Real-time Chord Recognition

Adam Belhouchat, Ashish Sareen

***Abstract*—This project describes the development of a real-time chord recognition system, which was built using the TI LCDK embedded chip and Code Composer Studio IDE [1], [2]. Prior chord recognition systems have been developed [3]–[11], but our implementation is specifically intended for use by a musician during the real-time composition process. The subsystems are, in order: recording audio of live chord playing, extracting chroma features, performing chord recognition, and providing output to the user. The output is both in real-time to the computer's console and to an annotated text file after the recording is complete. Our results indicate that as long as the chord is played in time to the program (with live prompts to the user), the system works as intended with minor undesirable effects.**

***Index Terms*—chord, octave, chord recognition, chroma features, pitch class profile, pattern matching, cosine similarity, filter, downsampling, FFT, LCDK**

I. Introduction

A. History with References

Automatic chord recognition (ACR) is the process of determining what chord is being played given an audio sample by labelling the audio sample over time with the appropriate chord [3]. A chord is defined as two or more notes being played at the same time or close together [4]. Chord recognition has many applications, such as music segmentation or determining the similarity between two music samples [6], but one of the most attractive applications is automatic transcription. Manual transcription can be tedious and difficult, and automatic chord recognition systems can help musicians work more quickly and more accurately [7]. Because of its wide range of applications, chord recognition has been the subject of much research in the past few decades. These researchers try to improve both the number of chords their systems can recognize and its accuracy through novel methods and approaches [10].

One of the first ACR systems was developed by Takuya Fujishima in 1999 [10]. He used the discrete Fourier transform (DFT) of a music sample to generate a pitch class profile, also known as chroma feature or chroma vector, which encodes the harmonic information of the sample over twelve pitch classes. The chroma feature is then compared to chord templates, one for each of 27 different chords, and the closest match is chosen as the chord label in a process called pattern matching [11].

Since then, chord recognition has evolved greatly. Pattern matching is simple and effective, but it is not always accurate [9]. Some researchers have found better success using hidden Markov models (HMMs) rather than template matching to perform ACR [12] while others have explored the use of deep learning and neural networks to both detect chords [13] and extract features from the audio sample [14]. However, both HMMs and deep learning require a significant amount of training data to set up, and good training data may be difficult to come by. Manual transcription is prone to errors due to human subjectivity [10] and, as mentioned before, it can be time consuming. Nevertheless, these approaches do show significant accuracy improvements over pattern matching [9], [13]. Each of these approaches has its pros and cons, and which one is most fitting depends on the specific application.

B. Global Constraints

The main constraints dealt with are the hardware limitations of the TI LCDK platform, which is an all-in-one microcontroller (MCU) with digital signal processor (DSP) and ARM processor (CPU), and other common peripheral components [1]. The entire system is built around the operation of the LCDK unit. The two major limitations are the memory size and processor speed. The memory specifications are 128 MB SDRAM and 128 MB NAND Flash [1]. The processor speed is listed as 456 MHz for both the DSP and CPU [1].

These hardware constraints pose implications on the software needed to build the system. The LCDK board is programmed using the TI Code Composer Studio (CCS) IDE in the C programming language [1]. The memory constraints imply that the program's memory allocation must be limited to prevent data overflow. To that end, memory is dynamically allocated when needed and freed immediately after use with the in-built C `malloc()` and `free()` functions. Additionally, since this is a real-time system, the processor speed places a constraint on the execution time of the program. The system leverages the interrupt service routine (ISR) to take audio samples at a sampling rate equal to 16000 Hz. The interrupt loop in the code must have an execution time less than 62.5 microseconds in order to guarantee that samples are taken properly. Functions that take a long time to complete, such as `fprintf()`, are used only at the end of the program when the real-time component is complete.

II. Motivation

One shortcoming of many modern chord recognition systems is the vast amount of data required to train these systems [10]. As mentioned earlier, large datasets of accurately transcribed music are not easy to find, so we wanted to develop a system that would work without training. Additionally, these data-driven approaches are opaque and it is difficult to understand how they determine the chords in an audio sample [10]. Part of our motivation for pursuing this project was to increase our skills in real-time digital signal processing and to understand the concepts behind automatic chord recognition. Throwing data at a neural network until it works does not help us understand chord recognition.

Another shortcoming of HMMs and neural networks is they are often more computationally expensive than pattern

matching [3]. They work well when a music sample has been prerecorded and is later fed into the system, but they are not fast enough to perform chord recognition and transcription in real-time. We wanted to make a chord recognition system that would record musical chords and transcribe them in real-time, so we needed to approach our system differently from most modern systems.

