FRC Programming Done Right Documentation

Release 0.2

Tim Winters

Jul 14, 2018

Introduction to Programming


FRC
PROGRAMMINGDONERIGHT

CHAPTER 1

Code Structure

1.1 Structure of Your Robot Program

It would be wrong of me to say there is one right way of setting up your code. There is not. You can set up your code any way you would like, as long as you can understand the structure and effectively navigate your code. I am going to explain one structure of code that is useful for teams using SampleRobot, non-command IterativeRobot, PeriodicRobot, and MagicRobot. It is the same structure used by 5 time Innovation in Control Award recipient Team 1418 Vae Victis.

```
Robot/
    robot/
        autonomous/ # In this folder are separate autonomous modes
        components/ # Different components of the robot such as drive, shooter,␣
↪intake, etc.
        common/ # Parts of the robot that belong to all parts, such as drivers for␣
↪sensors.
        robot # Main Robot File
    tests/ # Tests for your code. Unit tests are very important.
```

For command based robots, the structure is pretty similar.

```
Robot/
    robot/
        commands/ # Commands that cause the robot to perform actions
        subsystems/ # Similar to components in Iterative. Subsystems such as drive,␣
↪shooter, intake, etc.
        robot # Main robot File
    tests/
```

1.1.1 RobotMap and OI

A common occurrence in robot code is a file called `RobotMap`. This file contains **constants** use throughout the robot. Such constants include motor controller port numbers, button mapping for certain robot functions, and PID

constants for your control loops. Many teams use RobotMap for keeping track of constants, but to me it makes sense for constants such as PID to be within their respective component classes, and just having the values in `robotInit` where you initialize all of your hardware.

`OI` goes hand in hand with `RobotMap`, since they both serve similar purposes. The main purpose is for all of your inputs (such as joysticks) to go into `OI` and the main robot program will call functions from `OI`. A simple setup will look like this (syntax aside)

```
teleopPeriodic() # Method inside main robot code file
    if OI.getShooter()
        shooter.spin()



# In OI file
getShooter()
    return Joystick.getRawButton(1)
```

The use of an extra file for some more readability may be a worthwhile tradeoff. Talk with your team to decide if you want to use this form of structuring your code. Both ways are fine, but the most important thing is to be consistent. It is actually *less* helpful to only have *some* of the joystick inputs in `OI` than to have no `OI` file at all.

1.1.2 Main Robot Structure

You know the old saying "Cleanliness is close to godliness". The same goes for programming. Clean code -> God Tier code. The first step to having clean code is to have good organization. Let's start with `robotInit`.

robotInit

Note: The following examples show IterativeRobot, however the same logic an be applied to Sample and Periodic.

Java

C++

Python

```
public class MyRobot(wpilib.IterativeRobot) {

    Joystick joystick1, joystick2;
    Drive drive;

    public void robotInit(){
        joystick1 = new Joystick(0);
        joystick2 = new Joystick(1);
        drive = new Drive(new Talon(1), new Talon(2));
    }
}
```

```
class MyRobot : wpilib.IterativeRobot{
    Joystick joystick1, joystick2;
    Talon motor1, motor2;
    Drive drive;
public:
```

(continues on next page)

(continued from previous page)

```
    void robotInit(){
        joystick1 = Joystick(0);
        joystick2 = Joystick(1);
        drive = Drive(Talon(1), Talon(2));
    }

}
```

```
class MyRobot(wpilib.IterativeRobot):

    def robotInit(self):
        self.joystick1 = Joystick(0)
        self.joystick2 = Joystick(1)
        self.motor1 = Talon(1)
        self.motor2 = Talon(2)
        self.drive = Drive(motor1, motor2)
```

Let's talk about what's happening in these methods. In the Java and C++ examples, the code starts with declaring the variables in the class scope (outside of any method). This allows the other methods you will use such as `teleopPeriodic` or `operatorControl` to have access to your robot components.

Inside `robotInit` is where we actually initializing the variables. There is no real significance to doing this inside `robotInit` or when you declare the variables except structure, which is what we're going for. Also notice how I never created variables for the drive motors. If you aren't going to use the variables outside of the drive class, there is no need to declare them as variables here. It makes more sense to declare them as variables inside the drive class, where you can customize them (such as setting PID if they are CANTalons, or reversing them if need be).

Note: If you are using RobotMap, this is where the values stored in RobotMap would be used. Instead of `joystick1 = new Joystick(0)` you might do `joystick1 = new Joystick(RobotMap.LEFT_JOYSTICK)`

Teleop

Now that you've created all of the robot components, we can focus on teleop code. The main basis of teleop code is using `if` statements to check for input, and the performing some action based on these events. For example

```
drive.drive(joystick.getY(), joystick.getX())

if joystick.getRawButton(1)
    shootBall()

if joystick.getRawButton(2)
    intakeBall()

if joystick.getRawButton(3)
    climb()
```

This structure allows for easy configuration of joystick -> action. The drive code shouldn't involve an if statement, since you always want control over the drivetrain, and you should call the command that drives every loop. You probably want it to be at the top, that way if you have any code that edits the drive (such as angle rotation code) the values will not get overwritten by the joysticks.

1.1.3 Components

The components should be made up of setters, getters, and an execute method. The setters will be used to set variables used in the execute method. A good example is a function *move* in the drive class that sets the *fwd* and *rot* variables. These variables can then be used in the execute method to set motors. In order for this structure to work, it is crucial that the only place motors, relays, etc. get set is in the execute method. This prevents different parts of the robot overwriting each other. Here's an example of a move function in the drive class.

```
function move(fwd, rot):
    global fwd = fwd
    global rot = rot


function execute():
    DifferentialDrive.arcadeDrive(fwd, rot)
```

The reason for the setters setting variables and then an execute method passing those variables to the motors is to prevent 'race conditions'. Essentially, imagine you have two buttons on your joystick. One is to set a motor to full forward, the other, full reverse. If in your code you had

```
if button1:
    component.setFullForward()

if button2:
    component.setFullReverse()
```

then as the code looped, it would constantly switch the motors between forward and reverse. Now you could you an else if loop, but it can be annoying to manage precedence like this. Using verb methods allows every button to affect the outcome, but only the last one to actually show on the robot. This is helpful if you create autonomous commands that interact like a human. You can put them all the way at the bottom or top, and guarantee they either always take precedence, always yield, or a mix of the two.

CHAPTER 2

Debugging

Debugging. More commonly known as that thing you don't need to do because your code works always. Debugging is a natural part of the programming process, but knowing the best way to do it can be a challenge. Mainly because there isn't a 'right way' to do it.

2.1 Print statements

Print statements allow for quick and dirty debugging. They can be used to determine where in the code execution stops by placing many of them in succession in code. They can be as simple as printing out the letter 'A', then 'B' a few lines later, etc. or you can put in useful statements. This type of debugging is meant to be short-lived, and should be removed once the problem has been fixed.

2.2 Loggers

Loggers can be used for debugging, but are more long term. Loggers are usually built into the language you're using because they require a bit more code than just print statements. Loggers generally include timestamps, as well as possible stacktraces, and can write to files more easily.

2.2.1 Filters

Filters are also an integral part of logging. They allow you to..well... filter what gets logged. This can be used to only allow logs of certain degrees of severity

Java

C++

Python

```
SEVERE
WARNING
INFO
CONFIG
FINE
FINER
FINEST
```

```
// C++ doesn't include builtin logging. You will have to look online for some or␣
↪create your own.
```

```
ERROR   40
WARNING 30
INFO    20
DEBUG   10
NOTSET  0

Python log levels have numerical values associated, so setting the log level to 30 is␣
↪the same as setting it to WARNING
```

By filtering out certain level codes, you can have logs that only show the most important info, and therefore you can sift through them faster, giving you more time to fix any bugs in the code.

