Dice Integrity Check
EE4830 Senior Design II
5/7/2021

Sean Watts
Undergraduate, Computer Engineering
University of Wyoming, College of Engineering
swatts5@uwyo.edu

Advisor: Dr. John E. McInroy

Abstract

The goal of this project was to create a device that would automatically verify the fairness of dice. This is accomplished with a servo to spin a clear compartment containing a six-sided die, a camera to take a photo of the roll result, and software to determine what value was rolled. This process is repeated for a number of tests as designated by the user. Once all of the tests are completed, a Chi-squared fairness calculation is performed and the result is displayed on the LCD display. The project was a success, with a finished project capable of examining a die and calculating a Chi-squared result.

Keywords: Dice, Roll, Image Recognition, Camera, Raspberry Pi, Chi

Overview

The project objectives are to be able to verify the integrity of any six-sided die (that is roughly square and fits within the compartment), have a self-contained and user-friendly design, and be reasonably inexpensive. In the Technical Project Description section, a breakdown is given for the project functionality and components that make it happen, both physical and code-wise. The Testing section covers the different tests performed throughout the development process. Packaging discusses the final look and modularity of the project. The section Other Considerations exists for discussing things like project cost and manufacturability. Lessons Learned gives a moment to talk about design choices and situations that would have made the process easier. References provides links to sourced material, and finally the Appendices list graphics and code entries.

Background

A necessary piece of information for this project is an understanding of how Chi-Squared values work. These values are used to give a single numerical result for how well a set of random numbers correspond to the expected results. As the set of numbers approach the values of the expected results, the Chi-Squared value goes to zero. As the numbers diverge from their expected values, the corresponding Chi-Squared value gets larger and larger. Another way to think of a Chi-Squared value is a numeric value for divergence. A value of 0 means no divergence at all, whereas a value above 100 means a lot of divergence. In order to use Chi-Squared values for this design, the expected results were set to be that of an ideal die-- all sides occuring equally likely. Therefore, when comparing against tallies of rolls from testing, the expected value for each side is the number of total tests divided by 6. Given that patterns become more apparent with large sample sizes, it should be understood that large volumes of tests are preferred for calculating a Chi-Squared value. For an exact mathematical definition of Chi-Squared, refer to the Appendices section.

Another essential element of this project is an understanding of the Hough transform. This transform works by examining an image and finding the lines present in that image. More specifically, it takes in an image that has had edge detection performed on it and iterates over the entire image. As it passes over each pixel, it considers each possible angle that a line could pass through that point and calculates a score based on the volume of common edges along that hypothetical line. The Hough transform returns a list of rho-theta pairs describing the most prevalent lines in the image (the ones with the highest score).

Technical Project Description

The functionality of the project is as follows: the user will select a die to test and power on the device by plugging it in. Once the device has powered on, the user can use the 'Go' and 'Stop' buttons to navigate the user prompts and test setup. The user will first specify the number of tests they would like to perform. The user is then asked if they would like to reuse the data from the last test. This allows an individual to test a die again without needing to repeat the setup phase. If they say no, the prompts on the LCD will direct the user to load the die in the clear compartment with its 'one' face facing upwards. The user will sequentially be guided through exposing the camera to the rest of the faces. If the user instead decided to reuse old test data, that section would be skipped and the photos from the last test would be reused for face detection. Old test data should not be reused when testing a new die or when testing in a new environment. After the data has been settled, the user can then push the 'Go' button to start the tests. For every test specified, the device will roll the die by spinning the compartment, take a photo of the rolled die, determine what value is present, and then tally the result. After all of the tests have been

completed, the LCD will display a Chi-squared value to give the user a sense of how fair the die is.

The general structure of this project is a wooden frame with components mounted on it. For a diagram of the build, refer to the Construction Diagram section in the Appendices. The base is a flat white piece of wood that serves as the backdrop for photos taken. Two uprights extend up from either side of the base board. Another piece of wood runs across the top, bridging the two uprights. The camera is mounted on the underside of this piece. Adjacent to the baseboard and one of the uprights is a wedge cut to exactly 55°. This angle is used to hold the box level to the camera while still allowing it to spin about its diagonals. The servo is mounted to this wedge. Extending from the servo gear is a foam stopper. This stopper serves to hold the clear box for the dice. In the opposite corner, an identical wedge sets between the other upright and the top piece of wood. This wedge contains an inset bearing to house a metal rod. This metal rod connects to another foam stopper to hold the opposite corner of the clear box. The Raspberry Pi, the buttons, and the LCD screen are all mounted along the uprights and top piece.

The most integral part of this project is the Raspberry Pi. All of the peripherals, image processing and general functionality were carried out through code on the Pi. Connected directly to the Raspberry Pi were the following elements: the Raspberry Pi Camera, the 360-degree servo, two push-buttons, and a LCD display. Refer to the Raspberry Pi Pinout section in the Appendices to see the pins used and what they do. The general functionality of the code can be seen in the block diagram in figure 2 in the appendices. To efficiently modularize the tasks of the code and also to individually test peripherals, the code was broken into 8 different files. For the commented code files, refer to the Code section in the Appendices. All of the code is written in Python. The code files are broken down as follows:

- program.py

This file defines the overarching functionality of the project. It serves as the framework for all tasks within the design. It specifies all of the button pushes, writes words printed to the LCD display, triggers the camera, and also tells the servo when to operate. Consequently, this code calls functions from many of the following files. Refer to the Code Structure Chart in the Appendices for a graphic on the functionality of this file.

- hough.py

This code contains the isolate function. This piece of code is by far the most computationally intensive part of the whole project. The function takes in an image and attempts to return the most defined square shape within that photo. The core of this code relies on the cv2 library and specifically its HoughLines function [1]. This function examines an image and returns a list of rho-theta pairs corresponding to the most clearly defined lines in the image. The isolate function takes these pairs and examines them to determine which collection of lines best

constitute a square. Once it settles on four lines forming a square, the code calculates the intersection points of these lines. The image is then rotated to make one edge of the square completely flat. A simple crop function is performed on the image to remove all parts of the image outside of the square. The resulting photo is returned as the output of the function.

