Vision Processing

Last Updated: 08-16-2019

Table of Contents

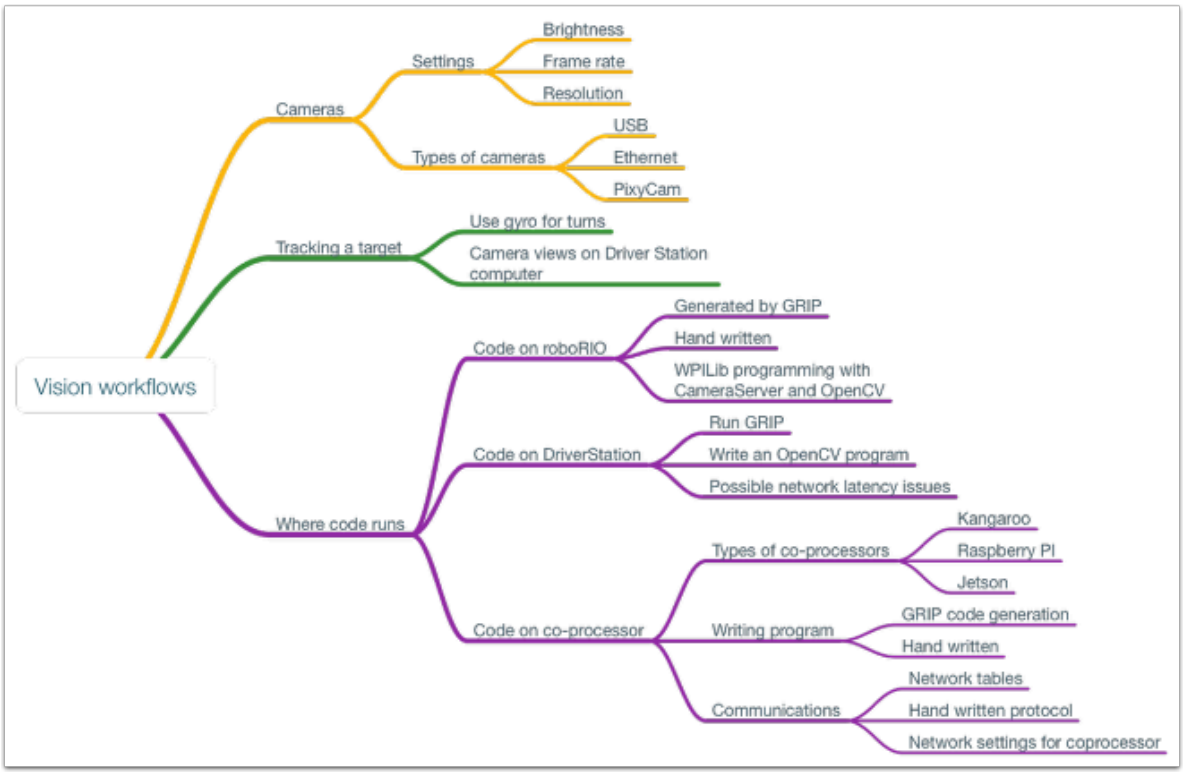
Vision Processing

Resources for vision programming

Strategies for vision programming

Using computer vision is a great way of making your robot be responsive to the elements on the field and make it much more autonomous. Often in FRC games there are bonus points for autonomously shooting balls or other game pieces into goals or navigating to locations on the field. Computer vision is a great way of solving many of these problems. And if you have autonomous code that can do the challenge, then it can be used during the teleop period as well to help the human drivers.

There are many options for choosing the components for vision processing and where the vision program should run. WPILib and associated tools support a number of options and give teams a lot of flexibility to decide what to do. This article will attempt to give you some insight into many of the choices and tradeoffs that are available.

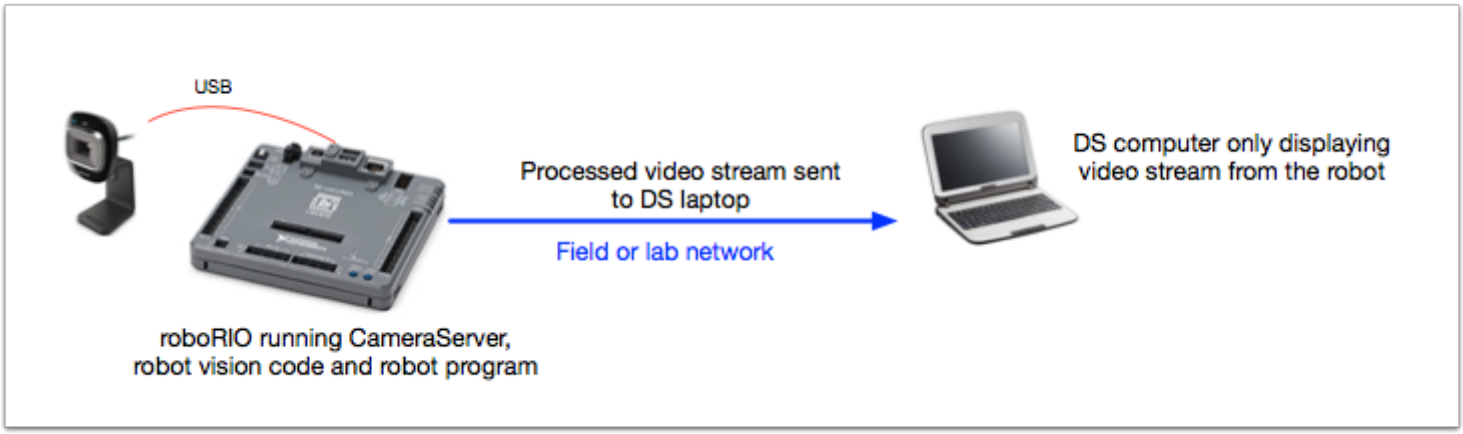


OpenCV computer vision library

OpenCV is an open source computer vision library that is widely used throughout academia and industry. It has support from hardware manufactures providing GPU accelerated processing, it has

bindings for a number of languages including C++, Java, and Python. It is also well documented with many web sites, books, videos, and training courses so there are lots of resources available to help learn how to use it. For these reasons and more we have adopted OpenCV for use with the C++ and Java versions of WPILib. Starting in 2017 **we have prebuilt OpenCV libraries included with WPILib**, support in the library for capturing, processing and viewing video, and tools to help you create your vision algorithms. For more information about OpenCV see [http://opencv.org.](http://opencv.org)

Vision code on roboRIO

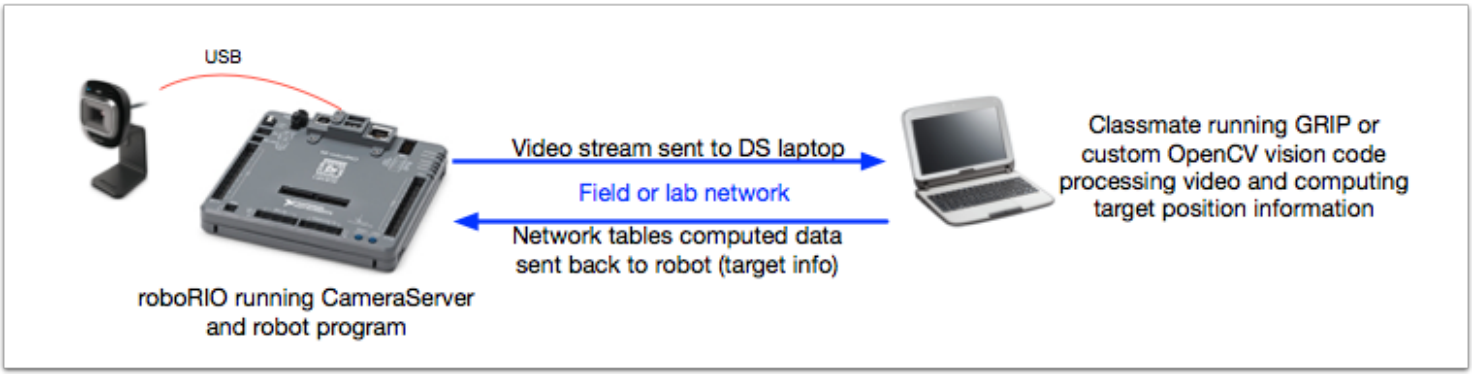


The programming is fairly straightforward since the vision program is just part of the overall robot program. Program can be written by hand or generated by GRIP in either C++ or Java. The disadvantage is that having vision code running on the same processor as the robot program can have performance issues. This is something you will have to evaluate depending on the requirements for your robot and vision program.

The vision code simply produces results that the robot code uses. Be careful about synchronization issues when writing robot code that is getting values from a vision thread. The GRIP generated code and the VisionRunner class in WPILib will make this easier.

The video stream can be sent to the SmartDashboard by using the camera server interface so operators can see what the camera sees. In addition, you can add annotations to the images using OpenCV commands so targets or other interesting objects can be identified in the dashboard view.

Vision code on DS computer

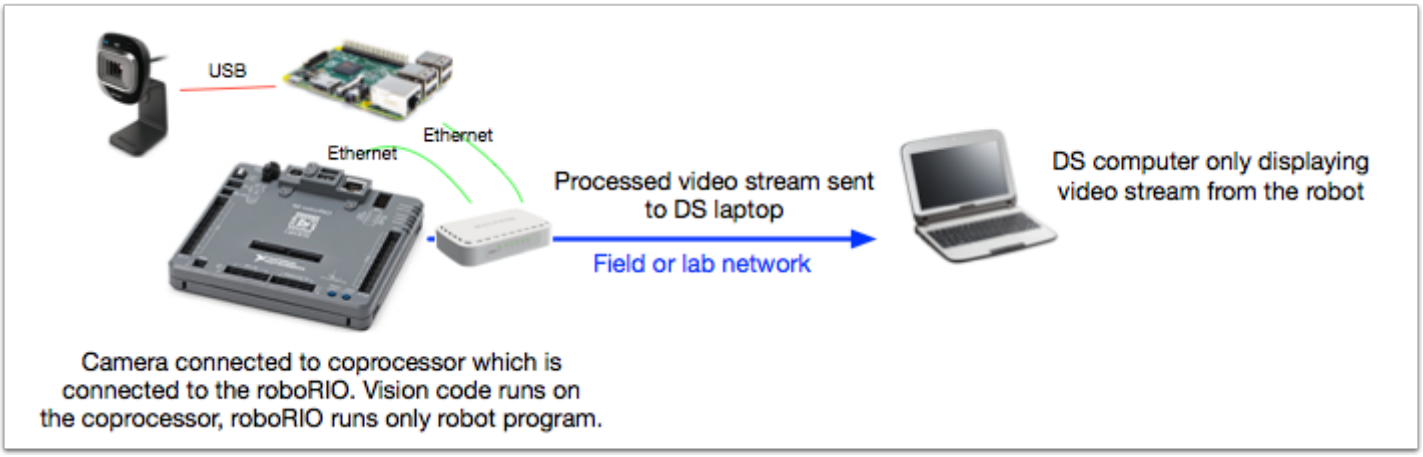


The video is streamed back to the Driver Station laptop for processing. Even the Classmate laptops are substantially faster at vision processing than the roboRIO and don't have real-time programs running on them. GRIP can be run on the Driver Station laptop directly with the results sent back to the robot using NetworkTables. Alternatively you can write your own vision program using a language of your choosing. Python makes a good choice since there is a native NetworkTables implementation and the OpenCV bindings are very good.

After the images are processed, the key values such as the target position, distance or anything else you need can be sent back to the robot with NetworkTables. Latency may be an issue since there might be delays in the images getting to the laptop and the results getting back.

The video stream can be displayed on SmartDashboard or in GRIP.

Vision code on a coprocessor



Coprocessors such as the Raspberry PI or Kangaroo are ideal for supporting vision code. The advantage is that they can run full speed and not interfere with the robot program. In this case, the camera is probably connected to the coprocessor or (in the case of ethernet cameras) an ethernet switch on the robot. The program can be written in any language although Python is a good choice because of it's simple bindings to OpenCV and NetworkTables. Some teams have used high performance vision coprocessors such as the Nvidia Jetson for fastest speed and highest resolution although it might be too complex for inexperienced programmers.

Data can be sent from the vision program on the coprocessor to the robot using NetworkTables or a private protocol over a network or serial connection.

Camera options

There are a number of camera options supported by WPILib. Cameras have a number of parameters than can determine how it operates. For example, frame rate and image resolution effect the quality of the received images but when set too high impact processing time and the bandwidth which may have serious effect on your robot program.

New for 2017 there is a CameraServer class available in C++ and Java that interfaces with cameras connected to the robot. It retrieve frames for local processing through a Source object and send stream to your driver station for viewing or processing there. The video stream is processed by the CameraServer.

Vision Processing

Details on using cameras with WPILib are detailed in the next section.

Read and process video: CameraServer class

Concepts

The cameras typically used in FRC (commodity USB and Ethernet cameras such as the Axis camera) offer relatively limited modes of operation. In general, they provide only a single image output (typically in an RGB compressed format such as JPG) at a single resolution and frame rate. USB cameras are particularly limited as only one application may access the camera at a time.

The 2017 CameraServer supports multiple cameras. It handles details such as automatically reconnecting when a camera is disconnected, and also makes images from the camera available to multiple "clients" (e.g. both your robot code and the dashboard can connect to the camera simultaneously).

Camera names

Each camera in CameraServer must be uniquely named. This is also the name that appears for the camera in the Dashboard. Some variants of the CameraServer startAutomaticCapture() and addAxisCamera() functions will automatically name the camera (e.g. "USB Camera 0" or "Axis Camera"), or you can give the camera a more descriptive name (e.g. "Intake Cam"). The only requirement is that each camera have a unique name.

USB Camera Notes

CPU Usage

The CameraServer is designed to minimize CPU usage by only performing compression and decompression operations when required and automatically disabling streaming when no clients are connected.

To minimize CPU usage, the dashboard resolution should be set to the same resolution as the camera; this allows the CameraServer to not decompress and recompress the image, instead, it can simply forward the JPEG image received from the camera directly to the dashboard. It's important to note that changing the resolution on the dashboard does *not* change the camera resolution; changing the camera resolution may be done either through NetworkTables or in your robot code (by calling setResolution on the camera object).

USB Bandwidth

The roboRio can only transmit and receive so much data at a time over its USB interfaces. Camera images can require a lot of data, and so it is relatively easy to run into this limit. The most common cause of a USB bandwidth error is selecting a non-JPEG video mode or running too high of a resolution, particularly when multiple cameras are connected.

Architecture

The CameraServer for 2017 consists of two layers, the high level WPILib **CameraServer class** and the low level **cscore library**.

CameraServer class

The CameraServer class (part of WPILib) provides a high level interface for adding cameras to your robot code. It also is responsible for publishing information about the cameras and camera servers to NetworkTables so that Driver Station dashboards such as the LabView Dashboard and SmartDashboard can list the cameras and determine where their streams are located. It uses a singleton pattern to maintain a database of all created cameras and servers.

Some key functions in CameraServer are:

- **startAutomaticCapture()**: Add a USB camera (e.g. Microsoft LifeCam) and starts a server for it so it can be viewed from the dashboard.
- **addAxisCamera()**: Add an Axis camera. Even if you aren't processing images from the Axis camera in your robot code, you may want to use this function so that the Axis camera appears in the Dashboard's drop down list of cameras. It also starts a server so the Axis stream can still be viewed when your driver station is connected to the roboRio via USB (useful at competition if you have both the Axis camera and roboRio connected to the two robot radio Ethernet ports).
- **getVideo()**: Get OpenCV access to a camera. This allows you to get images from the camera for image processing on the roboRio (in your robot code).
- **putVideo()**: Start a server that you can feed OpenCV images to. This allows you to pass custom processed and/or annotated images to the dashboard.

cscore Library

The cscore library provides the lower level implementation to:

- Get images from USB and HTTP (e.g. Axis) cameras
- Change camera settings (e.g. contrast and brightness)

- Change camera video modes (pixel format, resolution, and frame rate)
- Act as an web server and serve images as a standard M-JPEG stream
- Convert images to/from OpenCV "Mat" objects for image processing

Sources and Sinks

The basic architecture of the cscore library is similar to that of MJPGStreamer, with functionality split between sources and sinks. There can be multiple sources and multiple sinks created and operating simultaneously.

An object that generates images is a source and an object that accepts/consumes images is a sink. The generate/consume is from the perspective of the library. Thus cameras are sources (they generate images). The MJPEG web server is a sink because it accepts images from within the program (even though it may be forwarding those images on to a web browser or dashboard). Sources may be connected to multiple sinks, but sinks can be connected to one and only one source. When a sink is connected to a source, the cscore library takes care of passing each image from the source to the sink.

- **Sources** obtain individual frames (such as provided by a USB camera) and fire an event when a new frame is available. If no sinks are listening to a particular source, the library may pause or disconnect from a source to save processor and I/O resources. The library autonomously handles camera disconnects/reconnects by simply pausing and resuming firing of events (e.g. a disconnect results in no new frames, not an error).
- **Sinks** listen to a particular source's event, grab the latest image, and forward it to its destination in the appropriate format. Similarly to sources, if a particular sink is inactive (e.g. no client is connected to a configured M-JPEG over HTTP server), the library may disable parts of its processing to save processor resources.

User code (such as that used in a FRC robot program) can act as either a source (providing processed frames as if it were a camera) or as a sink (receiving a frame for processing) via OpenCV source and sink objects. Thus an image processing pipeline that gets images from a camera and serves the processed images out looks like the below graph:

UsbCamera (VideoSource) --> (cscore internal) --> CvSink (VideoSink)

--> (your OpenCV processing code) --> CvSource (VideoSource)

--> (cscore internal) --> MjpegServer (VideoSink)

Because sources can have multiple sinks connected, the pipeline may branch. For example, the original camera image can also be served by connecting the UsbCamera source to a second MjpegServer sink in addition to the CvSink, resulting in the below graph:

UsbCamera --> CvSink --> (your code) --> CvSource --> MjpegServer [2]

\-> MjpegServer [1]

When a new image is captured by the camera, both the CvSink and the MjpegServer [1] receive it.

The above graph is what the following CameraServer snippet creates:

```
// Creates UsbCamera and MjpegServer [1] and connects them
CameraServer.getInstance().startAutomaticCapture()
// Creates the CvSink and connects it to the UsbCamera CvSink cvSink =
CameraServer.getInstance().getVideo()
// Creates the CvSource and MjpegServer [2] and connects them CvSource
outputStream = CameraServer.getInstance().putVideo("Blur", 640, 480);
```

The CameraServer implementation effectively does the following at the cscore level (for explanation purposes). CameraServer takes care of many of the details such as creating unique names for all cscore objects and automatically selecting port numbers. CameraServer also keeps a singleton registry of created objects so they aren't destroyed if they go out of scope.

```
UsbCamera usbCamera = new UsbCamera("USB Camera 0", 0);
MjpegServer mjpegServer1 = new MjpegServer("serve_USB Camera 0", 1181);
mjpegServer1.setSource(usbCamera); CvSink cvSink = new CvSink("opencv_USB
Camera 0");
cvSink.setSource(usbCamera);
CvSource outputStream = new CvSource("Blur", PixelFormat.kMJPEG, 640, 480,
30);
MjpegServer mjpegServer2 = new MjpegServer("serve_Blur", 1182);
mjpegServer2.setSource(outputStream);
```

Reference Counting

All cscore objects are internally reference counted. Connecting a sink to a source increments the source's reference count, so it's only strictly necessary to keep the sink in scope. The CameraServer class keeps a registry of all objects created with CameraServer functions, so sources and sinks created in that way effectively never go out of scope (unless explicitly removed).

2017 Vision Examples

LabVIEW

The 2017 LabVIEW Vision Example is included with the other LabVIEW examples. From the Splash screen, click Support->Find FRC Examples or from any other LabVIEW window, click Help->Find Examples and locate the Vision folder to find the 2017 Vision Example. The example images are bundled with the example.

C++\Java

A more complete example is coming soon.

For now, we have provided a GRIP project and the description below, as well as the example images, bundled into a ZIP that can be found on TeamForge.

Details about integrating GRIP generated code in your robot program is here: Using Generated Code in a Robot Program

The code generated by the included GRIP project will find OpenCV contours for green particles in images like the ones included in the Vision Images folder of this ZIP. From there you may wish to further process these contours to assess if they are the target. To do this:

1. Use the boundingRect method to draw bounding rectangles around the contours
2. The LabVIEW example code calculates 5 separate ratios for the target. Each of these ratios should nominally equal 1.0. To do this, it sorts the contours by size, then starting with the largest, calculates these values for every possible pair of contours that may be the target, and stops if it finds a target or returns the best pair it found.

In the formulas below, 1 followed by a letter refers to a coordinate of the bounding rect of the first contour, which is the larger of the two (e.g. 1L = 1st contour left edge) and 2 with a letter is the 2nd contour. (H=Height, L = Left, T = Top, B = Bottom, W = Width)

- Group Height = 1H /((2B - 1T) *.4) Top height should be 40% of total height (4in / 10 in.)
- dTop = (2T - 1T) /((2B - 1T) * .6) Top of bottom stripe to top of top stripe should be 60% of total height (6in / 10 in.)
- LEdge = ((1L - 2L) / 1W) + 1 The distance between the left edge of contour 1 and the left edge of contour 2 should be small relative to the width of the 1st contour (then we add 1 to make the ratio centered on 1

- Width ratio = 1W / 2W The widths of both contours should be about the same
- Height ratio = 1H / (2H * 2) The larger stripe should be twice as tall as the smaller one

Each of these ratios is then turned into a 0-100 score by calculating: 100 - (100 * abs (1 - Val))

3. To determine distance, measure pixels from top of top bounding box to bottom of bottom bounding box

distance = Target height in ft. (10/12) * YRes / (2*PixelHeight*tan(viewAngle of camera))

The LabVIEW example uses height as the edges of the round target are the most prone to noise in detection (as the angle points further from the camera the color looks less green). The downside of this is that the pixel height of the target in the image is affected by perspective distortion from the angle of the camera. Possible fixes include:

1. Try using width instead
2. Empirically measure height at various distances and create a lookup table or regression function
3. Mount the camera to a servo, center the target vertically in the image and use servo angle for distance calculation (you'll have to work out the proper trig yourself or find a math teacher to help!)
4. Correct for the perspective distortion using OpenCV. To do this you will need to calibrate your camera with OpenCV as described here: http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html This will result in a distortion matrix and camera matrix. You will take these two matrices and use them with the undistortPoints function to map the points you want to measure for the distance calculation to the "correct" coordinates (this is much less CPU intensive than undistorting the whole image)

Background

Target Info and Retroreflection

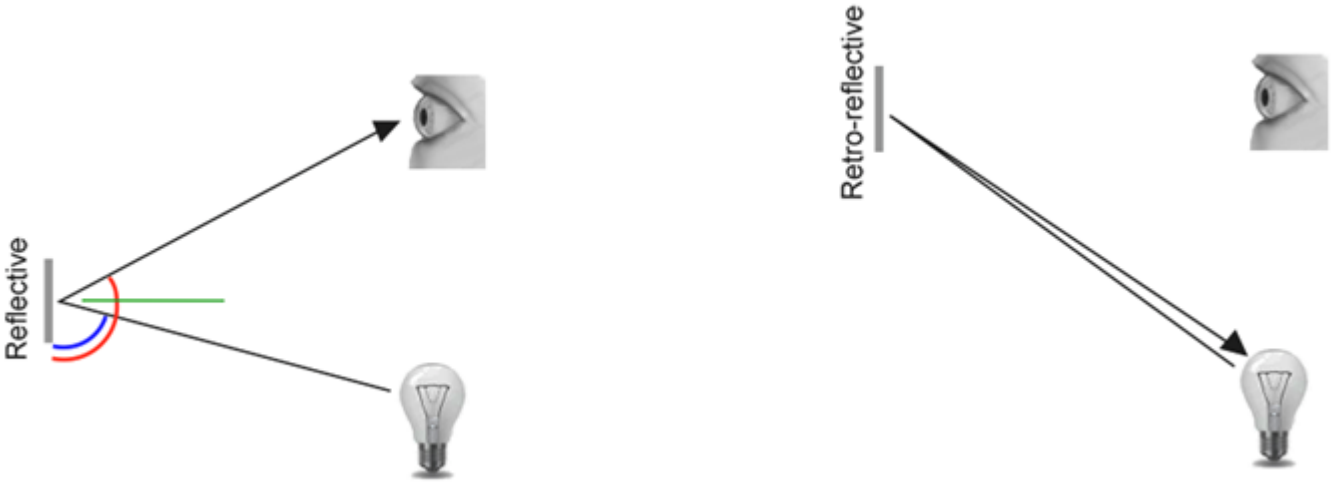
This document describes the Vision Targets from the 2016 FRC game and the visual properties of the material making up the targets. Note that for official dimensions and drawings of all field components, please see the Official Field Drawings

Targets

Targets

Each Target consists of a 1' 8" wide, 1' tall U-shape made of 2" wide retroreflective material (3M 8830 Silver Marking Film). The targets are located immediately adjacent to the bottom of each high goal. When properly lit, the retroreflective tape produces a bright and/or color-saturated marker.

Retroreflectivity vs. Reflectivity



Highly reflective materials are generally mirrored so that light “bounces off” at a supplementary angle. As shown above-left, the blue and red angles sum to 180 degrees. An equivalent explanation is that the light reflects about the surface normal the green line drawn perpendicular

to the surface. Notice that a light pointed at the surface will return to the light source only if the blue angle is ~90 degrees.

Retro-reflective materials are not mirrored, but it will typically have either shiny facets across the surface, or it will have a pearl-like appearance. Not all faceted or pearl-like materials are retro-reflective, however. Retro-reflective materials return the majority of light back to the light source, and they do this for a wide range of angles between the surface and the light source, not just the 90 degree case. Retro-reflective materials accomplish this using small prisms, such as found on a bicycle or roadside reflector, or by using small spheres with the appropriate index of refraction that accomplish multiple internal reflections. In nature, the eyes of some animals, including house cats, also exhibit the retro-reflective effect typically referred to as night-shine. The Wikipedia articles on retro-reflection go into more detail on how retro-reflection is accomplished.

Examples of Retroreflection

This material should be relatively familiar as it is often used to enhance nighttime visibility of road signs, bicycles, and pedestrians.

Initially, retro-reflection may not seem like a useful property for nighttime safety, but when the light and eye are near one another, as shown below, the reflected light returns to the eye, and the material shines brightly even at large distances. Due to the small angle between the driver's eyes and vehicle headlights, retro-reflective materials can greatly increase visibility of distant objects during nighttime driving.

Demonstration

To further explore retro-reflective material properties:

1. Place a piece of the material on a wall or vertical surface
2. Stand 10-20 feet away, and shine a small flashlight at the material.
3. Start with the light held at your belly button, and raise it slowly until it is between your eyes. As the light nears your eyes, the intensity of the returned light will increase rapidly.
4. Alter the angle by moving to other locations in the room and repeating. The bright reflection should occur over a wide range of viewing angles, but the angle from light source to eye is key and must be quite small.

Experiment with different light sources. The material is hundreds of times more reflective than white paint; so dim light sources will work fine. For example, a red bicycle safety light will demonstrate that the color of the light source determines the color of the reflected light. If possible, position several team members at different locations, each with their own light source. This will show that the effects are largely independent, and the material can simultaneously appear different colors to various team members. This also demonstrates that the material is largely immune to environmental lighting. The light returning to the viewer is almost entirely determined by a light source they control or one directly behind them. Using the flashlight, identify other retro-reflective articles already in your environment … on clothing, backpacks, shoes, etc.

Lighting

We have seen that the retro-reflective tape will not shine unless a light source is directed at it, and the light source must pass very near the camera lens or the observer's eyes. While there are a number of ways to accomplish this, a very useful type of light source to investigate is the ring flash, or ring light, shown above. It places the light source directly on or around the camera lens and provides very even lighting. Because of their bright output and small size, LEDs are particularly useful for constructing this type of device.

As shown above, inexpensive circular arrangements of LEDs are available in a variety of colors and sizes and are easy to attach to the Axis cameras. While not designed for diffuse even lighting, they work quite well for causing retro-reflective tape to shine. A small green LED ring is available through FIRST Choice. Other similar LED rings are available from the supplier, SuperBrightLEDs.com

Sample Images

Sample Images are located with the code examples for each language (packaged with LabVIEW, in a separate ZIP in the same location as the C++\Java samples).

Identifying and Processing the Targets

Once an image is captured, the next step is to identify Vision Target(s) in the image. This document will walk through one approach to identifying the 2016 targets. Note that the images used in this section were taken with the camera intentionally set to underexpose the images, producing very dark images with the exception of the lit targets, see the section on Camera Settings for details.

Additional Options

This document walks through the approach used by the example code provided in LabVIEW (for PC or roboRIO), C++ and Java. In addition to these options teams should be aware of the following alternatives that allow for vision processing on the Driver Station PC or an on-board PC:

1. RoboRealm
2. SmartDashboard Camera Extension (programmed in Java, works with any robot language)
3. GRIP

Original Image

The image shown below is the starting image for the example described here. The image was taken using the green ring light available in FIRST Choice. combined with an additional ring light of a different size. Additional sample images are provided with the vision code examples.

