ARIZONA STATE UNIVERSITY

HONORS THESIS

DIY Supercube

Author:
Joseph HALE

Supervisor:
Dr. Robert HEINRICHS

A thesis submitted in fulfillment of the requirements
for the degree of Honors Thesis in Software Engineering

in the

Fulton Schools of Engineering
Barrett, The Honors College

November 2, 2021

Declaration of Authorship

I, Joseph HALE, declare that this thesis titled, "DIY Supercube" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

ARIZONA STATE UNIVERSITY

Abstract

Dr. Robert Heinrichs

Barrett, The Honors College

Honors Thesis in Software Engineering

DIY Supercube

by Joseph HALE

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too…

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

Declaration of Authorship iii

Abstract vii

Acknowledgements ix

1 Introduction 1
- 1.1 The Advent of Smart Cubes . . . 1
- 1.2 Obstacles to Adoption . . . 1
- 1.3 Purpose of this Thesis . . . 2
- 1.4 Thesis Overview . . . 2

2 Background 3
- 2.1 A Brief History of the Rubik's Cube . . . 3
- 2.2 The Anatomy of a Rubik's Cube . . . 3
  - 2.2.1 Algorithm Notation . . . 4
  - 2.2.2 The Laws of the Cube . . . 4
- 2.3 Speedsolving . . . 4
  - 2.3.1 The Rise of Smart Cubes . . . 4
  - 2.3.2 Competitions . . . 4
    - The World Cube Association . . . 4
    - Competition Regulations . . . 4

3 State of the Art 5
- 3.1 Introduction . . . 5
- 3.2 Commercial Smartcubes . . . 5
  - 3.2.1 Giiker Cube . . . 5
  - 3.2.2 Go Cube . . . 6
  - 3.2.3 Gans 356i . . . 6
- 3.3 Academia . . . 7
  - 3.3.1 Computer Vision . . . 7
    - Sticker Color Classification . . . 7
    - Measuring a Face's Angle of Rotation . . . 7
    - Classification of Single Moves and Entire Move Sequences . . . 8
  - 3.3.2 Magnetic Resonance . . . 8
  - 3.3.3 Muscle-Tracking Armband . . . 9
- 3.4 Other Relevant Research . . . 9
  - 3.4.1 Sound . . . 9
  - 3.4.2 Radio Frequency Identification (RFID) . . . 9

3.4.3 Off-Axis Magnetic Angle Sensors . . . 10
3.5 Research Questions . . . 10

4 The Protocol **13**
4.1 Introduction . . . 13
4.2 Requirements . . . 13
4.2.1 Signal-to-Noise Ratio . . . 13
How to Read a Spectrogram . . . 14
Key Observations from the Spectrogram . . . 14
4.2.2 Tone Distinctiveness . . . 14
4.2.3 Frequency Response Range of Consumer Hardware . . . 17
4.2.4 Human Auditory Range . . . 18
4.3 Specification . . . 18
4.3.1 Representing the Cube's Current State . . . 18
4.3.2 Tracking Face Turns . . . 19
4.3.3 Conveying State Through Sound . . . 19
4.4 Alternative Protocol . . . 20
4.5 Summary . . . 21

5 The Receiver **23**
5.1 Introduction . . . 23
5.2 Creating Synthetic Audio . . . 23
5.2.1 Representing the Audio Protocol . . . 24
5.2.2 Representing the Rubik's Cube . . . 25
5.2.3 Creating Synthetic Audio for an Arbitrary Algorithm . . . 25
5.3 Decoding the Synthetic Audio . . . 27
5.3.1 Computing the Spectrogram . . . 27
The Brief Overview of the Math behind the Spectrogram . . . 28
5.3.2 Extracting the Dominant Component Frequencies . . . 29
5.3.3 Translating Component Frequencies to Centerpiece States . . . 30
5.3.4 Extracting Move Sequences from Centerpiece State Sequences . . . 32
5.3.5 Full Example: Extracting Moves from Synthetic Audio . . . 33
5.4 Creating Realistic Audio . . . 34
5.5 Decoding Realistic Audio . . . 34
5.5.1 Fine-Tuning the Threshold . . . 35
5.5.2 Filtering through Similar Peak Frequencies . . . 37
5.5.3 Ignoring Noise when Extracting Move Sequences . . . 38
5.5.4 Optimizing Algorithm Parameters . . . 39
5.6 Summary . . . 39

6 The Transmitter **41**
6.1 Introduction . . . 41
6.2 Requirements . . . 41
6.2.1 Prospects of Miniaturization . . . 41
6.2.2 Precision of Tone Generation . . . 41
6.2.3 Responsiveness to Face Turns . . . 42
6.2.4 Signal-to-Noise Ratio . . . 42
6.3 Hardware Selection . . . 42

6.3.1 Minimizing Sound Obstruction 42
6.4 Prototyping 42

7 Evaluation **43**
7.1 Compatibility with Standard Speedcubes 43
7.2 Move Tracking Accuracy 43
7.3 Move Tracking Granularity 43
7.4 Competition Legality 43

8 Conclusion **45**
8.1 Summary 45
8.2 Limitations 45
8.3 Ideas for Future Research 45

A Longer Code Snippets **47**
A.1 Spectrogram Generation for Figure 5.5 47
A.2 Spectrogram Generation for Figure 5.7 48

Bibliography **49**

List of Figures

3.1 The internal components of the Giiker Cube . . . 6
3.2 The internal components of the Go Cube [11] . . . 6
3.3 Gans 356i Teardown . . . 7
3.4 The IM3D technology as used in the Cube Harmonic . . . 8
3.5 Off-Angle Magnetic Sensor Calculations . . . 10

4.1 Background Noise of a Quiet Room while Speedsolving . . . 15
4.2 Distinctiveness of Two Increasingly Similar Tones . . . 17
4.3 "A typical frequency response curve for a microphone." [29] . . . 18
4.4 The four possible rotations of a face of a Rubik's Cube [32] . . . 19

5.1 Centerpiece State to Frequency Mapping . . . 24
5.2 A simple abstraction of a Rubik's Cube . . . 25
5.3 Generating audio for any Rubik's Cube algorithm . . . 26
5.4 Example Audio Generation for a Rubik's Cube Algorithm . . . 26
5.5 Spectrogram of the synthetic audio created in Section 5.2.3 . . . 27
5.6 Function to compute the spectrogram of a .wav file . . . 28
5.7 Spectrogram from Figure 5.5 with a threshold at 85% of the maximum strength of the component frequencies. . . . 29
5.8 Extracting dominant frequencies from one time step of audio . . . 29
5.9 Example peak frequency decoding at a specific time step . . . 30
5.10 Converting peak frequencies to centerpiece states . . . 30
5.11 Example conversion of peak frequencies to states . . . 31
5.12 Function to list the cube's state at each time step . . . 31
5.13 Example listing of states over time . . . 31
5.14 Code to extract move sequences from state sequences . . . 32
5.15 Example move sequence extraction . . . 33
5.16 Full Example: Audio generation and move extraction . . . 33
5.17 Spectrogram of a realistic audio sample . . . 34
5.18 An 85% threshold applied to realistic audio . . . 35
5.19 Dominance of background noise between moves . . . 35
5.20 Updated version of threshold computation in Figure 5.8a . . . 36
5.21 Fine-Tuned Threshold . . . 36
5.22 Updated version of state extraction in Figure 5.8b . . . 37
5.23 Example: Refined conversion of peak frequencies to states . . . 37
5.24 Updated version of move detection in Figure 5.14b . . . 38
5.25 Example: Refined move sequence extraction . . . 39

List of Tables

2.1 The number of each type of Rubik's Cube cubie 3

4.1 Sample frequencies for encoding a Rubik's Cube's state 20

List of Abbreviations

TPS	**T**urns **P**er **S**econd
WCA	**W**orld **C**ube **A**ssociation

Physical Constants

Speed of Light $c_0 = 2.997\,924\,58 \times 10^8\,\mathrm{m\,s^{-1}}$ (exact)

List of Symbols

a	distance	m
P	power	W ($\mathrm{J\,s^{-1}}$)
ω	angular frequency	rad

For/Dedicated to/To my…

Chapter 1

Introduction

1.1 The Advent of Smart Cubes

Speedsolving, the sport of solving twisty puzzles like the Rubik's Cube as fast as possible, has seen a resurgence of popularity since the early 2000s. [**TODO**] Over the past two decades many advances in cube technology have produced ever higher performing puzzles.

Recently, the speedcubing community has seen the entrance of smart cubes, special versions of a Rubik's Cube built around hardware that can connect to a mobile device over Bluetooth. These smart cubes have sparked a wave of excitement with the vast opportunities they offer for automatic turn tracking, performance analysis, personalized improvement feedback, and networked competition.

1.2 Obstacles to Adoption

While a revolutionary idea, smart cubes still face several obstacles to widespread adoption.

- *Cost*: Smart cubes can cost up to eight times as much as a comparable non-smart speedcube. [1]
- *Performance*: Existing smart cubes turn slower than comparable non-smart cubes. [**TODO**]
- *Reliability*: Many smart cube owners report inability to connect the smart cube to a mobile device and missed/inaccurate turn tracking. [**TODO**]
- *Regulation*: Current competition rules ban the use of electronics during timed solves, thus banning the use of smart cubes. There is no foreseeable change to this rule. [**TODO**]

As a result of these obstacles, many speedcubers refrain from purchasing a smart cube, despite expressing significant interest in the opportunities smart cubes offer.

[1] For example, one popular budget speedcube, the Moyu Weilong, costs only $5, while the cheapest smartcube, the Giiker Cube, starts at $40. [**TODO**]. On the higher end, a premium speedcube, like the Gans 356 XS, retails for just over $60 while a premium smartcube, like the GoCube, retails for over $100. [**TODO**]

Furthermore, all current smartcubes have been specifically built for the primary purpose of providing move-tracking functionality. There is no existing way to automatically track the moves of a standard, "non-smart" speedcube.

1.3 Purpose of this Thesis

The primary goal of this thesis is to create a proof-of-concept for a smart cube design that can enable a speedcuber to use his/her personal favorite cube, while still having all the benefits of a smart cube.

In other words, this thesis seeks to answer the following question:

Is it possible to track the face turns of a standard, "non-smart" speedcube in a non-destructive, competition-legal way?

1.4 Thesis Overview

TODO give an overview of the rest of the Thesis document.

Chapter 2

Background

TODO Each chapter starts with a paragraph that briefly outlines the purpose of each of the sections.

2.1 A Brief History of the Rubik's Cube

In 1974, Erno Rubik, a Hungarian professor of architecture, sought to help his students visualize space in three dimensions. To that end, he created a special cube whose faces could independently rotate around all three physical axes [1]. When he added colored stickers to further aid in visualizing the movements, Mr. Rubik realized he had created a new puzzle. He patented his cube in 1975, [2] and since then over 450 million units have been sold [3], allowing an estimated 1 in 7 humans on earth to try their hand at solving it [4].

Since then, the cube has been the subject of academic research, competition, leisure, and cultural iconography.

2.2 The Anatomy of a Rubik's Cube

The Rubik's Cube, like a standard geometric cube, has six faces, all of which are squares and positioned at right angles to each other. When solved, each of these faces has a single, unique color.

The Rubik's Cube is further subdivided into a 3x3x3 arrangement of smaller "cubies" such that each face consists of nine individual colored stickers/tiles. There are three different types of cubies: centers, edges, and corners. Each type of cubie is distinguished by the number of unique colors it binds together into a single physical unit.

TABLE 2.1: The number of each type of Rubik's Cube cubie

Type of Cubie	Number of Colors	Count per Rubik's Cube
Center	1	6[a]
Edge	2	12
Corner	3	8

[a] All six center pieces are attached a single core

Each of the six center cubies are also attached to a common core which allows them to rotate freely, but fixes their position relative to each other. As such, the single color of each center cubie is also the color shared by the corresponding face when the entire cube is solved.

2.2.1 Algorithm Notation

TODO

2.2.2 The Laws of the Cube

TODO Describe the basic concepts of group theory that stipulate what positions are and aren't legal. It might also be fun to discuss the derivation of the 43 quintillion possible positions on the cube.

2.3 Speedsolving

TODO

2.3.1 The Rise of Smart Cubes

TODO

2.3.2 Competitions

The World Cube Association

TODO

Competition Regulations

According to WCA regulation 2i, "While competing, competitors must not use electronics or audio equipment (e.g. cell phones, MP3 players, dictaphones, additional lighting) apart from the Stackmat timer or stopwatch." [5]

Chapter 3

State of the Art

3.1 Introduction

This chapter seeks to provide a comprehensive summary of the existing approaches to tracking the face turns of a Rubik's Cube. The most widely used solutions to date are found in Commercial Smartcubes (3.2), but significant research has also been carried out into Computer Vision based solutions (3.3.1). Other researchers have also explored the use of magnetic resonance and a muscle-tracking armband (3.4)

This chapter will also explore a selection of wireless communication techniques that at the time of writing have not been applied to the challenge of tracking the moves of a Rubik's Cube. Specifically, this chapter will review the potential usage of sound (3.4.1), Radio Frequency Identification (RFID) (3.4.2), and Off-Axis Magnetic Angle Sensors (3.4.3).

Finally, this chapter will close by detailing the specific research questions this thesis will seek to answer (3.5).