- rotate.py

rotate.py defines the rotate function. This function enables the Raspberry Pi to operate the servo with precision. It takes in two arguments, 'direction' and 'rotations'. Direction should be set to 1 for clockwise rotation, -1 for counter clockwise, and any other integer to hold in place. The rotations argument gives a number of times the servo will rotate in the given direction past the 0° mark. The code sets a pin to give an output PWM (pulse-width modulated) signal and sets its duty cycle given the direction value. The code also enables another pin to take input as a PWM signal. This pin allows the pi to monitor the angle of the servo. Once all of the pins have been set up, the servo control pin is set high. A counter counts up every time the servo passes the zero degree angle. Once the counter hits the number specified by 'rotations', the servo is turned off again. This code uses functions from the pigpio library [2].

- rotret.py

This code defines a rolling pattern for randomizing the value on the die. It accomplishes this by calling the rotate.py function a number of times in different directions. After tumbling the die for some time, it makes the servo halt the box at an angle perpendicular to the camera for clean image capturing.

- lcdprint.py

This code takes in a string and prints it on the LCD display. If the string is too long to fit all on one line of the LCD, this code truncates the string and sets the remaining letters on the second line.

- I2C_LCD_driver.py

The I2C LCD driver file accomplishes all of the tasks necessary to write characters to the LCD display. It sets up the pins for I2C protocol as well as defining base functions for operating with the LCD display. lcdprint.py is built upon this driver. Most of this file came from an online resource referenced both in the code and the references section [3].

- takephoto.py

As the name suggests, this file contains all of the picture taking code. Given a picture name (and path) this code will turn on the camera, preview the image, and save a photo under that name.

- compare.py

This code handles image comparisons. Given two photos, it first trims them down to the same dimensions. It then performs a histogram equalization and compares the two images. It generates a score for the pair of images based on their similarity. A small value means the two are a good match, whereas a large value means there are many discrepancies.

Other Design Ideas

There were a number of other ideas present in this design process that were scrapped or updated to other plans for one reason or another. One of the big issues of debate was the mechanism for rolling. Early plans involved using a spring loaded base plate that would compress until it released underneath the die, flinging it upwards. This idea was scrapped due to its questionable consistency, a mobile image plane (would require the camera to adjust), and the difficulty of ensuring a spring-loaded mechanism wouldn't deteriorate over time. Another idea was to have a base plate that would elevate a certain distance before dropping, allowing the die to fall for a while as a rolling mechanism. Similar to the last idea, this one was scrapped because the imaging plane for the camera would be variable and the guarantee of a good roll with this mechanism was questionable. While it did resolve any reliance on springs, it still wasn't a great solution. The idea settled on was using a box that would rotate, but there were some different ideas related to this design as well. While simply rotating the box about a horizontal axis would be very easy, this idea was passed up because a cube inside a cubical box rotating such that flat sides meet flat sides didn't seem like a good way to ensure a random result of a roll. One considered workaround was to use some other shape of box or to include unnatural geometry. This was not used because of the difficulty of getting clean, consistent photos through an uneven surface sounded unnecessarily hard. The design settled on was to use a square box, but to have it rotate about its diagonal axis. This way, the die will come in contact with the sides of the box at odd angles rather than flat sides. Additionally, this design maintains the same level for the camera to focus on. Another reason for this design is that it only depends on a servo and a suspending rod of axis for physical components. While this did introduce some difficult problems in accessing the die and allowing the clear box to be removed, it was the most obvious choice.

Camera mounting was another issue that went through several changes. One idea was to have the camera rotate with the box. This one was quickly ruled out for the difficulty of maintaining connections to the camera as it rotates. Another idea was to mount the camera

beneath the die enclosure. The two main issues with this were that the background of the photos would be everything above the device, and also that the camera would pick up the bottom face of the die, not the top. While some sort of cover could have resolved the background issue and a simple lookup table could have resolved the issue of top/bottom face, this isn't a very elegant solution and it requires a lot more set up from the user. A top mounted camera allows the baseplate of the design to serve as the background while also taking photos of the correct side of the die.

Testing

Testing has been an ongoing focus for this project. The image recognition has gone through testing for every step in its development. Before the Raspberry Pi or it's camera shipped, the code was being tested on a desktop with fake, flawless images of dice. As the code improved, testing was changed to work with real photos of dice. The pictures for testing became progressively rougher and rougher to see how much variance the program could handle. Once the parts arrived, the code was moved onto the pi and testing evolved to taking photos of the dice in real time. Similarly, the servo and its corresponding code have gone through stages of testing. Varying voltages and duty cycles have been applied to the servo to find an ideal speed for spinning the box. In the same fashion, the feedback line from the servo has been tested on multiple fronts-- testing for angles of rotation, testing for number of rotations, testing the ability to stop perpendicular to the camera frame. Once the design of the project was completed, the entire system was tested with different colors and styles of dice at varying numbers of tests. At all steps, tests are performed to make sure the code correctly isolates the die and accurately determines what value is on it. During this stage, it was concluded that it is helpful to slide a piece of construction paper over the baseboard to improve contrast between the die and background. For example, a white piece of paper was used when testing dark colored dice, and a black piece of construction paper was used when testing white dice. After verifying desirable functionality on small scale tests, the project was examined at 100 tests per operation. Four different dice of varying styles, colors and qualities were tested at 100 rolls each. The first die tested was a standard white surface die with black dots. After the 100 rolls, its Chi-Squared value was 47.72. The next die tested was a black die with grey written numbers on its sides that came from a table-top dice set. It's Chi-Squared value was 33.56. The next test was with a cheap red die with white dots. This one gave a resulting Chi-Squared value of 38.36. The final test was with a homemade die weighted to favor landing on six. This die had a wooden base color and black dots drawn on it. This final test produced a Chi-Squared value of 176.96. These tests proved the goal functionality of the project. The deliberately unfair die was easy to isolate given it's massive Chi-Squared value, whereas all of the reasonably fair dice were within the same range of one another.

