FRC Programming Done Right Documentation

Release 0.2

Tim Winters

Jun 01, 2017

Introduction


FRC
PROGRAMMING DONE RIGHT

CHAPTER 1

PID Control

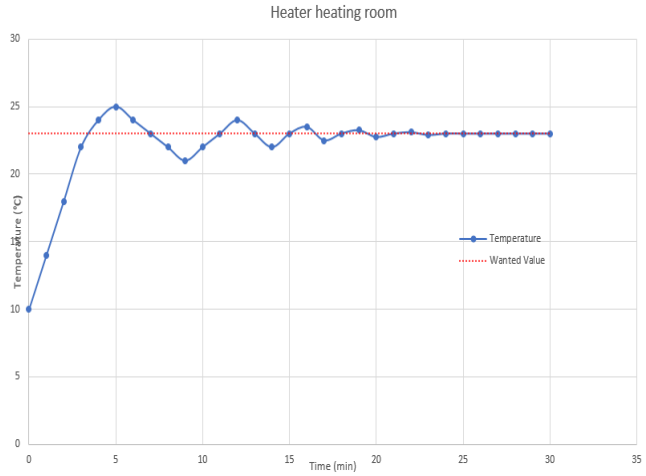
PID Proportional, Integral, Derivative feedback loop

Gain Amount of output given from each of the components of PID.

Feedforward A known value supplied to the output as a guesstimate so the PID only has to make minor corrections.

PID Control lies at the heart of any advanced robotics motion. Essentially, it is a way of controlling something, i.e. a wheel or an arm, using information gathered by the surroundings, In robotics, data is usually gathered through sensors, like encoders, range sensors, light sensors, etc. Using this data, robots can determine how they should act.

Let's say you have a cold room, like 10 degrees. You want to warm it up to a nice 23 degrees (celsius). Luckily you have a heater. You set it's thermostat to 23, and it starts heating. It heats as fast as it can, and quickly gets to 23 degrees. It immediatly turns off. However, the coils on the heater are still warm, and continue to heat the air for a while after. The room heats up to 25, before the coils cool down, and the room loses heat to the environment. It dips down to 23 degrees, and the heater turns on - but it takes time for the coils to turn on, and during this time the room cools down to 21 degrees. This oscilliation around the set point slowly dies out, over a long period of time.



PID is designed to intelligently approach the target to reach it as quickly as possible. So in this example, the heater would have turned off before it hit 23, say at 21 degrees, such that it natrually warms up to 23.

In robotics, the same concept can be applied. Many teams use PID control to drive during autonomous, using encoders as their sensor, shooting, using cameras as their sensor, or rotating, using gyros as their sensor.

The main equation for PID Control is

$$output = P \times error + I \times \sum error + D \times \frac{\delta error}{\delta t}$$

Proportional

$P \times error$

Proportional control is using a predetermined constant (`kP`)to control how much a mechanism can move. Every time the PID code is run, the error is calculated, and the proportional gain is multiplied to this. In the car analogy, lets say the pressure you were applying was inversely proportional to the distance you were from the stop sign. `P = 1/(error^2)` and `error` is distance from the stop sign. (Note, P is 1/error^2 because you want the output to be 1/error.)

Distance	Output
200	.005
150	.0067
100	.001
50	.02
10	.1
1	1

Using this P value, we apply more pressure the closer we get, causing us to slow down.

Using only Proportional control can be done, and is usually better for slow moving mechanisms, or mechanisms where you don't need com

Integral

$I \times \sum error$

When controlling a motor with just P, you may find that it oscillates. This happens because it has too much power when it gets to the setpoint, and it overshoots. Then when it tries to correct, it overshoots again, and this cycle continues. One way to reduce this is to lower P. This could have some bad side effects though. By reducing P, your motor may not get *all* the way to where you want it to. It may be off by a few degrees or rotations. To overcome this **steady-state** error, an Integral gain is introduced.

If you've taken calculus, you know the integral is the area under a curve or line. It's the same with PID. The Integral gain is the sum of all the past error. This means the gain will increase more and more the longer the motor isn't where it's supposed to be.

Even though this reduces steady state error, it may increase settling time. If you notice it oscillating a little bit before settling, you may need a Derivate gain.

Derivative

$D \times \frac{\delta error}{\delta t}$

Derivative gain works by calculating the change in error. By finding this change, it can predict future system behavior, and reduce settling time. It does this by applying a brake more or less. This can be useful if it is imperative that you don't overshoot. This isn't even used in the industry much, but if you find yourself with long settling times, it may help to introduce a Derivative gain.

Tuning Methods

Zeigler-Nichols

Zeigler-Nichols tuning method works by increasing `P` until the system starts oscillating, and then using the period of the oscillation to calculate `I` and `D`.

1. Start by setting `I` and `D` to 0.
2. Increase `P` until the system starts oscillating for a period of `Tu`. You want the oscillation to be large enough that you can time it. This maximum `P` will be referred to as `Ku`.
3. Use the chart below to calculate different `P`, `I`, and `D` values.

Control Types	P	I	D
P	.5*Ku	-	-
PI	.45*Ku	.54*Ku/Tu	-
PID	.6*Ku	1.2*Ku/Tu	3*Ku*Tu/40

Note: The period of oscillation is one full 'stroke', there and back. Imagine a grandfather clock with a pendulum, when it is all the way to the right, swings to the left, and hits the right again, that is 1 period.

Which ones to use

P control is best used on slow moving parts that aren't subject to overshooting, or parts of the robot that don't need complete accuracy. Turning to a certain degree, for example, can be done with just P in some cases (but not all).

The most common control loop is PI. It combines simple P control with the fine tuning feature of an Integral gain. This is teams are most likely to use.

Complete PID may be overkill for an FRC robot, but if you find that PI isn't working *enough*, feel free to add D gain

CHAPTER 2

Encoders

An encoder is a sensor that measures motor movement, usually in the form of position or rotation. They are useful for getting precise movement from a mechanism by providing feedback to a PID (or other feedback) loop.

Relative Encoders

Relative encoders detect changes in position relative to their starting position, which resets when they power off. Hall-effect quadrature encoders, the most common encoders in FRC, have A & B outputs that are used to count the number and direction of steps.

Relative encoders are generally used as either velocity or distance sensors. For example you can use the rate of the steps to calculate the RPM of a flywheel launcher. If you put an encoder on a wheel you can calculate the distance traveled by multiplying the number of steps by a constant.

