

ZebraVision 3.0 – Team 900

ZebraVision 3.0 represents the culmination of many hours of work and research by the programming students for FRC Team 900, also known as the Zebracorns. It encompasses a library and several applications that are designed to use a Logitech C920 camera mounted to a robot to detect recycling bins in the 2015 FRC game: Recycle Rush. It is designed to run in real time using the OpenCV library onboard a NVIDIA Jetson TK1 DevKit mounted to the robot. The detection is done using the Cascade Classification method and Local Binary Patterns.

Evolution of ZebraVision

Our first version of ZebraVision was used during the 2012 competition season. The game was Rebound Rumble and we decided to use stereoscopic vision with dual Axis 206 cameras. As this was the year before bandwidth limitations we abused the network as both cameras were streaming to our driver station to perform the calculations. We used stereoscopic vision to accurately determine the distance to the hoop from any point around the key. After locating the target in both cameras and matching we used the four corners and the centroid to increase the accuracy of angle and distance. At further distances we used the 4 targets together to determine the distance. With the addition of bandwidth limits on the network and the implementation of QoS, we could not implement a heavy off robot solution in the future.

The second version of ZebraVision was used in the 2014 competition season, Aerial Assist. We used a single Axis m1011 to perform 'hot' goal detection during autonomous and with the aid of a Frisbee lined with retro-reflective tape were able to perform human player target detection during teleop to align the robot for our full-court shot. For this implementation we relied on single frame captures rather than streaming to perform target detection. This resulted in a very robust system when it came to utilizing the available network bandwidth.

Rationale for Onboard Vision Processing

In the course of our research to increase automation with the aid of video processing, some people have asked us, "Why process video on the robot? Why not process video on the driver station?" The answer is both simple and complex. It is simple because it comes down to an issue of control, and the variables involved in sending large datagrams over the network at an FRC event are beyond our control in addition to not being guaranteed to complete as only robot control packets is given priority. It is complex because there is a lot of variance in the wireless communications used for FRC competitions. For instance, some of the variables that determine the speed, latency, and reliability of the data being sent over the network:

- The placement of the radio on the robot, including where in the chassis the radio is placed, and the orientation of the radio
- The proximity of the radio to sources of interference and noise from the robot
- The manufacturing deviation from one radio to another (Yes, seriously, there are some unexplained differences between radios)
- The movement of the radio on the field (It's a moving target!)
- The interference between radios on the field

- The interference between radios and signals off the field
- The black box nature of the Field Management System (FMS). Not all fields are managed the same.
- The interference between the radio and various field elements
- The interference between the radio and field personnel (Humans are bags of water to radios!)

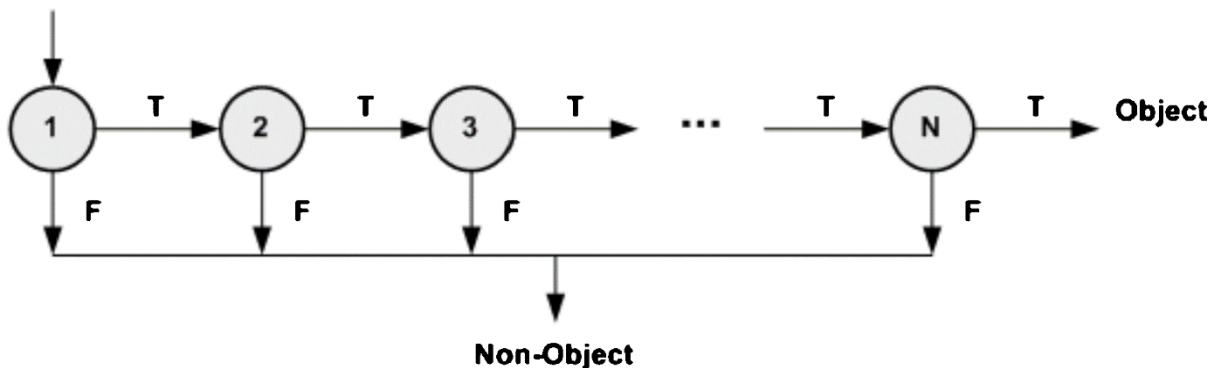
While we can't control many of these variables, we can minimize the use of the wireless network. Instead of streaming video (or even compressed video) over the wireless network back to a driver station, we can stick to simple control signals and process the video on the robot's internal wired network instead. As technology continues to evolve and FIRST adopts newer communications between the robot and the driver station, we will continue to re-evaluate many of these variables. For now, however, we have learned to avoid relying on the wireless network for our video processing.

