FIRST Robotics Team 399 Eagle Robotics
2012 Software Overview

By Jeremy Paul Germita

Spring/Summer 2012

Revision 1.3: 6/6/2012

Abstract

This writeup was written in conjunction with the completely revamped version of the code for FIRST team 399's 2012 season robot, X-1. The redoing of code was prompted by a review of old code. This review facilitated a reimplementation of the elements that worked well and a removal of unnecessary elements. Much motivation for this project was through the Japanese philosophy of Kaizen, or improvement for the better.

Contents

Section I. Overall Structure

Section overview

This section will discuss the overall structure of the code. Topics include language used and code architecture.

A. Language

The language used is Java. Java was chosen after past experience and success with the language. FIRST team 399 has used java since the language's first introduction into the FIRST Robotics Competition in the fall of 2009, where we served as Java Language beta testers. For the following competition season (2010 - Breakaway), the use of Java was decided by a combination of factors including ease of development, speedy compilation and debugging, and members' past experience with object oriented languages.

Java was chosen for the third consecutive year due to similar factors and with similar success. There have been some setbacks in using the language, but they only proved to improve our mastery of it.

B. Object Oriented Programming

Object Oriented programming is a practice in which “objects” in code may be separated into different classes to allow for easy reuse of code. The Wikipedia article

(http://en.wikipedia.org/wiki/Object-oriented_programming) on OOP has a good explanation of the concept.

This software was developed with object oriented programming fully in mind. All subsystems of the robot are separated into their own separate classes with all related functions available for use by any portion of code that requires it. While at first glance it seems unorganized and cumbersome to deal with code in many files, through implementation, it proves to be very intuitive to group often used code with their related mechanisms.

Section II. Systems

Section overview

This section will discuss the different systems on the robot and their software components. This section is organized with progressively more complex systems towards the end of the section

A. Intake

Mechanism Overview:

The intake system is an amalgation of the bridge manipulator and conveyor belt system. The bridge manipulator's pneumatic actuators are controlled by a solenoid. The conveyor's Banebots RS-550 motor is controlled by a software linearized Victor motor controller.

The intake class contains all of the code to run the combined intake/dropper mechanism. This includes functions to operate the motor and actuator in this system. This class is most often accessed by the Robot and AutonInterpreter classes

Relevant Files:

- Intake.java
  - in package org.team399.y2012.Robot.systems

B. Drivetrain

Mechanism Overview:

The drivetrain system is comprised of four 4” wheels(the center two are dropped 1/8”) on each side. These are driven by dual CIM motor Super Shifter gears with the 6:1 high gear ratio and 24:1 low gear ratio. The four CIM motors in the drivetrain are controlled by 4 Black Jaguars via the CAN bus. The actuators for shifting are controlled by one solenoid.

The drivetrain class provides basic functionality, including tank and arcade drive methods, for the drivetrain system on the robot. The drivetrain also contains various alternative drive algorithms for competitive driving.

The AutoDrivetrain class provides various automated tasks for the drivetrain.

These classes are most often accessed by the Robot and AutonInterpreter classes.

Relevant Files:

- Drivetrain.java
  - in package org.team399.y2012.Robot.systems
- AutoDrivetrain.java
  - in package org.team399.y2012.Robot.controls.Automation

C. Turret

Mechanism Overview:

The turret system is comprised of a Denso Window motor controlled by a Closed PID loop

running on a Black Jaguar via the CAN bus. The PID loop takes input from a Vishay-Spectrol 10 turn potentiometer.

The turret provides basic access to the CAN commands used to control the PID control loop on the Jaguar. This offloads all turret positioning to the Jaguar rather than the cRIO allowing for easy tuning and maintenance.

The AutoAimController class provides automated aiming and positioning for easy targeting on the field. It makes use of the vision system(discussed later) code and basic trigonometry to determine the proper angle to turn the turret to. For example, if the target is X pixels offset from the center and the target's straightline distance from the camera is D feet, one could derive the angle offset like so:

$$\sin(\theta)=\frac{opp}{hyp} \rightarrow \sin(\theta)=\frac{X}{D} \rightarrow \theta=\sin^{-1}(\frac{X}{D})$$

This offers a huge improvement over a previous method of centering the target. The previous method would use the target's X coordinate as an input to a PID controller. The derivative of the pid controller's output was added to the current position of the turret. That sum was then the input for the positional PID loop running inside of the Jaguar. This inefficient algorithm could be expressed as:

$$PID_{cRIO}(X)=(K_p\times e(t)+K_i\times\int e(t)\,dt+\frac{(K_d\times e(t))}{dt})$$

$$output(x)=PID_{jag}(Position_{current}(t)-PID_{cRIO}(x))$$

As is visible, the position passes through 2 PID loops. Also, the Camera's framerate is slow compared to the execution of this loop(30 frames per second vs 50 Hz execution rate). Those two factors add together to produce very laggy tracking/positioning.