Packaging

The initial hopes for this project were to have the entire frame 3d printed, however this was unfeasible due to time constraints. Instead, the final packaging is the wooden frame as described in the Technical Project Description section. The Raspberry Pi is housed inside of a case attached to the upright of the wooden frame. The wires are kept organized by zip-ties and electrical tape. This project only needs one power wire running to the Raspberry Pi. The HDMI port is left accessible for easy access. This modular and clean design allows the project to easily be picked up, moved and set up in other locations.

Other Considerations

One major goal of this project was to keep it inexpensive. Thanks to careful consideration of parts and pricing, the overall project cost was under the goal cost of $200 dollars.

- BaseBall Display Cube (3-1/8" sq. x 3-1/8" h) by The Container Store: $3.99
- Feedback 360 Degree - High Speed Continuous Rotation Servo by Adafruit: $27.99
- Raspberry Pi 3 Model B+ at Adafruit: $35.00
- Raspberry Pi Camera Board v2 - 8 Megapixels at Adafruit: $29.95
- SunFounder LCD1602 Module with 3.3V Backlight at Amazon: $8.99
- MJTP1230 SPST Tactile Switch by APEM Inc.: $0.20
- GenBasic 80 Piece Female to Female Jumper Wires at Amazon: $5.99
- Fuel Cell Anti-Slosh Safety Foam Tank Baffle Inserts 14x4x6: $8.70
- 3/8″ OD Round Solid Rod 6in by Max-Gain Systems: $1.31
- CanaKit - Premium Case for Raspberry Pi at Best Buy: $9.99
- 100 Count Wire Ties at O'Reilly: $3.99
- 2 in. x 4 in. x 96 in. Prime Whitewood Stud at Home Depot: $7.37

- 1 in. x 2 in. x 8 ft. Eastern White Pine Furring Strip Board at Home Depot: $1.89

Total: $145.36

Another consideration is the manufacturability of this project. For the design given, all of the components are readily purchasable. Similarly, all of the steps to assemble the design are fairly simple and standard processes that could readily be made into an assembly manual.

One major aim of this project is to have a clean and discrete design. In aiming for this goal, the design is easy to move but also reasonably aesthetic. All of the wooden components are cleanly cut and at satisfying square angles. The wires are neatly packaged and without excess length. The modularity of this project also tied nicely into making a visually pleasing design.

A noteworthy concern for this project is the ways it could be used. A company could use this project to advertise a quality guarantee on their dice-- "tested to 1000 rolls" or something of the like. This could set a benchmark for dice quality. Similarly however, this same project could easily be manipulated to produce whatever goal results the user has in mind. In this way, a clearly defined operating method and environment ought to be specified and included with this product.

As for environmental concerns, this design was made to have a minimal environmental impact. All of the components are either renewable resources or are intended to last. One exception to this is the foam stoppers used to hold the dice chamber. During testing these have exhibited signs of breaking down, so in future designs alternatives to these components would be a great consideration.

The safety of this project is another point of interest. All of the powered wires have been carefully covered to prevent electrical shocks. Another safety consideration is the danger of moving parts in this project. One could easily crimp a hand with the rolling dice chamber or get hair caught in the rotating rod. While the servo doesn't seem to be powerful enough to cause any serious damage, warning labels are a good idea to prevent harm to users.

This concludes all of the pertinent considerations for this project.

Lessons Learned

There were many takeaways from this design process. One of the biggest lessons learned is that major projects are much more difficult during a global pandemic. Online wares become unavailable for months on end, shipping times are abysmal, in-person resources are harder to access if still available at all, and all communication with pertinent persons becomes more difficult. The entire project completion process would probably be way easier and less frustrating during a normal calendar year. Another consideration would be to reign in the difficulty of the project. Simplifications could have been made to the project that would have substantially

reduced the workload. For instance, a simpler rolling mechanism would've reduced the project complexity quite a bit. Similarly, a larger group size could have offset workload substantially.

References

1. *OpenCV*, OpenCV Team, 26 Feb. 2021, opencv.org/.

2. “Python Interface.” *Pigpio Library*, abyz.me.uk/rpi/pigpio/python.html.

3. Campbell, Scott. “How to Setup an I2C LCD on the Raspberry Pi.” *Circuit Basics*, 15 July 2020, www.circuitbasics.com/raspberry-pi-i2c-lcd-set-up-and-programming/.

Appendices

Schedule:

December 10th: Finish some level of image recognition code

February 12th: Purchase parts, get image recognition code running on Pi

March 5th: Get the Pi working with all peripherals

April 11st: Complete project structure

April 15th: Present working demo

April 26th: Finish presentation video

May 7th: Final Report Submitted

Parts List:

- BaseBall Display Cube (3-1/8" sq. x 3-1/8" h)
- Feedback 360 Degree - High Speed Continuous Rotation Servo
- Raspberry Pi 3 Model B+
- Raspberry Pi Camera Board v2 - 8 Megapixels
- SunFounder LCD1602 Module with 3.3V Backlight
- MJTP1230 SPST Tactile Switch by APEM Inc. x2
- GenBasic 80 Piece Female to Female Jumper Wires
- Fuel Cell Anti-Slosh Safety Foam Tank Baffle Inserts
- 3/8″ OD Round Solid Rod 6in
- CanaKit - Premium Case for Raspberry Pi

- 100 Count Wire Ties
- 2 in. x 4 in. x 96 in. Prime Whitewood Stud
- 1 in. x 2 in. x 8 ft. Eastern White Pine Furring Strip Board

Chi-Squared Equation

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

The mathematical definition of Chi-Squared. O_i is the observed value, E_i is the expected value, and X^2 is the resulting Chi-Squared value.

