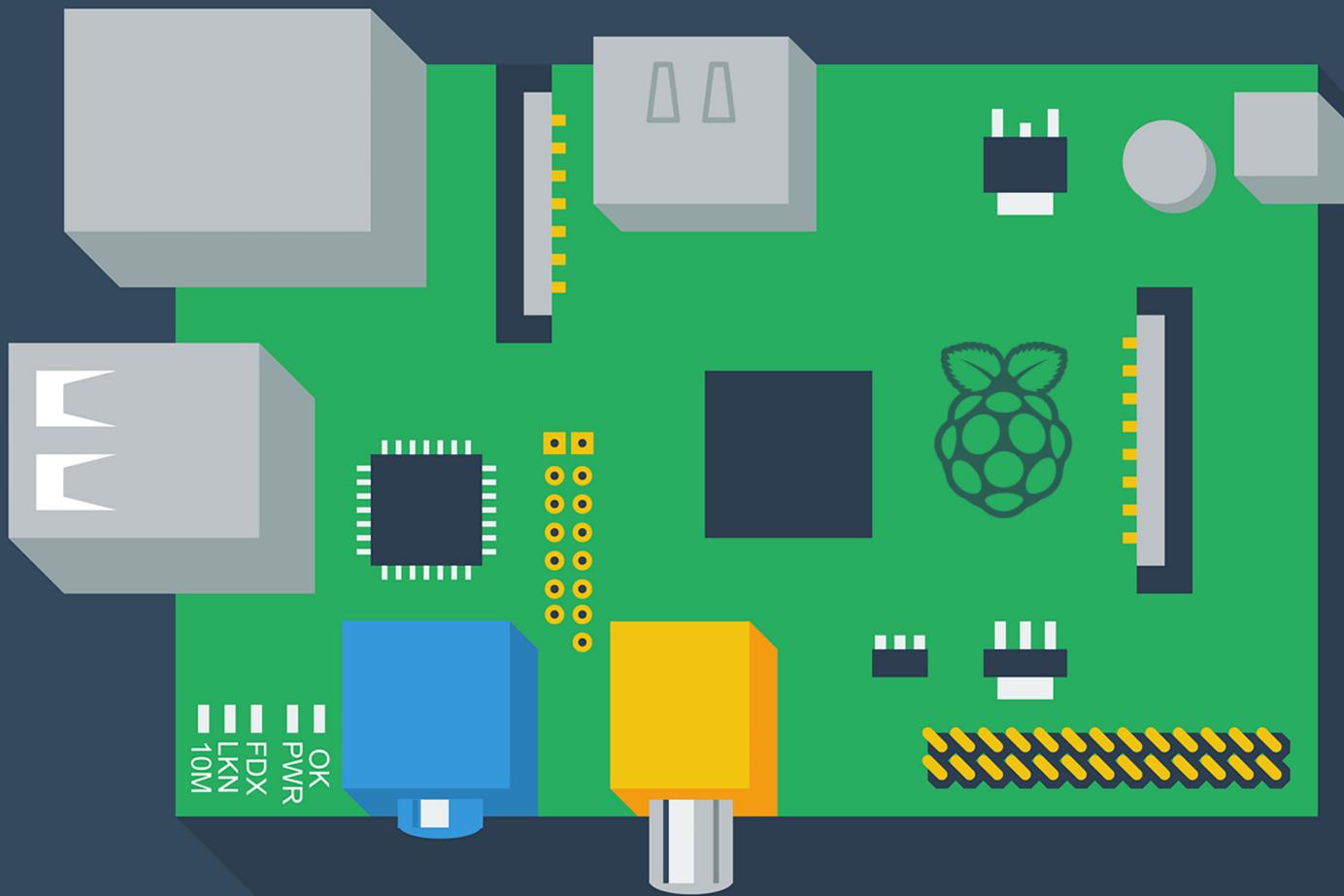


# Raspberry Pi



## for Computer Vision

# Raspberry Pi for Computer Vision

Hobbyist Bundle - v1.0.0

Adrian Rosebrock, PhD

Dave Hoffman, MSc

David McDuffee

Abhishek Thanki

Sayak Paul



The contents of this book, unless otherwise indicated, are Copyright ©2019 Adrian Rosebrock, PylImageSearch.com. All rights reserved. Books like this are made possible by the time invested by the authors. If you received this book and did not purchase it, please consider making future books possible by buying a copy at <https://www.pyimagesearch.com/raspberry-pi-for-computer-vision/> today.

# Contents

<b>Contents</b>	<b>3</b>
<b>1 Why the Raspberry Pi?</b>	<b>17</b>
1.1 Chapter Learning Objectives . . . . .	18
1.2 Can We Use the RPi for CV and DL? . . . . .	18
1.2.1 The RPi for CV and DL Applications . . . . .	19
1.2.2 Cheap, Affordable Hardware . . . . .	20
1.2.3 Computer Vision, Compiled Routines, and Python . . . . .	20
1.2.4 The Resurgence of Deep Learning . . . . .	21
1.2.5 IoT and Edge Computing . . . . .	21
1.2.6 The Rise of Coprocessor Devices . . . . .	22
1.2.7 Embedded Boards and Devices . . . . .	22
1.3 Summary . . . . .	23
<b>2 What is Computer Vision and Deep Learning?</b>	<b>25</b>
2.1 Chapter Learning Objectives . . . . .	25
2.2 Untangling Computer Vision and Deep Learning . . . . .	25
2.2.1 Artificial Intelligence . . . . .	26
2.2.2 Machine Learning . . . . .	27
2.2.3 Deep Learning . . . . .	27
2.2.4 Computer Vision . . . . .	28
2.3 Bring the Right Tools for the Job . . . . .	29
2.4 Summary . . . . .	30
<b>3 Installing Required Packages and Libraries</b>	<b>33</b>
3.1 Chapter Learning Objectives . . . . .	34
3.2 Libraries and Packages . . . . .	34
3.2.1 Python . . . . .	34

3.2.2	OpenCV . . . . .	35
3.2.3	scikit-image and scikit-learn . . . . .	35
3.2.4	dlib . . . . .	35
3.2.5	Keras and TensorFlow . . . . .	35
3.2.6	Caffe . . . . .	36
3.3	Configuring your Raspberry Pi . . . . .	36
3.4	Pre-configured Raspbian .img File . . . . .	36
3.5	How to Structure Your Projects . . . . .	37
3.6	Summary . . . . .	38
<b>4</b>	<b>A Brief Tutorial on OpenCV</b> . . . . .	<b>41</b>
4.1	Chapter Learning Objectives . . . . .	41
4.2	Project Structure . . . . .	41
4.3	OpenCV Basics . . . . .	42
4.3.1	Loading an Image with OpenCV . . . . .	43
4.3.2	Accessing Individual Pixel Values . . . . .	45
4.3.3	Array Slicing and Cropping . . . . .	46
4.3.4	Resizing Images . . . . .	47
4.3.5	Rotating Images . . . . .	48
4.3.6	Blurring Images . . . . .	49
4.3.7	Drawing Methods . . . . .	50
4.3.8	Counting Objects . . . . .	51
4.3.9	Image Subtraction . . . . .	54
4.3.10	Object and Face Detection . . . . .	61
4.4	Summary . . . . .	64
<b>5</b>	<b>Accessing RPi Camera/USB Webcam</b> . . . . .	<b>67</b>
5.1	Chapter Learning Objectives . . . . .	67
5.2	Should I use a RPi Camera Module or USB Webcam? . . . . .	68
5.3	Accessing Cameras via the <code>VideoStream Class</code> . . . . .	69
5.4	Running Our Script . . . . .	71
5.5	Tips and suggestions . . . . .	73
5.5.1	Enable the RPi Camera via <code>raspi-config</code> . . . . .	73
5.5.2	Ensure Your RPi Camera Module is Correctly Plugged In . . . . .	73
5.5.3	Run a Quick Sanity Test with <code>raspistill</code> . . . . .	74
5.5.4	Understanding <code>NoneType</code> Errors . . . . .	74
5.5.5	Confusing the Parameters to <code>VideoStream</code> . . . . .	75

5.6	Summary	75
<b>6</b>	<b>Changing Camera Parameters</b>	<b>77</b>
6.1	Chapter Learning Objectives	77
6.2	Changing Camera Parameters	78
6.2.1	Project Structure	78
6.2.2	Method #1: Changing USB Camera Parameters from a GUI	78
6.2.3	Method #2: Changing USB Camera Parameters from the Command Line	79
6.2.4	Method #3: Changing USB Camera Settings from OpenCV's API	82
6.2.5	Method #4: Changing PiCamera Settings with the PiCamera API	89
6.3	Tips and Suggestions	98
6.4	Summary	99
<b>7</b>	<b>Building a Time Lapse Capture Camera</b>	<b>101</b>
7.1	Chapter Learning Objectives	101
7.2	Capturing a Time Lapse Sequence	102
7.3	Running the Time Lapse Capture Script	107
7.4	Processing Time Lapse Images into a Video	108
7.5	Running our Time Lapse Processor	111
7.6	Tips and Suggestions	112
7.6.1	Common Errors and Problems	113
7.7	Summary	114
<b>8</b>	<b>Automatically Starting Scripts on Reboot</b>	<b>115</b>
8.1	Chapter Learning Objectives	115
8.2	Starting Scripts on Reboot	116
8.2.1	Understanding the Project Structure	116
8.2.2	Implementing the Python Script	116
8.2.3	Creating the Shell Script	118
8.2.4	Method #1: Using crontab	119
8.2.5	Method #2: Updating the LXDE Autostart	120
8.3	An Example of Running a Script on Reboot	121
8.4	Common Issues and Debugging Tips	121
8.5	Summary	122
<b>9</b>	<b>Creating a Bird Feeder Monitor</b>	<b>125</b>
9.1	Chapter Learning Objectives	125

9.2 Creating a Bird Feeder Monitor . . . . .	126
9.2.1 What is Background Subtraction? . . . . .	126
9.2.2 Saving Key Event Clips . . . . .	127
9.2.3 Project Structure . . . . .	127
9.2.4 Our Configuration Files . . . . .	128
9.2.5 Developing Our Bird Monitor Script . . . . .	131
9.2.6 Deploying Our Bird Monitor System . . . . .	138
9.3 Common Issues and Debugging Tips . . . . .	140
9.4 Summary . . . . .	140
<b>10 Sending Notifications from Your RPi to Your Smartphone</b> . . . . .	<b>141</b>
10.1 Chapter Learning Objectives . . . . .	141
10.2 Sending Text Messages Via an API . . . . .	142
10.2.1 What is Twilio? . . . . .	142
10.2.2 Registering for a Twilio Account . . . . .	142
10.2.3 Installing the Twilio Library . . . . .	143
10.3 Implementing a Python Script to Send Texts . . . . .	144
10.3.1 Project Structure . . . . .	144
10.3.2 Our Configuration File . . . . .	145
10.3.3 Sending Texts via Python and Twilio . . . . .	145
10.4 Implementing the Driver Script . . . . .	146
10.4.1 Sending the Actual Text Messages . . . . .	148
10.5 Summary . . . . .	148
<b>11 Detecting Mail Delivery</b> . . . . .	<b>149</b>
11.1 Chapter Learning Objectives . . . . .	150
11.2 Detecting Mail Delivery . . . . .	150
11.2.1 SMS Messages with Twilio . . . . .	150
11.2.2 Project Structure . . . . .	150
11.2.3 Our Configuration File . . . . .	152
11.2.4 Our Mail Delivery Detection Script . . . . .	153
11.2.5 Deploying Our Mail Delivery Detector . . . . .	158
11.3 Summary . . . . .	159
<b>12 Working in Low Light Conditions with the RPi</b> . . . . .	<b>161</b>
12.1 Chapter Learning Objectives . . . . .	162
12.2 Light, Low Light, and Invisible Light . . . . .	162

12.2.1 What is Visible Light? . . . . .	163
12.2.2 What is Infrared Light? . . . . .	164
12.2.3 Types of Lights You Can Add to Your Projects . . . . .	164
12.3 Controlling Lights with an RPi . . . . .	167
12.4 Cameras and Non-standard Cameras . . . . .	170
12.5 Photography Tips . . . . .	171
12.6 Summary . . . . .	172
<b>13 Building a Remote Wildlife Detector</b> . . . . .	<b>173</b>
13.1 Chapter Learning Objectives . . . . .	173
13.2 Chapter Layout . . . . .	174
13.3 Hardware: Building a Deployable Wildlife Detector Box . . . . .	174
13.3.1 Hardware Requirements . . . . .	175
13.3.2 Assembling your Wildlife Detector Box . . . . .	177
13.4 Software: Writing the Remote Wildlife Detector . . . . .	180
13.4.1 Project Structure . . . . .	180
13.4.2 Our Configuration File . . . . .	181
13.4.3 Our Wildlife Monitor Driver Script . . . . .	183
13.5 Deploying Our Wildlife Monitor . . . . .	188
13.5.1 Results . . . . .	189
13.5.2 Tips and Suggestions . . . . .	189
13.6 Summary . . . . .	190
<b>14 Video Surveillance and Web Streaming</b> . . . . .	<b>191</b>
14.1 Chapter Learning Objectives . . . . .	192
14.2 Project Structure . . . . .	192
14.3 Implementing a Basic Motion Detector . . . . .	193
14.4 Sending Frames from an RPi to Our Web Browser . . . . .	196
14.4.1 The Flask Web Framework . . . . .	196
14.4.2 Combining OpenCV with Flask . . . . .	197
14.4.3 The HTML Page Structure . . . . .	203
14.5 Putting the Pieces Together . . . . .	203
14.6 Summary . . . . .	205
<b>15 Using Multiple Cameras with the RPi</b> . . . . .	<b>207</b>
15.1 Chapter Learning Objectives . . . . .	207
15.2 Multiple Cameras and the Raspberry Pi . . . . .	208

15.2.1 A Template for Working with Multiple cameras . . . . .	208
15.2.2 Project Structure . . . . .	209
15.2.3 Our Panorama Stitcher . . . . .	210
15.2.4 Pedestrian Detection . . . . .	216
15.2.5 A Two-camera Panorama Pedestrian Detector Driver Script . . . . .	217
15.2.6 Executing the Panorama Pedestrian Detector . . . . .	220
15.2.7 Tips and Suggestions . . . . .	221
15.3 Summary . . . . .	222
<b>16 Detecting Tired, Drowsy Drivers Behind the Wheel</b>	<b>223</b>
16.1 Chapter Learning Objectives . . . . .	224
16.2 Understanding the "Eye Aspect Ratio" (EAR) . . . . .	225
16.3 Detecting Drowsy Drivers at the Wheel with the RPi . . . . .	227
16.3.1 Project Structure . . . . .	227
16.3.2 Our Drowsiness Configuration . . . . .	229
16.3.3 Developing our Drowsiness Detection Script . . . . .	230
16.3.4 Results . . . . .	240
16.3.5 Tips and Suggestions . . . . .	242
16.4 Summary . . . . .	243
<b>17 What's a PID and Why do we need it?</b>	<b>245</b>
17.1 Chapter Learning Objectives . . . . .	245
17.2 The Purpose of Feedback Control Loops . . . . .	246
17.3 PID Control Loops . . . . .	247
17.3.1 The Concept Behind Proportional Integral Derivative Controllers . . . . .	247
17.3.2 Our Custom PID Class . . . . .	249
17.3.3 How to Tune a PID . . . . .	251
17.3.4 Alternatives and Improvements to PIDs . . . . .	252
17.3.5 Further Reading . . . . .	253
17.4 Summary . . . . .	254
<b>18 Face Tracking with Pan/Tilt Servos</b>	<b>255</b>
18.1 Chapter Learning Objectives . . . . .	255
18.2 Pan/tilt Face Tracking . . . . .	256
18.2.1 Hardware Requirements . . . . .	257
18.2.2 A Brief Review on PIDs . . . . .	258
18.2.3 Project Structure . . . . .	258

## CONTENTS

9

18.2.4 Implementing the Face Detector and Object Center Tracker . . . . .	259
18.2.5 The Pan and Tilt Driver Script . . . . .	261
18.2.6 Manual Tuning . . . . .	269
18.2.7 Run Panning and Tilting Processes at the Same Time . . . . .	270
18.3 Improvements for Pan/Tilt Tracking with the Raspberry Pi . . . . .	271
18.4 Summary . . . . .	272
<b>19 Creating a People/Footfall Counter</b>	<b>273</b>
19.1 Chapter Learning Objectives . . . . .	273
19.2 Background Subtraction + Haar Based People Counter . . . . .	274
19.2.1 Project Structure . . . . .	274
19.2.2 Centroid Tracking . . . . .	275
19.2.2.1 Fundamentals of Object Tracking . . . . .	276
19.2.2.2 The Centroid Tracking Algorithm . . . . .	276
19.2.2.3 Centroid Tracking Implementation . . . . .	279
19.2.3 Trackable Objects . . . . .	286
19.2.4 The Centroid Tracking Distance Relationship . . . . .	286
19.2.5 The DirectionCounter Class . . . . .	288
19.2.6 Implementing Our People Counting App Based on Background Subtraction	291
19.2.7 Deploying the RPi People Counter . . . . .	300
19.2.8 Tips and Suggestions . . . . .	302
19.3 Summary . . . . .	302
<b>20 Building a Traffic Counter</b>	<b>305</b>
20.1 Chapter Learning Objectives . . . . .	306
20.2 Similarities and Differences in this Project Compared to People Counting . . . . .	306
20.3 Traffic Counting via Background Subtraction on the RPi . . . . .	307
20.3.1 Project Structure . . . . .	307
20.3.2 Our Configuration File . . . . .	308
20.3.3 Trackable Objects . . . . .	309
20.3.4 Centroid Tracker . . . . .	310
20.3.5 Direction Counter . . . . .	318
20.3.6 Traffic counting Driver Script . . . . .	321
20.3.7 Traffic Counting Results . . . . .	334
20.4 Leading Up to a Successful Project Requires Multiple Revisions . . . . .	335
20.5 Summary . . . . .	338

<b>21 Measuring Object Sizes</b>	<b>339</b>
21.1 Chapter Learning Objectives	339
21.2 A Simple Camera Calibration	340
21.2.1 The “Pixels Per Metric” Ratio	340
21.3 Implementing Object Size Measurement	341
21.3.1 Project Structure	341
21.3.2 Measuring Object Sizes with OpenCV	342
21.3.3 Object Size Measurement Results	346
21.4 Summary	347
<b>22 Building a Prescription Pill Recognition System</b>	<b>349</b>
22.1 Chapter Learning Objectives	349
22.2 The Case for Prescription Pill Recognition	350
22.2.1 Injuries, Deaths, and High Insurance Costs	350
22.2.2 How Computer Vision Can Help	350
22.2.3 Why Not OCR?	351
22.3 Our Pill Recognition System	352
22.4 Characterizing Pills with Features	354
22.4.1 Color Histograms	354
22.4.2 Shape	355
22.4.2.1 How are Hu Moments Computed?	355
22.4.3 Texture	357
22.4.4 Size	359
22.5 Prescription Pill Recognition with Computer Vision	361
22.5.1 Our Project Structure	361
22.5.2 Our Configuration File	362
22.5.3 Finding Pills in Images	363
22.5.4 Quantifying Pill Color	370
22.5.5 Quantifying Pill Shape	372
22.5.6 Quantifying Pill Texture	372
22.5.7 Creating the Pill Identifier	374
22.5.8 Putting the Pieces Together	377
22.5.9 Pill Recognition Results	383
22.6 Improvements and Suggestions	384
22.6.1 Proper Camera Calibration	384
22.6.2 Segmentation in Uncontrolled Environments	385

<i>CONTENTS</i>	11
22.6.3 Triplet Loss, Siamese Network, and Deep Metric Learning . . . . .	386
22.7 Summary . . . . .	387
<b>23 OpenCV Optimizations and Best Practices</b>	<b>389</b>
23.1 Chapter Learning Objectives . . . . .	389
23.2 Package and Library Optimizations . . . . .	390
23.2.1 NEON and VFPV3 . . . . .	391
23.2.2 TBB . . . . .	391
23.2.3 OpenCL . . . . .	392
23.3 Benchmarking and Profiling Your Scripts . . . . .	393
23.3.1 Project Structure . . . . .	393
23.3.2 Timing Operations and Functions . . . . .	394
23.3.3 Measuring FPS Throughput . . . . .	396
23.3.4 Python's Built-in Profiler . . . . .	398
23.3.4.1 Command Line Profiling . . . . .	398
23.3.4.2 Profiling Within a Python Script . . . . .	400
23.4 Improving Computation Speed with Cython . . . . .	402
23.5 Summary . . . . .	403
<b>24 Your Next Steps</b>	<b>405</b>
24.1 So, What's next? . . . . .	406



*To the PylImageSearch team;  
the Raspberry Pi community;  
and all the PylImageSearch readers who  
have made this book possible.*



# Companion Website

Thank you for picking up a copy of *Raspberry Pi for Computer Vision!* To accompany this book I have created a companion website which includes:

- **Up-to-date installation instructions** on how to configure your Raspberry Pi development environment
- Instructions on how to use the **pre-configured Raspbian .img file(s)**
- **Supplementary material** that I could not fit inside this book
- **Frequently Asked Questions (FAQs)** and their suggested fixes and solutions

Additionally, you can use the “*Issues*” feature inside the companion website to report any bugs, typos, or problems you encounter when working through the book. I don’t expect many problems; however, this is a brand new book so myself and other readers would appreciate reporting any issues you run into. From there, I can keep the book updated and bug free.

**To create your companion website account, just use this link:**

<http://pyimg.co/qnv89>

Take a second to create your account now so you’ll have access to the supplementary materials as you work through the book.



## Chapter 1

# Why the Raspberry Pi?

*"The Raspberry Pi is a low cost, credit-card sized computer that plugs into a computer monitor or TV, and uses a standard keyboard and mouse. It is a capable little device that enables people of all ages to explore computing, and to learn how to program in languages like Scratch and Python."* — Raspberry Pi foundation

Out of all the computer devices in the past decade that have facilitated not only *innovation*, but *education* as well, very few, if any devices have surpassed the Raspberry Pi. And at only \$35, this single board computer packs a punch similar to desktop hardware a decade ago.

Since the Raspberry Pi (RPi) was first released back in 2012, the surrounding community has used it for fun, innovative, and practical projects, including:

- i. Creating a wireless print server
- ii. Utilizing the RPi as a media center
- iii. Adding a controller to the RPi and playing retro Atari, NES, SNES, etc. games via a software emulator
- iv. Building a stop motion camera

*...and the list goes on.*

At the core, the Raspberry Pi and associated community has its roots in both:

- **Practicality** — Whatever is built with the RPi should be *useful* in some capacity.
- **Education** — The hacker ethos, at the core, is about *education* and learning something new. When using a RPi, you should be pushing the limits of your understanding and enabling yourself to learn a new technique, method, or algorithm.

I published my first series of Raspberry Pi tutorials on the PyImageSearch blog back in 2015. These tutorials taught PyImageSearch readers how to:

- i. Install OpenCV (with Python bindings) on the RPi
- ii. Access the Raspberry Pi camera module
- iii. Build a home surveillance system capable of detecting motion and sending notifications to the user

These blog posts fit the RPi ethos perfectly. Not only is building a home security application *practical*, but it was also *educational*, both for myself as well as the PyImageSearch community.

Nearly five years later, I feel incredibly lucky and privileged to bring this book to you, keeping the Raspberry Pi ethos close to heart. **Inside this text expect to find highly practical, hands-on computer vision and deep learning projects that will challenge your education and enable you to build real-world applications.**

What are you waiting for? Let's get started!

## 1.1 Chapter Learning Objectives

In this chapter you will:

- Discover the history of the Raspberry Pi
- Learn how computer vision (CV) can be applied on the RPi
- Briefly review how IoT and Edge Computing is fueling the RPi
- Discover alternative devices for CV and Deep Learning (DL)

## 1.2 Can We Use the RPi for CV and DL?

In the first part of this chapter we'll briefly review the history of the Raspberry Pi. I'll then discuss how computer vision can be applied to the RPi, followed by how the current trends in Internet of Things (IoT) and Edge Computing applications are helping drive innovation in embedded devices (both at the software and hardware level).

We'll then wrap up by looking at coprocessor devices, such as Intel's Movidius NCS [1] and Google's Coral USB Accelerator [2], and how they are facilitating state-of-the-art deep learning on the RPi.

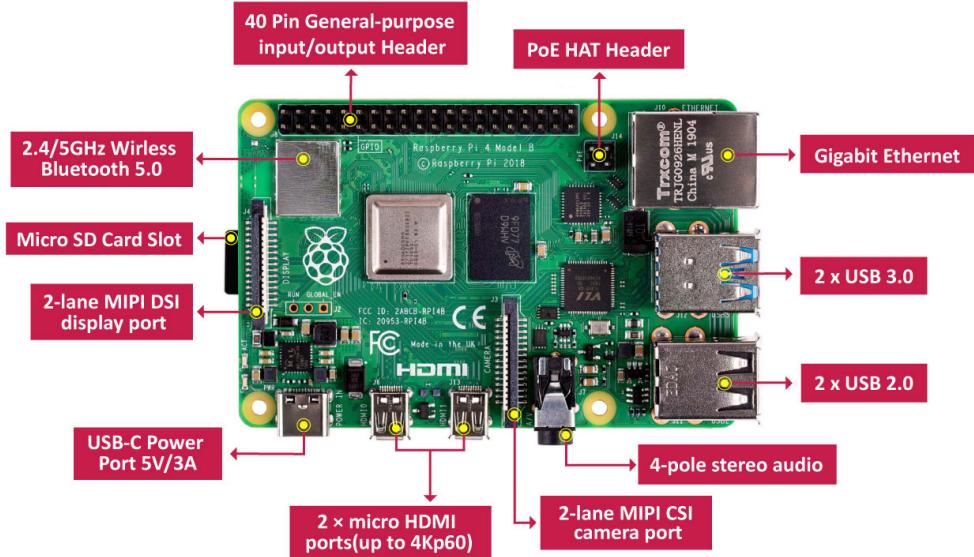


Figure 1.1: The Raspberry Pi 4 comes with a 64-bit 1.5GHz processor and 1-4GB of RAM, all in a device approximately the size of a credit card and under \$35. Credit: Seeed Studio [3]

### 1.2.1 The RPi for CV and DL Applications

The very first Raspberry Pi was released in 2012 with a 700 MHz processor and 512GB of RAM. Specs have improved over the years — the current iteration, the RPi 4B (Figure 1.1), has a 64-bit 1.5GHz quad-core processor and 1-4GB of RAM (depending on the model).

The Raspberry Pi has similar specs to desktop computers from a decade ago, meaning it's still incredibly underpowered, especially compared to our current laptop/desktop devices — **but why should we be interested?**

To start, consider the field of computer vision from a research perspective — image understanding algorithms that were *incredibly computationally expensive* and *only capable of running on high-end machines* ten years ago can now be effectively executed on the RPi.

We can also look at it from a practitioner viewpoint as well — computer vision libraries have matured to the point where they are straightforward to install, simple to use (once you understand them), and *are so highly optimized* that even more recent computer vision algorithms can be run on the RPi.

**Effectively, the Raspberry Pi has brought computer vision to embedded devices, whether you are a hobbyist or an experienced practitioner in the field.**

And furthermore, we are *not* limited to computer vision algorithms. Using coprocessor de-

vices, such as the Movidius NCS or Google Coral Accelerator, we are now capable of deploying *state-of-the-art* deep neural networks to the RPi as well!

This is an *incredibly* exciting time to be involved in computer vision on embedded devices — the possibility for innovation is nearly endless.

### 1.2.2 Cheap, Affordable Hardware

Part of what makes the Raspberry Pi so attractive is the relatively cheap, affordable hardware. At the time of this writing, a Raspberry Pi 4 costs \$35, making it a minimal investment for both:

- i. **Hobbyists** who wish to teach themselves new algorithms and build fun projects.
- ii. **Professionals** who are creating products using the RPi hardware.

At only \$35, the RPi is well positioned, enabling hobbyists to afford the hardware while providing enough value and computational horsepower for industry professionals to build production-level applications with it.

### 1.2.3 Computer Vision, Compiled Routines, and Python

It is no secret that computer vision algorithms can be computationally expensive. Typically, when a programmer needs to extract *every last bit of performance* out of a routine, they'll ensure every variable, loop, and construct is optimized, typically by implementing the method in C or C++ (or even dropping down to assembler).

While compiled binaries are undoubtedly fast, the associated code takes significantly longer to write and is often harder to maintain. But on the other hand, languages such as Python, which tends to be easier to write and maintain, often suffer from slower code execution.

**Luckily, computer vision, machine learning, and deep learning libraries are now providing compiled packages.** Libraries such as OpenCV, scikit-learn, and others:

- Are implemented directly in C/C++ *or* provide compiled Cython optimized functions (Python-like functions with C-like performance).
- Provide Python bindings to interact with the compiled functions.

**Effectively, this combination of compiled routines and Python bindings gives us the best of both worlds.** We are able to leverage the *speed* of a compiled function while at the same time *maintaining* the ease of coding with Python.



Figure 1.2: **Left:** Intel's Movidius Neural Compute Stick. **Right:** Google Coral USB Accelerator.

#### 1.2.4 The Resurgence of Deep Learning

As we'll discuss in the next chapter, the latest resurgence in deep learning has created an additional interest in embedded device, such as the Raspberry Pi. Deep learning algorithms are *super powerful*, demonstrating unprecedented performance in tasks such as image classification, object detection, and instance segmentation.

The *problem* is that deep learning algorithms are *incredibly* computationally expensive, making them challenging to run on embedded devices.

But just as computer vision libraries are making it easier for CV to be applied to the RPi, the same is true with deep learning. Libraries such as TensorFlow Lite [4] enable deep learning practitioners to train a model on a custom dataset, optimize it, and then deploy it to resource constrained devices as the RPi, obtaining faster inference.

#### 1.2.5 IoT and Edge Computing

The Raspberry Pi is often used for Internet of Things applications. A great example of such a project would be building a remote wildlife detector (which we'll do later in Chapter 13).

Such a system is deployed in the wilderness and is either powered by batteries and/or a solar panel. The camera then captures and processes images of wildlife, useful for approximating species counts and detecting intruders.

Another great example of IoT and edge computing with the RPi comes from Jeff Bass (<http://pyimg.co/h8is2>), a PyImageSearch reader who uses RPis around his farm to monitor temperature, humidity, sunlight levels, and even detect water meter usage.

**Remark.** If you’re interested in learning more about how Jeff Bass is using computer vision and RPis around his farm, you can read the full interview on the PyImageSearch blog: <http://pyimg.co/sr2gj>.

### 1.2.6 The Rise of Coprocessor Devices

As mentioned earlier, deep learning algorithms are computationally expensive, which is a *big* problem on the resource constrained Raspberry Pi. In order to run these computationally intense algorithms on the RPi we need additional hardware.

Both the Intel (Movidius NCS) [1] and Google (Coral USB Accelerator) [2] have released what are essentially “USB sticks for deep learning inference” that can be plugged into the RPi (Figure 1.2). We call such devices “coprocessors” as they are designed to augment the capabilities of the primary CPU.

Combined with the optimized libraries from both Google and Intel, we can obtain faster inference on the RPi than using the CPU alone.

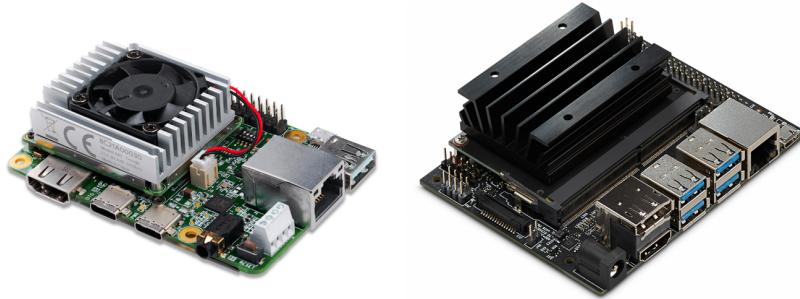


Figure 1.3: **Left:** Google Coral Dev Board. **Right:** NVIDIA Jetson Nano.

### 1.2.7 Embedded Boards and Devices

Of course, there are situations where the Raspberry Pi itself is not sufficient and additional computational resources are required beyond what coprocessors can achieve. In those cases, you would want to look at Google Coral’s Dev Board [5] and NVIDIA’s Jetson Nano [6] (Figure 1.3) — these single board computers are similar in size to the RPi but are *much* faster (albeit more expensive).

**While the Raspberry Pi is the primary focus of this book, all code is meant to be compatible with minimal (if any) changes on both the Coral and Jetson Nano.** I’ve also included notes in the relevant chapters regarding where I would suggest using an alternative to the RPi.

### 1.3 Summary

In this chapter we discussed how the Raspberry Pi can be used for computer vision, including how deep learning, IoT, and edge computing are pushing innovation in embedded devices, both at the *hardware* and *software* level.

In the next chapter we'll expand on our knowledge of the RPi within the CV and DL fields, paving the way for you to build computer vision applications on the Raspberry Pi.



## Chapter 2

# What is Computer Vision and Deep Learning?

In this chapter we'll discuss the fields of computer vision and deep learning, including similarities between them and how they overlap.

This chapter will be fairly high-level and won't dive into code or go into the weeds with implementations. Instead, it will help you understand each field as a whole, and namely how they interact with each other (including clearing up misconceptions about both fields along the way).

### 2.1 Chapter Learning Objectives

In this chapter you will:

- Receive a high-level overview of the computer vision and deep learning.
- Understand the similarities between computer vision and deep learning, including how these fields overlap.

### 2.2 Untangling Computer Vision and Deep Learning

I have been teaching computer vision, and now deep learning, for over five years at this point and two of the most prominent misconceptions I see among beginners in the fields is that computer vision and deep learning are *binary*, meaning they either:

- i. Do not overlap.
- ii. Or that all computer vision problems can now be solved with deep learning.

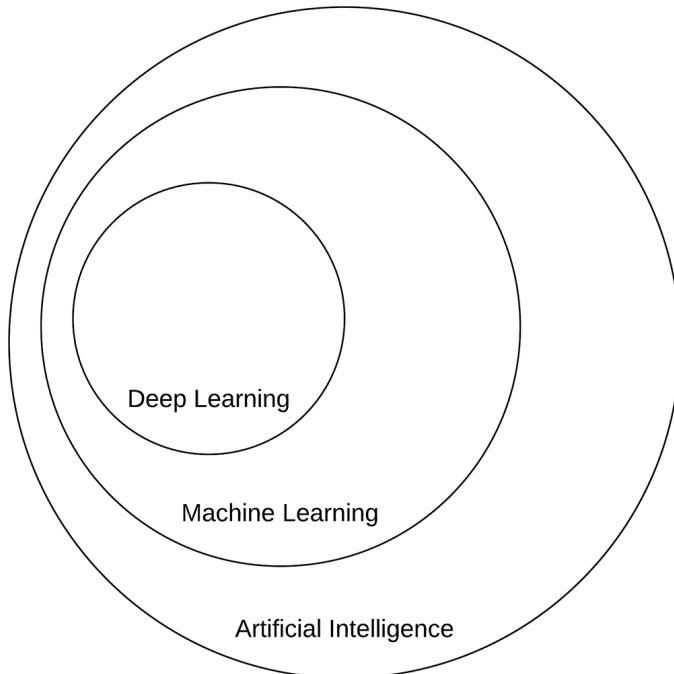


Figure 2.1: A Venn diagram describing deep learning as a subfield of machine learning which is in turn a subfield of artificial intelligence. (Image inspired by Figure 1.4 of Goodfellow et al. [7])

These are unfortunate misconceptions as they really hurt beginners in the field get their foot in the door — they pick one of the fields, study it, and not realize that *there so much more knowledge to be gained*.

This trend is *especially* prominent with new deep learning practitioners. I've seen *thousands* of programmers start studying deep learning, as it's currently one of the hottest trends in computer science.

They pour hours, days, and weeks into their studies...but when confronted with a simple computer vision problem they attempt to bang in a screw with a hammer — they lack the proper tools in their toolbox to appropriately and effectively complete the project.

In the rest of this chapter I'll provide a brief history of the computer vision and deep learning fields, including how Artificial Intelligence, Machine Learning, and Deep Learning all overlap (Figure 2.1), and most importantly, I'll show you how you can fill your toolbox with the proper tools for the job.

### 2.2.1 Artificial Intelligence

Let's first consider the field of **Artificial Intelligence (AI)** — it's a broad field, encompassing nearly every facet of computer science that involves machines making *intelligent, informed* decisions based on input data that could be dynamic and always changing:

*“The central goal of AI is to provide a set of algorithms and techniques that can be used to solve problems that humans perform intuitively and near automatically, but are otherwise very challenging for computers. A great example of such a class of AI problems is interpreting and understanding the contents of an image — this task is something that a human can do with little-to-no effort, but has proven to be extremely difficult for machines to accomplish.” — Adrian Rosebrock, Deep Learning for Computer Vision with Python [8]*

Any algorithm that accepts input data, learns from that data, and attempts to make intelligent decisions/reasoning based on that data, can be seen as an AI algorithm.

### 2.2.2 Machine Learning

While AI embodies a large, diverse set of work related to automatic machine reasoning (inference, planning, heuristics, etc.), the **machine learning** subfield tends to be *specifically interested in pattern recognition and learning from data* [8].

In essence, machine learning algorithms are given a set of input data, and then instructed to “make sense” of this data *without* applying explicit instructions on hard-coded rules. Once a machine learning method has learned the underlying patterns in the input data, it can attempt to perform *inference* and make predictions based on new input data points.

A great example of machine learning is building a spam classifier for your inbox. A machine learning algorithm would require a set of training examples, including emails that are:

- i. Spam
- ii. Not spam

A machine learning algorithm would accept these emails, learn patterns based on the distribution of characters and word counts, and then *classify* incoming emails as either “spam” or “not spam”.

### 2.2.3 Deep Learning

The latest incarnation of neural networks is called **deep learning**, a subfield of machine learning. It surprises many practitioners new to the field that Artificial Neural Networks (ANNs) have been around *for over 60 years*, going by different names based on popular research trends at the time.

Paraphrasing Geoff Hinton [9], what makes deep learning *different* than the previous incarnations of ANNs is that we now have:

- i. Faster computers
- ii. Highly optimized hardware (i.e., GPUs)
- iii. Large, labeled datasets in the order of millions of data points
- iv. A better understanding of weight initialization functions and what does/does not work.
- v. Superior activation functions and an understanding of why previous nonlinear activation functions stagnated research.

Furthermore, deep learning is *not* limited to applications in computer vision — it is also applied to other fields, including information retrieval, text understanding, natural language processing, and audio understanding.

Since the rebirth of neural networks in the early 2010s, deep learning has touched nearly every facet of computer vision, but deep learning algorithms are *still* contained within the machine learning and AI subfields.

#### 2.2.4 Computer Vision

In July 1966 Marvin Minsky gave a project to student Gerald Sussman:

*“Connect a camera to a computer and have the computer understand what’s in the image.”*

The project mostly stipulated performing background and foreground subtraction, followed by analyzing the objects detected as “foreground” [10].

With that, the field of computer vision was born (and it’s worth noting that the researchers found computer vision *far more challenging* than they originally thought, even in its most simple, controlled form).

At the highest level, **computer vision** encompasses all methods used for **image understanding**, which is at the very core, *understanding what is in an image*. Some people also lump **image processing** into the computer vision field as well, as many image processing techniques are also included in computer vision pipelines.

Since computer vision includes all algorithms used to understand the contents of images, by definition, **computer vision overlaps with the deep learning algorithms used for image understanding as well** (Figure 2.2).

But given the popularity and proliferation of deep learning, that raises the question:

*“Are ‘traditional’ computer vision algorithms relevant anymore? Or has deep learning taken over the computer vision field?”*

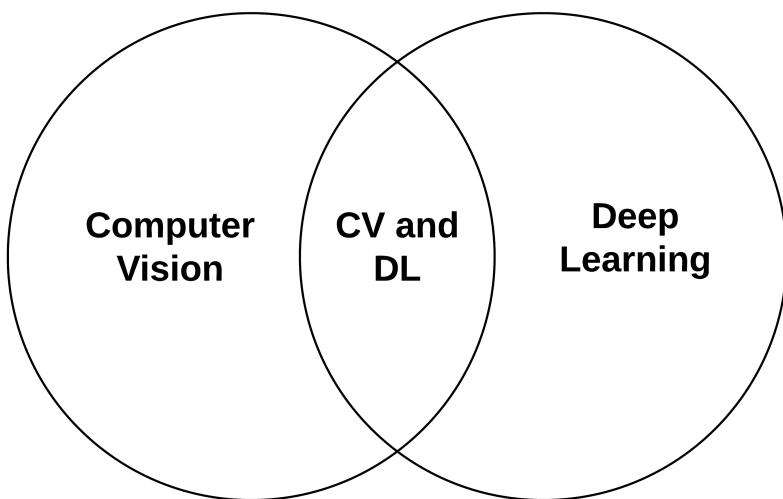


Figure 2.2: The deep learning field provides techniques and algorithms for *more* than just computer vision, including Natural Language Processing (NLP), audio classification, and more. Computer vision also includes algorithms that are *not* related to deep learning. That said, there is a substantial overlap between the fields in modern day computer science.

That line of thinking is arguably the **largest misconception** I have seen over the years — and untangling this question all starts with bringing the right tools to the job.

## 2.3 Bring the Right Tools for the Job

Imagine this scenario — you spent the past two years in a trade school learning how to be a plumber and you've started your apprenticeship working with a master plumber. But on the first day of the job, you decide to leave your wrenches and solder at home, and instead bring nothing but a table saw. At best, you may be able to awkwardly and inefficiently cut the pipes — but that's about it — you won't be able to fit the actual pipes and plumb them properly.

Computer vision and deep learning are no different — **you need to bring the right tools to the job.**

Just because you encounter a computer vision problem *doesn't mean* that you should attempt to throw deep learning at it. Instead, what you need is the *discipline* and *understanding* of what tool to use. Just because you have hammer doesn't mean you should use it to bang in a screw. Pick a screwdriver out of your toolbox instead.

*Raspberry Pi for Computer Vision* takes an **example-based approach** to show you where and when you should apply *traditional computer vision* versus *deep learning algorithms*.

Inside the *Hobbyist Bundle* you will primarily use standard computer vision methods. Then,

in the *Hacker Bundle* and *Complete Bundle*, you will learn how to incorporate deep learning into your computer vision pipelines, making them more powerful, but *still* relying on the fundamentals of computer vision.

If you're new to computer vision and/or deep learning and want to study these fields more in-depth, be sure to refer to my books/courses, including:

- **Practical Python and OpenCV:** A gentle introduction to the world of computer vision and image processing through the OpenCV library (<http://pyimg.co/ppao>) [11].
- **PylImageSearch Gurus:** Similar to a college survey course on computer vision, but *far* more hands-on and practical (<http://pyimg.co/gurus>) [12].
- **Deep Learning for Computer Vision with Python:** An in-depth dive into the intersection of computer vision and deep learning, showing you how to train state-of-the-art deep learning models on your own custom datasets (<http://pyimg.co/dl4cv>) [8].

These resources are *not required* when reading through *Raspberry Pi for Computer Vision* — I've simply included them if you wish to learn more about the fields and study them in greater depth, enabling you to add more tools to your toolbox.

## 2.4 Summary

Inside this chapter you learned about the fields of computer vision and deep learning, a subfield of machine learning which is in turn a subfield of artificial intelligence.

The computer vision field encompasses all sets of algorithms and methods for image understanding, or more simply, *understanding what is “in” an image*. Since computer vision includes all methods for image understanding, by definition, it also includes all deep learning methods used to understand image contents as well.

Computer vision are deep learning thus **overlap** but do not entirely consume each other, as deep learning algorithms can be applied to text and audio understanding as well (which is not part of the computer vision field).

Finally, keep in mind that deep learning has not “taken over” computer vision. Deep learning methods, while powerful, are just tools in your toolbox — these tools do not solve all problems in computer vision. Instead, you need to bring the right tool to the job.

The rest of this book provides hands-on examples showing you which computer vision algorithms are appropriate in a given situation: Sometimes it will be applications of strict, traditional computer vision techniques. Other times we'll be utilizing deep learning, and computer vision will only *facilitate* the deep learning algorithms.

These hands-on chapters will help you understand that computer vision and deep learning are not an “*either/or*” choice — they both feed into each other, enabling us to form a complete solution to a problem.

**And most importantly, the rest of this text will help you fill your toolbox with the right tools for the job.**



## Chapter 3

# Installing Required Packages and Libraries

When learning a new programming language, library, or API, configuring your development environment is usually the first challenge you must surmount — and in some cases, it can feel like a tall order.

When it comes to the Raspberry Pi, your install issues are only further compounded. This is because:

- i. Computer vision and deep learning libraries change *rapidly*, pushing innovation, but also *breaking* previous versions.
- ii. The RPi CPU is quite slow compared to your laptop/desktop, so building and compiling libraries takes far longer than it typically would.

It can be quite frustrating to let your RPi compile a library overnight just to wake up in the morning to find that it failed *or* the library you just compiled isn't compatible with the rest of the packages on your Pi!

Due to the rapidly changing nature of computer vision, deep learning, and edge computing libraries, I've moved the majority of the development environment configuration instructions to the **Companion Website** (<http://pyimg.co/ods37>). Having install instructions on the *website* (versus this *book*) makes it easier for me to guarantee they are always fresh and up-to-date, ensuring you are able to configure your RPi with minimal (and ideally zero) headaches.

Additionally, I have provided a pre-configured Raspbian .img file with all the computer vision and deep learning libraries you need to be successful when (1) working through this text and (2) applying what you've learned to your own projects.

The .img file is included with your purchase of *Raspberry Pi for Computer Vision* — if you

haven't downloaded it yet, check your inbox for your purchase link, click it, and download the .img file. The pre-configured Raspbian .img file will save you *hours* of compile time (and have you up and running in a matter of minutes).

### 3.1 Chapter Learning Objectives

In this chapter you will:

- i. Learn about the libraries and packages we'll use in this text.
- ii. Find the link to configure your Raspberry file from scratch (if you so desire).
- iii. Learn about the pre-configured Raspbian.img file included in your purchase.

### 3.2 Libraries and Packages

In order to build computer vision and deep learning applications on the Raspberry Pi, you'll need to have the right tools. This section highlights the most important libraries and packages we'll be using throughout the text.

The list is *not* exhaustive, however. For the complete list of libraries and packages you need, be sure to refer to the development environment configuration instructions inside the companion website.

#### 3.2.1 Python

All examples inside *Raspberry Pi for Computer Vision* will use the Python programming language.

Python is hands-down the **best language** for working with computer vision and deep learning. Even if you've never used Python before, you'll find that the simple, intuitive syntax allows you to focus on the actual *algorithm/application* you're building rather than spending hours trying to debug compiler errors.

Furthermore, many of the computer vision and deep learning libraries we'll use are actually *compiled binaries*, meaning that function calls in those libraries will be super fast (or as fast as they can be on a RPi). Using Python will enable you to retain the speed of these C/C++ compiled routines.

### 3.2.2 OpenCV

OpenCV is the *de facto* standard for computer vision and image processing. The goal of the OpenCV library is to facilitate real-time computer vision and image processing [13].

The library has been around since 1999, but it wasn't until the v2.0 release in 2009 that we saw increased Python support, including being able to represent images as NumPy arrays.

NumPy array support has paved the way for v3.0 and v4.0, which have provided additional optimizations and a dedicated Deep Neural Network (dnn) module for performing inference using pre-trained models via Caffe, Torch, and TensorFlow.

OpenCV itself is written in C/C++, but Python bindings are provided with the library, enabling you to compile super fast OpenCV routines via the Python API.

### 3.2.3 scikit-image and scikit-learn

To complement OpenCV, we'll be using scikit-image [14], a collection of algorithms for image processing. The scikit-image library, while smaller than OpenCV, includes algorithms that are either (1) not implemented in OpenCV or (2) easier to use via the scikit-image API.

We'll also be using scikit-learn [15], an open-source Python library for machine learning.

### 3.2.4 dlib

The dlib [16] library was created by Davis King as collection of algorithms he used in his day-to-day job. Over the years the library has evolved to include computer vision and deep learning algorithms, two of the most notable ones being **face recognition** and **object detection**.

We'll be using *both* of these algorithm implementations inside dlib in both the *Hacker Bundle* and *Complete Bundle*.

### 3.2.5 Keras and TensorFlow

To implement and train our deep learning networks, particularly in the *Hacker Bundle*, we'll primarily be using the Keras library [17] with a TensorFlow backend [18]. Using Keras makes it *super easy* to build and train networks quickly (you can also use TensorFlow 2.0 and the `tf.keras` module if you so desire – both are compatible with this book.)

In the *Complete Bundle* you'll learn how to use TensorFlow Lite to train models on your own custom datasets, optimize them, and deploy them to your RPi for faster inference.

### 3.2.6 Caffe

Caffe is one of the original deep learning frameworks [19] and is still used today. The Caffe framework is *less* of a library and *more* of a set of tools used to *train* a deep neural network. Using the tools provided with Caffe, you can create a dataset and serialize it to an optimized format on disk, followed by training a model on the dataset.

The benefit of using Caffe is that most coprocessor units, including the Movidius NCS and Google Coral, provide tools to enable you to take a trained Caffe model and optimize/convert it for the respective coprocessor unit. Once optimized and converted, you can take the model and deploy it to the NCS or Coral for faster inference.

We'll primarily be using Caffe in the *Complete Bundle* of this book.

## 3.3 Configuring your Raspberry Pi

If you would like to configure your Raspberry Pi from scratch, just use the link below to find the most up-to-date instructions: <http://pyimg.co/ods37>.

**Remark.** If you have not already created your account on the companion website for *Raspberry Pi for Computer Vision*, refer to the first few pages of this book (immediately following the Table of Contents) for the registration link. From there, you can create your account and access the supplementary material (including install instructions).

## 3.4 Pre-configured Raspbian .img File

Configuring your Raspberry Pi for computer vision and deep learning can be *time consuming* and *tedious*, especially since:

- i. Computer vision and deep learning libraries change quickly and new versions are pushed out all the time.
- ii. The RPi CPU is slow compared to laptop/desktop systems, so any install commands take additional time to complete.

The last thing you want to do is start a compile for a particular library on the RPi and let it run overnight, only to find that it's not compatible with the rest of the packages on the system!

To that end, I have put together a *pre-configured* Raspbian .img file included in your purchase of *Raspberry Pi for Computer Vision*. This .img file ships with all the necessary computer vision, image processing, and deep learning libraries you'll need to be successful when working through this book.

All you need to do is download the Raspbian .img file, flash it to your micro-SD card, and boot — *you'll be up and running in a matter of minutes.*

Make sure you download the `Raspbian.zip` file included with your bundle to have access to the .img file. Additional instructions on how to use the pre-configured Raspbian file can be found in the `README.pdf` included with your download of the book.

## 3.5 How to Structure Your Projects

Now that you've configured your Raspberry Pi, let's take a second and explore how projects in this text are structured on disk. Make sure you've downloaded the `.zip` file containing the code for the *Hobbyist Bundle* and unarchive it — you should then see the following directory structure:

---

```
|--- Hobbyist_Code
|   |--- chapter04-opencv_tutorial
|   |--- chapter05-accessing_camera
|   |--- chapter06-changing_camera_parameters
|   |--- chapter07-time_lapse
|   |--- chapter08-automatically_starting_scripts
|   |--- chapter09-bird_feeder_monitor
|   |--- chapter10-sending_notifications
|   |--- chapter11-detecting_mail_delivery
|   |--- chapter13-remote_wildlife_detector
|   |--- chapter14-video_surveillance
|   |--- chapter15-using_multiple_cameras
|   |--- chapter16-drowsiness_detection
|   |--- chapter18-face_tracking_pantilt
|   |--- chapter19-people_counter_bs
|   |--- chapter20-traffic_counter
|   |--- chapter21-object_size
|   |--- chapter22-pill_recognition
|   |--- chapter23-opencv_optimizations
```

---

Each chapter (that includes code) has its own directory. Each directory then includes:

- i. The source code for the chapter.
- ii. A module named `pyimagesearch` which contains implementations of CV and DL algorithms. We use a module here to (1) keep our code neat and organized while (2) ensuring we can *reuse* any of our implementations in later chapters.
- iii. Any additional files needed to run the examples in the chapter.

Some chapters may have a directory named `dataset` — this is where we store any image

data required to train a model to perform a particular task (i.e., image classification or object detection).

Inside each chapter I provide instructions for running the associated Python scripts via the command line. For example, let's say I wanted to run the “*Bird Feeder Monitor*” project from Chapter 9.

I would first change into `bird_monitor` and then execute the `bird_mon.py` script:

---

```
$ cd bird_monitor
$ python bird_mon.py --conf config/mog.json
```

---

Notice how I am supplying a command line argument named `--conf`. The `--conf` switch points to a file named `mog.json`, a configuration file in JSON format. We use configuration files in many examples throughout the book to reduce the number of command line arguments we need to supply to a script.

If you are new to the command line and how to use command line arguments, I ***highly recommend*** that you read up on them (<http://pyimg.co/0e97u>) [20] before getting too far into this book.

Becoming comfortable with the command line (and how to debug errors using the terminal) is a *very* important skill to develop. Take a second now to explore the directory structure for the source code included with your purchase of this book.

You'll note that the directory structure is similar for nearly all chapters, making it easy for you to follow along. I encourage you to adopt a similar directory structure for your own projects as well.

## 3.6 Summary

In this chapter you learned about the libraries and packages we'll be using to build computer vision and deep learning projects on the Raspberry Pi. Most notably, we'll be using the Python programming language and OpenCV in the majority of our projects. OpenCV is the *de facto* standard for building image processing applications.

Additionally, we'll be using scikit-learn, scikit-image, dlib, Keras, and TensorFlow to perform feature extraction, machine learning, object tracking, deep learning, *and more!*

Since computer vision and deep learning libraries change so quickly, pushing innovation, but also breaking previous installs, I have moved much of the development environment configuration instructions to the companion website (<http://pyimg.co/ods37>).

Keeping the instructions on the website ensures that I can more easily keep the directions up to date, ensuring you are able to configure your RPi with minimal, and ideally zero, headaches.

Finally, I recognize that you may not want to configure your RPi and instead jump right in! To that end, I have provided a pre-configured Raspbian .img file with all the libraries listed above (and more) pre-installed. All you need to do is download the .img file associated with your purchase, flash it to a micro-SD card, and boot.

Take the time now to configure your RPi, whether:

- i. Using the instructions listed on the companion website.
- ii. Or simply utilize the pre-configured Raspbian .img file.

I'll be waiting for you in the next chapter where I'll provide a brief tutorial on the OpenCV library.



## Chapter 4

# A Brief Tutorial on OpenCV

Before we can start implementing computer vision applications on the Raspberry Pi, we first need to review the basics of the OpenCV library, including common functions and techniques that will be used throughout the rest of this text.

This chapter is *not* meant to be an exhaustive review of the OpenCV library nor a review of the computer vision field as a whole. If you would like a deeper review of OpenCV I suggest reading through my previous book, *Practical Python and OpenCV* (<http://pyimg.co/ppao>) [11].

And for an in-depth dive into not only OpenCV, but the computer vision field as a whole, you should refer to the PyImageSearch Gurus course (<http://pyimg.co/gurus>) [12].

Let's get started!

### 4.1 Chapter Learning Objectives

In this chapter you will:

- i. Learn about basic image processing operations with OpenCV (i.e., resizing, rotation, blurring, etc.).
- ii. Discover how to count basic object shapes using contours.
- iii. Perform image subtraction.
- iv. Utilize OpenCV for face and object detection.

### 4.2 Project Structure

Before we get started, let's first review the directory structure for this project:

```
|-- basics.py
|-- count_shapes.py
|-- detect_faces.py
|-- haarcascade_frontalface_default.xml
|-- image_sub.py
|-- images
|   |-- 30th_birthday.png
|   |-- adrian.jpg
|   |-- bg.jpg
|   |-- face_example.png
|   |-- shapes.png
```

We'll be reviewing four Python scripts:

- `basics.py`: Contains our implementation of basic image processing operations so we can use them to get hands-on experience.
- `count_shapes.py`: Performs shape counting via OpenCV and contour/outline detection.
- `image_sub.py`: Uses image background subtraction to segment the background of a scene from the foreground.
- `detect_faces.py`: Detects faces in video streams.

The `.xml` file in the project structure is Haar cascade that has been trained by the OpenCV library and then serialized to disk.

We'll load the model and then use it to perform face detection inside `detect_faces.py`.

The `images/` directory contains various images that we'll be using as examples in this chapter.

### 4.3 OpenCV Basics

In this section you will learn about basic image processing operations using OpenCV and Python. This section is a shorter, condensed version of my *OpenCV Tutorial: A Guide to Learn OpenCV*, which you can find on the PyImageSearch blog (<http://pyimg.co/jy25o>) [21].

I've also included additional scripts that are specific to the Raspberry Pi which we'll later analyze and profile for optimization purposes in Chapter 23.

Use this section to quickly help you learn the basics, but be sure to refer to either the above mentioned tutorial or *Practical Python and OpenCV* [11] for a more in-depth review.

### 4.3.1 Loading an Image with OpenCV

Let's begin by learning how to load an image from disk and then access individual pixel values.

Open up the `basics.py` file and insert the following code:

---

```
1 # import the necessary packages
2 import imutils
3 import cv2
4 import os
5
6 # load the input image and show its dimensions, keeping in mind that
7 # images are represented as a multi-dimensional NumPy array with shape:
8 # num rows (height) * num columns (width) * num channels (depth)
9 p = os.path.sep.join(["images", "30th_birthday.png"])
10 image = cv2.imread(p)
11 (h, w, d) = image.shape
12 print("width={}, height={}, depth={}".format(w, h, d))
13
14 # display the image to our screen -- we will need to click the window
15 # opened by OpenCV and press a key on our keyboard to continue execution
16 cv2.imshow("Image", image)
17 cv2.waitKey(0)
```

---

On **Lines 2-4** we import `imutils`, `cv2`, and `os`.

- i. The `cv2` import is the actual OpenCV library.
- ii. `imutils` contains a series of convenience functions that make performing basic image processing with OpenCV a bit easier.
- iii. The built-in Python `os` module allows us to easily build file paths.

Now that we have the required packages at our fingertips, let's load an image from disk into memory.

We start on **Line 9** by building the path to our input image. The input image we wish to load is named `30th_birthday.png` and resides in the `images` directory of the project. Using the `os` Python we construct the path to the input image, ensuring that the code will run on the RPi, Linux, macOS, or even Windows system (since Unix and Windows use different path separators). Given the input path, `p`, we can load the image using `cv2.imread`. As you can see from **Line 10**, we assign the result to `image`.

Our image is actually just a NumPy array. Every NumPy array has a `.shape` attribute, which corresponds to the dimensions of the NumPy matrix. We extract the height, width, and depth of the `image` on **Line 11** and then display the dimensions to our screen on **Line 12**.

It may seem a bit confusing that the height comes *before* the width in the `shape`, but think of it this way:

- i. We describe matrices by *# of rows × # of columns*
- ii. The number of *rows* is our *height*
- iii. The number of *columns* is our *width*

Therefore, the dimensions of the image, represented by a NumPy array, are ordered in terms of `(height, width, depth)`. The depth is the number of channels in the image — in our case, it's 3 as we're working with three color channels: Red, Green, and Blue.

If you were to execute the script, you would see the output of the `print` statement on **Line 10** to be:

---

```
$ python basics.py
width=600, height=457, depth=3
```

---



Figure 4.1: Loading an image and displaying its width, height, and number of channels.

To display the image on the screen using OpenCV we utilize the `cv2.imshow` function on **Line 16** (Figure 4.1).

The subsequent line waits for a keypress using `cv2.waitKey`.

We need to include the `cv2.waitKey` call, otherwise our image would display and then automatically close.

### 4.3.2 Accessing Individual Pixel Values

All images consist of pixels. Pixels are the raw building blocks of images. Images are made of pixels in a grid. A  $640 \times 480$  image has 640 columns (the width) and 480 rows (the height). There are  $640 \times 480 = 307,200$  pixels in an image with those dimensions.

Each pixel in a grayscale image has a value representing the shade of gray. In OpenCV, there are 256 shades of gray, from 0 to 255, respectfully.

Pixels in color have additional information.

There are several color spaces in image processing, but the most used one (and the most familiar) is the RGB (Red, Green, Blue) color space. In OpenCV, color images in the RGB color space have a 3-tuple associated with each pixel: (B, G, R).

Notice the ordering is BGR rather than RGB. This ordering dates back to when OpenCV was first being developed many years ago [22]. Back then, BGR was the standard ordering. Over the years, the standard has now become RGB, but OpenCV still maintains this “legacy” BGR ordering to ensure no existing code breaks. Each value in the BGR 3-tuple has a range of [0, 255].

How many color possibilities are there for each pixel in an RGB image with OpenCV?

It just takes a bit of math to figure that out:  $256 \times 256 \times 256 = 16,777,216$

Overall, there are 16,777,216 color possibilities in an RGB image.

Now that we know exactly what a pixel is, let’s see how to retrieve the value of an individual pixel in an image:

---

```

19 # access the RGB pixel located at x=430, y=200, keeping in mind that
20 # OpenCV stores images in BGR order rather than RGB (the pixel value
21 # at this location is part of the "red" in the jeep)
22 (B, G, R) = image[200, 430]
23 print("R={}, G={}, B={}".format(R, G, B))

```

---

As shown previously, our input image has a width of 600px, height of 457px, and depth of 3 pixels.

We can access individual pixel values in the array by specifying the coordinates, so long as they are within the maximum width and height. The code `image[200, 430]` yields a 3-tuple of BGR values from the pixel located at  $x=430$  and  $y=200$  (again, keeping in mind that the *height* is the number of *rows* and the *width* is the number of *columns* — take a second now to convince yourself this is true).

The resulting pixel value from **Lines 22 and 23** would be:

---

```
$ python basics.py
width=600, height=457, depth=3
R=217, G=0, B=48
```

---

This pixel value corresponds to the “red” in the Jurassic Park logo on the door of the jeep. Notice the red component is almost fully saturated, the green value is zero, and the blue value is small, indicating that we do indeed have a shade of red.

### 4.3.3 Array Slicing and Cropping

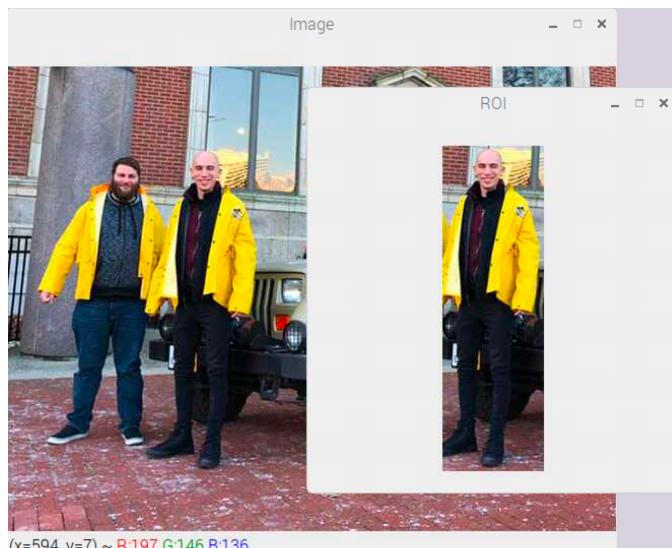


Figure 4.2: **Left:** The original input image. **Right:** Output after applying array slicing to crop a person (me) from the image. Later in this text you’ll learn how to *automatically* determine the  $(x, y)$ -coordinates needed to crop an object from an image.

Extracting “regions of interest” (ROI) is an important skill to have for image processing. In this example I will show you how to extract the person on the right (me) from our input image (Figure 4.2, *left*).

Let’s continue with our coding:

---

```

25 # extract a 100x100 pixel square ROI (Region of Interest) from the
26 # input image starting at x=150,y=80 and ending at x=250,y=400
27 roi = image[80:400, 150:250]
28 cv2.imshow("ROI", roi)
29 cv2.waitKey(0)

```

---

Array slicing is shown on **Line 27** with the format: `image[startY:endY, startX:endX]`.

This code grabs the ROI, which we then display to our screen on **Lines 28 and 29**.

As you can see from the output (Figure 4.2, right), we have successfully extracted me from the image.

So, how did I determine those starting and ending coordinates?

I determined them by manually examining the  $(x, y)$ -coordinates in Photoshop. In Section 4.3.9 you will learn how to *automatically* determine these coordinates.

#### 4.3.4 Resizing Images

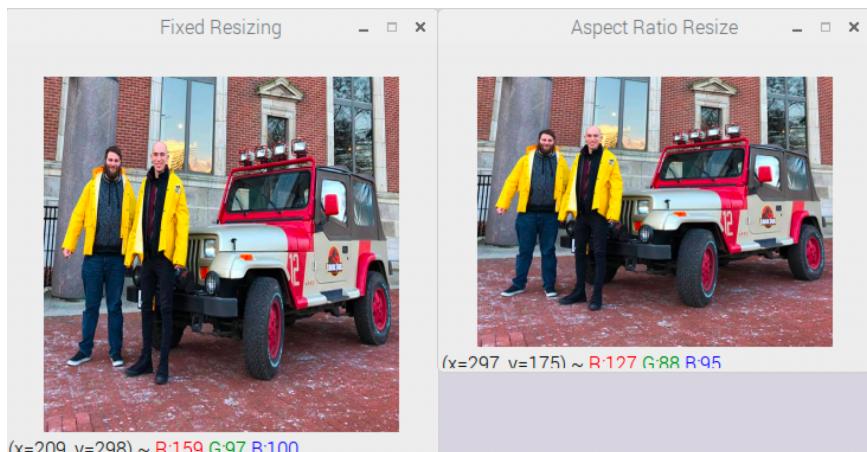


Figure 4.3: **Left:** Fixed resizing *without* maintaining aspect ratio. Notice how the image appears distorted. **Right:** Resizing an image *with* the aspect ratio maintained. The image no longer appears distorted.

Resizing an image is important for a number of reasons. First, you might want to resize a larger image to fit on your screen. Secondly, image processing is also *faster* on smaller images because there are fewer pixels to process.

In the case of deep learning, we often resize images, ignoring aspect ratio, so the volume fits into a network which requires an image be square and of a certain size.

Let's resize our image to  $300 \times 300\text{px}$ , ignoring aspect ratio:

---

```

31 # resize the image to 300x300px, ignoring aspect ratio
32 resized = cv2.resize(image, (300, 300))
33 cv2.imshow("Fixed Resizing", resized)
34 cv2.waitKey(0)

```

---

In Figure 4.3 (*left*) you can see the image has been resized; however, the image is distorted as we did not take into account the aspect ratio.

Using the `imutils.resize` function, we can resize an image while *automatically* maintaining the aspect ratio:

---

```

36 # resize the image, maintaining aspect ratio
37 resized = imutils.resize(image, width=300)
38 cv2.imshow("Aspect Ratio Resize", resized)
39 cv2.waitKey(0)

```

---

As Figure 4.3 (*right*) shows, our image has now been resized to have a width of 300px, but this time ensuring the height is resized proportionally, maintaining the aspect ratio.

**Remark.** If you are using a Raspberry Pi Zero, you'll need to change the default `inter` method from `cv2.INTER_AREA` to `cv2.INTER_NEAREST`, like this:

```
image = imutils.resize(image, width=500, inter=cv2.INTER_NEAREST)
```

For whatever reason, using the `cv2.INTER_AREA` interpolation on the RPi Zero will cause a segfault and your script will crash. I'm not sure why that is, but I hope it will be resolved in future versions of OpenCV.

### 4.3.5 Rotating Images

For this next example we are going to rotate our input image:

---

```

41 # rotate the image 45 degrees clockwise
42 rotated = imutils.rotate(image, -45)
43 cv2.imshow("Rotation", rotated)
44 cv2.waitKey(0)

```

---

The `imutils.rotate` function accepts an input `image` along with the number of degrees to rotate the image.

**Positive values correspond to clockwise rotation and negative angles to clockwise rotation** — here we are rotating our image -45 degrees, clockwise.

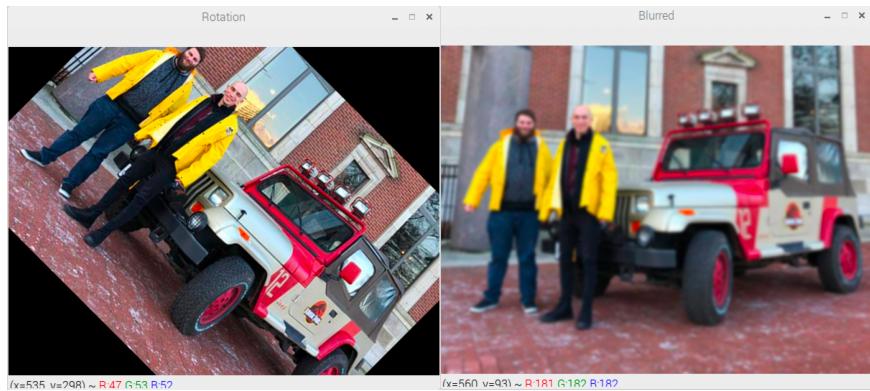


Figure 4.4: **Left:** Rotating an image with OpenCV about the center point requires three steps: (1) compute the center point using the image width and height, (2) compute a rotation matrix with `cv2.getRotationMatrix2D`, and (3) use the rotation matrix to warp the image with `cv2.warpAffine`. **Right:** Blurring is an important preprocessing step as it can be used to reduce high frequency noise, enabling our code to focus on the structural components of the image rather than the foreground details (which may confuse our algorithms).

The output of the rotation can be seen in Figure 4.4 (*left*).

Note how our image has been successfully rotated; however, parts of the image are cut off. This behavior is due to the fact that OpenCV does not *automatically* resize the image matrix to hold the full rotated image.

In some cases this behavior is acceptable and in other cases it is not.

If you need to have the *entire* image after rotation you can use the `imutils.rotate_bound`, which is covered in my tutorial, *Rotate images (correctly) with OpenCV and Python* (<http://pyimg.co/7xnk6>) [23].

### 4.3.6 Blurring Images

In many image processing pipelines, we must blur an image to reduce high-frequency noise, making it easier for our algorithms to detect and understand the actual *contents* of the image rather than just *noise* that will “confuse” our algorithms.

Blurring and smoothing an image is a very easy in OpenCV.

One of the most common methods you’ll encounter is Gaussian blurring:

---

```

46 # apply a Gaussian blur with a 11x11 kernel to the image to smooth it,
47 # useful when reducing high frequency noise
48 blurred = cv2.GaussianBlur(image, (11, 11), 0)
49 cv2.imshow("Blurred", blurred)
50 cv2.waitKey(0)
```

---

Here we perform a Gaussian blur with an  $11 \times 11$  kernel, the result of which is shown in Figure 4.4 (right). Larger kernels will yield a more blurry image, while smaller kernels a less blurry image.

To read more about kernels and their role in computer vision and image processing, refer to this tutorial (<http://pyimg.co/8om5g>) [24], along with the PyImageSearch Gurus course [25].

### 4.3.7 Drawing Methods

A good place to start with drawing methods is to learn how to draw a rectangle, circle, and line on an input image, as well as overlay text on the image. Here's how it's done:

---

```

52 # draw a rectangle, circle, and line on the image, then draw text on
53 # the image as well
54 cv2.rectangle(image, (150, 80), (250, 400), (255, 0, 255), 5)
55 cv2.circle(image, (490, 240), 30, (255, 0, 0), -1)
56 cv2.line(image, (0, 0), (600, 457), (0, 0, 255), 5)
57 cv2.putText(image, "You're learning OpenCV!", (10, 435),
58     cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
59 cv2.imshow("Drawing", image)
60 cv2.waitKey(0)

```

---

On **Line 54** we draw a rectangle on our `image` via the `cv2.rectangle` function. Here we supply the input `image`, the starting  $(x, y)$ -coordinates of the rectangle, the ending  $(x, y)$ -coordinates of the rectangle, the color of the rectangle (in BGR ordering), and finally the thickness of the rectangle.

When executed, this code will draw a pink rectangle surrounding myself in the image (Figure 4.5).

**Line 55** draws a circle via the `cv2.circle` function. This method requires that we pass in the input `image`, the center  $(x, y)$ -coordinates of the circle, the radius of the circle, the color of the circle, and finally the thickness of a circle. Using a negative value for the thickness indicates that the circle should be drawn *filled in*.

Looking at Figure 4.5 again, you can see that we have drawn a blue circle overtop the *Jurassic Park* logo on the jeep.

**Line 56** draws a line on our image. Similar to the `cv2.rectangle` function, we must supply the `image`, the starting  $(x, y)$ -coordinates of the line, the ending  $(x, y)$ -coordinates of the line, the color of the line, and the thickness.

Referring once again to Figure 4.5, you'll see that we have drawn a red line from the upper-left corner of the image to the bottom-right corner.

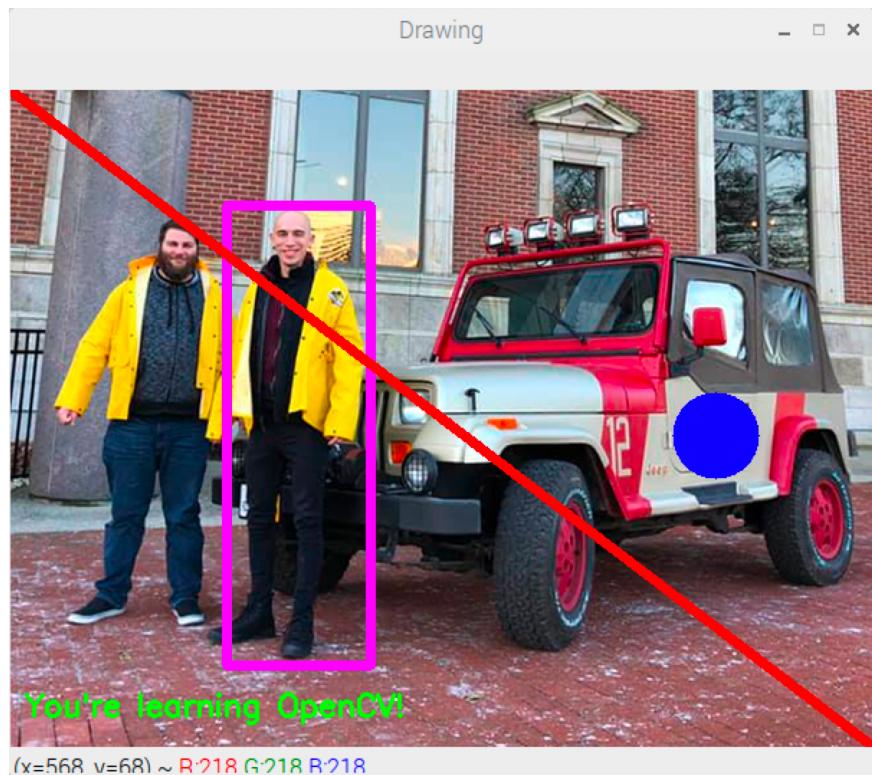


Figure 4.5: Drawing functions with OpenCV are easy and intuitive to use. Refer to the source code in this section for details on how to use OpenCV's drawing functions.

Finally, **Lines 57 and 58** enable us to draw text on our image via the `cv2.putText` function. This method requires that we pass in the input `image` to draw on, the text string itself, the `(x, y)`-coordinates of where the text is to start, the font type, font size, BGR color tuple, and finally the thickness.

Look at Figure 4.5 one final time to see that we have drawn the text “*You’re learning OpenCV!*” in the bottom-left corner of the image.

We’ll be using these drawing functions throughout the text, so take a second now to play around with them and familiarize yourself with how they work.

#### 4.3.8 Counting Objects

Now that we’ve learned about basic OpenCV functions, let’s put them to work to build a complete image processing application used to count the number of objects in the image.

Open up the `count_shapes.py` file in your directory structure and insert the following code:

---

```

1 # import the necessary packages
2 import argparse
3 import imutils
4 import cv2
5
6 # construct the argument parser and parse the arguments
7 ap = argparse.ArgumentParser()
8 ap.add_argument("-i", "--image", required=True,
9     help="path to the input image")
10 args = vars(ap.parse_args())

```

---

**Lines 2-4** handle our imports while **Lines 7-10** parse our command line arguments. We only need a single argument here, `--image`, which is the path to our input image.

Given the path to the input image, let's load it from disk, and utilize our image processing functions from Section 4.3.6 to prepare it for object counting:

---

```

12 # load the input image and convert it to grayscale
13 image = cv2.imread(args["image"])
14 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
15
16 # blur the image (to reduce false-positive detections) and then
17 # perform edge detection
18 blurred = cv2.GaussianBlur(gray, (3, 3), 0)
19 edged = cv2.Canny(blurred, 50, 130)

```

---



Figure 4.6: **Left:** Original input image. **Middle:** Blurring the image to reduce noise. **Right:** Performing edge detection to reveal the boundaries of each shape in the image.

On **Line 13** we load our input `image` from disk.

As you can see from Figure 4.6 (*left*), we'll be building a simple shape counting application. Using OpenCV, we'll be able to count the number of shapes in the image.

But before we can do that, we first need to preprocess the image:

- We first convert the image to grayscale on **Line 14**.
- Given the grayscale image, we blur it on **Line 18** to help reduce noise and false-positive shape detections (Figure 4.6, *middle*).
- The final preprocessing step is to take the `blur` image and apply **edge detection (Line 19)**. Edge detection reveals the outlines/boundaries of each shape in the image (Figure 4.6, *right*).

At this point we have been able to successfully identify the outlines of each shape in the image — **but how do we access these outline and count them?**

The solution is to apply **contour detection**:

---

```

21 # find contours in the edge map and initialize the total number of
22 # shapes found
23 cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
24     cv2.CHAIN_APPROX_SIMPLE)
25 cnts = imutils.grab_contours(cnts)
26 total = 0

```

---

**Lines 23 and 24** perform contour detection via the `cv2.findContours` function. This method processes the `edged` image and accumulates a list of  $(x, y)$ -coordinates for each separate outline.

Since the return signature of the `cv2.findContours` function is different between OpenCV 2.4, OpenCV 3, and OpenCV 4, we use the `imutils.grab_contours` method to parse the returned tuple and return our list of contours (**Line 25**).

Finally, **Line 26** initializes an integer, `total`, which will be used to count the total number of shapes in the image.

Let's now perform the shape counting:

---

```

28 # loop over the contours one by one
29 for c in cnts:
30     # if the contour area is small, then the area is likely noise, so
31     # we should ignore the contour
32     if cv2.contourArea(c) < 25:
33         continue
34
35     # otherwise, draw the contour on the image and increment the total
36     # number of shapes found
37     cv2.drawContours(image, [c], -1, (204, 0, 255), 2)
38     total += 1

```

---

On **Line 29** we loop over each of the contours.

**Line 32** computes the area of the contour and checks to see if the contour is smaller than 25 pixels. Even though we blurred our image, it's possible that noisy, false-positive detections slipped through — checking the contour area and ignoring regions with a small area (via the `continue` call on **Line 33**) is a good way to help reduce these false-positive detections.

Provided our contour, `c`, passes the area test, we then draw it on our `image` via the `cv2.drawContours` function (**Line 37**).

The `total` object count is then incremented on **Line 38**.

The final step is to show the output image and final object count:

---

```
40 # show the output image and the final shape count
41 print("[INFO] found {} shapes".format(total))
42 cv2.imshow("Image", image)
43 cv2.waitKey(0)
```

---

To execute the `count_shapes.py` script, just execute the following command:

---

```
$ python count_shapes.py --image images/shapes.png
[INFO] found 17 shapes
```

---

Figure 4.7 displays our output image. Note how our script has successfully (1) counted each of the 17 shapes and then (2) drawn the outline of each shape on the image.

While a basic image processing method, contour detection is a computationally efficient technique that can easily run in real-time on the Raspberry Pi.

As we'll see in Chapter 14, we'll be able to build an entire motion detection/home surveillance system using contour detection.

Finally, if you are interested in learning more about contours, including more advanced contour techniques that can even recognize and identify *the type of shape* in our example image, be sure to take a look at this post on the PyImageSearch blog (<http://pyimg.co/jy25o>) [21]. This tutorial will give you additional experience working with contours and associated contour properties.

### 4.3.9 Image Subtraction

One of the most common methods of performing background subtraction is **image subtraction**.



Figure 4.7: By using contours we are able to count the total number of shapes in the image.

When performing background subtraction we make the assumption that we have two images — the **background** and the **foreground**. *The background is assumed to be a static representation of the scene* (i.e., without any motion or objects that we wish to detect).

An example of such an image can be seen in Figure 4.8 (*left*) — this image is captured from the computer on my office desk. I wish to use image subtraction to detect if anyone is sitting at my desk and using my computer.

**The foreground image contains any potential objects or regions of interest that we want to detect.** Figure 4.8 (*right*) shows an example of a foreground image (i.e., myself sitting at my computer).

Using image subtraction, we'll be able to **detect** the differences between the two images and even **locate** where in the image the difference is.

Let's get started!

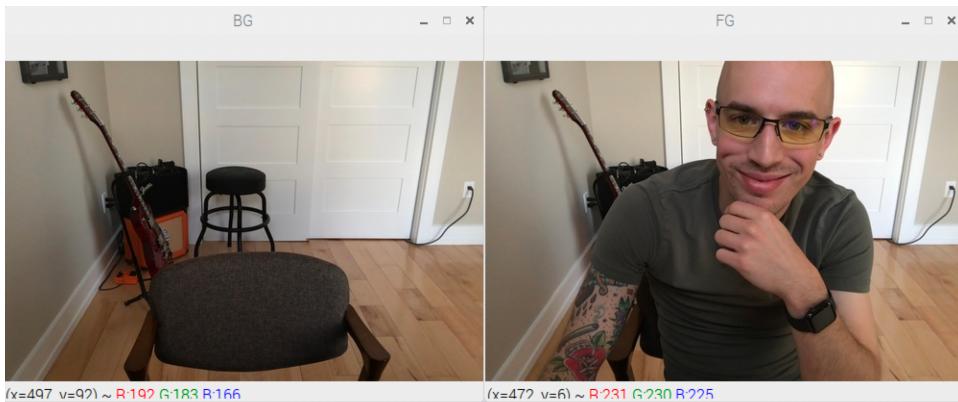


Figure 4.8: **Left:** The "background" image which we assume to be a static scene. **Right:** The "foreground" image. Here you can see that I am sitting in my chair. By applying image subtraction we'll be able to find the difference between the two images (i.e., the person in the chair).

Open up the `image_sub.py` file and insert the following code:

---

```

1 # import the necessary packages
2 import numpy as np
3 import argparse
4 import imutils
5 import cv2
6
7 # construct the argument parser and parse the arguments
8 ap = argparse.ArgumentParser()
9 ap.add_argument("-b", "--bg", required=True,
10     help="path to background image")
11 ap.add_argument("-f", "--fg", required=True,
12     help="path to foreground image")
13 args = vars(ap.parse_args())

```

---

**Lines 2-5** handle our imports while **Lines 8-13** parse our command line arguments. We'll need two arguments for this script:

- `--bg`: The path to the *background* image.
- `--fg`: The path to the *foreground* image.

Next, let's load these two images from disk and convert them to grayscale:

---

```

15 # load the background and foreground images
16 bg = cv2.imread(args["bg"])
17 fg = cv2.imread(args["fg"])
18
19 # convert the background and foreground images to grayscale

```

---

---

```
20 bgGray = cv2.cvtColor(bg, cv2.COLOR_BGR2GRAY)
21 fgGray = cv2.cvtColor(fg, cv2.COLOR_BGR2GRAY)
```

---

The grayscale background and foreground images can be seen in Figure 4.9 (*top-left and top-right*).

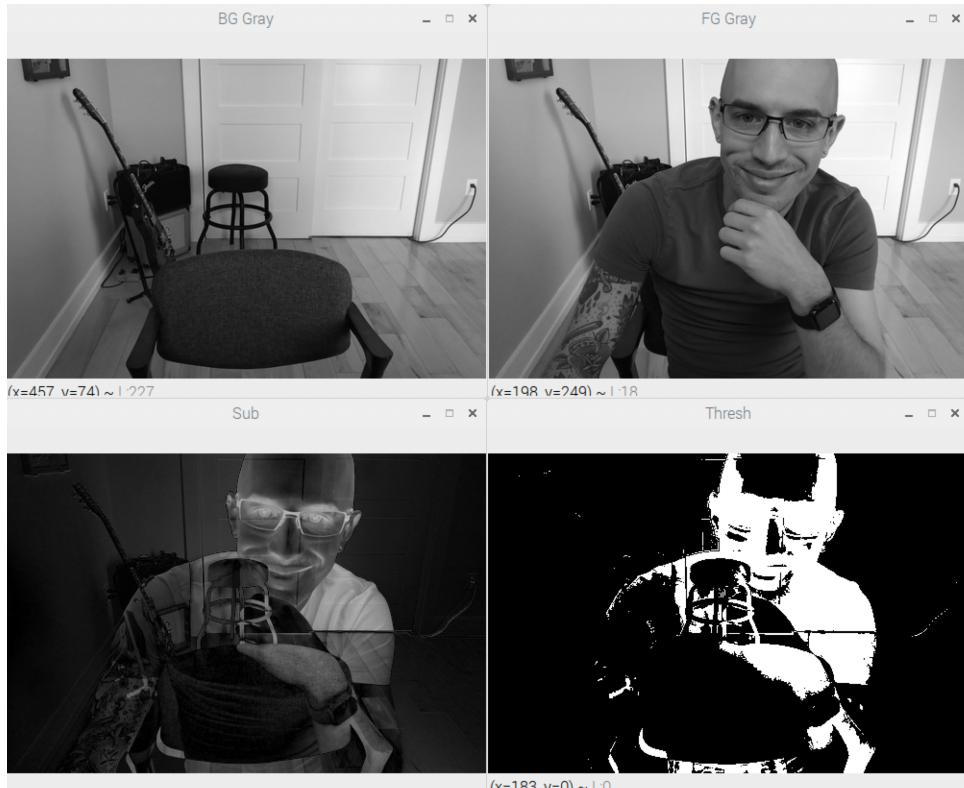


Figure 4.9: **Top-left:** Grayscale representation of the background image. **Top-right:** Foreground grayscale image. **Bottom-left:** Subtracting the foreground from the background creates a "ghost-like effect" of where the differences between the foreground and background are. **Bottom-right:** Thresholding the difference image to create a binary representation of where the differences are.

Given both the grayscale representations of the `bg` and `fg`, we can perform image subtraction:

---

```
23 # perform background subtraction by subtracting the foreground from
24 # the background and then taking the absolute value
25 sub = bgGray.astype("int32") - fgGray.astype("int32")
26 sub = np.absolute(sub).astype("uint8")
```

---

On **Line 25** we subtract the foreground from the background.

We convert the images to 32-bit integers to ensure we can have *negative* values in the image (by default, OpenCV represents images as 8-bit unsigned integers which cannot take

on negative values).

Once we have the `sub`, we then take the absolute value (so we have absolute pixel differences) and convert back to the 8-bit unsigned integer type that OpenCV expects.

The output of the background subtraction can be seen in Figure 4.9 (*bottom-left*) — note the “ghost-like” effect in the image. This “ghostliness” represents the absolute pixel value differences between the grayscale images. Regions of the image that are **dark** contain **no difference** while regions of the image that are **light** are regions that do **contain difference**.

Gradients between white and black represent varying levels of difference.

In order to actually detect and extract the regions of the image that contain difference, we’ll be using contour detection, just as we did in Section 4.3.8.

The issue here is that contour detection assumes we are working with a **binary image**. A binary image only contains two pixel values — 0 (black, for background), and 255 (white, for foreground). But as you can see, our `sub` image is grayscale, having pixel values that can range between 0–255.

In order to apply contour detection we need to take this grayscale image and then binarize it.

The solution?

Use thresholding to take the `sub` grayscale image and binarize it:

---

```

28 # threshold the image to find regions of the subtracted image with
29 # larger pixel differences
30 thresh = cv2.threshold(sub, 0, 255,
31     cv2.THRESH_BINARY | cv2.THRESH_OTSU) [1]
32
33 # perform a series of erosions and dilations to remove noise
34 thresh = cv2.erode(thresh, None, iterations=1)
35 thresh = cv2.dilate(thresh, None, iterations=1)

```

---

**Lines 30 and 31** perform thresholding via Otsu’s thresholding method [26]. This method allows us to *automatically* threshold the image into background or foreground *without* having to worry about tuning parameters. The output of Otsu’s thresholding can be seen in Figure 4.9 (*bottom-right*).

After thresholding, the `thresh` image is now binary, with values of 0 being background and values of 255 belonging to the foreground; however, we’re not quite done yet.

We first apply an *erosion* that will “eat away” at the foreground, allowing us to remove noise (**Line 34**). After erosion, it’s common to apply a *dilation* which “regrows” the foreground (**Line 35**). Provided that the erosion removed the noise, we don’t need to worry about regrowing it

via the dilation.

The output of the thresholding, erosion, and dilation operations can be seen in Figure 4.10. Notice how we have a series of white blobs on a black ground — these white blobs represent the actual *foreground difference* between our images.



Figure 4.10: Using a series of erosions and dilations to clean up our thresholded image and remove noise.

The next step is to take this `thresh` image, and then apply contour detection to extract each of the individual regions:

---

```

37 # find contours in the thresholded difference map and then initialize
38 # our bounding box regions that contains the *entire* region of motion
39 cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
40     cv2.CHAIN_APPROX_SIMPLE)
41 cnts = imutils.grab_contours(cnts)
42 (minX, minY) = (np.inf, np.inf)
43 (maxX, maxY) = (-np.inf, -np.inf)

```

---

On **Lines 39-41** we perform contour detection, just like in Section 4.3.8.

