Noteworthy Features of the Average Joes' 2020-2021 FRC Code

Doug Wegscheid

January 8, 2022

Abstract

FRC Team 3620, "The Average Joes", from St. Joseph High School in St. Joseph, Michigan, is releasing their 2020-2021 Java code for other FRC teams to learn from.

The robot code is available on github:
`https://github.com/FRC3620/FRC3620_2020_GalacticSenate.git`
The `test` branch is typically what the team used for 2020 competition, and other branches were used for various parts of the 2021 remote competitions.

The Shuffleboard configuration is available on github:
`https://github.com/FRC3620/FRC3620_2020_Shuffleboard.git`

The vision code running on top of the WPILibPi environment was derived from ChickenVision, and can be found at:
`https://github.com/FRC3620/FRC3620_2019_Vision.git`

This paper is a summary of what the team programming mentor considers to be features that other FRC teams may want to look at and use: swerve drive, event logging, data logging, using a Pixy2 to determine which path to follow during 2021 Galactic Search Challenge, and using vision to guesstimate range and set up the shooter. This last contributed greatly to the Joes winning the Innovation in Control Award at the St. Joseph District competition in 2020.

Some of the information contained here was not discovered or developed by The Average Joes; much of it was gleaned from other team's postings of code, discussions on Chief Delphi, or other publicly available sources. We didn't generally keep track of where we got ideas that we reused, and apologize for not being able to attribute them properly.

1 Overview

Our 2020-2021 FRC programming was done in Java (new) command-based code. We did *not* use Robot-Builder this year. We did version control with git, using github as our central repository.

Branch `master` had our code through the end of the 2020 season; for 2021, we had several branches go off from that, one each for a specific challenge, I believe that branch `Slalom` ended up with a merge of all the 2021 work.

The code was developed by Horatio Moreno (graduated 2020, now at University of Michigan[1]), Dashiel Matlock (graduated 2020, now at Michigan State University[2]), Nick Zimanski (graduated 2020, now at Michigan Tech[3]), Sean Thursby (graduated 2020), Drew Kellerer (graduating 2022, if he behaves himself[4]), Jared Svetlov[5] (graduating 2022), Liam Allen (graduating 2023, if **he** behaves himself[6]), Charlie Vaughn (graduating 2023?[7]), Grace Yuan (graduating 2024), and myself (graduated 1977[8]).

[1] he's ok anyway.
[2] Go Green!
[3] 188.4 inches of snow last winter, down from the 280.1 inches the previous year, and 317 inches the preceding year.
[4] not really an issue, just seeing if anyone actually reads this part!
[5] our build lead, but he learned enough programming to defend himself against programmers.
[6] yeah, he'll graduate.
[7] He *will* graduate, I don't remember what class he's in.
[8] My high school diploma was painted on a cave wall. My college diploma was carved in stone.

Here are the features of the 2010-2021 code that may be of use to other teams:

- Swerve drive

  We used Swerve Drive Specialties MK3 swerve drive modules in 2020-2021, and wrote our own swerve drive code.

- Event logging

  Event logging is the timestamped logging to a file of significant events during the execution of robot code. Events worth logging could include:

  – transitions between autonomous, teleop, test, and disabled states.
  – initialization, end, and interruption of commands.
  – blockage of operation because of limit switches.
  – acquisition and loss of vision targets.
  – brownouts.
  – selection of different autonomous modes.

  One of our requirements was that all logging be done at the roboRIO (not the driver station PC, which would consume valuable bandwidth).