Construction Diagram

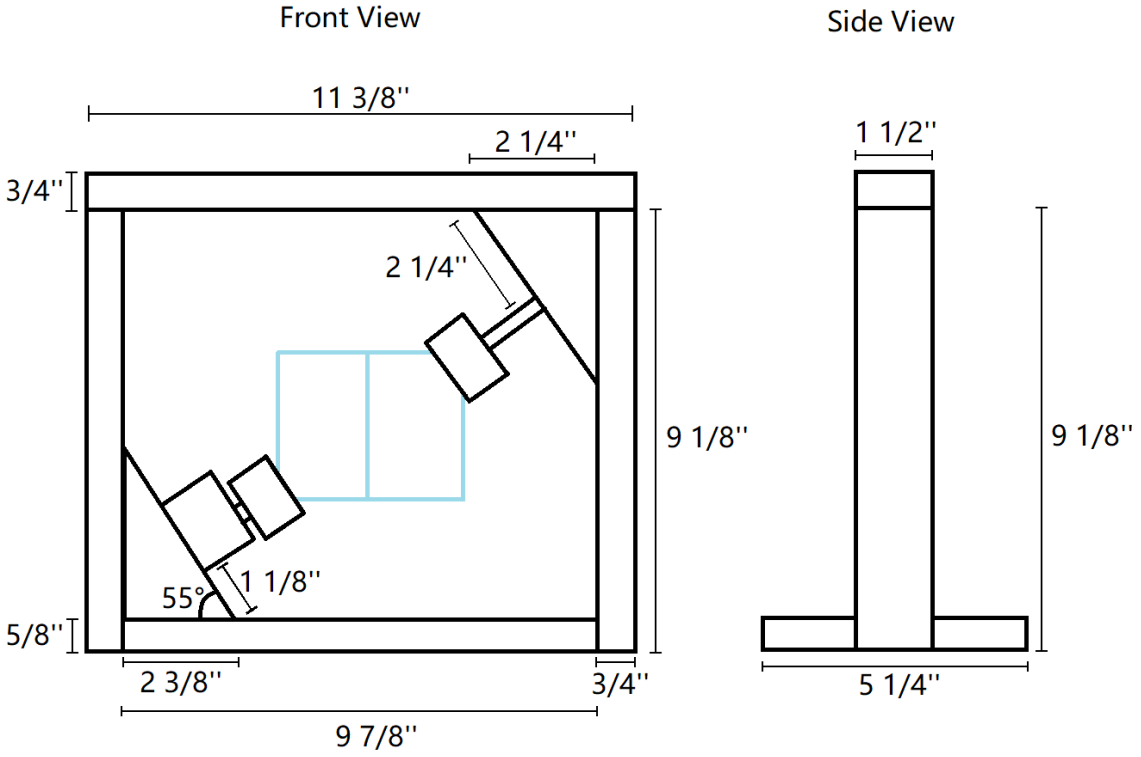

Image 1) a depiction of all important measurements for the construction of the project. All measurements in inches.

Code Structure Chart

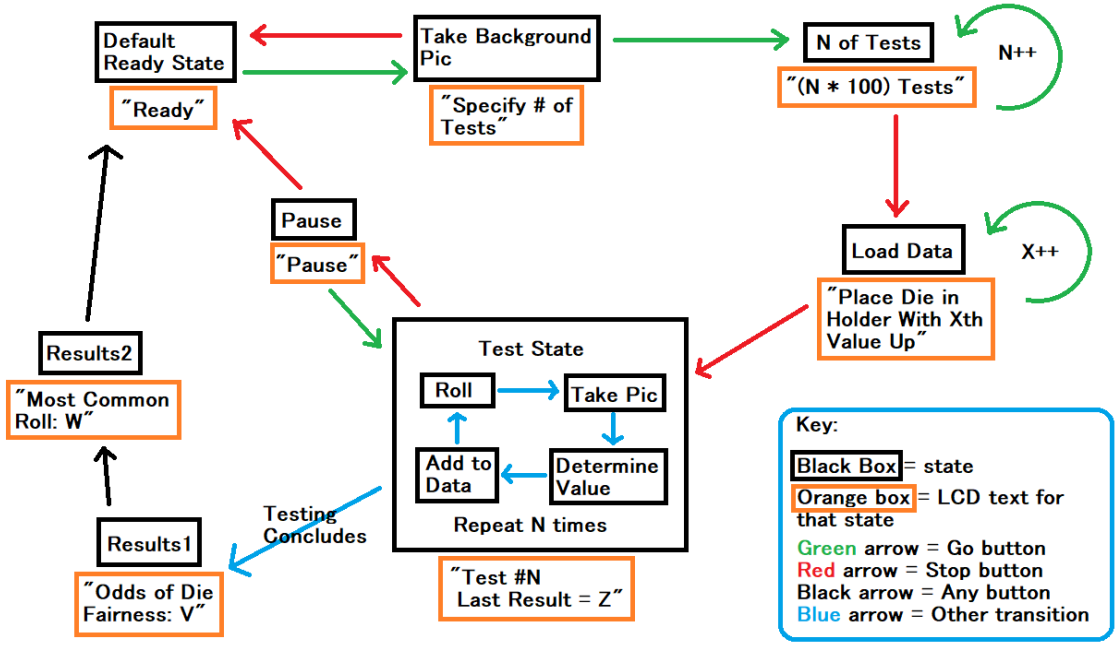

Image 2) A flowchart for the general code functionality.

