2018-19 FTC 3216 Autonomous and Control

Musings of an Overworked Programmer

Edward Yang

February 2019

Contents

1 Introduction

This year we fully intend to learn from the experience and mistakes that we've had from previous years. The autonomous portion of the game is fairly straightforward, and we have aimed to do every part of it: delatching, sampling, claiming, and parking. The majority of our time has been spent perfecting the ins-and-outs of the autonomous section and the control of the robot in general. We've reused and improved on much of the code of previous years, this time enhancing the modularization of the code and adding more sane defaults, while also integrating more sensors and simplifying unnecessarily complex parts. We've made our robot move more precisely with PID controllers and tuning, and gave it the ability to "see" with gyroscopes, limit switches, new I2C sensors, and computer vision algorithms. Motion is smoother with addition of new control schemes and greater usage of odometry, both autonomously and when the robot is controlled by a driver. In my time on 3216, I fully intend to continue push both my own and my team's knowledge of these fields to the limit.

Tons of opportunities for enhancement still lie in wait: from more advanced control algorithms like Ramsete [6] to greater customization of software, the sheer number of interactions of robotics with fields like control theory and software are vast and daunting. There's still far too much to be learned and done that's possible in a year, but hopefully the next generation of my team will be able to take the torch and go from here. Programming isn't isn't all just about raw, brute efficiency: sometimes it's just as much about the beauty and aesthetics of it all. At times, simply seeing a clean, simple structure and the clarity of thought behind it, like in some of the assignments I've graded, is enough to make it all worthwhile.

2 General Program Structure

In general, we split our code into 3 major sections: `autonomous`, `common`, and `components`. The `autonomous` section holds, as the name would imply, our autonomous code, which includes the command system structure, implemented commands, the main autonomous program, and the starting-position specific opmodes to run them. The `common` section holds some of the code that is reused throughout the program, including vector math global constants, and enumerations like `SamplingConfiguration` or `StartLocation`. Finally, the `components` section holds all of our hardware or software components: parts that can be split and modularized for ease of testing and usage. A graphical overview can be found in appendix A.

3 Autonomous

We accomplish all of the main tasks in our autonomous: de-latching, sampling, claiming, and finally parking. However, I also intend on accomplishing these tasks in a "clean" way: this means well-structured, easy-to-read code, modularization, and maximum efficiency.

3.1 Abstractification for Great Good

Over time, we've realized that there are many common repeated actions: movement to a specific location, sampling, de-latching, etc. This has led us to abstract over these tasks, as detailed in the next subsections.

3.1.1 Command System

This year, we've decided to implement a command system. What this means is that commonly used sequences of code are split into their own class. For example, the code for moving to a certain location is all put into `MovementCommand`. Each one of these extends the abstract `Command` class, which defines an abstract method `executeCommand` that is used by a defined method `execute`, which allows them to be initialized with their own sensible set of parameters while maintaining a common interface. `execute` takes in all of the current robot component classes as its parameters, and so each `Command` can use any of those provided component classes: e.g. `NavigationalState` or `Drivebase`. Continuing the example, the `MovementCommand` is initialized with its own target direction and vector location, and then is executed with the `execute` method common to all `Command`s. This is the structure that all of the `Command`s follow:

```
public class MovementCommand extends Command {
    // Command-specific fields
    private VectorF targetPosition;
    private float targetHeading;
    private boolean forward;

    // allows for appropriate/efficient constructors
    public MovementCommand(float x, float y, float targetHeading, boolean
        forward) {
        this(new VectorF(x, y), targetHeading, forward);
    }
```

```
    public MovementCommand(VectorF targetPosition, float targetHeading,
        boolean forward) {
        this.targetPosition = targetPosition;
        this.targetHeading = targetHeading;
        this.forward = forward;
    }

    void executeCommand(NavigationalState navigationalState,
        InertialSensor imu, VisionProcessor visionProcessor, MainRobot
        mainRobot, Telemetry telemetry) {
        ...
    }
}
```

With this common interface, we can simply create a list of `Command`s and iterate through that list, calling `execute` on each one. Following from this idea, we simply have a separate list of `Command`s for each starting location, which are well-defined and easily understandable. An example list of these `Command`s is given below:

```
Command[] commandList = new Command[] {
    new BeginCommand(), // run beginning code
    new HookControlCommand(ElevatorHook.State.FullyExtended), // fully
        extend the elevator
    new MovementCommand(609.6f, -609.6f, -45, true), // move to (609.6mm,
        -609.6mm), face -45, move forwards
    new SampleCommand(), // sample
    new MovementCommand(525f, -525f, -45, false), // move to (525mm,
        -525mm), face -45, move backwards
    new MovementCommand(1524f, 304.8f, 90, true), // move to (1524mm,
        304.8mm), face 90, move forwards
    new MovementCommand(1524f, 1219.2f, 90, true), // move to (1524mm,
        -1219.2mm), face 90, move forwards
    new ClaimCommand(),
    new MovementCommand(1524f, -609.6f, 90, false), // move to (1524mm,
        -609.6mm), face 90, move backwards
    new ArmDeployCommand(), // deploy the arm
    new FinishCommand() // run finishing code (especially cleanup)
};
```

As one can clearly see, this list of `Command`s simply runs something at the beginning, then fully extends the hook, moves, samples, moves, claims, moves, deploys

the arm, and finishes.

3.1.2 Compatibility with the FTC Stop System

This system of having separate components and separate `Command`s is incredible for clarity and modularization; however, it does not come without its downsides. The FTC app relies on `LinearOpMode`s to take into account the state of the output of `OpModeIsActive()` method in order to stop the program on time. If this variable is not taken into account and the program is not ended soon after the state of the `OpModeIsActive()` method is changed to false, the FTC app simply crashes. Because this method is only accessible from the `LinearOpMode` class and all of the components and `Command`s do not necessarily have access to the class that is using them, we are faced with a problem of stopping on time. In order to rectify this issue, we decided to spawn another thread that would check the output of the method in question and change a global variable `OPMODE_ACTIVE` that is accessible by the components and `Command`s. For thread-safety, this variable is wrapped in a `Mutex` which ensures that changes are synchronized throughout threads.