How it Works

Cascade Classifier

Cascade Classification is a method of finding a specific pattern in an image. This can be used to detect many different types of objects. A **classifier** is an algorithm that classifies observations (in this case a portion of an image) into different categories. In this case the classifier is deciding between two categories: object or not object. The word “cascade” refers to the fact that the classifier is made up of many simple classifiers (known as **stages**). The classifier will take an input image and put it through many stages.

Input window



[Image from Journal of Electronic Imaging⁹](#)

This diagram illustrates how the classifier works. The input window is a section of an image, and each number represents a stage. To detect an object in an image the input window is moved over the image and resized to detect objects of different sizes. At each stage the input window can be rejected (false) or passed on to the next stage (true). This turns out to be a powerful method. Each classifier stage is fast since most images are rejected quickly which means that not all stages need to be applied to each input. This drastically reduces the amount of computation necessary and increases the speed of the classifiers.

Training and Generating Classifiers

Detecting anything using cascade classification requires a **classifier**. A classifier is a file that contains a description of what should be detected. Classifiers need to be generated by **training**. Training is done by code that comes with OpenCV (see the guide [here](#)¹ if you want to learn more about how it works). ZebraVision contains prewritten scripts to automate usage of these functions. At this point, if you don't have the ZebraVision library, you can pull it from GitHub with [this link](#)². Here are a few other requirements to make sure that you're set up to begin working:

- Knowledge of Linux/Unix and basic command line navigation skills
- Make sure you're running on either Linux or Cygwin on Windows. We have tested the code on ARM and x86 architectures, and it works the same on both.
- The training code will run faster when given more RAM. Make sure you have at least 4GB; more is always better.
- Packages required:
 - Boost
 - OpenCV (To install OpenCV on Cygwin follow [these](#)³ steps.)
 - Cmake

This tutorial will walk you through detecting the game balls from the 2014 game Aerial Assist. This is a good example because they are square (in an image) and a good illustration of how color does not affect the classifier. There are two things that you need to start training:

- Videos with a ball in it
- Videos without a ball in it

These videos should be in different conditions and capture the object from different angles. From now on we will refer to the videos with balls in them as **positive videos** and the videos without as **negative videos**. Thanks to teams [1986](#)⁴ and [1939](#)⁵ whose release videos were used as positive videos. We also used [our video](#)⁶ from the NC regional in 2014. The first step is to extract images of the ball from these videos that we can use for training. Start by going into the imageclipper directory and building the tool:

```
cmake .  
make  
mv ./bin/imageclipper.exe .
```

This will build the imageclipper and put it in the root directory so it's easier to use.

Imageclipper is a utility that lets you step through a video and extract samples of certain frames. This will be useful for acquiring **positive images** or **samples** for the classifier. Now run it with a positive video as a parameter:

```
./imageclipper.exe [path to video]
```

Now you should get a window with the first frame of the video. Pressing the 'f' key advances one frame, and you can drag with your mouse to select a portion of the image. The idea is to select the object when it is fully visible and not blurry. The training works better when the aspect ratio of the samples are near constant. The line around the sample turns yellow and green when you are close to a 1:1 aspect ratio.

To save a selection as an image press the 's' key. When you are done, move the samples to the positive_images folder within cascade_training.