  Figure 1 is an example of an event log.

```
2020/03/07 08:35:25.822 [org.usfirst.frc3620.misc.CANDeviceFinder] INFO - calling find()
2020/03/07 08:35:26.218 [frc.robot.RobotContainer] INFO - CAN bus: [PDP 0, PCM 0, TALON 1, TALON 2, TALON 3, TALON 4, TALON 5, SPARK_MAX 1,
      SPARK_MAX 2, SPARK_MAX 3, SPARK_MAX 4, SPARK_MAX 5, SPARK_MAX 6, SPARK_MAX 7, SPARK_MAX 8, SPARK_MAX 9, SPARK_MAX 10, SPARK_MAX 11]
2020/03/07 08:35:27.762 [org.usfirst.frc3620.logger.DataLogger] INFO - setupOutputFile filename is 20200307-083525.csv
2020/03/07 08:35:27.764 [org.usfirst.frc3620.logger.DataLogger] INFO - setting file to /home/lvuser/logs/20200307-083525.csv
2020/03/07 08:35:27.773 [org.usfirst.frc3620.logger.DataLogger] INFO - DataLogger interval = 1000ms
2020/03/07 08:35:27.790 [org.usfirst.frc3620.logger.DataLogger] INFO - Writing dataLogger to /home/lvuser/logs/20200307-083525.csv
2020/03/07 08:35:27.806 [frc.robot.Robot] INFO - Switching from INIT to DISABLED
2020/03/07 08:40:28.902 [frc.robot.Robot] INFO - Switching from DISABLED to AUTONOMOUS
2020/03/07 08:40:28.908 [frc.robot.Robot] INFO - Initialized GoldenAutoCommand
2020/03/07 08:40:28.916 [frc.robot.Robot] INFO - Initialized ManuallyMoveColorMotor
2020/03/07 08:40:28.920 [frc.robot.Robot] INFO - Initialized BeltIdleCommand
2020/03/07 08:40:33.042 [frc.robot.subsystems.DriveSubsystem] INFO - Switching to Manual Spin Mode
...
2020/03/07 08:40:36.605 [frc.robot.commands.AutoCreateShootingSolutionCommand] INFO - pixel Height = 360.0, calculated Position =
      13.418805399935655, calculated RPM = 3657.547772056719
2020/03/07 08:40:40.644 [frc.robot.Robot] INFO - Ended GoldenAutoCommand
2020/03/07 08:40:40.648 [frc.robot.commands.TeleOpDriveCommand] INFO - Init tod
2020/03/07 08:40:40.650 [frc.robot.subsystems.DriveSubsystem] INFO - setting heading to -140.3499984741211
2020/03/07 08:40:40.653 [frc.robot.Robot] INFO - Initialized TeleOpDriveCommand
2020/03/07 08:40:44.262 [frc.robot.Robot] INFO - Switching from AUTONOMOUS to DISABLED
2020/03/07 08:40:44.268 [frc.robot.Robot] INFO - Interrupted ManuallyMoveColorMotor
2020/03/07 08:40:44.273 [frc.robot.Robot] INFO - Interrupted BeltIdleCommand
2020/03/07 08:40:44.278 [frc.robot.Robot] INFO - Interrupted TeleOpDriveCommand
2020/03/07 08:40:44.502 [frc.robot.Robot] INFO - Switching from DISABLED to TELEOP
2020/03/07 08:40:44.508 [frc.robot.Robot] INFO - FMS attached. Event name MISJO, match type Elimination, match number 20, replay number 1
2020/03/07 08:40:44.515 [frc.robot.Robot] INFO - Alliance Red, position 1
```

Figure 1: Event log sample

- Data logging

  Data logging is the continuous timestamped logging of significant sensor data, actuator status, power statistics (voltage and current draw), and operator inputs to a file. This data could be processed after a test session, practice session, or match using LibreOffice Calc, Veusz, gnuplot, Excel[9], or some other graphics or analytical software to provide plots of such data. Figure 2 is an example of such a data log.

  A typical Veusz plot[10] of battery voltage during a match is shown in Figure 3.

  We used this data in 2016 to ensure that both motors on each side of our drive were actually drawing current (didn't have an open connection).

  We used this data in 2015 to determine if it was possible to lose weight by changing to a smaller air compressor. We logged the current drawn by the compressor, and by looking at the plot of current draw, we were able to determine exactly how much the compressor was run during any given match. See Figure 4 (data from the 2020 robot).