3.1.3 Asynchronous Programming

The increased modularization we have introduced in the previous sections, coupled with an update to the phones that allows us to use Java 8, has given us an interesting opportunity in the form of the possibility of easy asynchronous programming. With Java 8 comes `CompletableFuture`, which allows the creation of `CompletableFuture`s that can be executed either with a threadpool or simply asynchronously on one thread. A `CompletableFuture` represents the result of an asynchronous computation, and provides many methods to chain, join, or run them in parallel. This is far simpler and easier to keep track of than keeping state variables for each and every action that we want to do, and lets us create arbitrary execution trees of `Command`s that we want to run in a combination of serially and in parallel. These `CommandTree`s can then be simply parsed and run as a series of `CompletableFuture`s. However, because of the difficulty of debugging asynchronous commands and general time constraints, the `CommandTree` system has been largely left unused despite its completion. In our first competition, we had issues within our code that were hidden under many layers of callbacks and were not immediately obvious when they showed up, as the error logs only indicated that the threadpool had broken somewhere, and not where or in what computation.

3.1.4 Future Improvements

In the future, we fully intend on continuing our current modularization approach, and potentially modelling the components in a similar way to the `Command` system: one single abstract `Component` class that would provide both wrappers and a shared interface that each `Component` would implement. The `ArmDeployCommand` is still to be implemented, and could be helpful in providing a set way of deploying our arm without direct control from the driver, saving us driver-controlled time. There is definitely some way of working around the FTC app stop system that is better than our current approach, likely through the usage of callbacks. Finally, the asynchronous system is likely to be pushed into production code, as this would allow us to bring down the elevator on our robot while the robot performs other actions, saving us valuable time during the autonomous section, and potentially even in the driver-controlled section as well. This would involve further expansion on the debugging process present with this method and having a better way to initialize a tree than nested initialization calls.

3.2 I Know De Way: Path System

We have briefly touched on our motion commands in the previous section. However, it is worth having an in-depth discussion of our motion and path system on its own. Our motion commands all work with an absolute coordinate system. This means that any `MovementCommand` we run takes in absolute coordinates and angles, and so makes the adjustment or creation of autonomous paths far easier. After the `MovementCommand` is written, the most we have to do is write in the numbers that correspond to the wanted target positions on the field. No extra calculations with relative coordinates have to be done by hand: the program does it for us. The way we implement this system is fairly simple. at the beginning, we know the absolute position and orientation of the robot can only be of 4 set positions, which is known by which `OpMode` is run. We store this information (position and heading) in a `NavigationalState` as a vector and a scalar, respectively. Then, whenever we want to move to a new position, we find the needed orientation and position changes, and use a transformation matrix determined by the robot's current orientation and position to convert this displacement from field-relative to robot-relative coordinates. After the motion is complete, we update the `NavigationalState` with the target position and orientation.

3.2.1 Future Improvements

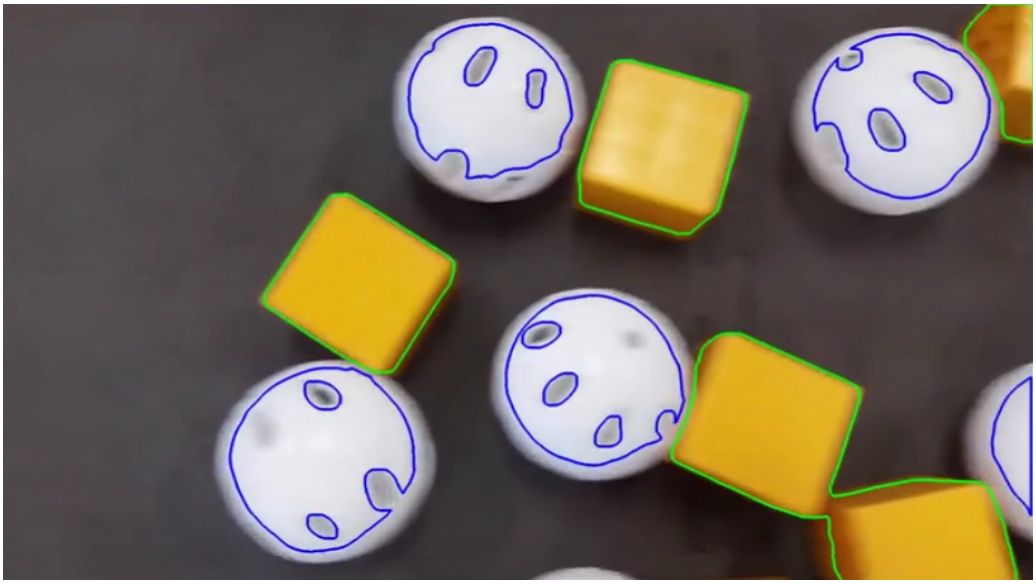
There's still much to be done in this respect. Lots of methods are available for smoother or faster motions, such as the usage of actual trajectories and splines to plan paths, especially for a 2DOF robot. We intend on reading through the code provided and perfected by ACME Robotics (#8367), the Road Runner library [4], which implements all of these wanted capabilities with adequate documentation. In addition, the current system assumes that the robot perfectly reaches its target location, which many times is not true. This should be easily solvable by using the odometry vales rather than the target values for the update after movement. Finally, in the far future we want to do some experimentation with SLAM libraries (specifically ORB2-SLAM [2]) to give our robot navigational capabilities in case of disturbance. This idea is elaborated upon in section 3.5.

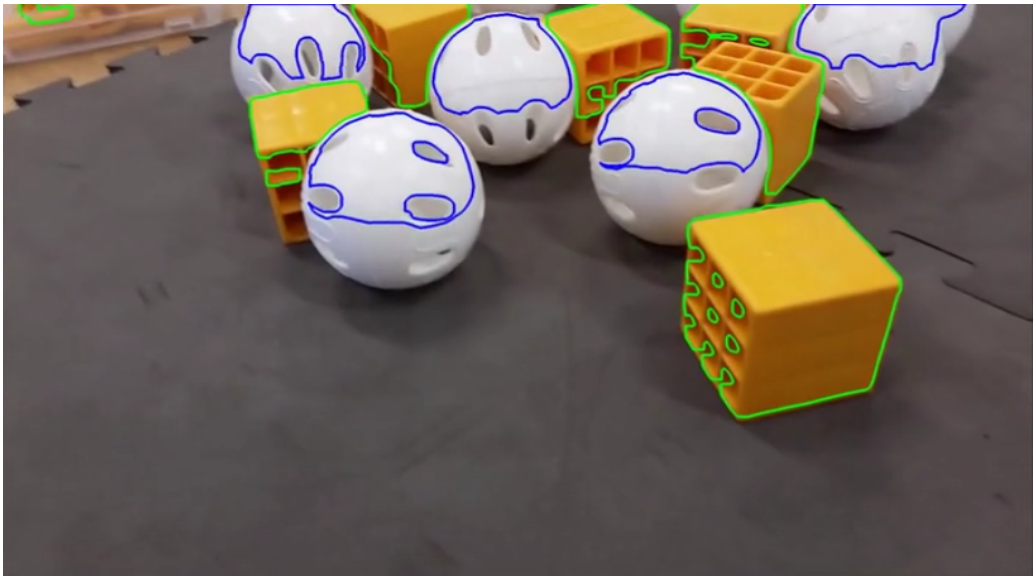
3.3 I Can See Clearly Now: Sampling

Sampling initially seemed like the hardest part of this year's game, with materials that didn't conform to the usual RGB cutoff values and the necessity of detecting multiple items. However, with the existence of OpenCV and the later release of builtin TensorFlow Lite code and models for the FTC SDK, this task has grown to be fairly straightforward.