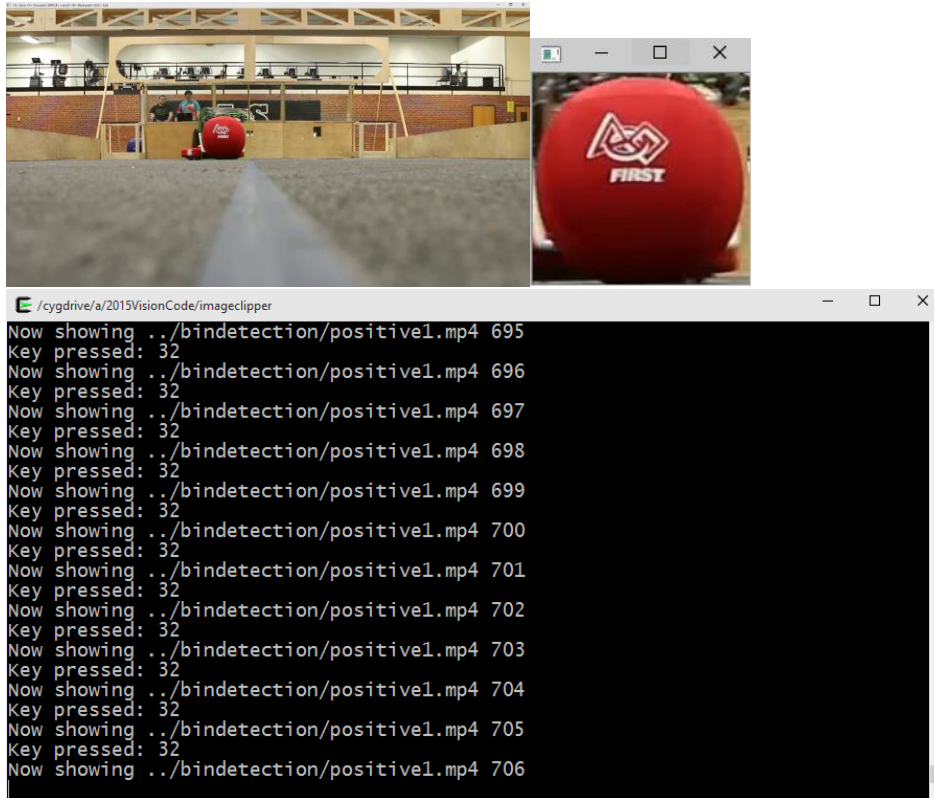


Image created by Alon Greyber

The more positives you have, the better. For our detection during the 2015 season we used about 1,300; for this demonstration I used 47. If you want to use my dataset and try it yourself you can get the positives [here](#)⁷. Now it's time to generate some negative images. Since we don't have a classifier we'll just use the frames of a negative video as negatives for now. The framegrabber code will grab a specified percentage of frames of a video. Go into the folder and build the project. Then run the project with a negative video as the parameter. This will output 100 images by default. Put these images in the negative_images folder within cascade_training directory.

Now we need to put these positives and negatives into a format readable by the training application. The easiest way to do this is with the automated script *prep.sh*. Before you run it there is one parameter you need to change. The code takes each positive and randomly rotates it and scales it slightly so that we can detect things that aren't exactly like the positives. In the code there's a line that lets you specify how many of these it should generate. This should be about 30x the number of positives that you have. In this case I chose 1410:

```
40 # For each positive image, create a number of randomly rotated versions of that i
41 # This creates a .vec file for each positive input image, each containing multipl
42 # rotated random amounts
43 perl createtrainsamples.pl positives.dat negatives.dat . 1410 | tee foo.txt
44
45 # Merge each set of randomized versions of the images into one big .vec file
```

Now you can go ahead and run the script. This will initially create many files and then delete them all at the end, leaving just a negatives.dat and positives.vec in the cascade_training folder.