Lastly, as mentioned earlier, we chose this project to learn more about real-time digital signal processing (DSP) and to gain more experiences with different applications of it. Chord recognition requires many DSP techniques, including sampling a signal, processing it to extract features, performing some evaluation based on those features, and then returning the results of that evaluation to the user [11]. This project would give us valuable experience with all the important elements of DSP with the added constraint of performing it in real-time. This required us to perform diligent memory management and pay close attention to our code to make sure all functions were optimized, which are valuable skills for embedded software development.

III. Approach

A. Team Organization

The team for the project consisted of two people, and the majority of the work was done together in the lab during lab time. Having two people worked well, because we could split up the work into its natural software and hardware components. A typical workflow would have one person controlling the software of the LCDK with CCS and the other monitoring the interfaces, such as microphone and speakers, to the LCDK. The software person would write code and run it on the LCDK. The hardware person would be responsible for testing the program by giving the microphone input through recorded or live chord sequences. Being able to incrementally test units in this fashion enabled us to quickly develop and debug our code. We were generally successful at managing our time, utilizing the full extent of our given lab time and not needing to come in after hours. With proper time management and team organization, we were able to accomplish our main goals.

B. Plan and Implementation

Our plan for implementing the real-time chord recognition was to separate the task into major chunks. We would prototype the system in a higher-level language as a proof of concept and then implement the system in C for the LCDK. Our initial plan for this project is outlined in Table I, while the actual timeline of our implementation is outlined in Table II.

Our actual implementation followed our plan well except for two major differences. First, we used Python with the LibROSA to build our prototype instead of MATLAB. As explained in Table II, LibROSA is a more modern library for audio processing that is still maintained as opposed to the deprecated MATLAB libraries we found. It also had better documentation than the MATLAB libraries which helped us understand the principles and operations behind what we were doing.

TABLE I
Outline of our plan for building the chord recognition system.

Weeks 1–2	Find and format data. Find a data set of various guitar chords. Format data into frames.
Weeks 3–4	Generate chroma features. Test feature extraction in MATLAB. Implement final version of feature extraction on LCDK.
Weeks 5–7	Perform chord recognition on chroma features. Test different classification algorithms (template, HMM, linear classifier, etc.) in MATLAB and find the best (most accurate) one. Implement final classification algorithm on LCDK.
Weeks 8–9	Test chord recognition (all on LCDK). Test on data set. Test on live performance. Write output to console.
Weeks 9–10	Write chord sequences to a file. Write a plain text file for testing. Write to standard format (guitar tabs, sheet music, etc.).

TABLE II
Outline of the timeline of our implementation of the chord recognition system.

Weeks 1–2	We found several potential data sets from which samples were taken. Then, we were able to construct a data set with several audio samples for each of the 24 chords that were implemented.
Weeks 3–4	Instead of MATLAB, we used Python to prototype chroma feature extraction. Python had an up-to-date library for chroma processing, and was easier to use than MATLAB [15]. Referring to this library and another paper, we were able to implement feature extraction on the LCDK [3], [15].
Weeks 5–7	Instead of beginning with Python or MATLAB, we started developing on the LCDK to save time. Our first attempt was to adapt a template-matching algorithm, which ended up working very well [3]. It was the simplest to implement and had the lowest computational complexity. We ended up not having to attempt any of the other classifiers, which included linear classifiers, HMM, and SVM.
Weeks 8–9	We tested chord recognition by playing sequences of guitar chords and recording the classification accuracy. We were pleased to find that when played live, the program was able to classify all of the chords.
Weeks 9–10	We were able to write the sequences to a text file by assigning timestamps for the duration that a chord was played. We did not get to our stretch goal, which was to write the sequence to a standard format.

Second, we decided to prototype the whole system in Python first before doing anything on the LCDK instead of taking a more piecemeal approach as we initially planned. This allowed us to gain a better understanding of how the system would work as a whole, such as graphing the chroma features and seeing how similar chroma representations would need to be accounted for during template matching. It also let us debug fundamental issues on Python before we ported it over to the LCDK. Debugging C code on the embedded system would have been much more annoying than debugging Python on a laptop, so this change in our approach gave us a smoother transition into working on the LCDK.