Original Image

What is HSL/HSV?

The Hue or tone of the color is commonly seen on the artist's color wheel and contains the colors of the rainbow Red, Orange, Yellow, Green, Blue, Indigo, and Violet. The hue is specified using a radial angle on the wheel, but in imaging the circle typically contains only 256 units, starting with red at zero, cycling through the rainbow, and wrapping back to red at the upper end. Saturation of

a color specifies amount of color, or the ratio of the hue color to a shade of gray. Higher ratio means more colorful, less gray. Zero saturation has no hue and is completely gray. Luminance or Value indicates the shade of gray that the hue is blended with. Black is 0 and white is 255.

The example code uses the HSV color space to specify the color of the target. The primary reason is that it readily allows for using the brightness of the targets relative to the rest of the image as a filtering criteria by using the Value (HSV) or Luminance (HSL) component. Another reason to use the HSV color system is that the thresholding operation used in the example runs more efficiently on the roboRIO when done in the HSV color space.

Masking

In this initial step, pixel values are compared to constant color or brightness values to create a binary mask shown below in yellow. This single step eliminates most of the pixels that are not part of a target's retro-reflective tape. Color based masking works well provided the color is relatively saturated, bright, and consistent. Color inequalities are generally more accurate when specified using the HSL (Hue, Saturation, and Luminance) or HSV (Hue, Saturation, and Value) color space than the RGB (Red, Green, and Blue) space. This is especially true when the color range is quite large in one or more dimension.

Notice that in addition to the target, other bright parts of the image (overhead light and tower lighting) are also caught by the masking step.

Teams may find it more computationally efficient, though potentially less robust, to filter based on only a single criteria such as Hue or Value/Luminance.

Masking

Particle Analysis

After the masking operation, a particle report operation is used to examine the area, bounding rectangle, and equivalent rectangle for the particles. These are used to compute several scored terms to help pick the shapes that are most rectangular. Each test described below generates a score (0-100) which is then compared to pre-defined score limits to decide if the particle is a target or not.

Coverage Area

The Area score is calculated by comparing the area of the particle compared to the area of the bounding box drawn around the particle. The area of the retroreflective strips is 80 square inches. The area of the rectangle that contains the target is 240 square inches. This means that the ideal ratio between area and bounding box area is 1/3. Area ratios close to 1/3 will produce a score near 100, as the ratio diverges from 1/3 the score will approach 0.

Aspect Ratio

The aspect ratio score is based on (Particle Width / Particle Height). The width and height of the particle are determined using something called the "equivalent rectangle". The equivalent rectangle is the rectangle with side lengths x and y where 2x+2y equals the particle perimeter and x*y equals the particle area. The equivalent rectangle is used for the aspect ratio calculation as it is less affected by skewing of the rectangle than using the bounding box. When using the bounding box rectangle for aspect ratio, as the rectangle is skewed the height increases and the width decreases.

The target is 20" wide by 12" tall, for a ratio of 1.6. The detected aspect ratio is compared to this ideal ratio. The aspect ratio score is normalized to return 100 when the ratio matches the target ratio and drops linearly as the ratio varies below or above.

Moment

The moment measurement calculates the particles moment of inertia about it's center of mass. This measurement provides a representation of the pixel distribution in the particle. The ideal score for this test is ~0.28. Moment of Inertia

X/Y Profiles

X/Y Profiles

The edge score describes whether the particle matches the appropriate profile in both the X and Y directions. As shown, it is calculated using the row and column averages across the bounding box

extracted from the original image and comparing that to a profile mask. The score ranges from 0 to 100 based on the number of values within the row or column averages that are between the upper and lower limit values.

Measurements

If a particle scores well enough to be considered a target, it makes sense to calculate some real-world measurements such as position and distance. The example code includes these basic measurements, so let's look at the math involved to better understand it.

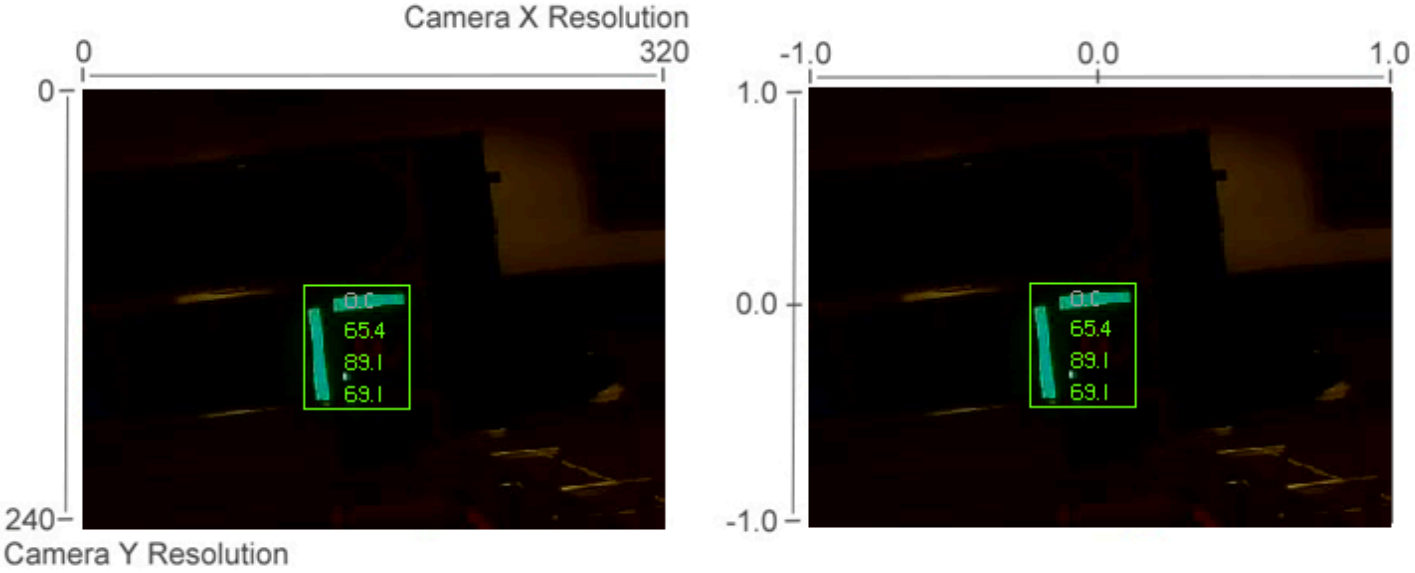
Position

The target position is well described by both the particle and the bounding box, but all coordinates are in pixels with 0,0 being at the top left of the screen and the right and bottom edges determined by the camera resolution. This is a useful system for pixel math, but not nearly as useful for driving a robot; so let's change it to something that may be more useful.

To convert a point from the pixel system to the aiming system, we can use the formula shown below.

The resulting coordinates are close to what you may want, but the Y axis is inverted. This could be corrected by multiplying the point by [1,-1] (Note: this is not done in the sample code). This coordinate system is useful because it has a centered origin and the scale is similar to joystick outputs and RobotDrive inputs.

$$A_{x,y} = (P_{x,y} - \frac{resolution_{x,y}}{2}) / \frac{resolution_{x,y}}{2}$$



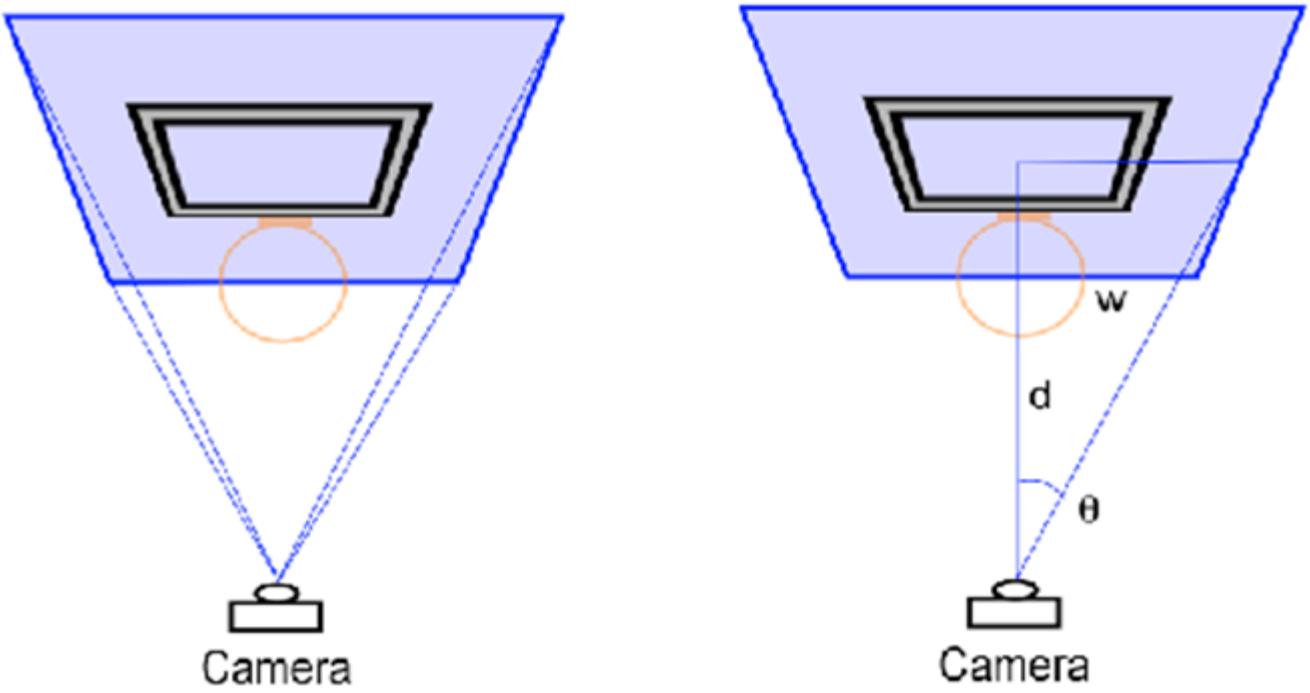
Pixel Coordinate System – $P_{x,y}$ Aiming Coordinate System – $A_{x,y}$

Distance

The target distance is computed with knowledge about the target size and the camera optics. The approach uses information about the camera lens view angle and the width of the camera field of view. Shown below-left, a given camera takes in light within the blue pyramid extending from the focal point of the lens. Unless the lens is modified, the view angle is constant and equal to 2Θ. As shown to the right, the values are related through the trigonometric relationship of ...

tanΘ = w/d

The datasheets for the cameras can be found at the following URLs:Axis 206, AxisM1011, Axis M1013, Lifecam HD3000. These give rough horizontal view angles for the lenses. Remember that this is for entire field of view, and is therefore 2Θ. This year's code uses the vertical field-of-view and it is therefore highly recommend to perform calibration (as described in the next article) to determine the appropriate view angle for your camera (empirically determined values for each camera type are included in the code as a reference).



Distance Continued

The next step is to use the information we have about the target to find the width of the field of view the blue rectangle shown above. This is possible because we know the target rectangle size in both pixels and feet, and we know the FOV rectangle width in pixels. We can use the relationships of …

Tft/Tpixel = FOVft/FOVpixel and FOVft = 2*w = 2*d*tanΘ

to create an equation to solve for d, the distance from the target:

d = Tft*FOVpixel/(2*Tpixel*tanΘ)

Notice that the datasheets give approximate view angle information. When testing, it was found that the calculated distance to the target tended to be a bit short. Using a tape measure to measure the distance and treating the angle as the unknown it was found that view angles of 41.7˚ for the 206, 37.4˚ for the M1011, and 49˚ for the M1013 gave better results. Information on performing your own distance calibration is included in the next article.

Setup

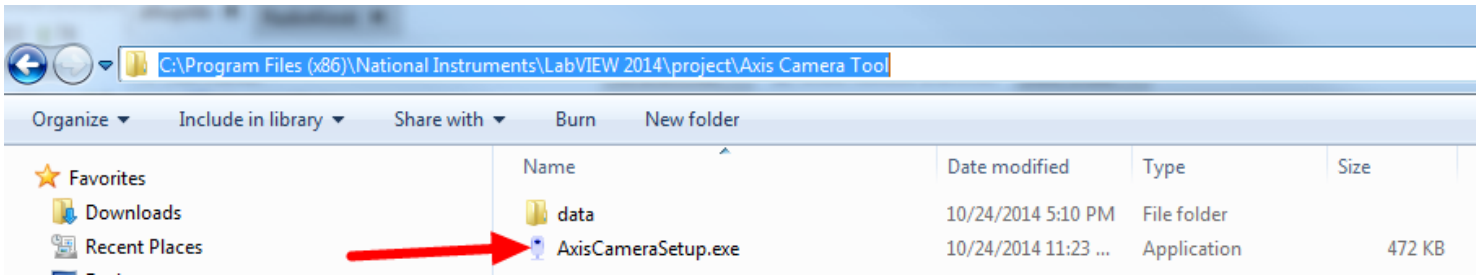
Configuring an Axis Camera

Three different Axis camera models are supported by the FRC software, the Axis 206, Axis M1011 and Axis M1013. This document provides instructions on how to configure one of these cameras for FRC use. To follow the instructions in this document, you must have installed the NI FRC Update Suite and Configured your radio

Connect the camera

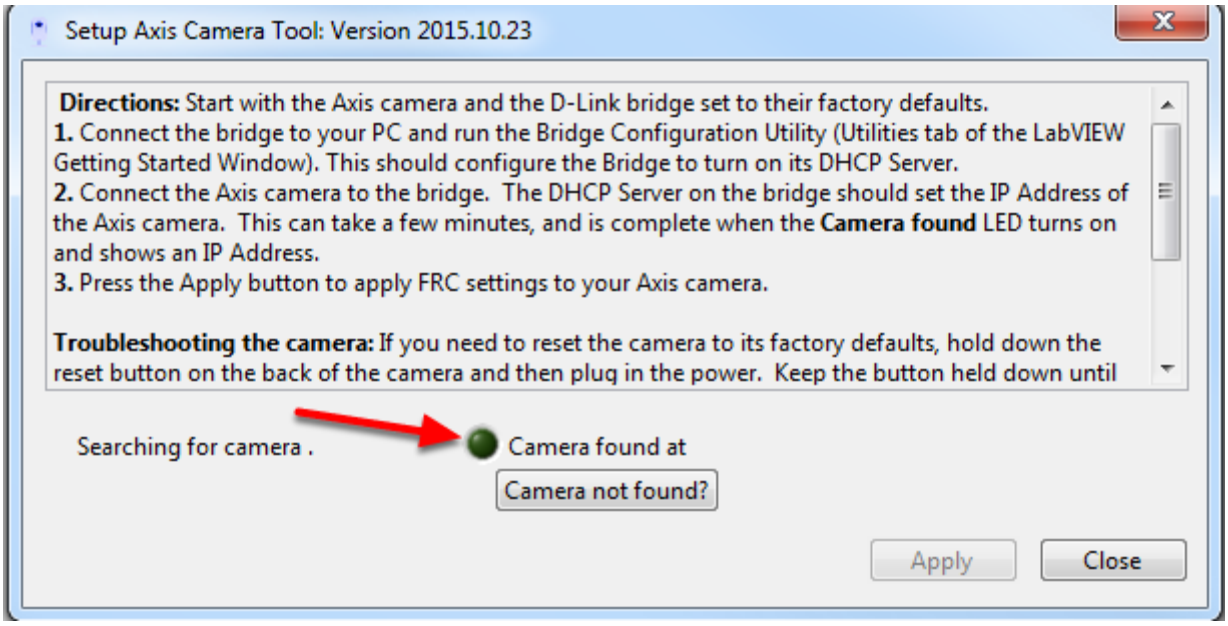
Connect the Axis camera to the DAP-1522 radio using an Ethernet cable. Connect your computer to the radio using an ethernet cable or via a wireless connection.

Axis Camera Setup Tool



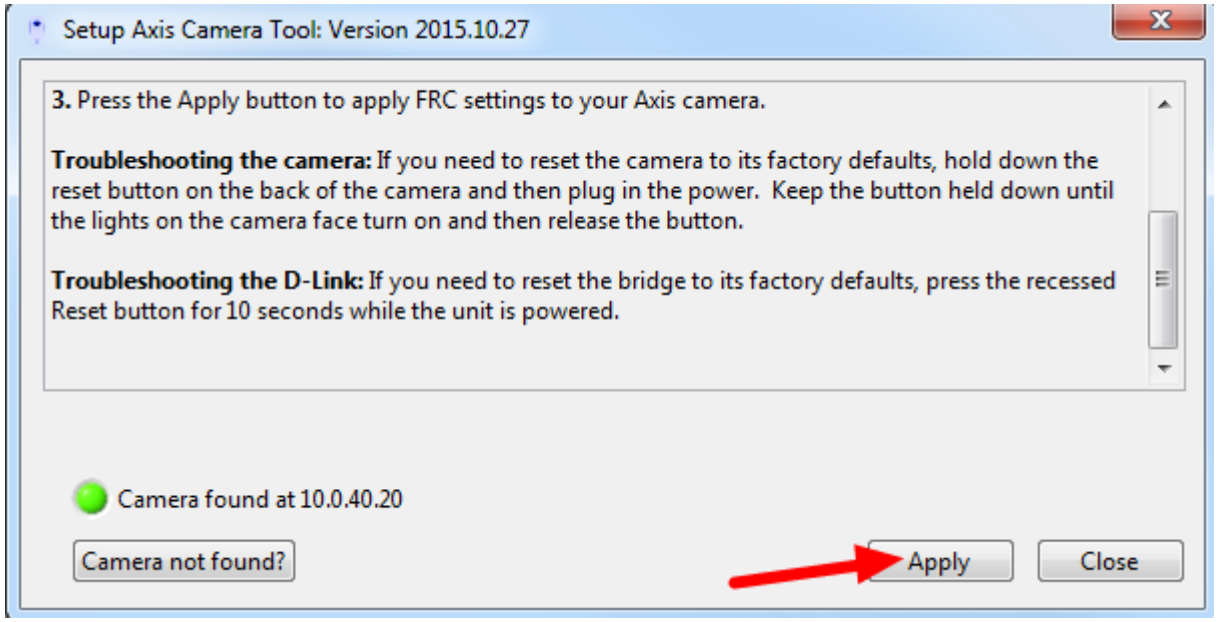
Browse to **C:\Program Files (x86)\National Instruments\LabVIEW 2014\project\Axis Camera Tool** and **double-click** on **AxisCameraSetup.exe** to start the Axis Camera Setup Tool.

Tool Overview



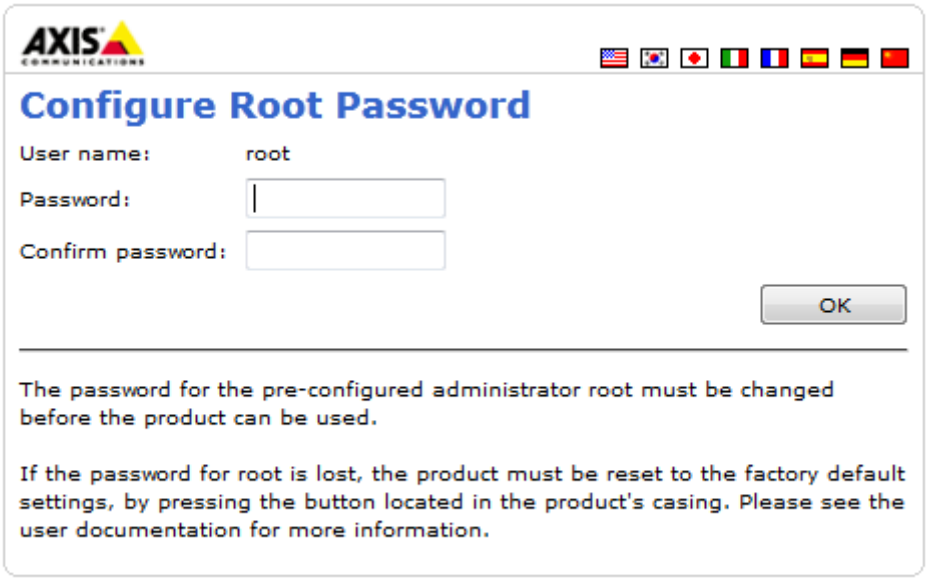
The camera should be automatically detected and the green indicator light should be lit. If it is not, make sure the camera is powered on (the ring on the camera face should be green) and connected to your computer. If the indicator remains off follow the instructions in the tool textbox next to **Troubleshooting the camera** to reset the camera. You can also use the **Camera not found?** button to check the IP address of your computer, one of the addresses listed should be of the form 10.TE.AM.XX where TEAM is your 4 digit team number.

Setup the Camera



To configure the camera, press Apply. This will configure many of the necessary/recommended settings for using the camera for FRC. Currently the tool does not properly configure the DNS name of the camera in many cases.

Camera Webpage

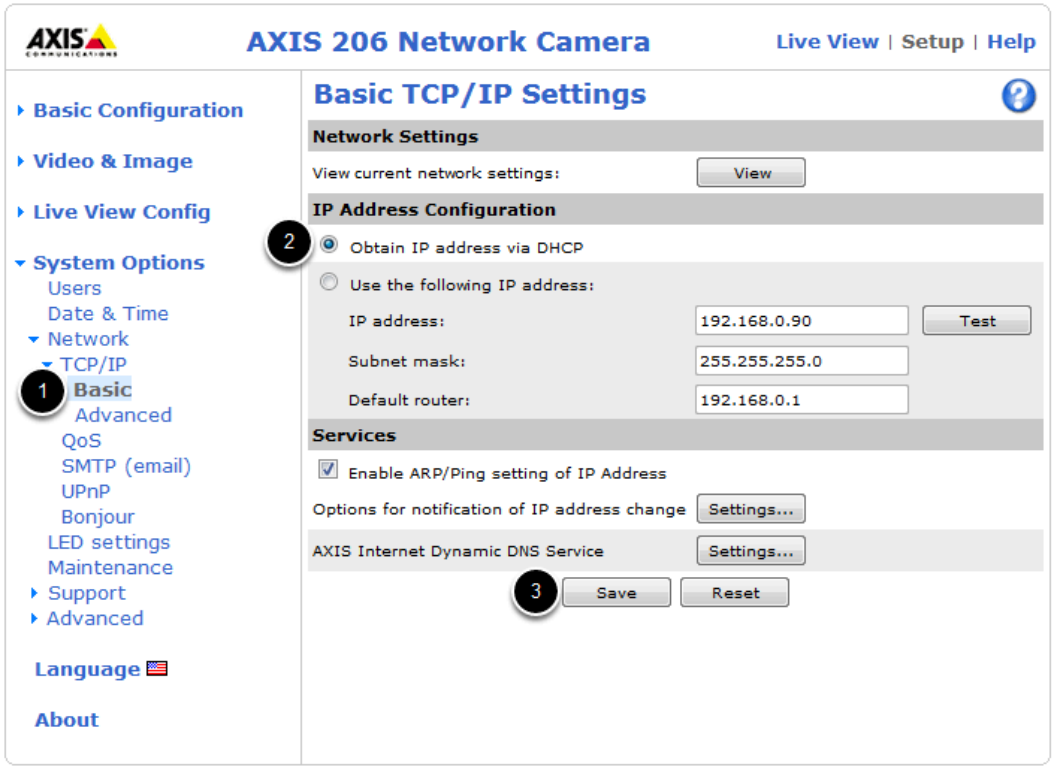


To set the network settings, open a web browser and enter the address shown next to **Camera found at** in the tool (in the example above this is 10.0.40.20) in the address bar and press enter. You should see a Configure Root Password page, set this password to whatever you would like, but **admin** is recommended.

Setup Page

Click **Setup** to go to the setup page.

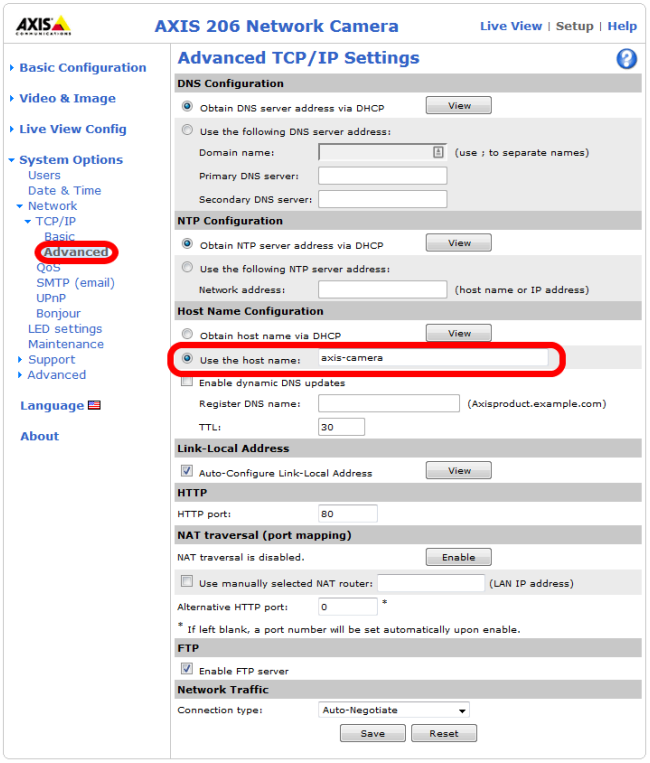
Configure Basic Network Settings



To configure the network settings of the camera, click the arrow to expand the **System Options** pane, then click the arrow to expand **Network**, then expand **TCP/IP** and select **Basic.** Set the camera to obtain an IP address via DHCP by selecting the bubble. Alternately, you may choose to set a static IP in the range 10.TE.AM.3 to 10.TE.AM.19. This is outside the range handed out by the DAP-1522 radio (home use) or FMS system (event use) so you will avoid any IP conflicts.

Click **Save**.

Configure Advanced Network Settings



Next click **Advanced** under **TCP/IP**. Set the **Host Name Configuration** to **"Use the host name:"** and set the value to **"axis-camera"** as shown. If you plan to use multiple cameras on your robot, select a unique host name for each. You will need to modify the dashboard and/or robot code to work with the additional cameras and unique host names.

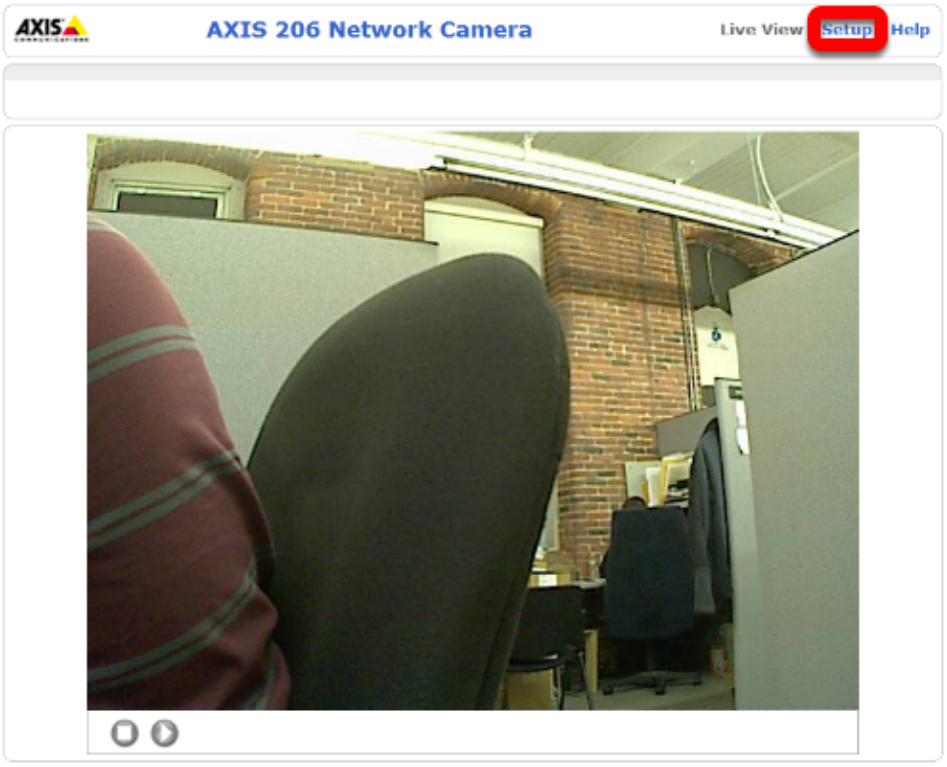
Click **Save.**

Manual Camera Configuration

It is recommended to use the Setup Axis Camera Tool to configure the Axis Camera. If you need to configure the camera manually, connect the camera directly to the computer, configure your computer to have a static IP of 192.168.0.5, then open a web browser and enter **192.168.0.90** in the address bar and press enter. You should see a Configure Root Password page, set this password to whatever you would like, but **admin** is recommended.