Before we begin the training, there are a multitude of parameters that can be adjusted in the run_training.pl script. There is a lot of documentation in the script itself, but here is an explanation of the parameters that you should tweak:

-data	Name of the directory that the classifier will be stored in. Our naming convention was classifier_X_# where X is the object being detected and # is which number classifier this is. Make sure to create this directory before starting; otherwise the code will not save your progress.
-numStages	Number of stages to generate before the code stops. You will typically not reach this value as stages get exponentially slower so the default is usually fine (55).
-numPos	Number of positives that the training uses. This should be about 85% of the number of positives that you generated in the step above.
-numNeg	Number of negatives that the training uses. There isn't agreement about how this number should be picked, but using a number in a similar range to numPos should work. The higher this number is the longer training will take.
-precalcValBufSize and -precalcIdxBufSize	These parameters set the amount of memory the classifier will use. There is some overhead in addition to these values, so we found that each should be set to around 1/3 of your system's available memory.

Finished parameters list example:

```
61 my $pid = open(PIPE, "/bin/opencv_traincascade -data classifier_ball_1 -vec positives
    .vec -bg negatives.dat -w 20 -h 20 -numStages 55 -minHitRate 0.999 -
    maxFalseAlarmRate 0.5 -numPos 1200 -numNeg 1200 -featureType LBP -
    precalcValBufSize 4000 -precalcIdxBufSize 4000 -maxWeakCount 1000 |");
```

Now you can run the training:

```
perl run_training.pl
```

You should now see a list of all of your parameters followed by a line that says "TRAINING 0-stage". This indicates that your training started. The first few stages will finish pretty quickly. Let's look at a stage of training and decipher the output:

```
===== TRAINING 1-stage =====
1. <BEGIN
2. POS count : consumed    1200 : 1201
3. NEG count : acceptanceRatio    1200 : 0.186829
4. Precalculation time: 0
   +---+-----+-----+
5. |  N  |      HR      |      FA      |
   +---+-----+-----+
6. |   1 |           1 |           1 |
   +---+-----+-----+
7. |   2 |           1 |           1 |
   +---+-----+-----+
8. |   3 |           1 |           1 |
```

```

      +-----+-----+-----+
  9. |    4| 0.999167| 0.631667|
      +-----+-----+-----+
 10. |    5|          1| 0.769167|
      +-----+-----+-----+
 11. |    6|          1| 0.349167|
      +-----+-----+-----+
 12. END>
 13. Training until now has taken 0 days 0 hours           0
minutes 13 seconds.

```

Line 2: This line shows how many positives were generated and then how many are actually being used. The number being used is larger because some extra positives are required.

Line 3: The first number on this line is the number of negatives being used for training and the second number is the **acceptance ratio**. The acceptance ratio is the percentage of negatives that are being let through to this stage of the classifier. In this case the previous classifier stage (stage 0) has already removed around 81% of the negative images.

Line 6-11: Each line shows the result of testing a new filter on the current stage. There are two criteria that need to be satisfied before one of these becomes a new stage. These criteria are represented by the two columns HR and FA:

- **HR – Hit Rate** is the percentage of times that the classifier accepts a positive image. Each stage must have a hit rate of >0.999. This is a parameter in the run_training script (minHitRate)
- **FA – False Alarms** is the percentage of negative images that are allowed through the stage. By default the stage must accept at most 50% of the negative images. In the case above, the classifier accepted only 34.9%, much better than was required. This does not usually happen, especially in later stages. Maximum false alarms is also a parameter in the run_training script (maxFalseAlarmRate).

We can calculate the end hit rate and false alarms given a number of stages, minimum hit rate, and maximum false alarms as follows:

$$\text{overall hit rate} = \text{minHitRate}^n \quad \text{and} \quad \text{overall false alarms} = \text{minFalseAlarms}^n$$

Where n represents the number of stages completed. For example in this case I generated 28 stages so

$$\text{overall hit rate} = 0.999^{28} = 0.972 = 97.2\%$$

$$\text{overall false alarms} = 0.50^{28} = 0.00000000373 = 0.000000373\%$$

The overall false alarms rate is the same as the **acceptance ratio**, or the percentage of images that survive being filtered out through all classifier stages. This means that if 1 billion negatives were added to the classifier on average, it would only recognize about 3.7 falsely as positives. That sounds pretty good but the classifier is moving the input window over the image thousands of times and repeating this at hundreds of different sizes. It's a good start but it can be improved.