Absolute Encoders

Absolute encoders provide their position relative to a static point, even after a power loss or reboot. Absolute encoders have more complex outputs, like analog or PWM, and have higher resolutions than most relative encoders. Absolute encoders do everything relative encoders do along with providing accurate position, but they are more expensive and complex.

Absolute encoders are used in applications like swerve drive modules where their direction always needs to be relative to the robot. Absolute encoders must be used when the position readings need to be relative to the encoder and not its starting position.

CHAPTER 3

Limit Switches

Limit Switches are sensors that tell you when a component is touching it or not. They can be used to prevent mechanisms from moving too far in one direction or another, or - if placed on the outside of the robot - if you've hit something. They do this by sending a signal to the motor controller or roboRIO reporting it has gone the maximum distance.

There is nothing special about a limit switch that makes it a limit switch. Any type of switch will do, as long as it can reset itself when nothing is touching it (e.g. a light switch wouldn't work well as a limit switch since it doesn't automatically flip one way or another). Commonly used as limit switches are microswitches, since they have settings for Normally Open and Normally Closed, however, their small size makes them prone to breaking easily. They also require the mechanism to contact it head on, which can cause for awkward placement in a rotating object such as an arm. For these more advanced mechanisms, industrial limit switches might be useful because they can work with many different motions.

Wiring a Limit Switch

Limit switches generally need two wires. One for ground, and one for signal. On the limit switch, you will generally see 3 connectors. Normally Open (NO), Normally Closed (NC) (see above left for examples of labeling), and ground. Normally Open means the switch is normally in the unpressed/untriggered position. When something depresses the switch, it will send a signal back to the roborio/motor controller saying it has been tripped. Normally Closed is the opposite. Connect the white/yellow wire of a PWM cable to either *NO* or *NC*, depending on the application, and the black/brown wire to ground.

Industrial limit switches sometimes have 4 contacts. 2 *NO* and 2 *NC* to allow for multi-directional limiting, i.e. clockwise vs counterclockwise. Make sure to wire the correct side or your switch may not be tripped.

RoboRIO

To wire a limit switch to the roborio, you can use normal PWM cable. You can solder the wire directly to the limit switch, however certain connectors such as these andymark connectors should allow you to make a quick-connect wire. I recommend soldering because the small wires are hard to get a good crimp with, and connectors can become unplugged, but I recommend connectors as well because you can unplug them if you need to change something.

Once you create the wire, connect it to the correct port on the switch (NC or NO). The other end of the switch should go to a DIO, or Digital In/Out port on the roborio.

Talon SRX

The easiest way to connect a limit switch to a TalonSRX is through the breakout board. Linked is the encoder breakout board, but the analog breakout board will work as well. Theoretically you could solder directly to the SRX pin but it is not recommended.

If you don't want to buy the breakout board, you can use the roboRIO method. Connecting directly to the SRX is recommended because it interacts at the hardware level, and will respond faster.

Spark

The spark is the easiest to connect a limit switch to, since there is no soldering and the PWM cable can plug direcly into it. Connect the clipped end of the PWM cable to the limit switch, then plug the end with the housing still intact into the spark with the ground on the outside.

Programming a Limit Switch

The status of the limit switch can be determined using the `get` method of `DigitalInput`. Using them to control motor behavior is a bit harder. Your first instinct may be to call `get` on the limit switch, and if `get` returns `true`, set the motor to 0. This is... *not* going to work, unless you want the motor to be disabled for the rest of the game. When the limit switch is enabled, you need to limit motor speed **only in that direction** or else you can't reverse the motor. We can do this with a simple min/max function. JavaC++Python

```
public class MyRobot extends IterativeRobot{
    public void robotInit(){
        DigitalInput forwardLimitSwitch = new DigitalInput(1);
        DigitalInput reverseLimitSwitch = new DigitalInput(2);
        Talon motor = new Talon(1);
        Joystick joystick1 = new Joystick(1);
```

```
    }

    public void teleopPeriodic()
    {
        int output = joystick1.getY(); //Moves the joystick based on Y value
        if (forwardLimitSwitch.get()) // If the forward limit switch is pressed, we
↪want to keep the values between -1 and 0
            output = Math.min(output, 0);
        else if(reverseLimitSwitch.get()) // If the reversed limit switch is pressed,
↪we want to keep the values between 0 and 1
            output = Math.max(output, 0);
        motor.set(output);
    }
}
```

```
#include <math.h>

class Robot: public IterativeRobot
{
    DigitalInput forwardLimitSwitch, reverseLimitSwitch;
    Joystick joystick1;
    Talon motor;

public:
    Robot() {

    }

    void RobotInit(){
        forwardLimitSwitch = new DigitalInput(1);
        reverseLimitSwitch = new DigitalInput(2);
        joystick = new Joystick(1);
        motor = new Talon(1);
    }

    void teleopPeriodic() {
        int output = joystick1->getY(); //Moves the joystick based on Y value
        if (forwardLimitSwitch->get()) // If the forward limit switch is pressed, we
↪want to keep the values between -1 and 0
            output = fmin(output, 0);
        else if(reverseLimitSwitch->get()) // If the reversed limit switch is pressed,
↪ we want to keep the values between 0 and 1
            output = fmax(output, 0);
        motor->set(output);
    }
```

```
class MyRobot(wpilib.IterativeRobot):

    def robotInit(self):
        self.forwardLimitSwitch = wpilib.DigitalInput(1)
        self.reverseLimitSwitch = wpilib.DigitalInput(2)
        self.joystick1 = wpilib.Joystick(1)
        self.motor = wpilib.Talon(1)

    def teleopPeriodic(self):
        output = self.Joystick1.getY()
        if self.forwardLimitSwitch.get():
```

```
            output = min(0, output)
        elif self.reverseLimitSwitch.get():
            output = max(0, output)

        motor.set(output)
```

CHAPTER 4

Introduction

This guide aims to give the reader the knowledge to be able to use OpenCV to solve FRC vision tasks. But more importantly, this guides hopes the reader develops an understanding of the computer vision algorithms that OpenCV provides.

Along with theory, this series of pages will also give the appropriate OpenCV code in C++, Java, as well as Python to provide a grounding to the theory. While the math may get complex at times, we highly suggest you try your best to understand it as we want you to have an understanding of the algorithms themself, not just an understanding of how to call a function.