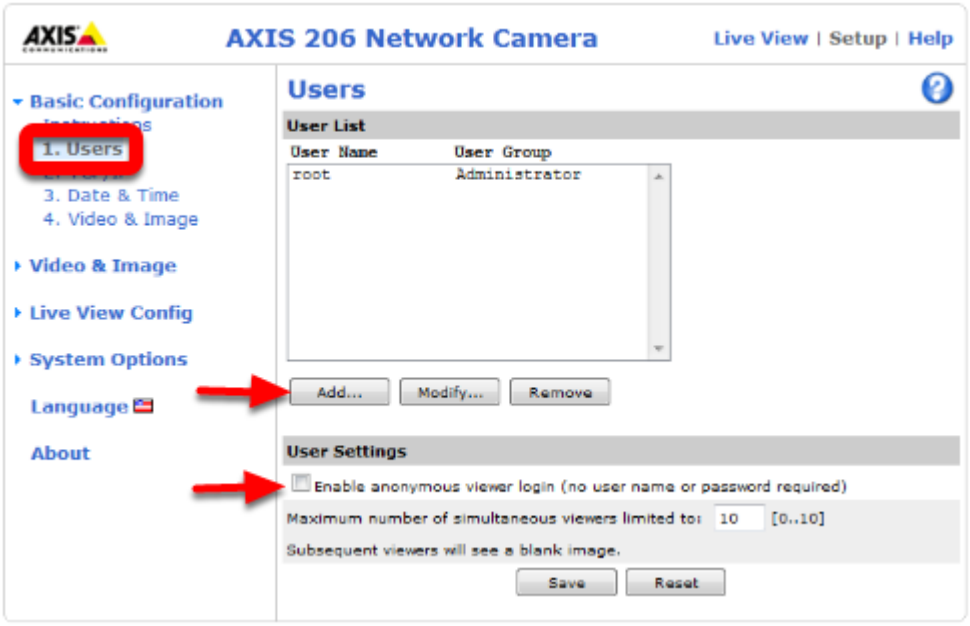
If you do not see the camera webpage come up, you may need to reset the camera to factory defaults. To do this, remove power from the camera, hold the reset button while applying power to the camera and continue holding it until the lights on the camera face turn on, then release the reset button and wait for the lights to turn green. The camera is now reset to factory settings and should be accessible via the 192.168.0.5 address.

Setup Page



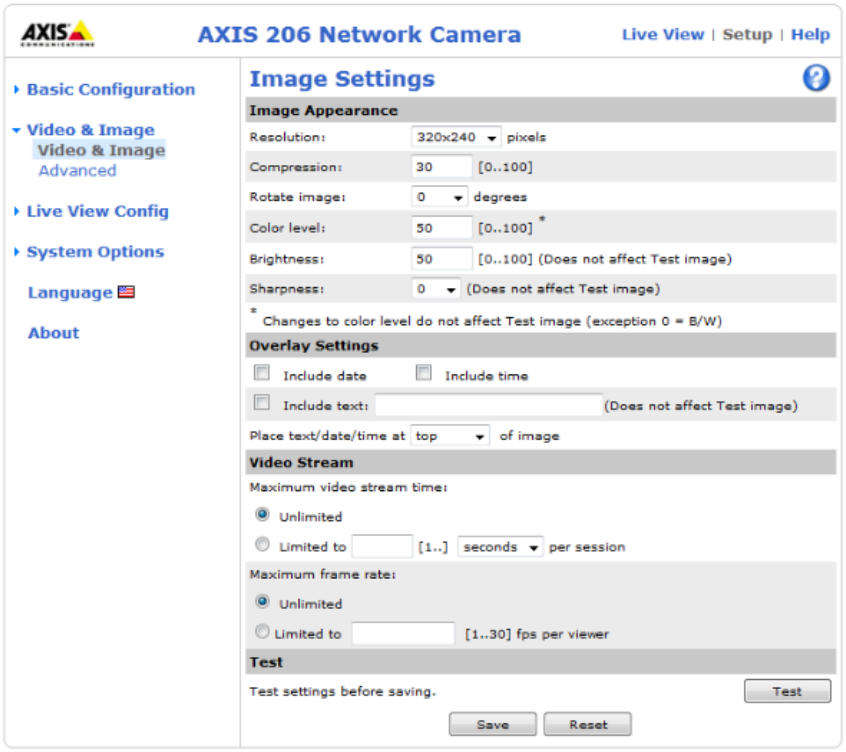
Click **Setup** to go to the setup page.

Configure Users



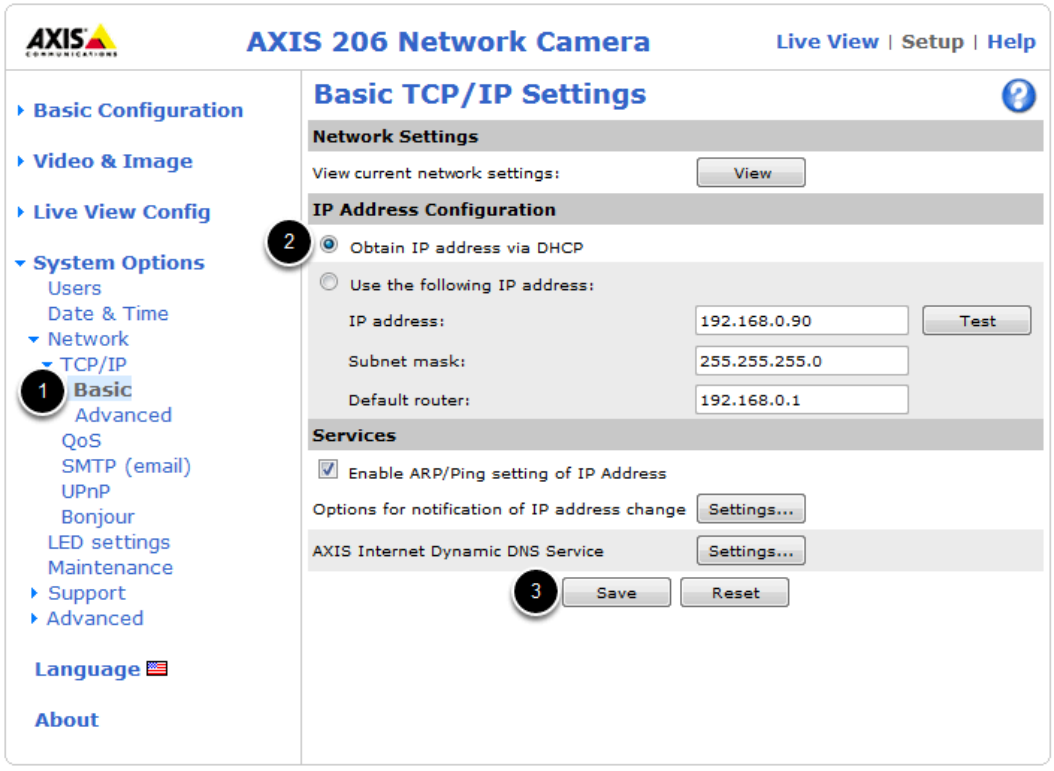
On the left side click **Users** to open the users page. Click **Add** then enter the Username **FRC** Password **FRC** and click the **Administrator** bubble, then click **OK.** If using the SmartDashboard, check the **Enable anonymous viewer login** box. Then click **Save**.

Configure Image Settings



Click **Video & Image** on the left side to open the image settings page. Set the **Resolution** and **Compression** to the desired values (recommended **320x240, 30**). To limit the framerate to under 30 FPS, select the **Limited to** bubble under **Maximum frame rate** and enter the desired rate in the box. Color, Brightness and Sharpness may also be set on this screen if desired. Click **Save** when finished.

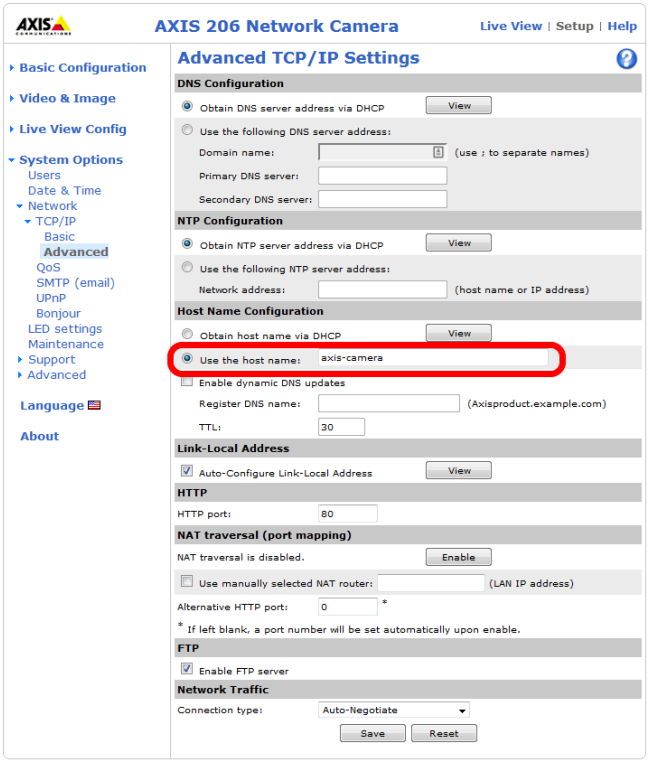
Configure Basic Network Settings



To configure the network settings of the camera, click the arrow to expand the **System Options** pane, then click the arrow to expand **Network**, then expand **TCP/IP** and select **Basic.** Set the camera to obtain an IP address via DHCP by selecting the bubble. Alternately, you may choose to set a static IP in the range 10.TE.AM.3 to 10.TE.AM.19. This is outside the range handed out by the DAP-1522 radio (home use) or FMS system (event use) so you will avoid any IP conflicts.

Click **Save**.

Configure Advanced Network Settings



Next click **Advanced** under **TCP/IP**. Set the **Host Name Configuration** to **"Use the host name:"** and set the value to **"axis-camera"** as shown. If you plan to use multiple cameras on your robot, select a unique host name for each. You will need to modify the dashboard and/or robot code to work with the additional cameras and unique host names.

Click **Save.**

Using the Microsoft Lifecam HD-3000

The Microsoft Lifecam HD-3000 is a USB webcam that was tested with the roboRIO as part of the Beta testing and software development effort. While other USB webcams may work with the roboRIO, this camera has been tested to be compatible with the provided software.

Connecting the camera to the roboRIO

Connecting the camera to the roboRIO

The camera can be connected to either of the roboRIO USB ports.

Using the camera - LabVIEW

Using the camera - LabVIEW

To stream the camera back to the Dashboard using LabVIEW, no additional code is necessary. Simply select USB HW (image compression done by the camera, fewer options but lower roboRIO CPU usage) or USB SW (image compressed by roboRIO, more options, but higher roboRIO CPU usage) and the image should begin streaming back.

Note: The camera should be plugged in before your LabVIEW code starts running to work properly. If you just plugged in the camera rebooting the roboRIO is a quick way to make sure it is recognized properly.

The default LabVIEW templates and the image processing examples are already set up for the USB camera if you want to do image processing. On the LabVIEW splash screen, click Tutorials, then click Tutorial 8 for more information about integrating Vision processing in your LabVIEW code.

Using the Camera - C++\Java

To stream the camera back to the Dashboard using C++ or Java robot code, you will need to add some code to your robot project. Example have been provided in Eclipse to illustrate the use of the CameraServer class to automatically stream images back to the Dashboard (SimpleVision) and send back modified images (IntermediateVision and the 2015 Vision examples). This class will allow images to be streamed back to either the SmartDashboard "USB Webcam Viewer" or the LabVIEW Dashboard (set to USB HW).

Determining the Camera name

Determining the Camera name

Unlike the LabVIEW code which attempts to determine the camera name of the camera you want to use, the C++\Java code requires you to specify the camera name. To determine the name of the desired camera, you will need to use the roboRIO webdashboard. For more information about accessing the roboRIO webdashboard see RoboRIO Webdashboard. Open the roboRIO webdashboard and locate the camera in the left pane and note the Name, this is the string you will pass into the camera server or IMAQDx open (in the image above, the camera name is cam0).

Using the SmartDashboard USB Camera Viewer

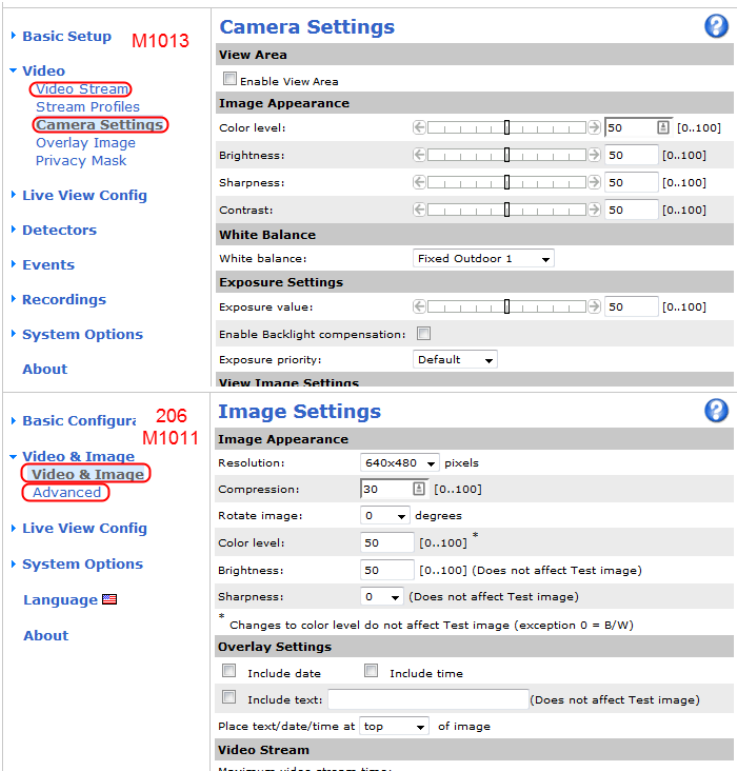
Using the SmartDashboard USB Camera Viewer

To view the camera stream from the LabVIEW dashboard, set the camera dropdown to USB HW. To view the stream from the SmartDashboard you will need to add a USB Webcam Viewer widget to your layout. For more information on the SmartDashboard and widgets see the SmartDashboard manual. To add the USB Webcam Viewer widget to the SmartDashboard, select View -> Add -> USB Webcam Viewer. To move or resize the viewer widget, make the layout Editable by selecting that option in the View menu (select again to disable).

Camera Settings

It is very difficult to achieve good image processing results without good images. With a light mounted near the camera lens, you should be able to use the provided examples, the dashboard or SmartDashboard, NI Vision Assistant or a web browser to view camera images and experiment with camera settings.

Changing Camera Settings



To change the camera settings on any of the supported Axis cameras (206, M1011, M1013), browse to the camera's webpage by entering it's address (usually 10.TE.AM.11) in a web browser. Click **Setup** near the top right corner of the page. On the M1013, the settings listed below are split between the **Video Stream** page and the **Camera Settings** page, both listed under the **Video** section.

Focus

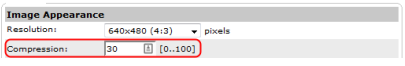
The Axis M1011 has a fixed-focus lens and no adjustment is needed. The Axis 206 camera has a black bezel around the lens that rotates to move the lens in and out and adjust focus. The Axis M103 has a silver and black bezel assembly around the lens to adjust the focus. Ensure that the images you are processing are relatively sharp and focused for the distances needed on your robot.

Compression

320x240 Color Image:
Compression set to 0. Image file size is 20,715 bytes.
High quality, but large in size. May be slower to compress and decompress -- which impacts frame rate.

320x240 Color Image:
Compression set to 30. Image file size is 6,450 bytes. Good quality, relatively small in size. Some image artifacts are present on edges.

320x240 Color Image:
Compression set to 100. Image file size is 2,222 bytes. Poor quality for processing. Notice blocky artifacts and rough edges.



The Axis camera returns images in BMP, JPEG, or MJPG format. BMP images are quite large and take more time to transmit to the cRIO and laptop. Therefore the WPILib implementations typically use MJPG motion JPEG. The compression setting ranges form 0 to 100, with 0 being very high quality images with very little compression, and 100 being very low quality images with very high compression. The camera default is 30, and it is a good compromise, with few artifacts that will degrade image processing. Due to implementation details within the VxWorks memory manager, you may notice a performance benefit if you keep the image sizes consistently below 16 kBytes.

Teams are advised to consider how the compression setting on the camera affects bandwidth if performing processing on the Driver Station computer, see the FMS Whitepaper for more details.

Resolution

Image sizes shared by the supported cameras are 160x120, 320x240, and 640x480. The M1011 and 1013 have additional sizes, but they aren't built into WPILib. The largest image size has four times as many pixels that are one-fourth the size of the middle size image. The large image has sixteen times as many pixels as the small image.

The tape used on the target is 4 inches wide, and for good processing, you will want that 4 inch feature to be at least two pixels wide. Using the distance equations above, we can see that a medium size image should be fine up to the point where the field of view is around 640 inches, a little over 53 feet, which is nearly double the width of the FRC field. This occurs at around 60 feet away, longer than the length of the field. The small image size should be usable for processing to a distance of about 30 feet or a little over mid-field.

Image size also impacts the time to decode and to process. Smaller images will be roughly four times faster than the next size up. If the robot or target is moving, it is quite important to minimize image processing time since this will add to the delay between the target location and perceived location. If both robot and target are stationary, processing time is typically less important.

Note: When requesting images using LabVIEW (either the Dashboard or Robot Code), the resolution and Frame Rate settings of the camera will be ignored. The LabVIEW code specifies the framerate and resolution as part of the stream request (this does not change the settings stored in the camera, it overrides that setting for the specific stream). The SmartDashboard and robot code in C++ or Java will use the resolution and framerate stored in the camera.

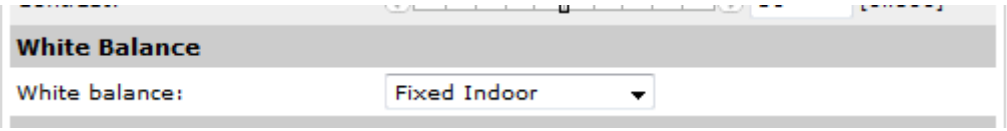
Frame Rate

The Axis Cameras have a max framerate of 30 frames per second. If desired, a limit can be set lower to reduce bandwidth consumption.

Color Enable

The Axis cameras typically return color images, but are capable of disabling color and returning a monochrome or grayscale image. The resulting image is a bit smaller in file size, and considerably quicker to decode. If processing is carried out only on the brightness or luminance of the image, and the color of the ring light is not used, this may be a useful technique for increasing the frame rate or lowering the CPU usage.

White Balance

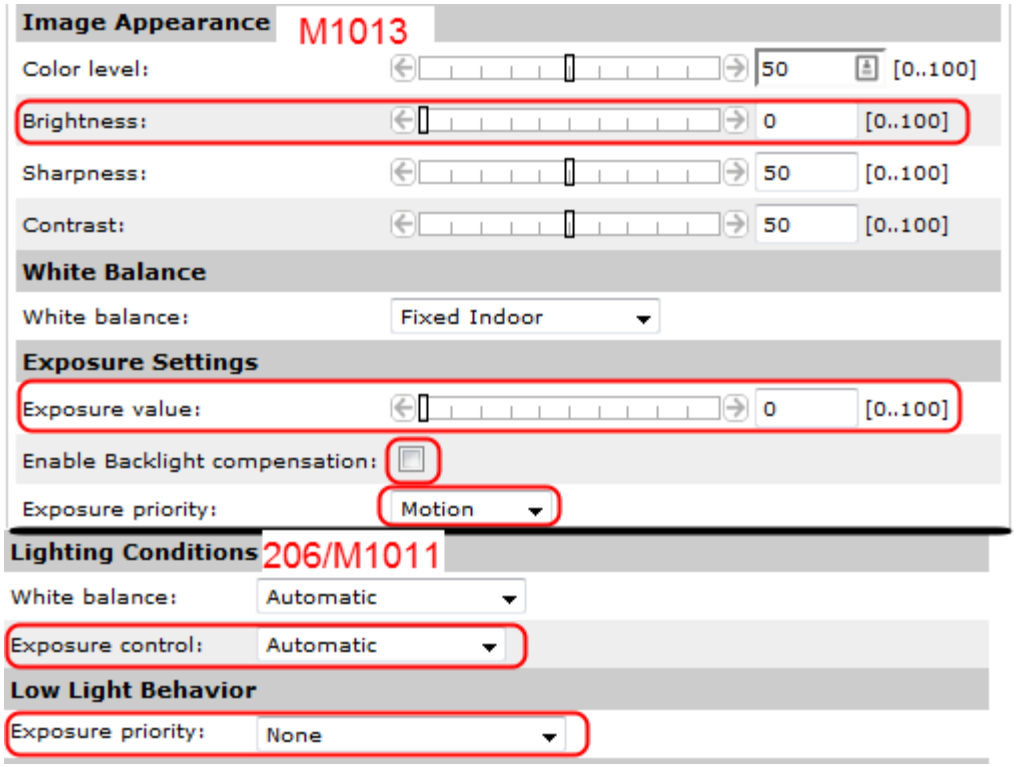


If the color of the light shine is being used to identify the marker, be sure to control the camera settings that affect the image coloring. The most important setting is white balance. It controls how the camera blends the component colors of the sensor in order to produce an image that matches the color processing of the human brain. The camera has five or six named presets, an auto setting that constantly adapts to the environment, and a hold setting -- for custom calibration.

The easiest approach is to use a named preset, one that maintains the saturation of the target and doesn't introduce problems by tinting neutral objects with the color of the light source.

To custom-calibrate the white balance, place a known neutral object in front of the camera. A sheet of white paper is a reasonable object to start with. Set the white balance setting to auto, wait for the camera to update its filters (ten seconds or so), and switch the white balance to hold.

Exposure



The brightness or exposure of the image also has an impact on the colors being reported. The issue is that as overall brightness increases, color saturation will start to drop. Lets look at an example to see how this occurs. A saturated red object placed in front of the camera will return an RGB measurement high in red and low in the other two e.g. (220, 20, 30). As overall white lighting increases, the RGB value increases to (240, 40, 50), then (255, 80, 90), then (255, 120, 130), and then (255, 160, 170). Once the red component is maximized, additional light can only increase the blue and green, and acts to dilute the measured color and lower the saturation. If the point is to identify the red object, it is useful to adjust the exposure to avoid diluting your principle color. The desired image will look somewhat dark except for the colored shine.

There are two approaches to control camera exposure times. One is to allow the camera to compute the exposure settings automatically, based on its sensors, and then adjust the camera's brightness setting to a small number to lower the exposure time. The brightness setting acts

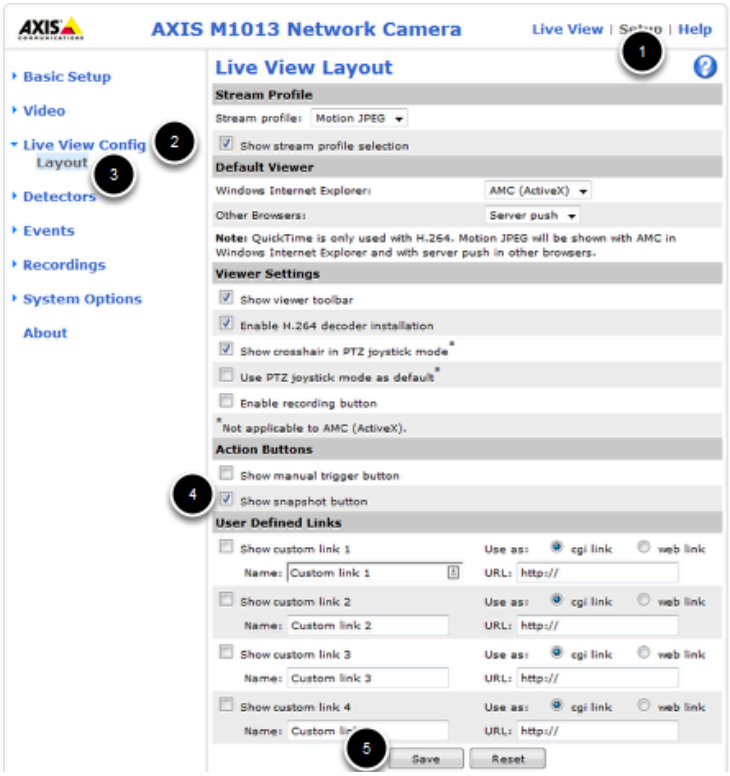
similar to the exposure compensation setting on SLR cameras. The other approach is to calibrate the camera to use a custom exposure setting. To do this on a 206 or M1011, change the exposure setting to auto, expose the camera to bright lights so that it computes a short exposure, and then change the exposure setting to hold. Both approaches will result in an overall dark image with bright saturated target colors that stand out from the background and are easier to mask.

The M1013 exposure settings look a little different. The Enable Backlight compensation option is similar to the Auto exposure settings of the M1011 and 206 and you will usually want to un-check this box. Adjust the Brightness and Exposure value sliders until your image looks as desired. The Exposure Priority should generally be set to Motion. This will prioritize framerate over image quality. Note that even with these settings the M1013 camera still performs some auto exposure compensation so it is recommended to check calibration frequently to minimize any impact lighting changes may have on image processing. See the article on Calibration for more details.

Calibration

While many of the numbers for the Vision Processing code can be determined theoretically, there are a few parameters that are typically best to measure empirically then enter back into the code (a process typically known as calibration). This article will show how to perform calibration for the Color (masking), and View Angle (distance) using the NI Vision Assistant. If you are using C++ or Java and have not yet installed the NI Vision Assistant, see the article Installing NI Vision Assistant.

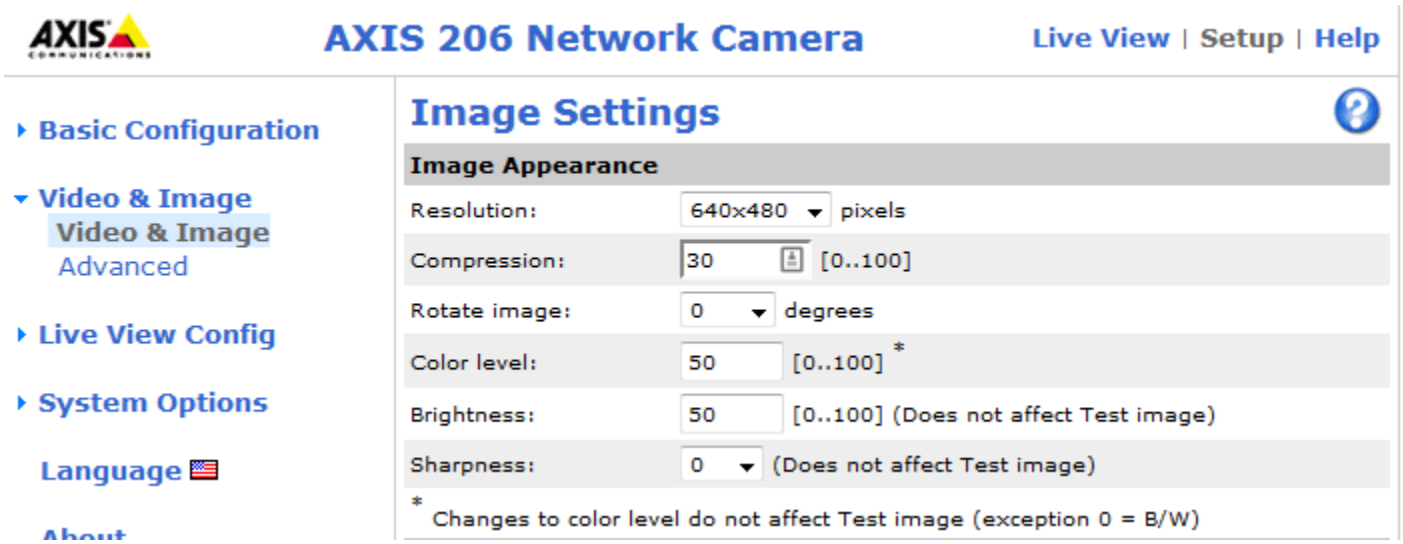
Enable Snapshots



To capture snapshots from the Axis camera, you must first enable the Snapshot button. Open a web-browser and browse to camera's address (10.TE.AM.11), enter the Username/Password

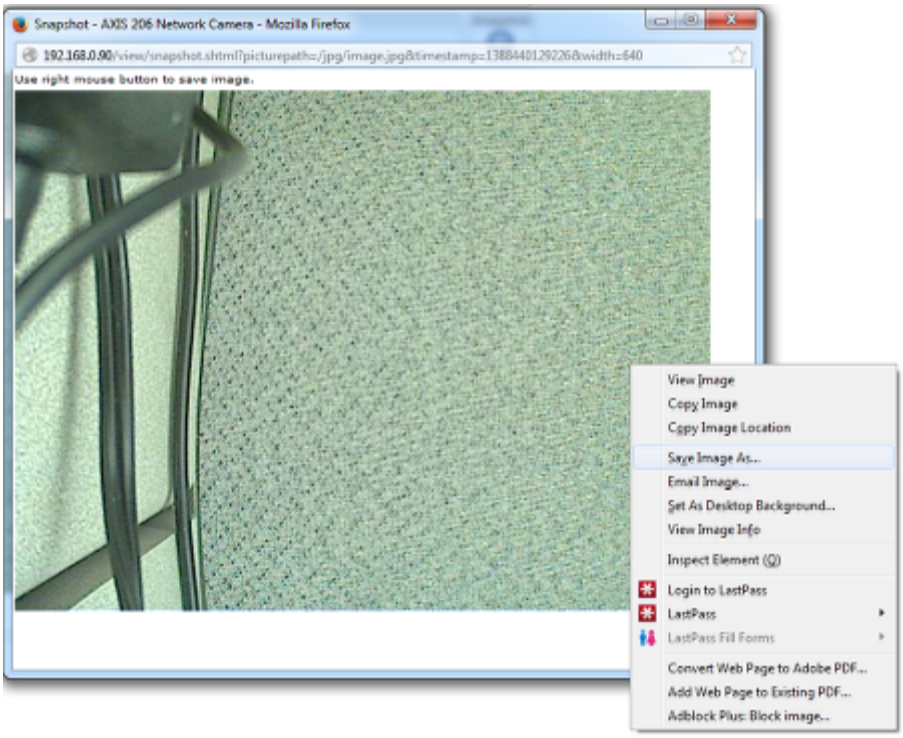
combo FRC/FRC if prompted, then click Setup->Live View Config->Layout. Click on the checkbox to Show snapshot button then click Save.