After about 25 stages, stop training. You may even have to stop before because the code tends to run very slowly at later stages. The next thing we are going to do is generate new negatives, known as **hard negatives**, using the classifier. Hard negatives are negatives that are falsely detected by the current classifier. We can generate these by running the classifier on videos with no balls in them and looking at what the classifier produces. Run the create_cascade script:

```
User@MSI-GT70 /cygdrive/a/2015VisionCode/cascade_training
$ ./create_cascade.sh ./classifier_ball_1/
```

This will take each stage and create a full classifier from it. Now go into the generate_negatives directory and build using Cmake. You should have a binary called generate_negatives. The parameters are listed in the README file but this is what you need to know for now:

1. Path to a negative video. You can use any of them; it doesn't really matter.
2. Path to the classifier. This should be an xml file within the directory that your classifier is in. You should use the latest stage that you have, in my case 28.
3. -negative_count X - (optional) This lets you specify the maximum number of negatives to generate. Some videos can generate more than 100,000 negatives, so this parameter is important for longer videos

For me the command ended up being something like this:

```
User@MSI-GT70 /cygdrive/a/2015VisionCode/cascade_training/generate_negatives
$ ./generate_negatives.exe ./negative.mp4 ../classifier_ball_1/cascade_28.xml
-negative_count 10000
```

This will generate 10,000 negative images. Copy them from the negatives folder in generate_negatives to the negative_images folder in the cascade_training directory. Now that you have more negatives, you can go ahead and start training again. The code will prep all of the negatives and pick up where it left off.

Another thing that you may want to do at some point is restart the training again. This is necessary if you want to add more positives. For our season detection code we have multiple directories (classifier_bin_1, classifier_bin_2, etc.) that each represent one run of the classifier. During the season we ran the classifiers on positive videos using our detection code and found bins that the code did not recognize. We took samples of these and used them as additional positives to construct a better classifier. To restart training you need to do a few things:

1. Delete your hard negatives. Old hard negatives won't do any good because it's a new classifier. You can however reuse your negatives that you generated with framegrabber.
2. Change the --data argument in run_training so that it's not overwriting your older classifier. Remember that you need to create the directory before starting the training.
3. Rerun the prep.sh script. This is necessary whenever new positives are added.
4. Run the run_training script.

That's it! Once you're happy with the classifier you can use it with the [OpenCV example cascade detection code](#)⁸ or use it with the bindetection code.

Detection Code

Overview

The bindetection code is the code that was used in 2015 to detect recycling bins using an NVIDIA Jetson TK1 DevKit. The main file is creatively named “test”. The entire project is quite large and split up into a few main areas.

- **Classifier Interaction** – loading and switching between classifiers at runtime
- **GPU and CPU Detection** – Automatic switching between GPU and CPU depending on the system’s capabilities
- **Arguments** – Processing command line arguments
- **Network tables** – Passing data back to LabView navigation code
- **Bin tracking** – Tracking bins across multiple frames
- **Detection** – Detecting bins
- **Genetic Algorithm** – Tuning parameters using a genetic algorithm approach

Building and Running

All of the code is configured to be built using Cmake. To build first enter the bindetection/galib247 directory. This is a genetic algorithm library. It is not in use actively in the code but was used to determine values for multiple constants in our code. It is now listed as a dependency and must be built before anything else.

```
make -j5
```

You can stop the make as soon as it finishes building libga. The examples are not required. Exit the galib247 directory and do another Cmake followed by make.

```
cmake
```

```
make -j5
```

This will build all of the code. To run type

```
./test
```

This will start the code running off of a webcam connected to your computer using the default classifier. To run off of a video file use

```
./test <video path>
```

There are other arguments that you can use as well. These are discussed in the Arguments section below.