**Lines 43 and 44** define four variables which will be used to compute a bounding box that encompasses **all** foreground regions, thereby giving us a rectangular region that contains all significant differences between the background and foreground.

To see how we can update these values, let's move on to the next code block:

---

```

45 # loop over the contours
46 for c in cnts:
47     # compute the bounding box of the contour
48     (x, y, w, h) = cv2.boundingRect(c)
49
50     # reduce noise by enforcing requirements on the bounding box size
51     if w > 20 and h > 20:
52         # update our bookkeeping variables

```

---

---

```

53     minX = min(minX, x)
54     minY = min(minY, y)
55     maxX = max(maxX, x + w - 1)
56     maxY = max(maxY, y + h - 1)

```

---

On **Line 46** we loop over all contours found in the `thresh` image.

For each contour, `c`, we compute the bounding box rectangle via the `cv2.boundingRect` function.

The `cv2.boundingRect` function examines the  $(x, y)$ -coordinates of the contour and then computes the *smallest possible* rectangle that encompasses *all* of the coordinates.

Therefore, this method returns four values:

- `x`: The starting  $x$ -coordinate of the rectangle.
- `y`: The starting  $y$ -coordinate of the rectangle.
- `w`: The width of the rectangle.
- `h`: The height of the rectangle.

**Line 51** checks to ensure that the bounding box has a minimum width and height of 20 pixels (to help filter out small regions of noise).

Provided the width and height pass the test, we update our bookkeeping variables on **Lines 53-56**.

Using this combination of `min` and `max` functions we can determine the smallest possible rectangular region that encompasses *all* bounding boxes — take a second now to make sure you understand this operation.

Our final code block handles drawing the minimum enclosing rectangle on the `fg` and displaying it to our screen:

---

```

58 # draw a rectangle surrounding the region of motion
59 cv2.rectangle(fg, (minX, minY), (maxX, maxY), (0, 255, 0), 2)
60
61 # show the output image
62 cv2.imshow("Output", fg)
63 cv2.waitKey(0)

```

---

To execute this script, open up a command line and issue the following command:

---

```
$ python image_sub.py --bg images/bg.jpg --fg images/adrian.jpg
```

---

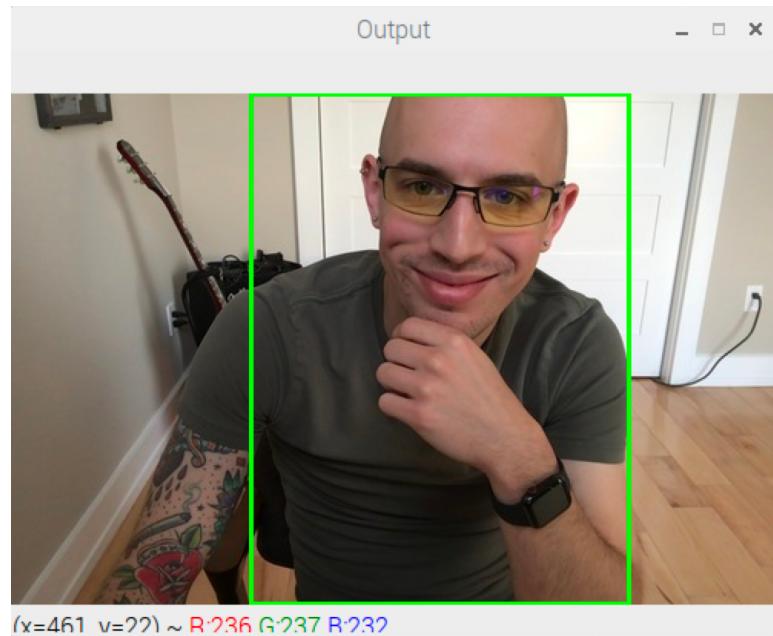


Figure 4.11: By using image subtraction we can detect and mark the differences between two images. While simple, this technique is *extremely* useful. We'll be building on this method in later chapters in this text.

Figure 4.11 visualizes our output — notice how we have successfully located the difference between the two images (i.e., me sitting in the chair in front of my desk).

As we'll see in Chapter 14, these techniques, while simplistic, make it possible to build motion detection and home surveillance systems with minimal effort.

#### 4.3.10 Object and Face Detection

The final script we'll be reviewing is `detect_faces.py`. As the name suggests, this script will be used to detect the presence of faces in an image.

We'll be using OpenCV's Haar cascades for face detection as they are *built into the library* and do not require us to train the model (which is a more advanced technique).

Keep in mind that **face detection** is different than **face recognition**. When performing face detection, we're simply *locating* the bounding box coordinates of an image where a face is found — face detection does not tell us *anything* about the actual *identity* of the face.

Face recognition, on the other hand:

- Accepts an input face (which is found via face detection).
- Performs the actual identification of the person.

Face detection is thus a necessary step *before* face recognition can be applied. We'll be covering face recognition on the Raspberry Pi inside the *Hacker Bundle* of this text. All that said, let's learn how to perform face detection with OpenCV!

Open up the `detect_faces.py` file in the project structure and insert the following code:

---

```

1 # import the necessary packages
2 import argparse
3 import cv2
4
5 # construct the argument parser and parse the arguments
6 ap = argparse.ArgumentParser()
7 ap.add_argument("-i", "--image", required=True,
8     help="path to the input image")
9 ap.add_argument("-d", "--detector", required=True,
10    help="path to Haar cascade face detector")
11 args = vars(ap.parse_args())

```

---

**Lines 2 and 3** handle our imports while **Lines 5-9** parse our command line arguments.

We need to supply two command line arguments to our script:

- `--image`: The path to our input image we wish to apply face detection on.
- `--detector`: The path to the face detector `.xml` model.

Given the path to our input image we can load it from disk and then convert it to grayscale:

---

```

13 # load our image and convert it to grayscale
14 image = cv2.imread(args["image"])

```

---

We are now ready to perform face detection on the `gray` image:

---

```

17 # load the face detector and detect faces in the image
18 detector = cv2.CascadeClassifier(args["detector"])
19 rects = detector.detectMultiScale(gray, scaleFactor=1.05, minNeighbors=9,
20     minSize=(40, 40), flags=cv2.CASCADE_SCALE_IMAGE)
21 print("[INFO] detected {} faces".format(len(rects)))

```

---

We load our face `detector` from disk on **Line 18**.

The type of face detector we're using is called a **Haar cascade** which was introduced by Viola and Jones in their 2001 paper, *Rapid Object Detection using a Boosted Cascade of Simple Features* [27].

While Haar cascades are nearly 20 years old, they are still used today in certain situations. We still use them because they are *very* fast, and compared to more advanced object detectors, including HOG + Linear SVM [28], and deep learning-based detectors [29, 30, 31, 32, 33, 34], they require comparatively little computation, making them efficient to use on resource-constrained devices such as the RPi.

**Lines 19-20** call the `.detectMultiScale` function of the `detector` which is used to detect the actual faces in the image.

This function returns `rects`, a *list* of the **bounding box locations** of each face in the image. Each bounding box is represented by four values:

- `x`: The starting x-coordinate of the rectangle.
- `y`: The starting y-coordinate of the rectangle.
- `w`: The width of the rectangle.
- `h`: The height of the rectangle.

Taken together, these values allow us to locate each face in the image.

The length of the `rects` determines the *total number of faces* in the image. We then display the number of faces found to our terminal on **Line 21**.

Now that we know the locations of the faces in the image, let's draw our `rects` on the image:

---

```

23 # loop over the bounding boxes and draw a rectangle around each face
24 for (x, y, w, h) in rects:
25     cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
26
27 # show the detected faces
28 cv2.imshow("Faces", image)
29 cv2.waitKey(0)

```

---

On **Line 24** we loop over each of the bounding boxes.

We then take the bounding box coordinates and draw the rectangle surrounding the face on **Line 25** via the `cv2.rectangle` function (which we learned about in Section 4.3.7). **Lines 28 and 29** display our output image to our screen.

To execute the script, issue the following command:

---

```
$ python detect_faces.py --image images/faces_example.png \
--detector haarcascade_frontalface_default.xml
```

---

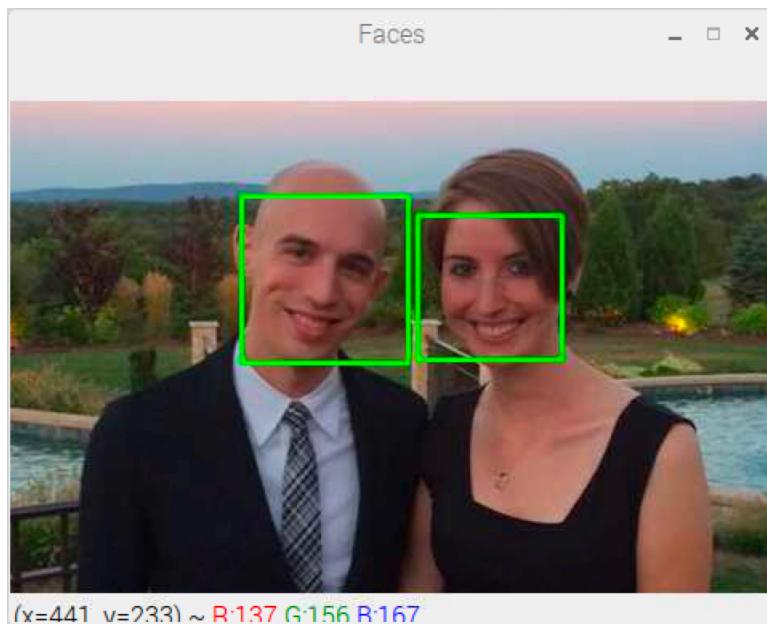


Figure 4.12: Performing face detection with OpenCV.

Figure 4.12 visualizes our output. Notice how two faces are detected in the image — mine and my wife's.

As you can see, performing face detection via OpenCV is quite easy using the pre-supplied Haar cascade model!

The problem with Haar cascades is that they are especially prone to false positives and they sometimes even miss face detections altogether. To obtain reasonable accuracy with a Haar cascade you may have to tune the values `.detectMultiScale` on a per-image basis, which does not allow for robust face detection. That said, Haar cascades are computationally efficient, which makes them suitable for real-time face detection on the Raspberry Pi.

We'll learn about the more advanced (1) HOG + Linear SVM and (2) deep learning-based face detectors in the *Hacker Bundle* and *Complete Bundle* of this text.

## 4.4 Summary

In this chapter you learned about basic OpenCV functions and computer vision techniques that we'll commonly use in this text. Our implementations will start off basic and then build upon these fundamentals, eventually creating more complex computer vision applications on the Raspberry Pi.

That said, this chapter is *not* meant to be an exhaustive review of the OpenCV library, nor a review of the computer vision field as a whole. If this is your first time working with

OpenCV I would **strongly encourage** you to read through *Practical Python and OpenCV* (PPaO) (<http://pyimg.co/ppao>) [11] in tandem with this text — PPaO will get you up to speed quickly and ensure you get the most value out of *Raspberry Pi for Computer Vision*.

Secondly, if you would like to study computer vision in depth, be sure to take a look at PyImageSearch Gurus course (<http://pyimg.co/gurus>) [12]. The course is similar to a college-level survey course in computer vision but much more hands on and practical.

Now that you have a handle on basic computer vision functions, let's put them to use! In the next chapter you'll learn how to access your RPi camera module/USB webcam via OpenCV.



## Chapter 5

# Accessing RPi Camera/USB Webcam

Before you can enable your Raspberry Pi to “see” we first need to give it a set of “eyes”. In the context of computer vision, our “eyes” will be cameras.

When it comes to the RPi you have two choices of camera type:

- A Camera Serial Interface (CSI) camera, called the **Raspberry Pi camera module**, that connects to pins on the RPi board directly.
- A **USB-based webcam** that connects to the RPi via one of its USB ports.

We’ll start this chapter with a brief discussion of both types of webcams. I’ll then show you how to access *either* type of camera using a unified Python class called `VideoStream` — this class is compatible with *both* the RPi camera module and USB webcam, making it straightforward and simple to bring “eyes” to your Pi, regardless of which camera you are using.

In Chapter 6 we’ll go into more detail regarding changing camera parameters for both the RPi camera module and USB cameras.

### 5.1 Chapter Learning Objectives

In this chapter you will:

- i. Learn about the RPi camera module and USB webcams
- ii. Pick a camera and get started with it
- iii. Learn about the `VideoStream` class and how it can access *either* a USB webcam or RPi camera module using OpenCV

- iv. Discover my tips, suggestions, and best practices when working with cameras on the RPi (including how to debug common problems).

Let's go ahead and get started!



Figure 5.1: **Left:** A Raspberry Pi camera module that connects to your RPi via CSI. **Right:** An example of a USB webcam (Logitech C920) that connects to your Raspberry Pi via a USB port.

## 5.2 Should I use a RPi Camera Module or USB Webcam?

When working with the Raspberry Pi, you have two choices for a cameras: a RPi camera module (Figure 5.1, *left*) or a USB webcam (Figure 5.1, *right*).

So, which one is better? The answer is usually dependent on your use case, but I'll typically say that if you are brand new to the world of computer vision on the RPi, it doesn't matter which camera you decide to use. Pick whichever one you have laying around or use the camera module included with your RPi kit (if you purchased a kit, of course).

**At this point your goal is to simply get started.** Don't agonize over which camera is "better", as it simply doesn't matter right now. Your goal is to connect your camera to the RPi and start learning.

If you *already* have experience with the RPi and have a very specific project or goal in mind, then you may want to consider your camera options a bit more.

To start, the RPi camera module is cheap and affordable, but more expensive USB webcams can often obtain a much higher quality frame capture. If you want a higher quality image, you'd be better off going with a USB webcam.

You also need to consider any hardware-level operations the camera should be required to perform. Many USB webcams have the ability to auto-focus which may or may not be a good feature for your particular project. As we'll see in Chapter 22 when building a prescription pill identification system, a camera with auto-focus can actually *hurt* our system, making it harder for pills to be correctly identified.

However, there are a number of *optimization* reasons you may choose a CSI camera, such as the RPi camera module. The RPi camera module connects *directly* to the RPi GPU, leaving precious CPU cycles available for additional processing of the frames, running more computationally-expensive computer vision algorithms, or simply taking care of system tasks.

USB webcams, on the other hand, are not only limited by the data transfer speed of USB, but also the fact that moving data from USB to CPU to your Python script is not “free” — more CPU cycles will be eaten up compared to what is required by a CSI camera.

**There is certainly a tradeoff between the two types of cameras.** My personal suggestion is to have both CSI and USB cameras to test your projects and try to avoid making blanket assumptions such as “*The CSI camera will always help me reduce CPU load*” or “*USB webcams will always give me better image quality*” — those types of assumptions will only lead to trouble later.

**Instead, try your project out on both types of cameras and let your project guidelines, goals, and empirical results guide you on which camera to utilize.**

## 5.3 Accessing Cameras via the `VideoStream` Class

To access *either* a Raspberry Pi camera module or a USB webcam, we’ll be using the `VideoStream` class. This class is actually part of my `imutils` package [35], a series of convenience functions to make working with OpenCV easier.

I implemented this class to not only make it seamless to access a RPi camera module or USB camera, **but also to help with performance!** Typical video stream applications you see in online tutorials are single threaded, meaning that the entire main thread of execution will block until the next frame is polled from a camera.

In most computer vision applications, this blocking operation is wasteful — our goal is to process our frames *as fast as possible* and if we have to wait for a frame to be read, that’s unnecessary overhead that will slow down our frames per second processing.

Instead, what we can do is leverage **threading**. Internally, the `VideoStream` class starts a thread that constantly pulls frames from the camera, ensuring that whenever we are ready for a new frame, **it will already be there waiting for us!**

If you’re interested in learning more about the optimizations utilized inside the `VideoStream` class, be sure to refer to the following three tutorials on the PyImageSearch blog, which discuss the internal optimizations (and rational behind them) in depth:

- *Increasing webcam FPS with Python and OpenCV* (<http://pyimg.co/les2l>) [36]
- *Increasing Raspberry Pi FPS with Python and OpenCV* (<http://pyimg.co/3igfs>) [37]

- *Unifying picamera and cv2.VideoCapture into a single class with OpenCV (<http://pyimg.co/sdopi>) [38]*

But for now, let's look at an example of how to use the `VideoStream` class. Taking a look at the directory structure for this project you'll see that it's quite simple:

---

```
|-- access_camera.py
```

---

We only have a single Python file, `access_camera.py`, which as the name suggests will enable us to access our camera attached to the RPi (regardless of whether it's a CSI camera or USB webcam). Open up the `access_camera.py` file now and insert the following code:

---

```
1 # import the necessary packages
2 from imutils.video import VideoStream
3 import imutils
4 import time
5 import cv2
6
7 # initialize the video stream and allow the cammera sensor to warmup
8 print("[INFO] starting video stream...")
9 vs = VideoStream(src=0).start()
10 #vs = VideoStream(usePiCamera=True, resolution=(640, 480)).start()
11 time.sleep(2.0)
```

---

**Lines 1-5** we import our required Python packages. As you can see, we'll be using the implementation of `imutils` of the `VideoStream` class. If you do not have `imutils` installed on your RPi yet, refer to Chapter 3 where instructions are provided to configure your development environment.

**Line 9** creates a `VideoStream` object for a **USB webcam**. Take note of the `src` argument — this parameter controls the index of the camera on your RPi. A value of 0 indicates the first camera, 1 for the second camera, and so on.

If you want to instead utilize your **Raspberry Pi camera module** you should *comment out Line 9* and then *uncomment Line 10*.

As **Line 10** demonstrates, you can supply the `usePiCamera=True` argument to `VideoStream` to indicate that the RPi camera module should be used instead.

Also take note of the `resolution` parameter — we can specify this parameter if we know the spatial dimensions of the frames we want to read from the camera sensor. Specifying the resolution frees up CPU cycles, as the CSI and GPU will perform the resizing for us if no dimensions are specified.

**Line 11** pauses execution of the script for two seconds, enabling the camera sensor to warm up.

Now that we have instantiated the `VideoStream` class, we can start polling frames from the camera:

---

```

13 # loop over the frames from the video stream
14 while True:
15     # grab the frame from the video stream and resize it to have a
16     # maximum width of 400 pixels
17     frame = vs.read()
18     frame = imutils.resize(frame, width=400)
19
20     # show the output frame
21     cv2.imshow("Frame", frame)
22     key = cv2.waitKey(1) & 0xFF
23
24     # if the `q` key was pressed, break from the loop
25     if key == ord("q"):
26         break
27
28     # do a bit of cleanup
29     cv2.destroyAllWindows()
30     vs.stop()

```

---

**Line 17** reads the next `frame` from our camera (again, regardless of whether a USB webcam or RPi camera module is being used). **Line 18** takes the `frame` and then resizes it to have a width of 400 pixels. If you are using the RPi camera module you can actually comment out this line and instead specify the resolution to the `VideoStream`.

We then take the resized frame and display it on our screen via **Lines 21 and 22**. If the `q` key was pressed on our keyboard then we break from the `while` loop (**Lines 25 and 26**) and cleanup our video stream pointer (**Line 30**).

At this point, our script to access our camera on the RPi is fully implemented! In the next section we'll actually execute the script and look at the results.

## 5.4 Running Our Script

To access your camera on the Raspberry Pi, open up a terminal and execute the following command:

---

```
$ python access_camera.py
[INFO] starting video stream...
```

---

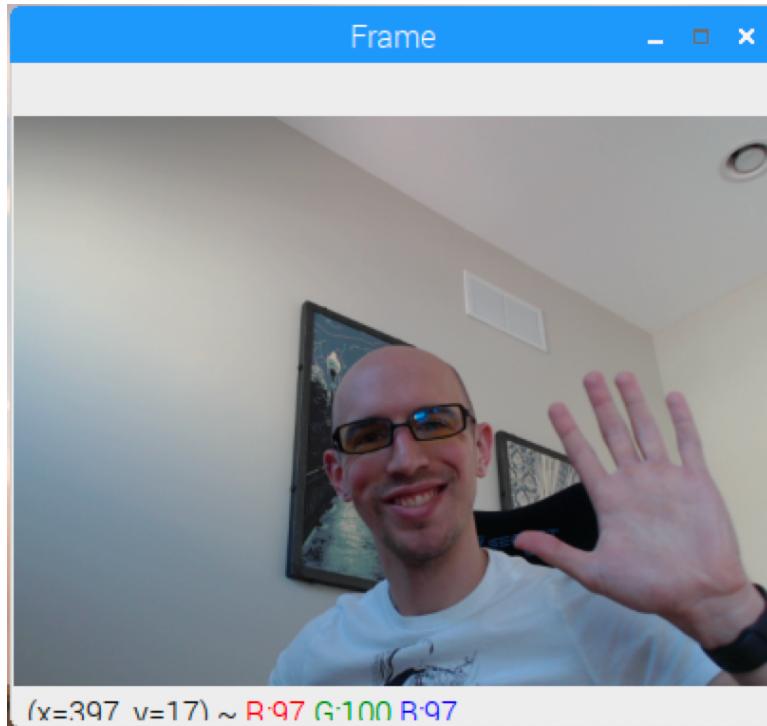


Figure 5.2: Accessing a camera on the Raspberry Pi programmatically via the `VideoStream` class. Our code is compatible with *both* USB webcams and the RPi camera module.

**Remark.** If you are using the RPi camera module, make sure you have enabled it via the `raspi-config` command. Execute `sudo raspi-config`, select Interfacing Options, followed by Camera. From there you can enable your RPi camera module (you may need to reboot your RPi for the change to take effect).

Your output should be similar to mine in Figure 5.2. Notice how frames are being read from your video stream and then displayed to your screen in real time.

Of course, this script doesn't do anything interesting (yet). Instead, think of this example as a **template**. Whenever you need to create a script to access your camera, **start with this template first**. You'll then be able to insert code blocks inside the `while` loop to handle performing face detection, object recognition, and *more!*

In the next chapter, you'll learn how to take this template and use the `VideoStream` class to build an automatic time lapse builder. The rest of the text will then continue to utilize our template, building on previous examples, and creating more advanced projects.

Take the time now to run the script, play with the code (including accessing both the RPi camera module and USB webcam), and ensure you fully understand how it works — **this template will be used many times throughout the rest of the text**.

## 5.5 Tips and suggestions

The following section provides my tips, suggestions, and best practices when debugging why you may be running into an error or having a problem accessing your camera on the Raspberry Pi.

### 5.5.1 Enable the RPi Camera via `raspi-config`

A common reason you may not be able to access your RPi camera module is forgetting to enable it via the `raspi-config` command. Take a second now to launch the program from a terminal:

---

```
$ sudo raspi-config
```

---

You'll then want to select `Interfacing Options`, followed by `Camera`. From there you can enable your RPi camera module (a reboot of your Raspberry Pi may be required).

### 5.5.2 Ensure Your RPi Camera Module is Correctly Plugged In

Does the following error message look familiar?

---

```
mmal: Cannot read camera info, keeping the defaults for OV5647
mmal: mmal_vc_component_create: failed to create component
    'vc.ril.camera' (1:ENOMEM)
mmal: mmal_component_create_core: could not create component
    'vc.ril.camera' (1)
mmal: Failed to create camera component
mmal: main: Failed to create camera component
mmal: Camera is not enabled in this build.
      Try running "sudo raspi-config" and ensure that "camera"
      has been enabled
```

---

If you are using a Raspberry Pi camera module and encounter the above error, then your camera module may not be properly connected to the Pi. Double-check that the ribbon is correctly seated and housed in the connector.

I've run into this problem myself a handful of times and found that sometimes the housing can be finicky — make sure the camera module is connected properly, otherwise the RPi will not be able to access it.

### 5.5.3 Run a Quick Sanity Test with `raspistill`

Before trying to run a Python script that accesses the RPi camera module, I first like to run a quick sanity check using the `raspistill` command:

---

```
$ raspistill -o test.jpg
$ ls
test.jpg
```

---

This command accesses our RPi camera module, snaps a photo, and saves it to disk. You should check the contents of `test.jpg` to ensure it looks correct. If so, your RPi camera module is working properly and you can try to access it via the `VideoStream` class.

### 5.5.4 Understanding `NoneType` Errors

A common error I see PylImageSearch readers encounter when they try to access a webcam or RPi camera module is the following:

---

```
$ python access_camera.py
Traceback (most recent call last):
  File "access_camera.py", line 17, in <module>
    (H, W) = frame.shape[:2]
AttributeError: 'NoneType' object has no attribute 'shape'
```

---

What does this `NoneType` error mean? And why does it seem to happen when you are performing an arbitrary operation on a frame, such as resizing it, converting it to grayscale, etc?

It's important to understand that OpenCV will *not* return an error message when either:

- i. A frame could not be read from a video stream
- ii. The path to an input image supplied to the `cv2.imread` function is invalid

Instead, OpenCV will return `None` (the Python equivalent of `null`).

If an image/frame is `None` (versus the NumPy array we expect), we cannot possibly apply any operations to the object, hence the error. The vast majority of the time this error is due to either:

- i. An invalid path to an input image/video

## ii. OpenCV being unable to access your video stream

If you are using a RPi camera module, make sure you have enabled it via `raspi-config` using Section 5.5.1 earlier in this chapter. If you are using a USB webcam, you may want to ensure it's properly plugged in and that you have installed the proper drivers.

Provided you have followed my install/configuration scripts you shouldn't run into any problems; however, keep in mind that not all USB webcams are plug-and-play compatible with OpenCV and the RPi — you should refer to the documentation for your webcam.

### 5.5.5 Confusing the Parameters to `VideoStream`

If you're still encountering issues accessing your USB webcam or RPi camera module, you should double-check your parameters to the `VideoStream` class.

**For a USB webcam**, your `VideoStream` instantiation should supply the `src` integer index of the webcam (typically 0):

---

```
1 vs = VideoStream(src=0).start()
```

---

The `src` is an integer representing the number of the webcam. The first webcam connected to your system has an index of 0, the second webcam an index of 1, and so on. If you only have a single USB webcam the index is almost always 0.

**If you are instead using a RPi camera module**, your `VideoStream` instantiation should look like this:

---

```
1 vs = VideoStream(usePiCamera=True).start()
```

---

Notice how the `usePiCamera` flag is set to `True`, indicating we are to use the RPi camera module. Additionally, since we are using the RPi camera module we *do not* need to supply a `src` index.

Double and triple-check your `VideoStream` class is instantiated properly depending on whether you are using a USB webcam or RPi camera module.

## 5.6 Summary

In this chapter we discussed two different types of cameras you can use on your Raspberry Pi:

- i. The Raspberry Pi camera module (a CSI camera that connects directly to the Pi board)
- ii. USB webcams which connect to the RPi via one of its USB ports

Exactly which camera is best for you is heavily dependent on your project. For pure speed and reduced CPU load, the RPi camera module may be a good choice as it will reduce the overhead of moving data from USB, across the system bus, to your Python script. But on the other hand, USB webcams give you more flexibility, including more options for camera quality, lens, auto-focus etc.

My personal recommendation is to try both when you are (1) first implementing your project and (2) then again when you are looking to optimize your scripts. Make sure you note which camera is:

- Giving you higher quality frames that are sufficient for completing your project goals
- Reducing your CPU load to acceptable levels

Use these empirical results to guide you when choosing a camera to deploy your computer vision project to the RPi.

Finally, we demonstrated how to use the `VideoStream` class to access *either* a RPi camera module or USB webcam. Switching between the two is as simple as changing a single argument to the class when instantiating.

The `VideoStream` class will save you a *ton* of time and effort when accessing frames from a camera on the RPi, so make sure you use it!

## Chapter 6

# Changing Camera Parameters

When it comes to developing successful computer vision and deep learning applications, your *code* and your *models* are only half the battle — lighting conditions and camera parameters are the other half. Writing computer vision code to accomplish a given task is *significantly easier* when you can control lighting conditions, enabling you to make assumptions regarding hardcoded parameters for edge detection, thresholding, and other standard image processing algorithms.

Of course, there are situations where lighting conditions cannot be controlled. This is *especially* true if you want to use your RPi in outdoor scenarios where one hour the sun could be shining, the next it could be a thunderstorm, and then three hours later, it's nighttime.

We'll be discussing working in low light conditions with the RPi in Chapter 12, but for now, let's review various USB and RPi camera module parameters you can tweak to make it easier to develop computer vision applications.

### 6.1 Chapter Learning Objectives

In this chapter we'll learn the following:

- i. Camera settings basics
- ii. How to change USB camera settings from GNU application
- iii. How to change USB camera settings from the command line with the Video4Linux driver
- iv. Write our own Python + OpenCV script to change USB camera parameters
- v. Write another Python + PiCamera script to alter PiCamera settings

With this capability, you will be able to customize the picture you're working with **before** it enters your computer vision pipeline. Sometimes this can make all the difference.

## 6.2 Changing Camera Parameters

In the first part of this chapter we'll work with `guvcview`, a GUI for changing settings on most USB webcams that interface with the Video4Linux drivers in Linux. We'll then work with OpenCV's and imutils' APIs to change cameras within a Python script. Similarly, we'll define a script that works with PiCamera settings. Both scripts will allow us to visualize changes on-the-fly as we change parameters.

### 6.2.1 Project Structure

This chapter's project structure is straightforward:

---

```
|--- change_usb_camera_parameters.py
|- change_picamera_parameters.py
```

---

We will walk through two scripts: one for changing USB camera parameters (which will work for most USB cameras), and one for changing PiCamera parameters.

But first, let's review GUI and commandline based methods for adjusting camera settings.

### 6.2.2 Method #1: Changing USB Camera Parameters from a GUI

In this section we'll discuss a tool called `guvcview`. The GTK UVC Viewer application (Figure 6.1) is GNU-licensed software which allows you to change camera parameters using a clunky, but workable GUI. The basic method of operation is as follows:

First, install the program onto your RPi via `apt`, and then run the application from the command line (with your USB webcam hooked up):

---

```
$ sudo apt-get install guvcview
$ guvcview
```

---

From there, go ahead and familiarize yourself with the GUI. As you go, you can adjust the settings until the webcam feed looks good. Next, close the program. Your settings will be preserved until you unplug your USB camera or restart your RPi. You can then fire up your Python computer vision script.

I like to use the program to help me dial in the settings, and then I apply methods #2 and #3 to recall the settings quickly. It is also possible to save settings files with `guvcview`, but it is less convenient — you will need to load the clunky GUI app to recall them.

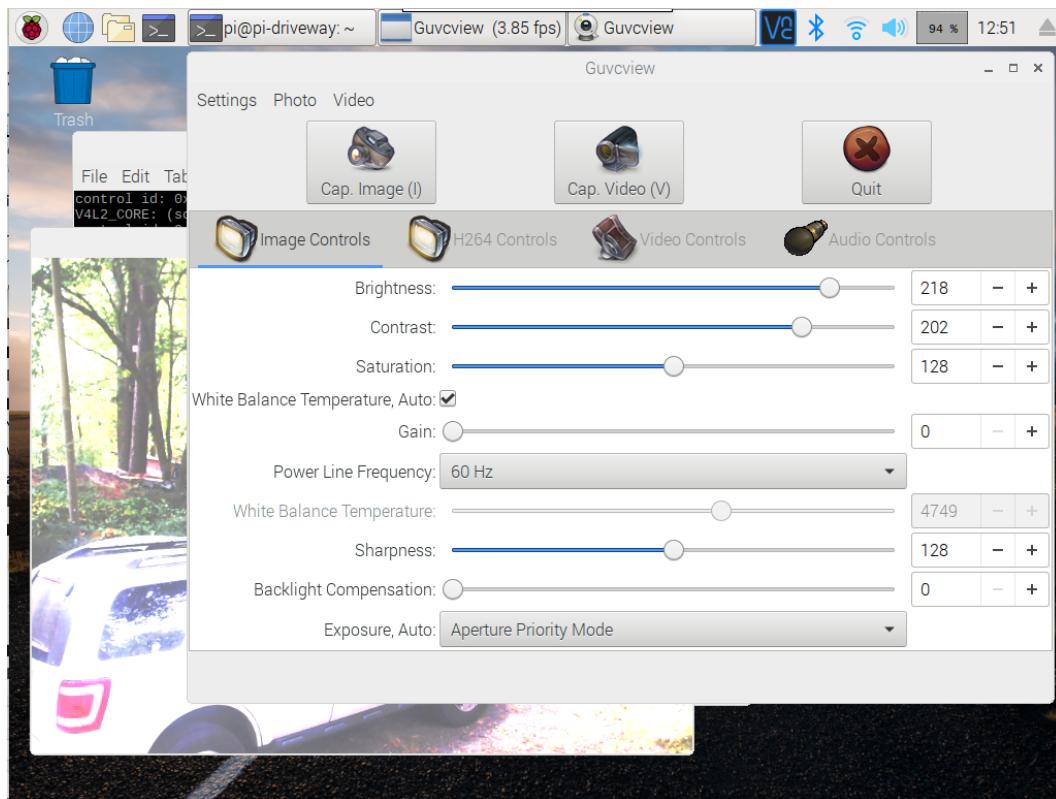


Figure 6.1: The guvcview application allows you to control camera parameters and save them using a convenient GUI.

### 6.2.3 Method #2: Changing USB Camera Parameters from the Command Line

Video4Linux (V4L) is a set of drivers for interfacing with cameras on Linux machines. OpenCV has interfaced well with V4L for years. Whenever you are working with a USB camera and a video stream in OpenCV (on a Linux box), OpenCV is talking to the V4L drivers in order to grab frames from your USB camera. This certainly includes Raspbian.

If you're working in Windows or macOS, OpenCV actually works with different drivers, so this method will not work on those operating systems. There are ways to port V4L functionality to macOS, though I cannot support readers who enter this territory.

Provided you are on Linux/Raspbian and you're using a compatible camera, you can query settings from the command line and even change most of them.

The tool that provides this functionality is called `v4l-ctl`.

Using my Logitech C920, let's query the camera its settable/adjustable parameters:

---

```
$ v4l2-ctl -l
brightness (int)      : min=0 max=255 step=1 default=-8193
```

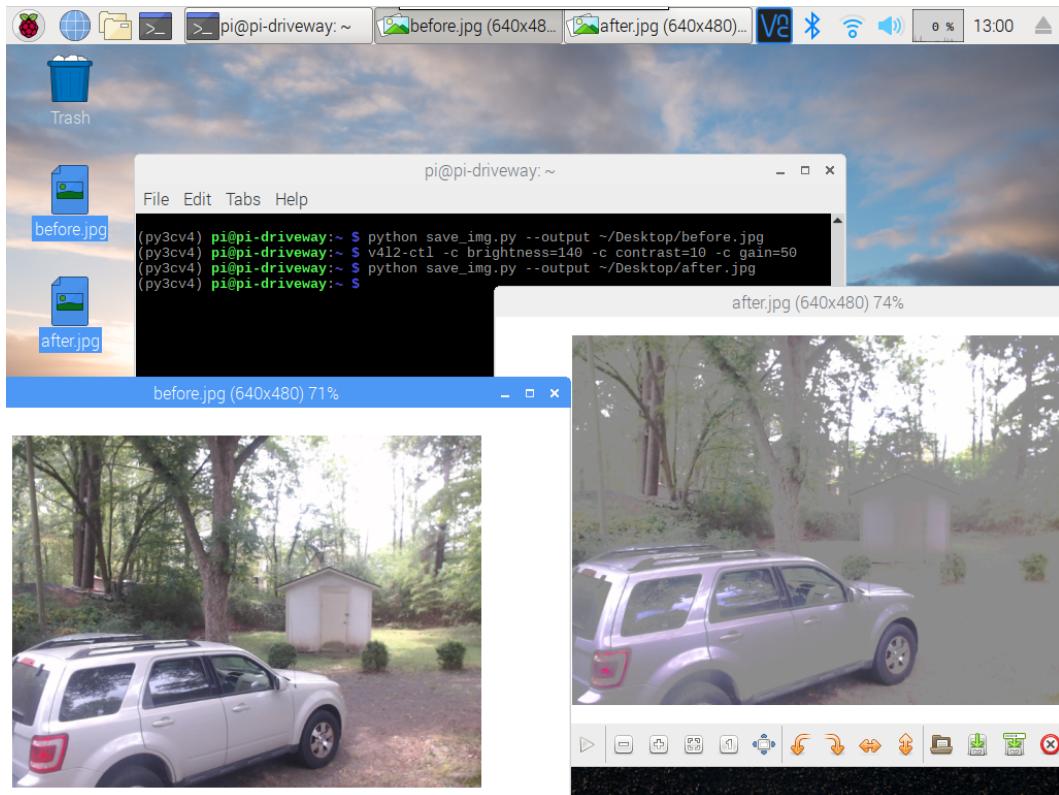


Figure 6.2: Changing USB camera settings with the Video4Linux driver command line tool.

```

        value=128
contrast (int)   : min=0 max=255 step=1 default=57343
                    value=128
saturation (int) : min=0 max=255 step=1 default=57343
                    value=128
white_balance_temperature_auto (bool)
gain (int)       : default=1 value=1
                    : min=0 max=255 step=1 default=57343
                    value=57
power_line_frequency (menu)
white_balance_temperature (int)
                    flags=inactive
sharpness (int)  : min=0 max=255 step=1 default=57343
                    value=128
backlight_compensation (int)
                    flags=inactive
exposure_auto (menu)
exposure_absolute (int)
                    flags=inactive
exposure_auto_priority (bool)
pan_absolute (int)
                    flags=inactive
tilt_absolute (int)
focus_absolute (int)
                    flags=inactive
focus_auto (bool)
                    flags=inactive

```

---

```
zoom_absolute (int)      : min=100 max=500 step=1 default=57343
                           value=100
```

---

With the list of parameters that can be controlled in hand, now we can go ahead and change one or many. In this example, we will look at the ranges from the output above and change brightness, contrast, and gain:

---

```
$ v4l2-ctl -c brightness=140 -c contrast=10 -c gain=50
```

---

Figure 6.2 shows the resulting image (before and after):

Of course, if you ever feel lost with the command, be sure to use the help flag:

---

```
$ v4l2-ctl -h
General/Common options:
--all           display all information available
-C, --get-ctrl=<ctrl>[,<ctrl>...]
                  get the value of the controls [VIDIOC_G_EXT_CTRLS]
-c, --set-ctrl=<ctrl>=<val>[,<ctrl>=<val>...]
                  set the value of the controls [VIDIOC_S_EXT_CTRLS]
-D, --info        show driver info [VIDIOC_QUERYCAP]
-d, --device=<dev> use device <dev> instead of /dev/video0
                  if <dev> starts with a digit, then /dev/video<dev> is used
-e, --out-device=<dev> use device <dev> for output streams instead of the
                  default device as set with --device
                  if <dev> starts with a digit, then /dev/video<dev> is used
-h, --help         display this help message
--help-all        all options
--help-io         input/output options
--help-misc       miscellaneous options
--help-overlay    overlay format options
--help-sdr         SDR format options
--help-selection  crop/selection options
--help-stds        standards and other video timings options
--help-streaming  streaming options
--help-tuner      tuner/modulator options
--help-vbi         VBI format options
--help-vidcap     video capture format options
--help-vidout     vidout output format options
--help-edid        edid handling options
-k, --concise     be more concise if possible.
-l, --list-ctrls   display all controls and their values [VIDIOC_QUERYCTRL]
-L, --list-ctrls-menus
                  display all controls and their menus [VIDIOC_QUERYMENU]
-r, --subset=<ctrl>[,<offset>,<size>]+
                  the subset of the N-dimensional array to get/set for control
                  <ctrl>, for every dimension an (<offset>, <size>) tuple is
                  given.
-w, --wrapper     use the libv4l2 wrapper library.
--list-devices   list all v4l devices
--log-status     log the board status in the kernel log [VIDIOC_LOG_STATUS]
--get-priority   query the current access priority [VIDIOC_G_PRIORITY]
```

---

```
--set-priority=<prio>
    set the new access priority [VIDIOC_S_PRIORITY]
    <prio> is 1 (background), 2 (interactive) or 3 (record)
--silent
    only set the result code, do not print any messages
--sleep=<secs>
    sleep <secs>, call QUERYCAP and close the file handle
--verbose
    turn on verbose ioctl status reporting
```

---

Again, with this method, the settings are preserved until the camera is unplugged or you restart your RPi.

#### 6.2.4 Method #3: Changing USB Camera Settings from OpenCV's API

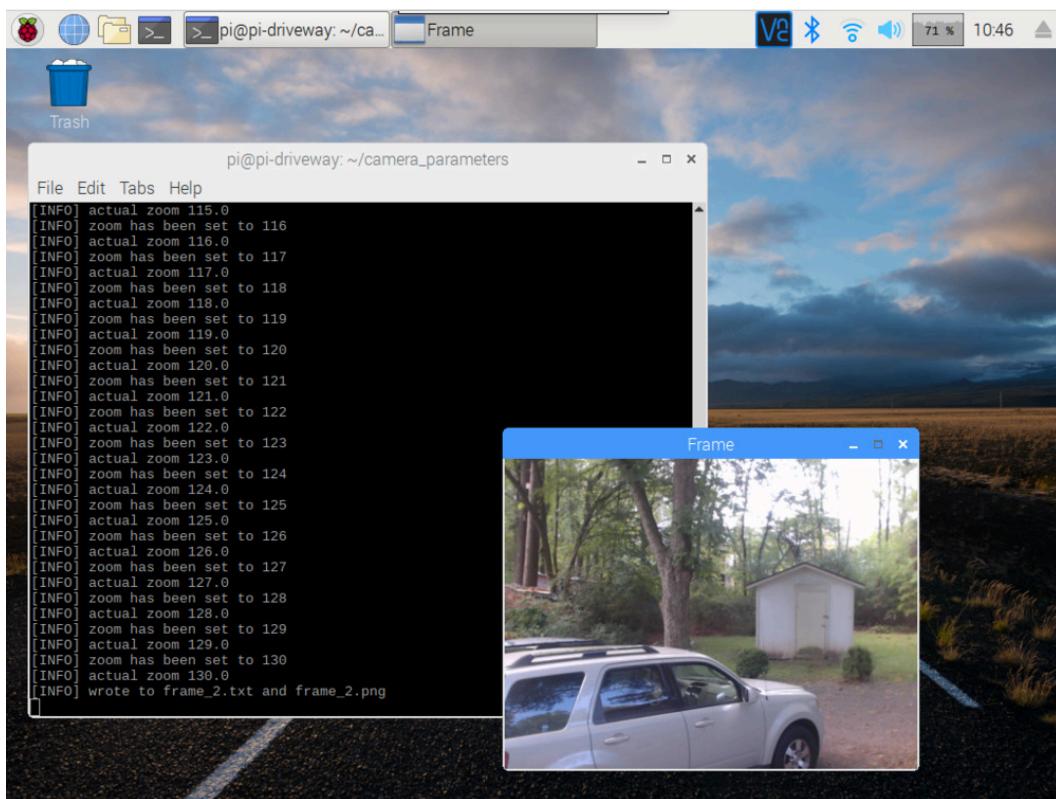


Figure 6.3: Changing USB camera settings with the OpenCV API and a GUI that accepts key presses.

Since we are often working with OpenCV, it is nice to be able to control the camera from our own scripts rather than the command line. Changing settings with OpenCV is easy. Here is the basic syntax for turning autofocus off:

---

```
1 # import necessary packages
2 import cv2
3
```

---

---

```
4 # start the webcam video stream and turn off the autofocus setting
5 stream = cv2.VideoCapture(0)
6 stream.set(cv2.CAP_PROP_AUTOFOCUS, 0)
```

---

Notice how the `stream.set` function call is used to set a particular property.

In Chapter 5, we discussed how to use the threaded `VideoStream` class from `imutils`. The syntax is nearly the same if you are using my `VideoStream` class:

---

```
1 # import necessary packages
2 from imutils.video import VideoStream
3
4 # start the webcam video stream and turn off the autofocus setting
5 vs = VideoStream(src=0).start()
6 vs.stream.set(cv2.CAP_PROP_AUTOFOCUS, 0)
```

---

Of course, there are many settings other than autofocus to tweak. Here are some that you may wish to work with:

- `cv2.CAP_PROP_CONTRAST`
- `cv2.CAP_PROP_BRIGHTNESS`
- `cv2.CAP_PROP_FOCUS`
- `cv2.CAP_PROP_ISO_SPEED`
- `cv2.CAP_PROP_AUTO_WB`

An exhaustive list of parameters you may wish to tune is available in the OpenCV documentation: <http://pyimg.co/14fbb>. Take note that different cameras will support different settings. There are also warnings in the OpenCV documentation that there could be breakdowns at point in this linkage:

*"Reading / writing properties involves many layers. Some unexpected results might happen along this chain. `VideoCapture` -> API Backend -> Operating System -> Device Driver -> Device Hardware. The returned value might be different from what is really used by the device or it could be encoded using device dependent rules (eg. steps or percentage). Effective behavior depends from device driver and API Backend"*

In my experience working with Logitech, Microsoft, and knock-off cameras, Logitech cameras are always the most reliable and feature-rich. I also like the Microsoft LifeCam series of cameras. I officially endorse the Logitech C920 or a similar grade Logitech camera.

Using the C920, I developed a script called `change_usb_camera_parameters.py` that accepts keyboard input to change settings on the fly. It is certainly not as fancy as `guvcview`, but it demonstrates a quick example of changing camera settings. Let's go ahead and implement it.

When you're ready, open `change_usb_camera_parameters.py` and let's review:

---

```

1 # import the necessary packages
2 from imutils.video import VideoStream
3 import time
4 import cv2
5
6 def toggle_autofocus(vs, autofocus=True):
7     # set the autofocus camera property on or off
8     vs.stream.set(cv2.CAP_PROP_AUTOFOCUS, 1 if autofocus else 0)
9     print("[INFO] autofocus has been set to {}".format(
10         "ON" if autofocus else "OFF"))
11
12     # read back the property to ensure it was set
13     actualAutofocus = vs.stream.get(cv2.CAP_PROP_AUTOFOCUS)
14     print("[INFO] actual autofocus {}".format(actualAutofocus))

```

---

Our packages are imported on **Lines 2-4**. We then begin by defining the `toggle_autofocus` function on **Line 6**. This function has two actions: (1) Setting the autofocus camera property, and (2) reading the value back so we can ensure it was set properly.

The function accepts two parameters:

- `vs`: The `VideoStream` object.
- `autofocus`: A boolean indicating whether the autofocus should be on or off.

The autofocus setting is changed via **Line 8** using the `set` method and by passing the setting flag along with the desired value.

To keep the script simple, no verification on the return value from action 2 is performed. You, as the reader, should add that functionality as appropriate. It is especially important to validate the return of this call if you run your script with unfamiliar cameras. In other words, you shouldn't wishfully change USB camera parameters. Rather you should set and validate; if validation fails, you may wish to throw an exception.

Let's define a similar method to set the auto white balance setting:

---

```

16 def toggle_auto_whitebalance(vs, autowb=True):
17     # set the auto whitebalance camera property on or off

```

---

---

```

18     vs.stream.set(cv2.CAP_PROP_AUTO_WB, 1 if autowb else 0)
19     print("[INFO] auto white balance has been set to {}".format(
20         "ON" if autowb else "OFF"))
21
22     # read back the property to ensure it was set
23     actualAutoWhitebalance = vs.stream.get(cv2.CAP_PROP_AUTO_WB)
24     print("[INFO] actual auto white balance {}".format(
25         actualAutoWhitebalance))

```

---

As you can see on **Lines 16-24**, `toggle_auto_whitebalance` follows the same format as the previous function to wrap setting the `cv2.CAP_PROP_AUTO_WB` value.

Similarly, the zoom can be set:

---

```

27 def set_zoom(vs, zoom=100):
28     # set the zoom camera property on or off
29     vs.stream.set(cv2.CAP_PROP_ZOOM, zoom)
30     print("[INFO] zoom has been set to {}".format(zoom))
31
32     # read back the property to ensure it was set
33     actualZoom = vs.stream.get(cv2.CAP_PROP_ZOOM)
34     print("[INFO] actual zoom {}".format(actualZoom))

```

---

I found that with the Logitech C920, a zoom value of 100 is default and values up to approximately 275 zoom in. Zooming out (less than 100) made no difference.

I believe on some cameras the zoom feature may be optical, and on others, it may be a digital/software based zoom. I haven't been able to confirm this, but if you know, please send me an email.

From there, we'll initialize our stream and settings:

---

```

36 # initialize the camera's video stream
37 vs = VideoStream(src=0).start()
38 time.sleep(2.0)
39
40 # initialize the camera parameter settings
41 autofocus = True
42 autowb = True
43 zoom = 100

```

---

**Line 35** initializes our stream. **Lines 41-43** then initialize the settings that we'll be controlling in this demo.

Let's begin our frame processing loop which also handles key presses and calling appropriate functions:

---

```

45 # loop over frames
46 while True:
47     # grab a frame
48     frame = vs.read()
49
50     # display the frame
51     cv2.imshow("Frame", frame)
52     key = cv2.waitKey(1)

```

---

The first part of our `while` loop simply grabs and displays our frame. We'll also capture key presses via **Line 51**.

Let's process the key presses now:

---

```

54     # handle *q* keypresses for "quit"
55     if key == ord("q"):
56         break
57
58     # handle *f* keypresses for "autofocus"
59     elif key == ord("f"):
60         # toggle autofocus and set the camera property
61         autofocus = not autofocus
62         toggle_autofocus(vs, autofocus)
63
64     # handle *w* keypresses for "auto white balance"
65     elif key == ord("w"):
66         # toggle auto white balance and set the camera property
67         autowb = not autowb
68         toggle_auto_whitebalance(vs, autowb)
69
70     # handle *i* keypresses for "zoom in"
71     elif key == ord("i"):
72         # increase zoom and set the camera property
73         zoom += 1
74         set_zoom(vs, zoom)
75
76     # handle *o* keypresses for "zoom out"
77     elif key == ord("o"):
78         # decrease zoom and set the camera property
79         zoom -= 1
80         set_zoom(vs, zoom)

```

---

**Lines 53-79** will call appropriate functions based on a keypress. The following keys are activated:

- **q**: Breaks and quits.
- **f**: Toggles autofocus.

- **w**: Toggles auto white balance.
- **i**: Zooms in.
- **o**: Zooms out.

As you can see, each `elif` block simply toggles or increments/decrements a value and then calls the respective function with the fresh value. There's really nothing to it!

Let's clean up and exit (when the `q` key is pressed):

---

```

82 # reset camera parameter settings
83 toggle_autofocus(vs, 1)
84 toggle_auto_whitebalance(vs, 1)
85 set_zoom(vs, 100)
86
87 # cleanup
88 vs.stop()
89 cv2.destroyAllWindows()

```

---

**Lines 82-84** reset our settings to default since the settings would be preserved otherwise. From there, we stop the stream and close GUI windows (**Lines 87 and 88**).

To run the script simply enter the following command (and be sure to inspect the output as you press the keys):

---

```

$ python change_usb_camera_parameters.py
[INFO] autofocus has been set to OFF
[INFO] actual autofocus 0.0
[INFO] autofocus has been set to ON
[INFO] actual autofocus 1.0
[INFO] auto white balance has been set to OFF
[INFO] actual auto white balance 0.0
[INFO] zoom has been set to 101
[INFO] actual zoom 101.0
[INFO] zoom has been set to 102
[INFO] actual zoom 102.0
[INFO] zoom has been set to 103
[INFO] actual zoom 103.0
[INFO] zoom has been set to 104
[INFO] actual zoom 104.0
[INFO] zoom has been set to 105
[INFO] actual zoom 105.0
[INFO] zoom has been set to 106
[INFO] actual zoom 106.0
[INFO] zoom has been set to 105
[INFO] actual zoom 105.0
[INFO] zoom has been set to 104

```

---

```
[INFO] actual zoom 104.0
[INFO] auto white balance has been set to ON
[INFO] actual auto white balance 1.0
[INFO] autofocus has been set to ON
[INFO] actual autofocus 1.0
[INFO] auto white balance has been set to ON
[INFO] actual auto white balance 1.0
[INFO] zoom has been set to 100
[INFO] actual zoom 100.0
```

---



**USB Logitech C920 Parameters**

Figure 6.4: An assortment of camera parameter settings changes. The *top-left* image shows the default settings.

Changing USB settings with the OpenCV API was quite straightforward. The zoom capability of the Logitech C920 camera is a really nice feature and might drive some future projects (Figure 6.4, *top row*). The autofocus and auto-whitebalance parameters seem to only make a difference when a target is moving towards/away from the camera, but it is hard to see a difference in Figure 6.4 (*bottom row*).

Be sure to refer to the OpenCV documentation (<http://pyimg.co/14fbb>) to learn more about what settings the API supports. And remember to keep in mind that there could be breakdowns between the hardware drivers, Video4Linux drivers, and OpenCV API, which may make changing parameters troublesome.

The screenshot shows the PiCamera ReadTheDocs website. At the top, there's a navigation bar with the title 'PiCamera' and 'release-1.13'. Below it is a search bar labeled 'Search docs'. The main content area has a sidebar on the left containing links to various sections: 1. Installation, 2. Getting Started, 3. Basic Recipes, 4. Advanced Recipes, 5. Frequently Asked Questions (FAQ), 6. Camera Hardware, 7. Development, 8. Deprecated Functionality, and 9. API - The PiCamera Class. Under '9. API - The PiCamera Class', there are further sub-links: 9.1. PiCamera, 9.2. PiVideoFrameType, 9.3. PiVideoFrame, 9.4. PiResolution, 9.5. PiFramerateRange, and 10. API - Streams. At the bottom of the sidebar, there are 'Read the Docs' and 'v: release-1.13' buttons. The main content area has a header '9. API - The PiCamera Class' and a paragraph explaining that the library contains numerous classes, with the primary one being `PiCamera`. It also includes a code snippet for importing `picamera` and the definition of the `PiCamera` class.

```
import picamera

class picamera.PiCamera(camera_num=0, stereo_mode='none',
    stereo_decimate=False, resolution=None, framerate=None, sensor_mode=0,
    led_pin=None, clock_mode='reset', framerate_range=None) [source]
```

Provides a pure Python interface to the Raspberry Pi's camera module. Upon construction, this class initializes the camera. The `camera_num` parameter (which defaults to 0) selects the camera module that the

Figure 6.5: There is a wealth of information about the PiCamera hardware and API on the PiCamera ReadTheDocs website (<http://pyimg.co/jq78m>). You should have a general knowledge of what the PiCamera is capable of and where to find the information on this site as you need it.

### 6.2.5 Method #4: Changing PiCamera Settings with the PiCamera API

The PiCamera is completely separate from both Video4Linux and Universal Video Controller (UVC), with its own self-named API that we can work with to change settings (<http://pyimg.co/jq78m>) [39].

The nice thing about changing settings on the PiCamera is that it is very much controlled hardware. There are few vendors and few models. Everyone is working with the same hardware and software, so there likely won't be any driver issues.

That said, be sure you are running at least `imutils==0.5.3` and `picamera==1.13`. You can check your versions via the following commands:

---

```
$ pip freeze | grep 'imutils'
imutils==0.5.3
$ pip freeze | grep 'picamera'
picamera==1.13
```

---

If you need to update your packages, run the following command:

---

```
$ pip install --upgrade imutils "picamera[array]"
```

---

It's important to run the right versions, or you'll encounter an `AttributeError` because some linkages are not made in the previous `imutils` API.

This time, let's dive right into an example script — the best way to learn is to roll up your sleeves and slide under the engine bay. Open up `change_picamera_settings.py` and insert the following code:

---

```
1 # import the necessary packages
2 from imutils.video import VideoStream
3 from itertools import cycle
4 from pprint import pprint
5 import time
6 import cv2
7
8 # set the video stream as a global variable
9 global vs
```

---

**Lines 2-6** import our required packages. Again, we'll use the `VideoStream` class to set camera parameters but in a slightly different way. The `cycle` generator will be used to circularly iterate through a list.

If you're not familiar with `pprint`, it is essentially a fancy version of the `print` function; we will use `pprint` to print the settings dictionary such that it is pleasing to the eye. Of course `cv2` will display our video frame, but that is all we will use it for.

Our `vs` video stream object is `global` in this script for convenience (**Line 9**). When we need to access it from within a function, we'll use the `global` keyword.

First, let's define a function to read our settings:

---

```
11 def get_picam_settings(output=False):
12     # access globals
13     global picamSettings
14     global vs
15
16     # initialize variable to hold current settings
17     currentPicamSettings = {}
18
19     # print status message if the output will be displayed
20     if output:
21         print("[INFO] reading settings...")
22
23     # grab picamera attributes from the object
24     for attr in picamSettings.keys():
```

---

```

25         currentPicamSettings[attr] = getattr(vs.camera, attr)
26
27     # print settings to the terminal if required
28     if output:
29         pprint(currentPicamSettings)
30
31     # update the global
32     picamSettings = currentPicamSettings
33
34     # return the settings to the calling function
35     return currentPicamSettings

```

---

The `get_picam_settings` function will read all settable PiCamera attributes (<http://pyimg.co/jq78m>) [39], which we've predefined manually on **Lines 101-126**. Per the docs, some settings are read-only and for our purposes in this chapter, we won't pay attention to those settings. We will revisit **Lines 101-126** soon.

The function accepts a boolean `output` which indicates whether we should print status messages. It is `False` by default because this method will be called often and we want to avoid cluttering our terminal output.

Notice that we access two globals in this function: `vs` and `picamSettings` (**Lines 13 and 14**). The purpose of making `picamSettings` global is so that we can store `picamSettings` without passing it around like a soccer ball — it makes for a cleaner script. The `vs` object follows suit because we'll destroy and recreate it each time we change settings.

**Line 17** defines a variable to hold our latest/current PiCamera settings.

A status message is printed, if required (**Lines 20 and 21**).

The heart of the function is in **Lines 24 and 25**. For every key in the global `picamSettings` dictionary, we'll grab the value of the corresponding attribute in the PiCamera object contained within `vs.camera`. **This is different than calling `get` to grab camera settings as we did in Method #3 for USB cameras.**

We then print the freshly-formed `currentPicamSettings` dictionary via `pprint` (**Lines 28 and 29**).

Our global `picamSettings` are updated and stored on **Line 32**. As you can imagine, the purpose of saving the settings is so that we can make small adjustments to them or even save them to disk for later use.

Let's now define a sister function to grab a single setting (rather than all of them as we have just demonstrated):

---

```

37 def get_single_picam_setting(setting):
38     currentPicamSettings = get_picam_settings()

```

---

---

```
39     return currentPicamSettings[setting]
```

---

**Lines 37-39** grab the value of a single setting. Effectively, all settings are grabbed in the process, but only a single setting value is reported. You could implement this to directly read an attribute if you wish, but it is plenty fast without making this modification.

Now we'll define a function to set all settings:

---

```
41 def _set_picam_setting(**kwargs):
42     # create a new video stream with the settings
43     global vs
44     vs.stop()
45     time.sleep(0.25)
46     vs = VideoStream(usePiCamera=True, **kwargs).start()
47     time.sleep(1.5)
48     print("[INFO] success")
```

---

The `set_picam_settings` function accepts and passes `kwargs` to the `VideoStream` (**Lines 41-48**). This helper function assists `set_picam_setting`. The `kwargs` can be one or multiple `PiCamera` keyword argument attributes. Please refer to the `PiCamera` docs (<http://pyimg.co/jq78m>) for more details on possible attributes and associated values.

**Remark.** *It is imperative that the `usePiCamera=True` flag is explicitly set so that the `PiVideoStream` class (abstracted) is used.*

Take note that this function requires at least 1.75 seconds of time to run. In the process, the `vs` object is destroyed and a new one is created. This is the best way I found to recreate the object.

Let's now head down to our `set_picam_setting` function:

---

```
50 def set_picam_setting(**kwargs):
51     # access globals
52     global picamSettings
53     global vs
54
55     # read the current settings
56     print("[INFO] reading settings...")
57     currentPicamSettings = get_picam_settings()
58
59     # print the past and new values
60     for (attr, value) in kwargs.items():
61         print("[INFO] changing {} from {} to {}".format(attr,
62             currentPicamSettings[attr], value))
63         currentPicamSettings[attr] = value
64
```

---

```

65      # initialize variable to hold the duplicate attributes to delete
66      # since we can't have duplicates in kwargs
67      attrsToDelete = []
68
69      # loop over current settings attributes
70      for attr in currentPicamSettings.keys():
71          # test for a value of None and if so, mark the attribute for
72          # deletion
73          if currentPicamSettings[attr] == None:
74              attrsToDelete.append(attr)
75
76      # delete all duplicate attributes that have been marked
77      for attr in attrsToDelete:
78          currentPicamSettings.pop(attr)
79
80      # reassign kwargs and set camera settings
81      kwargs = currentPicamSettings
82      _set_picam_setting(**kwargs)

```

---

This function performs the following actions:

- Reads current settings (**Line 57**).
- Prints past and new settings values (**Lines 60-63**).
- Deletes duplicate values since we can't have duplicate attributes (**Lines 67-78**).
- Updates the settings via the helper function (**Lines 81 and 82**).

Let's define our video stream, a few modes, and cycle generators:

---

```

84  # initialize the camera's video stream
85  vs = VideoStream(usePiCamera=True).start()
86  time.sleep(2.0)
87
88  # auto white balance and ISO modes
89  awbModes = ["off", "auto", "sunlight", "cloudy", "shade",
90             "tungsten", "fluorescent", "flash", "horizon"]
91  isoModes = [0, 100, 200, 320, 400, 500, 640, 800, 1600]
92
93  # initialize two cycle pools
94  isoModesPool = cycle(isoModes)
95  awbModesPool = cycle(awbModes)

```

---

Our PiCamera video stream is defined on **Line 85**. **Lines 89 and 91** define the possible automatic white balance and ISO modes. Corresponding `cycle` generators are made for each list of modes, and these generators will iterate around all modes in the list. All of our “settable” PiCamera parameters are defined next:

---

```

97 # the following dictionary consists of PiCamera attributes that can be
98 # *changed*; the list is not exhaustive because some settings can only
99 # be changed based on the values of others, so be sure to refer to
100 # the docs
101 picamSettings = {
102     "awb_mode": None,
103     "awb_gains": None,
104     "brightness": None,
105     "color_effects": None,
106     "contrast": None,
107     "drc_strength": None,
108     "exposure_compensation": None,
109     "exposure_mode": None,
110     "flash_mode": None,
111     "hflip": None,
112     "image_denoise": None,
113     "image_effect": None,
114     "image_effect_params": None,
115     "iso": None,
116     "meter_mode": None,
117     "rotation": None,
118     "saturation": None,
119     "sensor_mode": None,
120     "sharpness": None,
121     "shutter_speed": None,
122     "vflip": None,
123     "video_denoise": None,
124     "video_stabilization": None,
125     "zoom": None
126 }
```

---

As stated previously, the `picameraSettings` defined on **Lines 101-126** is an not exhaustive list. Attributes that can be read but not set are excluded. Furthermore, a handful of settable parameters are only activated when certain settings are made, and even those are excluded.

If you want to put your camera into a specific mode and tweak values, I encourage you to refer to the PiCamera docs (<http://pyimg.co/jq78m>) and update the `picamSettings` dictionary accordingly. It is also likely that the list will change with different versions of the PiCamera API as well as PiCamera hardware.

The rest of the script mimics our previous USB script. We will loop over frames and accept/handle key presses. Let's not waste any time:

---

```

128 # loop over frames
129 while True:
130     # grab a frame
131     frame = vs.read()
132
133     # display the frame
```

---

```

134     cv2.imshow("Frame", frame)
135     key = cv2.waitKey(1)
136
137     # handle *q* keypresses for "quit"
138     if key == ord("q"):
139         break

```

---

Our while loop begins by reading a new frame and displaying it (**Lines 129-135**).

The `q` key is handled when we desire to quit (`break` is called) on **Lines 138 and 139**.

Let's create actions for the remainder of the keys:

---

```

1      # handle *w* keypresses for "auto white balance"
2      elif key == ord("w"):
3          # read the white balance mode and change it
4          awbMode = get_single_picam_setting("awb_mode")
5          set_picam_setting(awb_mode=next(awbModesPool))
6
7      # handle *i* keypresses for "ISO"
8      elif key == ord("i"):
9          # read the ISO and increase it (looping back after 1600)
10         iso = get_single_picam_setting("iso")
11         set_picam_setting(iso=next(isoModesPool))
12
13     # handle *b* keypresses for "brightness"
14     elif key == ord("b"):
15         # read the brightness and increase it
16         brightness = get_single_picam_setting("brightness")
17         brightness += 1
18         set_picam_setting(brightness=brightness)
19
20     # handle *d* keypresses for "darken brightness"
21     elif key == ord("d"):
22         # read the brightness and decrease it
23         brightness = get_single_picam_setting("brightness")
24         brightness -= 1
25         set_picam_setting(brightness=brightness)
26
27     # handle *r* keypresses for "read settings"
28     elif key == ord("r"):
29         get_picam_settings(output=True)
30
31     # handle *c* keypresses for "custom settings"
32     elif key == ord("c"):
33         set_picam_setting(brightness=30, iso=800, awb_mode="cloudy",
34         vflip=True)

```

---

In addition to `q`, the following keys are associated with actions:

- `w`: Cycles through white balance modes.

- **i**: Cycles through ISO modes.
- **b**: Increments brightness.
- **d**: Decrements brightness (darkens).
- **r**: Reads and prints all settings.
- **c**: Demonstrates that multiple settings can be set with a single function call via multiple keyword arguments (`kwarg`s). You can use the format of **Lines 173 and 174** to customize your PiCamera photo settings "to the T".

Finally, we can perform a bit of cleanup:

---

```
176 # cleanup
177 vs.stop()
178 cv2.destroyAllWindows()
```

---

**Lines 177 and 178** stop our stream and close the GUI windows.

Let's put our script to work. Go ahead and enter the following command in your terminal:

---

```
$ python change_picamera_parameters.py
...
[INFO] changing awb_mode from auto to sunlight
[INFO] success
[INFO] reading settings...
{'awb_gains': (Fraction(225, 128), Fraction(351, 256)),
 'awb_mode': 'sunlight',
 'brightness': 50,
 'contrast': 0,
 'drc_strength': 'off',
 'exposure_compensation': 0,
 'exposure_mode': 'auto',
 'flash_mode': 'off',
 'hflip': False,
 'image_denoise': True,
 'image_effect': 'none',
 'iso': 0,
 'meter_mode': 'average',
 'rotation': 0,
 'saturation': 0,
 'sensor_mode': 0,
 'sharpness': 0,
 'shutter_speed': 0,
 'vflip': False,
 'video_denoise': True,
 'video_stabilization': False,
 'zoom': (0.0, 0.0, 1.0, 1.0)}
```

---

```
[INFO] wrote to frame_1.txt and frame_1.png
...
[INFO] changing brightness from 69 to 70
[INFO] success
[INFO] reading settings...
{'awb_gains': (Fraction(55, 32), Fraction(405, 256)),
 'awb_mode': 'auto',
 'brightness': 70,
 'contrast': 0,
 'drc_strength': 'off',
 'exposure_compensation': 0,
 'exposure_mode': 'auto',
 'flash_mode': 'off',
 'hflip': False,
 'image_denoise': True,
 'image_effect': 'none',
 'iso': 0,
 'meter_mode': 'average',
 'rotation': 0,
 'saturation': 0,
 'sensor_mode': 0,
 'sharpness': 0,
 'shutter_speed': 0,
 'vflip': False,
 'video_denoise': True,
 'video_stabilization': False,
 'zoom': (0.0, 0.0, 1.0, 1.0)}
[INFO] wrote to frame_5.txt and frame_5.png
...
```

---

As shown in Figure 6.6, I've used keypresses to change the "brightness"=70 (using the **b** key), "awb\_mode"="auto" (with the **w** key), and "iso"=0 (using the **i** key). By default, the brightness is set to 50. When it's set to 70, the image appears brighter and washed out.

I also implemented another key, **s**, for saving full resolution frame .png image files and settings .txt files to disk on command. I'll leave this as a five minute exercise for you, the reader, to implement as you experiment with camera parameters.

Figure 6.7 shows images taken with a PiCamera V2 during daylight but with different settings (using the **s** key to save the frames and keep track of the settings).

The impact of "awb\_mode" on the image are evident in Figure 6.7 (*top row*). On the *top-left*, the "awb\_mode" is "auto". In the *top-middle*, the "awb\_mode" is "sunlight". Then, in the *top-right*, the "awb\_mode" is "tungsten". The changes are best viewed using the color PDF version of this book.

A few levels of brightness from lower to higher are shown (Figure 6.7, *middle row*).

ISO does not affect results unless (1) it is dark outside to begin with, or (2) there is action/motion. Neither of these conditions have been met, so there is no discernible difference

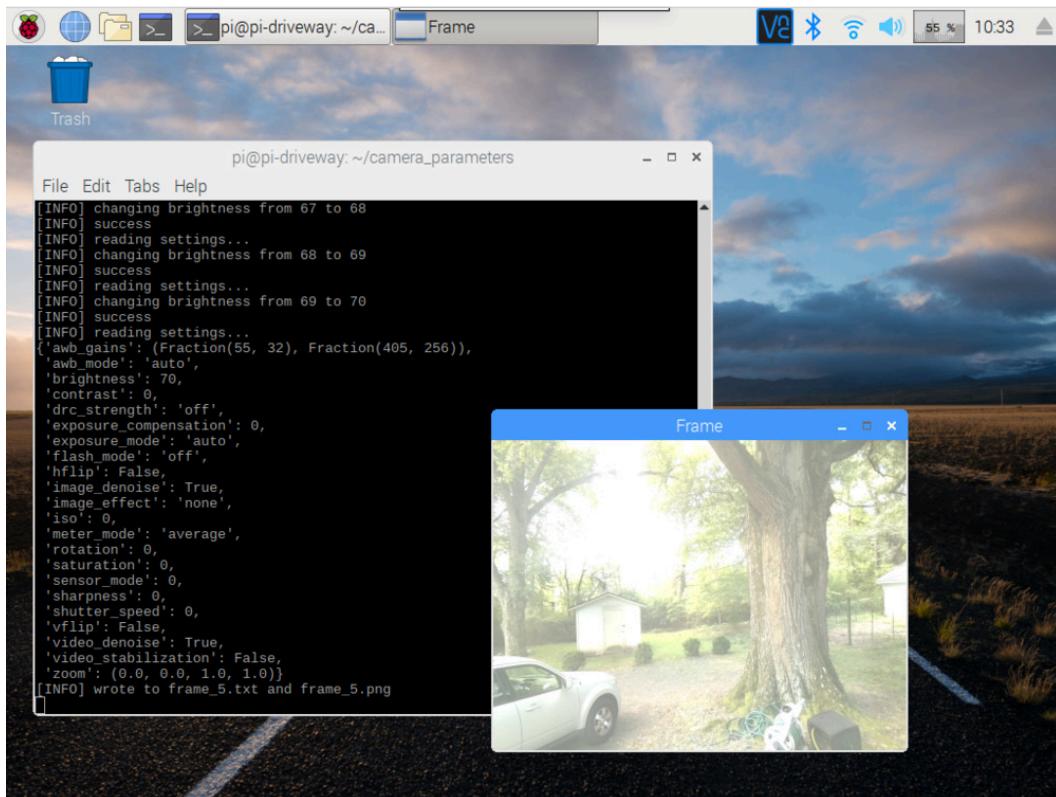


Figure 6.6: Changing PiCamera settings with the PiCamera API and keypresses.

changing the ISO settings (Figure 6.7, *bottom row*).

### 6.3 Tips and Suggestions

Whenever you develop a computer vision application, you should first consider your lighting conditions and camera parameters — are there any assumptions you can make about either? If so, how may these assumptions enable you to more easily write code for your project?

Always keep in mind that it's far easier to write OpenCV code if you're able to (1) control your lighting conditions and/or (2) control your camera parameters. Before diving too deep into a project, you should first explore camera parameters you can utilize to make your life easier.

For controlling your lighting conditions and working with low light, make sure you refer to Chapter 12.



### PiCamera V2 Parameters

Figure 6.7: A montage of images taken under different PiCamera settings. The default settings image is shown in the *top-left*.

## 6.4 Summary

In this chapter, we learned three ways to change USB camera settings and one way to change PiCamera settings:

- **Method #1:** GUI method — `guvvcview`
- **Method #2:** Command line option — `v4l2-ctl`
- **Method #3:** OpenCV's API combined with `imutils.video`
- **Method #4:** PiCamera's API combined with `imutils.video`

However, knowing how to change settings is just one piece of the puzzle. The next step is for you to start applying these settings to your projects. Take the time now to explore each of the parameters and note how they change the resulting frame.

Secondly, make sure you refer to the documentation for camera parameters with OpenCV (<http://pyimg.co/14fbb>) and the `picamera` Python module (<http://pyimg.co/jq78m>). An exhaustive review of each of these parameters is outside the scope of this text, so make sure you refer to the associated documentation for more details on which parameters are available to you.

Over time, with hands-on experience and tinkering, you will develop a “sixth sense” for these parameters, enabling you to look at a problem and near unconsciously know which camera parameters to adjust (if any).

For a more detailed review of camera parameters you may want to consider picking up a photography/camera book or enrolling in a photography class at your local community college.

## Chapter 7

# Building a Time Lapse Capture Camera

Our first foray into the world of computer vision with the Raspberry Pi is a practical project building a time lapse camera with Python and OpenCV.

A time lapse is a sequence of images captured from a camera (similar to a video). Unlike a video; however, there is a noticeable time delay between frames. These frames are captured at a frequency different from their display video frequency. Of course, the timing parameters are up to you so you can capture and display as you wish.

Typically, the result is a video (sequence of frames) which is far from "realtime" but tells the story of a scene in a very short duration.

**Remark.** *The Raspbian OS has a time lapse feature built into the `raspistill` command line tool, which I encourage you to check out (<http://pyimg.co/91l6h>). That said, in this chapter we're going to learn how to develop our own custom time lapse program using Python and OpenCV (i.e., without `raspistill`). Implementing our own time lapse system will help us become comfortable using the camera on our Raspberry Pi, paving the way for more advanced applications.*

### 7.1 Chapter Learning Objectives

In this chapter, we're going to learn and reinforce the following concepts:

- i. Accessing the Pi camera or USB camera
- ii. Handling command line arguments with `argparse`
- iii. Interrupt signals
- iv. Working with video streams

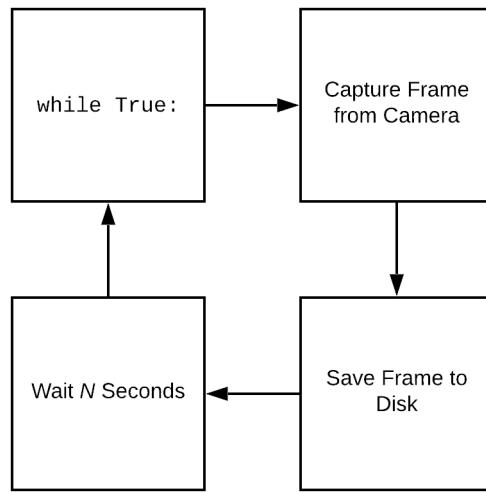


Figure 7.1: Our time lapse capture script consists of four steps. In the first step we start an infinite loop that will continue until we manually kill the script. We then capture a frame from our camera, save it to disk (so we can create the actual video time lapse sequence in a separate Python script), and then finally wait  $N$  seconds until the next frame is captured and logged.

- v. Writing images/frames to disk with OpenCV
- vi. Reading images/frames from disk with OpenCV
- vii. Writing to a video file with OpenCV

The last bullet is especially important. If you are new to Python, I encourage you to work through this chapter slowly. Try to understand what each line of code is doing. You'll need to use code fragments from this chapter as building blocks for future chapters.

If you find that you are struggling with the basics, definitely refer to my entry level book, *Practical Python and OpenCV* (<http://pyimg.co/ppao>) [11].

Let's get to work!

## 7.2 Capturing a Time Lapse Sequence

A time lapse sequence is a set of frames captured at even time intervals, the general algorithm of which is outlined in Figure 7.1. The time between frame captures from here forward will be known as the "delay".

The first task in building a time lapse animation is to capture frames from the camera sensor. We can accomplish this in less than 100 lines of Python.

As part of this script, we will allow for dynamic input from the command line, ensuring that we don't need to open the .py file each time to make an adjustment. The command line arguments are passed to the program at runtime to:

- Specify where photos will be stored.
- Set delay between captures.
- Determine whether we should display the frames in a GUI window.

When you're ready, open up a text editor or IDE. Name your file `capture_timelapse_frames.py`, and let's start coding:

---

```
1 # import the necessary packages
2 from imutils.video import VideoStream
3 from datetime import datetime
4 import argparse
5 import signal
6 import time
7 import cv2
8 import sys
9 import os
```

---

**Lines 2-9** import our necessary packages. We will use `VideoStream` to read frames from the camera along with `cv2` to annotate and save frames to disk. Both `datetime` and `time` will be used for timestamps and delays between captures. The `os` module will allow us to create our output directories.

Knowing how to work with the `signal` import is important because it will enable us to gracefully exit our script:

---

```
11 # function to handle keyboard interrupt
12 def signal_handler(sig, frame):
13     print("[INFO] You pressed `ctrl + c`! Your pictures are saved" \
14           " in the output directory you specified...")
15     sys.exit(0)
```

---

We need a way to gracefully exit our time lapse capturing system – we may interrupt the signal with `ctrl + c` (and we will), but let's do this carefully.

I've defined a `signal_handler` function on **Lines 12-15**. The concept is simple. We capture the interrupt `sig` at the current execution `frame` (which is *different* than a frame captured from a webcam). From there we can perform whatever operation we wish.

In this case, all we're doing is printing a message to the user and exiting the program. But you can actually work with existing variables in the stack in this function despite it appearing to have a different scope. However, if you have an image defined as `frame` it will conflict with the execution `frame`, which will take priority.

Soon, we'll set the signal trap (i.e. before our time lapse loop). We'll see other implementations of interrupt signal handling later in this book.

Now let's learn about the concept of `argparse` and command line arguments:

---

```

17 # construct the argument parse and parse the arguments
18 ap = argparse.ArgumentParser()
19 ap.add_argument("-o", "--output", required=True,
20                 help="Path to the output directory")
21 ap.add_argument("-d", "--delay", type=float, default=5.0,
22                 help="Delay in seconds between frames captured")
23 ap.add_argument("-dp", "--display", type=int, default=0,
24                 help="boolean used to indicate if frames should be displayed")
25 args = vars(ap.parse_args())

```

---

When you run your Python script from the command line, typically you would execute it like this:

```
$ python script_name.py
```

But what if we need to pass additional information to our script? Say you'd like to pass a file or directory path? Or maybe you need to pass a string, float, int, or boolean?

With **command line arguments** and a built-in Python module called `argparse`, we can accomplish passing additional information (arguments) to our script at runtime. The analogy is: **function:parameters :: script:arguments**.

In the case of Python we work with scripts, but you could also pass command line arguments to a binary executable (and I'll bet you have if you've worked with the command line on any OS before).

We'll start by defining our command line arguments with `argparse` (**Lines 18-25**). Later on, we'll see how to run the script from the terminal. If you need a more detailed explanation of command line arguments, please refer to my *Python, argparse, and command line arguments* (<http://pyimg.co/0e97u>) [20] blog post.

Our script requires one command line argument and accepts two optional arguments:

- `--output`: The path to the output directory.
- `--delay`: Delay in seconds between frames captured. By default, frames will be captured

at five-second intervals, but I encourage you to pass different values, especially for longer time lapse sequences.

- `--display`: Whether or not the frames should be displayed on your screen. By default, we will not display the frames as you may want to deploy your RPi without an HDMI display monitor.

Again, no changes to the code are necessary on **Lines 18-24**. Simply use the command line argument flags and values at runtime when you execute your script. We'll see how this is done shortly.

Let's go ahead and create our output directories and initialize our camera feed:

---

```

27 # initialize the output directory path and create the output
28 # directory
29 outputDir = os.path.join(args["output"],
30     datetime.now().strftime("%Y-%m-%d-%H%M"))
31 os.makedirs(outputDir)
32
33 # initialize the video stream and allow the camera sensor to warmup
34 print("[INFO] warming up camera...")
35 # vs = VideoStream(src=0).start()
36 vs = VideoStream(usePiCamera=True, resolution=(1920, 1280),
37     framerate=30).start()
38 time.sleep(2.0)

```

---

**Lines 29-31** create our output directory which will have the form:

`output/images/2019-06-10-1131/`

The value of the `--output` command line argument specifies where the images will be stored. In the example, we'll pass `output/images/`. Then, the datetime-stamped sub-directory will *automatically* be generated.

At this point, **Lines 35-38** should look familiar (if not, refer to Chapter 5). These lines initialize our camera stream with the specified resolution. If you want to use a USB camera, uncomment **Line 35** and comment **Lines 36 and 37**.

Let's set our frame count and signal trap. From there we'll begin our time lapse loop:

---

```

40 # set the frame count to zero
41 count = 0
42
43 # signal trap to handle keyboard interrupt
44 signal.signal(signal.SIGINT, signal_handler)
45 print("[INFO] Press `ctrl + c` to exit, or 'q' to quit if you have" \

```

---

```

46     " the display option on...")
47
48 # loop over frames from the video stream
49 while True:
50     # grab the next frame from the stream
51     frame = vs.read()

```

---

Our frame count is initialized to zero on **Line 41**. This value serves one purpose: counting and naming our output image files.

**Line 44** sets our signal trap, with the signal being `signal.SIGINT`. In general, the only interrupt will be a *ctrl + c* keyboard press, but of course there could be other interrupts as well (when the disk is nearly full, for example). Either the signal will be caught by the signal handler or Python's exception system will catch the problem.

**Remark.** We typically won't handle Python exceptions in the example code for this book. Handling exceptions is a best practice and we encourage you to do so. That said, it makes the code more cluttered, requires additional indentation, and adds to the overall page count/lines of code. Adding exception handling is therefore left as an exercise to you, the reader. To learn more about Python exceptions, including `try/catch` blocks, be sure to refer to this excellent resource from RealPython: <http://pyimg.co/0t8ge> [40].

We'll be looping over our frames beginning on **Line 49**, indefinitely, until any of the following occur:

- Power loss
- *Ctrl + c* or other interrupt
- Unhandled exception

Inside the loop, first we grab a `frame` (**Line 51**). Let's go ahead and process the `frame` before we loop back to read the next one:

---

```

53     # draw the timestamp on the frame
54     ts = datetime.now().strftime("%A %d %B %Y %I:%M:%S%p")
55     cv2.putText(frame, ts, (10, frame.shape[0] - 10),
56                 cv2.FONT_HERSHEY_SIMPLEX, 0.35, (0, 0, 255), 1)
57
58     # write the current frame to output directory
59     filename = "{}.jpg".format(str(count).zfill(16))
60     cv2.imwrite(os.path.join(outputDir, filename), frame)

```

---

**Lines 54-56** annotate the `frame` with the timestamp. You can comment these lines out if you don't want the timestamp shown on every frame.

**Lines 59 and 60** then write the image file to disk. Filenames will start at 0 and increase from there depending on the value of `count`. As you can see, writing an image to disk is quite straightforward.

Next, we'll display the frame on the screen if required:

---

```

62     # display the frame and detect keypresses if the flag is set
63     if args["display"]:
64         cv2.imshow("frame", frame)
65         key = cv2.waitKey(1) & 0xFF
66
67         # if the `q` key is pressed, break from the loop
68         if key == ord("q"):
69             break

```

---

If the command line argument `--display` is set, then the frame is shown on the screen. If "q" is pressed, we'll break from the loop.

Let's prepare for the next iteration of the loop and end our script:

---

```

71     # increment the count
72     count += 1
73
74     # sleep for specified number of seconds
75     time.sleep(args["delay"])
76
77     # close any open windows and release the video stream pointer
78     print("[INFO] cleaning up...")
79     if args["display"]:
80         cv2.destroyAllWindows()
81     vs.stop()

```

---

Prior to going back to the top of the loop, we have two important actions:

- i. Increment `count` so our next filename is in sequence.
- ii. Delay for the time specified in the `--delay` command line argument.

Upon completion of these two steps, the next frame will be grabbed and we'll repeat the process. Of course, if we break out of the loop we'll perform cleanup (**Lines 79-81**).

## 7.3 Running the Time Lapse Capture Script

For this step you can test the program at your desk. Eventually, you'll want to deploy the time lapse capture system to a fixed and fun location. Here are a handful of ideas:

- A nature scene — perhaps a scene containing clouds, shadows, and wildlife
- A flower in the process of blooming — control the scene with a light source and capture over 48 hours to see the flower open up
- Construction workers building a house or other building — you could capture once per hour or once per day for a longer project
- People moving in a crowded public place (such as a mall or stadium) — try to set up early in the morning before people arrive and let the system run until all the people leave

When your camera is set up, you have two options to run it for an extended period of time:

- i. Using the `screen` method so that you can safely disconnect your SSH session and the program will continue to run (if you've never used `screen` before I recommend reading this tutorial first: <http://pyimg.co/k9e77>).
- ii. Starting the script on reboot. See Chapter 8 to learn how to start the time lapse camera when the power cable is plugged in and the RPi boots.

In both cases, the command to start the program is:

---

```
$ python capture_timelapse_frames.py --output output/images --delay 2
[INFO] warming up camera...
[INFO] Press `ctrl + c` to exit, or 'q' to quit if you have the display
option on...
^C[INFO] You pressed `ctrl + c`! Your pictures are saved in the output
directory you specified...
```

---

Using the `--delay` argument, you could capture images every hour (3600 seconds) or even every day (86400 seconds). Here we arbitrarily set the `--delay` to two seconds.

To stop the program, you can press `ctrl + c`, use `pkill python` from a terminal, or pull the plug. This may involve re-establishing a `screen` session if you used that method for deployment.

In the next section, we'll package all the images together into a shareable video file.

## 7.4 Processing Time Lapse Images into a Video

In this section we'll write a single Python script to process a directory of time lapse photographs into a video file.

Recall that we have a sequence of images stored on disk. These images were captured with an  $N$ -second delay in between each capture. Now we're going to package all the images into a video file at a faster rate (usually 30 FPS or higher). The result will be an impressive animation of your scene.

When you are ready, go ahead and open a new file called `timelapse_process_images.py` and insert the following code:

---

```

1 # import the necessary packages
2 from imutils.video import VideoStream
3 from imutils import paths
4 import progressbar
5 import argparse
6 import cv2
7 import os
8
9 # function to get the frame number from the image path
10 def get_number(imagePath):
11     return int(imagePath.split(os.path.sep)[-1][-4])

```

---

**Lines 2-7** import our required packages. The `paths` module from my `imutils` package allows us to grab all image file paths in a specific directory.

The `get_number` function defined on **Lines 10 and 11** simply allows a convenient way of accessing the image number from its filename/path. For example, if the filepath is:

`output/images/2019-06-05-1737/0000000000000027.jpg`

Then the output of `get_number` will be an integer (27).

Let's handle our command line arguments for this script:

---

```

13 # construct the argument parse and parse the arguments
14 ap = argparse.ArgumentParser()
15 ap.add_argument("-i", "--input", required=True,
16                 help="Path to the input directory of image files")
17 ap.add_argument("-o", "--output", required=True,
18                 help="Path to the output video directory")
19 ap.add_argument("-f", "--fps", type=int, default=30,
20                 help="Frames per second of the output video")
21 args = vars(ap.parse_args())

```

---

Our script requires two command line arguments and can also handle one optional one:

- `--input`: The path to the input directory containing our timelapse images.
- `--output`: The output video file path.

- `--fps`: Frames per second of the output video. By default the video will play at 30 FPS.

Now that our command line arguments are parsed, let's initialize our video writer and grab our `imagePaths`:

---

```

23 # initialize the FourCC and video writer
24 fourcc = cv2.VideoWriter_fourcc(*'MJPG')
25 writer = None
26
27 # grab the paths to the images, and initialize output file name and
28 # output path
29 imagePaths = list(paths.list_images(args["input"]))
30 outputFile = "{}.avi".format(args["input"].split(os.path.sep)[2])
31 outputPath = os.path.join(args["output"], outputFile)
32 print("[INFO] building {}...".format(outputPath))

```

---

Our `VideoWriter` is initialized via **Lines 24 and 25**. Our codec is 'MJPG' for Motion JPEG. To learn more about writing videos to disk, be sure to read this blog post: <http://pyimg.co/i2z98> [41].

Using our `paths` module, **Line 29** then grabs all of our input image file paths. Our `outputPath` for the time lapse video is concatenated via **Lines 30 and 31**.

We're going to monitor progress in our terminal via a text-based progress bar called `pbar`. Let's go ahead and initialize it:

---

```

34 # initialize the progress bar
35 widgets = ["Building Video: ", progressbar.Percentage(), " ",
36             progressbar.Bar(), " ", progressbar.ETA()]
37 pbar = progressbar.ProgressBar(maxval=len(imagePaths),
38                               widgets=widgets).start()

```

---

Our progress bar is initialized to display the bar itself along with the ETA for completion on **Lines 35-38**.

Now we'll loop over all `imagePaths`:

---

```

40 # loop over all sorted image paths
41 for (i, imagePath) in enumerate(sorted(imagePaths, key=get_number)):
42     # load the image
43     image = cv2.imread(imagePath)
44
45     # initialize the video writer if needed
46     if writer is None:
47         (H, W) = image.shape[:2]

```

---

---

```

48         writer = cv2.VideoWriter(outputPath, fourcc, args["fps"],
49                         (W, H), True)
50
51     # write the image to output video
52     writer.write(image)
53     pbar.update(i)

```

---

Inside the loop beginning on **Line 41**, we:

- i. Sort frames by frame number via `get_number`.
- ii. Read the image from disk (**Line 43**).
- iii. Instantiate our video `writer` if required (**Lines 46-49**).
- iv. Write the `image` to disk and update our `pbar` (**Lines 52 and 53**).

