A Step by Step Runthrough of FRC Vision Processing

David Gerard and Paul Rensing
FRC Team 2877
LigerBots

Contents

1 Abstract **2**

2 Goals of Vision Processing **2**

3 Vision System Configurations **2**
- 3.1 Vision Processing on the roboRIO 4
- 3.2 Vision Processing on a Coprocessor 4
- 3.3 Vision Processing on Cell Phones 5
- 3.4 Vision Processing on the Driver's Station 6
- 3.5 LigerBots 2018 Choice 6

4 Comparisons of Coprocessors **6**
- 4.1 Coprocessor Operator Interface 8

5 Coprocessor Communication **9**

6 Cameras **9**
- 6.1 Choice of Camera 9
- 6.2 Camera Placement 11
- 6.3 Camera Mounting 11
- 6.4 Camera Cabling Challenges 12

7 Programming Environment **12**
- 7.1 Language Choice: Python vs. C++ vs. Java 12
- 7.2 Vision Processing Framework 13

8 Code Breakdown **13**
- 8.1 Camera Calibration 14

9 Coordinate Systems and Geometry **14**
- 9.1 Coordinate Systems 15
- 9.2 Coordinate Transformations 16
- 9.3 OpenCV Python Code 17

10 Additional Problems We Encountered **19**

11 Appendix **20**
- 11.1 Compiling OpenCV for the ODROID 20
- 11.2 Camera Tilt 21

1 Abstract

This paper provides a detailed analysis of the implementation of vision processing for both rookie and veteran FRC robotics teams. After first hand experiences of vision processing throughout the 2018 Power Up season and years prior, we have recognized both our successes and failures. After compiling these key findings together, we created this guide to lead teams through current and future challenges, in hope to aid them in putting together a complete and accurate vision system in sync with their robot.

2 Goals of Vision Processing

When first discussing ways to implement vision processing into a robot, it's important to consider the impact the work will have on your team's success. Often, depending on certain circumstances and requirements, vision may not be useful enough to consider adding to a repertoire of abilities. Many teams in the FRC 2018 game, FIRST Power Up, chose not to include vision processing in their robot due to the limited uses of cube and switch target detection throughout the game. Although, if chosen to be incorporated into a system correctly and efficiently, vision processing can in fact prove to be incredibly helpful in:

- Creating autonomous routines
- Constructing predefined actions for teleop
- Enhancing the driver's abilities (streaming a camera feed to the driver station)
- Programming the robot

With vision processing, rather than using brute force reckoning during the autonomous and teleop periods, a team can instead ensure certain aspects of their gameplay and completely change their strategies for the better.

In addition to increasing the accuracy of the robot code, having a solid system for tracking targets helps drivers during a match with tough visibility or under the pressure of time. Just helping a robot reduce even a second or two from its cycle time by simply aligning to a small target or providing the driver with helpful information can make a significant difference, allowing a team to help out their alliance and possibly win close matches. Although, after deciding to incorporate vision processing into a design, many other aspects and trade-offs of the system must first be considered before continuing. They can notably affect the efficiency and effectiveness of the system.

3 Vision System Configurations

A team must consider system configurations carefully and thoroughly examine what fits their requirements. Here are a few configurations the LigerBots considered throughout the years. Some have worked better for us than others, as explained later on, but it's up to a team themselves to figure out what works best for them.

- Processing directly on the roboRIO (USB or Ethernet camera)
- Processing on a coprocessor (USB or Ethernet camera, results sent to roboRIO)
- Processing on the driver's station (USB or Ethernet camera, results sent to roboRIO)
- Processing on cell phone(s) (results sent to roboRIO)

As seen in Figure 1, there is increasingly complex depth in assembling a complete set-up. We recommend going through at least the top level of the workflow below to understand each individual aspect.

