Facial Recognition-Enabled Lock

Nicholas Franchini · Ryan Gonzalez · Sravani Gandu
NAF345 · RAG580 · SG6176

ME-GY 6933 Raspberry Pi Project
May 14, 2020

ABSTRACT

In this report we detail the design, construction, and application of a locking mechanism that uses facial recognition to validate the identity of permissible users, in tandem with conventional analog input. The machine learning algorithms used to perform facial recognition are implemented on a Raspberry Pi, which uses serial communication to signal an Arduino handling mechanical actuation. Together they provide multiple means of identity validation, either through facial recognition, or keypad input.

INTRODUCTION

One of the most prominent fields of machine learning is computer vision, the detection and recognition of objects and events through photo and video feedback. Relatively recent developments in the fields of AI and consumer technologies have pushed their degrees of accessibility to levels never seen before; now anyone from a hobbyist to a data scientist can claim to have had experience with machine learning, and attempt to develop and experiment with their own models. The past decade in particular has seen incredible leaps in technology within the homes of consumers, leading to the Internet of Things. The emergence of companies such as Ring reflects this.

Despite our limited experience, we too decided to attempt to develop a product, albeit a prototype, that integrates emergent technologies with a product in the realm of home security. While a conventional lock system requires passwords, combinations, and physical keys, we drew inspiration from these emerging technologies and attempted to create a "smart virtual doorman" - a lock that recognizes user faces, and unlocks a door when queried. This is a robust approach that relies not upon the classic keypad or key input but rather the unique facial features of an individual.

Advances not only in hardware but also software make this an application of machine learning that was impossible five years ago. With the new iterations of microcontrollers and microcomputers, and the rapid development of lightweight ML libraries like Tensorflow Lite, we were able to successfully develop a system that recognizes and responds to user input and recognition. We apply a Haar cascade classifier provided by OpenCV to identify users' faces in an image, or a video clip if necessary, to classify users and determine whether they have access based upon recognition models. The main objective of this project is to create a mechanical locking system that uses machine learning to identify the faces of users in order to allow or deny access to the appropriate users.

PROJECT DEVELOPMENT

We have chosen to use machine learning to take image data and process it into meaningful information, effectively determining if a person should be granted access. After preparing the necessary files for our software, we use a Raspberry Pi to run a Python facial recognition script which communicates with an Arduino, which handles the mechanical locking mechanism and mechanical inputs and outputs. This separation of tasks allows for efficient delegation between the boards, and plays to their respective strengths in computing and actuation.

Many avenues were explored in developing the software architecture for this project, which we had little to no prior experience with. Though we originally intended to work with tensorflow lite and a Google Coral Edge TPU, we found that the training and retraining methods for classifying an individual's face, in particular our own, was an incredibly intensive and difficult process that required more computing power and time than we were capable of committing to our project, though ideally we would have worked in depth with Tensorflow Lite and Google's new AutoML API. However, we came across a blog called PyImageSearch that detailed an incredibly lightweight and effective way of constructing our software and rapidly generating foundational software we could modify for our own purposes. We credit them at the end of this report.

Raspberry Pi Software

The first step in developing a working facial recognition model was creating a dataset. Shown in Figure 1 below is the structure of our project folder, including our dataset consisting of approximately twenty unlabeled photos per person, organized by folder. The names of these folders were important, as they were used in generating the labels for our recognition software.

```
$ tree --dirsfirst
.
├──── dataset
│     ├──── nick_franchini
│     │     ├──── IMG_0077.JPG
│     │     └──── ...
│     ├──── ryan_gonzalez
│     │     ├──── ...
│     └──── sravani_gandu
│         ├──── ...
├──── encode_faces.py
├──── encodings.pickle
├──── haarcascade_frontalface_default.xml
└──── pi_face_recognition.py
```

Figure 1: Software architecture of our project file.