These classes are most often accessed by the Robot and AutoAimController classes.

Relevant Files:

- Turret.java
  - in package org.team399.y2012.Robot.systems
- AutoAimController.java
  - in package org.team399.y2012.Robot.controls.Automation

D. Shooter

Mechanism Overview:

The shooter system is comprised of two Fisher Price 0673 motors and 1 pneumatic actuator. The motors provide power to the shaft driving a pair of 6 inch wheels to propel the balls through the air. The actuator moves the hood system to alter the balls trajectory by changing the exit angle.

The shooter class provides velocity calculation and control loops to the rest of the program as well as code to operate the pneumatic hood. The class uses custom developed algorithms to derive velocity and output power to maintain a set velocity.

The velocity calculation algorithm passes the first derivative of the position of the shooter wheel (derived from a digital encoder mounted on the shooter) with respect to time through an Infinite Impulse Response Low Pass Filter to filter out noise. That value is scaled up or down as necessary to produce a clean velocity reading that only yields +/- 150 RPM of noise. With the shooter's max actual speed being 4100 RPM, that is only +- 3.6% of noise.

The velocity PID control loop produces an output power based on past, present, and predicted errors in velocity along with the desired velocity. A velocity PID loop is very similar, but also different from a traditional positional PID loop. With a P-PID loop, the Proportional term would approach zero as the

error approaches zero. That is very undesirable for a V-PID loop as in order to maintain a velocity, power must be continually supplied. Also, with a P-PID loop, the Integral term approaches infinity as the summed error over time approaches infinity. This is something that is desireable in a V-PID loop. Remember the definition of velocity: the change in position divided by the change in time. If time increases, change in position must also increase to reach that velocity. Thus, in a Velocity PID loop, the roles of the P and I terms are reversed. It makes tuning intuitive to swap them within the algorithm rather than mentally while tuning.

With that in mind, our Velocity PID algorithm is as follows:

$$out(t) = (K_p \times \int e(t)) + (K_i \times e(t)) - (K_d \times (de\frac{(t)}{dt})) + (K_f * V_{set})$$

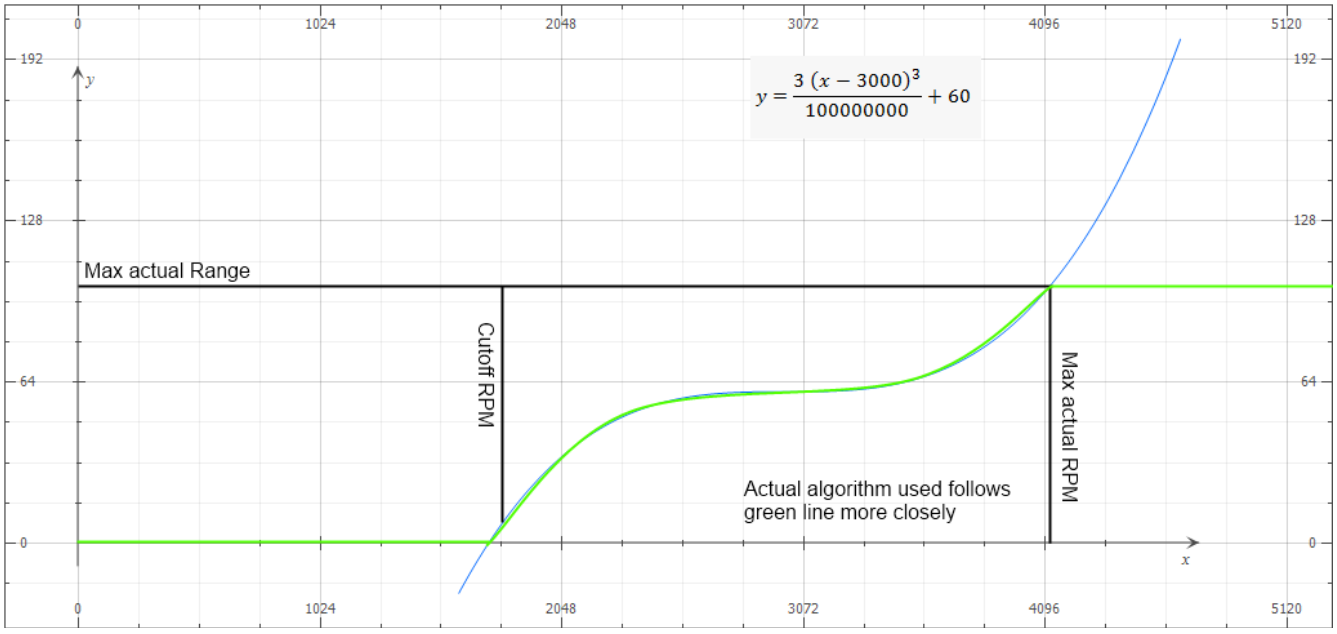
Where e(t) is the error as a function of time, Kp, Ki, Kd, Kf are tuning constants, and Vset is the desired set velocity. The Kf term acts as a safety net to keep the shooter spinning in the event of an encoder failure as it assumes a linear relationship between the set velocity and output speed.