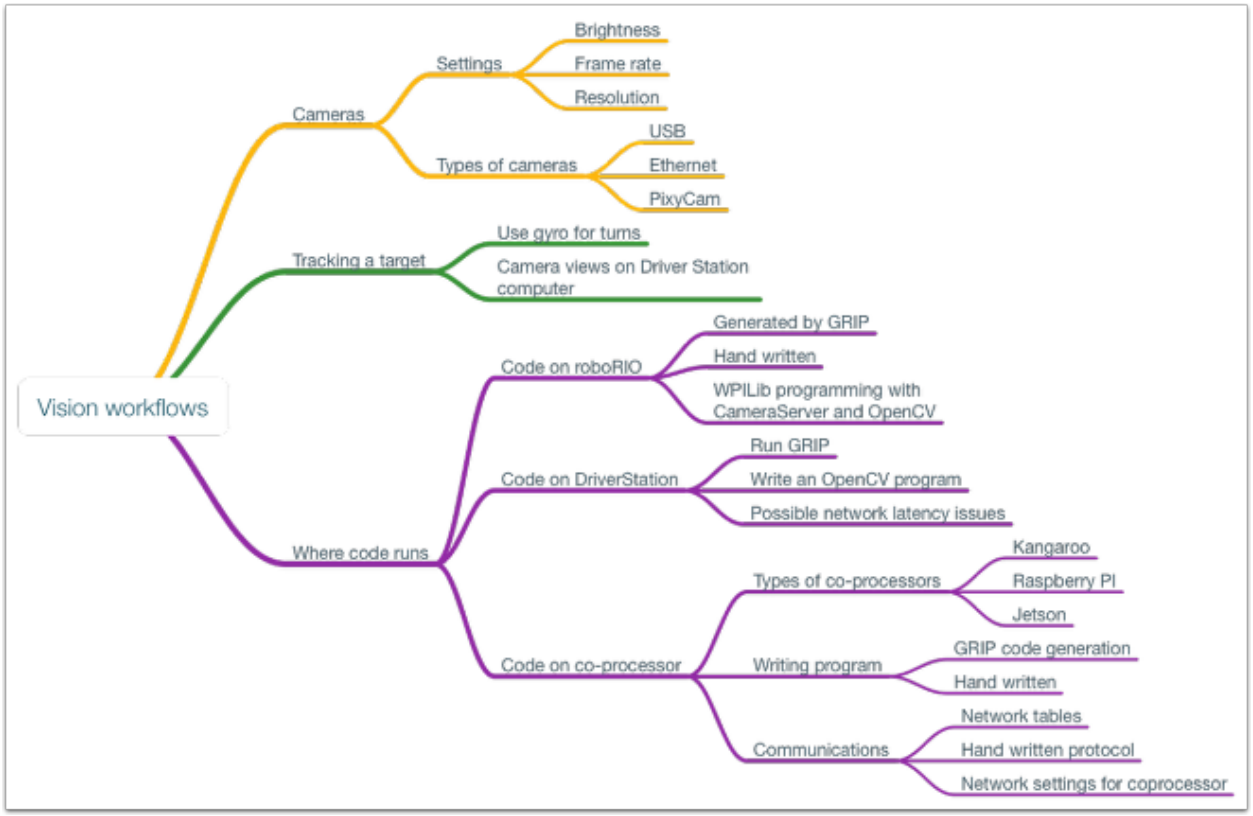


Figure 1: Various choices for the complete vision processing system

3.1 Vision Processing on the roboRIO

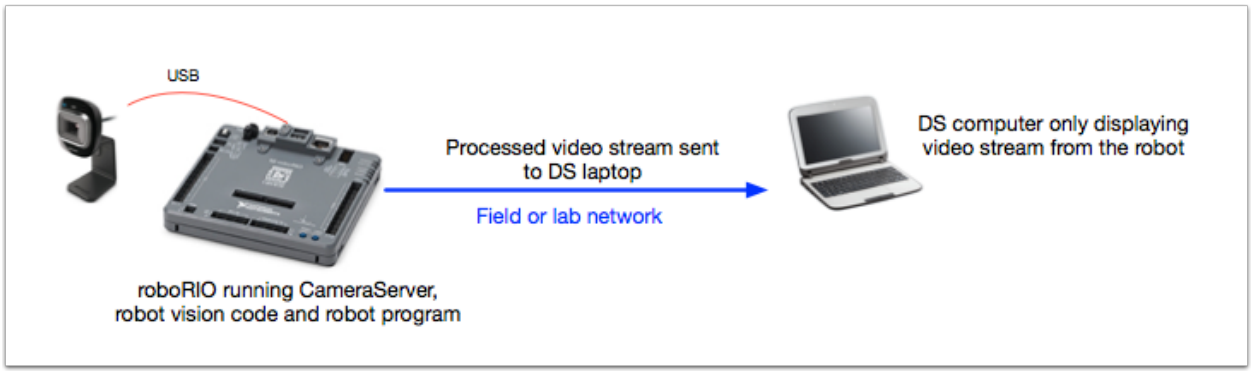


Figure 2: Schematic of vision processing on the roboRIO

One crucial item to consider while creating the basis for an efficient system is the platform on which to run processing code. This can be done either on the main processor itself or on a separate coprocessor. It is recommended to have a coprocessor run the vision code rather than running it directly on a roboRIO (unless the camera is simply being streamed directly to the driver's station without additional processing) because the vision processing may delay the robot code's cycle time on the roboRIO. This will create jolting movements due to a long loop period and numerous other possible problems.

- OK for simple streaming to DS
- Potential lag in robot control

3.2 Vision Processing on a Coprocessor

Having a coprocessor will ensure that these types of problems will not happen and will increase the speed of the processing.

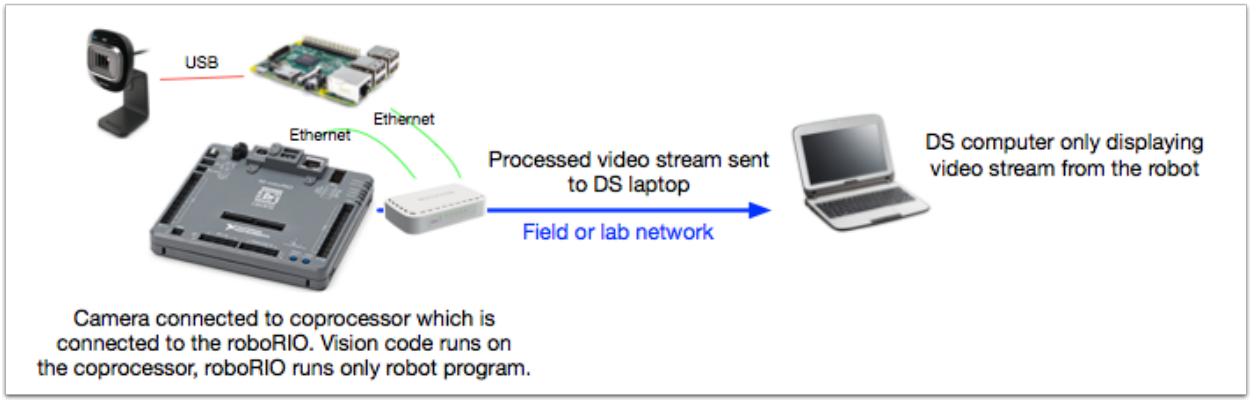


Figure 3: Schematic for vision processing on a coprocessor

As shown in Figure 3, the radio connected to the roboRIO is also connected to the co-processor. The coprocessor is constantly processing the images it receives from the attached USB camera. Then, through a system such as NetworkTables, the coprocessor sends the processed information to the roboRIO for the robot code to utilise during operation.