Data Structures

A data structure is a programming term that describes the format for organizing and storing data. In computer vision, the type of data we care about are images and points. While there are other data structures within OpenCV, we will cover those as they come up so this section won't be too long.

Images

The core of computer vision is the analysis of images, thus making the problem of how to efficiently encode images for programming is an essential one. Unfortunately, OpenCV is not consistent across its languages, so feel free to read about the language you are interested in using.

Generally, the images are 8 bit, meaning that the pixels take a value in the range [0, 255] where 255 is white and 0 is black, but OpenCV does not limit the user to 8 bit images. For instance, the Microsoft Kinect's depth camera produces a 16 bit image.

What is color for OpenCV? In OpenCV, that means that the image has 3 channels, where each channel represents a color. Again, OpenCV does not limit the user to 1 channel (grayscale) or 3 channel (RBG or HSV) images, they can construct 6 channel images if they wish, however that has no practical application to FRC, so shall restrict our images to single channel or 3 channels.

$$\begin{pmatrix} 39 & 230 & 0 \\ 205 & 79 & 255 \\ 15 & 114 & 165 \end{pmatrix}$$

C++

OpenCV's C++ binding encodes images as matrices, where each element in the matrix directly encodes for the pixel value at that particular index.

```
// creates empty matrices
cv::Mat A, C;

// Copy constructor
cv::Mat B(A);

// Assignment operator
C = A;
```

Java

Java's bindings also utilize matrices to encode images, however, as is explained later on, it is a bit trickier to display images.

```
// allocates memory for a new matrix
Mat frame = new Mat();
```

Python

The Python binding for OpenCV utilizes numpy to handle the heavy lifting of storing and operating on images as numpy is a highly optimized library specifically designed for fast computations. More specifically, the numpy.array is used to store images. This being said, python is not a user specified type casted language, so it is difficult to illustrate how to declare a np.array. While one can create a np.array of size 0, there is no application of that.

Points

Points in OpenCV are indentical to what you learned in your geometry class. They take the form (x,y), where x and y denote the location in an image, and the origin is the top left corner, positive x is to the right and positive y is down, as is common across computer vision libraries. JavaC++Python

```
Point pt;
pt.x = 10;
pt.y = 8;
Point pt2 = new Point(10,8);
```

```
Point pt;
pt.x = 10;
pt.y = 8;
Point pt2 =  Point(10, 8);
```

```
#Python uses tuples
(5, 10)
```

The Basics

The following few sections will let you get started with OpenCV in the language of your choosing.

Reading an image from a file

While reading images into memory isn't terribly useful for FRC, it is useful for debugging purposes. JavaC++Python

```
Mat img = Highgui.imread("image.png", Highgui.CV_LOAD_IMAGE_GRAYSCALE);
```

```
// 1 denotes to load it as a color image
img = imread("/home/faust/Documents/vision2013/boilerraw.jpg", 1);
```

```
# 0 denotes to load the image as grayscale
img = cv2.imread('picture.jpg',0)
```

Saving an image to a file

It is recommended that you save an image every so often for debugging purposes. However, take note that this is a time expensive operation and it is not necessary to do on every frame, one frame a second should be enough. JavaC++Python

```
Imgcodecs.imwrite("picture.png",img);
```

```
imwrite("picture.png", img)
```

```
cv2.imwrite('picture.png',img)
```

Displaying an Image

In order to help the development process, it helps to see the image and the effects of the operations that you apply to an image. When you display an image, make sure to have a waitKey(x) where x is the time to wait in milliseconds. Typically 10 milliseconds is sufficient. If -1 is passed as the arguement, the program will wait indefinitely until a key is pressed. JavaC++Python

```
public BufferedImage Mat2BufferedImage(Mat m){
    int bufferSize = m.channels()*m.cols()*m.rows();
    byte [] b = new byte[bufferSize];
    m.get(0,0,b); // get all the pixels
    BufferedImage image = new BufferedImage(m.cols(),m.rows(), type);
    final byte[] targetPixels = ((DataBufferByte) image.getRaster().getDataBuffer()).
↪getData();
    System.arraycopy(b, 0, targetPixels, 0, b.length);
    return image;
}

public void displayImage(Image img2) {
    ImageIcon icon=new ImageIcon(img2);
    JFrame frame=new JFrame();
    frame.setLayout(new FlowLayout());
    frame.setSize(img2.getWidth(null)+50, img2.getHeight(null)+50);
```

```
    JLabel lbl=new JLabel();
    lbl.setIcon(icon);
    frame.add(lbl);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

```
imshow("window name", img);
char keypress = waitKey(10);
```

```
cv2.imshow('window name',img)
cv2.waitKey(0)
```

As you noticed, OpenCV Java does not have an "imshow" method, making displaying an image a much more complicated ordeal.

Getting an image from a usb camera

Note the 0 in the capture commands. This denotes the first camera the computer recognizes. If you have a webcam, it will default to that. If using multiple cameras, it is best practice to plug them in in the order that you are getting them in your program every time you restart the computer. JavaC++Python

```
VideoCapture camera = new VideoCapture(0);
Mat frame = new Mat();
camera.read(frame);
//use frame for image processing from here
```

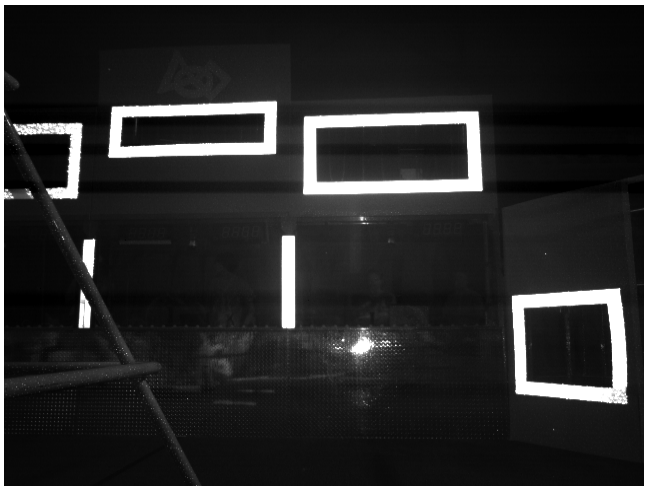
```
VideoCapture cam;
Mat image;
cam.open(0);
cam >> image;
```