3.3.1 Initial OpenCV Experimentation

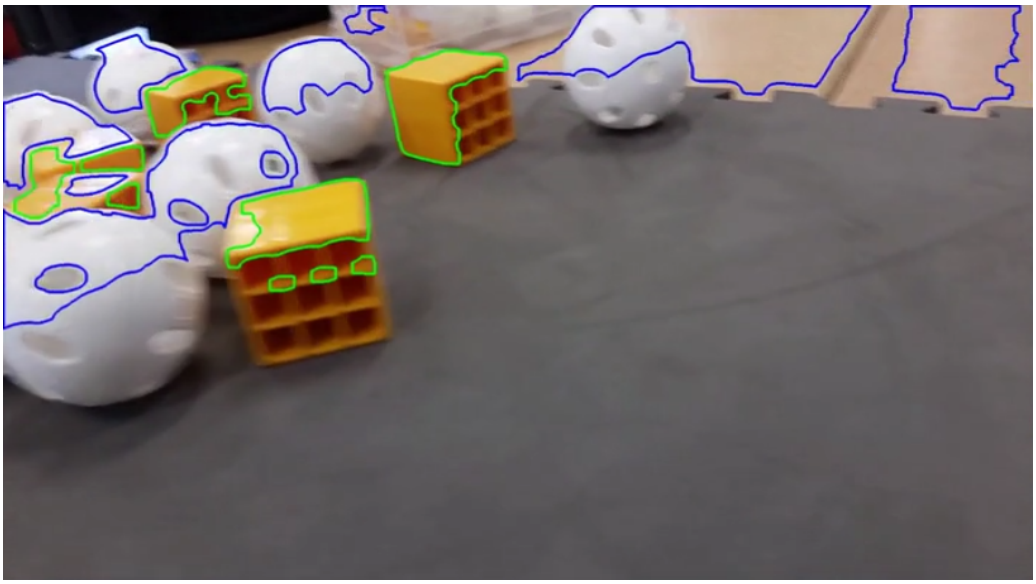
With the release of the game came multiple images and videos of the field and game elements provided by people such as Sohom Roy. Prior to the game, we had knowledge of the GRIP [5] program, provided by WPI, which allows for fast prototyping of OpenCV algorithms along with live preview of the pipeline on sample images. Upon suggestion by other teams, we started with experimenting with thresholding and filters on the HSV, YUV, and LAB color spaces, which gave fairly good results when combined and merged together, contoured, and filtered by area. Some example results are shown below:





3.3.2 OpenCV Issues

However, this approach still has caveats. For one, when the lighting was darker or lighter than expected, less than the required amount or huge chunks of the field would be detected as balls and cubes. An example of this issue is shown below:



We tried to solve this problem with the watershed algorithm, which essentially erodes, contours, and then re-expands groups of pixels in the image. Even then, the results were questionable, but they seemed to work well enough for the given parameters (few objects, spaced far apart, in ok lighting). In trying to add in the watershed algorithm, we found that the GRIP pipeline had issues with its implementation, so we had to export our existing code and then add in the watershed code manually.

3.3.3 The Current State (TensorFlow Lite)

Around a month after the release of the game, our buddies at FTC decided to release an update to the FTC SDK that included builtin TensorFlow Lite support and a pretrained model, which made sampling almost trivial. The major concern with this method is increasing the accuracy of object detection, generally by improving lighting conditions or adjusting the camera's distance from the minerals. During this time, we tried with the phone's flashlight both on and off, with varying amounts of success, and have added a webcam in order to have a closer sample. A bare minimum object detection program is shown below that demonstrates how simple the usage is:

```
TFObjectDetector tf; // initialization sequence omitted
List<Recognition> updatedRecognitions = tf.getUpdatedRecognitions();
if (updatedRecognitions != null) {
    for (Recognition recognition : updatedRecognitions) {
        if (recognition.getLabel().equals(LABEL_GOLD_MINERAL)) {
            // process a gold mineral
        } else {
            // process a silver mineral
        }
    }
}
```

3.3.4 Future Improvements

The most that can be done on this front is perhaps an integration of OpenCV algorithms for preprocessing with TensorFlow, or our own model, which has potential to further increase accuracy. However, annotating and tuning the model is extremely time-intensive, which is why improving this aspect of our autonomous is fairly low on our priority list.

3.4 I've Gotta Feeling: Sensor Integration

This year, our robot has grown to be more of a well-informed citizen than our robot from last year. It has gained a sense of direction and control in the form of limit switches, gyroscopes, and encoders, which allow it to be far more effective in the autonomous mode.

3.4.1 Limit Switches

Currently, there is one limit switch mounted on our elevator with 3 triggers, which allow the robot to extend and contract the elevator on command without going too far up or down. This essentially eliminates any sense of encoder drift or inaccuracy, and has proven to be extremely effective no matter what we change about our elevator (switching from chain to string, rack and pinion to linear slide). This system is described easily with a state machine, with 3 on-switch states and 2 off-switch states that are in-between the on-switch states. By simply giving the direction of motion and the initial state of the elevator, we can maintain knowledge of the current position of the elevator and constrain its movement.

3.4.2 Gyroscope

This year we've begun to use the gyroscope to get our orientation relative to our starting position, which has allowed us to gain greater accuracy and remove much of the error introduced by encoder drift in both straight driving and turning. With this sensor feedback, we have effectively started to gain more variables that we can sense and control through PID feedback, opening the doors for far faster and more accurate motion. Compared to the previous encoder-based method, our turning and straight-line motion is far more accurate and consistent, and allows the robot an accurate way of self-correction.