Check Camera Settings



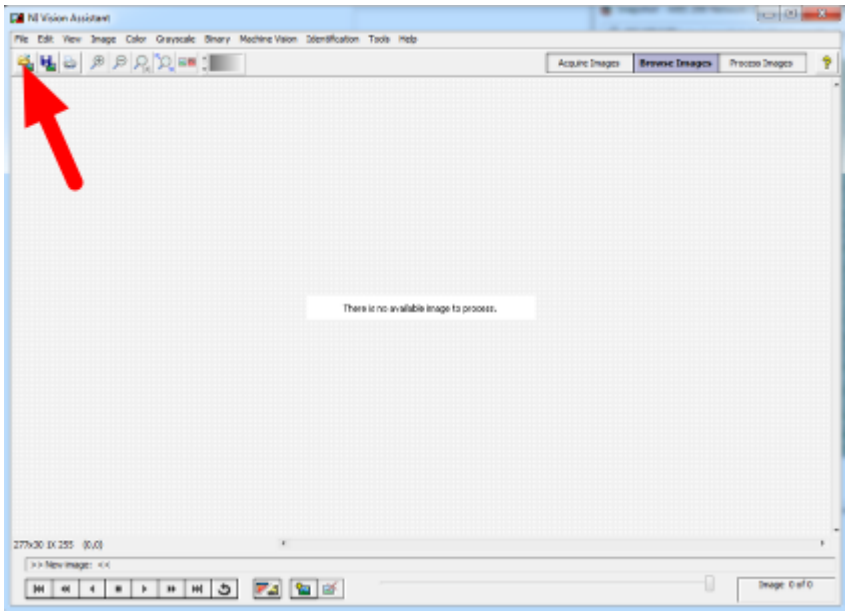
Depending on how you are capturing the image stream in your program, it may be possible to stream a different resolution, framerate and/or compression than what is saved in the camera and used in the Live View. Before performing any calibration it is recommended you verify that the settings in the camera match the settings in your code. To check the settings in the camera, click on the Video and Image header on the left side of the screen, then click Video and Image.

Capture Images



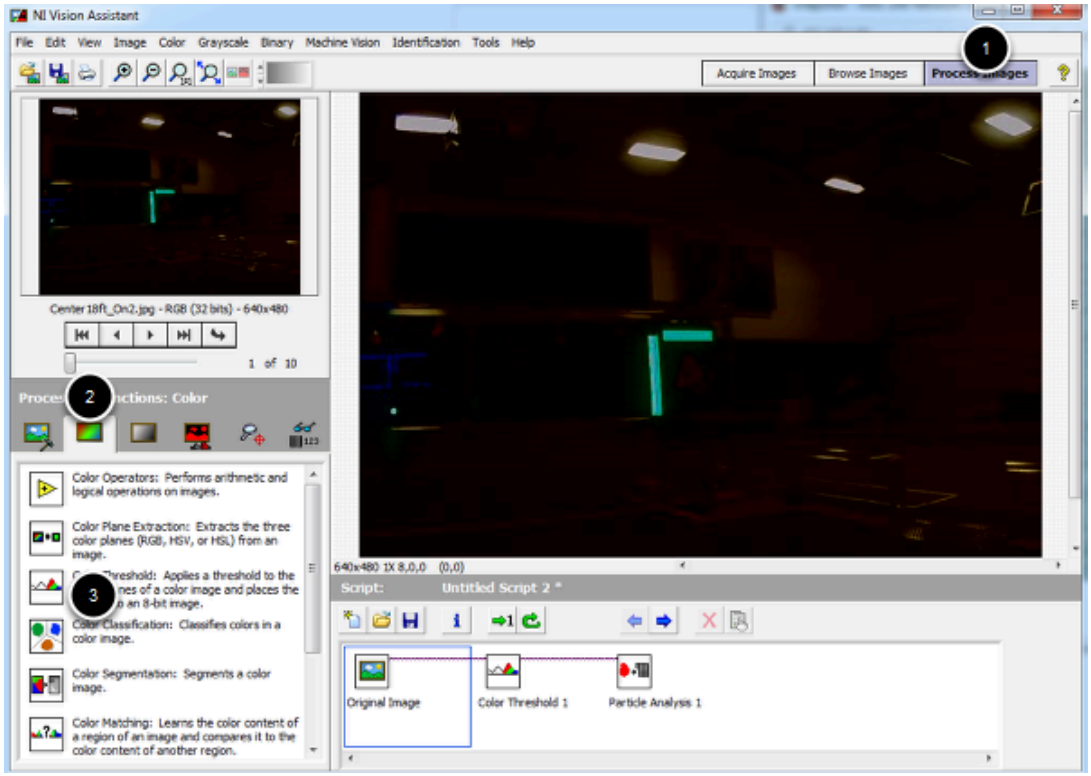
Click the Live View button to return to the Live View page and you should now see a Snapshot button. Clicking this button opens a pop-up window with a static image capture. Right-click on this image, select Save Image as and select your desired location and file name, then save the image.

Load Image(s) in Vision Assistant



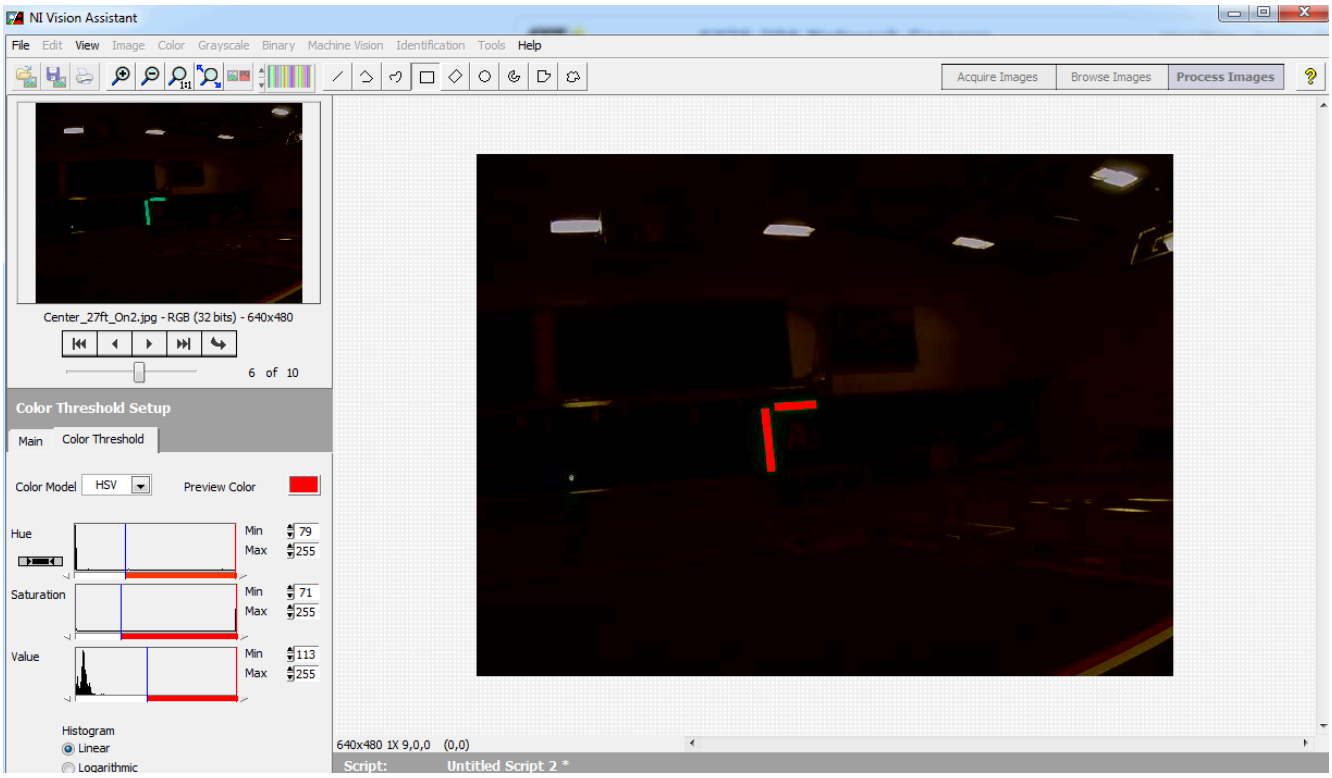
Open the NI Vision Assistant and select the Browse Images option. Select the Open Images icon in the top left of the Toolbar, then locate your images. Repeat as necessary to load all desired images.

Color Threshold



Click Process Images in the top right, then select the color tab on the bottom right and click the Color Threshold icon.

HSV Calibration



Change the Color Model dropdown to HSV. Next tune the window on each of the three values to cover as much of the target as possible while filtering everything else. If using a green light, you may want to use the values in the sample code as a starting point. If you have multiple images you can use the controls in the top left to cycle through them. Use the center two arrow controls or the slider to change the preview image in the top left window, then click the right-most arrow to make it the active image. When you are happy with the values you have selected, note down the ranges for the Hue, Saturation and Value. You will need to enter these into the appropriate place in the vision code. Click OK to finish adding the step to the script.

You may wish to take some new sample images using the time for camera calibration at your event to verify or tweak your ranges slightly based on the venue lighting conditions.

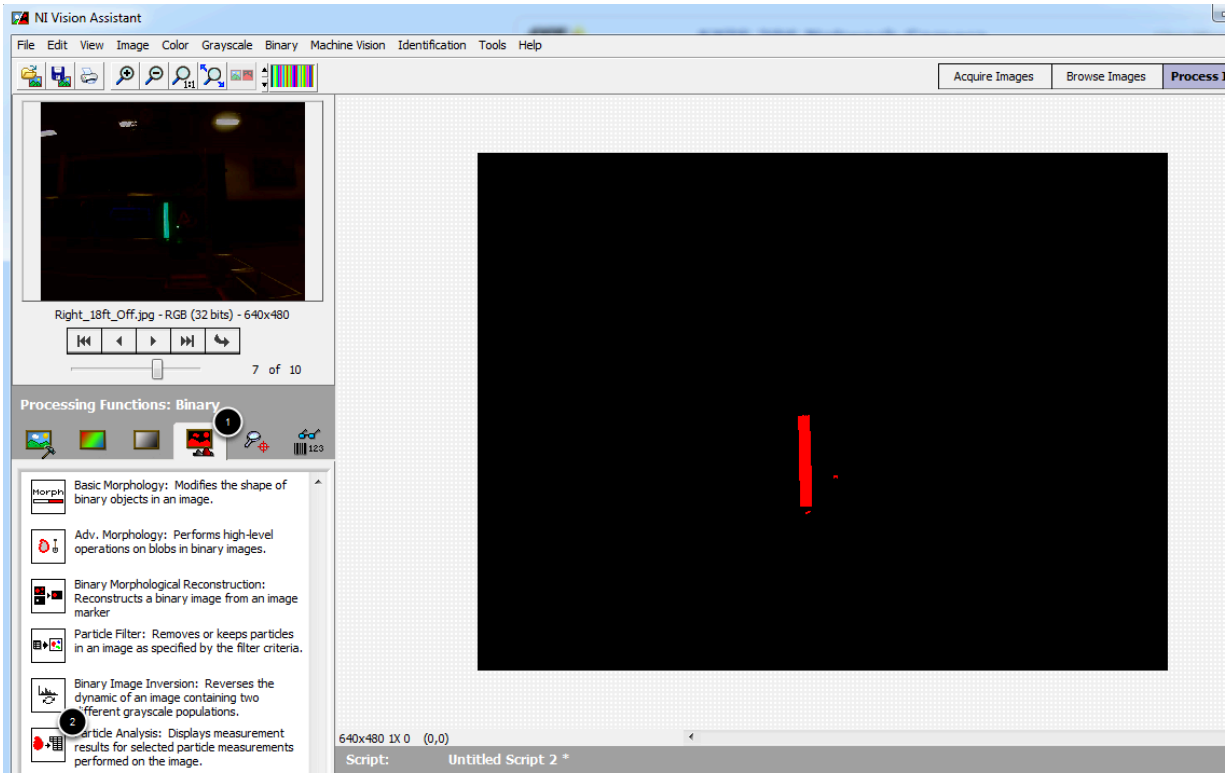
View Angle/Distance Calibration

While a theoretical view angle for each camera model can be found in the datasheet, empirical testing has found that these numbers may be a bit off even for the horizontal view angle. Given that this year's code uses the vertical field-of-view it is best to perform your own calibration for your camera (though empirical values for each camera type are included in the code as a reference). To do this set up an equation where the view angle, Θ, is the only unknown. To do this, utilize a target of known size at a known distance, leaving the view angle as the only unknown. Let's take our equation from the previous article, d = Tft*FOVpixel/(Tpixel*tanΘ), and re-arrange it to solve for Θ:

tanΘ = Tft*FOVpixel/(Tpixel*d)

Θ = arctan(Tft*FOVpixel/(Tpixel*d))

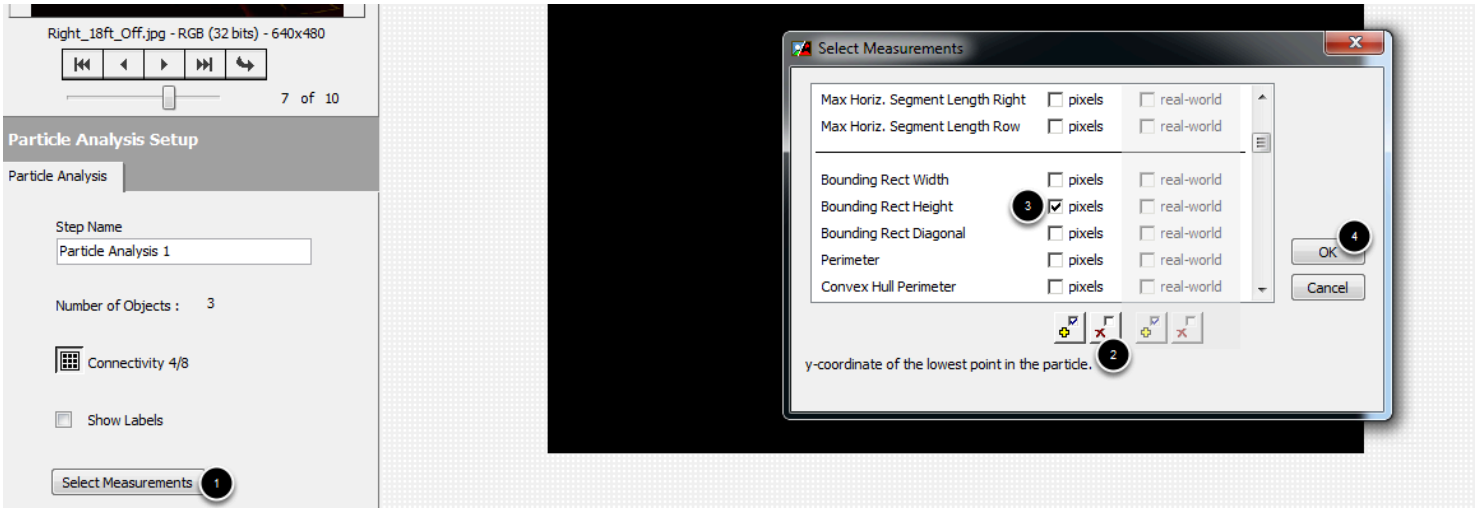
Taking measurements



One way to take the required measurements is to use the same images of the retro-reflective tape that were used for the color calibration above. We can use Vision Assistant to provide the height of the detected blob in pixels. By measuring the real-world distance between the camera and the target, we now have all of the variables to solve our equation for the view angle.

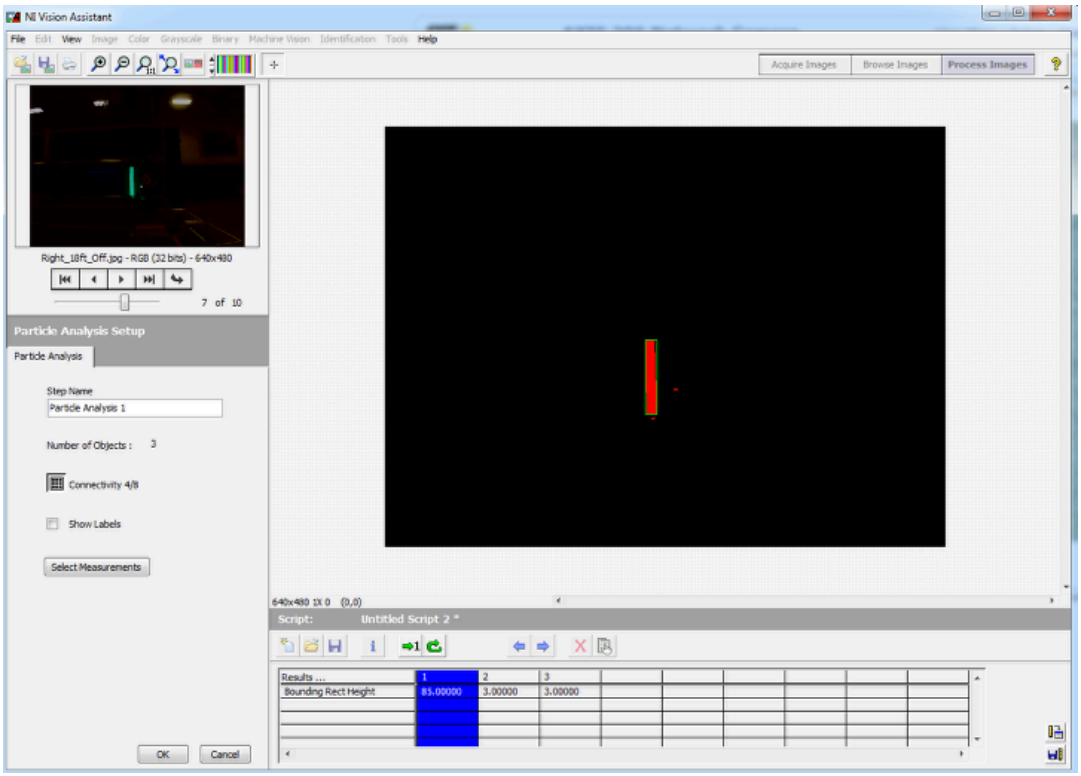
To measure the particles in the image, click the Binary tab, then click the Particle Analysis icon.

Selecting Measurements



Click on the Select Measurements button. In this case, we are only interested in the bounding box height. Click on the button with the X to deselect all measurements, then locate the Bounding Rect Height measurement and check the box. Click OK to save.

Measuring the Particle



The measurements for each particle will now be displayed in the window at the bottom of the screen. If your image has multiple particles, you can click in each box to have Vision Assistant highlight the particle so you can make sure you have the right one. This article will show the calculation using a single image, but you may wish to perform the calculation on multiple images from multiple distances and use a technique such as averaging or least squares fit to determine the appropriate value for the View angle. You can use the same arrow controls described in the color section above to change the active image.

Calculation

As seen in the previous step, the particle representing the 32in tall vertical target in this example measured 85 pixels tall in a 640x480 image. The image shown was taken from (very roughly) 18 ft. away. Plugging these numbers into the equation from above....

Θ = arctan(2.66*480/(2*85*18)) = 22.65 degrees

Depending on what you use to calculate the arctangent, your answer may be in radians, make sure to convert back to degrees if entering directly into the sample code as the view angle.

Note: The code uses **View Angle** and we just calculated Θ. Make sure to multiply Θ by 2 if replacing the constants in the code.Multiplying our result by 2 yields 45.3 degrees. This image is from a M1013 camera, so our value is a bit off from the previously measured 29.1 but given that the 18ft. was a very rough measurement this shows that we are in the ballpark and likely performed the calculation correctly.

Axis M1013 Camera Compatibility

It has come to our attention that the Axis M1011 camera has been discontinued and superseded by the Axis M1013 camera. This document details any differences or issues we are aware of between the two cameras when used with WPILib and the provided sample vision programs.

Optical Differences

The Axis M1013 camera has a few major optical differences from the M1011 camera:

1. The M1013 is an adjustable focus camera. Make sure to focus your M1013 camera by turning the grey and black lens housing to make sure you have a clear image at your desired viewing distance.
2. The M1013 has a wider view angle (67 degrees) compared to the M1011 (47 degrees). This means that for a feature of a fixed size, the image of that feature will span a smaller number of pixels

Using the M1013 With WPILib

The M1013 camera has been tested with all of the available WPILib parameters and the following performance exceptions were noted:

1. The M1013 does not support the 160x120 resolution. Requesting a stream of this resolution will result in no images being returned or displayed.
2. The M1013 does not appear to work with the Color Enable parameter exposed by WPILib. Regardless of the setting of this parameter a full color image was returned.

All other WPILib camera parameters worked as expected. If any issues not noted here are discovered, please file a bug report on the WPILib tracker (note that you will need to create an account if you do not have one, but you do not need to be a member of the project).

Using the Axis Camera at Single Network Events

The 2015 convention for using the Axis camera uses mDNS with the camera name set to axis-camera.local At home this works fine as there is only one camera on the network. At official events, this works fine as each team is on their own VLAN and therefore doesn't have visibility to another team's camera. At an offseason using a single network, this will cause an issue where all teams will connect to whichever team's camera "wins" and becomes "axis-camera", the other cameras will see that the name is taken and use an alternative name. This article describes how to modify the Dashboard and/or robot code to use a different mDNS name to separate the camera streams.

Changing the camera mDNS name

To change the mDNS name in the camera, follow the instructions in Configuring an Axis Camera but substitute the new name such as axis-cameraTEAM where TEAM is your team number.

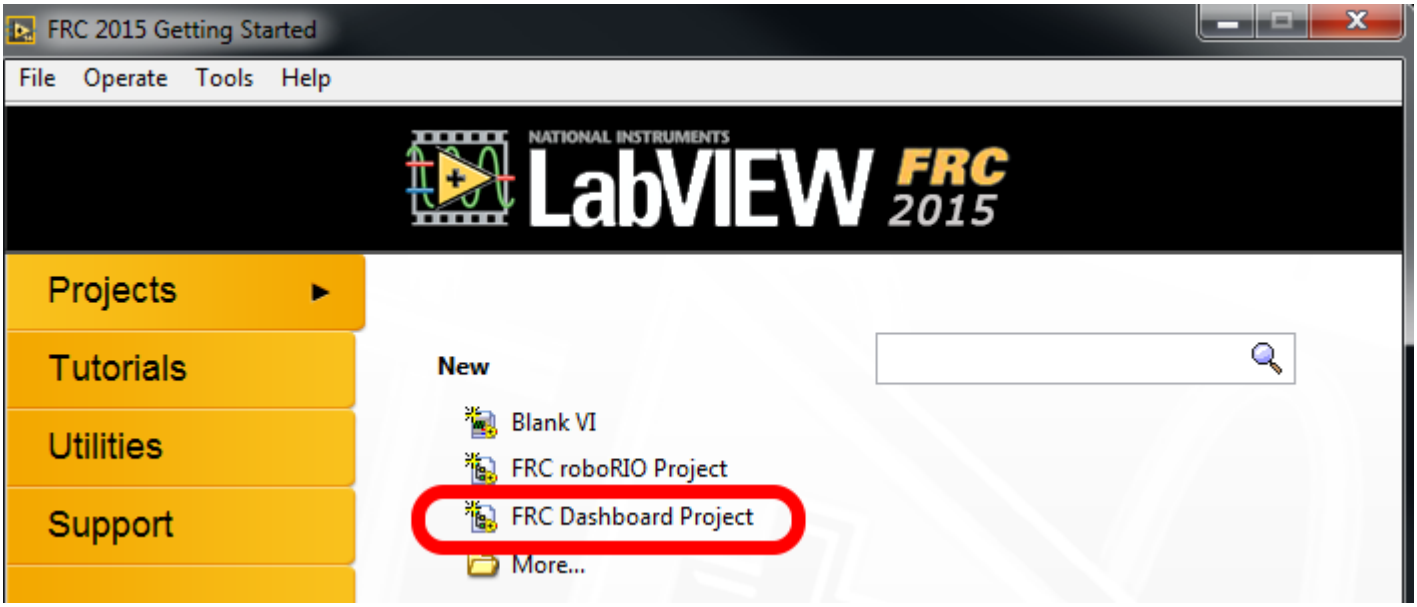
Viewing the camera on the DS PC - Browser or Java SmartDashboard

If you are using a web-browser or the updated Java SmartDashboard (which accepts mDNS names for the Simple Camera Viewer widget), updating to use the new mDNS name is simple. Simply change the URL in the browser or the address in the Simple Camera Viewer widget properties to the new mDNS name and you are all set.

Viewing the camera on the DS PC - LabVIEW Dashboard

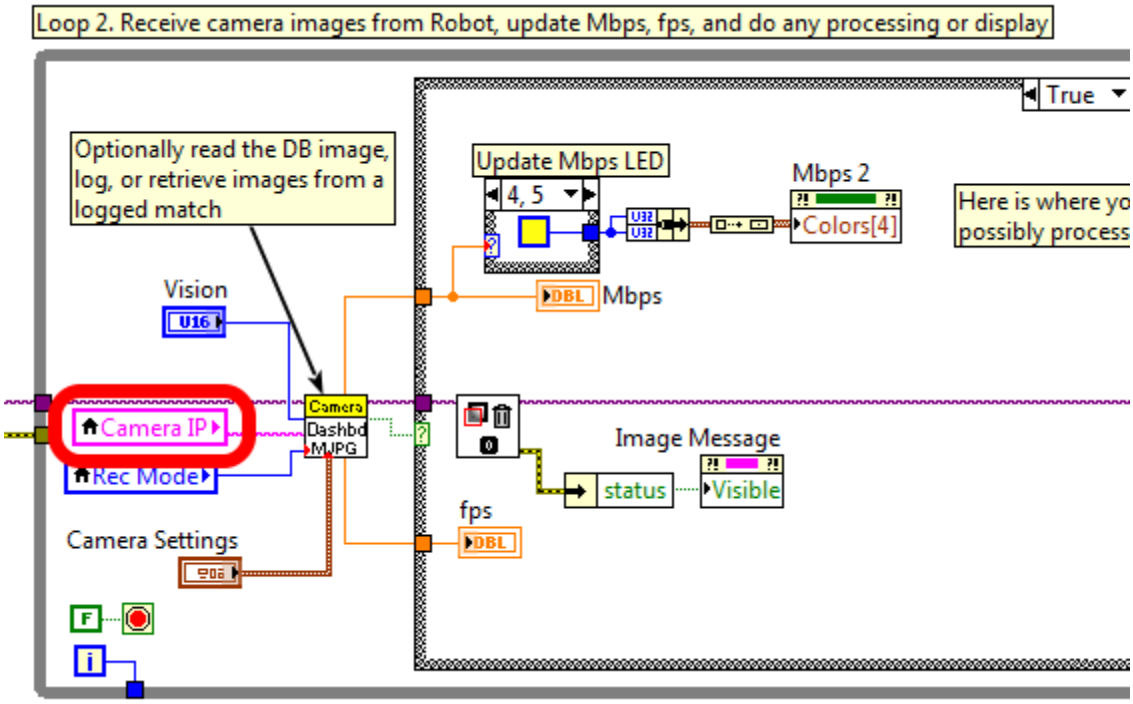
To view the camera stream in the LabVIEW Dashboard, you will need to build a customized version of the Dashboard. Note that this customized version will only work for the Axis camera and will no longer work for a USB camera, revert to the default Dashboard to use a USB camera.

Creating a Dashboard Project



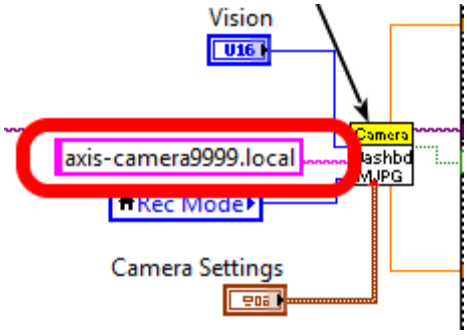
From the LabVIEW Splash screen, select "FRC Dashboard Project". Name the project as desired, then click Finish.

Locating Loop 2 - Camera IP



Double click on Dashboard Main.vi in the project explorer to open it and press Ctrl+e to see the block diagram. Scroll down to the loop with the comment that says Loop 2 and locate the "Camera IP" input.