Lastly, we did not meet our stretch goal of outputting the

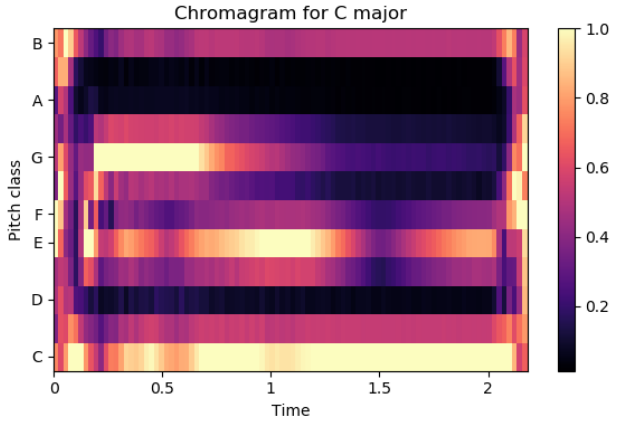

Fig. 1. Chromagram for the C major chord generated using LibROSA. Note that the distortions and noise are due to the chord being played imperfectly on the guitar and due to silence at the beginning before the chord is played and at the end when the chord trails off.

chord sequences into a standard format like sheet music or guitar tabs that could be loaded into a music notation program. When we explored different file formats, we discovered that we would need a lot of additional information that our system did not keep track of to output our chord sequences in these formats, such as clef or time signature. Therefore it would have been much more work to reach our stretch goal and may have fundamentally changed the objective of this project if we were to pursue it. For now, it is left as potential future work.

C. Standard

The most common features used in automatic chord recognition systems to extract chord information from a music sample are chroma features [16]. First introduced into chord recognition by Fujishima [11], chroma features represent the pitch content of a music sample. Each chroma vector has twelve elements, one for each note in the standard Western octave from C to B, and the energy in each element determines what the pitch composition of the chord is and thus determines what the chord is [9]. Basic chroma features are calculated by summing the magnitude of the DFT of the music signal over certain frequency bins [11]. These bins correspond roughly to the energy in the different pitch classes. Because of how ubiquitous and useful chroma features are in chord recognition, we chose to use them as our features for chord recognition.

To actually determine what chord the chroma features corresponds to, we use pattern matching, which is one of the most common methods of determining chords [16]. Specifically, we use binary pattern matching, where each chord has a corresponding template which is a specific arrangement of zeros and ones. Since the chroma vector represents the pitch content of a music sample, we can create ideal template vectors that represent what the pitch content of a pure chord would look like. For example, the major C has pitch contributions from the C, E, and G pitch classes as we can see in Figure 1. Therefore, the template vector for the major C chord would be $\mathbf{v}_{\mathrm{C}} = (1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0)$, with ones at the C, E, and G pitch classes and zeros everywhere else.

In general, the tones that should be present in the ideal chord are one, while all the other tones not present are zero [16]. To find the matching template, many implementations look for the minimum Euclidean distance between a template and the chroma feature [3], but other approaches exist such as finding the template with the minimum angle to the chroma vector [9]. We chose to go with this latter approach as it gave us slightly better accuracy when testing.

D. Theory

The key mathematical elements of our chord recognition system are the low-pass filter, the chroma feature calculations, and pattern matching algorithm. In this section, we will explain in detail the theory behind each of these steps.

The low-pass filter is generated using the bilinear transform, which converts a continuous transfer function to a discrete one. To do so, the bilinear transform uses the approximate map

$$s \to \frac{2}{T}\frac{1 - z^{-1}}{1 + z^{-1}} \tag{1}$$

where T is the sampling period of the discrete signal. We can then substitute this into the continuous transfer function $H_a(s)$ to get the discrete one $H_d(z)$ by

$$H_d(z) = H_a\left(\frac{2}{T}\frac{1 - z^{-1}}{1 + z^{-1}}\right) \tag{2}$$

Once we have $H_d(z)$, we can get the discrete time-domain equation by partial fractions and inverse Z-transform pairs.

In general, the gain of an nth order Butterworth filter with DC gain of 1 is

$$|H(j\omega)|^2 = \frac{1}{1 + (j\omega/j\omega_c)^{2n}} \tag{3}$$

where ω_c is the cutoff frequency. For this project, we used a 6th order Butterworth low-pass filter with a cutoff frequency at 6000 Hz, so our gain would be

$$|H(j\omega)|^2 = \frac{1}{1 + (j\omega/j12000\pi)^{12}}$$

This cutoff frequency preserved most of the important harmonic information while still filtering out high-frequency noise, and the order gave a reasonably steep dropoff around the cutoff frequency without being computationally expensive. From here we used a program which used the bilinear transform to generate the time-domain filter function [17]. The frequency response of the discrete filter is given in Figure 2.

Chroma features are features that describe the pitch content of a sample of music [9]. A chroma feature has 12 elements, one for each note in the standard Western octave from C through B. The value of one of these elements corresponds to the energy present in the music signal that is contributed by that pitch. Because of this, it is straightforward to determine the chords in a piece of music by analyzing the chroma features.