3.4.3 Vuforia

While not used this year because of the lack of necessity, we have still retained and updated our Vuforia code from last year. Our Vuforia code allows us to get the absolute position of the robot on the field through a series of transformation matrices, and so may be used later in order to track and further eliminate motion error.

3.4.4 Future improvements

In the future, we plan to add more limit switches and a gyroscope to the arm, in order to allow automatic deployment of the arm. There is a potential to add distance sensors on the sides of the robot for collision detection, but from what we've seen, this is not yet necessary.

3.5 The Circle of Life: PID and General Control

PID control [7] is perhaps one of the most fundamental methods of feedback control in control theory. In essence, the error, derivative of error, and integral of error are linearly combined in order to get the final amount by which to tune whatever process is capable of changing the error, packed beautifully into the below equation:

$$u(t) = K_{\mathrm{p}} e(t) + K_{\mathrm{i}} \int_0^t e(t')dt' + K_{\mathrm{d}} \frac{de(t)}{dt}$$

In this way, the robot can adapt to minor changes due to disturbance and rely more on its odometry than by dead reckoning in order to get both consistent and accurate results.

3.5.1 Usage

On our robot, we use PID control for a lot of subsystems: forward motion, turning, and arm movement, and fully intend on making it used by more. The major usage is in forward motion and turning. In forward motion, each motor has its own PID controller, and there is another PID controller running on the error of the gyroscope. The output of all three of these PID controllers are combined to get the motor powers wanted in order to keep the robot driving straight. In turning, we have only the gyroscope PID controller, which controls both motors and allows for precise turning without overshooting, thanks to the integral coefficient.

3.5.2 Tuning by "Experienced Personnel"

In tuning, we chose to go with the manual method in order to gain more experience with the process, and because we don't have a sufficiently accurate physical model of our robot to use some of the automatic methods. The manual method consists of first setting K_{i} and K_{d} to 0, and then increasing K_{p} until the system reaches steady oscillations [7]. After this point is reached, K_d is increased until the system is critically damped, and then K_{i} is increased until the system overshoots very little or does not overshoot at all. This whole process was made far easier with the help of the FTC Dashboard [3] software made by ACME Robotics (#8367), which allows for live hot-swapping and tuning of variables, especially those of K_{p}, K_{i}, and K_{d}.

3.5.3 Issues

Over the course of the tuning process, we've learned that PID control is extremely hard to execute when the voltages are constantly changing. Over the course of the

match, voltage is steadily dropping, and so unless voltage is constantly increased, the tuned coefficient values can make the robot over- or undershoot even when it works at another voltage.

3.5.4 Future Improvements

To solve the aforementioned issues, we intend to add a voltage feedforward coefficient to the motors in order to account for differences in voltage. In addition, we are considering using better tuning methods in order to increase the speed that we tune at (we can tune about one PID controller a day). But even then, PID or even PIDF has its limitations. To this end, we plan on learning more control theory to the point that we have a basic understanding of Ramsete and can implement it, which is touted to be virtually impervious to outside disturbances when given good odometry, and can account for the physical constraints of our robot in terms of size, maximum acceleration, maximum velocity, etc.

4 Driver Controlled

4.1 Towards a Sensor-Assisted Driver

Even with the brain power that humans have, a driver can't do everything at once. Accounting for error and making motion smooth can go a long ways towards improving the interaction between hardware, software, and driver.

4.1.1 Learning to Dual Wield: Control Layout

Our control layout was finalized as soon as our strategy was finalized, and is as follows:

- **Left Joystick** Drive/Turn
- **Right Joystick** Drive/Turn
- **DPad** Arm Rotation
- **R/L Bumper** Arm Extension
- **R/L Trigger** Intake Control

One may notice that both the left and right joysticks control the same thing. This is intentional, as it allows the driver to use one side of the gamepad to drive while using the other side to control other parts of the robot, effectively allowing control to be switched on a whim. In addition, the joystick inputs are taken to the 5th power: this is to preserve sign and in order to account for the nonlinearity of the motors. By taking the inputs to a higher power, lower values have less of an effect while higher values gain more effect, which maps better to the logarithmic sensory processing that humans do.

4.1.2 The Elevator

Because all of the modules and components used by autonomous are also available to the driver, we are able to use the same limit switching system in driver-controlled without any additional code. This means that the driver can move the elevator in a constrained manner to prevent undue strain.

4.1.3 Arm Control (WIP)

Controlling the arm has been one of the biggest challenges that the software side has faced. Because our arm rotates on an axis perpendicular to gravitational force, it has nearly constantly changing force applied on it. Even the simple PID control has trouble dealing with this. In order to remedy this, we intend on implementing sensors on the arms that will detect the angle of the arm from horizontal, θ, and the angular velocity of the arm, ω and feeding this forward into a control algorithm. The form of this theoretical algorithm is given below:

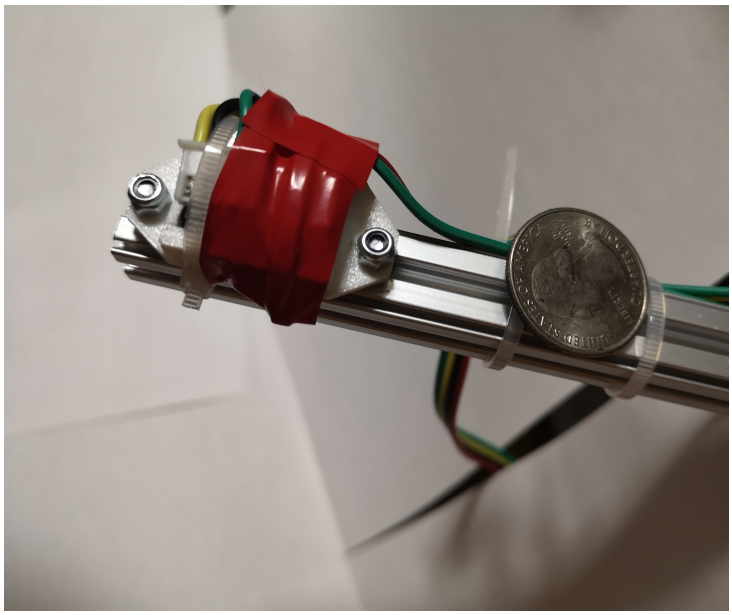
$$P_{\text{arm}} = K_{\text{p}}\epsilon_\omega + K_{\text{d}}\frac{d\epsilon_\omega}{dt} + K_{\text{i}}\int_0^T \epsilon_\omega dt + K_\omega\frac{1}{\omega} + K_\theta \cos\theta$$