This dataset was then input to the `encode_faces.py` file, which processed and encoded images as a `.pickles` file. In short, this encoding script takes every photo in the dataset, and performs a process known as deep metric learning to compute for each photo a list of 128 floating point values. These lists are known as 128-d vectors, and quantify the features of each facial picture. The deep metric learning process is enabled by the **dlib** library, which is wrapped by the **facial_recognition** library, both of which work in tandem with the **OpenCV** library to provide the functionality of our Python scripts. This file's structure and usage case is detailed in Appendix A at the end of the report.

The network trains it's model on these photos in triplets, using two photos known to be correctly labelled and the third randomly picked from an incorrect label. By comparing two correct inputs the machine learning algorithm learns to better correctly identify a specific category (or individual face), while simultaneously learning when an input is incorrect by comparing against the third photo. This lends to a model that is *unusually accurate* at performing predictions with datasets that are by normal standards *incredibly small*, in our case on the order of approximately 10-20 photos per class, or individual. To put this into perspective, the usual dataset is usually on the order of hundreds of labelled photos per class.

Once the `encode_faces.py` file has generated the encodings file (named `encodings.pickle` in this case), we move to the `pi_facial_recognition.py` script which, aptly named, performs facial recognition. This script is instantiated at the beginning of RPi operation, and needs to run continuously as an Arduino would to perform facial recognition. Through a camera module interfaced with the RPi, the script uses the OpenCV library to grab each frame and process the features with the `facial_recognition` library and `encodings.pickle` file.

To reduce strain on the Raspberry Pi during operation, we employ a Haar cascade. This form of detection is capable of recognizing objects at *multiple scales in real time*. Moreover, our Haar cascade in particular came pre-trained by OpenCV, and simply needed to have parameters tweaked by our program during implementation through our encodings file. This framework is incredibly lightweight; the `While(True)` loop handling the bulk of our face detection was approximately *70 lines of code*, and includes extraneous snippets such as whitespace, commenting, and visualization methods for the developer-end of the project. We skip the OpenCV visualization methods in our report, as they are mostly unnecessary in our final product; however, we leave them in for replicability and ease of use.

Another important feature of the `pi_facial_recognition.py` script is handling crosstalk to the Arduino. Through the **PySerial** library, we instantiate a Serial object that enables communication through the serial port - circumventing the need for any GPIO pins during prototyping. Whenever the Raspberry Pi detects *and* recognizes an individual's face through the camera, an integer encoded as bytes is sent to the Arduino,signalling a check for unlock conditions. An important distinction is that this signalling occurs *regardless* of whether the user is attempting to unlock the mechanism. It has no final say in whether the security system should disarm; rather, it simply identifies and alerts the Arduino that a person is at the door. Like our `encode_faces.py` file, the code for our pi_facial_recognition.py file is also detailed with it's usage case in Appendix A at the end of the report.

Arduino Software

The ArduinoIDE was used to develop the code for controlling the Arduino. In this we assign pin constants for a buzzer and the locking mechanism. A keypad has been attached to the Arduino in order to allow access to users who have not been added to the facial recognition dataset. There are five functions within the Arduino code for the following processes: buzzer for incorrect keypad entry, buzzer for correct keypad entry, a function to clear the keypad entry string (Input), a lock function, and an unlock function. Serial communication is used to transmit data between the Raspberry Pi and Arduino.