The algorithm we use to generate chroma features comes from Stark and Plumbley's paper [3]. The first step is to apply a Hamming window $w(n)$ to the signal $x(n)$, given by

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{N - 1}\right) \tag{4}$$

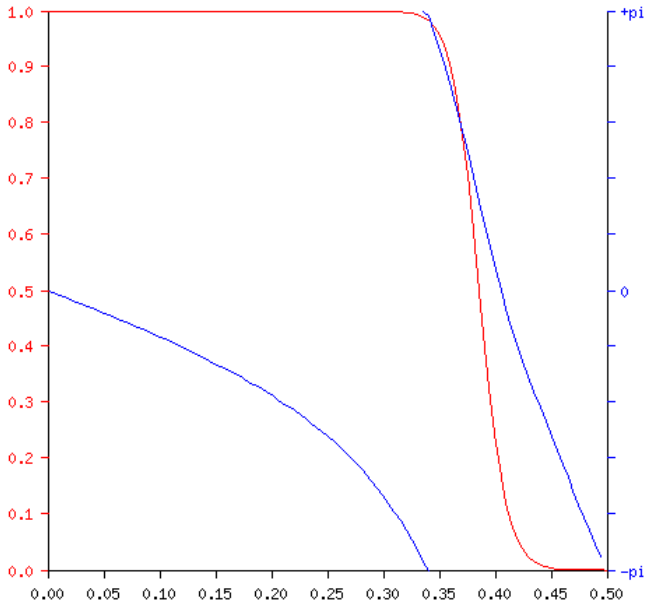

Fig. 2. Frequency response of the low-pass filter used in our chord recognition system. The red curve is the normalized magnitude of the frequency response and the blue curve is the phase. Generated from Tony Fisher's website [17].

where N is the size of the signal. Once we have the windowed signal $x_w(n) = x(n)w(n)$ we take the DFT,

$$X(k) = \sum_{n=0}^{N-1} x_w(n) e^{-\frac{i2\pi kn}{N}} \tag{5}$$

From here, the original chroma feature algorithm would sum the squared magnitudes of the DFT over certain frequency bins to get the chroma values [11]. However, this may include unwanted energy like noise in the chroma features. To avoid this, Stark and Plumbley only consider the maximum amplitude in a given frequency bin, thus only taking into account the energy in the note we want.

First, we determine the frequencies to search over. We start with $f_{\text{C3}} = 130.81$ Hz, which is lower C. Then for $n = 0, 1, \ldots, 11$, we calculate

$$f(n) = f_{\text{C3}} 2^{(n/12)} \tag{6}$$

which gives the 12 notes in an octave. We then do this for two octaves so we get all 24 notes in the two octaves from $f_{\text{C3}} = 130.81$ Hz to $f_{\text{C5}} = 523.25$ Hz. For each of the 12 elements of the chroma feature, we search through two octaves and within each octave search through two harmonics. This is because Stark and Plumbley have found that most instruments of interest use the lower register within this frequency range and to account for inharmonicities in real instruments [3]. The chroma feature is then given by

$$c_n = \frac{1}{h} \sum_{\phi=1}^{2} \sum_{h=1}^{2} \max_{k_0^{(n,\phi,h)} \le k \le k_1^{(n,\phi,h)}} X(k) \tag{7}$$

where c_n are the elements of the chroma vector $\mathbf{c}$ for $n = 0, 1, \ldots, 11$, ϕ is the number of the octave, h is the number of the harmonic, and

$$k_0^{(n,\phi,h)} = k'^{(n,\phi,h)} - rh$$
$$k_1^{(n,\phi,h)} = k'^{(n,\phi,h)} + rh$$

are the lower and upper bounds respectively of the bins to search over. Here, $r = 2$ is the number of bins to search over for each harmonic and

$$k'^{(n,\phi,h)} = \text{round}\left(\frac{f(n)\phi h}{f_s/N}\right)$$

is the central bin where f_s is the sampling rate.

Lastly, we use pattern matching to match the chroma feature to the most fitting chord. Pattern matching involves the use of template vectors for each chord, such as $\mathbf{v}_{\text{C}} = (1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0)$ for major C [9]. We try to match the chroma vector to the closest template vector and from that get the corresponding chord. One simple method is to just find the Euclidean distance $||\mathbf{c} - \mathbf{v}||$ between the chroma vector and each of the template vectors and see which is the smallest. Another approach, which we use, is to find the cosine similarity between two vectors, defined as

$$\text{cossim}(\mathbf{c}, \mathbf{v}) = \frac{\langle \mathbf{c}, \mathbf{v} \rangle}{||\mathbf{c}|| \cdot ||\mathbf{v}||} \tag{8}$$

which finds the cosine of the angle between the two vectors and choose the template vector with the maximum cosine similarity [9].