[9]$$$ ugh.
[10]all plots in this paper were done with Veusz.

```
time,timeSinceStart,matchTime,robotMode,robotModeInt,batteryVoltage,pdp.totalCurrent,pdp.totalPower,pdp.totalEnergy,compressorCurrent,
      liftCurrent,liftOutputPower,shooterRequestedHoodPosition,shooterActualHoodPosition,getRequestedTopShooterVelocity,
      getActualTopShooterVelocity,getTargetHeading
03-07-2020 08:35:27.851,0.085055,15,INIT,0,12.84,0,0,0,1.06,0,0,0,0,0,0,0
03-07-2020 08:35:28.783,1.017053,15,DISABLED,1,12.84,0,0,0,1.06,0,0,0,0,0,0,0
...
03-07-2020 08:40:28.828,301.058789,15,DISABLED,1,12.79,0,0,0,1.06,0,0,0,-0,0,0,0
03-07-2020 08:40:29.828,302.059180,15,AUTONOMOUS,2,11.91,22.5,273.38,21424.62,7.91,0,0,0,0,0,0,0
...
03-07-2020 08:40:43.830,316.061001,1,AUTONOMOUS,2,12.69,0,0,318086.5,1.06,0,0,13.42,13.48,12484.43,7746,-140.35
03-07-2020 08:40:44.831,317.061499,135,TELEOP,3,12.72,0,0,318086.5,1.06,0,0,13.42,13.48,12484.43,6518,-140.39
03-07-2020 08:40:45.830,318.061051,134,TELEOP,3,10.51,127.62,1371.88,340427.25,1.06,0,0,13.42,13.48,12484.43,5329,-140.39
...
```