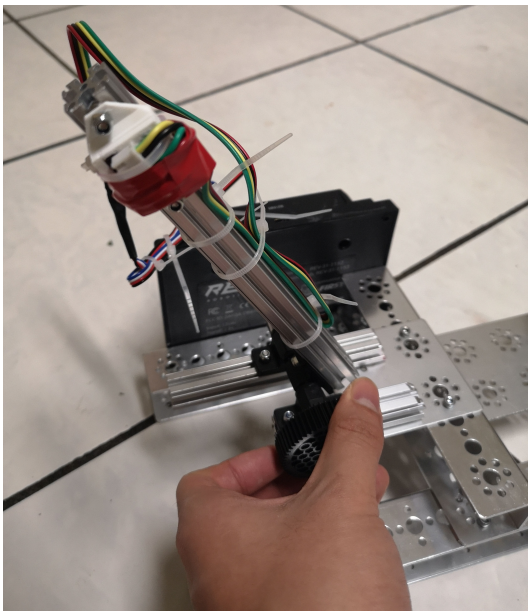
The first part of this is a normal velocity PID, but then there are two additional terms. The K_ω term should account for the motor speed-torque tradeoff, while the K_θ term should account for the change in gravitational force due to angle. However, this requires a gyroscope small enough to be mounted on the arm, which is discussed next.

5 Writing an I2C Driver

5.1 Introduction

In order to add a gyroscope to the arm, an MPU-6050 gyroscope was purchased and prepared hardware-wise by soldering on headers, printing a case, and creating the correct I2C wires. In addition, a physical test-bed to replicate its usage on the arm was made. However, because this sensor has no pre-existing Java I2C driver, we had to implement it ourselves with the help of the FTC I2C Guide.





5.2 Register Mapping

The first thing we did was find the register mappings on the sensor that we wanted. These were easily found through the official register mapping document provided by the manufacturer. The register mappings we used are listed below.

Addr (Hex)	Addr (Dec.)	Register Name	Serial I/F	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1A	26	CONFIG	R/W	-	-	EXT_SYNC_SET[2:0]			DLPF_CFG[2:0]		
1B	27	GYRO_CONFIG	R/W	-	-	-	FS_SEL [1:0]		-	-	-
1C	28	ACCEL_CONFIG	R/W	XA_ST	YA_ST	ZA_ST	AFS_SEL[1:0]				

3B	59	ACCEL_XOUT_H	R	ACCEL_XOUT[15:8]							
3C	60	ACCEL_XOUT_L	R	ACCEL_XOUT[7:0]							
3D	61	ACCEL_YOUT_H	R	ACCEL_YOUT[15:8]							
3E	62	ACCEL_YOUT_L	R	ACCEL_YOUT[7:0]							
3F	63	ACCEL_ZOUT_H	R	ACCEL_ZOUT[15:8]							
40	64	ACCEL_ZOUT_L	R	ACCEL_ZOUT[7:0]							
41	65	TEMP_OUT_H	R	TEMP_OUT[15:8]							
42	66	TEMP_OUT_L	R	TEMP_OUT[7:0]							
43	67	GYRO_XOUT_H	R	GYRO_XOUT[15:8]							
44	68	GYRO_XOUT_L	R	GYRO_XOUT[7:0]							
45	69	GYRO_YOUT_H	R	GYRO_YOUT[15:8]							
46	70	GYRO_YOUT_L	R	GYRO_YOUT[7:0]							
47	71	GYRO_ZOUT_H	R	GYRO_ZOUT[15:8]							
48	72	GYRO_ZOUT_L	R	GYRO_ZOUT[7:0]							
6B	107	PWR_MGMT_1	R/W	DEVICE _RESET	SLEEP	CYCLE	-	TEMP_DIS	CLKSEL[2:0]		
6C	108	PWR_MGMT_2	R/W	LP_WAKE_CTRL[1:0]		STBY_XA	STBY_YA	STBY_ZA	STBY_XG	STBY_YG	STBY_ZG
75	117	WHO_AM_I	R	-	WHO_AM_I[6:1]						-

These registers were each one byte long, which differed from the original FTC and had to be modified to fit. In addition, some values were offset bit-wise, which meant that we had to use bit-shifts to set and get the correct values.

5.3 Register Caching

The I2C device generally should have cached/batched reads, as individual register reads are extremely expensive and slow. However, as we would later find out, the maximum register address space that can be cached by the builtin `I2cDeviceSynch` is only 26 register addresses long, and so we couldn't just cache the entire register space. Instead, we chose only the accelerometer and gyroscope registers (`3B` to `48`) to batch read, as those were the registers that we would be consistently reading at a high frequency.

5.4 Generalization (Parameters)

The MPU 6050 sensor is actually highly customizable in its settings, in that it allows for modification of the sensitivity of its measurements. For example, the accelerometer can be set on a range of $\pm 2g$ $\pm 4g$, $\pm 8g$, and $\pm 16g$, depending on the values written to its configuration registers. We attempted to generalize these settings by creating enumerations representing all of the possible values in each

configuration bit or register, and then asking for a set of these enumerations, or `Parameters` on initialization. Some of the included configuration options are:

- `EXT_SYNC_SET` External Frame Synchronization bit location
- `DLPF_CFG` Digital Low-Pass Filter Configuration
- `FS_SEL` Gyroscope Sensitivity
- `AFS_SEL` Accelerometer Sensitivity
- `TEMP_DIS` Temperature Sensor Switch
- `CLKSEL` Clock Selection (internal vs gyroscope vs external)

5.5 Unresolved Issues

For the most part, there were actually very few bugs in our experience coding this driver, the cache buffer-length error being one and an issue with the reset being another. The register that is supposed to allow the device to reset is, according to the documentation, supposed to be switched back to 0 after successfully resetting. However, when we tested this, the switch simply never happened, and so we haven't yet been able to figure out how to reset the device, although bug-fixing is in progress.

5.6 Sensor Fusion (WIP)

By themselves, the accelerometer and gyroscope provide very poor readings of the gyroscope, with the accelerometer doing well in stable positions and being utterly demolished in motion, while the gyroscope's variance tends to accumulate into angular error when integrated over long periods of time. To solve this, we intend to use sensor fusion algorithms in order to get the best of both worlds. We consider a model where the sensor is mounted on an arm at an angle θ from the horizontal and rotating with an instantaneous angular velocity ω. The sensor is located a distance r from the axis of rotation, and has its y-axis orthogonal to the radial vector while the x-axis points towards the right of the y-axis. Let the values experienced by the sensors be a_y, a_x, and ω, being y-axis acceleration, x-axis acceleration, and angular velocity, respectively. We can apply a complementary filter to the values of $\theta_{\text{accelometer}}$ and $\theta_{\text{gyroscope}}$ in general:

$$\theta = K_a\theta_{\text{accelerometer}} + K_g\theta_{\text{gyroscope}}$$

For some values K_a and K_g such that $K_a + K_g = 1$.