Raspberry Pi Pinouts

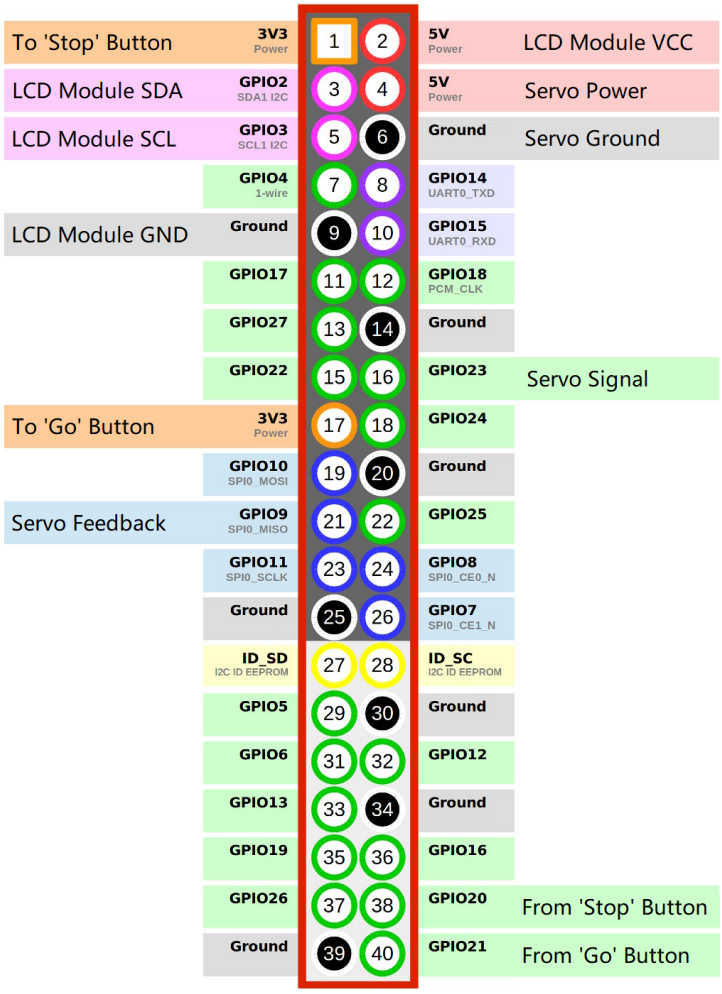

Image 3) This figure shows the pinouts of the Raspberry Pi in relation to which pins are used in this project and to what effect.

Code Entries

All code is written in Python. Some of the code extends past the boundaries of the image, so word wrap automatically carries these statements to the next line. Any words after a pound symbol (#) are comments used for explaining the code or allowing for quick and easy adjustments.

compare.py

```
import cv2
import numpy as np
import os

# this function takes two images, compares them and generates a score
# for how similar they are.
def compare(image1, image2):
	# convert both to grayscale, get their dimensions
	image1 = cv2.cvtColor(image1,cv2.COLOR_BGR2GRAY)
	x1dim, y1dim = image1.shape

	image2 = cv2.cvtColor(image2,cv2.COLOR_BGR2GRAY)
	x2dim, y2dim = image2.shape

	# these four while loops trim down the images to be the same dimensions
	while x1dim > x2dim:
		image1 = image1[1:x1dim,0:y1dim]
		x1dim -=1
		if x1dim > x2dim:
			image1 = image1[0:x1dim-1,0:y1dim]
			x1dim -=1
		else: break
	while x1dim < x2dim:
		image2 = image2[1:x2dim,0:y2dim]
		x2dim -=1
		if x1dim < x2dim:
			image2 = image2[0:x2dim-1,0:y2dim]
			x2dim -=1
		else: break
	while y1dim > y2dim:
		image1 = image1[0:x1dim,1:y1dim]
		y1dim -=1
		if y1dim > y2dim:
			image1 = image1[0:x1dim,0:y1dim-1]
			y1dim -=1
		else: break
	while y1dim < y2dim:
```

```
		image2 = image2[0:x2dim,1:y2dim]
		y2dim -=1
		if y1dim < y2dim:
			image2 = image2[0:x2dim,0:y2dim-1]
			y2dim -=1
		else: break
	# a crop factor here is used to trim off some of the noisy edges
	cropx = int(x1dim/10)
	cropy = int(y1dim/10)
	image1 = image1[cropx:x1dim-cropx,cropy:y1dim-cropy]
	image2 = image2[cropx:x2dim-cropx,cropy:y2dim-cropy]

	minimum = 1000
	maximum = 0
	result = 0
	# perform histogram equalization on both images
	im1 = cv2.equalizeHist(image1)
	im2 = cv2.equalizeHist(image2)
	# loop over the two images, add to result when incongruities are found
	for i in range (x1dim-2*cropx):
		for j in range (y1dim-2*cropy):
			i1 = int(im1[i][j])
			i2 = int(im2[i][j])
			if i1 - i2 > 100:
				result += 1
			if i1 > maximum:
				maximum = i1
			if i1 < minimum:
				minimum = i1
	# return the calculated score
	return result
```

