wheels and the inertial sensor readout to determine the actual change in position. The following is an explanation of this concept.

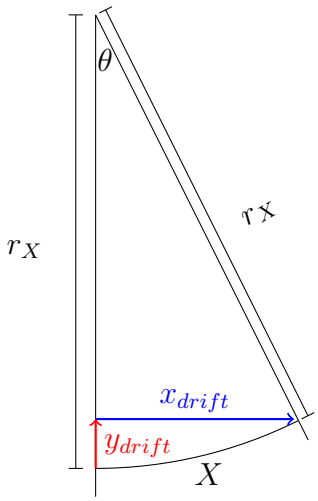
Deriving Formulas to find Δx and Δy in the current plane when $\Delta\theta \neq 0$

Finding the Δx and Δy is done from the following diagrams.

Figure 4.2: *Given C, X, and θ find y_{norm}, x_{norm} and $drift$ counterparts.*



(a) Diagram of normal movement(not accounting drift)



(b) Diagram of drift movement

For the purposes of this section we are assuming the tracking wheels are perfectly in the center of the robot. This is not possible in real robot design, but it simplifies the math. The corrections for real life applications are detailed in Section 4.1.4. The *norm* values do not account for drift and are determined from the y-wheel. C is the value read from the sensor on the y-wheel. The $drift$ values are determined from the x-wheel. Likewise, X is the value read from the sensor on the x-wheel. We will first focus of the *norm* values.

To find the norm values we must find r_C. This value is the radius the arc C, or the arc which the robot travels along(assuming no drifting). We

use the formula for arc length to find the value r_C.

Formula for Arc Length

$$s = \frac{\theta}{2\pi}(2\pi r)$$

Which can then be reduced by canceling 2π:

$$s = r\theta$$

Where A is the arc length, θ is the internal angle of the arc in radians, and r is the radius of the arc.

We can substitute C and r_C and solve for r_C.

$$C = \theta r_C$$

Dividing theta from both sides.

$$\frac{C}{\theta} = r_C$$

We also add the distance from the y-center of the robot to the wheel in actual application. We assume that distance is zero for theory purposes.

By using r_C we can find values for x_{norm} and y_{norm}. We can do this by using sine and cosine. Recall that on the unit circle $\sin\theta$ is the y value of a certain point and $\cos\theta$ is the x value of a certain point. Recognizing that Figure 4.2a invokes the unit circle of radius r_C, we see that $y_{norm} = r_C \sin\theta$. We also find that $x_{norm} = -(r_C - r_C\cos\theta)$. We must subtract from the radius because we wish to find the x distance from the start position to the current position, not from the origin to the start position. We also must make this value negative because the robot x is moving in the negative direction (Direction of movement is shown by the arrows in Figure 4.2a). Finally, we can factor out r_C to simplify the formula for x_{norm}. So finally we come to the following two formulas for our *norm* values.

$$y_{norm} = r_C \sin\theta$$

$$x_{norm} = -r_C(1 - \cos\theta)$$

To find the drift values, we first must recognize that Figure 4.2b is a rotation 90° clockwise of Figure 4.2a. The computation for r_X is identical to that of r_C just with values swapped for the different diagram. This means that we use cos for y_{drift} and sin for x_{drift}. Contrary to the *norm* values, neither term including r_X is negative. This is seen in Figure 4.2b as both arrows are pointing positive. So using the formulas from the *norm* values, we derive the following formulas.

$$x_{drift} = r_X \sin\theta$$

$$y_{drift} = r_X(1 - \cos\theta)$$

The last step of this process is to add the values together. We find each $\Delta value$ in the current plane by adding the corresponding *norm* and $drift$ values together.

$$\Delta x = x_{norm} + x_{drift}$$

$$\Delta y = y_{norm} + y_{drift}$$

We can then expand these values using our equations from the previous two subsections. Thus,

$$\Delta x = -r_C(1 - \cos\theta) + r_X \sin\theta$$

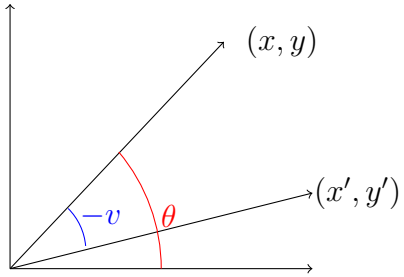
$$\Delta y = r_C(1 - \sin\theta) + r_X(1 - \cos\theta)$$

Q.E.F

Rotating Into The Field's Plane

We have now found the change in position, but it is from the robots previous frame of reference. We must rotate the point from the robots frame of reference to the fields frame of reference. We know the heading of the robot relative to the fields, so we can use it to find the new point. We will be referencing Figure 4.3.
Figure 4.3 depicts a clockwise rotation. It is important to note that the rotation from the robots frame of reference to the fields frame is a rotation counterclockwise, this means that we rotate the point clockwise. The way to think of the logic here is to imagine rotating an object counterclockwise.

Figure 4.3: Point Rotation Diagram



Reset the object then imagine orbiting your head around the object clockwise. The object looks the same after both rotations. With this knowledge, the following is a derivation of formulas for x' and y'.