First, we consider getting the values of θ from the sensors themselves in a simplified version. By inspection:

$$\theta_{\text{accelerometer}} = \arctan(\frac{-a_y}{a_x})$$

$$\theta_{\text{gyroscope}} = \int_0^T \omega(t)dt$$

In discretized form for easier translation to code:

$$\theta_{\text{gyroscope}} = \theta_{t-1} + \omega(t)dt$$

Applying the complementary filter to this and substituting, we find the discretized form:

$$\theta_{\text{simple}} = K_a \arctan(\frac{-a_y}{a_x}) + K_g(\theta_{t-1} + \omega(t)dt)$$

Now we move onto a more complex variant of this, where the values are affected by both centripetal and gravitational accelerations $\vec{a_c}$ and $\vec{a_g}$ respectively. The gyroscope remains unaffected. These can both be absolutely represented in vector form:

$$\vec{a_c} = -\omega^2 r \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}$$

$$\vec{a_g} = \begin{bmatrix} 0 \\ -g \end{bmatrix}$$

The vector sum of absolute external accelerations on the sensor is thus

$$\vec{a_{\text{net}}} = \begin{bmatrix} -\omega^2 r\cos\theta \\ -\omega^2 r\sin\theta - g \end{bmatrix}$$

To convert from absolute to sensor coordinates, we can use the transformation matrix T:

$$T_\theta \vec{a_{\text{net}}} = \vec{a_{\text{sensor}}}$$

where

$$T_\theta = \begin{bmatrix} -\sin\theta & -\cos\theta \\ \cos\theta & -\sin\theta \end{bmatrix}$$

Solving for θ, we find that

$$\theta_{\text{accelerometer}} = \arctan(\frac{\omega^2 r - a_y}{a_x})$$

This simplifies to our original equation when the arm is not moving. Our final sensor fusion equation is thus:

$$\theta_{\text{complex}} = K_a \arctan(\frac{\omega^2 r - a_y}{a_x}) + K_g(\theta_{t-1} + \omega(t)dt)$$

5.7 Credits

During this process, the FTC I2C Driver Writing Guide [1] was extremely helpful - without it we wouldn't know where to start.

6 Teaching

6.1 Philosophy and Objectives

To me, teaching is extremely important. It's a way to show that I actually know what I'm talking about. It's a way to pass down knowledge and experience to the next generation. But more importantly, it's also how I get to see the up-and-coming programmers on the team think and reason.

"Missing an edge case—but otherwise good idea."

"Could use some work: have you considered using a boolean instead?"

"Good idea, I hadn't thought of that!"

These were some of the comments I wrote while grading my first robotics programming assignment. As the lead programmer for my school's robotics team, I was and still am in charge of training the new programmers on the team. My first assignment was fairly simple: given an "elevator" which could move up and down, and two sensors at different heights, I wanted my teammates to write a program to move the elevator to one of those heights. But grading those assignments turned out to be far more eye-opening than I expected.

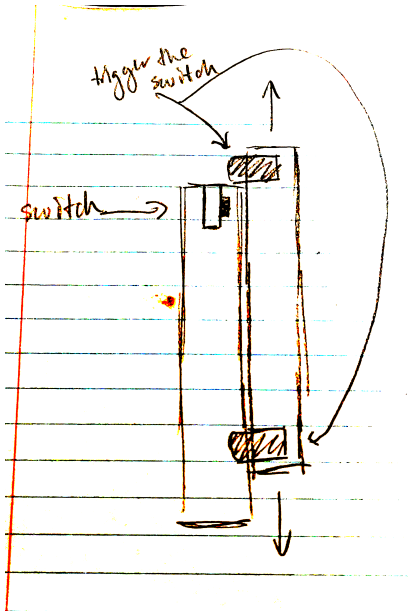
Before I started grading, I had a notion of how I would solve the problem: use a boolean to keep track of whether the sensor is triggered, and repeatedly check for when it did to determine when to stop. While some of the solutions I saw were close to this, none of them were ever "exactly" what I had envisioned.

Some of the solutions took advantage of how the elevator took a certain time to move. Others used different variables. Some of these methods I had thought of before, but others I had not. Some even had better ideas than my own. And although some of them were wrong, each and every one of their solutions was unique and had its own reasoning.

Grading these assignments has given me insight into how diverse my teammates' ideas are and how they implement them. While I value my own opinions and ideas, as a leader and a teacher I strive to maintain a balance between my own and others' ways of thinking. Every comment I leave on their assignments has the potential to shape how they problem solve in the future and potentially how they teach others. Because of this, I place an extremely large emphasis on preserving the eccentricities of the ways my teammates think. It's only when we develop a diversity of ideas that we can truly innovate.

6.2 First Assignment: 2-Stop Elevator

The first assignment was to implement my first elevator design, which was a single limit switch with two triggers. The objective was to implement two methods, `extend` and `contract` which would set the elevator to an extended or contracted state, respectively. It was given that the elevator started in a fully contracted state. The code can be found in appendix B.1.



