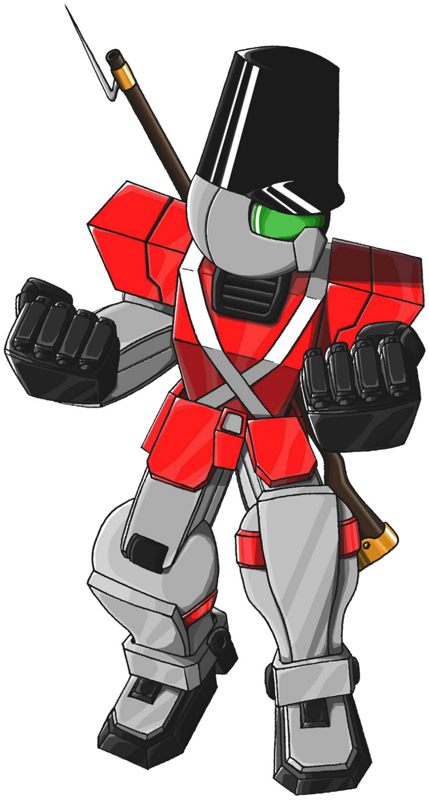


SIMBOTICS TEAM 1114

CONTROLS AND PROGRAMMING

SIMBOTICS TEAM 1114

2012 Season

Simbot Jordan

Overview

Programming Environment

- Using Java
- Iterative robot template
- Modular Programming structure
  - Separate class for each part of robot functionality
  - Over 100 classes in total, broken down into directories by component
- Autonomous Structure
  - created modular commands that can be reused in different autonomous modes

Sensors on Robot

- Encoder on each side of drive for distance
- Gyro for robot angle
- Encoder for turret position
- Encoder for shooter wheel speed
- Potentiometer for Collector position
- Camera for aiming at target
- Light sensors for ball position (ready to fire/fully loaded)

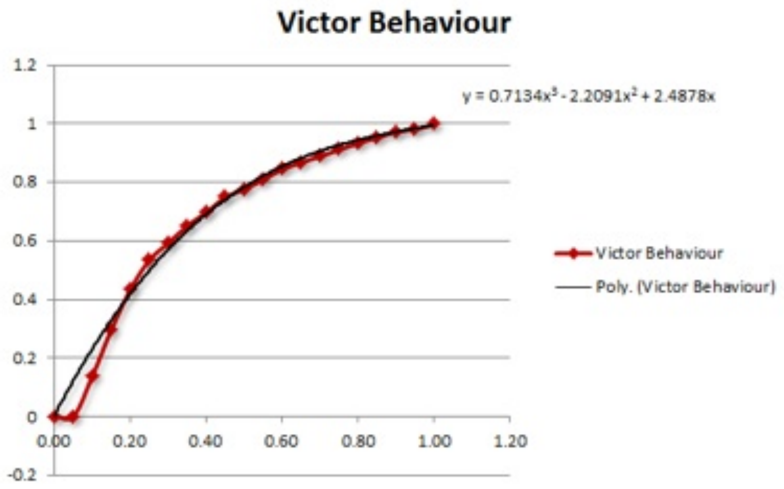
Key Programming Elements

1. Linearization of Victors
2. PID Control
3. Base Lock
4. Camera Aiming
5. Shooter Speed Control
6. Autonomous Structure
7. Controls in Teleop
8. Positional System
9. Use of WPI SmartDashboard

Linearization of Victors

Reason

- Inspired by the work done last year by FRC254 The Cheesy Poofs
- Desired consistent voltage change from an output change in code
- Current behaviour of victor is non-linear:



Functionality

- More consistent control of components
  - a desired output of 0.25 actually generates 25% of total power
  - with default behaviour an output of 0.25 would generate approximately 50% output
- Allowed more consistent PID control of robot components

How We Did It

- Plotted (normalized) voltage readings from victors at set increments
- Found a curve of best fit for this data
- Calculated the inverse of this curve
- Found simplified curve of best fit to fit this new data for domain [0, 1]
- Used to convert all desired outputs to actual victor outputs
- Function used to approximate inverse:
  - $linearize(x) = 3.1199x^4 - 4.4664x^3 + 2.2378x^2 + 0.1222x$