hough.py

```
import cv2
import numpy as np
import time
# this code attempts to find a die in the image and returns that image cropped around the die.
def isolate(img):
	# convert to grayscale and perform canny edge detection
	gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
	v = np.median(gray)
	lower = int(max(0,(0.67 * v)))
	upper = int(min(255, (1.33 * v)))
	edges = cv2.Canny(gray,50,130,apertureSize = 3) #defaults = 50, 150 ap = 3
	cv2.imwrite('edges.jpg', edges)
	course = 100
```

```
	# while a square has not been found repeat these steps
	while True:
		# get all of the most prominent lines in the image
		lines = cv2.HoughLines(edges,1,np.pi/180,course, min_theta = 0, max_theta = 3.1)
		x1dim, y1dim = edges.shape
		counter = 0
		erdis = min(x1dim, y1dim)/ 10

		erdeg = 0.15
		vals = []
		bucket = []

		# bad photo
		if course == 0:
			print("bad photo")
			return -1
		# if we didn't find enough lines for a box, relax the strictness of the
houghlines
		# function via the course variable and try again
		course -= 20
		if lines is None:
			continue
		if len(lines) < 4:
			continue

		# look through the lines, sort the lines into containers with roughly the same
theta (angle)
		for thing in lines:
			rho, theta = thing[0]
			if bucket:
				i = 0
				valset = 0
				for rotat in bucket:
					i = i + 1
					if (rotat[0][1]+erdeg > theta) and (rotat[0][1]-erdeg <
theta):
						for tab in range(len(rotat)):
							if (rotat[tab][0]+erdis > rho) and
(rotat[tab][0]-erdis < rho) and (valset == 0):
								rotat[tab][2] = rotat[tab][2] + 1
								valset = 1
								break
							if (rotat[tab][0] > rho) and (valset == 0):
								rotat.insert(tab, [rho, theta, 0])
								valset = 1
								break
						if (valset == 0):
							rotat.append([rho, theta, 0])
							valset = 1
							break
					elif( i == len(bucket) and valset == 0):
						bucket.append([[rho, theta, 0]])
						valset = 1
```

```
                                        break
                        else:
                                bucket.append([[rho, theta, 0]])

                done = 0
                # check to see if any combination of the lines in the containers form a nice
square
                if(len(bucket) > 1):
                        for b1 in range(len(bucket)):
                                if len(bucket[b1]) > 1:
                                        for b2 in range(b1 + 1, len(bucket)):
                                                if len(bucket[b2]) > 1:
                                                        angv = abs(bucket[b1][0][1] -
bucket[b2][0][1])
                                                        if(angv < np.pi/2 + erdeg and angv >
np.pi/2 - erdeg):
                                                                for d1 in range(len(bucket[b1])):
                                                                        for d2 in range(d1 + 1,
len(bucket[b1])):
                                                                                for d3 in
range(len(bucket[b2])):
                                                                                        for d4 in
range(d3 + 1, len(bucket[b2])):
                                                                                                dist1 =
abs(bucket[b1][d1][0] - bucket[b1][d2][0])
                                                                                                dist2 =
abs(bucket[b2][d3][0] - bucket[b2][d4][0])
                                                                                                xdist =
max(dist1 * 0.1, dist2 * 0.1)

if(dist1 < 300 and dist2 < 300 and dist1 + xdist > dist2 and dist1 - xdist < dist2):
                                                                                                # only
hits this condition when 4 lines are found that form a desired square

done = 1

# add lines to the finished list

vals.append(bucket[b1][d1])

vals.append(bucket[b1][d2])

vals.append(bucket[b2][d3])

vals.append(bucket[b2][d4])

break
                                                                                        if(done == 1):
                                                                                                break
                                                                                if(done == 1):
                                                                                        break
                                                                        if(done == 1):
                                                                                break
```

```
                                    if(done == 1):
                                           break
                if(done == 1):
                       break

        if (len(vals) == 4):
                       break;

# calculate some values to make following calculations easier
r1 = vals[0][0]
sin1 = np.sin(vals[0][1])
cos1 = np.cos(vals[0][1])
r2 = vals[1][0]
sin2 = np.sin(vals[1][1])
cos2 = np.cos(vals[1][1])
r3 = vals[2][0]
sin3 = np.sin(vals[2][1])
cos3 = np.cos(vals[2][1])
r4 = vals[3][0]
sin4 = np.sin(vals[3][1])
cos4 = np.cos(vals[3][1])

# calculate intersections of the four lines
d = (sin3 * cos1 - sin1 * cos3)
fx1=(r1 * sin3 - r3 * sin1) / d
fy1=(cos1 * r3 - cos3 * r1) / d

d = (sin4 * cos1 - sin1 * cos4)
fx2=(r1 * sin4 - r4 * sin1) / d
fy2=(cos1 * r4 - cos4 * r1) / d

d = (sin3 * cos2 - sin2 * cos3)
fx3=(r2 * sin3 - r3 * sin2) / d
fy3=(cos2 * r3 - cos3 * r2) / d

d = (sin4 * cos2 - sin2 * cos4)
fx4=(r2 * sin4 - r4 * sin2) / d
fy4=(cos2 * r4 - cos4 * r2) / d

# find outer edges of the newly found box
xmax = int(max(fx1, fx2, fx3, fx4))
xmin = int(min(fx1, fx2, fx3, fx4))
ymax = int(max(fy1, fy2, fy3, fy4))
ymin = int(min(fy1, fy2, fy3, fy4))

# find center of box
ycent = (ymax - ymin) / 2 + ymin
xcent = (xmax - xmin) / 2 + xmin
axis = (xcent, ycent)
xdim, ydim = edges.shape

angle = 0
# find angle of offset of box
```

```
        smallest = min(vals[0][1], vals[1][1], vals[2][1], vals[3][1])
        for item in [vals[0][1], vals[1][1], vals[2][1], vals[3][1]]:
                while item > smallest + 0.11:
                        item -= np.pi/2
                angle += item
        angle = angle / 4
        # rotate the entire image to make the box oriented
        rot_mat = cv2.getRotationMatrix2D(axis, angle*57.2958, 1.0)
        result = cv2.warpAffine(img, rot_mat, img.shape[1::-1], flags=cv2.INTER_LINEAR)


        temsin = np.sin(angle)
        temcos = np.cos(angle)
        coord = [[fx1, fy1], [fx2, fy2], [fx3, fy3], [fx4, fy4]]

        xmax2 = 0
        xmin2 = 10000
        ymax2 = 0
        ymin2 = 10000
        # calculate new corner locations after the rotation
        for item in coord:
                val = (item[1] - ycent) * temcos - (item[0] - xcent)*temsin + xcent
                val2 = (item[1] - ycent) * temsin + (item[0] - xcent)*temcos + ycent
                if val > xmax2: xmax2 = val
                if val < xmin2: xmin2 = val
                if val2 > ymax2: ymax2 = val2
                if val2 < ymin2: ymin2 = val2

        # crop the image and return the result
        crop = result[int(ymin2):int(ymax2),int(xmin2):int(xmax2)]
        cv2.imwrite('final.jpg',crop)
        return crop
```