6.3 Second Assignment: Drivebase Math

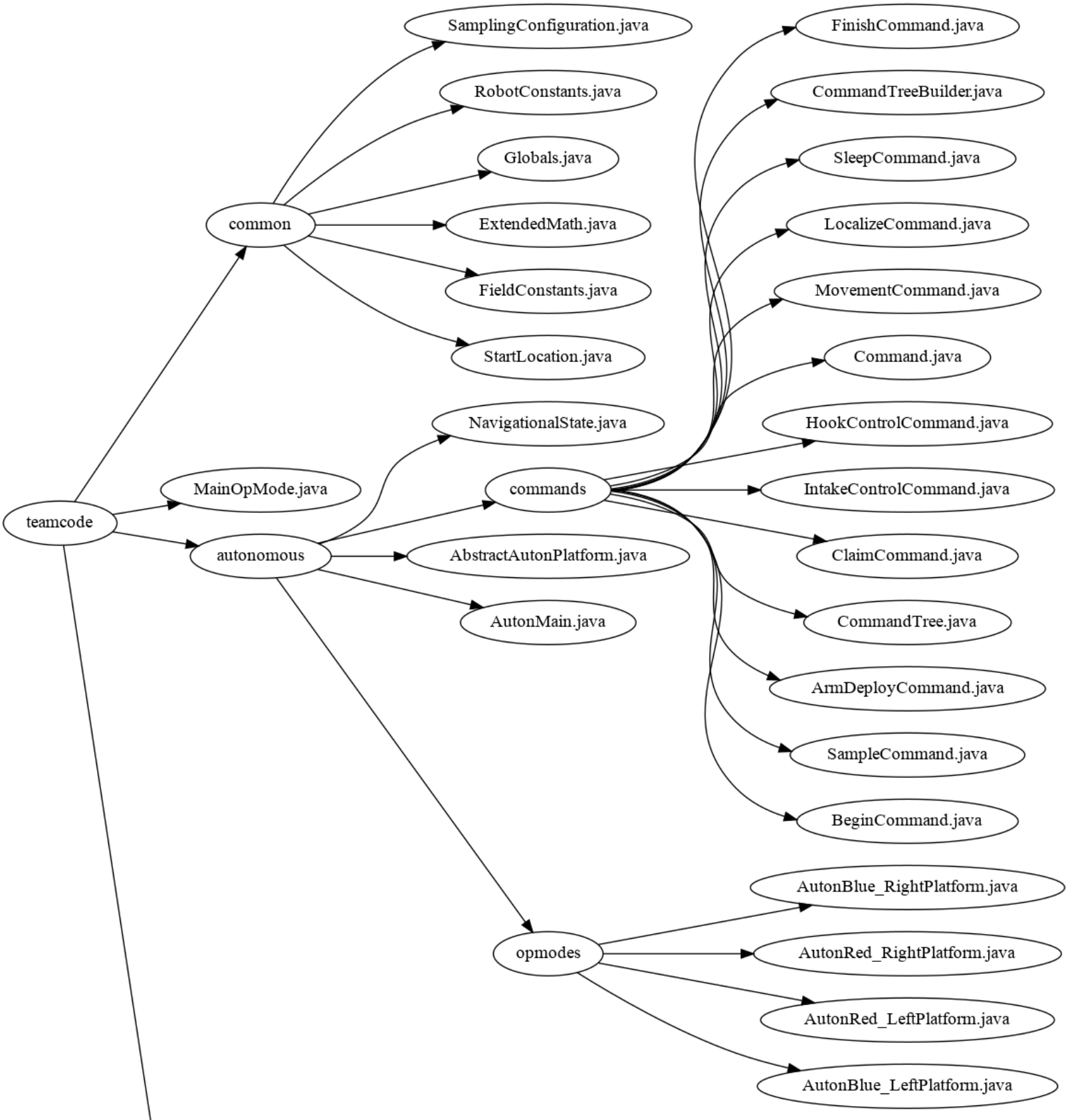
The second assignment [8] involved implementing the main math function for autonomous driving, which was to convert and get the relative movement given the target movement. It mirrors the actual code very closely in order to give some sense of realism. I asked the students to specify mathetmatical helper functions to normalize angles, and then to define the sequence of calculations and movements that would correctly move the robot to the target location and heading. The code can be found in appendix B.2.

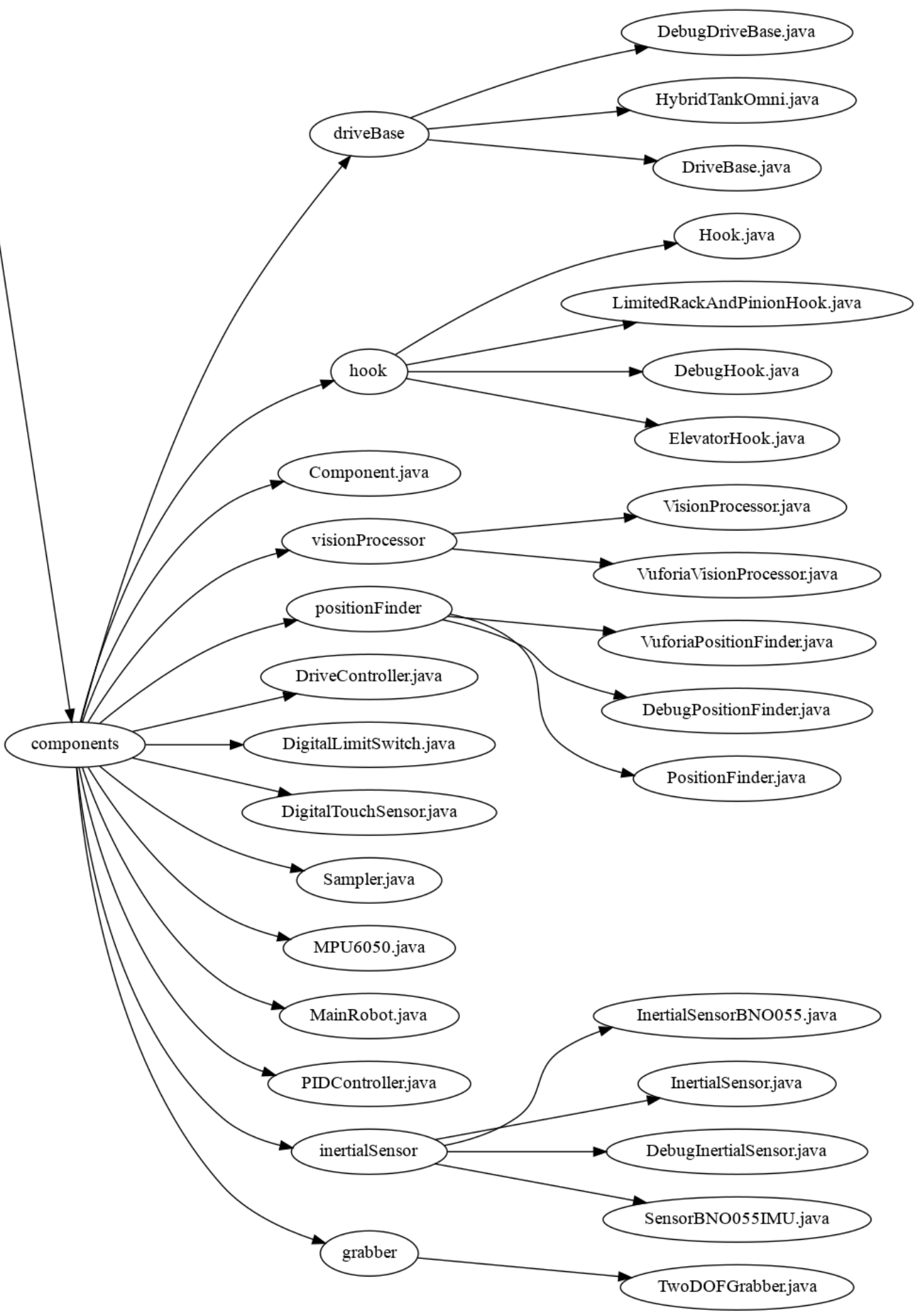
6.4 Third Assignment: Further Implementation of Drivebase (WIP)

The third assignment is intended to build upon the second one, leading students towards PID control and explaining some of the theory behind it.