E. Software/Hardware

There were two main types of software used for the project: the LibROSA Python package for prototyping, and Code Composer Studio (CCS) C software for implementation [2], [15]. The LibROSA Python package was used in the early stages of development for chroma feature extraction. The main motivation was to familiarize ourselves with the chroma structure and the type of inputs/outputs that our program would need. The LibROSA package also provided functions to visualize the contents of a chroma feature, which helped to further understand the nature of the audio transformation [15]. After using this package, we were able to better organize our C code for the LCDK, keeping in mind its real-time constraints.

The majority of the software was done in Code Composer Studio in order to program the LCDK. We were familiar with CCS since we used it extensively in prior projects. The main challenge was to adapt the chroma feature extraction and chord recognition into the C programming language, which requires more low-level knowledge than Python. Fortunately, with prior experience in C/C++ and relative simplicity of the algorithms used, we were able to quickly implement our program. An example code snippet from the chroma feature extraction, containing the `process_audio_frame()` function, found in `chromagram.h`, is provided in Listing 1 at the end of this report.

As aforementioned, the LCDK platform is the embedded system on which the compiled C program operates [1]. The

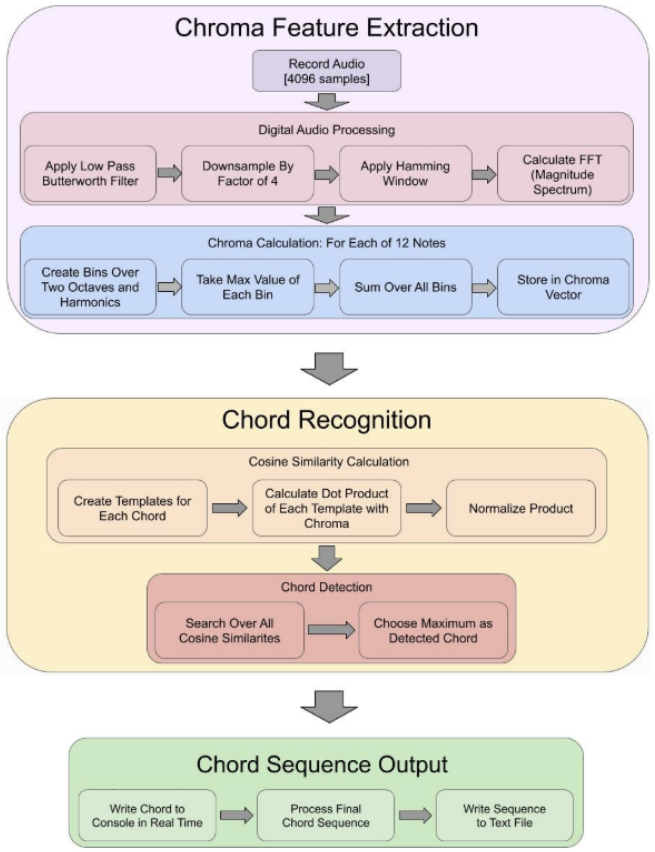

Fig. 3. Block diagram flowchart of the system. There are three main steps: chroma feature extraction, chord recognition, and chord sequence output.

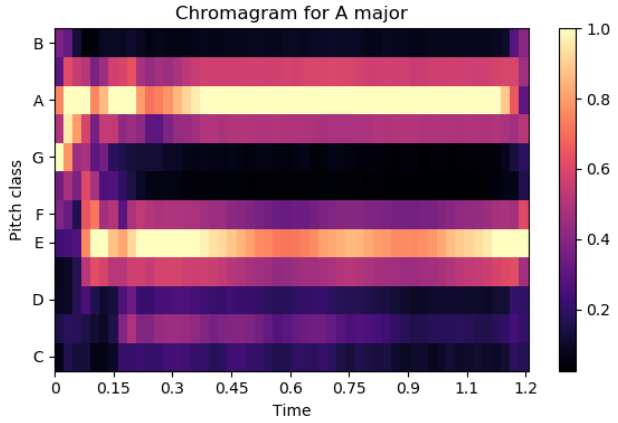

Fig. 4. Chromagram for the A major chord. The distortion at the beginning is due to the initial strum of the guitar. Generated with LibROSA using Python.