I2C_LCD_Display

```
import smbus
from time import sleep
I2CBUS = 1
ADDRESS = 0x27

# this code defines classes for easy use of the LCD module.
# code borrowed from:
# https://www.circuitbasics.com/raspberry-pi-i2c-lcd-set-up-and-programming/
class i2c_device:
        def __init__(self, addr, port=I2CBUS):
                self.addr = addr
                self.bus = smbus.SMBus(port)

        def write_cmd(self, cmd):
                self.bus.write_byte(self.addr, cmd)
                sleep(0.0001)
```

```
	def write_cmd_arg(self, cmd, data):
		self.bus.write_byte_data(self.addr, cmd, data)
		sleep(0.0001)

	def write_block_data(self, cmd, data):
		self.bus.write_block_data(self.addr, cmd, data)
		sleep(0.0001)

	def read(self):
		return self.bus.read_byte(self.addr)

	def read_data(self, cmd):
		return self.bus.read_byte_data(self.addr, cmd)

	def read_block_data(self, cmd):
		return self.bus.read_block_data(self.addr, cmd)


class lcd:
		def __init__(self):
			self.lcd_device = i2c_device(ADDRESS)

			self.lcd_write(0x03)
			self.lcd_write(0x03)
			self.lcd_write(0x03)
			self.lcd_write(0x02)

			self.lcd_write(0x20 | 0x08 | 0x00 | 0x00)
			self.lcd_write(0x08 | 0x04)
			self.lcd_write(0x01)
			self.lcd_write(0x04 | 0x02)
			sleep(0.2)

		def lcd_strobe(self, data):
			self.lcd_device.write_cmd(data | 0b00000100 | 0x08)
			sleep(0.0005)
			self.lcd_device.write_cmd((data & 0b11111011) | 0x08)
			sleep(0.0001)

		def lcd_write_nibble(self, data):
			self.lcd_device.write_cmd(data | 0x08)
			self.lcd_strobe(data)

		def lcd_write(self, cmd, mode=0):
			self.lcd_write_nibble(mode | (cmd & 0xF0))
			self.lcd_write_nibble(mode | ((cmd << 4) & 0xF0))

		def lcd_write_char(self, charvalue, mode=0):
```

```
                self.lcd_write_nibble(mode | (charvalue & 0xF0))
                self.lcd_write_nibble(mode | ((charvalue << 4) & 0xF0))

        def lcd_display_string(self, string, line=1, pos=0):
                if line == 1:
                        pos_new = pos
                elif line == 2:
                        pos_new = 0x40 + pos
                elif line == 3:
                        pos_new = 0x14 + pos
                elif line == 4:
                        pos_new = 0x54 + pos

                self.lcd_write(0x80 + pos_new)

                for char in string:
                        self.lcd_write(ord(char), 0b00000001)

        def lcd_clear(self):
                self.lcd_write(0x01)
                self.lcd_write(0x02)

        def backling(self, state):
                if state == 1:
                        self.lcd_device.write_cmd(0x08)
                elif state == 0:
                        self.lcd_device.write_cmd(0x00)

        def lcd_load_custom_chars(self, fontdata):
                self.lcd_write(0x40)
                for char in fontdata:
                        for line in char:
                                self.lcd_write_char(line)
```

lcdprint.py

```
import I2C_LCD_driver
from time import *

# this function takes in a string and prints it to the LCD. It
# breaks strings into two lines as needed. No overflow detection.
def lcdprint(string):

        mylcd = I2C_LCD_driver.lcd()
        # if the string is too long for one line, put it on two
        if len(string) > 16:
```

```
            string2 = string[16:]
            mylcd.lcd_display_string(string)
            mylcd.lcd_display_string(string2, 2)
      else:
            mylcd.lcd_display_string(string)
```