The AutoShooterSpeedController class provides functions to convert a distance value into a shooter velocity value. It is as follows:

$$distance(v) = \frac{(3 \times (v - RPM_{mid})^3)}{10^8} + Distance_{max}$$

$$RPM(d) = \sqrt[3]{\frac{(d - Distance_{max}) \times 10^8}{3}} + RPM_{mid}$$

Where RPMmid is 50-75% of the maximum shooter RPM, Distancemax is the maximum distance in inches. The functions were derived based on a graph of data similar to the following:



The AutoshootController class provides an automatic shooting routine using sensors on the shooter and the intake system. When commanded to shoot, it will check to see if there is a ball in posession inside the intake. If there appears to not be, it will run the intake upwards until a ball appears to be posessed. Once a ball is posessed, the code checks the shooter speed. If there is a very small amount of error between set and actual speed(very close to the target speed), it runs the intake further to shoot the ball.

Relevant Files:

- Shooter.java
  - in package org.team399.y2012.Robot.systems
- AutoShooterSpeedController.java
  - in package org.team399.y2012.Robot.controls.Automation
- AutoShootController.java
  - in package org.team399.y2012.Robot.controls.Automation

E. Vision

Mechanism Overview:

The Vision system is comprised of an Axis camera and an RGB light ring. The light ring is controlled by a custom 3 channel LED driver board to allow for simple control of the light ring's color.

The EagleEye Vision System is one of the more sophisticated pieces of software developed by team 399. It is simpler to explain the system in terms of the “journey” an image takes through the vision system. First, the RGB Light Ring is commanded so illuminate the targets with a specific color. The camera acquires the image and it gets processed within ImageProcessor.java. While there, the target data goes through a series of tests: color, rectangularity(is it closer to a rectangle or a rhombus?), size, and aspect ratio(does the width vs height ratio closely match those of a real target?). After being processed, the program now has data on anywhere between 0 and 4 targets. This data includes distance, X and Y positions, and size. Other sections of the program can use this data to manipulate various components.

The EagleEye system includes an RGB light ring that allows us to set any one of up to 16 million colors, but through testing have found 3 colors to be sufficient. Red is excellent for ranges less than 7 feet. It isn't as intense, so it doesn't appear to “wash out” at very close ranges. Although because of this, it very quickly fades out as distance increases. Green is very good for most effective ranges(between 4 and 25 feet) as the camera's sensor is more sensitive to the color. Blue is very effective at long ranges(> 10 feet). But it is not very effective at short ranges because it starts to

oversaturate the image, making the targets appear very bright white rather than blue.

In the event that EagleEye loses its target, it will wait 5 seconds for it to come back into view. If it doesn't reaquire a target in that time, it shifts to the next color in the list. The sequence goes Red → Green → Blue.

Related Files:

- EagleEye.java
  - in package org.team399.y2012.Robot.systems.Vision
- ImageProcessor.java
  - in package org.team399.y2012.Robot.systems.Vision
- Camera.java
  - in package org.team399.y2012.Robot.systems.Vision
- Color.java
  - in package org.team399.y2012.Robot.systems.Vision
- LightRing.java
  - in package org.team399.y2012.Robot.systems.Vision
- Threshold.java
  - in package org.team399.y2012.Robot.systems.Vision

Section III. Controls

Section overview

This section will discuss the automated portions of the robot.

A. AutonFile and AutonInterpreter

These two classes work together to read from a text file stored on the cRIO to memory and interpret the contents to run an autonomous program. The advantage to this method is the ability to create autonomous programs without really programming. Editing files can be done on any platform with a text editor. IOS, Android, Windows, Mac OS, Windows Phone, all flavors of Linux, etc. The scripting language is very simple for a human to read and write. More information on this in the "Autonomous Commands and syntax" section.