- Pros:
  - Can run vision processing at a frame rate independent of the rate of images sent to DS
  - No risk of overloading the roboRIO
- Cons:
  - Extra hardware: power supply, Ethernet
  - Communication delays (NetworkTables)

3.3 Vision Processing on Cell Phones

For the 2017 season, the LigerBots did vision processing on cell phone cameras. We used 2 phones on the robot, one aimed forward for the gear placement mechanisms, and one angled upward to aid in targeting the Boiler. The phones were connected to the roboRIO using IP networking over USB cables. This has a number of perceived advantages:

- A good camera is built into the phone
- Vision processing can happen directly on the phone. Cell phone processors can be quite powerful, depending on the model.
- The price can be reasonable, especially if you purchase past generation used phones. The LigerBots used two Nexus 5 phones.

However, while the cameras did work reasonably well, the experience gained during the season convinced us that it is not our preferred solution:

- The phones are large, heavy, and hard to mount. The mounting required a significant amount of space, particularly since we needed to remove the phone frequently (for charging, etc.).
- The weight of the phone plus 3d printed mount meant that the cameras were significantly affected by vibration when the robot was moving. This made the hoped-for driver assist very tricky.
- An unexpected twist was that using 2 phones with USB networking required modifying, compiling, and installing a custom Android kernel on one of the phones. The issue was that the IP address assigned to the phone for USB networking is ***hard-coded in the kernel***, and therefore the two phones had the same IP address, which would not work. A student installed a new kernel, but this is very specialized knowledge.

- The phones would get very hot during a match (we used them throughout for driver assist), and needed to be removed frequently to cool and be charged.
- The vision processing needs to be programmed in Java (only), and it needs to be packaged as an Android app which auto-starts. Again, this is more specialized knowledge.

Given all the factors, we decided we did not want to continue using the phones for vision processing.

3.4 Vision Processing on the Driver's Station

While the LigerBots do not have any experience with this setup, many teams have successfully used vision processing on the Driver's Station (DS) laptop. In this setup, a streaming camera is set up on the robot (Ethernet camera, or USB camera connected to roboRIO). A program runs on the DS which grabs a frame, processes it, and sends the necessary results back to the roboRIO via NetworkTables.

- Pros:
  - Plenty of processing power on DS
  - Easy programming environment (Windows laptop with system of your choosing)
- Cons:
  - Extra lag between image capture and results getting to roboRIO. Image frames and results are both transmitted over WiFi FMS.
  - Captured images should be shared between processing and Driver's Station (for display to driver), otherwise you will use double the bandwidth.

3.5 LigerBots 2018 Choice

During the 2018 season, the LigerBots chose to incorporate a coprocessor into the robot design due to our experiences over the years. The coprocessor allowed our team to run a processing routine without interrupting the roboRIO. Additionally, having a coprocessor enabled additional further freedoms not only in the code, but also in the methods of communication between the two processors and other systems.

4 Comparisons of Coprocessors

We wanted to investigate numerous coprocessors to find which processor was realistic and could quickly run the algorithms we required. Some processors which we looked at early on were the RaspberryPi 3, ODROID-C2, ODROID-XU4, and the Jetson.

- Raspberry Pi 3
  - $35

  - 4-core ARM with custom CPU
  - Faster than RPi 2 but still somewhat slow
  - Geekbench 4 specs: around 1,000 (multi-core)

- ODROID-C2
  - https://www.hardkernel.com/shop/odroid-c2/
  - $46
  - 4-core Cortex-A53 with Mali-450 GPU
  - Geekbench 4 specs: about 1675 (multi-core), about 1.6x faster than the RPi3

- ODROID-XU4
  - https://www.hardkernel.com/shop/odroid-xu4/
  - $59
  - Samsung Exynos5422 (8-core: 4 fast, 4 slow) + Mali-T628 GPU
  - Geekbench 4 specs: 2800-3000 (multi-core), about 3x faster than the RPi3, about 2x faster than a C2
  - Mali-T628 has full OpenCL 1.1 profile available, so OpenCV can use the GPU.
  - Uses 5V 4 Amp (max) power, so more than typical. Not USB powered.
  - Has cooling fan, but runs only when needed.

- Jetson
  - Actually a line of boards from nVIDIA
  - NVIDIA Kepler GPU with 192 CUDA cores
  - NVIDIA 4-Plus-1 quad-core ARM Cortex-A15 CPU
  - $250 - very expensive and large compared to the other coprocessors

These computer boards are of course not the only possibilities for coprocessors and it is recommended to search deeper for ones that meet a team's requirements. We decided to test the RaspberryPi 3 (since we could borrow one), and an ODROID-XU4 because of its favorable price and performance specs. We did not do any testing of the ODROID-C2 nor Jetson boards.