PID Control

Reason

- Accurate and repeatable control of components in autonomous and teleop presets
- Original version created before WPI included a PID class
- Desired more control than the default PID Class allowed
    - Arm control (different constants in each direction)
    - Speed control (derivative term removed at times)

Functionality

- Reusable class for accurate control of various robot components

How We Did It

- Development of PID code goes back to 2005 in C
- In 2009 it was formalized to be a separate class instead of a calculation embedded in code
    - Updated each year to move closer to industry style PID control
- PID Calculation
    - P proportional: how far we are from the goal
    - I integral: how long have we been away from the goal
    - D derivative: how quickly are we moving towards the goal
- Ouput = P * C_P + I * C_I - D * C_D
    - Where C_P, C_I, and C_D are constants set for each separate application
- Tuning Constants
    - Start with all three constants at 0
    - Increase P until the component oscillates consistently around desired endpoint
    - Increase D until the component comes to rest near endpoint without oscillation
    - Add I until the component comes to rest at the proper endpoint
- Speeding up the tuning of constants:
    - We have used a file of constants on the robot to change constants so the robot does not need to be rebooted for every change
    - We now use the WPI SmartDashboard for ever faster adjustment of constants

Base Lock

Reason

- More efficient bridge balancing. Preventing the robot from rolling off of an unbalanced bridge
- Used in previous years for robot to hold position on field while completing a task, even with heavy defence

Functionality

- Driver able to hold position on bridge by pressing a button
    - also able to make small adjustments to position to aid in balancing

How We Did It

- PID drive control with the desired “end point” being the initial location of the robot
- Pressing the Drive-Y control on joystick adjusts this desired endpoint slightly to allow fine movement while still holding on bridge

Camera Aiming

Reason

- Accurate targeting of goals in both autonomous and teleop period

Functionality

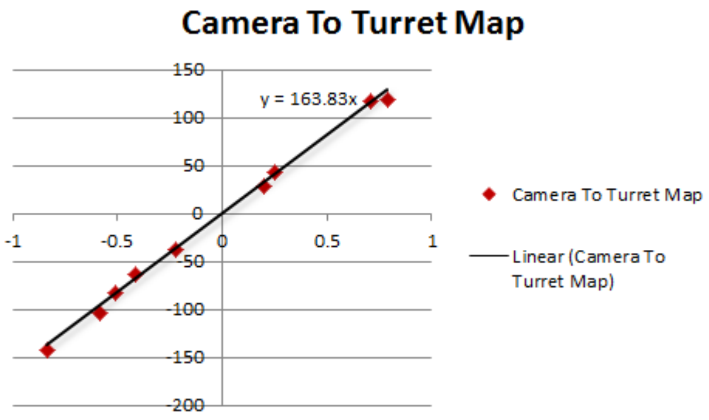
- Identifying and tracking centre target goal for aiming turret

How We Did It

- Identifying Target
  - Green LED “Angel Eye” used to illuminate target
  - HSL Colour Thresholding to filter out other particles
    - NI Vision Assistant used to find HSL ranges
  - Remaining particles are rated based on four criteria to identify most likely target:
    - Size of particle
    - Particle Aspect Ratio (closest to 4:3 target rectangle)
    - Height of particle in camera view
    - Fill percentage of particle/quality (only outer area should be filled)
  - Highest scoring particle is considered to be the target
  - Different criteria for key vs fender images - data collected from recorded images
- Driver Queues
  - SmartDashboard has a light and audible queue to let the drive team know the target is locked
- Tracking described on next page

Camera Aiming (continued)

Tracking to the Target



<u>Version #1</u>

- Camera images updated every 10 cycles
- Offset calculated from current turret position to the position that would centre the target
- Aiming finished when image is centred in camera view
- Issues
  - Often overshot target
    - inaccurate conversion factor to turrent encoder ticks
    - stale images from camera delay/image processing time
  - Took a long time to settle on target
    - stale images
    - shaking of shooter