```
cap = cv2.VideoCapture(0)
ret, frame = cap.read()
# use frame for image processing from here
```

CHAPTER 5

Thresholding

Thresholding is the process of converting an image to a binary image. If a pixel passes the threshold, it turns white (255 for 8 bit images), else, it turns black (0).



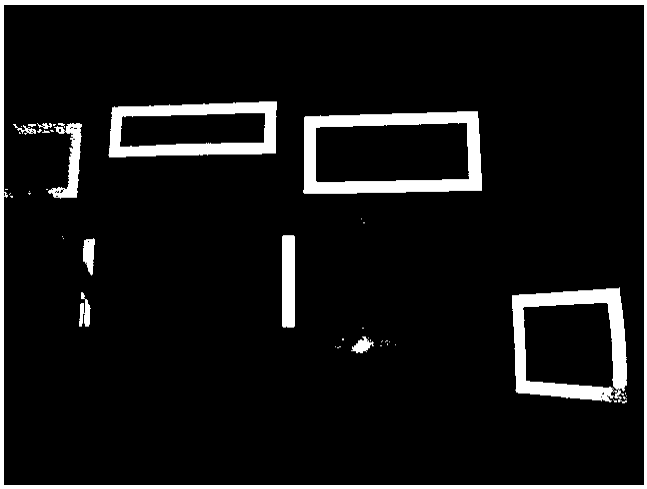
The rest of the page will talk about the various types of thresholding techniques OpenCV provides.

Threshold

The "threshold" algorithm is defined mathematically as

$$dst(x,y) = \begin{cases} maxValue & \text{if } src(x,y) > value \\ 0 & \text{if } src(x,y) <= value \end{cases}$$

In layman's terms, if $low < pixelvalue$, then the pixel passes the test and is turned white, else it is turned black. JavaC++Python



```
Imgproc.threshold(src, dst, value, maxValue, Imgproc.THRESH_OTSU);
```

```
cv::threshold(src, dst, value, maxValue, THRESH_BINARY);
```

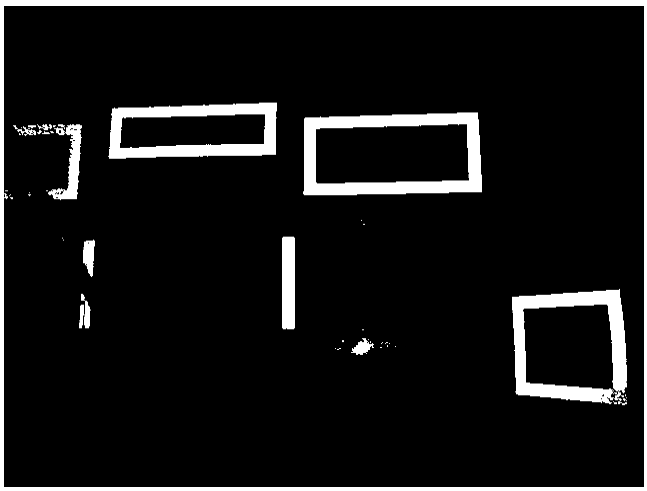
```
ret,dst = cv2.threshold(src, value, maxValue, cv2.THRESH_BINARY)
```

While other methods besides THRESH_BINARY exist in OpenCV, there is not a good application to use any of them in FRC.

Note that this is a high pass filter, and nothing more.

inRange

OpenCV's "inRange" function checks if $low < pixelvalue < high$, and if it is, then the pixel passes the test and is turned white, else it is turned black.



Note that this is identical to threshold's output because the parameters used made inRange behave the same. InRange is useful when thresholding for certain colors, as it is more than a simply high pass filter. JavaC++Python