2.3 NetConsole

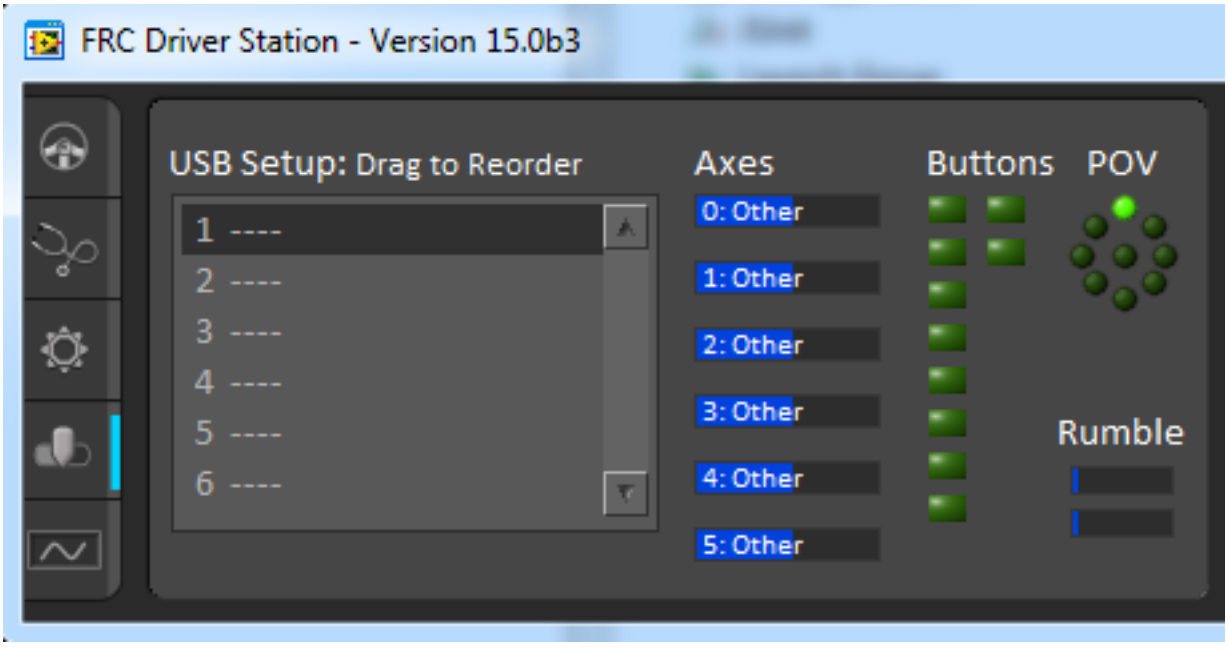
When you output anything to the console (via print statements to stdout) it will show up in both the log viewer on the dashboard, as well as on NetConsole. You can connect your computer to a RoboRIO and open the console viewer (using either the NI tool or pynetconsole) to view the console remotely.

CHAPTER 3

Controllers

3.1 Determining Joystick Mappings

One way to determine joystick mapping is by writing robot code to display axis and button values via the dashboard or console, loading it on the robot, then testing the joystick. A simpler way is to use the Driver Station. The 2015 FRC Driver Station contains indicators of the values of axes buttons and the POV that can be used to determine the mapping between physical joystick features and axis or button numbers. Simply click the joystick in the list to select it and the indicators will begin responding to the joystick input.



3.2 Xbox Controller

When using an Xbox controller, it can be a pain to determine the mappings yourself. Thankfully, WPILib has already done that, and put them into a class called XboxController

CHAPTER 4

Joystick Utilities

4.1 Toggles

If you want to be able to turn on a system with the push of a button, but not have to hold the button the entire time (but have the option to), or push a different button to turn it off, you would want a toggle. The concept of it is simple, press the button, it turns on, press it again, it turns off. The execution of it is slightly more difficult, requiring a few variables to store the current state of the toggle.

Java

C++

Python

```
public class MyRobot extends IterativeRobot{
    public void robotInit(){
        Joystick joystick = new Joystick(0);
    }

    boolean toggleOn = false;
    boolean togglePressed = false;

    public void teleopPeriodic(){
        updateToggle();

        if(toggleOn){
            // Do something when toggled on
        }else{
            // Do something when toggled off
        }
    }

    public void updateToggle()
    {
        if(joystick.getRawButton(1)){
```

(continues on next page)

(continued from previous page)

```
            if(!togglePressed){
                toggleOn = !toggleOn;
                togglePressed = true;
            }
        }else{
            togglePressed = false;
        }
    }
}
```

```
//This still needs to be done. If you'd like to do it, fork the github repository at
↪https://github.com/FRC-PDR/ProgrammingDoneRight
```

```
'''
NOTE: Uses robotpy_ext/control/toggle.py, which isn't
merged with the latest version of robotpy yet (v2017.1.5)
'''
class MyRobot(wpilib.IterativeRobot):
    def robotInit(self):
        self.joystick = wpilib.Joystick(0)
        self.toggle = Toggle(self.joystick, 0)

    def teleopPeriodic(self):
        if self.toggle:
            # Do something when button pressed
        if self.toggle.on:
            # Do Something when toggled on
        if self.toggle.off:
            # Do Something when toggled off
```

It is probably helpful to extend the toggle logic to a class, that way you can create many different toggle buttons without having repeating code. For an example of this, look at the robotpy version.

4.2 Debouncers

When you get a joystick button input, sometimes the mechanical switch will bounce and register one press as 2 hits. To fix this, you should use something called a **Debouncer**. This will make it so the button is only registered as pressed once, making it much easier to control your inputs. Debouncers are also useful when you want a surefire way of only sending one pulse (instead having to press and release really quickly).

Java

C++

Python

```
public class MyRobot extends IterativeRobot{
    public void robotInit(){
        Joystick joystick = new Joystick(0);
        ButtonDebouncer debouncer = new ButtonDebouncer(joystick, 1, .5);
    }


    public void teleopPeriodic(){
```

(continues on next page)

(continued from previous page)

```
        if(debouncer.get()){
            System.out.print() // This print statement will only get called every .5
↪seconds
        }
    }
}

public class ButtonDebouncer(){

    Joystick joystick;
    int buttonnum;
    double latest;
    double debounce_period;

    public ButtonDebouncer(Joystick joystick, int buttonnum){
        this.joystick = joystick;
        this.buttonnum = buttonnum;
        this.latest = 0;
        this.debounce_period = .5;
    }
    public ButtonDebouncer(Joystick joystick, int buttonnum, float period){
        this.joystick = joystick;
        this.buttonnum = buttonnum;
        this.latest = 0;
        this.debounce_period = period;
    }

    public void setDebouncePeriod(float period){
        this.debounce_period = period;
    }

    public boolean get(){
        double now = Timer.getFPGATimestamp();
        if(joystick.getRawButton(buttonnum)){
            if((now-latest) > debounce_period){
                latest = now;
                return true;
            }
        }
        return false;
    }
}
```

```
class MyRobot(wpilib.IterativeRobot){

public:
    ButtonDebounce debouncer (joystick, 1, .5)
    public void teleopPeriodic()
    {
        if debouncer.get(){
            cout << endl; // This print line will only get called every .5 seconds
        }
    }
}
class ButtonDebouncer{
```

(continues on next page)

(continued from previous page)

```
    Joystick joystick;
    int buttonnum;
    double latest;
    double debounce_period;

public:
    ButtonDebouncer(Joystick joystick, int buttonnum){
        this.joystick = joystick;
        this.buttonnum = buttonnum;
        this.latest = 0;
        this.debounce_period = .5;
    }
    ButtonDebouncer(Joystick joystick, int buttonnum, float period){
        this.joystick = joystick;
        this.buttonnum = buttonnum;
        this.latest = 0;
        this.debounce_period = period;
    }

    void setDebouncePeriod(float period){
        this.debounce_period = period;
    }

    bool get(){
        double now = Timer.getFPGATimestamp();
        if(joystick.getRawButton(buttonnum)){
            if((now-latest) > debounce_period){
                latest = now;
                return true;
            }
        }
        return false;
    }
}
```

```
from robotpy_ext.control import ButtonDebouncer
class MyRobot(wpilib.IterativeRobot):

    def robotInit(self):
        self.joystick1 = wpilib.Joystick(1)
        # Joystick object, Button Number, Period of time before button is pressed␣
↪again
        self.button = ButtonDebouncer(self.joystick, 1, period=.5)

    def teleopPeriodic(self):
        if self.button.get():
            print() # This print statement will only get called every .5 seconds
```

CHAPTER 5

Driving Straight

A common problem in FRC is driving straight, in autonomous mode or otherwise. This article describes two ways to accomplish this using either encoders or a gyro. Recommended prerequisite reading is PID Control.