Finally we clean up our progress bar and video `writer` on **Lines 57 and 58**:

---

```

55 # release the writer object
56 print("[INFO] cleaning up...")
57 pbar.finish()
58 writer.release()

```

---

Let's learn how to execute the script in our terminal.

## 7.5 Running our Time Lapse Processor

In this section, we'll run the `timelapse_process_images.py` script to build our time lapse video. For shorter sequences, you can run the script on your RPi. For longer time lapses, it may be faster to create the video using a laptop/desktop. You can transfer the files with SCP quite easily using these optional instructions.

To transfer all .jpg files with SCP command, simply execute the following commands *on your laptop* while replacing both the RPi IP address and paths with those which are relevant:

---

```

$ cd ~/RPi4CV/Hobbyist_Code/time_lapse
$ mkdir -p output/images/2019-06-10-1131
$ mkdir -p output/videos
$ scp pi@192.168.1.xxx:~/time_lapse/output/images/2019-06-10-1131/*.jpg \
    output/images/2019-06-10-1131/

```

---

From there, still on your laptop, navigate into the directory and execute the command to process the images:

---

```
$ cd ~/RPi4CV/Hobbyist_Code/time_lapse
$ python timelapse_process_images.py --input output/images/2019-06-10-1131/ \
--output output/videos --fps 30
[INFO] building output/videos/2019-06-10-1131.avi...
Building Video: 33% | #####| ETA: 0:00:19
```

---



Figure 7.2: **Top-left:** 5:30AM. The sun is just starting to come up. **Top-right:** 11:08AM. Sun is fully up and is shining. **Bottom-left:** 3:00PM. The sun is starting to shine directly into the camera, causing glare. **Bottom-right:** 8:45PM. Sun is setting, lights in skyscrapers and neighbor's houses are starting to turn on. A full video of the 24-hour time lapse can be found here: <http://pyimg.co/h0wn2>.

Now, using a media player such as VLC, QuickTime, or Windows Media Player, you can view the results of your hard work!

But until you have a chance to put together your own time lapse, I have included a sample output in Figure 7.2. This figure includes sample still frames from a 24-hour time lapse facing the Philadelphia, PA skyline from my office window. Using the code from this chapter you can create your own time lapses!

## 7.6 Tips and Suggestions

Here are a handful of suggestions for when you create your own time lapse videos:

- Your camera should be mounted so that it is stationary, ideally in a tripod.
- Keep lighting in mind. If your script will run overnight, do you need a light source to come on? In many cases, the answer is no. But in some cases you might need to turn on a light. If you want to turn on an IoT WiFi Smart Plug (or Smart Bulb/Smart Switch), be sure to refer to the **Hacker Bundle** of this text where we control a smart bulb programmatically.
- Before a deployment, always perform a one-minute time lapse sanity check to ensure your images are being recorded. This is especially important if you are “running the script on reboot” as demonstrated in Chapter 8.
- Delay values are in seconds. Be sure to calculate how many seconds you need between frames. For example, one hour is equal to 3600 seconds, so for the capturing process your delay parameter would be `--delay 3600` and the time lapse would start when the script starts.

### 7.6.1 Common Errors and Problems

A common error readers often experience when working with command line arguments for the first time is what we call a "USAGE error". Are you encountering the following error message?

---

```
$ python capture_timelapse_frames.py
usage: capture_timelapse_frames.py [-h] -o OUTPUT [-d DELAY] [-dp DISPLAY]
capture_timelapse_frames.py: error: the following arguments are required:
-o/--output
```

---

This isn’t an error message. It is simply a message indicating how to actually run (“use”, hence “usage”) the program from the command line.

Once you add your arguments properly, the script will execute. For example, the command line arguments for `capture_timelapse_frames.py` may look like this:

---

```
$ python capture_timelapse_frames.py --output output/images --delay 2
[INFO] warming up camera...
[INFO] Press `ctrl + c` to exit, or 'q' to quit if you have the display
option on...
^C[INFO] You pressed `ctrl + c`! Your pictures are saved in the output
directory you specified...
```

---

Be sure to refer to my full blog post on command line arguments and argparse (<http://pyimg.co/0e97u>) [20] if you are stuck.

## 7.7 Summary

In this chapter, you learned how to make a custom time lapse system using two Python scripts and our Raspberry Pi. More importantly, we learned key skills along the way.

We learned and reinforced the objectives listed at the beginning of the chapter, including how to access the RPi camera module/USB webcam, how to work with interrupt signals, how to read/write images to disk with OpenCV, and finally, how to read/write video to disk with OpenCV.

The knowledge learned by exploring these concepts will be directly applicable to future chapters. Take the time to understand them and burn some of the code syntax into your mind.

You can always refer to this chapter to “copy and paste” code fragments to build your own programs. In due time you’ll have them memorized and it will become second nature to develop scripts for the Raspberry Pi. You can do it!

In Chapter 8, we will learn how to start scripts on reboot. You’ll be able to simply plug in the power cable and your time lapse capture script (or a script of your choosing) will come alive.

## Chapter 8

# Automatically Starting Scripts on Reboot

Whether developing a computer vision project to monitor your backyard for furry friends, creating a wildlife camera to the wilderness to count animals, or deploying a production-level application with the Raspberry Pi, eventually you will need to know how to *automatically* start and execute your program on reboot. However, while simple in concept, it can actually be a bit of a pain to setup and debug.

In this chapter I will provide you with my best practices, tips, and suggestions to ensure your computer vision applications will run whenever your RPi is powered on or rebooted.

### 8.1 Chapter Learning Objectives

In this chapter, you will learn two methods to launch your scripts on reboot:

- i. **crontab:** For *background* processes.
- ii. **LXDE autostart:** For applications that run in the *foreground* of the RPi GUI/window manager.

You'll first implement a simple Python + OpenCV script to log frames to disk. You'll then use both crontab and LXDE autostart to launch the script on reboot.

By the end of this chapter you will be able to take your own computer vision applications and launch them on boot/reboot on the Raspberry Pi.

## 8.2 Starting Scripts on Reboot

In the first part of this chapter we'll review our project structure and then implement a simple Python script that will be executed on reboot. I'll then show you how to create a shell script that accesses your Python virtual environment, changes directory to your project, and then executes the Python script.

Once our shell script is ready, we'll review my two favorite methods that can be used to execute a program on boot – crontab (for background processes) and LXDE autostart (for foreground processes).

### 8.2.1 Understanding the Project Structure

Before we get started, let's briefly review our directory structure for the project:

---

```
|--- output/
|   |--- 0.jpg
|   |--- ...
|   |--- 999.jpg
|--- instructions.txt
|--- on_reboot.sh
|--- save_frames.py
```

---

Our Python script, `save_frames.py`, will access your camera (whether an RPi camera module or USB webcam), capture a total of 1,000 frames and save them to disk in the `output` directory.

The `on_reboot.sh` script encapsulates all necessary logic to access our Python virtual environment and execute our `save_frames.py` script. Since we need to call multiple commands, it's convenient (and easier to debug) to use a shell script that contains all of our necessary commands.

### 8.2.2 Implementing the Python Script

The Python script we're implementing in this chapter will simply log frames to disk (a total of 1,000 frames). After the frames have been saved, the script will automatically exit. This script is entirely *arbitrary* — you could replace it with any other Python script in this book or bring your own Python script.

That said, let's go ahead and implement it now. Open up `save_frames.py` and insert the following code.

---

```

1 # import the necessary packages
2 from imutils.video import VideoStream
3 import argparse
4 import time
5 import cv2
6 import os
7
8 # construct the argument parser and parse the arguments
9 ap = argparse.ArgumentParser()
10 ap.add_argument("-o", "--output", required=True,
11     help="Path to the output directory")
12 ap.add_argument("-d", "--display", type=int, default=0,
13     help="boolean used to indicate if frames should be displayed")
14 args = vars(ap.parse_args())

```

---

**Lines 1-6** import our Python packages while **Lines 8-14** parse our command line arguments. We need two command line arguments here:

- i. `--output`: The path to the output directory where frames will be saved to.
- ii. `--display`: A flag used to indicate whether or not we should be displaying frames to our screen using the `cv2.imshow` function.

Next, let's access our `VideoStream`:

---

```

16 # initialize the video stream and allow the camera sensor to warmup
17 print("[INFO] warming up camera...")
18 #vs = VideoStream(src=0).start()
19 vs = VideoStream(usePiCamera=True).start()
20 time.sleep(2.0)
21
22 # set the frame count to zero
23 count = 0

```

---

**Line 19** accesses our RPI camera module. If you are using a USB webcam you can uncomment **Line 18** and comment out **Line 19**. **Line 23** initializes `count`, used to count the total number of frames we have saved to disk thus far.

The following code block handles reading frames from the video stream, saving them to disk, and optionally displaying the frames to our screen:

---

```

25 # loop over frames from the video stream
26 while True:
27     # grab the next frame from the stream
28     frame = vs.read()

```

---

---

```

29
30     # write the current frame to output directory
31     p = os.path.sep.join([args["output"], "{}.jpg".format(count)])
32     cv2.imwrite(p, frame)
33
34     # check to see if the display flag is set
35     if args["display"] > 0:
36         # show the output frame
37         cv2.imshow("frame", frame)
38         key = cv2.waitKey(1) & 0xFF
39
40         # if the `q` key is pressed, break from the loop
41         if key == ord("q"):
42             break

```

---

**Line 28** reads the next `frame` from our stream while **Lines 31 and 32** (1) build the path to the output file path and (2) write the `frame` to disk.

**Line 35** checks to see if the `frame` should be displayed to our screen via `cv2.imshow`, and if so, we display it. If the `q` key is pressed we'll break from the `while` loop early.

The final step is to keep track of our `count`:

---

```

44     # increment the count
45     count += 1
46
47     # if the count reaches 1000, stop saving frames by breaking out
48     # of the loop
49     if count % 1000 == 0:
50         break
51
52     # cleanup the camera and close any open windows
53     cv2.destroyAllWindows()
54     vs.stop()

```

---

With each iteration of the `while` loop increments our `count` (**Line 45**). Once we've saved 1,000 frames to disk, we'll exit out of the loop (**Lines 49 and 50**).

### 8.2.3 Creating the Shell Script

Now that our `save_frames.py` script is implemented, we can move on our shell script. The `on_reboot.sh` script will be responsible for:

- i. Accessing our Python virtual environment
- ii. Changing directory to where our code/project lives

### iii. Executing the `save_frames.py` script

Take a look at the contents of `on_reboot.sh`:

---

```
1 #!/bin/bash
2
3 source `which virtualenvwrapper.sh`
4 workon py3cv3
5 cd /home/pi/pyimagesearch-rpi-code/script_on_reboot
6 python save_frames.py --output output --display 1
```

---

On **Line 3** we use the `source` command to load the contents of `virtualenvwrapper.sh`, the helper utility used to access our Python virtual environment via the `workon` command (**Line 4**).

Once inside our Python virtual environment, we can change directory to where our project lives on disk (**Line 5**) and then execute the Python script itself (**Line 6**).

After adding your commands to your shell script, exit the editor, **but don't forget to make it executable!** You can accomplish this task using the `+x` option to the `chmod` command:

---

```
$ chmod +x on_reboot.sh
```

---

In the next two sections you'll find out how to actually execute the `on_reboot.sh` script (and therefore, your Python script) on reboot.

#### 8.2.4 Method #1: Using crontab

The first method to running a script on reboot is to use `crontab`. The `cron` program is a daemon process that runs in the background on Unix machines. This process is responsible for executing scripts at certain times (i.e., every five minutes, once a day, at exactly this date and time, etc.) and when certain actions are performed on the system (such as a reboot).

We can edit `cron`'s list of jobs via the `crontab` command. However, there is a caveat to running jobs via this method.

The `cron` process runs in the **background** which means that if you want to launch a window via `cv2.imshow` you will *not* be able to. Again, `cron` is *only* for background processes (for foreground processes we'll be reviewing the LXDE autostart method in the next section).

All that said, let's go ahead and check out how we can use `crontab` to schedule our `on_reboot.sh` script to run on reboot.

To get started, you'll want to execute the following command:

```
$ sudo crontab -e
```

---

Notice here that we *require* sudo permissions to edit the cron schedule.

Scroll down to the bottom of the file and insert the following action:

```
@reboot /path/to/your/project/on_reboot.sh
```

---

The @reboot command tells cron to run our script whenever our system is rebooted. We then supply the path to our shell script after the @reboot command.

However, as I mentioned, cron can only run scripts in the background — for foreground running scripts, we'll need to use LXDE autostart.

### 8.2.5 Method #2: Updating the LXDE Autostart

LXDE is the name of the GUI/window manager provided with the Raspbian operating system. We can instruct LXDE to launch a Python script in the **foreground** of the window manager on reboot by editing its autostart configuration file.

There are two places where the autostart configuration file may live on your Raspbian installation.

- i. /home/pi/.config/lxsession/LXDE-pi/autostart
- ii. /etc/xdg/lxsession/LXDE-pi/autostart

If you edit the autostart file in your home directory then the global autostart file in /etc is ignored; however, I have run into situations where the autostart file in my home directory *is not automatically created*.

In those situations I instead recommend you edit the autostart file in your /etc directory. To edit this file, use the following command:

```
$ sudo nano /etc/xdg/lxsession/LXDE-pi/autostart
```

---

Again, sudo permissions are required to edit this file. Scroll down to the bottom of the file and add the following line:

```
@/path/to/your/project/on_reboot.sh
```

---

This command instructs LXDE to execute the path to our supplied script on reboot. Note that you can perform the same steps to edit the `autostart` file in your `/etc` directory (just make sure you update your file path to the `nano` editor).

### 8.3 An Example of Running a Script on Reboot

Now that we've implemented our example Python script, as well as edited either our `crontab` or `autostart` file, we can put our reboot method to the test.

Reboot your Pi, then navigate to the `output` directory of our project — you should see that directory start to populate with frames captured from our video stream (Figure 8.1, *left*).

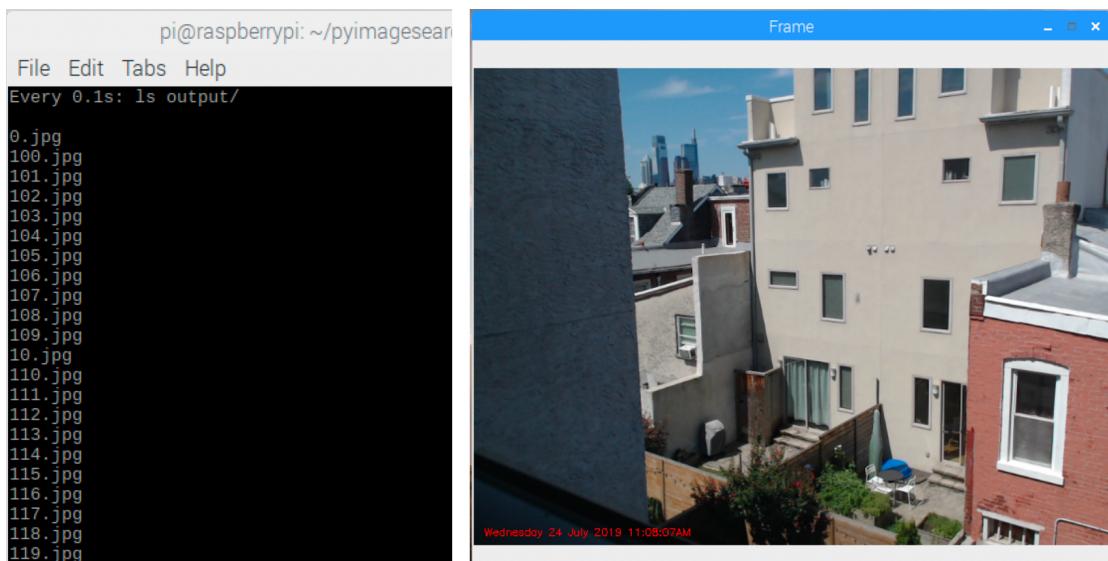


Figure 8.1: **Left:** Once your RPi has rebooted you can use the `watch` command to monitor the `output` directory for frames logged to disk. Here you can see that our `output` directory is being successfully populated. **Right:** An example frame from our video stream, thus demonstrating how we can run a script in the *foreground* (versus a cronjob that runs in the *background*) upon reboot.

If you've used the `autostart` method of LXDE you should see a foreground window of the video stream displayed as well (Figure 8.1, *right*).

### 8.4 Common Issues and Debugging Tips

By far, the most common issue I see when readers run into problems executing their scripts on reboot is **invalid file paths**. Typically, either the path to the shell script is incorrect *or* the path to your Python file is incorrect. **Double and triple-check your file paths, as they are the most likely cause of any issues.**

The second most common problem I see is forgetting to access your Python virtual environ-

ment (providing you are using one) before trying to exit your script. If you are not in the correct environment your imports will fail (and thus your script will not run).

**Before even attempting to run your shell script on reboot, execute it via the command line just as you would any other program.** Do you see any errors? If so, fix them — those errors will still be there if you try to run it on reboot. Your script is ready to be run on reboot when there aren't any errors.

**The point here is that you shouldn't go blindly editing shell scripts and cronjobs/autostart commands.** Take the time to execute the script via the terminal *first* — doing so will save you *a lot* of debugging time later.

Finally, I'll add that a great way to debug your scripts on reboot is to update the shell script to include `echo` commands (like a `print` statement) and shell redirects.

For example, going back to the `on_reboot.sh` script, perhaps I want to debug (1) my current working directory and (2) validate that my Python script finished executing. My `on_reboot.sh` script would look like this:

---

```
#!/bin/bash

echo "loading virtualenvwrapper.sh..." >>> /home/pi/test.log
source `which virtualenvwrapper.sh`
echo "accessing virtualenv..." >>> /home/pi/test.log
workon py3cv3
cd /home/pi/pyimagesearch-rpi-code/script_on_reboot
echo "running Python script..." >>> /home/pi/test.log
python save_frames.py --output output --display 1
echo "script exiting..." >>> /home/pi/test.log
```

---

Now, when I reboot my RPi, I can check the contents of `/home/pi/test.log` to validate that my script was successfully executing. Use these tips to your advantage — they will save you considerable time and frustration.

## 8.5 Summary

In this chapter you learned how to execute your Python and computer vision scripts on reboot using two different methods:

- i. `cron`, the Unix scheduling program
- ii. LXDE autostart, the autostart utility for the LXDE window manager

For applications meant to run in the **background** you should use `cron`. If the script you

are executing opens a window via `cv2.imshow` (or any other GUI library) then you should use LXDE autostart to run the program in the **foreground**.



## **Chapter 9**

# **Creating a Bird Feeder Monitor**

Capturing the perfect shot of wildlife is never easy.

All too often, I see something cool and fumble to grab the only camera I have available — a smartphone. By the time I orient the phone and start the camera application two things have happened:

First, the wildlife has left the scene. Second, I'm mad because my eyes were on my phone and not on the animal. I missed it with my own eyes and with those of my camera.

That got me thinking...

Can I automate wildlife photography and videography with a Raspberry Pi?

I most certainly can, and you can too.

### **9.1 Chapter Learning Objectives**

With a well aimed camera connected to a Raspberry Pi, you can capture wildlife in digital photos. By the end of this chapter, you'll understand the following concepts:

- i. Working with JSON configuration files
- ii. Background subtraction
- iii. Working with contours
- iv. Reading and writing video/image files
- v. Saving key event clips

## 9.2 Creating a Bird Feeder Monitor

We'll start this chapter by learning about background subtraction and why it is often used on resource-constrained devices such as the RPi. I'll direct you to where you can learn more about saving key event clips with my custom `KeyClipWriter` class (but I think you'll catch on by working through this chapter). From there, we'll review our project structure as well as our configuration file. And finally we'll develop and deploy our bird feeder monitor.

I recommend using a high quality USB webcam for this project. If you elect to use the `PiCamera`, ensure that you are grabbing high resolution images using the `resolution` parameter when you initialize your `VideoStream`. You may want to adjust camera parameters for the ideal image – refer to Chapter 6.

### 9.2.1 What is Background Subtraction?

Birds and many other animals move quickly. When monitoring animals, it is important to use a method that is both fast and reliable to determine motion events so that video recordings and stills can be made. Background subtraction is a concept and skill to learn when developing any type of monitoring application, like security, or in our case, wildlife photography.

Background subtraction can also be used in counting applications when object detectors and object trackers consume far more resources than a Raspberry Pi has at its disposal.

In order to successfully apply background subtraction, **we need to make the assumption that our background is mostly static and unchanging over consecutive frames of a video.**

Therefore, if we can model the background and monitor it for substantial changes, we can detect those changes and mark them as containing motion.

For this chapter we'll be using a “pre-baked” algorithm for background subtraction, implemented in OpenCV, enabling us to get our feet wet and quickly build our bird feeder application. In Chapter 14 we'll implement a background subtraction algorithm by hand, giving you more hands-on experience with the algorithm (and enabling you to better modify it for your specific project needs). However, for the time being let's start with a pre-implemented background subtractor – we can add in bells and whistles later.

I'll wrap up this section by saying that background subtraction algorithms are *not* perfect.

Shadows and lighting (caused by a fast moving cloud or a falling tree branch, for example), can “confuse” background subtraction, as the algorithm has no semantic understanding of the image/frame (i.e., the algorithm doesn't know if the background subtraction is caused by an event we don't care about, such as a falling tree branch, or by an event we do care about, such

as a bird or animal entering the scene).

In the *Hacker Bundle* of this text we use a combination of both deep learning-based object detection and dedicated object tracking algorithms to obtain better accuracy (but at the cost of more CPU cycles). Be sure to refer to the *Hacker Bundle* if you are interested in utilizing these more accurate methods, but for the time being, let's get our feet wet by building a simple bird feeder monitor via background subtraction.

### 9.2.2 Saving Key Event Clips

When saving motion videos, how can you ensure that you capture that *very split second* where the motion starts?

**The answer is by buffering frames.** The concept is simple — keep the camera rolling at all times and buffer frames so that there is padding on the front and the back of a motion event, that way, once motion is detected you can write any frames in the buffer to disk and then continue writing frames until no further motion is detected.

We'll be using my implementation of key event writing from a 2016 PyImageSearch tutorial titled *Saving key event video clips with OpenCV* (<http://pyimg.co/hvskf>) [42]. Reviewing the entire implementation is outside the scope of this book, so if you're interested in the algorithm, please refer to the blog post. Otherwise, **what you need to know for our purposes is that the algorithm maintains a buffer of frames and then dumps the frames to disk if motion is detected, ensuring you have the frames that not only include the motion, but also the frames leading up to the motion.**

### 9.2.3 Project Structure

Let's inspect our project structure:

---

```
|-- config
|   |-- gmg.json
|   |-- mog.json
|-- output_gmg
|   |-- 20181014-143423.jpg
|   |-- ...
|-- output_mog
|   |-- 20181014-143511.jpg
|   |-- ...
|-- pyimagesearch
|   |-- utils
|   |   |-- __init__.py
|   |   |-- conf.py
|   |-- __init__.py
```

---

```
|   |-- keyclipwriter.py
|-- bird_mon.py
|-- birds_10min.mp4
```

---

Our configuration lies in `config.json` as opposed to passing many command line arguments. The `Conf` class inside `conf.py` will parse the JSON configuration and allow us to access the settings from within our driver script.

Inside the `pyimagesearch` module is the `Conf` class introduced in Chapter 11. We also have the `KeyClipWriter` class inside `keyclipwriter.py`. Refer to Section 9.2.2 for more information on saving buffered event clips.

Our driver script for monitoring birds is `bird_mon.py`. This script is designed to work with a live webcam feed. For testing/development purposes you may use David McDuffee's `birds_10min.mp4` video, which he recorded near his residence.

#### 9.2.4 Our Configuration Files

In many of my projects I like to include a configuration file as opposed to having a long list of command line arguments. A configuration file knocks out a few birds with one stone.

Configuration files eliminate having to type many command line arguments. They allow for saving key constants, ensuring that we don't have to figure them out again after months away from the code. Different runtime configurations can be saved and recalled via separate config files.

There are many options for implementing configuration files with Python, each with their own advantages. XML, YAML, JSON, and .py files come to mind. I've used them all, but my two favorites are JSON and .py.

JSON is great — its syntax is the same as a Python dictionary.

The use of .py files is also beneficial when you need to perform calculations or insert other Python code directly in your config. This approach works great for joining file paths and ensuring your code is “OS agnostic”.

Going forward in this book, most chapters will use JSON configuration files.

Let's review the `Conf` class inside of `conf.py`:

---

```
1 # import the necessary packages
2 from json_minify import json_minify
3 import json
4
5 class Conf:
```

---

```

6     def __init__(self, confPath):
7         # load and store the configuration and update the object's
8         # dictionary
9         conf = json.loads(json_minify(open(confPath).read()))
10        self.__dict__.update(conf)
11
12    def __getitem__(self, k):
13        # return the value associated with the supplied key
14        return self.__dict__.get(k, None)

```

---

**Line 2 and 3** import both `json` and `json_minify`. We'll use "minify" to strip out C-like comments from our JSON files. Technically comments don't follow the JSON standard which is why we need to strip them out before letting `json` parse the file.

Our `Conf` class is defined on **Line 5** followed by the constructor on **Line 6**. The constructor accepts the `confPath` — the path to the configuration file.

From there, **Line 9** (1) opens the JSON file, (2) strips out comments, and (3) parses the JSON. **Line 10** simply loads the JSON data into a dictionary.

Due to how the `__getitem__` function is defined on **Lines 12-14**, we'll be able to access our JSON data just like we would a regular Python dictionary. You'll see how this works in our `mail_detector.py` driver script.

Now that we've defined the `Conf` class, we'll **review our configuration for this project** contained a JSON file.

As shown in the project structure, two example configurations are provided. Let's review just one of them with the major difference being that the MOG background subtraction method is used as opposed to GMG. Go ahead and open `mog.json`:

---

```

1  {
2      // indicate whether the picamera should be used
3      "picamera": false,
4
5      // background subtractor
6      // valid values are [CNT GMG MOG GSOC LSBP]
7      "bg_sub": "MOG",
8
9      // MxN-size and number of iterations of erode kernel
10     "erode": {"kernel": [3, 3], "iterations": 1},
11
12     // MxN-size and number of iterations of dilate kernel
13     "dilate": {"kernel": [5, 5], "iterations": 2},
14
15     // minimum pixels required for a valid foreground (motion) blob
16     "min_radius": 80,

```

---

You may toggle between USB webcam and PiCamera mode with the "picamera" variable on **Line 3**.

The background subtraction method is a string contained in "bg\_sub" on **Line 7**. The comment on **Line 6** shows the valid options. I encourage you to experiment with each of them, noting how different background subtraction algorithms can produce different results.

Erosion and dilation kernels for morphological operations are held on **Lines 10 and 13**. The dimensions as well as the number of iterations are both able to be set directly in the config. Different background subtraction methods may require adjustments to the kernels.

**Line 16** contains the minimum radius for a valid motion detection blob.

The remaining lines contain configuration settings related to storage and display of our clips and photos:

---

```

18 // key clips before/after buffer and codec
19 "keyclipwriter_buffersize": 50,
20 "codec": "MJPG",
21
22 // should snapshots be written, and if so, how many frames should
23 // there be between snapshots?
24 "write_snaps": true,
25 "frames_between_snaps": 30,
26
27 // should boxes and circles be annotated for display?
28 "annotate": true,
29
30 // should the video be shown on the screen?
31 "display": true,
32
33 // path to which output files will be written
34 "output_path": "output_mog",
35
36 // frames per second for output video
37 "fps": 20
38 }
```

---

Our key clip writer buffers frames. You can set the buffer size on **Line 19**. I've found that OpenCV writes videos to disk best with the MJPG "codec" (**Line 20**); however, if you have a different codec you want to use, you change it here.

Still snapshots of birds can be written to disk if the "write\_snaps" boolean is set to `true` (**Line 24**). We don't want to save every snapshot, so the "frames\_between\_snaps" is set to 30 (**Line 25**). You should customize this value as you keep in mind the FPS of your camera and the overall pipeline.

If you'd like rectangles and circles to be drawn around the birds, you can set the "annotate"

boolean on **Line 28**. For testing/debugging purposes, you should turn the "display" option on (**Line 31**).

Videos and photos will be output to the path contained in "output\_path" (**Line 34**). The output video's framerate is configurable via **Line 37**.

### 9.2.5 Developing Our Bird Monitor Script

With the concepts of background subtraction and key clip writing behind us, now we're ready to develop our bird monitoring driver script. The driver script will manage our video stream, background subtractor, and key clip writer all with the goal of catching our flying friends while they are in their feeding frenzy.

Open a file called `bird_mon.py` and insert the following code:

---

```

1 # import the necessary packages
2 from pyimagesearch.keyclipwriter import KeyClipWriter
3 from pyimagesearch.utils import Conf
4 from imutils.video import VideoStream
5 import numpy as np
6 import argparse
7 import datetime
8 import imutils
9 import signal
10 import time
11 import sys
12 import cv2
13 import os

```

---

**Lines 2-13** import necessary packages including our `KeyClipWriter`, `Conf`, and `VideoStream` classes.

From there, we'll handle "ctrl + c" interrupt signals:

---

```

15 def signal_handler(sig, frame):
16     # show message to user
17     print("\n[INFO] You pressed `ctrl + c`!")
18     print("[INFO] Your files are saved in the `{}` directory".format(
19         conf["output_path"]))
20
21     # check if we're recording and wrap up
22     if kcw.recording:
23         kcw.finish()
24
25     # gracefully exit
26     sys.exit(0)

```

---

Our `signal_handler` is very straightforward. It simply prints a message letting you know where your stills and videos are saved (**Lines 18 and 19**) as well as stops any key clips that are currently being recorded (**Lines 22 and 23**). From there, the program will exit (**Line 26**). Remember, the signal handler can access variables in the current execution frame, so as long as a variable/object exists, you can read/mutate it.

Moving on, we'll parse command line arguments and load our configuration:

---

```

28 # construct the argument parser and parse the arguments
29 ap = argparse.ArgumentParser()
30 ap.add_argument("-c", "--conf", required=True,
31     help="path to the JSON configuration file")
32 ap.add_argument("-v", "--video", type=str,
33     help="path to optional input video file")
34 args = vars(ap.parse_args())
35
36 # load our configuration settings
37 conf = Conf(args["conf"])

```

---

Our script can handle two command line arguments via `argparse`: (1) `--conf`, the path to our JSON configuration file, and (2) `--video`, the path to a video file for testing. If the second argument is not provided via the terminal at runtime, your camera will be used by default.

Let's go ahead and initialize our video stream now:

---

```

39 # check if we are using a camera and start the stream
40 if not args.get("video", False):
41     print("[INFO] starting video stream...")
42     vs = VideoStream(usePiCamera=conf["picamera"]).start()
43     time.sleep(3.0)
44
45 # otherwise, start a video file stream
46 else:
47     print("[INFO] opening video file `{}`.".format(args["video"]))
48     vs = cv2.VideoCapture(args["video"])

```

---

**Lines 40-43** kick off a video stream using your camera (either USB or PiCamera). Otherwise, if the `--video` command line argument exists, then a video file will be used as our video stream.

Let's initialize our background subtraction method:

---

```

50 # OpenCV background subtractors
51 OPENCV_BG_SUBTRACTORS = {
52     "CNT": cv2.bgsegm.createBackgroundSubtractorCNT,

```

```

53     "GMG": cv2.bgsegm.createBackgroundSubtractorGMG,
54     "MOG": cv2.bgsegm.createBackgroundSubtractorMOG,
55     "GSOC": cv2.bgsegm.createBackgroundSubtractorGSOC,
56     "LSBP": cv2.bgsegm.createBackgroundSubtractorLSBP
57 }
58
59 # create our background subtractor
60 fgbg = OPENCV_BG_SUBTRACTORS[conf["bg_sub"]]()
61
62 # create erosion and dilation kernels
63 eKernel = np.ones(tuple(conf["erode"]["kernel"]), "uint8")
64 dKernel = np.ones(tuple(conf["dilate"]["kernel"]), "uint8")

```

---

The OPENCV\_BG\_SUBTRACTORS dictionary (**Lines 51-57**) combined with the instantiation on **Line 60** sets up our foreground-background subtraction method.

We'll need erosion and dilation kernels to remove and cleanup our background subtraction mask. **Lines 63 and 64** set up the kernels using the parameters in our config file.

The last setup we need to perform is initializing our KeyClipWriter object:

```

66 # initialize key clip writer, the consecutive number of frames without
67 # motion and frames since the last snapshot was written
68 kcw = KeyClipWriter(bufSize=conf["keyclipwriter_buffersize"])
69 framesWithoutMotion = 0
70 framesSinceSnap = 0
71
72 # begin capturing "ctrl + c" signals
73 signal.signal(signal.SIGINT, signal_handler)
74 images = " and images..." if conf["write_snaps"] else "..."
75 print("[INFO] detecting motion and storing videos{}".format(images))

```

---

**Line 68** instantiates the KeyClipWriter as kcw. The buffer is pulled from our config and passed as the bufSize parameter.

Two frame counter variables are then initialized to zero. The first one, framesWithoutMotion (**Line 69**) is simply a counter for the number of frames that have passed since our background subtraction and contour processing last detected motion. The framesSinceSnap simply holds a count of the frames since the last still photo of a bird has been stored to disk.

**Lines 73-75** initialize our signal\_handler and print a status message indicating that we're starting to do some real work.

Now we'll begin looping over frames from our stream:

```

77 # loop through the frames
78 while True:

```

---

```

79     # grab a frame from the video stream
80     fullFrame = vs.read()
81
82     # if no frame was read, the stream has ended
83     if fullFrame is None:
84         break
85
86     # handle the frame whether the frame was read from a VideoCapture
87     # or VideoStream
88     fullFrame = fullFrame[1] if args.get("video", False) \
89     else fullFrame
90
91     # increment number of frames since last snapshot was written
92     framesSinceSnap += 1
93
94     # resize the frame apply the background subtractor to generate
95     # motion mask
96     frame = imutils.resize(fullFrame, width=500)
97     mask = fgbg.apply(frame)

```

---

A frame is grabbed and checked for validity (**Lines 80-84**). This frame is called `fullFrame` because it is the original, full-resolution frame. **Lines 88 and 89** simply handle whether the frame came from a camera or file stream.

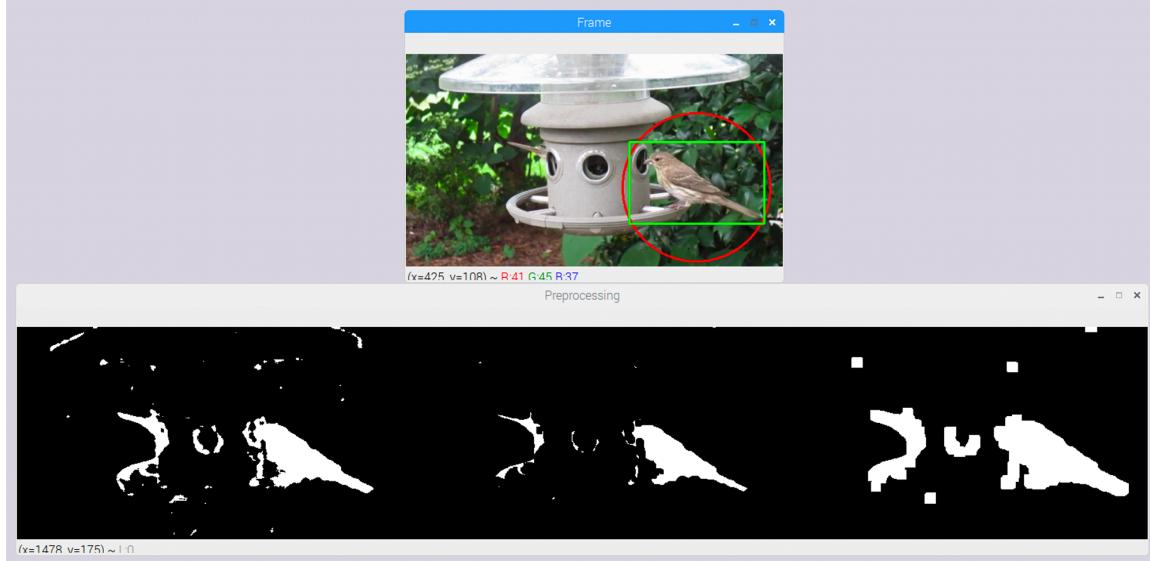


Figure 9.1: **Top:** Our output frame after motion detection, preprocessing, and annotation. **Bottom-left:** The raw mask obtained from our motion detector/background subtractor. **Bottom-middle:** Applying erosions to remove noise. **Bottom-right:** Using dilations to "regrow" mask regions after removing noise.

At this point, we'll increment our `framesSinceSnap` (**Line 92**). Once this value is greater than or equal to 30 (or another value specified in the config), it will be possible to write a photo to disk.

**Lines 96 and 97** resize our frame and apply background subtraction. Our motion detection is based on background subtraction which is the first step to generate a `mask`. Let's perform morphological operations on the `mask` now:

---

```

99      # perform erosions and dilations to eliminate noise and fill gaps
100     mask = cv2.erode(mask, eKernel,
101         iterations=conf["erode"]["iterations"])
102     mask = cv2.dilate(mask, dKernel,
103         iterations=conf["dilate"]["iterations"])
104
105     # find contours in the mask and reset the motion status
106     cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
107         cv2.CHAIN_APPROX_SIMPLE)
108     cnts = imutils.grab_contours(cnts)
109     motionThisFrame = False

```

---

The output of these operations can be seen in Figure 9.1 (with the goal of producing the *top* output image). Here we can see the raw `mask` after applying motion detection (*bottom-left*). Erosions are performed to remove noise in the `mask` (*bottom-middle*). Subsequently, we apply dilations to connect close by blobs together (*bottom-right*) prior to searching for contours.

**Lines 106-108** find and grab contours in our image. The `motionThisFrame` flag is reset to `False` — we'll have to determine its new value by processing contours:

---

```

111     # loop over the contours
112     for c in cnts:
113         # compute the bounding circle and rectangle for the contour
114         ((x, y), radius) = cv2.minEnclosingCircle(c)
115         (rx, ry, rw, rh) = cv2.boundingRect(c)
116
117         # convert floating point values to integers
118         (x, y, radius) = [int(v) for v in (x, y, radius)]

```

---

We begin looping over contours (**Line 112**). To estimate the size of a contour, we'll calculate its minimum enclosing circle radius (**Line 114**). The  $(x, y)$ -coordinates and radius will be used for annotation. We also calculate the bounding box coordinates for annotating where snapshots are be stored from the `fullFrame` (**Line 115**).

**Line 118** uses Python's list comprehension syntax to convert the bounding circle components to integers.

We'll now ensure that our contour is large enough and continue to process it:

---

```

120     # only process motion contours above the specified size
121     if radius < conf["min_radius"]:

```

---

```

122     continue
123
124     # grab the current timestamp
125     timestamp = datetime.datetime.now()
126     timestring = timestamp.strftime("%Y%m%d-%H%M%S")
127
128     # set our motion flag to indicate we have found motion and
129     # reset the motion counter
130     motionThisFrame = True
131     framesWithoutMotion = 0
132
133     # check if we need to annotate the frame for display
134     if conf["annotate"]:
135         cv2.circle(frame, (x, y), radius, (0, 0, 255), 2)
136         cv2.rectangle(frame, (rx, ry), (rx + rw, ry + rh),
137                       (0, 255, 0), 2)

```

---

**Lines 121 and 122** ensure the contour is sufficiently large according to its radius — if not, we'll move on the next contour. Depending on how close to your bird feeder your camera is, you may need to adjust the "min\_radius" parameter in your config.

A timestamp is grabbed and converted into a readable string (**Lines 125 and 126**). The timestring will be part of our video/snapshot filenames.

**Lines 130 and 131** mark this frame as having motion and reset the `framesWithoutMotion` counter to zero.



Figure 9.2: An example drawing a bounding box and minimum enclosing circle surrounding the bird in our input frame.

From there, **Lines 134-137** annotate the frame with a circle and rectangle around the contour for visualization purposes as shown in Figure 9.2.

---

```

139     # frame to disk
140     writeFrame = framesSinceSnap >= conf["frames_between_snaps"]

```

---

```

141      # check to see if should write the frame to disk
142      if conf["write_snaps"] and writeFrame:
143          # construct the path to the output photo and save it
144          snapPath = os.path.sep.join([conf["output_path"],
145              timestamp])
146          cv2.imwrite(snapPath + ".jpg", fullFrame)
147
148          # reset the counter between snapshots
149          framesSinceSnap = 0

```

---

We will write a frame (snapshot/photo) if there have been enough "frames\_between\_snaps" and if the "write\_snaps" boolean is set in the config (**Lines 140-147**).

Let's work with our KeyClipWriter:

---

```

152      # start recording if we aren't already
153      if not kcw.recording:
154          # construct the path to the video file
155          videoPath = os.path.sep.join([conf["output_path"],
156              timestamp])
157
158          # instantiate the video codec object and start the key
159          # clip writer
160          fourcc = cv2.VideoWriter_fourcc(*conf["codec"])
161          kcw.start("{}{}.avi".format(videoPath), fourcc, conf["fps"])

```

---

If we haven't started recording (**Line 153**), then we specify the videoPath and start recording (**Lines 155-161**).

Let's now determine when to stop recording:

---

```

163      # check if no motion was detected in this frame and then increment
164      # the number of consecutive frames without motion
165      if not motionThisFrame:
166          framesWithoutMotion += 1
167
168      # update the key frame clip buffer
169      kcw.update(frame)
170
171      # check to see if the number of frames without motion is above our
172      # defined threshold
173      noMotion = framesWithoutMotion >= conf["keyclipwriter_buffersize"]
174
175      # stop recording if there is no motion
176      if kcw.recording and noMotion:
177          kcw.finish()

```

---

**Lines 165 and 166** increment `framesWithoutMotion` if there is no motion in this frame. **Line 169** updates our `kcw`, a step that must be performed during each iteration of our frame processing `while` loop.

If there is `noMotion` (i.e. `framesWithoutMotion` is greater than the `kcw` buffer) and we are currently recording, then, we'll finish recording (**Lines 173-177**).

Finally, we'll display our frame (if necessary) and perform cleanup:

---

```

179 # check to see if we're displaying the frame to our screen
180 if conf["display"]:
181     # display the frame and grab keypresses
182     cv2.imshow("Frame", frame)
183     key = cv2.waitKey(1) & 0xFF
184
185     # if the `q` key was pressed, break from the loop
186     if key == ord("q"):
187         break
188
189 # check if we're recording and stop recording
190 if kcw.recording:
191     kcw.finish()
192
193 # stop the video stream
194 vs.stop() if not args.get("video", False) else vs.release()

```

---

**Lines 180-183** wrap up our loop by displaying the frame on the screen. Displaying the frame is optional and depends on the "display" config setting.

If the "q" (quit) key is pressed (during display) or "ctrl + c" is pressed within the terminal, the `kcw` will finish up if needed (**Lines 186-191**). The video stream is then stopped (**Line 194**).

## 9.2.6 Deploying Our Bird Monitor System

You'll want to deploy your camera in a controlled environment. I recommend aiming your camera at a bird feeder with a relatively static background and in a place where the bird feeder doesn't blow in the wind so much that it sets off the motion detector

This project does not require Internet connectivity, so the most logical way to deploy it is upon reboot (Chapter 8). That said, you may find deploying via SSH and `screen` to be just as convenient.

I recommend collecting a few images using `raspistill` and then inspecting them to ensure that the camera is positioned well with the bird feeder in view.

Once your physical environment is set up, the command you'll need to use is:

---

```
$ python bird_mon.py --conf config/mog.json --video birds_10min.mp4
[INFO] opening video file `birds_10min.mp4`
[INFO] detecting motion and storing videos and images...
```

---

Note that this command is processing the `birds_10min.mp4` file included in the downloads of this text. If you wish to use your live webcam, leave off the `--video` switch:

---

```
$ python bird_mon.py --conf config/mog.json --video birds_10min.mp4
[INFO] detecting motion and storing videos and images...
```

---

You can replace the config file path with a config file of your choosing. As mentioned, we've provided two example configs in the project; however, I would suggest playing with the configurations, changing the values, re-running the script, and noting how different parameters can have quite a big difference in your output.

Wrapping up, Figure 9.3 contains some of the best stills captured from the provided example video recorded by PyImageSearch correspondence coordinator, David McDuffee:

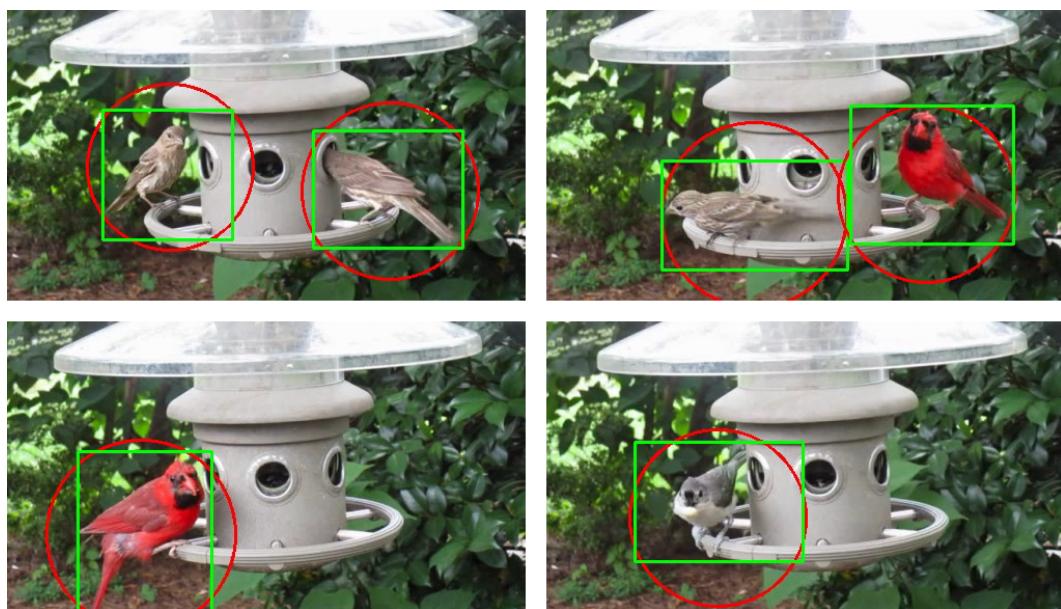


Figure 9.3: Sample stills of our bird feeder monitor successfully detecting and localizing birds in our video stream. Key event video clips were also created for each of these detections (but obviously cannot be displayed in a book). You can run the `python` commands provided in this section to reproduce these results.

Our algorithm, while simple, has enabled us to successfully build an intelligent bird feeder monitor using the Raspberry Pi.

### 9.3 Common Issues and Debugging Tips

Ideally your bird feeder won't move much (even in the wind) so that it doesn't trigger motion detection. I recommend that you aim your camera and place your bird feeder in such a way that there is a consistent background. Trees and other greenery are okay in the background so long as they aren't moving a lot in the wind.

**If no images are saved in the output folder, I recommend turning on annotation via the config file.** From there you'll be able to visualize if background subtraction and contour processing is working properly. You may also want to check your paths to ensure that your output path exists.

Make obvious accommodations for inclement weather — either monitor the weather and bring the gear inside before it rains or waterproof your electronics.

Lastly, use birdseed that attracts the birds you want to capture!

### 9.4 Summary

In this chapter we built a wildlife monitor for recording birds at a bird feeder. The bird feeder monitor uses background subtraction processing to determine if there is motion. We took advantage of a key clip writer to buffer our videos before and after motion events. Using the key clip writer ensures that we see the bird flying to and from the bird feeder.

Birds are very photogenic creatures, but don't let your imagination stop with birds. Maybe you want to monitor for another type of critter.

But what if the animals you want to monitor are nocturnal?

In Chapter 12 we'll learn to work in low light environments and then in Chapter 13 we'll deploy a nighttime wildlife monitor in the field.

## Chapter 10

# Sending Notifications from Your RPi to Your Smartphone

The primary goal of building Internet of Things (IoT) and edge computing applications is to enable devices to be “smart” — these programs should be able to detect certain patterns in the world, and once detected, perform some sort of action.

The action could be triggering an alarm, communicating with GPIO pins to control a robot, or most commonly, *sending some sort of notification*. In the world of smartphones, *text messages* and *push notifications* are the most common types of alerts we receive.

In this chapter we’ll explore the basics of sending notifications with the RPi. We’ll be specifically examining **text message notifications**, as they can be utilized anywhere in the world, regardless of your location, provided you have a cellular connection.

If you wish to swap in a different type of notification (including push notifications, Telegram, or WhatsApp alerts), the same techniques in this chapter can be applied — **just swap out the text message API calls for whatever action you want to take**.

Just like in Chapter 5 when we talked about accessing your camera, the code covered here will serve as a template that we will build on for the rest of the text.

### 10.1 Chapter Learning Objectives

In this chapter you will:

- i. Learn about the Twilio API and how it enables you to send text message notifications programmatically
- ii. Register for a Twilio account and grab your API keys

- iii. Install the Twilio Python package on your system
- iv. Implement a Python class that encapsulates sending texts with Twilio (ensuring we can reuse the code in other projects in this book)
- v. Use your Python class in a separate script to send the texts

By the time you are done with this chapter you will have a good understanding of how to send notifications from your RPi.

## 10.2 Sending Text Messages Via an API

The RPi does not come with a built-in library to send text messages, and in fact, sending text messages is actually a pretty challenging problem *without* using external APIs.

My favorite API for sending text messages is Twilio. The Twilio service is cheap, easy to use, reliable, and will work for most countries in the world.

If you do not have access to Twilio, or simply don't want to use it, don't worry — *this chapter is still applicable to you!* You'll easily be able to swap in whatever notification service you want to use by following the template in this chapter.

### 10.2.1 What is Twilio?

Twilio enables developers to send/receive text messages and make/receive phone calls (<http://pyimg.co/62u18>) [43]. Twilio provides a simple set of cloud-based APIs to perform these actions.

Since Twilio is cloud-based, an internet connection *is required* to utilize it — if your RPi is deployed somewhere where internet is not available (via WiFi or cellular) then you won't be able to utilize Twilio (and of course you wouldn't be able to send a notification if you didn't have an internet connection in the first place).

There are other services that can be used for sending text messages and non-text notifications, including WhatsApp and Telegram, but for simplicity we will be using Twilio in this chapter as it's widely available and super reliable.

### 10.2.2 Registering for a Twilio Account

In order to utilize the Twilio API you first need to register for an account on Twilio's official website: <http://pyimg.co/hhg6l>.

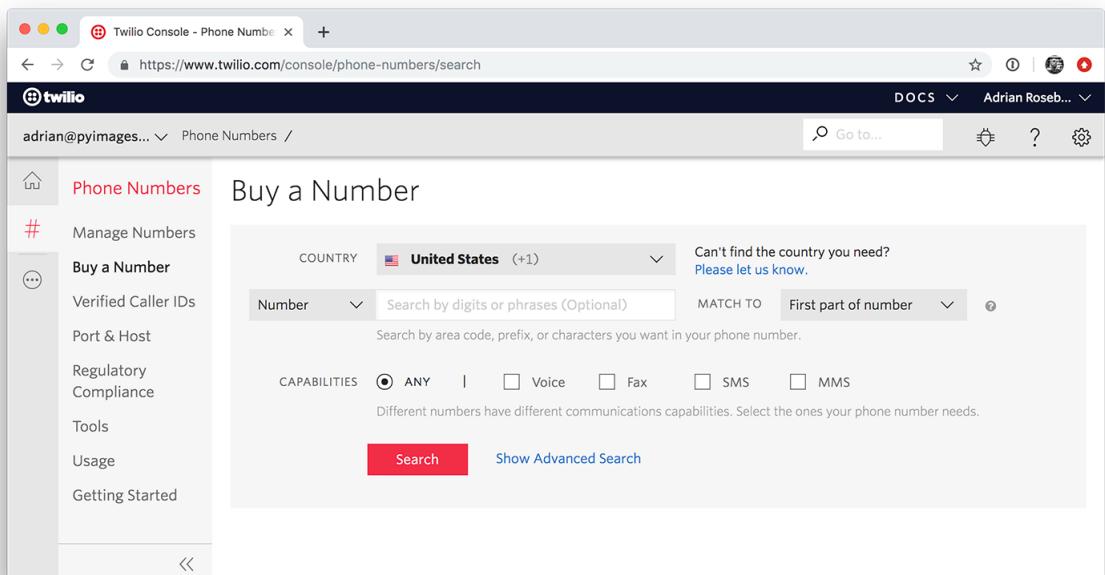


Figure 10.1: At the time of this writing, Twilio phone numbers start at \$1/month. If you choose to use Twilio, make sure you select a phone number with SMS and MMS capability.

After registering, you will be prompted to create a phone number (<http://pyimg.co/9bitq>), as shown in Figure 10.1. At the time of this writing, a Twilio phone number starts at \$1/month. A phone number *is required* to send/receive text messages, so the \$1 is a worthwhile investment (you can cancel your account at any time).

After creating your phone number you'll want to grab your API keys: <http://pyimg.co/uvno6>

Make note of your keys but *do not share them* as they are used for authentication when making Twilio API calls. Now that you have your Twilio API keys, we can move on to reviewing our project structure and creating a Python script to send text messages.

### 10.2.3 Installing the Twilio Library

If you followed the development environment configuration instructions in Chapter 3, then you should have the Twilio Python library installed on your system.

If you did not follow the instructions, or are configuring a fresh machine, you can install Twilio via the following command:

---

```
$ pip install twilio
```

---

You can then verify that Twilio can be imported via a Python shell:

---

```
$ python
>>> import twilio
>>>
```

---

Provided you have successfully imported the `twilio` package, you are safe to move on with the rest of the chapter.

## 10.3 Implementing a Python Script to Send Texts

In the first part of this section we'll review our project structure for the chapter as well as review the configuration file used to house our Twilio API keys. From there, we'll implement a Python script that will be used to send text messages via Twilio.

### 10.3.1 Project Structure

Our project has the following directory structure:

---

```
|-- config
|   |-- config.json
|-- pyimagesearch
|   |-- __init__.py
|   |-- notifications
|       |-- __init__.py
|       |-- twilionotifier.py
|   |-- utils
|       |-- __init__.py
|       |-- conf.py
|-- send_plain_message.py
```

---

Inside the `config` directory we have a `config.json` file — this file is in JSON format (making it easy to edit) and will house our Twilio API credentials. The `conf.py` file contains our implementation of the `Conf` class used to load our JSON file.

To actually *send* the text message, we'll encapsulate all necessary Twilio logic and API calls inside the `twilionotifier.py` file in the `notifications` submodule of `pyimagesearch`.

Finally, `send_plain_message.py` will put all the pieces together, enabling us to send a simple text message notification.

### 10.3.2 Our Configuration File

Let's take a look at our configuration file. For a review of the `Conf` class which parses JSON configurations, be sure to refer to Section 9.2.4.

Go ahead and open `conf.json` and you will see the following contents:

---

```
1  {
2      // variables to store your twilio account credentials
3      "twilio_sid": "YOUR_TWILIO_SID",
4      "twilio_auth": "YOUR_TWILIO_AUTH_ID",
5      "twilio_to": "YOUR_PHONE_NUMBER",
6      "twilio_from": "YOUR_TWILIO_PHONE_NUMBER"
7 }
```

---

As you can see, all Twilio API authentication information is stored in this file. **You will need to edit it and provide your own details** (I provided placeholder values as I cannot share my own API keys as they are specific to my account).

The `twilio_sid` is your sender ID while the `twilio_auth` is the authentication key for your account. You can find both of these parameters on your account page: <http://pyimg.co/uvn06>.

The `twilio_to` is the phone number that will be **receiving** text messages from your device while the `twilio_from` is your Twilio phone number (which can be found on this page: <http://pyimg.co/rk0hw>).

### 10.3.3 Sending Texts via Python and Twilio

Assuming you have edited the `conf.json` file to provide your own Twilio credentials and information, we can now move on to utilizing the `twilio` Python package to actually send text messages.

Open up the `twilionotifier.py` file in our directory structure and insert the following code:

---

```
1 # import the necessary packages
2 from twilio.rest import Client
3
4 class TwilioNotifier:
5     def __init__(self, conf):
6         # store the configuration object
7         self.conf = conf
```

---

**Line 1** imports our only required class, `Client`, from the `twilio` library. The `Client` class will be used to communicate with the Twilio REST API.

**Line 5** defines our constructor to the `TwilioNotifier` which only requires a single parameter, `conf`, which we assume to be a `Conf` object (which loads our JSON configuration file from disk).

To actually send text messages we'll utilize the `send` method:

---

```

9  def send(self, msg):
10     # initialize the twilio client and send the message
11     client = Client(self.conf["twilio_sid"],
12                      self.conf["twilio_auth"])
13     client.messages.create(to=self.conf["twilio_to"],
14                           from_=self.conf["twilio_from"], body=msg)

```

---

The `send` function accepts a single parameter, the message (`msg`) that we want to send.

We first instantiate a `client` on **Lines 11 and 12** using our Twilio API credentials. We then use the `client` to actually send the `msg` to the specified phone number.

In future chapters I'll show you how to modify the `TwilioNotifier` class to send **image** and **video** text messages as well!

## 10.4 Implementing the Driver Script

Let's put all the pieces together — open up the `send_plain_message.py` file and insert the following code:

---

```

1  # import the necessary packages
2  from pyimagesearch.notifications import TwilioNotifier
3  from pyimagesearch.utils import Conf
4  import argparse
5
6  # construct the argument parser and parse the arguments
7  ap = argparse.ArgumentParser()
8  ap.add_argument("-c", "--conf", required=True,
9                  help="Path to the input configuration file")
10 args = vars(ap.parse_args())

```

---

**Lines 2-4** import our required Python packages. We'll be using the `TwilioNotifier` class we just implemented to send the actual text message, while the `Conf` class will be used to load our configuration file.

Our command line arguments are parsed on **Lines 7-10**. We only need a single argument here, `--conf`, which is the path to our input configuration file.

It's now time to send the text message:

```
12 # load the configuration file and initialize the Twilio notifier
13 conf = Conf(args["conf"])
14 tn = TwilioNotifier(conf)
15
16 # send a text message
17 print("[INFO] sending txt message...")
18 tn.send("Incoming message from your RPi!")
19 print("[INFO] txt message sent")
```

**Line 13** loads our JSON configuration file while **Line 14** initializes our `TwilioNotifier`. The actual text message is sent on **Line 18**.

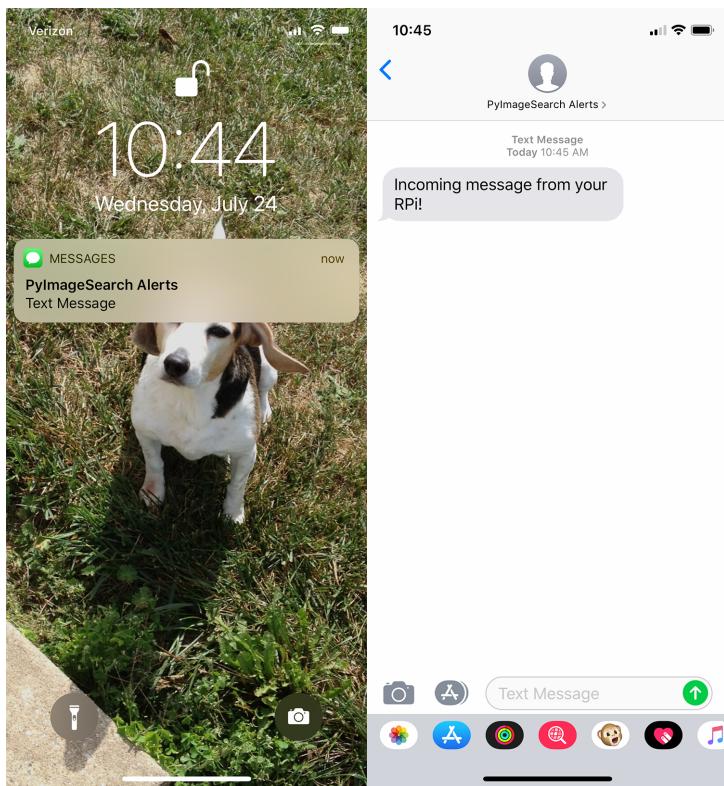


Figure 10.2: After executing the `send_plain_message.py` script a text message is delivered to my phone. In the next chapter you will learn how to send images/videos through the Twilio API as well.

### 10.4.1 Sending the Actual Text Messages

Now that our driver script is implemented, you can execute it using the following command:

---

```
$ python send_plain_message.py --conf config/config.json
[INFO] sending txt message...
[INFO] txt message sent
```

---

A few seconds after executing the script you should see a text message delivered to your phone from your Twilio phone number (Figure 10.2).

Again, the code we covered in this chapter is meant to be a *template* — **use it as a starting point for your own IoT/edge applications that need to send text messages**. I will be showing you how to send **image and video text messages** in Chapter 11.

## 10.5 Summary

In this tutorial you learned how to send text message notifications from IoT/edge devices using the Twilio API. While there are other API services you could use, including WhatsApp and Telegram, we chose to use Twilio here as the service is reliable and available for use in nearly every country.

It's also important to note that the examples in this chapter serve as a template. You should use this template as a starting point to build your own applications that need to send notifications — simply swap out the Twilio API calls and insert whatever API calls you need to perform your given action.

## Chapter 11

# Detecting Mail Delivery

In today's information age world, we expect information to be sent from or delivered to us near instantaneously. Unfortunately, mail and package delivery entities will never be able to meet that expectation fully (although attempts are being made with drones).

Have you ever expected a letter or important information in the mail?

Maybe you were waiting on a university acceptance letter. A birthday check from grandpa.

A passport for a travel excursion. A concert ticket. Or even a mail order prescription medications.

All too often, I find that I'm waiting on the United States Post Office (USPS). My friends in other countries may have even worse experiences than me with their mail systems.

I've mailed postcards to friends in other countries and six months later they ask "*How was your trip?*" I'm thinking, "*I didn't go anywhere.*" I ask them, "*Which trip?*" and they let me know they received a postcard from me. I'll be damned — I mailed that postcard six months ago and it is just now delivered to their mailbox?

In addition to slow delivery services, package and mail theft is on the rise.

Is there a way we can get to our goods before the thieves or catch them in the act?

We can't make the mail service faster or more efficient. We can't make the thieves better members of society.

But we *can* get notifications to our smartphone whenever the mailman or someone else opens the mailbox!

## 11.1 Chapter Learning Objectives

As the chapter title suggests, we're going to detect when a mailbox is *open* versus *closed* — and send an alert to your phone immediately. We'll also determine if someone left your mailbox open for a given period of time, at which point we'll also send a reminder.

Through this chapter, we'll learn and reinforce these objectives:

- i. Detecting a change in light with a camera sensor
- ii. Sending SMS notifications with Twilio

This chapter will help you learn the fundamentals of applying OpenCV on the Raspberry Pi.

Future chapters in this text will become progressively more complex and solve more real-world problems using computer vision and the RPi.

## 11.2 Detecting Mail Delivery

In this chapter, we'll walk through the project structure. From there, we'll review the configuration management class, `Conf`, and take a look at the configuration file for this project. We'll then develop and deploy our script to our mailbox. You'll be able to ensure that the script continuously runs on your Pi via the `screen` or "start on reboot" approaches.

Let's get started.

### 11.2.1 SMS Messages with Twilio

I've chosen to use Twilio for sending SMS message alerts, but other services and APIs do exist. Be sure to refer to Chapter 10 to learn more about sending messages with Twilio and other services.

### 11.2.2 Project Structure

Our project is laid out in the following manner:

---

```
|-- config
|   |-- config.json
|-- pyimagesearch
|   |-- notifications
|       |-- __init__.py
```



Figure 11.1: An overview of our mail delivery notification system. The Raspberry Pi + Camera sit hidden in the back of the mailbox to measure light thresholds. When the mailbox is opened and then closed, the Python Twilio API connects to your Twilio SMS account and sends a text message to a phone number of your choosing to let you know "Your mailbox at 12 Grimmauld Place was opened on Wednesday, August 07 2019 at 10:43AM for 3 seconds."

---

```

|   |   |-- twilionotifier.py
|   |-- utils
|   |   |-- __init__.py
|   |   |-- conf.py
|   |   |-- __init__.py
|   |-- mail_detector.py

```

---

Our configuration lies in `config.json` as opposed to passing many command line arguments. The `Conf` class inside `conf.py` will parse the JSON configuration and allow us to access the settings from within our driver script.

Twilio SMS notifications are made possible with `twilionotifier.py` and the `TwilioNotifier` class.

Our driver script, `mail_detector.py`, is responsible for determining if the mailbox is open or closed and sending notifications accordingly.

### 11.2.3 Our Configuration File

Let's go ahead and review our configuration. For a review of the `Conf` class which parses JSON configurations, be sure to refer to Section 9.2.4.

When you're ready, go ahead and open `config.json`:

---

```

1  {
2      // two constants, first threshold for detecting if the mailbox is
3      // open, and a second threshold for the number of seconds the
4      // mailbox is open
5      "thresh": 10,
6      "open_threshold_seconds": 60,
7
8      // flag used to indicate if frames must be displayed
9      "display": true,
10
11     // variables to store your twilio account credentials
12     "twilio_sid": "YOUR_TWILIO_SID",
13     "twilio_auth": "YOUR_TWILIO_AUTH_ID",
14     "twilio_to": "YOUR_PHONE_NUMBER",
15     "twilio_from": "YOUR_TWILIO_PHONE_NUMBER",
16     "address_id": "YOUR_ADDRESS"
17 }
```

---

Our configuration file has C-like comments beginning with `//`. JSON minify can also parse multiline C comments separated by `/*` and `*/`, but we've chosen to just show single line comments. Other than the comments, JSON files follow Python dictionary syntax.

Our light threshold is set to 10 (**Line 5**), meaning that at the average pixel intensity for a given frame must be greater than `thresh` for us to consider the mailbox as "open".

The maximum mailbox open threshold in seconds is set to 60 (**Line 6**). If the mailbox is left open for 60 seconds, an SMS alert will be sent to your cell phone indicating that the mailbox was left open by mistake.

You probably won't have a display connected next your mailbox, but for testing purposes, you may want to work with a display at your desk as you hack with the code. The `"display"` option on **Line 9** can either be `true` or `false`. **Keep in mind that booleans in Java, JavaScript, and JSON are lower case unlike Python where they begin with a capital letter.** Don't fall into the trap of entering booleans incorrectly.

All of our Twilio settings are shown on **Lines 12-16**. You need to find your `"twilio_sid"`,

"`twilio_auth`", and "`twilio_from`" values on the Twilio console online. The "`twilio_from`" phone number must be one of the phone numbers you have purchased (it could also be a free trial phone number).

Be sure to refer to Chapter 10 for screenshots of the Twilio settings console. Keep in mind that the settings panel is subject to change, so you may need to look around to find the keys and phone number.

#### 11.2.4 Our Mail Delivery Detection Script

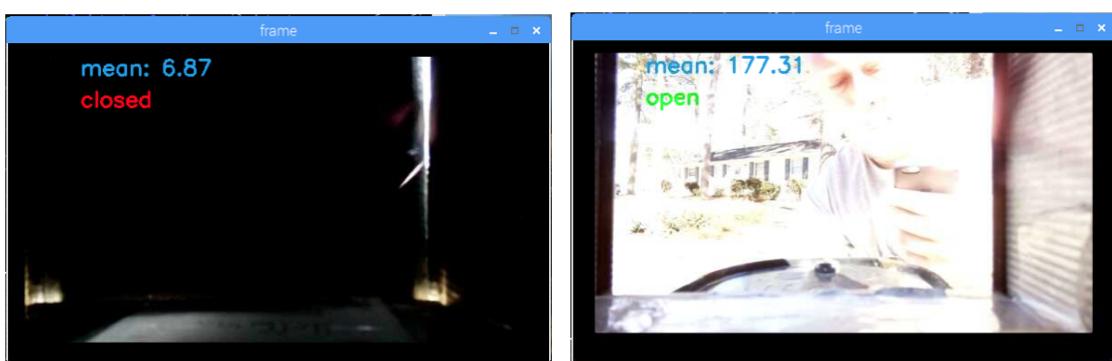


Figure 11.2: Original images are shown from the application. The mailbox closed/open detection algorithm simply (1) converts to grayscale, and (2) calculates the mean (average) of all pixels. From there the mean is compared to a threshold (in this case the threshold is 10 in the configuration file, but you should configure it based on experimental testing) to determine either closed (*left*) or open (*right*).

Our mail detection script will detect when the mailbox is opened by measuring the average light as shown in Figure 11.2.

This method makes the following assumption: Inside the mailbox is normally dark and when the mailbox is opened light enters the box. Be sure to keep this assumption in mind when working on the code and deploying your Raspberry Pi.

Go ahead and open a new file called `mail_detector.py` and insert the following code:

---

```
 1 # import the necessary packages
 2 from pyimagesearch.notifications import TwilioNotifier
 3 from pyimagesearch.utils import Conf
 4 from imutils.video import VideoStream
 5 from datetime import datetime
 6 from datetime import date
 7 import numpy as np
 8 import argparse
 9 import imutils
10 import signal
```

---

```

11 import time
12 import cv2
13 import sys
14
15 # function to handle keyboard interrupt
16 def signal_handler(sig, frame):
17     # handle keyboard interrupt by stopping the stream and exiting
18     print("[INFO] You pressed `ctrl + c`! Closing mail detector" \
19           " application...")
20     vs.stop()
21     sys.exit(0)

```

---

**Lines 2-13** import our necessary packages. Our `TwilioNotifier` class will be used for sending SMS notifications. The `Conf` class is for parsing and accessing our JSON configuration file.

The `signal` import on **Line 10** works with our `signal_handler` function on **Lines 16-21**. Whenever an interrupt signal is received (i.e. “ctrl + c”), this function will run at the current execution `frame` (i.e., not a frame that is displayed to your screen).

Let’s parse our command line argument and perform initializations:

---

```

23 # construct the argument parser and parse the arguments
24 ap = argparse.ArgumentParser()
25 ap.add_argument("-c", "--conf", required=True,
26                  help="Path to the input configuration file")
27 args = vars(ap.parse_args())
28
29 # load the configuration file and initialize the Twilio notifier
30 conf = Conf(args["conf"])
31 tn = TwilioNotifier(conf)
32
33 # initialize the flags for mailbox open and notification sent
34 mailboxOpen = False
35 notifSent = False

```

---

We have a single command line argument, `--conf`, the path to our JSON config file (**Lines 24-27**). Our `conf` is then initialized by creating a `Conf` object with the path parameter (**Line 30**). We also go ahead and initialize our `TwilioNotifier` (**Line 31**).

Our script will utilize two status variables: (1) `mailboxOpen` is a boolean indicating if the mailbox is currently open or closed, and (2) `notifSent` indicates if we have sent an SMS notification yet.

Let’s continue by initializing our video stream and signal trap:

---

```

37 # initialize the video stream and allow the camera sensor to warmup
38 print("[INFO] warming up camera...")

```

---

---

```

39 #vs = VideoStream(src=0).start()
40 vs = VideoStream(usePiCamera=True).start()
41 time.sleep(2.0)
42
43 # signal trap to handle keyboard interrupt
44 signal.signal(signal.SIGINT, signal_handler)
45 print("[INFO] Press `ctrl + c` to exit, or 'q' to quit if you have" \
46     " the display option on...")

```

---

Our `VideoStream` is initialized on **Line 40**. You'll likely want to use a PiCamera inside your mailbox for this project. **Line 44** initializes our “ctrl + c” interrupt `signal_handler`. Any time this signal is received, the `signal_handler` function will be run.

We're now ready to begin looping over incoming frames and implementing our application logic:

---

```

48 # loop over the frames of the stream
49 while True:
50     # grab both the next frame and preprocess it
51     frame = vs.read()
52     frame = imutils.resize(frame, width=200)
53     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
54
55     # set the previous mailbox status
56     mailboxPrevOpen = mailboxOpen

```

---

**Lines 51-53** grab a frame and preprocess it into a resized grayscale image.

**Line 56** then sets the previous mailbox open status from the value acquired during the last iteration of the loop.

The heart of today's app takes place where we **determine if the mailbox is currently open**:

---

```

58     # calculate the average of all pixels where a higher mean
59     # indicates that there is more light coming into the mailbox,
60     # then determine if the mailbox is currently open
61     mean = np.mean(gray)
62     mailboxOpen = mean > conf["thresh"]
63
64     # if the mailbox is open and previously it was closed, it means
65     # the mailbox has been just opened
66     if mailboxOpen and not mailboxPrevOpen:
67         # record the start time
68         startTime = datetime.now()

```

---

**Lines 61 and 62** determine if the mailbox is open. The calculation is very simple. First we

calculate the average/mean of all grayscale pixels. Remember values are in the range  $[0, 255]$  where 0 is darker and 255 is brighter. If the average of all pixel values is high, then there is a lot of light (i.e. the mailbox is open). The `mean` is compared to our threshold on **Line 62** to determine `mailboxOpen` status. To see an example, refer to Figure 11.2.

If the mailbox is freshly open (i.e. it was not previously open), we'll grab the current timestamp (**Lines 66-68**).

Otherwise, the mailbox was already open and we need to send notifications accordingly:

---

```

70      # if the mail box is open then there are 2 possibilities,
71      # 1) it's left open for more than the *threshold* seconds.
72      # 2) it's closed in less than or equal to the *threshold* seconds.
73  elif mailboxPrevOpen:
74      # determine if the mailbox was left open for longer than the
75      # threshold time
76      elapsedTime = (datetime.now() - startTime).seconds
77      mailboxLeftOpen = elapsedTime > conf["open_threshold_seconds"]:
78
79      # handle when the mailbox was left open
80  if mailboxOpen and mailboxLeftOpen:
81      # if a notification has not been sent yet, then send a
82      # notification
83  if not notifSent:
84      # build the message and send a notification
85      msg = "Your mailbox at {} has been left open for " \
86          "longer than {} seconds. It is possible that you" \
87          " or the mailman didn't close your mailbox.".format(
88              conf["address_id"], conf["open_threshold_seconds"])
89      tn.send(msg)
90      notifSent = True

```

---

When the mailbox is left open for two iterations of the `while` loop, we need to take action beginning on **Line 73**. **When the mailbox is left open too long (Lines 76-80)**, then we'll send a notification to the homeowner (if required). The notification alerts the homeowner that someone opened the mailbox but did not close it within the threshold time (in our config it is set to 60 seconds, but you are welcome to change it).

Otherwise, the mailbox has recently been closed (i.e. **mail has been delivered**), so we'll send a different notification:

---

```

92      # check to see if the mailbox has been closed
93  elif not mailboxOpen:
94      # if a notification has already been sent, then just set
95      # the our boolean to False for the next iteration
96  if notifSent:
97      notifSent = False

```

---

```

98
99      # if a notification has *not* been sent, then send a
100     # notification
101  else:
102      # record the end time and calculate the total time in
103      # seconds
104      endTime = datetime.now()
105      totalSeconds = (endTime - startTime).seconds
106      dateOpened = date.today().strftime("%A, %B %d %Y")
107
108      # build the message and send a notification
109      msg = "Your mailbox at {} was opened on {} at {} " \
110          "for {} seconds.".format(conf["address_id"],
111          dateOpened, startTime.strftime("%I:%M%p"),
112          totalSeconds)
113      tn.send(msg)

```

---

When the mailbox is opened and closed within the threshold time, it is likely that new mail has been delivered (or maybe a criminal has opened the mailbox to steal your goods).

If the notification has already been sent, we simply reset the `notifSent` flag (**Lines 96 and 97**).

Otherwise, we calculate a timestamp and determine today's date (**Lines 104-106**) and send an appropriate message to your cell phone (**Lines 109-113**). The notification contains your address (in case you have multiple properties), the date, time, and number of seconds your mailbox was opened.

That takes care of all of our mailbox and notification logic. Let's display our `frame` and clean up:

```

115      # check to see if we should display the frame to the screen
116  if conf["display"]:
117      # show the frame and record any keypresses
118      cv2.imshow("frame", frame)
119      key = cv2.waitKey(1) & 0xFF
120
121      # if the `q` key is pressed, break from the loop
122  if key == ord("q"):
123      break
124
125  # cleanup the camera and close any open windows
126  cv2.destroyAllWindows()
127  vs.stop()

```

---

Optionally, we'll display the frame until the "q" (quit) key has been pressed (**Lines 116-123**). **Lines 126 and 127** then perform cleanup.

### 11.2.5 Deploying Our Mail Delivery Detector



Figure 11.3: A Raspberry Pi is discretely deployed with a USB battery pack in the rear of my mailbox. Now instant alerts will be delivered to my cell phone when my concert tickets arrive!

Let's deploy our mail delivery detector.

But first, I recommend that you test it once or twice to ensure that you are receiving the notifications on your phone. You can test it at your desk by putting a dark piece of fabric over the camera sensor. Start the script and then remove the fabric for a short period of time, replace it, and see if you receive an alert on your phone in a few moments.

When the system is working properly, you'll need a power source at your mailbox. For proof-of-concept purposes, I recommend a battery pack before you run power. This is also the perfect project for a Raspberry Pi Zero W as it draws less power than the Pi 3B+ and Pi 4.

Once you're powered up, deploy the Pi and battery secretly into your mailbox, preferably in such a way that the mailman won't bump it, as shown in Figure 11.3. You'll also need to ensure that your Pi has a steady WiFi signal in the mailbox. If that isn't possible, then cellular is an option. There are a number of Raspberry Pi Cellular HATs — just keep in mind the additional power draw. A good directional WiFi antenna aimed at your mailbox may do the trick as well.

You can deploy this program using the `screen` command or the “start on boot” method

discussed in Chapter 8.

The command for launching the program is:

```
$ python mail_detector.py --conf config/config.json
[INFO] Press 'ctrl + c' to exit, or 'q' to quit if you have the
display option on...
```

You will now receive alerts when the mailbox is opened or when the mailbox is left open.



Figure 11.4: **Left:** An alert for a normal mailbox delivery. **Right:** An alert for when the mailbox is left open for greater than 60 seconds.

## 11.3 Summary

In this chapter, you learned how to deploy a mail delivery notification system. While the project is admittedly simple, you learned how to build an entire computer vision and OpenCV pipeline

on the Raspberry Pi. This pipeline will be adapted to more complex applications later in the text.

The crux of our project takes advantage of a simple pixel average, where a higher number indicates the presence of light, to determine if our mailbox is open. From there, the rest of the script uses the open/closed logic to send messages to our cell phone.

For a similar project, be sure to review my blog post where we learned how to send MMS security *videos* with the Raspberry Pi [44] (<http://pyimg.co/gdy2a>).

More advanced security applications leverage motion detection, which we'll be covering in Chapters 13, 14, 19, and 20. The *Hacker Bundle* of this text includes more advanced security applications that leverage object detection and deep learning as well.

Finally, If you don't have the luxury of a controlled light environment (as is the basis of this chapter), you may wish to refer to Chapters 6 and 12 where you will learn how to change camera settings and work in low light conditions.

## Chapter 12

# Working in Low Light Conditions with the RPi

Light. The force of nature that makes it possible for humans and animals to see and interpret their surroundings. The sensor on your camera turns photons of light into signals that are processed and turned into an image.

That is where our image processing begins — with light.

But what can we do in the absence of light (or more than likely, lack of *sufficient* light)?

For example, what are our options when we want to film wildlife at night without startling them?

Luckily for us, engineers and scientists have measured the electromagnetic spectrum. We know that most of the spectrum is invisible to humans. We can only see a very tiny band of the entire spectrum. Animals see in a different spectrum from humans as well.

Cameras are different. With proper engineering, a camera can be designed to be sensitive to a frequency band of light that our human eye is not. Meet infrared, ultraviolet, and wide-band spectrum cameras.

Or maybe we can't call these light detectors "cameras" at all. Maybe it is some other sort of sensor, like a system that uses multiple satellites around the world to image a black hole.

In addition to imaging in a specific spectrum, you can pair your special camera with the right lighting, and then the invisible becomes visible. Image processing somewhere in the pipeline will translate invisible signals into visible light that we can see on our computer screen.

Part of the focus of this chapter will be on infrared cameras and lights, but you shouldn't let your imagination stop there. You can apply the concepts of this chapter to ultraviolet light or any electromagnetic spectrum such as radio or X-ray.

Knowledge of photography will help you succeed in making the ideal settings changes to your camera in low light conditions.

Let's dive into the darkness.

## 12.1 Chapter Learning Objectives

Our objectives for this chapter revolve around low light conditions. We'll learn:

- i. What is visible light
- ii. What is infrared light
- iii. Types of lights you can add to projects
- iv. Controlling lights with an RPi
- v. Non-traditional cameras
- vi. Photography tips

By the end of this chapter, that Edison light bulb idea beacon should be flickering on for your projects.

## 12.2 Light, Low Light, and Invisible Light

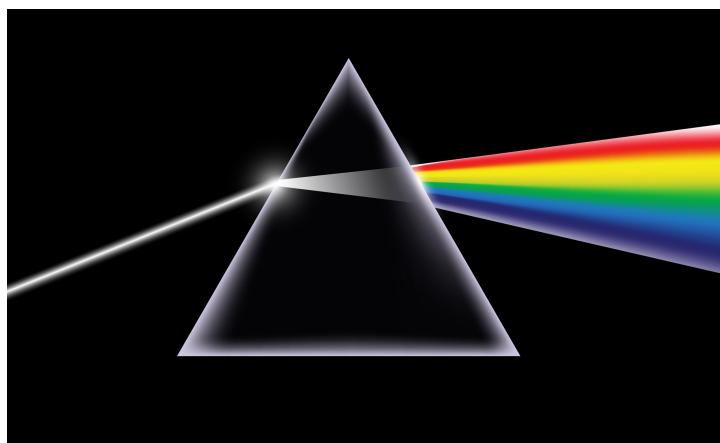


Figure 12.1: Light is critical to *every* computer vision project. You not only need to consider the lighting itself, but also how you may *control* lighting conditions — remember, it's *far easier* to write code for *controlled* lighting conditions than for *uncontrolled conditions*.

Light takes on many forms and is **essential to every computer vision project**. If one of the computer vision examples in this book (or my blog) hasn't worked for you, consider

that it may not be the actual *code*, but instead, that your *environmental lighting conditions* are different than mine. Simply turning on/off a lamp, drawing the blinds, or going outside could have a *dramatic* effect on the results.

When you work with computer vision, it is important to *not only consider lighting, but also how to control lighting*.

After all, light is the basis of how a camera works and thus is extremely important.

All too often, I see that computer vision software developers focus solely on their software and don't give any thought to their environment and a little bit of elementary physics.

### 12.2.1 What is Visible Light?

The electromagnetic spectrum (Figure 12.2) is divided into seven bands of increasing frequency (decreasing wavelength):

- i. Radio waves
- ii. Microwaves
- iii. Infrared (IR)
- iv. Visible light
- v. Ultraviolet (UV)
- vi. X-rays
- vii. Gamma-rays

For most (but not all) computer vision applications, we typically only care about #4: *Visible light*.

**Visible light** includes wavelengths of light that are visible to the *human* eye and sits between the IR and UV spectrums. Humans see visible light in the form of color, but not all animals do. Color exists only in the mind! Color bands follow the ROYGBIV spectrum ordering:

- i. Red
- ii. Orange
- iii. Yellow
- iv. Green
- v. Blue

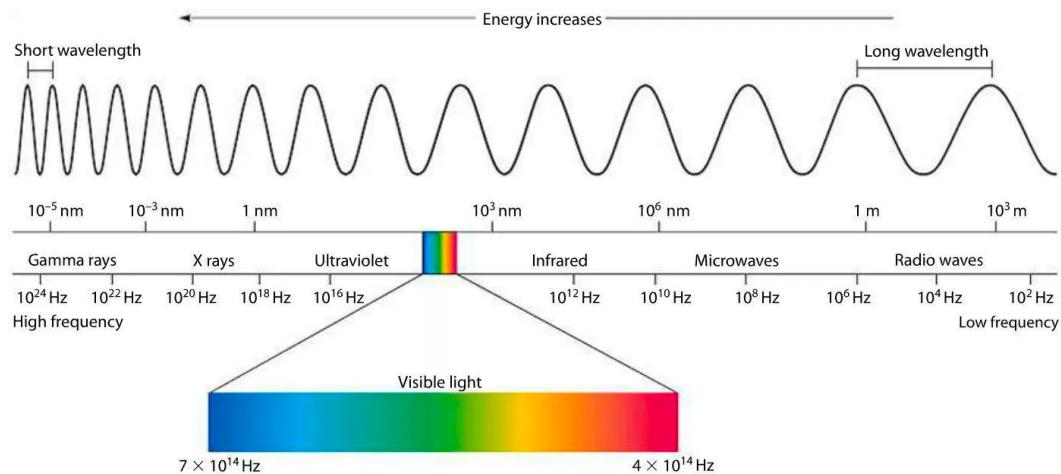


Figure 12.2: Visible light is only one *tiny subset* of the electromagnetic spectrum. Most light, including radio waves, microwaves, infrared, gamma rays, and X-rays are *invisible*. (Image source: <http://pyimg.co/c73pm>)

vi. Indigo

vii. Violet

Camera sensors have been engineered to be sensitive to visible light. Color perception is accomplished via more engineering/math in the camera hardware/firmware before being passed to software. As we learned in Chapter 4, typically we represent colors in code as a 3-tuple of (Blue, Green, Red) values from [0-255]. Be sure to review Section 4.3.2 for more information.

### 12.2.2 What is Infrared Light?

Infrared light is a specific frequency band that sits between microwaves and visible light; thus, it is invisible to humans unless special cameras are used. Most animals aren't bothered by infrared light, so if you are imaging wildlife at night, utilizing both an infrared light source and an infrared camera is a great way to see nighttime critters.

There are many suitable infrared lights and cameras available for purchase in online marketplaces — finding these products in stores is unlikely so we've included a set of recommended cameras on the companion website. You can find a link to access in the companion website in the first few pages of this text.

### 12.2.3 Types of Lights You Can Add to Your Projects

When choosing your lighting, it is important to consider the use case scenario.

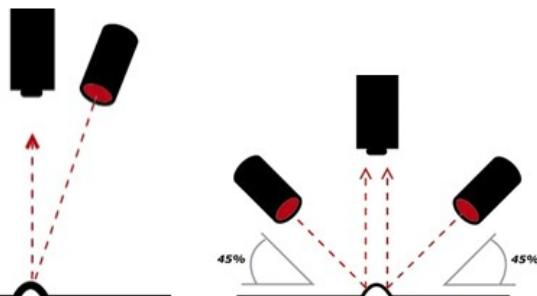


Figure 18. Directional Bright Field

Figure 19. Dark Field Lighting

### Directional and low angle lighting sources

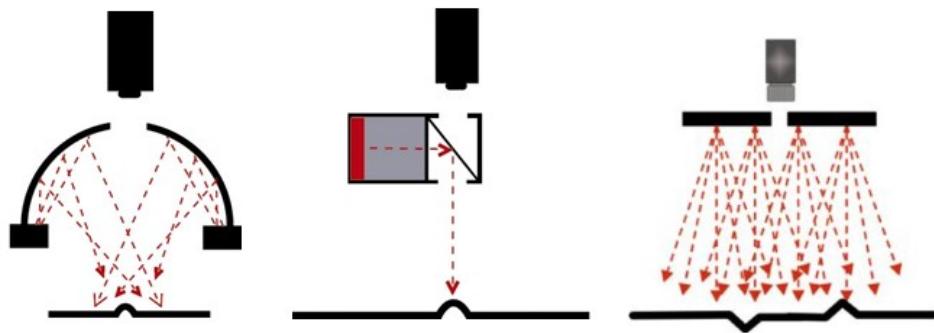


Figure 17a. Dome Diffuse

Figure 17b. On-axis Diffuse

Figure 17c. Flat Diffuse

### Diffuse lighting sources

Figure 12.3: Example light source diagrams [45]. **Top:** Directional and low angle light sources. **Bottom:** Diffuse light sources. [46]

Light is very directional. Light can also bend. Light can cause optical illusions. Light can either help or hinder your project goals. You need to choose your light source(s) wisely and likely try several methods before you land on the most effective one.

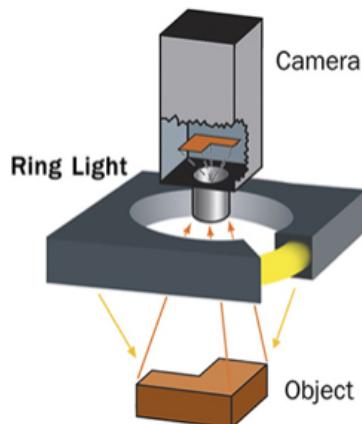
For example, you trying to prevent reflections? Maybe you need a polarizing filter.

Are you trying to increase overall light without bringing in shadows? Try diffuse lighting.

Do you need to illuminate a reflective surface? Place the lighting at a low angle and the camera above the target so that the light doesn't directly reflect into the camera lens and sensor.

Will a specific color/frequency of light better illuminate the target? Sometimes colored or white light can make all the difference.

Due to how expensive machine vision cameras and software are, hackers like us like to



**Ring lighting source**

Figure 12.4: Ring lighting is a form of low angle lighting which illuminates a target from all directions [46].

keep hundreds of extra dollars in our wallets by putting Raspberry Pis on our machines and assembly lines. Don't forget the most important part of engineering your vision system, though: the lighting. Whether you use a \$30USD RPi or a \$800USD Microscan/Cognex camera, you still need to add auxiliary lighting to your project in most cases.