```
Core.inRange(src, low, high, dst);
```

```
cv::inRange(src, low, high, dst);
```

```
ret, dst = cv2.inRange(src, low, high)
```

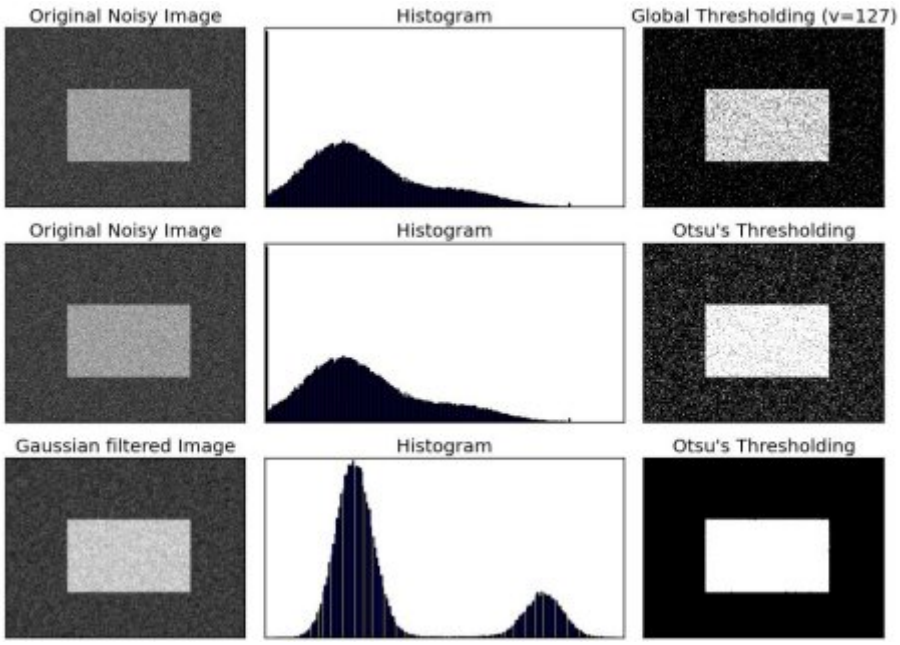
Otsu

Otsu thresholding is an old algorithm that is an adaptive thresholding technique. The algorithm assumes that the image contains two classes of pixels following a bi-modal histogram (foreground pixels and background pixels), it then calculates the optimum threshold separating the two classes so that their combined spread is minimal, or equivalently so that their inter-class variance is maximal.

Otsu's method exhaustively searches for the threshold that minimizes the intra-class variance, defined as a weighted sum of variances of the two classes:

$\sigma_w^2(t) = \omega_0(t)\sigma_0^2(t) + \omega_1(t)\sigma_1^2(t)$

Weights ω_0 and ω_1 are the probabilities of the two classes separated by a threshold t and σ_0^2 and σ_1^2 are variances of these two classes. JavaC++Python

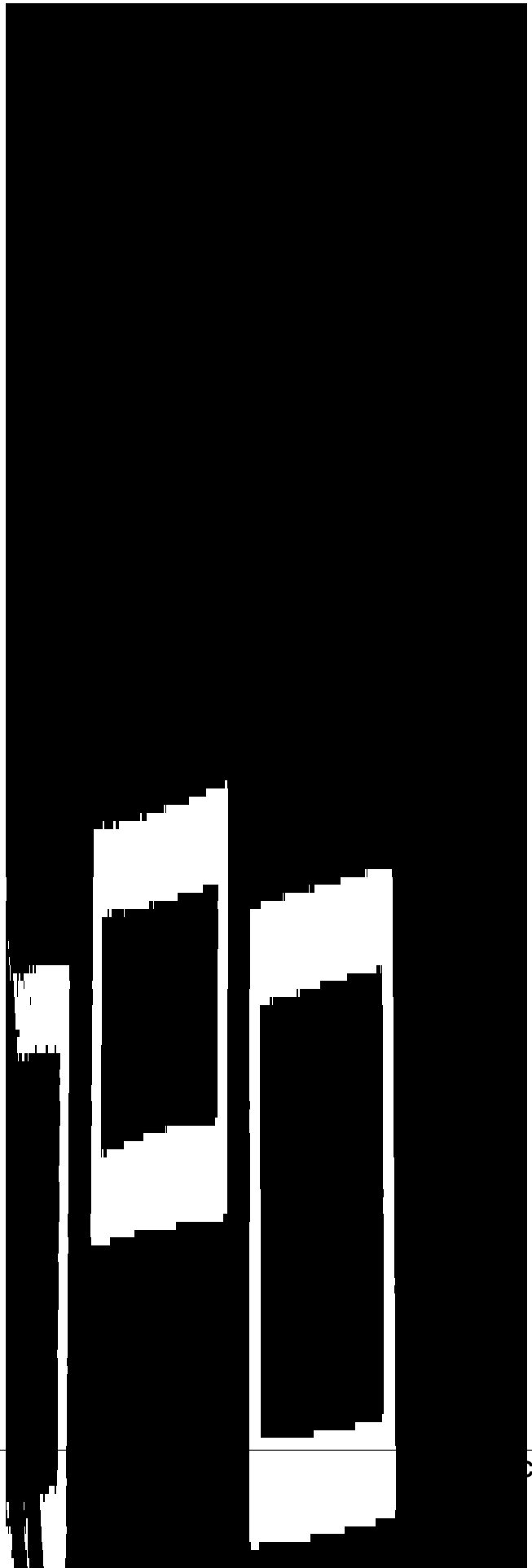



```
Imgproc.threshold(src, dst, 0, 255, Imgproc.THRESH_OTSU);
```

```
cv::threshold(src, dst, 0, 255, CV_THRESH_BINARY | CV_THRESH_OTSU);
```

```
ret2, dst = cv2.threshold(src ,0 , 255, cv2.THRESH_OTSU)
```

Otsu thresholding optimizes the upper and lower bounds, so 0 and 255 are simply placeholders as OpenCV doesn't use a separate function for Otsu thresholding.



In a typical FRC game, your environment is not drastically changing, so it is best practice to use inRange with hand tuned values instead of Otsu for speed purposes.

Thresholding with Color Images

Up until now, the examples have been with grayscale images. Color images are different in the fact that they have 3 channels instead of one, meaning that threshold values must be provided for each channel (color). This is a very slow and tedious process. To make it easier, use this program: https://github.com/rr1706/Multi-Thresh. This utilizes sliders that dynamically changes the threshold values for each color, and also allows the user to tune HSV images as well. Always use inRange when thresholding RGB images

The syntax for each language changes slightly, as observed: JavaC++Python

```
Core.inRange(src, new Scalar(low1, low2, low3), new Scalar(high1, high2, high3), dst);
```

```
cv::inRange(src, Scalar(low1, low2, low3), Scalar(high1, high2, high3), dst);
```

```
dst = cv2.inRange(src, np.array([low1, low2, low3]), np.array([high1, high2, high3]))
```

Using HSV Thresholding

HSV thresholding uses hue, saturation, and value to threshold images. Unlike RGB, HSV separates the image intensity, from the color information. This is very useful in computer applications such as vision tracking. In FRC, HSV is a great tool to detect the reflective vision tape if using a LEDs to illuminate the tape. It is also possible to use an IR camera with IR LEDs which would output a grayscale image. However, there are other threshold options that seperate image intensity with color but HSV is often used simply because the code for converting between RGB and HSV is widely available and easily implemented. Before using HSV to threshold the image, you must convert the image retrieved from the camera to the HSV colorspace (see code below). JavaC++Python