5.1 Using a Gyro

See the Gyro article for gyro basics. A gyro automatically corrects your turn as you drive. A simple way to accomplish this is using a P loop in your drive routine.:

```
function drive_straight_gyro(power):
    error = -gyroAngle  # Our target angle is zero
    turn_power = kP * error
    drive.arcadeDrive(power, turn_power, squaredInputs=False)
```

This works well without much oscillation at speed because most of the nonlinearities in a drivetrain are taken up by the main drive power. However, at low speed, small P values may not correct as well. This can be fixed by "gain scheduling", which is using different values of kP/kI/kD for different situations, or by adding terms to the P loop.

Note that this example is similar to the Gyro article's rotate to angle function but with the target angle set to 0. Depending on the robot, field layout, direction of travel, drift, and more, your gyro may not be zero when you want it to be. A good idea would be to zero the gyro reading before starting the drive straight routine.

Here we use the DifferentialDrive arcadeDrive function because it does the straight/rotate math for us. We also pass an argument in telling it not to square the inputs. This is so the output is linear.

5.2 Using Encoders

How this works conceptually is by using a PID loop to drive the *difference* between the left and right side encoders to zero.:

```
function drive_straight_enc(power):
    error = left_encoder - right_encoder
    turn_power = kP * error
    drive.arcadeDrive(power, turn_power, squaredInputs=False)
```

Make note of the sign of *error* and make sure that it reacts in the same direction as your turn direction, otherwise you will continually turn.

Again, this works well at high speeds, but may have difficulty correcting small errors at low speed. Also similarly to the gyro, the difference may start at something that's not zero, so it's a good idea to zero both those before starting the drive routine.

5.3 What to use?

Both these approaches work well. A gyro is a fantastic sensor for angle measurement, especially over short terms. As turning to a specific angle is an almost-as-if-not-more-common problem than driving straight, the same sensor can easily double up for both tasks.

Encoders are less useful for turning to an angle (though technically usable), so a gyro is the recommended approach. However, again, encoders are acceptable if you do not have a gyro on your robot for whatever reason.

Both of these approaches can be combined with other drive methods, such as driving to a distance or using PID for drive distance.

CHAPTER 6

Gyros

Probably the most critical sensor on any FRC robot is the gyro. The gyro (short for gyroscope) is used to measure heading, or which direction the robot is pointing. Gyros can be used to get your robot to point at a certain angle, as well as keeping an angle while driving.

6.1 Rotating to an Angle

In order to rotate to an angle, we are going to use PID. There are two ways of implementing this. We can use WPILib's builtin PIDController class, or we can create our own calculator. Because rotating to an angle can be done with a simple P coefficient, we'll create our own. Here's the

```
function calculateRotateValue(targetAngle):
    error = targetAngle - gyroAngle
    if error > threshold
        rotation =  error*kP
        return False
    else:
        rotation = 0
        return True
```

This function will take a target angle, calculate the error, and if it's within a threshold, it will return True, signaling that the robot is pointed where it needs to be. If you follow the structure of code shown in [structure] and have a class dedicated to drive, this function should go in that class.

Note: A good threshold value is 1-3 degrees, since friction will make it hard for the robot to move only one degree without overshooting.

Let's see how to use this in pseudocode

```
function rotateToAngle(targetAngle):
    error = targetAngle - gyroAngle # check out wpilib documentation for getting the␣
↪angle from the gyro
```

(continues on next page)

(continued from previous page)

```
    if error > threshold
        this.rotation =  error*kP
        return False
    else:
        this.rotation = 0
        return True

function move(fwd, rotation):
    // This function allows for joystick input
    this.fwd = fwd
    this.rotation = rotation

function execute():
    // Execute function that should be called every loop
    this.robotdrive.drive(this.fwd, this.rotation)

    this.fwd = 0
    this.rotation = 0
```

In this example, you would want to call execute every iteration of teleop or autonomous. To rotate to 90 degrees:

```
drive.rotateToAngle(90)
drive.execute()
```

To drive straight while using joysticks:

```
drive.move(joystick.Y, joystick.X) // You can still use the X joystick here
drive.rotateToAngle(gyro.currentAngle) // It gets overwritten here
drive.execute()
```

This behavior works in autonomous as well. Just give the move values (and a value of 0 for rotation) and then call your rotate function. See Driving Straight for a more thorough discussion of this topic.

CHAPTER 7

PID Control

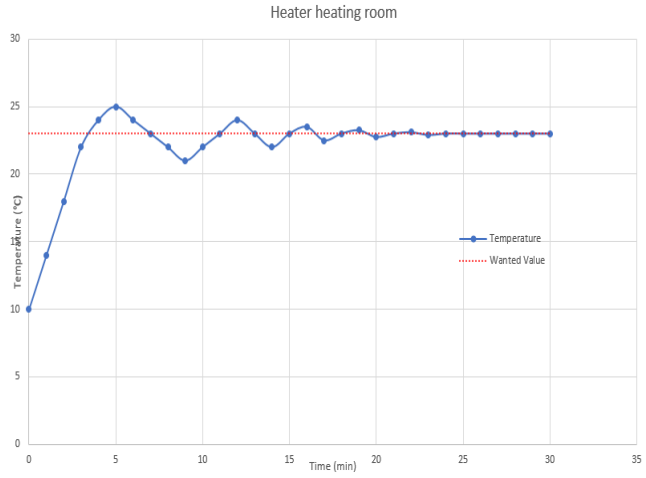
PID Proportional, Integral, Derivative feedback loop

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

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

7.1 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

7.2 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.

7.3 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.

7.4 Feed-Forward

The most essential part of a good control loop is a well tuned feed-forward. Feed-forward accounts for the known dynamics of the system, whereas feedback accounts for deviations from the known behavior like some friction and minor changes in weight. Feed-forward models can be derived using physics principles like torque and gravity in conjunction with motor characteristics.

7.4.1 Flywheel

The most simple feed-forward model is the "flywheel" model, where a static voltage is always applied to get the motor to spin at a constant speed. For example, if your motor's maximum RPM is 6000 RPM at 12V, then you should apply $\frac{12}{6000} \times Setpoint$ volts to get the motor to spin at `Setpoint` RPM.

7.4.2 Arm affected by Gravity

Another common model is the rotational arm. The model can be derived using the effect of gravity on the arm. The result of this calulation is the voltage theoretically required to keep the arm perfectly static.

$A \cos \theta$

In this calculation, θ is the angle of the arm above the horizontal. A has units of volts and can be found by using the motor's stall and free torque in combination with the weight and length of the arm. Derivation and constant determination can be found in this post.

7.4.3 Cascade Elevator

With this model, a static voltage is added to the output to oppose the force of gravity. The amount of voltage can be determined using the weight of the elevator, the motor torque, and spool radius.

7.4.4 Drivetrain

With the robot drivetrain, many nonlinear factors should be considered because of the large mass and high friction. These terms are kF (static voltage for friction) kV (volts per speed) and kA (volts per acceleration). This model can be used to improve following motion profiles as well as improve open loop control. Team 449's paper on FRC drive characterization shows the derivation of these terms as well as an empirical method of determining them.