Your lighting (whether it is visible or invisible) can take on the following forms:

- **Directional** lighting (also known as "partial bright field" lighting) is the most common light source. The sun is a directional light source. With this type of lighting, a point source delivers light to the target. Refer to Figure 12.3 (*top-left*).
- **On-axis** lighting is when the light and camera are in the same plane and on the same axis. Usually I find that this type of lighting requires a polarizing filter due to glare/hotspots.
- **Low-angle** lighting (a form of *directional* lighting) works well if you are imaging a shiny surface such as metal or plastic. The low angle will cause shadows in a positive way that may make the target stand out. Figure 12.3 (*top-right*) shows an example of low angle lighting.
- **Diffuse** lighting to eliminate reflections and glare. Light passes through a diffuser or is aimed backwards into an umbrella-shaped object to scatter the light in all different directions. Figure 12.3 (*bottom*) shows three example forms of diffuse lighting.
- **Ring** lighting is another form of lighting (typically *low-angle*) where a ring of light goes around the target or around the camera's lens as shown in Figure 12.4. In these situations usually the camera's lens is really close to the target and low angle lighting is needed from

all sides. I've found this helpful when imaging laser-etched plastic parts for OCR. Ring lighting tends to be expensive because it is hard to improvise this type of lighting on your own.

- **Backlighting** is not always for low light scenarios, but is worth mentioning. When you have the ability to place light behind the target, it can make it easier to detect shapes and contours. For example, you may wish to aim a camera at a fish tank and then place the light source behind the fish tank, aiming through it and towards the camera so that the fish appear like dark contours. You can then use image processing techniques such as background subtraction, allowing you to extract and track the fish.

Refer to this whitepaper [45] by National Instruments on lighting sources for more information.

Now that we have an understanding of light, let's learn how we can control lighting conditions with the Raspberry Pi.

## 12.3 Controlling Lights with an RPi

Raspberry Pis have the capability to interface with all sorts of hardware in addition to cameras and USB devices. Thus, when you work with your camera, you might find that you want to control a light with your Raspberry Pi.

There are two main ways to control lights with your Raspberry Pi:

- **GPIO:** Using the general purpose input/output pins on the RPi header to send on/off voltage signals to your light source.
- **IoT lights with APIs:** Internet of Things lights have their own hardware for controlling a light and come with some sort of software interface (usually via IP or Z-band wireless) to turn the light on and off.

We will learn both methods by example in this book, but first let's get to know them better.

**GPIO light control** works well for controlling just about any type of light that can be powered by DC or AC power.

However, there's a problem that non-electronics tinkerers may overlook. Almost always, the pins on your Raspberry Pi can't source enough current to directly power a light. The pins provide 3.3V, but at a very low current. Two great options for overcoming this problem so that you can control the lights with GPIO pins include MOSFETs and Relays.

**MOSFETs** are special transistors that act as switches. You can buy a single finger-size MOSFET to control power to a device. The MOSFET accepts a signal from GPIO (on or off)

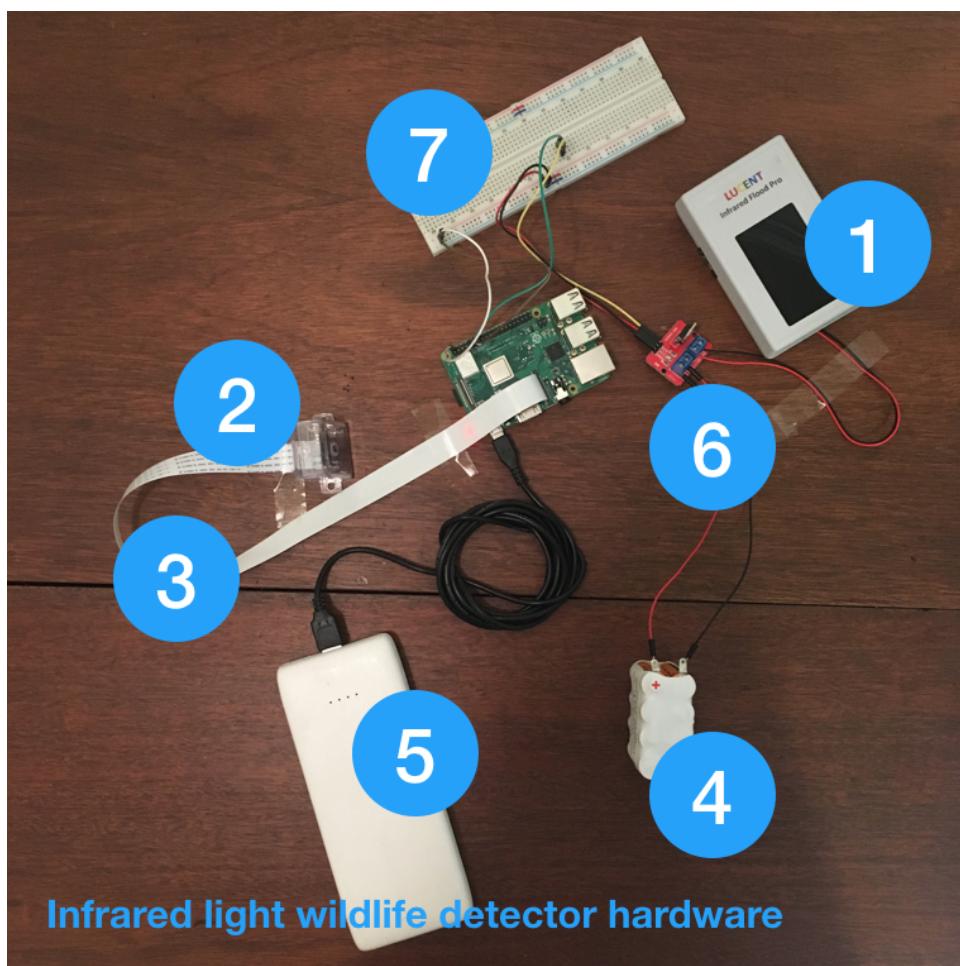


Figure 12.5: The nocturnal wildlife monitoring project (Chapter 13) uses a **MOSFET module** to take a 3.3V signal and power a 9V light from a battery. Raspberry Pis cannot power high voltage or high current lights from GPIO pins directly without using a MOSFET or relay. Electronic hardware for our deployable nocturnal wildlife monitoring box includes an infrared light (#1), infrared camera (#2), 18-inch ribbon cable (#3), 9.6V 2Ah rechargeable battery (#4), 5V 5.8A 22Ah portable USB power bank (#5), **MOSFET module** (#6), breadboard (#7), and Raspberry Pi (center).

and then distributes power from a larger source (battery/power supply) to the target device. We'll use MOSFETs to control a battery powered IR light in Chapter 13.

**Relays** (either traditional mechanical relays or solid state relays) are another means of controlling power. Traditional relays have a physical switch that clicks into place, completing a circuit. You might hear relays in your home thermostat, oven, or refrigerator from time to time. There's even a product called IOTRelay [47] that is geared towards Arduino and Raspberry Pi users for discrete lighting control. These days there are more modern solid state relays that are similar to MOSFETs.

Looking up MOSFETs and Relays on a search engine will yield many options and circuit diagrams. Building your own circuit can be a great way to save a few bucks if you have both

time and skill with a soldering iron.

I prefer to take the lazy (and usually more reliable) approach by buying a "module". A MOSFET module conveniently packs the MOSFET and other hardware (mainly wire receptacles/clamps and a heatsink) into a small package that is ready to go. *Voila!* No soldering necessary for us software folks (you'll still need to wire it up, of course). Similar modules are available for relays as well.



Figure 12.6: **Left:** Smart plug. **Center:** Smart light bulb. **Right:** Smart switch.

**IoT + API light control** is great when you are near WiFi (in the case of WiFi smart lights/switches), Zigbee, or a Z-band light controller. Smart plugs (Figure 12.6, *left*) allow you to plug a small, network-enabled brick into your wall to turn lights (or other devices) on and off using an API or smartphone app. We'll use a smart plug product in Chapter 21 of the *Hacker Bundle* to illuminate someone's face automatically.

Smart lights (Figure 12.6, *center*) screw into a light socket and smart switches (Figure 12.6, *right*) hide inside your wall and enable you to control the lights from an API or app.

Zigbee and Z-wave lights use a separate protocol from WiFi; however, they accomplish the same goal. While similar technologies, Zigbee is an open standard whereas Z-wave is proprietary. These products are still used, but not as convenient because you have to have another base station for the specific wireless signal (usually 908 MHz radio signals). Some people prefer it to keep their home network devices separate from their lighting for obvious security reasons.

Be sure to refer to the book's companion site hardware listing for links to recommended hardware. I've elected not to list specific hardware in this chapter as part numbers and distributor hyperlinks go out of date faster than my book makes it off the printing press.

## 12.4 Cameras and Non-standard Cameras

Cameras attempt to match the human eye, though as you can imagine, cameras and eyes are quite different.

An eye is filled with nerve endings that are sensitive to light. They also convert light into electrochemical signals that our brain can interpret. It is great that our eyes are so close to our brain because there's a lot going on between them as they communicate back and forth.

Infrared, ultraviolet, and multi-spectral cameras are no different in concept. Their key difference is that the sensors are sensitive to different frequencies that we typically can't see.

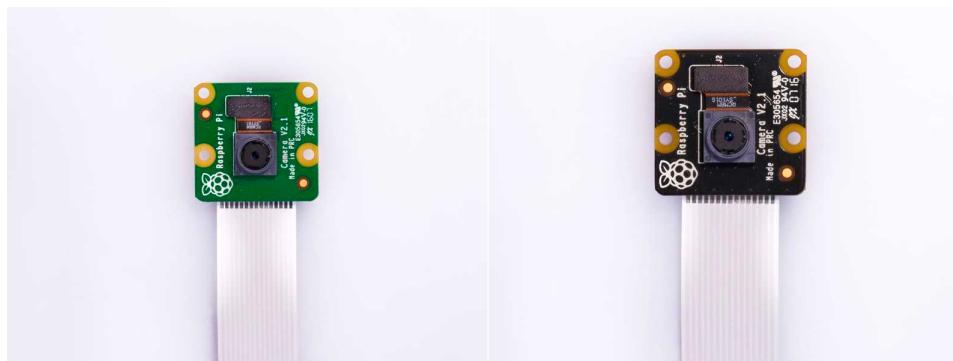


Figure 12.7: **Left:** The standard Raspberry Pi camera module. **Right:** The Pi NoIR V2 infrared camera module suitable for working in low light conditions.

A camera sensor is an array of photo sensors that receive photons, convert the information into voltage signals, and finally into a digital image. When it comes to the Raspberry Pi, there are plenty of options. The most obvious and popular choice is the **PiCamera V2** (Figure 12.7, *left*). For infrared light, the **PiCamera NoIR V2** (Figure 12.7, *right*) also interfaces to the PiCamera port. We'll put the NoIR camera to use in Chapter 21 of the *Hacker Bundle*.

A low cost **ultraviolet camera** has been built from the standard PiCamera [48] though only experts with equipment and supplies should attempt this.

And then there is always USB. For example, Joseph Howse spoke about the low-cost XNiteUSB2S-MUV ultraviolet camera during his PyImageConf 2018 talk, “*Visualizing the Invisible*” [49].

Due to so many options on the market for camera sensors, I cannot begin to cover them all. Be sure to do your research, read reviews, and ask in online communities such as Py-

ImageSearch Gurus (<http://pyimg.co/gurus>). It is also important to become familiar with the specification sheets when choosing cameras and lights. Ensure that your light spectrum range covers the sensor's range to ensure beautiful photos.

## 12.5 Photography Tips

Adapting lighting to your needs comes naturally with experience working with many different computer vision projects. We software folks need to stay cognizant of the fact that not everything can be solved from our keyboards. Sometimes we need to step away from the code and physically adjust lights, wiring, and circuits. Of course, there are times where knowledge of software as well as photography can be very helpful.

The PiCamera V2 API allows you to control **ISO** (60-1600) and **shutter speed** (sub-second to 10s) and two of the three sides of the exposure triangle (the third is **aperture** which is fixed at 2.0 on the PiCamera V2).

**Remark.** *There are a number of PiCamera-compatible cameras on the market, but none seem to allow for the aperture to be controlled (at least as far as I know).*

When you're working in low light conditions, you should try increasing your **ISO**, just beware that higher values lead to more noise in your image.

**Shutter speed** is the length of time your shutter is open. The longer you leave your shutter open, the more light will enter the sensor and the brighter your image will appear. Slow shutter speeds are not generally appropriate for fast motion (unless you want a cool blended motion effect).

Some cameras will allow you to control **aperture**. Aperture is the size of the hole where light enters your camera sensor. The larger the aperture, the brighter the image. You can experiment with larger apertures in low light conditions.

A detailed discussion of ISO, shutter speed, and aperture is outside the scope of this book. If you're interested in learning more about them from a high-level photography perspective, be sure to refer to the following articles:

- *Complete Guide to ISO for Beginner Photographers* (<http://pyimg.co/2nfuv>)
- *Introduction to Shutter Speed in Photography* (<http://pyimg.co/gx0oe>)
- *Introduction to Aperture in Photography* (<http://pyimg.co/4jbzp>)

Other “lighting conditions” controls include **exposure mode** and **automatic white balance** settings.

A wide-angle lens collects more light, so cameras that include wide-angle lenses will provide more low-light flexibility.

**Gamma correction** can be applied to images to accommodate different perceptions of luminance. The process of gamma correction is a translation between the sensitivity of our eyes and the sensors of a camera. Be sure to review my *OpenCV Gamma Correction* blog post (<http://pyimg.co/p9rir>) [50] where a simple equation is presented that may help your software in some lighting conditions. Automatic gamma correction is a significantly more complicated subject and is not covered in the article (or this book, for that matter).

To review setting camera settings for both USB webcams and PiCameras, go back to [6](#).

## 12.6 Summary

In this chapter we briefly reviewed physics of light. From there we discussed adding light to your low light projects and learned that in some cases you may want to add infrared light so as not to disturb animals.

The RPi is able to control lighting via GPIO and IoT — you just need to be willing to work with hardware.

We also briefly discussed a handful of photography terms which I highly encourage you to read more about in other resources as photography concepts are outside the scope of this book.

When you are working in low light, you have the option to (1) add auxiliary lighting, (2) select the ideal camera, and/or (3) tune your camera sensor parameters. Any combination of these approaches will lead to a successful project in low light conditions.

Armed with the knowledge of this chapter, in Chapter [13](#) we're going to build a wildlife observation box so we can catch some friendly critters on camera.

## Chapter 13

# Building a Remote Wildlife Detector

Have you ever woken up in the mountains, cooked a nice breakfast over a cast iron skillet, and then cleaned up before a hike?

On your way out, you surely remembered to take the trash to the outside canister since the smell of bacon grease isn't one you want lingering in the cabin.

When you reach the trash can you discover that some unwanted overnight guests made a huge mess to pick up.

Was it bears?

Raccoons?

You'll never know unless you monitor your trash for the little buggers with a Raspberry Pi infrared camera.

We learned to monitor birds at a feeder in Chapter 9. Then we learned how to work in low light conditions in Chapter 12.

It only makes sense that now we're going to monitor the night for nocturnal critters and beasts.

### 13.1 Chapter Learning Objectives

In this chapter, we'll:

- i. Build a deployable wildlife detector using inexpensive hardware
- ii. Control lights with a Raspberry Pi using MOSFETs
- iii. Reinforce background subtraction motion detection

iv. Reinforce key clip writing

## 13.2 Chapter Layout

This chapter is laid out a little differently. In the next section, we'll cover hardware for designing and assembling an inexpensive, waterproof, battery powered, nocturnal wildlife monitoring box.

From there we'll write the Python software which interacts with the infrared camera, MOS-FET switch, and infrared light. The software will mimic a few aspects of the bird feeder monitor discussed in Chapter 9, so you may want to review that chapter first.

Finally, we will deploy our system in the wild and discuss tips and suggestions to successfully catch the nocturnal beasts on camera!

## 13.3 Hardware: Building a Deployable Wildlife Detector Box

This chapter is the most hardware tinkering-centric of the book and should appeal to the Raspberry Pi community of hardware folks. In fact, I'm guessing that people who have built Raspberry Pi hardware projects are even more experienced than me.

This project is meant to appeal to those who are more software oriented, so we'll be sure to cover the basics.

For those who are more software-oriented, just take this section one step at a time. If you purchase the hardware that I recommend, there will be little-to-no soldering and more plugging in wires with the right polarity.

Our task in this section is simple: We're going to design and assemble a box to connect our gear and keep it safe. We have the following goals:

- Emphasize resourcefulness
- Keep costs to a minimum
- Make our system easy to assemble
- Ensure our system can run without external power all night long
- Keep the electronics safe from moisture

Of course, you're welcome to get creative with the hardware, especially if you are deploying the system for an extended period of time and need to add daytime solar charging and more.

### 13.3.1 Hardware Requirements

Our list of hardware includes batteries, an infrared light, an infrared sensitive camera, and a MOSFET so that we can turn the lights on and off. As we learned in Chapter 6, standard cameras don't see much at night unless auxiliary lighting is present or camera parameters are changed.

The same is true for an infrared camera — they need infrared lights. We use an infrared light and infrared camera so that the wildlife is not disturbed by bright white or yellow colors.

For this chapter, we need a light that we can power with a battery for at least 10 hours overnight, before requiring a recharge. You can certainly deviate here if you plan on deploying your system with a connection to the power grid, as there are many lights and fixtures available for this option (see the companion site for a listing on suitable alternative lights).

But for our purposes, we'll use the hardware discussed in this section.

We'll start with the **IFP-1 Infrared Flood Pro** light by Lucent (Figure 13.1, item #1; <http://pyimg.co/ft7gf>). This light will be paired with the official Raspberry Pi **NoIR camera V2** camera (Figure 13.1, item #2; <http://pyimg.co/l1hpi>). You'll likely find that the standard PiCamera ribbon cable is not long enough. Go ahead and purchase an **18inch PiCamera ribbon cable** by Adafruit or an alternative supplier (Figure 13.1, item #3; <http://pyimg.co/l1cfp>). I *would not* suggest a cable longer than 18 inches, due to voltage drop and signal degradation.

The Infrared Flood Pro runs on a standard 9V battery. It should run for 6-8 hours according to their FAQs page. That won't cut it for us. So we're going to need a larger 9V battery (Figure 13.1, item #4), a battery cable, and to make this project more fun we'll need a way to turn the power on and off electronically. To keep things simple, we'll use a *separate* battery for the Raspberry Pi itself (Figure 13.1, item #5). This battery pack can be used for other RPi projects, or even to charge your smartphone.

How can we turn our light source on and off electronically?

The Raspberry Pi digital General Purpose Output pins output 3.3V, which sources hardly any current. Therefore we need a way to use the 3.3V logic with a higher voltage/current light. Meet relays and MOSFETs. For this project we're going to use a MOSFET (Figure 13.1, item #6) which takes in a signal and power and outputs that power when the signal is high which the RPi is perfectly capable of. For power, the hardware requirements consist of:

- **5V, 5.8A, 22Ah, Portable USB Charger** by RavPower: <http://pyimg.co/d2ih7>
- **9.6V, 2Ah, Rechargeable battery** by Panasonic: <http://pyimg.co/qnk8o>
- **9V battery connector dongle 12-inch** by MPD (or equivalent manufacturer): <http://pyimg.co/egxj6>

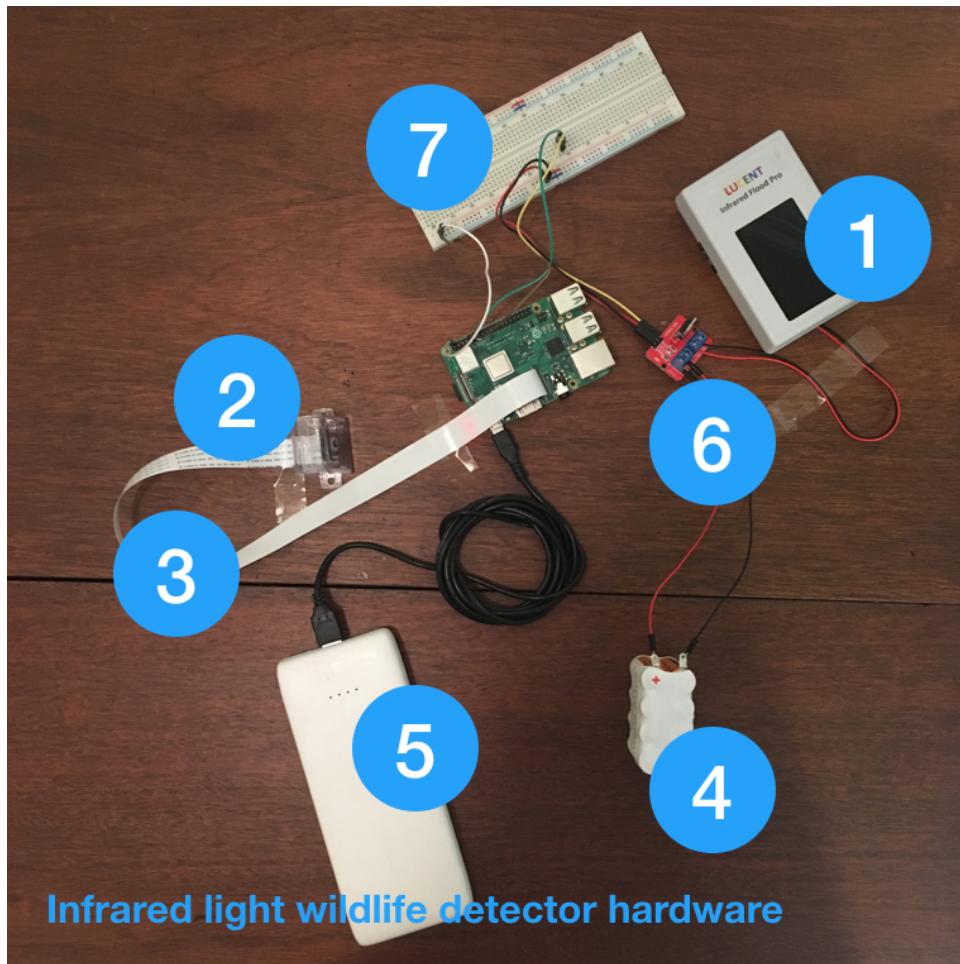


Figure 13.1: Electronic hardware for our deployable nocturnal wildlife monitoring box includes an infrared light (#1), infrared camera (#2), 18-inch ribbon cable (#3), 9.6V 2Ah rechargeable battery (#4), 5V 5.8A 22Ah portable USB power bank (#5), MOSFET module (#6), breadboard (#7), and Raspberry Pi (center).

- **MOSFET + Driver Module** by WINGONEER: <http://pyimg.co/eiyog>
- **Jumper wires** by Dupont (or equivalent manufacturer): <http://pyimg.co/azyyt>
- **Smart Universal Charger for NiMH/NiCD Battery Packs: 7.2v-12v** by Tenergy: <http://pyimg.co/b0fhe>

Finally, we need a waterproof box to seal all of our electronics in. Depending on what environmental conditions you expect your system to be in, you could spend a lot or a little in this area. I deployed mine in the backyard and put it to the test in the rain. A cheap liquid container from Wal-Mart that I already had on hand worked perfectly in these conditions. For your enclosure, you can use a plastic container such as a **Sterilite 1 Gal Pitcher** (plastic) sold by WalMart (Figure 13.3). I ran the ribbon cable through the liquid dispensing hole in the top and it was plenty waterproof!

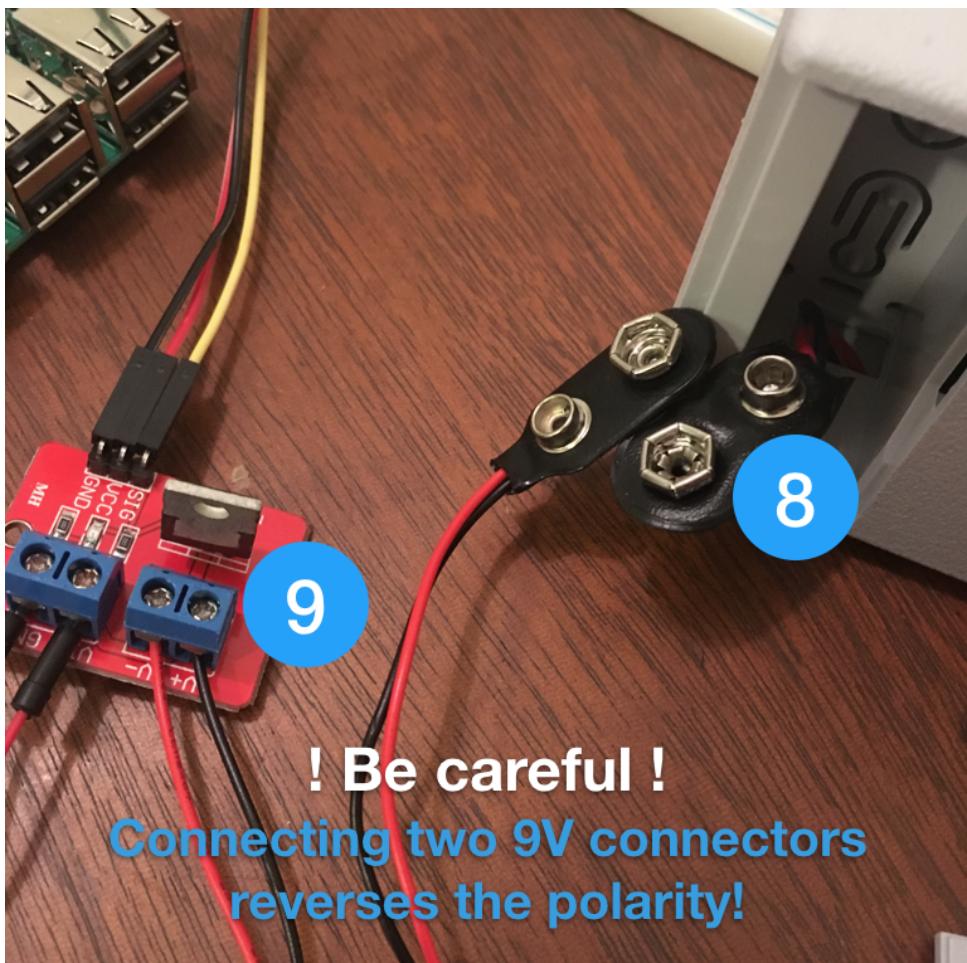


Figure 13.2: Be sure to connect the wires with proper polarity (it is reversed) when you connect the light to the MOSFET. Two 9V adapters reverses the polarity. Refer to Step #1 for more details.

**Remark.** Parts are subject to availability and can be discontinued at any given time. Use your creativity and favorite search engine. The companion site contains a full, up-to-date parts list for compatibility. You can find a link to create your account and access the companion website in the first few pages of this book.

### 13.3.2 Assembling your Wildlife Detector Box

Budget about one hour to assemble and test your wildlife box. Assembly requires **6 Steps**.

**Step #1:** Connect the battery connector dongle to the IR Flood Pro (Figure 13.2 item #8) and the bare wires to the MOSFET output (Figure 13.2 item #9). **Ensure that the polarity is correct.** Since we are using the 9V adapter connected to another one, the polarity becomes swapped. Therefore, **black** will be **positive or V+** instead of negative and **red** will be **negative or V-** instead of positive. Refer to Figure 13.2.



Figure 13.3: A cheap 1-gallon liquid pitcher can serve as your enclosure for less than \$4USD, available at WalMart or similar box stores in the USA. (<http://pyimg.co/bij9r>)

**Step #2:** Connect the rechargeable battery (Figure 13.1 item #5), to the MOSFET (Figure 13.1 item #6). You can use the spare connector that comes with your Tenergy battery charger if you'd like. Or just get the job done using some spare wire. Soldering will make for a better connection, but it isn't required if you clamp the leads with pliers.

**Step #3:** Connect the MOSFET control to the Raspberry Pi GPIO. On my Raspberry Pi 3B+, I connected the signal wire to PIN 7 (corresponds to GPIO 4), power wire to 5V PIN 2 or PIN 4 (note, the WINGONEER MOSFET will work with 3.3V as well, so PIN 1 is an option), and the ground wire can go to any available ground such as PIN 6 or PIN 9. I used a breadboard, but it isn't necessary if you have jumper wires that are the right gender.

**Step #4:** Ensure that your NoIR PiCamera V2 (Figure 13.1 item #2) is plugged into your Raspberry Pi. I recommend an 18 inch ribbon cable for this project (Figure 13.1 item #3).

**Step #5:** Power your Raspberry Pi with the RavPower battery pack (Figure 13.1 item #5) or another brand with similar specs.



Figure 13.4: Carefully fit all electronic components and wires inside your waterproof enclosure. Route the camera ribbon and light power lines through the lid of the container.

**Step #6:** Fit everything inside the plastic enclosure carefully (Figure 13.4). The only things outside the box should be the (1) IR Flood Pro, and (2) NoIR camera. I used a lot of packing tape to ensure that my box wouldn't fill up with rain water. I taped the IR Flood pro to the side of the plastic container, and ran the NoIR camera + ribbon cable up a short stick. I'd recommend a paint stick or yardstick, but since I didn't have one on hand, I used a wooden stirring utensil from the kitchen.

I didn't do anything to protect the NoIR camera from water. It definitely got rained on and still worked the next day but this was risky. What I would recommend is making a roof for it, but do not put glass/plastic in front of the sensor since IR light won't penetrate unless it is a special material.

Now you're ready to write some well crafted software!



Figure 13.5: The fully assembled and waterproof nocturnal wildlife monitoring system is battery powered so you can deploy it anywhere overnight. I used a lot of packing tape to hold the light to the front of the box. More tape was used to mount the household stirring stick to the back of the box to hold the camera.

## 13.4 Software: Writing the Remote Wildlife Detector

The software for this project is quite similar to the bird feeder monitor except that (1) we have light control, and (2) we'll only capture key video clips (no stills). In the first part of this section we'll discuss the project structure. From there we'll review our configuration file. And finally we'll develop our driver script and deploy our system in the wild.

### 13.4.1 Project Structure

Let's inspect our project structure:

---

```

|--- config
|   |-- config.json
|--- output
|--- |-- 20180928-032712.avi
|--- |-- ...
|--- pyimagesearch
|   |-- keyclipwriter
|   |   |-- __init__.py
|   |   |-- keyclipwriter.py
|   |-- utils
|   |   |-- __init__.py
|   |   |-- conf.py
|   |-- __init__.py
|--- 20180928-015847.avi
|--- ir_wildlife_monitor.py

```

---

Our configuration parameters are stored in `config.json`. We'll review the configuration in the next section.

Output videos (i.e. animal or other motion key clips) will be stored in the `output/` directory.

The `pyimagesearch` module contains our `KeyClipWriter` and `Conf` classes.

For tuning your motion detector parameters, one demo video (`20180928-015847.avi`) is included.

Our computer vision application which takes care of turning on the IR light, looping over a video stream, and capturing motion key clips is contained within `ir_wildlife_monitor.py`.

### 13.4.2 Our Configuration File

Let's inspect our configuration, `config.json`:

---

```

1  {
2      // flag indicating if pi camera should be used
3      "picamera": true,
4
5      // GPIO IR light pin number
6      "IR_GPIO_pin": 17,
7
8      // delay before light and loop start
9      "deployment_seconds": 120,
10
11     // flag indicating if post processing should be done or not
12     // and path to the input video (if post processing)
13     // NOTE: set "post_process" and "display" to false if
14     // deploying in the field
15     "post_process": true,

```

---

```

16     "input_video_path": "20180928-015847.avi",
17
18     // boolean variable used to indicate if frames must be displayed
19     // default is false
20     "display": false,

```

---

You should definitely use the PiCamera (**Line 3**), unless you have a better quality USB IR camera.

The MOSFET will be connected on pin 17 (**Line 6**). You can use a different GPIO output pin if you are using pin 17 for a different purpose.

I allowed myself exactly 2 minutes for deployment (**Line 9**).

If you'll post-process from a pre-recorded video file, set "post\_process" to true and ensure you have a valid video file path (**Lines 15 and 16**). You'll only need the display option if you are post-processing a video (**Line 20**).

We have a few more parameters to setup:

---

```

22     // path to output directory
23     "output_videos_path": "output",
24
25     // fps of output video
26     "output_fps": 32,
27
28     // codec of output video
29     "codec": "MJPEG",
30
31     // buffer size of key clip writer
32     "buffer_size": 64,
33
34     // minimum area for motion event
35     "min_area": 10000,
36
37     // MOG background subtraction parameters
38     "mog_history": 500,
39     "mog_nmixtures": 5,
40     "mog_bg_ratio": 0.7,
41     "mog_noise_sigma": 0
42 }

```

---

Videos will be outputted to the path specified on **Line 23** with the defined frame rate and codec (**Lines 26-29**).

Our keyclip writer buffer is set to 64 frames on **Line 32**.

The remaining settings are for our MOG background subtraction (**Lines 35-41**).

### 13.4.3 Our Wildlife Monitor Driver Script

Go ahead and open a new file called `ir_wildlife_monitor.py` and insert the following code:

---

```

1 # import the necessary packages
2 from pyimagesearch.keyclipwriter import KeyClipWriter
3 from pyimagesearch.utils import Conf
4 from imutils.video import VideoStream
5 import numpy as np
6 import argparse
7 import datetime
8 import imutils
9 import time
10 import cv2
11
12 # construct the argument parser and parse the arguments
13 ap = argparse.ArgumentParser()
14 ap.add_argument("-c", "--conf", required=True,
15     help="path to configuration file")
16 args = vars(ap.parse_args())
17
18 # load the configuration file
19 conf = Conf(args["conf"])

```

---

**Lines 2-10** import our required packages. Nothing is out of the ordinary here — this project is very similar to the bird monitor project (Chapter 9), although *there is one additional import which we'll review in our next code block*.

**Lines 13-19** parse the command line argument path to our config file and load it so we can access the parameters.

Let's set up our video stream and **turn on the infrared light**:

---

```

21 # if we are not post processing, we'll work with the NoIR PiCamera
22 if not conf["post_process"]:
23     # start the NoIR PiCamera video stream, allow the camera sensor to
24     # warmup, and import GPIO to turn the light on and off
25     print("[INFO] starting video stream...")
26     vs = VideoStream(usePiCamera=conf["picamera"]).start()
27     time.sleep(2.0)
28     import RPi.GPIO as GPIO
29
30     # set GPIO parameters for IR light
31     GPIO.setmode(GPIO.BCM)
32     GPIO.setup(conf["IR_GPIO_pin"], GPIO.OUT, initial=GPIO.LOW)
33
34     # OPTIONAL give yourself x seconds to go set it up outside

```

---

---

```

35     # NOTE: run inside of *screen* since you will likely lose wifi
36     time.sleep(conf["deployment_seconds"])
37
38     # turn on the light
39     GPIO.output(conf["IR_GPIO_pin"], GPIO.HIGH)
40
41 # otherwise, grab a reference to the video file
42 else:
43     print("[INFO] opening video file...")
44     vs = cv2.VideoCapture(conf["input_video_path"])
45     time.sleep(2.0)

```

---

Our system has two modes: (1) normal with a NoIR camera and IR light and (2) post-processing with a video file.

If we are not post-processing (**Line 22**), we launch our video stream (**Line 26**). We then import `RPi.GPIO` as `GPIO`. This import will allow us to interact with the MOSFET switch which is connected to the I/O port on our RPI.

In order to turn our MOSFET on and off, we need to set the pin mode as `GPIO.BCM` and set it up as an output pin `GPIO.OUT` (**Lines 31 and 32**).

We then sleep for the period of time specified in the config ("deployment\_seconds") on **Line 36**. This will allow you to start everything up in your house, and then walk the wildlife monitor box out to the deployment location and set the bait — you could even delay until it is dark out.

**Remark.** *This would also be a good place to insert time of day logic either programmatically with Python or via cronjob (say you wanted the system to start at 8pm). I'll leave that as an exercise for you, the reader.*

Then we go ahead and turn on the light via the MOSFET (`GPIO.HIGH`) via **Line 39**.

Assuming you'd rather post-process a video that you collected with another means you must set "post\_process" to `false` in the config file. **Lines 42-45** will then load the video from the path contained in "input\_video\_path" for post-processing.

Now let's initialize our key clip writer and background subtractor:

---

```

47 # initialize key clip writer and the consecutive number of
48 # frames that have *not* contained any action
49 kcw = KeyClipWriter(bufSize=conf["buffer_size"])
50 consecFrames = 0
51
52 # initialize the MOG foreground background subtractor
53 fgbg = cv2.bgsegm.createBackgroundSubtractorMOG(
54     history=conf["mog_history"], nmixtures=conf["mog_nmixtures"],

```

---

```

55     backgroundRatio=conf["mog_bg_ratio"],
56     noiseSigma=conf["mog_noise_sigma"])
57
58 # create erosion and dilation kernels
59 eKernel = np.ones((7, 7),np.uint8)
60 dKernel = np.ones((3, 3),np.uint8)

```

---

Our key clip writer is initialized on **Line 49**. We briefly discussed this class in Chapter 9, but to recap, the `KeyClipWriter` class maintains a buffer of frames and then dumps the frames to disk if motion is detected, ensuring you have the frames that not only *include* the motion, but also the frames *leading up to the motion* as well.

Our `consecFrames` holds the number of non-motion activity frames and it is initialized along with

We'll use the MOG background subtractor for this chapter. For a brief review on background subtraction, refer to Chapter 9. **Lines 53-56** set up our background subtraction object with the necessary parameters directly from our config file. **Lines 59 and 60** then set our erosion and dilation kernels that we'll use for morphological operations on our background subtraction mask.

Let's begin looping over frames:

---

```

62 # try until an exception such as ctrl+c being pressed in the headless
63 # terminal, or the OS shutting down Python due to a lack of battery
64 # power
65 try:
66     # keep looping
67     while True:
68         # grab the current frame, resize it, and initialize a
69         # boolean used to indicate if the consecutive frames
70         # counter should be updated
71         frame = vs.read()
72         frame = frame[1] if conf["post_process"] else frame
73         frame = imutils.resize(frame, width=600)
74
75         # flag to update the counter consecutive frames with no motion
76         updateConsecFrames = True
77
78         # apply foreground background subtraction with the
79         # GMG algorithm
80         mask = fgbg.apply(frame)
81
82         # erode the mask to eliminate noise and then
83         # dilate the mask to fill in holes
84         mask = cv2.erode(mask, eKernel, iterations=3)
85         mask = cv2.dilate(mask, dKernel, iterations=2)

```

---

The `try/finally` block that begins on **Line 65** is a way of catching an exception and then cleaning up (turning off the IR light to save battery, finishing any in-progress key clips, and releasing video streams). I chose to implement `try/finally` as you never know what may go wrong in the wild — maybe your RPi's voltage gets low and the OS stops Python and powers down.

**Line 67** begins our frame processing loop and we grab our first frame on **Line 71**. We resize it to a known `width` of 600 while preserving the aspect ratio (**Line 73**).

We set a boolean indicating that we need to update our consecutive frame counter (**Line 76**).

Then we apply background subtraction with morphological operations (**Lines 80-85**).

Let's find and process contours in the mask:

---

```

87     # find contours
88     cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
89         cv2.CHAIN_APPROX_SIMPLE)
90     cnts = imutils.grab_countours(cnts)
91
92     # only proceed if at least one contour was found
93     if len(cnts) > 0:
94         # find the largest contour in the mask, then use it
95         # to compute the minimum enclosing circle
96         c = max(cnts, key=cv2.contourArea)
97         (x, y, w, h) = cv2.boundingRect(c)
98         updateConsecFrames = (w * h) <= conf["min_area"]
99
100    # only proceed if the area meets a minimum size
101    if updateConsecFrames:
102        # reset the number of consecutive frames with
103        # *no* action to zero and draw the circle
104        # surrounding the object
105        consecFrames = 0
106        green = (0, 255, 0)
107        cv2.rectangle(frame, (x, y), (x + w, y + h), green, 2)
108
109        # if we are not already recording, start recording
110        if not kcw.recording:
111            timestamp = datetime.datetime.now()
112            p = "{}.avi".format(conf["output_videos_path"],
113                timestamp.strftime("%Y%m%d-%H%M%S"))
114            kcw.start(p, cv2.VideoWriter_fourcc(
115                *conf["codec"]), conf["output_fps"])

```

---

Contours are grabbed (**Lines 88-90**). Assuming we find at least one (**Line 93**), we'll begin to process the contours. First we grab the contour with the largest area and determine if it meets or exceeds the "min\_area" (**Lines 96-98**).

If so, we reset our `consecFrames` counter, draw a green box around the motion, and begin recording (**Lines 105-115**).

Otherwise, we either (1) didn't find motion or (2) are finished processing contours:

---

```

117      # otherwise, no action has taken place in this frame, so
118      # increment the number of consecutive frames that contain
119      # no action
120      if updateConsecFrames:
121          consecFrames += 1
122
123      # update the key frame clip buffer
124      kcw.update(frame)
125
126      # if we are recording and reached a threshold on consecutive
127      # number of frames with no action, stop recording the clip
128      if kcw.recording and consecFrames == conf["buffer_size"]:
129          kcw.finish()

```

---

We increment our `consecFrames` if necessary an **Lines 120 and 121**.

Then we update our key clip writer object (**Line 124**). If the consecutive non-motion frames exceeds the key clip writer buffer, we finish recording (**Lines 128 and 129**).

Let's display our frame if necessary:

---

```

131      # if the display option is set
132      if conf["display"]:
133          # show the frame
134          cv2.imshow("Frame", frame)
135          key = cv2.waitKey(1) & 0xFF
136
137          # if the `q` key was pressed, break from the loop
138          if key == ord("q"):
139              break

```

---

We optionally display our frame while watching for a "q" (quit) keypress (**Lines 132-139**).

From there, if we experienced an exception for any reason, we'll perform cleanup and exit:

---

```

141      # if an exception/error occurred, take these final steps
142      finally:
143          # show a message to the user
144          print("[INFO] cleaning up...")
145
146          # if we are in the middle of recording a clip, wrap it up
147          if kcw.recording:

```

```

148         kcw.finish()
149
150     # if we are live (not post processing) turn off the IR light to
151     # save battery and stop the video stream
152     if not conf["post_process"]:
153         GPIO.output(conf["IR_GPIO_pin"], GPIO.LOW)
154         GPIO.cleanup()
155         vs.stop()
156
157     # otherwise, release the video file pointer
158     else:
159         vs.release()
160
161     # if the display option is set, close all windows opened by OpenCV
162     if conf["display"]:
163         cv2.destroyAllWindows()

```

---

The code inside our `finally` block, beginning on **Line 142**, will be executed if there's an exception in the `try` block. It could be a "ctrl + c" interrupt, an RPi shutdown signal due to low battery, or any other error condition. Cleanup steps include:

- Finishing recording (**Lines 147 and 148**).
- Turning off the IR light and cleaning up GPIO (**Lines 152-154**).
- Releasing the video stream (**Line 155 or 159**).
- Destroying GUI windows if necessary (**Lines 162-163**).

## 13.5 Deploying Our Wildlife Monitor

Be sure to review section [13.3.2](#) for assembly instructions on your deployable nocturnal wildlife box. Once your box is assembled, power it on and transfer the files for the project.

Next, you should edit the configuration file. Most notably you should edit your `"deployment_seconds"` variable. If you know that sunset is 7:30pm and it is 5:00pm now (a difference of 5400 seconds), then set the value to 5400 so that your IR light will come on at 7:30, but your system can sleep until then, consuming very little power.

You could also set deployment seconds to zero and "start the script on boot" (see Chapter [8](#)) or you can use the `screen` command.

The command to execute the IR wildlife monitor is simply:

---

```
1 $ python ir_wildlife_monitor.py --config config/config.json
```

---

### 13.5.1 Results



Figure 13.6: **Left:** Our RPi is monitoring an area outside of our house. Currently no motion is detected. **Middle:** A raccoon enters the scene. Our background subtractor picks up the motion and draws a bounding box surrounding the raccoon. **Right:** After feeding, the raccoon heads back into the wild.

Our PylImageSearcher, David McDuffee, has captured some fantastic clips of raccoons near his house. Figure 13.6 contains few still frames extracted from the video clip.

On the *left* we can see an empty scene. Our Raspberry Pi is monitoring for motion. In the *middle*, a raccoon enters the view of the camera and starts feeding. Our background subtractor is able to detect the motion and draw a bounding box, surrounding the raccoon. After feeding, the raccoon heads back into the wild (*right*).

### 13.5.2 Tips and Suggestions

When working on this project, we learned the following points to keep in mind:

- **Remember to test your light functionality.** Ensure that the light is able to turn on and off and that nighttime pictures with the NoIR camera look good with how the camera and light are aimed. You can write a short script to blink the light on and off with a delay for testing purposes.
- **Be creative with the hardware.** You don't need to use my exact hardware list. I encourage you to find cheaper and/or better alternatives.
- **Keep things simple for your first deployments.** You could add all sorts of features to the code to make the light come on and off based on time of day. For example, a second "white light" could turn on to get a standard video of the wildlife (in addition to an IR video). The key is to prove that you can get something on camera first.
- **Put bait out.** Trying to catch deer on camera? Figure out where the deer roam and put out some grain that they like (check with a garden or hunting store). Other animals like raccoons, opossums, foxes, and snakes will eat just about anything with a meaty smell.

Raw chicken, beef, tuna, or wet cat food are great options. Rabbits and other vegetarians will eat carrots and other vegetables.

**Make sure you educate yourself on local wildlife laws before using bait!** Some counties, cities, or states do not allow baiting – make sure you check with your local government first.

## 13.6 Summary

In this chapter we learned how to build a nocturnal wildlife detector. Our nocturnal wildlife monitoring system required both hardware and software engineering.

We learned about controlling lights with GPIO and a MOSFET connected to a power source and light. We tied our system together with a Python script that you can deploy again and again (assuming you recharge your batteries!)

Hardware projects are a lot of fun and I encourage you to try this one out. Be creative with the hardware to (1) save money like I did, or (2) engineer the ultimate system, sparing no expense.

Maybe you'll get really into it and you could implement a small solar cell to charge the batteries during the daytime so they are ready to go at night. Maybe you'd like to add a cellular HAT, so your videos get uploaded to the cloud from a remote location.

Use your imagination and have fun!

## Chapter 14

# Video Surveillance and Web Streaming

I first started playing guitar twenty years ago when I was in middle school. I wasn't very good at it and gave it up only a couple years after. Looking back, I strongly believe the reason I didn't stick with it was because I wasn't learning in a practical, hands-on manner.

Instead, my music teacher kept trying to drill theory into my head — but as an eleven year old kid I was just trying to figure out whether I even *liked* playing guitar, let alone if I wanted to *study the theory* behind music in general.

About a year and a half ago, I decided to start taking guitar lessons again. This time I took care to find a teacher who could blend theory and practice together, showing me how to play songs or riffs while at the same time learning a particular theoretical technique.

The result? My finger speed is now faster than ever, my rhythm is on point, and I can annoy my wife to no end rocking "*Sweet Child of Mine*" on my Les Paul.

My point is this — whenever you are learning a new skill, whether it's computer vision, hacking with the Raspberry Pi, or even playing guitar, one of the fastest, fool-proof methods to pick up the technique is to design (small) real-world projects around the skill and try to solve it.

For guitar, that meant learning short riffs that not only taught me *parts of actual songs*, but gave me a *valuable technique* (such as mastering a particular pentatonic scale, for instance).

In computer vision and image processing, your goal should be to brainstorm mini-projects and then try to solve them. Don't get too complicated too quickly, that's a recipe for failure.

Instead, pay attention to the chapters in this book and note which chapters interest you the most. Then, when you're done reading, go back to those chapters and see how you can extend them in some manner, *even if it's just applying the same technique to a different dataset*.

Solving new mini-projects you brainstorm will not only keep you interested in the subject (since you personally thought of them), but they'll teach you hands-on skills at the same time.

The subject of this chapter — motion detection and streaming to a web browser — is a great starting point for such a mini-project.

**While going through this chapter, brainstorm ideas of how you may extend this project to your own applications.** Keep a list of these projects and refer to them once you are finished the book.

## 14.1 Chapter Learning Objectives

In this chapter you will:

- Learn the fundamentals of *motion detection*
- Implement motion detection by means of a *background subtractor*
- Discover the Flask web framework
- Combine Flask with OpenCV
- Stream frames from your RPi to a web browser

We'll then put all these pieces together and build a home surveillance system capable of performing motion detection, and stream the result to your web browser.

## 14.2 Project Structure

Before we get started, let's take a look at our directory structure for the project:

---

```
|-- pyimagesearch
|   |-- __init__.py
|   |-- motion_detection
|       |-- __init__.py
|       |-- singlemotiondetector.py
|-- templates
|   |-- index.html
|-- webstreaming.py
```

---

To perform background subtraction and motion detection, we'll be implementing a class named `SingleMotionDetector` — this class will live inside the `singlemotiondetector.py` file inside the `motion_detection` submodule of `pyimagesearch`.

The `webstream.py` file will use OpenCV to access our web camera, perform motion detection via `SingleMotionDetector`, and then serve the output frames to our web browser via the Flask web framework.

In order for our web browser to have something to display, we need to populate the contents of `index.html` with HTML used to serve the video feed. We'll only need to insert some basic HTML markup — Flask will handle actually sending the video stream to our browser for us.

## 14.3 Implementing a Basic Motion Detector

Our motion detector algorithm will detect motion via ***background subtraction***. Most background subtraction algorithms work by:

- i. Accumulating the *weighted average* of the previous  $N$  frames
- ii. Taking the *current frame* and subtracting it from the weighted average of frames
- iii. Thresholding the output of the subtraction to highlight the regions with substantial differences in pixel values (“white” for foreground and “black” for foreground)
- iv. Applying basic image processing techniques such as erosions and dilations to remove noise
- v. Utilizing contour detection to extract the regions containing motion

Our motion detection implementation will live inside the `SingleMotionDetector` class which can be found in `singlemotiondetector.py`.

We call this a “single motion detector” as the algorithm itself is only interested in finding the *single, largest* region of motion. We can easily extend this method to handle multiple regions of motion as well.

Let's go ahead and implement the motion detector. Open up the `singlemotiondetector.py` file and insert the following code:

---

```
1 # import the necessary packages
2 import numpy as np
3 import imutils
4 import cv2
5
6 class SingleMotionDetector:
7     def __init__(self, accumWeight=0.5):
8         # store the accumulated weight factor
9         self.accumWeight = accumWeight
10
11     # initialize the background model
12     self.bg = None
```

---

**Lines 2-4** handle our required imports. All of these are fairly standard, including NumPy for numerical processing, imutils for our convenience functions, and cv2 for our OpenCV bindings.

We then define our SingleMotionDetector class on **Line 6**. The class accepts an optional argument, accumWeight, which is the factor used to our accumulated weight average.

The *larger* accumWeight is, the *less* the background (`bg`) will be factored in when accumulating the weighted average. Conversely, the *smaller* accumWeight is, the *more* the background `bg` will be considered when computing the average. Setting `accumWeight=0.5` weights the background and foreground equally — I often recommend this as a starting point value (you can then adjust it based on your own experiments).

Next, let's define the `update` method, which will accept an input frame and compute the weighted average:

---

```

14     def update(self, image):
15         # if the background model is None, initialize it
16         if self.bg is None:
17             self.bg = image.copy().astype("float")
18             return
19
20         # update the background model by accumulating the weighted
21         # average
22         cv2.accumulateWeighted(image, self.bg, self.accumWeight)

```

---

In the case that our `bg` frame is `None` (implying that `update` has never been called), we simply store the `bg` frame (**Lines 15-18**). Otherwise, we compute the weighted average between the input `frame`, the existing background `bg`, and our corresponding `accumWeight` factor.

Given our background `bg` we can now apply motion detection via the `detect` method:

---

```

24     def detect(self, image, tVal=25):
25         # compute the absolute difference between the background model
26         # and the image passed in, then threshold the delta image
27         delta = cv2.absdiff(self.bg.astype("uint8"), image)
28         thresh = cv2.threshold(delta, tVal, 255, cv2.THRESH_BINARY) [1]
29
30         # perform a series of erosions and dilations to remove small
31         # blobs
32         thresh = cv2.erode(thresh, None, iterations=2)
33         thresh = cv2.dilate(thresh, None, iterations=2)

```

---

The `detect` method requires a single parameter along with an optional one:

- `image`: The input frame/image that motion detection will be applied to
- `tVal`: The threshold value used to mark a particular pixel as “motion” or not

Given our input `image` we compute the absolute difference between the `image` and the `bg` (**Line 27**). Any pixel locations that have a difference  $> tVal$  are set to 255 (white; foreground), otherwise they are set to 0 (black; background) (**Line 28**).

A series of erosions and dilations are performed to remove noise and small, localized areas of motion that would otherwise be considered false-positives (likely due to reflections or rapid changes in light).

The next step is to apply contour detection to extract any motion regions:

---

```

35      # find contours in the thresholded image and initialize the
36      # minimum and maximum bounding box regions for motion
37      cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
38          cv2.CHAIN_APPROX_SIMPLE)
39      cnts = imutils.grab_contours(cnts)
40      (minX, minY) = (np.inf, np.inf)
41      (maxX, maxY) = (-np.inf, -np.inf)

```

---

**Lines 37-38** perform contour detection on our `thresh` image. We then initialize two sets of bookkeeping variables to keep track of the location where any motion is contained. These variables will form the “bounding box”, which will tell us the where the motion in the frame is taking place.

The final step is to populate these variables (provided motion exists in the frame, of course):

---

```

42      # if no contours were found, return None
43      if len(cnts) == 0:
44          return None
45
46      # otherwise, loop over the contours
47      for c in cnts:
48          # compute the bounding box of the contour and use it to
49          # update the minimum and maximum bounding box regions
50          (x, y, w, h) = cv2.boundingRect(c)
51          (minX, minY) = (min(minX, x), min(minY, y))
52          (maxX, maxY) = (max(maxX, x + w), max(maxY, y + h))
53
54      # otherwise, return a tuple of the thresholded image along
55      # with bounding box
56      return (thresh, (minX, minY, maxX, maxY))

```

---

On **Lines 43-45** we make a check to see if our contours list is empty. If that’s the case, then there was no motion found in the frame, and we can safely ignore it.

Otherwise, motion *does* exist in the frame, so we need to start looping over the contours (**Line 48**).

For each contour we compute the bounding box and then update our bookkeeping variables (**Lines 47-53**), finding the minimum and maximum  $(x, y)$ -coordinates that *all* motion has taken place it.

Finally, we return the bounding box location to the calling function.

## 14.4 Sending Frames from an RPi to Our Web Browser

In the first part of this section, we'll briefly discuss the Flask web framework and how to install it on your system. From there, you will learn how to combine Flask with OpenCV, enabling you to:

- i. Access frames from RPi camera module or USB webcam
- ii. Process the frames and apply an arbitrary algorithm (here we'll be using background subtraction)
- iii. Stream the results to a web page/web browser

Additionally, the code we'll be covering will be able to support *multiple clients* (i.e., more than one person/web browser/tab accessing the stream at once), something the vast majority of examples you will find online cannot handle.

### 14.4.1 The Flask Web Framework

Flask is a micro web framework written in the Python programming language [51]. Along with Django [52], Flask is one of the *most common* web frameworks you'll see when building web applications using Python. However, unlike Django, Flask is very lightweight, making it *super easy* to build basic web applications.

As we'll see in this section, we'll only need a small amount of code to facilitate live video streaming with Flask — the rest of the code either involves (1) OpenCV and accessing our video stream or (2) ensuring our code is thread safe and can handle multiple clients.

If you are using the Raspbian .img file included with this book, you'll find that Flask is *already installed* for you. However, if you ever need to install Flask on a machine, it's as simple as the following command:

---

```
$ pip install flask
```

---

Provided the above command executes successfully, the Flask web framework is now installed on your system.

#### 14.4.2 Combining OpenCV with Flask

Let's go ahead and combine OpenCV with Flask to serve up frames from a video stream (running on a Raspberry Pi) to a web browser. Open up the `webstreaming.py` file in your project structure and insert the following code:

---

```
1 # import the necessary packages
2 from pyimagesearch.motion_detection import SingleMotionDetector
3 from imutils.video import VideoStream
4 from flask import Response
5 from flask import Flask
6 from flask import render_template
7 import threading
8 import argparse
9 import datetime
10 import imutils
11 import time
12 import cv2
```

---

**Lines 2-12** handle our required imports. **Line 2** imports our `SingleMotionDetector` class which we implemented in Section 14.3. The `VideoStream` class will enable us to access our Raspberry Pi camera module or USB webcam.

**Lines 4-6** handle importing our required Flask packages — we'll be using these packages to render our `index.html` template and serve it up to clients.

**Line 7** imports the `threading` library to ensure we can support concurrency (i.e., multiple clients, web browsers, and tabs at the same time).

Let's move on to performing a few initializations:

---

```
14 # initialize the output frame and a lock used to ensure thread-safe
15 # exchanges of the output frames (useful for multiple browsers/tabs
16 # are viewing the stream)
17 outputFrame = None
18 lock = threading.Lock()
19
20 # initialize a flask object
21 app = Flask(__name__)
22
23 # initialize the video stream and allow the camera sensor to
24 # warmup
25 #vs = VideoStream(usePiCamera=1).start()
```

---

```
26 vs = VideoStream(src=0).start()
27 time.sleep(2.0)
```

---

First, we initialize our `outputFrame` on **Line 17** — this will be the frame (post-motion detection) that will be served to the clients.

We then create a `lock` on **Line 18** which will be used to ensure thread-safe behavior when updating the `outputFrame` (i.e., ensuring that one thread isn't trying to read the frame as it is being updated).

**Line 21** initializes our Flask app itself while **Lines 25-27** access our video stream.

The next function, `index`, will render our `index.html` template and serve up the output video stream:

---

```
29 @app.route("/")
30 def index():
31     # return the rendered template
32     return render_template("index.html")
```

---

This function is quite simplistic — all it's doing is calling the Flask `render_template` on our HTML file. We'll be reviewing the `index.html` file in the next section so we'll hold off on a further discussion on the file contents until then.

Our next function, `detect_motion`, is responsible for:

- i. Looping over frames from our video stream
- ii. Applying motion detection
- iii. Drawing any results on the `outputFrame`

Furthermore, this function must perform all of these operations in a thread safe manner to ensure concurrency is supported.

Let's take a look at this function now:

---

```
34 def detect_motion(frameCount):
35     # grab global references to the video stream, output frame, and
36     # lock variables
37     global vs, outputFrame, lock
38
39     # initialize the motion detector and the total number of frames
40     # read thus far
41     md = SingleMotionDetector(accumWeight=0.1)
42     total = 0
```

---

Our `detection_motion` function accepts a single argument, `frameCount`, which is the *minimum* number of required frames to build our background `bg` in the `SingleMotionDetector` class: If we don't have at least a total of `frameCount` frames, we'll continue to compute the accumulated weighted average. Once `frameCount` is reached, we'll start performing background subtraction.

**Line 37** grabs global references to three variables:

- `vs`: Our instantiated `VideoStream` object
- `outputFrame`: The output frame that will be served to clients
- `lock`: The thread lock that we must obtain before updating `outputFrame`

**Line 41** initializes our `SingleMotionDetector` class with a value of `accumWeight=0.1`, implying that the `bg` value will be weighted higher when computing the weighted average.

**Lines 42** then initializes the `total` number of frames read thus far — we'll need to ensure a sufficient number of frames have been read to build our background model. From there, we'll be able to perform background subtraction.

With these initializations complete, we can now start looping over frames from the camera:

---

```

44     # loop over frames from the video stream
45     while True:
46         # read the next frame from the video stream, resize it,
47         # convert the frame to grayscale, and blur it
48         frame = vs.read()
49         frame = imutils.resize(frame, width=400)
50         gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
51         gray = cv2.GaussianBlur(gray, (7, 7), 0)
52
53         # grab the current timestamp and draw it on the frame
54         timestamp = datetime.datetime.now()
55         cv2.putText(frame, timestamp.strftime(
56             "%A %d %B %Y %I:%M:%S%p"), (10, frame.shape[0] - 10),
57             cv2.FONT_HERSHEY_SIMPLEX, 0.35, (0, 0, 255), 1)

```

---

**Line 48** reads the next frame from our camera while **Lines 49-51** perform preprocessing, including:

- Resizing to have a width of 400px (the *smaller* our input frame is, the *less* data there is, and thus the *faster* our algorithms will run).
- Converting to grayscale.
- Gaussian blurring (to reduce noise).

We then grab the current timestamp and draw it on the `frame` (**Lines 54-57**).

With one final check, we can perform motion detection:

---

```

59     # if the total number of frames has reached a sufficient
60     # number to construct a reasonable background model, then
61     # continue to process the frame
62     if total > frameCount:
63         # detect motion in the image
64         motion = md.detect(gray)

65
66         # check to see if motion was found in the frame
67         if motion is not None:
68             # unpack the tuple and draw the box surrounding the
69             # "motion area" on the output frame
70             (thresh, (minX, minY, maxX, maxY)) = motion
71             cv2.rectangle(frame, (minX, minY), (maxX, maxY),
72                           (0, 0, 255), 2)

73
74         # update the background model and increment the total number
75         # of frames read thus far
76         md.update(gray)
77         total += 1

78
79         # acquire the lock, set the output frame, and release the
80         # lock
81         with lock:
82             outputFrame = frame.copy()

```

---

On **Line 62** we ensure that we have read at least `frameCount` frames to build our background subtraction model. If so, we apply the `.detect` motion of our motion detector, which returns a single variable, `motion`.

If `motion` is `None`, then we know no motion has taken place in the current `frame`. Otherwise, if `motion` is *not* `None` (**Line 67**), then we need to draw the bounding box coordinates of the motion region on the `frame`.

**Line 76** updates our motion detection background model while **Line 77** increments the total number of frames read from the camera thus far.

Finally, **Lines 81 and 82** acquire the `lock` (required to support thread concurrency) and sets the `outputFrame`. We need to acquire the `lock` to ensure the `outputFrame` variable is not accidentally being *read* by a client while we are trying to *update* it.

Our next function, `generate`, is a Python generator used to encode our `outputFrame` as JPEG data — let's take a look at it now:

---

```

84     def generate():

```

---

```

85      # grab global references to the output frame and lock variables
86      global outputFrame, lock
87
88      # loop over frames from the output stream
89      while True:
90          # wait until the lock is acquired
91          with lock:
92              # check if the output frame is available, otherwise skip
93              # the iteration of the loop
94              if outputFrame is None:
95                  continue
96
97              # encode the frame in JPEG format
98              (flag, encodedImage) = cv2.imencode(".jpg", outputFrame)
99
100             # ensure the frame was successfully encoded
101             if not flag:
102                 continue
103
104             # yield the output frame in the byte format
105             yield(b"--frame\r\n" b'Content-Type: image/jpeg\r\n\r\n' +
106             bytearray(encodedImage) + b'\r\n')

```

---

**Line 86** grabs global references to our `outputFrame` and `lock`, similar to the `detect_motion` function.

The `generate` starts an infinite loop on **Line 89** that will continue until we kill the script. Inside the loop, we:

- First acquire the `lock` (**Line 91**).
- Ensure the `outputFrame` is not empty (**Line 94**), which may happen if a frame is dropped from the camera sensor.
- Encode the `frame` as a JPEG image on **Line 98** — JPEG compression is performed here to reduce load on the network and ensure faster transmission of frames.
- Check to see if the success `flag` has failed (**Lines 101 and 102**), implying that the JPEG compression failed and we should ignore the frame.
- Finally, serve the encoded JPEG frame as a byte array that can be consumed by a web browser.

That was quite a lot of work in a short amount of code, so definitely make sure you review this function a few times, to ensure you understand how it works.

The next function, `video_feed` calls our `generate` function:

---

```

108 @app.route("/video_feed")
109 def video_feed():
110     # return the response generated along with the specific media
111     # type (mime type)
112     return Response(generate(),
113                     mimetype = "multipart/x-mixed-replace; boundary=frame")

```

---

Notice how this function has an `app.route` decorator, just like the `index` function above. The `app.route` signature tells Flask that this function is a URL endpoint and that data is being served from `http://your_ip_address/video_feed`.

The output of `video_feed` is the live motion detection output, encoded as a byte array via the `generate` function. Your web browser is smart enough to take this byte array and display it in your browser as a live feed.

Our final code block handles parsing command line arguments and launching the Flask app:

---

```

115 # check to see if this is the main thread of execution
116 if __name__ == '__main__':
117     # construct the argument parser and parse command line arguments
118     ap = argparse.ArgumentParser()
119     ap.add_argument("-i", "--ip", type=str, required=True,
120                     help="ip address of the device")
121     ap.add_argument("-o", "--port", type=int, required=True,
122                     help="ephemeral port number of the server (1024 to 65535)")
123     ap.add_argument("-f", "--frame-count", type=int, default=32,
124                     help="# of frames used to construct the background model")
125     args = vars(ap.parse_args())
126
127     # start a thread that will perform motion detection
128     t = threading.Thread(target=detect_motion, args=(
129         args["frame_count"],))
130     t.daemon = True
131     t.start()
132
133     # start the flask app
134     app.run(host=args["ip"], port=args["port"], debug=True,
135             threaded=True, use_reloader=False)
136
137     # release the video stream pointer
138     vs.stop()

```

---

**Lines 118-125** handle parsing our command line arguments. We need three arguments here, including:

- i. `--ip`: The IP address of the system you are launching the `webstream.py` file from.

- ii. `--port`: The port number that the Flask app will run on (you'll typically supply a value of 8000 for this parameter).
- iii. `--frame-count`: The number of frames used to accumulate and build the background model before motion detection is performed.

**Lines 128-131** launches a thread that will be used to perform motion detection. Using a thread ensures the `detect_motion` function can safely run in the background — it will be constantly running and updating our `outputFrame` so we can serve any motion detection results to our clients.

Finally, **Lines 134 and 135** launches the Flask app itself.

#### 14.4.3 The HTML Page Structure

As we saw in `webstreaming.py`, we are rendering an HTML template named `index.html`. The template itself is populated by the Flask web framework and then served to the web browser. Your web browser then takes the generated HTML and renders it to your screen.

Let's inspect the contents of our `index.html` file:

---

```
1 <html>
2   <head>
3     <title>Pi Video Surveillance</title>
4   </head>
5   <body>
6     <h1>Pi Video Surveillance</h1>
7     
8   </body>
9 </html>
```

---

As we can see, this is a super basic web page; however, pay close attention to **Line 7** — notice how we are instructing Flask to *dynamically render* the URL of our `video_feed` route.

Since the `video_feed` function is responsible for serving up frames from our webcam, the `src` of the image will be automatically populated with our output frames. Our web browser is then smart enough to properly render the webpage and serve up the live video stream.

## 14.5 Putting the Pieces Together

Now that we've coded up our project, let's put it to the test. Open up a terminal and execute the following command:

---

```
$ python webstreaming.py --ip 0.0.0.0 --port 8000
* Serving Flask app "webstreaming" (lazy loading)
* Environment: production
WARNING: Do not use the development server in a production environment.
Use a production WSGI server instead.
* Debug mode: on
* Running on http://0.0.0.0:8000/ (Press CTRL+C to quit)
10.0.0.75 -- [26/Jul/2019 08:22:55] "GET / HTTP/1.1" 200 -
10.0.0.75 -- [26/Jul/2019 08:22:55] "GET /video_feed HTTP/1.1" 200 -
```

---

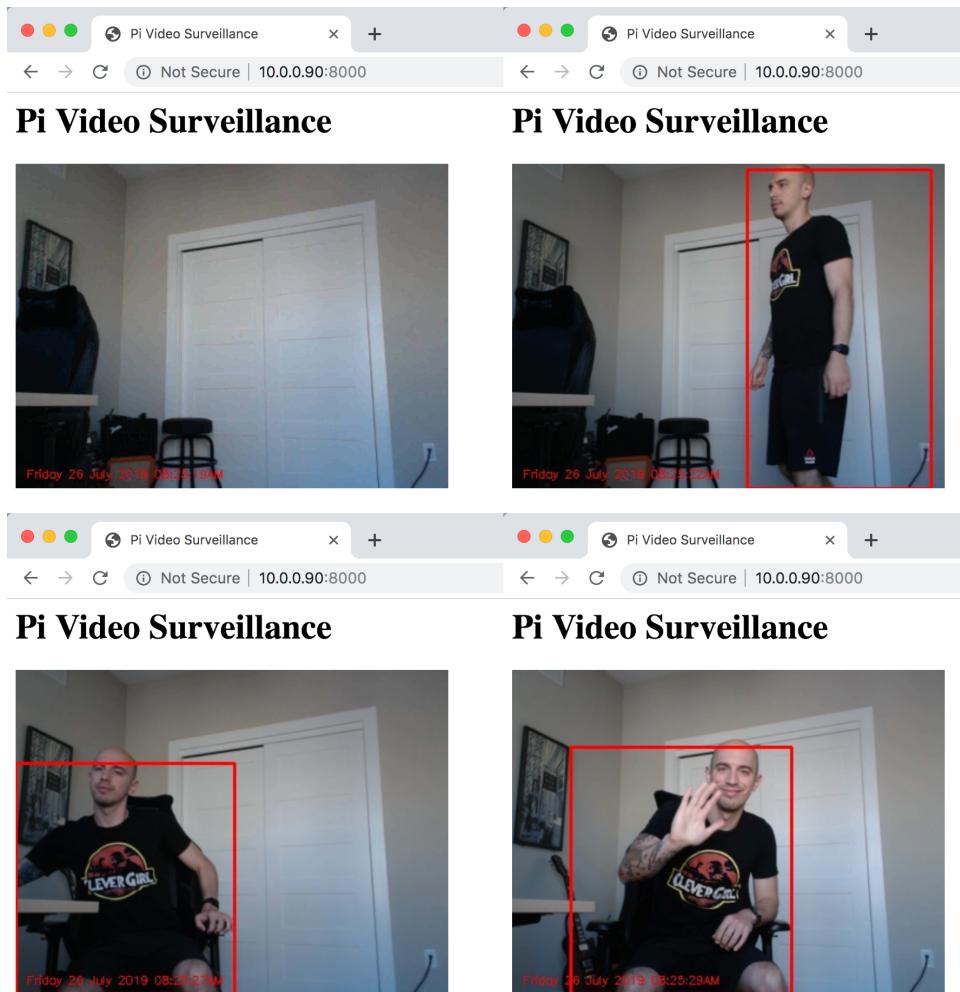


Figure 14.1: **Top-left:** A live stream from the webcam connected to our RPi is being streamed to our web browser. **All others:** Motion is detected in the scene and drawn on the output frame.

An example of our output can be seen in Figure 14.1. In the *top-left* we can see that the frames captured from our camera connected to our RPi are being streamed to our web browser. Then, when motion occurs, our RPi detects the area motion is contained in and draws the bounding box surrounding the area.

## 14.6 Summary

In this chapter you learned how to perform motion detection via a background subtraction algorithm. Background subtraction is an *extremely common* method utilized in computer vision. Typically, these algorithms are computationally efficient, making them suitable for resource constrained devices, such as the Raspberry Pi.

We'll be using background subtraction quite a bit in this text, so if you're feeling at all confused by it, definitely take the time to re-read this chapter before continuing.

After implementing our background subtractor, we combined it with the Flask web framework, enabling us to:

- i. Access frames from RPi camera module/USB webcam
- ii. Apply background subtraction/motion detection to each frame
- iii. Stream the results to a web page/web browser

Furthermore, our implementation **supports multiple clients, browsers, or tabs** — something that you will not find in most other implementations. Whenever you need to stream frames from a device to a web browser, definitely use this code as a template/starting point.

Finally, use this chapter as a starting point for your own mini-projects. One of the best ways to learn is to take an existing project and modify it slightly to achieve a certain goal.

My personal suggestion to you would be to take this project and then update it to perform motion detection on *only a specific region of the input frame*. I'll give you a hint – you should be using Region of Interest (ROI) cropping using NumPy array slicing.



## Chapter 15

# Using Multiple Cameras with the RPi

Have you ever wondered if you can use multiple cameras on a single Raspberry Pi? Perhaps you wish you could stitch multiple feeds into a single frame, and then process the frame to detect objects, people, or motion. Or, maybe you need multiple perspectives of a scene, enabling you to construct a depth map from stereo images.

The good news is that yes, the Raspberry Pi does indeed support multiple cameras.

In this chapter we're going to demonstrate working with two cameras connected to the same Raspberry Pi. These two cameras could be two USB cameras or a USB and PiCamera. There is even special hardware you can use to connect multiple PiCameras, but hardware like this tends to go off the market just as quickly as it came on the market. Special hardware will not be covered.

Our use case for this chapter will focus on panorama stitching. We'll stitch feeds together from two separate cameras and then detect pedestrians in the panorama.

### 15.1 Chapter Learning Objectives

We'll learn the following concepts in this chapter:

- i. Multiple cameras with OpenCV and the Raspberry Pi
- ii. Panorama stitching with OpenCV
- iii. Pedestrian detection

## 15.2 Multiple Cameras and the Raspberry Pi

In this section, we'll first create a template for working with multiple cameras using OpenCV. From there, we'll review our project structure and dive into the panorama stitcher by implementing a `Stitcher` class used to stitch together frames.

Next, we'll design the `PedestrianDetector` class using a Haar-based approach. Then we'll walk through our driver script, which utilizes both the `Stitcher` and `PedestrianDetector` classes. Finally, we'll execute our code and review some recommendations when working with multiple cameras.

### 15.2.1 A Template for Working with Multiple cameras

In this section we'll cover two concepts: (1) Multiple camera hardware configurations and (2) Template code for using OpenCV/imutils to access multiple cameras.

Our hardware can be configured in either one of two ways:

- i. Two USB cameras
- ii. One USB camera and one PiCamera

**Remark.** Your RPi's Universal Serial Bus (USB) is likely shared. This means that working with more than two USB cameras on the same bus is likely not possible. Don't get your hopes up for connecting four USB cameras or more of them using a hub. It simply won't work.

With either hardware configuration, you can use the `video` module from `imutils` to set up two separate video streams. Here is our template:

---

```

1 # import the necessary packages
2 from imutils.video import VideoStream
3 import time
4
5 # initialize the video streams and allow them to warmup
6 print("[INFO] starting cameras...")
7 leftStream = VideoStream(src=0).start()
8 rightStream = VideoStream(src=1).start()
9 #rightStream = VideoStream(usePiCamera=True).start()
10 time.sleep(2.0)
11
12 # loop over frames from the video streams
13 while True:
14     # grab the frames from their respective video streams
15     left = leftStream.read()
16     right = rightStream.read()
```

---

```

17
18     # do something with each frame

```

---

**Lines 7 and 8** set up two USB webcam streams. Alternatively, you could change the parameter to use a PiCamera for one stream via **Line 9** (currently commented out). From there you can go about your business working with the `left` and `right` images in your loop (**Lines 13-18**).

That's really all there is to it! Now let's proceed to our panorama and pedestrian detection example.

Be sure to refer to my blog post on *Multiple cameras with the Raspberry Pi and OpenCV* for additional details (<http://pyimg.co/n9upt>) [53] on this template.

### 15.2.2 Project Structure

Our panorama + pedestrian detection project is laid out in the following manner:

---

```

|--- cascade
|   |--- haarcascade_fullbody.xml
|   |--- haarcascade_upperbody.xml
|--- pyimagesearch
|   |--- __init__.py
|   |--- panorama.py
|   |--- pedestriandetector.py
|--- realtime_stitching.py

```

---

Two OpenCV Haar Cascades are provided:

- i. `haarcascade_fullbody.xml`: A full body detector.
- ii. `haarcascade_upperbody.xml`: An upper body detector.

Using the Haar Cascades, we will detect people in a single panorama frame comprised of two camera feeds.

Our `pyimagesearch` module consists of two Python classes. In `panorama.py` the `Stitcher` class is implemented to perform image stitching based on keypoint matching. And in `pedestriandetector.py`, the `PedestrianDetector` class uses a Haar Cascade to detect and localize people in images.

Our driver script is `realtime_stitching.py`. This script handles two camera feeds and performs stitching as well as pedestrian detection.

### 15.2.3 Our Panorama Stitcher

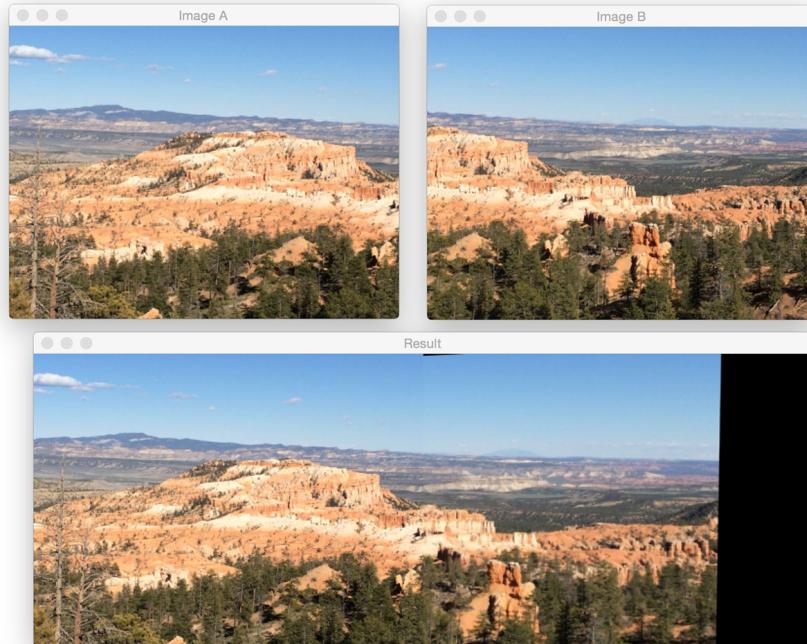


Figure 15.1: An example of taking two input images and stitching them together to form a panorama.

Our panorama stitching algorithm is inspired by Matthew Brown and David Lowe's seminal 1999 paper *Automatic Panoramic Image Stitching using Invariant Features* [54]. The algorithm itself consists of four steps:

- i. **Step #1:** Detect keypoints (DoG, Harris, etc.) and extract local invariant descriptors (SIFT, SURF, etc.) from the two input images
- ii. **Step #2:** Match the descriptors between the two images
- iii. **Step #3:** Use the RANSAC algorithm to estimate a homography matrix using our matched feature vectors
- iv. **Step #4:** Apply a warping transformation using the homography matrix obtained from **Step #3**

The output of our image stitcher will look similar to Figure 15.1. The *top-left* and *top-right* displays our example input images, including an overlap of the rocky tableau space. The *bottom* visualizes the output of our stitching algorithm, merging the two.

We'll encapsulate all four of these steps inside `panorama.py` where we'll define a `Stitcher` class used to construct our panoramas.

In order to (1) create a real-time image stitcher and (2) perform motion detection on the panorama image, we'll assume that both cameras are *fixed* and *non-moving*, like in Figure 15.1.

Why is the fixed and non-moving assumption so important?

Performing keypoint detection, local invariant description, keypoint matching, and homography estimation is a *computationally expensive* task. If we were to perform stitching on *each* set of frames, it would make it near-impossible to run in real-time (especially for resource constrained hardware like the Raspberry Pi). However, if we assume that the cameras are fixed, ***we only have to perform the homography matrix estimation once!*** After the initial homography estimation, we can use the same matrix to transform and warp the images to construct the final panorama — doing this enables us to skip the computationally expensive steps of keypoint detection, local invariant feature extraction, and keypoint matching in each set of frames.

Let's review `panorama.py` which contains the `Stitcher` class:

---

```

1 # import the necessary packages
2 import numpy as np
3 import imutils
4 import cv2
5
6 class Stitcher:
7     def __init__(self):
8         # determine if we are using OpenCV v3.X and initialize the
9         # cached homography matrix
10        self.isv3 = imutils.is_cv3(or_better=True)
11        self.cachedH = None

```

---

We start off on **Lines 2-4** by importing our necessary packages. We'll be using NumPy for matrix/array operations, `imutils` for our set of OpenCV convenience methods, and finally `cv2` for our OpenCV bindings.

From there, we define the `Stitcher` class on **Line 6**. The constructor to `Stitcher` simply checks which version of OpenCV we are using by making a call to the `is_cv3` method. Since there are major differences in how OpenCV 2.4 and higher versions handle keypoint detection and local invariant descriptors, it's important that we ensure that we are using OpenCV 3 or better.

**Line 11** defines `cachedH` the cached homography matrix. We will use it later to prevent duplicate and expensive computation.

Next up, let's start working on the `stitch` method:

---

```

13     def stitch(self, images, ratio=0.75, reprojThresh=4.0):
14         # unpack the images

```

---

```

15     (imageB, imageA) = images
16
17     # if the cached homography matrix is None, then we need to
18     # apply keypoint matching to construct it
19     if self.cachedH is None:
20         # detect keypoints and extract
21         (kpsA, featuresA) = self.detectAndDescribe(imageA)
22         (kpsB, featuresB) = self.detectAndDescribe(imageB)
23
24         # match features between the two images
25         M = self.matchKeypoints(kpsA, kpsB,
26             featuresA, featuresB, ratio, reprojThresh)
27
28         # if the match is None, then there aren't enough matched
29         # keypoints to create a panorama
30         if M is None:
31             return None
32
33         # cache the homography matrix
34         self.cachedH = M[1]
35
36         # apply a perspective transform to stitch the images together
37         # using the cached homography matrix
38         result = cv2.warpPerspective(imageA, self.cachedH,
39             (imageA.shape[1] + imageB.shape[1], imageA.shape[0]))
40         result[0:imageB.shape[0], 0:imageB.shape[1]] = imageB
41
42         # return the stitched image
43         return result

```

---

The `stitch` method requires only a single parameter, `images`, which is the list of (two) images that we are going to stitch together to form the panorama. We can also optionally supply `ratio`, used for David Lowe's ratio test when matching features, and `reprojThresh` which is the maximum pixel "wiggle room" allowed by the RANSAC algorithm.

**Line 15** unpacks the `images` list (which again, we presume to contain only two images). The ordering to the `images` list is important: **we expect images to be supplied in *left-to-right order***. If images are *not* supplied in this order, then our code will still run — but the output panorama will only contain one image, not both.

On **Line 19** we make a check to see if the homography matrix has been computed before. If not, we proceed to detect keypoints and extract local invariant descriptors from the two images via a call to the `detectAndDescribe` method on **Lines 21 and 22**. This method simply detects keypoints and extracts local invariant descriptors (i.e., SIFT) from the two images.

This is followed by applying keypoint matching to match the features in the two images (**Lines 25 and 26**). We'll define this method later in the lesson.

If the returned matches, `M`, are `None`, then not enough keypoints were matched to create a

panorama, so we simply `return` to the calling function (**Lines 30 and 31**).

We then cache the homography matrix on **Line 34**. Subsequent calls to `stitch` will use this cached matrix, allowing us to sidestep detecting keypoints, extracting features, and performing keypoint matching on *every* set of frames.

Finally, we apply a perspective transform and concatenate our images via NumPy slicing (**Lines 38-40**). The `result` is then returned to the caller.

Now that the `stitch` method has been defined, let's look into some of the helper methods that it calls. We'll start with `detectAndDescribe`:

---

```

45     def detectAndDescribe(self, image):
46         # convert the image to grayscale
47         gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
48
49         # check to see if we are using OpenCV 3.X
50         if self.isv3:
51             # detect and extract features from the image
52             descriptor = cv2.xfeatures2d.SIFT_create()
53             (kps, features) = descriptor.detectAndCompute(image, None)
54
55         # otherwise, we are using OpenCV 2.4.X
56         else:
57             # detect keypoints in the image
58             detector = cv2.FeatureDetector_create("SIFT")
59             kps = detector.detect(gray)
60
61             # extract features from the image
62             extractor = cv2.DescriptorExtractor_create("SIFT")
63             (kps, features) = extractor.compute(gray, kps)
64
65             # convert the keypoints from KeyPoint objects to NumPy
66             # arrays
67             kps = np.float32([kp.pt for kp in kps])
68
69             # return a tuple of keypoints and features
70             return (kps, features)

```

---

As the name suggests, the `detectAndDescribe` method accepts an image, then detects keypoints and extracts local invariant descriptors. In our implementation we use the Difference of Gaussian (DoG) keypoint detector and the SIFT feature extractor [55].

On **Line 50** we check to see if we are using OpenCV 3 or better. If we are, then we use the `SIFT_create` function to instantiate both our DoG keypoint detector and SIFT feature extractor. A call to `detectAndCompute` handles extracting the keypoints and features (**Lines 52 and 53**).

**Remark.** It's important to note that you **must** have compiled OpenCV 3, 4, or higher with

`opencv_contrib` support enabled. You should also ensure you have included the “`NON_FREE`” algorithms. If you did not, you’ll get an error such as `AttributeError: 'module' object has no attribute 'xfeatures2d'`.

If that’s the case, head over to my OpenCV tutorials page (<http://pyimg.co/cuph2>) where I detail how to install OpenCV with `opencv_contrib` and “`NON_FREE`” support enabled for your Raspberry Pi. The Raspbian images that come with this book include all of the OpenCV functionality.

**Lines 56-63** handle if we are using OpenCV 2.4 for legacy support.

Finally, our keypoints are converted from `KeyPoint` objects to a NumPy array (**Line 67**) and returned to the calling method (**Line 70**).

Next up, let’s look at the `matchKeypoints` method:

---

```

72     def matchKeypoints(self, kpsA, kpsB, featuresA, featuresB,
73                         ratio, reprojThresh):
74         # compute the raw matches and initialize the list of actual
75         # matches
76         matcher = cv2.DescriptorMatcher_create("BruteForce")
77         rawMatches = matcher.knnMatch(featuresA, featuresB, 2)
78         matches = []
79
80         # loop over the raw matches
81         for m in rawMatches:
82             # ensure the distance is within a certain ratio of each
83             # other (i.e. Lowe's ratio test)
84             if len(m) == 2 and m[0].distance < m[1].distance * ratio:
85                 matches.append((m[0].trainIdx, m[0].queryIdx))
86
87         # computing a homography requires at least 4 matches
88         if len(matches) > 4:
89             # construct the two sets of points
90             ptsA = np.float32([kpsA[i] for (_, i) in matches])
91             ptsB = np.float32([kpsB[i] for (i, _) in matches])
92
93             # compute the homography between the two sets of points
94             (H, status) = cv2.findHomography(ptsA, ptsB, cv2.RANSAC,
95                                             reprojThresh)
96
97             # return the matches along with the homography matrix
98             # and status of each matched point
99             return (matches, H, status)
100
101         # otherwise, no homography could be computed
102         return None

```

---

The `matchKeypoints` function requires six parameters:

- `kpsA` and `featuresA`: First image keypoints and feature vectors
- `kpsB` and `featuresB`: Second image keypoints and feature vectors
- `ratio`: David Lowe's ratio test variable
- `reprojThresh`: The RANSAC re-projection threshold.

Matching features together is actually a fairly straightforward process. We simply loop over the descriptors from both images, compute the distances, and find the smallest distance for each pair of descriptors.

Since this is a very common practice in computer vision, OpenCV has a built-in function called `DescriptorMatcher_create` that constructs the feature matcher for us. The `BruteForce` value indicates that we are going to *exhaustively* compute the Euclidean distance between *all feature vectors* from both images, and find the pairs of descriptors that have the smallest distance.

A call to `knnMatch` on **Line 77** performs k-NN matching [54, 55] between the two feature vector sets using `k=2` (indicating the top two matches for each feature vector are returned). The reason we want the top *two* matches rather than just the top one match is because we need to apply David Lowe's ratio test for false-positive match pruning.

Again, **Line 77** computes the `rawMatches` for each pair of descriptors — but there is a chance that some of these pairs are false positives, meaning that the image patches are not actually true matches.

In an attempt to prune these false-positive matches, we can loop over each of the `rawMatches` individually (**Line 81**) and apply Lowe's ratio test (**Line 84**), which is used to determine high-quality feature matches. Typical values for Lowe's ratio are normally in the range  $[0.7, 0.8]$ .

Once we have obtained the matches using Lowe's ratio test, we can compute the homography between the two sets of keypoints. Computing a homography between two sets of points requires, *at a bare minimum*, an initial set of four matches (**Line 88**). For a more reliable homography estimation, we should have substantially more than just four matched points.

Provided that there are *at least* four matches, **Lines 90-99** compute and return the homography. Instantiation and usage of the `Stitcher` class will be covered in the driver script in the next section.

This section included content from two of my previous blog posts. Be sure to refer to them to reinforce the concepts in this section: (1) *OpenCV Panorama Stitching* (<http://pyimg.co/nzfn2>) [56], and (2) *Real-time Panorama Stitching with OpenCV* (<http://pyimg.co/asto0>) [57].

### 15.2.4 Pedestrian Detection

Our pedestrian detection algorithm is based on the Haar Cascade approach as it will run quickly on most Raspberry Pi versions. For more accurate detections, you could also try the pre-trained Caffe MobileNet SSD [58] either with the CPU or a coprocessor such as the Coral or Movidius (covered in the *Hacker Bundle* and *Complete Bundle* of this book).

Let's implement our Haar-based `PedestrianDetector` class now inside `pedestriandetector.py`:

---

```

1 # import the necessary packages
2 import cv2
3
4 class PedestrianDetector:
5     def __init__(self, cascadePath):
6         # load the pedestrian detector
7         self.pedestrianCascade = cv2.CascadeClassifier(cascadePath)
8
9     def detect(self, image, scaleFactor=1.1, minNeighbors=5,
10               minSize=(30, 30)):
11         # detect pedestrians in the image
12         rects = self.pedestrianCascade.detectMultiScale(image,
13                 scaleFactor=scaleFactor, minNeighbors=minNeighbors,
14                 minSize=minSize, flags=cv2.CASCADE_SCALE_IMAGE)
15
16         # return the rectangles representing bounding boxes around
17         # the pedestrians
18         return rects

```

---

**Line 2** imports OpenCV. From there, we define the `PedestrianDetector` class and constructor on **Lines 4-7**. The constructor simply accepts a path to a Haar cascade and initializes the `CascadeClassifier` object.