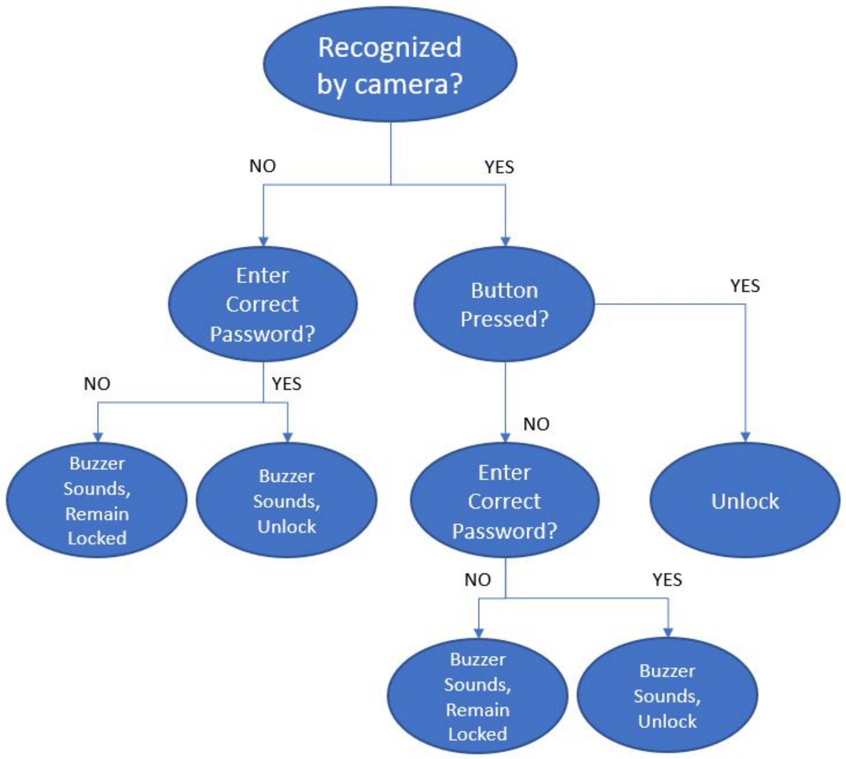

Figure 2: System Access Flowchart

The flowchart above is a visualization of the access process for the system. The Raspberry Pi's facial recognition software is always running and taking in images from the camera. If a face is successfully identified as one of the permitted users, a signal is sent from the Raspberry Pi to the Arduino. If the "START" button on the keypad is pressed while the Raspberry Pi is sending this signal, the Arduino then runs the unlock function which unlocks the mechanism. If no permitted user is identified by the Raspberry Pi, no signal is sent to the Arduino and the mechanism remains locked. During this time, nothing will happen if the button is pressed because the recognition confirmation signal is not being sent by the Raspberry Pi. A user has the option to enter the password on a keypad attached to the Arduino. If the six-digit password is entered correctly, the buzzer sounds three ascending tones and the mechanism is unlocked. If the password is entered incorrectly, the buzzer plays a different series of tones, three in descending order, indicating that the password was incorrect and the mechanism remains locked. If a user has begun entering a password on the keypad, but wants to start over, they can press the “STOP” key in the bottom right corner, which runs the *ClearData* function to clear the entry string.

The void loop of the Arduino program has two primary *if* statements. The first monitors the keypad for button presses and the second monitors the length of the input string. If a button is pressed, the first loop is entered. If the button pressed is the “START” button, the Arduino checks the serial monitor and retrieves any available data. If that data matches the established code for a successful user recognition, “555”, the Arduino initiates the *UnlockDoor* function. If the “STOP” button is pressed, the arduino initiates the *ClearData* function and if any other button is pressed, the password input string is appended with that value. The second *if* statement simply monitors the length of that string and once the string is six characters long, it compares the entered string to the password string. If they match, the *BuzzerRightFunction* and *UnlockDoor* functions are initiated. If the strings do not match, the *BuzzerWrongFunction* and *ClearData* functions are initiated and the lock remains engaged.

Hardware

The recognition-based locking system is conceptually simple and we have already detailed the two primary components, the Raspberry Pi and the Arduino, but there are a few additional components necessary for the mechanical system that are all connected to the Arduino in order to allocate I/O functions away from the Raspberry Pi. A keypad is used for two purposes. The “START” button on the keypad is pressed when a permitted user is present and requesting access. It may be necessary for users who need access but do not have recognition models to unlock the system. In this case, they can use the keypad to enter the unlock password. A buzzer is used for audible feedback when entering a password and to determine if the password was correct or not.

The locking mechanism is represented by a servo motor. Its limits represent the lock and unlock positions. The specific application of the lock, whether it be a door, a filing cabinet, or some other container, is beyond the scope of this project so the lock has intentionally not been designed for a given object.

Pin 10 is connected to the buzzer which uses a 220 Ohm in-line resistor. Pins 2 through 8 are dedicated to reading the four-row by three-column keypad.

