SOFTWARE SUBSYSTEMS - TELEOP AND AUTONOMOUS

Summary: Development of software (Java) in the Android Studio environment that allows the robot to navigate during the Autonomous period and the Drive Team to operate the robot during Teleop.

Pros: Made software reusable from past season's work; Android Studio is the most capable environment for implementing complex software, including machine vision.

Cons: Android Studio is also the most difficult programming environment to use, and the software designer was new to the job this season.

Solution: A great deal of mentoring and software expertise was available.

Software Reuse:

- Mecanum drivetrain software was provided by mentor Ian from past season's robots. On the advice of Ian, all of our OpModes are of type LinearOpMode (thus the "L" in the naming of the AutoLxxxxx and TeleOpLxxxxx class names).
- Ring Height Detection using the Webcam and OpenCV was accomplished by starting with an approach by team 9794, Wizards.exe (EasyOpenCVExample). Their example makes use of an OpenCV "pipeline", and was refined, streamlined, and clarified for use in Vera.
- Launcher motor PID tuning was accomplished using a standalone OpMode (VeloPIDTuner) from Noah Bresler (team 10940) that utilizes the FTC-Dashboard and RoadRunner software packages to graph the motor velocity in real-time and to allow modification of Proportional, Integral, Derivative, and Feed-Forward terms so that the motor quickly returns to its commanded velocity quickly, without overshoot, and without oscillations.

Mentoring:

- The team software mentor took Joel through the on-line book "Learn Java for FTC" by Alan G. Smith. The team coach provided a hardware testbed that included a Robot Control Hub and one of each type of hardware component we would be working with. All of the examples in the first 12 sections of the book were worked to learn how to add hardware components to the robot and how to control them from software. The book also explained how to separate the code that deals directly with the hardware into its own "mechanisms" class (known as abstraction or compartmentalization). This allows the OpMode code to be more readable and requires fewer changes to all those classes when hardware code needs to be tweaked.
- The Android Studio Guide was used as a reference resource to fill in gaps not covered in

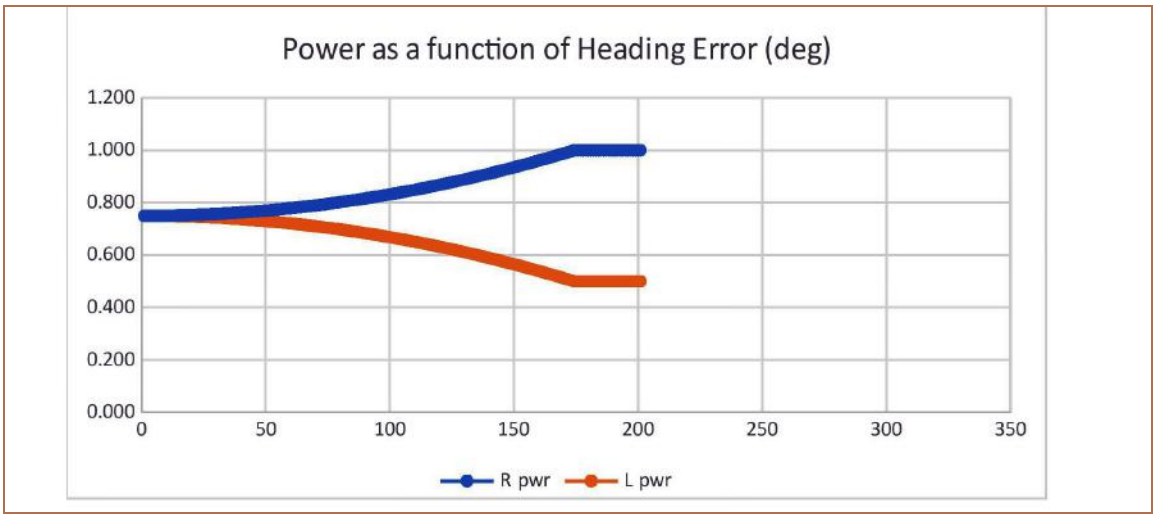
the book and to ensure we were current with the 2020/2021 library requirements.

- Joel already had some experience with PyCharm – a modern Integrated Development Environment (IDE). The team software mentor helped him get up-to-speed with the Android Studio IDE.
- Since Joel was brand new to *FIRST* robotics programming this year, our mentor assisted with the design of the more complex portions of the code (such as the state machine and machine vision). The goal of the mentor was to NOT re-use much software, but for the team to code from scratch as much as possible in straightforward, understandable chunks to maximize learning.

Design History:

- **Gamepad**: The gamepad controller button assignments were proposed and provided to the drive team during the software design review for feedback. As more subsystems were added to the robot during the season, button assignments were added and re-assigned based on feedback from the drive team.
- **Drivetrain**: The first system that was integrated was the Mecanum drivetrain that allowed driver control of the bare robot "foundation" using joysticks in "Drone Mode" and to create an Autonomous mode that could park the robot on the launch line using the drive motor encoders for speed and distance control.
- **Machine Vision – Ring Detection**: The next system that was integrated was the Webcam. We tackled this early because we viewed the machine vision problem as the highest software risk since OpenCV was not covered in the Learn Java for FTC book, and none of the team members, mentors, or coaches had hands-on experience with machine vision. After getting the Webcam integrated, we implemented the Ring Height Detection feature for Autonomous. This pipeline extracts a small rectangular region where the ring stack is expected to be and takes the average of how much "Red" (close enough to Orange for our purposes) appears in that region. Comparing that average to experimentally determined thresholds determines whether zero, one, or four rings are present.
- **Autonomous Navigation**: Each Autonomous route was developed and choreographed as a simple series of discrete steps executed sequentially. Functions were written to "Rotate to Heading (degrees)", "Drive to a Field Position (X, Y)", "Drive Forward/Backward (distance)". The built-in gyro (IMU) was used to control heading, and the drive motor encoders (average of all four motors) were used for distance. Joel came up with the mathematical approach to steer the robot in a straight line using IMU heading error to feed back into the commanded power for the left or right motors (see "error squared" relationship graph below). In addition, we used a trapezoidal acceleration/cruise/deceleration curve to ensure the robot didn't start/stop/travel too fast, because wheel

slippage would degrade our odometry and thus our positional accuracy on the field. Historical Note: Prior to integrating the wobble goal arm, we simply used our Mecanum drivetrain to push the wobble goal to the correct zone, while avoiding the ring stack.