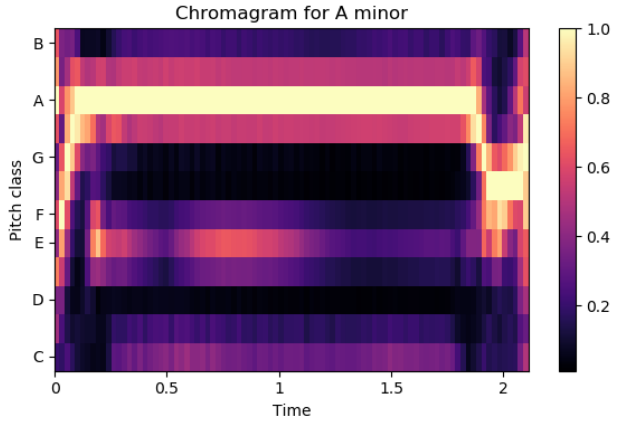

Fig. 5. Chromagram for the A minor chord. The distortions at the beginning and end are due to the initial strum of the guitar and the audio trailing off. Note how two out of the three most prominent notes are the same as for the A major chord. Generated with LibROSA using Python.

main subsystems used were the CPU and DSP, memory, and audio codec. The audio codec was easy to use, since the manufacturer provided the encapsulated functions needed to sample and save audio information to the C built-in data types. Encapsulation enabled us to use the hardware without needing to know many of its low-level details and instead use our knowledge of the software.

F. Operation

1) How the System was Built: We broke our chord recognition system into three major parts based on the three major components we needed to implement: chroma feature extraction, chord recognition, and chord sequence output. The block diagram flowchart outlining our system is given in Figure 3.

The first block we worked on, and the first component that is used in chord recognition, is chroma feature extraction. We first decided to record our audio through a microphone (so we could perform live chord recognition) at a sampling rate of 16000 Hz and to record 4096 samples. This would give us roughly a quarter-second of audio, long enough to achieve good resolution for the chroma features, and the sampling rate was high enough that we could capture important high frequency components to the audio. We also needed the number of samples to be a power of 2 for the FFT and chroma feature calculations. As the microphone was not very high quality, we needed to implement a low-pass filter to remove high frequency noise from the input. Our reasoning for choosing a 6th order Butterworth filter with cutoff at 6000 Hz was given in the Theory section.

After low-pass filtering the recorded audio, we needed to downsample it by a factor of 4, which reduces the size of the audio sample so that subsequent calculations do not take too long. As we want to perform real-time chord recognition, we need the chroma feature calculations to be fast enough that they do not significantly slow down the program's speed. We did attempt to bypass the downsampling step by recording at a lower sampling rate, but this gave very poor results and so we decided to stick with downsampling.

After downsampling the filtered audio signal, we calculate the chroma feature through the algorithm described in the Theory section. Examples of the chroma features are given in Figures 4 and 5 for the A major and A minor chords respectively played on a guitar. For each frame of recorded audio (4096 samples), we generate one chroma vector. Due to the limited memory of the LCDK, we cannot store very many frames of audio and their chroma features at once, so

we run our pipeline one at a time, recording a single frame and generating a single chroma vector and performing chord recognition on that chroma vector before moving on to the next frame of audio.

After we have generated the chroma feature, we move on to chord recognition. We first create the templates for each chord, and we also normalize their magnitudes in order to make comparison easier. After generating the templates, we calculate the cosine similarities between the chroma feature and each of the templates and find the maximum cosine similarity and choose the corresponding chord as the detected chord.

Lastly, we output the detected chord. We first write the chord to the CCS console in real-time in order to provide immediate feedback to the user about which chord is being played. We also wanted to write the chord to a final output file with timestamps of when the chord was played, so we decided to keep track of the chords detected during each time frame by storing them in an array. However, because of this, we now had to limit the runtime of the program to a fixed time. If we were to let the program run indefinitely, we would run into memory issues with a continuously growing and reallocating array, so we decided to force the program to terminate after a fixed number of loops. Therefore, once the program completes the given number of iterations, it processes the final chord sequence to assign timestamps, using the sampling rate and frame size to estimate the length of each loop, and writes the timestamped sequence of chords to a text file.

To make this process easier on the user, we decided after implementing all three blocks to play a "metronome" through speakers connected to the output port of the LCDK. During our initial testing, we discovered that the accuracy of the chord recognition system was highly dependent on whether the chord was played in time with the program's recording loop. If the chord was played at the same time as the LCDK started recording, we would get good performance, but if the two were out of sync then we would have very poor performance. To fix this, our metronome tells the user how to time their audio to achieve good performance. At every loop, the beat plays once the LCDK begins recording the audio frame so that the user can play their audio in time with the recording.