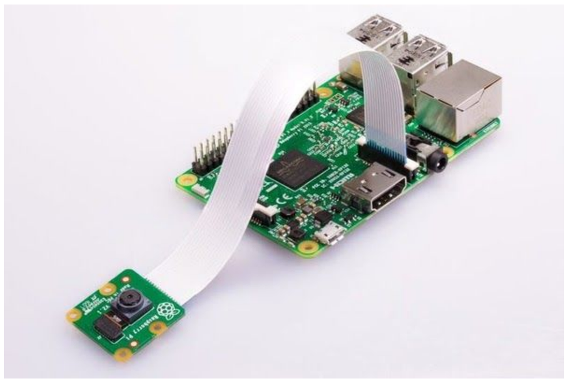
Figure 3: Raspberry Pi Camera Module V2 Installed On Board

A Raspberry Pi Camera Module V2 camera is used for video signal input into the Raspberry Pi. This and a serial communications USB cable are the only devices attached to the Raspberry Pi in our application.

Circuit Diagrams

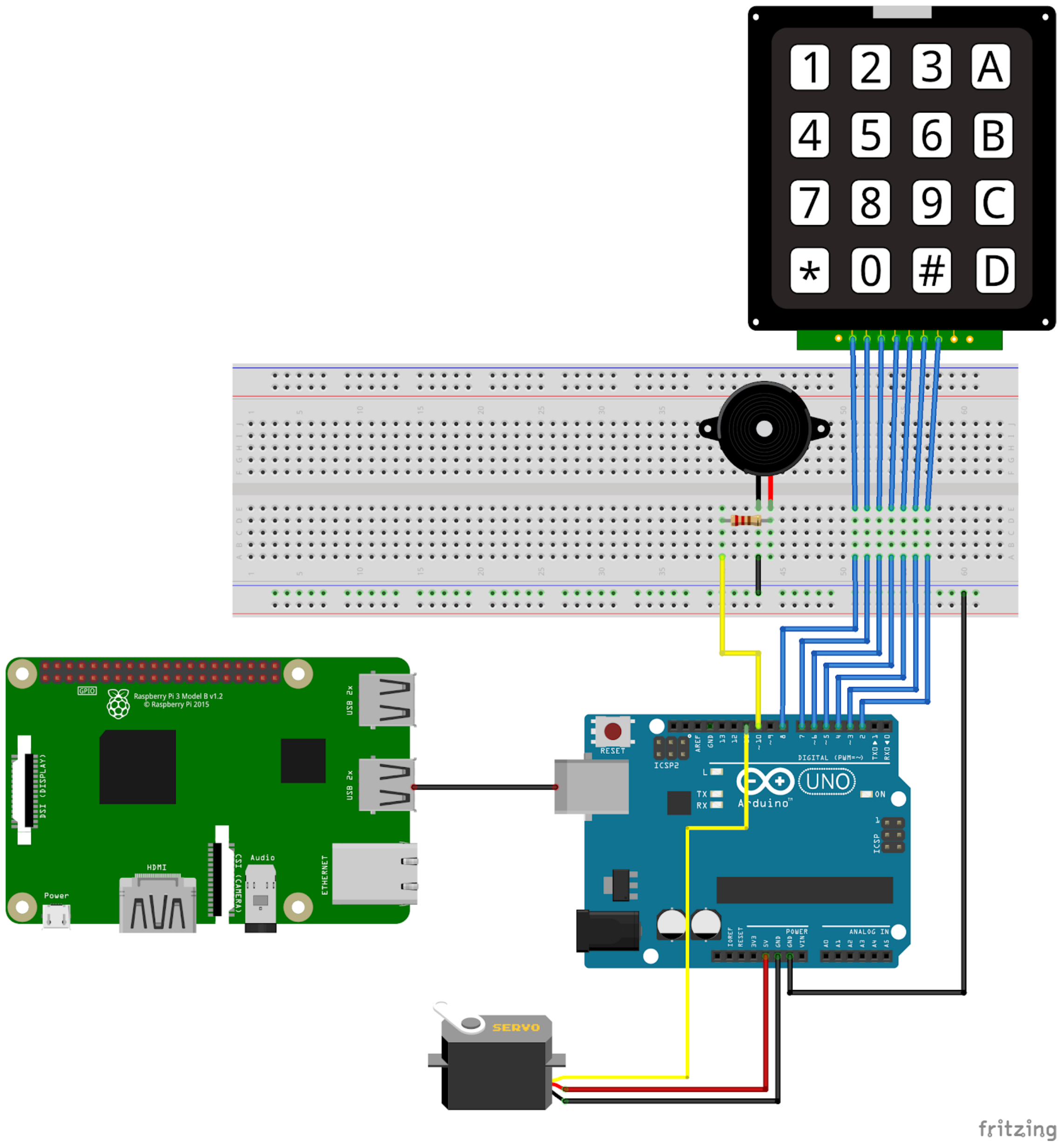

Figure 4: Electrical Layout of System

The diagram above does not depict the Raspberry Pi Camera Module V2 because the model was not available in the Fritzing program. The camera plugs directly into the mezzanine connector on the Raspberry Pi board.

CONCLUSION

We were able to successfully design, program, and build a locking system that employs machine learning to identify faces and validate them against defined permitted user recognition files. The system utilizes Python on a Raspberry Pi to take video input from a Raspberry Pi Camera Module and successfully identify users. It also allows users the option to enter a password to unlock the system, accompanied by audible feedback of the password state. Room for future improvement would include notifying a master user via WhatsApp text message when a user tried to gain access to the system and was not recognized or entered an incorrect password. Additional ability to build new user recognition models via the onboard camera would constitute a more robust user interface without the need to upload images to a recognition file on the Raspberry Pi.

BILL OF MATERIALS

Component	Qty
Raspberry Pi	1
Arduino UNO	1
Raspberry Pi Camera Module V2	1
USB to USB type B serial comm cable	1
12-button keypad	1
Piezoelectric buzzer	1
220 Ohm Resistor	1
Standard Servo motor	1

Electrical wires	13

Appendix A: Raspberry Pi Software

encode_faces.py

```
# USAGE
# When encoding on laptop, desktop, or GPU (slower, more accurate):
# python encode_faces.py --dataset dataset --encodings encodings.pickle
--detection-method cnn