- **Machine Vision – Aiming**: Next, the Webcam was utilized to be able to aim the robot when launching rings. The mentor and I brainstormed how to best detect the Tower Goal. We wanted to use this feature in TeleOp, especially to help with Power Shot aiming, so we were concerned with CPU usage. A complex machine vision algorithm might negatively impact Gamepad responsiveness. We observed that the "Low Goal" was very, very black. The 45-degree angle of the backstop ensured no light would reflect from this area, and we could be pretty confident that nothing else THAT black and THAT wide would be in the camera's field of vision when pointing at the back wall. To detect the goal, we only look at a narrow horizontal band (20 pixels in height) of the image that cuts across the middle of the Low Goal. This narrow band is processed as follows:
  - Convert the band to grayscale (to reduce the size of the data)
  - Apply a Gaussian Blur to the band (to eliminate any noise)
  - Search ONLY the center row of pixels to find the longest contiguous stretch of black pixels (black = a brightness less than 10). The graph below shows RGB data collected across this band, with the tower goal black region clearly distinguishable.

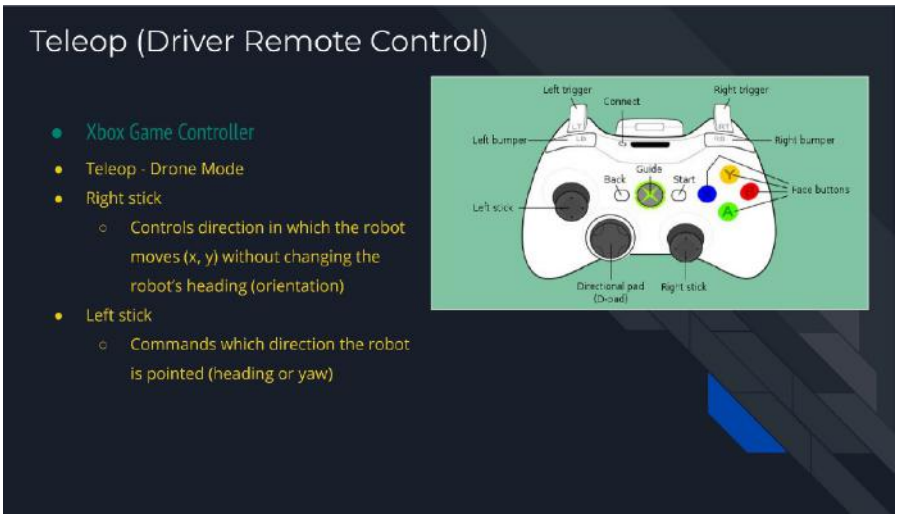
- If the contiguous black region is large enough, we consider the Tower Goal detected. We use the robot's estimated distance from the goal (based on IMU and odometry navigation) to estimate how wide the goal should appear.
  - The actual width of this region (in pixels) is in turn used to provide a more accurate estimate of the robot's distance to goal. This value is provided for potential use in automatically adjusting the ramp elevation angle to account for the distance of ring launches.
  - This is a very lightweight algorithm with a very small amount of data to process.
- **The Intake, Storage ("Water Wheel"), and Launcher (flywheel)** subsystems were integrated next. The software joins these three subsystems into a single integrated system.
  - A state machine was implemented to control the ring-handling interactions between these three hardware subsystems. The software for this turned out to be quite complex as the teeth on the Storage wheel must be precisely aligned as rings are fed in from the Intake.
  - Storage Wheel Tooth Alignment: To accomplish this, a color/distance sensor had to be added to detect the presence of a tooth as the wheel turns. Whenever the robot starts up in TeleOp, and whenever a driver manually "clears" the system, the robot executes a "Wheel Alignment" algorithm. This algorithm slowly turns the wheel, looking for a tooth. After it finds the tooth, it turns the wheel just enough so that a ring coming in from the intake can slide all the way into the bottom of the Storage wheel without striking any teeth. From this point on, the Storage wheel operates in RUN_TO_POSITION mode so that it always turns in