Figure 2: Data log sample

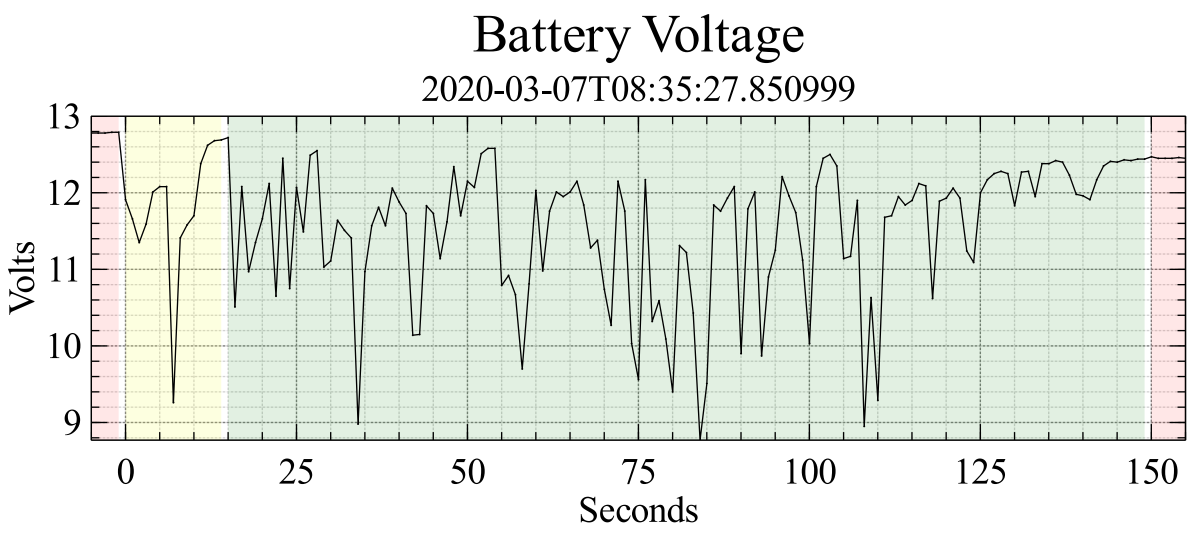


Figure 3: Battery voltage during a match

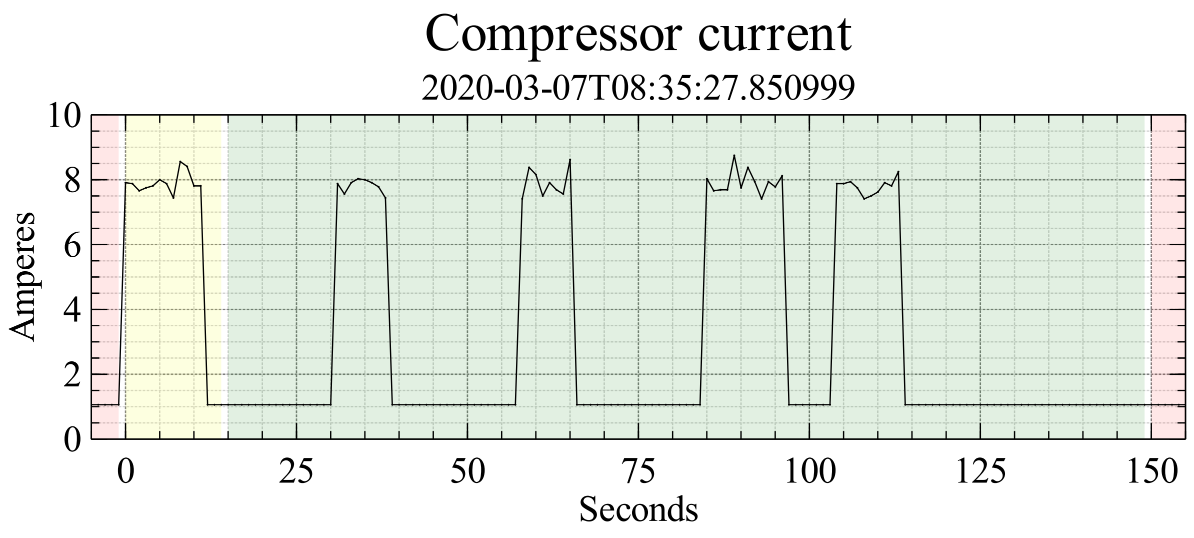


Figure 4: Compressor amperage during a match

There is also a facility to log data very quickly; this was useful while dialing in PID for the shooter (see Figure 5), and was useful when our shooter (final stage with two motors driving a common shaft) started coming up short. A look at the motor data for the two motors was interesting, show that both motors were not running at the same speed: Figure 6. A good onceover by the build team revealed a slipping motor coupling. Once fixed, all was well: Figure 7.