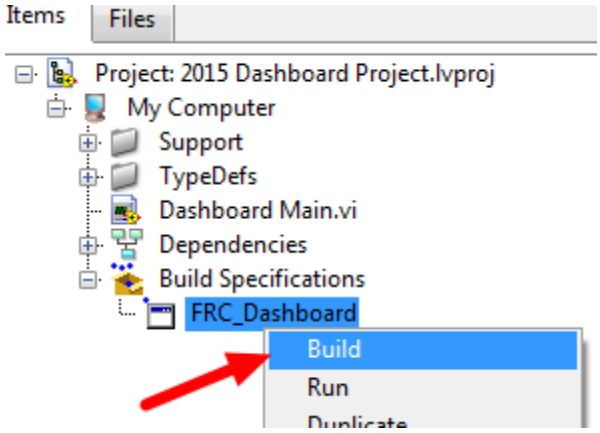
Editing the camera IP



Delete the Camera IP node, right click on the broken wire and click Create Constant (connect the constant to the wire if necessary). In the box, enter the mDNS name of your camera with a ".local" suffix (e.g. "axis-cameraTEAM.local" where TEAM is replaced with your team number). In this example I have used a sample name for team 9999. Then click File->Save or Ctrl+S to save the VI.

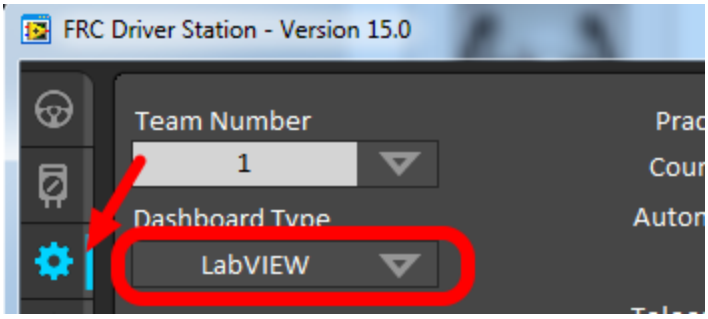
Note: You may also wish to make a minor modification to the Front Panel to verify that you are running the right dashboard later.

Building the Dashboard



To build the new dashboard, expand Build Specifications in the Project Explorer, right click on FRC_Dashboard and select Build.

Setting the Driver Station to launch the modified Dashboard



On the Setup tab of the Driver Station, change to dropdown box to LabVIEW to launch your new Dashboard.

Accessing the camera from Robot Code

If you wish to access the renamed camera from your robot code, you will have to modify it as well. In C++ and Java, just change the String used for the camera host name to match the new name. In LabVIEW follow the step below.

Modifying LabVIEW Robot Code

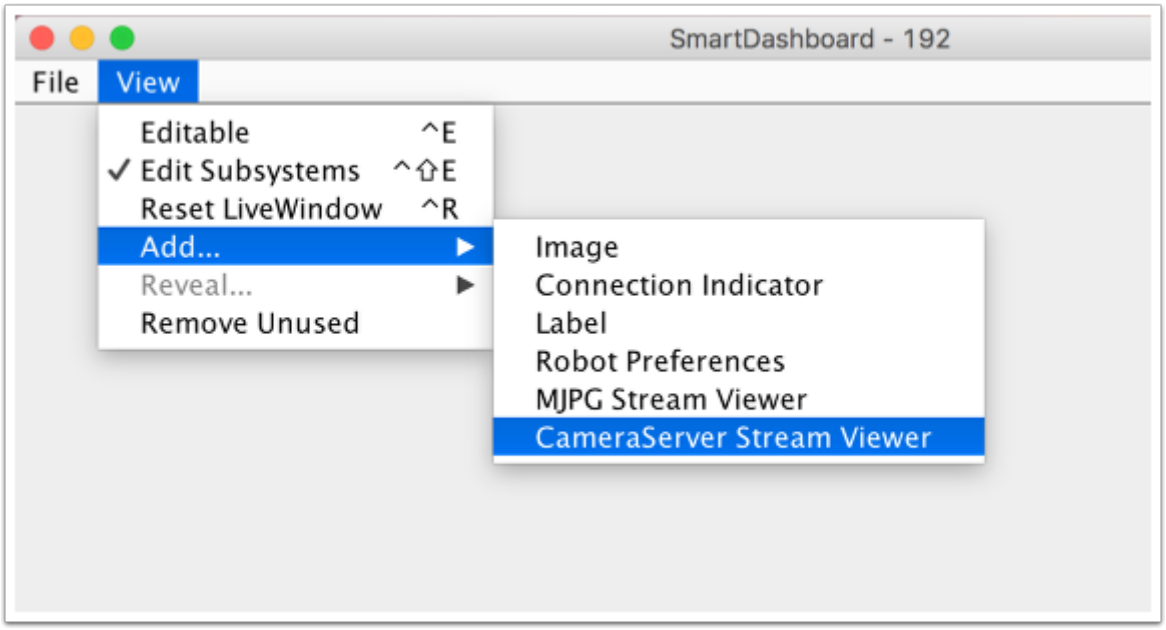
In the Project Explorer, locate Vision Processing.VI and double click to open it. Then press Ctrl+e to open the Block Diagram. Locate the string "axis-camera.local" near the left of the image and replace with "axis-cameraTEAM.local" Also make sure the constant is set to "False" to use the Axis camera instead of USB.

Vision programming

Using the CameraServer on the roboRIO

Simple CameraServer program

The following program gets a CameraServer instance and starts automatic capture of a USB camera like the Microsoft LifeCam that is connected to the roboRIO. In this mode, the camera will capture frames and send them to the SmartDashboard. To view the images, create a CameraServer Stream Viewer widget using the "View", then "Add" menu in the dashboard. The images are unprocessed and just forwarded from the camera to the dashboard.




```
package org.usfirst.frc.team190.robot;

import edu.wpi.first.wpilibj.CameraServer;
import edu.wpi.first.wpilibj.IterativeRobot;

public class Robot extends IterativeRobot {

    public void robotInit() {
        CameraServer.getInstance().startAutomaticCapture();
    }
}
```

```
#include "WPILib.h"
class Robot: public IterativeRobot
{
private:
    void RobotInit()
    {
        CameraServer::GetInstance()->StartAutomaticCapture();
    }
};
START_ROBOT_CLASS(Robot)
```

Advanced camera server program

In the following example a thread created in robotInit() gets the Camera Server instance. ¬†Each frame of the video is individually processed, in this case converting a color image (BGR) to gray scale using the OpenCV cvtColor() method. The resultant images are then passed to the output stream and sent to the dashboard. You can replace the cvtColor operation with any image processing code that is necessary for your application. You can even annotate the image using OpenCV methods to write targeting information onto the image being sent to the dashboard.

```
package org.usfirst.frc.team190.robot;

import org.opencv.core.Mat;
import org.opencv.imgproc.Imgproc;

import edu.wpi.cscore.CvSink;
import edu.wpi.cscore.CvSource;
import edu.wpi.cscore.UsbCamera;
import edu.wpi.first.wpilibj.CameraServer;
import edu.wpi.first.wpilibj.IterativeRobot;

public class Robot extends IterativeRobot {

    public void robotInit() {
            new Thread(() -> {
                UsbCamera camera = CameraServer.getInstance().startAutomaticCapture();
                camera.setResolution(640, 480);
```

```
                CvSink cvSink = CameraServer.getInstance().getVideo();
                CvSource outputStream = CameraServer.getInstance().putVideo("Blur", 640,
480);

                Mat source = new Mat();
                Mat output = new Mat();

                while(!Thread.interrupted()) {
                    cvSink.grabFrame(source);
                    Imgproc.cvtColor(source, output, Imgproc.COLOR_BGR2GRAY);
                    outputStream.putFrame(output);
                }
            }).start();
    }
}
```

```
#include "WPILib.h"
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/core/core.hpp>
class Robot: public IterativeRobot
{
private:
    static void VisionThread()
    {
        cs::UsbCamera camera = CameraServer::GetInstance()->StartAutomaticCapture();
        camera.SetResolution(640, 480);
        cs::CvSink cvSink = CameraServer::GetInstance()->GetVideo();
        cs::CvSource outputStreamStd = CameraServer::GetInstance()->PutVideo("Gray", 640,
480);
        cv::Mat source;
        cv::Mat output;
        while(true) {
            cvSink.GrabFrame(source);
            cvtColor(source, output, cv::COLOR_BGR2GRAY);
            outputStreamStd.PutFrame(output);
        }
    }
    void RobotInit()
    {
```

```
        std::thread visionThread(VisionThread);
        visionThread.detach();
    }
};
START_ROBOT_CLASS(Robot)
```

Notice that in these examples, the PutVideo method writes the video to a named stream. To view that stream on the SmartDashboard set the properties on the CameraServerStreamViewer to refer the the named stream. In this case that is "Blur" for the Java program and "Gray" for the C++ sample.

Using multiple cameras

Switching the driver views

If you're interested in just switching what the driver sees, and are using SmartDashboard, the SmartDashboard CameraServer Stream Viewer has an option ("Selected Camera Path") that reads the given NT key and changes the "Camera Choice" to that value (displaying that camera). The robot code then just needs to set the NT key to the correct camera name. Assuming "Selected Camera Path" is set to "CameraSelection", the following code uses the joystick 1 trigger button state to show camera1 and camera2.

```
cs::UsbCamera camera1;
cs::UsbCamera camera2;
frc::Joystick joy1{0};
bool prevTrigger = false;
void RobotInit() override {
  camera1 = frc::CameraServer::GetInstance()->StartAutomaticCapture(0);
  camera2 = frc::CameraServer::GetInstance()->StartAutomaticCapture(1);
}
void TeleopPeriodic() override {
  if (joy1.GetTrigger() && !prevTrigger) {
    printf("Setting camera 2\n");
    nt::NetworkTableInstance::GetDefault().GetTable("")->PutString("CameraSelection",
camera2.GetName());
  } else if (!joy1.GetTrigger() && prevTrigger) {
    printf("Setting camera 1\n");
    nt::NetworkTableInstance::GetDefault().GetTable("")->PutString("CameraSelection",
camera1.GetName());
  }
  prevTrigger = joy1.GetTrigger();
}
```

If you're using some other dashboard, you can change the camera used by the camera server dynamically. If you open a stream viewer nominally to camera1, the robot code will change the stream contents to either camera1 or camera2 based on the joystick trigger.

```
cs::UsbCamera camera1;
cs::UsbCamera camera2;
```

```
cs::VideoSink server;
frc::Joystick joy1{0};
bool prevTrigger = false;
void RobotInit() override {
  camera1 = frc::CameraServer::GetInstance()->StartAutomaticCapture(0);
  camera2 = frc::CameraServer::GetInstance()->StartAutomaticCapture(1);
  server = frc::CameraServer::GetInstance()->GetServer();
}
void TeleopPeriodic() override {
  if (joy1.GetTrigger() && !prevTrigger) {
    printf("Setting camera 2\n");
    server.SetSource(camera2);
  } else if (!joy1.GetTrigger() && prevTrigger) {
    printf("Setting camera 1\n");
    server.SetSource(camera1);
  }
  prevTrigger = joy1.GetTrigger();
}
```

Keeping Streams Open

By default, the cscore library is pretty aggressive in turning off cameras not in use. What this means is that when you switch cameras, it may disconnect from the camera not in use, so switching back will have some delay as it reconnects to the camera. To keep both camera connections open, use the SetConnectionStrategy() method to tell the library to keep the streams open, even if you aren't using them.

```
cs::UsbCamera camera1;
cs::UsbCamera camera2;
cs::VideoSink server;
frc::Joystick joy1{0};
bool prevTrigger = false;
void RobotInit() override {
  camera1 = frc::CameraServer::GetInstance()->StartAutomaticCapture(0);
  camera2 = frc::CameraServer::GetInstance()->StartAutomaticCapture(1);
  server = frc::CameraServer::GetInstance()->GetServer();
  camera1.SetConnectionStrategy(cs::VideoSource::ConnectionStrategy::kConnectionKeepOpen);
  camera2.SetConnectionStrategy(cs::VideoSource::ConnectionStrategy::kConnectionKeepOpen);
```

```
}
void TeleopPeriodic() override {
  if (joy1.GetTrigger() && !prevTrigger) {
    printf("Setting camera 2\n");
    server.SetSource(camera2);
  } else if (!joy1.GetTrigger() && prevTrigger) {
    printf("Setting camera 1\n");
    server.SetSource(camera1);
  }
  prevTrigger = joy1.GetTrigger();
}
```

If both cameras are USB, it's worth noting that you may run into USB bandwidth limitations with higher resolutions, as in all of these cases the roboRio is going to be streaming data from both cameras to the roboRio simultaneously (for a short period in options 1 and 2, and continuously in option 3). It is theoretically possible for the library to avoid this simultaneity in the option 2 case (only), but this is not currently implemented.

Different cameras report bandwidth usage differently. The library will tell you if you're hitting the limit; you'll get this error message: "could not start streaming due to USB bandwidth limitations; try a lower resolution or a different pixel format (VIDIOC_STREAMON: No space left on device)". If you're using Option 3 it will give you this error during RobotInit(). Thus you should just try your desired resolution and adjusting as necessary until you both don't get that error and don't exceed the radio bandwidth limitations.

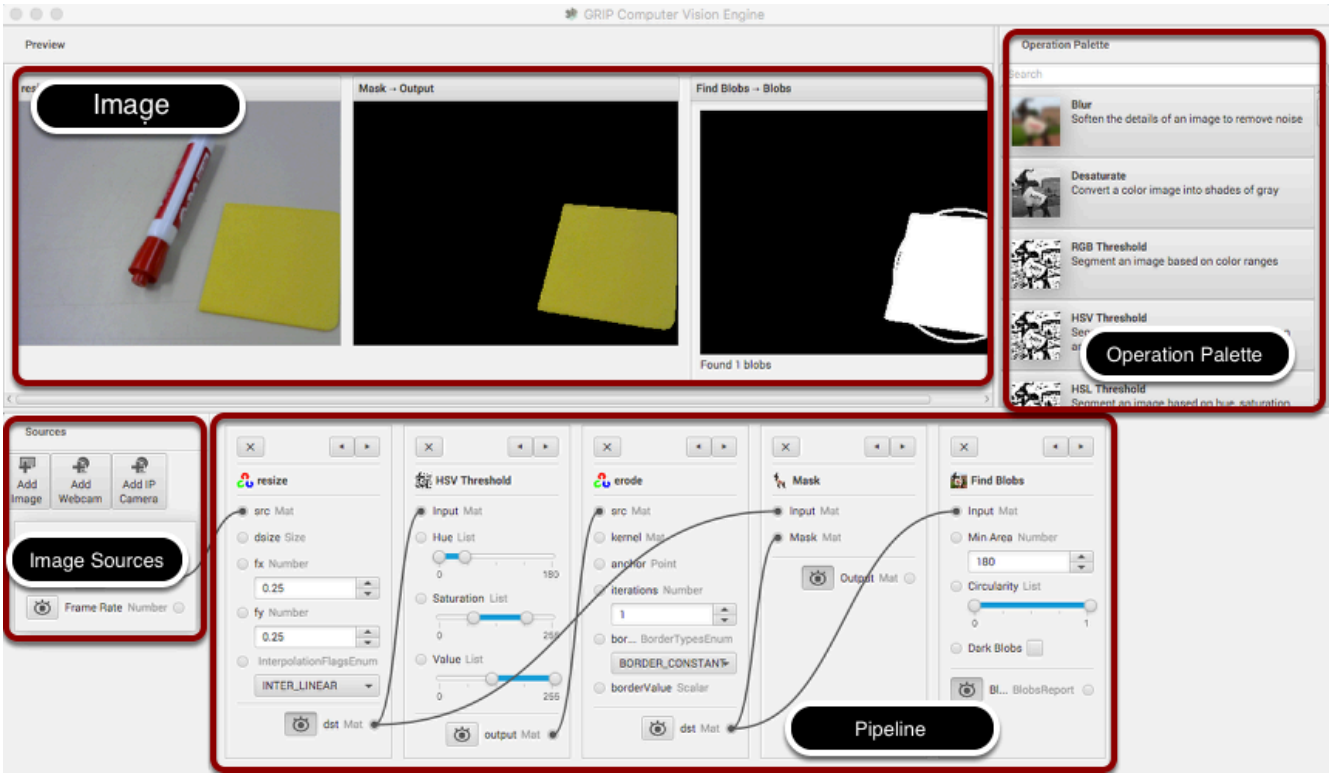
GRIP

Introduction to GRIP

GRIP is a tool for developing computer vision algorithms interactively rather than through trial and error coding. After developing your algorithm you may run GRIP in headless mode on your roboRIO, on a Driver Station Laptop, or on a coprocessor connected to your robot network. With Grip you choose vision operations to create a graphical pipeline that represents the sequence of operations that are performed to complete the vision algorithm.

GRIP is based on OpenCV, one of the most popular computer vision software libraries used for research, robotics, and vision algorithm implementations. The operations that are available in GRIP are almost a 1 to 1 match with the operations available if you were hand coding the same algorithm with some text-based programming language.

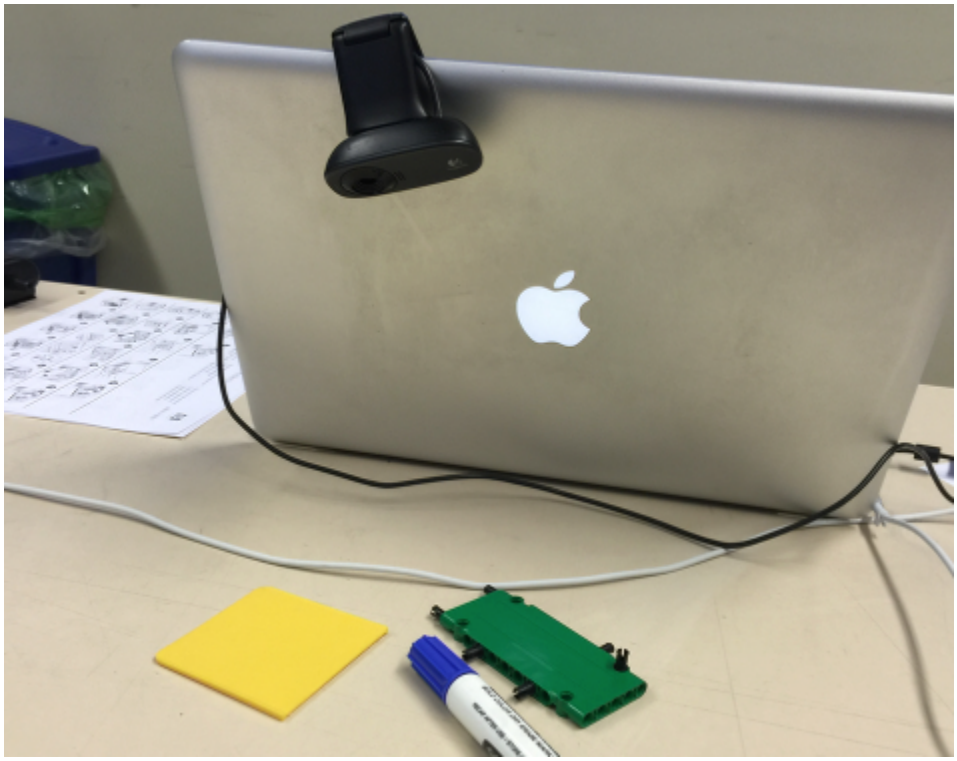
The GRIP user interface



The GRIP user interface consists of 4 parts:

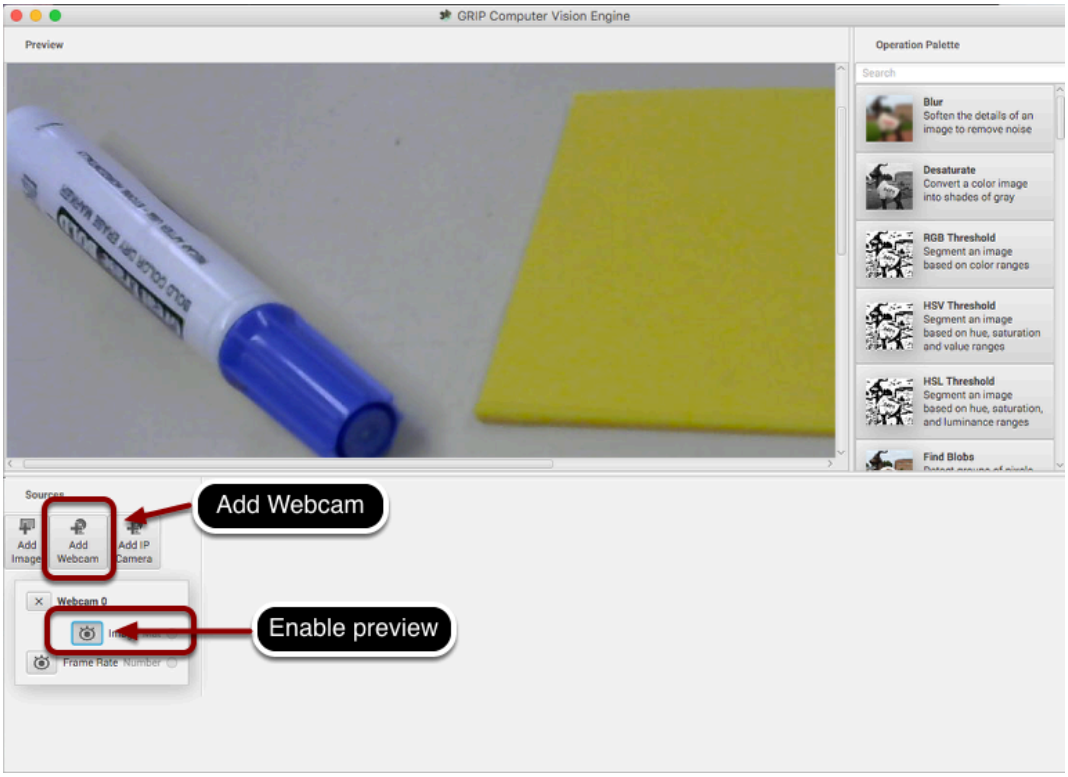
- **Image Sources** are the ways of getting images into the GRIP pipeline. You can provide images through attached cameras or files. Sources are almost always the beginning of the image processing algorithm.
- **Operation Palette** contains the image processing steps from the OpenCV library that you can chain together in the pipeline to form your algorithm. Clicking on an operation in the palette adds it to the end of the pipeline. You can then use the left and right arrows to move the operation within the pipeline.
- **Pipeline** is the sequence of steps that make up the algorithm. Each step (operation) in the pipeline is connected to a previous step from the output of one step to an input to the next step. The data flows from generally from left to right through the connections that you create.
- **Image Preview** are shows previews of the result of each step that has it's preview button pressed. This makes it easy to debug algorithms by being able to preview the outputs of each intermediate step.

Finding the yellow square



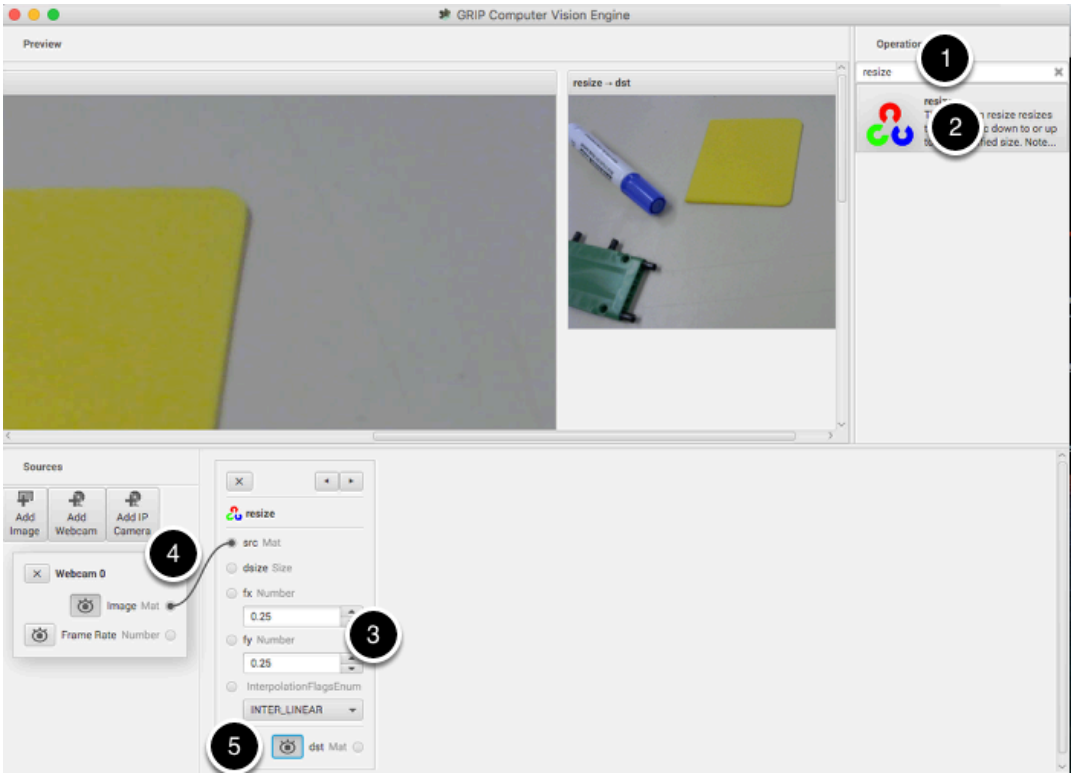
In this application we will try to find the yellow square in the image and display it's position. The setup is pretty simple, just a USB web camera connected to the computer looking down at some colorful objects. The yellow plastic square is the thing that we're interested in locating in the image.

Enable the image source



The first step is to acquire an image. To use the source, click on the "Add Webcam" button and select the camera number. In this case the Logitech USB camera that appeared as Webcam 0 and the computer monitor camera was Webcam 1. The web camera is selected in this case to grab the image behind the computer as shown in the setup. Then select the image preview button and the real-time display of the camera stream will be shown in the preview area.

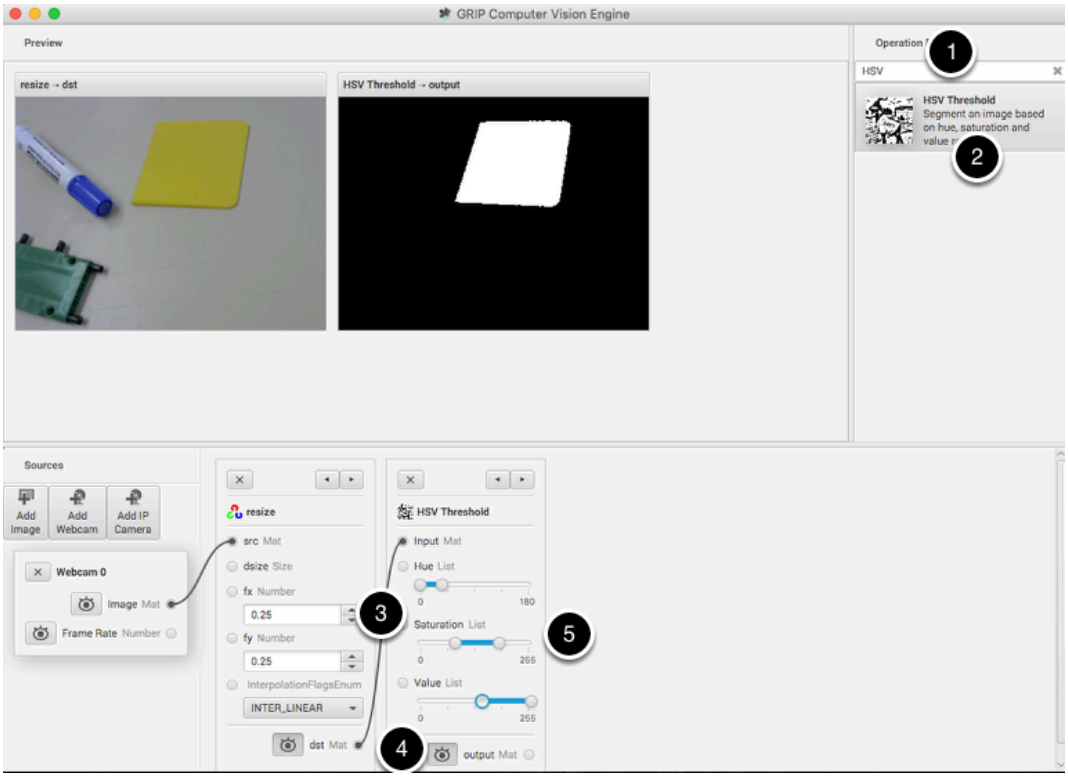
Resize the image



In this case the camera resolution is too high for our purposes, and in fact the entire image cannot even be viewed in the preview window. The "Resize" operation is clicked from the Operation Palette to add it to the end of the pipeline. To help locate the Resize operation, type "Resize" into the search box at the top of the palette. The steps are:

1. Type "Resize" into the search box on the palette
2. Click the Resize operation from the palette. It will appear in the pipeline.
3. Enter the x and y resize scale factor into the resize operation in the popeline. In this case 0.25 was chosen for both.
4. Drag from the Webcam image output mat socket to the Resize image source mat socket. A connection will be shown indicating that the camera output is being sent to the resize input.
5. Click on the destination preview button on the "Resize" operation in the pipeline. The smaller image will be displayed alongside the larger original image. You might need to scroll horizontally to see both as shown.
6. Lastly, click the Webcam source preview button since there is no reason to look at both the large image and the smaller image at the same time.