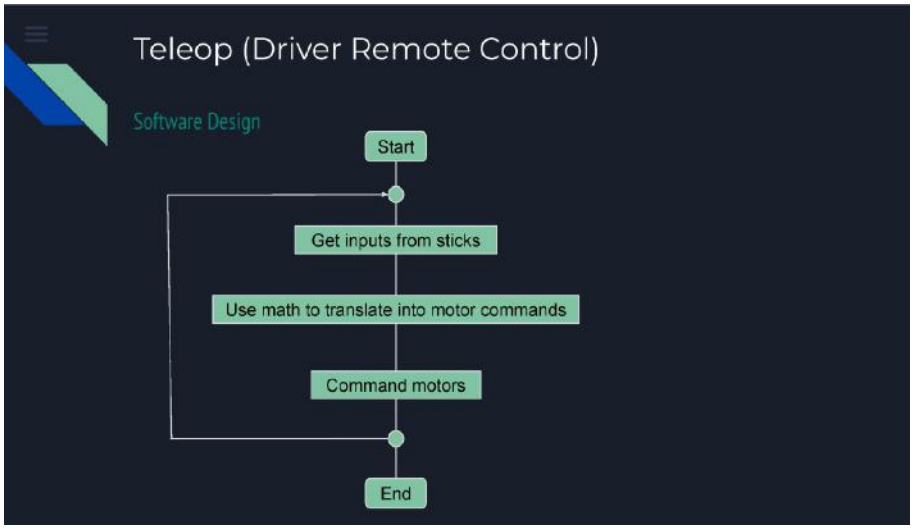
exact Tooth Increments (multiples of 1/7 revolution) to ensure it stays aligned.
  - The Intake also has a color/distance sensor so the robot can detect when a ring is in the Intake.
  - The Launcher flywheel velocity can be nudged faster/slower using buttons on the gamepad.
  - We found that the Intake draws a lot of power from the system, which can affect the Launcher motor speed and recovery. As a result, our goal was to store all 3 rings on the Storage wheel so that all 3 can be launched without running the Intake.
  - There is also a "clearing" feature where the driver team can manually turn the Storage wheel in forward or reverse to clear rings from the robot that may have gotten jammed or that got stuck in the robot due to a state machine malfunction. The Intake can also be run in reverse to help clear rings.
  - Both the Launcher Motor and the Storage Wheel Motor were PID tuned to provide precision control for velocity and position respectively.
- **Launch Flap**: The flap is controlled by a servo which turns a cam which controls the height of the flap so that rings can be launched at different elevations. There are preset elevations for the High Tower Goal and for the Power Shot Goals that can be selected by buttons on the gamepad. In addition, there are gamepad buttons to nudge the flap up/down for finer tuning based on robot performance (e.g., if a low battery slows down the launcher).
- **Indicator Lights and Telemetry**: Information is reported to the drivers via indicator LEDs on the robot and by telemetry text printed on the phone display.
  - LEFT LED: Indicates the number of rings the robot "thinks" it has loaded (OFF=none, RED=one, AMBER=two, GREEN=three). If the light blinks red, a manual "clearing" operation is underway. In Autonomous Modes the left LED indicates how many rings were detected (OFF=detection failed, RED=zero rings, AMBER=one ring, GREEN=four rings).
  - RIGHT LED: Indicates whether the tower goal is detected by the webcam (OFF=goal not detected, AMBER=goal detected, but NOT properly aimed, GREEN=Robot heading is properly aimed for a shot on goal). In Autonomous modes GREEN indicates the goal was detected and is being used for aiming. RED indicates the goal was not detected and aiming is based on IMU alone.
  - Telemetry text on the phone displays the following:
    - Tower goal detected/not detected
    - Tower goal estimated distance
    - Tower goal estimated heading delta (aiming error in pixels and degrees)
    - Launch Ramp position setting

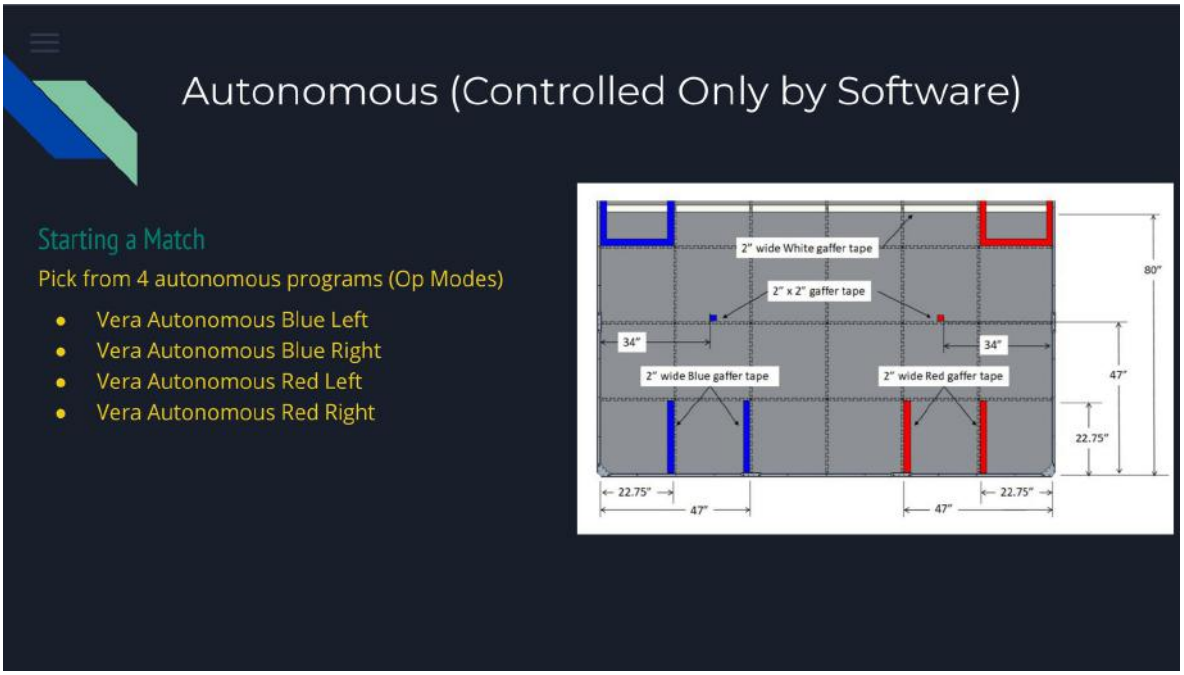
- Launcher motor commanded and actual speeds

- **Wobble Arm/Claw**: The Wobble subsystem was integrated last. The secondary gamepad controls a motor that moves the arm forward and backward, while a button toggles a servo to open/close the claw/gripper. The software limits the travel of the arm to prevent the arm from going too far back and striking the robot hardware or too far forward to strike the ground. There is a separate OpMode to allow the arm position to be "reset" to its stowed position. The arm limits assume the robot was started with the arm in the stowed position.