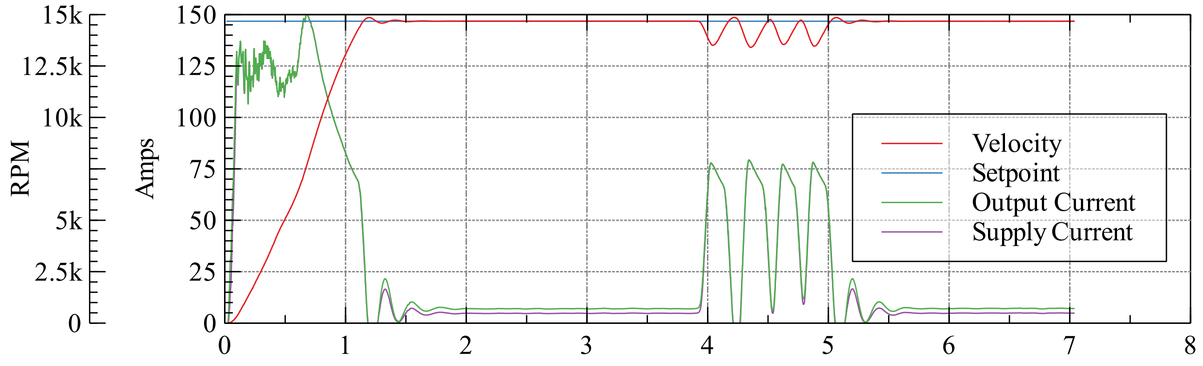


Figure 5: Shooter PID data

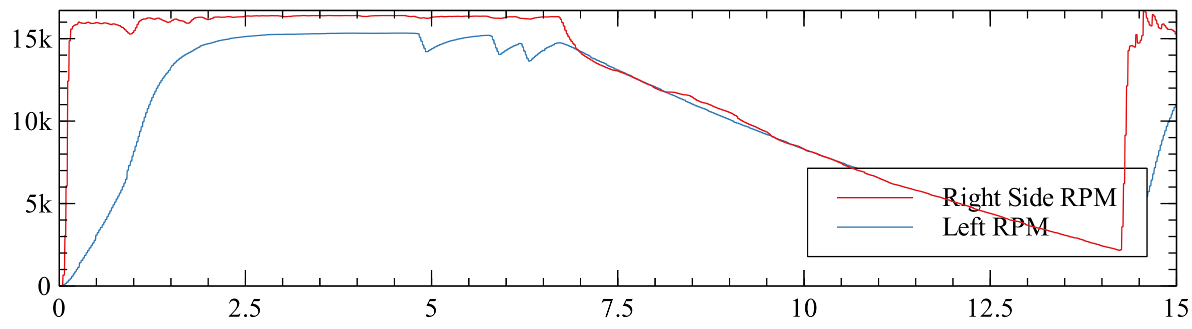


Figure 6: Shooter RPM with problem

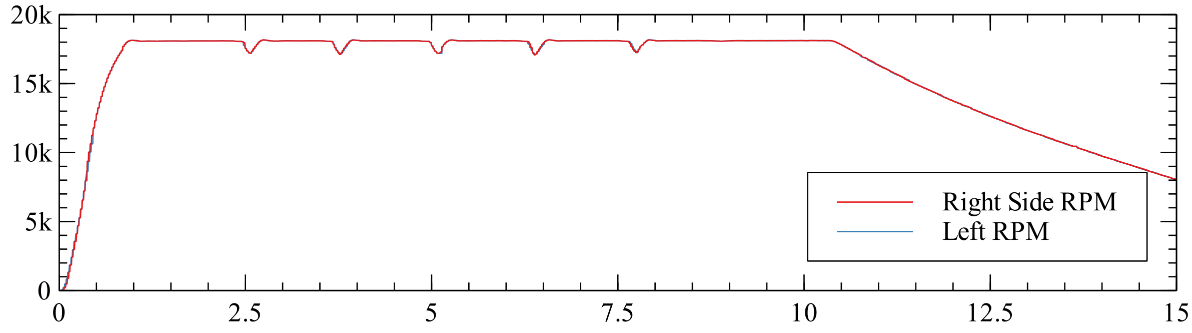


Figure 7: Shooter RPM with problem fixed

- Using a Pixy2 to determine which path to follow during 2021 Galactic Search

  The problem of determining what path to follow in the 2021 Galactic Search challenge required identifying the field configuration of red and blue field pieces.

  In 2020, we had extremely good vision capabilities using an adaptation of ChickenVision running on top of the WPILibPi distribution. However, the students with experience in Raspberry Pi based vision graduated in 2020, and we did not have a chance to meet much during the post-2020 offseason.

  The team prototyped using a Pixy2 to identify the field pieces, and quickly decided to add the Pixy2 to the robot. The Pixy2 has limited capability relative to a OpenCV based solution, but the capabilities were sufficient for the problem at hand, and was *much* simpler to implement.

- Using vision to guesstimate range and set up the shooter

  We used a derivation of `ChickenVision` to pick out the reflective tape around the Power Port. Communication of the target information from the Raspberry Pi to the roboRIO was done by placing a JSON string in the roboRIO NetworkTables.

  Centering the tape in the camera frame would aim correctly from side to side. Looking at the Y position of the tape in the camera frame tells us the range to the target (the closer we are, the higher the target appears in the frame).

  Our shooter had a PID velocity-controlled top wheel, and a worm-screw driven tilting hood to direct the power cells. Both of these need to be set for a given range. We picked 4 different ranges on the field, empirically determined good values for the top wheel RPM and hood position at each, and use that to develop a spline fit for setting those for intermediate ranges. The calculations for this are in `ShooterSubsystem.calcHoodPosition()` and `shooterSubsystem.calcTopRPM()`.

2 Swerve Drive

The code is mostly in class `DriveSubsystem`, but significant parts of the code are in package `frc.robot.miscellaneous`. The code does optimization to minimize the amount of steering necessary (it may set a wheel to the opposite direction of the desired azimuth and use reverse power if that results in less movement from the current module azimuth than simply turning the wheel to the requested azimuth). It makes extensive use of PID at the motor controllers for setting azimuth and drive speed.

3 Log File Naming and Location

One of our requirements for logging was that the event and data logs should be placed in the same directory (on a thumb drive, if one is inserted in the roboRIO), and have the same name (but different file types). The filename format we decided on was based on the current time, and was formatted "`yyyyMMdd-HHmmss`" (which sorts correctly). For simplicity (and because none of our matches spanned any Daylight Savings Time changes) we decided to format the date and time using the Eastern Time Zone, where all our practices and matches took place (except for CMP). We considered using GMT (commonly used in industry because of it's unambiguous nature and lack of glitches around DST changes), but those advantages had no value here, and were offset by the disadvantage of having to do timezone conversion when trying to find the files for a particular time.

New data and event log files (with new names) are written every time the robot code is restarted (at roboRIO boot, user code load, or user code restart). If multiple practice matches are played in a row, with no intervening roboRIO restarts, the data for all the matches will be in the same data and event logs.