3.2 Commercial Smartcubes

Commercial Smartcubes are special Rubik's Cubes built around sensors that can detect face turns and transmit that information over Bluetooth. Some models can also measure and transmit data about the cube's orientation.

At the time of writing, there are four major smartcubes on the market: the Giiker Cube, the Go Cube, the Rubik's Connected (which is powered by GoCube technology) and the Gans 356i. This section will discuss the internal components of each of these cubes that provide this move-tracking functionality.

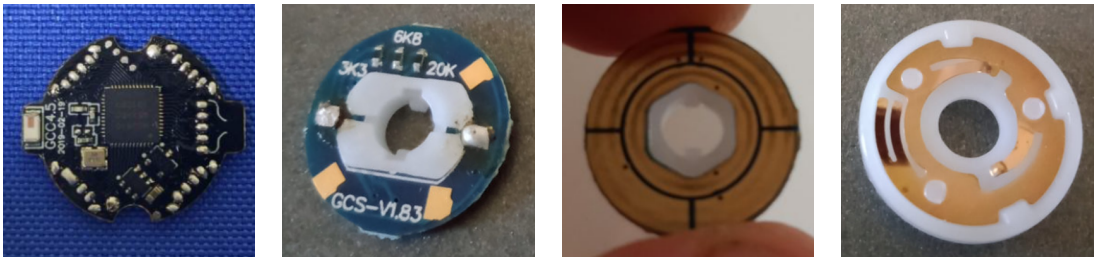
3.2.1 Giiker Cube

The Xiaomi Giiker Cube was released in September 2018 making it the first commercial smartcube on the market [6]. This "Supercube" as it was branded, used a relatively simple system for tracking the cube's movements. The core of the cube is built around a small circuit board with a microcontroller that measures the cube's movements and

a Bluetooth antenna that transmits those moves wirelessly (Figure 3.1a). The microcontroller detects each face turn by reading the voltage drop across a small circuit embedded within each center cap (Figure 3.1b). The center cap circuit controls its output voltage by using a copper brush to change between four separate electric paths, three different resistors and ground, as each face rotates (Figure 3.1d). [7]

FIGURE 3.1: The internal components of the Giiker Cube

(A) Internal Board [7] (B) Center Cap Board [8] (C) Center Cap Pads [8] (D) Center Cap Brush [8]



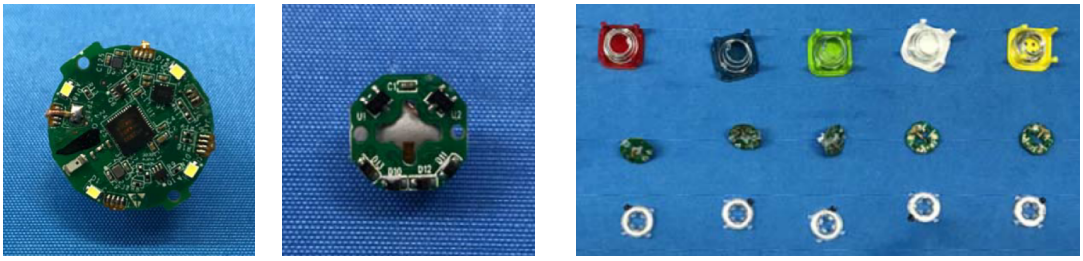
3.2.2 Go Cube

Announced on Kickstarter in June 2018, Patricula's GoCube was the first smartcube to include a gyroscope that would track a Rubik's Cube's orientation in addition to the face turns applied to it. [9] Like the Giiker Cube before it, the GoCube's core contains a small circuit board with the main electronics including a microcontroller, Bluetooth antenna, and the added gyroscope (Figure 3.2a). Though the teardown pictures from the Go Cube's FCC filing aren't particularly clear, it appears that the cube registers face turns similarly to the Giiker Cube: by producing a voltage drop via changing which one of the four resistors shown across the bottom of the center cap board in Figure 3.2b is in series with the circuit.

GoCube also serves as the underlying technology for the Rubik's Connected, the official smartcube from The Rubik's Company. [10]

FIGURE 3.2: The internal components of the Go Cube [11]

(A) Internal Board (B) Center Cap Board (C) Full Center Cap Teardown

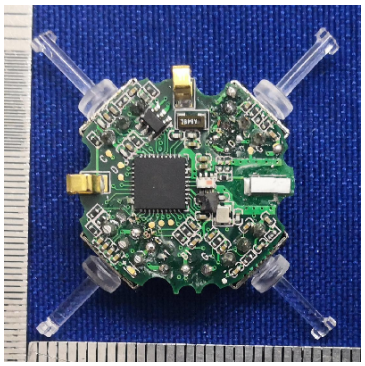


3.2.3 Gans 356i

Released in July 2019, the Gans 356i was the first commercial smartcube produced by a traditional speedcube manufacturer. [12] While the Gans 356i also uses a microcontroller to process the face turns and Bluetooth to transmit the move data, it tracks moves

not through changing resistors in and out of a circuit, but via six plastic rods that connect the outer center caps to internal rotary encoders (Figure 3.3).

FIGURE 3.3: The Gans 356i Cube's internal components. [13]



3.3 Academia

In addition to the various commercial smartcubes, many academic research projects have involved some element of tracking the state/face turns of a Rubik's Cube.

This section summarizes the current state of academic research into using computer vision, magnetic resonance, and a muscle-tracking armband to track the state of a Rubik's Cube.

3.3.1 Computer Vision

Computer Vision refers to the "field of Artificial Intelligence (AI) that enables computers and systems to derive meaningful information from digital images, videos, and other visual inputs." [14] Since human manipulation of a Rubik's Cube is a physical, observable process, Computer Vision algorithms could be developed to extract face turn information from videos of Rubik's Cube solutions.

This section summarizes some of the relevant research in this area, including computer vision algorithms capable of extracting individual sticker colors from video, measuring the angle of rotation of a specific face, and detection of entire face turns and face turn sequences.

Sticker Color Classification

In 2015, Jay Hack, a graduate student studying Computer Science at Stanford developed a neural network capable of recognizing the colors of a Rubik's Cube face from video in various lighting conditions. His algorithm could classify frames within 7 milliseconds with 92% accuracy. [15]

Measuring a Face's Angle of Rotation

In 2019, OpenAI et al. published a viral video of a robot hand that had taught itself to solve a Rubik's Cube. While the final, most successful version of the robot hand's

software used a Giiker Cube to obtain the current rotational state of the cube, OpenAI et al. also researched the viability of tracking a Rubik's Cube's position using only computer vision. Their most successful vision-only algorithm measured only the rotation angle of the top-most face on the Rubik's Cube and assumed significant hardware requirements: a modified sticker set for the Rubik's Cube, a well-lit environment, three strategically positioned RBG Basler cameras, and a neural network trained on "a pool of optimizer nodes, each of which uses 8 NVIDIA V100 GPUs and 64 CPU cores". At peak performance, their vision-only algorithm's average error (the difference between the predicted face angle and the actual face angle) was 15.92°, nearly three times the 5.90° average error of the hardware-based face angle measurement. [16]

Classification of Single Moves and Entire Move Sequences

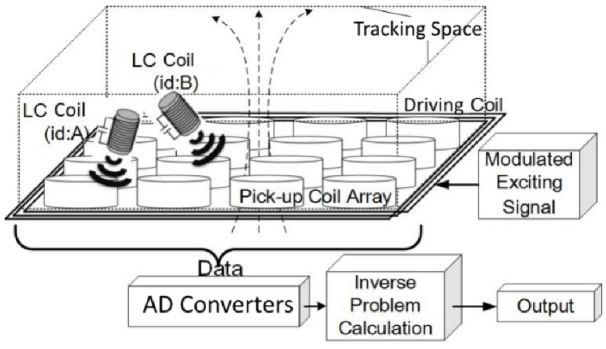
In 2020, Junshen Kevin Chen, Wanze Xie, and Zhouheng Sun, graduate Computer Science students at Stanford created the DeepCube dataset consisting of over 20,000 videos of Rubik's Cube face turns with consistent lighting and backgrounds. They also built a neural network to classify the videos with the face turn they contained. Their best performing model only made "one mistake every 15 moves" which corresponds to a 93.3% accuracy. [17]

3.3.2 Magnetic Resonance

In 2018, Maria Mannone et al. used the IM3D magnetic 3D motion tracking technology introduced by Huang et al. [18] to track the state of a Rubik's Cube across various movements for the purpose of generating a sequence of musical chords. This approach to turn tracking requires a special array of magnetic coils as shown in Figure 3.4a and the installation of "multiple small, light-weight, wireless markers (LC coils) with unique IDs" (a process that requires permanent modifications to the cube as evidenced by the damaged plastic in Figure 3.4b). Mannone et al. reported no issues with mistakes in this move tracking technology. [19]

FIGURE 3.4: The IM3D technology as used in the Cube Harmonic

(A) IM3D System Architecture [18]



(B) IM3D trackers in a Rubik's Cube [20]

3.3.3 Muscle-Tracking Armband

In 2017, Richard Polfreman and Benjamin Oliver researched ways to use the face turns of a Rubik's Cube as controls for a music synthesizer. They explored the use of a muscle-tracking armband (specifically the Myo Armband) to track the human solver's finger movements while manipulating the cube. However, since "the Myo moved a little when 'cubing'", they ultimately found greater success with a computer vision based tracking solution similar to those discussed in 3.3.1.

3.4 Other Relevant Research

Finally, there are a number of research papers/commercial products that seek to transmit data in highly-constrained environments that are potentially relevant to the challenge of tracking the turns of a Rubik's Cube.

This section discusses some of these potential alternate move tracking mediums, specifically sound, RFID, and off-angle magnetic rotation sensors.

3.4.1 Sound

Sound is another communication medium that could be leveraged to track the moves of a Rubik's Cube. In 2015, Jonas Michel, a researcher at The University of Texas at Austin documented his exploration of the viability of creating an "acoustic modem" to transmit an arbitrary sequence of bits using sound. He observed that "as commercial off-the-shelf (COTS) smartphones become more powerful, it is worthwhile to revisit the use of sound as a medium for aerial digital device-to-device communications." [21]

Indeed, since many speedcubers practice by timing solves on a microphone-equipped smartphone or laptop[1], sound is a promising alternative communication medium to existing Bluetooth-based smartcubes.

3.4.2 Radio Frequency Identification (RFID)

Radio Frequency Identification (RFID) is a wireless technology often used in supply-chain systems [24] based on individual tags that can transmit a fixed set of information to a nearby reader. [25]. In 2018, Genovesi et al. proposed a rotary encoder based on RFID that produced angle measurements accurate to within 3° after a calibration that "must only be done once (when the sensor is put in place) if the distance between the reader and the tag does not change." [24]

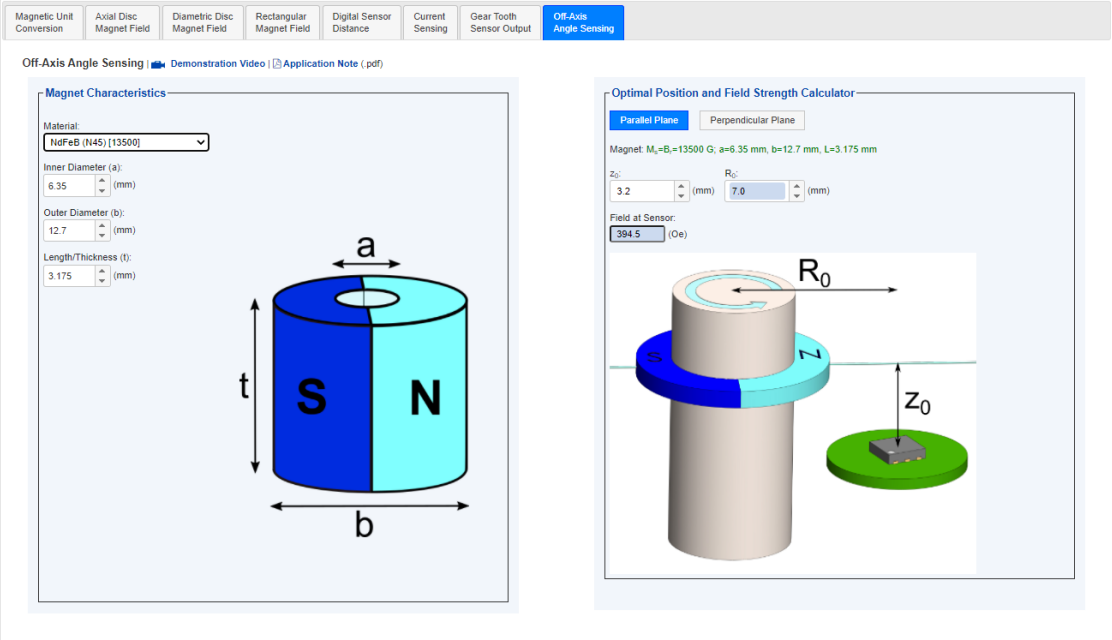
While this approach may not be particularly useful for tracking the moves of human speedcubers since their movement of the cube would require constant re-calibration, it could be used in robotics based applications desiring to track the ongoing state of the cube.