A File Hierarchy Diagram

The total code is far too large to put in, so we have included a diagram of the overall structure of our project on the next page.


SamplingConfiguration.java
RobotConstants.java
Globals.java
common
ExtendedMath.java
FieldConstants.java
StartLocation.java
NavigationalState.java
MainOpMode.java
commands
teamcode
autonomous
AbstractAutonPlatform.java
AutonMain.java
FinishCommand.java
CommandTreeBuilder.java
SleepCommand.java
LocalizeCommand.java
MovementCommand.java
Command.java
HookControlCommand.java
IntakeControlCommand.java
ClaimCommand.java
CommandTree.java
ArmDeployCommand.java
SampleCommand.java
BeginCommand.java
AutonBlue_RightPlatform.java
opmodes
AutonRed_RightPlatform.java
AutonRed_LeftPlatform.java
AutonBlue_LeftPlatform.java


components
driveBase
DebugDriveBase.java
HybridTankOmni.java
DriveBase.java
hook
Hook.java
LimitedRackAndPinionHook.java
DebugHook.java
ElevatorHook.java
Component.java
visionProcessor
VisionProcessor.java
VuforiaVisionProcessor.java
positionFinder
VuforiaPositionFinder.java
DebugPositionFinder.java
PositionFinder.java
DriveController.java
DigitalLimitSwitch.java
DigitalTouchSensor.java
Sampler.java
MPU6050.java
MainRobot.java
PIDController.java
inertialSensor
InertialSensorBNO055.java
InertialSensor.java
DebugInertialSensor.java
SensorBNO055IMU.java
grabber
TwoDOFGrabber.java

B Assignment Code

B.1 Assignment 1

```
public class RackAndPinionWithLimitSwitch {
    // NOTE: you're given that the rack and pinion starts in a fully
        contracted state,
    // with the switch initially pressed.
    //
    // GIVEN FUNCTIONS: ignore the body, just consider input and output as
        if they were true,
    // the dummy body is just to make it compile so its easier to work on

    // sets the motor speed (up is 1, down is -1, stop is 0)
    private void setMotorSpeed(int speed) {}
    // gets whether the switch is currently pressed
    private boolean switchPressed() {return false;}



    // WORKSHEET
    // add your stuff here (variables, private functions, body of extend
        and contract, etc)

    // move the rack and pinion to a fully extended state
    public void extend() {

    }

    // move the rack and pinion to a fully contracted state
    public void contract() {

    }
}
```

B.2 Assignment 2

Drivebase.java

```
public interface Drivebase {
    // assume that all of these functions work perfectly
    void rotate(float angle); // angle in radians
    void forwardDrive(float x); // x in mm
}
```

ExtendedMath.java

```
public class ExtendedMath {
    // TODO: implement this
    public static float normalizeRadians(float radians) {
        // should take in any number of radians, and output the equivalent
            in the range [-pi, pi]
        // do NOT use a loop.
        // hint: maybe make another function called normalizeRadians2pi
            which normalizes to [0, 2pi] first
        return 0;
    }
}
```

NavigationalState.java

```
public class NavigationalState {
    private float heading;
    private Vector2F position;

    public NavigationalState(float heading, Vector2F position) {
        this.heading = heading;
        this.position = position;
    }

    public float getHeading() {
        return heading;
    }

    public Vector2F getPosition() {
        return position;
    }
```

```
    public void setHeading(float heading) {
        this.heading = heading;
    }

    public void setPosition(Vector2F position) {
        this.position = position;
    }
}
```

Robot.java

```
public class Robot {
    // assume that these are set/nonnull
    private Drivebase drivebase;
    private NavigationalState navigationalState;

    public void moveTo(float x, float y, float heading) {
        moveTo(new Vector2F(x, y), heading);
    }

    // TODO: implement this
    public void moveTo(Vector2F targetPosition, float targetHeading) {
        // move to the given position and heading
        // you should only have to use things that are within these files
    }
}
```

Vector2F.java

```
public class Vector2F {
    private float x;
    private float y;

    public float getX() {
        return x;
    }

    public float getY() {
        return y;
    }
```

```
    public Vector2F(float x, float y) {
        this.x = x;
        this.y = y;
    }

    // for all of these functions, you are allowed to use java Math
        builtins only.
    // TODO: implement this
    public Vector2F subtract(Vector2F subtrahend) {
        // return this - subtrahend
        return null;
    }

    // TODO: implement this
    public float getAngle() {
        // get the angle CCW from the +x axis
        return 0;
    }

    // TODO: implement this
    public float getMagnitude() {
        // get the magnitude of the vector
        return 0;
    }
}
```

This assignment was graded with this grading scale:

- **Style** consistent spacing, naming (camelCase), indentation, structure, etc. - 20 points
- **Clarity** comments explaining your methodology and thought process, reasonable variable naming - 20 points
- **Functionality** does everything work, is it as specified, and if it is a "good"/correct solution - 60 points
- **Total** 100 points

Bibliography

[1] FTC Engineering. Writing an I2C Driver. https://github.com/ftctechnh/ftc_app/wiki/Writing-an-I2C-Driver.

[2] J. M. M. Montiel Raul Mur-Artal, Juan D. Tardos and Dorian Galvez-Lopez. ORB-SLAM2. https://github.com/raulmur/ORB_SLAM2.

[3] ACME Robotics. FTC Dashboard. https://acmerobotics.github.io/ftc-dashboard/.

[4] ACME Robotics. Road Runner. https://github.com/acmerobotics/road-runner.

[5] WPI Robotics. GRIP computer vision engine. http://wpiroboticsprojects.github.io/GRIP/.

[6] Tyler Veness. Practical Guide to State-space Control.

[7] Wikipedia. PID Controller. https://en.wikipedia.org/wiki/PID_controller.

[8] Edward Yang. auton_drive_learn. https://github.com/efyang/auton_drive_learn.