# When encoding on Raspberry Pi (faster, more accurate):
# python encode_faces.py --dataset dataset --encodings encodings.pickle
--detection-method hog

# import the necessary packages
from imutils import paths
import face_recognition
import argparse
import pickle
import cv2
import os

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--dataset", required=True,
      help="path to input directory of faces + images")
ap.add_argument("-e", "--encodings", required=True,
      help="path to serialized db of facial encodings")
ap.add_argument("-d", "--detection-method", type=str, default="cnn",
      help="face detection model to use: either `hog` or `cnn`")
args = vars(ap.parse_args())

# grab the paths to the input images in our dataset
print("[INFO] quantifying faces...")
imagePaths = list(paths.list_images(args["dataset"]))

# initialize the list of known encodings and known names
knownEncodings = []
knownNames = []

# loop over the image paths
for (i, imagePath) in enumerate(imagePaths):
	# extract the person name from the image path
	print("[INFO] processing image {}/{}".format(i + 1,
		len(imagePaths)))
	name = imagePath.split(os.path.sep)[-2]

	# load the input image and convert it from RGB (OpenCV ordering)
	# to dlib ordering (RGB)
	image = cv2.imread(imagePath)
	rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

	# detect the (x, y)-coordinates of the bounding boxes
	# corresponding to each face in the input image
	boxes = face_recognition.face_locations(rgb,
		model=args["detection_method"])

	# compute the facial embedding for the face
	encodings = face_recognition.face_encodings(rgb, boxes)

	# loop over the encodings
	for encoding in encodings:
		# add each encoding + name to our set of known names and
		# encodings
		knownEncodings.append(encoding)
		knownNames.append(name)

# dump the facial encodings + names to disk
print("[INFO] serializing encodings...")
data = {"encodings": knownEncodings, "names": knownNames}
f = open(args["encodings"], "wb")
f.write(pickle.dumps(data))
f.close()
```