The `detect` method wraps the `detectMultiScale` Haar method with the following parameters:

- `image`: The panorama image
- `scaleFactor`: Parameter specifying how much the image pyramid should be scaled
- `minNeighbors`: The number of overlapping detections required to mark the pedestrian as a positive
- `minSize`: The width and height in pixels

**Lines 12-14** call the `detectMultiScale` function returning the detected rectangles. **Line 18** then returns the `rects` (containing our localized detections) to the caller.

### 15.2.5 A Two-camera Panorama Pedestrian Detector Driver Script

We now have all the components to build a successful computer vision app to work with multiple cameras, stitch images, and then detect people in the panorama.

Let's get started!

Open up `realtime_stitching.py` and insert the following code:

---

```

1 # import the necessary packages
2 from pyimagesearch.pedestriandetector import PedestrianDetector
3 from pyimagesearch.panorama import Stitcher
4 from imutils.video import VideoStream
5 import numpy as np
6 import argparse
7 import datetime
8 import imutils
9 import time
10 import cv2
11
12 # construct the argument parser and parse the arguments
13 ap = argparse.ArgumentParser()
14 ap.add_argument("-c", "--cascade", required = True,
15     help = "path to where the pedestrian cascade resides")
16 args = vars(ap.parse_args())

```

---

**Lines 2-10** import our necessary packages. Most notably, we're using our new `PedestrianDetector` and `Stitcher` classes.

**Lines 13-16** parse a single command line argument, `--cascade`, the path to the Haar Cascade file residing on disk.

Let's initialize our `Stitcher` and `PedestrianDetector`:

---

```

18 # initialize the image stitcher and create a pedestrian detector
19 # object
20 stitcher = Stitcher()
21 pd = PedestrianDetector(args["cascade"])

```

---

**Line 20** initializes our `stitcher` object which we will soon put to work to create our panorama images.

**Line 21** then initializes `pd`, our pedestrian detector. The path to the Haar Cascade from the command line `args` is provided as a parameter.

From here, let's initialize our video feeds and begin reading frames (following our previous template):

---

```

23 # initialize the video streams and allow them to warmup
24 print("[INFO] starting cameras...")
25 leftStream = VideoStream(src=0).start()
26 rightStream = VideoStream(src=1).start()
27 time.sleep(2.0)
28
29 # loop over frames from the video streams
30 while True:
31     # grab the frames from their respective video streams
32     left = leftStream.read()
33     right = rightStream.read()
34
35     # resize the frames
36     left = imutils.resize(left, width=500)
37     right = imutils.resize(right, width=500)

```

---

**Lines 25 and 26** initialize our `leftStream` and `rightStream` video feeds. Be sure to refer to the explanation in Section 15.2.1 on working with multiple cameras if you need help. Remember, you can work with two USB cameras or a USB + PiCamera.

Our frame processing loop begins on **Line 30**.

First, we read a frame from the `left` and `right` cameras (**Lines 32 and 33**). Order *does matter*, so adjust the source numbers on **Lines 25 and 26** if necessary.

We go ahead and resize both frames to 500 pixels wide (**Lines 36 and 37**). Both frames must have the same dimensions.

**Remark.** On **Lines 36 and 37** we preserve the aspect ratios while resizing via `imutils.resize`. If your frames' aspect ratios are different, then you'll need to use `cv2.resize`, otherwise it will not be possible to stitch the images.

From here, we'll **stitch the images together**:

---

```

39     # stitch the frames together to form the panorama
40     # IMPORTANT: you might have to change this line of code
41     # depending on how your cameras are oriented; frames
42     # should be supplied in left-to-right order
43     result = stitcher.stitch([left, right])
44
45     # no homography could be computed
46     if result is None:
47         print("[INFO] homography could not be computed")
48         break

```

---

**Line 43** stitches the `left` and `right` images using our custom `stitch` method. The function expects that the *first* image in the list is the actual *left camera* and the *second* is the actual *right camera*.

Our script exits if a homography cannot be computed. This would only occur during the first iteration of the loop (if at all) as we cache the homography matrix the first time the method is called.

Now let's **detect** **pedestrians**:

---

```

50      # convert the stitched frame to grayscale and find pedestrians in
51      # the image
52      gray = cv2.cvtColor(result, cv2.COLOR_BGR2GRAY)
53      pedestrianRects = pd.detect(gray, scaleFactor=1.1,
54          minNeighbors=5, minSize=(30, 30))
55
56      # loop over the pedestrians and draw a rectangle around each
57      for (x, y, w, h) in pedestrianRects:
58          cv2.rectangle(result, (x, y), (x + w, y + h), (0, 255, 0), 2)

```

---

Our Haar detector requires a grayscale image. **Line 52** facilitates the conversion to grayscale.

From there, we call `detect` on our `pd` object while providing our `gray` image and the other parameters covered in the previous section. **Lines 57 and 58** then draw rectangles around each detection on the panorama image.

From here, we'll annotate the frame with a timestamp and display it:

---

```

60      # draw the timestamp on the image
61      timestamp = datetime.datetime.now()
62      ts = timestamp.strftime("%A %d %B %Y %I:%M:%S%p")
63      cv2.putText(result, ts, (10, result.shape[0] - 10),
64          cv2.FONT_HERSHEY_SIMPLEX, 0.35, (0, 0, 255), 1)
65
66      # show the output image
67      cv2.imshow("Result", result)
68      key = cv2.waitKey(1) & 0xFF
69
70      # if the `q` key was pressed, break from the loop
71      if key == ord("q"):
72          break
73
74      # do a bit of cleanup
75      print("[INFO] cleaning up...")
76      cv2.destroyAllWindows()
77      leftStream.stop()
78      rightStream.stop()

```

---

This is a security application, so a timestamp is generated and drawn in the corner of the panorama (**Lines 61-64**). The image is then displayed until the "q" key is pressed (**Lines 67-72**).

Necessary cleanup is performed on **Lines 76-78**. Don't forget to stop both camera streams!

### 15.2.6 Executing the Panorama Pedestrian Detector



Figure 15.2: An example of stitching together two video streams (USB on the *left* and PiCamera on the *right* in each frame) while also detecting full-body pedestrians.

When you're ready, go ahead and enter the following command to run the script with the full-body Haar Cascade:

---

```
$ python realtime_stitching.py --cascade cascade/haarcascade_fullbody.xml
[INFO] starting cameras...
[INFO] cleaning up...
```

---

You can substitute the `--cascade` argument with the `haarcascade_upperbody.xml` file as well.

Figure 15.2 demonstrates stitching together two video sources (USB on the *left* and PiCamera on the *right*) while also detecting full-body pedestrians.

It may take some practice and manual tuning to set the Haar cascade parameters for adequate pedestrian detection. Specifically, you should play with the `scaleFactor`, `minNeighbors`, and `minSize`:

- `scaleFactor`: **How much the image size is reduced at each image scale. This value is used to create the scale pyramid.** It detects objects at multiple scales in the image (some objects may be closer to the foreground, and thus be larger, other objects may be smaller and in the background, thus the usage of varying scales). A value of `1.05` indicates that we are reducing the size of the image by 5% at each level in the pyramid.
- `minNeighbors`: **How many neighbors each window should have for the area in the window to be considered an object.** The cascade classifier will detect multiple windows around an object. This parameter controls how many rectangles (neighbors) need to be detected for the window to be labeled as a positive object detection.
- `minSize`: **A tuple of width and height (in pixels) indicating the minimum size of the window.** Bounding boxes smaller than this size are ignored. It is a good idea to start with `(30, 30)` and fine-tune from there.

Additionally, don't forget that you could use a more accurate pedestrian detector such as the "person" class as part of MobileNet SSD's pre-trained object detector (<http://pyimg.co/5gmse>) [59]. While deep learning-based object detectors are slower, they tend to be more accurate and have less parameters to configure. We'll be covering how to use these detectors on the Raspberry Pi in the *Hacker Bundle* of this text.

### 15.2.7 Tips and Suggestions

As you work with this code to develop your own projects, keep the following suggestions in mind:

- This program requires the `opencv_contrib` and "NON\_FREE" algorithms. Be sure you compile OpenCV with these features. See the companion website associated with this text for instructions on how to install OpenCV's "NON\_FREE" module.
- Ensure that your left camera and right camera are set up properly, both physically and programmatically, otherwise a valid homography matrix cannot be calculated. Multiple USB camera devices have a tendency to swap without notice when power is taken away from the Pi. You can do a quick test by running the template code in Section 15.2.1 and adding `cv2.imshow` calls at the bottom of the `while` loop, so you can determine which camera is which and make programmatic changes accordingly.
- This script and algorithm only supports two cameras. One reason for this limitation is the `Stitcher` class, which only supports two images. If you're not going to be stitching

camera feeds, please also keep in mind that USB shared bus may only support streaming from two cameras (i.e. the shared bus likely does not have enough bandwidth and cannot handle more than 2 streams at a time).

- If you want to use two or more PiCameras, you could look into the “*Arducam Multi Camera Adapter*” board (<http://pyimg.co/z3ham>). I’m not sure how long this product will be available on the market, which is why I’m not covering it in this book, but wanted to provide it to you just in case you are interested.

### 15.3 Summary

In this chapter, we learned how to work with multiple cameras and the Raspberry Pi. We took it two steps further with a practical example that (1) creates a panorama image, and (2) detects people.

Let your imagination run wild with multiple cameras using the Raspberry Pi.

If your cameras will be further than a cable length apart, and you are dead set on using one RPi, then wired or wireless IP-based cameras are an option. IP cameras tend to be pricy, so instead I encourage you to use multiple RPis as is demonstrated in Chapter 9 (“*Monitoring Your Home with Deep Learning and Multiple RPis*”) of the *Hacker Bundle*. In that chapter, we learn how to stream video feeds from cameras around your WiFi network to a central location.

## Chapter 16

# Detecting Tired, Drowsy Drivers Behind the Wheel

Back in May 2017, I wrote about my Uncle John on the PyImageSearch blog [60]. I'm going to share the story again here in this book — what we learned in that blog post is that simple technology can be put in vehicles to save lives.

John is a long-haul tractor trailer truck driver. For each new assignment, he picks his load up from a local company early in the morning and then sets off on a lengthy, enduring cross-country trek across the United States that takes him *days* to complete.

John is a nice, outgoing guy who carries a smart, witty demeanor. He also fits the “*cowboy of the highway*” stereotype to a T, sporting a big ole’ trucker cap, red-checkered flannel shirt, and a faded pair of Levi’s that have more than one splotch of oil stain from quick and dirty roadside fixes. He also loves his country music.

I caught up with John during a family dinner and asked him about his trucking job. I was genuinely curious — before I entered high school, I thought it would be fun to drive a truck or a car for a living (personally, I find driving to be a pleasurable, therapeutic experience).

### **But my question was also a bit self-motivated as well.**

Earlier that morning, I had just finished writing the code for my first drowsiness detector iteration and wanted to get his take on how computer science (and more specifically, computer vision) was affecting his trucking job.

The truth was that **John was scared about his employment, his livelihood, and his future**. The *first five sentences* out of his mouth included the words “Tesla”, “Self-driving cars”, and “Artificial Intelligence (AI)”. Many proponents of autonomous, self-driving vehicles argue that the first industry that will be *completely and totally overhauled* by self-driving cars/trucks (even before consumer vehicles) is the long-haul tractor trailer business.

As self-driving tractor trailers becomes a reality, John has good reason to be worried — he'll be out of a job, one that he's been doing his entire life. He's also getting close to retirement and needs to finish out his working years strong.

This isn't speculation either: NVIDIA has partnered with PACCAR [61], a leading global truck manufacturer. The goal of this partnership is to make self-driving semi-trailers a reality. Tesla already has self driving semi-trucks on the road [62].

After John and I were done discussing self-driving vehicles, I asked him the critical question that this very chapter hinges on:

*"Have you ever fallen asleep at the wheel?"*

I could tell instantly that John was uncomfortable. He didn't look me in the eye. And when he finally did answer, it wasn't a direct one — instead he recalled a story about his friend (name left out on purpose) who fell asleep after disobeying company policy on maximum number of hours driven during a 24 hour period.

The man ran off the highway, the contents of his truck spilling all over the road, blocking the interstate almost the entire night. Luckily, no one was injured, but it gave John quite the scare as he realized that if it could happen to other drivers, it could happen to him as well.

I then explained to John my work — **a computer vision system that can automatically detect driver drowsiness in a real-time video stream and then play an alarm if the driver appears to be drowsy**. While John said he was uncomfortable being directly video surveyed while driving, he did admit that it the technique would be helpful in the industry and ideally reduce the number of fatigue-related accidents.

In this chapter, I am going to show you my next iteration of the code, which detects not only sleepy eyes, but also counts yawns which are also an indication of a drowsy driver.

## 16.1 Chapter Learning Objectives

In this chapter, we will learn how to build a drowsiness detector. We will accomplish the following objectives:

- i. Detect facial landmarks with dlib
- ii. Calculate Eye Aspect Ratio (EAR) and use it to detect when the driver's eyes are closed too long
- iii. Calculate Mouth Aspect Ratio (MAR) and use it to count the number of yawns in a consecutive period

- iv. Interface with the aptly named TrafficHAT which includes a buzzer and LED warning lights
- v. Combine the techniques mentioned above and use them to detect drowsy drivers in real-time

Let's go ahead and get started.



Figure 16.1: According to the United States National Highway Traffic Safety Administration, as many as 6,000 fatal crashes a year are caused by drowsy driving, in addition to over 40,000 injuries and over 70,000 total crashes. The goal of this chapter is to develop a computer vision method to help reduce the number of injuries and deaths each year attributed to drowsy driving.

## 16.2 Understanding the "Eye Aspect Ratio" (EAR)

We can employ dlib's [16] pre-trained facial landmark detector to localize important regions of the face, including eyes, eyebrows, nose, ears, and mouth as shown in Figure 16.2 [63]. This also implies that we can extract specific facial structures by knowing the indices of the particular face parts (Figure 16.3) [64].

In terms of blink detection, we are only interested in two sets of facial structures — the eyes. Each eye is represented by six  $(x, y)$ -coordinates, starting at the left-corner of the eye (as if you were looking at the person), and then working clockwise around the remainder of the region (Figure 16.4, *top-left*).

Based on this image, you should notice the relation between the width and the height of these coordinates. In the work by Soukupová and Čech in their 2016 paper *Real-Time Eye*

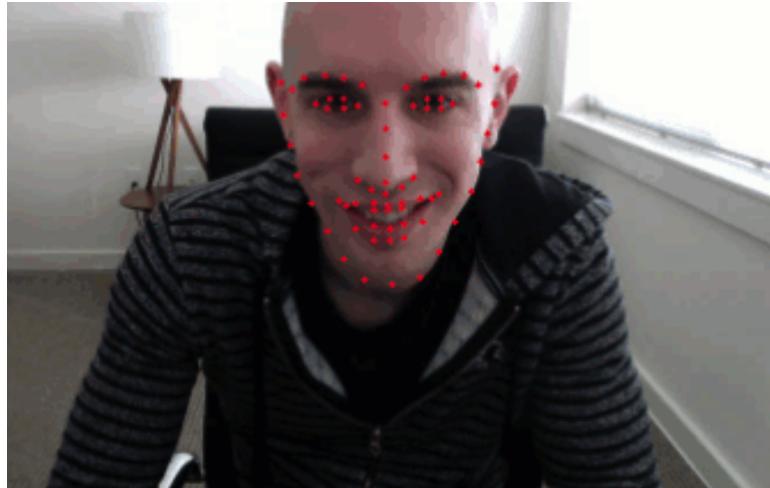


Figure 16.2: Detecting 68 facial landmarks is as simple as employing a pre-trained dlib shape predictor model.

*Blink Detection using Facial Landmarks* [65], we can then derive an equation that reflects this relation called the **eye aspect ratio (EAR)**, where  $p_1, \dots, p_6$  are 2D facial landmark locations.

$$EAR = \frac{||p_2 - p_6|| + ||p_3 - p_5||}{2||p_1 - p_4||} \quad (16.1)$$

The numerator of this equation computes the distance between the vertical eye landmarks while the denominator computes the distance between horizontal eye landmarks, weighting the denominator appropriately since there is only one set of horizontal points but two sets of vertical points.

Why is this equation so interesting?

As we'll find out, the eye aspect ratio is approximately constant while the eye is open, but will rapidly fall to zero when a blink is taking place. Using this simple equation, we can avoid image processing techniques and simply rely on the ratio of eye landmark distances to determine if a person is blinking.

To make this more clear, consider Figure 16.4 from Soukupová and Čech. On the top-left we have an eye that is fully open — the eye aspect ratio here would be large(r) and relatively constant over time. However, once the person blinks (Figure 16.4, top-right) the eye aspect ratio decreases dramatically, approaching zero.

The bottom of Figure 16.4 plots a graph of the eye aspect ratio over time for a video clip. As we can see, the eye aspect ratio is constant, then rapidly drops close to zero, then increases again, indicating a single blink has taken place.

In the remainder of this chapter, we'll both implement and apply the EAR formula to detect



Figure 16.3: Knowing the indices of facial landmarks allows us to extract, highlight, or take measurements from specific landmarks such as an eye or mouth.

drowsy blinks. We'll also form a similar equation for detecting yawns. The **Mouth Aspect Ratio (MAR)** uses the exact concept described in this section.

### 16.3 Detecting Drowsy Drivers at the Wheel with the RPi

In this section, we'll first cover our project structure and walk through the configuration file. From there, we'll develop our drowsiness detection script. Finally, we'll learn how to deploy the drowsiness detector to our Raspberry Pi, install it in our vehicle, and achieve the best results.

***DISCLAIMER:*** *Do not exclusively rely on this system to keep you awake while driving! Do not get distracted with electronics while driving. This is a proof of concept system used to teach computer vision concepts. Thus, any testing or actual usage is at your own risk.*

#### 16.3.1 Project Structure

Our project structure is as follows:

```
|-- config
|   |-- config.json
|-- pyimagesearch
|   |-- utils
|       |-- __init__.py
```

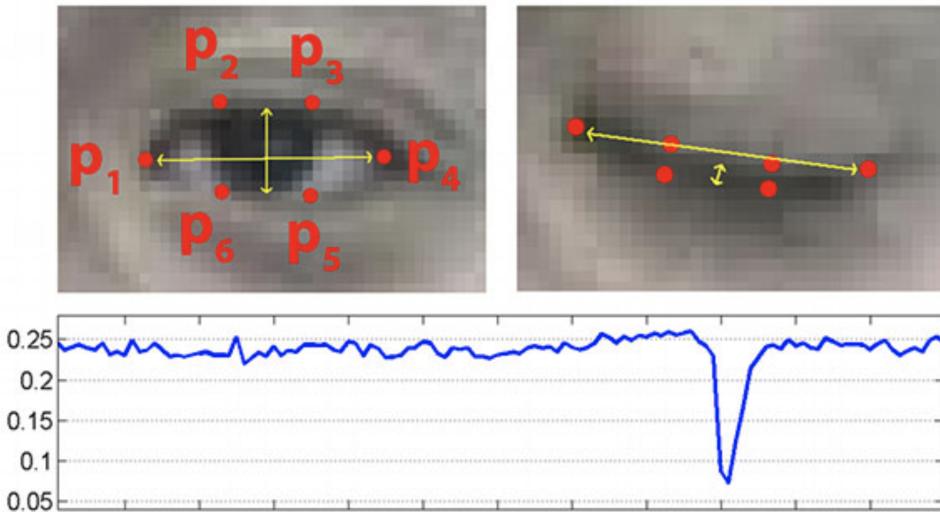


Figure 16.4: **Top-left:** A visualization of eye landmarks when then the eye is open. **Top-right:** Eye landmarks when the eye is closed. **Bottom:** Plotting the eye aspect ratio over time. The dip in the eye aspect ratio indicates a blink (Figure 1 of Soukupová and Čech [65]).

---

```

|     |     |-- conf.py
|     |     |-- __init__.py
|-- detect_drowsiness.py
|-- haarcascade_frontalface_default.xml
|-- shape_predictor_68_face_landmarks.dat

```

---

Our JSON configuration file contains key parameters for our application. These are pulled out in `config.json` so that they can be conveniently stored and adjusted without scrolling through many lines of code and introducing mistakes and bugs.

The `pyimagesearch` module contains the `Conf` class used to parse the JSON configuration.

Our Haar Cascade XML file is used to localize the face within the frame, keeping in mind that the face closest to the center of the frame is the one used in case other faces are in the camera's view.

To detect landmarks on the face (and calculate EAR/MAR), we'll use dlib's pretrained shape predictor. This shape predictor predicts 68 points on a face.

The heart of today's chapter is inside `detect_drowsiness.py`. We'll review all of the logic and code inside that script later in this chapter.

### 16.3.2 Our Drowsiness Configuration

One change from my implementations on the PyImageSearch blog is that in this book, we will use a configuration file for our application settings. The configuration makes changing settings a lot more convenient.

Go ahead and open up config.json in your terminal:

---

```

1  {
2      // path to OpenCV's face detector haar cascade file
3      "cascade_path": "haarcascade_frontalface_default.xml",
4
5      // path to dlib's facial landmark predictor
6      "shape_predictor_path": "shape_predictor_68_face_landmarks.dat",
7
8      // two boolean variables, one used to indicate if frames must be
9      // displayed, and second used to indicate if TrafficHat should be
10     // used
11     "display": true,
12     "alarm": false,
13
14     // two constants, one for the eye aspect ratio to indicate
15     // blink and then a second constant for the number of consecutive
16     // frames the eye must be below the threshold for to set off the
17     // alarm
18     "EYE_AR_THRESH": 0.3,
19     "EYE_AR_CONSEC_FRAMES": 8,
20
21     // three constants, one for the mouth aspect ratio to indicate
22     // yawn, a second constant for the yawn threshold, and a third
23     // constant for the yawn threshold time
24     "MOUTH_AR_THRESH": 0.65,
25     "YAWN_THRESH_COUNT": 120,
26     "YAWN_THRESH_TIME": 600
27 }
```

---

**Line 3** contains the path to our pre-trained Haar cascade for detecting/localizing faces. This is the exact face detector included in the OpenCV repository. It is fast and reasonably accurate making it perfect for the RPi.

**Line 6** is the path to our pre-trained dlib facial landmark detector (also known as a shape predictor). To learn more about facial landmarks, be sure to refer to this tutorial on the PyImageSearch blog: <http://pyimg.co/x0f5r> [63].

If you have a display, you should set "display" to true on **Line 11**. If you are using the TrafficHAT module on your Raspberry Pi, be sure to set "alarm" to true (**Line 12**).

The EAR (Eye Aspect Ratio) threshold and consecutive "eyes closed" frames are shown on

**Lines 18 and 19.** Similarly the MAR (Mouth Aspect Ratio) and yawn count threshold within a given amount of time are on **Lines 24-26**.

### 16.3.3 Developing our Drowsiness Detection Script

With our configuration settings in hand, now we're ready to develop our drowsiness detection script. Though the script is 263 lines, it is relatively straightforward.

Let's get started by opening `detect_drowsiness.py` and inserting the following code:

---

```

1 # import the necessary packages
2 from pyimagesearch.utils import Conf
3 from imutils.video import VideoStream
4 from imutils import face_utils
5 from datetime import datetime
6 import numpy as np
7 import argparse
8 import imutils
9 import time
10 import dlib
11 import cv2

```

---

Notable packages and modules for import include:

- `Conf`: For parsing our configuration
- `VideoStream`: For reading frames from a camera
- `face_utils`: Convenient face landmark access
- `numpy`: Used to calculate the Euclidean distance between two landmarks
- `imutils`: OpenCV convenience functions
- `dlib`: For shape prediction (also known as landmark detection)

Let's define a function to calculate the distance between two landmarks:

---

```

13 def euclidean_dist(ptA, ptB):
14     # compute and return the euclidean distance between the two
15     # points
16     return np.linalg.norm(ptA - ptB)

```

---

The `euclidean_dist` function on **Lines 13-16** simply calculates the distance “as the crow flies” between two  $(x, y)$ -points. The Euclidean distance will allow us to detect if an eye or mouth is open or closed — the basis of our drowsiness detection method.

Let's go ahead and define two functions to calculate the **eye and mouth aspect ratios**:

---

```

18 def eye_aspect_ratio(eye):
19     # compute the euclidean distances between the two sets of
20     # vertical eye landmarks (x, y)-coordinates
21     a = euclidean_dist(eye[1], eye[5])
22     b = euclidean_dist(eye[2], eye[4])
23
24     # compute the euclidean distance between the horizontal
25     # eye landmark (x, y)-coordinates
26     c = euclidean_dist(eye[0], eye[3])
27
28     # compute the eye aspect ratio
29     ear = (a + b) / (2.0 * c)
30
31     # return the eye aspect ratio
32     return ear
33
34 def mouth_aspect_ratio(mouth):
35     # compute the euclidean distances between the three sets of
36     # vertical mouth landmarks (x, y)-coordinates
37     a = euclidean_dist(mouth[1], mouth[7])
38     b = euclidean_dist(mouth[2], mouth[6])
39     c = euclidean_dist(mouth[3], mouth[5])
40
41     # compute the euclidean distance between the horizontal
42     # mouth landmark (x, y)-coordinates
43     d = euclidean_dist(mouth[0], mouth[4])
44
45     # compute the mouth aspect ratio
46     mar = (a + b + c) / (2.0 * d)
47
48     # return the mouth aspect ratio
49     return mar

```

---

On **Lines 18-32**, the **Eye Aspect Ratio (EAR)** is calculated. The algorithm is quite simple. First, we compute the distances between the *vertical* eye landmarks (**Lines 21 and 22**). Similarly, we compute the distances between the *horizontal* eye landmarks (**Line 26**). A visualization of all facial landmarks available to us, for reference, is included in Figure 16.5.

The `ear` is the ratio between the *vertical* and *horizontal* landmark distances (**Line 29**). The value is returned to the caller via **Line 32**. The value will be approximately constant when the eye is open and will decrease towards zero during a blink. If the eye is closed, the EAR will remain constant at a much smaller value.

The **Mouth Aspect Ratio (MAR)** is calculated in the exact same way, albeit with different landmarks. I encourage you to review **Lines 34-49** to see the parallels between the two functions.

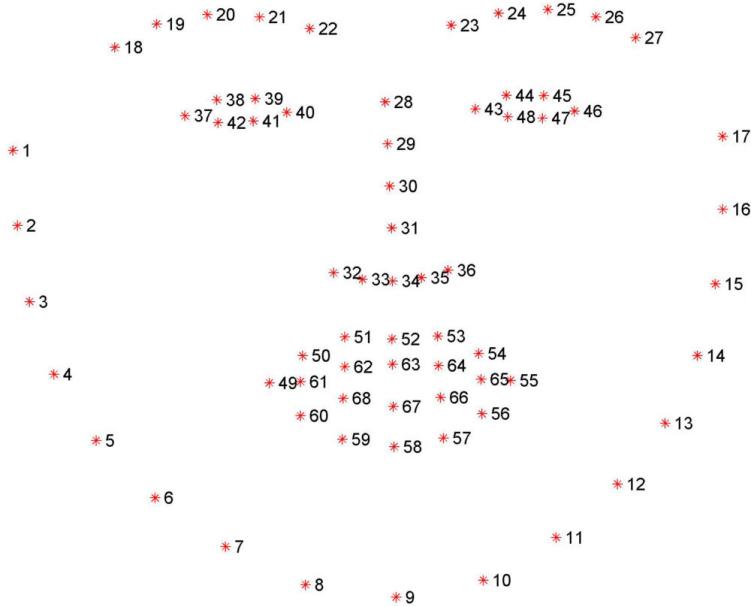


Figure 16.5: Visualizing the 68 facial landmark coordinates from the iBUG 300-W dataset [66].

Armed with our EAR and MAR algorithms, we're now ready to dig into the rest of the script. First, let's load our configuration and initialize our alarm system:

```
51 # construct the argument parser and parse the arguments
52 ap = argparse.ArgumentParser()
53 ap.add_argument("-c", "--conf", required=True,
54                 help="Path to the input configuration file")
55 args = vars(ap.parse_args())
56
57 # load the configuration file
58 conf = Conf(args["conf"])
59
60 # check to see if we are using GPIO/TrafficHat as an alarm
61 if conf["alarm"]:
62     from gpiozero import TrafficHat
63     th = TrafficHat()
64     print("[INFO] using TrafficHat alarm...")
```

**Lines 52-58** load our configuration via command line argument. Be sure to review Chapter 4 where we first covered command line arguments if you find yourself struggling with this code block.

**Lines 61-64** check the configuration for the "alarm" setting. If the alarm is to be used, we import TrafficHAT and instantiate the object.

The TrafficCHAT (Figure 16.6) is a Raspberry Pi module including a buzzer and lights. It is

listed on my hardware list on the book's companion website (link in the first few pages of this text). If you are using alternative hardware, be sure to hack this script to accommodate. A good rule of thumb would be to search for all instances of `th`, the TrafficHAT object.



Figure 16.6: The TrafficHat board containing button, buzzer, and lights.

Next we'll perform initializations:

---

```

66 # initialize the frame center coordinates
67 centerX = None
68 centerY = None
69
70 # initialize the blink counter, yawn counter, a boolean used to
71 # indicate if the alarm is going off, and start time
72 blinkCounter = 0
73 yawnCounter = 0
74 alarmOn = False
75 startTime = None
76
77 # load OpenCV's Haar cascade for face detection (which is faster than
78 # dlib's built-in HOG detector, but less accurate), then create the
79 # facial landmark predictor
80 print("[INFO] loading facial landmark predictor...")
81 detector = cv2.CascadeClassifier(conf["cascade_path"])
82 predictor = dlib.shape_predictor(conf["shape_predictor_path"])
83
84 # grab the indexes of the facial landmarks for the left, right eye,
85 # and inner part of the mouth respectively
86 (lStart, lEnd) = face_utils.FACIAL_LANDMARKS_IDXS["left_eye"]
87 (rStart, rEnd) = face_utils.FACIAL_LANDMARKS_IDXS["right_eye"]
88 (mStart, mEnd) = face_utils.FACIAL_LANDMARKS_IDXS["inner_mouth"]
89
90 # start the video stream thread
91 print("[INFO] starting video stream thread...")

```

---

```
92 vs = VideoStream(src=0).start()
93 #vs = VideoStream(usePiCamera=True).start()
94 time.sleep(2.0)
```

---

Let's review the initializations:

- `centerX` and `centerY`: The center of the frame coordinates (**Lines 67 and 68**).
- `blinkCounter` and `yawnCounter`: These values (**Lines 72 and 73**) will be used to measure the time the eyes are closed or the number of yawns within a specified amount of time. Be sure to refer to `conf["EYE_AR_CONSEC_FRAMES"]`, `conf["YAWN_THRESH_COUNT"]`, and `conf["YAWN_THRESH_TIME"]` in the configuration and throughout the script. Determining how long eyes are closed and counting yawns are essential to the drowsiness detection concept we are developing.
- `alarmOn`: A flag indicating whether the buzzer is currently actively trying to wake the driver up (**Line 74**).
- `startTime`: The time in which the yawn counting begins. The `startTime` will be measured against `conf["YAWN_THRESH_TIME"]` to determine if  $N$  yawns have passed within  $T$  seconds, indicating that the alarm needs to sound.

Continuing with our initializations, our face `detector` is initialized with our Haar Cascade on **Line 81**. Haar Cascades are sufficiently fast and reasonably accurate making them a better choice for face detection than HOG or DL based methods. Keep this in mind when detecting faces on the Raspberry Pi.

We also initialize the dlib shape `predictor` on **Line 82**. Refer to my facial landmarks blog post to learn the concept of shape prediction and facial landmarks (<http://pyimg.co/x0f5r>) [63]. Both the `detector` and `predictor` are pre-trained and ready to go, thanks to the hard work of the OpenCV team and Davis King [16] (maintainer of dlib).

**Lines 86-88** then grab all the left eye, right eye, and inner mouth landmark indices from the `face_utils` module in `imutils`. There's no magic here — if you want to see the indices for yourself, be sure to visit this direct link to the code on GitHub: <http://pyimg.co/v5b5k>.

Lastly, **Line 92 or 93** initializes our `VideoStream`. I recommend a USB camera here so you can position it on your car's dashboard or visor (hence why **Line 92** is active and **Line 93** is commented out). Our standard camera warmup time is coded on **Line 94**.

From there, let's get to business and begin looping over frames from the video stream:

---

```
96 # loop over frames from the video stream
97 while True:
```

---

```

98     # grab the frame from the threaded video file stream, resize,
99     # flip horizontally, and convert to grayscale
100    frame = vs.read()
101    frame = imutils.resize(frame, width=450)
102    frame = cv2.flip(frame, 1)
103    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
104
105    # set the frame center coordinates
106    if centerX is None and centerY is None:
107        (centerX, centerY) = (frame.shape[1] // 2,
108            frame.shape[0] // 2)
109
110    # detect faces in the grayscale frame
111    rects = detector.detectMultiScale(gray, scaleFactor=1.1,
112        minNeighbors=5, minSize=(30, 30),
113        flags=cv2.CASCADE_SCALE_IMAGE)
114
115    # loop over the detected faces
116    for rect in rects:
117        # draw a bounding box surrounding the face
118        (x, y, w, h) = rect
119        cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0), 2)

```

---

The beginning of this loop should look familiar, since you have the basics down at this point.

We read a frame, resize it (for efficiency), and flip on the horizontal axis (for mirror-like purposes if you are using a display).

Finally we convert it to grayscale (**Lines 100-103**).

The center of the frame is calculated on **Lines 106-108**. We will use the center of the frame to find the most likely driver (i.e. we do not care if passengers are drowsy).

We then detect faces in the grayscale image with our detector on **Lines 111-113**. Subsequently, we'll go ahead and draw a box around the faces (**Lines 116-119**). These few lines are great for debugging our script but aren't necessary if you are making an embedded product with no screen.

Now let's loop over the detections:

```

121    # check if the number of faces detected is greater than zero
122    if len(rects) > 0:
123        # sort the detected rectangles by their position relative to
124        # the center and grab the face that's closest to the center
125        centerRect = sorted(rects, key=lambda r: abs((
126            r[0] + (r[2] / 2)) - centerX) + abs((
127            r[1] + (r[3] / 2)) - centerY))[0]
128
129        # get the coordinates of the rectangle in the center and
130        # construct a dlib rectangle object from the Haar cascade

```

---

```

131     # bounding box
132     (x, y, w, h) = centerRect
133     rect = dlib.rectangle(int(x), int(y), int(x + w),
134                           int(y + h))
135
136     # determine the facial landmarks for the face region, then
137     # convert the facial landmark (x, y)-coordinates to a NumPy
138     # array
139     shape = predictor(gray, rect)
140     shape = face_utils.shape_to_np(shape)

```

---

**Line 122** begins a lengthy if-statement which is broken down into several code blocks beginning here.

First, we grab the face that is closest to the center of the frame (i.e. the driver) on **Lines 125-127**. Next, we extract the coordinates and width and height of the `rects` detections (**Line 132**).

**Lines 133 and 134** construct a `dlib.rectangle` object using the information extracted from the Haar cascade bounding box.

From there, we determine the **facial landmarks** for the face region (**Line 139**) and convert the facial landmark  $(x, y)$ -coordinates to a NumPy array (**Line 140**).

Given our NumPy array and shape we can extract each eye's coordinates and compute the EAR:

---

```

142     # extract the left and right eye coordinates, then use the
143     # coordinates to compute the eye aspect ratio for both eyes
144     leftEye = shape[lStart:lEnd]
145     rightEye = shape[rStart:rEnd]
146     leftEAR = eye_aspect_ratio(leftEye)
147     rightEAR = eye_aspect_ratio(rightEye)
148
149     # average the eye aspect ratio together for both eyes
150     ear = (leftEAR + rightEAR) / 2.0

```

---

Utilizing the indexes of the eye landmarks, we can slice the shape array to obtain the  $(x, y)$ -coordinates for each eye (**Lines 144 and 145**).

We then calculate the EAR for each eye on **Lines 146 and 147**.

Soukupová and Čech recommend averaging both eye aspect ratios together to obtain a better estimation (**Line 150**).

This next block is strictly for visualization purposes:

---

```

152     # compute the convex hull for the left and right eye, then
153     # visualize each of the eyes
154     leftEyeHull = cv2.convexHull(leftEye)
155     rightEyeHull = cv2.convexHull(rightEye)
156     cv2.drawContours(frame, [leftEyeHull], -1, (0, 255, 0), 1)
157     cv2.drawContours(frame, [rightEyeHull], -1, (0, 255, 0), 1)

```

---

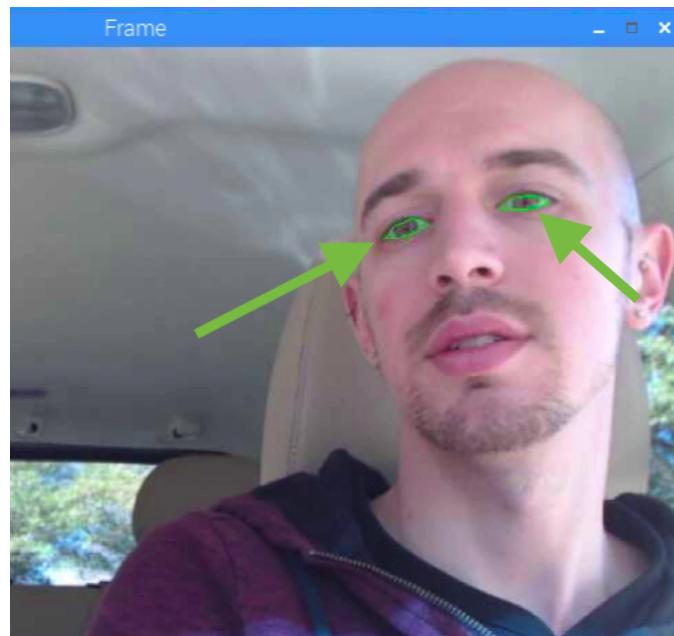


Figure 16.7: Drawing the convex hull of the eye facial landmark coordinates is very useful for debugging your script and ensuring it's working properly.

We can visualize each of the eye regions on our frame by using `cv2.drawContours` and supplying the `cv2.convexHull` calculation of each eye using **Lines 154-157** (Figure 16.7). These few lines are great for debugging our script, but aren't necessary if you are making an embedded product with no screen.

From here, we will check our Eye Aspect Ratio (`ear`) and frame counter (`conf["EYE_AR_THRESH"]`) to **see if the eyes are closed, while sounding the alarm to alert the drowsy driver if needed:**

---

```

159     # check to see if the eye aspect ratio is below the blink
160     # threshold
161     if ear < conf["EYE_AR_THRESH"]:
162         # increment the blink frame counter
163         blinkCounter += 1
164
165         # if the eyes were closed for a sufficient number of
166         # frames, then sound the alarm

```

---

```

167     if blinkCounter >= conf["EYE_AR_CONSEC_FRAMES"]:
168         # if the alarm is not on, turn it on
169         if not alarmOn:
170             alarmOn = True
171
172             # check to see if the TrafficHat buzzer should
173             # be sounded and red light set to blink
174             if conf["alarm"]:
175                 th.buzzer.blink(0.1, 0.1, 30,
176                                 background=True)
177                 th.lights.red.blink(0.1, 0.1, 30,
178                                 background=True)
179
180             # draw an alarm on the frame
181             cv2.putText(frame, "DROWSINESS ALERT! - eyes",
182                         (10, 60),
183                         cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
184
185             # otherwise, the eye aspect ratio is not below the blink
186             # threshold, so reset the counter and alarm
187             else:
188                 blinkCounter = 0
189                 alarmOn = False

```

---

On **Line 161** we check the `ear` against the `conf["EYE_AR_THRESH"]` — if it is less than the threshold (eyes are closed), we increment our `blinkCounter` (**Line 163**) and subsequently check it to see if the eyes have been closed for enough consecutive frames to sound the alarm (**Line 167**).

If the alarm isn't on, we turn it on (buzzer and blinking red light) for a few seconds to wake up the drowsy driver. This is accomplished on **Lines 169-178**.

Optionally (if you're implementing this code with a screen), you can draw the alarm on the frame as I have done on **Lines 181 and 182**.

That brings us to the case where the `ear` wasn't less than the threshold — in this case we reset our `blinkCounter` to 0 and make sure our alarm is turned off (**Lines 186-188**).

The next two code blocks compute the **Mouth Aspect Ratio (MAR)**, **count yawns**, and **sound an alarm if necessary**:

```

190     # extract the inner mouth coordinates, then use the
191     # coordinates to compute the mouth aspect ratio for the mouth
192     mouth = shape[mStart:mEnd]
193     mar = mouth_aspect_ratio(mouth)
194
195     # compute the convex hull for the left and right eye, then
196     # visualize each of the eyes
197     mouthHull = cv2.convexHull(mouth)
198     cv2.drawContours(frame, [mouthHull], -1, (0, 0, 255), 1)

```

---

Similar to EAR, the `mar` is calculated and the convex hull is visualized ([Lines 192-198](#)).

Now let's **count yawns and sound alarms when the driver is drowsy**:

---

```

200      # check to see if the mouth aspect ratio is above the yawn
201      # threshold
202      if mar > conf["MOUTH_AR_THRESH"]:
203          # increment the yawn frame counter and set the start time
204          yawnCounter += 1
205          startTime = datetime.now() if startTime == None else \
206                          startTime
207
208          # check to see if yawn frame counter is greater than yawn
209          # frame threshold and if the difference between current
210          # time and start time is less than or equal to yawn
211          # threshold time (in which case the person is drowsy)
212          if yawnCounter >= conf["YAWN_THRESH_COUNT"] and \
213              (datetime.now() - startTime).seconds <= \
214                  conf["YAWN_THRESH_TIME"]:
215              # if the alarm is not on, turn it on
216              if not alarmOn:
217                  alarmOn = True
218
219              # check to see if the TrafficHat buzzer should
220              # be sounded and red light set to blink
221              if conf["alarm"]:
222                  th.buzzer.blink(0.1, 0.1, 10,
223                                  background=True)
224                  th.lights.red.blink(0.1, 0.1, 30,
225                                  background=True)
226
227              # draw an alarm on the frame
228              cv2.putText(frame, "DROWSINESS ALERT! - yawning",
229                          (10, 85), cv2.FONT_HERSHEY_SIMPLEX, 0.7,
230                          (0, 0, 255), 2)
231
232          # check to see if the start time is set
233      elif startTime != None:
234          # check if the difference between current time and start
235          # time is greater than yawn threshold time
236          if (datetime.now() - startTime).seconds > \
237              conf["YAWN_THRESH_TIME"]:
238              # reset yawn counter, alarm flag and start time
239              yawnCounter = 0
240              alarmOn = False
241              startTime = None

```

---

Counting yawns is the same as counting blinks: every time a new frame is captured, the counter is incremented when there is a yawn ([Line 204](#)) or reset ([Line 239](#)) when the time has passed. Provided that the `yawnCounter` is both (1) above the count threshold, and (2) time is still less than or equal to the timing threshold ([Lines 212-214](#)), then we'll:

- Turn on the alarm (**Lines 216 and 217**).
- Sound the buzzer and blink the red light (**Lines 221-225**).
- Draw the alarm text on the frame (**Lines 228-230**).

Otherwise, we'll go ahead and reset the `yawnCounter`, `alarm`, and `startTime`.

Let's annotate and display our frame as the last steps of the long loop, which began back on **Line 97**:

---

```

243     # draw the computed aspect ratios on the frame
244     cv2.putText(frame, "EAR: {:.3f} MAR: {:.3f}".format(
245         ear, mar), (175, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.7,
246         (0, 0, 255), 2)
247
248     # if the 'display' flag is set, then display the current frame
249     # to the screen and record if a user presses a key
250     if conf["display"]:
251         cv2.imshow("Frame", frame)
252         key = cv2.waitKey(1) & 0xFF
253
254     # if the `q` key was pressed, break from the loop
255     if key == ord("q"):
256         break
257
258     # check to see if we have any open windows, and if so, close them
259     if conf["display"]:
260         cv2.destroyAllWindows()
261
262     # release the video stream pointer
263     vs.stop()
```

---

On **Lines 244-246** we annotate our frame with the EAR and MAR calculations for debugging purposes. This ends the if-statement which began on **Line 122**.

If we have a display and keyboard hooked up while we debug our program, **Lines 250-256** display the frame and wait for a “q” keypress to quit. You could also use the button on the TrafficHAT to start/stop drowsiness detection, but that is up to you.

The remaining lines close windows and release our video stream.

We did it! Let's check the results in the next section.

#### 16.3.4 Results

As you learned in the last section, computer vision scripts can become quite lengthy when you have status variables, counters, timers, and more in order to develop logic. Surprisingly, detect-

ing the landmarks was quite easy (thanks to Davis King's dlib) and calculating the eye/mouth aspect ratios was more straightforward in comparison.

To run this program, I recommend that you test it at your desk *before* installing it in your car. When you're ready (at your desk), hook up a camera and TrafficHAT to your RPi.

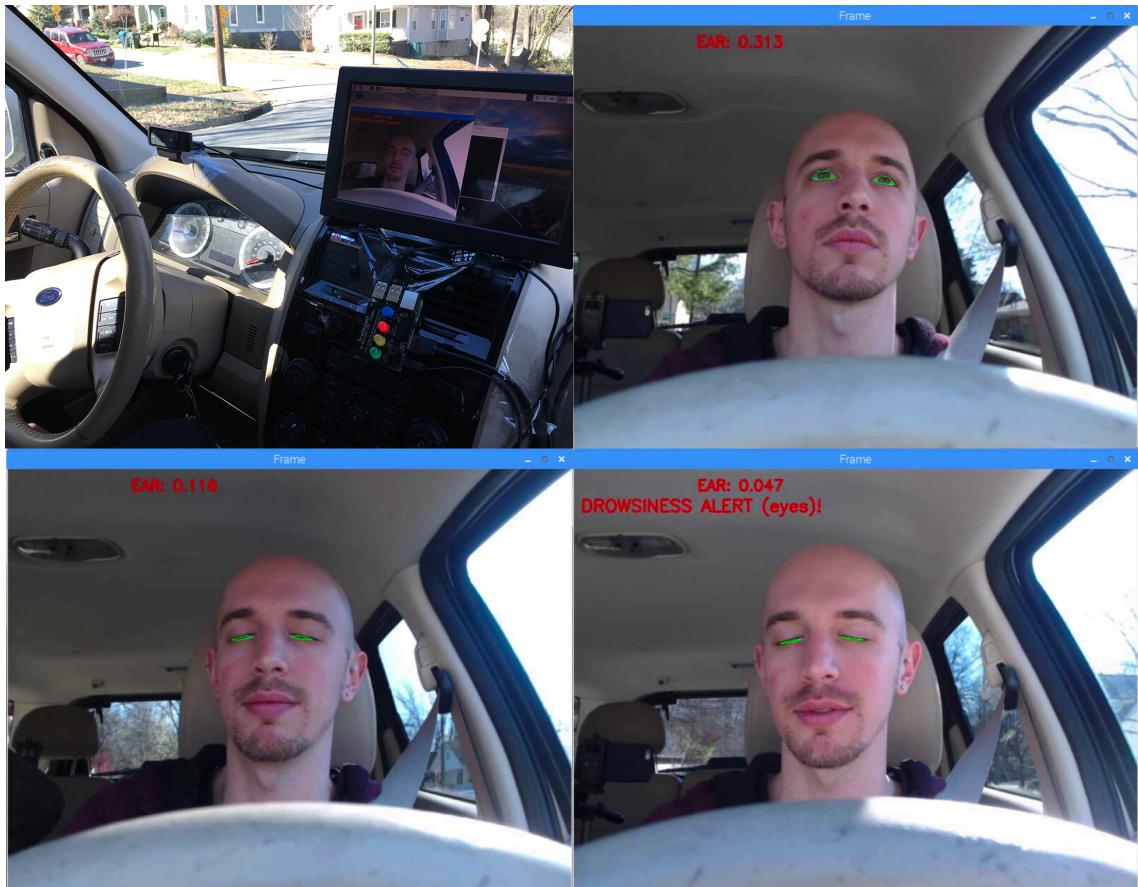


Figure 16.8: **Top-left:** Our drowsy driver detector setup includes an RPi, TrafficHAT (with buzzer/alarm), and monitor, all attached to the dashboard of our car. **Top-right:** Driving with my eyes open. **Bottom-left:** Closing my eyes. Notice how the EAR value drops considerably. **Bottom-right:** After my eyes have been marked as "closed" for a sufficient number of frames, an alarm is sounded to wake me up.

From there, enter the following command:

---

```
$ python detect_drowsiness.py --conf config/config.json
```

---

Figure 16.8 displays a few screen captures from our experimentation on the road.

- i. The *top-left* shows our in-car setup. Here we have attached a TrafficHAT to our RPi and then connected the RPi to a monitor. The monitor and TrafficHAT are taped to our

dashboard. We'll leave it up to you to decide if you want to mount the hardware on your car.

- ii. The *top-right* shows a view of my face as I'm driving a car. Notice how the eye landmarks are visualized by drawing the convex hull surrounding the eye location.
- iii. In the *bottom-left* you can see my eyes are now closed. Notice how the EAR value has dropped to 0.11.
- iv. After my eyes have been marked as "closed" for a sufficient number of frames, the alarm is sounded (*bottom-right*).

Additionally, you may want to refer to Chapter 8 so that the application will start when the RPi power comes on.

### 16.3.5 Tips and Suggestions

The following suggestions may go a long way toward your success with this project.

The camera works better when it is mounted below the face and angling up (for detecting open eyes). In other words, don't mount the camera on your visor or rear view mirror, because there won't be as good of a view of the open eye.

Everyone's EAR and MAR thresholds will be slightly different. Be sure to adjust the value in the config file. I challenge you to write an automatic calibration system that can be run for each new person (maybe when the button on the TrafficCHAT is pushed).

Lighting makes a big difference with face detection and facial landmark prediction. Ensure that you have adequate lighting. This program has not been tested at night with an IR light and NoIR camera, although that may be a logical next step for night-time drowsiness detection.

We had best results when testing with a Logitech C920 camera — the PiCamera will work, just not as well. Additionally the USB cable that comes with a webcam allows you more flexibility in where you mount the camera in your car.

Finally, due to noise in a video stream, subpar facial landmark detections, or fast changes in viewing angle, a simple threshold on the eye aspect ratio could produce a false-positive detection, reporting that a blink had taken place when in reality the person had not blinked.

To make our blink detector more robust to these challenges, Soukupová and Čech recommend: [65]

- i. Computing the eye aspect ratio for the  $N$ -th frame, along with the eye aspect ratios for  $N - 6$  and  $N + 6$  frames, then concatenating these eye aspect ratios to form a 13 dimensional feature vector.

- ii. Training a Support Vector Machine (SVM) on these feature vectors.

Soukupová and Čech report that the combination of the temporal-based feature vector and SVM classifier helps reduce false-positive blink detections and improves the overall accuracy of the blink detector.

Have fun and impress your friends, but be safe. As I've said before, I do not recommend solely relying on this system to keep you awake. I also don't recommend fumbling with a camera, screen, and RPi while your vehicle is in motion.

## 16.4 Summary

In this chapter, we learned how to optimize facial landmarks on the Raspberry Pi by swapping out a HOG + Linear SVM-based face detector for a Haar cascade.

Haar cascades, while less accurate, are significantly faster than HOG + Linear SVM detectors. Given the detections from the Haar cascade we were able to construct a `dlib.rectangle` object corresponding to the bounding box  $(x, y)$ -coordinates in the image. This object was fed into dlib's facial landmark predictor, which in turn gave us the set of localized facial landmarks on the face. From there, we applied the Eye Aspect Ratio (EAR) and Mouth Aspect Ratio (MAR) algorithms to determine when eyes and mouth are opened/closed.

Using the status of the eyes and mouth, we were able to develop logic to determine if someone is drowsy in front of the camera. When the person is drowsy, we sounded an audible alarm to wake the person up.

I hope you enjoyed this chapter as much as I did!



## Chapter 17

# What's a PID and Why do we need it?

Modern machines are able to operate completely independently and accomplish most tasks even better than a human can. We deploy machines in everyday life to solve problems. Sometimes we even send more advanced robots into places humans cannot (deep space, fathoms under sea level, small spaces, extreme temperatures, radioactive environments).

But how do we control a dynamic machine?

In some cases, it is as simple as turning a machine on and off when a sensor is triggered. Other cases require acceleration and dampening to reach a desired setpoint. Even more advanced machines and robots will use input from cameras for computer vision and deep learning to make so called "intelligent" decisions.

In this chapter we'll step outside the realm of computer vision, machine learning, and deep learning and enter an entirely new territory called "control theory", enabling us to deploy our machines to accomplish new goals. And of course, modern machines have cameras so this chapter will lay the foundation for future computer vision projects.

Let's begin!

### 17.1 Chapter Learning Objectives

In this chapter we will learn:

- i. The purpose of feedback control loops
- ii. The concept of a PID (with equation and code)
- iii. Our custom PID class
- iv. How to tune a PID

v. Alternatives and improvements to PIDs

vi. Further reading resources

## 17.2 The Purpose of Feedback Control Loops



Figure 17.1: HVAC (*left*) and thermostats (*right*) are systems that require a feedback loop. Thermostats read the temperature, and in turn, the HVAC system turns on or off based on the feedback from the thermostat.

Almost any machine that is controlled automatically requires a feedback loop.

Let's analyze a simple machine we're all familiar with: your heating, ventilation, and air conditioning system, more commonly known as HVAC (Figure 17.1).

Your HVAC is a system that is always working for you. When it is too hot in your house, the A/C comes on to cool you down. Similarly, when it is too cold, the heat comes on to warm you up. Some areas of the world can get away without either heat or A/C, but that's not the point. The point is that your HVAC is an example of a **feedback loop**.

The **thermostat** in your house is a sensor that reads the temperature. The **HVAC** residing nearby turns on and off based on the **feedback** from the thermostat.

Heating and air conditioning systems are most efficient when they run at a certain rate; therefore, for your house, they typically turn on and off to keep your system in a  $\pm$  degree range before turning on again. This is known as "**fuzzy logic**".

**Remark.** We will put the concept of fuzzy logic to use in the Hacker Bundle to control a robot that follows a line that it sees with a camera.

Other examples of feedback control loops which impact your life are:

- Your laptop/smartphone screen dims/brightens automatically depending on the ambient light.

- Your oven heats automatically to a temperature setpoint and then levels off to cook your food at the exact right temperature.
- A satellite maintains proper orbit and alignment so that a communication link is maintained and/or so that an imaging system works properly.
- Nuclear power plant control rods are raised and lowered in the reactor impacting the speed of the steam turbine so that the energy grid receives more or less power.

No matter how simple or complex an unmanned machine is, there's likely at least one control loop, and possibly many more than just one.

## 17.3 PID Control Loops

In this section, we'll discuss a fundamental control loop algorithm: The PID.

We'll review how the algorithm works conceptually and implement one in a Python class. From there, we'll discuss how to tune your PID with three key constants. We'll discuss a handful of alternatives and improvements to the PID algorithm as well.

Let's begin.

### 17.3.1 The Concept Behind Proportional Integral Derivative Controllers

A "Proportional Integral Derivative" controller (commonly known as a PID) is one of the most fundamental control loops. PIDs go further than on and off fuzzy logic. A PID actually can control machines that require outputs to be "smoothened" and "dampened".

Let's consider the concept of an unmanned airplane or quadcopter (also known as a "drone"). There are obviously multiple control loops at play here, such as one that keeps the drone following a GPS course. Another control loop that maintains a specific elevation above the ground or above sea level. In the case of a quadcopter, multiple PIDs keep the device stable.

For this example, let's only consider elevation (the simplest control loop of those mentioned).

What is the "process variable" or sensor? It could be a pressure transducer readout. Or a radar that pings the ground to determine distance. Or even the output of a GPS system which uses triangulation from satellites to determine the elevation.

No matter which sensor is used, let's just call **elevation** our "process variable". We have a goal of maintaining an elevation of 300m in the sky so we can take footage of agricultural fields.

How can we design a control loop such that our drone will go up or down automatically?

The answer is in the form of a PID (although there are more complex control loops we will discuss later). Figure 17.2 provides great drawing of the concept of a PID controller.

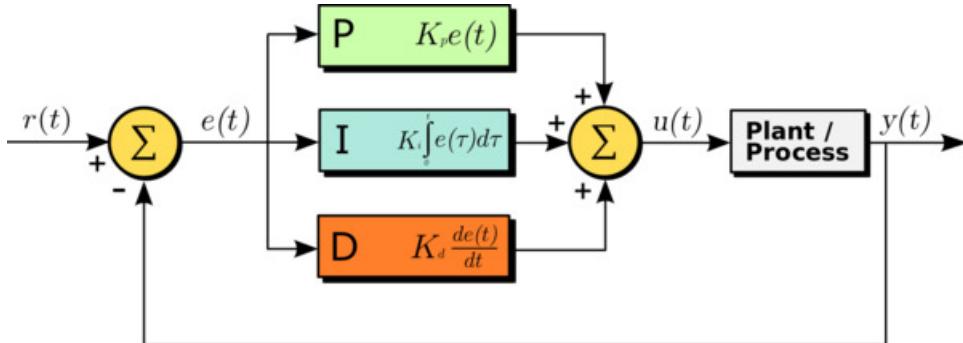


Figure 17.2: A Proportional Integral Derivative (PID) control loop will be used for each of our panning and tilting processes [67].

Notice how the output loops back into the input. This feedback loop allows our algorithm to compute large and small corrections automatically to reach our setpoint (300m elevation). The PID controller (also known as “three term controller”) calculates an error term (the difference between a desired set point and sensor reading) and has a goal of compensating for the error.

What this means is that as our drone climbs in elevation from takeoff, it is probably at max thrust until it nears the setpoint of 300m. Then it will dampen and ideally not overshoot the 300m target. At this point, our drone seeks to maintain 300m above the ground while it flies its course to collect images. When it goes to high, the PID will drive it down; likewise if it is too low. The elevation PID will constantly help the drone to battle the effects of wind whereas a completely separate PID will ensure that the drone drives to its waypoints, in the most optimal flight path.

The following equation matches the figure:

$$u(t) = K_p e(t) + K_i \int_0^t e(t') dt' + K_d \frac{de(t)}{dt} \quad (17.1)$$

The Proportional, Integral, and Derivative terms are calculated based on the error and summed. The result is an output known as a “process” (an electromechanical process, not what us computer science/software engineer types think of as a “computer process”). The sensor output is known as the “process variable” and serves as constant feedback input to the equation. Throughout the feedback loop, timing is captured and it is input to the equation as well.

Let's review what **P**, **I**, and **D** mean:

- **P (proportional):** If the *current* error is large, the output will be proportionally large to cause a significant correction.

- **I (integral):** *Historical* values of the error are integrated over time. Less significant corrections are made to reduce the error. If the error is eliminated, this term won't grow.
- **D (derivative):** This term anticipates the *future*. In effect, it is a dampening method. If either P or I will cause a value to overshoot (i.e. a servo was turned past an object or a steering wheel was turned too far), D will dampen the effect before it gets to the output.

Be sure to keep those points in mind when it comes to tuning your PID constants later in this chapter.

### 17.3.2 Our Custom PID Class

Using the equation of a PID, it is quite straightforward to implement a PID Python class. The PID class demonstrated in this section can be re-used in many projects with little to no modification. Go ahead and open a new file named `pid.py` and insert the following code:

---

```

1 # import necessary packages
2 import time
3
4 class PID:
5     def __init__(self, kP=1, kI=0, kD=0):
6         # initialize gains
7         self.kP = kP
8         self.kI = kI
9         self.kD = kD
10
11    def initialize(self):
12        # initialize the current and previous time
13        self.currTime = time.time()
14        self.prevTime = self.currTime
15
16        # initialize the previous error
17        self.prevError = 0
18
19        # initialize the term result variables
20        self.cP = 0
21        self.cI = 0
22        self.cD = 0

```

---

Our only import is `time` on **Line 2** — the rest of our code is just simple math.

**Lines 4-9** define our `PID` class and constructor. The three parameters our constructor accepts are our PID gain constants: `kP`, `kI`, and `kD`. These values are then assigned on **Lines 7-9**.

Our `initialize` method begins on **Line 11**. Inside the function, the following values are initialized:

- currTime: The current timestamp.
- prevTime: The timestamp for the most recent update.
- prevError: We'll be calculating deltaError for each new update, so we'll need to know the previous update's error value.
- kP, kI, and kD: Our PID term result variables. We'll calculate these in the update method.

From here, let's move to the heart of our PID class, the update method:

---

```

24     def update(self, error, sleep=0.2):
25         # pause for a bit
26         time.sleep(sleep)
27
28         # grab the current time and calculate delta time
29         self.currTime = time.time()
30         deltaTime = self.currTime - self.prevTime
31
32         # calculate the delta error
33         deltaError = error - self.prevError
34
35         # calculate the proportional term
36         self.cP = error
37
38         # calculate the integral term
39         self.cI += error * deltaTime
40
41         # calculate the derivative term (and prevent divide by zero)
42         self.cD = (deltaError / deltaTime) if deltaTime > 0 else 0
43
44         # save previous time and error for the next update
45         self.prevTime = self.currTime
46         self.prevError = error
47
48         # sum the terms and return
49         return sum([
50             self.kP * self.cP,
51             self.kI * self.cI,
52             self.kD * self.cD])

```

---

Our update method accepts two parameters: (1) the current `error` value and (2) an amount to `sleep` in seconds.

Inside the `update` method, we:

- Sleep for a predetermined amount of time on **Line 26**, thereby preventing updates so fast that our electro-mechanical actuator can't respond fast enough. The sleep value

should be chosen wisely based on knowledge of mechanical, computational, and even communication protocol limitations. Without prior knowledge, you should experiment to see what works best.

- Calculate `deltaTime` (**Line 30**). Updates won't always come in at the exact same time (we have no control over the RPi since it doesn't run an RTOS). Thus, we calculate the time difference between the previous update and now (this current update). This will affect our `cI` and `cD` terms.
- Compute `deltaError`, the difference between the provided `error` and `prevError` (**Line 33**).

We then calculate our PID control terms based on our equation:

- `cP`: Our proportional term is equal to the `error` term (**Line 36**).
- `cI`: Our integral term is simply the `error` multiplied by `deltaTime` (**Line 33**).
- `cD`: Our derivative term is `deltaError` over `deltaTime`. Division by zero is accounted for (**Line 42**).

We then set the `prevTime` and `prevError` (**Lines 45 and 46**). We'll need these values during our next `update`. Finally, we'll return the summation of calculated terms multiplied by constant terms (**Lines 49-52**).

Keep in mind that updates will be happening in a fast-paced loop. Depending on your needs, you should adjust the `sleep` parameter when you call the method.

To implement a `PID` object into your project, I recommend using threading or multiprocessing (as we will in this book). Once the object is instantiated, you should reserve a thread or process for the PID updates. I recommend implementing thread-safe or process-safe variables for both the PID "process" (the output) and your PID "process variable" (the feedback value which could be from a sensor or a computer vision pipeline in our case).

When a new "process variable" reading is ready, simply call `update` and you'll have your "process" value to output to your hardware.

**Remark.** *The Python class presented in this section was based on Wikipedia's pseudocode [68] as well as Erle Robotics GitBook [69]. It is formatted in the style PylImageSearch readers have come to expect.*

### 17.3.3 How to Tune a PID

If you've been following along carefully, you noticed that our PID controller requires three constants known as "gains". These three values in our `PID` class are `kP`, `kI`, and `kD`. An obvious

question you probably have is:

*"How can I determine the values for the constants?"*

The secret sauce to a PID is not the code or the equation. The code and equation have been proven time and time again, for many years. Instead, the most important part of the PID controller lies in the gain constants.

There are two options for finding these constants: (1) Manually, and (2) Automatically.

Don't jump so quickly to automatic methods. This requires additional code complexity and in some scenarios, it isn't possible. While manual tuning can be a pain, it is what will lead you to fully respect the PID algorithm and become intimate with how it works.

The basic manual process outlined in Wikipedia's writeup [70] follows:

- i. Set  $k_I$  and  $k_D$  to zero.
- ii. Increase  $k_P$  from zero until the output oscillates (i.e. the servo process bounces back and forth around the setpoint). Then set the value to half.
- iii. Increase  $k_I$  until offsets are corrected quickly, knowing that too high of a value will cause instability.
- iv. Increase  $k_D$  until the output settles on the desired output reference quickly after a load disturbance (i.e. if you introduce significant error manually and quickly). Too much  $k_D$  will cause excessive response and make your output overshoot where it needs to be.

**I cannot stress this enough:** *Make small changes while tuning.* If your machine has multiple PIDs, you should also only tune one PID at a time, if at all possible. We will walk through the process of tuning two PIDs for ***pan/tilt face tracking*** in Chapter 18.

#### 17.3.4 Alternatives and Improvements to PIDs

PIDs help us conquer machine control in many ways. As with any algorithm, there are always new and creative ways people find to push the limits or to shortcut and simplify. Modifications to the PID algorithm that you may encounter upon further study include:

- PI controller (eliminating D)
- Feed-forward (in addition to feedback)
- Integral Windup improvements (initializations, bounding, and back-calculating to reduce overshooting)

- Derivative Kick accommodations (methods to reduce the undesirable spike in output)

I encourage you to dive into these modifications on your own after you've implemented a pan/tilt face tracker in Chapter 18.

Alternatives to improve your machine and robot control may include (1) LQR Controller ("Linear Quadratic Regulator"), (2) Kalman filtering (also known as "Linear Quadratic Estimation"), and more!

Self driving cars and drones also rely on "sensor fusion". Many sources of data for a self-driving car might include RADAR, LIDAR, cameras, and GPS. Our drone example earlier in the chapter might lead to a drone with three sensors for estimating altitude: pressure transducer, GPS, and RADAR. Kalman filters are great when you have multiple sensor sources, especially if any of them are particularly noisy. A benefit of PIDs is that you can cascade PIDs together. The output of one PID becomes the input to another for special applications.

### 17.3.5 Further Reading

Control theory is a large subject area within electrical, computer, and software engineering. Thus, there are countless books and online resources on the subject. Some are heavy on mathematics, some conceptual. Some are easy to understand, some not. A number of resources are domain-specific as well.

That said, as a software programmer, you just need to know how to implement and tune the algorithm at hand. In the case of a PID, **even if you think the mathematical equation looks complex, when you see the code, you should be able to follow and understand.**

PIDs are easier to tune if you understand how they work, *but as long as you follow the manual tuning guidelines discussed in this chapter*, you don't have to be intimate with the equation at all times.

**Just remember:**

- **P** – proportional, "present" (large corrections)
- **I** – integral, "in the past" (historical)
- **D** – derivative, "dampening" (anticipates the future)

For more information, the Wikipedia PID controller page [67] is really great and also links to other great guides in its references.

With control theory, the possibilities are truly endless. You may wish to enroll in a bachelor's or master's level class if you are interested in theory. Nothing beats hands-on, practical tinkering and experimentation as well.

## 17.4 Summary

In this chapter we stepped outside the realm of computer vision, learning about control systems and theory. More specifically we learned about the Proportional Integral Derivative (PID) feedback control loop.

PIDs and other control algorithms are prevalent in many areas of our daily lives and whether we realize it or not, they are all around us. There are multiple systems in a car or airplane that use PIDs, even those where a driver or pilot is also controlling the vehicle.

Of course, PIDs barely scratch the surface of the complex field of “Control Theory”, but a basic understanding the PID allows computer vision practitioners to accomplish fun projects.

In the next chapter, we’ll apply two PID control loops to pan/tilt face tracking project!

## Chapter 18

# Face Tracking with Pan/Tilt Servos

Ever since I started working with computer vision, I thought it would be really cool to have a camera track objects. We've accomplished this on the PyImageSearch blog with object detection and tracking algorithms (<http://pyimg.co/aiqr5>) [71].

But what happens if the object (ex. person, dog, cat, horse, etc.), goes out of the frame?

In that case, usually there is nothing we can do. That is, unless we add mechanics to our camera. There are certainly plenty of pan/tilt security cameras on the market, but usually they are manually controlled by an operator.

Luckily for us, there's a great HAT for us made by Pimoroni which makes *automatic* pan/tilt tracking possible using the Pi Camera. In this chapter we're going to use the Raspberry Pi pan/tilt servo HAT and PIDs to track a moving target using servo mechanics.

### 18.1 Chapter Learning Objectives

In this chapter, we'll apply and reinforce our knowledge of PIDs from Chapter 17 to track objects. We will learn about the following concepts to accomplish our goal:

- i. Multi-processing
- ii. Process-safe variables
- iii. The Haar Cascade face detector
- iv. Tuning our PID gain constants

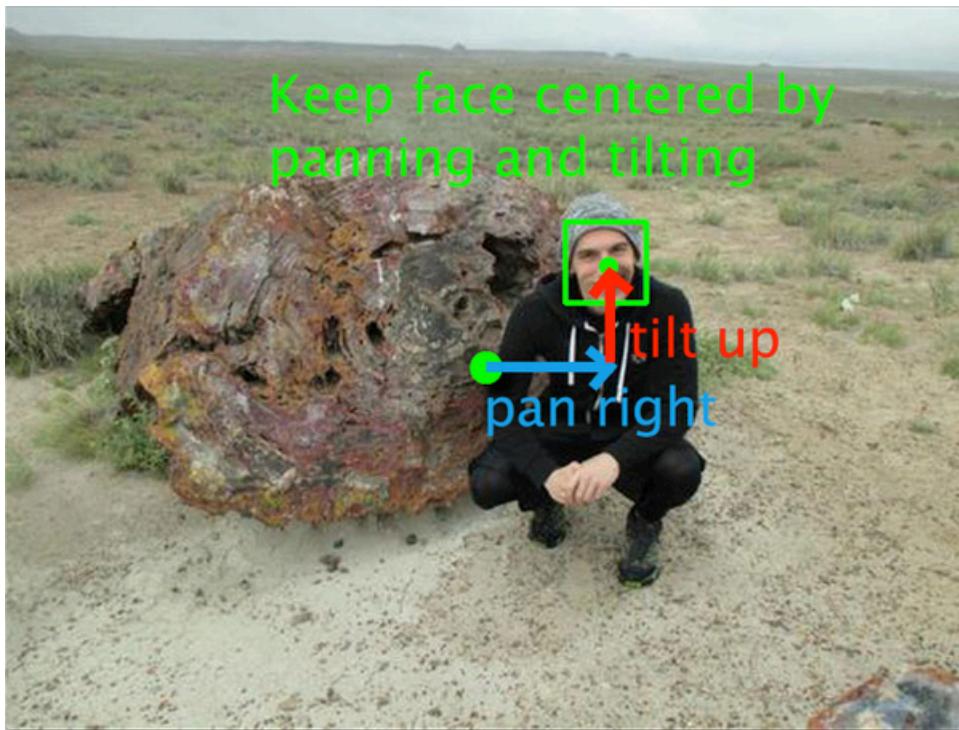


Figure 18.1: The goal of pan/tilt face tracking is to use mechanical servo actuators to follow a moving face in a scene while keeping the face centered in the view of the camera. The following steps take place: (1) the face detector localizes the face, (2) the PID process accepts an error between where the face is and where it should be, (3) the PID update method calculates a new angle to send to the servo, (4) rinse and repeat. These steps are performed by two independent processes for both panning and tilting.

## 18.2 Pan/tilt Face Tracking

In the first part of this chapter, we'll discuss the hardware requirements. Then we'll briefly review the concept of a PID. From there, we'll dive into our project structure. Our object center algorithm will find the object we are tracking. For this example, it is a face, but don't limit your imagination to tracking only faces.

We'll then walk through the script while coding each of our processes. We have four:

- i. **Object center** - Finds the face
- ii. **PID A** - Panning
- iii. **PID B** - Tilting
- iv. **Set servos** - Takes the output of the PID processes and tells each servo the angle it needs to steer to

Finally, we'll tune our PIDs independently and deploy the system.

### 18.2.1 Hardware Requirements



Figure 18.2: The PiMoroni Pan-Tilt HAT kit for a Raspberry Pi (<http://pyimg.co/lh4on>).

The goal of pan and tilt object tracking is for the camera to **stay centered upon an object**. Typically this tracking is accomplished with two servos. In our case, we have one servo for ***panning left and right***. We have a separate servo for ***tilting up and down***. Each of our servos and the fixture itself has a range of 180 degrees (some systems have a greater or lesser range than this).

You will need the following hardware to replicate this project:

- **Raspberry Pi** – I recommend the 3B+ or greater, but other models may work provided they have the same header pin layout.
- **PiCamera** – I recommend the PiCamera V2.
- **Pimoroni pan tilt HAT full kit** (<http://pyimg.co/lh4on>) The Pimoroni kit is a quality product and it hasn't let me down. Budget about 30 minutes for assembly. The SparkFun kit (<http://pyimg.co/tsny0>) would work as well, but it requires a soldering iron and additional assembly. Go for the Pimoroni kit if at all possible.
- **2.5A, 5V power supply** – If you supply less than 2.5A, your Pi might not have enough current, causing it to reset. Why? Because the servos draw necessary current away. Grab a **2.5A** power supply and dedicate it to this project hardware.

- **HDMI Screen** – Placing an HDMI screen next to your camera as you move around will allow you to visualize and debug, essential for manual tuning. Do not try X11 forwarding — it is simply too slow for video applications. VNC is possible if you don't have an HDMI screen.
- **Keyboard/mouse** - To accompany your HDMI screen.

### 18.2.2 A Brief Review on PIDs

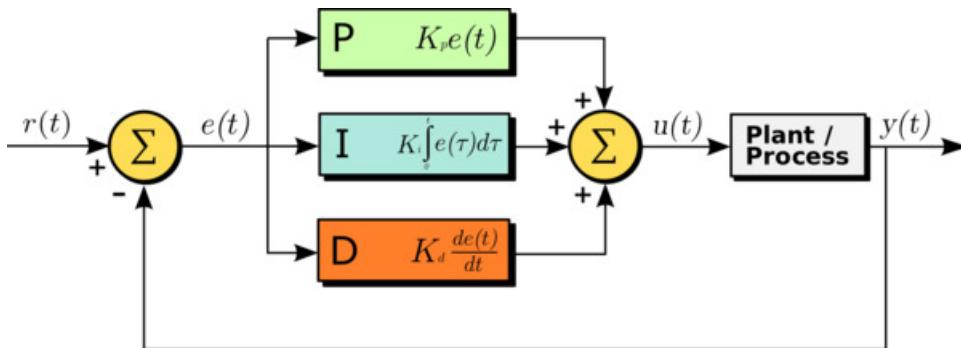


Figure 18.3: A Proportional Integral Derivative (PID) control loop will be used for each of our panning and tilting processes [67].

Be sure to refer to Chapter 17 for a thorough explanation of the Proportional Integral Derivative control loop (Figure 18.3). Let's review the essential concepts.

A PID has a target setpoint for the “process”. The “process variable” is our feedback mechanism. Corrections are made to ensure that the process variable adjusts to meet the setpoint.

In this chapter, our “process variable” for panning is the  $x$ -coordinate of the center of the face. Our “process variable” for tilting is the  $y$ -coordinate of the center of the face. We will use the `PID` class and call the `.update` method inside of two independent control loops, to update the angles of both servos.

The result and goal are that our servos will move to keep the face centered in the frame. Recall the following when adjusting your PID gain constants:

- **P** – proportional, "present" (large corrections)
- **I** – integral, "in the past" (historical)
- **D** – derivative, "dampening" (anticipates the future)

### 18.2.3 Project Structure

Let's review the project structure:

---

```
|-- pyimagesearch
|   |-- __init__.py
|   |-- objcenter.py
|   |-- pid.py
|-- haarcascade_frontalface_default.xml
|-- home.py
|-- pan_tilt_tracking.py
```

---

In the remainder of this chapter, we'll be reviewing three Python files:

- `objcenter.py` : Calculates the center of a face bounding box using the Haar Cascade face detector. If you wish, you may detect a different type of object and place the logic in this file.
- `pan_tilt_tracking.py` : This is our pan/tilt object tracking driver script. It uses multi-processing with four independent processes (two of which are for panning and tilting, one is for finding an object, and one is for driving the servos with fresh angle values).

The `haarcascade_frontalface_default.xml` file is our pre-trained Haar Cascade face detector. Haar works great with the RPi in this context as it is much faster than HOG or Deep Learning. If your object detection method is not fast enough, then your camera tracking will lag behind the target.

#### 18.2.4 Implementing the Face Detector and Object Center Tracker

The goal of our pan and tilt tracker will be to keep the camera centered on the object itself as shown in Figure 18.1. To accomplish this goal, we need to (1) detect the object itself, and (2) compute the center ( $x$ ,  $y$ )-coordinates of the object.

Let's go ahead and implement our `ObjCenter` class inside `objcenter.py` which will accomplish both goals:

---

```
1 # import necessary packages
2 import cv2
3
4 class ObjCenter:
5     def __init__(self, haarPath):
6         # load OpenCV's Haar cascade face detector
7         self.detector = cv2.CascadeClassifier(haarPath)
```

---

We begin by importing OpenCV. Our `ObjCenter` class is defined on **Line 4**.

The constructor accepts a single argument — the path to the Haar Cascade face detector. Again, we're using the Haar method to find faces. Keep in mind that the RPi (even a 3B+ or

4) is a resource-constrained device. If you elect to use a slower (but more accurate) HOG or a CNN, the camera may not keep up with the moving object. You'll want to slow down the PID calculations so they aren't firing faster than you're actually detecting new face coordinates.

**Remark.** You may also elect to use a Movidius NCS or Google Coral TPU USB Accelerator for deep learning face detection with the Raspberry Pi. Refer to the Hacker Bundle and Complete Bundle for examples on using these hardware accelerators.

The `detector` itself is initialized on **Line 7**.

Let's define the `update` method which **finds the center** ( $x, y$ )-coordinate of a face:

---

```

9  def update(self, frame, frameCenter):
10     # convert the frame to grayscale
11     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
12
13     # detect all faces in the input frame
14     rects = self.detector.detectMultiScale(gray, scaleFactor=1.05,
15                                            minNeighbors=9, minSize=(30, 30),
16                                            flags=cv2.CASCADE_SCALE_IMAGE)
17
18     # check to see if a face was found
19     if len(rects) > 0:
20         # extract the bounding box coordinates of the face and
21         # use the coordinates to determine the center of the
22         # face
23         (x, y, w, h) = rects[0]
24         faceX = int(x + (w / 2.0))
25         faceY = int(y + (h / 2.0))
26
27         # return the center (x, y)-coordinates of the face
28         return ((faceX, faceY), rects[0])
29
30     # otherwise no faces were found, so return the center of the
31     # frame
32     return (frameCenter, None)

```

---

Our project has *two update methods* so let's review the difference:

- i. We previously reviewed the the `PID` class `update` method in Chapter 17. This method executes the PID algorithm to help calculate a new servo angle to keep the face/object in the center of the camera's field of view.
- ii. Now we are reviewing the `ObjCenter` class `update` method. This method simply finds a face and returns its center current coordinates.

The `update` method (for finding the face) is defined on **Line 9** and accepts two parameters:

- `frame`: An image ideally containing one face
- `frameCenter`: The center coordinates of the frame

**Line 12** converts the `frame` to grayscale (a preprocessing step for Haar object detection).

From there we **perform face detection** using the Haar Cascade `detectMultiScale` method. On **Lines 19-25** we check that faces have been detected and from there calculate the center ( $x, y$ )-coordinates of the face itself. **Lines 19-23** make an important assumption: we assume that only *one* face is in the `frame` at all times and that face can be accessed by the 0-th index of `rects`.

**Remark.** *Without this assumption holding true, additional logic would be required to determine which face to track. See the section below on “Improvements for pan/tilt face tracking with the Raspberry Pi”, where I describe how to handle multiple face detections with Haar.*

The center of the face, as well as the bounding box coordinates, are returned on **Line 28**. We'll use the bounding box coordinates to draw a box around the face for display purposes.

Otherwise, when no faces are found, we simply return the center of the frame on **Line 32** (ensuring that the servos stop and do not make any corrections until a face is found again).

### 18.2.5 The Pan and Tilt Driver Script

Now that we have a system in place to find the center of a face as well as our PID class, let's work on the driver script to tie it all together. Go ahead and create a new file called `pan_tilt_tracking.py` and insert these lines:

---

```

1 # import necessary packages
2 from pyimagesearch.objcenter import ObjCenter
3 from pyimagesearch.pid import PID
4 from multiprocessing import Manager
5 from multiprocessing import Process
6 from imutils.video import VideoStream
7 import pantilthat as pth
8 import argparse
9 import signal
10 import time
11 import sys
12 import cv2
13
14 # define the range for the motors
15 servoRange = (-90, 90)

```

---

On **Line 2-12** we import necessary libraries:

- Process and Manager will help us with multiprocessing and shared variables.
- VideoStream will allow us to grab frames from our camera
- ObjCenter will help us locate the object in the frame, while PID will help us keep the object in the center of the frame by calculating our servo angles
- pantilthat is the library used to interface with the RPi's Pimoroni pan tilt HAT

Our servos on the pan tilt HAT have a range of 180 degrees (-90 to 90) as is defined on **Line 15**. These values should reflect the limitations of your servos or any soft limits you'd like to put in place (i.e. if you don't want your camera to aim at the floor, wall, or ceiling).

Let's define a “ctrl + c” signal\_handler:

---

```

17 # function to handle keyboard interrupt
18 def signal_handler(sig, frame):
19     # disable the servos and gracefully exit
20     print("[INFO] You pressed `ctrl + c`! Exiting...")
21     pth.servo_enable(1, False)
22     pth.servo_enable(2, False)
23     sys.exit()

```

---

This multiprocessing script can be tricky to exit from. There are a number of ways to accomplish it, but I decided to go with a signal\_handler approach (first introduced in Chapter 7).

Inside the signal handler, **Line 20** prints a status message, **Lines 21 and 22** disable our servos, and **Line 23** exits from our program.

You might look at this script as a whole and think “*If I have four processes, and signal\_handler is running in each of them, then this will occur four times.*”

You are absolutely right, but this is a compact and understandable way to go about killing off our processes, short of pressing “ctrl + c” as many times as you can in a sub-second period to try to get all processes to die off. Imagine if you had 10 processes and were trying to kill them with the “ctrl + c” approach. There are certainly better approaches presented online, and I encourage you to investigate them.

Now that we know how our processes will exit, let's define our first process:

---

```

25 def obj_center(args, objX, objY, centerX, centerY):
26     # set the signal trap to handle keyboard interrupt and then
27     # initialize the object center finder
28     signal.signal(signal.SIGINT, signal_handler)
29     obj = ObjCenter(args["cascade"])

```

---

```

30
31     # start the video stream and wait for the camera to warm up
32     vs = VideoStream(usePiCamera=True).start()
33     time.sleep(2.0)
34
35     # loop indefinitely
36     while True:
37         # grab the frame from the threaded video stream and flip it
38         # vertically (since our camera was upside down)
39         frame = vs.read()
40         frame = cv2.flip(frame, 0)
41
42         # calculate the center of the frame as this is (ideally) where
43         # we will we wish to keep the object
44         (H, W) = frame.shape[:2]
45         centerX.value = W // 2
46         centerY.value = H // 2
47
48         # find the object's location
49         objectLoc = obj.update(frame, (centerX.value, centerY.value))
50         (objX.value, objY.value), rect) = objectLoc
51
52         # extract the bounding box and draw it on the frame
53         if rect is not None:
54             (x, y, w, h) = rect
55             cv2.rectangle(frame, (x, y), (x + w, y + h),
56                           (0, 255, 0), 2)
57
58         # display the frame to the screen
59         cv2.imshow("Pan-Tilt Face Tracking", frame)
60         cv2.waitKey(1)

```

---

Our `obj_center` thread begins on **Line 29** and accepts five parameters:

- `args`: Our command line arguments dictionary (created in our main thread).
- `objX` and `objY`: The  $(x, y)$ -coordinates of the object. We'll continuously calculate these values.
- `centerX` and `centerY`: The center of the frame.

On **Line 28** we start our `signal_handler`.

Our `ObjCenter` is instantiated as `obj` on **Line 29**. Our cascade path is passed to the constructor.

Then, on **Lines 32 and 33**, we start our `VideoStream` for our PiCamera, allowing it to warm up for two seconds.

From here, our process enters an infinite loop on **Line 36**. The only way to escape out of the loop is if the user types “*ctrl + c*” (you’ll notice the lack of no `break` statement).

Our `frame` is grabbed and flipped on **Lines 39 and 40**. We must flip the `frame` because the PiCamera is physically upside down in the pan tilt HAT fixture by design.

**Lines 44-46** set our frame width and height as well as calculate the center point of the frame. You'll notice that we are using `.value` to access our center point variables — this is required with the `Manager` method of sharing data between processes.

To calculate where our object is, we'll simply call the `update` method on `obj` while passing the video `frame`. The reason we also pass the center coordinates is because we'll just have the `ObjCenter` class return the frame center if it doesn't see a Haar face. Effectively, this makes the PID error 0 and thus, the servos stop moving and remain in their current positions until a face is found.

**Remark.** *I choose to return the frame center if the face could not be detected. Alternatively, you may wish to return the coordinates of the last location a face was detected. That is an implementation choice that I leave up to you.*

The result of the `update` (**Line 49**) is parsed on **Line 50** where our object coordinates and the bounding box are assigned.

The last steps are to (1) draw a rectangle around our face and (2) display the video frame (**Lines 53-60**).

Let's define our next process, `pid_process`:

---

```

62 def pid_process(output, p, i, d, objCoord, centerCoord):
63     # set the signal trap to handle keyboard interrupt, then create a
64     # PID and initialize it
65     signal.signal(signal.SIGINT, signal_handler)
66     p = PID(p.value, i.value, d.value)
67     p.initialize()
68
69     # loop indefinitely
70     while True:
71         # calculate the error and update the value
72         error = centerCoord.value - objCoord.value
73         output.value = p.update(error)

```

---

Our `pid_process` is quite simple as the heavy lifting is taken care of by the PID class. Two of these processes will be running at any given time (panning and tilting). If you have a complex robot, you might have many more PID processes running. The method accepts six parameters:

- `output`: The servo angle that is calculated by our PID controller. This will be a pan or tilt angle.

- `p`, `i`, and `d`: Our three PID constants.
- `objCoord`: This value is passed to the process so that the process has access to keep track of where the object is. For panning, it is an `x`-coordinate. For tilting, it is a `y`-coordinate.
- `centerCoord`: Used to calculate our `error`, this value is just the center of the frame (either `x` or `y` depending on whether we are panning or tilting).

*Be sure to trace each of the parameters back to where the process is started in the main thread of this program.*

On **Line 65**, we start our `signal_handler`.

Then we instantiate our `PID` on **Line 65**, passing each of the `P`, `I`, and `D` values. Subsequently, the `PID` object is initialized (**Line 67**).

Now comes the fun part in just two lines of code:

- i. Calculate the `error` on **Line 72**. For example, this could be the frame's `y`-center minus the object's `y`-location for tilting.
- ii. Call `update` (**Line 73**), passing the new error (and a sleep time if necessary). The returned value is the `output.value` (an angle in degrees).

We have another thread that “watches” each `output.value` to drive the servos. Speaking of driving our servos, let's implement a servo range checker and our servo driver now:

---

```

75 def in_range(val, start, end):
76     # determine the input vale is in the supplied range
77     return (val >= start and val <= end)
78
79 def set_servos(pan, tlt):
80     # set the signal trap to handle keyboard interrupt
81     signal.signal(signal.SIGINT, signal_handler)
82
83     # loop indefinitely
84     while True:
85         # the pan and tilt angles are reversed so multiply by -1
86         panAngle = -1 * pan.value
87         tltAngle = -1 * tlt.value
88
89         # if the pan angle is within the range, pan
90         if in_range(panAngle, servoRange[0], servoRange[1]):
91             pth.pan(panAngle)
92
93         # if the tilt angle is within the range, tilt

```

---

```
94     if in_range(tltAngle, servoRange[0], servoRange[1]):  
95         pth.tilt(tltAngle)
```

---

**Lines 75-77** define an `in_range` method to determine if a value is within a particular range.

From there, we'll drive our servos to specific pan and tilt angles in the `set_servos` method. This method will be running in another process. It accepts `pan` and `tlt` values, and will watch the values for updates. The values themselves are constantly being adjusted via our `pid_process`.

We initialize our `signal_handler` for this process on **Line 81**.

From there, we'll start our infinite loop until a signal is caught:

- Our `panAngle` and `tltAngle` values are made negative to accommodate the orientation of the servos and camera (**Lines 86 and 87**)
- Then we check each value ensuring it is in the range as well as drive the servos to the new angle (**Lines 90-95**)

That was easy.

Now let's parse command line arguments:

---

```
97 # check to see if this is the main body of execution  
98 if __name__ == "__main__":  
99     # construct the argument parser and parse the arguments  
100    ap = argparse.ArgumentParser()  
101    ap.add_argument("-c", "--cascade", type=str, required=True,  
102                    help="path to input Haar cascade for face detection")  
103    args = vars(ap.parse_args())
```

---

The main body of execution begins on **Line 98**.

We parse our command line arguments on **Lines 100-103**. We only have one — the path to the Haar Cascade on disk.