<u>Version #2</u>

- More data collected for more accurate conversion from image location of target to encoder movement
- Only a single image used to move to goal
- Encoder position recorded at time image is received instead of after processing time
  - if turret was in motion, the image would be from an old position, not current position - lead to overshooting target
- Result
  - Much faster/more accurate target locking
- Issues
  - Compressed target at edge of vision (when only part of the target is in the view) leads to inaccurate conversion to turret movement

Shooter Speed Control

Reason

- To have consistent shooter speed for more accurate shots
- Constant motor output did not have a consistent speed throughout matches with different batteries

Functionality

- Able to control speed of shooter at desired target +/- 25 RPM
- Partially limited by resolution of turret encoder and maximum encoder ticks the cRIO can handle

How We Did It

- Different from standard PID control. Instead of the PID calculation setting the output, the PID calculation is an adjustment to the output
  - output = output + calcPID(currentSpeed)
- Large “D” constant used to reduce overshoot (starts throttling back before getting to desired speed)

Autonomous Structure

Reason

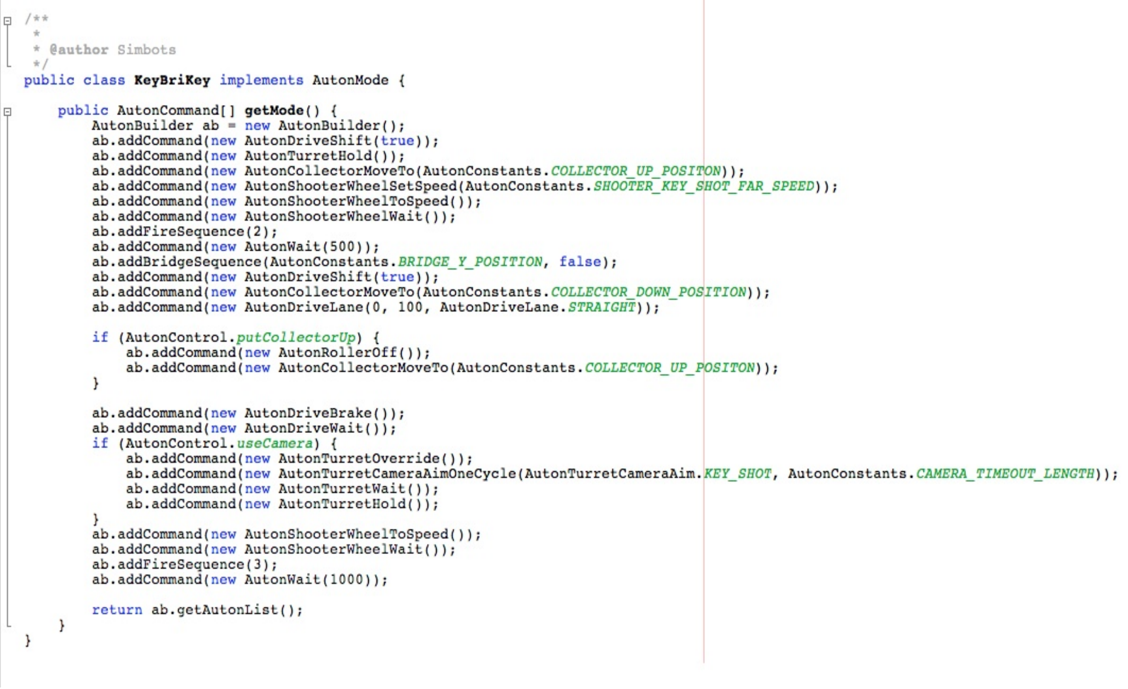
- Be able to quickly build new autonomous modes and modify existing modes

Functionality

- Separate commands for each robot component
- Lists of commands made to create autonomous modes
- Driver can select from list of autonomous modes before start of match