7.5 Using PID on your robot

Now that you know the math behind PID, it's time to implement it with your robot. There are two ways to do this. One is to create the PID calculaitons yourself, the other is to use WPILib's PIDController. Let's talk about both

Let's create an example drive class

Java

C++

Python

```
public class Drive(){
    int P, I, D = 1;
    int integral, previous_error, setpoint = 0;
    Gyro gyro;
    DifferentialDrive robotDrive;


    public Drive(Gyro gyro){
        this.gyro = gyro;
    }

    public void setSetpoint(int setpoint)
    {
        this.setpoint = setpoint;
    }

    public void PID(){
        error = setpoint - gyro.getAngle(); // Error = Target - Actual
        this.integral += (error*.02); // Integral is increased by the error*time␣
↪(which is .02 seconds using normal IterativeRobot)
        derivative = (error - this.previous_error) / .02;
        this.rcw = P*error + I*this.integral + D*derivative;
    }

    public void execute()
    {
        PID();
        robotDrive.arcadeDrive(0, rcw);
    }
}
```

```
class Drive:
{
    int P, I, D, error, setpoint, rcw;
    public:

    Drive(){
        P, I, D = 0;
        error, setpoint, rcw = 0;

    }

    void setSetpoint(int setpoint){
        this.setpoint = setpoint;
    }
```

(continues on next page)

(continued from previous page)

```
    void PID(){
        error = setpoint - gyro.getAngle() // Error = Target - Actual
        this.integral += (error*.02) // Integral is increased by the error*time
↪(which is .02 seconds using normal IterativeRobot)
        derivative = (error - this.previous_error) / .02
        rcw = P*error + I*self.integral + D*derivative
    }

    void execute(){
        PID();
        robotDrive.arcadeDrive(0, rcw);
    }
}
```

```
class Drive:

    def __init__(leftMotor, rightMotor, gyro):
        self.gyro = gyro
        self.setpoint = 0
        self.robotDrive = wpilib.drive.DifferentialDrive(leftMotor, rightMotor)

        # PID Values
        self.P = 1
        self.I = 1
        self.D = 1

        self.integral = 0
        self.previous_error = 0


    def setSetpoint(self, setpoint):
        self.setpoint = setpoint

    def PID(self):
        """PID for angle control"""
        error = self.setpoint - self.gyro.getAngle() # Error = Target - Actual
        self.integral = integral + (error*.02)
        derivative = (error - self.previous_error) / .02
        self.rcw = self.P*error + self.I*self.integral + self.D*derivative


    def execute(self):
        """Called every iteration of teleopPeriodic"""
        self.PID()
        self.robotDrive.arcadeDrive(0, self.rcw)
```

7.6 Tuning Methods

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.

7.7 Which ones to use

Feedforward control is necessary on all but the absolute simplest of systems. It's incredibly difficult to get a good response without a feedforward calculation.

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 8

Using WPILib's PID Controller Class

You need an understanding of PID Theory to understand this article. If you don't already understand PID, I would recommend looking at the previous PID Control article

This article was written primarily for java. Most of the details are the same between java and C++, however some details may be different. This article is not applicable to python.

So, you now know all about PID and it's control theory magic, and would like to run some PID on your robot! Well, you've come to the right place, because this article is just for that. This article explains how you can use the inbuilt WPILib PID Controller class for all your PID needs.

> Note: If you are using the Talon SRX motor controller, you can use it's in-built PID feature to run PID *on the Talon*, if you are wired up using CAN. The Talon is able to run it's PID loop faster, resulting in better control. The Talon SRX user manual has details on how to set this up.

8.1 Advantages of the WPILib PID Controller Class

It is often tempting to roll your own PID code to control your motors, however using the WPILib version offers several advantages, mainly:

Threading The PID Controller class utilizes threading so the PID loop runs much faster, giving it much better response and resulting in better control.

Extra functionality The PID Controller class provides more than just a basic PID loop. For example, PID Controller provides ways to set maximum and minimum output's, wrap endpoints around, and give tolerances for the setpoint.

Pre-exisiting tested nature The PID Controller class has alread been written, tested and debugged. Doing it your self dosen't provide that certainty, and will near definitely take longer.

8.2 Disadvantges of the WPILib PID Controller Class

Could be better control-theory wise While the WPILib PID Controller class is quite good, for advanced teams, its lack of an acceleration term for example, means that it could be improved upon. But, if this is why you don't want to use it, then you probably don't need this article.

Potentially slow(er) on the RIO As mentioned earlier, the PID Controller class must run on the RoboRIO. Other alternatives, like using the Talon SRX's inbuilt PID controller, will operate faster and so with better control.

Ramping motors up The PID Controller class can result in very jerky motion which can be bad for geraboxes. However, increasing the D term should smooth out sudden changes. Alternatively, see "Adding Ramping for Motors".

Linear output assumption The PID Controller, like nearly all versions of PID, assumes a linear output, however in reality motor responses are curves.

8.3 Implementing a basic PID Control

PIDController Javadoc PIDController C++ Reference

1. **create a new instance of a PIDController. In the full / largest constructor, the values are:**

Constructur var name	Explanation
double Kp	The P term constant. See the PID Theory article if you don't understand this. Set to 0 to not use P control.
double Ki	The I term constant. See the PID Theory article if you don't understand this. Set to 0 to not use I control.
double Kd	The D term constant. See the PID Theory article if you don't understand this. Set to 0 to not use D control.
double Kf	The F term constant for feedforward control. See the PID Theory article if you don't understand this. Set to 0 to not use Feedforward control. See Velocity PID Control.
PIDSource source	The input device to the PID loop. For example, an encoder or gyro. Note that this must implement PIDSource. WPILib's Gyro and Encoder classes implement PIDSource. If you've written your own class, or are using a 3rd party gyro class (say for a NavX), you may need to implement PIDSource yourself.
PIDOutput output	The output device for the PID loop. For example, a motor controller. Note that, like PIDSource, you must pass the object itself. Note that it must implelement PIDOutput. Anything that implelements SpeedController implelements PIDOutput, so nearly all motor controllers do. But if you use your own class, or a 3rd party class (like the CANTalon), then you might need to implement it yourself.
double period	How often the PID loop should run. Defaults to 50ms.

2. **Set options you want (see options of PID Control)**
    - Mandatory options: setSetpoint, setTolerance(or the %/abs versions)
    - Optional ones: setContinuous, setInputRange
3. enable
4. You can set options - such as the PID constants, setpoints, ranges, etc. while the PIDLoop is running - you may want to call reset() if you do though (particularly if you change te setpoint)
5. disable
6. free, if you want to clear up the memory and are done with the PIDController

8.4 Options of PID Control

Function/option	Explanation
disable	Sets output to zero and stops running.
enable	Starts running the PID loop.
free	Sets all it's variables to null to free up memory.
reset	
setInputRange	Set's the minimm and maximum values expected from the input. Needed to use setContinuous.
setOutputRange	Set's minimum and maximum output values. Should also constrain the totalError I integral.
setContinuous	Treats the input ranges as the same, continuous point rather than two boundaries, so it can calculate shorter routes. For example, in a gyro, 0 and 360 are the same point, and should be continuous. Needs setInputRanges.
setPID	Set's the P,I,D,F constants.
setSetpoint	Set's the target point for the PID loop to reach.
setTolerance	Let's you implemenet your own Tolerance object. PidController.onTarget() will return True when the Tolerance object returns True - for example to let you to know to disable the PID loop and end the command.
setAbsoluteTolerance	Makes PIDController.onTarget() return True when PIDInput is within the Setpoint +/- the absolute tolerance.
setPercentTolerance	Makes PIDController.onTarget() return True when PIDInput is within the Setpoint * (+/- the percent tolerance).
setToleranceBuffer	Sets the number of previous error samples to average for tolerances before onTarget() will become True, so you don't get a false true if it is temporarily within the tolerance or has a noisy sensor.

8.5 Velocity PID Control

To use PID Controller to maintain a velocity - say for a shooter fly wheel or closed loop driving:

- You should use a feedforward term (Kf)
- Your PIDSource should probably have a PIDSourceType of kRate

- Be careful of what your PIDSource is giving - for example, if you use an encoder, and it gives encoder positions, but you want speed, then you might need to wrap it with your own code that gives the rate of change instead.

8.6 Using PID Subsystem

WPILib provides the PID Subsytem class to provide convenience methods to run a PIDController on a subsytem for simple cases. For example, if you had an elevator subsytem that needed to stay at the same height, you could use a PIDSubsystem for that.

To use, rather than extending Subsystem, extend PIDSubsytem.