2) How to Use the System: To use the chord recognition system, first plug a microphone into the LCDK microphone input port and plug speakers into the LCDK output port. If you want, you can change the sampling rate, frame size, buffer size, and runtime. The first three parameters are defined in `chromagram.h`: `FS` is sampling rate, `FRAME_SIZE` is frame size, and `BUFFER_SIZE` is buffer size which is fed into the FFT. Note that `FRAME_SIZE` must be a power of 2 and `BUFFER_SIZE` must be one-fourth of `FRAME_SIZE` due to the downsampling. We choose to define them separately just to save that little bit of processing power required to perform the division by four. The program length is implicitly defined by `ITERATIONS` in `main.c`. Strictly speaking, it controls how many loops the program will run for, which is also the number of chords that will be processed.

Once all the parameters are defined, run the program through Code Composer Studio and you will hear the "metronome" start to play through the speakers. The metronome is really just a square wave that plays on beat to when the recording starts. Once you hear the metronome, you can begin playing your audio samples into the microphone. For best results, play the chords in time with the metronome, because otherwise the chord will be out of sync with the recording and the performance of the chord recognition system will be poor.

Once you start the program, the system will write the chords it recognizes to console in real time. Once the program has run through all iterations, it will print "Done :)" at which point you will find a file named `chord_sequence.txt` in the Debug folder. This prints out all the chords that were played during recording along with the times that the chord was played.

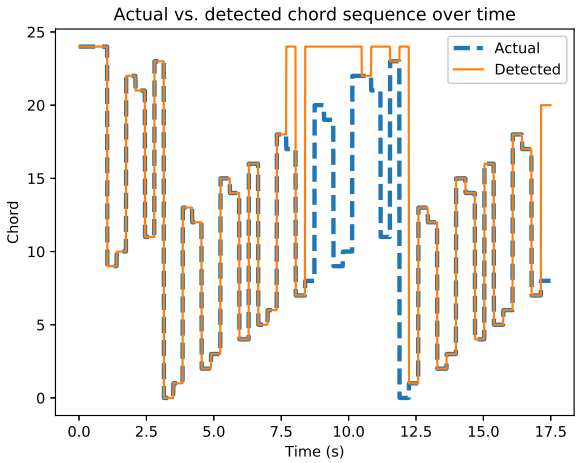

Fig. 6. Graph of the performance of the system for a naturally ordered chord progression.

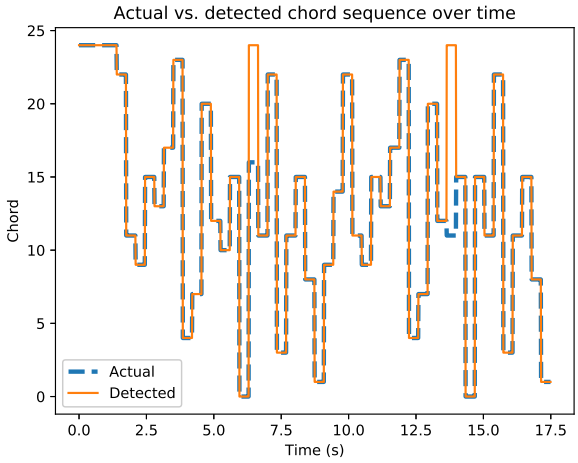

Fig. 7. Graph of the performance of the system for a randomly ordered chord progression.

IV. Results

A. Description

We graphed the performance of our chord recognition system over two different chord sequences, one ordered naturally

in Figure 6 and one ordered randomly in Figure 7. Both were generated from live performance. We follow the graph convention described by [5], where the 24 chords from C major to B minor are numbered 0 to 23, with no chord assigned 24, and we plot the number of each chord over time.

B. Discussion

After performing many tests using different chord sequences, we can properly evaluate the performance of our project. We chose two particular tests to display due to some typical properties that they showcase. The first test, in Figure 6, shows correct recognition of about 80% of the chords played. The behavior in the other 20% is notable, because it is due to playing a note off of the metronome beat. This meant that the full audio frame of that chord was cut off and the program did not have enough audio of decent quality to properly perform chord recognition, so the chord was incorrectly recorded. In most cases where the audio is out of sync with the metronome, the chord is classified as chord 24 or "no chord" as seen in Figure 6. In fewer cases, the chord is classified as the most similar chord, which is usually the minor or major chord in the same key as they share two out of three notes.