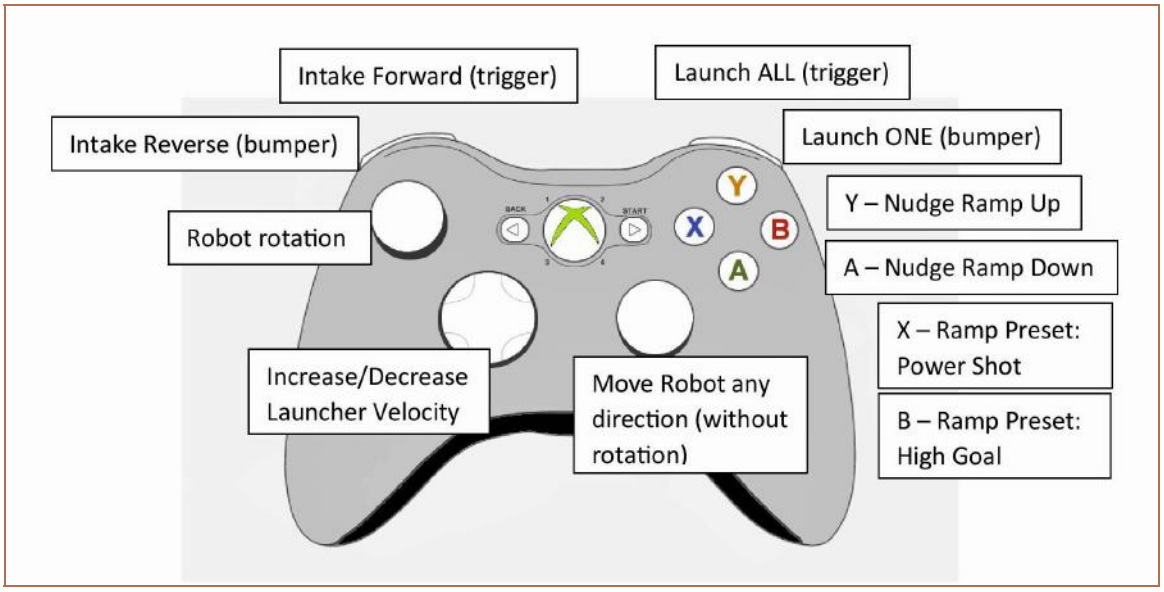
Preliminary Software Design:
During preliminary software design, the following charts were presented to the team.



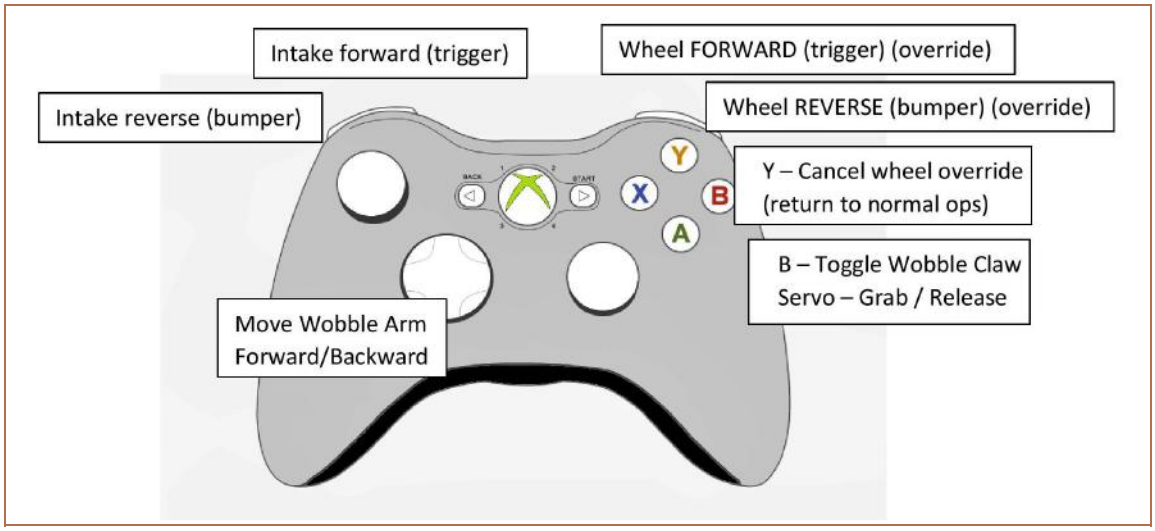




Current Design:
Gamepad Controls – Primary Driver



Gamepad Controls – Secondary Driver



Software Classes and Functions

Class: globals/CONSTANTS

This class provides a central place to define all constants for the system. These constants define values used by the software and cover the gamut of categories:

- Electronics: IMU and WebCam
- Drivetrain Behavior (e.g., scale factors for sensitivity)
- Robot dimensions
- Field dimensions
- Ring Height Detection Settings and Thresholds
- Tower Goal Detection Settings
- Wobble Arm/Claw Motion Extents and Preset Positions
- Intake Motor Settings and Run-Duration Times
- Storage "Water" Wheel Settings
- Launching (Aiming, Launcher Motor, Ramp)

Class: mechanisms/VeraHardware

This class compartmentalizes all interfaces to the hardware. The OpMode classes should not

interface directly with the hardware. The OpMode classes call into the wrapper functions defined in this VeraHardware class.

FUNCTION	SUMMARY
init	Initializes all hardware components. Each hardware component must be mapped from the phone configuration, where each device is assigned a port and a name. Any special settings are applied here, such as motor direction, motor behavior settings, motor tuning coefficients, and digital channel directions (Input/Output).
clearBulkCache	Clears the bulk cache in all hubs. Bulk caching improves robot performance which lowers cycle times for the main OpMode loop.
logCsvString	Adds a comma-separated-value string to a buffer which gets written to a CSV file when the robot stops. This is used to log debug data.
stopRobot	Writes out the CSV file buffer (if it is not empty) and returns a status string.
invertMotorDirections	Inverts all four drive motor directions (toggles Forward/Reverse).
setMotorsConstant	Sets all four motors to the same constant power value. This is primarily used to stop all four motors by setting them to zero power.
setLeftRightMotors	Sets one power value to both left drive motors and another power value to both right drive motors.
resetMotorRevolutions	This function causes the class to “remember” the current position of all the drive motors.
getMotorRevolutions	Gets the AVERAGE number of revolutions that all the drive motors turned since the last reset.
translateSticksDroneFlightControls	Translates drone style stick inputs (pitch, yaw, roll, thrust) to drive motor power values.
scaleMotorPower	If any of the motor power commands are greater than 1, all the motor power commands are scaled down equally to ensure power commanded is not greater than 1.
commandDriveMotors	After calling scaleMotorPower, this function sets the commanded powers to each of the four drive motors.
setIntakePower	Sets the power of the Intake motor.
setLauncherPower	Sets the power of the Launcher motor. This is only used when running WITHOUT the PID encoder.
getLauncherSpeed_tps	Gets the velocity of the launcher motor in ticks per second.
setLauncherSpeed	Sets the launcher target velocity in ticks per second.
getWaterWheelPosition_rev	Gets the position of the water wheel in revolutions.
getWaterWheelSpeed_tps	Gets the velocity of the water wheel in ticks per second.
turnWaterWheelToPosition	Turns the water wheel to the specified position at the specified velocity.