pi_face_recognition:

```
### CREDITS GO TO ADRIAN ROSEBROCK FOR TUTORIAL CODE IN RPI FACE
RECOGNITION SOFTWARE
### Tutorial found at:
https://www.pyimagesearch.com/2018/06/25/raspberry-pi-face-recognition/

# USAGE
# python pi_face_recognition.py --cascade
haarcascade_frontalface_default.xml --encodings encodings.pickle

# import the necessary packages (facial recognition)
from imutils.video import VideoStream
from imutils.video import FPS
import face_recognition
import argparse
import imutils
import pickle
import time
import cv2

# imports for Arduino-to-RPi communication
import serial

# instantiate connection to Arduino serial port
ser = serial.Serial('/dev/ttyACM0',9600)
ser.flush()

# construct the argument parser and parse the arguments
# forms the "constructor" for our python code when we run through command
line
ap = argparse.ArgumentParser()
ap.add_argument("-c", "--cascade", required=True,
    help = "path to where the face cascade resides")
ap.add_argument("-e", "--encodings", required=True,
    help="path to serialized db of facial encodings")
args = vars(ap.parse_args())

# load the known faces and embeddings along with OpenCV's Haar
# cascade for face detection
print("[INFO] loading encodings + face detector...")
data = pickle.loads(open(args["encodings"], "rb").read())
detector = cv2.CascadeClassifier(args["cascade"])

# initialize the video stream and allow the camera sensor to warm up
print("[INFO] starting video stream...")
vs = VideoStream(src=0).start()
# vs = VideoStream(usePiCamera=True).start()
time.sleep(2.0)
```

```
# start the FPS counter
fps = FPS().start()

# loop over frames from the video file stream
while True:
    # grab the frame from the threaded video stream and resize it
    # to 500px (to speedup processing)
    frame = vs.read()
    frame = imutils.resize(frame, width=500)

    # convert the input frame from (1) BGR to grayscale (for face
    # detection) and (2) from BGR to RGB (for face recognition)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    # detect faces in the grayscale frame
    rects = detector.detectMultiScale(gray, scaleFactor=1.1,
        minNeighbors=5, minSize=(30, 30),
        flags=cv2.CASCADE_SCALE_IMAGE)

    # OpenCV returns bounding box coordinates in (x, y, w, h) order
    # but we need them in (top, right, bottom, left) order, so we
    # need to do a bit of reordering
    boxes = [(y, x + w, y + h, x) for (x, y, w, h) in rects]

    # compute the facial embeddings for each face bounding box
    encodings = face_recognition.face_encodings(rgb, boxes)
    names = []

    # loop over the facial embeddings
    for encoding in encodings:
        # attempt to match each face in the input image to our known
        # encodings
        matches = face_recognition.compare_faces(data["encodings"],
            encoding)
        name = "Unknown"

        ### THIS IS WHERE WE SEND TO THE ARDUINO###

        # check to see if we have found a match
        if True in matches:
            # find the indexes of all matched faces then initialize a
            # dictionary to count the total number of times each face
            # was matched
            matchedIdxs = [i for (i, b) in enumerate(matches) if b]
            counts = {}

            # loop over the matched indexes and maintain a count for
            # each recognized face face
            for i in matchedIdxs:
                name = data["names"][i]
                counts[name] = counts.get(name, 0) + 1

            # determine the recognized face with the largest number
            # of votes (note: in the event of an unlikely tie Python
            # will select first entry in the dictionary)
            name = max(counts, key=counts.get)

        # update the list of names
        names.append(name)

        # Sends '555' as a string converted to byted when Raspberry Pi
        # detects a face.
        if True in matches:
              ser.write(str("555\n").encode())

    # loop over the recognized faces
    for ((top, right, bottom, left), name) in zip(boxes, names):
        # draw the predicted face name on the image
        cv2.rectangle(frame, (left, top), (right, bottom),
            (0, 255, 0), 2)
        y = top - 15 if top - 15 > 15 else top + 15
        cv2.putText(frame, name, (left, y), cv2.FONT_HERSHEY_SIMPLEX,
            0.75, (0, 255, 0), 2)

    # display the image to our screen
    cv2.imshow("Frame", frame)
    key = cv2.waitKey(1) & 0xFF

    # if the `q` key was pressed, break from the loop
    if key == ord("q"):
        break

    # update the FPS counter
    fps.update()

# stop the timer and display FPS information
fps.stop()
print("[INFO] elasped time: {:.2f}".format(fps.elapsed()))
print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))

# do a bit of cleanup
cv2.destroyAllWindows()
vs.stop()
```