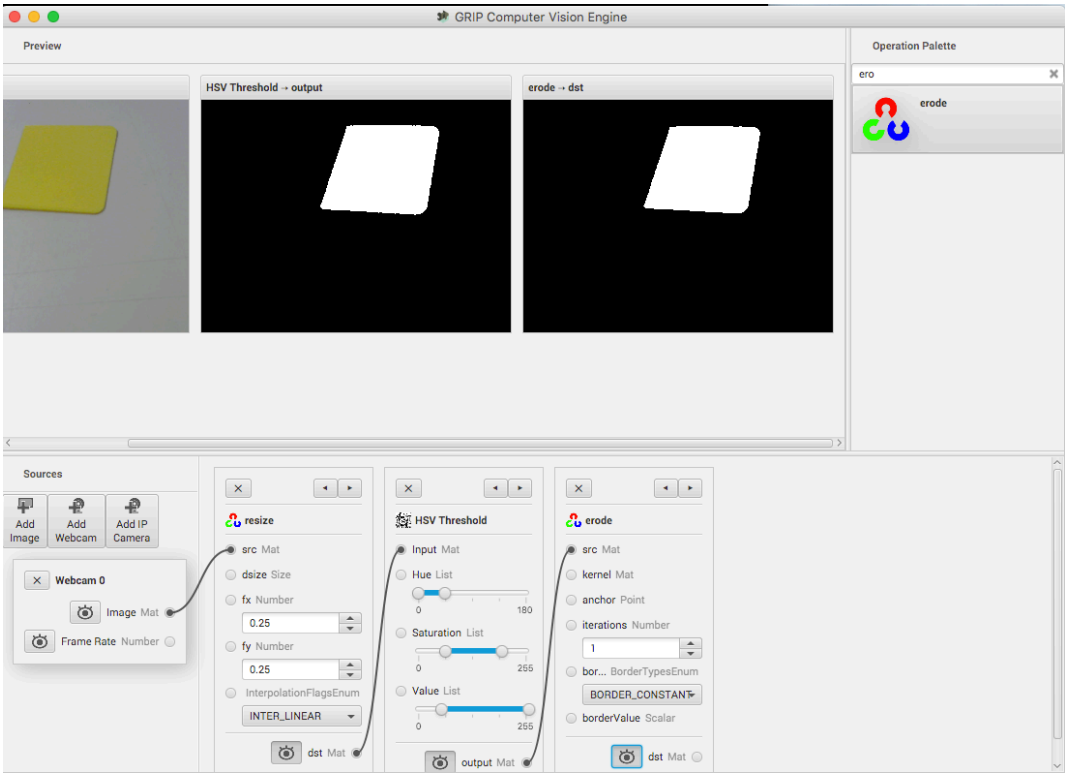
Find only the yellow parts of the image



The next step is to remove everything from the image that doesn't match the yellow color of the piece of plastic that is the object being detected. To do that a HSV Threshold operation is chosen to set upper and lower limits of HSV values to indicate which pixels should be included in the resultant binary image. Notice that the target area is white while everything that wasn't within the threshold values are shown in black. Again, as before:

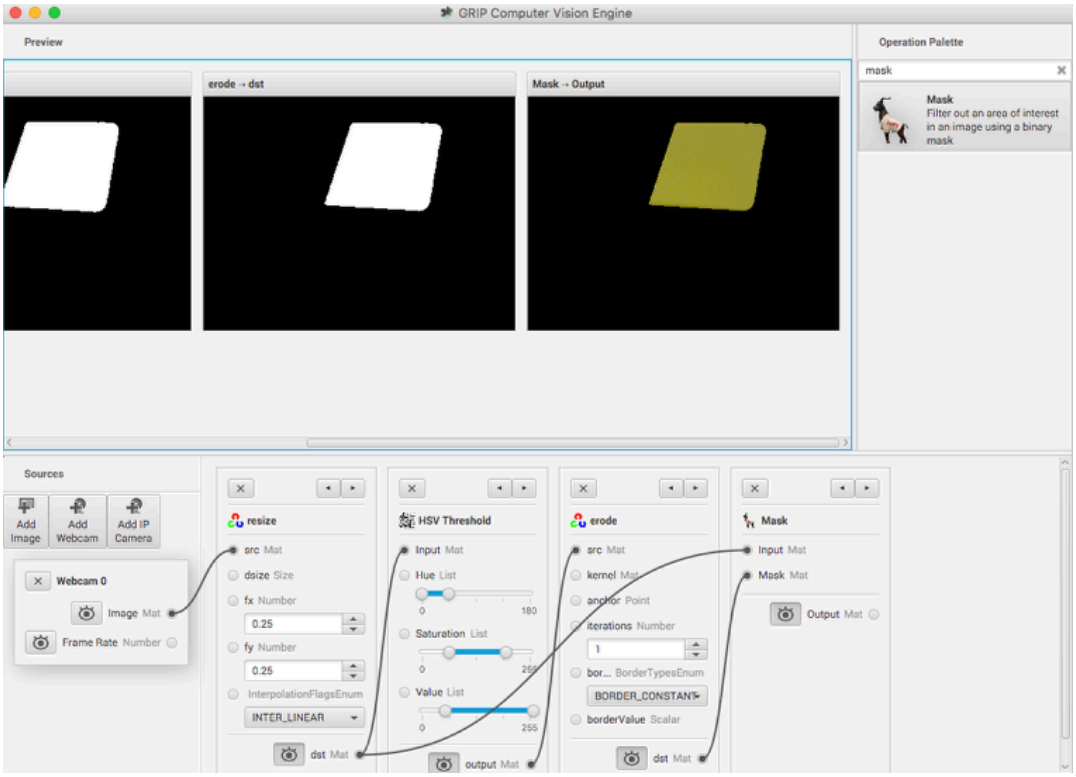
1. Type HSV into the search box to find the HSV Threshold operation.
2. Click on the operation in the palette and it will appear at the end of the pipeline.
3. Connect the dst (output) socket on the resize operation to the input of the HSV Threshold.
4. Enable the preview of the HSV Threshold operation so the result of the operation is displayed in the preview window.
5. Adjust the Hue, Saturation, and Value parameters only the target object is shown in the preview window.

Get rid of the noise and extraneous hits



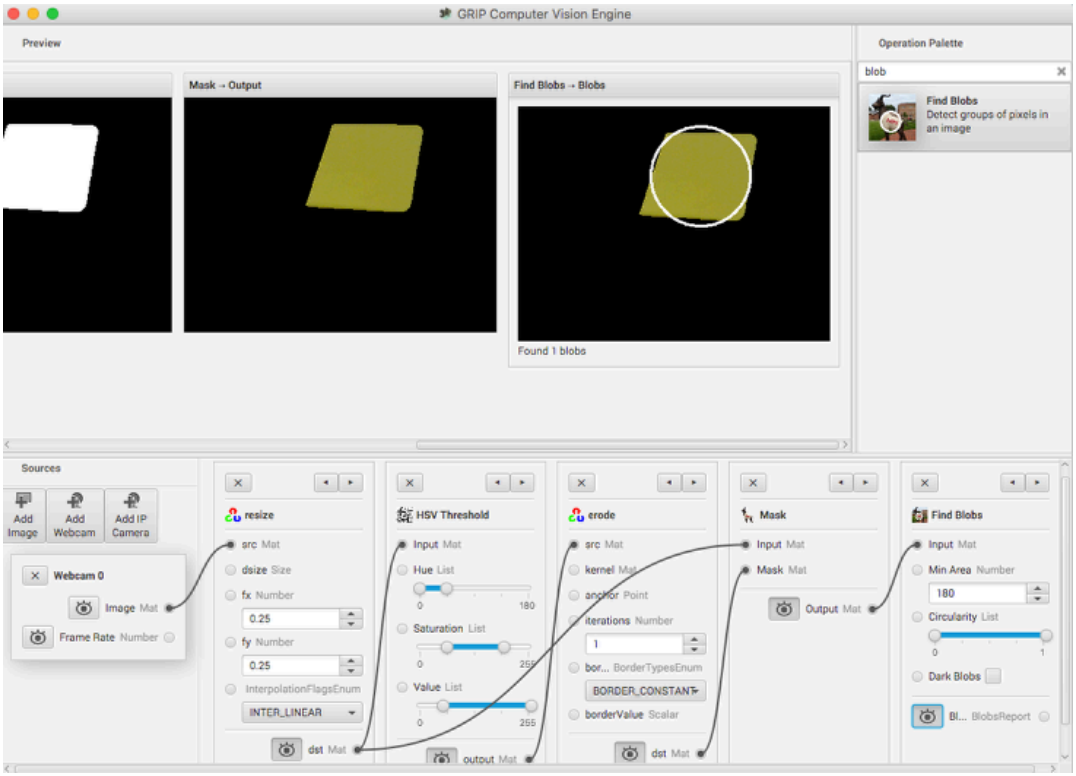
This looks pretty good so far, but sometimes there is noise from other things that couldn't quite be filtered out. To illustrate one possible technique to reduce those occasional pixels that were detected, an Erosion operation is chosen. Erosion will remove small groups of pixels that are not part of the area of interest.

Mask the just the yellow area from the original image



Here a new image is generated by taking the original image and masking (and operation) it with the the results of the erosion. This leaves just the yellow card as seen in the original image with nothing else shown. And it makes it easy to visualize exactly what was being found through the series of filters.

Find the yellow area (blob)



The last step is actually detecting the yellow card using a Blob Detector. This operation looks for a grouping of pixels that have some minimum area. In this case, the only non-black pixels are from the yellow card after the filtering is done. You can see that a circle is drawn around the detected portion of the image. In the release version of GRIP (watch for more updates between now and kickoff) you will be able to send parameters about the detected blob to your robot program using Network Tables.

Status of GRIP

As you can see from this example, it is very easy and fast to be able to do simple object recognition using GRIP. While this is a very simple example, it illustrates the basic principles of using GRIP and feature extraction in general. Over the coming weeks the project team will be posting updates to GRIP as more features are added. Currently it supports cameras (Axis ethernet camera and web cameras) and image inputs. There is no provision for output yet although Network Tables and ROS (Robot Operating System) are planned.

Vision Processing

You can either download a pre-built release of the code from the github page "Releases" section (https://github.com/WPIRoboticsProjects/GRIP) or you can clone the source repository and built it yourself. Directions on building GRIP are on the project page. There is also additional documentation on the project wiki.

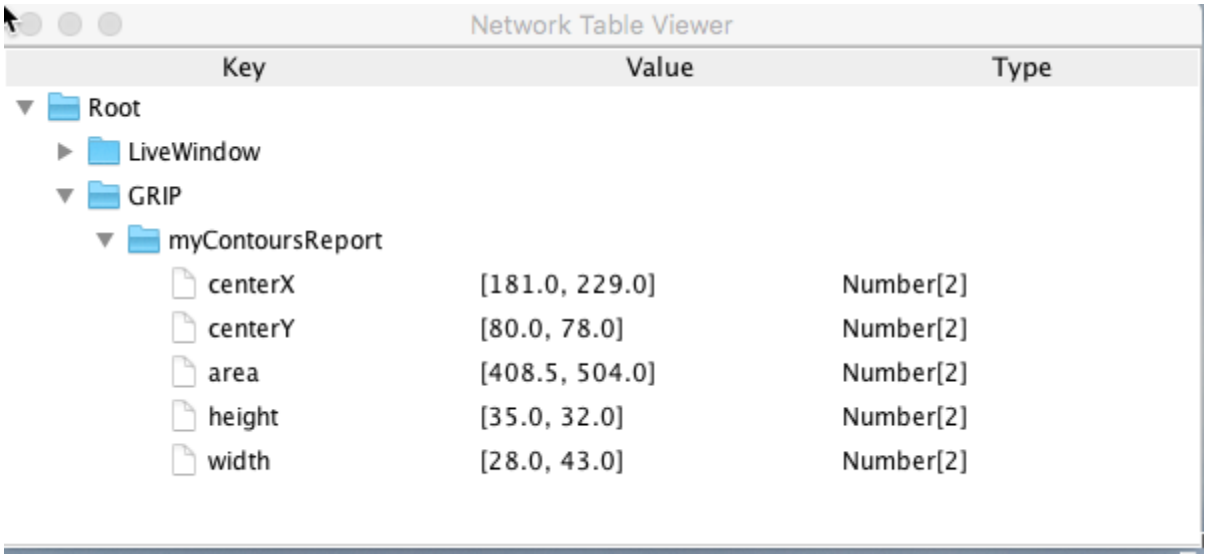
So, please play with GRiP and give us feedback here on the forum. If you find bugs, you can either post them here or as a Github project issue on the project page.

Reading array values published by NetworkTables

This article describes how to read values published by NetworkTables using a program running on the robot. This is useful when using computer vision where the images are processed on your driver station laptop and the results stored into NetworkTables possibly using a separate vision processor like a raspberry pi, or a tool on the robot like GRIP, or a python program to do the image processing.

Very often the values are for one or more areas of interest such as goals or game pieces and multiple instances are returned. In the example below, several x, y, width, height, and areas are returned by the image processor and the robot program can sort out which of the returned values are interesting through further processing.

Verify the network table keys being published



You can verify the names of the network table keys used for publishing the values by using the Network Table Viewer application. It is a java program in your user directory in the wpilib/tools

folder. The application is started by selecting the "WPILib" menu in eclipse then "OutlineViewer". In this example, wth the image processing program running (GRIP) you can see the values being put into NetworkTables.

In this case the values are stored in a table called GRIP and a sub-table called myContoursReport. You can see that the values are in brackets and there are 2 values in this case for each key. The network table key names are centerX, centerY, area, height and width.

Both of the following examples are extremely simplified programs that just illustrate the use of NetworkTables. All the code is in the robotInit() method so it's only run when the program starts up. In your programs, you would more likely get the values in code that is evaluating which direction to aim the robot in a command or a control loop during the autonomous or teleop periods.

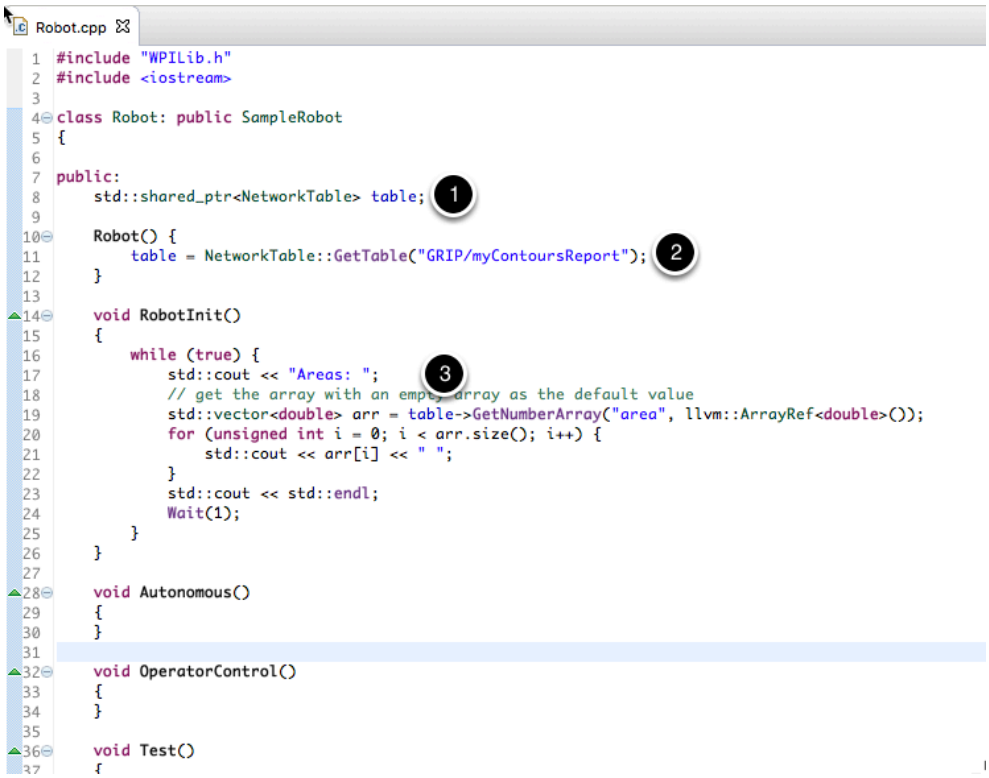
Writing a Java program to access the keys

The steps to getting the values and, in this program, printing them are:

1. Declare the table variable that will hold the instance of the subtable that have the values.
2. Initialize the subtable instance so that it can be used later for retrieving the values.

3. Read the array of values from NetworkTables. In the case of a communicating programs, it's possible that the program producing the output being read here might not yet be available when the robot program starts up. To avoid issues of the data not being ready, a default array of values is supplied. This default value will be returned if the network table key hasn't yet been published. This code just loops forever and reads values and prints them to the console.

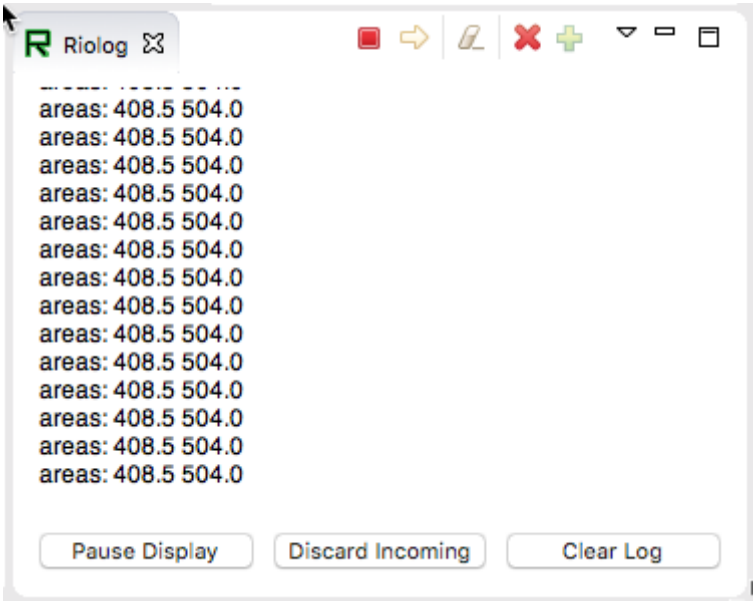
Writing a C++ program to access the keys



The steps to getting the values and, in this program, printing them are:

1. Declare the table variable that will hold the instance of the subtable that have the values. It is a shared pointer where the library takes care of allocation and deallocation automatically.
2. Initialize the subtable instance so that it can be used later for retrieving the values.
3. Read the array of values from NetworkTables. In the case of a communicating programs, it's possible that the program producing the output being read here might not yet be available when the robot program starts up. To avoid issues of the data not being ready, a default array of values is supplied. llvm::ArrayRef<double> creates this temporary array reference of zero length that would be returned if the network table key hasn't yet been published. This code just loops forever and reads values and prints them to the console.

Program output



In this case the program is only looking at the array of areas, but in a real example all the values would more likely be used. Using the Riolog in eclipse or the DriverStation log you can see the values as they are retrieved. This program is using a sample static image so they areas don't change, but you can imagine with a camera on your robot, the values would be changing constantly.

Generating Code from GRIP

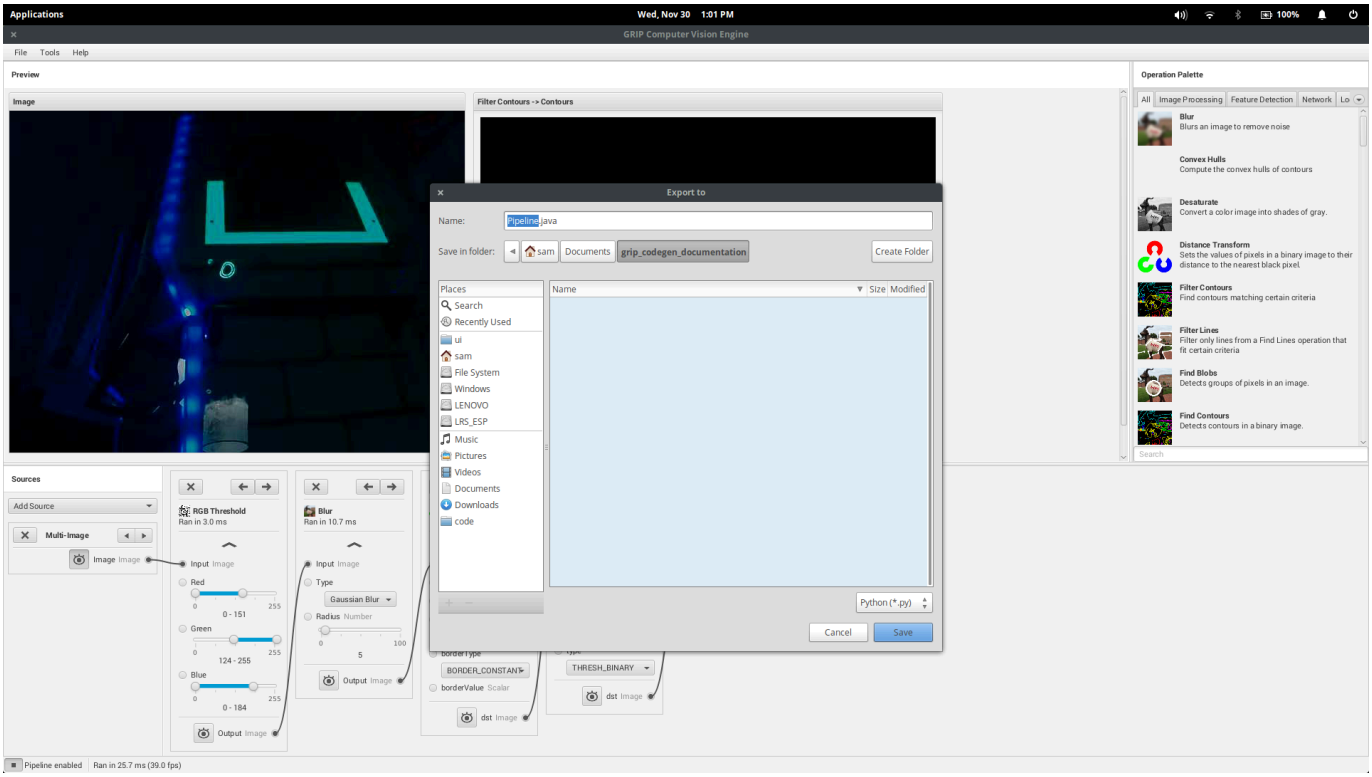
GRIP Code Generation

When running your vision algorithm on a small processor such as a roboRIO or Raspberry PI it is encouraged to run OpenCV directly on the processor without the overhead of GRIP. To facilitate this, GRIP can generate code in C++, Java, and Python for the pipeline that you have created. This generated code can be added to your robot project and called directly from your existing robot code.

Input sources such as cameras or image directories and output steps such as NetworkTables are not generated. Your code must supply images as OpenCV mats. On the roboRIO, the CameraServer class supplies images in that format. For getting results you can just use generated getter methods for retrieving the resultant values such as contour x and y values.

Generating Code

To generate code, go to `Tools > Generate Code`. This will bring up a save dialog that lets you create a C++, Java, or Python class that performs the steps in the GRIP pipeline.



If generating code to be used in a pre-existing project, choose a relevant directory to save the pipeline to.

C++ Users: the pipeline class is split into a header and implementation file

Java Users: the generated class lacks a package declaration, so a declaration should be added to match the directory where the file was saved.

Python Users: the module name will be identical to the class, so the import statement will be something like `from Pipeline import Pipeline`

Structure of the Generated Code

```
Pipeline:
  // Process -- this will run the pipeline
```

```
process(Mat source)

// Output accessors
getFooOutput()
getBar0Output()
getBar1Output()
...
```

Running the Pipeline

To run the Pipeline, call the process method with the sources (webcams, IP camera, image file, etc) as arguments. This will expose the outputs of every operation in the pipeline with the `getFooOutput` methods.

Getting the Results

Users are able to the outputs of every step in the pipeline. The outputs of these operations would be accessible through their respective accessors. For example:

Operation	Java/C++ getter	Python variable
RGB Threshold	`getRgbThresholdOutput`	`rgb_threshold_output`
Blur	`getBlurOutput`	`blur_output`
CV Erode	`getCvErodeOutput`	`cv_erode_output`
Find Contours	`getFindContoursOutput`	`find_contours_output`
Filter Contours	`getFilterContoursOutput`	`filter_contours_output`

If an operation appears multiple times in the pipeline, the accessors for those operations have the number of that operation:

Operation	Which appearance	Accessor
Blur	First	`getBlur0Output`

Operation	Which appearance	Accessor
Blur	Second	`getBlur1Output`
Blur	Third	`getBlur2Output`

Using Generated Code in a Robot Program

GRIP generates a class that can be added to an FRC program that runs on a roboRIO and without a lot of additional code, drive the robot based on the output.

Included here is a complete sample program that uses a GRIP pipeline that drives a robot towards a piece of retroreflective material.

This program is designed to illustrate how the vision code works and does not necessarily represent the best technique for writing your robot program. When writing your own program be aware of the following considerations:

1. **Using the camera output for steering the robot could be problematic**. The camera code in this example that captures and processes images runs at a much slower rate that is desirable for a control loop for steering the robot. A better, and only slightly more complex solution, is to get headings from the camera and it's processing rate, then have a much faster control loop steering to those headings using a gyro sensor.
2. **Keep the vision code in the class that wraps the pipeline**. A better way of writing object oriented code is to subclass or instantiate the generated pipeline class and process the OpenCV results there rather than in the robot program. In this example, the robot code extracts the direction to drive by manipulating the resultant OpenCV contours. By having the OpenCV code exposed throughout the robot program it makes it difficult to change the vision algorithm should you have a better one.

Iterative program definitions

```
package org.usfirst.frc.team190.robot;

import org.usfirst.frc.team190.grip.MyVisionPipeline;

import org.opencv.core.Rect;
import org.opencv.imgproc.Imgproc;

import edu.wpi.cscore.UsbCamera;
import edu.wpi.first.wpilibj.CameraServer;
import edu.wpi.first.wpilibj.IterativeRobot;
import edu.wpi.first.wpilibj.RobotDrive;
import edu.wpi.first.wpilibj.vision.VisionRunner;
```

```
import edu.wpi.first.wpilibj.vision.VisionThread;

public class Robot extends IterativeRobot {

        private static final int IMG_WIDTH = 320;
        private static final int IMG_HEIGHT = 240;

        private VisionThread visionThread;
        private double centerX = 0.0;
        private RobotDrive drive;

        private final Object imgLock = new Object();
```

In this first part of the program you can see all the import statements for the WPILib classes used for this program.

- The **image width and height** are defined as 320x240 pixels.
- The **VisionThread** is a WPILib class makes it easy to do your camera processing in a separate thread from the rest of the robot program.
- **centerX** value will be the computed center X value of the detected target.
- **RobotDrive** encapsulates the 4 motors on this robot and allows simplified driving.
- **imgLock** is a variable to synchronize access to the data being simultaneously updated with each image acquisition pass and the code that's processing the coordinates and steering the robot.

```
@Override
public void robotInit() {
    UsbCamera camera = CameraServer.getInstance().startAutomaticCapture();
    camera.setResolution(IMG_WIDTH, IMG_HEIGHT);

    visionThread = new VisionThread(camera, new MyVisionPipeline(), pipeline -> {
        if (!pipeline.filterContoursOutput().isEmpty()) {
            Rect r = Imgproc.boundingRect(pipeline.filterContoursOutput().get(0));
            synchronized (imgLock) {
                centerX = r.x + (r.width / 2);
            }
        }
    });
    visionThread.start();
```

```
    drive = new RobotDrive(1, 2);
}
```

The **robotInit()** method is called once when the program starts up. It creates a **CameraServer** instance that begins capturing images at the requested resolution (IMG_WIDTH by IMG_HEIGHT).

Next an instance of the class **VisionThread** is created. VisionThread begins capturing images from the camera asynchronously in a separate thread. After processing each image, the pipeline computed **bounding box** around the target is retrieved and it's **center X** value is computed. This centerX value will be the x pixel value of the center of the rectangle in the image.

The VisionThread also takes a **VisionPipeline** instance (here, we have a subclass **MyVisionPipeline** generated by GRIP) as well as a callback that we use to handle the output of the pipeline. In this example, the pipeline outputs a list of contours (outlines of areas in an image) that mark goals or targets of some kind. The callback finds the bounding box of the first contour in order to find its center, then saves that value in the variable centerX. Note the **synchronized** block around the assignment: this makes sure the main robot thread will always have the most up-to-date value of the variable, as long as it also uses synchronized blocks to read the variable.

```
@Override
public void autonomousPeriodic() {
    double centerX;
    synchronized (imgLock) {
        centerX = this.centerX;
    }
    double turn = centerX - (IMG_WIDTH / 2);
    drive.arcadeDrive(-0.6, turn * 0.005);
}
```

This, the final part of the program, is called repeatedly during the **autonomous period** of the match. It gets the **centerX** pixel value of the target and **subtracts half the image width** to change it to a value that is **zero when the rectangle is centered** in the image and **positive or negative when the target center is on the left or right side of the frame**. That value is used to steer the robot towards the target.