Arguments

The bindetection code contains a separate class for parsing command line arguments (Args.hpp/cpp). Running the “Usage” function in the class will also display usage information. Here is a list of command line arguments possible:

```
./test [arguments] [camera number | video name]
```

Argument	Action
----------	--------

--frame=<frame num>	Start at a given frame in the input video. Must be used in conjunction with the video name argument.
--all	Write every detected sample to disk
--batch	Run without GUI
--ds	Driver station mode – run without GUI and look for 4 bins
--calibrate	bring up crosshair to calibrate camera position
--capture	Save camera output to disk
--save	Save processed video to disk. This includes rectangles around each detected bin
--no-rects	Start with detection rectangles disabled. Detection will still take place just silently
--no-tracking	Start with tracking rectangles disabled
--classifierDir=	Allows you to specify an initial directory for the classifier
--classifierStage=	Allows you to specify an initial classifier stage

Detection

The actual code to search the image for an object is only one line (detectMultiScale). However, there is a lot that goes into this. All of the code for this is in imagedetect.cpp/hpp and some is in the test.cpp file. In imagedetect there are a few different functions. It is important to notice is that there are two classes: one for CPU detection and one for GPU detection. They are very similar except that the GPU version uploads and downloads from the GPU.

Another important part of the imagedetect script is the definition at the top called DETECT_ROTATED. This definition will enable detection of bins that are rotated -90, 90, and 180 degrees. This works by rotating the image and applying the classifier 4 times, which also decreases the framerate. By default it is disabled.

The bindetection code will automatically detect whether your system has a CUDA enabled graphics processor and accelerate the processing accordingly. The code for this can be found in the imagedetect.cpp/hpp files. The character 'G' can be used to quickly switch between GPU and CPU modes. Please note that the only function that is accelerated with the GPU is the detectMultiScale function because it is the main function and other ones would not be measurably faster on the GPU.

Bin Tracking

The bindetection code has the ability to locate bins and track them for extended periods of time. Because the classifier draws a rectangle around the bins, we can use the size of the rectangle in pixels plus the known width of the bin to compute the distance to the bin. It's a short step from there to find the angle that the robot needs to turn to navigate towards the bin.

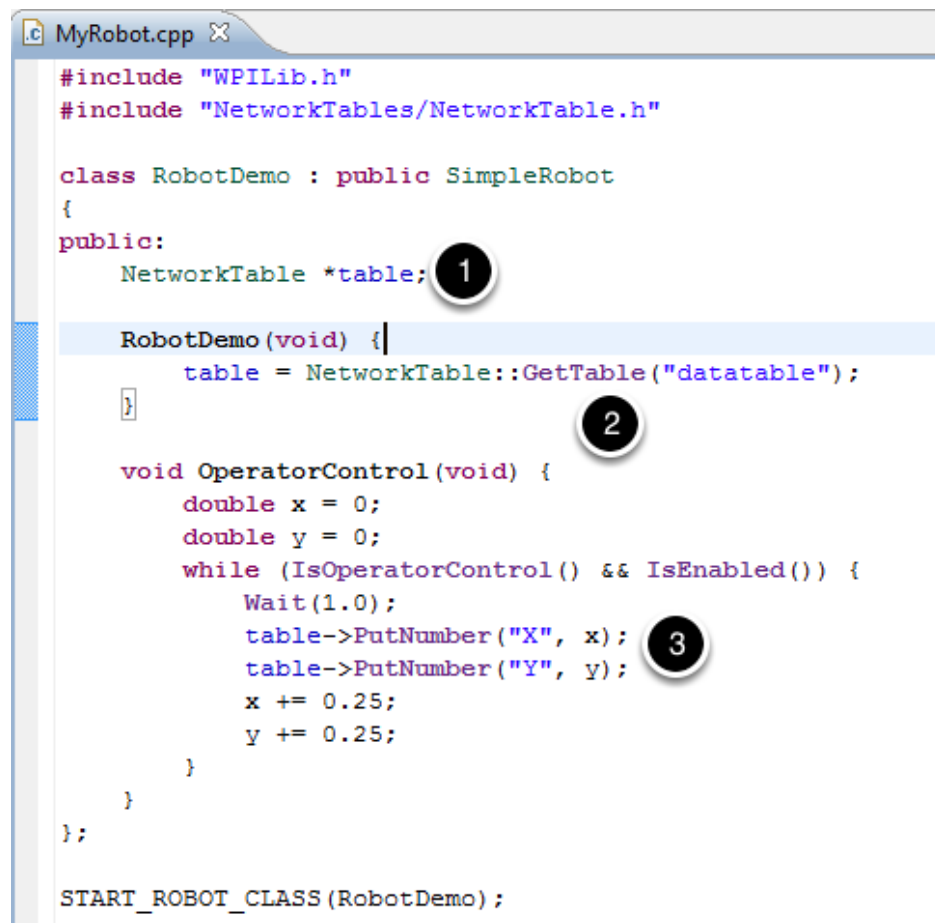
The code also remembers where an object was in a previous frame and tries to find the same object again in the current frame. If the code finds the object in multiple frames in a row, it builds up a confidence rating which delays the disappearance of the bin after not being sighted.