You will need to define the functions returnPIDInput and usePIDOutput to give to the PIDController, and you will want to in the constructor for your subsytem call:

```
super(name, p, i, d, f, period)
```

You can access the internal PIDController with getPIDController()

Example PIDSubsystem to control the angle of a wrist join (taken from WPI's FRC Control System Screensteps live)

```
1  package org.usfirst.frc.team1.robot.subsystems;
2  import edu.wpi.first.wpilibj.*;
3  import edu.wpi.first.wpilibj.command.PIDSubsystem;
4  import org.usfirst.frc.team1.robot.RobotMap;
5
6
7  public class Wrist extends PIDSubsystem { // This system extends PIDSubsystem
8
9          Victor motor = RobotMap.wristMotor;
10         AnalogInput pot = RobotMap.wristPot();
11
12         public Wrist() {
13                 super("Wrist", 2.0, 0.0, 0.0);// The constructor passes a name␣
   ↪for the subsystem and the P, I and D constants that are useed when computing the␣
   ↪motor output
14                 setAbsoluteTolerance(0.05);
15                 getPIDController().setContinuous(false); //manipulating the raw␣
   ↪internal PID Controller
16         }
17
18         public void initDefaultCommand() {
19         }
20
21         protected double returnPIDInput() {
22                 return pot.getAverageVoltage(); // returns the sensor value that␣
   ↪is providing the feedback for the system
23         }
24
25         protected void usePIDOutput(double output) {
26                 motor.pidWrite(output); // this is where the computed output␣
   ↪value fromthe PIDController is applied to the motor
27         }
28 }
```

8.7 Explanation of the various PID WPILib class's

These are all found at edu.wpi.first.wpilibj, except for PIDSubsystem which is at edu.wpi.first.wpilibj.command

PID WPILib Class	Function/role
PIDController	The main PID Class that runs your PID loop and has been referenced many times in this article.
PIDSubsystem	See
PIDInterface	A generic PID interface with generic methods. Extends controller. If you wanted you could implement this if you made your own PID Controller.
PIDOutput	An interface for the function PIDWrite to be implemented by an output device such as a motor.
PIDSource	An interface to be implemented by input sensors.
PIDSourceType	An enum for the two types of PIDSources - Displacement and Rate.

8.8 Adding Ramping for motors

As mentioned earlier, the best way is generally to increase your D term as it will smooth out sudden changes. However, alternative options, if for some reason you could not change your D term:

- Create a wrapper function for PIDWrite that dampens motors. This function would store the previous output to the motor, and if given a new output that was say greater than 0.2 higher, it would only increase it by 0.2, and then increase it more after a brief wait. Note that this will reduce the effictiveness of your control, and will most likely mess up the I term of the PID loop
- Dynamically change the minimum / maximum values of your PID Controller. Say, whever PIDWrite get's called, change the PIDController's maximum and minimum values to be around a certain band. This is basically the first option, but a bit better as it will limit the I term and stop it from going crazy.

Dampener function

```
public class PIDMotor implements PIDOutput
{
        /** The motor that will be set based on the {@link PIDController} results. */
        public PWMSpeedController motor;
        private double previousOutput = 0.0;
        private double rampBand;
        private double output;

        /**
         * Constructor for a PID controlled motor, with a controllable multiplier.
         *
         * @param motor The motor being set.
         * @param rampBand The acceptable range for a motor change in one loop
         */
        public PIDMotor(PWMSpeedController motor, double rampBand) {
                this.motor = motor;
                this.rampBand = rampBand;
        }

        public void pidWrite(double pidInput) {
```

(continues on next page)

(continued from previous page)

```
                if (Math.abs(pidInput - previousOutput) > rampBand) { //If the change␣
↪is greater that we want
                        output = pidInput - previousOutput > 0 ? previousOutput +␣
↪rampBand : previousOutput - rampBand; //set output to be the previousOutput␣
↪adjusted to the tolerable band, while being aware of positive/negative
                }
                else {
                        output = pidInput;
                }
                motor.set(output);
                previousOutput = output;
        }
}
```

Dynamically changing function

```
public class PIDMotor implements PIDOutput
{
        /** The motor that will be set based on the {@link PIDController} results. */
        public PWMSpeedController motor;
        private PIDController controller;
        private double rampBand;

        /**
         * Constructor for a PID controlled motor, with a controllable multiplier.
         *
         * @param motor The motor being set.
         * @param rampBand The acceptable range for a motor change in one loop
         * @param controller The PIDController this was passed as output to
         */
        public PIDMotor(PWMSpeedController motor, double rampBand, PIDController␣
↪controller) {
                this.motor = motor;
                this.controller = controller;
                this.rampBand = rampBand;
                controller.setOutputRange(0 - rampBand, 0 + rampBand);
        }

        public void pidWrite(double pidInput) {
                motor.set(pidInput);
                controller.setOutputRange(pidInput - rampBand, pidInput + rampBand);
        }
}
```

CHAPTER 9

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.

9.1 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.

9.2 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 10

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.

10.1 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.

10.1.1 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.

10.1.2 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.

10.1.3 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.

10.2 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.

Java

C++

Python

```
public class MyRobot extends IterativeRobot{
    DigitalInput forwardLimitSwitch, reverseLimitSwitch;
    Talon motor;
    Joystick joystick1;
    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 11

Introduction to Data Analysis

So you're at a competition, your robot's running great, and you're one of the top teams. Unfortunately, you have no clue what to do during alliance selections because you didn't do any data analysis.

11.1 Phases

There are several phases of data analysis: - Requirements - Collection - Processing - Modeling - Analysis

I will be going through all of these phases in detail to show good practices for all of them, as well as giving some examples on how to execute them properly. For this tutorial, I will be using The Blue Alliance's API to get data quickly, since I don't have ready access to a competition to collect data from.

11.2 Data Requirements

The first step when starting your data analysis is deciding which pieces of information are needed. Now obviously, you're not going to need to know what color the robot is to know if they're a good robot to pick. You need to pick attributes that compliment your robot, while also being effective to the game. Some pieces of information you could want is, using 2017 as an example, the accuracy of their gear autonomous, cycle times for gear, consistency for shooting fuel, or how well the team can play defense.

11.3 Data Collection

The collection of the data is one of the most important steps for data analysis. Without having a dataset that has high quality information, it will be nearly impossible to create a good analysis. Depending on the situation, there are several different styles of data collection that could be used.

11.3.1 Scouting Apps

Using scouting apps is the most common form of electronic scouting. There are several apps that teams create and publish to Google Play and the App Store. These let your team quickly jump into scouting without devoting time to creating the app, but don't allow you to create one with only the data you need. While getting excess data is never bad, it can distract the scouters from the information that's actually needed.

11.3.2 Paper Scouting

Many teams opt out of technology for their scouting and go with a paper and pencil method. While primitive, this works extremely well, allowing the scouters to very quickly pick up on how to fill out the scouting forms. Teams that go with paper scouting usually prepare a printed spreadsheet for every single team at a competition and store them in a box until they need to be used, then distribute them out to all the scouters for each match.

11.4 Data Processing

This is predominantly a thing that needs to be done in electronic scouting, where the data that is collected gets transformed into a more computer readable format. This will make the modelling phase go much smoother, as the data is already organized in an easy to understand format for the program to interact with.

11.5 Data Modelling

This part is, in my opinion, the most fun part of data analysis. This is the part where all the data that was collected gets put into easy to understand graphs, making it much easier to analyze and explain. It can help to show trends between matches, or even compare teams side by side. This is also one of the most important steps for the analysis, as this can make or break the scouting data. There are several different styles of graphs, all being used to represent different things.

11.5.1 Line Graph

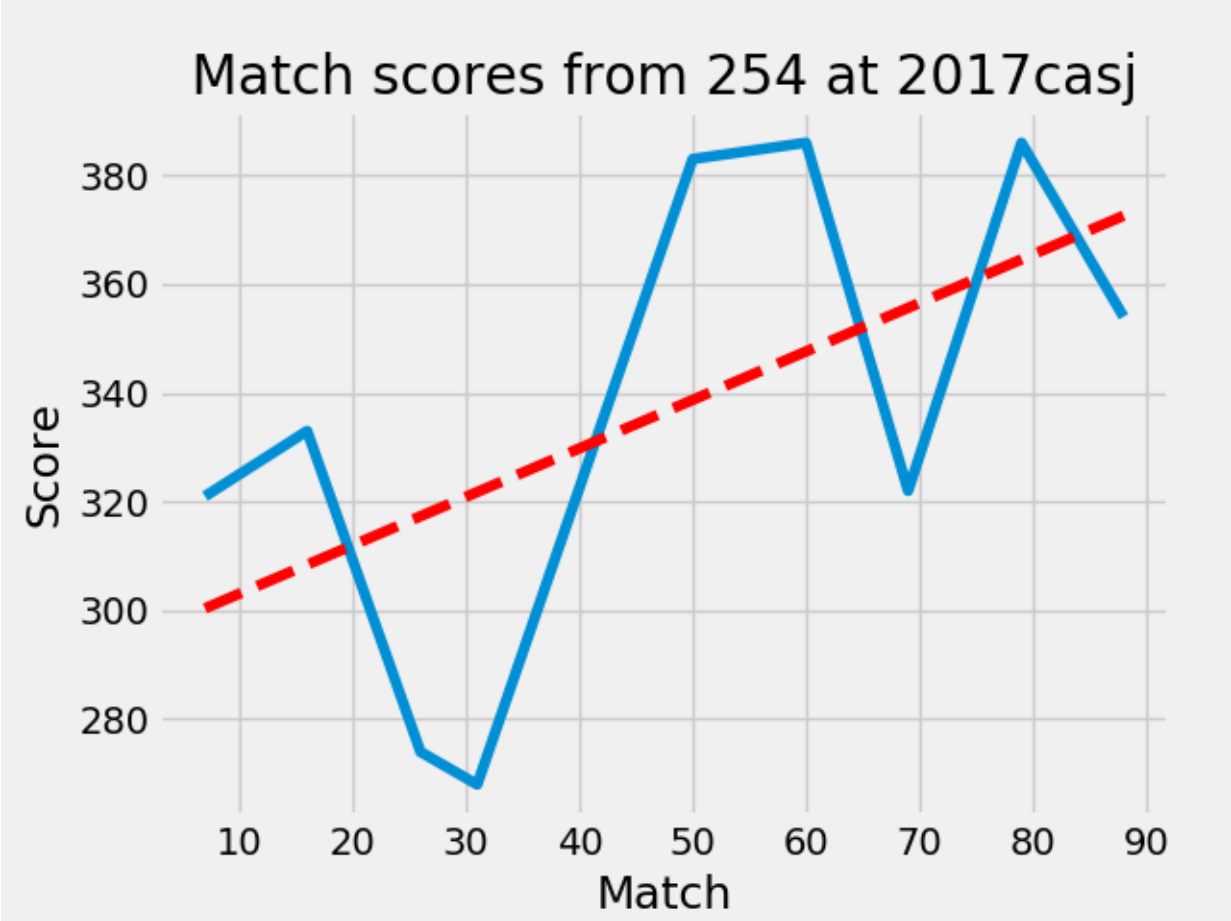
One of the most common styles of graph, a line graph shows the trends between data using lines that connect points together. It can very quickly allow you to compare data by match, so it can be easy to see whether a team improved over their competitions.

11.5.2 Scatter Plots

Best for printing out raw data, you can easily see the trends of where a team lies with data. Very similar to a line graph, it plots points, but doesn't draw lines connecting them together. If using python and matplotlib for the visualization, you can use numpy to calculate a trendline between all of the points and see the average between all of the points.

11.5.3 Using matplotlib

One of the most popular python modules for data visualization is matplotlib. It provides very easy functions for plotting data, allowing you to use several types of graphs. I wrote up some example code that retrieves data using my python api for The Blue Alliance, and then plots match scores vs. match number to show whether a team improved throughout qualification matches. This picture shows the result of the code, while also giving a very basic example of what's possible with matplotlib.


Match scores from 254 at 2017casj
Score
380
360
340
320
300
280
10
20
30
40
50
60
70
80
90
Match