Appendix B: Arduino Software

```
#include <Keypad.h>
#include <Servo.h>
Servo myServo;

int const ServoPin = 11;     // Servo control ouput pin
int const BuzzPin = 10;      // Buzzer control ouput pin
int const Unlock = 70;       // Servo angle to UNLOCK door
int const Lock = 130;        // Servo angle to LOCK door
int const ROW_NUM = 4;       //four rows
int const COLUMN_NUM = 3;    //three columns
int const Length = 7;        // how long the keypad password can be. 6 digits
plus 1 null = 7
char PASSWORD[] = "123456"; // correct password
char Input[Length];
byte index = 0, attempts = 0;

char keys[ROW_NUM][COLUMN_NUM] = {
  {'1','2','3'},
  {'4','5','6'},
  {'7','8','9'},
  {'*','0','#'}
};

byte pin_rows[ROW_NUM] = {8, 7, 6, 5}; //connect to the row pinouts of the
keypad
byte pin_column[COLUMN_NUM] = {4, 3, 2}; //connect to the column pinouts of the
keypad

Keypad keypad = Keypad( makeKeymap(keys), pin_rows, pin_column, ROW_NUM,
COLUMN_NUM );
// Define Functions:
int BuzzerWrongFunction(){
    tone(BuzzPin, 1000);
    delay(200);
    tone(BuzzPin, 700);
    delay(200);
    tone(BuzzPin, 400);
    delay(200);
    noTone(BuzzPin);
    delay(50);
}
```

```
int BuzzerRightFunction(){
    tone(BuzzPin, 400);
    delay(200);
    tone(BuzzPin, 700);
    delay(200);
    tone(BuzzPin, 1000);
    delay(200);
    noTone(BuzzPin);
    delay(50);
}
int ClearData(){
  while (index != 0){
    Input[index--] = 0;
  }
}
int LockDoor(){
    myServo.write(Lock);
    delay(50);
}
int UnlockDoor(){
  myServo.write(Unlock);
  delay(5000);          // five second delay for time to open door
  myServo.write(Lock);
  delay(50);            // lock door at end
}

void setup(){

  pinMode(BuzzPin, OUTPUT);  // Set BuzzerPin as output
  myServo.attach(ServoPin);  // Assign ServoPin to the servo

  Serial.begin(9600);
  myServo.write(Lock);       // Initial door state is locked
}

void loop(){
  char key = keypad.getKey();  // Read input from keypad
  if (key){
    if (key == '*'){ // If 'START' key on keypad is pressed
      if (Serial.available() > 0) { // Check if message is available in serial
com (from Raspberry Pi)
        String data = Serial.readStringUntil('\n');  // Classify incoming
message
        if (data == "555"){ // if RPi is sending '555/n'
```

```
          UnlockDoor();
          delay(20);
        }
      }
    }
    else if (key == '#'){ // If 'STOP' button on keypad is pressed, clear input
string
      ClearData();
    }
    else{ // If any other button on the keypad is pressed, append the input
string with that digit
      Input[index] = key;
      index++;
      tone(BuzzPin, 700);
      delay(50);
      noTone(BuzzPin);
      delay(50);
    }
  }
  if (index == Length-1){ // If there are 6 entries in the input string
    if (!strcmp(Input,PASSWORD)){  //If the input string matches the password
      BuzzerRightFunction();  // Success buzzer sound
      UnlockDoor();
      delay(20);
      ClearData();        // Clear input string data
  }
    else{   // If the input string does not match the password
      ClearData();
      BuzzerWrongFunction();
  }
}
delay(20);
}
```

References

Computer Vision Blog:
https://www.pyimagesearch.com/2018/06/25/raspberry-pi-face-recognition/