[1] One of the highest rated cubing timers on Android, Twisty Timer, has over 100,000 downloads on the Google Play Store [22]. By comparison, SpeedSolving.com, the central forum for speedcubing related discussion, has about 43,000 members [23]

3.4.3 Off-Axis Magnetic Angle Sensors

Another unique type of rotary encoder is an off-angle magnetic rotational sensor like the ones produced by NVE Corporation. [26] In the context of a Rubik's Cube, a properly sized diametric ring magnet could be fastened to the inner screw below the center cap of each face of the cube and the resulting optimal position for the magnetic sensor would stay within the walls of the center cap as shown by the the z_0 and R_0 values in Figure 3.5 which are smaller than the 8 mm distance between the center screw and the inner wall of the center cap in a standard-sized cube like the Gans 356.

FIGURE 3.5: Off-Angle Magnetic Sensor Calculations [27]



3.5 Research Questions

While there are many effective solutions for tracking the moves of a Rubik's Cube if the cube is built for that purpose, there does not yet exist a comparably effective technique for tracking the moves of a standard, "non-smart" Rubik's Cube. As shown above, significant effort has been spent researching and developing solutions that can leverage the Bluetooth transmitters and camera sensors of consumer grade smartphones and laptops with varying degrees of success. However, little to no research has been carried out exploring the viability of using the microphones readily available on the same devices for this purpose.

Thus, this thesis will seek to answer the overarching question from Chapter 1 *"Is it possible to track the face turns of a standard, "non-smart" speedcube in a non-destructive, competition-legal way?"* by detailing a proof-of-concept for a sound-based smartcube design created in response to the following questions:

1. **Feasibility on Consumer Hardware**: *What are the constraints for a sound-based smart cube design compatible with consumer-grade microphones like those found in common smartphones and laptops?*
2. **Compatibility with Standard Speedcubes**: *How could such a sound-based smart cube design be deployed within a standard, "non-smart" speedcube without requiring permanent modifications to the original cube?*
3. **Move Tracking Accuracy**: *How could such a sound-based smart cube design track the face turns of a Rubik's Cube with high accuracy?*
4. **Move Tracking Granularity**: *How could such a sound-based smart cube design record the time spent executing each individual face turn of a Rubik's Cube?*
5. **Competition Legality**: *How could such a sound-based smartcube design comply with competition regulations prohibiting the use of electronics while performing a competitive solve?*

Chapter 4

The Protocol

4.1 Introduction

Designing a sound-based protocol for tracking the moves of a Rubik's Cube requires great care. Lots of things produce sound that could interfere with the protocol: people talking, machines operating, nature stirring, and so forth. Furthermore, the structure of the Rubik's Cube itself imposes stringent physical constraints on the size of any components used to produce the sounds used in the protocol.

This chapter first seeks to clearly detail the specific constraints that must be considered when designing a sound-based protocol for tracking the moves of a Rubik's Cube (4.2). From there, a specific protocol that meets these constraints will be proposed in Section 4.3. This proposed protocol will then be contrasted with an alternative option in Section 4.4. Finally, an overview of the plan to implement the proposed protocol in a proof-of-concept will be given in Section 4.5

4.2 Requirements

This section will detail the constraints required for designing a protocol for tracking the moves of a Rubik's Cube.

These constraints include a sufficiently strong signal-to-noise ratio (4.2.1), sufficient distinctiveness between tones (4.2.2), the frequency response range of consumer hardware (4.2.3), and the human auditory range (4.2.4).

4.2.1 Signal-to-Noise Ratio

Sound is an easily accessible, and therefore noisy, medium of communication. As a result, any data transmission protocol based on sound must be resilient to the presence of additional noise unrelated to the signal being transmitted. For a sound-based protocol to be effective, its tones must be easily distinguishable from background noise.

For the purpose of this thesis, "background noise" will be considered the ambient noise present in a quiet room when a speedcuber is actively solving a Rubik's Cube. Measurement of the background noise was carried out by using the Android app Spectroid [28] to create live visualizations of the background noise in a household bedroom (a typical place for a speedcuber to practice solving the cube) while solving three types of

speedcubes selected based on their "noisiness". A fourth visualization of just the ambient noise in the bedroom (i.e. while no cubes were being solved) was also recorded as a control. These visualizations are shown in Figure 4.1.

How to Read a Spectrogram

These visualizations are formally known as an audio "spectrogram", and show both the specific frequencies present in each audio sample and their intensity.

The graph on top shows the current and maximum strengths of each frequency present in the background audio with the yellow and red lines respectively. The x-axis measures the specific frequencies in Hz, while the y-axis shows the intensity (aka loudness) of those frequencies in dB.

The lower graph is the actual spectrogram of the background audio. It shares the same x-axis as the top graph, but its y-axis instead measures time with time = 0 at the bottom of the graph and time = now at the top. Here, the brighter colors represent a more intense (aka louder) frequency with the exact key shown on the left side of the graph. As such, this lower graph can be considered a "bird's eye view" of the top graph over time.

Key Observations from the Spectrogram

As shown in Figure 4.1a, the predominant background noises in a quiet bedroom are the tones between 80Hz - 500Hz which reach a max strength of -54dB.

In contrast, Figures 4.1b and 4.1c show that solving a speedcube creates additional noise in the frequency ranges from 1000Hz - 20000Hz, with Figure 4.1d showing particular strength from the noisy QiYi cube with the frequencies in the 1000Hz - 10000Hz range reaching up to -30dB.

Thus, a sound-based move tracking protocol must account for the following items in order to achieve an adequate signal-to-noise ratio:

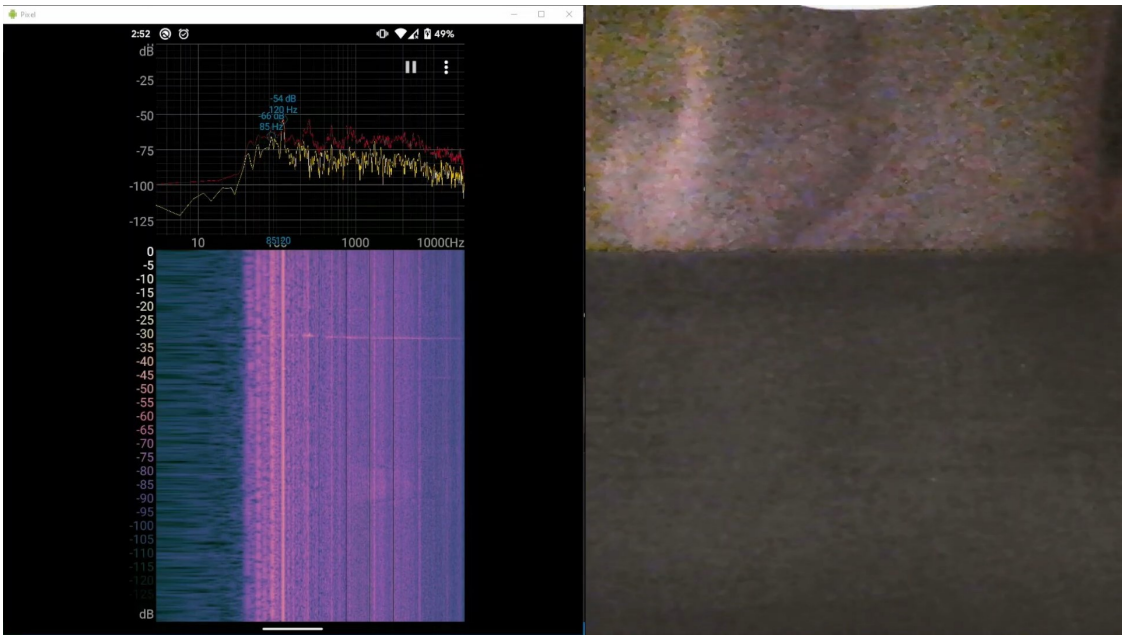
- Tones between 500Hz and 1000Hz are the easiest to detect while solving a speedcube since they have the least competition from other background sounds while solving a speedcube.
- Tones between 1000Hz and 20000Hz must be significantly louder than -30dB or they risk being indistinguishable from the background noise created by a noisy speedcube like the QiYi Qimeng.

4.2.2 Tone Distinctiveness

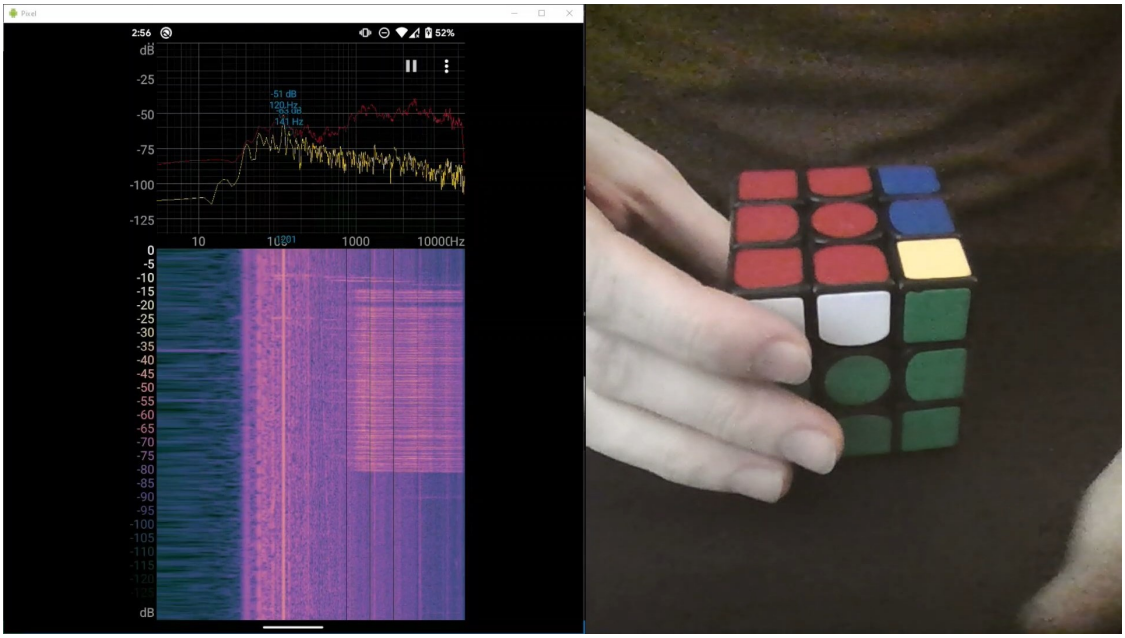
If more than one tone is required for the protocol, each tone must be unique enough to be easily distinguished from each other tone. Since a standard smartphone or laptop microphone will be used on the listening end of this protocol, the definition of "easily distinguishable" must be based on an assessment of how clearly smartphone and laptop-grade microphones can distinguish similar frequencies.

FIGURE 4.1: Background Noise of a Quiet Room while Speedsolving

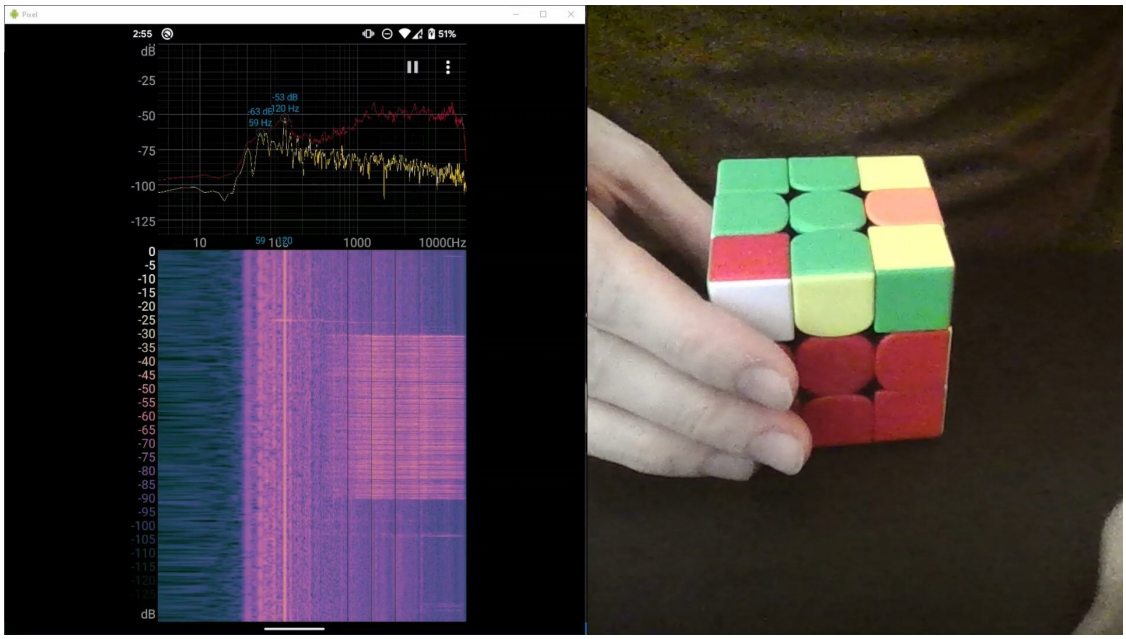
(A) No Speedsolving (Quiet Room)