```
'''
NOTE: The following code requires a few modules to run.
            numpy
            matplotlib
            frctba
'''

import numpy as np
import matplotlib.pyplot as plt
from tba import tbaapi3 as tbapi

# Sets Plot styling.
# https://tonysyu.github.io/raw_content/matplotlib-style-gallery/gallery.html for␣
↪list of styles
plt.style.use('fivethirtyeight')

# Get the raw json from The Blue Alliance
teamkey, event = 'frc5203', '2017mimid'
matches = tbapi._fetch('team/%s/event/%s/matches/simple' % (teamkey, event))
matchdata = []

# Collect match scores for qualification matches
for match in matches:
if match['comp_level'] == 'qm':
    matchnum = match['match_number']
    if teamkey in match['alliances']['blue']['team_keys']:
        score = match['alliances']['blue']['score']
    else:
        score = match['alliances']['red']['score']
    matchdata.append((matchnum, score))
# Sort by match. Not necessary, but makes it easier to read when printing.
matchdata.sort()
# Create a numpy array for ease of use
data = np.array(matchdata)

# Create Trendline
x, y = data[:,0], data[:,1]
polyfit = np.polyfit(x, y, 1)
trend = np.poly1d(polyfit)

# Print numpy array and plot the data with a trendline.
print(data)
plt.plot(x,y)
plt.plot(x, trend(x), 'r--')

# Sets plot title and axis names.
plt.title('Match scores from %s at %s' % (teamkey[3:], event))
plt.xlabel('Match')
plt.ylabel('Score')

# Shows the plot in a window
plt.show()
```

These 42 lines above show 4 phases of data analysis. Before starting to write the code, I decided that I wanted to plot the scores in relation to the match number. At the beginning of the code, the data is collected, just using some basic API calls to retrieve match data from a competition. After that, the data is processed to use only the information that is necessary for the graph. Finally, I do some modelling, plotting points using the match number as the x-axis, with the scores on the y-axis. After seeing the very sporadic results, I used numpy to find a trend line, which gives a more

accurate view of how much the team has improved.

CHAPTER 12

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.

12.1 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.

12.1.1 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.

12.1.2 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.

Java

C++

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)
```