Now let's work with process-safe variables and start each independent process:

---

```
105    # start a manager for managing process-safe variables  
106    with Manager() as manager:  
107        # enable the servos  
108        pth.servo_enable(1, True)  
109        pth.servo_enable(2, True)  
110
```

---

---

```

111      # set integer values for the object center (x, y)-coordinates
112      centerX = manager.Value("i", 0)
113      centerY = manager.Value("i", 0)
114
115      # set integer values for the object's (x, y)-coordinates
116      objX = manager.Value("i", 0)
117      objY = manager.Value("i", 0)
118
119      # pan and tilt values will be managed by independent PIDs
120      pan = manager.Value("i", 0)
121      tlt = manager.Value("i", 0)

```

---

Inside the Manager block, our process safe variables are established. We have quite a few of them.

First, we enable the servos on **Lines 108 and 109**. Without these lines, the hardware won't work.

Let's look at our first handful of process safe variables:

- The frame center coordinates are integers (denoted by "i") and initialized to 0 (**Lines 112 and 111**).
- The object center coordinates, also integers and initialized to 0 (**Lines 116 and 117**).
- Our pan and tlt angles (**Lines 120 and 121**) are integers that I've set to start in the center, pointing towards a face (angles of 0 degrees).

Now we'll set the P, I, and D constants:

---

```

123      # set PID values for panning
124      panP = manager.Value("f", 0.09)
125      panI = manager.Value("f", 0.08)
126      panD = manager.Value("f", 0.002)
127
128      # set PID values for tilting
129      tiltP = manager.Value("f", 0.11)
130      tiltI = manager.Value("f", 0.10)
131      tiltD = manager.Value("f", 0.002)

```

---

Our panning and tilting PID constants (process safe) are set on **Lines 124-131**. These are floats. Be sure to review the PID tuning section (Section 18.2.6) next to learn how we found suitable values. **To get the most value out of this project, I would recommend setting each to zero and following the tuning method/process** (not to be confused with a computer science method/process).

With all of our process safe variables ready to go, let's launch our processes:

---

```

133     # we have 4 independent processes
134     # 1. objectCenter - finds/localizes the object
135     # 2. panning      - PID control loop determines panning angle
136     # 3. tilting       - PID control loop determines tilting angle
137     # 4. setServos    - drives the servos to proper angles based
138     #                   on PID feedback to keep object in center
139     processObjectCenter = Process(target=obj_center,
140         args=(args, objX, objY, centerX, centerY))
141     processPanning = Process(target=pid_process,
142         args=(pan, panP, panI, panD, objX, centerX))
143     processTilting = Process(target=pid_process,
144         args=(tilt, tiltP, tiltI, tiltD, objY, centerY))
145     processSetServos = Process(target=set_servos, args=(pan, tilt))
146
147     # start all 4 processes
148     processObjectCenter.start()
149     processPanning.start()
150     processTilting.start()
151     processSetServos.start()
152
153     # join all 4 processes
154     processObjectCenter.join()
155     processPanning.join()
156     processTilting.join()
157     processSetServos.join()
158
159     # disable the servos
160     pth.servo_enable(1, False)
161     pth.servo_enable(2, False)

```

---

Each process is defined on **Lines 139-145**, passing required process safe values. We have four processes:

- i. A process which **finds the object in the frame**. In our case, it is a face.
- ii. A process which **calculates panning (left and right) angles with a PID**.
- iii. A process which **calculates tilting (up and down) angles with a PID**.
- iv. A process which **drives the servos**.

Each of the processes is started and then joined (**Lines 148-157**).

Servos are disabled when all processes exit (**Lines 160 and 161**). This also occurs in the `signal_handler` for when our program receives an interrupt signal.

### 18.2.6 Manual Tuning

That was a lot of work, but we're not done yet. Now that we understand the code, we need to perform manual tuning of our two independent PIDs (one for panning and one for tilting).

Tuning a PID ensures that our servos will track the object (in our case, a face) smoothly.

Be sure to refer to the “*How to tune a PID*” section from the previous chapter (Section 17.3.3). Additionally, the manual tuning section of Wikipedia’s PID article is a great resource [67].

You should follow this process:

- i. Set `kI` and `kD` to zero.
- ii. Increase `kP` from zero until the output oscillates (i.e. the servo goes back and forth or up and down). Then set the value to half.
- iii. Increase `kI` until offsets are corrected quickly, knowing that a value that is too high will cause instability.
- iv. Increase `kD` until the output settles on the desired output reference quickly after a load disturbance (i.e. if you move your face somewhere really fast). Too much `kD` will cause excessive response and make your output overshoot where it needs to be.

**I cannot stress this enough: *Make small changes while tuning.***

We will be tuning our PIDs independently, first by **tuning the tilting process**.

Go ahead and **comment out the panning process** in the driver script:

---

```

147      # start all 4 processes
148      processObjectCenter.start()
149      #processPanning.start()
150      processTilting.start()
151      processSetServos.start()
152
153      # join all 4 processes
154      processObjectCenter.join()
155      #processPanning.join()
156      processTilting.join()
157      processSetServos.join()
```

---

From there, open up a terminal and execute the following command:

---

```
$ python pan_tilt_tracking.py --cascade haarcascade_frontalface_default.xml
```

---

You will need to follow the manual tuning guide above to tune the tilting process. While doing so, you'll need to:

- i. Start the program and move your face up and down, causing the camera to tilt. I recommend doing squats at your knees while looking directly at the camera.
- ii. Stop the program and adjust values, per the tuning guide.
- iii. Repeat **until you're satisfied with the result** (and thus, the values). It should be tilting well with small displacements, as well as large changes, in where your face is. Be sure to test both.

At this point, let's **switch to the other PID**. The values will be similar, but it is necessary to tune them as well. Go ahead and **comment out the tilting process** (which is fully tuned) and **uncomment the panning process**:

---

```

147      # start all 4 processes
148      processObjectCenter.start()
149      processPanning.start()
150      #processTilting.start()
151      processSetServos.start()

152
153      # join all 4 processes
154      processObjectCenter.join()
155      processPanning.join()
156      #processTilting.join()
157      processSetServos.join()
```

---

And once again, execute the following command:

---

```
$ python pan_tilt_tracking.py \
--cascade haarcascade_frontalface_default.xml
```

---

Now follow the steps above again to **tune the panning process**, only this time moving from side to side rather than up and down.

### 18.2.7 Run Panning and Tilting Processes at the Same Time

With our freshly tuned PID constants, let's put our pan and tilt camera to the test.

**Assuming you followed the section above, ensure that both processes (panning and tilting) are uncommented and ready to go.**

From there, open up a terminal and execute the following command:

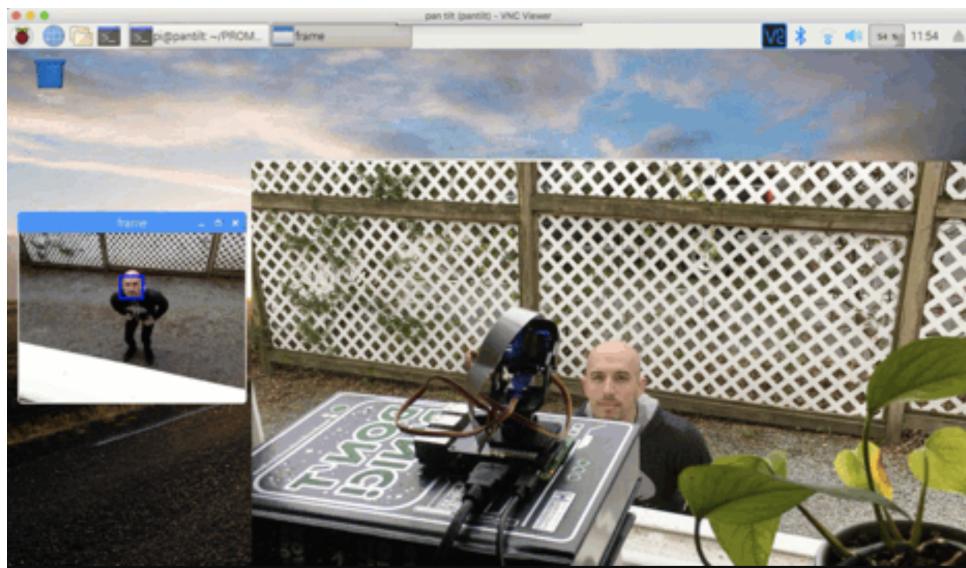


Figure 18.4: Pan/tilt face tracking demonstration. Full GIF demo available here: <http://pyimg.co/pb9f2>

---

```
$ python pan_tilt_tracking.py \
--cascade haarcascade_frontalface_default.xml
```

---

Once the script is up and running you can walk in front of your camera. If all goes well, you should see your face being *detected* and *tracked* as shown in Figure 18.4. Click this link to see a demo in your browser: <http://pyimg.co/pb9f2>.

### 18.3 Improvements for Pan/Tilt Tracking with the Raspberry Pi

There are times when the camera will encounter a false positive face, causing the control loop to go haywire. Don't be fooled! Your PID is working just fine, but your computer vision environment is impacting the system with false information.

We chose Haar because it is fast; however, Haar *can* lead to false positives:

- Haar isn't as accurate as HOG. HOG is great, but is resource hungry compared to Haar.
- Haar is far from accurate compared to a Deep Learning face detection method. The DL method is too slow to run on the Pi and real-time. If you tried to use the DL face detector, panning and tilting would be pretty jerky.

My recommendation is that you set up your pan/tilt camera in a new environment and see if that improves the results. For example, when we were testing the face tracking, we found that it didn't work well in a kitchen due to reflections off the floor, refrigerator, etc.

Then when we aimed the camera out the window and I stood outside, and the tracking improved drastically, because `ObjCenter` was providing legitimate values for the face. Thus our PID could do its job.

What if there are two faces in the frame? Or what if I'm the only face in the frame, but consistently there is a false positive?

This is a great question. In general, you'd want to track only one face, so there are a number of options:

- Use the **confidence value and take the face with the highest confidence**. This is **not possible using the default Haar detector code** as it doesn't report confidence values. Instead, let's explore other options.
- Try to get the `rejectLevels` and `rejectWeights` from the Haar cascade detections. I've never tried this, but the following links may help:
  - OpenCV Answers thread: <http://pyimg.co/iej12>
  - StackOverflow thread: <http://pyimg.co/x2f6e>
- **Grab the largest bounding box** — easy and simple.
- **Select the face closest to the center of the frame**. Since the camera tries to keep the face closest to the center, we could compute the Euclidean distance between all centroid bounding boxes and the center  $(x, y)$ -coordinates of the frame. The bounding box closest to the centroid would be selected.

## 18.4 Summary

In this chapter, we learned how to build a pan/tilt tracking system with our Raspberry Pi.

The system is capable of tracking any object provided that the Pi has enough computational horsepower to find the object. We took advantage of the fast Haar Cascade algorithm to find and track our face. We then deployed two PIDs using our `PID` class and multiprocessing. Finally we learned out to tune PID constants — a critical step. The result is a relatively smooth tracking algorithm!

Finally, be sure to refer to the PyImageSearch blog post from April 2019 on the topic as well (<http://pyimg.co/aiqr5>) [71]. The "*Comments*" section in particular provides a good discussion on pan/tilt tips, tricks, and suggestions from other PyImageSearch readers.

## Chapter 19

# Creating a People/Footfall Counter

A highly requested topic by readers of the PyImageSearch blog has been to create a people counter application. In August 2018, I wrote such a tutorial, entitled *OpenCV People Counter* (<http://pyimg.co/vgak6>) [72].

The tutorial was based upon a (1) a pre-trained MobileNet SSD object detector, (2) dlib's correlation tracker (<http://pyimg.co/yqlsy>) [73], and (3) my implementation of centroid tracking (<http://pyimg.co/nssn6>) [74].

Soon after the post published, readers asked a followup question:

*"I want to track people entering/exiting my store, but I want to keep costs down and mount an RPi near the doorway. Can the OpenCV People Counter run on a Raspberry Pi?"*

More courageous readers even tried out the code on a Raspberry Pi 3B+ with no modifications. They were disheartened by the fact that the RPi 3B+ achieved a minuscule 4.89 FPS making the methodology unusable — the people moved in “slow motion” using a video file as input and had we deployed this in the field, people would not have been counted properly.

That's not to say it is impossible to count people using an RPi though!

Working on resource-constrained hardware requires that you bring the right weapons to the fight. In the case of using the Raspberry Pi CPU for people counting, your nunchucks are background subtraction and your dagger is Haar Cascade. In this chapter, we will learn to make a handful of modifications to the original people tracking system I presented.

### 19.1 Chapter Learning Objectives

In this chapter we will learn:

- i. Saving CPU cycles with the efficient background subtraction algorithm
- ii. Object detection tradeoffs and how Haar Cascades are a great alternative
- iii. Object tracking tradeoffs for the Raspberry Pi
- iv. How to put these concepts together to build a reliable people counting application that will run on your Raspberry Pi

## 19.2 Background Subtraction + Haar Based People Counter

In this section, we'll briefly review our project structure. From there, we'll develop our `TrackableObject`, `CentroidTracker`, and `DirectionCounter` classes. We'll then implement the classes into our `people_counter.py` driver script which is optimized and tweaked for the Raspberry Pi. Finally, we'll learn how to execute the program; we'll report statistics for both the RPi 3B+ and RPi 4.

### 19.2.1 Project Structure

---

```

|-- videos
|   |-- horizontal_01.avi
|   |-- vertical_01.mp4
|   |-- vertical_02.mp4
|-- output
|   |-- output_01.avi
|-- pyimagesearch
|   |-- __init__.py
|   |-- centroidtracker.py
|   |-- directioncounter.py
|   |-- trackableobject.py
|-- people_counter.py

```

---

Input videos for testing are included in the `videos/` directory. The input videos are provided by David McDuffee.

Our `output/` folder will be where we'll store processed videos. One example output video is included.

The `pyimagesearch` module contains three classes which we will review: (1) `TrackableObject`, (2) `CentroidTracker`, and (3) `DirectionCounter`. Each trackable object is assigned an ID number. The centroid tracker associates objects and updates the trackable objects. Each trackable object has a list of centroids — its current location centroid, and all previous locations in the frame. The direction counter analyzes the current and historical lo-

cations to determine which direction an object is moving in. It also counts an object if it has passed certain horizontal or vertical line in the frame.

We'll also walk through the driver script in detail, `people_counter.py`. This script takes advantage of all the aforementioned classes in order to count people on a resource-constrained device.

### 19.2.2 Centroid Tracking

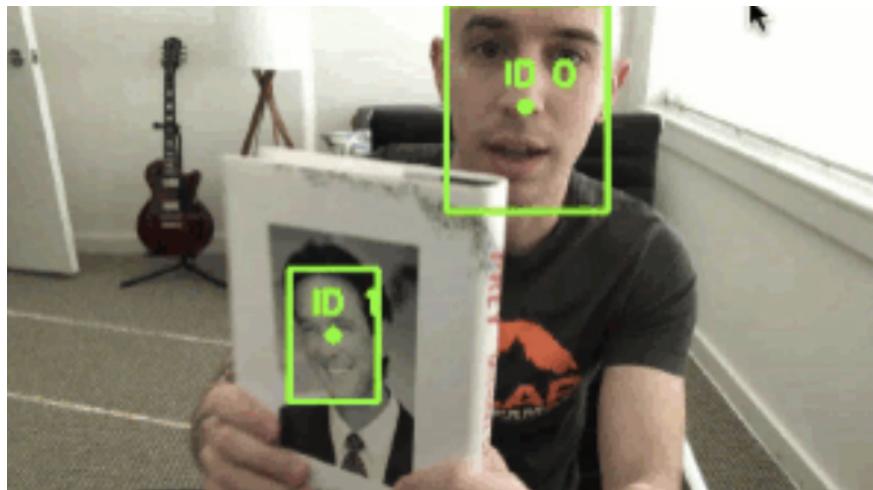


Figure 19.1: An example of centroid tracking in action. Notice how each detected face has a unique integer IDs associated with it. A full GIF of the demo can be found here: <http://pyimg.co/jhuw>

I first presented a simple object tracker in a blog post in July 2018 (<http://pyimg.co/nssn6>) [74]. Object tracking, is arguably one of the most requested topics here on PyImageSearch.

#### **Object tracking is the process of:**

- i. Taking an initial set of object detections (such as an input set of bounding box coordinates)
- ii. Creating a unique ID for each of the initial detections
- iii. And then tracking each of the objects as they move around frames in a video, maintaining the assignment of unique IDs

Furthermore, object tracking allows us to **apply a unique ID to each tracked object**, making it possible for us to count unique objects in a video. Object tracking is paramount to building a **person counter**.

An ideal object tracking algorithm will:

- Only require the object detection phase once (i.e., when the object is initially detected)

- Be extremely fast — *much* faster than running the actual object detector itself
- Be able to handle when the tracked object “disappears” or moves outside the boundaries of the video frame
- Be robust to occlusion
- Be able to pick up objects it has “lost” in between frames

This is a tall order for any computer vision or image processing algorithm, and there are a variety of tricks we can play to help improve our object trackers on resource-constrained devices. But before we can build such a robust method, we first need to study the fundamentals of object tracking.

### 19.2.2.1 Fundamentals of Object Tracking

A simple object tracking algorithm relies on keeping track of the centroids of objects.

Typically an *object tracker* works hand-in-hand with a less-efficient *object detector*. The *object detector* is responsible for localizing an object. The *object tracker* is responsible for keeping track of which object is which by assigning and maintaining identification numbers (IDs).

This object tracking algorithm we’re implementing is called *centroid tracking* as it relies on the Euclidean distance between (1) *existing* object centroids (i.e., objects the centroid tracker has already seen before), and (2) new object centroids between subsequent frames in a video.

We’ll review the centroid algorithm in more depth going forward.

### 19.2.2.2 The Centroid Tracking Algorithm

The centroid tracking algorithm is a multi-step process. The five steps include:

- i. **Step #1:** Accept bounding box coordinates and compute centroids
- ii. **Step #2:** Compute Euclidean distance between new bounding boxes and existing objects
- iii. **Step #3:** Update  $(x, y)$ -coordinates of existing objects
- iv. **Step #4:** Register new objects
- v. **Step #5:** Deregister old/lost objects that have moved out of frame

Let’s review each of these steps in more detail.

### 19.2.2.2.1 Step #1: Accept Bounding Box Coordinates and Compute Centroids

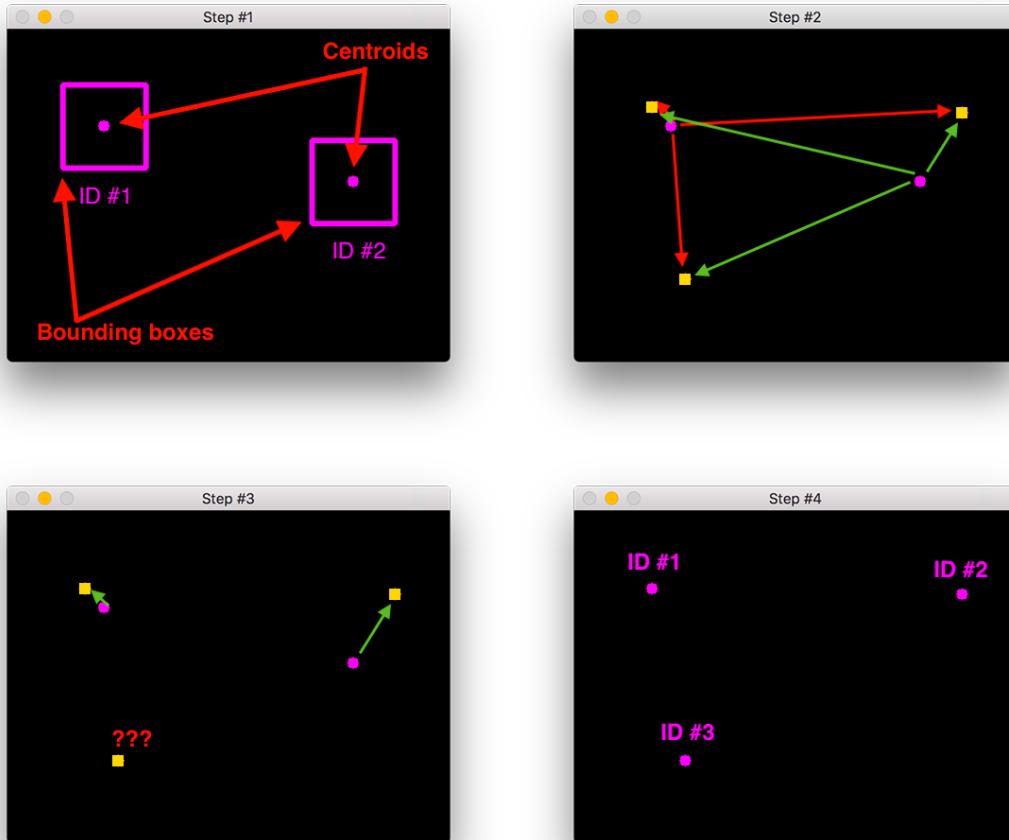


Figure 19.2: **Top-left:** To build a simple object tracking algorithm using centroid tracking, the first step is to accept bounding box coordinates from an object detector and use them to compute centroids. **Top-right:** In the next input frame, three objects are now present. We need to compute the Euclidean distances between each pair of original centroids (circle) and new centroids (square). **Bottom-left:** Our simple centroid object tracking method has associated objects with minimized object distances. What do we do about the object in the bottom left though? **Bottom-right:** We have a new object that wasn't matched with an existing object, so it is registered as object ID #3.

The centroid tracking algorithm assumes that we are passing in a set of bounding box  $(x, y)$ -coordinates for each detected object in ***every single frame***. These bounding boxes can be produced by any type of object detector you would like (color thresholding + contour extraction, Haar cascades, HOG + Linear SVM, SSDs, Faster R-CNNs, etc.), provided that they are computed for every frame in the video.

Once we have the bounding box coordinates we must compute the “centroid”, or more simply, the center  $(x, y)$ -coordinates of the bounding box. Figure 19.2 (*top-left*) demonstrates accepting a set of bounding box coordinates and computing the centroid.

Since these are the first initial set of bounding boxes presented to our algorithm we will assign them unique IDs.

#### 19.2.2.2 Step #2: Compute Euclidean Distance Between New Bounding Boxes and Existing Objects

For every subsequent frame in our video stream, we apply **Step #1** of computing object centroids; however, instead of assigning a new unique ID to each detected object (which would defeat the purpose of object tracking), we first need to determine if we can *associate* the *new* object centroids (circles) with the *old* object centroids (squares). To accomplish this process, we compute the Euclidean distance (highlighted with green arrows) between each pair of existing object centroids and input object centroids.

From Figure 19.2 (*top-right*) you can see that we have this time detected three objects in our image. The two pairs that are close together are two existing objects. We then compute the Euclidean distances between each pair of original centroids (yellow) and new centroids (purple).

But how do we use the Euclidean distances between these points to actually match them and associate them?

The answer is in **Step #3**.

#### 19.2.2.3 Step #3: Update $(x, y)$ -coordinates of Existing Objects

The primary assumption of the centroid tracking algorithm is that a given object will potentially move in between subsequent frames, but the *distance* between the centroids for frames  $F_t$  and  $F_{t+1}$  will be *smaller* than all other distances between objects.

Therefore, if we choose to associate centroids with minimum distances between subsequent frames we can build our object tracker.

In Figure 19.2 (*bottom-right*) you can see how our centroid tracker algorithm chooses to associate centroids that minimize their respective Euclidean distances.

But what about the lonely point in the bottom-left? It didn't get associated with anything — what do we do with it?

#### 19.2.2.2.4 Step #4: Register New Objects

In the event that there are more input detections than existing objects being tracked, we need to register the new object. “Registering” simply means that we are adding the new object to our list of tracked objects by (1) assigning it a new object ID, and (2) storing the centroid of the bounding box coordinates for that object

We can then go back to **Step #2** and repeat the pipeline of steps for every frame in our video stream.

Figure 19.2 (*bottom-right*) demonstrates the process of using the minimum Euclidean distances to associate existing object IDs and then registering a new object.

#### 19.2.2.5 Step #5: Deregister Old Objects

Any reasonable object tracking algorithm needs to be able to handle when an object has been lost, disappeared, or left the field of view. Exactly how you handle these situations is really dependent on where your object tracker is meant to be deployed, but for this implementation, we will deregister old objects when they cannot be matched to any existing objects for a total of  $N$  subsequent frames.

#### 19.2.2.3 Centroid Tracking Implementation

Before we can apply object tracking to our input video streams, we first need to implement the centroid tracking algorithm. While you’re digesting this centroid tracker script, just keep in mind **Steps #1 through Step #5 detailed above** and review the steps as necessary. As you’ll see, the translation of steps to code requires quite a bit of thought, and while we perform all steps, they aren’t linear due to the nature of our various data structures and code constructs.

I would suggest re-reading the steps above, re-reading the code explanation for the centroid tracker, and finally reviewing it all once more. This will bring everything full circle, and allow you to wrap your head around the algorithm.

Once you’re sure you understand the steps in the centroid tracking algorithm, open up the `centroidtracker.py` inside the `pyimagesearch` module, and let’s review the code:

---

```
1 # import the necessary packages
2 from scipy.spatial import distance as dist
3 from collections import OrderedDict
4 import numpy as np
5
6 class CentroidTracker:
```

---

```

7  def __init__(self, maxDisappeared=50, maxDistance=50):
8      # initialize the next unique object ID along with two ordered
9      # dictionaries used to keep track of mapping a given object
10     # ID to its centroid and number of consecutive frames it has
11     # been marked as "disappeared", respectively
12     self.nextObjectID = 0
13     self.objects = OrderedDict()
14     self.disappeared = OrderedDict()
15
16     # store the number of maximum consecutive frames a given
17     # object is allowed to be marked as "disappeared" until we
18     # need to deregister the object from tracking
19     self.maxDisappeared = maxDisappeared
20
21     # store the maximum distance between centroids to associate
22     # an object -- if the distance is larger than this maximum
23     # distance we'll start to mark the object as "disappeared"
24     self.maxDistance = maxDistance

```

---

On **Lines 2-4** we import our required packages and modules — `distance`, `OrderedDict`, and `numpy`.

Our `CentroidTracker` class is defined on **Line 6**. The constructor accepts a single parameter, the maximum number of consecutive frames a given object has to be lost/disappeared for until we remove it from our tracker (**Line 7**). Our constructor builds five class variables:

- `nextObjectID`: A counter used to assign unique IDs to each object (**Line 12**). In the case that an object leaves the frame and does not come back for `maxDisappeared` frames, a new (next) object ID would be assigned.
- `objects`: A dictionary that utilizes the object ID as the key and the centroid ( $x$ ,  $y$ )-coordinates as the value (**Line 13**).
- `disappeared`: Maintains number of consecutive frames (value) a particular object ID (key) has been marked as “lost” (**Line 14**).
- `maxDisappeared`: The number of consecutive frames an object is allowed to be marked as “lost/disappeared” until we deregister the object.
- `maxDistance`: The maximum distance between centroids to associate an object. If the distance exceeds this value, then the object will be marked as disappeared.

Let’s define the `register` method which is responsible for adding new objects to our tracker:

---

```

26 def register(self, centroid):
27     # when registering an object we use the next available object

```

---

```

28         # ID to store the centroid
29         self.objects[self.nextObjectID] = centroid
30         self.disappeared[self.nextObjectID] = 0
31         self.nextObjectID += 1
32
33     def deregister(self, objectID):
34         # to deregister an object ID we delete the object ID from
35         # both of our respective dictionaries
36         del self.objects[objectID]
37         del self.disappeared[objectID]
```

---

The `register` method is defined on **Line 26**. It accepts a centroid and then adds it to the `objects` dictionary using the next available object ID.

The number of times an object has disappeared is initialized to 0 in the `disappeared` dictionary (**Line 30**).

Finally, we increment the `nextObjectID` so that if a new object comes into view, it will be associated with a unique ID (**Line 31**).

Similar to our registration method, we also need a `deregister` method. Just like we can add new objects to our tracker, we also need the ability to remove old ones that have been lost or disappeared from our input frames themselves. The `deregister` method is defined on **Line 33**. It simply deletes the `objectID` in both the `objects` and `disappeared` dictionaries, respectively (**Lines 36 and 37**).

The heart of our centroid tracker implementation lives inside the `update` method:

---

```

39     def update(self, rects):
40         # check to see if the list of input bounding box rectangles
41         # is empty
42         if len(rects) == 0:
43             # loop over any existing tracked objects and mark them
44             # as disappeared
45             for objectID in list(self.disappeared.keys()):
46                 self.disappeared[objectID] += 1
47
48             # if we have reached a maximum number of consecutive
49             # frames where a given object has been marked as
50             # missing, deregister it
51             if self.disappeared[objectID] > self.maxDisappeared:
52                 self.deregister(objectID)
53
54             # return early as there are no centroids or tracking info
55             # to update
56             return self.objects
57
58         # initialize an array of input centroids for the current frame
59         inputCentroids = np.zeros((len(rects), 2), dtype="int")
```

---

---

```

60
61     # loop over the bounding box rectangles
62     for (i, (startX, startY, endX, endY)) in enumerate(rects):
63         # use the bounding box coordinates to derive the centroid
64         cX = int((startX + endX) / 2.0)
65         cY = int((startY + endY) / 2.0)
66         inputCentroids[i] = (cX, cY)

```

---

The `update` method, defined on **Line 39**, accepts a list of bounding box rectangles, presumably from an object detector (Haar cascade, HOG + Linear SVM, SSD, Faster R-CNN, etc.). The format of the `rects` parameter is assumed to be a tuple with this structure: `(startX, startY, endX, endY)`.

If there are no detections, we'll loop over all object IDs and increment their `disappeared` count (**Lines 42-46**). We'll also check if we have reached the maximum number of consecutive frames a given object has been marked as missing. If that is the case we need to remove it from our tracking systems (**Lines 51 and 52**). Since there is no tracking info to update, we go ahead and `return` early on **Line 51**. Otherwise, we have quite a bit of work to do to complete our `update` method implementation.

On **Line 59** we'll initialize a NumPy array to store the centroids for each `rect`.

Then, we loop over bounding box rectangles (**Line 62**) and compute the centroid and store it in the `inputCentroids` list (**Lines 64-66**).

If there are currently no objects we are tracking, we'll `register` each of the new objects:

---

```

68     # if we are currently not tracking any objects take the input
69     # centroids and register each of them
70     if len(self.objects) == 0:
71         for i in range(0, len(inputCentroids)):
72             self.register(inputCentroids[i])

```

---

Otherwise, we need to update any existing object  $(x, y)$ -coordinates based on the centroid location that minimizes the Euclidean distance between them:

---

```

74     # otherwise, are are currently tracking objects so we need to
75     # try to match the input centroids to existing object
76     # centroids
77     else:
78         # grab the set of object IDs and corresponding centroids
79         objectIDs = list(self.objects.keys())
80         objectCentroids = list(self.objects.values())
81
82         # compute the distance between each pair of object
83         # centroids and input centroids, respectively -- our

```

---

---

```

84         # goal will be to match an input centroid to an existing
85         # object centroid
86         D = dist.cdist(np.array(objectCentroids), inputCentroids)
87
88         # in order to perform this matching we must (1) find the
89         # smallest value in each row and then (2) sort the row
90         # indexes based on their minimum values so that the row
91         # with the smallest value as at the *front* of the index
92         # list
93         rows = D.min(axis=1).argsort()
94
95         # next, we perform a similar process on the columns by
96         # finding the smallest value in each column and then
97         # sorting using the previously computed row index list
98         cols = D.argmin(axis=1)[rows]

```

---

The updates to existing tracked objects take place beginning at the `else` on [Line 77](#). The goal is to track the objects and to maintain correct object IDs — this process is accomplished by computing the Euclidean distances between all pairs of `objectCentroids` and `inputCentroids`, followed by associating object IDs that minimize the Euclidean distance.

Inside of the `else` block, beginning on [Line 77](#), we will first grab `objectIDs` and `objectCentroid` values ([Lines 79 and 80](#)). We then compute the distance between each pair of existing object centroids and new input centroids ([Line 86](#)). The output NumPy array shape of our distance map `D` will be (# of object centroids, # of input centroids).

To perform the matching we must (1) find the smallest value in each row, and (2) sort the row indexes based on the minimum values ([Line 93](#)). We perform a very similar process on the columns, finding the smallest value in each, and then sorting them based on the ordered rows ([Line 98](#)). Our goal is to have the index values with the smallest corresponding distance at the front of the lists.

The next step is to use the distances to see if we can associate object IDs:

---

```

100         # in order to determine if we need to update, register,
101         # or deregister an object we need to keep track of which
102         # of the rows and column indexes we have already examined
103         usedRows = set()
104         usedCols = set()
105
106         # loop over the combination of the (row, column) index
107         # tuples
108         for (row, col) in zip(rows, cols):
109             # if we have already examined either the row or
110             # column value before, ignore it
111             if row in usedRows or col in usedCols:
112                 continue
113

```

---

```

114     # if the distance between centroids is greater than
115     # the maximum distance, do not associate the two
116     # centroids to the same object
117     if D[row, col] > self.maxDistance:
118         continue
119
120     # otherwise, grab the object ID for the current row,
121     # set its new centroid, and reset the disappeared
122     # counter
123     objectID = objectIDs[row]
124     self.objects[objectID] = inputCentroids[col]
125     self.disappeared[objectID] = 0
126
127     # indicate that we have examined each of the row and
128     # column indexes, respectively
129     usedRows.add(row)
130     usedCols.add(col)

```

---

Inside the code block, we start by initializing two sets to determine which row and column indexes we have already used (**Lines 103 and 104**). Keep in mind that a set is similar to a list but it contains only *unique* values.

Then we loop over the combinations of `(row, col)` index tuples (**Line 108**) in order to update our object centroids. If we've already used either this row or column index, ignore it and continue to loop (**Lines 111 and 112**). Similarly, if the distance between centroids exceeds the `maxDistance`, then we will not associate the two centroids (**Lines 117 and 118**).

**Remark.** *This is new functionality compared to my original blog post implementation because we are tracking objects via a background subtraction and centroids rather than an object detector.*

Otherwise, we have found an input centroid that (1) has the smallest Euclidean distance to an existing centroid, and (2) has not been matched with any other object. In that case, we update the object centroid and make sure to add the row and col to their respective `usedRows` and `usedCols` sets (**Lines 123-130**)

There are also likely indexes in our `usedRows` and `usedCols` sets that we have *NOT* examined yet:

```

132             # compute both the row and column index we have NOT yet
133             # examined
134             unusedRows = set(range(0, D.shape[0])).difference(usedRows)
135             unusedCols = set(range(0, D.shape[1])).difference(usedCols)

```

---

We must determine which centroid indexes we haven't examined yet and store them in two new convenient sets (`unusedRows` and `unusedCols`) on **Lines 134 and 135**.

Our final check handles any objects that have become lost or if they've potentially disappeared:

---

```

137      # in the event that the number of object centroids is
138      # equal or greater than the number of input centroids
139      # we need to check and see if some of these objects have
140      # potentially disappeared
141      if D.shape[0] >= D.shape[1]:
142          # loop over the unused row indexes
143          for row in unusedRows:
144              # grab the object ID for the corresponding row
145              # index and increment the disappeared counter
146              objectID = objectIDs[row]
147              self.disappeared[objectID] += 1
148
149              # check to see if the number of consecutive
150              # frames the object has been marked "disappeared"
151              # for warrants deregistering the object
152              if self.disappeared[objectID] > self.maxDisappeared:
153                  self.deregister(objectID)

```

---

To finish up, if the number of object centroids is greater than or equal to the number of input centroids (**Line 141**), we need to verify if any of these objects are lost or have disappeared by looping over unused row indexes, if any (**Line 143**). In the loop, we will increment their disappeared count in the dictionary (**Line 147**). Then we check if the disappeared count exceeds the maxDisappeared threshold (**Line 152**). If so, we'll deregister the object (**Line 153**).

Otherwise, the number of input centroids is greater than the number of existing object centroids, so we have new objects to register and track:

---

```

155      # otherwise, if the number of input centroids is greater
156      # than the number of existing object centroids we need to
157      # register each new input centroid as a trackable object
158      else:
159          for col in unusedCols:
160              self.register(inputCentroids[col])
161
162      # return the set of trackable objects
163      return self.objects

```

---

We loop over the unusedCols indexes (**Line 159**) and we register each new centroid (**Line 160**).

Finally, we'll return the set of trackable objects to the calling method (**Line 163**).

### 19.2.3 Trackable Objects

In order to track and count an object in a video stream, we need an easy way to store information regarding the object itself, including:

- It's object ID
- It's previous centroids (so we can easily compute the direction the object is moving)
- Whether or not the object has already been counted

To accomplish all of these goals we can define an instance of `TrackableObject` — open up the `trackableobject.py` file and insert the following code:

---

```

1  class TrackableObject:
2      def __init__(self, objectID, centroid):
3          # store the object ID, then initialize a list of centroids
4          # using the current centroid
5          self.objectID = objectID
6          self.centroids = [centroid]
7
8          # initialize a boolean used to indicate if the object has
9          # already been counted or not
10         self.counted = False

```

---

We will have multiple trackable objects — one for each person that is being tracked in the frame. Each object will have the three attributes shown on **Lines 5-10**.

The `TrackableObject` constructor accepts an `objectID` + `centroid` and stores them. The `centroids` variable is a list because it will contain an object's centroid location history. The constructor also initializes `counted` as `False`, indicating that the object has not been counted yet.

### 19.2.4 The Centroid Tracking Distance Relationship

Recall that each `TrackableObject` has a `centroids` attribute — a list of the object's location history. The algorithm described in the `update` method relies on the history to associate centroids and object IDs by calculating the distance between them.

Calculating the distance between points on a cartesian coordinate system can be done in several ways. The most common method is “as the crow flies”, or more formally known as the Euclidean distance.

If you're having trouble following along with **Lines 86-99** open a Python shell and let's practice:

---

```

1  >>> from scipy.spatial import distance as dist
2  >>> import numpy as np
3  >>> np.random.seed(42)
4  >>> objectCentroids = np.random.uniform(size=(2, 2))
5  >>> centroids = np.random.uniform(size=(3, 2))
6  >>> D = dist.cdist(objectCentroids, centroids)
7  >>> D
8  array([[0.82421549, 0.32755369, 0.33198071],
9         [0.72642889, 0.72506609, 0.17058938]])

```

---

Once you've started a Python shell in your terminal with the `python` command, import `distance` and `numpy` as shown on **Lines 1 and 2**.

Then, set a seed for reproducibility (**Line 3**) and generate two (random) existing `objectCentroids` (**Line 4**) and three `inputCentroids` (**Line 5**).

From there, compute the Euclidean distance between the pairs (**Line 6**) and display the results (**Lines 7-9**). The result is a matrix `D` of distances with two rows (# of existing object centroids) and three columns (# of new input centroids).

Just like we did earlier in the script, let's find the minimum distance in each row and sort the indexes based on this value:

---

```

10 >>> D.min(axis=1)
11 array([0.32755369, 0.17058938])
12 >>> rows = D.min(axis=1).argsort()
13 >>> rows
14 array([1, 0])

```

---

First, we find the minimum value for each row, allowing us to figure out which existing object is closest to the new input centroid (**Lines 10 and 11**). By then sorting on these values (**Line 12**) we can obtain the indexes of these rows (**Lines 13 and 14**).

In this case, the second row (index 1) has the smallest value and then the first row (index 0) has the next smallest value.

Let's perform a similar process for the columns:

---

```

15 >>> D.argmin(axis=1)
16 array([1, 2])
17 >>> cols = D.argmin(axis=1)[rows]
18 >>> cols
19 array([2, 1])

```

---

We first examine the values in the columns and find the index of the value with the smallest

column (**Lines 15 and 16**). We then sort these values using our existing rows (**Lines 17-19**). Let's print the results and analyze them:

---

```
20 >>> print(list(zip(rows, cols)))
21 [(1, 2), (0, 1)]
```

---

The final step is to combine them using `zip` (**Lines 20**). The resulting list is printed on **Line 21**.

Analyzing the results, we can make two observations. First, `D[1, 2]` has the smallest Euclidean distance implying that the second existing object will be matched against the third input centroid. And second, `D[0, 1]` has the next smallest Euclidean distance which implies that the first existing object will be matched against the second input centroid.

I'd like to reiterate here that now that you've reviewed the code, you should go back and review the steps to the algorithm in the previous section. From there you'll be able to associate the code with the more linear steps outlined here.

### 19.2.5 The DirectionCounter Class

Another difference (and highly requested feature) between my August 2018 blog post implementation and this book's implementation of people counting is that we can count people from (a) up/down, or (b) left/right. Previously, the app only counted upwards/downwards movement through the frame.

With this new functionality comes a new class, `DirectionCounter`, to manage the task of counting in either vertically or horizontally.

Let's go ahead and build our constructor:

---

```
1 # import the necessary packages
2 import numpy as np
3
4 class DirectionCounter:
5     def __init__(self, directionMode, H, W):
6         # initialize the height and width of the input image
7         self.H = H
8         self.W = W
9
10        # initialize variables holding the direction of movement,
11        # along with counters for each respective movement (i.e.,
12        # left-to-right and top-to-bottom)
13        self.directionMode = directionMode
14        self.totalUp = 0
15        self.totalDown = 0
```

---

```

16         self.totalRight = 0
17         self.totalLeft = 0
18
19         # the direction the trackable object is moving in
20         self.direction = ""

```

---

Our constructor accepts three parameters. We must pass the `directionMode`, either "vertical" or "horizontal", that we will be counting our objects.

We also must pass the the height and width of the input image, `H` and `W`.

Instance variables are then initialized. Draw your attention to **Lines 14-17** where the totals are initialized to zero. If our `directionMode` is "vertical", we only care about `totalUp` and `totalDown`. Similarly, if our `directionMode` is "horizontal" we are only concerned with `totalRight` and `totalLeft`.

Next, draw your attention to a trackable object's direction `direction` on **Line 21**. We will calculate an object's direction in the the following `find_direction` function:

---

```

22     def find_direction(self, to, centroid):
23         # check to see if we are tracking horizontal movements
24         if self.directionMode == "horizontal":
25             # the difference between the x-coordinate of the
26             # *current* centroid and the mean of *previous* centroids
27             # will tell us in which direction the object is moving
28             # (negative for 'left' and positive for 'right')
29             x = [c[0] for c in to.centroids]
30             delta = centroid[0] - np.mean(x)
31
32             # determine the sign of the delta -- if it is negative,
33             # the object is moving left
34             if delta < 0:
35                 self.direction = "left"
36
37             # otherwise if the sign is positive, the object is moving
38             # right
39             elif delta > 0:
40                 self.direction = "right"
41
42             # otherwise we are tracking vertical movements
43             elif self.directionMode == "vertical":
44                 # the difference between the y-coordinate of the
45                 # *current* centroid and the mean of *previous* centroids
46                 # will tell us in which direction the object is moving
47                 # (negative for 'up' and positive for 'down')
48                 y = [c[1] for c in to.centroids]
49                 delta = centroid[1] - np.mean(y)
50
51             # determine the sign of the delta -- if it is negative,
52             # the object is moving up

```

---

---

```

53     if delta < 0:
54         self.direction = "up"
55
56     # otherwise, if the sign of the delta is positive, the
57     # object is moving down
58     elif delta > 0:
59         self.direction = "down"

```

---

The `find_direction` function accepts a trackable object (`to`) and a single centroid. Recall that `centroids` contains a historical listing of an object's position. **Line 30 or 49** grabs all `x`-values or `y`-values from the trackable object's historical centroids, respectively.

The `delta` is calculated by averaging all previous centroid `x` or `y`-coordinate values and subtracting it from the very first value.

From there, if the `delta` is negative, the object is either moving "left" or "down" (**Lines 34 and 35** plus **Lines 53 and 54**).

Or if the `delta` is positive, the object is either moving "right" (**Lines 39 and 40**) or "up" (**Lines 58 and 59**).

Now let's implement `count_object`, a function which actually performs the counting:

---

```

61     def count_object(self, to, centroid):
62         # initialize the output list
63         output = []
64
65         # check if the direction of the movement is horizontal
66         if self.directionMode == "horizontal":
67             # if the object is currently left of center and is
68             # moving left, count the object as moving left
69             leftOfCenter = centroid[0] < self.W // 2
70             if self.direction == "left" and leftOfCenter:
71                 self.totalLeft += 1
72                 to.counted = True
73
74             # otherwise, if the object is right of center and moving
75             # right, count the object as moving right
76             elif self.direction == "right" and not leftOfCenter:
77                 self.totalRight += 1
78                 to.counted = True
79
80             # construct a list of tuples with the count of objects
81             # that have passed in the left and right direction
82             output = [("Left", self.totalLeft),
83                     ("Right", self.totalRight)]

```

---

The `count_object` function accepts a trackable object and `centroid`. It will create/update a list (`output`) of 2-tuples indicating left/right counts or up/down counts.

**Lines 66-83** handle the "horizontal" direction mode. If the object is `leftOfCenter` and is moving left, then it is marked as counted and the `totalLeft` is incremented.

Similarly, if the object is not `leftOfCenter` (i.e. right of center) and it is moving "right", then `totalRight` is incremented and the object is marked as counted.

The next code block operates in the exact same fashion, but for the "vertical" direction:

---

```

85      # otherwise the direction of movement is vertical
86      elif self.directionMode == "vertical":
87          # if the centroid is above the middle and is moving
88          # up, count the object as moving up
89          aboveMiddle = centroid[1] < self.H // 2
90          if self.direction == "up" and aboveMiddle:
91              self.totalUp += 1
92              to.counted = True
93
94          # otherwise, if the object is moving down and is below
95          # the middle, count the object as moving down
96          elif self.direction == "down" and not aboveMiddle:
97              self.totalDown += 1
98              to.counted = True
99
100         # return a list of tuples with the count of objects that
101         # have passed in the up and down direction
102         output = [("Up", self.totalUp), ("Down", self.totalDown)]
103
104     # return the output list
105     return output

```

---

This time we're testing if the object is `aboveMiddle` or not and whether it is moving "up" or "down". The logic is identical.

In the next section, we will proceed to implement our people counter driver script.

### 19.2.6 Implementing Our People Counting App Based on Background Subtraction

We have all components/classes ready at this point. Now let's go ahead and implement our people counting driver script.

Go ahead and open a new file named `people_counter.py` and insert the following code:

---

```

1 # import the necessary packages
2 from pyimagesearch.directioncounter import DirectionCounter
3 from pyimagesearch.centroidtracker import CentroidTracker
4 from pyimagesearch.trackableobject import TrackableObject

```

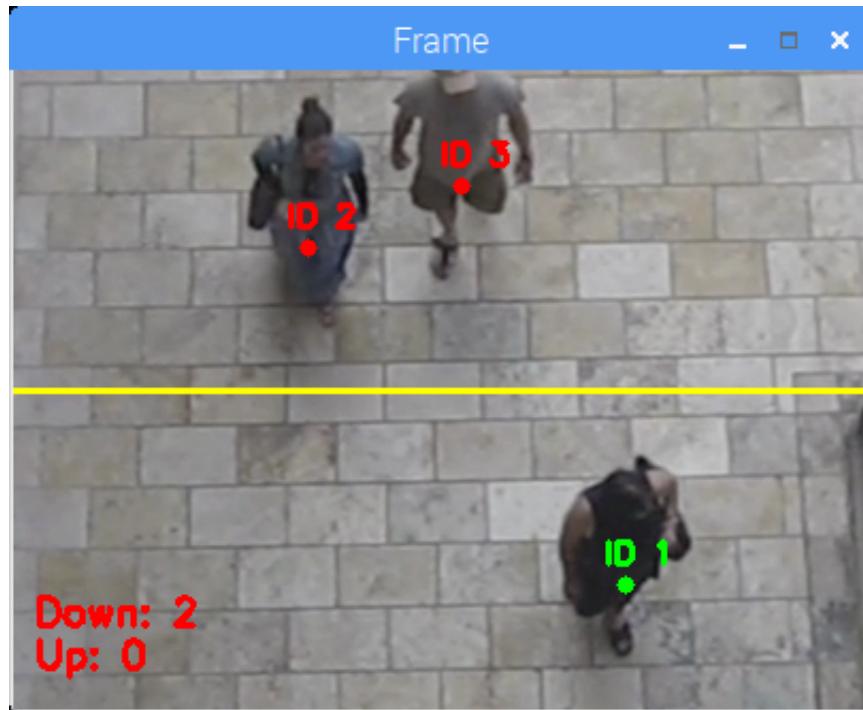


Figure 19.3: Our people counting GUI. The *yellow* counting line will be where objects have to pass to be counted as moving up or down (counts are shown in *red* in the *bottom-left*). Object entroids and object IDs are shown as either *red* (not counted yet) or *green* counted. In this frame, ID #0 (not pictured) has been counted as moving down, ID #1 has been counted as moving down (this person moved down and across the counting line), and IDs #2 and #3 have not yet been counted (they are moving down but have not crossed the counting line).

```

5 from multiprocessing import Process
6 from multiprocessing import Queue
7 from multiprocessing import Value
8 from imutils.video import VideoStream
9 from imutils.video import FPS
10 import argparse
11 import imutils
12 import time
13 import cv2

```

**Lines 2-13** import necessary packages. We will use our `DirectionCounter`, `CentroidTracker`, and `TrackableObject`. Taking advantage of `multiprocessing`, we need `Process`, `Queue`, and `Value` for writing to video in a separate process to achieve higher FPS.

Let's define that process for writing to video files now:

---

```

15 def write_video(outputPath, writeVideo, frameQueue, W, H):
16     # initialize the FourCC and video writer object
17     fourcc = cv2.VideoWriter_fourcc(*"MJPG")
18     writer = cv2.VideoWriter(outputPath, fourcc, 30,

```

```

19         (W, H), True)
20
21     # loop while the write flag is set or the output frame queue is
22     # not empty
23     while writeVideo.value or not frameQueue.empty():
24         # check if the output frame queue is not empty
25         if not frameQueue.empty():
26             # get the frame from the queue and write the frame
27             frame = frameQueue.get()
28             writer.write(frame)
29
30     # release the video writer object
31     writer.release()

```

---

The `write_video` function will run in an independent process, writing frames from the `frameQueue` to a video file. It accepts five parameters: (1) `outputPath`, the filepath to the output video file, (2) `writeVideo`, a flag indicating if video writing should be ongoing, (3) `frameQueue`, a process-safe `Queue` holding our frames to be written to disk in the video file, and finally, (4 and 5) the video file dimensions.

**Lines 18-20** initialize the video writer. From there, an infinite loop starts on **Line 24** and will continue to write to the video until `writeVideo` is `False`. The frames are written as they become available in the `frameQueue`.

When the video is finished, it is released (**Line 32**).

With our video writing process out of the way, now let's define our command line arguments:

```

33     # construct the argument parser and parse the arguments
34     ap = argparse.ArgumentParser()
35     ap.add_argument("-m", "--mode", type=str, required=True,
36                     choices=["horizontal", "vertical"],
37                     help="direction in which people will be moving")
38     ap.add_argument("-i", "--input", type=str,
39                     help="path to optional input video file")
40     ap.add_argument("-o", "--output", type=str,
41                     help="path to optional output video file")
42     ap.add_argument("-s", "--skip-frames", type=int, default=30,
43                     help="# of skip frames between detections")
44     args = vars(ap.parse_args())

```

---

Our driver script accepts four command line arguments:

- `--mode`: The direction (either `horizontal` or `vertical` in which people will be moving through the frame).
- `--input`: The path to an optional input video file.

- `--output`: The path to an optional output video file. When an output video filepath is provided, the `write_video` process will come to life.
- `--skip-frames`: The number of skip frames used to improve computational efficiency on the RPi. By default we'll skip 30 frames between detections. In between we'll simply correlate and track the detections that have already been made.

Our script can handle webcam or video file streams:

---

```

46 # if a video path was not supplied, grab a reference to the webcam
47 if not args.get("input", False):
48     print("[INFO] starting video stream...")
49     # vs = VideoStream(src=0).start()
50     vs = VideoStream(usePiCamera=True).start()
51     time.sleep(2.0)
52
53 # otherwise, grab a reference to the video file
54 else:
55     print("[INFO] opening video file...")
56     vs = cv2.VideoCapture(args["input"])
57
58 # initialize the video writing process object (we'll instantiate
59 # later if need be) along with the frame dimensions
60 writerProcess = None
61 W = None
62 H = None

```

---

**Lines 48-52** start a PiCamera video stream. If you'd like to use a USB camera, you can swap the comment symbol between **Lines 50 and 51**. Otherwise, a video file stream will be initialized and started (**Lines 55-57**).

We have a handful more initializations to take care of:

---

```

63 # instantiate our centroid tracker, then initialize a list to store
64 # each of our dlib correlation trackers, followed by a dictionary to
65 # map each unique object ID to a trackable object
66 ct = CentroidTracker(maxDisappeared=15, maxDistance=100)
67 trackers = []
68 trackableObjects = {}
69
70 # initialize the direction info variable (used to store information
71 # such as up/down or left/right people count)
72 directionInfo = None
73
74 # initialize the MOG foreground background subtractor and start the
75 # frames per second throughput estimator
76 mog = cv2.bgsegm.createBackgroundSubtractorMOG()
77 fps = FPS().start()

```

---

Our video `writerProcess` placeholder is initialized as `None` on **Line 61** along with the placeholders for the dimensions (**Lines 62 and 63**).

The `directionInfo` is a list of two tuples in the format `[("Up", totalUp), ("Down", totalDown)]` or `[("Left", totalLeft), ("Right", totalRight)]`. This variable contains the counts of people that have moved up and down or left and right. **Line 74** initializes the `directionInfo` as `None`.

Our last initializations include our MOG background subtractor (**Line 78**) and our FPS counter (**Line 79**).

Now let's get to work processing frames:

---

```

80 # loop over frames from the video stream
81 while True:
82     # grab the next frame and handle if we are reading from either
83     # VideoCapture or VideoStream
84     frame = vs.read()
85     frame = frame[1] if args.get("input", False) else frame
86
87     # if we are viewing a video and we did not grab a frame then we
88     # have reached the end of the video
89     if args["input"] is not None and frame is None:
90         break
91
92     # set the frame dimensions and instantiate direction counter
93     # object if required
94     if W is None or H is None:
95         (H, W) = frame.shape[:2]
96     dc = DirectionCounter(args["mode"], H, W)

```

---

Our `while` loop begins on **Line 82**. A frame is grabbed and indexed depending on if it is from a webcam or video stream (**Lines 86-91**).

If the dimensions of the frame haven't been initialized, it is our signal to initialize them in addition to our `DirectionCounter` (**Lines 95-97**). Notice that we pass the `"mode"` to the direction counter, which will be either `"vertical"` or `"horizontal"`, as well as the frame dimensions.

Next, we'll start our writer process (if we will be writing to a video file):

---

```

98     # begin writing the video to disk if required
99     if args["output"] is not None and writerProcess is None:
100         # set the value of the write flag (used to communicate when
101         # to stop the process)
102         writeVideo = Value('i', 1)
103

```

---

---

```

104     # initialize a shared queue for the exchange frames,
105     # initialize a process, and start the process
106     frameQueue = Queue()
107     writerProcess = Process(target=write_video, args=
108         args["output"], writeVideo, frameQueue, W, H))
109     writerProcess.start()

```

---

If we have an "output" video path in args and the writerProcess doesn't yet exist (**Line 100**), then **Lines 103-110** set the writeVideo flag, initialize our frameQueue, and start our writerProcess.

Moving on, let's preprocess our frame and **apply background subtraction**:

---

```

111     # initialize a list to store the bounding box rectangles returned
112     # by background subtraction model
113     rects = []
114
115     # convert the frame to grayscale and smoothen it using a
116     # gaussian kernel
117     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
118     gray = cv2.GaussianBlur(gray, (5, 5), 0)
119
120     # apply the MOG background subtraction model
121     mask = mog.apply(gray)
122
123     # apply a series of erosions to break apart connected
124     # components, then find contours in the mask
125     erode = cv2.erode(mask, (7, 7), iterations=2)
126     cnts = cv2.findContours(erode.copy(), cv2.RETR_EXTERNAL,
127         cv2.CHAIN_APPROX_SIMPLE)
128     cnts = imutils.grab_contours(cnts)

```

---

**Line 114** initializes an empty list to store bounding box rectangles which will be returned by the background subtraction model.

**Lines 118 and 119** convert the frame to grayscale and blur it slightly. Background subtraction is applied to the gray frame (**Line 122**). From there, a series of erosions are applied to reduce noise (**Line 126**).

Then contours are found and extracted via **Lines 127-129**. Let's process our contours and add the bounding boxes to rects:

---

```

130     # loop over each contour
131     for c in cnts:
132         # if the contour area is less than the minimum area
133         # required then ignore the object
134         if cv2.contourArea(c) < 2000:

```

---

---

```

135         continue
136
137     # compute the bounding box coordinates of the contour
138     (x, y, w, h) = cv2.boundingRect(c)
139     (startX, startY, endX, endY) = (x, y, x + w, y + h)
140
141     # add the bounding box coordinates to the rectangles list
142     rects.append((startX, startY, endX, endY))

```

---

For each contour, if it is sufficiently large, we extract its bounding rectangle coordinates (**Lines 132-140**). Then we add it to the `rects` list (**Line 143**).

Now we will split the screen with either a vertical or horizontal line:

---

```

144     # check if the direction is vertical
145     if args["mode"] == "vertical":
146         # draw a horizontal line in the center of the frame -- once an
147         # object crosses this line we will determine whether they were
148         # moving 'up' or 'down'
149         cv2.line(frame, (0, H // 2), (W, H // 2), (0, 255, 255), 2)
150
151     # otherwise, the direction is horizontal
152     else:
153         # draw a vertical line in the center of the frame -- once an
154         # object crosses this line we will determine whether they
155         # were moving 'left' or 'right'
156         cv2.line(frame, (W // 2, 0), (W // 2, H), (0, 255, 255), 2)

```

---

If our "mode" is "vertical", we draw a horizontal line in the middle of the screen (**Lines 146-150**). Otherwise, if our "mode" is "horizontal", we draw a vertical line in the center of the screen (**Lines 153-157**).

Either line will serve as a visual indication of the point at which a person must pass to be counted.

**Let's count our objects — this is the heart of the driver script:**

---

```

158     # use the centroid tracker to associate the (1) old object
159     # centroids with (2) the newly computed object centroids
160     objects = ct.update(rects)
161
162     # loop over the tracked objects
163     for (objectID, centroid) in objects.items():
164         # grab the trackable object via its object ID
165         to = trackableObjects.get(objectID, None)
166         color = (0, 0, 255)
167
168         # create a new trackable object if needed

```

```

169     if to is None:
170         to = TrackableObject(objectID, centroid)
171
172     # otherwise, there is a trackable object so we can utilize it
173     # to determine direction
174     else:
175         # find the direction and update the list of centroids
176         dc.find_direction(to, centroid)
177         to.centroids.append(centroid)
178
179     # check to see if the object has been counted or not
180     if not to.counted:
181         # find the direction of motion of the people
182         directionInfo = dc.count_object(to, centroid)
183
184     # otherwise, the object has been counted and set the
185     # color to green indicate it has been counted
186     else:
187         color = (0, 255, 0)

```

---

**Line 160** calls `update` on our centroid tracker to associate old object centroids with the freshly computed centroids. Under the hood, this is where **Steps #2 through #5** from Section [19.2.2.2](#) from the previous section take place.

From there, we'll loop over the centroid `objects` beginning on **Line 163**.

The goals of this loop includes (1) tracking objects, (2) determining the direction the objects are moving, and (3) counting the objects depending on their direction of motion.

**Line 165** grabs a trackable object via its ID.

**Line 166** sets the default centroid + ID `color` to *red* for now. It will soon become *green* as the person becomes counted.

**Lines 169 and 170** create a new trackable object if needed.

Otherwise, the trackable object's direction is determined from its centroid history using the `DirectionCounter` class (**Line 176**). The centroid history is updated via **Line 177**. Finally, if the trackable object hasn't been counted yet (**Line 180**), it is counted (**Line 182**). Otherwise, it has already been counted and we must ensure that its `color` is *green* as it has passed the counting line (**Lines 186 and 187**). **Line 190** stores the trackable object in our `trackableObjects` dictionary.

Be sure to refer to the `CentroidTracker`, `TrackableObject`, and `DirectionCounter` classes at this point to see what is going on under the hood in **Lines 160-190**.

The remaining three codeblocks are for display/video annotation and housekeeping. Let's annotate our frame:

---

```

189     # store the trackable object in our dictionary
190     trackableObjects[objectID] = to
191
192     # draw both the ID of the object and the centroid of the
193     # object on the output frame
194     text = "ID {}".format(objectID)
195     cv2.putText(frame, text, (centroid[0] - 10, centroid[1] - 10),
196                 cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)
197     cv2.circle(frame, (centroid[0], centroid[1]), 4, color, -1)

```

---

**Lines 190-197** annotate each object with a small dot and an ID number.

Now let's annotate the object counts in the corner:

---

```

199     # extract the people counts and write/draw them
200     if directionInfo is not None:#
201         for (i, (k, v)) in enumerate(directionInfo):
202             text = "{}: {}".format(k, v)
203             cv2.putText(frame, text, (10, H - ((i * 20) + 20)),
204                         cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255), 2)

```

---

Using the `directionInfo`, we extract the up/down or left/right object counts. The text is displayed in the corner of the frame.

From here we'll finish out our loop and then clean up:

---

```

206     # put frame into the shared queue for video writing
207     if writerProcess is not None:#
208         frameQueue.put(frame)
209
210     # show the output frame
211     cv2.imshow("Frame", frame)
212     key = cv2.waitKey(1) & 0xFF
213
214     # if the `q` key was pressed, break from the loop
215     if key == ord("q"):
216         break
217
218     # update the FPS counter
219     fps.update()
220
221     # stop the timer and display FPS information
222     fps.stop()
223     print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))
224     print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
225
226     # terminate the video writer process
227     if writerProcess is not None:

```

```

228     writeVideo.value = 0
229     writerProcess.join()
230
231 # if we are not using a video file, stop the camera video stream
232 if not args.get("input", False):
233     vs.stop()
234
235 # otherwise, release the video file pointer
236 else:
237     vs.release()
238
239 # close any open windows
240 cv2.destroyAllWindows()

```

---

**Lines 207 and 208** add an annotated frame to the `frameQueue`. The `writerProcess` will process the frames in the background using multiprocessing. The frame is shown on the screen until the `q` (quit) keypress is detected (**Lines 211-216**). The FPS counter is updated via **Line 219**. Assuming we've broken out of the `while` loop, **Lines 222-240** perform cleanup and print FPS statistics.

In the next section, we'll deploy our people counter!

### 19.2.7 Deploying the RPi People Counter

In this chapter so far, we've reviewed the mechanics of tracking and counting objects. Then we brought it all together with our driver script. We've now reached the fun part — putting it to the test.

Remember, this people counter differs from the one presented on my blog (<http://pyimg.co/gak6>) [72]. This script is optimized such that the Raspberry Pi can successfully count people.

When you're ready, go ahead and execute the following command using the input video to test with:

---

```
$ python people_counter.py --mode vertical \
    --input videos/veritcal_01.mp4 \
    --output output/vertical_01.avi
```

---

Figure 19.4 shows three separate frames from people counter processing. There are no issues with the *left* and *center* frames. However, on the *right* two people are only being tracked as one centroid (ID #9). Obviously, this results in a counting error.

But is there a way around it?

Of course you could tune your morphological operations (i.e. the erosion and dilation kernels). Doing so would fix the problem in this case, but not under all scenarios.



Figure 19.4: Examples of people counting using a background subtraction + centroid tracking method. On the right, notice that two contours blobs were touching (ID #9), indicating that only one centroid is tracked; this results in a counting error and shows the disadvantage of this chapter's methodology. A more accurate object detector (HOG, SSD) would have resulted in both of these people being independent objects (covered in the *Hacker Bundle*).

Does this limit us? Are there alternatives?

Yes, but with a severely resource-constrained device, it seems that we have no choice but to rely on a pipeline similar to the one discussed in this chapter (background subtraction with centroid tracking). An alternative, assuming you have the processing horsepower, would be to rely on an object detection model (HOG or SSD) every  $N$  frames which will better discern the difference between separate people. We add the horsepower to a RPi (a Movidius Neural Compute Stick) and will learn such a method in the *Hacker Bundle*.

Our RPi 3B+ achieved **19.65 FPS** when benchmarking this project. Our goal was essentially 8FPS or higher to ensure that we counted each walking person, so we did well on the pipeline speed.

The RPi 4 (1GB version) achieved a whopping **35.74 FPS**. Taking it even further, the RPi 4 (4GB version) achieved an **insanely high 50.06 FPS**. As you can see, the RPi 4 is more than than 2x faster than the RPi 3B+.

With a live webcam, FPS will certainly help with accuracy such as during a running race while people are moving faster. The more often centroids are correlated via the `CentroidTracker`, the better.

You can run the code with other command line arguments to suit your needs. To run the people counting app on a live video feed, be sure to omit the `--input` command line argument (you can leave off the `--output` argument if you don't want to save the likely large video file to disk either):

---

```
$ python people_counter.py --mode vertical
```

---

Or, if you want to count people moving horizontally, use a live webcam feed, and output

the resulting video to disk, be sure to change the `--mode` accordingly:

---

```
$ python people_counter.py --mode horizontal  
--output output/webcam_output.avi
```

---

### 19.2.8 Tips and Suggestions

When you mount your people counter in a public/private place, be sure you follow all surveillance laws in your area.

Using the background subtraction method, you will need to tune the minimum contour size (**Line 135** of the driver script). The size you choose will be dependent upon how far the camera is from the people and the type of lens you use. Furthermore any frame resizing you perform to speed up your pipeline will also impact contour sizes. The only way to determine this value is by experimentation.

For the example video used during development, only erosion morphological operations were applied to the background subtraction mask. You may need to add dilation. Additionally, you may need to adjust the kernel size and number of iterations. This will be dependent upon noise in the image (there shouldn't be too much for a static ground) and the size of the objects/frame.

Finally, it may be the case that simple background subtraction is not sufficient for your particular task. Background subtraction, while efficient (especially on resource constrained devices such as the RPi), is still a basic algorithm that has *no semantic understanding* of the video stream itself.

Instead, you may need to leverage dedicated object detection algorithms, including Haar cascades, HOG + Linear SVM, and deep learning-based detectors. We'll be covering how to utilize these more advanced methods inside the *Hacker Bundle* of this text.

## 19.3 Summary

In this chapter, we designed a people counter based on a number of concepts.

Background subtraction and contours were used to find moving objects (i.e. people). We implemented a Haar Cascade object detector to locate people.

We developed a `CentroidTracker` to maintain a lookup table for objects in the frame. This class was responsible for registering/deregistering objects, and updating their locations. Our `TrackableObject` class keeps track of a person's ID, current location, and historical locations. From there, the `DirectionCounter` class counted left/right (horizontal) or up/down

(vertical) moving people depending on the set mode.

Given that this chapter spans many pages including three new classes and a driver script, it is easy to become lost. If you're experiencing this feeling, then I suggest you open up the project folder in your editor/IDE and study it on your screen. Then open up the book and review the chapter once again.

In the next chapter, we're going to apply what we learned here to traffic counting. The concepts are the same, but there are a selection of additional considerations we have to make for implementation.



## Chapter 20

# Building a Traffic Counter

Counting traffic is valuable data for municipalities and governments so that they can plan road and highway projects in the future. Traffic data per hour or per five minutes may be useful to morning and evening news reporters, so they can inform us of traffic slowdowns.

Some solutions are quite expensive and rely on cellular tower triangulation of cell phone signals from our phones as they zoom down the highway. But can we do this in a much cheaper manner with computer vision?

It only makes sense to have an Internet connected device with a camera looking at and analyzing the traffic. As a first thought, you may consider streaming raw video somewhere. While acceptable, streaming video from an IoT camera 24x7 to a central server to process the video would be very costly (especially over a cellular connection).

Instead, counting traffic “at the edge” makes more sense. Then you can simply stream textual/numerical traffic count data (not video frames) to a database somewhere in the cloud for analysis.

But the question remains:

*“Can we count traffic on a resource-constrained, low-power, and low-cost IoT device?”*

In fact we can, using the Raspberry Pi as our edge device.

In this chapter we’ll use the same concept described in our previous chapter on people counting to count traffic. There a few differences and additions we’ll highlight which ensure accurate traffic counting.

Buckle up and let’s go!

## 20.1 Chapter Learning Objectives

In this chapter, we will:

- i. Review similarities between our traffic counting and people counting implementations
- ii. Discuss additions and changes such that our traffic counting system is more accurate
- iii. Learn how and where to deploy the traffic counter
- iv. Discuss iterative computer vision development, and review how the development of this chapter and the people counting chapter progressed

Let's get started.

## 20.2 Similarities and Differences in this Project Compared to People Counting



Figure 20.1: When the "diff\_flag" is set in the configuration, the program will open with the "set\_points" window. Using your mouse, select the centerpoint with your mouse (green dot), between the opposing traffic flows (white arrows) and at the far end of where vehicles are visible. From there, press the **s** key on your keyboard to "save" the setting. The program will then open a fresh GUI window to begin traffic counting.

There are many parallels with this project as compared to people counting on the Raspberry Pi (Chapter 19). First, we will use a combination of:

- Background subtraction and contours
- Trackable objects
- Centroid tracking
- Direction counting

Each of these components are identical or nearly identical to our previous chapter's project. There is a key difference in the `CentroidTracker` class which facilitates better association between two centroids. Traffic tends to move in two directions and this project makes the assumption that you aren't working with a one-way street.

To explain the concept here, go ahead and make the assumption that traffic is moving up and down through your frame as shown in Figure 20.1 (the software also supports traffic moving left/right through the frame, just make sure you set the centerpoint accordingly).

When the script first launches, you will be able to select the centerpoint (using your mouse) to indicate the spot between the *left side* and *right side* of traffic flow.

This point will usually be in the top half of the frame in approximately the middle of the frame as shown in Figure 20.1. From there, inside the `CentroidTracker` class, a check is made to see if two different centroids are on opposite sides of the road. If so, we know that these two centroids should not be associated with each other.

**Remark.** *For best performance, try to mount your camera on an overpass or pole in the middle of the flow of traffic moving past in two directions (up/down). The camera should be able to see about 1/4 to 1/2 mile of highway, as shown in Figure 20.1.*

## 20.3 Traffic Counting via Background Subtraction on the RPi

In this section, we'll begin by reviewing our project structure and configuration file.

From there, we will go through each of our classes which make traffic counting possible. Both `CentroidTracker` and `DirectionCounter` are a little different than those presented in the previous chapter, whereas `TrackableObject` remains the same.

Finally, we'll walkthrough the traffic counting application code and deploy it in the field.

### 20.3.1 Project Structure

---

```
|-- config
|   |-- config.json
|-- output
```

---

```

|   |-- highway-resized-short.avi
|-- pyimagesearch
|   |-- utils
|   |   |-- __init__.py
|   |   |-- conf.py
|   |-- centroidtracker.py
|   |-- directioncounter.py
|   |-- trackableobject.py
|-- videos
|   |-- highway-orig.mp4
|   |-- highway-resized-short.avi
|-- traffic_counter.py

```

---

Input videos for testing are included in the `videos/` directory. Our input video credits go to Vlad Kiraly, a YouTube user who has posted the “*Relaxing Highway Traffic*” video [75].

Our `output/` folder will be where we’ll store processed videos. One example video is included.

The `pyimagesearch` module contains three classes key to traffic counting: (1) `TrackableObject`, (2) `CentroidTracker`, and (3) `DirectionCounter`. As noted in Section 20.2, there are a handful of differences and features in this project made possible by the three classes that are a little different than people counting; we will review those changes in detail.

We’ll also review the `traffic_counter.py` implementation. This script takes advantage of all the aforementioned classes, in order to count traffic on a resource-constrained device.

### 20.3.2 Our Configuration File

Go ahead and open up the `config.json` file and inspect the configuration parameters:

---

```
{
    // flag indicating if the user wants to initialize a difference
    // point
    "diff_flag": true,

    // minimum vehicle contour area
    "min_area": 350,

    // centroid tracker arguments
    "max_disappeared": 10,
    "max_distance": 120,

    // offset used to draw the counting line
    "x_offset": 0,
    "y_offset": 100,
```

---

```
// limit used to avoid vehicles that are too far away
"limit": 130
}
```

---

Our "diff\_flag" (**Line 4**) is a boolean indicating if user intervention is required at the beginning of the program to set the difference point.

The minimum vehicle contour area should be set appropriately so that small motion areas are ignored (**Line 7**).

The maximum number of frames until a `TrackableObject` is marked as disappeared is held by "max\_disappeared" on **Line 10**. Centroids will be associated together only if they are no greater than the value contained in "max\_distance" on **Line 11**.

Our counting line will be drawn based on an offset (**Lines 14 and 15**).

Vehicles that are further than "limit" pixels from the `diffPt` will be ignored.

### 20.3.3 Trackable Objects

The `TrackableObject` class is identical to the one presented in Section 19.2.3 of Chapter 19 on people counting.

As a brief review, in order to track and count an object in a video stream, we need an easy way to store information regarding the object itself, including:

- Its object ID
- Its previous centroids (so we can easily compute the direction the object is moving)
- Whether or not the object has already been counted

To accomplish all of these goals we can define an instance of `TrackableObject` — open up the `trackableobject.py` file and review the following code:

---

```
1  class TrackableObject:
2      def __init__(self, objectID, centroid):
3          # store the object ID, then initialize a list of centroids
4          # using the current centroid
5          self.objectID = objectID
6          self.centroids = [centroid]
7
8          # initialize a boolean used to indicate if the object has
9          # already been counted or not
10         self.counted = False
```

---

We will have multiple trackable objects — one for each vehicle that is being tracked in the frame. Each object will have the three attributes shown on **Lines 5-10**. The `TrackableObject` constructor accepts and stores the `objectID` and `centroid`. The constructor initializes `centroids` as a list with the first `centroid`. Soon, `centroids` will have a history of a trackable object's locations.

The constructor also initializes `counted` as `False`, indicating that the object has not been counted yet. Whenever the `DirectionCounter` detects that a vehicle has moved across the counting line, the `counted` flag will be updated.

#### 20.3.4 Centroid Tracker

Our `CentroidTracker` class is responsible for keeping track of our trackable objects by associating and dissociating centroids.

As discussed in Section [20.2](#), the `CentroidTracker` class contains an added feature for traffic counting such that our objects are more accurately associated. As a quick review, our traffic counting app allows the user to select the middle of two opposite-direction flows of traffic. Figure [20.1](#) shows a green dot where the user clicked their mouse to mark the middle of the traffic flow. The `CentroidTracker` class has additional code (**Lines 27 and 28** as well as **Lines 124-163**) that accommodate this additional datapoint to ensure better tracking.

We'll walk through the entire class together to ensure we understand the additions.

Let's begin — go ahead and open up `centroidtracker.py`:

---

```

1 # import the necessary packages
2 from scipy.spatial import distance as dist
3 from collections import OrderedDict
4 import numpy as np
5
6 class CentroidTracker:
7     def __init__(self, maxDisappeared=50, maxDistance=50):
8         # initialize the next unique object ID along with two ordered
9         # dictionaries used to keep track of mapping a given object
10        # ID to its centroid and number of consecutive frames it has
11        # been marked as "disappeared", respectively
12        self.nextObjectID = 0
13        self.objects = OrderedDict()
14        self.disappeared = OrderedDict()
15
16        # store the number of maximum consecutive frames a given
17        # object is allowed to be marked as "disappeared" until we
18        # need to deregister the object from tracking
19        self.maxDisappeared = maxDisappeared
20
21        # store the maximum distance between centroids to associate

```

---

```

22         # an object -- if the distance is larger than this maximum
23         # distance we'll start to mark the object as "disappeared"
24         self.maxDistance = maxDistance
25
26         # initialize the movement direction and difference point
27         self.direction = None
28         self.diffPt = None

```

---

**Lines 2-4** import necessary packages (the same ones as our previous chapter).

The first change is to our `CentroidTracker` constructor. ***There are two new class variables noted in the last two bullets:***

- `nextObjectID`: A counter used to assign unique IDs to each object (**Line 12**). In the case that an object leaves the frame and does not come back for `maxDisappeared` frames, a new (next) object ID would be assigned.
- `objects`: A dictionary that utilizes the object ID as the key and the centroid ( $x$ ,  $y$ )-coordinates as the value (**Line 13**).
- `disappeared`: Maintains number of consecutive frames (value) a particular object ID (key) has been marked as “lost” (**Line 14**).
- `maxDisappeared`: The number of consecutive frames an object is allowed to be marked as “lost/disappeared” until we deregister the object (**Line 19**).
- `maxDistance`: The maximum distance between centroids to associate an object. If the distance exceeds this value, then the object will be marked as disappeared (**Line 24**).
- `direction`: Indicates the direction mode, either “vertical” or “horizontal”. **Line 27** is **new** to this chapter’s project and will enable us to track objects that are moving left-to-right or up-and-down.
- `diffPt`: Marks the center of where traffic is flowing in opposite directions. Either an  $x$ -coordinate in the case of vertical traffic or a  $y$ -coordinate in the case of horizontal traffic. Our `traffic_counter.py` app will set this value directly via a mouse click event. **Line 28** is **new** to this chapter’s project.

Both of these **new** values (`direction` and `diffPt`) are initialized to `None` upon instantiation of the centroid tracking object.

Next comes our `register` and `deregister` methods, both of which are identical to the previous chapter:

---

```

30     def register(self, centroid):
31         # when registering an object we use the next available object

```

```

32     # ID to store the centroid
33     self.objects[self.nextObjectID] = centroid
34     self.disappeared[self.nextObjectID] = 0
35     self.nextObjectID += 1
36
37     def deregister(self, objectID):
38         # to deregister an object ID we delete the object ID from
39         # both of our respective dictionaries
40         del self.objects[objectID]
41         del self.disappeared[objectID]

```

---

When an `update` is made to our `CentroidTracker` object, it will register new vehicle centroids with an ID and/or remove (`deregister`) objects that have disappeared for greater than `maxDisappeared` frames. These two methods work to manage the `objects` and `disappeared` ordered dictionaries in addition to the `nextObjectID`.

Let's review the `update` method now, ***including 40 lines of new code***. Briefly, here's what the `update` method accomplishes:

- **Accepts** bounding box `rects` of our motion contours (from the background subtraction motion tracking method concept).
- **Registers** new trackable objects.
- **Associates** existing trackable objects with previously registered objects, based upon Euclidean distance between centroids.
- **Deregisters** missing trackable objects if they have disappeared for a specified amount of frames.
- **Returns** the updated list of trackable objects.

So where's our change to accommodate our `diffPt` (the point between opposite moving flows of traffic)?

Simply, the change is in the **association** component of this method. Recall that we want our cars to only be associated if they're in the same-direction-flowing lanes (`lanes` is plural in the case of a highway with multiple lanes in each direction). We need to insert logic to test for this condition. These changes occur later in **Lines 124-163**; however, let's review the lines leading up to **Line 124** for completeness:

---

```

43     def update(self, rects):
44         # check to see if the list of input bounding box rectangles
45         # is empty
46         if len(rects) == 0:
47             # loop over any existing tracked objects and mark them

```

```

48         # as disappeared
49     for objectID in list(self.disappeared.keys()):
50         self.disappeared[objectID] += 1
51
52         # if we have reached a maximum number of consecutive
53         # frames where a given object has been marked as
54         # missing, deregister it
55         if self.disappeared[objectID] > self.maxDisappeared:
56             self.deregister(objectID)
57
58         # return early as there are no centroids or tracking info
59         # to update
60     return self.objects
61
62     # initialize an array of input centroids for the current frame
63     inputCentroids = np.zeros((len(rects), 2), dtype="int")
64
65     # loop over the bounding box rectangles
66     for (i, (startX, startY, endX, endY)) in enumerate(rects):
67         # use the bounding box coordinates to derive the centroid
68         cX = int((startX + endX) / 2.0)
69         cY = int((startY + endY) / 2.0)
70         inputCentroids[i] = (cX, cY)

```

---

The `update` method, defined on **Line 43**, accepts a list of bounding box rectangles, presumably from an object detector (Haar cascade, HOG + Linear SVM, SSD, Faster R-CNN, etc.). The format of the `rects` parameter is assumed to be a tuple with this structure: `(startX, startY, endX, endY)`.

If there are no detections, we'll mark `objectIDs` as `disappeared` and go ahead and deregister them if they haven't returned for `maxDisappeared` frames (**Lines 46-60**). Then, we go ahead and return the `objects` early.

Otherwise, we have a lot of work to do.

We begin by initializing and populating `inputCentroids` from our `rects` provided to this `update` method (**Lines 63-70**).

If there are currently no objects we are tracking, we'll register each of the new objects:

```

72     # if we are currently not tracking any objects take the input
73     # centroids and register each of them
74     if len(self.objects) == 0:
75         for i in range(0, len(inputCentroids)):
76             self.register(inputCentroids[i])

```

---

Otherwise, we need to update any existing object  $(x, y)$ -coordinates based on the centroid location that minimizes the Euclidean distance between them:

---

```

78     # otherwise, are are currently tracking objects so we need to
79     # try to match the input centroids to existing object
80     # centroids
81 else:
82     # grab the set of object IDs and corresponding centroids
83     objectIDs = list(self.objects.keys())
84     objectCentroids = list(self.objects.values())
85
86     # compute the distance between each pair of object
87     # centroids and input centroids, respectively -- our
88     # goal will be to match an input centroid to an existing
89     # object centroid
90     D = dist.cdist(np.array(objectCentroids), inputCentroids)
91
92     # in order to perform this matching we must (1) find the
93     # smallest value in each row and then (2) sort the row
94     # indexes based on their minimum values so that the row
95     # with the smallest value is at the *front* of the index
96     # list
97     rows = D.min(axis=1).argsort()
98
99     # next, we perform a similar process on the columns by
100    # finding the smallest value in each column and then
101    # sorting using the previously computed row index list
102    cols = D.argmin(axis=1)[rows]

```

---

The updates to existing tracked objects take place beginning at the `else` on [Line 81](#). The goal is to track the objects and to maintain correct object IDs — this process is accomplished by computing the Euclidean distances between all pairs of `objectCentroids` and `inputCentroids`, followed by associating object IDs that minimize the Euclidean distance.

[Lines 83 and 84](#) grab `objectIDs` and `objectCentroid` values. Using the centroids data, we perform centroid matching/association. We seek to minimize the index values with the smallest corresponding distance at the front of the lists.

In order to accomplish the association, we must (1) find the smallest value in each row and (2) sort the row indexes based on the minimum values ([Line 97](#)). We perform a very similar process on the columns, finding the smallest value in each column, and then sorting them based on the ordered rows ([Line 102](#)).

The next step is to use the distances to try to associate object IDs:

---

```

104         # in order to determine if we need to update, register,
105         # or deregister an object we need to keep track of which
106         # of the rows and column indexes we have already examined
107         usedRows = set()
108         usedCols = set()
109

```

---

---

```

110      # loop over the combination of the (row, column) index
111      # tuples
112      for (row, col) in zip(rows, cols):
113          # if we have already examined either the row or
114          # column value before, ignore it
115          if row in usedRows or col in usedCols:
116              continue
117
118          # if the distance between centroids is greater than
119          # the maximum distance, do not associate the two
120          # centroids to the same object
121          if D[row, col] > self.maxDistance:
122              continue

```

---

Here we initialize two sets to determine which row and column indexes we have already used (**Lines 107 and 108**). Keep in mind that a set is similar to a list but it contains only *unique* values.

Then we loop over the combinations of (row, col) index tuples (**Line 112**) in order to update our object centroids.

If we've already used either this row or column index, we ignore it and `continue` to loop (**Lines 115 and 116**)

Similarly, if the distance between centroids exceeds the `maxDistance`, then we will not associate the two centroids (**Lines 121 and 122**).

***Then comes two new conditionals with logic to handle our diffPt (this next block is entirely new compared to the previous chapter's people counting implementation of Centroid Tracker):***

---

```

124      # check if the direction is vertical and the
125      # difference point is set
126      diffPtExists = self.diffPt is not None
127      if self.direction == "vertical" and diffPtExists:
128          # get the x-axis coordinates from both object and
129          # input centroid
130          (oX, iX) = (objectCentroids[row][0],
131                        inputCentroids[col][0])
132
133          # check if oX is less than the difference point
134          # and iX is greater than difference point, if so
135          # skip this iteration
136          if oX < self.diffPt and iX > self.diffPt:
137              continue
138
139          # otherwise, check if oX is greater than the
140          # difference point and iX is less than difference
141          # point, if so skip this iteration

```

```

142         elif oX > self.diffPt and iX < self.diffPt:
143             continue
144
145             # otherwise, check if the direction is horizontal and
146             # the difference point is set
147             elif self.direction == "horizontal" and diffPtExists:
148                 # get the y-axis coordinates from both object and
149                 # input centroid
150                 (oY, iY) = (objectCentroids[row][1],
151                             inputCentroids[col][1])
152
153                 # check if oY is less than the difference point
154                 # and iY is greater than difference point, if so
155                 # skip this iteration
156                 if oY < self.diffPt and iY > self.diffPt:
157                     continue
158
159                 # otherwise, check if oY is greater than the
160                 # difference point and iY is less than difference
161                 # point, if so skip this iteration
162                 elif oY > self.diffPt and iY < self.diffPt:
163                     continue

```

---

First, we check to see if the `diffPtExists` (i.e. if the user has intervened with the mouse and keyboard to save the `diffPt`) via **Line 126**. Then we have two conditions to handle.

When the `direction` is "vertical" and the `diffPtExists` (**Line 127**), we'll proceed to grab the `x`-axis coordinates from the existing object and the input centroid (**Lines 130 and 131**). Then we'll **ignore** the input centroid if it is on the wrong side (left/right) of the `diffPt` (**Lines 136-143**).

Similarly, when the `direction` is "horizontal" and the `diffPtExists` (**Line 147**), we'll proceed to grab the `y`-axis coordinates from the existing object and the input centroid (**Lines 150-151**). Then we'll **ignore** the input centroid if it is on the wrong side (above/below) of the `diffPt` (**Lines 156-163**).

***That's it for additions to this class, everything else is the same.***

Our final two code blocks wrap up the `update` method:

---

```

165             # otherwise, grab the object ID for the current row,
166             # set its new centroid, and reset the disappeared
167             # counter
168             objectID = objectIDs[row]
169             self.objects[objectID] = inputCentroids[col]
170             self.disappeared[objectID] = 0
171
172             # indicate that we have examined each of the row and
173             # column indexes, respectively

```

---

```

174         usedRows.add(row)
175         usedCols.add(col)

```

---

If we've made it to this point (i.e. none of the `continue` control statements on **Line 116, 122, 137, 143, 157, or 163** have triggered), then we have found an input centroid that (1) has the smallest Euclidean distance to an existing centroid and (2) has not been matched with any other object. In this case, we update the object centroid and make sure to add the `row` and `col` to their respective `usedRows` and `usedCols` sets (**Lines 168-175**)

That concludes the `for` loop which began on **Line 112**.

There are also likely indexes in our `usedRows + usedCols` sets that we have *NOT* examined yet:

---

```

177     # compute both the row and column index we have NOT yet
178     # examined
179     unusedRows = set(range(0, D.shape[0])).difference(usedRows)
180     unusedCols = set(range(0, D.shape[1])).difference(usedCols)
181
182     # in the event that the number of object centroids is
183     # equal or greater than the number of input centroids
184     # we need to check and see if some of these objects have
185     # potentially disappeared
186     if D.shape[0] >= D.shape[1]:
187         # loop over the unused row indexes
188         for row in unusedRows:
189             # grab the object ID for the corresponding row
190             # index and increment the disappeared counter
191             objectID = objectIDs[row]
192             self.disappeared[objectID] += 1
193
194             # check to see if the number of consecutive
195             # frames the object has been marked "disappeared"
196             # for warrants deregistering the object
197             if self.disappeared[objectID] > self.maxDisappeared:
198                 self.deregister(objectID)
199
200             # otherwise, if the number of input centroids is greater
201             # than the number of existing object centroids we need to
202             # register each new input centroid as a trackable object
203             else:
204                 for col in unusedCols:
205                     self.register(inputCentroids[col])
206
207             # return the set of trackable objects
208             return self.objects

```

---

We must determine which centroid indexes we haven't examined yet and store them in two new convenient sets (`unusedRows` and `unusedCols`) on **Lines 179 and 180**.

Our conditional handles any objects that have become lost or if they've potentially disappeared. **Lines 186-201** deregister objects that have disappeared for greater than `maxDisappeared frames`.

Conversely, **Lines 203-205** register new centroids as trackable objects.

**Line 205** concludes the `else` block that began way back on **Line 81**.

Finally, we'll return the *updated* set of trackable objects to the calling method (**Line 208**).

Now that we've reviewed the `CentroidTracker` class, take the time now to (1) refer to the people counting Section 19.2.2 for a deeper explanation of the `update` method, and (2) consider how these additions in `CentroidTracker` will allow us to **associate vehicles only if they are going in the same directional traffic flow lanes**.

**Remark.** *Of course, once in a blue moon, there will be a drunk/dumb motorist driving the wrong way on the street/highway. We won't count them in this "edge case". Instead, we'll hope that the police find this person and bring them to a stop before they cause a serious head-on collision accident.*

### 20.3.5 Direction Counter

Recall from Chapter 19 that our `DirectionCounter` class determines the direction of a trackable object (in this case it will be vehicles) relative to its list of historical centroids. Then, if the vehicle goes above/below or left-of/right-of the counting threshold line, this class counts the vehicle.

The changes in the `DirectionCounter` class are minimal, but they do have an impact on the counting line. These changes will be highlighted as we review the class.