Note the **synchronized** block at the beginning. This takes a snapshot of the most recent *centerX* value found by the VisionThread.

Using GRIP with a Kangaroo Computer

A recently available computer called the Kangaroo looks like a great platform for running GRIP on FRC robots. Some of the specs for this processor include:

- Quad core 1.4Ghz Atom processor
- HDMI port
- 2 USB ports (1 USB2 and 1 USB3)
- 2GB RAM
- 32GB Flash
- Flash card slot
- WiFi
- Battery with 4 hours running time
- Power supply
- Windows 10
- and a fingerprint reader

All this is only $99 or $90 for a student or faculty member from Microsoft.

The advantage of this setup is that it offloads the roboRIO from doing image processing and it is a normal Windows system so all of our software should work without modification. Be sure to read the caveats at the end of this page before jumping in.

More detailed instructions for using a Kangaroo for running GRIP can be found in the following PDF document created by Scott Taylor and FRC 1735. His explanation goes beyond what is shown here, detailing how to get the GRIP program to auto-start on boot and many other details.

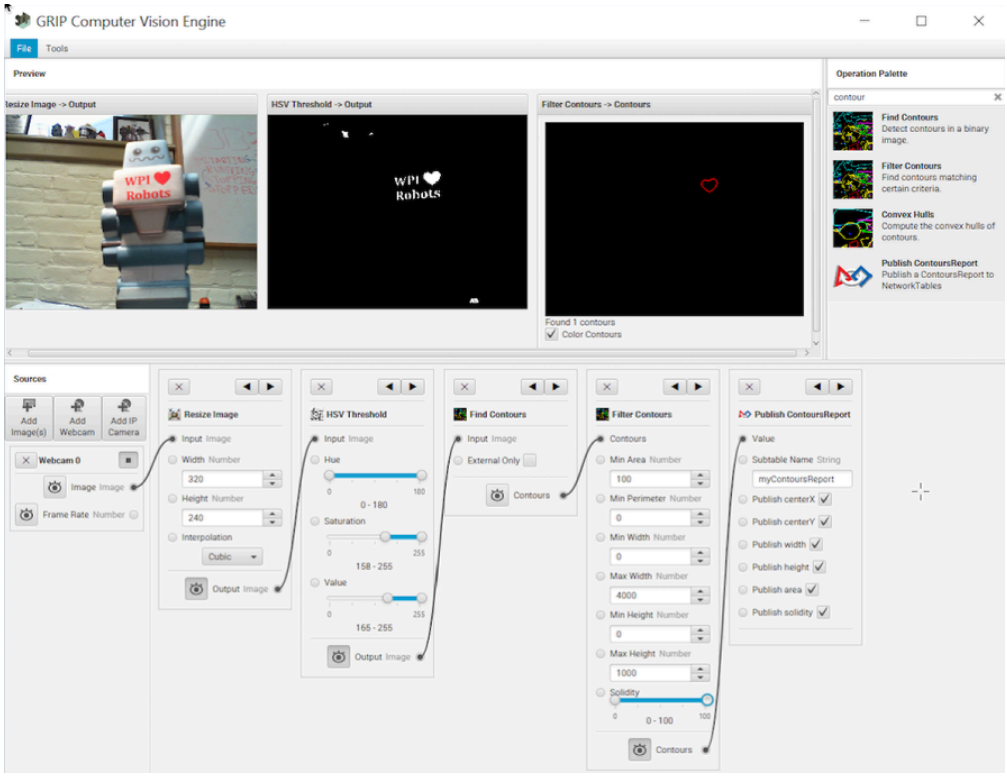
Grip_plus_Kangaroo.pdf

Setup

The nice thing about this setup is that you just need to plug in a monitor, keyboard, mouse and (in this case) the Microsoft web camera and you are good to go with programming the GRIP pipeline. When you are finished, disconnect the keyboard, mouse and monitor and put the Kangaroo on your robot. You will need to disable the WiFi on the Kangaroo and connect it to the robot with a USB ethernet dongle to the extra ethernet port on the robot radio.

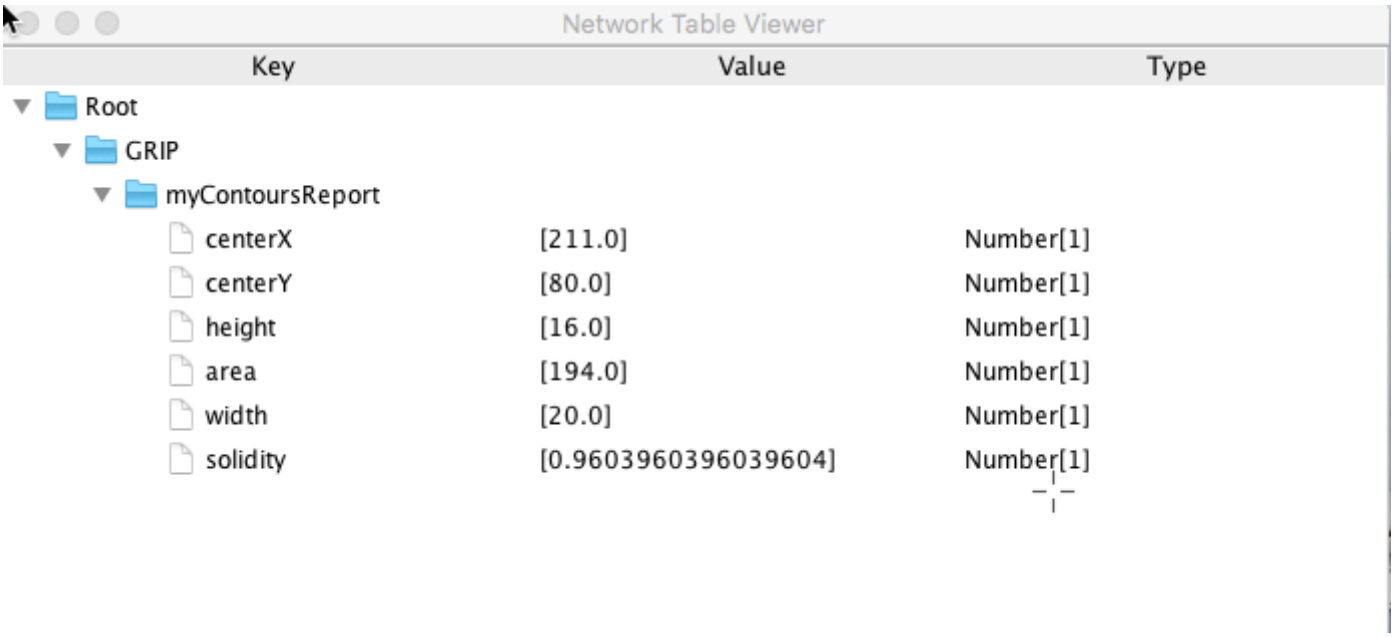
In this example you can see the Kangaroo computer (1) connected to a USB hub (3), keyboard, and an HDMI monitor for programming. The USB hub is connected to the camera and mouse.

Sample GRIP program



Attached is the sample program running on the Kangaroo detecting the red heart on the little foam robot in the image (left panel). It is doing a HSV threshold to only get that red color then finding contours, and then filtering the contours using the size and solidity. At the end of the pipeline, the values are being published to NetworkTables.

Viewing Contours Report in NetworkTables



This is the output from the OutlineViewer (<username>/WPILib/tools/OutlineViewer.jar), running on a different computer as a server (since there is no roboRIO on the network in this example) and the values being reported back for the single contour that the program detected that met the requirements of the Filter Contours operation.

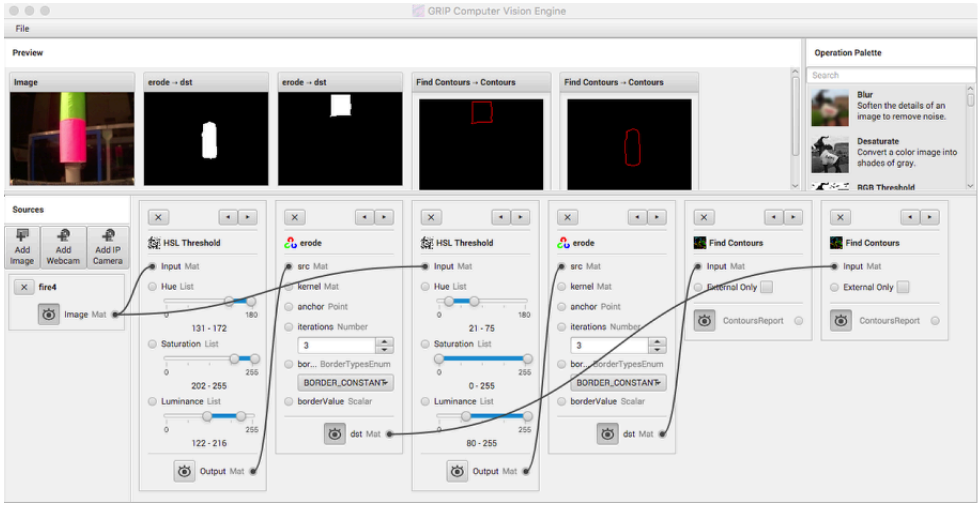
Considerations

The Kangaroo runs Windows 10, so care must to be taken to make sure GRIP will keep running on the robot during a match or testing. For example, it should not try to do a Windows Update, Virus scan refresh, go to sleep, ect. Once configured, it has the advantage of being a normal Intel Architecture and should give predictable performance since it is running only one application.

Processing Images from the 2009 FRC Game

In the FRC 2009 game, Lunacy, robots were required to put orbit balls into the trailers of opponents robots. To differentiate robots on each of the two alliances, a "flag" was attached to the center of the goal. The flag was a cylinder that was green on top and red on the bottom or red on top with green on the bottom. During the autonomous period, robots could look for opponent robots and shoot balls into their trailer using the vision targets.

Using the Find Contours operation to find targets



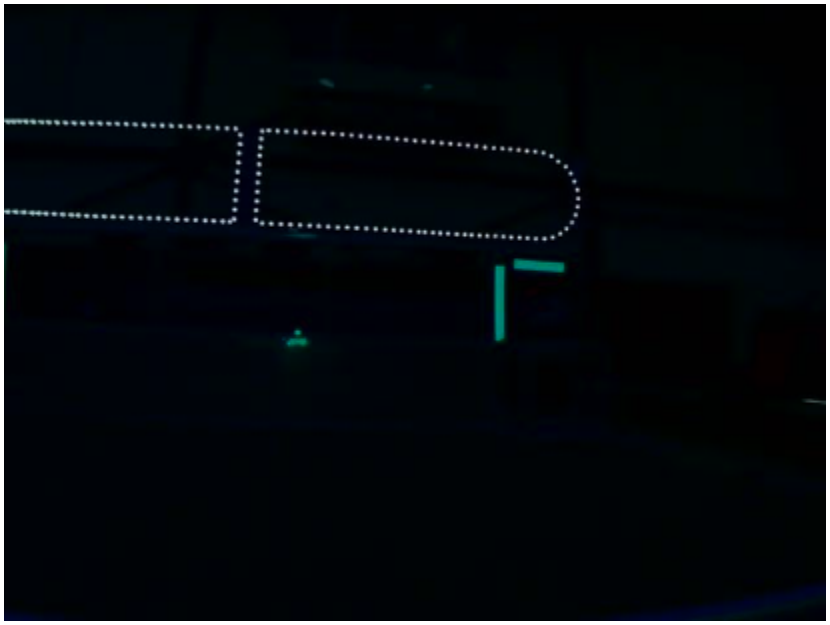
In this example the image, in this case a file, you can see the green and red halves of the vision target. The strategy is to:

1. Look for objects that are either green or red using two different HSL Threshold operations, one for each color. Setting the appropriate parameters on the threshold allow it to detect the two halves of the target.
2. Erode the images to get rid of any very small objects that slipped through
3. Find contours in each of the resultant binary images
4. Send the red and green contours lists to the robot, It will look for objects with the correct aspect ratio, proximity, and orientation with respect to each other. From this the robot program can determine which sets are targets.

Processing Images from the 2014 FRC Game

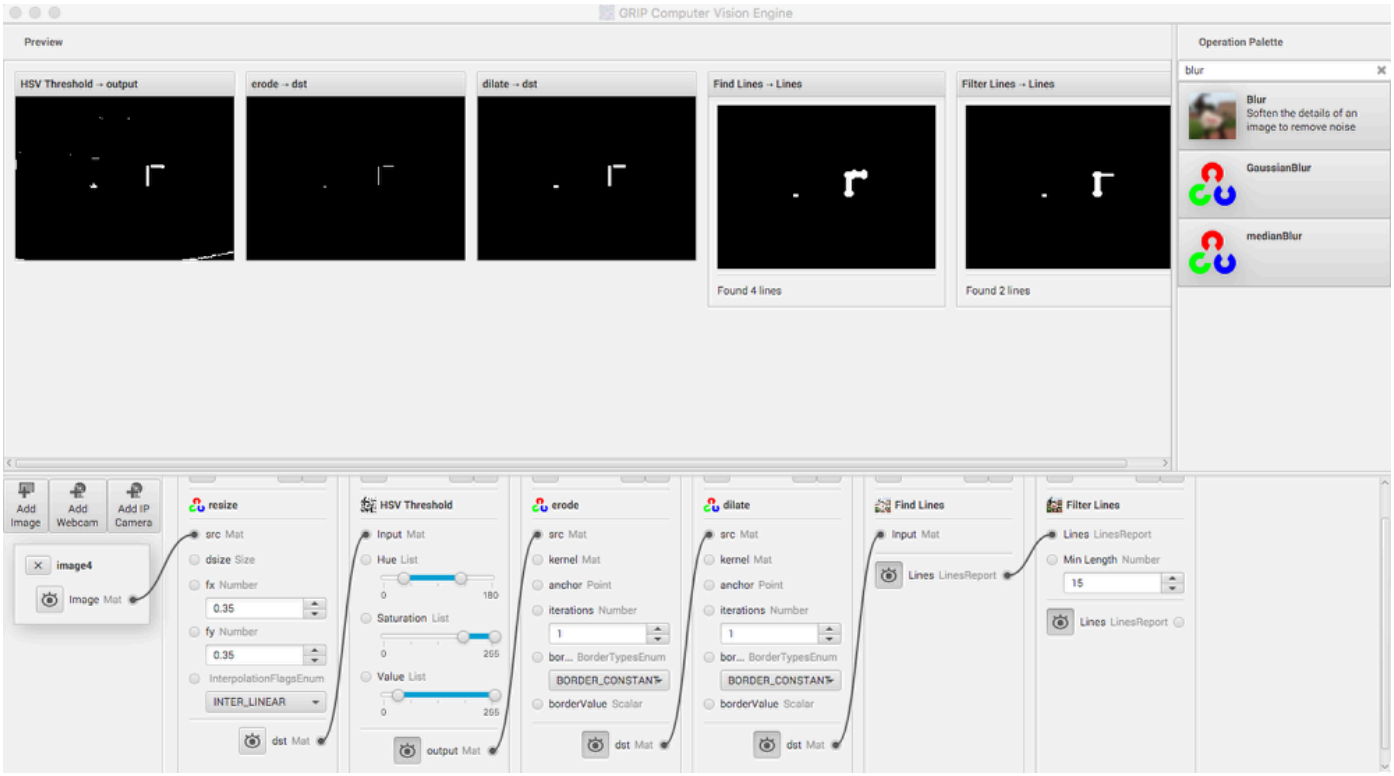
This is a quick sample using GRIP to process a single image from the FRC 2014 Aerial Assist. Keep in mind that this is just a single image (all that I could find in a hurry) so it is not necessarily a particularly robust algorithm. You should really take many pictures from different distances, angles, and lighting conditions to ensure that your algorithm will perform well in all those cases.

The sample image (I'll get more later)



The original image is shown here and represents the hot goal being indicated by the retroreflective tape on the moveable board. When the goal is hot, the horizontal piece of tape is facing the robot. When it is not hot, the board that the tape is attached is flipped up so it doesn't reflect and that horizontal line would not be present.

The actual algorithm depicted in GRIP

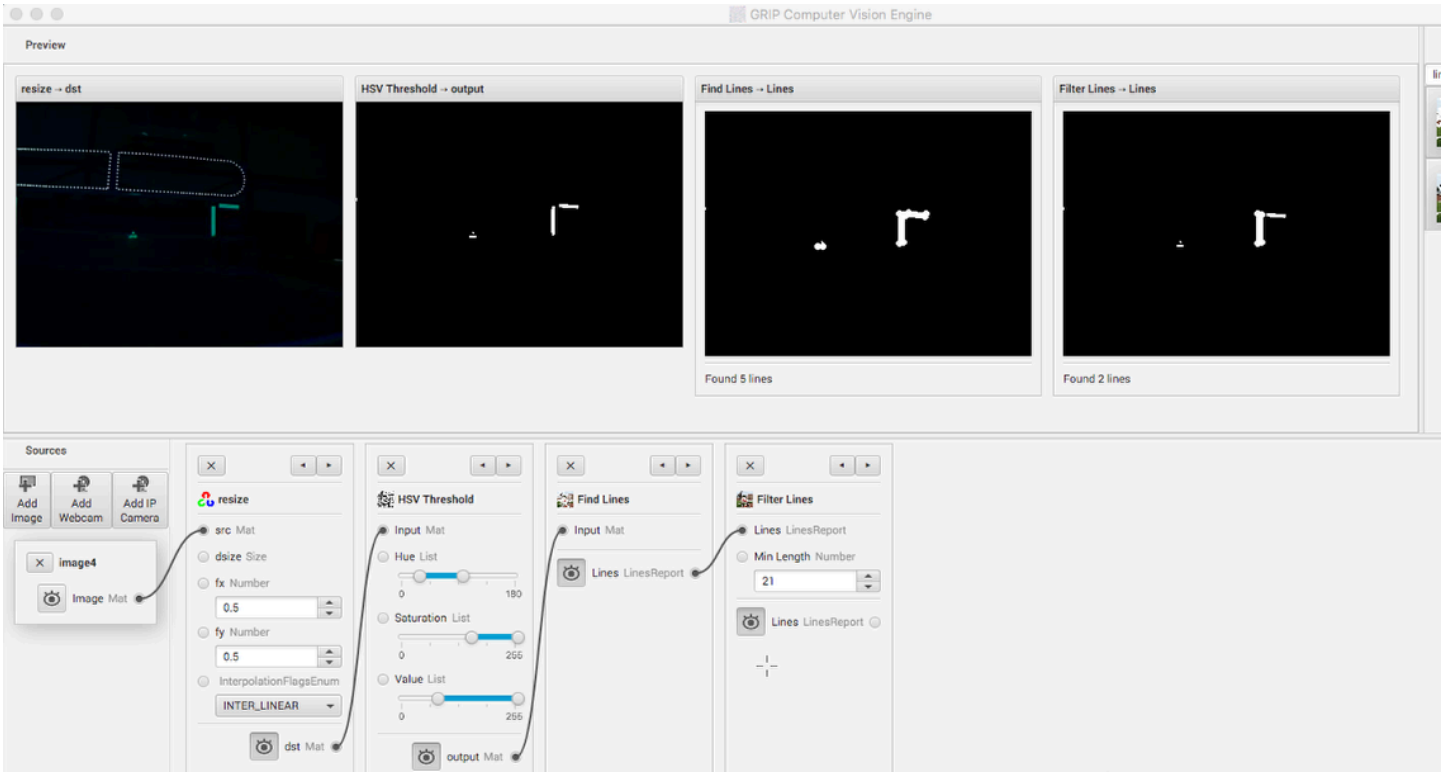


This shows the pipeline as it was developed in GRIP. It took only a few minutes to do this and was the first attempt:

1. The image was resized by 0.35 to make it fit better for this presenation.
2. A HSV threshold operation is done to filter out everything but the blue-green lines that are shown in the initial image.
3. An erosion operation was done to reduce a bunch of the small artifacts that the threshold was not able to filter. This also reduced the line thickness.
4. A dilation is done to make the lines a little thicker in hopes of better detecting them, although this might not have been necessary.
5. Then a line detection operation found 4 lines in this case, the few artifacts registered as lines as well as the actual lines.
6. The smaller artifacts were filtered using the Filter Lines operation as shown in the final step. Filter lines allows the specification of the minimum line size so the extra mis-detected lines are removed from the list.

In the final release, the output of the filter lines operation can be sent to network tables to make it accessible to the robot program to decided whether or not to take the shot.

A simpler approach

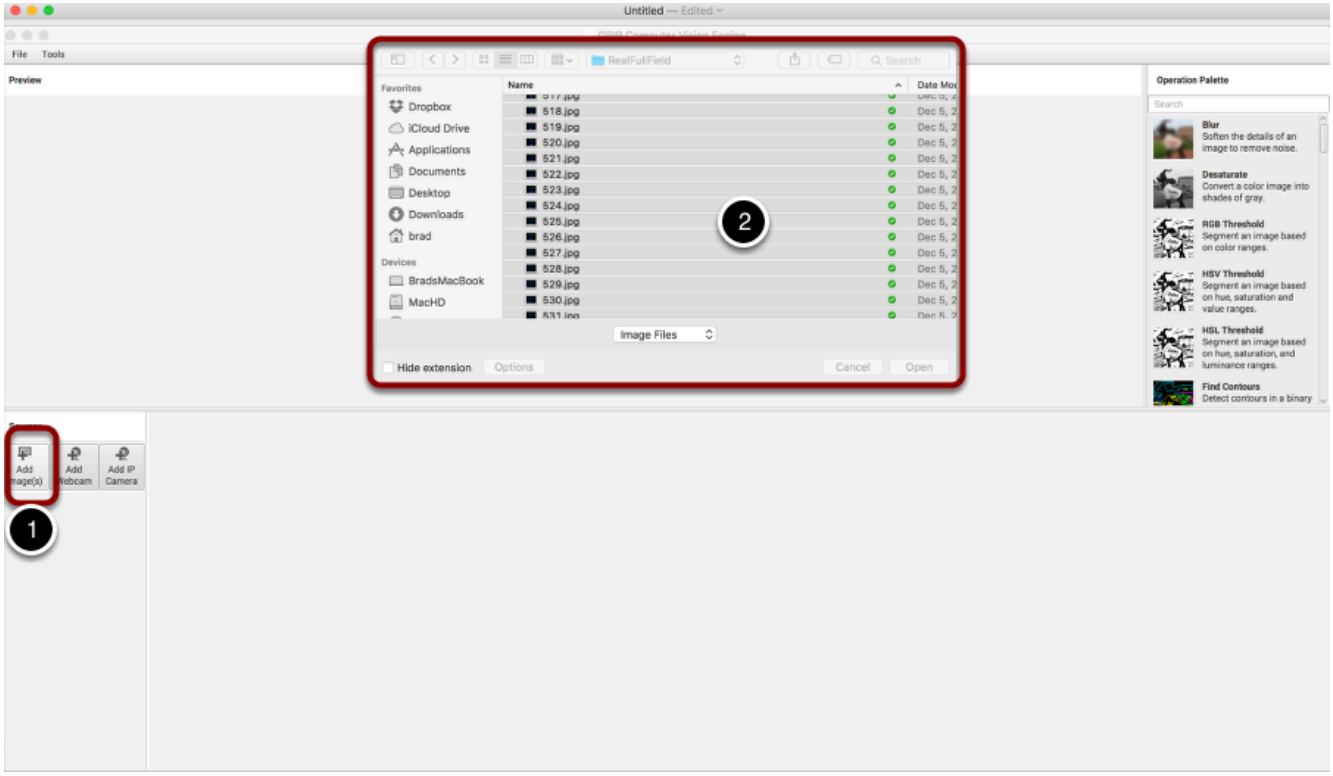


It seemed like the erosion and dilation might not be necessary since there was very little noise showing up in the images. So for this image (again - in real life test this with more images), the same algorithm was attempted successfully without those extra steps. In this case the algorithm is shorter and would run more quickly, but might not be as robust as the one with the potentially better filtering. And again, it also found 2 lines representing the hot goal targets.

Processing Images from the 2016 FRC Game

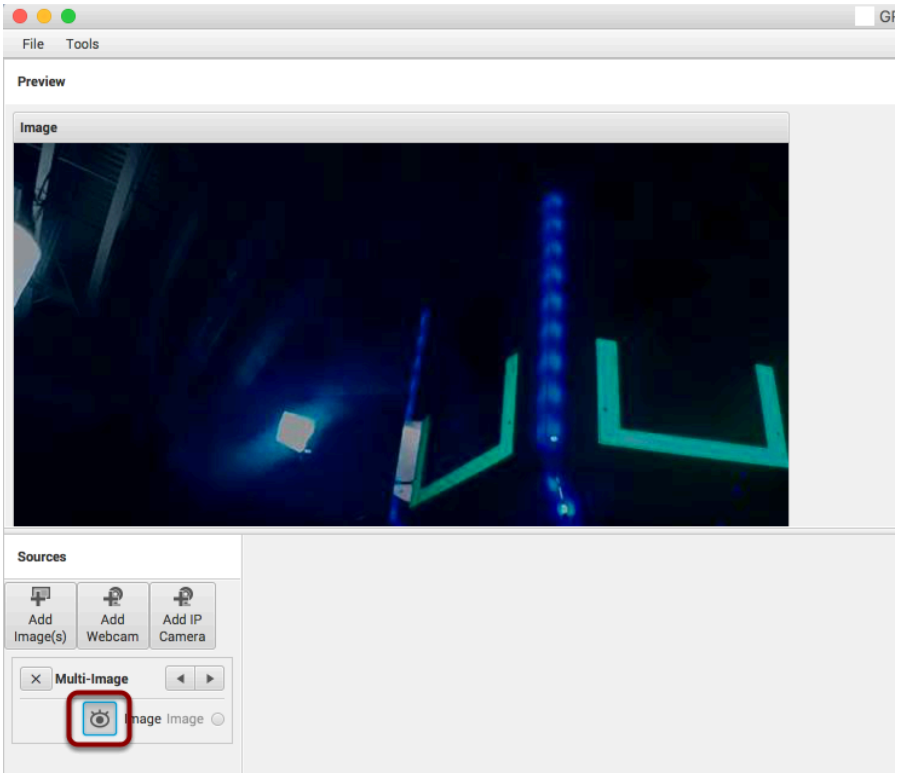
GRIP can be used to locate goals for the FIRST Stronghold by using a series of thresholding and finding contours. This page describes the algorithm that was used. You should download the set of sample images from the WPILib project on http://usfirst.collab.net. The idea is to load up all the images into a multi image source and test your algorithms by cycling through the pictures.

Select all the vision samples as multi image source



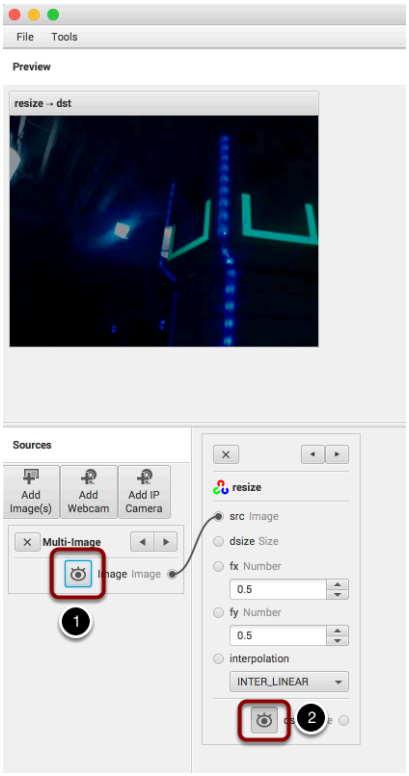
Click the Add images button to create the multi-image source, then select all the images from the directory where they were unzipped.

Images are too big (maybe)



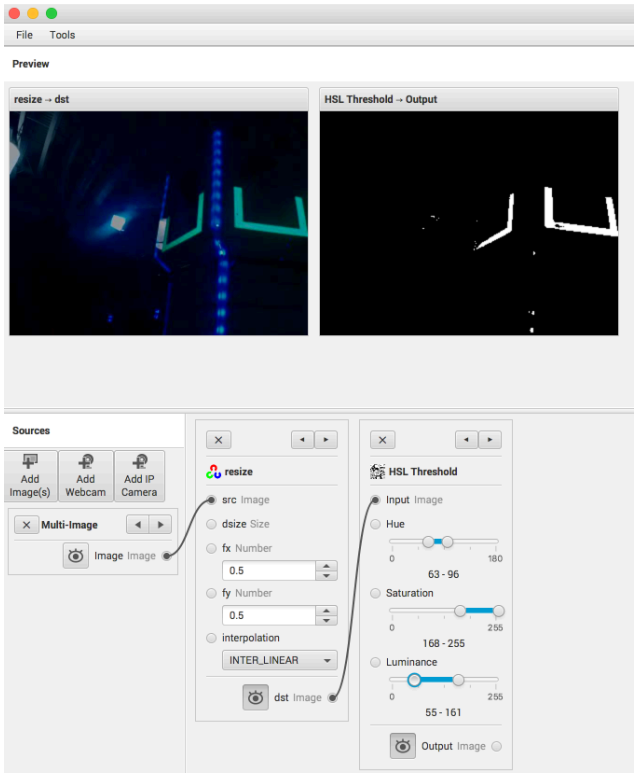
I decided that in this example the images were too big and would work just as well if they were resized. That may or may not but true, but it made the pictures small enough to capture easily for this set of examples. You can decide based on your processor and experience whether you want to resize them or not. The images are previewed by pressing the preview button shown above.