12.2 The Basics

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

12.2.1 Reading an image from a file

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

Java

C++

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)
```

12.2.2 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.

Java

C++

Python

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

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

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

12.2.3 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.

Java

C++

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.

12.2.4 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.

Java

C++

Python

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

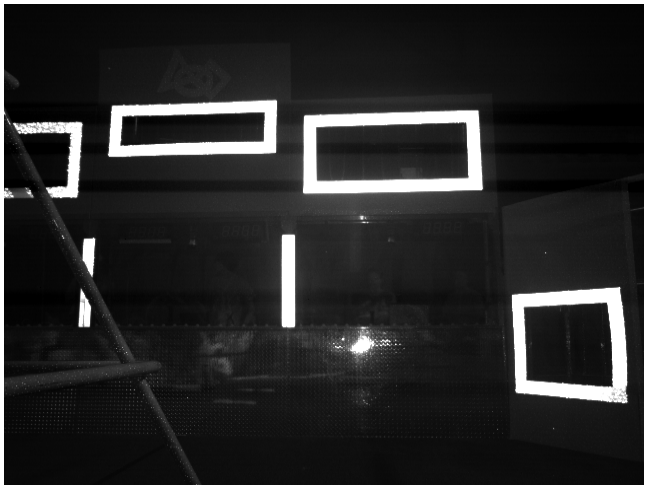
```
VideoCapture cam;
Mat image;
cam.open(0);
cam >> image;
```

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

CHAPTER 13

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.

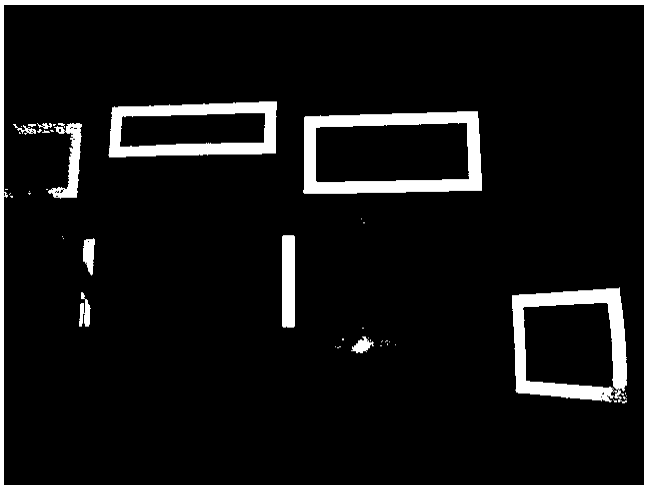
13.1 Threshold

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.

Java



C++

Python

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

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

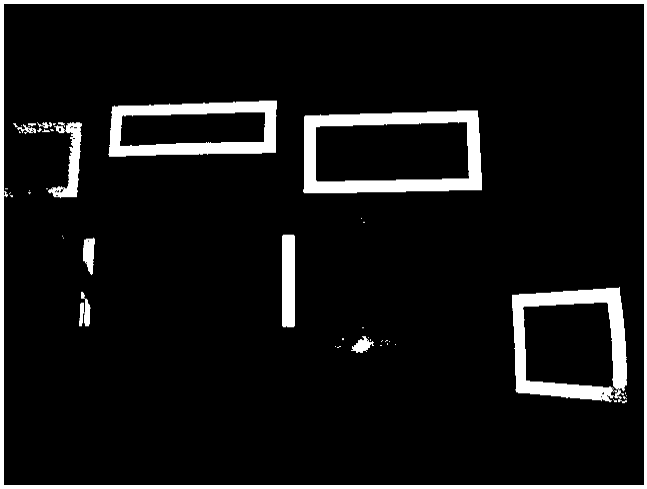
```
ret, dst = cv2.threshold(src, value, maxValue, cv2.THRESH_BINARY)
```

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.

13.2 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.

Java

C++

Python

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

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

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

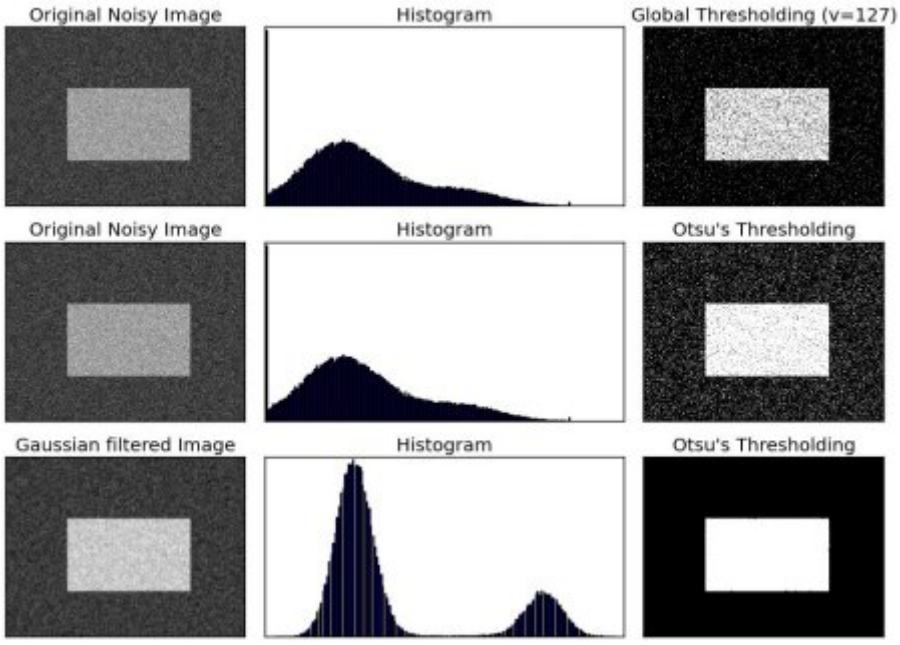
13.3 Otsu

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.



Java

C++

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.

13.4 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:

Java

C++

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]))
```

13.5 Using HSV Thresholding

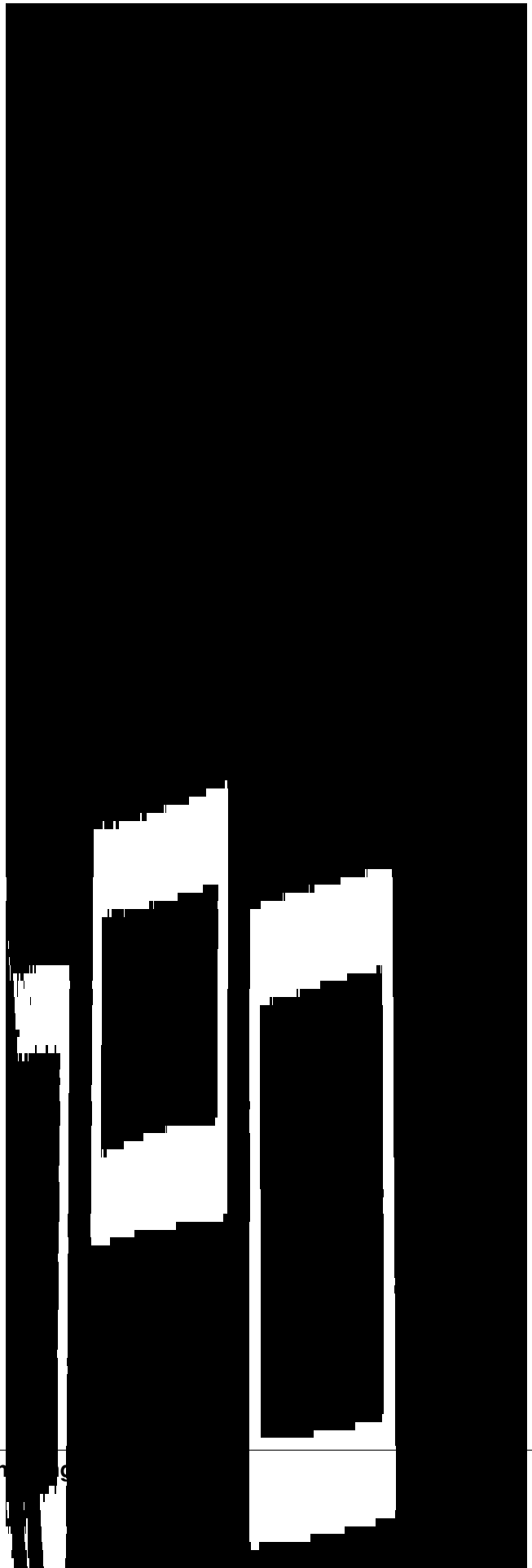
HSV thresholding uses hue, saturation, and value to threshold images. Unlike RGB, HSV separates the image intensity, from the color information. This is very useful in computer applications such as vision tracking. In FRC, HSV is a great tool to detect the reflective vision tape if using a LEDs to illuminate 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).

Java

C++

Python

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



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

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

13.6 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

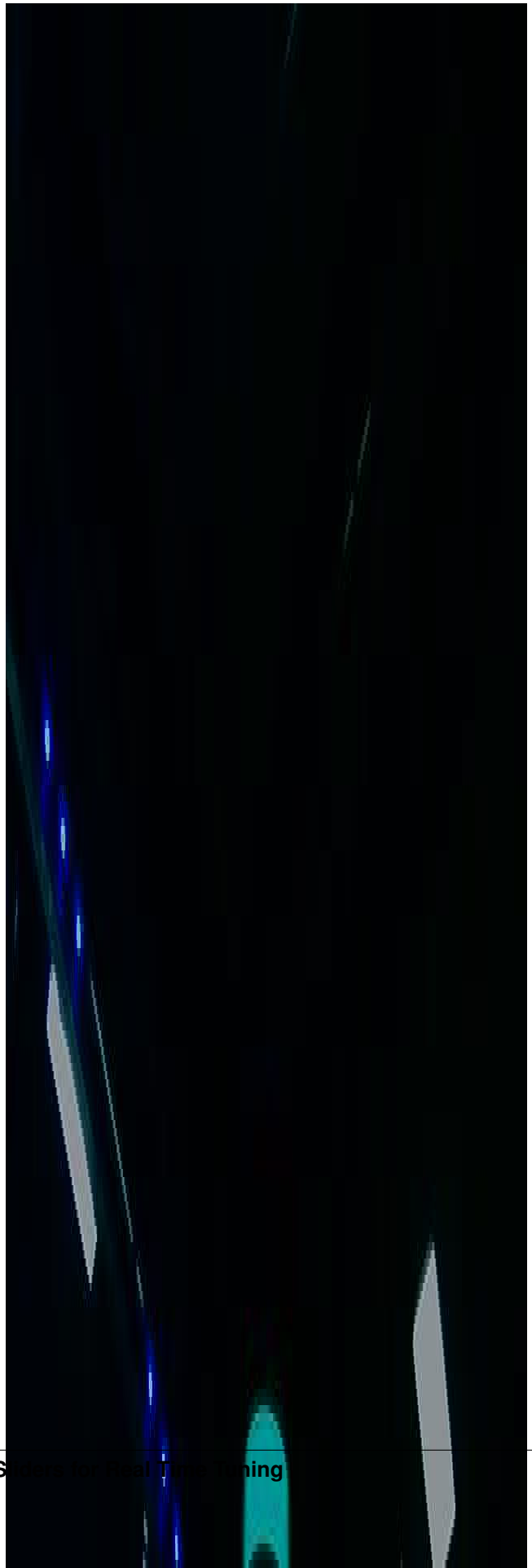
```
cv::namedWindow("Title of Window");
cv::createTrackbar("Title of slider", "Title of Window", &variable, highest number);
```