Go ahead and open up `directioncounter.py` and review the constructor:

---

```

1 # import the necessary packages
2 import numpy as np
3
4 class DirectionCounter:
5     def __init__(self, directionMode, X, Y):
6         # initialize the X and Y coordinates of the line
7         self.X = X
8         self.Y = Y
9
10        # initialize variables holding the direction of movement,
11        # along with counters for each respective movement (i.e.,
12        # left-to-right and top-to-bottom)
13        self.directionMode = directionMode
14        self.totalUp = 0
15        self.totalDown = 0

```

---

```

16         self.totalRight = 0
17         self.totalLeft = 0
18
19     # the direction the trackable object is moving in
20     self.direction = ""

```

---

In the constructor beginning on **Line 5**, we set the `x` and `y` coordinates rather than `H` and `W` frame dimensions, as we did in the last chapter (**Lines 7 and 8**).

Why?

Our `diffPt` will also serve as the very top or most right point where we'll then split the rest of the frame into two (based on a configurable offset) for traffic counting. Both `x` and `y` become the place where the counting line is established. Recall that for people counting, we just split the frame (i.e. place our counting line) exactly in the middle/center which is why we needed the frame dimensions.

From there, `directionMode`, the count totals, and trackable object's direction (**Lines 13-20**) are the same.

Our `find_direction` method is the exact same as our previous chapter's implementation:

---

```

22     def find_direction(self, to, centroid):
23         # check to see if we are tracking horizontal movements
24         if self.directionMode == "horizontal":
25             # the difference between the x-coordinate of the
26             # *current* centroid and the mean of *previous* centroids
27             # will tell us in which direction the object is moving
28             # (negative for 'left' and positive for 'right')
29             x = [c[0] for c in to.centroids]
30             delta = centroid[0] - np.mean(x)
31
32             # determine the sign of the delta -- if it is negative,
33             # the object is moving left
34             if delta < 0:
35                 self.direction = "left"
36
37             # otherwise if the sign is positive, the object is moving
38             # right
39             elif delta > 0:
40                 self.direction = "right"
41
42             # otherwise we are tracking vertical movements
43             elif self.directionMode == "vertical":
44                 # the difference between the y-coordinate of the
45                 # *current* centroid and the mean of *previous* centroids
46                 # will tell us in which direction the object is moving
47                 # (negative for 'up' and positive for 'down')
48                 y = [c[1] for c in to.centroids]

```

---

---

```

49     delta = centroid[1] - np.mean(y)
50
51     # determine the sign of the delta -- if it is negative,
52     # the object is moving up
53     if delta < 0:
54         self.direction = "up"
55
56     # otherwise, if the sign of the delta is positive, the
57     # object is moving down
58     elif delta > 0:
59         self.direction = "down"

```

---

The `find_direction` function accepts a trackable object (`to`) and a single `centroid`.

Recall that `to.centroids` contains a historical listing of an object's position. **Line 30 or 49** grabs all `x`-values or `y`-values from the trackable object's historical centroids, respectively.

The `delta` indicates an object's direction. It is calculated by averaging all previous centroid `x` or `y`-coordinate values and subtracting it from the very first (current) value.

The `direction` will either be left/right or up/down.

The `count_objects` method then uses the `directionMode`, object's `direction`, in conjunction with which side of the counting line the object is on to count objects:

---

```

61 def count_object(self, to, centroid):
62     # initialize the output list
63     output = []
64
65     # check if the direction of the movement is horizontal
66     if self.directionMode == "horizontal":
67         # if the object is currently left of center and is
68         # moving left, count the object as moving left
69         leftOfCenter = centroid[0] < self.X
70         if self.direction == "left" and leftOfCenter:
71             self.totalLeft += 1
72             to.counted = True
73
74         # otherwise, if the object is right of center and moving
75         # right, count the object as moving right
76         elif self.direction == "right" and not leftOfCenter:
77             self.totalRight += 1
78             to.counted = True
79
80         # construct a list of tuples with the count of objects
81         # that have passed in the left and right direction
82         output = [("Left", self.totalLeft),
83                   ("Right", self.totalRight)]
84
85     # otherwise the direction of movement is vertical

```

```

86         elif self.directionMode == "vertical":
87             # if the centroid is above the middle and is moving
88             # up, count the object as moving up
89             aboveMiddle = centroid[1] < self.Y
90             if self.direction == "up" and aboveMiddle:
91                 self.totalUp += 1
92                 to.counted = True
93
94             # otherwise, if the object is moving down and is below
95             # the middle, count the object as moving down
96             elif self.direction == "down" and not aboveMiddle:
97                 self.totalDown += 1
98                 to.counted = True
99
100            # return a list of tuples with the count of objects that
101            # have passed in the up and down direction
102            output = [("Up", self.totalUp), ("Down", self.totalDown)]
103
104        # return the output list
105    return output

```

---

**The two notable but minor changes** here are on **Line 69** and **Line 80**.

Our `x` value (from the constructor) is used directly to determine left/right of center. Similarly, our `Y` value (from the constructor) is used directly to determine above/below the middle.

For people counting, on the other hand, we used `w // 2` (instead of `x`) or `H // 2` (instead of `Y`) to mark the counting line.

**Lines 66-83** count horizontally flowing traffic, building the `output` which contains the left-right counts. And likewise, **Lines 85-102** count vertically flowing traffic and build a similar output containing up/down counts.

**Line 105** returns the `output` count information to the driver script which we will review next.

### 20.3.6 Traffic counting Driver Script

Our `CentroidTracker` and `DirectionCounter` modifications are complete.

Now comes the fun part — implementing the app!

Let's build our traffic counting app script. There are a number of differences in comparison to the `people_counter.py` driver, so we'll walkthrough the `traffic_counter.py` script in its entirety.

When you're ready, go ahead and open a new file called `traffic_counter.py` and insert the following code:

---

```

1 # import the necessary packages
2 from pyimagesearch.directioncounter import DirectionCounter
3 from pyimagesearch.centroidtracker import CentroidTracker
4 from pyimagesearch.trackableobject import TrackableObject
5 from pyimagesearch.utils import Conf
6 from multiprocessing import Process
7 from multiprocessing import Queue
8 from multiprocessing import Value
9 from imutils.video import VideoStream
10 from imutils.video import FPS
11 import numpy as np
12 import argparse
13 import imutils
14 import time
15 import cv2

```

---

We begin with our imports including our `DirectionCounter`, `CentroidTracker`, and `TrackableObject` on **Lines 2-4**.

**Lines 6-8** import three classes from `multiprocessing` used for our `write_video` function that will be used as a separate `Process`. A `Queue` of frames added by the main process will be consumed by the `write_video` process. A boolean flag `Value` will indicate when writing to video should begin.

You should be familiar with all other imports at this point.

Let's implement our mouse click event function:

---

```

17 def set_points(event, x, y, flags, param):
18     # declare a global variable to store difference point
19     global diffPt
20
21     # check if a left button down event has occurred
22     if event == cv2.EVENT_LBUTTONDOWN:
23         # if the direction is set as vertical, set the difference
24         # point as the x-coordinates, otherwise set it as the
25         # y-coordinate
26         diffPt = x if param[0] == "vertical" else y

```

---

The `set_points` function handles a left mouse click event and assigns the `x` or `y` coordinate as appropriate.

As you can see, the function accepts five parameters. We only care about four of them:

- `event`: The type of event such as a left or right mouse click.
- `x`: The `x`-coordinate where the mouse was clicked.

- `y`: The `y`-coordinate where the mouse was clicked.
- `param`: Custom callback function parameters may be set when we set the mouse callback. We pass the direction mode (either "vertical" or "horizontal") as the only parameter.

We access the global `diffPt` variable (**Line 19**), which we will then set if we receive the correct `event` (**Line 22**). The specific `event` we're looking for is `cv2.EVENT_LBUTTONDOWN` when the left mouse button is pressed (also corresponds to a regular tap on your trackpad if you don't have a physical mouse). When that `event` triggers, we set the `diffPt` with the `x` or `y` coordinate in the frame depending upon whether our direction mode is "vertical" or "horizontal" (**Line 26**).

Next up, we'll define our `write_video` process:

---

```

28 def write_video(output, writeVideo, frameQueue, W, H):
29     # initialize the FourCC and video writer object
30     fourcc = cv2.VideoWriter_fourcc(*"MJPG")
31     writer = cv2.VideoWriter(output, fourcc, 30,
32         (W, H), True)
33
34     # loop while the write flag is set or the output frame queue is
35     # not empty
36     while writeVideo.value or not frameQueue.empty():
37         # check if the output frame queue is not empty
38         if not frameQueue.empty():
39             # get the frame from the queue and write the frame
40             frame = frameQueue.get()
41             writer.write(frame)
42
43     # release the video writer object
44     writer.release()

```

---

The `write_video` process was implemented to **speed up FPS** using multiprocessing (refer to **Rev #6 and Rev #7** in Section 20.4 to how this was determined to be necessary).

The main process will add newly processed frames to a `frameQueue` and the `write_video` process will churn on the frames as they become available. Let's review the parameters for `write_video` (**Line 28**):

- `output`: The output path plus filename to the video file (.avi is usually most compatible).
- `writeVideo`: A boolean implemented with the process-safe `Value` class. It indicates when/if the video writing should commence/cease.
- `frameQueue`: A process-safe queue of frames to be consumed by this process. The frames will be written to disk in FIFO order.

- W: Width of frame.
- H: Height of frame.

**Lines 30-31** initialize the video `VideoWriter` as `writer`.

From there, a `while` loop begins on **Line 36**. The `while` loop will run provided that the `writeVideo` flag is true or the `frameQueue` is not empty. Frames are then written to disk as they become available (**Lines 40-41**). When the `writeVideo` flag is `False` and the `frameQueue` becomes empty, then the writer is released (**Line 44**).

Back to our main process, command line arguments need to be parsed next:

---

```

46 # construct the argument parser and parse the arguments
47 ap = argparse.ArgumentParser()
48 ap.add_argument("-c", "--conf", required=True,
49     help="Path to the input configuration file")
50 ap.add_argument("-m", "--mode", type=str, required=True,
51     choices=[ "horizontal", "vertical"],
52     help="direction in which vehicles will be moving")
53 ap.add_argument("-i", "--input", type=str,
54     help="path to optional input video file")
55 ap.add_argument("-o", "--output", type=str,
56     help="path to optional output video file")
57 args = vars(ap.parse_args())
58
59 # load the configuration file
60 conf = Conf(args["conf"])

```

---

Our script handles four command line arguments:

- `--conf`: The path to our configuration file
- `--mode`: The direction mode with two choices, either "horizontal" or "vertical"
- `--input`: An optional path to a video file, if you are testing and not using a live webcam stream
- `--output`: An optional path to an output video file

Both `--conf` and `--mode` are required arguments.

The configuration is loaded via **Line 60** and from here forward values will be accessible via the `conf` dictionary.

Now we will begin initializing components and variables:

---

```

62 # initialize the MOG foreground background subtractor object
63 mog = cv2.bgsegm.createBackgroundSubtractorMOG()
64
65 # initialize and define the dilation kernel
66 dKernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
67
68 # initialize the video writer process
69 writerProcess = None
70
71 # initialize the frame dimensions (we'll set them as soon as we read
72 # the first frame from the video)
73 W = None
74 H = None
75
76 # instantiate our centroid tracker and initialize a dictionary to
77 # map each unique object ID to a trackable object
78 ct = CentroidTracker(conf["max_disappeared"], conf["max_distance"])
79 trackableObjects = {}

```

---

**Line 63** instantiates our MOG background subtraction object and **Line 66** sets our dilation kernel.

Our writerProcess and width/height dimensions are each set to None (**Lines 69-74**). We will initialize these values in the while loop soon.

**Line 78** instantiates our CentroidTracker as ct with our configuration settings.

**Line 79** then initializes an empty dictionary to hold trackableObjects. The CentroidTracker will manage this dictionary.

From here, let's go ahead and initialize our camera or video file stream:

---

```

81 # if a video path was not supplied, grab a reference to the webcam
82 if not args.get("input", False):
83     print("[INFO] starting video stream...")
84     # vs = VideoStream(src=0).start()
85     vs = VideoStream(usePiCamera=True).start()
86     time.sleep(2.0)
87
88 # otherwise, grab a reference to the video file
89 else:
90     print("[INFO] opening video file...")
91     vs = cv2.VideoCapture(args["input"])

```

---

And then we'll handle our "diff\_flag" when it is set to true in the configuration file:

---

```

93 # check if the user wants to use the difference flag feature
94 if conf["diff_flag"]:

```

```

95     # initialize the start counting flag and mouse click callback
96     start = False
97     cv2.namedWindow("set_points")
98     cv2.setMouseCallback("set_points", set_points,
99         [args["mode"]])
100
101    # otherwise, the user does not want to use it
102 else:
103     # set the start flag as true indicating to start traffic counting
104     start = True
105
106    # initialize the direction info variable (used to store information
107    # such as up/down or left/right vehicle count) and the difference
108    # point (used to differentiate between left and right lanes)
109    directionInfo = None
110    diffPt = None

```

---

If the "diff\_flag" is set to true in config.json (**Line 94**), we need to prepare for our mouse click intervention. Recall that the purpose of our diffPt is to ensure better centroid tracking of motion objects. The goal is that vehicles moving in one flow of traffic will stay associated without being incorrectly associated with vehicles moving in the other direction. This is accomplished by clicking and saving a point between the traffic flows.

**Line 96** sets the start flag. In this "diff\_flag" mode, traffic counting will not begin (i.e. it will not start) until the diffPt is set.

**Line 97** creates a GUI window called "set\_points" which is subsequently activated with a mouse callback event handler. **Line 98** initializes/sets the mouse callback with the name of the GUI window ("set\_points"), event handler function (set\_points), and the direction mode from the command line args. Take the time now to go back and review the set\_points callback function on **Lines 17-26**.

Otherwise, when the configuration variable, "diff\_flag" is set to false, we mark the counting as ready to start (**Lines 102-104**).

**Line 109** initializes directionInfo to None. This value will hold a tuple of up/down or left/right traffic counts.

**Line 110** initializes diffPt as None. This value will only be set if both (1) the config value, "diff\_flag" is set, and (2) the user intervenes with the mouse and keyboard. Otherwise, the diffPt will not be used.

We're finally ready to commence processing frames:

```

112    # loop over frames from the video stream
113 while True:
114     # grab the next frame and handle if we are reading from either

```

---

---

```

115      # VideoCapture or VideoStream
116      frame = vs.read()
117      frame = frame[1] if args.get("input", False) else frame
118
119      # if we are viewing a video and we did not grab a frame then we
120      # have reached the end of the video
121      if args["input"] is not None and frame is None:
122          break

```

---

In the while loop beginning, first the frame is grabbed, extracted and checked (**Lines 113-122**).

Let's handle the case where we have started counting which spans from **Line 126** to **Line 295**:

---

```

124      # check if the start flag is set, if so, we will start traffic
125      # counting
126      if start:
127          # if the frame dimensions are empty, grab the frame
128          # dimensions, instantiate the direction counter, and set the
129          # centroid tracker direction
130          if W is None or H is None:
131              # start the frames per second throughput estimator
132              fps = FPS().start()
133
134          (H, W) = frame.shape[:2]
135          dc = DirectionCounter(args["mode"],
136                                W - conf["x_offset"], H - conf["y_offset"])
137          ct.direction = args["mode"]
138
139          # check if the difference point is set, if it is, then
140          # set it in the centroid tracker object
141          if diffPt is not None:
142              ct.diffPt = diffPt

```

---

**Line 126** begins the block indicating that traffic counting is ongoing.

On the first iteration, the FPS counter isn't started, our frame dimensions aren't set, Direction Counter isn't initialized, and our centroid tracker's mode and diffPt are not ready. **Lines 130-142** perform these initializations.

There are two differences here, as compared to the people counting implementation. First, our DirectionCounter's (x, y)-coordinates are set based on both an "x\_offset" and "y\_offset". Second, the centroid tracker's diffPt (ct.diffPt) is set if the diffPt is available from our set\_points mouse callback function.

We'll now start our writerProcess if we are writing to an output video:

---

```

144     # begin writing the video to disk if required
145     if args["output"] is not None and writerProcess is None:
146         # set the value of the write flag (used to communicate when
147         # to stop the process)
148         writeVideo = Value('i', 1)
149
150         # initialize a shared queue for the exchange frames,
151         # initialize a process, and start the process
152         frameQueue = Queue()
153         writerProcess = Process(target=write_video, args=(
154             args["output"], writeVideo, frameQueue, W, H))
155         writerProcess.start()

```

---

If a path to an "output" video is provided in the command line args and the writerProcess hasn't been started yet (**Line 145**), then we'll go ahead and prepare for writing video frames to disk:

- **Line 148** sets the writeVideo flag to true (using an integer value of 1).
- **Line 152** initializes our frameQueue.
- **Lines 153-155** then start a new Process called writerProcess, passing the required parameters.

Each iteration of frame processing undergoes background subtraction — let's work on that aspect now:

---

```

157     # initialize a list to store the bounding box rectangles
158     # returned by background subtraction model
159     rects = []
160
161     # convert the frame to grayscale image and then blur it
162     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
163     gray = cv2.GaussianBlur(gray, (5, 5), 0)
164
165     # apply the MOG background subtraction model which returns
166     # a mask
167     mask = mog.apply(gray)
168
169     # apply dilation
170     dilation = cv2.dilate(mask, dKernel, iterations=2)
171
172     # find contours in the mask
173     cnts = cv2.findContours(dilation.copy(), cv2.RETR_EXTERNAL,
174                             cv2.CHAIN_APPROX_SIMPLE)
175     cnts = imutils.grab_contours(cnts)

```

---

**Line 159** initializes an empty list to store bounding box rectangles which will be returned by the background subtraction model.

**Lines 162 and 163** convert the frame to grayscale and blur it slightly. Background subtraction is then applied to the gray frame (**Line 167**).

From there, a series of dilations are applied to enlarge contours (**Line 170**). Contours are then found and extracted (**Lines 173-175**).

Let's process our contours and add the bounding boxes to `rects`:

---

```

177     # loop over each contour
178     for c in cnts:
179         # if the contour area is less than the minimum area
180         # required then ignore the object
181         if cv2.contourArea(c) < conf["min_area"]:
182             continue
183
184         # get the (x, y)-coordinates of the contour, along with
185         # height and width
186         (x, y, w, h) = cv2.boundingRect(c)
187
188         # check if direction is *vertical and the vehicle is
189         # further away from the line, if so then, no need to
190         # detect it
191         if args["mode"] == "vertical" and y < conf["limit"]:
192             continue
193
194         # otherwise, check if direction is horizontal and the
195         # vehicle is further away from the line, if so then,
196         # no need to detect it
197         elif args["mode"] == "horizontal" and x > conf["limit"]:
198             continue
199
200         # add the bounding box coordinates to the rectangles list
201         rects.append((x, y, x + w, y + h))

```

---

For each contour (provided it is sufficiently large), we extract its bounding rectangle coordinates (**Lines 178-186**).

Depending on the direction "`mode`" (in command line `args`) and whether the `x` or `y`-coordinate is less than or greater than "`limit`" (configured in `config.json`), we'll skip adding contours to `rects` (**Lines 191-198**). This limit ensures that we *ignore contours which are likely small at the edge of our frame*.

If both of those conditions pass, then we append the contour coordinates to `rects` (**Line 201**)

Now we will split the screen visually with vertical and horizontal lines (a little differently than

how we did for people counting):

---

```

203     # check if the direction is vertical
204     if args["mode"] == "vertical":
205         # draw a horizontal line in the frame -- once an object
206         # crosses this line we will determine whether they were
207         # moving 'up' or 'down'
208         cv2.line(frame, (0, H - conf["y_offset"]),
209                 (W, H - conf["y_offset"]), (0, 255, 255), 2)
210
211         # check if a difference point has been set, if so, draw
212         # a line dividing the two lanes
213         if diffPt is not None:
214             cv2.line(frame, (diffPt, 0), (diffPt, H),
215                     (255, 0, 0), 2)
216
217         # otherwise, the direction is horizontal
218     else:
219         # draw a vertical line in the frame -- once an object
220         # crosses this line we will determine whether they were
221         # moving 'left' or 'right'
222         cv2.line(frame, (W - conf["x_offset"], 0),
223                 (W - conf["x_offset"], H), (0, 255, 255), 2)
224
225         # check if a difference point has been set, if so, draw a
226         # line dividing the two lanes
227         if diffPt is not None:
228             cv2.line(frame, (0, diffPt), (W, diffPt),
229                     (255, 0, 0), 2)

```

---

If our "mode" is "vertical", we draw a horizontal line in the middle of the screen based on "y\_offset" to serve as a visual indication of the point at which a vehicle must pass to be counted ([Lines 204-209](#)). Additionally, if the `diffPt` happens to be set, a vertical line is drawn to visually separate the two opposing flows of traffic ([Lines 213-215](#)). Be sure to refer to Figure [20.2](#) to see the lines in the case of *vertical mode*.

Otherwise, if our "mode" is "horizontal", we draw a vertical line in the center of the screen based on "x\_offset" to serve as a visual indication of the point at which a vehicle must pass to be counted ([Lines 218-223](#)). Additionally, if the `diffPt` happens to be set, a horizontal line is drawn to visually separate the two flows of traffic ([Lines 227-229](#)).

### **Let's count our objects — this is the heart of the driver script:**

---

```

231     # use the centroid tracker to associate the (1) old object
232     # centroids with (2) the newly computed object centroids
233     objects = ct.update(rects)
234
235     # loop over the tracked objects

```

---

```

236     for (objectID, centroid) in objects.items():
237         # check to see if a trackable object exists for the
238         # current object ID and initialize the color
239         to = trackableObjects.get(objectID, None)
240         color = (0, 0, 255)
241
242         # create a new trackable object if needed
243         if to is None:
244             to = TrackableObject(objectID, centroid)
245
246         # otherwise, there is a trackable object so we can
247         # utilize it to determine direction
248         else:
249             # find the direction and update the list of centroids
250             dc.find_direction(to, centroid)
251             to.centroids.append(centroid)
252
253             # check to see if the object has been counted or not
254             if not to.counted:
255                 # find the direction of motion of the vehicles
256                 directionInfo = dc.count_object(to, centroid)
257
258                 # otherwise, the object has been counted and set the
259                 # color to green indicate it has been counted
260             else:
261                 color = (0, 255, 0)
262
263             # store the trackable object in our dictionary
264             trackableObjects[objectID] = to
265
266             # draw both the ID of the object and the centroid of the
267             # object on the output frame
268             text = "ID {}".format(objectID)
269             cv2.putText(frame, text, (centroid[0] - 10,
270                                     centroid[1] - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
271                                     color, 2)
272             cv2.circle(frame, (centroid[0], centroid[1]), 4, color,
273                                     -1)

```

---

**Lines 231-273** are *identical* to **Lines 158-197** of `people_counter.py` (Section 19.2.6). Let's review.

**Line 233** calls `update` on our centroid tracker to associate old object centroids with the freshly computed centroids. Under the hood, this is where **Steps #2 through #5** from Section 19.2.2.2 (of the prior chapter) take place.

From there, we'll loop over the centroid objects beginning on **Line 236**.

**The goals of this loop include (1) tracking objects, (2) determining the direction the objects are moving, and (3) counting the objects depending on their direction of motion.**

**Line 239** grabs a trackable object via its ID. For now, the `color` of each trackable object will be *red* until it is counted (at which point it turns *green*).

**Lines 243 and 244** create a new trackable object if needed. Otherwise, the trackable object's direction is determined from its centroid history using the `DirectionCounter` class (**Line 250**). The centroid history is updated via **Line 251**.

If the trackable object hasn't been counted yet, it is counted (**Lines 254-256**). Otherwise, it is counted and we set the `color` to *green*.

**Line 264** stores the trackable object in our `trackableObjects` dictionary.

**Lines 268-273** draw the ID and centroids on the screen in *red* (not counted) or in *green* (counted once a centroid crosses the counting line).

Be sure to refer to the `CentroidTracker`, `TrackableObject`, and `DirectionCounter` classes at this point to see what is going on under the hood in **Lines 236-273**.

Now let's annotate the object counts in the corner:

---

```

275      # extract the traffic counts and write/draw them
276      if directionInfo is not None:
277          for i, (k, v) in enumerate(directionInfo):
278              text = "{}: {}".format(k, v)
279              cv2.putText(frame, text, (10, ((i * 20) + 20)),
280                          cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255), 2)

```

---

Once the object counts are written in the corner, we'll populate our `frameQueue`:

---

```

282      # put frame into the shared queue for video writing
283      if writerProcess is not None:
284          frameQueue.put(frame)

285      # show the output frame
286      cv2.imshow("Frame", frame)
287      key = cv2.waitKey(1) & 0xFF

288      # if the `q` key was pressed, break from the loop
289      if key == ord("q"):
290          break

291      # update the FPS counter
292      fps.update()

```

---

A frame is put at the end of the `frameQueue` for the `writerProcess` to handle in the background (**Lines 283-284**).

**Lines 287-295** then display the frame until the `q` key is pressed in addition to updating our FPS counter.

**Now we'll handle our new `diffPt` feature in the case that `start` is `False`:**

---

```

297     # otherwise, the user has to select a difference point
298 else:
299     # show the output frame
300     cv2.imshow("set_points", frame)
301     key = cv2.waitKey(1) & 0xFF
302
303     # if the `s` key was pressed, start traffic counting
304     if key == ord("s"):
305         # begin counting and eliminate the informational window
306         start = True
307         cv2.destroyWindow("set_points")

```

---

If we are in "diff\_mode", the first iteration of the `while` loop will land us in the `else` block beginning on **Line 298**. In this case, we simply display frames (**Line 300**) until the user intervenes with the mouse to find the `diffPt` and then saves with the `s` key (**Lines 301-307**).

That's a wrap for our `while` loop, so let's cleanup:

---

```

309 # stop the timer and display FPS information
310 fps.stop()
311 print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))
312 print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
313
314 # terminate the video writer process
315 if writerProcess is not None:
316     writeVideo.value = 0
317     writerProcess.join()
318
319 # if we are not using a video file, stop the camera video stream
320 if not args.get("input", False):
321     vs.stop()
322
323 # otherwise, release the video file pointer
324 else:
325     vs.release()
326
327 # close any open windows
328 cv2.destroyAllWindows()

```

---

FPS statistics are printed, the `writerProcess` is joined, the video stream is stopped, and the GUI windows are destroyed.

Great job implementing a traffic counter. This script's walkthrough spanned multiple pages.

Thus, at this point, I would recommend opening the script (accompanying your book downloads) to be able to scroll through it as a whole on your screen.

In this way, the indentation blocks will make more sense since some of them spanned multiple code blocks.

### 20.3.7 Traffic Counting Results

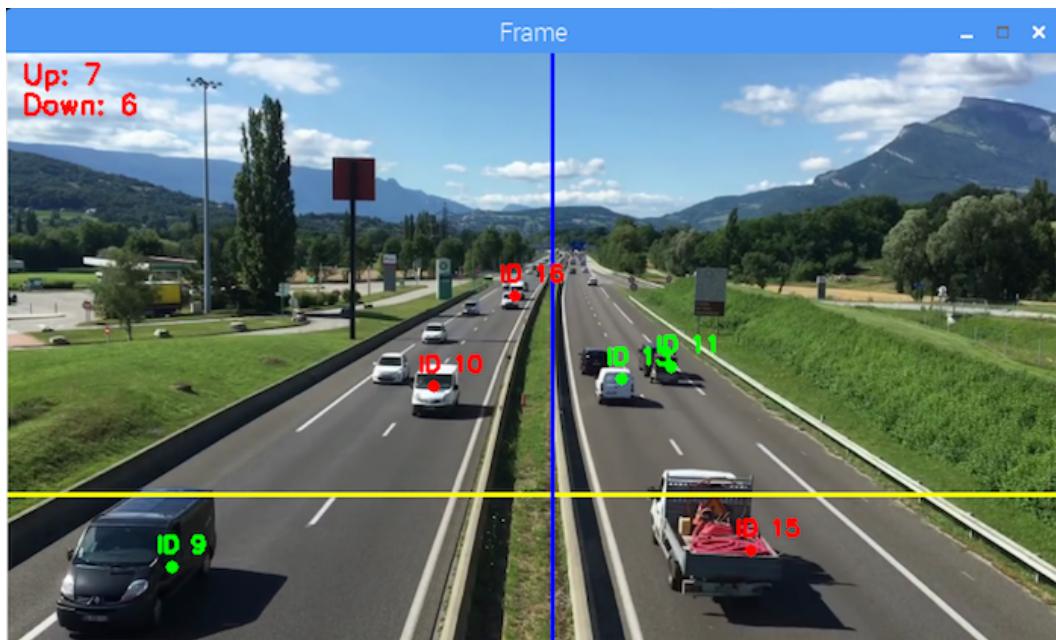


Figure 20.2: Traffic counting in action. *Red* centroids/IDs indicate vehicles that have not been counted yet (i.e. they haven't crossed the *yellow* counting line), whereas *green* centroids/IDs have been counted. The *blue* vertical line is based on wherever the user set the `diffPt` with their mouse and keyboard; this line ensures that centroids don't hop between opposite flowing traffic lane to different vehicles, ensuring more-accurate traffic counting.

With our changes to the classes in place and our traffic counting app in hand, we're now ready to put it to the test.

The best way to test for performance is using a video file *before* you deploy your RPi on the highway. Be sure to use the `--input` command line argument to test using an input video (otherwise your camera will be activated). You can also (optionally) write to an output video to share with people.

When you're ready, enter the following command to start the program (be ready to intervene with your mouse and keyboard):

---

```
$ python traffic_counter.py --conf config/config.json \
--mode vertical \
```

Revision	FPS on RPi 3B+	Revision Description
<b>Rev #1:</b>	<1	background subtraction + dlib correlation tracking
<b>Rev #2:</b>	$\approx 5$ to $\approx 8$	optical flow
<b>Rev #3:</b>	$\approx 4$ to $\approx 6.6$	background subtraction (no tracking)
<b>Rev #4:</b>	$\approx 3.7$	background subtraction + MOSSE tracking
<b>Rev #5:</b>	$\approx 6.4$	no <code>cv2.VideoWriter</code>
<b>Rev #6:</b>	$\approx 16$	no <code>cv2.VideoWriter</code> + no <code>imutils.resize</code>
<b>Rev #7:</b>	$\approx 17.65$ ( $\approx 32.8$ on RPi 4)	multiprocessing of <code>cv2.VideoWriter</code> + no <code>imutils.resize</code>

Table 20.1: A table summary people/traffic counting Frames Per Second (FPS) statistics for the various revisions undergone during development.

---

```
--input videos/highway-resized-short.mp4 \
--output output/output.avi
```

---

Now find the GUI window on your screen. Assuming you have the "diff\_flag" set to true in `config.json`, then (1) left click between the opposing direction flows of traffic about 1/3 of the way from the top, and (2) press the `s` key to save (Figure 20.1). Counting will commence as shown in Figure 20.2.

The result is an accurate and high FPS traffic counting system.

Be sure to read the next section for FPS statistics that were achieved as this project was *iteratively developed with incremental improvements*.

## 20.4 Leading Up to a Successful Project Requires Multiple Revisions

Reaching the point of obtaining  $\geq 15$  FPS (making it possible for real-time traffic counting or real-time people counting) was no easy feat. In this section, I'll briefly describe what my team and I did to ensure a successful project. Despite the resulting project fitting nicely into a chapter, there were multiple months of discussions and revisions logged in our project management software and Git repository.

It is important to incrementally improve your computer vision algorithms, and even discuss them with other engineers when you hit a bump in the road (if you own the *Complete Bundle*, I would suggest posting in the community forums).

I'm recalling our team discussions from our project notes and will explain each revision which led up to a successful methodology for counting traffic.

Let's dive into each of the revisions which are summarized in Table 20.1

**Rev #1:** First, we knew that we'd need to use background subtraction + centroid correlation instead of object detection (cars can appear small from far away). We proceeded with background subtraction. **Dlib's correlation tracker** was used initially. The problem is that it is too slow to run *one*, let alone *multiple*, tracker instances on the RPi. Discovering that dlib's correlation tracker would not be viable led to the following list of developmental experiments to perform:

- If we want to perform object tracking on the Pi, OpenCV's MOSSE is the only one suitable. All other trackers (including dlib's) will just be too slow. The issue is that *multiple* MOSSE trackers will not work on the Pi.
- Swap out the computationally expensive motion trackers for optical flow/Lucas-Kande. We weren't 100% sure this will work but we knew it would be faster. We hypothesized that it might not work on the RPi at all, but that would be a great learning experience.
- Try Haar cascades on every frame (no tracking).
- Attempt background subtraction on every frame (no tracking).

**Rev #2:** This revision involved **Optical Flow** based object tracking. It ran at  $\approx$  52 FPS on a laptop but only  $\approx$  5 FPS on a RPi 3B+. Resizing the frame to `width=300` bumped the FPS to  $\approx$  8 FPS. This method was definitely too slow and not viable for the RPi 3B+.

**Rev #3:** Next we tried **background subtraction only (no tracking)**. This led to  $\approx$  4 FPS. The same method with a resized `width=300` frame led to  $\approx$  6.6 FPS. This method is not viable either.

**Rev #4:** Background subtraction and the **MOSSE Object Tracker** yielded  $\approx$  3.7 FPS — similar results as background subtraction with no tracking.

**Pressing pause on experiments to profile our scripts:** Based on **Revs #2 through #4**, we decided we should profile our scripts for performance and consider optimizations. To learn about profiling a Python script, be sure to refer to Chapter 23, Section 23.3.4. Upon profiling, we made two observations from the data that printed out in the terminal: (1) Writing to videos takes significant CPU cycles, and (2) OpenCV's default resizing algorithm isn't fast on the RPi.

We profiled our experiments in **Rev #1 through Rev #4** (an example is shown in Figure 20.3). Profiling led us to realize that writing to video and resizing frames is slow on the RPi; resizing frames and writing to video are the first two lines listed in the output. Armed with this information, we decided our next revisions would be:

- Background subtraction (no tracking) **without** writing to video

```
pi@raspberrypi: ~object-tracking - □ x
File Edit Tabs Help
<pstats.Stats object at 0x76a28370>
>>> p = pstats.Stats('output_mosse.stats')
>>> p.strip_dirs().sort_stats("time").print_stats(15)
Sat May 25 02:22:19 2019      output_mosse.stats

    227834 function calls (224196 primitive calls) in 204.157 seconds

Ordered by: internal time
List reduced from 1559 to 15 due to restriction <15>

      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1341   52.291   0.039   52.291   0.039 {resize}
        1341   46.363   0.035   46.363   0.035 {method 'write' of 'cv2.VideoWriter' objects}
        1341   30.149   0.022   30.149   0.022 {method 'read' of 'cv2.VideoCapture' objects}
        1072   29.047   0.027   29.047   0.027 {method 'apply' of 'cv2.BackgroundSubtractor'
objects}
        1836   25.065   0.014   25.065   0.014 {method 'init' of 'cv2.Tracker' objects}
        1341    3.522   0.003    3.522   0.003 {waitKey}
        1072    2.666   0.002    2.666   0.002 {GaussianBlur}
        6228    2.516   0.000    2.516   0.000 {putText}
          1    2.385   2.385  204.158 traffic_counting_mosse.py:5(<module>)
        1072    1.976   0.002    1.976   0.002 {findContours}
         461    1.276   0.003    1.276   0.003 {method 'update' of 'cv2.Tracker' objects}
        1341    1.270   0.001    1.270   0.001 {imshow}
          45    0.972   0.022    0.986   0.022 {built-in method _imp.create_dynamic}
        1072    0.969   0.001    0.969   0.001 {dilate}
        1072    0.898   0.001    0.898   0.001 {cvtColor}

<pstats.Stats object at 0x768d4830>
>>> █
```

Figure 20.3: Performance profiling traffic counting with the MOSSE object tracker. Performance profiling will be covered in detail in Chapter 23.

- Background subtraction (no tracking) ***without*** writing to video ***or*** resizing

**Rev #5:** Background subtraction without VideoWriter yielded  $\approx 6.4$  FPS

**Rev #6:** Background subtraction without VideoWriter or imutils.resize yielded  $\approx 16$  FPS. This is a huge improvement!

**Back to the drawing board:** Based on Rev #5 and #6, we determined that we should move `VideoWriter` to a separate multiprocessing process with a `frameQueue` to be consumed by the process. Our next revision (the same revision presented in this chapter) includes multiprocessing. Since this chapter and the previous people counting chapter were developed in tandem, you should remember that the `write_video` function (which becomes instantiated as `writerProcess` for multiprocessing) is implemented in both "people counter" and "traffic counter".

**Rev #7:** This revision includes **multiprocessing** with a separate process for writing to video. The results are promising and achieved  $\approx 17.65$  **FPS** using the separate video writing process on the RPi 3B+.

**Bringing in the big guns:** We developed **Rev #1 through Rev #7** using the RPi 3B+ (the fastest Raspberry Pi at the time). Adding more computational horsepower is always an option

too. Ironically the RPi 4 was released at about this time and we later tested on an RPi 4 as soon as it arrived in the mail.

Astoundingly, the **Rev #7** implementation achieved  $\approx 30.14$  **FPS** on the 1GB RAM hardware and  $\approx 32.80$  **FPS** on the 4GB RAM hardware version. The crazy thing is that the folks at the Raspberry Pi Foundation have managed to approximately double performance (for most tasks) and they kept the price of the hardware the same. Kudos to the Raspberry Pi Foundation and engineers that make the little SBC possible with welcomed improvements.

All I can say at this point as I review our notes is: **Bingo! We did it!** Computer vision development is an iterative process — you should not expect to land on the perfect algorithm in a single attempt.

Instead, you should plan and expect to conduct experiments, collect data in your notes file, and make revisions. I urge you to keep this in mind if you are contracting/consulting, as well as estimating costs and the timeline of a successful project.

Engineers tend to underestimate costs and severely miss the mark when it comes to estimating the time it will take to complete a project. You'll improve this valuable skill with practice and by reading books like this one.

## 20.5 Summary

In this chapter we implemented a traffic counting system using a similar concept as our people counting system. This application was developed specifically with resource-constrained computers like the Raspberry Pi in mind.

While it may seem like we hit the nail on the head the first time, you need to realize that we missed the nail quite a few times before we drove it into the wood. I encourage you to review the previous section to learn about the bumps in the road we overcame in order to build a successful high FPS traffic counting system. High FPS is important because traffic is ideally moving quickly to get people from point A to point B.

If you have the luxury of a more powerful computer for counting traffic, you could use a combination of a deep learning object detector and more advanced object trackers – just keep in mind that deep learning object detectors don't always work well for small objects (i.e. vehicles that are far away). You could try to push the limits of a Raspberry Pi using a Movidius or Coral, but it may just be more advantageous to use a more powerful SBC or full size computer altogether.

That said, we've demonstrated successful traffic counting very efficiently for resource constrained devices in this chapter and there is a lot to learn from it. Discussions in the community are encouraged (provided you own the *Complete Bundle* of this text).

## Chapter 21

# Measuring Object Sizes

In this chapter you will learn how to measure the size of objects in images. The technique you will learn here is called the “**pixels per metric ratio**” — it is a simple method used to perform a rough estimation of object size.

More advanced methods exist, including computing the intrinsic/extrinsic parameters of a camera. We’ll be covering those methods in the *Hacker Bundle* of this text, but for the time being, the pixels per metric ratio will be sufficient.

Additionally, this chapter is meant to be a *precursor* to the next chapter on building a prescription pill recognition system. We’ll be measuring the size of pills in this chapter, but you could just as easily measure the size of another object.

Make sure you take the time to understand how we’re estimating object sizes — we’ll be using it as one component in our very extensive and highly detailed case study, coming up next.

### 21.1 Chapter Learning Objectives

In this chapter you will:

- Learn about the “pixels per metric ratio”, a simple calibration technique
- Implement a Python script to measure the size of objects in an image
- View the results of our object measurement script

In the following chapter we will build on this example and enable the script to:

- i. Measure the size of objects in *real-time video* (versus static images)

- ii. Build an entire pill measurement application



Figure 21.1: We'll use a United States quarter as our reference object and ensure it is always placed as the left-most object in the image, making it easy for us to extract it by sorting contours based on their location.

## 21.2 A Simple Camera Calibration

Before we start to implement Python code to measure object size, let's first explore the “pixels per metric” ratio.

### 21.2.1 The “Pixels Per Metric” Ratio

In order to compute the size of a pill, we'll be using a method called the **pixels per metric ratio** [76] which is based on the **triangle similarity**. More advanced camera calibration methods, including intrinsic/extrinsic camera parameters, can be used to compute object size as well, but they require significantly more work and attention. The pixels per metric ratio will be sufficient for our application here.

So, how does the pixel per metric method work?

In order to determine the size of an object in an image, we first need to perform a simple “calibration” using a reference object. Our reference object should have two important properties:

- **Property #1:** We should know the *dimensions of this object* (in terms of width or height) in a *measurable unit* (such as millimeters, inches, etc.).

- **Property #2:** We should be able to easily find this reference object in an image, either based on the *placement* of the object (for example, the reference object is always placed in the top-left corner of an image) or via *appearance* (for example, being a distinctive color or shape, unique from all other objects in the image). In either case, our reference object should be *uniquely identifiable* in some manner.

The example in this chapter:

- i. Uses the United States quarter as our reference object (Figure 21.1)
- ii. Ensures that the quarter is always the *left-most* object in the image

By guaranteeing the quarter is the left-most object, we can sort our object contours from left-to-right, grab the contour (which will by definition always be the first contour in the sorted list), and use it to define our `pixels_per_metric`, defined below:

```
pixels_per_metric = object_width / known_width
```

A US quarter has a `known_width` of 0.955 inches. Now, suppose that our `object_width` (measured in pixels) is computed to be 150 pixels (based on the associated bounding box). The `pixels_per_metric` is therefore:

```
pixels_per_metric = 150px / 0.955in = 157px
```

Thus, implying there are approximately 157 pixels per every 0.955 inches in our image. Using this ratio we can compute the size of objects in an image.

For more details on this metric, please refer to the following blog posts on PyImageSearch:

- *Ordering coordinates clockwise with Python and OpenCV* [77] (<http://pyimg.co/dj13h>)
- *Find distance from camera to object/marker using Python and OpenCV* [78] (<http://pyimg.co/l7fm3>)
- *Measuring distance between objects in an image with OpenCV* [79] (<http://pyimg.co/yf2of>)

## 21.3 Implementing Object Size Measurement

In this section we will implement our object size measurement script. We'll start by reviewing our directory structure, then move on to the implementation. Finally, we'll review our results.

### 21.3.1 Project Structure

The directory structure for this project is simple and straightforward:

---

```
|-- object_size.py
|-- pills.png
```

---

The `object_size.py` script will be used to measure the size of objects in our `pill.png` file. We will implement this script in the next section.

### 21.3.2 Measuring Object Sizes with OpenCV

Now that we've reviewed our project structure, we can get started with the implementation. Open up the `object_size.py` file and insert the following code:

---

```
1 # import the necessary packages
2 from scipy.spatial import distance as dist
3 from imutils import perspective
4 from imutils import contours
5 import numpy as np
6 import argparse
7 import imutils
8 import cv2
9
10 def midpoint(ptA, ptB):
11     # compute and return the midpoint between the two input points
12     return ((ptA[0] + ptB[0]) * 0.5, (ptA[1] + ptB[1]) * 0.5)
```

---

**Lines 2-8** import our required Python packages, namely:

- i. The `perspective` submodule of `imutils` contains a function named `order_points` — this method will be used to order  $(x, y)$ -coordinates of a bounding box.
- ii. The `contours` submodule contains a method called `sort_contours`. As the name suggests, this function will sort the detected contours.

**Lines 10-12** define a helper method, `midpoint`. This function is used to compute the midpoint between two sets of  $(x, y)$ -coordinates.

Let's move on to parsing our command line arguments:

---

```
14 # construct the argument parser and parse the arguments
15 ap = argparse.ArgumentParser()
16 ap.add_argument("-i", "--image", required=True,
17     help="path to the input image")
18 ap.add_argument("-w", "--width", type=float, required=True,
19     help="width of the left-most object in the image (in inches)")
20 args = vars(ap.parse_args())
```

---

We require two command line arguments here:

- `--image`: The path to our input image that contains the objects we wish to measure.
- `--width`: The width (in inches) of the left-most object (which serves as our reference object).

We can now load our image and preprocess it:

---

```

22 # load the image, convert it to grayscale, and blur it slightly
23 image = cv2.imread(args["image"])
24 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
25 gray = cv2.GaussianBlur(gray, (7, 7), 0)
26
27 # perform edge detection, then perform a dilation + erosion to
28 # close gaps in between object edges
29 edged = cv2.Canny(gray, 50, 100)
30 edged = cv2.dilate(edged, None, iterations=1)
31 edged = cv2.erode(edged, None, iterations=1)
32
33 # find contours in the edge map
34 cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
35                         cv2.CHAIN_APPROX_SIMPLE)
36 cnts = imutils.grab_contours(cnts)
37
38 # sort the contours from left-to-right and initialize the
39 # pixels per metric calibration variable
40 (cnts, _) = contours.sort_contours(cnts)
41 pixelsPerMetric = None

```

---

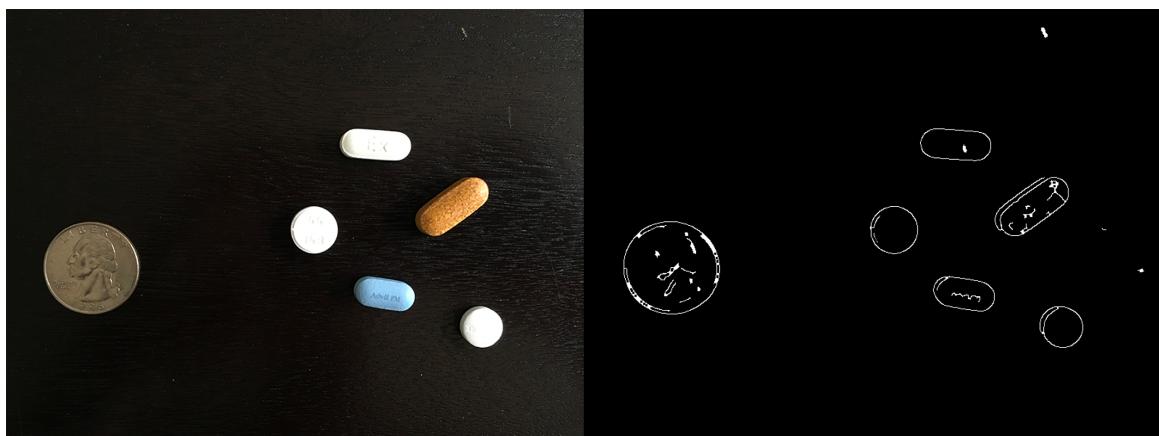


Figure 21.2: **Left:** Our original input image. **Right:** Output after applying grayscale conversion, blurring, edge detection, and a series of erosions and dilations. Notice how the outlines of each object in the image are clearly visible.

**Lines 23-25** load our image from disk, convert it to grayscale, and then smooth it using a Gaussian filter (used to reduce false-positive edges).

We then perform edge detection along with a dilation and erosion to close any gaps between edges in the map (**Lines 29-31**). Figure 21.2 visualizes our input in on the *left* and our output edge map (after morphological operations) on the *right*.

**Lines 34-36** find contours (i.e., outlines) that correspond to the objects in our edge map. These contours are then sorted from left-to-right (allowing us to later extract our reference object) on **Line 40**. We also initialize our `pixelsPerMetric` value on **Line 41**.

The next step is to loop over each of the contours:

---

```

43 # loop over the contours individually
44 for c in cnts:
45     # if the contour is not sufficiently large, ignore it
46     if cv2.contourArea(c) < 100:
47         continue
48
49     # compute the rotated bounding box of the contour
50     orig = image.copy()
51     box = cv2.boxPoints(cv2.minAreaRect(c))
52
53     # order the points in the contour such that they appear
54     # in top-left, top-right, bottom-right, and bottom-left
55     # order, then draw the outline of the rotated bounding
56     # box
57     box = perspective.order_points(box)
58     cv2.drawContours(orig, [box.astype("int")], -1, (0, 255, 0), 2)
59
60     # loop over the bounding box points and draw them in *red*
61     for (x, y) in box:
62         cv2.circle(orig, (int(x), int(y)), 5, (0, 0, 255), -1)

```

---

On **Line 44** we start looping over each of the individual contours. If the contour is not sufficiently large, we discard the region, presuming it to be noise left over from the edge detection process (**Lines 46 and 47**).

Provided that the contour region is sufficiently large, we compute the rotated bound box of object on **Line 51**.

**Line 57** handles arranging the rotated `box` coordinates in the following order:

- i. Top-left
- ii. Top-right
- iii. Bottom-right
- iv. Bottom-left

The implementation of the `order_points` function is outside the scope of this tutorial, but if you're interested in learning more about it, you can refer to the following blog post: <http://pyimg.co/djl3h> [77].

Lastly, **Lines 58-62** draw the *outline* of the object in *green*, followed by drawing the vertices of the bounding box rectangle in *small red circles*.

Now that we have our bounding box ordered, we can compute a series of midpoints:

---

```

64      # unpack the ordered bounding box, then compute the midpoint
65      # between the top-left and top-right coordinates, followed by
66      # the midpoint between bottom-left and bottom-right coordinates
67      (tl, tr, br, bl) = box
68      (tltrX, tltrY) = midpoint(tl, tr)
69      (blbrX, blbrY) = midpoint(bl, br)
70
71      # compute the midpoint between the top-left and top-right points,
72      # followed by the midpoint between the top-right and bottom-right
73      # points
74      (tblbIX, tblbIY) = midpoint(tl, bl)
75      (trbrX, trbrY) = midpoint(tr, br)
76
77      # draw the midpoints on the image in *blue*
78      cv2.circle(orig, (int(tltrX), int(tltrY)), 5, (255, 0, 0), -1)
79      cv2.circle(orig, (int(blbrX), int(blbrY)), 5, (255, 0, 0), -1)
80      cv2.circle(orig, (int(tblbIX), int(tblbIY)), 5, (255, 0, 0), -1)
81      cv2.circle(orig, (int(trbrX), int(trbrY)), 5, (255, 0, 0), -1)
82
83      # draw lines between the midpoints in *purple*
84      cv2.line(orig, (int(tltrX), int(tltrY)), (int(blbrX), int(blbrY)),
85                  (255, 0, 255), 2)
86      cv2.line(orig, (int(tblbIX), int(tblbIY)), (int(trbrX), int(trbrY)),
87                  (255, 0, 255), 2)

```

---

**Lines 67-69** unpack our ordered bounding box coordinates, followed by computing the midpoints between the top-left and top-right points, and finally the midpoint between the bottom-right points, respectively. We also compute the midpoints between the top-left/bottom-left and top-right/bottom-right points as well (**Lines 74 and 75**).

**Lines 78-81** draw *blue midpoints* on the image, followed by connecting the midpoints with *purple lines*.

Next, we need to initialize our `pixelsPerMetric` variable by investigating our reference object:

---

```

89      # compute the Euclidean distance between the midpoints
90      dA = dist.euclidean((tltrX, tltrY), (blbrX, blbrY))
91      dB = dist.euclidean((tblbIX, tblbIY), (trbrX, trbrY))

```

---

---

```

92     # if the pixels per metric has not been initialized, then
93     # compute it as the ratio of pixels to the supplied metric
94     # value (in this case, inches)
95
96     if pixelsPerMetric is None:
97         pixelsPerMetric = dB / args["width"]

```

---

First, we compute the Euclidean distance between the sets of midpoints (**Lines 90 and 91**). The `dA` variable will contain the *height* of the distance (in pixels) while `dB` will hold the *width* distance.

We make a quick check on **Line 96** to see if the `pixelsPerMetric` variable has been initialized, and if not, we divide `dB` by the `--width`, thus giving us our (approximate) pixels per inch.

Now that our `pixelsPerMetric` variable has been defined, we can measure the size of objects in an image:

---

```

99     # compute the size of the object along both axis
100    dimA = dA / pixelsPerMetric
101    dimB = dB / pixelsPerMetric
102
103    # draw the object sizes on the image in *white*
104    cv2.putText(orig, "{:.1f}in".format(dimA),
105                (int(tltrX - 15), int(tltrY - 10)), cv2.FONT_HERSHEY_SIMPLEX,
106                0.65, (255, 255, 255), 2)
107    cv2.putText(orig, "{:.1f}in".format(dimB),
108                (int(trbrX + 10), int(trbrY)), cv2.FONT_HERSHEY_SIMPLEX,
109                0.65, (255, 255, 255), 2)
110
111    # show the output image
112    cv2.imshow("Image", orig)
113    cv2.waitKey(0)

```

---

**Lines 100 and 101** compute the dimensions of the object (in inches) by dividing the respective Euclidean distances by the `pixelsPerMetric` value (see Section 21.2.1 for more information on how and why this ratio works).

**Lines 104-109** draw the dimensions of the object on our `image`, while **Lines 112 and 113** display the result to our screen.

### 21.3.3 Object Size Measurement Results

To test our object measurement script, just execute the following command:

---

```
$ python object_size.py --image pills.png --width 0.955
```

---

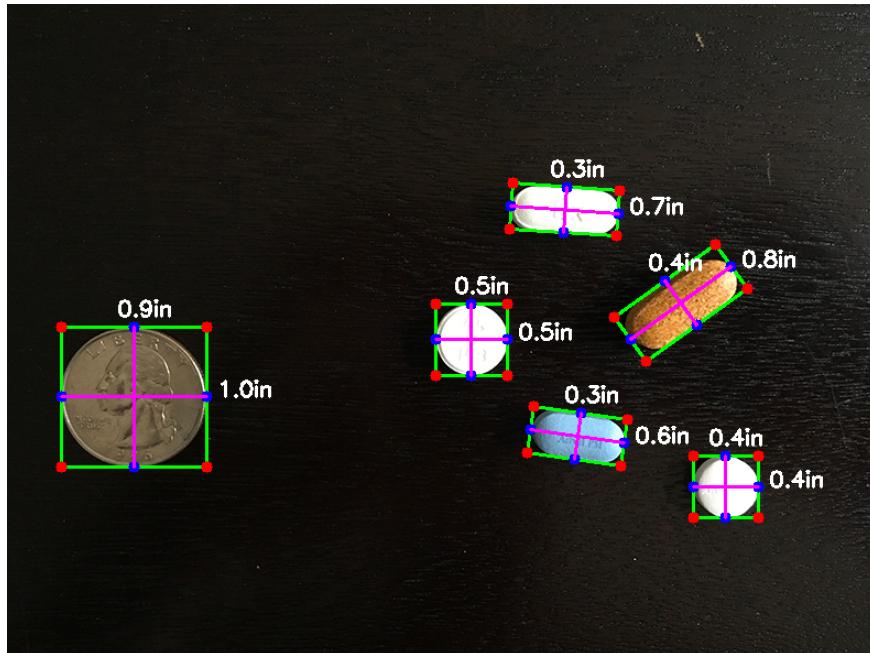


Figure 21.3: Our script is able to successfully measure the size of objects in our input image.

As Figure 21.3 demonstrates, we are able to successfully measure the size of our reference object (i.e., the quarter) and then use the reference object to measure the size of the *other objects* in the image (i.e., the pills).

While this particular example may not be too terribly exciting, it's important to note that nearly 50% of all 20,000+ prescription pills on the market in the United States are round and/or white, thus if we can filter pills based on their measurements, *we stand a better chance at accurately identifying the medication*.

We'll continue this example in the next chapter where we'll combine pill color, shape, and texture to aid shape in the identification.

## 21.4 Summary

In this chapter you learned how to use the “pixels per metric ratio” to perform a super basic calibration. More advanced calibration methods exist, namely computing the intrinsic/extrinsic parameters of a camera; however, for the purposes of this application, our simple calibration method was sufficient.

The pixels per metric ratio requires that we find a **reference object** in the image *before* we can measure the sizes of *other objects*.

The reference object must maintain two import properties:

- i. We should know the *dimensions* of the object (in measurable units)
- ii. We should be able to easily identify it in the image (based on color, location, etc.)

After understanding the pixels per metric ratio, we implemented it using Python and OpenCV to measure the size of objects in images. We used a United States quarter as a reference object, and ensured that the quarter was *always the left-most object* in the image.

After finding the quarter in the image we could perform our simple calibration using the pixels per metric ratio.

But what if you didn't know the location of your reference object ahead of time? Or, what if you wanted to create a "seamless" user experience that required a "less obvious" reference object? How would you go about building such an application? Our next chapter will address those exact questions.

## Chapter 22

# Building a Prescription Pill Recognition System

So far in this book we have explored a number of practical, hands-on projects that will enable you to use your Raspberry Pi for real-world computer vision projects — but perhaps the most *important* real-world project we are going to build is contained in this chapter.

Each year over 3.3 million injuries and deaths happen due to the incorrect prescription pill being taken. We call these occurrences *Adverse Drug Events* (ADEs).

Using computer vision we can reduce the number of ADEs that happen each year by empowering patients, doctors, and pharmacists with the ability to *recognize* and *visually validate* that the patient is indeed taking the correct pill.

This chapter is treated as a *case study*, and will be *very intensive* in various computer vision concepts including image processing, feature extraction, and performing Content-Based Image Retrieval (CBIR).

I've included abbreviated discussions of each algorithm inside this chapter, but I *strongly recommend* that you read through *Practical Python and OpenCV* (<http://pyimg.co/ppao>) [11] to ensure you understand the fundamentals before attempting to follow along with this chapter.

### 22.1 Chapter Learning Objectives

In this chapter you will:

- i. Understand the importance of prescription pill recognition
- ii. Learn how computer vision can help ID pills
- iii. Review three methods to characterize/quantify a pill

- iv. Write Python code to localize a pill in an image
- v. Create methods to quantify the color, shape, texture, and size of a pill
- vi. Build a database of pills to recognize
- vii. Put all the pieces to recognize a pill in an image

## 22.2 The Case for Prescription Pill Recognition

In this section we'll discuss why prescription pill identification is important, and how computer vision can help reduce the number of deaths and injuries that happen each year due to patients ingesting the incorrect medication.

### 22.2.1 Injuries, Deaths, and High Insurance Costs

Each year, over 3.3 million injuries and deaths occur due to taking the wrong prescription pill medication. These events are *not* limited to just overdoses — they also encompass:

- Patients confusing their medications, taking the wrong medication, or consuming too much/too little of the correct medication
- Doctors prescribing the incorrect medication
- Pharmacists filling the prescription with the incorrect medication

We call these types of occurrences Adverse Drug Events (ADEs). These ADEs result in billions of dollars per year in hospital and insurance costs, and not to mention, the psychological toll on the patient and loved ones.

### 22.2.2 How Computer Vision Can Help

The good news is that computer vision can be used to help patients, doctors, and pharmacists ensure the patient is taking their correct medication, essentially creating a checks and balances system (Figure 22.1).

For example, let's consider the following scenario when a patient visits a doctor. The patient is sick, and after diagnosing them, the doctor writes a prescription for a particular medication.

The patient takes the prescription slip and heads to their local pharmacy. The pharmacy takes the prescription slip, finds the container of pills, and then lays them out for counting.

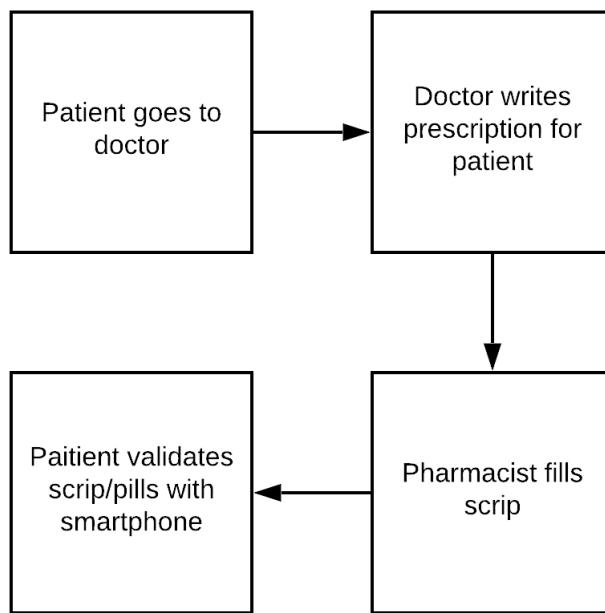


Figure 22.1: We can use computer vision to create a checks and balances system for patients. First a patient visits their doctor. After examination, the doctor writes a prescription. The patient then takes their prescription slip to the pharmacist who takes the time to ensure the prescription is correct. Finally, back at home, the patient can use a prescription pill identification app to verify that they are indeed taking the correct medication.

At this time, a camera facing the counter, along with computer vision algorithms, could be used to *automatically* recognize the pills and validate that the pharmacist is filling the prescription with the correct medication.

After the patient receives their medication from the pharmacist, they head home. Upon arriving at their house, they open the prescription, take out a pill, and use a smartphone app to snap a photo of the pill. The computer vision algorithms powering the app would then process the image and automatically recognize the pill, thereby validating they are taking not only the correct medication, but also the correct amount of pills.

While no system will ever cease all ADEs (we are human, and it's in our nature to make mistakes), such a computer vision system could *dramatically* reduce the number of injuries and deaths.

### 22.2.3 Why Not OCR?

Whenever I discuss using computer vision to recognize prescription pills, one of the first questions I get asked is “*Why can't you apply Optical Character Recognition (OCR)?*”



Figure 22.2: Simple OCR cannot be used for reliable prescription pill identification. **Left:** This pill will be extremely challenging to recognize as there are no characters or markings on the pill, other than a line used to help patients split/cut their medication. **Middle:** Some prescription drug manufacturers embellish their logos. Here the "R" is embellished, making it challenging for an OCR engine to recognize it. **Right:** Imprints in general are extremely challenging (and usually impossible) to accurately OCR.

It's a fair question — and it's one that can only be fully appreciated once you've both (1) researched pill identification and (2) started playing with pill recognition datasets.

The short answer is that it's *impossible* to use OCR for all pill identifications. The longer answer is a bit more complicated.

To start, keep in mind that not all pills have markings on them (Figure 22.2, *left*). Over-the-counter drugs, in particular vitamins, rarely have identifying markings on them. If a pill doesn't have any characters, you can't OCR it.

Secondly, some companies may choose to *embellish* or *stylize* their logo on the pill (Figure 22.2, *middle*). While a human may be able to perceive and understand the stylized logo, it's highly unlikely that the OCR algorithm will be able to do so.

Finally, most OCR systems perform best when the text to be OCR'd is *already* segmented from the input image. When working with prescription pill images, this assumption can very rarely be made, *especially* when working in uncontrolled environments or varying lighting conditions.

The very nature of how characters are imprinted on pills makes it extremely challenging to obtain a nice, clean segmentation of the text (Figure 22.2, *right*). Even if such a segmentation is obtained, there's no guarantee the OCR algorithm will still recognize the characters.

Given the reasons above, we instead need to rely on feature extraction algorithms to quantify the contents of a pill in an image.

## 22.3 Our Pill Recognition System

Before we get started reviewing the actual algorithms we'll use to build our prescription pill recognition system, let's first start by reviewing how our system will actually work.

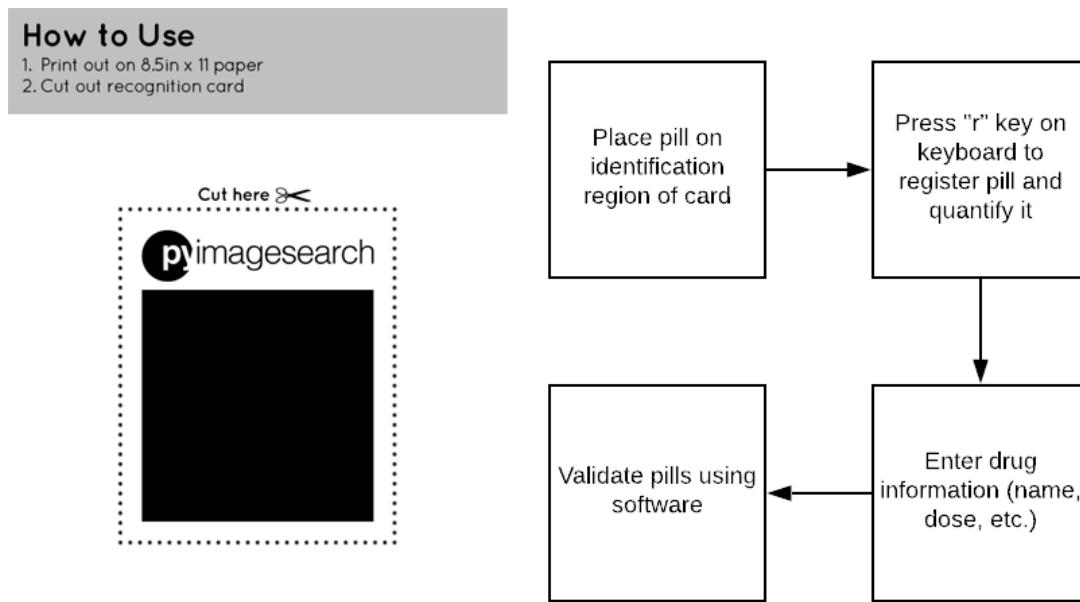


Figure 22.3: **Left:** In order to easily quantify our pills we'll be using a "recognition card". The recognition card will enable us to easily segment the pill from the image and quantify it, including estimating the pill's size. **Right:** The steps involved using the recognition card.

Our pill identification software will have two phases:

- i. Registration
- ii. Recognition/validation

During the **registration phase**, users of the system will place their pills on the black square of an ID card, like in Figure 22.3 (*left*).

We use the ID card for two reasons:

- i. **Known size:** The black square on the ID card has a known size (i.e.,  $100mm \times 100mm$ ), ensuring that we can compute the size of pills once they are placed on the square.
- ii. **Segmentation:** There are very, *very* few pills that are black; and therefore, using a black background will make it easier for us to segment the foreground (the pill) from the background (measurement square).

Once the pills are placed on the black square, the user will press the `r` key (for "register") on their keyboard, and type in any pertinent information, including the name, dose, etc. of the medication (Figure 22.3, *right*). This process is repeated for each pill on the ID card. If the user is a pharmacist they'll want to build a database of all pills they regularly use when filling prescriptions (i.e., registration).

Then, when filling a prescription (i.e., **recognition/validation**) the pharmacist can use the pill ID software to validate that they are indeed distributing the correct medication.

If the user is a patient, they'll want to build a database of all the pills they take (i.e., registration), that way, when they need to take their medication at a later date, they can validate that the pills are correct and that there are no mixups (i.e., recognition/validation).

Such a pill recognition system can be used to drastically reduce the number of ADEs that happen each year.

## 22.4 Characterizing Pills with Features

To characterize and quantify a pill in an image, we'll be using **feature extraction** computer vision algorithms. These algorithms will accept an input image and then produce a feature vector (i.e., a list of numbers) that quantifies a particular component of the pill.

We'll be focusing on four primary components:

- i. Color
- ii. Shape
- iii. Texture
- iv. Size

We'll use these four components as the building blocks for our prescription pill recognition system.

### 22.4.1 Color Histograms

A color histogram counts the number of times a given pixel intensity (or range of pixel intensities) occurs in an image. Using a color histograms we can express the actual *distribution* or "amount" of each color in an image. The counts for each color/color range are then used as a *feature vector* used to represent the color distribution of the image.

As image descriptors, color histograms tend to be quite useful, despite their simplistic nature. For example, let's suppose we want to utilize a 3D color histogram with 8 bins per channel. We could then represent **any image** of **any size** using only  $8 \times 8 \times 8 = 512$  bins, or a feature vector of 512-d.

The size of the input image has *no effect* on our output color histogram (although it's wise to resize large images to smaller dimensions to increase the speed of color histogram computation).

To compare color histograms we can compute the distance between them using a distance function or metric such as the Euclidean distance, cosine similarity, or  $\chi^2$  distance.

The *smaller* the distance between two histograms, the *more similar* the color distributions are. Conversely, the *larger* the distance between two histograms, the *less similar* the color distributions are.

We'll be using color histograms in this chapter to quantify the color of our input pills. You'll find the implementation of the color histogram extraction in Section 22.4.1 along with the histogram distance computation in Section 22.5.7.

If you would like to learn more about color histograms, you should refer to *Practical Python and OpenCV* (<http://pyimg.co/ppao>) [11] and the PyImageSearch Gurus course (<http://pyimg.co/gurus>) [12].

## 22.4.2 Shape

In computer vision and image processing, image moments are often used to characterize the **shape** of an object in an image. These moments capture basic information such as the **area** of the object, the **centroid** (i.e., the center  $(x, y)$ -coordinates of the object), the **orientation**, and other desirable properties.

In Ming-Kuei Hu's 1962 paper *Visual Pattern Recognition for Moment Invariants*, [80], Hu proposed seven moments (i.e., a  $7 - d$  feature vector) that can be used to characterize the shape of an object in an image.

Hu further demonstrated that these moments are (theoretically) invariant to changes in rotation, translation, scale, and reflection (i.e., mirroring).

In practice, the shape to be described can either be a segmented binary image or the boundary of the object (the "outline" of "contour" of the shape).

### 22.4.2.1 How are Hu Moments Computed?

From a strictly statistical point of view, "moments" are just statistical expectations of a random variable. In fact, you're probably *already* familiar with at least one moment, whether you realize it or not.

The most common moment is the first moment – the mean. You're also likely familiar with the second moment – the variance.

Taking the square-root of the variance leaves us with the standard deviation, which you're probably also familiar with. Skew and kurtosis round out the third and fourth moments, respectively.



Figure 22.4: **Top-left:** Input image containing an example pill. **Top-right:** Converting the input image to grayscale. **Bottom-left:** Applying thresholding to binarize the pill. **Bottom-right:** Computing Hu moments for the pill mask (only the centroid is visualized via the red circle).

Let's take a second to consider Figure 22.4 and an example input pill (*top-left*). That same pill is converted to grayscale in the *top-right*. We then apply threshold to obtain a binary image representing the mask of the pill segmented from the input image (*bottom-left*).

The regular moment of a shape in a binary image (in this case, a pill), is defined by:

$$M_{ij} = \sum_x \sum_y x^i y^j I(x, y) \quad (22.1)$$

Where  $I(x, y)$  is the pixel intensity value at the  $(x, y)$ -coordinate.

In order to obtain the translation variance, we need to take our shape measurements relative to the centroid of the shape. The centroid is simply the center  $(x, y)$ -coordinate of the shape, which we define as  $\bar{x}$  and  $\bar{y}$ —, respectively:

$$\bar{x} = M_{10}/M_{00}$$

$$\bar{y} = M_{01}/M_{00}$$

As Figure 22.4 (*bottom-right*) shows, we have marked the center  $(x, y)$ -coordinates (i.e., the centroid) with a green circle.

Now that we have our centroids, we can compute *relative moments*, which are centered about the centroid:

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q I(x, y) \quad (22.2)$$

However, these relative moments do not have much discriminative power to represent shapes, nor do they possess any invariant properties — which is *exactly* where the work of Hu comes in. Hu took these relative moments and constructed seven separate moments which are suitable for shape discrimination (*refer to the Hu paper [80] and the PyImageSearch Gurus course [81]* for more details on these equations).

We'll be using Hu moments in this chapter to quantify and represent the shape of our pills. Pills come in many shapes in sizes, including round/circular, capsules, diamond, and more. Using Hu moments we'll be able to quantify pill shape characteristics into a  $7 - d$  feature vector which we can then use to find pills with similar shapes.

#### 22.4.3 Texture

Local Binary Patterns, or LBPs for short, are a texture descriptor first introduced by Ojala et al. in their 2002 paper *Multiresolution Gray-Scale and Rotation Invariant Texture Classification with Local Binary Patterns* [82]. LBPs work by computing a *local representation* of texture by comparing each pixel in an input (grayscale) image with its surrounding neighborhood.

The first step in constructing the LBP texture descriptor is to convert the image to grayscale. For each pixel in the grayscale image, we select a neighborhood of size  $r$  surrounding the center pixel. An LBP value is then calculated for this center pixel and stored in an output 2D array, with the same width and height as our input image.

For example, let's take an example LBP descriptor, which operates on a fixed  $3 \times 3$  neighborhood of pixels (Figure 22.5, *top-left*). In this example we take the center pixel and threshold it against its neighborhood of 8 pixels. If the intensity of the center pixel is greater-than-or-equal-to its neighbor, then we set the value to 1; otherwise, we set it to 0. With 8 surrounding pixels, we have a total of  $2^8 = 256$  possible combinations of LBPs codes.

From there, we need to calculate the LBP value for the center pixel. We can start from any neighboring pixel and work our way clockwise or counter-clockwise, but our ordering must be *consistent* for all pixels in our input image *and* all images in our dataset.

Given a  $3 \times 3$  neighborhood, **we thus have 8 neighbors that we must perform a binary test on**. The results of this binary test are stored in an 8-bit binary array which, when converted to decimal, looks like Figure 22.5 (*top-right*).

In this example we start at the top-right point and work our way **counter-clockwise**, accumulating the binary strings as we go. We then convert this binary step to decimal, yielding a value of 23. This value is stored in the output LBP 2D array which, when visualized, can be

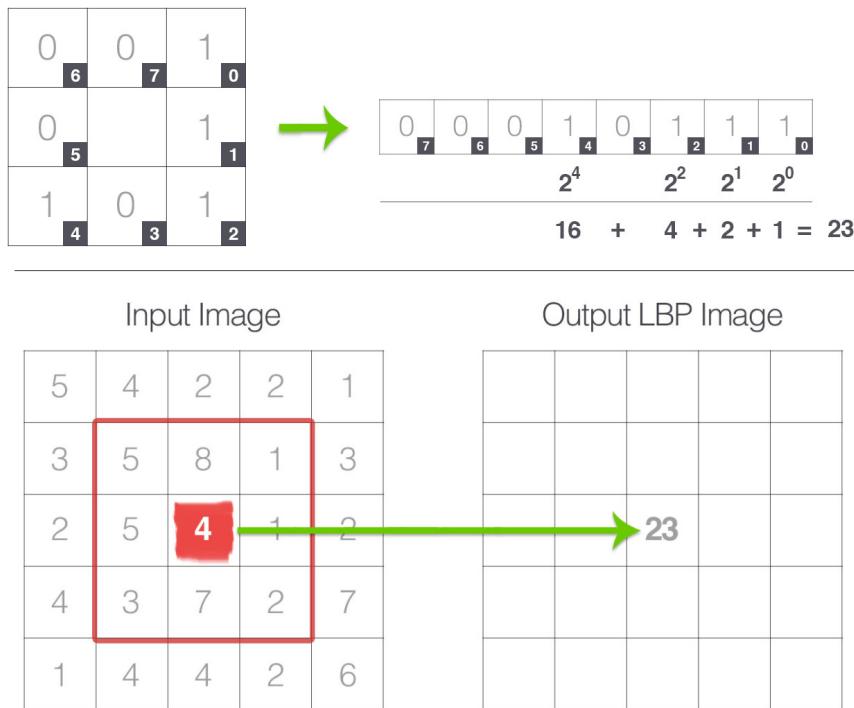
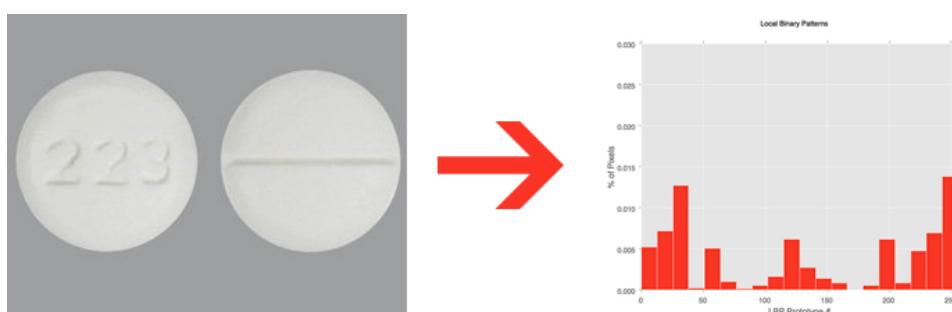


Figure 22.5: **Top:** LBPs work by taking a local neighborhood of pixels and thresholding it according to the center pixel. The resulting binary values are then converted to an integer used to quantify the texture of a region. **Bottom:** After computing the LBP for a local region, we store it in an output matrix with the same width and height as the input image.

seen in Figure 22.5 (bottom). We then repeat these steps for all pixels in the input image.

The final step is to compute a histogram over the output LBP array. Since a  $3 \times 3$  neighborhood has  $2^8 = 256$  possible binary patterns, our LBP 2D array thus has a minimum value of 0 and a maximum value of 255, allowing us to construct a 256-bin histogram of LBP codes as our feature vector (Figure 22.6).



handle such a scenario, Ojala et al. proposed an extension to the original LBP implementation to handle variable neighborhood sizes.

To account for these variable neighborhood sizes, two parameters were introduced:

- i. The number of points,  $p$ , in a circularly symmetric neighborhood to consider (thus removing relying on a square neighborhood)
- ii. The radius of the circle,  $r$ , which allows us to account for different scales

Using this approach, the LBP descriptor now has (theoretically) no limitations on the size of the neighborhood,  $r$  or the number of points,  $p$ , but typically values for these parameters are  $r = 8$  and  $p = 24$ . The final dimensionality of our LBP histogram will be  $p + 2$  (i.e., 26-d).

A further detailed discussion on LBPs, including the concept of uniform patterns, quantifying them, and tabulating them in a histogram, is outside the scope of this book — if you’re interested in learning more about LBPs, be sure to refer to the PyImageSearch Gurus course [12] which includes a detailed lesson on them.

In this chapter, we’ll be using LBPs to characterize the texture of our input pills — some pills are smooth with a dissolvable plastic casing while others are rough and compacted. LBPs will be able to quantify these types of textures into a feature vector which we can then use to compare pills.

#### 22.4.4 Size

Size is arguably *the most important* component of pill identification. As I mentioned before, In the United States, over 50% of the pills on the market are round and/or white. Color and shape are therefore useful for reducing the search space for non-round or non-white pills, but if a pill is round and/or white, we need additional measurements — and that’s where size comes in.

Using size, we can *immediately* reduce our search space of potential pills. Luckily for us, size can be fairly easy to compute/estimate!

To measure the size of a pill in an image, we’ll be extending our size estimation implementation from Chapter 21 — and instead of using a United States quarter as our reference object, we’ll be using a printout of an “pill ID card” (Figure 22.3, *left*).

This ID card has two important properties:

- i. The black square region, where the pill will be placed, is known to be  $100mm \times 100mm$  (I designed it in Photoshop to have these dimensions). This region will serve as our *reference object*. We’ll be able to locate it via edge detection and contour extraction. And since we know the dimensions, we can use it when using the pixels per metric ratio.

- ii. Secondly, the ID region is black. Since very, *very* few pills are black, this region will make it easy for us to segment the foreground (the pill) from the background (the ID card).

Thus, to measure the size of a pill in an image, we must:

- i. Localize the pill ID card
- ii. Compute the dimensions of the ID card (in pixels)
- iii. Utilize the `pixels_per_metric` equation from the previous chapter to “calibrate” our system.
- iv. Find all pills on the ID card
- v. Compute the ratio of the size of the pill ID card to the pill, enabling us to obtain the estimated size of the pill

As a refresher, the pixels per metric equation is the following:

$$\text{pixels\_per\_metric} = \text{object\_dim\_pixels}/\text{object\_dim\_mm} \quad (22.3)$$

The `object_dim_pixels` could be either the width or the height *in pixels* of the object we want to measure. The `object_dim_mm` is the *known dimension* (i.e., either width or height) of the object in *millimeters* (or in some other measurable unit, such as inches).

Since we know our pill ID card is *100mm x 100mm*, the `object_dim_mm` will be a constant value of *100* for this particular application. Once we segment the pill ID card from the image, we’ll know the `object_dim_pixels`.

We’ll modify this equation slightly to handle measuring the size of a pill via its length by computing the following ratio:

$$\text{pill\_size\_mm} = (\text{object\_dim\_mm}/\text{object\_dim\_pixels}) \times \text{pill\_length\_pixels} \quad (22.4)$$

Here we take the known dimensions of the pill ID card (in millimeters) and divide it by its measured length in pixels. The *ratio* of this value to the measured length of the pill (in pixels) give us the estimated size of the pill in millimeters.

This method will become more clear in Section 22.5.3 where we implement the algorithm used to both segment and measure the sizes of the pills on the ID card.

## 22.5 Prescription Pill Recognition with Computer Vision

So far in this chapter, we have reviewed the components of our prescription pill recognition system, but only at a high level. In this section, we'll get our hands dirty and look at the actual source code used to build the system.

### 22.5.1 Our Project Structure

Before we dive into any code, let's first review the directory structure for our project:

---

```
|-- config
|   |-- config.json
|-- pyimagesearch
|   |-- __init__.py
|   |-- pill_describer.py
|   |-- pill_finder.py
|   |-- pillsearcher.py
|   |-- utils
|       |-- __init__.py
|       |-- conf.py
|-- pill_id.py
```

---

Inside the `config` directory we will store a file named `config.json`. This JSON file includes any relevant configurations we need, including the size of our pill measurement area, path to our pill database, etc.

The `pyimagesearch` module includes three Python files:

- `pill_describer.py`: Implements all necessary logic to quantify the color, shape, and texture of a pill in an image
- `pill_finder.py`: Finds the pill ID card and then segments the pill from the background
- `pillsearcher.py`: Contains our implementation of the `PillSearcher` class, used to search our database of known pills based on an input query pill

Finally, `pill_id.py` combines all the pieces, enabling us to:

- i. Add/register pills in our database
- ii. Recognize pills

### 22.5.2 Our Configuration File

Let's get started with our configuration file. Open up the `config.json` file in your project directory structure and insert the following code:

---

```

1  {
2      // define the size of the pill measurement area (in millimeters)
3      "measurement_area_size": 100,
4
5      // define the path to our pill database
6      "db_path": "db.pickle",

```

---

On **Line 3** we define the size of the pill measurement area (in millimeters) on the pill ID card. As I mentioned in Section 22.4.4, the size of the pill measurement is 100mm.

I know the size because I designed the pill ID card myself and ensured that the measurement area is indeed 100mm. The ID card is included in your download of the text — be sure to print it out to follow along with the rest of this chapter.

**Line 6** defines the path to our database file. Our “database” is simply going to be a Python dictionary with a unique integer ID as the key and the name of the pill itself as the value.

Next, we have a few configurations related to how we'll handle weighting the various pill components when performing a pill recognition:

---

```

8      // when identifying pills we can weight the size, shape, color, and
9      // texture components differently, ultimately leading to a more
10     // accurate pill identification
11     "weights": {
12         "size": 0.5,
13         "shape": 0.3,
14         "color": 0.1,
15         "texture": 0.1
16     }
17 }
```

---

As we know from Section 22.4, we're quantifying pills via their size, shape, color, and texture. Some components are more important than others when performing a pill identification, namely size and shape, as these components allow us to dramatically reduce the search space of the pills. **Lines 11-16** define a `weights` dictionary that will enable us to *weight* the components as such.

### 22.5.3 Finding Pills in Images

Before we can actually *quantify* a pill, we first need to:

- i. Find the Pill ID card in the image
- ii. Search the Pill ID card for the measurement area
- iii. And finally, find all pills on the measurement area

Once we have both the measurement area along with all the pills on the measurement area, we can compute the respective pill sizes.

All of these tasks will be accomplished inside `pill_finder.py` in the `pyimagesearch` module:

---

```
1 # import the necessary packages
2 from imutils.perspective import four_point_transform
3 from skimage.segmentation import clear_border
4 from sklearn.metrics import pairwise
5 from scipy.spatial import distance as dist
6 from collections import namedtuple
7 import numpy as np
8 import imutils
9 import cv2
10
11 # define the detected pill object
12 DetectedPill = namedtuple("DetectedPill",
13     ["contour", "pill", "mask", "size"])
```

---

**Lines 2-9** handle importing our required Python packages. The `four_point_transform` function will enable us to obtain a top-down, birds-eye-view of the pill ID card, similar to when we built a document scanner in the following PyImageSearch blog post (<http://pyimg.co/djkn3>) [83].

The `clear_border` function inside of scikit-image allows us to remove any foreground pixels touching the borders of an image, making the assumption that any foreground pixels touching the borders are noise.

Using both the `pairwise` and `distance` modules (**Lines 5 and 6**) we'll be able to compute the size of the pill.

Finally, **Lines 11-12** define a `namedtuple` called `DetectedPill` which allows us to conveniently store all relevant information on the detected pill, including:

- The contour of the detected pill (so we can draw/visualize it later)

- The pill ROI itself
- The mask associated with the pill ROI
- The estimated size of the pill

Now that our imports are handled, let's move on to defining the `find_measurement_area` function which, as the name suggests, will find the pill measurement area on the ID card:

---

```
15 def find_measurement_area(image, keep=5):
16     # convert the image to grayscale and then perform edge detection
17     gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
18     edged = cv2.Canny(gray, 50, 128)
```

---

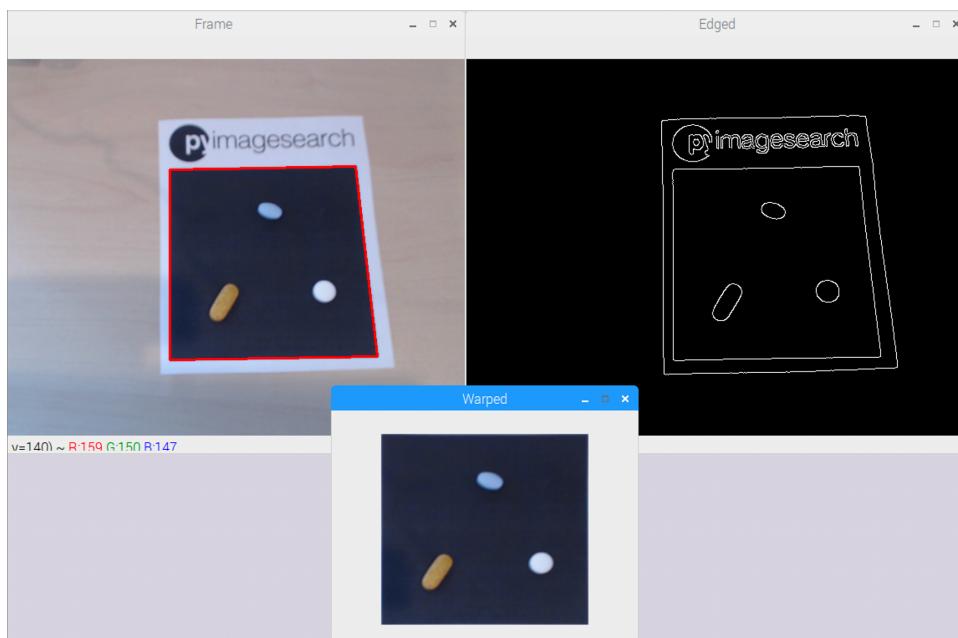


Figure 22.7: **Top-left:** Input frame from webcam connected to the RPi. **Top-right:** Apply edge detection to reveal the outlines of the pill recognition card. **Bottom:** After using contours to find the pill ID region, we apply a perspective transform to obtain a top-down, birds-eye-view of the card.

Our function accepts a single required parameter – `image`, which is the input image to find the pill measurement on, along with `keep` – an integer used to indicate how many potential contours to explore when searching for the measurement area.

**Line 17** converts the input `image` (Figure 22.7, *top-left*) to grayscale while **Line 18** takes the `gray` image and performs edge detection. The purpose of edge detection is to find the outlines of both the pill ID card along with the measurement area.

The output of applying edge detection can be seen in Figure 22.7 (*top-right*). Notice how the outlines of *both* the ID card and measurement area are clearly defined with *no* discontinuities (i.e., the outline is not “broken” at any point).

Now that we have our edge map, we need to find the contours of the edges:

---

```

20      # find contours in the edge map, keeping only the five largest
21      # ones
22      cnts = cv2.findContours(edged.copy(), cv2.RETR_LIST,
23          cv2.CHAIN_APPROX_SIMPLE)
24      cnts = imutils.grab_contours(cnts)
25      cnts = sorted(cnts, key=cv2.contourArea, reverse=True) [:keep]
26
27      # initialize the list of candidate regions that *could* be the
28      # pill measurement region along with (1) the contour corresponding
29      # to the measurement region and (2) the warped ROI of the
30      # measurement region
31      regions = []
32      areaCnt = None
33      warped = None

```

---

**Lines 22-24** find contours in the edge image while **Line 25** sorts the contours by their area (largest to smallest), keeping only the biggest ones.

**Lines 31-33** perform initializations, including:

- **regions**: A list of candidate regions that *could* be the pill measurement area
- **areaCnt**: The contour belonging to the measurement area
- **warped**: The output of the perspective transform, giving us a top-down view of the ID card

The next step is to loop over the contours and attempt to find the measurement area:

---

```

35      # loop over the contours
36      for c in cnts:
37          # approximate the contour
38          peri = cv2.arcLength(c, True)
39          approx = cv2.approxPolyDP(c, 0.02 * peri, True)
40
41          # check to see if our approximated contour has four points,
42          # implying that we have potentially found the pill measurement
43          # region
44          if len(approx) == 4:
45              # compute the bounding box of the contour region and then
46              # compute the average grayscale pixel intensity of the ROI

```

---

---

```

47     (x, y, w, h) = cv2.boundingRect(c)
48     roi = gray[y:y + h, x:x + w]
49     avg = cv2.mean(roi)[0]
50
51     # add the mean pixel intensity and contour approximation
52     # to the candidate regions list
53     regions.append((avg, approx))

```

---

**Line 36** starts a `for` loop over our contours list (keeping in mind that we are only processing the largest ones).

For each contour we perform contour approximation (**Lines 38 and 39**), which simplifies and reduces the number of points in the contour, thereby allowing us to check and see if we have found the pill measurement area (**Line 44**). If the approximated contour has four vertices, then we know we've found a rectangle.