To test the different boards' abilities and efficiency, and find which one to use, we wrote a simple test program. In order to not have to deal with the trouble of compiling, etc., we used OpenCV with Python. This is mostly not much slower than C++, since all the heavy lifting is done in the compiled OpenCV C++ code (see discussion later in this paper). The test routine was a re-implementation of our 2017 Peg finding code. The algorithm was about 90% the same as our older (Java) production code, so does actually represent a realistic processing chain.

Table 1 shows the test results, breaking down the timing of the important steps in the image processing. Times are in milliseconds needed to process one image, computed by

	RaspberryPi 3	**XU4 w/ OpenCL**	**XU4 OpenCL disabled**
cvtConvert BGR2HSV	12.38	0.55	4.97
threshold HSV image	14.52	2.61	3.25
findContours	6.49	4.16	1.79
cvtColor+threshold+findContour	33.12	7.29	10.21
peg recognition (read + process)	66.63	20.38	20.90
read Peg JPGs	24.21	7.03	6.90
process Peg JPGs	42.28	13.30	13.96

Table 1: Program Timing (milliseconds)

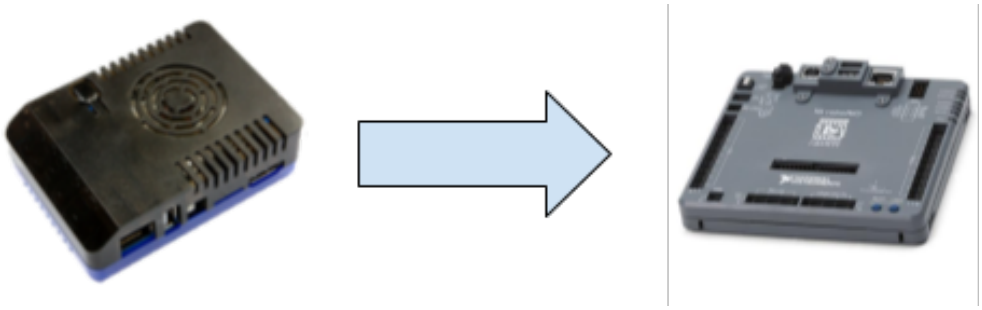
averaging 100 runs. The input data is a collection of pictures taken by WPI, both with and without the peg target; resolution is 640x480, which is higher than normally used in competition.

The tests indicate that the XU4 is about 3x faster than a RPi3, which matches the online specs of the boards. The RPi3 takes about 42 msec to process a frame (640x480), which is just too slow. The XU4, however, can do it in 14 msec, so it can keep up with a 30 fps stream, although with noticeable delay. Therefore, we chose use the ODROID-XU4.

4.1 Coprocessor Operator Interface

In order to configure and program the ODROID, we use the standard program PuTTY (https://putty.org/) to connect via SSH. If the ODROID is not on the robot, it can be connected directly to a laptop via an Ethernet cable, or connected to a normal (Ethernet) network. When the ODROID is on the robot, you would normally connect with Ethernet via the WiFi radio, just like connecting to the roboRIO. To transfer files to the ODROID, we use WinSCP (https://winscp.net/eng/index.php), but other file transfer programs will work, such as FileZilla or straight "scp" (Mac or Linux). Simply open the application and connect to the external file system. The interface enables the user to transfer files between the laptop and ODROID.

5 Coprocessor Communication



In order to communicate between the ODROID and the roboRIO, we used the standard NetworkTables ("NT") from WPILib, updating the information each cycle of the program. This allowed us to set up both a testing and main system.

In the test mode, the code sets up the ODROID as the NT server rather than the roboRIO. This allows changes of the NT properties to occur from the ODROID such as HSV thresholds, camera height, or an image writer state (a flag indicating that the program should save captured images to the ODROID's storage). This testing mode proved to be extremely helpful throughout the season during quick testing and tweaking of different constants to maximize the program's accuracy. Additionally, a NT variable named "tuning" defined whether or not the vision algorithms should fetch new data for image thresholding and distortion each loop. During matches, we kept "tuning" off, allowing the code to be more efficient, skipping over possible expensive operations that would not be required at the time. However, during testing phases, we kept tuning on so the values updated in NT could be implemented through each loop.

In the main NT initialization set-up, the ODROID is set as an NT client, with the roboRIO as the server. This allows the roboRIO to have control over the program during matches to control aspects such as which camera or mode to use. Make sure to keep track of which mode the NetworkTables are in to avoid confusion. Only one system can act as the NT server, and all components must talk to the same server in order for everyone to see the correct values and changes. We suggest creating some sort of indicator displaying the mode just to be safe.

6 Cameras

6.1 Choice of Camera

An important decision is choosing the correct camera for the job. Of course you could go all out and buy an expensive camera with amazing resolution and high fps, but within a smaller price range, there are many different options to consider, which each have different specific benefits. Often, algorithms do not even require such precise images. Some aspects to consider are:

- USB vs Ethernet

- Generally want a fixed focus lens
- Field of view
  - Field of view is usually specified on the *diagonal* of the image, so make sure to check that it meets your needs.
  - See Section 8.1 for measuring the field of view.
- Rolling shutter vs Global shutter
  - See https://en.wikipedia.org/wiki/Rolling_shutter
  - Rolling shutter distortion is bad when moving.
  - Global shutter cameras tend to be expensive.
- Resolution
  - High resolution is not especially worth it.
  - You can't stream the images at that high of a resolution, given the FMS bandwidth limits.
  - Vision processing slows down by the number of pixels (i.e. approx. square of the resolution), so the processing time will increase.
- Latency
  - Latency (or lag) is the delay between something happening in real life, and the image appearing on the screen.
  - High latency makes driving based on the streamed images harder.
  - It is often hard to get the specs for latency.