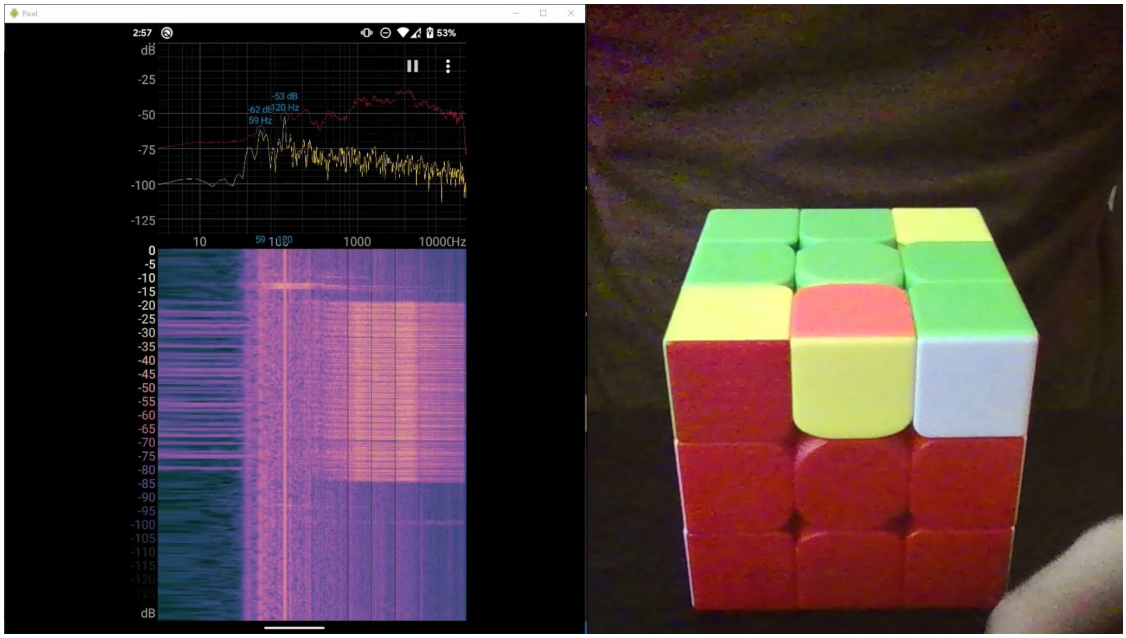
(B) Quiet Solving (Gans 356)



(C) Normal Solving (Gans XS)



(D) Loud Solving (QiYi Qimeng)

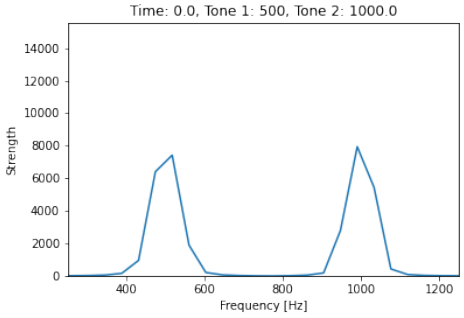


To measure the sensitivity of a standard smartphone microphone (in this case a Google Pixel 1), a recording was taken of two distinct tones that started 500 Hz apart and stepped closer together in 10 Hz increments every 0.5 seconds until their frequencies were identical.

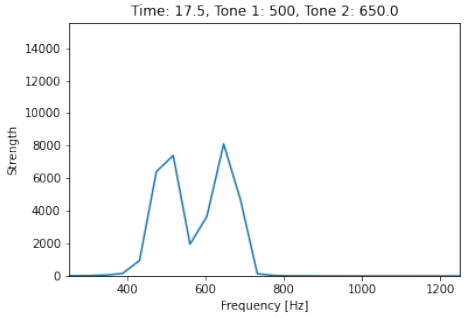
As shown in Figure 4.2, the two tones are clearly distinguishable from 500Hz apart to 150Hz apart. However, the distinction began waning once the tones came within 100Hz of each other, and by 80Hz they became entirely indistinguishable.

FIGURE 4.2: Distinctiveness of Two Increasingly Similar Tones

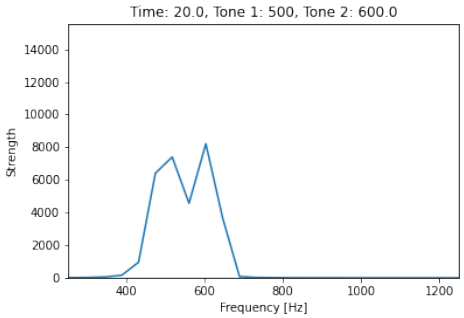
(A) 500Hz Separation



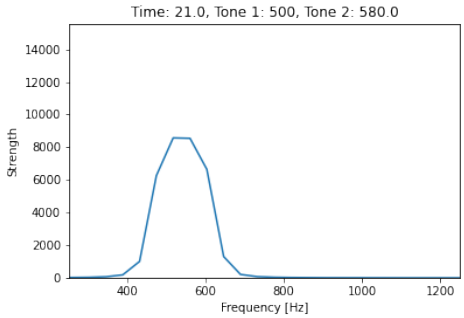
(B) 150Hz Separation



(C) 100Hz Separation



(D) 80Hz Separation



Thus, a sound-based move tracking protocol must account for the following items in order to achieve adequate tone distinctiveness:

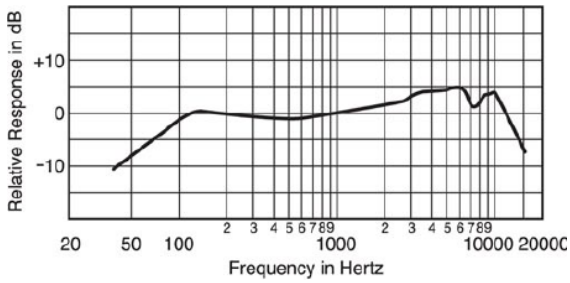
- Tones should be separated by at least 100Hz in order to be clearly distinguished from other tones in the protocol.

4.2.3 Frequency Response Range of Consumer Hardware

Since a standard smartphone or laptop microphone will be used on the listening end of this protocol, any tones used in the protocol must be within the range of tones that a smartphone or laptop-grade microphones can pick up. This range is formally known as the "frequency response range" of the microphone.

According to a Stanford research paper [29] that analyzed over 10,000 mobile devices, the typical smartphone microphone has a frequency response range of 20Hz to 20kHz as shown in Figure 4.3.

FIGURE 4.3: "A typical frequency response curve for a microphone." [29]



Thus, a sound-based move tracking protocol must account for the following restrictions on which tones can be used for the protocol:

- Tones used in the protocol must fall on the range of 20Hz-20kHz in order to be detectable by a typical smartphone or laptop microphone.

4.2.4 Human Auditory Range

An audible protocol could be distracting to a speedcuber. The human ear can detect audible frequencies from 20Hz to 20kHz, though this usually degrades with age with many people unable to notice sounds above 16kHz. [30]. However, not all tones are pleasant to listen to, particularly higher frequency tones. Tones within the frequency range of a piano (27.5Hz - 4kHz) are generally considered acceptable, while tones above the piano's upper range are often irritating. [31]

Thus, a sound-based move tracking protocol should be considerate of the human ear's sensitivity to various frequencies:

- The most acceptable tones for human speedsolvers are in the musical range of up to 4kHz or above the standard audible range of 16kHz

4.3 Specification

Given the above constraints, this section will detail a sound-based protocol for tracking the moves of a Rubik's Cube by continuously transmitting the current state of the cube. In this protocol, changes to the cube's state cause a change in the transmitted tones which can be recorded and analyzed to determine the face turn applied.

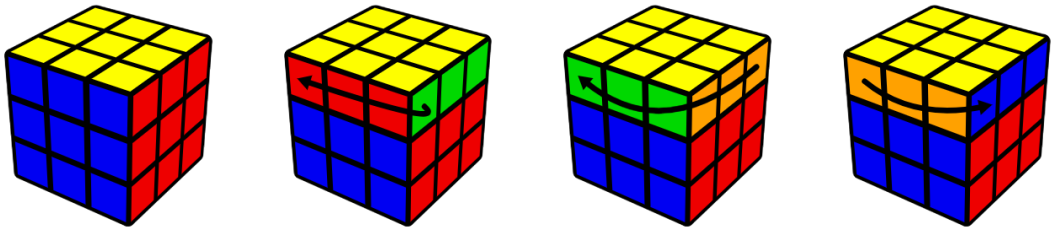
4.3.1 Representing the Cube's Current State

As mentioned in Section 2.2, a Rubik's Cube has six centerpieces that are fixed relative to each other, but can each rotate freely through four possible rotational positions as shown in Figure 4.4.

The state of a centerpiece is defined as its current rotational position. Implicit in this definition is the fact that a centerpiece is guaranteed to occupy one and only one of its four possible states at any given time.

FIGURE 4.4: The four possible rotations of a face of a Rubik's Cube [32]

(A) Full Alignment (B) 90° Misalignment (C) 180° Misalignment (D) 270° Misalignment



Each of the six centerpieces has an independent set of four possible states, yielding a total of 24 different centerpiece states for the Rubik's Cube, of which there will always be exactly six active at any given time.

It's important to note that a knowledge of the current state of each centerpiece does not imply knowledge of the exact state of each of edge and corner cubie. For example, after applying the algorithm R U R' U' all centerpieces have the same state they occupied prior to the algorithm's execution, while most of the edges and corners in the R and U layers will have been moved or rotated. However, in the reverse case, knowledge of the applied move sequence is sufficient information to determine the exact state of all cubies. Section 4.3.2 will explain how extract full move sequences using only the state information of the centers.

4.3.2 Tracking Face Turns

The only way to change a centerpiece's state is by applying a face turn. As such, when a centerpiece is rotated to a new position, its state changes to that of the new position. From there, a simple comparison of the new state to the previous state reveals the exact face turn applied to the cube.

Thus, in order to determine the sequences of moves applied to the Rubik's Cube, one simply needs to track how the state of its centerpieces changes over time.

4.3.3 Conveying State Through Sound

Conveying the current state of the cube's centerpieces through sound can be done by simply associating each of the 24 possible centerpiece states with a specific tone as shown in Table 4.1 and broadcasting a signal composed of the six tones corresponding to the active states of the cube's centerpieces.

Whenever a face turn is applied, the tone associated with the rotated centerpiece would change to the tone representing the piece's new state. From there, a microphone equipped device can record the broadcast frequencies, measure the changes in the frequencies over time, convert the frequency changes to state changes, and finally extract any move sequence applied to the cube.

Since there will only ever be a single tone broadcast for a specific face, the four tones used to represent each of its possible states can be chosen closer to the minimum separation specified in Section 4.2.2. However, the gap between tones for different faces should be larger to help prevent incorrectly classifying a tone as conveying a state of a different face. In the sample frequencies shown in Table 4.1, the frequency gap between faces is 200Hz, a value which performed well in testing.

TABLE 4.1: Sample frequencies for encoding a Rubik's Cube's state

Alignment	White	Yellow	Red	Orange	Green	Blue
Full	800 Hz	1300 Hz	1800 Hz	2300 Hz	2800 Hz	3300 Hz
90°	900 Hz	1400 Hz	1900 Hz	2400 Hz	2900 Hz	3400 Hz
180°	1000 Hz	1500 Hz	2000 Hz	2500 Hz	3000 Hz	3500 Hz
270°	1100 Hz	1600 Hz	2100 Hz	2600 Hz	3100 Hz	3600 Hz

4.4 Alternative Protocol

Instead of transmitting the current state of the cube's centerpieces, an alternative protocol design could seek to directly transmit the face turns applied to the cube.

This perspective focuses on the fact that all move sequences can be broken down into a series of 90° face turns. Since the cube consists of 6 faces, and each face can be turned either clockwise or counterclockwise, one could design a two-tone protocol using only 8 discrete audio frequencies to build the smart cube. The first tone would come from one of six predefined audio bands, one for each face of the cube. The second tone would come from one of two separately predefined audio bands, one for each possible direction of rotation. From this, an audio processing model could be designed to process a sequence of these two-tone pairs and reconstruct the sequence of face rotations by recording the rotated face followed by its direction of rotation.

However, while this model minimizes the number of discrete frequencies required to communicate changes in the cube's state, it carries many challenges. Consider the example of a speedcuber averaging 5 turns per second (TPS) with bursts up to 10 TPS (common for a speedcuber that averages 12-15 seconds per solution). The burst TPS would require the successful transmission of 20 sequential tones within a single second - only 50ms per tone, all in the midst of additional noise from the cube's pieces hitting each other harder at the higher turn speed. Adding to the difficulty, since each tone is only ever transmitted once, the audio detection model must achieve 100% tone recognition to be able to accurately reconstruct the originating move sequence. As a result, this model fails to support any sort of error correction that would make it resistant to the common challenges to data transmission through sound.

4.5 Summary

In summary, a sound based protocol for tracking the moves of a Rubik's Cube must be considerate of the following constraints:

1. Tones should be transmitted with a strength greater than -30dB to achieve a high signal-to-noise ratio. (Section 4.2.1)
2. Tones should be separated by at least 100Hz in order to be clearly distinguished from other tones in the protocol. (Section 4.2.2)
3. Tones must fall on the range of 20Hz-20kHz in order to be detectable by a typical smartphone or laptop microphone. (Section 4.2.3)
4. Tones should fall on the range 0Hz-4kHz or 16kHz-20kHz to minimize annoyance to the human speedsolver. (Section 4.2.4).