The roboRIO has no onboard realtime clock; the system clock appears to start at Linux epoch time 0 (January 1, 1970) when the roboRIO is booted. The system clock apparently is initialized to the driver station current time sometime during the establishment of communications between the driver station and the roboRIO. Until that initialization takes place, the roboRIO has no idea of the current time, and cannot determine the correct file name. We made the decision to just not log data or events until that occurs.

The filename is derived from the system time at the instant of the first data or event logging that takes place after the system clock is set.

Method `getTimestampString()` in class `org.usfirst.frc3620.logger.LoggingMaster` contains the code that determines the filename for the data and event logs; `getTimestampString()` will return a null if the system clock has not yet been sent, else it provides the“`yyyyMMdd-HHmmss`” data. Once `getTimestampString()` has determine the correct name to use, the name is cached so that calls to determine event and data log file names return the same result, even though the calls happen at different times.

The code to determine the directory to put the data and event logs into is in the `getLoggingDirectory()` method of `LoggingMaster`. It looks for a thumbdrive with a writable `logs` directory; if none is found, logging will go to /home/lvuser/logs.

4 Event Logging

We used the JRE provided java.util.logging framework to do the actual logging[11], but used the SLF4J logging API as our mechanism for getting log events into java.util.logging[12].

The new command package made it simple to log command initializes, ends, and interruptions without coding logging into the individual commands, as we did in the past. Figure 8 contains the code (executed at `robotInit()`)that made that possible. We did discover that the logging did not occur for commands executed as part of a `CommandGroup`.

```
CommandScheduler.getInstance().onCommandInitialize(new Consumer<Command>() {
        //whenever a command initializes, the function declared below will run.
        public void accept(Command command) {
                logger.info("Initialized {}", command.getClass().getSimpleName());
        }
});

CommandScheduler.getInstance().onCommandFinish(new Consumer<Command>() {
        //whenever a command ends, the function declared below will run.
        public void accept(Command command) {
                logger.info("Ended {}", command.getClass().getSimpleName());
        }
});

CommandScheduler.getInstance().onCommandInterrupt(new Consumer<Command>() {
        //whenever a command is interrupted, the function declared below will run.
        public void accept(Command command) {
                logger.info("Interrupted {}", command.getClass().getSimpleName());
        }
});
```

Figure 8: Telling the `CommandScheduler` to log command start/stops

We have a `EventLogging.commandMessage` method to be called from `Command.initialize()` and `Command.end()`. This simply logs the command and the method name[13]. It’s the same as `logger.info()` methods, but does not need to be updated as code is refactored or copied. The hook into the `CommandScheduler` has largely eliminated the need for this, but it is still helpful for any command this is part of a `CommandGroup`.

Method `setup()` in class `org.usfirst.frc3620.logger.EventLogging` contains the code for programmatically setting up java.util.logging. It sets up a console handler (so that logged messages go to Riolog), and also sets up a custom file handler. The custom file handler will silently ignore logging attempts until it is able to get a valid log file name from LogTimestamp. `EventLogging` also has a custom event formatter that formats the events reasonably concisely.

`EventLogging` has some useful convenience methods: `writeWarningToDS()` will write a message to the message area of the FRC Driver Station, and `exceptionToString()` makes a reasonably concise formatting of an exception.

`EventLogging` also has a convenience method for getting an `org.slf4j.Logger` object for a class, and setting it up for the desired logging level. Use of this can be seen in the `Robot` class. j.u.l and Log4j2 are typically

[11]We originally considered using Apache Log4j2 instead of java.util.logging, but Log4j2 has dependencies on parts of the Java runtime that were not present in the compact version of the Java runtime that FIRST has us install on the roboRIO in past years. We may look at this after the 2022 season, Log4j2 has substantial performance advantages over j.u.l., as well as a better user API than SLF4J).

[12]SLF4J has the capability to send events to different underlying loggers; using it insulates our code from changes if we decide to go to log4j2.

[13]It need to be fixed to print the interrupted flag on a call from `end()`.

configured by using a configuration file. We rejected this because it required students that were debugging to make changes in both the modules they were debugging and the central log configuration. We finally landed on this compromise where the logging level for a module was set in the module's code when creating the logger for that module.