We chose to use the Logitech c930e this past year for multiple reasons:

- wide field of view: 90° diagonal, or roughly 64° horizontal and 52° vertical
- well supported in the software
- robust construction
- readily available at a reasonable price

- favorable experience in past years

Although this was our final decision after much research, the camera choice is mostly dependant on the individual team's requirements and goals, be it a USB or Ethernet connection, cheap or expensive.

6.2 Camera Placement

One problem we ran into this year was how to mount the camera onto the robot. We made a decision to incorporate two cameras into our design. One camera was placed high on the side of the elevator to have a stationary and larger view of the field. The other camera was placed on the intake (which moved up and down with the elevator) so the driver could see the intake and cube as the cube was transferred around the field and into the Vault, Switches, and Scale.

6.3 Camera Mounting

In the beginning of a build season, it's crucial to decide on placement of any cameras before finalizing the layout of other subsystems. If a camera is placed awkwardly on a robot, it will make programming the vision system much harder, as you may need a complicated

conversion to translate positions and angle from centered on the camera to being centered on the robot. Cameras also need to be mounted firmly to the robot in order to minimize vibration when driving.

In terms of the mounting case, due to the curved shape of the Logitech c930e, we thought it would be best to take off the case of the camera and mount the camera circuit board onto a 3d printed piece, which is then mounted on the robot. This seemed to be a good idea at first, but after a few tests, the camera connections became unpredictable and dropped out because of the exposed wiring and loose connectors. In the end we were forced to buy two new cameras. We disassembled the outer cases and removed the "tripod" before putting the outer case back on (this can be done without disturbing the wiring).

We created two different 3D printed mounts to hold the cameras. The mount for the intake camera held the camera from the back and sides, with a hole out the back for the cable. The other mount, for on the side of the elevator, held the camera from the *front* and sides, with a large whole for the lens. This mount was also printed to hold the camera at an angle, so that the camera would be pointing toward the centerline of the robot. The up-down tilt of the camera could be adjusted by loosening the mounting screws and adjusting the tilt to match the driver's preference.

Contact us if you would like the plans for the mounts we used.

6.4 Camera Cabling Challenges

An unexpected problem we ran into during competition was that our intake camera was dropping frames and sometime not working at all. We guessed that this was probably due to the long USB cable, which was needed to accommodate the large motion of the elevator. At the time, the full cable run was approximately 23 feet long including 5 connections, partly due to a miscommunication during assembly. The specified limit for USB2 is 15 ft, and USB3 is actually specified to be shorter.

After our first competition, we rewired the intake camera using a 15 foot *Active* USB extension cable (for a total length of about 20 ft) and reduced the number of connections to 2. The Active cable includes a built-in signal amplifier and we did not experience any more dropouts.

7 Programming Environment

7.1 Language Choice: Python vs. C++ vs. Java



For vision processing (at least within FRC), Python is essentially as fast as C++, because all the heavy lifting for the processing is actually done in the OpenCV C++ code. The Python

is only a glue layer (for the most part) and the image frames are represented as "numpy" arrays, which are efficiently wrapped C arrays. The same is true of Java, but there is a suspicion (untested) that the JNI wrappers incur a tiny bit more overhead.

To test the speed of Python versus C++, we rewrote one of our Python routines for the coprocessor testing in C++, including a threaded wrapper for reading from the camera. The processing loop timing ran flat out on a laptop. Note that these processing speeds will be faster than the camera frame rate (15 fps) because the processing will just re-process the same image until a new one comes in.

Results (fps on a laptop with Intel processor):

- Python: 56.70, 57.30, 58.11 avg = 57.37
- C++: 59.45, 56.58, 57.98 avg = 58.00

Essentially, they are the same speed, so we were comfortable using Python over C++ or Java, due to its ease of use for team beginners and its vast documentation on OpenCV.

7.2 Vision Processing Framework

While you can write your own complete framework for image processing, there are existing frameworks which already handle most of the hard tasks, such as reading the camera, streaming the images over HTTP, etc. Two such common frameworks are cscore from WPILib (https://robotpy.readthedocs.io/projects/cscore/en/stable/) and mjpeg-streamer (https://github.com/jacksonliam/mjpg-streamer). We chose to use cscore, since we were already relatively familiar with WPILib, and it has good support for Python.

8 Code Breakdown

The basis of each year's vision code is the vision server class. This class handles the actual retrieving of images from the camera, processing the images through a processing routine located in a separate class, and sending the output information directly to the roboRIO.

In the main routine of the server, the program first checks that all necessary variables are present to calculate the output information such as the presence of the camera frame, since, if any individual part is absent, the whole program could later crash, causing catastrophe during a match. Once the computer verifies no errors have occurred, it calls the main processing routine and calculates all necessary information to send to the roboRIO.

For each target type, we write a specific processing class, containing the actual code to process an input image and output any target information. For example for 2018, we had separate classes to find the retroreflective switch target and Power Cubes. The general process steps are:

- Retrieve image from camera
- Convert image from BGR to HSV with cv2.cvtColor(). This makes object discrimination easier by color (comparing hue, saturation, and value rather than the different amounts of blue, green, and red on the image).