program.py

```
import cv2
import numpy as np
import os
import RPi.GPIO as GPIO
from time import sleep
from lcdprint import lcdprint
from takephoto import takephoto
from hough import isolate
from compare import compare
from rotate import rotate
from rotret import rotret
# this code contains all of the main program functionality. This file defines
# all of the testing, LCD prompts, camera and servo control.

# setup pushbuttons
GPIO.setmode(GPIO.BOARD)
GPIO.setwarnings(False)
GPIO.setup(40, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)
GPIO.setup(38, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)

path = 'data/'
libpath = 'lib/'
# start of program
print("Starting Program")
lcdprint("Ready, press Go button")
GPIO.wait_for_edge(40, GPIO.RISING)

# get the number of tests
lcdprint("Specify # of tests")
counter = 1
temp = 0
GPIO.wait_for_edge(40, GPIO.RISING)

lcdprint(str(counter) + " tests")

GPIO.add_event_detect(40, GPIO.RISING)
GPIO.add_event_detect(38, GPIO.RISING)
# loop until the user is happy with the # of tests
```

```
while temp == 0:
	sleep(1)
	if GPIO.event_detected(40):
		counter += 1
		lcdprint(str(counter) + " tests")
		#sleep(1)
	if GPIO.event_detected(38):
		temp += 1
		#sleep(1)

# decide if old data is to be kept or overwritten
lcdprint("Reuse Test data?")
reuse = 0
while temp > 0:
	if GPIO.event_detected(40):
		reuse = 1
		temp = 0
	if GPIO.event_detected(38):
		reuse = 0
		temp = 0

# Get data on all sides of the die
if reuse == 0:
	i = 1
	for i in range(6):
		valid = 0
		while(valid == 0):
			# ask user for a given side
			if i == 0:
				string = "Insert die with 1st face up"
			if i == 1:
				string = "Insert die with 2nd face up"
			if i == 2:
				string = "Insert die with 3rd face up"
			if i > 2:
				string = "Insert die with " + str(i+1) + "th face up"
			lcdprint(string)
			loc = path + str(i+1) + '.jpg'
			sleep(1)
			GPIO.wait_for_edge(40, GPIO.RISING)
			# take photo and find the image
			takephoto(loc)
			img = cv2.imread(loc)
			val = isolate(img)
			# make sure photo is good
			if type(val) == np.ndarray:
				valid = 1
		# in the library create 3 copies at 90 degree rotations
```

```
            for j in range(4):

                  thename = libpath + str(i+1) + "." + str(j * 90) + '.jpg'
                  cv2.imwrite(thename, val)
                  val = np.rot90(val)

# begin testing
lcdprint("Press Go to test, exit to quit")
GPIO.wait_for_edge(40, GPIO.RISING)
# keep track of number of rolls for each side
tally = [0,0,0,0,0,0]
# loop for the number of tests specified
for t in range(counter):
      valid = 0
      while(valid == 0):
            # rotate the die, take the photo, find the die in the image
            rotret()
            takephoto('current.jpg')
            testim = cv2.imread('current.jpg')
            val = isolate(testim)
            if type(val) == np.ndarray:
                  valid = 1

      # figure out what face is up
      scores = [np.Infinity, np.Infinity, np.Infinity, np.Infinity, np.Infinity,
np.Infinity]
      z = 0
      lcdprint("Test number: " + str(t))
      # test the new pic against everything in the library
      for entry in sorted(os.listdir('lib/')):
            string = libpath + entry
            filename = str(string)
            img = cv2.imread(filename)
            result = compare(val, img)
            zz = int(z/4)
            print('Testing ', int(z/4) + 1, result)
            # keep lowest score
            if result < scores[zz]:
                  scores[zz] = result
            z += 1

      decision = 1
      best = scores[0]
      for spot in range(len(scores)):
            if scores[spot] < best:
                  best = scores[spot]
                  decision = spot + 1
      # the lowest score among every comparison is the decided result.
```

```
	tally[decision-1] += 1
	print('The result is: ', decision)
	lcdprint("The result is: " + str(decision))

chi = 0
exp = counter/6.0
# calculate the corresponding chi value
for tab in tally:
	temp = (tab - exp) * (tab - exp)
	chi += temp/exp

print('The Chi value is: ', chi)
lcdprint("The chi value is: " + str(chi))
```

rotate.py

```
import RPi.GPIO as GPIO
from time import sleep
import pigpio
# set up input and output pins
GPIO.setmode(GPIO.BOARD)
GPIO.setwarnings(False)
GPIO.setup(16, GPIO.OUT)
pwm=GPIO.PWM(16, 50)
pwm.start(0)
# create a class to handle PWM input signals
class PWM_read:
	def __init__(self, pi, gpio):
		self.pi = pi
		self.gpio = gpio
		self._high_tick = None
		self._p = None
		self._hp = None
		self.cnt = 0
		self.rot = 0
		self._cb = pi.callback(gpio, pigpio.EITHER_EDGE, self._cbf)
	# create interupt (callback) function
	def _cbf(self, gpio, level, tick):

		if level == 1:
		# test for valid pwm data
			if self._high_tick is not None:
				self._p = pigpio.tickDiff(self._high_tick, tick)
			self._high_tick = tick
		elif level == 0:
			if self._high_tick is not None:
```

```
                        self._hp = pigpio.tickDiff(self._high_tick, tick)
                # test for readable pwm
                if (self._p is not None) and (self._hp is not None):
                        tmp = 100.0 * self._hp/self._p
                        # count up if a specific mark is hit
                        if self._hp == 500 and self.cnt == 0:
                                self.cnt = 1
                                self.rot += 1
                        if self._hp == 60 and self.cnt == 1:
                                self.cnt = 0

        def cancel(self):
                self._cb.cancel()

# this function takes in a direction and rotation. direction = 1 for clockwise,
# -1 for counterclockwise, any other number for no rotation. Rotation is # of passes
# by the marked angle on the servo.
def rotate(direction, rotations):
        #set up input and output pins
        pi = pigpio.pi()
        HALL = 9
        pi.set_mode(HALL, pigpio.INPUT)
        pi.set_pull_up_down(HALL, pigpio.PUD_DOWN)

        # duty = 6 for clockwise, 12 for cclockwise
        if direction == 1:
                duty = 6.4 #6.5 for slower speed
        elif direction == -1:
                duty = 8 #7.8 for slower speed
        else:
                duty = 7.25 # stop value

        # start servo turning
        GPIO.output(16, True)
        pwm.ChangeDutyCycle(duty)

        # start reading pwm
        p1 = PWM_read(pi, HALL)
        test = p1.rot
        sleep(0.1)
        while p1.rot < rotations:
                # check for rotations, update accordingly
                if (test < p1.rot):
                        print(p1._hp)
                        test = p1.rot
                temp = pi.read(HALL)
        # stop functionality
        pi.stop()
```

```
	GPIO.output(16, False)
	pwm.ChangeDutyCycle(0)
```

rotret.py

```
from rotate import rotate
from time import sleep
# this function defines a set of rotations to roll the die and stop at a flat angle.
def rotret():
	# rotate counter clockwise for 3 rotations
	rotate(-1, 3)
	sleep(1)
	# rotate clockwise for 3 rotations
	rotate(1, 3)
	sleep(1)
	# rotate counter clockwise for 3 rotations
	rotate(-1, 3)
	sleep(0.1)
	# halt the servo
	rotate(0,0)
```

takephoto.py

```
from picamera import PiCamera
from time import sleep
# This method takes a photo and saves it at at the given path/name
def takephoto(path):
	# enable the camera and start focusing
	camera = PiCamera()
	camera.start_preview()
	# give 5 seconds to focus and adjust
	sleep(5)
	# take and save photo
	camera.capture(path)
	# close peripherals
	camera.stop_preview()
	camera.close()
```