5 Data Logging

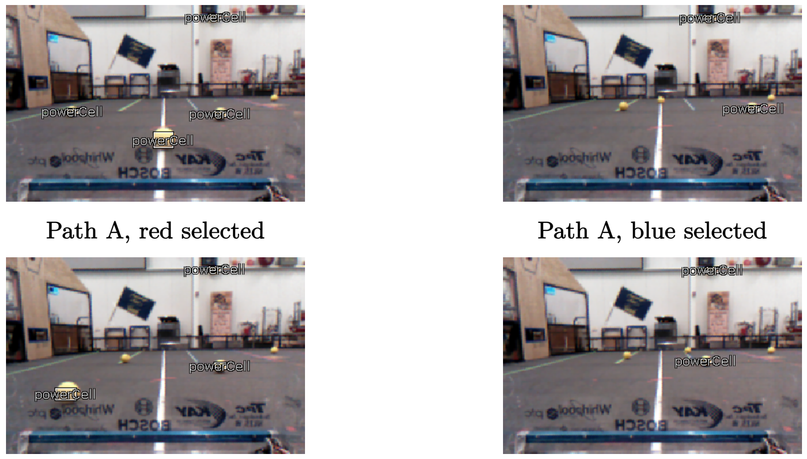
Data logging was taken care of in class `org.usfirst.frc3620.logger.DataLogger`. This class logs data to a file throughout the match.

The setup for data logging can be found in method `robotInit()` of class Robot; when logging, the caller needs to provide a object that implements the `IDataLoggerDataProvider` interface. This interface defines a method that provided the names of the data items, and a method that provided the values of the data items to be logged. Class `RobotDataLoggerDataProvider` provides this interface.

We also implemented class `FastDataLoggerCollections`, which is used to collect data quickly, then write it out at the end of an operation. This is what was used to generate the data for the plots for shooter PID tuning and shooter debugging.

6 Using a Pixy2 to determine which path to follow during 2021 Galactic Search Challenge

This code is on git branch `Slalom`, class `PixyPathFinder`. The Pixy2 was configured to see Power Cells. `PixyPathFinder` gets the blocks that match the Power Cells. The blocks lowest in the frame will be the closest, and the pre-programmed path to follow to collect the Power Cells can be determined by the X-Y location in the picture frame of the closest Power Cell (see Figure 9).



Path A, red selected

Path A, blue selected

Path B, red selected

Path B, blue selected

Figure 9: Pixy2 view of various paths and red/blue selections

`PixyPathFinder` adds `tags` information to the list of blocks it gets from the Pixy2, and `PixyTestCommand` runs the analysis and saves the list of blocks to a file for post-mortem analysis in case of a mistake (see Figure 10).

The code to interface to the Pixy2 was from `https://github.com/PseudoResonance/Pixy2JavaAPI`. We used version 1.4 instead of the latest version 1.4.1[14]. There is a problem we experienced where running the code

[14]We were unable to use 1.4.1 with WPILibJ 2021. I don't remember why, the only difference I see between 1.4 and 1.4.1 is that 1.4.1 runs the SPI at a faster clock speed. I need to do some retesting with WPILibJ 2022.

```
{
  "blocks": [
    {
      "block": {
        "signature": 1,
        "x": 66,
        "y": 144,
        "width": 28,
        "height": 15,
        "angle": 0,
        "index": 13,
        "age": 255
      },
      "aspectRatio": 1.8666666666666667,
      "area": 420,
      "bottom": 151.5,
      "tags": [
        "low",
        "decider"
      ]
    },
    {
      "block": {
        "signature": 1,
        "x": 220,
        "y": 13,
        "width": 12,
        "height": 8,
        "angle": 0,
        "index": 216,
        "age": 255
      },
      "aspectRatio": 1.5,
      "area": 96,
      "bottom": 17.0,
      "tags": []
    },
    {
      "block": {
        "signature": 1,
        "x": 225,
        "y": 115,
        "width": 14,
        "height": 6,
        "angle": 0,
        "index": 155,
        "age": 255
      },
      "aspectRatio": 2.3333333333333335,
      "area": 84,
      "bottom": 118.0,
      "tags": [
        "low"
      ]
    }
  ],
  "path": "B_RED",
  "when": "20210302-135131"
}
```

Figure 10: Debug information from `PixyPathFinder`

for a long time while detecting many blocks could cause an `"LLVL: out of memory"` error[15]. In practice, the amount of time it took for this to happen would much longer than the length of a match.

[15]See `https://github.com/PseudoResonance/Pixy2JavaAPI/issues/15`. I need to retest with WPILibJ 2022.