The protocol that will be used in subsequent chapters consists of 24 unique tones, one for each possible rotational state of each centerpiece. A transmitter embedded in the Rubik's Cube will continuously broadcast the six tones corresponding to the current rotational state of each centerpiece adjust those tones as face turns are applied (Details in Chapter 6). A software receiver will then listen to the transmitted audio and measure the changes in the transmitted frequencies over time to reconstruct the originally applied sequence of face turns (Details in Chapter 5).

Chapter 5

The Receiver

5.1 Introduction

The receiver for the sound-based move tracking protocol is in charge of decoding a sequence of face turns from the transmitted audio.

To do this, the receiver must go through the following steps:

1. Record the transmitted audio.
2. Measure the audible frequencies at each time step.
3. Convert each time step's audible frequencies to the cube's state at that moment.
4. Decode the applied face turns from the sequence of cube states.

This chapter will describe the development of a software algorithm in Python that can serve as a receiver for this sound-based move tracking protocol. This development began with the creation of synthetic audio recordings representing the tones that would be emitted by a Rubik's Cube equipped with an ideal transmitter (Section 5.2) followed by the implementation of an algorithm capable of decoding that ideal synthetic audio (Section 5.3). The algorithm was then made more robust by adding realistic noise to the synthetic audio to better simulate a real speedcubing environment (Section 5.4) followed by enhancing the previously designed algorithm to continue to decode the applied move sequence in the midst of the added noise (Section: 5.5). [1]

5.2 Creating Synthetic Audio

The first step of designing this receiver is to synthesize an audio signal representative of the output of the ideal transmitter.

This audio synthesis starts with encoding the frequency corresponding to each centerpiece state (Section 5.2.1), then involves creating a virtual Rubik's Cube (Section 5.2.2), and finally culminates in generating the audio signal from the virtual Rubik's Cube's state (Section 5.2.3).

[1] The contents of this chapter have been specially written so that the reader can copy them into a Jupyter Notebook running a Python 3.9 kernel and see the same results for himself/herself.

5.2.1 Representing the Audio Protocol

For the synthetic audio generator to produce a realistic signal, it needs to know which frequencies to transmit for each centerpiece state. This is easily accomplished by creating a dictionary to map each possible state of each face to its corresponding frequency.

For this design, the frequency assignments from Table 4.1 are converted to the dictionary shown in Figure 5.1a with two small changes. First, by assuming no cube rotations, the face color (e.g. "White") can be converted to a layer name (e.g. "U") which simplifies the code for both encoding and decoding a move sequence from audio. Second, the alignments from Figure 4.4 are divided by 90 to get a simple sequence of consecutive integers that are easier to manipulate in code.

With this dictionary in place, the frequency to transmit for any particular centerpiece's current rotation can be determined by looking up the centerpiece and its rotation in the dictionary as shown in Figure 5.1b.

FIGURE 5.1: Centerpiece State to Frequency Mapping

(A) Frequency mapping dictionary

```
FREQUENCY_MAPPINGS = {
    "U": {
        0: 800, 1: 900, 2: 1000, 3: 1100
    },
    "D": {
        0: 1300, 1: 1400, 2: 1500, 3: 1600
    },
    "R": {
        0: 1800, 1: 1900, 2: 2000, 3: 2100
    },
    "L": {
        0: 2300, 1: 2400, 2: 2500, 3: 2600
    },
    "F": {
        0: 2800, 1: 2900, 2: 3000, 3: 3100
    },
    "B": {
        0: 3300, 1: 3400, 2: 3500, 3: 3600
    }
}
```

(B) Frequency lookup function

```
def frequency_of(centerpiece: str, rotation: int) -> float:
    return FREQUENCY_MAPPINGS[centerpiece][rotation]
```

5.2.2 Representing the Rubik's Cube

While there are many implementations of a digital Rubik's Cube that can track every cubie and render an interactive 3D cube, all that's needed for the synthetic audio generator is a representation of a Rubik's Cube on which individual face turns can be virtually applied and the resulting centerpiece state can be read out.

To do this, a `RubiksCube` object is created that encapsulates a dictionary containing the same keys for each face as the `FREQUENCY_MAPPINGS` dictionary in Figure 5.1a above, each associated with a single integer representing the current rotational state of that face. A method called `apply_move` is also defined with a parameter for a valid move like `U` or `U'` that will update the `RubiksCube`'s state to reflect the rotation.

FIGURE 5.2: A simple abstraction of a Rubik's Cube

```
1 class RubiksCube:
2
3     CLOCKWISE = 1             # aka a 90 degree turn
4     COUNTERCLOCKWISE = 3      # aka a 270 degree turn which achieves the
5                               # same resulting state as a -90 degree turn
6
7     def __init__(self):
8         self.state = { "U": 0, "D": 0, "R": 0, "L": 0, "F": 0, "B": 0 }
9
10    def apply_move(self, move: str):
11        # Extract the face and direction from the 'move' string.
12        face = move[0]
13        if len(move) == 1:                                  # e.g. U
14            direction = RubiksCube.CLOCKWISE
15        else:                                               # e.g. U'
16            direction = RubiksCube.COUNTERCLOCKWISE
17
18        # Update the state to apply the move.
19            # NOTE: The 'mod 4' exists to wrap back around to 0
20            # after a full 360 degrees of rotation.
21        self.state[face] = (self.state[face] + direction) % 4
```

5.2.3 Creating Synthetic Audio for an Arbitrary Algorithm

Now the synthetic audio can be generated for any valid algorithm. This is done using the `tones` library created by Erik Nyquist. [33]

First, a separate audio track is created for each centerpiece on the cube since there will be a tone continuously broadcast from each centerpiece of the cube (Figure 5.3a).

Second, a new function is written to add the tones corresponding to the virtual cube's current state to the audio mixer using the `frequency_of` function from Figure 5.1b (Figure 5.3b). This function includes a `tps` parameter that controls the duration of the added tones so that they change to the next tone at the same speed as the turns on a speedcube being solved at that number of turns per second.

Then, those two functions are tied together by iterating through each move in a Rubik's Cube `alg`orithm and saving the audio of the resulting state changes into a .wav file at a given file path (Figure 5.3c). Additionally, the `tps` parameter is propagated to this function to enable generating audio for an algorithm at different turn speeds.

FIGURE 5.3: Generating audio for any Rubik's Cube algorithm

(A) Function to create an audio mixer for a virtual Rubik's Cube

```
from tones.mixer import Mixer  # https://pypi.org/project/tones/
from tones import SINE_WAVE

def _create_mixer(rubiks_cube: RubiksCube) -> Mixer:
    mixer: Mixer = Mixer(sample_rate=44100, amplitude=1)
    # Add a separate audio track for each centerpiece.
    for face, _ in rubiks_cube.state.items():
        mixer.create_track(face, SINE_WAVE, attack=0, decay=0)
    return mixer
```

(B) Function to update an audio mixer with a virtual Rubik's Cube's state

```
def _render_cube_state(mixer: Mixer, rubiks_cube: RubiksCube, tps: float):
    for face, rotation in rubiks_cube.state.items():
        mixer.add_tone(face, frequency_of(face, rotation), duration=1 / tps)
```

(C) Function to create a .wav file of any Rubik's Cube algorithm

```
def render_audible_alg(alg: str, wav_path: str=None, tps: float=4):
    # Create the virtual Rubik's Cube.
    rubiks_cube: RubiksCube = RubiksCube()
    # Create the audio mixer used to create the synthesized audio.
    mixer = _create_mixer(rubiks_cube)
    # Add the initial cube state to the mixer.
    _render_cube_state(mixer, rubiks_cube, tps)
    # Iterate over the alg, updating the mixer after each move.
    moves = alg.split(" ")
    for move in moves:
        rubiks_cube.apply_move(move)
        _render_cube_state(mixer, rubiks_cube, tps)
    # Save the final audio to a .wav file.
    mixer.write_wav(wav_path if wav_path else f"{alg}.wav")
```

With these functions in place, synthetic audio can be easily created for any valid Rubik's Cube algorithm. For example, generating synthetic audio that sweeps through every possible centerpiece state can be done with the two lines of code in Figure 5.4.

FIGURE 5.4: Example Audio Generation for a Rubik's Cube Algorithm

```
demo_alg = "U U U U D D D D R R R R L L L L F F F F B B B B"
render_audible_alg(demo_alg, "demo_all_states.wav")
```

5.3 Decoding the Synthetic Audio

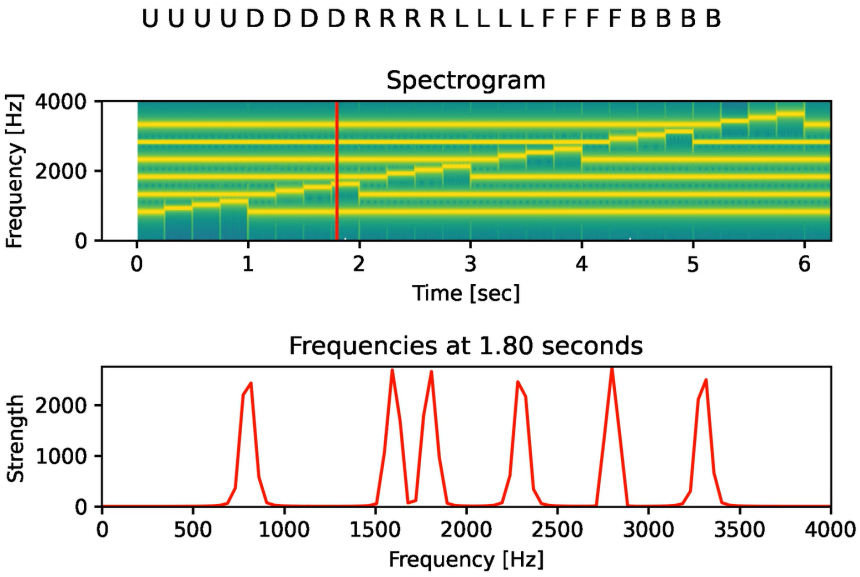
With synthetic audio now available for any valid Rubik's Cube algorithm, the next step is to create an initial software algorithm that can decode that audio back into the original move sequence.

Accomplishing this will require computing the synthetic audio's spectrogram (Section 5.3.1), followed by extracting the dominant frequencies present at each time step (Section 5.3.2), and converting those dominant frequencies into the corresponding Rubik's Cube centerpiece states (Section 5.3.3), all before finally recovering the originally applied move sequence (Section 5.3.4).

5.3.1 Computing the Spectrogram

The first step in decoding the synthetic audio is determining its component frequencies at any specific moment in time. These component frequencies can be easily visualized using a spectrogram, like the one in Figure 5.5 of the synthetic audio created in Section 5.2.3. [2]

FIGURE 5.5: Spectrogram of the synthetic audio created in Section 5.2.3



This spectrogram has the same pieces as the ones described in Section 4.2.1 with a few minor changes. First the order of the graphs has been reversed, so the top graph shows the actual spectrogram and the bottom graph shows the strength of the component frequencies at the specific point in time indicated by the vertical red line on the top graph. Additionally, the spectrogram has been rotated 90° so time is now on the x-axis and the component frequencies are shown on the y-axis. Finally, the color scheme is slightly different, with the presence of a strong component frequency indicated by a bright yellow color instead of bright purple or pink.

[2] The source code for Figure 5.5 is available in Appendix A.1.

The Brief Overview of the Math behind the Spectrogram

However, while Matplotlib's `specgram` plot [34] made it easy to create Figure 5.5, it didn't return the underlying data of the spectrogram required to decode the exact values and strengths of the component frequencies at each point in time. To compute that data directly, it is helpful to understand the basics of the math behind the spectrogram. Dr. Steve Brunton from the University of Washington created an excellent video series on this topic [35], of which the following few paragraphs are a short summary.

The foundational mathematical concept behind the spectrogram is the Fourier Transform, which is a technique for approximating any continuous function using only sine and cosine functions. In the case of an analog audio signal -which is inherently a composite of many sine functions (one for each component frequency)- applying a Fourier Transform would return data for a graph of all the frequencies present at any point in the entire signal and their overall strength throughout it.

But digital audio signals like .wav files are a series of discrete values, not continuous functions, which means the Fourier Transform cannot directly operate on them. Fortunately, there exists a variant of the Fourier Transform called the Discrete Fourier Transform which can operate on a list of discrete data points by assuming they sample a continuous function and then computing the Fourier Transform of that function.

However, when applied to a digital audio signal, the Discrete Fourier Transform still only computes which frequencies were present at *any* time during the duration of the signal, but not *when* they occurred. That data comes from another layer of computation called the Gabor Transform (i.e. the Short-Time Fourier Transform) which computes a Discrete Fourier Transform over a sliding window of the input samples to approximate the changes in the strength of each component frequency over time.

This knowledge, combined with the `numpy` [36] and `scipy` [37] libraries which can calculate these transforms, enables the creation of a function (Figure 5.6) to compute the spectrogram for the audio stored in a given .wav file. The return values `freq` and `time` are, respectively, lists of the .wav file's component frequencies and time steps, while `spectrogram` is a 2D array of the strengths of each component frequency at each time step. They are related by common indices, such that the strength of the frequency `freq[f_idx]` at the time `time[t_idx]` is found in `spectrogram[t_idx][f_idx]`.

