Iron Reign 6832 Control Award Submission

Key Algorithms

PID Control Loop

One of the key algorithms employed by Iron Reign is a feed-forward PID loop. A PID loop allows us to adjust the output of our motors in order to exactly reach a target position without overshooting or undershooting.

A PID loop uses the following equation, where K_P, K_I, K_D, K_0 are constants, and $e(t)$ is a function of the error over time.

$$P(t) = K_P \cdot e(t) + K_I \cdot \int_0^t e(u)\,\mathrm{d}u + K_D \cdot \frac{\mathrm{d}}{\mathrm{d}t}\,e(t) + K_0 \cdot sign(e(t))$$

K_P is the coefficient of the proportional response, i.e. the portion of the response which is directly in proportion to the error. The proportional response constitutes the main portion of the PID Control loop.

K_I is the coefficient of the integral response, i.e. the portion of the response which is proportional to the sum of the error over time. The integral response exists mainly to avoid oscillations when the output overshoots the error. In some cases, the K_I term is unnecessary, as it is already taken care of for the most part by the K_D term.

K_D is the coefficient of the derivative response, i.e. the portion of the response which is proportional to the change in error. The derivative response slows down the output of the control loop in order to avoid overshooting.

K_0 is the constant of kinetic friction, which exists in order to overcome friction. K_0 is always applied against the direction of error, which is the reason for $sign(e(t))$. $sign(n)$ is defined as follows:

$$sign(n) = \begin{cases} 1 & n > 0 \\ 0 & n = 0 \\ -1 & n \leq 0 \end{cases}.$$

Iron Reign has tuned our coefficients, and these are the optimal values we have found for our drivetrain:

$$\begin{aligned} K_P &= 0.010 \\ K_I &= 0.000 \\ K_D &= 0.001 \\ K_0 &= 0.050 \end{aligned}$$

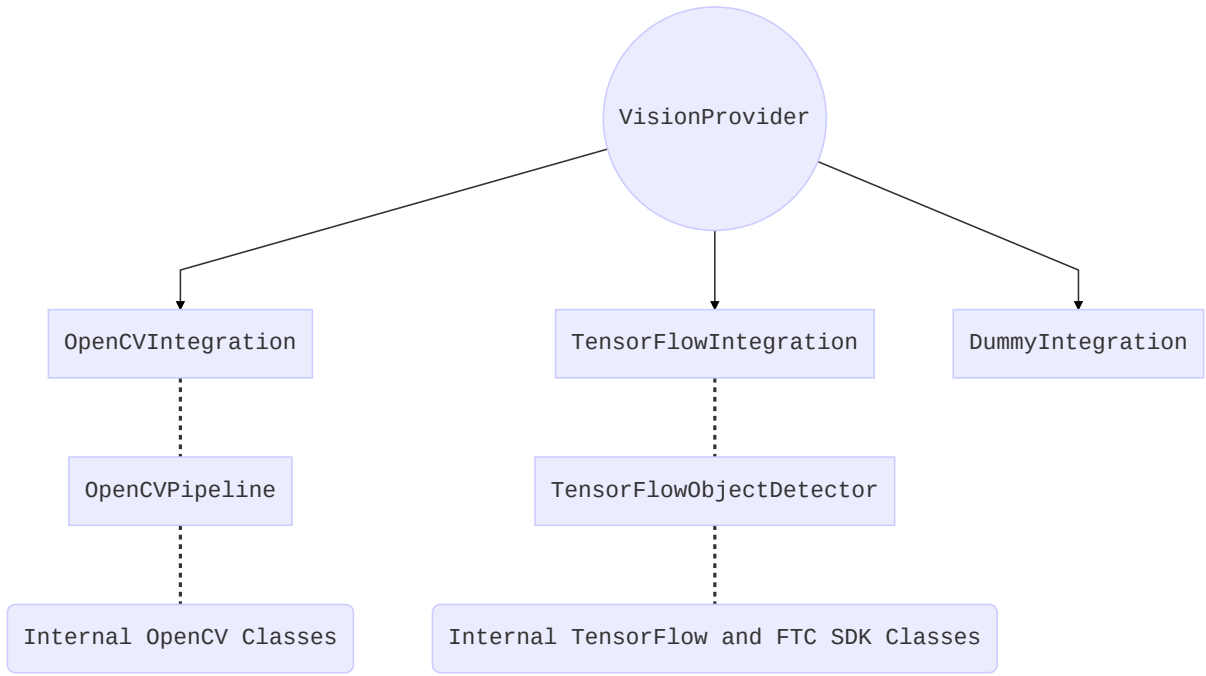
Computer Vision Pipelines

Iron Reign has been working on many different vision pipelines, including using the built-in TensorFlow, a custom OpenCV pipeline, and a custom Convolutional Neural Network. In order to support the development of the different backends, we designed a pluggable solution that allows us to swap out vision backends, or "providers", at runtime.