```
Imgproc.cvtColor(src, dst, Imgproc.COLOR_BGR2HSV);
```

```
cv::cvtColor(src, dst, CV_BGR2HSV);
```

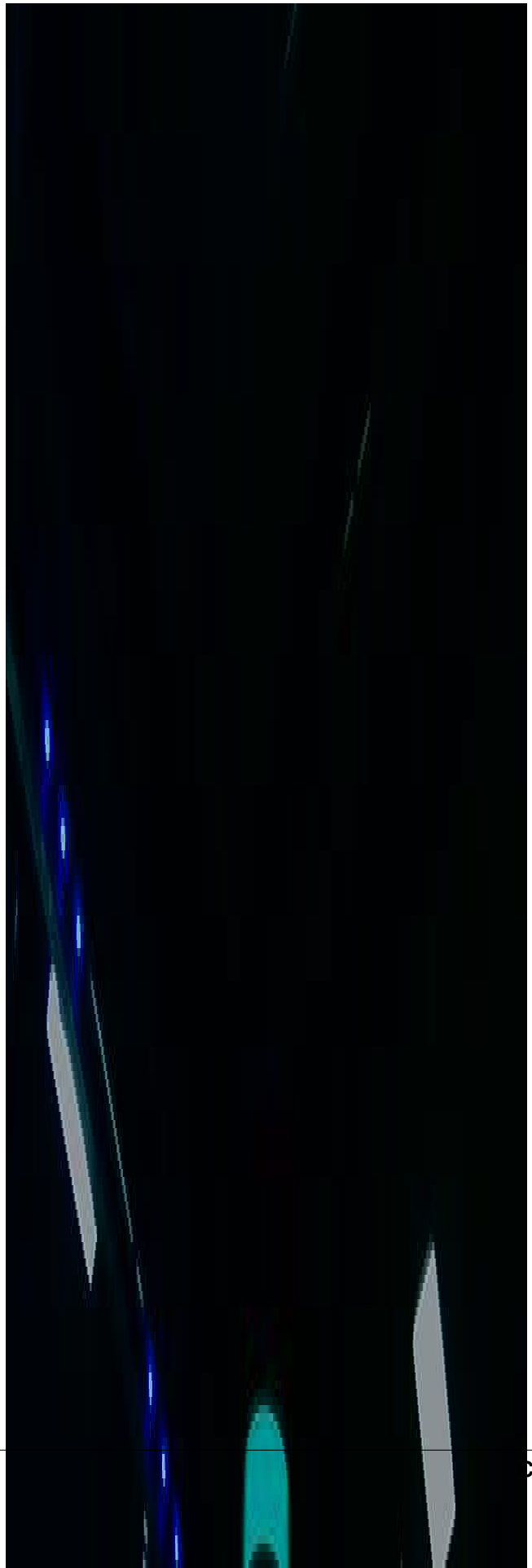
```
dst = cv2.cvtColor(src, cv2.COLOR_BGR2HSV)
```

Using Trackbars/Sliders for Real Time Tuning

As said above, sliders allow you to dynamically change HSV values, allowing you to fine tune the correct threshold values in real time. Here's how to create them. Note: Java does not support OpenCV to handle GUI so trackbars must be done with Swing and jsliders. More info on that here. C++Python

```
cv::namedWindow("Title of Window");
cv::createTrackbar("Title of slider", "Title of Window", &variable, highest number);
```

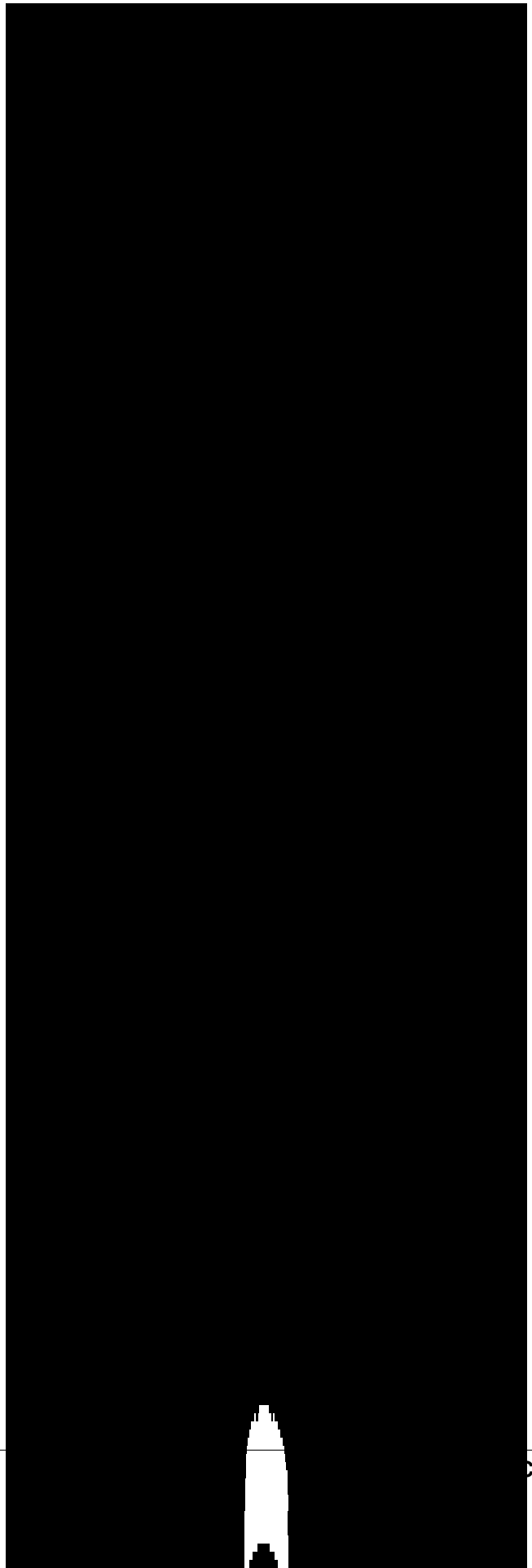
```
cv2.namedWindow('Title of Window')
cv2.createTrackbar('Title of Slider', 'Title of Window", 0, 255, nothing)
var = cv2.getTrackbarPos('title of slider', 'title of window');
```



Let's tackle an example. This is a pretty standard image that one might have if using green LEDs for the 2017 game.

The goal is to make the boiler tape white (255), and everything else black (0). By using the Multi-Thresh program, the RGB min and max values were found to be (0, 90, 0), (46, 255, 255), and they produce the following image:

If you find that you have noise, which is stray pixels, or if you thresholded away part of the inside of your target, please check out the morphological operations page.



CHAPTER 6

Morphological Operations

Binary images may contain noise (pixels that passed the initial threshold test but are not desired). Morphological image processing attempts to remove the noise of images while accounting for the form and structure of the image, as well as preserving the desired pixels' integrity. This guide dives into the math behind morphological operations, and then explains how they can be used in FRC.

Morphological image processing is a collection of non-linear operations related to the shape or morphology of features in an image. Morphological operations rely only on the relative ordering of pixel values and not on their numerical values, therefore making them especially suited to process binary images.

The core idea in binary morphology is to probe an image with a simple, pre-defined shape, and then draw conclusions on how this shape fits or misses the shapes in the image. This simple "probe" is called kernel, and is itself a binary image (i.e., a subset of the space or grid), The kernel dimensions are not limited to 3x3, but OpenCV limits them to be $N \times N$ where $N \in \mathbb{Z}_{odd}$, so here we will make the same restrictions.

Here are two common kernels used:

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

JavaC++Python

```
Mat kernel = Imgproc.getStructuringElement(Imgproc.RECT, new Size(3,3));
Mat kernel = Imgproc.getStructuringElement(Imgproc.MORPH_CROSS, new Size(3,3));
```