FIGURE 5.6: Function to compute the spectrogram of a .wav file

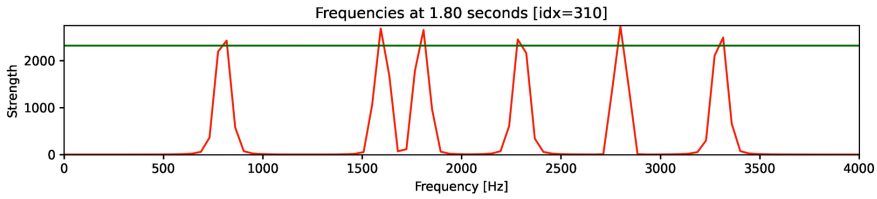
```
1  import numpy as np
2  from scipy import signal
3  from scipy.io import wavfile
4
5  def compute_spectrogram(wav_path: str):
6      SAMPLES_PER_WINDOW = 1024  # Balances frequency/time accuracy
7      sample_rate, audio_samples = wavfile.read(wav_path)
8      freq, time, Zxx = signal.stft(audio_samples, fs=sample_rate,
9          nperseg=SAMPLES_PER_WINDOW, noverlap=(SAMPLES_PER_WINDOW // 4) * 3)
10     spectrogram = np.abs(Zxx).transpose()
11     return freq, time, spectrogram
```

5.3.2 Extracting the Dominant Component Frequencies

Looking back at the component frequency graph in Figure 5.5 it is clear that it has six distinct peaks: these are the transmitted frequencies representing the current state of the virtual Rubik's Cube's six centerpieces at that specific instant of time.

The exact frequencies of these peaks can be extracted by filtering out all frequencies whose strength is not above a specific threshold. In this case, a simple threshold of 85% of the maximum strength of any component frequency (see the green line in Figure 5.7[3]) isolates the six dominant component frequencies.

FIGURE 5.7: Spectrogram from Figure 5.5 with a threshold at 85% of the maximum strength of the component frequencies.



Using the actual spectrogram data, the exact values of these peaks can be computed. First, the threshold computation is broken into its own function (Figure 5.8a). From there, a second function (Figure 5.8b) can iterate through all the component frequencies of the computed spectrogram data at a specific time index to compile and return a list of the frequencies whose strength exceeds the threshold at that time step.

FIGURE 5.8: Extracting dominant frequencies from one time step of audio

(A) Function to calculate a simple threshold to isolate the most dominant frequencies

```
1 def compute_threshold(values: list):
2     return max(values) * 0.85
```

(B) Function to extract all frequencies stronger than a threshold at a given time step

```
3 def extract_important_freqs(freq, time, spectrogram, t_idx):
4     # Create a list to store the important frequencies
5     important_freqs = []
6     # Compute the threshold for this time step
7     threshold = compute_threshold(spectrogram[t_idx])
8     # Check the strength of each frequency at this time step
9     for f_idx in range(len(freq)):
10        # If the frequency's strength is above the threshold ...
11        if spectrogram[t_idx][f_idx] > threshold:
12            # ... save the frequency and its strength for later processing
13            important_freqs.append(dict(
14                hz=freq[f_idx],
15                power=spectrogram[t_idx][f_idx]
16            ))
17    return important_freqs
```

[3]The source code for Figure 5.7 is available in Appendix A.2.

With these functions, finding the actual frequencies of the peaks in Figure 5.7 can be done in just a few lines of code as shown in Figure 5.9.

FIGURE 5.9: Example peak frequency decoding at a specific time step

```
freq, time, spectrogram = compute_spectrogram(demo_wav_path)
important_freqs = extract_important_freqs(
    freq, time, spectrogram, t_idx=310) # 1.80 seconds
print([f"{x["hz"]:.0f}Hz" for x in important_freqs])
```

```
>> ["818Hz", "1593Hz", "1809Hz", "2283Hz", "2799Hz", "3316Hz"]
```

5.3.3 Translating Component Frequencies to Centerpiece States

With the specific frequencies of each detected peak, the original state of the Rubik's Cube at that moment in time can be computed by finding the states whose corresponding frequency is closest to each detected peak frequency.

This is done by first defining a function that will find the closest centerpiece state for one peak frequency by iterating through every centerpiece state in the `FREQUENCY_MAPPINGS` dictionary from Section 5.2.1, comparing the difference between the state's frequency and the given peak frequency, and returning the state with the smallest difference (Figure 5.10a). This function can then be called for all the detected peak frequencies to recover the state of the cube at that time step (Figure 5.10b).

FIGURE 5.10: Converting peak frequencies to centerpiece states

(A) Function to find the state most similar to the detected frequency

```
def _closest_state(detected_freq):
    # Start with no assumptions regarding the closest state
    closest_state = None
    closest_difference = None
    # Iterate through each possible state to find the closest one
    for face, rotations in FREQUENCY_MAPPINGS.items():
        for rotation, freq in rotations.items():
            difference = abs(detected_freq - freq)
            if closest_difference is None or difference < closest_difference:
                closest_difference = difference
                closest_state = dict(face=face, rotation=rotation)
    return closest_state
```

(B) Function to determine the cube's state from a set of peak frequencies

```
def get_state_from_freqs(important_freqs: list) -> dict:
    # Start with an unpopulated state
    state = {}
    # Record the state represented by each of the peak frequencies
    for freq in important_freqs:
        hz, power = freq.values()
        closest_state = _closest_state(hz)
        state[closest_state["face"]] = closest_state["rotation"]
    return state
```

For example, Figure 5.11 shows how using these functions makes it easy to covert the peak frequencies extracted in Figure 5.9 to their corresponding centerpiece states.

FIGURE 5.11: Example conversion of peak frequencies to states

```
detected_state = get_state_from_freqs(important_freqs)
print(detected_state)

>> {"U": 0, "D": 3, "R": 0, "L": 0, "F": 0, "B": 0}
```

As such, obtaining a sequence of the Rubik's Cube's centerpiece states over the course of the recorded audio sequence only requires repeating this process for each time step in the spectrogram data. This can also be easily turned into another function as shown in Figure 5.12.

FIGURE 5.12: Function to list the cube's state at each time step

```
def get_state_over_time(freq, time, spectrogram):
    # Start with an empty list of centerpiece states
    state_over_time = []
    # For each time step ...
    for t_idx in range(len(time)):
        # find the peak frequencies,
        important_freqs = extract_important_freqs(
            freq, time, spectrogram, t_idx)
        # convert the peak frequencies to the corresponding state,
        state = get_state_from_freqs(important_freqs)
        # and save the state and time stamp for later processing.
        state_over_time.append(dict(
            time=time[t_idx],
            state=state
        ))
    return state_over_time
```

And, for completeness, Figure 5.13 shows an example of using that function to get the full sequence of states for the synthetic audio generated in Section 5.2.

FIGURE 5.13: Example listing of states over time

```
state_over_time = get_state_over_time(freq, time, spectrogram)
print(state_over_time)

>> [{"time": 0.0, "state": {"U":0, "D":0, "R":0, "L":0, "F":0, "B":0}},
    {"time": 0.0058 "state": {"U":0, "D":0, "R":0, "L":0, "F":0, "B":0}},
    ... ]
```

5.3.4 Extracting Move Sequences from Centerpiece State Sequences

The final step to recover the original move sequence is to iterate over the sequence of cube states and register any change to the cube state as a move applied to the cube.

This starts with a function (Figure 5.14a) that, given a starting and ending rotational state for a specific face, can calculate which direction a face was turned and return the text notation of the applied face turn. A second function (Figure 5.14b) then iterates through the state sequence extracted in Section 5.3.3 checking each state in the sequence against the previous one to detect when the state changes. Upon detecting a change, this second function then calls the first to get the actual move that caused the state change and saves it to a list of `detected_moves` to be returned after iterating through all states. This list of `detected_moves` is the final move sequence that was extracted from the synthetic audio.

FIGURE 5.14: Code to extract move sequences from state sequences

(A) Function to determine the face turn between two centerpiece states

```
def _move_from(face, current_rotation, new_rotation):
    direction = None
    # Check whether a Clockwise or Counterclockwise turn occurred
    # using the same logic as `RubiksCube.apply_move`
    if (current_rotation + RubiksCube.CLOCKWISE) % 4 == new_rotation:
        direction = ""  # Clockwise
    elif (current_rotation + RubiksCube.COUNTERCLOCKWISE) % 4 == new_rotation:
        direction = "'" # Counterclockwise
    # Create and return the notation for the applied face turn
    return f"{face}{direction}"
```

(B) Function to extract all moves from a centerpiece state sequence

```
def detect_moves(state_over_time):
    detected_moves = []
    current_state = {}
    # Iterate through all of the extracted cube states
    for idx, timed_state in enumerate(state_over_time):
        time, state = timed_state.values()
        for face, rotation in state.items():
            # Save the first state as the initial state
            if not (face in current_state):
                current_state[face] = rotation
            # If the state of any face changes...
            if rotation != current_state[face]:
                # ... get and save the applied move with its timestamp...
                detected_moves.append(dict(
                    time=time,
                    move=_move_from(face, current_state[face], rotation)
                ))
                # ... and update the tracked state based on the new move.
                current_state[face] = rotation
    return detected_moves
```

Figure 5.15 shows an example usage of the `detect_moves` function. Additionally, the extracted Rubik's Cube algorithm is compared to the original one used to create the synthetic audio.

FIGURE 5.15: Example move sequence extraction

```
detected_moves = detect_moves(state_over_time)

pretty_moves = " ".join([i["move"] for i in detected_moves])
print(pretty_moves)
print(f"Matches demo_alg? {demo_alg == pretty_moves}")

>> U U U U D D D D R R R R L L L L F F F F B B B B
>> Matches demo_alg? True
```

Clearly, the detected move sequence matches the demo algorithm used to create the synthetic audio in Section 5.2.3, which means this algorithm is a functional receiver for an ideal transmitter.

5.3.5 Full Example: Extracting Moves from Synthetic Audio

As a summary of this section on decoding synthetic audio, Figure 5.16 shows the full process for creating a synthetic audio clip and extracting the encoded move sequence from it. For this example, the chosen move sequence to transmit consists of every possible face turn, which will test that this strategy can correctly report back each face turn.

FIGURE 5.16: Full Example: Audio generation and move extraction

```
# Create the synthetic audio for a   new algorithm
demo2_alg = "U U' D D' R R' L L' F F' B B'"
demo2_wav_path = "demo_all_turns.wav"
render_audible_alg(demo2_alg, demo2_wav_path)

# Extract the move sequence from the new synthetic audio
freq2, time2, spectrogram2 = compute_spectrogram(demo2_wav_path)
state_over_time2 = get_state_over_time(freq2, time2, spectrogram2)
detected_moves2 = detect_moves(state_over_time2)

# Print out the results
pretty_moves2 = " ".join([i["move"] for i in detected_moves2])
print(pretty_moves2)
print(f"Matches demo2_alg? {demo2_alg == pretty_moves2}")

>> U U' D D' R R' L L' F F' B B'
>> Matches demo2_alg? True
```

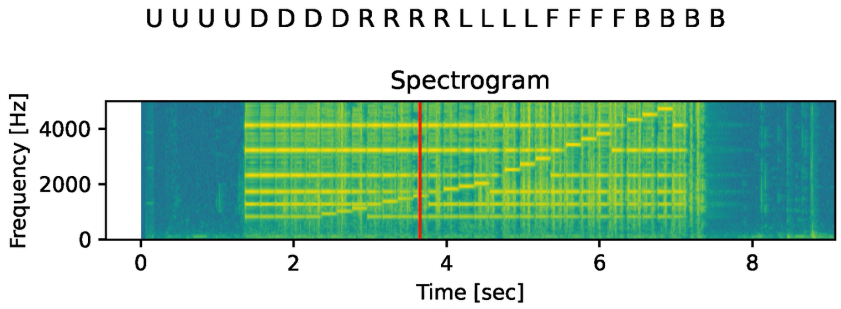
And again, the extracted move sequence does match the one used to generate the synthetic audio, further validating this strategy for decoding moves from audio.

5.4 Creating Realistic Audio

While Section 5.3's successful recovery of a sequence of moves applied to a virtual Rubik's Cube is impressive, the synthetic audio is not realistic. This is made obvious by a comparison between the spectrograms of the background audio in Figure 4.1 and the synthetic audio in Figure 5.5. In the latter, the bright yellow indicators of present frequencies are very clear and strong with no other frequencies present in the signal. In contrast, the former contains many areas with bright pink indicators of prominent frequencies (the color differences are due to the use of different applications to generate the diagrams). A more realistic signal would contain both the strong bands of the transmitted frequency along with the underlying background noise of whatever cube is actively being solved.

Adding this noise could be done in software by overlaying the synthetic audio with a recording of the background noise, but it was just as easy to play the synthetic audio from a laptop speaker and record it using a nearby smartphone while solving various Rubik's Cubes. An example of the spectrogram resulting from recording the audio of the demo algorithm from Section 5.2.3 on a Google Pixel smartphone while actively solving a Gans 356 speedcube can be seen in Figure 5.17. [4]

FIGURE 5.17: Spectrogram of a realistic audio sample



Notice how the horizontal bands representing the transmitted signal are dimmer in the realistic audio than in the purely synthetic audio. This reflects the loss of volume any tone experiences as it travels through the air. Additionally, the transmitted signal is also partially obscured by the additional audible noise of solving the speedcube.