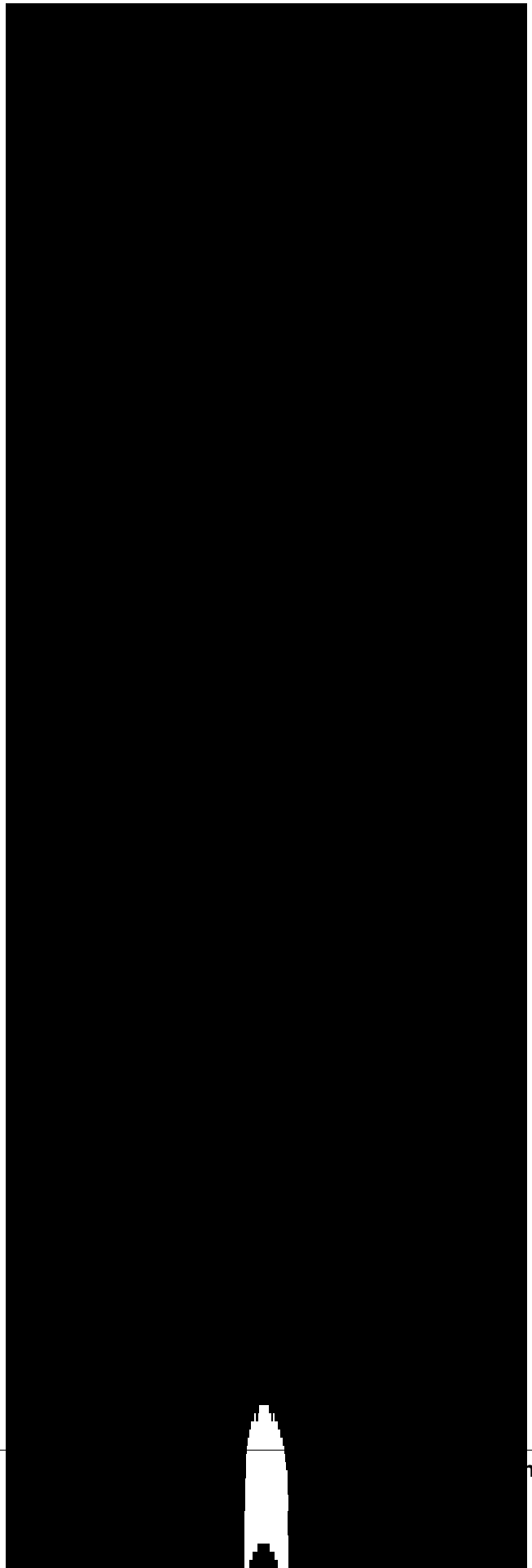
```
cv2.namedWindow('Title of Window')
cv2.createTrackbar('Title of Slider', 'Title of Window', 0, 255, nothing)
var = cv2.getTrackbarPos('Title of Slide', 'Title of Window')
```

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 14

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:

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

Java

C++

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 $A \in E$.

14.1 Erosion

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

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

Java

C++

Python

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

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

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

14.2 Dilation

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

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

Java

C++

Python

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

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

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

14.3 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.

14.4 Morophological Operations Playground

14.4.1 Open

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

Java

C++

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)
```

14.4.2 Close

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

Java

C++

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)
```

14.4.3 Morphological Gradient

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

Java

C++

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)
```

14.5 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.

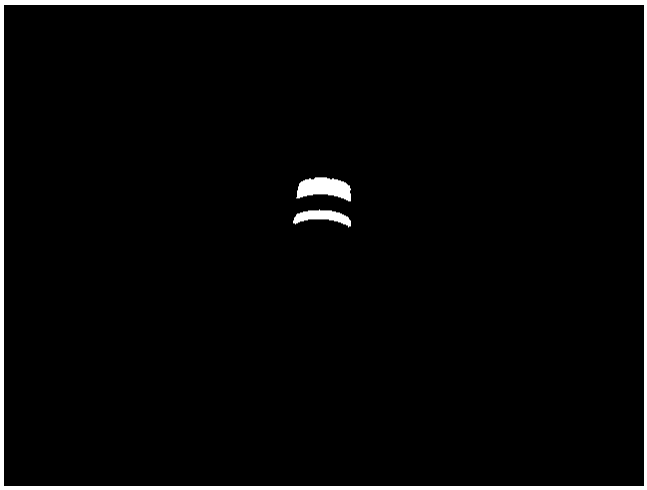
Likewise, this kernel will not alter the height of the pixel cluster, only the width, this is useful with targets who are close together vertically or if the target is not vey tall and you must erode.

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

CHAPTER 15

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.

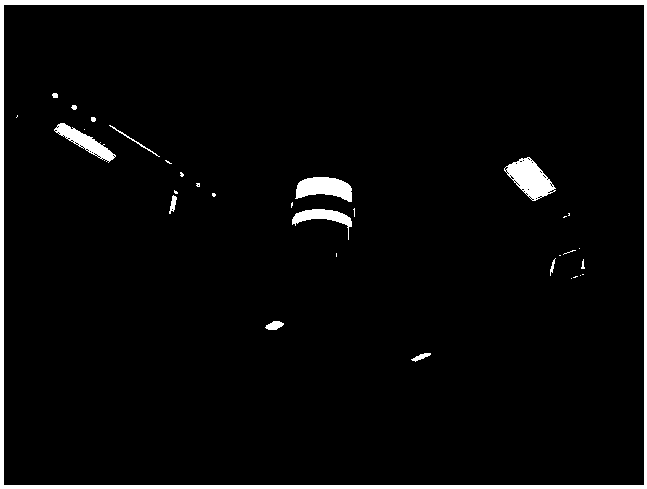
Java

C++

Python

```
List<MatOfPoint> contours = new ArrayList<MatOfPoint>();
Mat hierarchy = new Mat();
Imgproc.findContours(img, contours, hierarchy, Imgproc.RETR_EXTERNAL, Imgproc.CHAIN_
↪APPROX_NONE);
```

(continues on next page)



(continued from previous page)

```
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.

Java

C++

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....
```

15.1 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.

Java

C++

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.

15.2 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.

Java

C++

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.

15.3 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.

Java

C++

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)
```

15.4 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. Here is how to find the center using bounding rectangles. CenterX is the center x coordinate and CenterY is center y coordinate.

Java

C++

Python

```
Rect boundRect = Imgproc.boundingRect(contour);
double centerX = boundRect.x + (boundRect.width / 2)
double centerY = boundRect.y + (boundRect.height / 2)
```

```
Rect boundRect = boundingRect(contours[i]);
double centerX = boundRect.x + (boundRect.width / 2)
double centerY = boundRect.y + (boundRect.height / 2)
```

```
x,y,w,h = cv2.boundingRect(contour)
double centerX = boundRect.x + (boundRect.w / 2)
double centerY = boundRect.y + (boundRect.h / 2)
```

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. The centroid is the center of mass of an object. The centroid point is also the center of gravity. Using moments, the centroid, C is defined as $C_x = M_{10}/M_{00}$ and $C_y = M_{01}/M_{00}$.

Java

C++

Python

```
Moments m = Imgproc.moments(contour);
double centerX = m.get_m10() / m.get_m00();
double centerY = m.get_m01() / m.get_m00();
```

```
cv::Moments moment = cv::moments(contours[i]);
cv::Point center = cv::Point2f(moment.m10/moment.m00, moment.m01/moment.m00);
```

```
moments = cv2.moments(contour)
centerX = int(moments['m10']/moments['m00'])
centerY = int(moments['m01']/moments['m00'])
```

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.

15.5 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.

Java

C++

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)
```

15.6 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.