The behavior of the test shown in Figure 7 is more representative of the majority of our results. Only two of the chords were incorrectly classified (about 96% accuracy), and both of these chords were classified as "no chord." This is likely due to the same situation as mentioned above, in which the frame was cut off. We found that when the chord is played right on the metronome beat, it is nearly always classified correctly. Only when the full audio frame is misrepresented due to being cut off, does the recognition algorithm fail.

Another reason that could potentially incur failure is improper calibration of the microphone. Our low pass filter was needed to filter out noise inherent to our model of microphone. It is possible that another microphone would need more or less aggressive filtering. We used an average to lower-end microphone, so we estimate that few modifications would be needed to be made to the filter. Use of a higher quality microphone would most likely not worsen performance.

References

[1] *OMAP-L138/C6748 Low-Cost Development Kit (LCDK) User's Guide*, Texas Instruments, September 2019.

[2] (2020) Code Composer Studio (CCS) Integrated Development Environment (IDE). Texas Instruments. [Online]. Available: https://www.ti.com/tool/CCSTUDIO

[3] A. M. Stark and M. D. Plumbley, "Real-time chord recognition for live performance," in *Proceedings of the 2009 International Computer Music Conference, ICMC 2009*, 2009, pp. 85–88.

[4] T. Cho and J. P. Bello, "On the relative importance of individual components of chord recognition systems," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 22, no. 2, pp. 477–492, Feb 2014.

[5] C. Harte, "Towards automatic extraction of harmony information from music signals," Ph.D. dissertation, Department of Electronic Engineering, Queen Mary, University of London, 2010.

[6] K. Lee, "Automatic chord recognition from audio using enhanced pitch class profile," in *ICMC*, 2006.

[7] M. Mauch, "Automatic chord transcription from audio using computational models of musical context," Ph.D. dissertation, School of Electronic Engineering and Computer Science Queen Mary, University of London, 2010.

[8] M. Mauch and S. Dixon, "Simultaneous estimation of chords and musical context from audio," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 18, no. 6, pp. 1280–1289, Aug 2010.

[9] N. Jiang, P. Grosche, V. Konz, and M. Müller, "Analyzing chroma feature types for automated chord recognition," in *Audio Engineering Society Conference: 42nd International Conference: Semantic Audio*. Audio Engineering Society, 2011.

[10] J. Pauwels, K. O'Hanlon, E. Gómez, and M. B. Sandler, "20 years of automatic chord recognition from audio," in *ISMIR*, 2019.

[11] T. Fujishima, "Real-time chord recognition of musical sound: A system using common lisp music," *Proc. ICMC, Oct. 1999*, pp. 464–467, 1999.

[12] A. Sheh and D. P. Ellis, "Chord segmentation and recognition using EM-trained hidden Markov models," 2003.

[13] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent, "Audio chord recognition with recurrent neural networks," in *ISMIR*, 2013.

[14] F. Korzeniowski and G. Widmer, "Feature learning for chord recognition: The deep chroma extractor," 2016.

[15] B. McFee, C. Raffel, D. Liang, D. Ellis, M. McVicar, E. Battenberg, and O. Nieto, "librosa: Audio and music signal analysis in python," in *Proceedings of the 14th Python in Science Conference*. SciPy, 2015. [Online]. Available: https://doi.org/10.25080%2Fmajora-7b98e3ed-003

[16] T. Cho, R. Weiss, and J. Bello, "Exploring common variations in state of the art chord recognition systems," in *Proceedings of the 7th Sound and Music Computing Conference, SMC 2010*. Sound and music Computing network, 2010, p. 31, 7th Sound and Music Computing Conference, SMC 2010 ; Conference date: 21-07-2010 Through 24-07-2010.

[17] T. Fisher. (1999, 10) Interactive digital filter design. University of York. [Online]. Available: https://www-users.cs.york.ac.uk/~fisher/mkfilter/

```
void process_audio_frame(int16_t* input_audio_frame)
{
    /*
     * Processes the audio input and generates the chromagram.
     *
     * First downsamples the input audio signal and calculates
     * the magnitude of the FFT, then generates the chromagram.
     */

    chroma_ready = 0;

    downsampled_audio_frame_size = FRAME_SIZE / 4;
    downsampled_input_audio_frame = (float*)
    malloc(downsampled_audio_frame_size * sizeof(float));
    downsample_frame(input_audio_frame);

    int i;
    for (i = 0; i < BUFFER_SIZE; i++)
    {
        x_sp[2 * i] = downsampled_input_audio_frame[i];
        x_sp[2 * i + 1] = 0;
    }

    free(downsampled_input_audio_frame);
    calculate_chromagram();
}
```

Listing 1. An example code snippet from the chroma feature extraction containing the `process_audio_frame()` function. This function is located in the `chromagram.c` file.