5.5 Decoding Realistic Audio

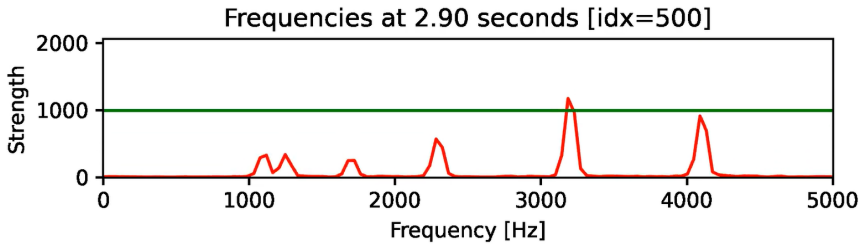
The loss of signal strength alongside the added noise creates several unique challenges that require more sophisticated analysis than that presented in Section 5.3. These challenges include added variation in the strength of each peak frequency (Section 5.5.1), the detection of conflicting centerpiece states (Section 5.5.2), and the erroneous reading of background noise as centerpiece states (Section 5.5.3).

[4] The astute reader will notice that the audio bands in Figure 5.17 fall on slightly different frequencies than in Figure 5.5. This is because the realistic audio samples encoded centerpiece states using slightly different frequency assignments than the synthetic audio generated in Section 5.2.

5.5.1 Fine-Tuning the Threshold

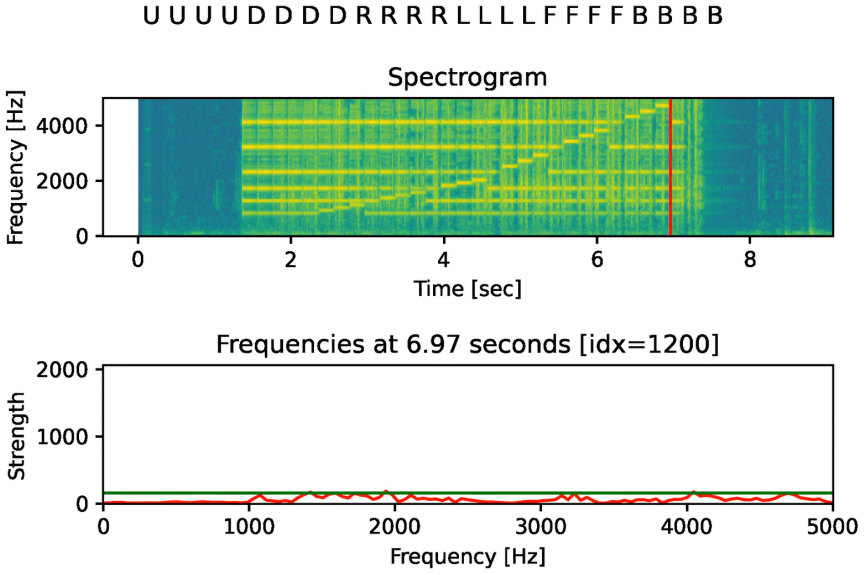
Because some audio frequencies are dampened more than others as they travel through the air to the recording microphone, a static threshold like the one used in Section 5.3.2 fails to capture all the dominant frequencies that compose the audible signal. For example, Figure 5.18 shows how the 85% threshold from Section 5.3.2 misses five of the six signal peaks in the realistic audio.

FIGURE 5.18: An 85% threshold applied to realistic audio



Furthermore, the natural background noise can also cause false positives during the moments between moves while the signal is weaker as a result of changing from one state to the next. For example, Figure 5.19 shows a moment between face turns when the audio signal is nearly non-existent, and the background noise can falsely register as important peak frequencies for centerpiece state detection. In this case, between 1000Hz and 2000Hz alone there are six different points where the background noise reaches the 85% threshold compared to the three centerpieces who state gets transmitted within that same band.

FIGURE 5.19: Dominance of background noise between moves



Resolving these issues starts by noticing that the power of each peak frequency bearing the audio signal is generally much higher than the power of all other component frequencies. As such, basing the threshold at a power equal to one standard deviation of the power of the component frequencies of a specific time step generally captures all the peak frequencies while still excluding the underlying noise of the cube/environment. Additionally, the use of a hard minimum value for the threshold helps reduce the number of false positives during the gaps in the signal between face turns.

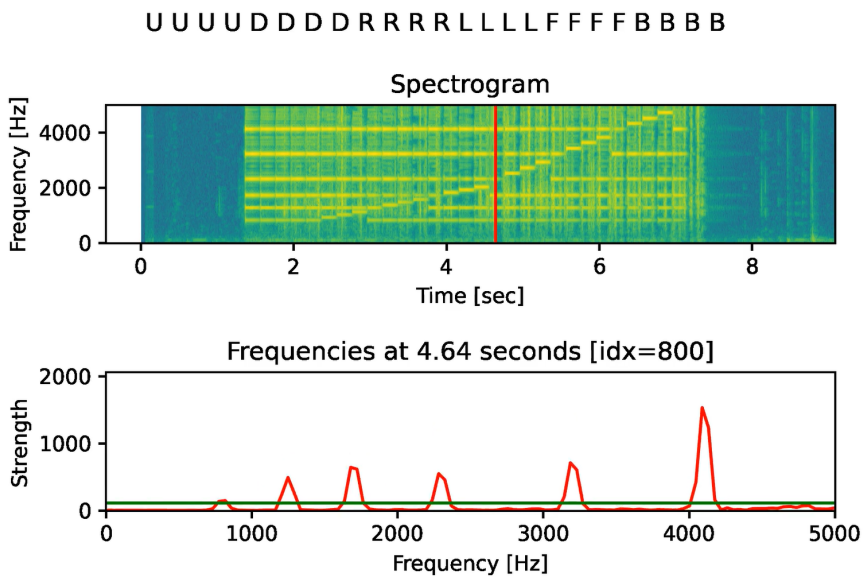
These two changes are encoded in an updated version of the `compute_threshold` function first defined in Section 5.3.2. This new version contains two new parameters: `stdv_pct`, which can be used to adjust the threshold to some multiple of the standard deviation for testing purposes, and `min_thresh` which is the "hard minimum value" discussed in the previous paragraph.

FIGURE 5.20: Updated version of threshold computation in Figure 5.8a

```
1 def compute_threshold(values: list, stdv_pct: float=1, min_thresh: int=50):
2     return max(min_thresh, np.std(values) * stdv_pct)
```

An example of the thresholds yielded by this new computation is shown in Figure 5.21, where the green line representing the threshold successfully captures all six peak frequencies of the audio signal while simultaneously staying just out of reach from the frequencies of the background noise.

FIGURE 5.21: Fine-Tuned Threshold



5.5.2 Filtering through Similar Peak Frequencies

While the new threshold calculation does properly capture all the peak frequencies, it also captures more samples than just the six tips of each peak. These extra samples can cause confusion when trying to extract each centerpiece's state if they correspond to different states for the same centerpiece. That said, as long as the power of each frequency is also saved, then any disagreements about a specific centerpiece's state can be resolved by accepting the one with the most intense frequency as the actual state.

In code, this is achieved by adding a temporary dictionary to the `get_state_by_freqs` method defined in Section 5.3.3 to track the strongest frequency associated with each face. Then, while iterating through the captured peak frequencies, only the strongest one for each face is used to determine the active centerpiece state at that time step.

FIGURE 5.22: Updated version of state extraction in Figure 5.8b

```
def get_state_from_freqs(important_freqs: list) -> dict:
    state = {}
    state_power = {"U": 0, "D": 0, "R": 0, "L": 0, "F": 0, "B": 0}  # New
    for freq in important_freqs:
        hz, power = freq.values()
        closest_state = _closest_state(hz)
        if power > state_power[closest_state["face"]]:                # New
            state[closest_state["face"]] = closest_state["rotation"]
            state_power[closest_state["face"]] = power                # New
    return state
```

Using this new version of `get_state_by_freqs` works just like it did in Figure 5.11. Here the extracted state is the one encoded by the frequencies depicted in Figure 5.21.

FIGURE 5.23: Example: Refined conversion of peak frequencies to states

```
transmitted_wav_path = "transmitted-356-5tps.wav"
freq, time, spectrogram = compute_spectrogram(transmitted_wav_path)
important_freqs = extract_important_freqs(freq, time, spectrogram,
    t_idx=800) # 4.64 seconds

detected_state = get_state_from_freqs(important_freqs)
print(detected_state)
```

```
>> {"U": 0, "D": 0, "R": 0, "L": 0, "F": 0, "B": 0}
```

5.5.3 Ignoring Noise when Extracting Move Sequences

However, despite these noise filtering measures, the background noise occasionally causes the detection of an incorrect centerpiece state. Since the approach in Section 5.3.4 registers an applied face turn for *any* detected change in a centerpiece's state, *any* mis-detection would incorrectly register a face turn that never happened.

Fortunately, this issue can be mitigated by requiring that the state change persists over several time steps instead of blindly accepting any detected change in a centerpiece's state as a new face turn.

This is implemented by adding a sliding window to the `detect_moves` function first created in Section 5.3.4. The window implementation starts with a new `window_size` function parameter to control the number of consecutive time steps over which a state change has to persist before it is recorded as a move applied to the cube. Within the function, two new dictionaries are created to serve as a "staging area" for state changes: one to stage the new state and the second to record the index of the time step where the state first changed. As the function iterates through each time step, it updates these staging dictionaries each time a new state is detected. Then, it checks to see if the current index of iteration is `window_size` steps after the last reported state change. If so, the current state is updated to the value of the staged state, the change is recorded as a new move, and the window counter gets reset. The full code for this is shown in Figure 5.24.

FIGURE 5.24: Updated version of move detection in Figure 5.14b

```
def detect_moves(state_over_time, window_size=8):                    # Edit
    detected_moves = []
    current_state = {}
    new_state = {"U":-1, "D":-1, "R":-1, "L":-1, "F":-1, "B":-1}  # New
    new_state_idx = {"U":0, "D":0, "R":0, "L":0, "F":0, "B":0}    # New
    for idx, timed_state in enumerate(state_over_time):
        time, state = timed_state.values()
        for face, rotation in state.items():
            # Update new_state whenever the rotation changes       # Edit ↓
            if rotation != new_state[face]:
                new_state[face] = rotation
                new_state_idx[face] = idx
            # If the new state has persisted over window_size time steps,
            if idx - new_state_idx[face] == window_size:
                if not (face in current_state):
                    current_state[face] = new_state[face]
                # and it is a different rotation than the current state,
                elif new_state[face] != current_state[face]:
                    # then a face turn has been detected!
                    detected_moves.append(dict(
                        time=state_over_time[new_state_idx[face]]["time"],
                        move=_move_from(face, current_state[face], rotation)
                    ))
                    current_state[face] = rotation
                    new_state_idx[face] = 0
    return detected_moves
```

And to validate that these changes work as expected, Figure 5.25 runs the realistic audio through this new `detect_moves` function and compares the returned move sequence with the one originally transmitted in the audio.

FIGURE 5.25: Example: Refined move sequence extraction

```
state_over_time = get_state_over_time(freq, time, spectrogram)
detected_moves = detect_moves(state_over_time)

pretty_moves = " ".join([i["move"] for i in detected_moves])
print(pretty_moves)
print(f"Matches demo_alg? {demo_alg == pretty_moves}")
```

```
>> U U U U D D D D R R R R L L L L F F F F B B B B
>> Matches demo_alg? True
```

And once again, despite the presence of the added noise of solving a speedcube, the detected algorithm perfectly matches the transmitted signal which further validates this approach as a viable proof-of-concept for tracking the moves of a Rubik's Cube using sound.

5.5.4 Optimizing Algorithm Parameters

Significant testing went into determining the best default values for each of the new function parameters `stdv_pct` (Section 5.5.1), `min_thresh` (Section 5.5.1), and `window_size` (Section 5.5.3). While each cube responded differently to various combinations of settings, this testing discovered multiple combinations for each cube that would yield a perfect extraction of the original move sequence. A detailed exploration of this testing and its findings is discussed in Chapter 7.

5.6 Summary

In summary, this chapter demonstrated a proof of concept for a sound analysis algorithm that could detect the face turns of a speedcube equipped with the proper transmitter (the details of which are explored in Chapter 6). This proof of concept algorithm works by computing the spectrogram of the transmitted audio, extracting the most intense frequencies from each time step, converting those dominant frequencies to centerpiece states at that time step, then iterating through that list of states over time to detect changes caused by face turns. Several noise-resisting measures were employed to help mitigate the impact of loud cubes and other background tones. Ultimately, this algorithm design successfully decoded both a purely synthetic audio signal and a realistic audio signal created by re-recording the synthetic audio while solving a speedcube.

Chapter 6

The Transmitter

6.1 Introduction

The transmitter for the sound-based move tracking protocol is in charge of creating the tones representing each centerpiece's current state, and updating those tones each time a centerpiece changes state.

This chapter will detail a prototype PCB design containing only nine discrete components capable of generating all the tones required for encoding one centerpiece's state.

6.2 Requirements

TODO lay out the requirements for the PCB

6.2.1 Prospects of Miniaturization

The transmitter must be both removable and small enough to fit in the center cap of each face of a speedcube. This requirement stems from two sources. First, in contrast to all existing smartcubes, most non-smartcubes have small, solid cores that provide no extra space for the inclusion of any electronics, but do have a small amount of open space within their center cubies. (TODO - add picture of Gans 356 core), Second, the use of a cube with non-removable, embedded electronics violates the WCA competition regulation 2i [5] (See also Section 2.3.2).