FUNCTION	SUMMARY
isWaterWheelBusy	Returns TRUE if the water wheel is busy moving to a specified position.
stopAndResetWaterWheel	Sets the water wheel power and velocity to 0, and stops the motor encoder.
clearWaterWheelForward	Runs the water wheel motor at full power without the encoder.
clearWaterWheelReverse	Runs the water wheel motor at full reverse power without the encoder.
getClawArmMotorPosition	Gets the current position of the wobble arm motor.
setClawArmSpeedPwr	Sets the power of the wobble arm motor.
moveClawArmToPosition	Moves the claw arm to the specified position at a constant speed.
setClawServoPosition	Sets the claw servo to the specified position.
setRampServoPosition	Sets the ramp servo to the specified position.
isGyroCalibrated	Returns TRUE if the IMU is calibrated.
getGyroCalibrationStatus	Returns a string containing the IMU calibration status.
getHeading	Gets the robot heading from the IMU in the specified angle units.
startWebcamStreaming	Sets the pipeline for the webcam to the input pipeline specified and starts asynchronously streaming frames through the pipeline.
changeWebcamPipeline	Changes the streaming pipeline of the webcam to the input pipeline specified.
stopWebcamStreaming	Stops the webcam streaming.
getIntakeSensorDistance	Gets the number of inches the Intake distance sensor is reporting (to detect presence of a ring in the intake).
getToothSensorDistance	Gets the number of inches the Storage wheel distance sensor is reporting (to detect presence of a wheel tooth).
setLedState	Sets a specified Red/Green LED pair of lights to the specified color (AMBER is achieved if both Red and Green are lit).
setAimingLED	Sets the color for the Aiming LED.
setLoadingLED	Sets the color for the Loading LED.

Class: pipelines/RingHeightPipeline

This class is responsible for machine vision to detect how many rings are in the Ring Stack during Autonomous operation. It sets up a webcam stream pipeline and processes image frames as they flow through the pipeline.

FUNCTION	SUMMARY
RingHeightPipeline	Constructs the class, setting the sample box size and position.
inputToCr	Takes an input RGB frame, converts it to YCrCb, and extracts the Cr channel to the m_matCr variable. Cr is the red-difference chroma component.

FUNCTION	SUMMARY
init	Calls inputToCr to extract the red-difference chroma component and clips out the sample box portion of the frame.
processFrame	Extracts the red-difference chroma component of the current frame; computes the "red average" of the sample box. Sets m_ringHeight variable depending on how much red was found in the sample box. Also draws boxes on the output image to indicate the location of the ring stack sample box and the tower goal detection box.
getRingHeightAnalysis	Returns the "red average" value found during processFrame.
getRingHeight	Returns the ring height value found during processFrame.

Class: pipelines/TowerGoalPipeline

This class is responsible for machine vision to detect the heading to the Tower Goal in Autonomous operation. It sets up a webcam stream pipeline and processes image frames as they flow through the pipeline. It is used to aim the robot for shots at either the Tower Goal or the Power Shot targets.

FUNCTION	SUMMARY
setYLocation	Sets the robot's Y location to the specified value. This allows the pipeline to know how wide the tower goal should appear.
getGoalHeadingDelta_deg	Returns an estimated heading delta for aiming at the center of the tower goal, in degrees.
getExpectedGoalWidthAtCurrentDistance_pix	Gets the pixel width the tower goal is expected to span at the current distance.
isTowerGoalDetected	Returns TRUE if the tower goal is detected.
init	Does nothing for this pipeline.
processFrame	Extracts the sample box (narrow horizontal band across the image); Converts the subimage to grayscale; Performs a Gaussian blur to remove noise (blur size 5, sigma 0); Scans the middle row of the subimage to find the longest contiguous run of black pixels; If a wide enough run is found, then sets m_goalWidth_pix to the width of the run and sets m_aimOffset_pix to the offset of the middle of the run from the middle of the camera view.

Class: RobotLocation

This class is used to keep track of where the robot thinks it is during Autonomous operation.

FUNCTION	SUMMARY
RobotLocation	Constructor that saves the specified X, Y position and heading.
getHeading_deg	Returns the current heading (normalized to -180..180).
toString	Returns the location formatted as a string.
setAngle	Sets (saves) the heading angle specified.
setPos	Sets (saves) the X, Y position specified.
getPos_in	Returns the current X, Y position.

Class: TeleOpLVera

This class contains all the "middle ware" as well as the "main loop" (runOpMode function) to run the robot during Driver Operated TeleOp mode.