Our system consists of an interface, `VisionProvider`, which is implemented by each of the vision providers, `TensorFlowIntegraton`, `OpenCVIntegration`, and `DummyIntegration`. `VisionProvider` looks as follows:

```
public interface VisionProvider {
    public void initializeVision(HardwareMap hardwareMap, Telemetry telemetry);
    public void shutdownVision();
    public GoldPos detect();
    public void reset();
}
```

A class diagram is shown below:



`OpenCVIntegration` uses a custom-built OpenCV pipeline to recognize the sampling order of minerals. It can be adapted to be used in situations where it can see only two minerals or all three minerals in the sampling field. `OpenCVIntegration` is our current vision provider of choice during the tournament.

`TensorFlowIntegration` uses the built-in TensorFlow model provided by the FTC SDK with some modifications to the code in order to allow it to detect the sampling order in more cases. Based on our testing, this is not as reliable as `OpenCVIntegration`, as it often fails to detect one or more of the minerals in the sampling field.

`DummyIntegration` is an implementation that always "detects" the gold as being in the middle position, useful as a fallback in case we do not want to use any vision, but still run an autonomous.

We are also developing a Convolutional Neural Network which is completely separate from the TensorFlow model provided by FIRST. We actually began developing this much before the built-in TensorFlow model was even announced. We designed a way to easily capture training images by driving around our robot, by grabbing frames from Vuforia as we drove and saving them to the internal memory of the phones. We also began writing the actual CNN in both Python (using the TensorFlow library) and Java (using the DL4J library, which is developed by Eclipse). Unfortunately, neither of these CNNs performed to the accuracy we were hoping for. At the moment we have not taken our Convolutional Neural Network past a prototype stage, and as such we are developing it off robot. Once we get to a point where we are confident in its performance, we will add it to our robot code.

An additional abstraction we created was the ability to switch between cameras at runtime. We originally wanted to use the front phone camera for our vision, but we later decided to test out a webcam. In order to make it easier to switch the cameras, we created a system that switches "viewpoints" between front, back, and webcam. This is now configured during robot setup, along with the vision provider selection. (See the robot setup section for more information.)

Please read the *Vision Summary* entry in our journal for more information.

Replay Autonomous

Replay Autonomous is a technique that records a drivers movements and "replays" those movements during the autonomous period. Replay Autonomous is extremely useful to quickly and efficiently prototype autonomous programs. We do not use Replay Autonomous during competition because it can be slightly inaccurate, but we still use it for prototyping and testing. We also used Replay Autonomous to help out a team at the Townview Qualifier write a full autonomous from scratch on the day of the tournament.

Replay Autonomous works by capturing the joystick values of the driver during teleop control, cleaning them up with some math, and then saving these values to a file. These values are then read during the autonomous period. For more permanent routines, our program can also output a Java file that is then copied over to a computer and compiled with the Robot Controller app. This Java file contains a class with one static member which contains the processed data, allowing the program to read the values without performing File IO (which can become very expensive on Android, due to application sandboxing).

Please read the *Replay Autonomous* entry in our journal for more information.

Non-Blocking State Machines

Our autonomous routines are structured as sequences of discreet actions, such as "drive forward 1 meter" or "turn to an angle of 180 degrees". We encode these actions as individual "states", which are then chained together into a state machine. We then continuously poll our state machine, checking whether it has completed its current state, if so, moving on to the next state. These state machines make our autonomous extremely easy to program. A sample autonomous is shown below.

```
private StateMachine auto_depotSample = getStateMachine(autoStage)
            .addNestedStateMachine(auto_setup) //common states to all autonomous
            .addMineralState(mineralStateProvider, //turn to mineral, depending on
mineral
                    () -> robot.rotateIMU(39, TURN_TIME), //turn left
                    () -> true, //don't turn if mineral is in the middle
                    () -> robot.rotateIMU(321, TURN_TIME)) //turn right
            .addMineralState(mineralStateProvider, //move to mineral
```

```
                    () -> robot.driveForward(true, .604, DRIVE_POWER), //move more on the
sides
                    () -> robot.driveForward(true, .47, DRIVE_POWER), //move less in the
middle
                    () -> robot.driveForward(true, .604, DRIVE_POWER))
            .addMineralState(mineralStateProvider, //turn to depot
                    () -> robot.rotateIMU(345, TURN_TIME),
                    () -> true,
                    () -> robot.rotateIMU(15, TURN_TIME))
            .addMineralState(mineralStateProvider, //move to depot
                    () -> robot.driveForward(true, .880, DRIVE_POWER),
                    () -> robot.driveForward(true, .762, DRIVE_POWER),
                    () -> robot.driveForward(true, .890, DRIVE_POWER))
            .addTimedState(4, //turn on intake for 4 seconds
                    () -> robot.collector.eject(),
                    () -> robot.collector.stopIntake())
            .build();
```

These state machines make programming an autonomous routines extremely easy. While previously taking over 400 lines before we converted them to this new system, our autonomous routines now fit within 30 lines each. Modifying and tuning these autonomous routines is also easy, since we just need to add an additional line of code to add an individual action. In addition, these state machines prevent us from making simple typos that cause our autonomous to fail, since we no longer need to have a lot of similarly named counter variables.