```
Mat kernel = cv::getStructuringElement(MORPH_RECT, Size(3,3) Point(-1,-1);
Mat kernel = cv::getStructuringElement(MORPH_CROSS, Size(3,3) Point(-1,-1);
```

```
kernel = cv2.getStructuringElement(cv2.MORPH_RECT,(3,3))
kernel = cv2.getStructuringElement(cv2.MORPH_CROSS,(3,3))
```

Defintion: Let E be a Euclidean space $\mathbb{R}^d$ or an integer grid $\mathbb{Z}^d$ for some dimension d, and A be a a binary image where $A \in E$.

Erosion

The erosion of the binary image A by the kernel B is defined by:

$A \ominus B = \bigcap_{b \in B} A_{-b}$. JavaC++Python

```
Imgproc.erode(src, dst, kernel);
```

```
cv::erode(src, dst, kernel, Point(-1,-1), 1);
```

```
erosion = cv2.erode(src, kernel, iterations=1)
```

Dilation

The dilation of A by the structuring element B is defined by:

$A \oplus B = \bigcup_{b \in B} A_b$. JavaC++Python

```
Imgproc.dilate(src, dst, kernel);
```

```
cv::dilate(src, dst, kernel, Point(-1,-1), 1);
```

```
dilation = cv2.dilate(src, kernel, iterations=1)
```

Properties of Morphologpical Operations

- They are translation invariant.
- Dilation is associative.
- Dilation is distributive over set union.
- Erosion is distributive over set intersection.
- Dilation is a pseudo-inverse of the erosion, and vice versa.

While understanding the set theory axioms of erosion and dilation is not necessary to understand them, they are still interesting.

Morophological Operations Playground

Open

Open is another name of erosion followed by dilation. It is useful in removing noise. JavaC++Python

```
Imgproc.morphologyEx(mFGMask,mFGMask,Imgproc.MORPH_OPEN,kernel);
```

```
cv::morphologyEx(src, dst, MORPH_OPEN, kernel, Point(-1,-1), 1);
```

```
opening = cv2.morphologyEx(src, cv2.MORPH_OPEN, kernel)
```

Close

Closing is reverse of Opening, Dilation followed by Erosion. It is useful in closing small holes inside the foreground objects. JavaC++Python

```
Imgproc.morphologyEx(mFGMask,mFGMask,Imgproc.MORPH_CLOSE,kernel);
```

```
cv::morphologyEx(src, dst, MORPH_CLOSE, kernel, Point(-1,-1), 1);
```

```
closing = cv2.morphologyEx(src, cv2.MORPH_CLOSE, kernel)
```

Morphological Gradient

It is the difference between dilation and erosion of an image. The result will look like the outline of the object. JavaC++Python

```
Imgproc.morphologyEx(mFGMask,mFGMask,Imgproc.MORPH_GRADIENT,kernel);
```

```
cv::morphologyEx(src, dst, MORPH_GRADIENT, kernel, Point(-1,-1), 1);
```

```
opening = cv2.morphologyEx(src, cv2.MORPH_GRADIENT, kernel)
```

Uses in FRC

FRC provides less than ideal environments for computer vision. Often times there is noise in your images that cannot be overcome by reducing the exposure of your camera and thresholding. When this occurs, consider using a morphological operation.

In many cases, however, these standard kernels will not suffice for FRC. Either erosion will remove too much of the target(s) or dilate will combine targets.

The following kernel will not alter the width of a pixel cluster, only the height. This is useful for when your targets are close together horizontally or if the target is not very tall and you must erode.

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

What is not pictured is the border on the top and left side of the image. this is a consequence of this kernel. Despite this consequence, this kernel may be very useful in many cases.

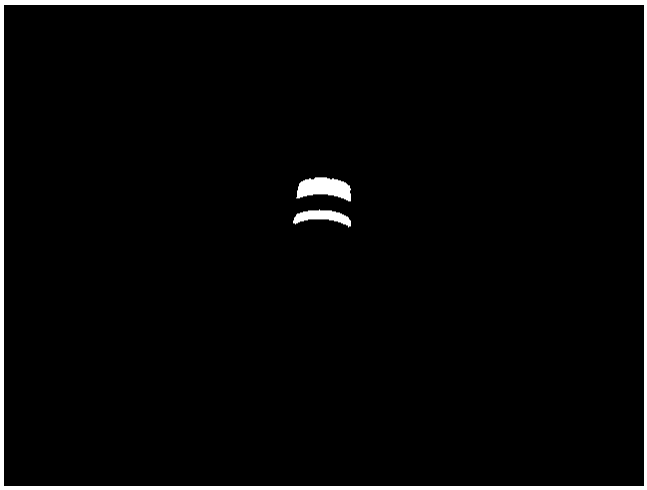
Likewise, this kernel will not alter the height of the pixel cluster, only the width, this is useful with targets who are close together vertically or if the target is not vey tall and you must erode.

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

CHAPTER 7

Contour Features

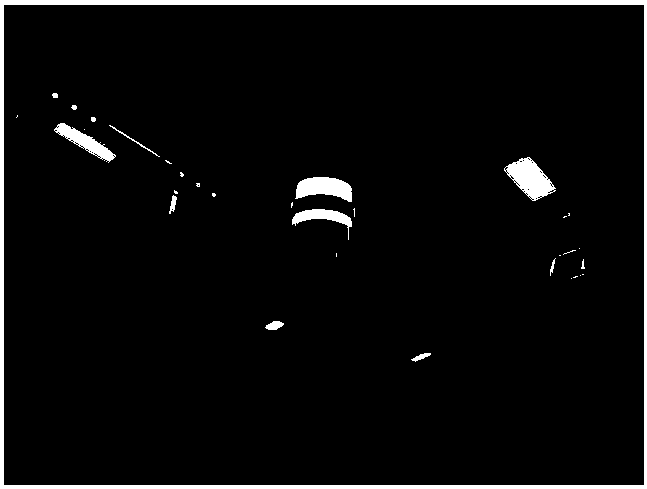
By this time, you should have an image that looks like this



but in all reality, your image probably looks something like this

We recommend you use "findcontours" for your edge detector. It is a rather old algorithm, but it is very effective as well as fast. Here is code that calls findcontours and draws them onto an image. Make sure that the image you are drawing to is RGB and not GrayScale if you want to draw things in color. JavaC++Python