FUNCTION	SUMMARY
initializeHardware	Initializes the "Vera" hardware mechanism, passing in flags relevant to TeleOp (enabling bulk caching, autonomous=false, use Launcher PID).
stopRobot	Calls the "Vera" hardware stop method.
getMotorInputsFromSticks	Passes the current inputs from the two primary driver joysticks to the Vera hardware. This translates the sticks using "Drone mode" into power commands for the four wheels.
getCommandsFromButtons	Gets inputs from both gamepads and saves them as commands in various variables that will be accessed later in the main loop processing.
getInputs	Clears the bulk cache; Gets inputs from all sensors on the robot (distance sensors, vision pipelines, IMU, motor/servo speeds and positions; Gets inputs from gamepads by calling getMotorInputsFromSticks and getCommandsFromButtons.
isRingInIntake	Returns TRUE if the distance sensor reports a ring in the intake.
isSecondRingInIntake	Returns TRUE if the distance sensor has reported a ring in the intake in the last several samplings. This is used to determine whether another ring has entered the intake while the previous ring was being moved to the Storage wheel.
RotateToHeading	Commands the robot to rotate to the specified heading.
AutoAimAtTowerGoal	If enabled and if the tower goal is detected, computes a new heading to aim at the tower goal; then calls RotateToHeading to perform the aiming.
turnWaterWheel	Commands the water wheel to turn to the specified position at the specified speed.
waterWheelClear	Commands the water wheel to turn full speed as specified (forward or reverse). Also controls the red blinking of the loading LED. If the user has notified that clear is complete, the state machine states are reset to initial conditions.

FUNCTION	SUMMARY
manageWaterWheelState	Ensures the water wheel is in TEETH_ALIGNED state. If not, various states are transitioned to run the wheel alignment algorithm to accurately detect a tooth and properly align the wheel based on tooth position.
commandIntakeAndLaunchSystem	Performs all the logic for the state machine. Based on the current m_launcherState value, logic is performed to transition to the next state. This logic may involve overriding the intake commands from the user. After processing the current state, the intake power is commanded.
commandClawArm	Sets the wobble arm power as requested, and toggles the claw servo position as requested.
runOpMode	Performs the following: •Calls initializeHardware •Starts the tower goal vision webcam pipeline •Waits for start •Executes the Main LOOP ◦getInputs ◦commandDriveMotors ◦setRampServoPosition ◦commandClawArm ◦commandIntakeAndLaunchSystem ◦Report information (LED lights and telemetry) •Calls stopRobot

Class: AutoLVera_BlueLeft

This class acts as a Base Class for all of the Autonomous OpModes. There are separate Autonomous OpModes depending on which start line the robot will be starting from, and also some separate modes for risking Power Shots instead of Tower Goal launches. This class contains all the "middle ware" for ALL the Autonomous OpModes. It also contains the "main loop" (runOpMode function) for the Blue-Left start line case.

FUNCTION	SUMMARY
initializeHardware	Initializes the "Vera" hardware mechanism, passing in flags relevant to TeleOp (disabling bulk caching, autonomous=true, use Launcher PID). Allows time for IMU to calibrate.

FUNCTION	SUMMARY
SetPosition	Sets the robot X, Y position as specified.
SetAngle	Sets the robot heading as specified.
GetRingStackHeight	Repeatedly samples the vision pipeline until non-zero samples start to flow, then returns the detected ring height.
DriveStraight	Drives in a straight line for the specified distance, in the specified direction (forward/reverse). The current heading and the new X, Y position are also passed in (to avoid recomputing them). Applies a trapezoidal velocity curve to smoothly accelerate/decelerate. Detects heading errors and applies steering corrections to maintain commanded heading.
DriveForward	Computes geometry and calls DriveStraight (forward)
DriveBackward	Computes geometry and calls DriveStraight (reverse)
RotateToHeading	Rotates the robot to the specified heading.
DriveToPosition	Computes geometry and calls both RotateToHeading and then DriveStraight to get to the commanded X, Y position.
StopRobot	Stops webcam, stops drive motors, and calls stopRobot in the hardware.
InitializeAndStart	Calls initializeHardware; Starts the ring height detection pipeline; Waits for start; Gets the ring stack height; Sets the loading LED based on ring stack height; Grabs the wobble goal; Switches to the tower goal webcam vision pipeline.
MoveFromStartWall	Moves away from the start wall so robot rotations do not strike the wall.
DropWobbleGoal	Opens the claw servo.
RotateToAimAtGoal	Computes heading to the tower goal and calls RotateToHeading.
DriveAndLaunchAllHigh	Starts up the launcher motor; Sets the ramp elevation; Drives to the launch position; Rotates due North; If the tower goal is detected, lights LED green and calls RotateToAimAtGoal; Otherwise lights LED red and calls RotateToHeading with a default. Launches three rings.
DriveAndLaunchPowerShots	Same as DriveAndLaunchAllHigh but with different launch position, different ramp elevations, and different rotations between each launch.

FUNCTION	SUMMARY
runOpMode	Performs the following sequential steps: • Calls InitializeAndStart • Calls MoveFromStartWall • Calls DriveToPosition (to avoid ring stack) • Based on ring stack height: Calls DriveToPosition, DropWobbleGoal, and DriveBackward. • Calls DriveAndLaunchAllHigh • DriveToPosition (to park on launch line)

AUTONOMOUS CLASSES – Each Inherits from AutoLVera_BlueLeft

Each autonomous OpMode (other than AutoLVera_BlueLeft) is a very simple class that contains only one function - the runOpMode function (which overrides the one in BlueLeft). In addition, the code in the runOpMode function is typically just a few lines of code that performs understandable, sequential steps.