Please read the *Autonomous Non-Blocking State Machines* entry in our journal for more information.

Key Sensors

- IMU (Inertial Measurement Unit) - The IMU is used to determine the heading of the robot, which is then used alongside a PID Control loop to allow the robot to make precise turns. We also use the IMU to determine whether our robot is tipping, so that we can perform corrective action to avoid us from completely tipping over our robot.
- Motor encoders - Motor encoders allow us to determine how far our robot has moved or what position its subsystems are at, which is used alongside a PID Control loop to allow the robot to make precise movements. Every subsystem on our robot uses encoders.
    - The drivetrain encoders allow us to determine how far our robot has moved. We also look at the differences between encoder values between both sides of our drivetrain in order to ensure that our robot is driving in a straight line. Along with a PID loop, this allows us to make precise lateral movements. We use these especially during autonomous, to ensure that our robot is driving the same distance and direction between each run. These are also used during teleop, to force the robot to drive in a straight line rather than veer to the side.
    - The superman arm encoders allow us to determine the current position of the arm. Using a PID contol loop, we can set a target position for the motors, and then the arm will move to that position. We use this extensively with the manual position control, as well as the preset system.
    - The elbow encoders allow us to determine the angle and extension of our elbow. Again, using a PID control loop, we can set a target position for both of these, and our elbow/belt system will then extend to that position. We use this extensively with the manual position control, as well as the preset system.

- Cameras (including the phone cameras and external webcams) - We use cameras to get input which is passed to our vision pipelines, which then are used to determine the sampling order. Vuforia is used to grab frames from the camera, and these frames are then passed to TensorFlow, OpenCV, or another backend, depending on which vision provider we are using. As mentioned above, we have devised a system for dynamically switching for between camera viewpoints during runtime. (See vision section for more information.)

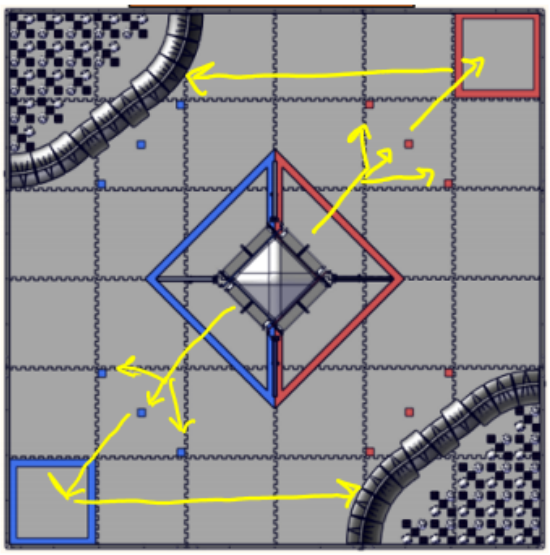
Autonomous

Our autonomous performs the following tasks:

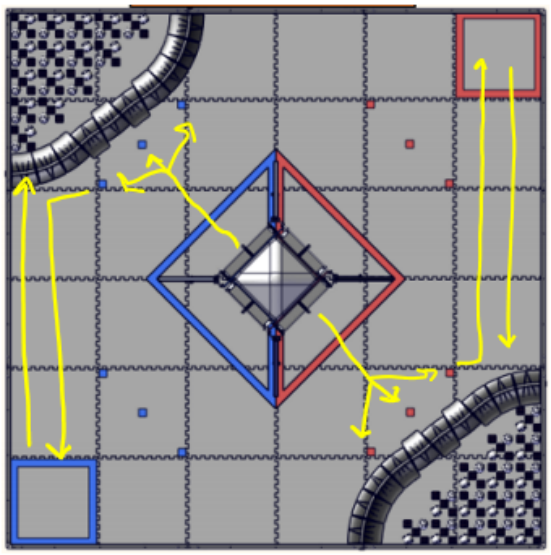
1. Delatch from the lander.
2. Sample the minerals in our sampling field.
3. Claim depot by placing team marker.
4. Park in the crater.

We have developed four different autonomous paths, shown below.