- Threshold the image for necessary colors (cv2.inRange())
- Find possible contours
- Loop through the contours, filter out unwanted objects, and find the one(s) that best match the target
- Solve to find required information for final output (i.e. angle(s) and distance to target)

For specific details of the LigerBots processing, you can find all our code for the past years on our Github: https://github.com/ligerbots/VisionServer and other repositories there.

8.1 Camera Calibration

In order to receive accurate values from vision algorithms, the camera must be calibrated to know its real field of view and to account for distortion. To do so, we created a utility ("utils/camera_calibration.py" in our code repository) to measure the camera's distortion coefficients. We collected a set of images of a 9 by 6 black and white "chessboard" and then analyzed them using cv2.findChessboardCorners() and cv2.calibrateCamera(). The saved calibration file includes numbers related to the field of view and effective center of the image, plus 5 parameters related to non-linear image distortions. These constants are used in routines like cv2.undistortPoints() and cv2.solvePnP() to accurately convert image coordinates to real world coordinates. For references, see:

- https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html
- https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html

For the calibration to be accurate, the analysis must be provided with sufficiently clear images. Make sure not to bend the chessboard when taking the images because the output coefficients will then not be reliable. As the program is being run, filter through the images to make sure the algorithm finds the correct points on the chessboard, and discard any images where the found corners are incomplete or inaccurate.

9 Coordinate Systems and Geometry

Other than streaming images to the Driver's Station for driver assist, the most important role for the vision processing is to find the coordinates of a particular object (target or game element) relative to the robot, so that it can navigate or target a shooter.

When finding a vision target with OpenCV, there are two typical ways to find coordinates. If you only need an angle between the robot "forward" and the target, you can compute it simply using the number of pixels from the center of the image, with appropriate scaling. On the other hand, if you want a full set of distance and angles, you can use the routine cv2.solvePnP(), which requires finding at least 3 well-known points of an object in the captured image, and knowing their actual, physical coordinates.

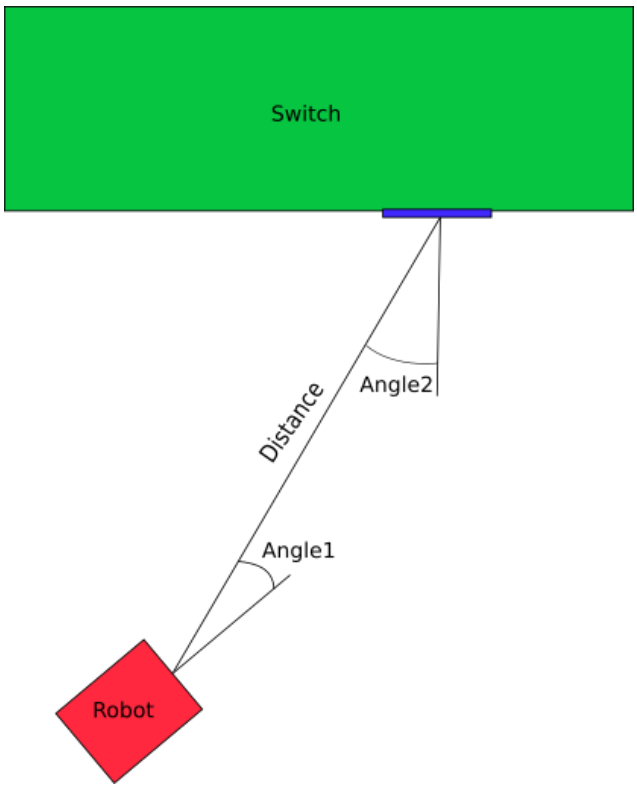


Figure 6: Schematic of the robot and target, showing relevant angles and distance

Figure 6 shows the view looking down, showing a robot and the switch target from 2018. The angles and distance that are desired are in the horizontal plane. What is relevant for robot navigation is:

- distance = the horizontal distance between the robot and the target
- angle1 = the horizontal angle from robot/camera forward and the robot-target line
- angle2 = the horizontal angle between the target perpendicular and the robot-target line

9.1 Coordinate Systems

The vision problem involves two different coordinate systems, camera coordinates and "world" coordinates. The camera coordinates are those attached to the camera, defined by the image. The array of pixels has the zero point at the top left, so the y direction increases going *down*. The z direction is defined by the right-hand rule, so positive z points out of the image forward. So:

- x_{camera} = the horizontal direction within the plane of the camera image, with positive values being to the right
- y_{camera} = the vertical direction, with **positive values going down**

- z_{camera} = the direction out of the image plane, positive facing forward

The world coordinates are those of the physical objects (the target) in the real world. To simplify the math, the origin of the system is usually defined by a particular point on the target (for example, the bottom middle point). x and y follow the usual convention of increasing to the right and up, respectively, so the right-hand rule means that positive z points to the camera.

- x_{world} = the horizontal direction within the plain of the target, positive to the right when facing the target
- y_{world} = the vertical direction, positive going **up**
- z_{world} = the direction perpendicular to the target. The standard right-hand rule applies, so positive is toward the camera.

For simplicity, the usual method is to define the target itself as the origin of the world coordinates, and thus everything is relative to the target's field location, but this is not required.