Given x,y,θ, and v; find x' and y'

$x = r\cos\theta$	Convert to Polar
$x' = r\sin(\theta + (-v))$	Polar Rotation
$x' = r\sin(\theta - v)$	Simplify
$x' = r\cos\theta\cos v + r\sin\theta\sin v$	Cosine Angle Subtraction Identity
$x' = x\cos v + y\sin v$	Substitute
$y = r\sin\theta$	Convert to Polar
$y' = r\sin(\theta + (-v))$	Polar Rotation
$y' = r\sin(\theta - v)$	Simplify
$y' = r\sin\theta\cos v - r\cos\theta\sin v$	Sine Angle Subtraction Identity
$y' = y\cos v - x\sin v$	Substitute

Thus, we have found that $x' = x\cos v + y\sin v$ and $y' = y\cos v - x\sin v$. *Q.E.F*

By applying this function to the inertial sensors previous position and our values for Δx and Δy we get our absolute change in field position; represented by Δx_{abs} and Δy_{abs}. It must be noted that we must multiply the value of the inertial sensor by -1 to make the value negative, this is so because the inertial sensor measures clockwise but we want counter-clockwise rotation.

Final Steps

Finally, we must add our absolute change values to the robots absolute position; represented by x_{avg0} and y_{avg0}. So the following formulas are true.

$$x_{avg1} = x_{avg0} + \Delta x_{avg}$$

$$y_{avg1} = y_{avg0} + \Delta y_{avg}$$

This concludes this section. In the coming sections we will put this knowledge into practice to control the robot.

Chapter 5

Moving the Robot with Odometry

5.1 The pointToPoint function

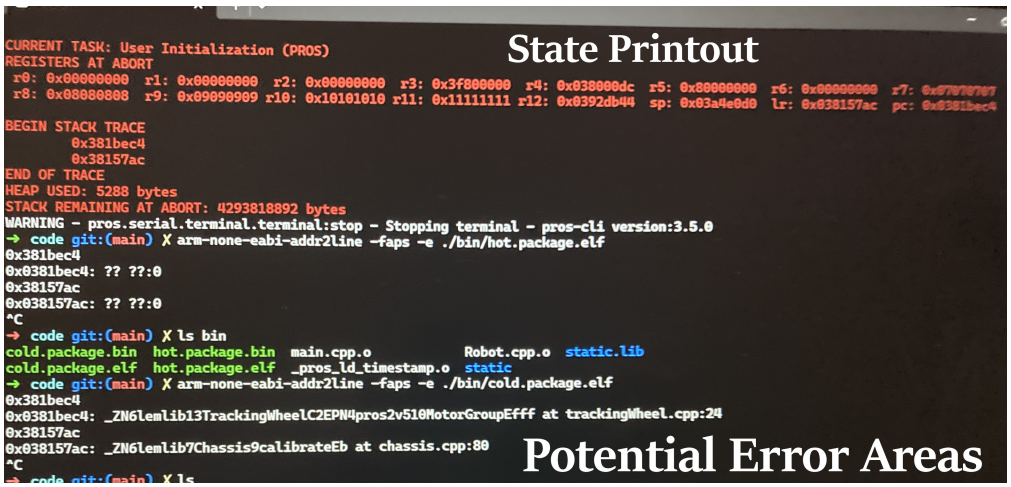
Chapter 6

Adventures in Debugging

6.1 Brain Crashes 2024-8-19

We ran into a runtime error during initial testing of our program. As soon as the program began it would crash. It gave us limited information, including a state printout. A state printout is where the program simply reports the current memory addresses on the CPU and runtime stack. See Figure 6.1 for an example.
By using a tool bundled with our cross-compiler we were able to locate possible error zones (Figure 6.1) Both of these error zones were within library files, which we do not have easy access to. The best course of action was decided to be to reinstall LemLib. This made no difference. Still thinking that this was a problem with the library, we tried installing an old version of LemLib. This also made no difference. After talking with another team and collaborating on the debugging process, we found that the problem was with the order of initialization in the Robot class. It seems to of stemmed from a misunderstanding in the use of tantalizer lists. We thought that the initializer list would begin at the top and step down, however it seems that the initializer list looks at the order of declarations in the class declaration and does it in that order. Due to the fact that we had declared chassis before its component pieces, the program crashed. We fixed this by moving the chassis declaration to the bottom of the declaration stack. Correct and incorrect code can be found in Figure 6.2.

Figure 6.1: State Printout and Potential Error Areas



Incorrect Initialization Order

```
class Robot
{
   Chassis chassis;
   Drivetrain drivetrain;
   ControllerSet PIDS;
   Sensors odom_sensors;
}
```

Correct Initialization Order

```
class Robot
{
   Drivetrain drivetrain;
   ControllerSet PIDS;
   Sensors odom_sensors;
   Chassis chassis;
}
```

Figure 6.2: Initialization Order(Simplified)

The appendix will detail background information and helper functions which are not specific to robotics.

Appendix A

Derivations for Helper Functions

A.1 Deriving the degToIn Function

The degToIn function uses basic circle geometry to convert the degrees a wheel rotates into the distance the wheel moves in inches using the wheel diameter. This function hinges on the principle that a wheel moves a distance equivalent to its circumference for each full rotation. We define circumference with the equation:

$$C = d\pi$$

Where C is the circumference and d is the diameter of the circle.
This formula gives us the distance that the wheel moves if the wheel moves one rotation at a time. Sadly, though, in real life the wheel moves in fractional increments. We can represent this fractional rotation by multiplying the circumference by the same fraction. We find this fraction by dividing the measured valued by the amount of one rotation, in this case 360 degrees. We can use this fraction of a rotation to get the distance by multiplying it by the wheel circumference. So we use the following formula.

$$D = \frac{\theta^\circ}{360} \cdot C$$

Where D is the distance the wheel moved, θ° is the degrees the wheel rotates, and C is the circumference of the wheel.

Finally, we add this function into our code. We use M PI to put π into the code. The function goes as follows:

```
double degToIn(double deg, double wheelDiam)
{
    return (deg * (M_PI / 360.0)) * (wheelDiam);
}
```

Q.E.F