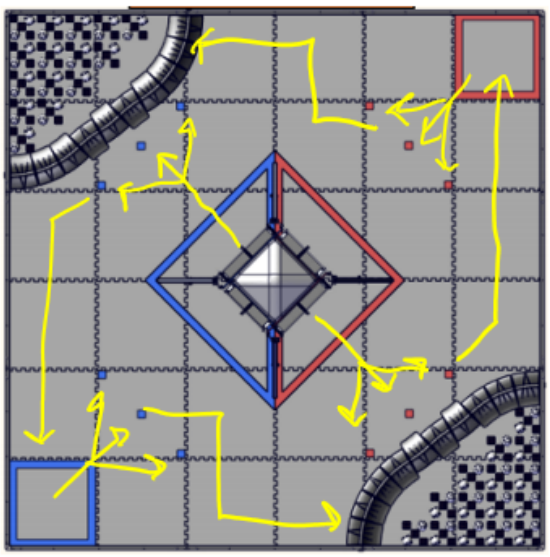
Path 1



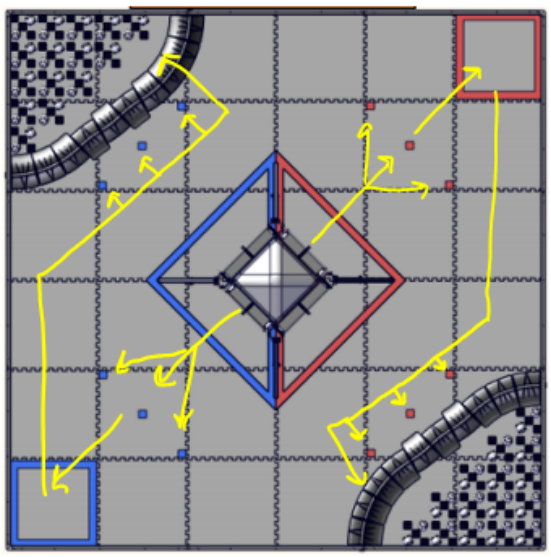
Path 2



Path 3



Path 4



Driver controlled enhancements

Hold-position control loops for elbow and superman

Both the elbow and superman motors use a PID control loop that allows the driver to directly manipulate their position rather than the power given to their motors. As the amount of motor power required to keep the motors in a steady state is variable, this provides a huge relief for the drivers and allows them to focus on the actual position of the robot's subsystems rather than the raw motor output.

Preset Controls and Articulations

In addition to the manual control above, we also use a preset system, where we have a set of already-known positions that we can switch to by button press. These preset positions include:

- Driving - a position which lowers the superman arm and moves the elbow to a position allowing the robot to comfortably drive without being imbalanced.
- Intake - a position which extends the elbow and arm to a position that allows it to intake minerals by reaching into the crater.
- Deposit - a position which extends the elbow and arm to a position that allows us to deposit minerals in the lander.
- Pre-latch - a position which allows us to prepare for latching by elevating our robot enough such that our hook is right next to the lander.
- Post-latch - a position which lifts our robot off the ground once we have latched.

We accomplish this internally using a system called Articulations, which "articulates," or sets our motor targets to the specified values. This system also allows us to have multi-step transitions, meaning that we can change the order of certain actions to keep our robot in a stable state, preventing it from tipping over and causing issues.

Tipping-risk damping

The IMU records the angle of the robot, and when it is at a dangerous level, corrective action is taken to avoid tipping. This includes damping motor powers and forcing the motors forward.

Automatic endgame latching

Since it was difficult for our drivers to latch during endgame, we designed an automatic latching routine using a similar framework as our autonomous routines. This allows us to perform latching with little to no driver input, however, drivers can still adjust the latching process and abort the routine if necessary. This new latching routine makes extensive use of the IMU to determine the angle of the robot.

Robot setup

During the initialization of the robot, there is a lot of configuration we need to perform. Much of this relates with vision parameters, but there are other things we do

- Vision provider configuration
    - As vision providers are all modularized, the actual initialization and configuration needs to be done at runtime, rather than in the code.
    - Basically the entire left side of the controller is occupied by vision configuration during initialization
- States and Active
    - Have multiple “states” rather than separate opmodes for teleop and individual autonomous routines.
    - State 0 is teleop, states 1-10 are for autonomous, demos, and other miscellaneous items.
    - When the robot is inactive, we can change state and configure it. When it is active, each state is running.
- Autonomous controls
    - Set delay for autonomous - to avoid conflicts with other teams
    - Toggle alliance color

Engineering notebook references:

A comprehensive list of our control entries can be found in the Table of Contents of our journal, as well as at https://www.ironreignrobotics.com/tags/control/toc/.

Recommended posts are Replay Autonomous, Vision Discussion, CNN Training, Autonomous Path Planning, Pose BigWheel, RIP CNN, Refactoring Vision Code, OpenCV Support, Vision Summary, Autonomous Non-Blocking State Machines.