Resized images



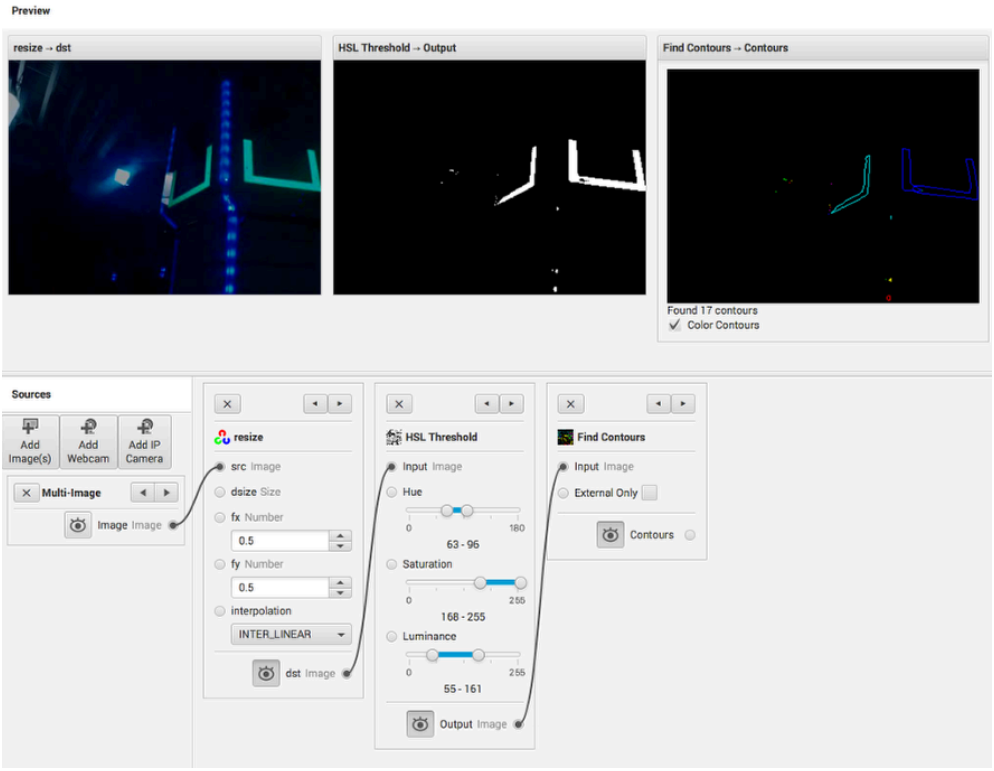
Changing the scale factor by 0.5 reduces the images to 1/4 the number of pixels and makes processing go much faster, and as said before, fits much better for this tutorial. In this example, the full size image preview is turned off and the smaller preview image is turned on. And, the output of the image source is sent to the resize operation.

Use HSV Threshold operation to detect the targets



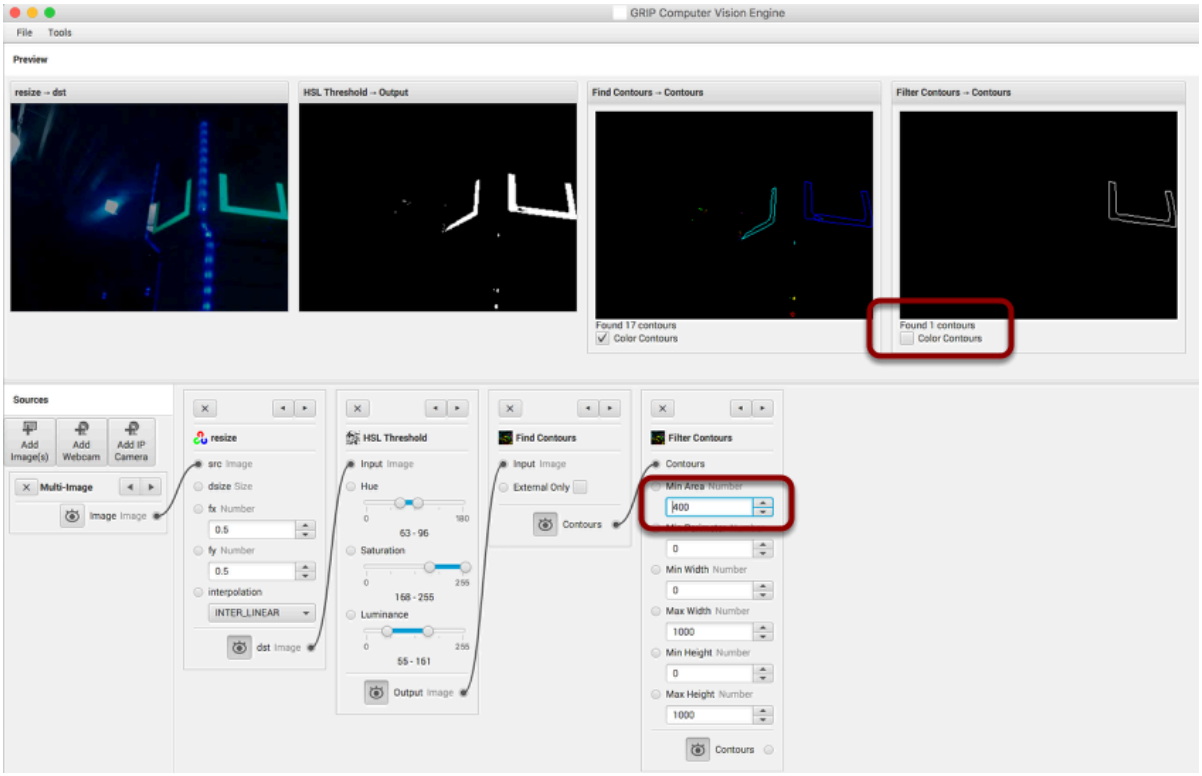
Since the targets were illuminated with a constant color ringlight, it's possible to do a threshold operation to detect only the pixels that are that color. What's left is a binary image, that is an image with 1's where the color matched the parameters and 0's where it didn't. By narrowing the H, S, and L parameters, only the target is included in the output image. You can see that the there are some small artifacts left over, but mostly, it is just the vision targets in the frame.

Find contours in the image



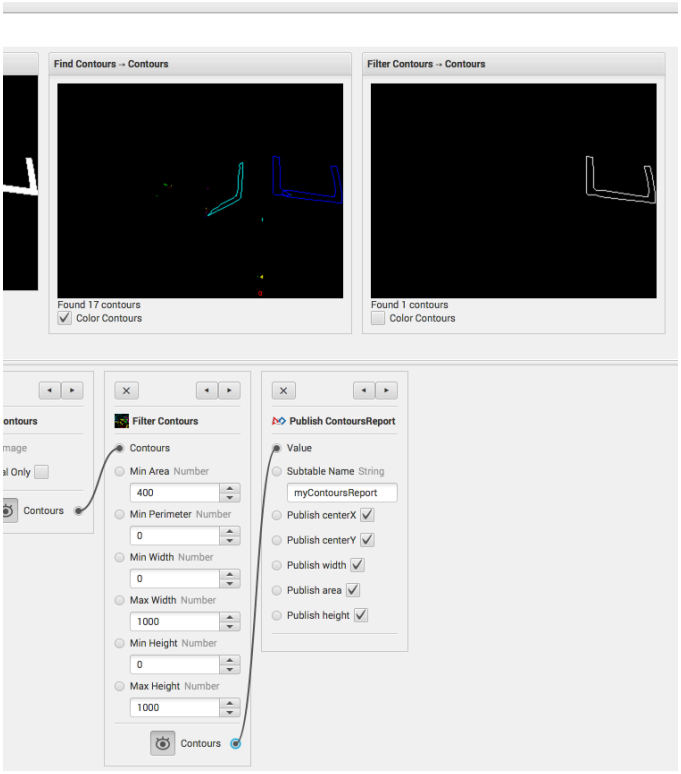
Now we can find contours in the image - that is looked for connected sets of pixels and the surrounding box. Notice that 17 contours were found that included all the smaller artifacts in the image. It's helpful to select the Color Contours checkbox to more distinctly color them to see exactly what was detected.

Filtering the discovered contours



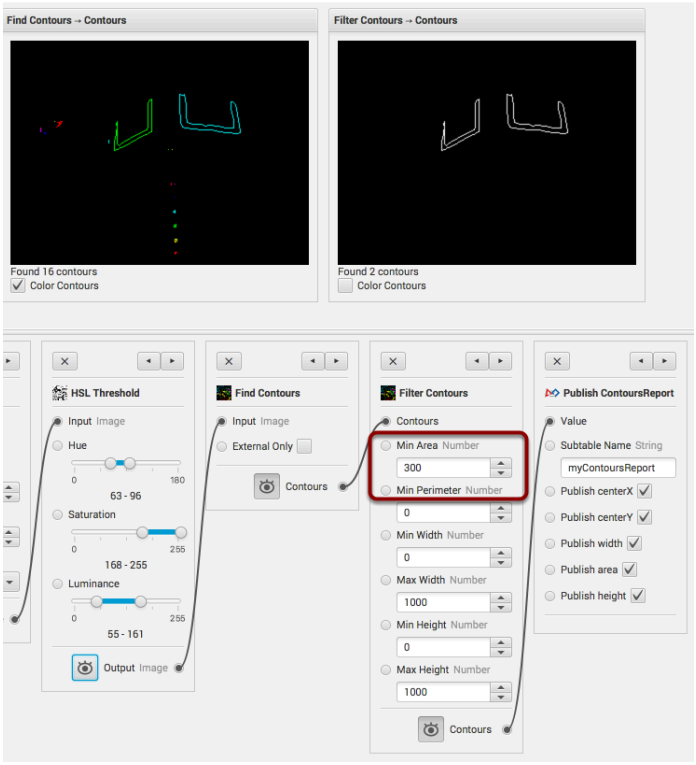
Filtering the contours with a minimum area of 400 pixels gets down to a single contour found. Setting these filters can reduce the number of artifacts that the robot program needs to deal with.

Publishing the results to network tables



The Publish ContoursReport step will publish the selected values to network tables. A good way of determining exactly what's published is to run the Outline Viewer program that's one of the tools in the <username>/wpilib/tools directory after you installed the eclipse plugins.

Changing filter values



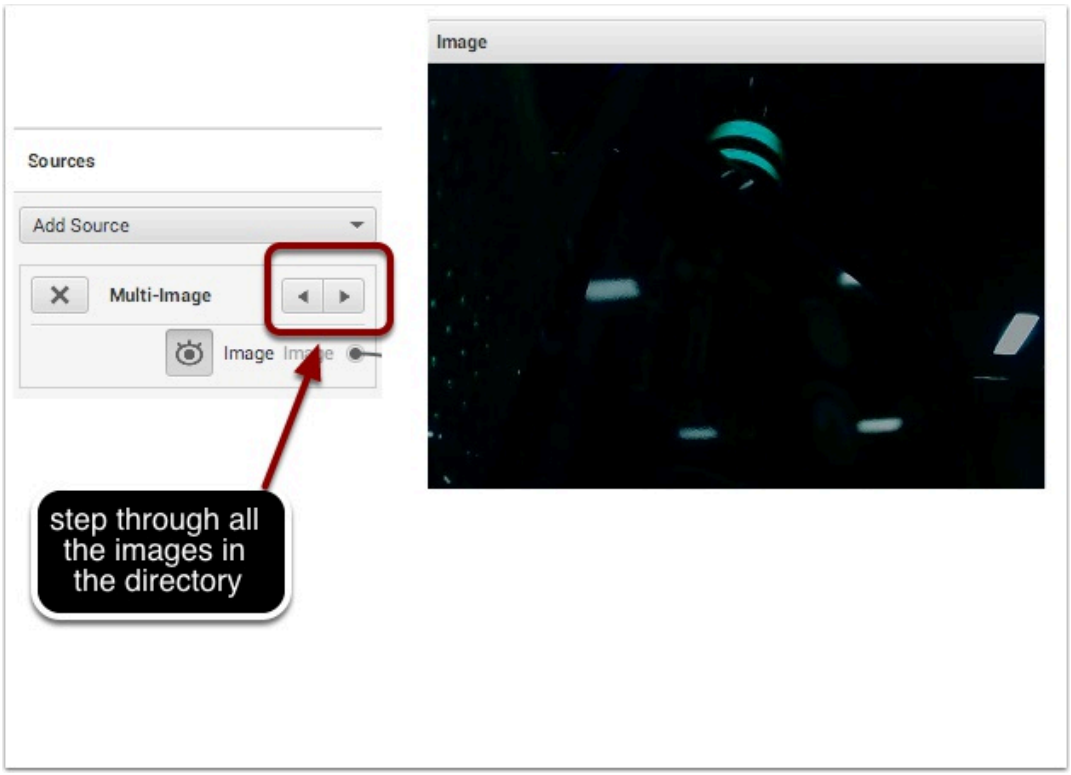
Reducing the minimum area from 400 to 300 got both sides of the tower targets, but for other images it might let in artifacts that are undesirable.

Allowing other colors

In some of the images, the two faces were slightly different colors. One way of handling this case is to open up the parameters enough to accept both, but that has the downside of accepting everything in between. Another way to solve this is to have two separate filters, one for the first color and one for the second color. Then two binary images can be created and they can be ORd together to make a single composite binary image that includes both. That is shown in this step. The final pipeline is shown here. A dilate operation was added to improve the quality of the results which adds additional pixels around broken images to try to make them appear as a single object rather than multiple objects.

Processing images from the 2017 FRC Game

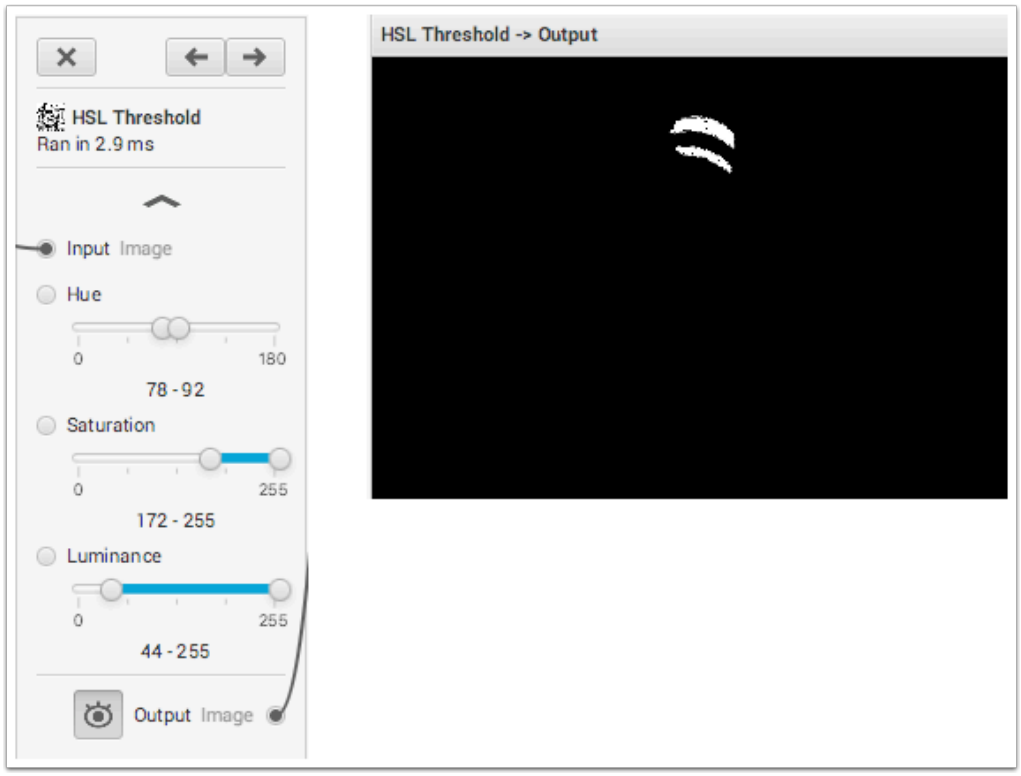
One stragegy for detecting the retroreflective vision targets is to use an LED ringlight around the lens on your camera and setting the camera brightness so only the target is visible. Then you can filter for the LED ringlight color and find contours (connected sets of points) in the image. There are a set of images and the GRIP save file provided by FIRST that were taking this way here. By starting the GRIP pipeline with a multi-image (files) set of sources you can use the left-right arrows to cycle through all the images in a set and make sure that your code is properly tuned to detect just the vision targets.



In this image you can see that everything that is not brightly lit is black and the only things visible are the reflected vision targets and some lights that were in the room. But the target is distinctly green (the LED color) and can be differentiated from the ceiling lights.

Filter out everything but the vision targets

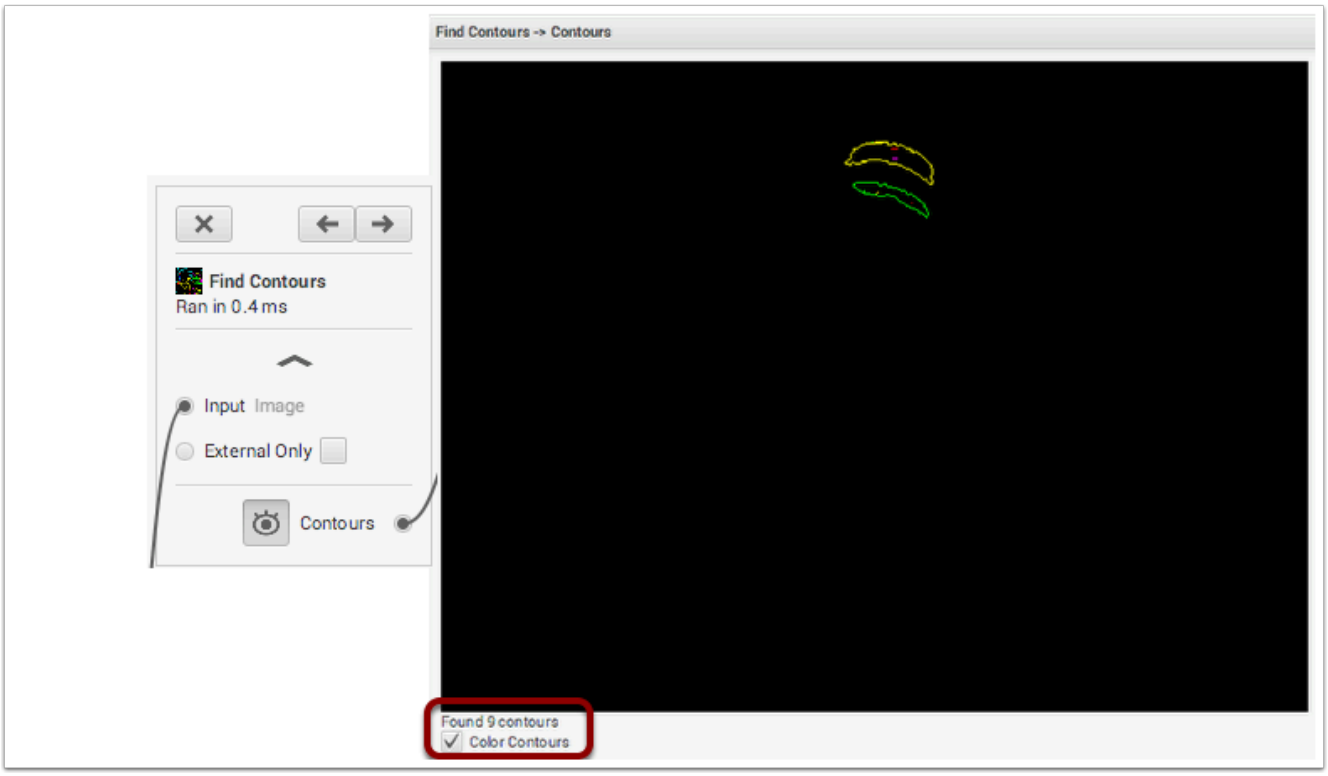
Using an HSL Threshold operation you can filter out almost everything except the vision targets as shown in the following image.



The HSL Threshold lets you specify an accepted range of Hue, Saturation, and Luminance values that will make up the pixels in the resultant binary (one bit per pixel) image.

Finding Contours in the image

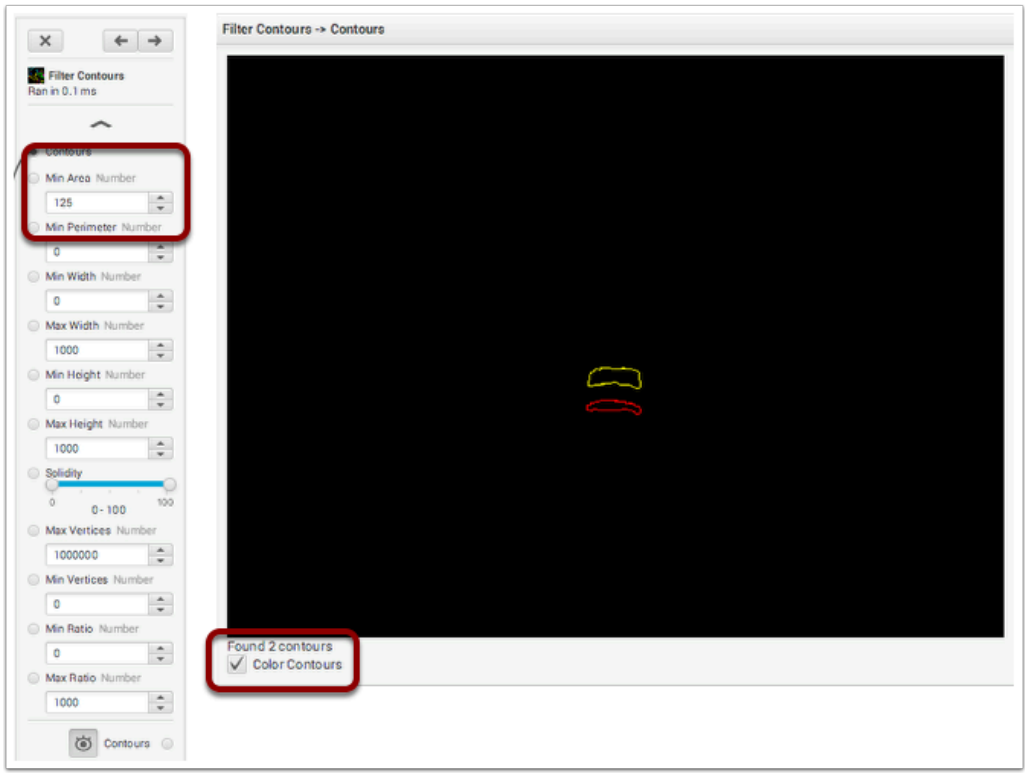
The next step is to identify the selected areas that make up the targets. This is done using a Find Contours operation. Contours are contiguous areas of pixels that are lit in the image.



Notice that 9 contours were found in the image because of noise tha other other very small areas that were selected. The idea is to come up criteria to reduce th number of false positives.

Filtering out noise

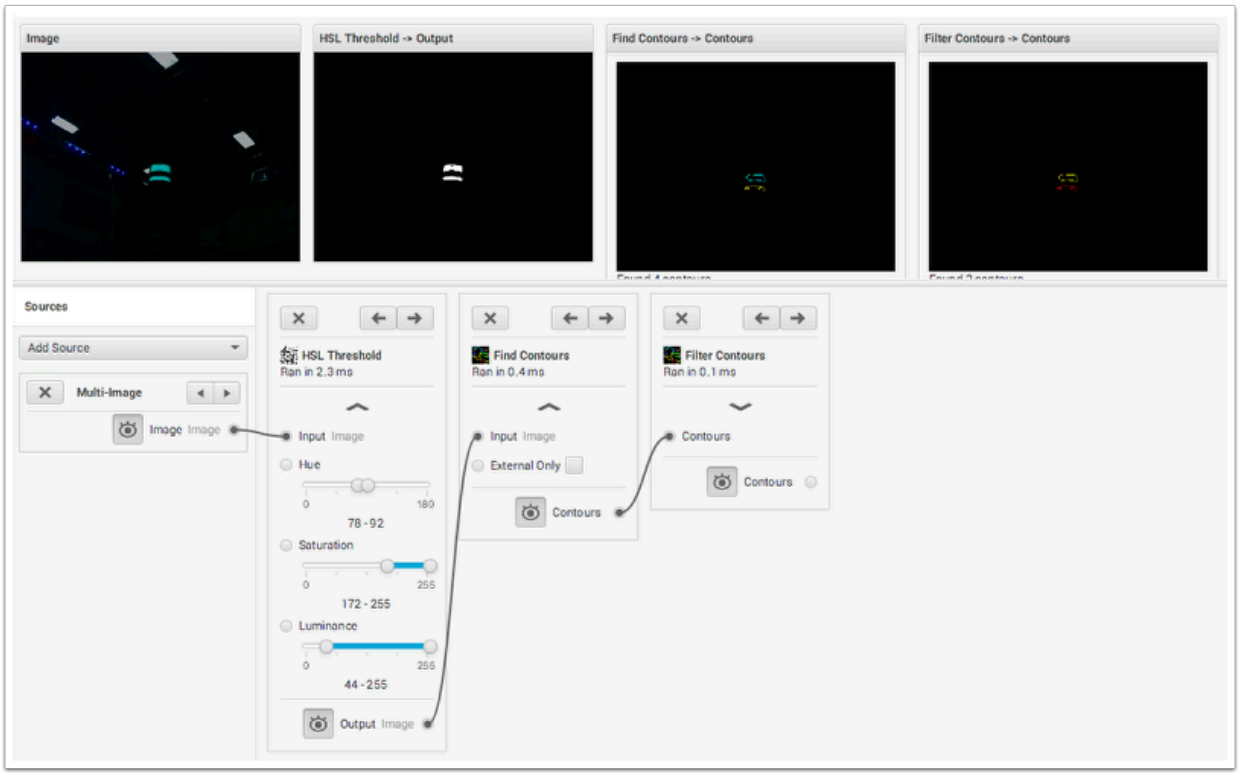
You can reduce the number of extraneous contours found a number of ways. One is to filter on the area of the contour. In the following example, a minimum area of 125 pixels was shown. The filtered contours are shown in the next image.



And now there are only 2 contours remaining representing the top and bottom piece of retrorecflective tape.

Here is the full GRIP pipeline that is described in this article.

Vision Processing



Further processing

From there you may wish to further process these contours to assess if they are the target. To do this:

1. Use the boundingRect method to draw bounding rectangles around the contours
2. Use the following sample heuristics as a starting point that you can apply to score your found targets.

Each of these ratios should nominally equal 1.0. To do this, it sorts the contours by size, then starting with the largest, calculates these values for every possible pair of contours that may be the target, and stops if it finds a target or returns the best pair it found In the forumlas below, 1 followed by a letter refers to a coordinate of the bounding rect of the first contour, which is the larger of the two (e.g. 1L = 1st contour left edge) and 2 with a letter is the 2nd contour. (H=Height, L = Left, T = Top, B = Bottom, W = Width)

// Top height should be 40% of total height (4in / 10 in.)
Group Height = 1H /((2B - 1T) *.4)

// Top of bottom stripe to top of top stripe should be 60% of total height (6in / 10 in.)
dTop = (2T - 1T) /((2B - 1T) * .6)

// The distance between the left edges of contours 1 and 2 should be small relative
// to the width of the 1st contour (then we add 1 to make the ratio centered on 1)
LEdge = ((1L - 2L) / 1W) + 1

// The widths of both contours should be about the same
Width ratio = 1W / 2W

// The larger stripe should be twice as tall as the smaller one
Height ratio = 1H / (2H * 2)

Each of these ratios is then turned into a 0-100 score by calculating: 100 - (100 * abs (1 - Val))

To determine distance, measure pixels from top of top bounding box to bottom of bottom bounding box
distance = Target height in ft. (10/12) * YRes / (2 * PixelHeight * tan(viewAngle of camera))

You can use the height as the edges of the round target are the most prone to noise in detection (as the angle points further from the camera the color looks less green). The downside of this is that the pixel height of the target in the image is affected by perspective distortion from the angle of the camera. Possible fixes include:

1. Try using width instead
2. Empirically measure height at various distances and create a lookup table or regression function
3. Mount the camera to a servo, center the target vertically in the image and use servo angle for distance calculation
4. Correct for the perspective distortion using OpenCV. To do this you will need to calibrate your camera with OpenCV as described here.

   This will result in a distortion matrix and camera matrix. You will take these two matrices and use them with the undistortPoints function to map the points you want to measure for the distance calculation to the "correct" coordinates (this is much less CPU intensive than undistorting the whole image)