Relevant Files:

- AutonFile.java
  - in package org.team399.y2012.Robot.Controls.Autonomous
- AutonInterpreter
  - in package org.team399.y2012.Robot.Controls.Autonomous

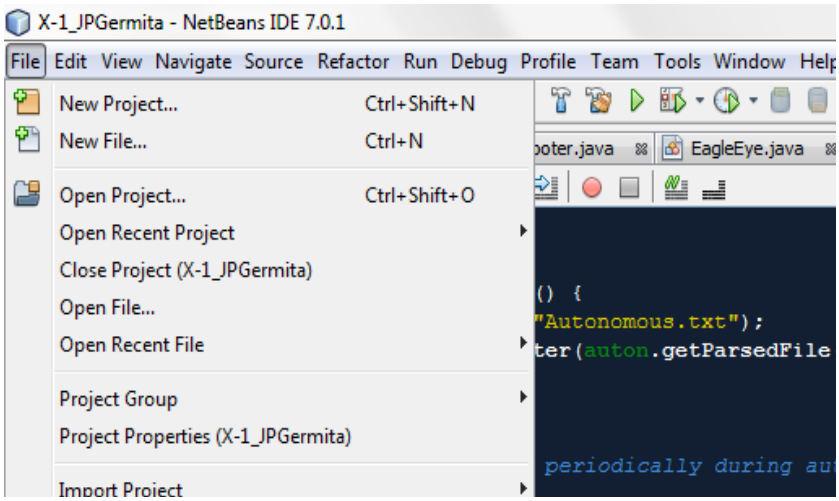
Section IV. How to use this code

Section overview

This section will discuss the usage of the code from opening as a project to compiling to editing.

A. Opening, Compiling

This code is written as a Netbeans IDE project. To open it, simply find the “Open Project” in Netbeans and navigate to the project folder.



To compile and download, simply set the project as the main project(if it isn't already) and press F6.

B. Editing Teleop Controls

To edit teleop controls, navigate to the Main.java file and scroll down to the drive() and

```
51
52      public void drive() {
53          double leftPower = 0, //Variables to customize controls easily
54                 rightPower = 0;
55          boolean shift = false,
56                  intake = false;
57
58          bot.drive.tankDrive(leftPower, rightPower);
59
60          if (shift) {
61              bot.drive.highGear();
62          } else {
63              bot.drive.lowGear();
64          }
65
66          bot.intake.setDropper(intake);
67      }
68
69      public void operate() {
70          double shooterSpeed = 0,
71                 intakeSpeed = 0,
72                 turretAngle = 0;
73          boolean hood = false,
74                  shoot = false;
75          boolean autoShoot = false,
76                  autoSpeed = false,
77                  autoAimLock = false,
78                  autoAimLFend = false,
79                  autoAimRFend = false,
80                  autoAimKey = false;
81
```

operate() methods. The variables that will be sent to the various mechanisms as commands will be listed at the top. Simply set the variables as your favorite input device's controls.

Section V. Appendix

Section overview

This section has various articles that didn't quite fit into the rest of the document.

A. Autonomous Commands and Syntax

Writing autonomous programs with the text file system is as simple as editing text. There are only two types of lines to learn: The comment and the command. Comments are denoted by either a '%' or a '#'. These are ignored by the program and are used for notes or removing certain lines from execution. Any line that is not a comment will be interpreted as a command. Commands, arguments, descriptions, and completion criteria are as follows:

Command	Arg1	Arg2	Arg3	Description	Completion Criteria
AUTOSHOOT	Belt Speed	Shooter Spd	Timeout	Autoshoot routine	Timeout has expired
BELT	Belt Speed	X	X	Starts belt at speed	None
DRIVE_DISTANCE	distance	timeout	X	Drives to distance in inches	Distance is reached or timeout
DRIVE_POWER	left pwr	right pwr	gear	Sets drivetrain to power	None
DROPPER	up/down	X	X	1 = up, 0 = down	None
SHOOTER	RPM	X	X	Sets shooter to speed	none
STOP	X	X	X	Stops autonomous execution	none
TURN_ANGLE	angle	timeout	X	Turns robot to angle in degrees	angle is reached or timeout
WAIT	time	X	X	Waits for time in ms	time expires

Sample program:

```
WAIT, 1000
% This is a comment
DRIVE_POWER, 1, 1, 0
# This is also a comment
WAIT, 5000
DRIVE_POWER, 0, 0, 0
# Notice how the commands and arguments are separated by a comma and
# a space. In that order. Program will fail if this order is not
# maintained
STOP
```

B. How to tune a PID controller

This section outlines a basic PID tuning procedure.

1. Start with all terms at 0
2. Start increasing the P term until a steady oscillation can be observed
3. Start increasing the D term until an undershoot is observed(slows as it approaches target)
4. Start increasing I until target is reached.Repeat 1-4 until system reaches target as quickly as desired.
5. Repeat 1-4 if system behaves differently between different setpoints.