9.2 Coordinate Transformations

The transformation between robot and world coordinates is defined by 2 matrices: "$rotMat$" is a 3x3 rotation matrix; "$tvec$" is a 3-dimensional translation vector. The transform between the coordinate vectors is defined by:

$$X_{camera} = rotMat * X_{world} + tvec$$

The operations are standard matrix math, so the "$*$" is a matrix multiply. With this equation, you can start with a point in world coordinates and compute the coordinates of the same point in robot coordinates.

So, to get the values for "$distance$" and "$angle1$" (defined above), we need the location of the target in robot coordinates. In world coordinates, the target is at

$$X_{world} = [0\ 0\ 0]^T$$

which results in

$$X_{camera} = tvec$$

From there, computing the horizontal angle and distance is straightforward:

$$distance = \sqrt{tvec\,[0]^2 + tvec\,[2]^2}$$

$$angle1 = atan2\,(tvec\,[0]\,, tvec\,[2])$$

That is, ignore the y dimension (distance in the vertical plane) and compute angle and distance using the Pythagorean Theorem and standard trigonometry.

Getting "$angle2$" seems to be difficult until you realize a trick. What we want is the angle of the robot/target line relative to the target, which is easy to calculate in the world

coordinates. We just need to figure out what that line is, in world coordinates. The way to compute this is to use the inverse transform of above. The steps of the math are easy (just ignore that they are matrices):

$$X_{camera} - tvec = rotMat * X_{world}$$

$$X_{world} = rotMat^{-1} * (X_{camera} - tvec)$$

where $rotMat^{-1}$ means the inverse of $rotMat$. Fortunately, a rotation matrix is special and the inverse is equal to the transpose:

$$rotMat^{-1} = rotMat^{T}$$

so

$$X_{world} = rotMat^{T} * (X_{camera} - tvec)$$

(Note that the transpose of a matrix is trivial, just flip the rows and columns.)

So, to calculate "$angle2$", we start with $X_{camera} = [0\ 0\ 0]^{T}$ and get:

$$X_{world} = -rotMat^{T} * tvec$$

and $angle2$ is then:

$$angle2 = atan2\left(X_{world}\left[0\right], X_{world}\left[2\right]\right)$$

A word of caution is called for here. The above math describes two coordinate systems which are rotated by 180° around the x axis from each other; that is, increasing y values go up in one system, but down in the other, and z is similarly flipped. We picked this so that the y world coordinates increase in the standard direction (positive up). However, this transform is actually established by the arguments to cv2.solvePnP() (see below). solvePnP() takes matched lists of camera and world coordinates; in order to properly establish the above coordinate systems, when camera y increases numerically (i.e. goes down in the image), world y should *decrease*. An easy mistake to make is to order the arguments such that both sets of y coordinates increase at the same time. If this is done, the resulting world coordinates will have x the same, but y and z will be the negative of the above.

9.3 OpenCV Python Code

You can use the OpenCV routine solvePnP() to find the full 6-dimensional location values of the target with respect to the camera. You then need to transform the results into the angles and distance which are more useful for robot navigation/targeting.

solvePnP() takes two lists of 3 or more points. One list is the (2D) coordinates in the image system, the other list is the exact matching (3D) coordinates of the target object in the "world system". Typically, these will be the coordinates of the corners of the target in the two systems. The "world" coordinates will have x and y set, but (usually) z = 0.

The return values of solvePnP() are two 3-dimensional vectors, typically called *tvec* and *rvec*. "*tvec*" is the translation vector and is exactly the same as "*tvec*" in our discussion on coordinate systems. "*rvec*" is the rotation *vector*; it is a packed representation of the

rotation matrix (above). The OpenCV routine Rodrigues() will unpack *rvec* into a proper 3x3 rotation matrix.

Here is the Python code for our 2018 Switch target processing. It takes the *rvec* and *tvec* directly from solvePnP() and computes the distance and 2 angles.

```
def compute_output_values(self, rvec, tvec):
    '''Compute the necessary output distance and angles'''

    x = tvec[0][0]
    z = tvec[2][0]
    # distance in the horizontal plane between camera and target
    distance = math.sqrt(x**2 + z**2)

    # horizontal angle between camera center line and target
    angle1 = math.atan2(x, z)

    rot, _ = cv2.Rodrigues(rvec)
    rot_inv = rot.transpose()
    pzero_world = numpy.matmul(rot_inv, -tvec)
    angle2 = math.atan2(pzero_world[0][0], pzero_world[2][0])

    return distance, angle1, angle2
```

For our 2018 Power Cube finding code, we chose not use the full 3D point finding method. This would have required finding at least 3 corners of the cube in the image, plus knowing the correct matching dimensions in the real world. This is not simple, since the Power Cube is not actually a cube (one side is 11 in. vs. the others at 13 in.), and the white zipper sometimes breaks up the found region. Instead, we chose to look for the center point of the nearest edge on the floor. This would give us both the horizontal and vertical angles between the camera and the cube. Given that the camera is a known height off the floor, the horizontal distance to the cube can be computed from vertical angle. This distance and the horizontal angle are what is need to drive to pick up the cube.