How We Did It

- Separate command class for movement of each component of the robot
- An array of these commands is created and executed one after another to build full autonomous mode
- Parameters in class creation allow modular use
- Sample autonomous mode:



```java
/**
 *
 * @author Simbots
 */
public class KeyBriKey implements AutonMode {

    public AutonCommand[] getMode() {
        AutonBuilder ab = new AutonBuilder();
        ab.addCommand(new AutonDriveShift(true));
        ab.addCommand(new AutonTurretHold());
        ab.addCommand(new AutonCollectorMoveTo(AutonConstants.COLLECTOR_UP_POSITON));
        ab.addCommand(new AutonShooterWheelSetSpeed(AutonConstants.SHOOTER_KEY_SHOT_FAR_SPEED));
        ab.addCommand(new AutonShooterWheelToSpeed());
        ab.addCommand(new AutonShooterWheelWait());
        ab.addFireSequence(2);
        ab.addCommand(new AutonWait(500));
        ab.addBridgeSequence(AutonConstants.BRIDGE_Y_POSITION, false);
        ab.addCommand(new AutonDriveShift(true));
        ab.addCommand(new AutonCollectorMoveTo(AutonConstants.COLLECTOR_DOWN_POSITION));
        ab.addCommand(new AutonDriveLane(0, 100, AutonDriveLane.STRAIGHT));

        if (AutonControl.putCollectorUp) {
            ab.addCommand(new AutonRollerOff());
            ab.addCommand(new AutonCollectorMoveTo(AutonConstants.COLLECTOR_UP_POSITON));
        }

        ab.addCommand(new AutonDriveBrake());
        ab.addCommand(new AutonDriveWait());
        if (AutonControl.useCamera) {
            ab.addCommand(new AutonTurretOverride());
            ab.addCommand(new AutonTurretCameraAimOneCycle(AutonTurretCameraAim.KEY_SHOT, AutonConstants.CAMERA_TIMEOUT_LENGTH));
            ab.addCommand(new AutonTurretWait());
            ab.addCommand(new AutonTurretHold());
        }
        ab.addCommand(new AutonShooterWheelToSpeed());
        ab.addCommand(new AutonShooterWheelWait());
        ab.addFireSequence(3);
        ab.addCommand(new AutonWait(1000));

        return ab.getAutonList();
    }
}
```

Controls in Teleop

Reason

- Aid drivers in precise control of robot components

Functionality

- Turret
  - Preset positions for shots at different locations on the field
  - Camera aiming for precise aiming of shots (discussed earlier)
- Collector
  - Positions for ball pickup (from bridge and floor), bridge approach, up within frame
- Shooter
  - Separate buttons for adjusting shooter speeds and camera constants at different firing locations
- Firing Sequence
  - Automated sequence to control of elevator/firing solenoid to prevent jamming while firing
- Drive
  - Finer control in centre range of joystick
  - Base Lock (discussed earlier)

How We Did It

- Used PID Controls from autonomous code to get similar automatic behaviour in teleop
- Driver buttons switch between automatic movements and preset movements
- Drive input (X/Y) is squared to give finer adjustment near centre of range while still getting full power at boundaries (-1 and 1)
- Timer delays used to coordinate elevator and cylinder trigger sequence

Positional System

Reason

- Keep track of location of robot on X/Y grid, with starting location as the origin

Functionality

- Used to keep the robot facing in a specific direction and travelling along a set lane to reach a desired goal downfield
- Allowed “S-curve” movements of robot to specific lanes, which allowed faster movement than separate turn-drive-turn commands

How We Did It

- The robot is assumed to move in a straight line for each individual cycle (20ms)
- The average of the two drive encoders is used as the distance the robot has covered
- The gyro is used to get the robot’s current heading
- Trigonometry is used to calculate the x- and y-changes to the robot position

Use of WPI SmartDashboard

Reason

- Feedback to drive team during matches
- Faster way to adjust values in code (compared to recompiling/loading code on the robot)

Functionality

- Used to quickly tune PID Constants for robot components
- Used to give drive team visual and audible signals when a ball is in the firing seat and the robot is fully loaded
- To view the current camera image with an overlay for where the robot believes the current target is
- To graph shooter speeds and other debug information.

How We Did It

- Tuning PID Constants uses built-in WPI SmartDashboard text boxes
- Created custom extension of BooleanBox to play wav file when state flips to true
  - Ball loaded and ready to fire
  - Camera properly aimed at target
- Created custom extension of camera to show target location and where turret is aimed