**Line 47** computes the bounding box rectangle of the contour and then **Line 48** extracts the `roi` from the `gray` image. Once we have the `roi`, **Lines 49-53** computes the mean of the `roi` and adds it to the `region` list.

Why bother computing the mean of the candidate ROI? Keep in mind that the pill measurement area is *black*. If we are able to find a rectangular area with a very small average pixel intensity (implying the region is very dark), then we can be fairly certain we've found the pill measurement area.

The next code block handles checking to see if we found such an area:

---

```

55     # ensure at least one candidate region was found
56     if len(regions) > 0:
57         # sort the regions according to the average grayscale pixel
58         # intensity, maintaining the assumption that the *darkest
59         # region* (due to the black square) will be our pill
60         # measurement area
61         areaCnt = sorted(regions, key=lambda x: x[0][0][1])
62
63         # apply the four point transform to obtain a top-down
64         # view of the input image
65         warped = four_point_transform(image, areaCnt.reshape(4, 2))
66
67         # return a tuple of the measurement area contour and the warped,
68         # top-down view of the pill region
69     return (areaCnt, warped)

```

---

**Line 56** makes a check to ensure at least one candidate area exists in the `regions` list. Provided one does, we sort the `regions` list according to their average grayscale pixel intensity, maintaining the assumption that the darkest area will be our pill measurement area.

**Line 65** then performs a perspective transform on the ROI, giving us a top-down, birds-eye view of the measurement area. A visualization of the transform can be seen in Figure 22.7 (bottom).

**Remark.** If you'd like more information on perspective transforms and how the output image is obtained, please refer to the following three tutorials on the PyImageSearch blog:

- i. 4 Point OpenCV getPerspective Transform Example (<http://pyimg.co/7t1dj>) [84]
- ii. How to Build a Mobile Document Scanner in Just 5 Minutes (<http://pyimg.co/djkn3>) [83]
- iii. Bubble sheet multiple choice scanner and test grader using OMR, Python and OpenCV (<http://pyimg.co/l0r5v>) [85]

Finally, **Line 69** returns 2-tuple of the area contour and warped image to the calling function. It's worth noting if the measurement area could *not* be found, then both of the values in the tuple will be `None`.

Now that we've obtained the pill measurement area, the next step is to find all pills in that area, which is exactly what `find_pills` will do:

---

```
71 def find_pills(image, height, keep=10, minArea=250):
72     # blur the image slightly to help reduce high frequency noise and
73     # then allocate memory for the output mask
74     blurred = cv2.GaussianBlur(image, (5, 5), 0)
75     mask = np.zeros(image.shape[:2], dtype="uint8")
```

---

The `find_pills` function requires two arguments along with two optional ones.

First, we must supply the input `image` (assumed to be the pill measurement ROI after a perspective transform) along with the `height`, which is the size of the pill measurement area (in millimeters) from our configuration file.

Optionally, we can also supply `keep` (the number of contours to explore when looking for pills) and `minArea` (the minimum area of a pill contour used to filter out false-positives).

**Line 74** performs a Gaussian blur to help reduce noise in the image, thereby making it easier to find the pills themselves.

**Line 75** then allocates memory for a `mask` which we'll build in the following code block:

---

```
77     # loop over the channels of the image
78     for chan in cv2.split(blurred):
79         # automatically threshold the channel using Otsu thresholding,
80         # then take the bitwise OR with the mask
```

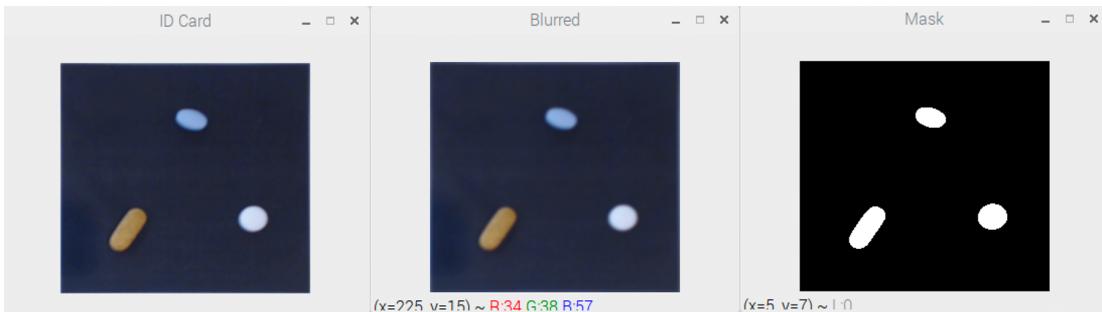


Figure 22.8: **Left:** Input pill recognition area after a top-down perspective transform. **Middle:** Blurring the region slightly, allowing us to focus on the structural outlines of the pills. **Right:** Output of applying channel-wise thresholding and combining the result to form the final pill masks.

---

```

81     thresh = cv2.threshold(chan, 0, 255,
82         cv2.THRESH_BINARY | cv2.THRESH_OTSU) [1]
83     mask = cv2.bitwise_or(mask, thresh)

```

---

On **Line 78** we loop over each of the RGB channels in the input `image`.

For each channel, we threshold it using Otsu's thresholding method (**Lines 81 and 82**). Given the output `thresh` image, we take the bitwise OR of the `thresh` and `mask`.

The reason we take bitwise OR is because we want to accumulate a mask based on the output of the threshold operation on each channel. Pills can be a variety of different colors and using a single channel or grayscale representation alone is not sufficient to segment them from a measurement area background — a neat trick we can perform is to thus apply thresholding to *each* channel and then accumulate the result (Figure 22.8).

Given our `mask` we can now preprocess it and find contours:

---

```

85     # remove any connected-components that are attached to the border
86     # of the mask
87     mask = clear_border(mask)
88
89     # find contours in the mask, keeping only the largest ones
90     cnts = cv2.findContours(mask.copy(), cv2.RETR_LIST,
91         cv2.CHAIN_APPROX_SIMPLE)
92     cnts = imutils.grab_contours(cnts)
93     cnts = sorted(cnts, key=cv2.contourArea, reverse=True)[:keep]
94
95     # initialize the list of detected pills
96     detectedPills = []

```

---

On **Line 87** we use the `clear_border` function to remove any pixels that are touching the borders of the image (we assume these foreground pixels are noise).

From there, **Lines 89-93** find contours in the `mask`, sort them by size (largest to smallest), and keep on the largest ones. **Line 96** initializes a list to keep track of the detected pills.

Let's loop over the contours now and attempt to detect each pill:

---

```

98     # loop over the contours
99     for c in cnts:
100         # ensure the contour area is sufficiently large
101         if cv2.contourArea(c) > minArea:
102             # compute the bounding box of the pill and then use the
103             # bounding box coordinates to extract both the pill and
104             # its mask
105             (x, y, w, h) = cv2.boundingRect(c)
106             pill = image[y:y + h, x:x + w]
107             pillMask = mask[y:y + h, x:x + w]
```

---

We start by looping over all contours on **Line 99**.

**Line 101** checks to ensure the area of the contour is sufficiently large (used to reduce false-positive pill detections due to noise).

Provided the contour *is* large enough, we continue to process it by computing the bounding box, and extracting the ROI of the `pill` and `pillMask` (respectively).

The next step is to compute the estimated *size* of the pill:

---

```

109         # extract the (x, y)-coordinates from the contour and
110         # then compute the pairwise euclidean distances between
111         # the set of coordinates
112         pts = [list(p[0]) for p in c]
113         D = pairwise.pairwise_distances(pts, metric="euclidean")
114
115         # find the indexes of the coordinates where the distance
116         # is the largest
117         maxDists = np.where(D == D.max())[0]
118         (mI, mJ) = (maxDists[0], maxDists[1])
119
120         # if the event there are *multiple* coordinates with the
121         # same distance, we'll just take the first two points out
122         # of convenience
123         if maxDists.shape[0] > 2:
124             mJ = maxDists[int(len(maxDists) / 2)]
125
126         # if the height of the image is greater than the width,
127         # then set the divisor equal to the width, otherwise set
128         # it equal to the height
129         div = image.shape[1] if image.shape[0] > image.shape[1] \
130             else image.shape[0]
```

---

```

132     # compute the distance (in pixels) between the two points
133     # and then convert the distance to millimeters to estimate
134     # the length of the pill
135     d = dist.euclidean(pts[mI], pts[mJ])
136     mm = (height / div) * d
137
138     # create the detected pill object and update the list of
139     # detected pills
140     dp = DetectedPill(c, pill, pillMask, mm)
141     detectedPills.append(dp)
142
143     # return the list of pills
144     return detectedPills

```

---

**Line 112** extracts list of all  $(x, y)$ -coordinates belonging to the pill contour while **Line 113** computes the pairwise Euclidean distances between *all* points. If our contour list has  $N$  coordinates, then the output matrix after computing the Euclidean distances will be  $N \times N$ .

Given our pairwise distances, **Line 117 and 118** finds the indexes where the distances are largest, indicating that these points will be the two points *farthest* from each other on the pill.

However, it could be the case that there are *multiple* points with the same distance. In that case, we'll take the first points out of convenience (**Lines 123 and 124**). Since the distances are the same, this convenience operation will *not* affect the estimated pill size computation.

**Line 129 and 130** compute the divisor (`div`) to be used when estimating the pill size. Following Equation 22.4 from Section 22.4.4, the `div` is the size of the pill measurement area in *pixels*. We'll use whatever is the smaller of the width and height of pill measurement area as our divisor.

**Lines 135 and 136** round out our pill size estimation procedure. We first compute the Euclidean distance between the two largest points along the pill contour. Once we have this distance, we can fill in all variables to Equation 22.4 and compute the pill size.

The final step is to construct a `DetectedPill` object using the contour of the detected pill, the ROI of the pill, the mask of the pill, and finally its size (**Lines 140 and 141**).

All `detectedPills` are then returned to the calling function.

#### 22.5.4 Quantifying Pill Color

In the previous section, we learned how to find all pills on our measurement card. The next step is to take each of these pills and *quantify* them.

All functions used to quantify pills will be stored in the `pill_describer.py` file inside the `pyimagesearch` module. Let's start by defining the `describe_color` function which will be

used to quantify the color distribution of the pill:

---

```

1 # import the necessary packages
2 from skimage.feature import local_binary_pattern
3 import numpy as np
4 import cv2
5
6 def describe_color(pill, mask, bins=(4, 4, 4)):
7     # convert the pill image to the HSV color space, then extract
8     # a 3D color histogram from the masked region of the pill
9     hsv = cv2.cvtColor(pill, cv2.COLOR_BGR2HSV)
10    hist = cv2.calcHist([hsv], [0, 1, 2], mask, bins,
11                      [0, 180, 0, 256, 0, 256])
12
13    # normalize the histogram and flatten it
14    hist = cv2.normalize(hist, hist).flatten()
15
16    # return the histogram
17    return hist

```

---

The `describe_color` function requires two arguments:

- `pill`: The pill ROI we are quantifying
- `mask`: The corresponding mask associated with the pill ROI

We can optionally supply the number of bins associated with our histogram. By default we are computing a 3D histogram with 4 bins per channel, resulting in a  $4 \times 4 \times 4 = 64 - d$  output feature vector.

**Line 9** starts by converting our input `pill` ROI to the HSV color space. We are using the HSV color space rather than RGB here as HSV tends to be a better color space for comparing color distributions.

**Line 10** then takes the `hsv` image and computes a color histogram for it using our supplied number of bins. You should also take note that we are supplying the `mask` to the `cv2.calcHist` function, implying that **only pixels belonging to the mask will be considered when computing the histogram**.

**Line 14** normalizes our histogram to use relative frequency, ensuring that two pills with the same color, even if they have different spatial dimensions will have the *same* histogram.

Finally, **Line 17** returns the resulting histogram to the calling function.

### 22.5.5 Quantifying Pill Shape

Our next function will quantify the shape of a pill. As discussed in Section 22.4.2, we'll be using Hu moments to represent shape.

Let's take a look at the `describe_shape` function now:

---

```

19 def describe_shape(mask):
20     # compute the Hu moments shape features
21     moments = cv2.HuMoments(cv2.moments(mask)).flatten()
22
23     # log transform the moments to make them comparable
24     sign = np.copysign(1.0, moments)
25     log = np.log10(np.absolute(moments))
26     moments = -1 * sign * log
27
28     # return the log transformed Hu moments
29     return moments

```

---

The `describe_shape` function requires only a single argument, the `mask` of the pill. Given this mask, we compute our Hu moments on **Line 21**.

However, we're not quite done yet. In order to make the moments comparable, we first must log transform them. The log transform works by:

- Grabbing the sign (positive or negative) of each of the 7 values in `moments` (**Line 24**)
- Taking the base-10 log of the absolute value of the `moments` (**Line 25**)
- And finally, taking the inverse of the sign, multiplied by the `log` transformed `moments`

The reason we take this log transform is to ensure the moments are comparable in scale.

**Line 29** takes our log transformed `moments` and returns them to the calling function.

### 22.5.6 Quantifying Pill Texture

The final component we'll be quantifying is the *texture* of the pill — to do that, we'll be using the `describe_texture` function:

---

```

31 def describe_texture(pill, mask, numPoints=24, radius=8, eps=1e-7):
32     # convert the pill image to grayscale
33     gray = cv2.cvtColor(pill, cv2.COLOR_BGR2GRAY)
34
35     # compute the Local Binary Pattern representation of the

```

```

36      # grayscale image, then grab the LBP values for *only* the
37      # masked region
38      lbp = local_binary_pattern(gray, numPoints, radius,
39          method="uniform")
40      lbp = lbp[mask > 0]
41
42      # compute a histogram based on the LBP values
43      (hist, _) = np.histogram(lbp, bins=np.arange(0, numPoints + 3),
44          range=(0, numPoints + 2))
45
46      # normalize the histogram
47      hist = hist.astype("float")
48      hist /= (hist.sum() + eps)
49
50      # return the LBP histogram
51      return hist

```

---

The `describe_texture` function requires two arguments:

- `pill`: The pill ROI
- `mask`: The mask associated with the pill ROI

We then supply `numPoints` and `radius`, two parameters to the LBPs feature extractor discussed in Section 22.4.3. Finally, the `eps` value is used to prevent division by zero errors when normalizing the extracted LBP histogram.

**Line 33** converts our pill ROI to grayscale, a requirement prior to extracting LBPs.

Once we have our `gray` image, we can compute our LBPs on **Lines 38 and 39**. **Line 40** uses our `mask` to select *only* LBP values for the pill region (ignoring all other values not associated with the pill).

But we're not quite done yet! Given the `lbps` we still need to compute the histogram.

**Lines 43 and 44** compute a histogram based on the `lbp` values. The total number of possible LBP values is `numPoints + 2`, which we supply to the `range` parameter. However, since the number of `bins` is *non-inclusive*, we need to add an additional value, hence `numPoints + 3` for the number of bins.

**Remark.** For more information on how we compute our histogram based on LBP values, refer to the following PyImageSearch blog post (<http://pyimg.co/94p11>) [86] along with the PyImageSearch Gurus course (<http://pyimg.co/gurus>) [87].

Now that we've computed our histogram, we normalize it on **Lines 47 and 48** by summing the values in the histogram, and then dividing by the sum. This operation is similar to our normalization procedure when computing the color histogram. Additionally, notice how we add

a constant value of `eps` to avoid any division by zero errors (which would happen if all entries in `hist` were zero).

Finally, **Line 51** returns the resulting LBP histogram to the calling function.

### 22.5.7 Creating the Pill Identifier

At this point we have implemented methods to quantify the shape, texture, color, and size of a pill — but the question remains, *how do we perform the actual pill identification?*

We'll be treating pill recognition as a **Content-Based Image Retrieval (CBIR)**, or more simply, **image search**.

When our pill ID system is presented with a new pill to recognize, we'll *independently* search our database for pills with similar shape, texture, color, and size. From there, we'll *combine* the results to find the most similar pill amongst each of the four components.

Let's go ahead and implement the `PillSearcher` class now. Open up the `pillsearcher.py` file in the `pyimagesearch` module and insert the following code:

---

```

1 # import the necessary packages
2 from scipy.spatial import distance as dist
3 import numpy as np
4
5 class PillSearcher:
6     def __init__(self, db):
7         # store the pill database
8         self.db = db

```

---

**Line 6** defines the constructor to the `PillSearcher` class. We need only a single argument here, `db`, which is a Python dictionary of our *registered* pills. The key to the dictionary is a unique ID and the value of the dictionary includes the name of the pill; shape, color, size, and texture features; and any other relevant important.

The `search` method will accept our input query features of the pill to be recognized and then attempt to identify the pill:

---

```

10     def search(self, size, shape, color, texture, weights, eps=1e-10):
11         # initialize the results dictionary
12         results = {"size": {}, "shape": {}, "color": {},
13                    "texture": {}, "combined": {}}

```

---

The `search` method requires a number of parameters:

- `size`: The computed size of the pill to be recognized
- `shape`: The extracted Hu moments of the query pill
- `color`: The color histogram of the pill we wish to ID
- `texture`: The LBPs for the query pill
- `weights`: The values which instruct how much to weight the size, shape, color, and texture features, respectively, when performing a search
- `eps`: A small  $\epsilon$  value used to prevent division by zero errors

**Lines 12 and 13** initialize a `results` dictionary that has a number of keys. For each size, shape, color, and texture component we'll perform an independent search, finding the most relevant pills for each. The `combined` key will then store the *final results* after combining each of the independent searchers.

Speaking of searches, let's go ahead and search for similar pills now:

---

```

15      # loop over the database
16      for pillID in self.db.keys():
17          # compute the distance between the pill sizes
18          sizeDist = dist.euclidean(size, self.db[pillID]["size"])
19          results["size"][pillID] = sizeDist
20
21          # compute the distance between the shape features
22          shapeDist = dist.euclidean(shape,
23              self.db[pillID]["shape"])
24          results["shape"][pillID] = shapeDist
25
26          # compute the distance between the color features
27          colorDist = self.chi2_distance(color,
28              self.db[pillID]["color"])
29          results["color"][pillID] = colorDist
30
31          # compute the distance bewteen the texture features
32          textureDist = self.chi2_distance(texture,
33              self.db[pillID]["texture"])
34          results["texture"][pillID] = textureDist

```

---

On **Line 16** we loop over each of the unique pill IDs in our `db`. For each of the pills we compute the distance between the *input query features* and the *features in our database*.

**Lines 18 and 19** computes the Euclidean distance between the query pill size and the size of the current pill from the `for` loop. The smaller the distance, the more similar the pills are in sizes.

**Lines 22-24** do the same, only for shape. Here we are taking the Euclidean distance between the Hu moments for the query pill and the Hu moments for the pill in the database. Again, the smaller the distance between the Hu moments, the more similar the pills are (in terms of shape).

When working with histograms and distributions, we often use the  $\chi^2$  distance. Since both our color features (color histograms) and texture features (LBP histograms) are histograms, we can use the  $\chi^2$  distance to compare the color and texture components (**Lines 27-34**).

At this point we've looped over our `db` and compared our *query features* to *all features in the database* for each of the size, shape, color, and texture components.

The next step is to actually *normalize them* so we can weight the results. The normalization procedure we'll use is to:

- i. Find the *largest* distance for each component
- ii. Scale all distances for the component to the range [0, 1]

This concept is more easily explained via code:

---

```

36     # loop over each set of search results
37     for t in results.keys():
38         # ignore the combined set of search results since they
39         # are not yet populated
40         if t == "combined":
41             continue
42
43         # find the largest similarity/distance score
44         maxDist = max(results[t].values())
45
46         # loop over all results again, this time scaling the
47         # compute distance to the range [0, 1]
48         for (pillID, d) in results[t].items():
49             results[t][pillID] = d / (maxDist + eps)

```

---

On **Line 37** we loop over each type of search results. Since we haven't yet computed any "combined" results, we'll ignore that component (**Lines 40 and 41**).

Otherwise, we find the largest distance for the current search result (**Line 44**). Given this distance we can loop over each of the distances in the `results[t]` dictionary and scale the distance to the range [0, 1].

Now that our distances are scaled, we can weight them and compute the final "combined" similarity:

---

```

51      # loop over the database keys a second time, this time to
52      # combine the search results
53      for pillID in self.db.keys():
54          # compute the weighted sum of distances
55          d = np.sum([
56              weights["size"] * results["size"][pillID],
57              weights["shape"] * results["shape"][pillID],
58              weights["color"] * results["color"][pillID],
59              weights["texture"] * results["texture"][pillID],
60          ])
61          results["combined"][pillID] = d
62
63      # sort our results, so that the smaller distances (i.e. the
64      # more similar pills) are at the front of the list
65      results["combined"] = sorted([(v, k) for (k, v) in \
66          results["combined"].items()])
67
68      # return the set of combined pill results
69      return results["combined"]

```

---

On **Line 53** we once again loop over each of the unique IDs in our `db`. For each pill in our database, we compute the ***weighted sum*** of components (**Lines 55-60**).

We use a weighted sum here so we can weight each of the components individually. We know from a previous discussion of pill identification in this chapter that `size` and `shape` will be the most discriminative features — thus, from our `config.json` file in Section 22.5.2, we'll be weighting these distances more.

**Line 61** then stores the weighted distances in our `results` dictionary.

The final step is to sort our results so that smaller distances are at the front of the list (**Lines 65 and 66**). The combined results are then returned to the calling function on **Line 69**.

The final function in the `PillSearcher` class is our  $\chi^2$  distance metric:

---

```

71  def chi2_distance(self, histA, histB, eps=1e-10):
72      # compute the chi-squared distance
73      d = 0.5 * np.sum(((histA - histB) ** 2) / (histA + histB + eps))
74
75      # return the chi-squared distance
76      return d

```

---

This function simply computes the chi-squared distance between two histograms.

### 22.5.8 Putting the Pieces Together

I know this has been quite a long chapter, but we are finally ready to put all the pieces together!

Open up the `pill_id.py` file and insert the following code:

---

```

1 # import the necessary packages
2 from imutils.video import VideoStream
3 from pyimagesearch.pillsearcher import PillSearcher
4 from pyimagesearch.utils import Conf
5 from pyimagesearch.pill_finder import find_measurement_area
6 from pyimagesearch.pill_finder import find_pills
7 from pyimagesearch.pill_describer import describe_shape
8 from pyimagesearch.pill_describer import describe_color
9 from pyimagesearch.pill_describer import describe_texture
10 import argparse
11 import imutils
12 import pickle
13 import time
14 import cv2
15 import os
16
17 # construct the argument parser and parse the arguments
18 ap = argparse.ArgumentParser()
19 ap.add_argument("-c", "--conf", required=True,
20     help="Path to the input configuration file")
21 args = vars(ap.parse_args())

```

---

**Lines 2-15** handle our imports. Notice how we'll be using *all* of our custom implementations in this chapter, including how to:

- Find the measurement area
- Localize each of the individual pills on the measurement area
- Quantify each pill in terms of shape, color, texture, and size
- Search our pill database and ID a given pill

We then parse our command line arguments on **Lines 18-21**. We only need a single argument here, `--conf`, the path to our configuration file.

The next step is to load our configuration file and perform a few initializations:

---

```

23 # load the configuration
24 conf = Conf(args["conf"])
25
26 # initialize the pills database
27 print("[INFO] initializing pill database...")
28 nextID = 0
29 db = {}
30

```

---

---

```

31 # if the pill database exists on disk, load it
32 if os.path.exists(conf["db_path"]):
33     print("[INFO] loading pill database...")
34     db = pickle.loads(open(conf["db_path"], "rb").read())
35     nextID = max(db.keys()) + 1
36
37 # initialize the pill searcher
38 ps = PillSearcher(db)
39
40 # start the video stream thread
41 print("[INFO] starting video stream thread...")
42 vs = VideoStream(src=0).start()
43 # vs = VideoStream(usePiCamera=True).start()
44 time.sleep(1.0)

```

---

**Line 24** loads our configuration file from disk. We then initialize the `nextID`, a unique ID (**Line 28**) of the *next* pill that will be added to our database.

The database (`db`) itself is initialized as an empty dictionary on **Line 29**. However, if an existing database exists on disk (**Line 32**), we'll load it (**Line 34**), and find the next unique ID (**Line 35**).

The `PillSearcher` class is then initialized on **Line 38** while we access our `VideoStream` on **Line 42-44**.

Let's start looping over frames from our video stream:

---

```

46 # loop over frames from the video stream
47 while True:
48     # read a frame from the camera sensor
49     frame = vs.read()
50
51     # if the height is greater than the width, then resize the frame
52     # to have a maximum height of 600 pixels
53     if frame.shape[0] > frame.shape[1]:
54         frame = imutils.resize(frame, height=600)
55
56     # otherwise, the width is greater than the height so resize the
57     # frame to have a maximum width of 600 pixels
58     else:
59         frame = imutils.resize(frame, width=600)

```

---

The next `frame` is read on **Line 49**. We then resize the frame such that the largest dimension has a maximum width or height of 600px (**Lines 51-59**).

Now that we have our frame, we can find the pill measurement area:

---

```

61     # find the pill measurement area in the frame

```

---

```

62     area = find_measurement_area(frame)
63     foundArea = False
64
65     # only continue if the pill measurement area was found
66     if area[0] is not None:
67         # draw the contour corresponding to the measurement area
68         (areaCnt, area) = area
69         foundArea = True
70         cv2.drawContours(frame, [areaCnt], -1, (0, 0, 255), 2)
71
72     # show the frame
73     cv2.imshow("Frame", frame)
74     key = cv2.waitKey(1) & 0xFF

```

---

**Line 62** calls our `find_measurement_area` function from Section 22.5.3. We then initialize a boolean, `foundArea`, to indicate if we have successfully found the measurement area.

**Line 66** ensures that the pill measurement area was indeed found, and if so:

- i. Unpacks the `area` 2-tuple to extract the (1) contour of the area and (2) area ROI
- ii. Indicates that we have indeed found the measurement area
- iii. Draws the outline of the measurement area on the `frame`

We then display the `frame` to our screen and register any keypresses.

But before we can *recognize* any pills, we must first *register* the pills in our database. To register a pill we must:

- i. Place one or more pills on the measurement area of the pill ID card
- ii. Press the `r` key on our keyboard to trigger the registration process

Let's implement the registration process now:

```

76     # check to see if we should register a new pill in the database
77     if key == ord("r") and foundArea:
78         # find all pills in the measurement area
79         pills = find_pills(area, conf["measurement_area_size"])
80
81         # loop over each of the detected pills
82         for pill in pills:
83             # quantify the color, shape, and texture of the pill
84             color = describe_color(pill.pill, pill.mask)
85             shape = describe_shape(pill.contour)
86             texture = describe_texture(pill.pill, pill.mask)
87

```

```

88         # grab the pill name from the user
89         name = input("Enter the pill name: ")
90
91         # construct the data for pill
92         data = {
93             "id": nextID,
94             "name": name,
95             "size": pill.size,
96             "color": color,
97             "shape": shape,
98             "texture": texture
99         }
100
101     # update the database with the new pill
102     print("[INFO] added pill ID {} with name '{}'".format(
103         nextID, name))
104     db[nextID] = data
105     nextID += 1
106
107     # write the new, updated database to disk
108     f = open(conf["db_path"], "wb")
109     f.write(pickle.dumps(db))
110     f.close()

```

---

**Line 77** checks to see if both (1) the `r` key was pressed on our keyboard and (2) our algorithm was able to find the measurement area. Provided both cases hold, we call `find_pills` (**Line 79**) to find all pills on the measurement area.

For each of the pills (**Line 82**), we loop over them individually, and then quantify their color, shape, and texture (**Lines 84-86**) using the methods we implemented earlier in this section.

**Line 89** prompts use to enter the name of the pill. Given the `nextID`, `name`, and respective components of the quantified pill, we can construct a `data` dictionary (**Lines 92-99**), and then add the `data` to our `db` (**Line 104**). We also increment the `nextID` for the next pill registration.

Finally, we round out the registration process by serializing our pill database to disk (**Lines 108-110**).

Now that the registration process is implemented, we can move on to implementing the recognition/verification process:

---

```

112     # check to see if we should recognize the pill(s) in the frame
113     elif key == ord("v") and foundArea:
114         # find all pills in the measurement area
115         pills = find_pills(area, conf["measurement_area_size"])
116
117         # ensure at least one pill was found
118         if len(pills) > 0:

```

---

```

119      # grab the original width of the measurement area, resize
120      # it, and then compute the ratio of the new area width to
121      # the old area width -- we'll use this ratio later when
122      # scaling the bounding boxes of the pills
123      origW = area.shape[1]
124      area = imutils.resize(area, width=256)
125      ratio = area.shape[1] / float(origW)

```

---

On **Line 113** we check to see if (1) the `v` (for verification) key is pressed and (2) that we have found our pill measurement area. Provided both cases hold, we then find all pills on the measurement area (**Line 115**).

**Line 118** makes a check to ensure at least one pill is found, and if so, we:

- i. Grab the original width of the measurement area in `pixels` (**Line 123**)
- ii. Resize the measurement area to have a width of 256 pixels (**Line 124**)
- iii. Compute the `ratio` of the *original* area measurement width to the *new* area measurement width (**Line 125**)

Again, it's important to understand that this computation is being performed in terms of `pixels`. We need this `ratio` so we can draw bounding boxes surrounding each of the pills later in this script.

The next step is to loop over each of the pills:

---

```

127      # loop over each of the detected pills
128      for pill in pills:
129          # quantify the color, shape, and texture of the pill
130          color = describe_color(pill.pill, pill.mask)
131          shape = describe_shape(pill.contour)
132          texture = describe_texture(pill.pill, pill.mask)
133
134          # identify the pill
135          results = ps.search(pill.size, shape, color,
136                               texture, conf["weights"])
137          pillID = results[0][1]
138          name = db[pillID]["name"]
139
140          # compute the bounding box of the pill and then scale
141          # the bounding box based on our computed ratio
142          (x, y, w, h) = cv2.boundingRect(pill.contour)
143          x = int(x * ratio)
144          y = int(y * ratio)
145          w = int(w * ratio)
146          h = int(h * ratio)
147

```

---

---

```

148             # draw a bounding box surrounding the pill along with
149             # the name of the pill itself
150             cv2.rectangle(area, (x, y),
151                         (x + w, y + h), (0, 0, 255), 2)
152             cv2.putText(area, name, (x, y - 15),
153                         cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
154
155             # show the output pill identifications
156             cv2.imshow("Pill IDs", area)

```

---

On **Line 128** we loop over each of the pills in the measurement area. For each pill, we quantify its color, shape, and texture, respectively (**Lines 130-132**).

Given the color, shape, texture, and size components, we can perform our search and attempt to ID the pill (**Lines 135-136**). **Lines 137 and 138** extract the unique ID and name of the medication.

**Lines 142-146** compute the bounding box of the pill contour and then scale the bounding box coordinates based on our original `ratio`. Performing this scaling allows us to draw the bounding box surrounding the pill, along with the medication name itself, on our `frame` via **Lines 150-153**.

Finally, **Line 156** displays the identified pills to our screen.

We're almost done! Just one code block to go:

---

```

158     # if the `q` key was pressed, break from the loop
159     elif key == ord("q"):
160         break
161
162     # do a bit of cleanup
163     cv2.destroyAllWindows()
164     vs.stop()

```

---

**Lines 159 and 160** check to see if the `q` key was pressed, indicating that we should exit from our script. **Lines 163 and 164** do a bit of cleanup.

### 22.5.9 Pill Recognition Results

To put our pill recognition system to the test, open up a terminal and execute the following command:

---

```
$ python pill_id.py --conf config/config.json
[INFO] initializing pill database...
```

---

```
[INFO] starting video stream thread...
Enter the pill name:
[INFO] added pill ID 0 with name 'Aleve'
...
```

**We'll start by registering a few pills first**, so gather your medications and put them, one-by-one on the measurement area of the pill ID card, like in Figure 22.9 (*left*).

When you're ready, press the `r` key on your keyboard to register the pills. For each pill, enter the name of the medication.

After registering your pills you can then recognize/verify them by pressing the `v` key. Take a look at Figure 22.9 (*right*) and you'll see that each of the prescription pills are correctly recognized.

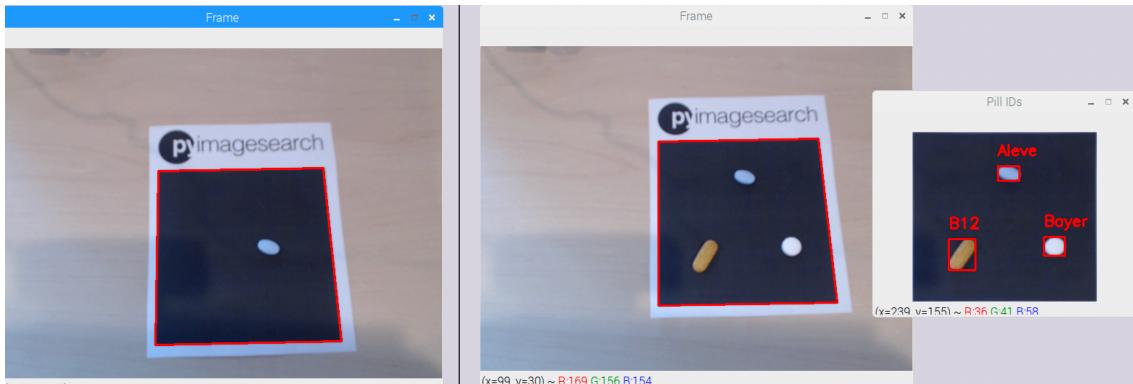


Figure 22.9: **Left:** During the pill registration phase you must place each pill on the verification card (one pill at a time), press the "`r`" key on your keyboard, and then enter the name of the medication. Repeat this process for every pill you want to register. **Right:** When you're ready to recognize the pills, place them on your verification card and press the "`v`" key. Your pills will then be automatically recognized!

## 22.6 Improvements and Suggestions

In this section I'll detail some of my tips and suggestions to improve prescription pill recognition. Some of these tips are fairly straightforward and are relatively simple to implement. Other suggestions are more advanced and will require significantly more effort and manual annotation of images to build.

### 22.6.1 Proper Camera Calibration

Experienced computer vision practitioners will be quick to point out that I did not compute the intrinsic/extrinsic camera parameters and perform a proper camera calibration when measuring pill size. Since we're using the *same* camera sensor we don't need such a calibration.

Yes, the actual “perceived” measured size of the pill may be incorrect in terms of real-world measurements ( $\pm$  a few millimeters, mainly for larger pills), but that doesn’t matter here. Since we’re using the same camera sensor, the distortion is thus a constant.

As long we we don’t use a different camera, our measurements will be off by the same factor. And since we’re comparing the perceived measurement sizes in our database, we can get away without performing a proper calibration.

All that said — if you wish to use *different* camera sensors for prescription pill recognition, then yes, you should *absolutely* compute the intrinsic/extrinsic camera parameters.

**Remark.** *If you’re interested in learning more about intrinsic/extrinsic camera parameter calibration, I’ll be covering the topic inside a future blog post on the PyImageSearch blog. I’m including a short link here (<http://pyimg.co/de8j1>) that will 404 until I publish the guide (then I will update the short link to point to the guide).*

### 22.6.2 Segmentation in Uncontrolled Environments

One drawback of the method we utilized in this chapter is that it *requires* the user to place their medication on the  $100\text{mm} \times 100\text{mm}$  measurement area. Using this measurement area ensures that:

- i. We can easily segment our pill from the input image
- ii. We can measure the pill size relative to our measurement area

However, if you were building a mobile app for pill identification, requiring the user to carry around such a measurement area is not only tedious, but simply unlikely as well. Instead, it would be *far* better to allow the user to place the pill on a counter or in their hand, and then have the app *automatically* segment the pill from an image.

While certainly delivering a better user experience, such a use case is *far* more challenging from a computer vision perspective as we’re now working in uncontrolled environments and can no longer make any assumptions regarding the background, lighting conditions, etc. Given not only the variation of pill colors themselves, but also the lack of assumption regarding background color, we should apply deep learning (and more specifically **instance segmentation**) to automatically segment the pill from the image.

If you’re new to instance segmentation, I would suggest reading the following guides on the PyImageSearch blog:

- i. Instance segmentation with OpenCV (<http://pyimg.co/w8yxv>) [88]
- ii. Semantic segmentation with OpenCV and deep learning (<http://pyimg.co/acgfp>) [89]

- iii. Keras Mask R-CNN (<http://pyimg.co/lSk0f>) [90]

I also cover how to train your own instance segmentation networks via the Mask R-CNN framework inside my book, *Deep Learning for Computer Vision with Python* (<http://pyimg.co/dl4cv>) [8]. Inside the text, I:

- i. Teach you how to train a Mask R-CNN to automatically detect and segment cancerous skin lesions — a first step in building an automatic cancer risk factor classification system
- ii. Provide you with my favorite image annotation tools, enabling you to create masks for your input images
- iii. Show you how to train a Mask R-CNN on your custom dataset
- iv. Provide you with my best practices, tips, and suggestions when training your own Mask R-CNN

**Additionally, I've included how to train a Mask R-CNN network for automatic pill segmentation as a case study in the text as well.**

### 22.6.3 Triplet Loss, Siamese Network, and Deep Metric Learning

We used traditional computer vision feature extraction methods to characterize the color, shape, and texture of a pill — no machine learning was utilized in this chapter. However, given enough labeled data, we could apply *deep metric learning*.

You may already be familiar with deep metric learning from the PyImageSearch tutorials on face recognition (<http://pyimg.co/i39fy>) [91]. In those tutorials, we use deep metric learning to construct a  $128 - d$  vector to quantify a face. Faces that are *more similar* have a small Euclidean distance between them while *less similar* faces have larger distances. The deep learning model automatically learns these embeddings for us using triplet loss and siamese networks.

These concepts are advanced and *well* outside the scope of this book. If you're interested in learning more, be sure to:

- i. Review to the PyImageSearch tutorials linked to above
- ii. Take a look at Schroff et al.'s 2015 CVPR publication, *FaceNet: A Unified Embedding for Face Recognition and Clustering* [92], which includes more details on the inner-workings of the neural network

## 22.7 Summary

In this chapter we looked at an in-depth case study for prescription pill identification. We then implemented such a system using a Raspberry Pi and computer vision algorithms.

This case study was (1) certainly more in-depth than the previous chapters and (2) covered more advanced computer vision algorithms that you may be unfamiliar with (i.e., Hu moments, LBPs, etc.). If you'd like to learn more about these algorithms be sure to refer to the PylImage-Search Gurus course (<http://pyimg.co/gurus>) [12] where I cover them in more detail.



## Chapter 23

# OpenCV Optimizations and Best Practices

As the final chapter in the *Hobbyist Bundle*, I thought it would be best for us to take a step back from implementing various projects and instead **learn how to optimize our code on the Raspberry Pi**.

Regardless of whether you choose to stop your studies here, or if you continue to work through the *Hacker Bundle* and *Complete Bundle*, knowing what various types of operations are available to you, including how to use them and measure them, will better enable you to increase the processing speed of your computer vision project.

Our optimizations are divided into two types:

- **Package and library-level optimizations:** These are special libraries you can install on your Raspberry Pi, enabling it to perform various operations faster
- **Code-level optimizations:** These performance boosts come from measuring your actual *code*, discovering where and what the bottleneck is, and then updating the code to remove the bottleneck

### 23.1 Chapter Learning Objectives

In this chapter you will:

- i. Learn about the NEON and FVPV3
- ii. Discover what “Threading Building Blocks” (TBB) are (and why they can improve performance)
- iii. Take a look at OpenCL (and whether or not it’s helpful on the RPi)

- iv. Discover how to optimize your Python scripts on the RPi
- v. Learn how to time and profile various functions *inside* your code
- vi. Measure Frames Per Second (FPS) throughput rate

## 23.2 Package and Library Optimizations

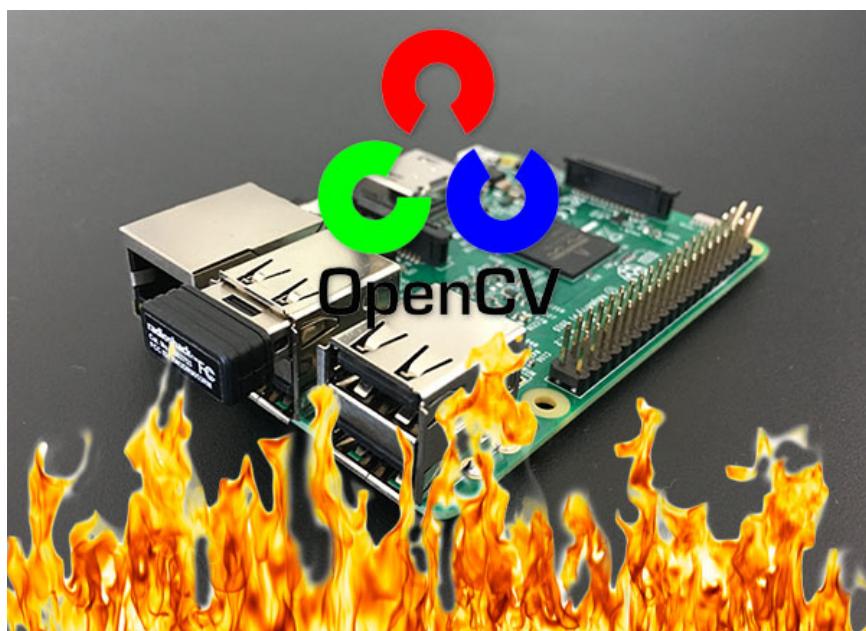


Figure 23.1: In this section you will learn how to utilize package and library-level optimizations to improve performance on your Raspberry Pi.

In this section you will learn about the **package-level optimizations** you can utilize to improve performance on your Raspberry Pi. Using these packages is typically a two step process:

- i. Install the respective package on your RPi (either via `apt-get` or by compiling from source)
- ii. Compile/re-compile OpenCV with the respective flags set to take advantage of our optimization libraries

From there we can enjoy better performance in our computer vision pipelines. We'll briefly discuss four package-level optimizations you may wish to utilize.

Installing these libraries, followed by configuring OpenCV to use them, is covered inside the companion website.

### 23.2.1 NEON and VFPV3

ARM NEON [93] is an optimization architecture extension for ARM processors. It was designed by the ARM engineers specifically for faster video processing, image processing, speech recognition, and machine learning. This optimization supports Single Instruction Multiple Data (SIMD) [94] (as opposed to SISD, MISD, MIMD), which describes an architecture where multiple processing elements in the pipeline perform operations on multiple data points (hardware), all executed with a single instruction.

The ARM engineers also built VFPv3 [95], a floating point optimization, into the chip our Raspberry Pi 3 and Raspberry Pi 4 use. The ARM page, cited above, describes features included in this optimization such as configurable rounding modes and customizable default not a number (NaN) behavior.

What this means for us is that our code can likely run faster, as the ARM processor on the Raspberry Pi 4 has hardware optimizations that we can take advantage of with the 4× BCM2711B0 ARM A72, 1.5GHz processor.

Both NEON and VFPv3 optimizations can be installed using CMake flags when you install OpenCV.

### 23.2.2 TBB

Threading Building Blocks (TBB) is a C++ template developed by Intel for parallel programming and multi-core processors (<http://pyimg.co/h8ng8>) [96].

Since (1) most modern processors are multi-core and (2) some computer vision algorithms can be made parallel, TBB can optimize various routines in OpenCV, enabling the library to run faster on the Raspberry Pi — *that is, at least in theory*.

In practice, TBB is a pain to install on the RPi with OpenCV support. TBB usage only leads to meager gains in performance for most operations.

For the majority of the computer vision pipelines covered in the *Hobbyist Bundle*, you won't see massive performance gains with TBB. That said, where TBB *does* shine is when we start getting into more complex pipelines, including leveraging deep learning and performing neural network inference on the RPi.

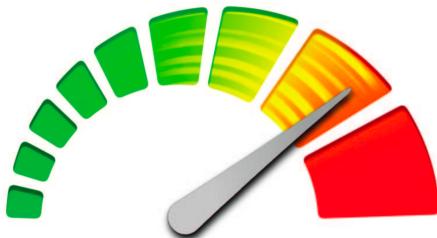
For example, when applying TBB to our *YOLO object detection with OpenCV* tutorial (<http://pyimg.co/6frf> [97]), ***we found that the code ran 2x faster!*** When it comes to deep learning on a resource constrained device, such as the RPi, that result is a *massive* improvement.

If you find yourself needing to run neural networks on the RPi using OpenCV's `dnn` module then TBB may be worth the time and headache to install.

Again, instructions to compile OpenCV with TBB is documented inside the companion website of this text.

Otherwise, using a dedicated coprocessor such as a Movidius NCS or Google Coral USB Accelerator will likely be easier (and obtain better results). Both the Movidius and Coral USB Accelerator are covered in the *Hacker* and *Complete Bundles*.

### 23.2.3 OpenCL



## OpenCL

Figure 23.2: OpenCL can be used to write software capable of executing on CPUs, GPUs, FGPAs, GPUs, and more.

OpenCL is a framework for writing software that can execute across multiple platforms, including CPUs, GPUs, DSPs, FPGAs, and other hardware accelerators and/or coprocessors. The benefit of using OpenCL is that you can write code as you normally would, then insert an additional function call, instructing OpenCL on where to run the computation (i.e., GPU instead of CPU).

Being able to specify a device for computation is *especially* interesting when performing deep learning. As we know, deep neural networks are incredibly computationally expensive — pushing that computation to a GPU versus a CPU can dramatically improve our performance.

#### **And believe it or not, there *is* a GPU on the Raspberry Pi.**

The Raspberry Pi 3B+ VideoCore IV GPU has 12 independent cores, is located on the same chip as the CPU, and has a theoretical maximum performance of 24 GFLOPS [98].

The Raspberry Pi 4 possess the updated Broadcom VideoCore VI GPU which has a main feature of supporting dual HDMI outputs with 4K capability. There is chatter on the RPi forums indicating a theoretical performance of 100 GFLOPS (in comparison to the NVIDIA Jetson

Nano's deep learning GPU with 250+ GFLOP performance) [99].

For computationally expensive operations it could be worthwhile to utilize our RPi GPU via OpenCL.

### So, what's the catch?

If you're thinking this is too good to be true, you're unfortunately correct (for right now, anyway).

In order to utilize OpenCL we need to install VC4CL, an implementation of OpenCL 1.2 for the Raspberry Pi (<http://pyimg.co/ca0ze>) [100]; however, keep in mind that VC4CL is *only* compatible with the Raspberry Pi 3. It will *not* work with the RPi 4.

VC4CL is under *heavy development* and is not fully ready to be utilized with OpenCV. That doesn't mean that OpenCL and OpenCV support for the Raspberry Pi is not coming — it is — but it may be awhile until you can fully enjoy the optimizations. If you're interested in attempting to install OpenCV with OpenCL support on your RPi, you can find our configuration instructions on the companion website.

Otherwise, if you need to obtain faster neural network inference results on your Raspberry Pi, I would instead suggest you utilize a coprocessor such as the Movidius NCS or Google Coral USB Accelerator (both of which are covered in the *Hacker* and *Complete Bundles* of this text).

## 23.3 Benchmarking and Profiling Your Scripts

So far we've only reviewed package and library level optimizations — ***but what about your Python scripts themselves?***

How can we profile our code, and then, based on those results, optimize our code so it runs faster on the Raspberry Pi?

In this section I'll show you both some simple, and some not so simple, ways to benchmark and profile your code.

### 23.3.1 Project Structure

Let's take a second and look at the directory structure for this project:

---

```
|-- coins.png
|-- haarcascade_frontalface_default.xml
|-- basic_operations.py
```

---

```
|-- measure_fps.py
|-- cprofile_measure_fps.py
```

---

The `basic_operations.py` script will be used to perform basic image processing operations. We'll be profiling our code to determine what steps of the pipeline take the longest. The `measure_fps.py` script, as the name suggests, will measure the FPS throughput rate of a particular pipeline. Our `cprofile_measure_fps.py` script simply has a few additional `cProfile` lines of code for printing profiling stats automatically.

### 23.3.2 Timing Operations and Functions

In this section we'll learn how to use Python's built-in `time` module so you can time various routines, ultimately giving you insight into how long a given step of a pipeline takes.

But before we can actually *profile* our code, we *first* need to implement a script. The script we'll be implementing here is similar to our object counter from Chapter 4; however, we'll now be inserting special statements to profile how long the execution of each given function takes.

Open up the `basic_operations.py` file and insert the following code:

---

```
1 # import the necessary packages
2 import numpy as np
3 import time
4 import cv2
5
6 def time_it(msg, start, end):
7     # show the time difference
8     print("[INFO] {} took {:.8} seconds".format(msg, end - start))
```

---

**Lines 2-4** handle importing our required packages.

**Line 6** defines our `time_it` function. This method requires three parameters:

- `msg`: An informative message that will be displayed to our screen (such as a description of what is being profiled/measured)
- `start`: A timestamp representing when the profiling started
- `end`: A timestamp indicating when the profiling ended

**Line 8** takes these variables, subtracts `start` from the `end` to obtain the time elapsed, and then displays the `msg` and elapsed time to our screen.

Given the `time_it` function, we can start profiling our code:

---

```

10 # load the image from disk
11 start = time.time()
12 image = cv2.imread("coins.png")
13 time_it("load", start, time.time())

```

---

**Line 11** grabs the `start` timestamp of our operation. **Line 12** loads our example image from disk. We then take the ending timestamp and determine how long the load operation took.

When profiling your own code, you should place the `start` timestamp at wherever the *starting point* of the code block you want to profile resides. The `time_it` call is then placed at the *end point* of the code block.

Next, let's profile how long it takes to convert the image to grayscale, blur it, and then perform edge detection:

---

```

15 # convert the image to grayscale
16 start = time.time()
17 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
18 time_it("grayscale", start, time.time())
19
20 # blur the image
21 start = time.time()
22 blurred = cv2.GaussianBlur(gray, (11, 11), 0)
23 time_it("blur", start, time.time())
24
25 # perform edge detection
26 start = time.time()
27 edged = cv2.Canny(blurred, 30, 150)
28 time_it("edge detection", start, time.time())

```

---

Given the edge map, let's stack the original image and edge map so we can visualize both on our screen via a single `cv2.imshow` call:

---

```

30 # stack the original input image and edge map image so we can
31 # visualize it with a single cv2.imshow call
32 start = time.time()
33 output = np.hstack([image, np.dstack([edged] * 3)])
34 time_it("stacking", start, time.time())
35
36 # show the output image
37 cv2.imshow("Output", output)
38 cv2.waitKey(0)

```

---

To run the example script, open up a terminal and execute the following command:

---

```
$ python basic_operations.py
[INFO] load took 0.0052177906 seconds
[INFO] grayscale took 0.00025629997 seconds
[INFO] blur took 0.001147747 seconds
[INFO] edge detection took 0.000903368 seconds
[INFO] stacking took 0.0015683174 seconds
```

---

As the results show, loading the image from disk requires the most amount of time (0.005s). Conversely converting the image to grayscale is the fastest operation out of all those measured (0.002s).

### 23.3.3 Measuring FPS Throughput

In our previous section we learned how to profile single code blocks — but what if we wanted to measure the FPS throughput rate of a pipeline?

In those situations a simple call to `time_it` will not be sufficient. Instead, we would need to loop over frames from a video stream, process them, increment a counter, and then approximate the frames per second throughput by dividing the total number of frames by the elapsed time. Luckily, performing such an operation is quite easy given the `FPS` implementation in the `imutils` library [35].

To see how we can measure the FPS throughput rate of a pipeline, let's look at an example of performing face detection, similar to Chapter 4. Go ahead and open `measure_fps.py`:

---

```
1 # import the necessary packages
2 from imutils.video import VideoStream
3 from imutils.video import FPS
4 import imutils
5 import time
6 import cv2
7
8 # initialize the video stream and allow the camera sensor to warmup
9 print("[INFO] starting video stream...")
10 vs = VideoStream(src=0).start()
11 #vs = VideoStream(usePiCamera=True, resolution=(640, 480)).start()
12 time.sleep(2.0)
13
14 # load OpenCV's face detector and start our FPS throughput measurement
15 # class
16 detector = cv2.CascadeClassifier("haarcascade_frontalface_default.xml")
17 fps = FPS().start()
```

---

On **Lines 2-6** we import our required Python packages. Notice the import of `FPS` — this class will be used to measure our FPS throughput rate.

**Line 10** starts our video stream, while **Line 16** loads the Haar cascade face detector. **Line 17** then **initiates the FPS measurement**.

Next, let's start looping over frames from the video stream:

---

```

19 # loop over the frames from the video stream
20 while True:
21     # grab the frame from the video stream, resize it to have a
22     # maximum width of 400 pixels, and then convert to grayscale
23     frame = vs.read()
24     frame = imutils.resize(frame, width=400)
25     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
26
27     # detect faces in the grayscale image
28     rects = detector.detectMultiScale(gray, scaleFactor=1.05, minNeighbors=9,
29                                         minSize=(40, 40), flags=cv2.CASCADE_SCALE_IMAGE)
30
31     # loop over the bounding boxes and draw a rectangle around each face
32     for (x, y, w, h) in rects:
33         cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
34
35     # show the output frame
36     cv2.imshow("Frame", frame)
37     key = cv2.waitKey(1) & 0xFF
38
39     # if the `q` key was pressed, break from the loop
40     if key == ord("q"):
41         break
42
43     # update the FPS counter
44     fps.update()

```

---

On **Line 23** we grab a `frame` from the video stream, resize it to have a width of 400px, and then convert it to grayscale.

Using the `gray` representation of the `frame`, we perform face detection using Haar cascades (**Lines 28 and 29**).

Given the bounding box `rects` for each face, we loop over them and draw the rectangles on the `frame` (**Lines 28-33**).

**Lines 35-41** display the output `frame` to our screen and record any keypresses.

**Line 44** updates the total number of frames processed by our pipeline.

After we've done a bit of processing we can press the `q` key on our keyboard, exit the `while` loop, and have approximate FPS throughput measurements displayed to our screen:

---

```
46 # stop the timer and display FPS information
```

---

```

47 fps.stop()
48 print("[INFO] elasped time: {:.2f}".format(fps.elapsed()))
49 print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
50
51 # do a bit of cleanup
52 cv2.destroyAllWindows()
53 vs.stop()

```

---

To measure the FPS throughput rate of our face detection pipeline, execute the following command:

---

```

$ python measure_fps.py
[INFO] starting video stream...
[INFO] elasped time: 77.40
[INFO] approx. FPS: 13.85

```

---

As you can see, I have let the script run for approximately 75 seconds. At the end of the run we are obtaining  $\approx 13.85$  FPS. The quickest way to improve on this result would be to process a smaller image. Try changing the `width=200` on **Line 24** and see what happens!

### 23.3.4 Python's Built-in Profiler

As we learned in the previous section, you can (and should) profile your scripts by manually inserting timestamps and computing the difference. But is there a way to *automatically* profile a Python computer vision script and collect statistics?

In fact, there is.

Python has a built-in tool called `cProfile`. The tool adds little overhead (i.e. requires very few CPU cycles) to your scripts, and is a great way to gauge the performance of your scripts. For reference and more of a backstory, you should be sure to visit the Python Profilers documentation <http://pyimg.co/pq2t2>.

I like to use `cProfile` in the following two ways: (1) Via the command line with no changes to my code, and (2) Inside my program so that I can highlight specific function statistics via regular expression. Let's review each method now.

#### 23.3.4.1 Command Line Profiling

As shown in the documentation, you can profile any script on the command line with the following syntax:

---

```
$ python -m cProfile -s tottime script.py --arg1 arg --arg2 arg
```

---

The only difference when calling your script is that you'll insert `-m cProfile -s tottime` in between your call to the Python executable and your script. This activates Python profiling from the command line.

Optionally, you can sort by one of the columns with the `-s <column>` command line argument. Usually, I sort by either `tottime` or `cumtime`. From there, you insert the script name and any of the script's command line arguments (shown are two example command line arguments).

In this section, we'll use the Haar Cascade script from earlier in this chapter: `measure_fps.py`.

Let's profile the `measure_fps.py` script while piping the output to a text file to keep track of our experiment:

---

```
$ python -m cProfile -s cumtime measure_fps.py
[INFO] starting video stream...
[INFO] elasped time: 78.50
[INFO] approx. FPS: 13.17
    93188 function calls (90419 primitive calls) in 81.936 seconds

Ordered by: cumulative time

      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        180/1   0.004    0.000   81.936   81.936 {built-in method builtins.exec}
          1    0.501    0.501   81.936   81.936 measure_fps.py:5(<module>)
        1035   68.510   0.066   68.510    0.066 {method 'detectMultiScale' of 'cv2.CascadeClassifier' objects}
        1035    0.073    0.000    6.797    0.007 convenience.py:65(resize)
        1035    6.725    0.006    6.725    0.006 {resize}
          1    2.002    2.002    2.002    2.002 {built-in method time.sleep}
        1035    1.646    0.002    1.646    0.002 {waitKey}
        206/1    0.004    0.000    0.695    0.695 <frozen importlib._bootstrap>:978(_find_and_load)
        206/1    0.002    0.000    0.695    0.695 <frozen importlib._bootstrap>:948(_find_and_load_unlocked)
        278/2    0.001    0.000    0.694    0.347 <frozen importlib._bootstrap>:211(_call_with_frames_removed)
        198/2    0.003    0.000    0.694    0.347 <frozen importlib._bootstrap>:663(_load_unlocked)
        166/2    0.002    0.000    0.693    0.347 <frozen importlib._bootstrap_external>:722(exec_module)
        325/1    0.001    0.000    0.688    0.688 {built-in method builtins.__import__}
          2    0.000    0.000    0.687    0.343 __init__.py:5(<module>)
          1    0.000    0.000    0.685    0.685 convenience.py:5(<module>)
        1035    0.588    0.001    0.588    0.001 {imshow}
        1035    0.574    0.001    0.574    0.001 {cvtColor}
...
```

---

There is a lot of output (in fact, many lines are omitted as there is over 1,000 lines of output). That said, notice in the output that `detectMultiScale` required 68.510s cumulatively. Per call it requires 0.066s, clearly less than the 2.002s per call for `time.sleep` (as to be expected).

There's a lot of noise, and with practice you'll develop the eye for picking out OpenCV functions which could be the culprit of a slow computer vision pipeline. With deeper knowledge of what is going on under the hood, you can adapt your computer vision applications for speed — a crucial skill for working on resource-constrained devices.

### 23.3.4.2 Profiling Within a Python Script

Profiling your script internally gives you more control over where you start and stop profiling. You can experiment with profiling the whole script (as we'll do) or just profiling sections of your script (such as a function or where DL inference and post-processing is handled).

Go ahead and make a copy of `measure_fps.py` and name it `cprofile_measure_fps.py`.

Open up the script and make the following changes:

First add the `cProfile` import on **Line 4**:

---

```

1 # import the necessary packages
2 from imutils.video import VideoStream
3 from imutils.video import FPS
4 import cProfile
5 import imutils
6 import time
7 import cv2

```

---

After the imports, insert the following lines to activate and enable the profiler:

---

```

9 # initialize and enable the profiler
10 pr = cProfile.Profile()
11 pr.enable()

```

---

From there, scroll to the very bottom of the script to disable the profiler and print sorted statistics:

---

```

60 # disable the profiler and print sorted statistics
61 pr.disable()
62 pr.print_stats(sort="time")

```

---

The statistics will be sorted by "time" — you should experiment with other means of sorting as well such as "percall", "cumulative", or any of the other options listed here: <http://pyimg.co/iybm6>.

From there, run the script as you normally would:

---

```

$ python cprofile_measure_fps.py
[INFO] starting video stream...
[INFO] elasped time: 69.72
[INFO] approx. FPS: 13.24

```

---

```

9293 function calls in 71.982 seconds
...

```

```

pi@raspberrypi: ~opencv_optimizations
File Edit Tabs Help
(pi3cv4) pi@raspberrypi:~/opencv_optimizations $ python cprofile_measure_fps.py
[INFO] starting video stream...
[INFO] elasped time: 69.72
[INFO] approx. FPS: 13.24
    9293 function calls in 71.982 seconds

Ordered by: internal time

      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
         924   60.547    0.066   60.547    0.066 {method 'detectMultiScale' of 'cv2.CascadeClassifier' objects}
           924    6.234    0.007    6.234    0.007 {resize}
             1    2.002    2.002    2.002    2.002 {built-in method time.sleep}
           924    1.452    0.002    1.452    0.002 {waitKey}
           924    0.531    0.001    0.531    0.001 {cvtColor}
           924    0.516    0.001    0.516    0.001 {imshow}
             1    0.264    0.264    0.264    0.264 {method 'read' of 'cv2.VideoCapture' objects}
             1    0.241    0.241    0.506    0.506 webcamvideostream.py:6(__init__)
           924    0.094    0.000    0.094    0.000 {rectangle}
           924    0.071    0.000    6.305    0.007 convenience.py:65(resize)
           923    0.018    0.000    0.018    0.000 fps.py:21(update)
           924    0.005    0.000    0.005    0.000 {built-in method builtins.ord}
           924    0.003    0.000    0.003    0.000 webcamvideostream.py:36(read)
             1    0.002    0.002    0.002    0.002 {destroyAllWindows}

```

Figure 23.3: An example of running a script with `cProfile` activated for calculating profiling statistics.

As you can see from the terminal output inside Figure 23.3, there is a lot of information to be gained by profiling your script (although it can be a bit of information overload).

Take note that the call to `detectMultiScale` (i.e., running our face detector) is the most computationally expensive operation.

Also note that both resizing and color space conversion are not "free" operations – each of these function calls consume computational resources, and thus time.

Reading a frame from the video stream (i.e. a frame from the video file on disk) also takes time, but there is no way around it – disk and I/O operations are inherently slow.

Use the profiling to your advantage to see what is slowing down your computer vision apps (along with manual timestamps). From there you can engineer improvements with your new-found knowledge.



Figure 23.4: Cython is a great tool to keep in your toolbox for optimizing custom pixel for-loops.

## 23.4 Improving Computation Speed with Cython

We all know that Python, being a high-level language, provides a lot of abstraction and convenience — that's the main reason why it is so great for image processing. What comes with this typically is slower speeds than a language which is closer to assembly, like C.

You can think of Cython as a combination of Python with traces of C which provides C-like performance. Cython differs from Python in that the code is translated to C using the CPython interpreter. This allows the script to be written mostly in Python, along with some decorators and type declarations.

### So when should you take advantage of Cython in image processing?

Probably the best time to use Cython would be when you find yourself looping pixel-by-pixel in an image, like this:

---

```

1 def threshold_slow(T, image):
2     # grab the image dimensions
3     h = image.shape[0]
4     w = image.shape[1]
5
6     # loop over the image, pixel by pixel
7     for y in range(0, h):
8         for x in range(0, w):
9             # threshold the pixel
10            image[y, x] = 255 if image[y, x] >= T else 0
11
12    # return the thresholded image
13    return image

```

---

If you find yourself writing any custom image processing functions in Python which analyze or modify images pixel-by-pixel, such as in the example above, it is *extremely likely* that your function won't run as fast as possible. In fact, it will run *very* slowly.

However, if you take advantage of Cython, which compiles with major C/C++ compilers, you

can achieve significant performance gains.

A full discussion of Cython and how to use it is outside the scope of this chapter, but if you are interested in learning more, be sure to refer to this tutorial on the PyImageSearch blog (<http://pyimg.co/fai5c>) [101].

## 23.5 Summary

In this chapter you learned how to optimize your Raspberry Pi for computer vision. There are two types of optimization we can perform: (1) Package/library-level, and (2) Code-level.

NEON, VFPv3, TBB, and OpenCL are all examples of library-level optimizations. We install these packages on our Raspberry Pi, compile OpenCV with the respective flags set, instructing OpenCV to optimize the library for these packages, install the optimized version of OpenCV, and then reap the benefits.

Currently, NEON and VFPv3 will give you the best bang for your buck when it comes to performance — they are also easy to install *and* the pip install of OpenCV (<http://pyimg.co/y8t7m>) [102] now has NEON and VFPv3 optimizations *pre-compiled* so you no longer have to compile OpenCV from source to take advantage of them. TBB is harder to install, but if successful, *can* improve your performance. Installing OpenCV with TBB support on the RPi has been documented on the companion website.

OpenCL is very much in its infancy for the Raspberry Pi. We were able to install OpenCL on the RPi 3 after a lengthy process (documented on the companion website), but were a bit underwhelmed by its performance gains.

Furthermore, the OpenCL implementation we tested is *only* compatible with the RPi 3. The library will *not* work with the RPi 4.

OpenCL has *incredible* potential but it needs to be optimized for usage with OpenCV first. Once that happens we'll likely see *much* better improvements in performance.

We then looked at code-level optimizations, including profiling your code with timing statements (or using `cProfile`), measuring FPS throughput, and finally, situations where you may want to consider using Cython.



## Chapter 24

# Your Next Steps

Take a second to congratulate yourself! You've worked through the entire *Hobbyist Bundle of Raspberry Pi for Computer Vision*. That's quite an achievement.

Let's take a second reflect on what your journey into the world of computer vision on the Raspberry Pi and resource constrained devices.

Inside this book you have:

- Learned how the Raspberry Pi (and other resource constrained devices, such as the Google Coral and NVIDIA Jetson Nano) can be used for computer vision.
- Studied the overlap and intersection of the computer vision and deep learning fields
- Configured your development environment on the RPi
- Discovered how to access *both* USB webcams as well as the PiCamera module
- Learned how to change, adjust, and set various camera parameters
- Launched your CV scripts on reboot
- Created a simple bird feeder monitor
- Used computer vision and Python to send text message notifications from your RPi
- Created a OpenCV script to detect when your mail has been delivered
- Discovered how to work in low light conditions with the RPi
- Built a remote wildlife monitor/camera capture system
- Implemented a video surveillance system capable of streaming to your web browser from the RPi
- Created a computer vision to detect tired, drowsy drivers behind the wheel of a vehicle

- Learned what a PID is (and how to tune it)
- Used PIDs to create a pan/tilt face tracking system
- Implemented a people/footfall counter
- Extended the footfall counter to count vehicles on the highway
- Learned how to measure the size of objects in images
- Created a custom computer vision system to recognize prescription pills in images, thereby reducing the number of injuries and deaths each year due to prescription pill mixups
- Discovered various optimization techniques to improve speed on the RPi

## 24.1 So, What's next?

At this point you have enough education to start building your own computer vision applications on the Raspberry Pi.

However, the *Hobbyist Bundle* of *Raspberry Pi for Computer Vision* is just the start – if you want to:

- Apply deep learning to the RPi
- Perform face recognition on the RPi
- Create your own image classifiers and deploy them to the RPi
- Train custom object detectors for embedded devices
- Deploy object detectors and obtain real-time performance
- Work with the Movidius NCS and the RPi
- Utilize the Google Coral for *super fast* deep learning inference
- Discover how to use the NVIDIA Jetson Nano
- Build more advanced computer vision and deep learning applications
- Discover my tips, suggestions, and best practices for CV and DL on resource constrained devices...

*then I would highly encourage you to not stop here.* Continue your journey and fully master computer vision and deep learning on embedded devices. I can guarantee you that the *Hacker Bundle* and *Complete Bundle* only get better from here.

I hope you'll allow me to continue to guide you on your journey. If you haven't already picked up a copy of the *Hacker Bundle* or *Complete Bundle*, you can do so here:

<http://pyimg.co/rpi4cv>

And if you have any questions, feel free to contact me:

<http://pyimg.co/contact>

Cheers,

-Adrian Rosebrock



# Bibliography

- [1] Intel Developer Zone. *Intel Movidius Neural Compute Stick*. <https://software.intel.com/en-us/movidius-ncs>. 2019 (cited on pages 18, 22).
- [2] Google Developers. *Google Coral USB Accelerator*. <https://coral.withgoogle.com/products/accelerator/>. 2019 (cited on pages 18, 22).
- [3] Seeed Studio. *Raspberry Pi 4 Computer Model B*. <https://www.seeedstudio.com/Raspberry-Pi-4-Computer-Model-B-4GB-p-4077.html>. 2019 (cited on page 19).
- [4] TensorFlow Community. *TensorFlow Lite: Deploy machine learning models on mobile and IoT devices*. <https://www.tensorflow.org/lite>. 2019 (cited on page 21).
- [5] Google Developers. *Google Coral Dev Board*. <https://coral.withgoogle.com/products/dev-board>. 2019 (cited on page 22).
- [6] NVIDIA Developers. *NVIDIA Jetson Nano Development Kit*. <https://developer.nvidia.com/embedded/jetson-nano-developer-kit>. 2019 (cited on page 22).
- [7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016 (cited on page 26).
- [8] Adrian Rosebrock. *Deep Learning for Computer Vision with Python, 2nd Ed.* PyImageSearch, 2018 (cited on pages 27, 30, 386).
- [9] Geoffrey Hinton. *What Was Actually Wrong With Backpropagation in 1986?* <https://www.axios.com/artificial-intelligence-pioneer-says-we-need-to-start-over-1513305524-f619efbd-9db0-4947-a9b2-7a4c310a28fe.html>. 2017 (cited on page 27).
- [10] Seymour A. Papert. “The Summer Vision Project”. In: *Massachusetts Institute of Technology* (1966) (cited on page 28).
- [11] Adrian Rosebrock. *Practical Python and OpenCV + Case Studies, 4th Ed.* PyImageSearch.com, 2019. URL: <https://www.pyimagesearch.com/practical-python-opencv/> (cited on pages 30, 41, 42, 65, 102, 349, 355).

- [12] Adrian Rosebrock. *PyImageSearch Gurus*. <https://www.pyimagesearch.com/pyimagesearch-gurus/>. 2019 (cited on pages 30, 41, 65, 355, 359, 387).
- [13] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000) (cited on page 35).
- [14] Stéfan van der Walt et al. "scikit-image: image processing in Python". In: *PeerJ* 2 (June 2014), e453. ISSN: 2167-8359. DOI: [10.7717/peerj.453](https://doi.org/10.7717/peerj.453). URL: <https://doi.org/10.7717/peerj.453> (cited on page 35).
- [15] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pages 2825–2830 (cited on page 35).
- [16] Davis E. King. "Dlib-ml: A Machine Learning Toolkit". In: *J. Mach. Learn. Res.* 10 (Dec. 2009), pages 1755–1758. ISSN: 1532-4435. URL: <http://dl.acm.org/citation.cfm?id=1577069.1755843> (cited on pages 35, 225, 234).
- [17] François Chollet et al. *Keras*. <https://github.com/fchollet/keras>. 2015 (cited on page 35).
- [18] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: <http://tensorflow.org/> (cited on page 35).
- [19] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *Proceedings of the 22Nd ACM International Conference on Multimedia*. MM '14. Orlando, Florida, USA: ACM, 2014, pages 675–678. ISBN: 978-1-4503-3063-3. DOI: [10.1145/2647868.2654889](https://doi.acm.org/10.1145/2647868.2654889). URL: <http://doi.acm.org/10.1145/2647868.2654889> (cited on page 36).
- [20] Adrian Rosebrock. *Python, argparse, and command line arguments*. <https://www.pyimagesearch.com/2018/03/12/python-argparse-command-line-arguments/>. 2018 (cited on pages 38, 104, 113).
- [21] Adrian Rosebrock. *OpenCV Tutorial: A Guide to Learn OpenCV*. <https://www.pyimagesearch.com/2018/07/19/opencv-tutorial-a-guide-to-learn-opencv/>. 2018 (cited on pages 42, 54).
- [22] Satya Mallick. *Why does OpenCV use BGR color format?* <https://www.learnopencv.com/why-does-opencv-use-bgr-color-format/>. 2015 (cited on page 45).
- [23] Adrian Rosebrock. *Rotate images (correctly) with OpenCV and Python*. <https://www.pyimagesearch.com/2017/01/02/rotate-images-correctly-with-opencv-and-python/>. 2017 (cited on page 49).
- [24] Adrian Rosebrock. *Convolutions with OpenCV and Python*. <https://www.pyimagesearch.com/2016/07/25/convolutions-with-opencv-and-python/>. 2016 (cited on page 50).

- [25] Adrian Rosebrock. *1.5: Kernels*. <https://gurus.pyimagesearch.com/lessons/kernels/>. 2017 (cited on page 50).
- [26] Mehmet Sezgin and Bülent Sankur. “Survey over image thresholding techniques and quantitative performance evaluation.” In: *J. Electronic Imaging* 13.1 (2004), pages 146–168. URL: <http://dblp.uni-trier.de/db/journals/jei/jei13.html#SezginS04> (cited on page 58).
- [27] Paul Viola and Michael Jones. “Rapid object detection using a boosted cascade of simple features”. In: 2001, pages 511–518 (cited on page 62).
- [28] Navneet Dalal and Bill Triggs. “Histograms of Oriented Gradients for Human Detection”. In: *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*. CVPR '05. Washington, DC, USA: IEEE Computer Society, 2005, pages 886–893. ISBN: 0-7695-2372-2. DOI: <10.1109/CVPR.2005.177>. URL: <http://dx.doi.org/10.1109/CVPR.2005.177> (cited on page 63).
- [29] Ross B. Girshick et al. “Rich feature hierarchies for accurate object detection and semantic segmentation”. In: *CoRR* abs/1311.2524 (2013). URL: <http://arxiv.org/abs/1311.2524> (cited on page 63).
- [30] Ross B. Girshick. “Fast R-CNN”. In: *CoRR* abs/1504.08083 (2015). arXiv: <1504.08083>. URL: <http://arxiv.org/abs/1504.08083> (cited on page 63).
- [31] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *CoRR* abs/1506.01497 (2015). URL: <http://arxiv.org/abs/1506.01497> (cited on page 63).
- [32] Wei Liu et al. “SSD: Single Shot MultiBox Detector”. In: *CoRR* abs/1512.02325 (2015). URL: <http://arxiv.org/abs/1512.02325> (cited on page 63).
- [33] Joseph Redmon and Ali Farhadi. “YOLOv3: An Incremental Improvement”. In: *CoRR* abs/1804.02767 (2018). arXiv: <1804.02767>. URL: <http://arxiv.org/abs/1804.02767> (cited on page 63).
- [34] Tsung-Yi Lin et al. “Focal Loss for Dense Object Detection”. In: *CoRR* abs/1708.02002 (2017). URL: <http://arxiv.org/abs/1708.02002> (cited on page 63).
- [35] Adrian Rosebrock. *imutils: A series of convenience functions to make basic image processing operations such as translation, rotation, resizing, skeletonization, and displaying Matplotlib images easier with OpenCV and Python*. <https://github.com/jrosebr1/imutils>. 2019 (cited on pages 69, 396).
- [36] Adrian Rosebrock. *Increasing webcam FPS with Python and OpenCV*. <https://www.pyimagesearch.com/2015/12/21/increasing-webcam-fps-with-python-and-opencv/>. 2015 (cited on page 69).

- [37] Adrian Rosebrock. *Increasing Raspberry Pi FPS with Python and OpenCV*. <https://www.pyimagesearch.com/2015/12/28/increasing-raspberry-pi-fps-with-python-and-opencv/>. 2015 (cited on page 69).
- [38] Adrian Rosebrock. *Unifying picamera and cv2.VideoCapture into a single class with OpenCV*. <https://www.pyimagesearch.com/2016/01/04/unifying-picamera-and-cv2-videocapture-into-a-single-class-with-opencv/>. 2016 (cited on page 70).
- [39] PiCamera Community. *API - The PiCamera Class*. [https://picamera.readthedocs.io/en/release-1.13/api\\_camera.html](https://picamera.readthedocs.io/en/release-1.13/api_camera.html). 2019 (cited on pages 89, 91).
- [40] Said van de Klundert. *Python Exceptions: An Introduction*. <https://realpython.com/python-exceptions/>. 2019 (cited on page 106).
- [41] Adrian Rosebrock. *Writing to video with OpenCV*. <https://www.pyimagesearch.com/2016/02/22/writing-to-video-with-opencv/>. 2016 (cited on page 110).
- [42] Adrian Rosebrock. *Saving key event video clips with OpenCV*. <https://www.pyimagesearch.com/2016/02/29/saving-key-event-video-clips-with-opencv/>. 2016 (cited on page 127).
- [43] Twilio Community. *Twilio - Communication APIs for SMS, Voice, Video and Authentication*. <https://www.twilio.com/>. 2019 (cited on page 142).
- [44] Adrian Rosebrock. *Building a Raspberry Pi security camera with OpenCV*. <https://www.pyimagesearch.com/2019/03/25/building-a-raspberry-pi-security-camera-with-opencv/>. 2019 (cited on page 160).
- [45] National Instruments Inc. *A Practical Guide to Machine Vision Lighting*. <https://www.ni.com/en-us/innovations/white-papers/12/a-practical-guide-to-machine-vision-lighting.html>. 2019 (cited on pages 165, 167).
- [46] Inc. Omron Microscan Systems. *Smart Series Ring Illuminators*. <https://www.ni.com/en-us/innovations/white-papers/12/a-practical-guide-to-machine-vision-lighting.html>. 2018 (cited on pages 165, 166).
- [47] Digital Loggers Community. *IoT - Digital Loggers Direct*. <https://dlidirect.com/products/iot-power-relay>. 2019 (cited on page 168).
- [48] Thomas C. Wilkes et al. “Ultraviolet Imaging with Low Cost Smartphone Sensors: Development and Application of a Raspberry Pi-Based UV Camera”. In: *Sensors* 16.10 (2016). ISSN: 1424-8220. URL: <https://www.mdpi.com/1424-8220/16/10/1649> (cited on page 170).
- [49] Josphe Howse. *Visualizing the Invisible*. [https://www.pyimageconf.com/static/talks/joseph\\_howse.pdf](https://www.pyimageconf.com/static/talks/joseph_howse.pdf). 2018 (cited on page 170).

- [50] Adrian Rosebrock. *OpenCV Gamma Correction*. <https://www.pyimagesearch.com/2015/10/05/opencv-gamma-correction/>. 2015 (cited on page 172).
- [51] Flask Community. *Flask - The Pallets Projects*. <https://palletsprojects.com/p/flask/>. 2019 (cited on page 196).
- [52] Django Community. *Django - The Web framework for perfectionists with deadlines*. <https://www.djangoproject.com/>. 2019 (cited on page 196).
- [53] Adrian Rosebrock. *Multiple cameras with the Raspberry Pi and OpenCV*. <https://www.pyimagesearch.com/2018/07/23/simple-object-tracking-with-opencv/>. 2016 (cited on page 209).
- [54] Matthew Brown and David G. Lowe. "Automatic Panoramic Image Stitching using Invariant Features". In: *International Journal of Computer Vision* 74.1 (Aug. 2007), pages 59–73. ISSN: 1573-1405. DOI: [10.1007/s11263-006-0002-3](https://doi.org/10.1007/s11263-006-0002-3). URL: <https://doi.org/10.1007/s11263-006-0002-3> (cited on pages 210, 215).
- [55] David G. Lowe. "Object Recognition from Local Scale-Invariant Features". In: *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*. ICCV '99. Washington, DC, USA: IEEE Computer Society, 1999, pages 1150–. ISBN: 0-7695-0164-8. URL: <http://dl.acm.org/citation.cfm?id=850924.851523> (cited on pages 213, 215).
- [56] Adrian Rosebrock. *OpenCV panorama stitching*. <https://www.pyimagesearch.com/2016/01/11/opencv-panorama-stitching/>. 2016 (cited on page 215).
- [57] Adrian Rosebrock. *Real-time panorama and image stitching with OpenCV*. <https://www.pyimagesearch.com/2016/01/25/real-time-panorama-and-image-stitching-with-opencv/>. 2016 (cited on page 215).
- [58] Adrian Rosebrock. *Object detection with deep learning and OpenCV*. <https://www.pyimagesearch.com/2017/09/11/object-detection-with-deep-learning-and-opencv/>. 2017 (cited on page 216).
- [59] Adrian Rosebrock. *Raspberry Pi: Deep learning object detection with OpenCV*. <https://www.pyimagesearch.com/2017/10/16/raspberry-pi-deep-learning-object-detection-with-opencv/>. 2017 (cited on page 221).
- [60] Adrian Rosebrock. *Drowsiness detection with OpenCV*. <https://www.pyimagesearch.com/2017/05/08/drowsiness-detection-opencv/>. 2017 (cited on page 223).
- [61] Danny Shapiro. *NVIDIA Working with Truckmaker PACCAR on Self-Driving Technology*. <https://blogs.nvidia.com/blog/2017/03/16/paccar/>. 2017 (cited on page 224).
- [62] Tesla Industries. *Tesla - Semi*. <https://www.tesla.com/semi>. 2019 (cited on page 224).

- [63] Adrian Rosebrock. *Facial landmarks with dlib, OpenCV, and Python*. <https://www.pyimagesearch.com/2017/04/03/facial-landmarks-dlib-opencv-python/>. 2017 (cited on pages 225, 229, 234).
- [64] Adrian Rosebrock. *Detect eyes, nose, lips, and jaw with dlib, OpenCV, and Python*. <https://www.pyimagesearch.com/2017/04/10/detect-eyes-nose-lips-jaw-dlib-opencv-python/>. 2017 (cited on page 225).
- [65] Tereza Soukupová and Jan Čech. “Real-Time Eye Blink Detection using Facial Landmarks”. In: *21st Computer Vision Winter Workshop. Luka Cehovin, Rok Mandeljc, Vito-mir Struc (eds.)* 2016 (cited on pages 226, 228, 242).
- [66] Christos Sagonas et al. “300 Faces In-The-Wild Challenge”. In: *Image Vision Comput.* 47.C (Mar. 2016), pages 3–18. ISSN: 0262-8856. DOI: [10.1016/j.imavis.2016.01.002](https://doi.org/10.1016/j.imavis.2016.01.002) (cited on page 232).
- [67] Wikipedia Contributors. *PID Controller*. [https://en.wikipedia.org/wiki/PID\\_controller](https://en.wikipedia.org/wiki/PID_controller). 2019 (cited on pages 248, 253, 258, 269).
- [68] Wikipedia Contributors. *PID Controller: Pseudocode*. [https://en.wikipedia.org/wiki/PID\\_controller#Pseudocode](https://en.wikipedia.org/wiki/PID_controller#Pseudocode). 2019 (cited on page 251).
- [69] Erle Robotics. *A simplified autopilot*. <https://erlerobot.github.io/erle-gitbook/en/beaglepilot/SimpleAutopilot.html>. 2019 (cited on page 251).
- [70] Wikipedia Contributors. *PID Controller: Manual Tuning*. [https://en.wikipedia.org/wiki/PID\\_controller#Manual\\_tuning](https://en.wikipedia.org/wiki/PID_controller#Manual_tuning). 2019 (cited on page 252).
- [71] Adrian Rosebrock. *Pan/tilt face tracking with a Raspberry Pi and OpenCV*. <https://www.pyimagesearch.com/2019/04/01/pan-tilt-face-tracking-with-a-raspberry-pi-and-opencv/>. 2019 (cited on pages 255, 272).
- [72] Adrian Rosebrock. *OpenCV People Counter*. <https://www.pyimagesearch.com/2018/08/13/opencv-people-counter/>. 2018 (cited on pages 273, 300).
- [73] Adrian Rosebrock. *Multi-object tracking with dlib*. <https://www.pyimagesearch.com/2018/10/29/multi-object-tracking-with-dlib/>. 2018 (cited on page 273).
- [74] Adrian Rosebrock. *Simple object tracking with OpenCV*. <https://www.pyimagesearch.com/2018/07/23/simple-object-tracking-with-opencv/>. 2018 (cited on pages 273, 275).
- [75] Vlad Kiraly. *OpenCV Face Recognition*. <https://www.youtube.com/watch?v=nt3D26lrkho>. 2017 (cited on page 308).

- [76] Adrian Rosebrock. *Measuring size of objects in an image with OpenCV*. <https://www.pyimagesearch.com/2016/03/28/measuring-size-of-objects-in-an-image-with-opencv/>. 2016 (cited on page 340).
- [77] Adrian Rosebrock. *Ordering coordinates clockwise with Python and OpenCV*. <https://www.pyimagesearch.com/2016/03/21/ordering-coordinates-clockwise-with-python-and-opencv/>. 2016 (cited on pages 341, 345).
- [78] Adrian Rosebrock. *Find distance from camera to object/marker using Python and OpenCV*. <https://www.pyimagesearch.com/2015/01/19/find-distance-camera-objectmarker-using-python-opencv/>. 2015 (cited on page 341).
- [79] Adrian Rosebrock. *Measuring distance between objects in an image with OpenCV*. <https://www.pyimagesearch.com/2016/04/04/measuring-distance-between-objects-in-an-image-with-opencv/>. 2016 (cited on page 341).
- [80] Ming-Kuei Hu. "Visual pattern recognition by moment invariants". In: *Information Theory, IRE Transactions on* 8.2 (Feb. 1962), pages 179–187. ISSN: 0096-1000. DOI: [10.1109/TIT.1962.1057692](https://doi.org/10.1109/TIT.1962.1057692) (cited on pages 355, 357).
- [81] Adrian Rosebrock. *10.4: Hu Moments*. <https://gurus.pyimagesearch.com/lessons/hu-moments/>. 2017 (cited on page 357).
- [82] T. Ojala, M. Pietikainen, and T. Maenpaa. "Multiresolution gray-scale and rotation invariant texture classification with local binary patterns". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 24.7 (2002), pages 971–987 (cited on page 357).
- [83] Adrian Rosebrock. *How to Build a Kick-Ass Mobile Document Scanner in Just 5 Minutes*. <https://www.pyimagesearch.com/2014/09/01/build-kick-ass-mobile-document-scanner-just-5-minutes/>. 2014 (cited on pages 363, 367).
- [84] Adrian Rosebrock. *4 Point OpenCV getPerspective Transform Example*. <https://www.pyimagesearch.com/2014/08/25/4-point-opencv-getperspective-transform-example/>. 2014 (cited on page 367).
- [85] Adrian Rosebrock. *Bubble sheet multiple choice scanner and test grader using OMR, Python and OpenCV*. <https://www.pyimagesearch.com/2016/10/03/bubble-sheet-multiple-choice-scanner-and-test-grader-using-omr-python-and-opencv/>. 2016 (cited on page 367).
- [86] Adrian Rosebrock. *Local Binary Patterns with Python & OpenCV*. <https://www.pyimagesearch.com/2015/12/07/local-binary-patterns-with-python-opencv/>. 2015 (cited on page 373).
- [87] Adrian Rosebrock. *10.7: Local Binary Patterns*. <https://gurus.pyimagesearch.com/lessons/local-binary-patterns/>. 2017 (cited on page 373).

- [88] Adrian Rosebrock. *Instance segmentation with OpenCV*. <https://www.pyimagesearch.com/2018/11/26/instance-segmentation-with-opencv/>. 2018 (cited on page 385).
- [89] Adrian Rosebrock. *Semantic segmentation with OpenCV and deep learning*. <https://www.pyimagesearch.com/2018/09/03/semantic-segmentation-with-opencv-and-deep-learning/>. 2018 (cited on page 385).
- [90] Adrian Rosebrock. *Keras Mask R-CNN*. <https://www.pyimagesearch.com/2019/06/10/keras-mask-r-cnn/>. 2019 (cited on page 386).
- [91] Adrian Rosebrock. *OpenCV Face Recognition*. <https://www.pyimagesearch.com/2018/09/24/opencv-face-recognition/>. 2018 (cited on page 386).
- [92] Florian Schroff, Dmitry Kalenichenko, and James Philbin. “FaceNet: A Unified Embedding for Face Recognition and Clustering.” In: *CoRR* abs/1503.03832 (2015). URL: <http://dblp.uni-trier.de/db/journals/corr/corr1503.html#SchroffKP15> (cited on page 386).
- [93] ARM NEON Developer Community. *SIMD ISAs Neon – Arm Developer*. <https://developer.arm.com/architectures/instruction-sets/simd-isas/neon>. 2019 (cited on page 391).
- [94] Wikipedia Contributors. *Single instruction, multiple data (SIMD)*. <https://en.wikipedia.org/wiki/SIMD>. 2019 (cited on page 391).
- [95] ARM NEON Developer Community. *Instruction Sets Floating Point – Arm Developer*. <https://developer.arm.com/architectures/instruction-sets/floating-point>. 2019 (cited on page 391).
- [96] Chuck Pheatt. “Intel&Reg; Threading Building Blocks”. In: *J. Comput. Sci. Coll.* 23.4 (Apr. 2008), pages 298–298. ISSN: 1937-4771. URL: <http://dl.acm.org/citation.cfm?id=1352079.1352134> (cited on page 391).
- [97] Adrian Rosebrock. *YOLO object detection with OpenCV*. <https://www.pyimagesearch.com/2018/11/12/yolo-object-detection-with-opencv/>. 2018 (cited on page 391).
- [98] Wikipedia Contributors. *VideoCore*. <https://en.wikipedia.org/wiki/VideoCore>. 2019 (cited on page 392).
- [99] RaspberryPi.org Forum Contributors. *Pi 4 - full specification of VideoCore 6*. <https://www.raspberrypi.org/forums/viewtopic.php?p=1491509>. 2019 (cited on page 393).
- [100] VC4CL Contributors. *VC4CL - OpenCL implementation running on the VideoCore IV GPU of the Raspberry Pi models*. <https://github.com/doe300/VC4CL>. 2019 (cited on page 393).

- [101] Adrian Rosebrock. *Fast, optimized ‘for’ pixel loops with OpenCV and Python*. <https://www.pyimagesearch.com/2017/08/28/fast-optimized-for-pixel-loops-with-opencv-and-python/>. 2017 (cited on page 403).
- [102] Adrian Rosebrock. *pip install opencv*. <https://www.pyimagesearch.com/2018/09/19/pip-install-opencv/>. 2018 (cited on page 403).