Below is our Python code which takes the found center point (in pixel coordinates) and computes the angle and distance between the robot and the cube. The one complicated part is the use of the camera calibration matrices to remove any lens/image distortion before using the coordinates to compute angles.

```
def get_cube_values_calib(self, center):
    '''Calculate the angle and distance from the camera to
    the center point of the cube. This routine uses the cameraMatrix
    from the calibration to convert to normalized coordinates'''

    # use the distortion and camera arrays to correct
    #   the location of the center point
    # got this from
```

```
# https://stackoverflow.com/questions/8499984/
# how-to-undistort-points-in-camera-shot-coordinates-and-obtain-corresponding-undi

center_np = numpy.array([[[float(self.center[0]), float(self.center[1])]]])
out_pt = cv2.undistortPoints(center_np, self.cameraMatrix, self.distortionMatrix,
                             P=self.cameraMatrix)
undist_center = out_pt[0, 0]

x_prime = (undist_center[0] - self.cameraMatrix[0, 2]) / self.cameraMatrix[0, 0]
y_prime = -(undist_center[1] - self.cameraMatrix[1, 2]) / self.cameraMatrix[1, 1]

# now have all pieces to convert to horizontal angle:
ax = math.atan2(x_prime, 1.0)

# corrected expression.
# As horizontal angle gets larger, real vertical angle gets a little smaller
ay = math.atan2(y_prime * math.cos(ax), 1.0)

# now use the x and y angles to calculate the distance to the target:
d = (self.target_height - self.camera_height) / math.tan(self.tilt_angle + ay)

return ax, d    # return horizontal angle and distance
```

To find out more details on the program from 2018 or from previous years, feel free to visit the Ligerbots Github under the subfolder VisionServer at https://github.com/ligerbots/VisionServer.

10 Additional Problems We Encountered

In the first couple of matches in the 2018 season at the NE district North Shore event, the camera stream did not come through to the driver computer. At the time, we were using Shuffleboard to stream and choose different autonomous commands. It turned out that in that version of Shuffleboard, camera streams would not automatically refresh if the stream was disconnected for some reason. You had to actually close out the widget and reopen a new stream to reset it. Due to this, we chose to use a separate custom dashboard to choose the auto commands and use a direct stream to the Google Chrome browser. This system proved to be increasingly consistent and worked for us throughout the rest of the season and all the way to the World Championship in Detroit. There have been Shuffleboard updates since our original testing which implement automatic reconnection, however we have not yet tried the latest version to see it works satisfactorily.

Furthermore, we experienced problems with camera lag causing difficulties for the driver during matches. The lag time was then testing on the ODROID with a USB camera and was tracked at roughly 110ms at 30fps. If you add in network transfer plus displaying the image in the browser, the lag is roughly 150ms. At 15 fps, it is approximately 190ms, and at 10fps, roughly 250ms. So, we switched from 640x480 at 10fps to 320x240 at 15fps. The image

at DS was considered acceptable (after adjusting compression) and latency was noticeably better. The driver afterwards had a much easier time seeing the field and navigating around.

Finally, very occasionally, a camera would go missing (the ODROID could not locate it). Rebooting the ODROID would solve this, but that is not always practical during a competition. We have not yet solved this mysterious problem, but are currently working to find out why this is occuring.

You will most definitely encounter your own problems when putting together a working system, so expect the worst and try your best to continue pushing through them.

11 Appendix

11.1 Compiling OpenCV for the ODROID

The easiest method to compile OpenCV (or any package) for a coprocessor is to compile it on the machine itself. While cross-compiling can work (e.g. compiling for ARM on a laptop), setting it up can be tricky.

- OpenCV takes a lot of space to compile ($\sim$ 4GB), so consider doing it on a USB3 stick
- Use version 3.3.0 or later https://github.com/opencv/opencv/archive/3.3.0.tar.gz
- We included the opencv_contrib, but this does not seem to necessary
- You will probably need various "dev" packages for compiling: python3-dev, cmake, etc.
- Create a directory, unpack each inside that directory
- Cd into opencv-3.3.0 and create a build directory "odroid"
- Cd into "odroid" and then do:
  - cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local -D INSTALL_PYTHON_EXAMPLES=ON -D OPENCV_EXTRA_MODULES_PATH=$\sim$ /opencv_contrib-3.3.0/modules -D BUILD_EXAMPLES=ON ..
- Check the enabled features. Make sure Python3 is available.
- When done, build with "make -j 2". Build takes a long time (hours).
- Do "sudo make install" to install for all users.

11.2 Camera Tilt

While it has not been tested yet, accounting for a tilt in the camera should be straightforward. Assuming the camera is tilted down by an angle of θ_{tilt}, ie. around a horizontal axis, you can define the rotation matrix for that tilt as

$$R_{tilt} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_{tilt} & -\sin\theta_{tilt} \\ 0 & \sin\theta_{tilt} & \cos\theta_{tilt} \end{bmatrix}$$

With that definition, the transform between world coordinates and robot coordinates becomes

$$X_{robot} = R_{tilt}\left(rotMat * X_{world} + tvec\right)$$

and the inverse transform is

$$X_{world} = rotMat^T\left(R_{tilt}^T * X_{robot} - tvec\right)$$

(Note that the angle of θ_{tilt} is possibly negative for a downward tilt; needs to be checked experimentally.)

Working through the math, we get the following. The position of the target in robot coordinates are:

$$X_{robot} = R_{tilt} * tvec$$
$$X_{robot}[0] = tvec[0]$$
$$X_{robot}[2] = \sin\theta_{tilt} * tvec[1] + \cos\theta_{tilt} * tvec[2]$$

and then "distance" and "angle1" are the similar to above (substitute these "x" and "z" values).

The position of the camera in world coordinates is the same as before with no tilt:

$$X_{world} = -rotMat^T * tvec$$