```
List<MatOfPoint> contours = new ArrayList<MatOfPoint>();
Mat hierarchy = new Mat();
Imgproc.findContours(img, contours, hierarchy, Imgproc.RETR_EXTERNAL, Imgproc.CHAIN_
↪APPROX_NONE);
for (int i = 0; i < contours.size(); i++) {
        //...contour code here...
}
```



```
vector<vector<Point> > contours;
vector<Vec4i> hierarchy;
findContours(img, contours, hierarchy, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_NONE,␣
↪Point(0, 0));
for (size_t i = 0; i < contours.size(); i++)
{
        drawContours(draw, contours, i, Scalar(255, 0, 255), 3, 8, hierarchy, 0,␣
↪Point() );
}
```

```
img2, contours, hierarchy = cv2.findContours(img,cv2.RETR_TREE,cv2.CHAIN_APPROX_NONE)
for contour in contours:
    cv2.drawContours(draw, [contour], 0, (255, 0, 255), 3)
```

CHAIN_APPROX_NONE is a flag that tells findContours to store every contour point - including contours inside other contours, or multiple along the same line. If you don't have the strongest of computers (like a Pi), I recommend using CHAIN_APPROX_SIMPLE because it compresses horizontal, vertical, and diagonal segments and leaves only their end points.

Moving forward, this is where you must get creative. If you have contours that are not wanted, you must come up with tests that will pass on the contours that you want, but fail on the ones you do not wish to keep.

An implementation note: The following tests are meant to be implemented in a for loop, where the programmer is looping through every contour. JavaC++Python

```
for(int i = 0; i < contours.size(); i++ )
{
    //contours.get(i)....
}
```

```
vector<vector<Point> > contours;

for (size_t i = 0; i < contours.size(); i++)
{
    //contours[i] ....
}
```

```
for contour in contours:
    # do stuff with contour....
```

Contour Area

A quick and easy way to filter out small contours is to check their area. Establish a minimum and maximum area, which will probably have to be found empirically, and check to see if the contour's area falls within that range. JavaC++Python

```
double contourArea = Imgproc.contourArea(contour);
if(contourArea < min_area || contourArea > maxArea)
{
    continue;
}
```

```
float contourArea = contourArea(contours[i]);
if contourArea > maxArea || contourArea < minArea)
{
    continue;
}
```

```
contourArea = cv2.contourArea(contour)
if contourArea < minArea or contourArea > maxArea:
    continue
```

For code readability, I prefer to do that continue approach when looping through contours, instead of nested every contour test. It makes the code more readable, as well makes the programmer have to worry about counting an absurd amount of curly brackets.

Aspect Ratio

Aspect Ratio refers to the ratio between the contour's width / contour's height. To do this, one could find the extreme points, but it is easier to apply a bounding rectangle and then compute the ratio of the bounded rectangle. JavaC++Python

```
Rect boundRect = Imgproc.boundingRect(contour);
float ratio = (float)boundRect.width/boundRect.height
```

```
Rect boundRect = boundingRect(contours[i]);
float ratio = (float)boundRect.width/boundRect.height
```

```
x,y,w,h = cv2.boundingRect(contour)
ratio = float(w)/h
```

Remember to cast to float, otherwise integer division will occur and you won't get precise ratios.

Solidity

The last test we will cover is solidity. That is, the ratio between the contour area and the bounding rectangle area. This is useful to determine the "rectangle-ness" of the contour. The more closely the bounding rectangle fits the contours, the closer this ratio will be to 1. JavaC++Python

```
Rect boundRect = Imgproc.boundingRect(contour);
float ratio = Imgproc.contourArea(contour)/(boundRect.width*boundRect.height)
```

```
Rect boundRect = boundingRect(contours[i]);
float ratio = contourArea(contours[i])/(boundRect.width*boundRect.height);
```

```
x,y,w,h = cv2.boundingRect(contour)
ratio = cv2.contourArea(contour)/(w*h)
```

Finding the center

In typical FRC fashion, you want your robot to line up with the center of the target (contour). In order to do this, one must first find the center. While you could simply apply a bounded rectangle and then find the center of that, there is a more precise way: N-th order moments.

Mathematically, a moment is defined as $\mu_n = \int_{-\infty}^{\infty} (x-c)^n \, f(x) \, dx$. For a 2D continuous function f(x,y) the moment of order (p + q) is defined as $M_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x^p y^q f(x,y) \, dx \, dy$. The area of a contour is the zeroth moment, and moments can be used to find the centroid of a contour. JavaC++Python

```
Rect boundRect = Imgproc.boundingRect(contour);
float ratio = Imgproc.contourArea(contour)/(boundRect.width*boundRect.height)
```

```
Rect boundRect = boundingRect(contours[i]);
float ratio = contourArea(contours[i])/(boundRect.width*boundRect.height);
```

```
x,y,w,h = cv2.boundingRect(contour)
ratio = cv2.contourArea(contour)/(w*h)
```

Note that in previous years for FRC, there hasn't been a vision challenge where the vision assistance tape wasn't symmetrical. But in future if the tape isn't symmetrical, you would need to consider whether the centroid is what you desire.

Drawing

A very useful function is drawing the contours on your images. While you can draw every contour using the -1 flag, instead of the contour index, it is recommended you only draw the contours that pass all your tests. This allows for fast and effective debugging. JavaC++Python

```
for (int i = 0; i < contours.size(); i++)
{
    //Tests....
    Imgproc.drawContours(contourImg, contours, i, new Scalar(255, 255, 255), -1);
}
```

```
for (size_t i = 0; i < contours.size(); i++)
{
    //Tests...
    drawContours(draw, contours, i, Scalar(255, 0, 255), 3, 8, hierarchy, 0, Point()␣
↪);
}
```

```
for contour in contours:
    #Tests...
    cv2.drawContours(draw, [contour], 0, (255, 0, 255), 3)
```

Putting it all together

While other contour tests can be done, such as approximating a polygon around a contour, this a a good basis that should allow you to solve just about every FRC computer vision challenge. Here is a quick rundown of the tests needed to successfully solve the vision challenges over the years:

Year	Tests
2012	Area, Solidity
2013	Area, Solidity, Aspect Ratio
2014	Area, Solidity
2015	Area, Solidity
2016	Area, Solidity
2017	Area, Solidity

As you can see, nearly every FRC vision challenge can be solved with a simple area and solidity test.

Here is an example of what a final image for 2017 with contours drawn on the image might look like.