The autonomous classes (OpModes) include:

- class **AutoLVera_BlueLeft** (the base class for all the others)
- class **AutoLVera_BlueRight**
- class **AutoLVera_RedLeft**
- class **AutoLVera_RedRight**
- class **AutoLVera_RedLeftPower** (shoots power shots instead of high tower goal)

The runOpMode function in each of these classes calls functions to perform straightforward tasks. During initialization, the following functions are typically called:

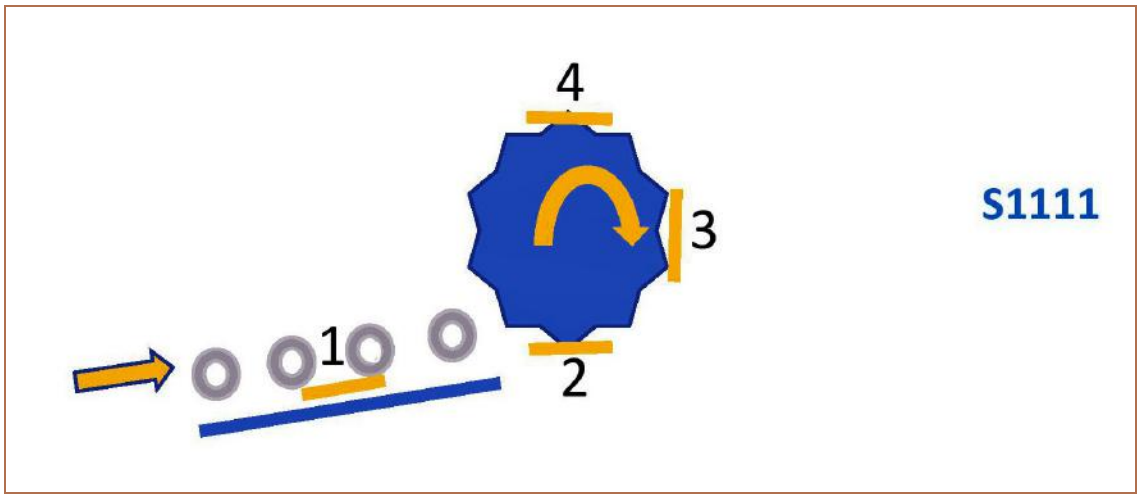
- InitializeAndStart (detects ring stack height)
- MoveFromStartWall
- DriveToPosition(x, y)
- waitForStart (waits for the driver to press Play on the phone)

To execute the autonomous tasks, the following functions are typically called in varying sequences:

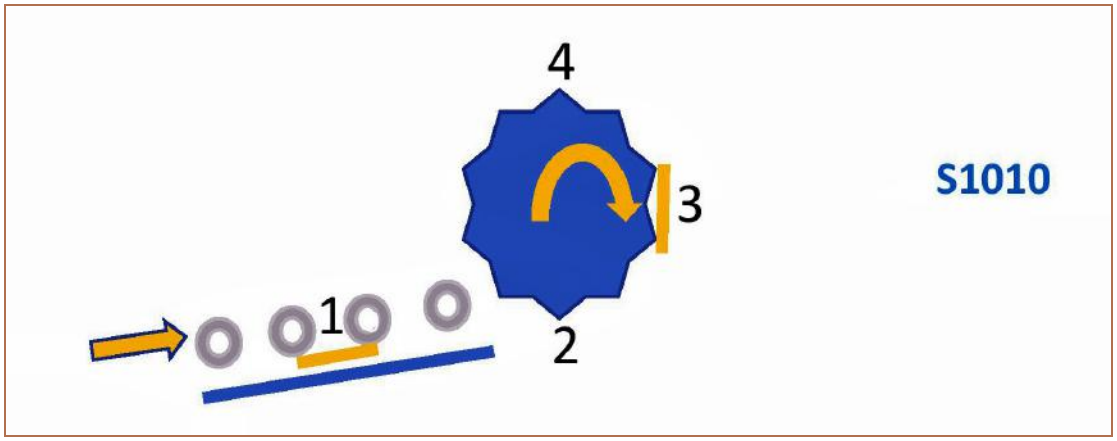
- DriveToPosition(x, y)
- DropWobbleGoal
- DriveForward(distance)
- DriveBackward(distance)
- RotateToHeading(angle)
- DriveAndLaunchAllHigh – Drives to a pre-defined launch position before launching
- DriveAndLaunchPowerShots – Drives to a pre-defined launch position before launching
- StopRobot – always called at the end

State Machine States, Events, and Transitions

A state machine is used in TeleOp to manage the logic of ring handling (intake, storage, and launching). The states are named to represent where rings are present within the Intake and Storage Wheel. A ring can move into the Intake and around the Storage Water Wheel and be in one of four positions as shown in the diagram below.



The states are named to graphically depict the progression of rings through the four possible positions, from left to right. In the diagram above, with all four rings present, the state would be S1111. In the diagram below, the state would be S1010.



The initial letter of each state represents different types of states.

- S = Normal "static" state.
- T = Transition state. The previous and new ring states are shown in the state name.
- TA = Transition AIM state. The robot will rotate to auto-aim (if enabled) in this state.
- TL = Transition LAUNCH state. Note that there are "Launch ONE" and "Launch ALL" commands.