All of this is accomplished using the TrackedObject and TrackedObjectList classes inside of track.cpp/hpp. There are a few important functions to note:

- processDetect(vector<Rect> objects) – Main function, takes all of the detected rects in a frame and updates locations of all of the objects
- adjustAngle(float angle) – Moves all rects a certain amount based on an angle and the distances of each rect. This is useful because if you can track robot location and send the data back to the detection code, the code can auto adjust the locations of the rects so as to not lose track of them
- nextFrame() – Moves onto the next frame. This will update the confidence ratings of each object

Network Tables

Networktables is a conceptually simple key/value database and is part of the WPILib. The robot runs as the server with clients connecting and updating frequently. Networktables can be run on the roboRIO/cRIO, and x86_64/ARM processors platforms running LabVIEW, C++, or Java.



```

#include "WPILib.h"
#include "NetworkTables/NetworkTable.h"

class RobotDemo : public SimpleRobot
{
public:
    NetworkTable *table; 1

    RobotDemo(void) {
        table = NetworkTable::GetTable("datatable"); 2
    }

    void OperatorControl(void) {
        double x = 0;
        double y = 0;
        while (IsOperatorControl() && IsEnabled()) {
            Wait(1.0);
            table->PutNumber("X", x); 3
            table->PutNumber("Y", y);
            x += 0.25;
            y += 0.25;
        }
    }
};

START_ROBOT_CLASS(RobotDemo);

```

[Image from WPILib⁹](#)

This example program simply reads or writes values from within the program. The instance of NetworkTables is automatically created by the WPILib runtime system. This example is the simplest robot program that can be written that continuously writes pairs of values (X, and Y) to a table called "datatable". Whenever these values are written on the robot, they can be read shortly after on the desktop client.

The variable "table" is of type NetworkTable. NetworkTables are hierarchical, that is tables can be nested by using their names for representing the position in the hierarchy.

The table is associated with values within the hierarchy, in this case the path to the data is /datatable/X and /datatable/Y.

Values are written to the "datatable" NetworkTable. Each value will automatically be replicated between all the NetworkTable programs running on the network.

When this program is run on the robot and enabled in Teleop mode, it will start writing incrementing X and Y values continuously, updating them 4 times per second (every 250 milliseconds).

```
NetworkTable::SetClientMode();
NetworkTable::SetIPAddress("10.9.0.2");
NetworkTable *netTable = NetworkTable::GetTable("VisionTable");
const size_t netTableArraySize = 7; // 7 bins?
NumberArray netTableArray;

// 7 bins max, 3 entries each (confidence, distance, angle)
netTableArray.setSize(netTableArraySize * 3);
```

We created a network table called "VisionTable," and instead of passing individual values, we passed an array that contains the acceptance ratio, the distance, and the relative angle of an object.