6.2.2 Precision of Tone Generation

While the receiver specified in Chapter 5 supports custom state to frequency mappings, it expects that the frequency corresponding to each centerpiece's state stays constant throughout the entire audio recording. As such, the chosen transmitter design can encode centerpiece states with any frequency (assuming the chosen frequencies work within the constraints specified in Section 4.2), but it must produce its chosen frequencies with high precision.

6.2.3 Responsiveness to Face Turns

The chosen transmitter design must respond to an applied face turn by changing the currently transmitted audio frequency to the frequency corresponding to the new centerpiece state.

6.2.4 Signal-to-Noise Ratio

The transmitter must create tones loud enough to be easily distinguished from ambient noise, including the sound of the Rubik's Cube's own turns. In light of the above requirement for the transmitter to fit within a center cubie (6.2.1), this requirement will also require the transmitter design to consider how to overcome any audio dampening caused by such an enclosure.

6.3 Hardware Selection

TODO outline the process of choosing specific hardware to use for this project

6.3.1 Minimizing Sound Obstruction

TODO Discuss the "tupperware" tests -> design of various center caps.

6.4 Prototyping

TODO detail the process of building a prototype. Include pictures of the board and the generated spectrograms.

Chapter 7

Evaluation

TODO review the core goals outlined in the introduction, and methodically review how well my final prototypes stack up against those goals. This section serves to prove (with all the data) that the approach I've described in the previous chapters actually works.

TODO a huge portion of this chapter can come from the "Interesting Graphs for my Thesis" document.

7.1 Compatibility with Standard Speedcubes

TODO discuss how well my design meets this requirement from the Introduction

The design must be deployable within a standard (non-smart) speedcube.

1. The design must not require permanent modifications to the original speedcube.
2. The design must not significantly impact the turn-speed of the original speedcube.

7.2 Move Tracking Accuracy

TODO discuss how well my design meets this requirement from the Introduction

The design must be capable of tracking the face turns of a Rubik's Cube with over 99% accuracy.

7.3 Move Tracking Granularity

TODO discuss how well my design meets this requirement from the Introduction

The design must be capable of recording the time spent executing each individual face turn of a Rubik's Cube.

7.4 Competition Legality

TODO discuss how well my design meets this requirement from the Introduction

The design must be competition legal, meaning it results in a cube that either does not violate existing competition rules against the use of electronics or can be easily modified to regain compliance.

Chapter 8

Conclusion

8.1 Summary

TODO summarize the goals from the Introduction and broadly describe what techniques were most helpful in reaching them. Also detail a summary of exactly how effective they were.

8.2 Limitations

TODO detail the limits to which this research can be more broadly applied. Be clear about the effects various assumptions/decisions have on the reliability of the conclusions of this thesis.

8.3 Ideas for Future Research

TODO If I had more time/resources to work on it, what would I do next with this project?

- Live algorithm - build it small enough to fit in the cube.

Appendix A

Longer Code Snippets

A.1 Spectrogram Generation for Figure 5.5

This code snippet actually generates a full animation of how the component frequencies change over time. Figure 5.5 is only one frame of this animation.

Make sure to run this code after running the code in Section 5.3.1.

```
# Heavily aided by https://matplotlib.org/stable/gallery/animation/simple_anim.html
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def compute_threshold(values: list):
    return -500  # Out of sight of the chart

def animate(wav_path: str, alg: str):
    # Parse the audio sample into its spectrogram
    freq, time, spectrogram = compute_spectrogram(wav_path)

    # Generate the plots
    fig, (ax_spectrogram, ax_fourier) = plt.subplots(2)

    # Set the spectrogram
    ax_spectrogram.specgram(audio_samples, Fs=sample_rate,
        NFFT=SAMPLES_PER_WINDOW,
        noverlap=SAMPLES_PER_WINDOW // 4 * 3)  # 3/4 of the window size
    line_spectrogram, = ax_spectrogram.plot([], [], '-', color='red')
    time_tracker_spectrogram = ax_spectrogram.axvline(0, color='red')
    ax_spectrogram.set_ylim(0, 4000)
    ax_spectrogram.set_title(f"Spectrogram")
    ax_spectrogram.set_ylabel('Frequency [Hz]')
    ax_spectrogram.set_xlabel('Time [sec]')

    # Set the Fourier transform
    STEP = 10
    indices = np.arange(0, len(time) + 1, step=STEP)
    line_fourier, = ax_fourier.plot(freq, spectrogram[0], color='red')
```

```
    threshold_fourier = ax_fourier.axhline(-500, color='green')
    ax_fourier.set_title("")
    ax_fourier.set_xlim(0, 4000)  # Frequency
    ax_fourier.set_xlabel("Frequency [Hz]")
    ax_fourier.set_ylim(0, np.amax(spectrogram))  # Strength
    ax_fourier.set_ylabel("Strength")

    # Space figures nicely
    fig.suptitle(alg)
    fig.tight_layout(h_pad=3)

    def draw_frame_at(i):
        time_tracker_spectrogram.set_xdata(time[i])
        line_fourier.set_ydata(spectrogram[i])
        threshold_fourier.set_ydata(compute_threshold(spectrogram[i]))
        ax_fourier.set_title(f"Frequencies at {time[i]:.2f} seconds [idx={i}]")
        return line_spectrogram, line_fourier

    ani = animation.FuncAnimation(fig, draw_frame_at,
        frames=indices, interval=1, blit=True, save_count=len(time)/STEP)
    ani.save(f"plt/animation/{wav_path[:-4]}.mp4",
        fps=len(time)/STEP/5, dpi=300)
    plt.show()

animate(demo_wav_path, demo_alg)
```

A.2 Spectrogram Generation for Figure 5.7

This code snippet actually generates a full animation of how the component frequencies change over time. Figure 5.7 is only one frame of this animation.

Make sure to run this code after running the code in Section 5.3.2 as the new *compute_threshold* function there is what adds the threshold line to this animation.

```
animate(demo_wav_path, demo_alg)
```

Bibliography

[1] Rubik's Brand Ltd. *About Us*. URL: https://www.rubiks.com/en-us/about. (accessed: 10 August 2021).

[2] Erno Rubik. *Rubik's Cube Patent*. URL: https://www.hipo.gov.hu/hu/anim/pics/HU-170062.pdf. (accessed: 10 August 2021).

[3] Joan Verdon. *Rubik's Cube And Spin Master: A $50 Million Deal With Endless Possibilities*. URL: https://www.forbes.com/sites/joanverdon/2020/11/15/rubiks-cube-and-spin-master-a-50-million-deal-with-endless-possibilities/. (accessed: 10 August 2021).

[4] Rubik's Brand Ltd. *The History of the Rubik's Cube*. URL: https://web.archive.org/web/20180908211659/https://www.rubiks.com/about/the-history-of-the-rubiks-cube. (accessed: 10 August 2021).

[5] The World Cube Association. *WCA Regulations*. May 2021. URL: https://www.worldcubeassociation.org/regulations/#2i. (accessed: 14 Oct 2021).

[6] The Cubicle. *XiaoMi Giiker Cube*. URL: https://www.thecubicle.com/products/xiaomi-giiker-cube/. (accessed: 25 Sep 2021).

[7] Federal Communications Commission. *FCC ID: 2ATHZ-SUPERCUBE - INT PHO*. URL: https://apps.fcc.gov/oetcf/eas/reports/ViewExhibitReport.cfm?mode=Exhibits&calledFromFrame=Y&application_id=xbxQDjxzoHU1WywTbpvIAQ%3D%3D&fcc_id=2ATHZ-SUPERCUBE. (accessed: 24 Sep 2021).

[8] Charlie Eggins and Geoff Eggins. *Giiker Cube Repair for Repeated Out of Sync Issues*. URL: https://swiftcubing.com/2019/10/14/giiker-cube-repair-for-repeated-out-of-sync-issues/. (accessed: 24 Sep 2021).

[9] Go Cube. *Go Cube - Official Kickstarter Video*. URL: https://www.youtube.com/watch?v=tMtmzoC_WUY. (accessed: 26 Sep 2021).

[10] Go Cube. *Rubik's Connected*. URL: https://www.getgocube.com/products/rubiks-connected/. (accessed: 27 Sep 2021).

[11] Federal Communications Commission. *FCC ID: 2ASMEGC33 - Internal photos*. URL: https://apps.fcc.gov/oetcf/eas/reports/ViewExhibitReport.cfm?mode=Exhibits&calledFromFrame=Y&application_id=tdk9t6CDQWEFnu4P%2B1WEbg%3D%3D&fcc_id=2ASMEGC33. (accessed: 24 Sep 2021).

[12] The Cubicle. *Gan 356 I*. URL: https://www.thecubicle.com/products/gan-356i. (accessed: 26 Sep 2021).

[13] Federal Communications Commission. *FCC ID: 2AT27-GAN-3X3 - Internal photos*. URL: https://apps.fcc.gov/oetcf/eas/reports/ViewExhibitReport.cfm?mode=Exhibits&calledFromFrame=Y&application_id=B75M7i7IVYNksPluyUFB1Q%3D%3D&fcc_id=2AT27-GAN-3X3. (accessed: 24 Sep 2021).

[14] IBM. *What is computer vision?* URL: https://www.ibm.com/topics/computer-vision. (accessed: 23 Sep 2021).

[15] Jay Hack and Kevin Shutzberg. *Rubik's Cube Localization, Face Detection, and Interactive Solving*.
[16] OpenAI et al. "Solving Rubik's Cube with a Robot Hand". In: *arXiv preprint* (2019).
[17] Junshen Kevin Chen, Wanze Xie, and Zhouheng Sun. "DeepCube: Transcribing Rubik's Cube Moves with Action Recognition". In: ().
[18] Jiawei Huang et al. "IM3D: Magnetic Motion Tracking System for Dexterous 3D Interactions". In: *ACM SIGGRAPH 2014 Emerging Technologies*. SIGGRAPH '14. Vancouver, Canada: Association for Computing Machinery, 2014. ISBN: 9781450329613. DOI: 10.1145/2614066.2614084. URL: https://doi-org.ezproxy1.lib.asu.edu/10.1145/2614066.2614084.
[19] Maria Mannone et al. "CubeHarmonic: A New Interface from a Magnetic 3D Motion Tracking System to Music Performance". In: *NIME Conference Proceedings*. 2018.
[20] Maria Mannone et al. "CubeHarmonic: a new musical instrument based on Rubik's cube with embedded motion sensor". In: *ACM SIGGRAPH 2019 Posters* (2019).
[21] Jonas Michel. *mobile-acoustic-modems-in-action Wiki*. URL: https://github.com/jonasrmichel/mobile-acoustic-modems-in-action/wiki/Wiki. (accessed: 28 Sep 2021).
[22] Ari Neto. *Twisty Timer*. URL: https://play.google.com/store/apps/details?id=com.aricneto.twistytimer. (accessed: 28 Sep 2021).
[23] SpeedSolving.com. *Home Page*. URL: https://www.speedsolving.com/. (accessed: 28 Sep 2021).
[24] Simone Genovesi et al. "Chipless Radio Frequency Identification (RFID) Sensor for Angular Rotation Monitoring". In: *Technologies* 6.3 (2018). ISSN: 2227-7080. DOI: 10.3390/technologies6030061. URL: https://www.mdpi.com/2227-7080/6/3/61.
[25] The Food and Drug Administration. *Radio Frequency Identification (RFID)*. URL: https://www.fda.gov/radiation-emitting-products/electromagnetic-compatibility-emc/radio-frequency-identification-rfid. (accessed: 28 Sep 2021).
[26] NVE Corporation. *Off-Axis Angle Sensing with NVE Angle Sensors*. URL: https://www.nve.com/Downloads/SB-SA-02_Off-Axis-Angle-Sensing.pdf. (accessed: 06 June 2020).
[27] NVE Corporation. *Calculators/WebApps: Off-Axis Angle Sensing*. URL: https://www.nve.com/spec/calculators#tabs-Off-Axis-Angle-Sensing. (accessed: 06 June 2020).
[28] Carl Reinke. *Spectroid*. URL: https://play.google.com/store/apps/details?id=org.intoorbit.spectrum. (accessed: 19 Oct 2021).
[29] Hristo Bojinov et al. "Mobile Device Identification via Sensor Fingerprinting". In: (Aug. 2014).
[30] Jeffrey Hass. *5. What is Frequency | Page 2*. 2020. URL: https://cmtext.indiana.edu/acoustics/chapter1_frequency2.php. (accessed: 09 Oct 2021).
[31] Carl Rod Nave. *The Piano*. 1999. URL: http://hyperphysics.phy-astr.gsu.edu/hbase/Music/pianof.html. (accessed: 09 Oct 2021).
[32] J Perm. *Rubik's Cube Move Notation*. 2021. URL: https://jperm.net/3x3/moves. (accessed: 10 Oct 2021).

[33] Erik Nyquist. *tones 1.2.0*. 2020. URL: https://pypi.org/project/tones/. (accessed: 29 Nov 2020).

[34] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[35] Steve Brunton. *Fourier Analysis*. Sept. 2021. URL: https://www.youtube.com/playlist?list=PLMrJAkhIeNNT_Xh3Oy0Y4LTj0Oxo8GqsC. (accessed: 20 Oct 2021).

[36] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[37] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.