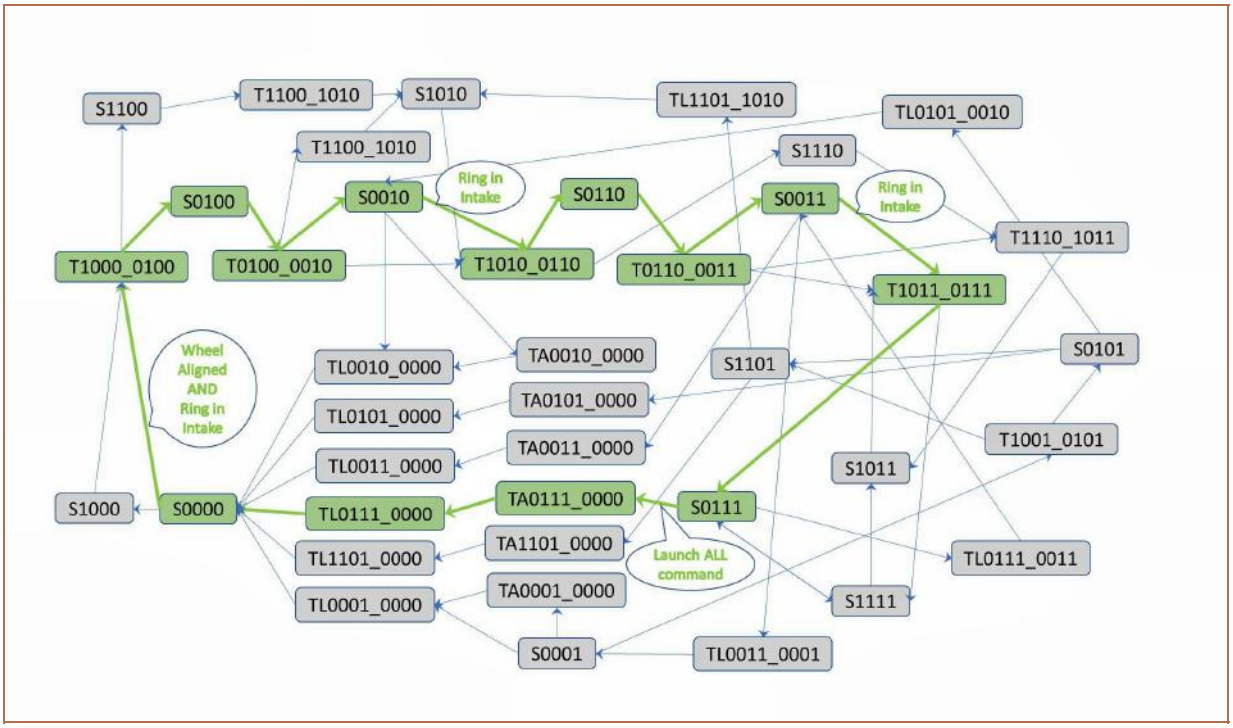
Normally, the "transition" states wouldn't be states at all, but would be the arcs between the states. However, it turned out to add clarity to the code to make formal states out of them in order to monitor and precisely control hardware operations and transitions.

Each state assigns the appropriate color for the LED Loading Indicator light (how many rings are loaded).

The diagram below illustrates the states and all possible transitions between them. A table follows on the next page which explains what events trigger the transition from one state to the next. The fact that the robot supports launching ONE ring at a time adds a lot of complexity. All of the "TL" states around the top, right, and bottom edges of the graph handle this job, and introduce many extra states and possible transitions (though perhaps rarely used in competition).

The Green states and transitions highlight the ideal path followed when the Driver Team loads one ring at a time for a full load of 3 rings and then launches them all. S0000 is the start state (no rings loaded).

State Machine States and Possible Transitions



CURRENT STATE	EVENT ------> TRANSITION If no event, the state remains the same	NEW STATE
The Initial State		
S0000	Ring in Intake AND wheel aligned	T1000_0100
	Ring in Intake	S1000
States where rings are stored in the robot		
S0010	Ring in Intake	T1010_0110
	Launch ALL command	TA0010_0000
	Launch ONE command	TL0010_0000
S0011	Ring in Intake	T1011_0111
	Launch ALL command	TA0011_0000
	Launch ONE command	TL0011_0001
S0111	Ring in Intake	S1111
	Launch ALL command	TA0111_0000
	Launch ONE command	TL0111_0011

CURRENT STATE	EVENT ------> TRANSITION If no event, the state remains the same	NEW STATE
S0001	Ring in Intake	T1001_0101
	Launch ALL command	TA0001_0000
	Launch ONE command	TL0001_0000
S0101	Ring in Intake	S1101
	Launch ALL command	TA0101_0000
	Launch ONE command	TL0101_0010
S1101	Launch ALL command	TA1101_0000
	Launch ONE command	TL1101_1010
Temporary states where rings on the wheel are about to advance		
S0100		T0100_0010
S0110		T0110_0011
S1100		T1100_1010
S1110		T1110_1011
States where a ring is about to move from intake to the wheel		
S1000	Wheel aligned	T1000_0100
S1010		T1010_0110
S1011		T1011_0111
States for advancing a ring from intake to wheel		
T1000_0100	If ring has advanced to wheel AND another ring has entered intake	S1100
	If ring has advanced to wheel	S0100
T1010_0110	If ring has advanced to wheel AND another ring has entered intake	S1110
	If ring has advanced to wheel	S0110
T1011_0111	If ring has advanced to wheel AND another ring has entered intake	S1111
	If ring has advanced to wheel	S0111
T1001_0101	If ring has advanced to wheel AND another ring has entered intake	S1101
	If ring has advanced to wheel	S0101
States for advancing rings on the wheel		
T0100_0010	Ring in Intake AND current ring not done	T1100_1010
	Ring in Intake AND current ring done	T1010_0110
	Current ring done	S0010
T0110_0011	Ring in Intake AND current rings not done	T1110_1011
	Ring in Intake AND current rings done	T1011_0111
	Current rings done	S0011
T1100_1010	Current ring done	S1010
T1110_1011	Current rings done	S1011

CURRENT STATE	EVENT ------> TRANSITION If no event, the state remains the same	NEW STATE
States for launching ONE ring		
TL0011_0001	Current ring launched	S0001
TL0101_0010	Current ring launched	S0010
TL0111_0011	Current ring launched	S0011
TL1101_1010	Current ring launched	S1010
States for aiming the robot prior to a launch ALL rings		
TA0001_0000		TL0001_0000
TA0010_0000		TL0010_0000
TA0011_0000		TL0011_0000
TA0101_0000		TL0101_0000
TA0111_0000		TL0111_0000
TA1101_0000		TL1101_0000
States for launching ALL rings		
TL0001_0000	Last ring is launched	S0000
TL0010_0000	Last ring is launched	S0000
TL0011_0000	Last ring is launched	S0000
TL0101_0000	Last ring is launched	S0000
TL0111_0000	Last ring is launched	S0000
TL1101_0000	Last ring is launched	S0000
States for error/override conditions		
S1111	Illegal ring has been ejected AND previous ring is located in Intake	S1011
	Illegal ring has been ejected AND previous ring is located in the wheel	S0111
WHEEL_CLEAR_FORWARDS	Driver indicates clear is complete	S0000
WHEEL_CLEAR_BACKWARDS	Driver indicates clear is complete	S0000