Classifier Interaction

The main files that deal with classifier interaction are the classifierio.cpp and classifierio.hpp files. The reason that these are necessary is that the detection code only interacts with one classifier at a time, and to switch classifiers the code must be edited, recompiled, and run again. This code enables you to switch classifiers and classifier stages at runtime. This is useful because sometimes certain classifier stages work better than others and it can be helpful to easily compare these. Once you find a good classifier stage you can hardcode the value into the code and this is no longer necessary.

The structure that the code uses is:

- classifier_bin_1
 - cascade_1.xml
 - cascade_2.xml
- classifier_bin_2
 - cascade_1.xml
 - etc...

Classifier_bin_x is a directory that represents one training run of the classifier and cascade_x.xml is a file that represents one stage of the classifier. This is useful because some stages will work better than others and it is nice to be able to switch quickly between them to find the differences. When running the code the following keys are used for navigation:

Key	Action	ClassifierIO Class Function
.	Higher stage	classifierIO.findNextClassifierStage(true)
,	Lower stage	classifierIO.findNextClassifierStage(false)

>	Higher directory	<code>classifierIO.findNextClassifierDir(true)</code>
<	Lower directory	<code>classifierIO.findNextClassifierDir(false)</code>

Genetic Algorithm

There are a multitude of parameters in the code that can be adjusted, such as

- Scale
- Neighbors
- Min Detect
- Max Detect

For a while we were tuning these values by hand each time we tried the application. With later stages and versions of the classifiers, these values became easier to set, but originally it was a big issue. We thought that using genetic algorithms could help us find the optimum values, especially for scale and neighbors. The file that has all of this code is `gaTesting.cpp`. The code ended up being too slow and inefficient, but it could be adapted for future uses. The basic idea is to provide an objective function that returns a value of how close you are to the target. Once you have the objective function the genetic algorithm will work and try to find a value for you. The objective function takes 10 different frames and tries to find bins in all of them. It returns a float, which is a fraction that was detected. The problem was that most values would not detect bins so the genetic algorithm would settle on an arbitrary value instead.

ZebraVision 4.0

ZebraVision has become an important project for the team and will be developed further. Cascade classification worked remarkably well for our team this season. We were able to use this code to adjust our robot positioning during [autonomous¹⁰](#) in real time. There are a few things that we could have done better:

- Much of our code is messy. This is a common problem while coding in build season but it is especially important to keep code readable and organized throughout the season. In the future we may switch to using an IDE to help organize our project.
- We never found a good way to tune values such as scale and neighbors.
- Values such as min and max detect heavily influenced the speed at which the code runs. Our code can detect the distance that a detected bin is and pass data back to the robot, creating a one way flow of data. However, if data about field location could have been transferred back to the vision code, adjustments to the min and max detect could be made in real time based on the maximum distance that a bin could be given the robot's location and orientation. This may be an offseason project for our team to implement.

The biggest downside to using cascade classifiers was the speed. We constantly had to make as many optimizations as possible, including sacrificing detection accuracy to run on the GPU. Next year, with help from NVIDIA, we will be working on using deep learning neural networks for detection. These will hopefully eliminate the need to tune values and help mitigate our concerns about speed.

Links

0. <http://electronicimaging.spiedigitallibrary.org/article.aspx?articleid=1100656>
1. http://docs.opencv.org/doc/user_guide/ug_traincascade.html
2. <https://github.com/FRC900/2015VisionCode>
3. <http://hvrl.ics.keio.ac.jp/kimura/opencv/opencv-2.4.11.html>
4. <https://youtu.be/sLlgDgQ-K70>
5. <https://youtu.be/xfsfdWAMzeU>
6. https://youtu.be/t1d4Y_ItZcw
7. <https://drive.google.com/file/d/0B87h8FxFUqjcGRU5DV1JaX2IPTVk/view?usp=sharing>
8. https://github.com/Itseez/opencv/blob/master/samples/cpp/tutorial_code/objectDetection/objectDetection2.cpp
9. <http://wpilib.screenstepslive.com/s/3120/m/7912/l/80205-writing-a-simple-networktables-program-in-c-and-java-with-a-java-client-pc-side>
10. https://youtu.be/WqHk50xX1_A