Real-time Chord Recognition

Adam Belhouchat, Ashish Sareen

***Abstract*—Blahdy blah.**

***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, labelling the audio sample over time with the appropriate chord [1]. A chord here is defined as two or more notes being played at the same time or close together [2]. Chord recognition has many applications, such as music segmentation or determining the similarity between two music samples [3], but one of the most attractive applications is in automatic transcription. Manual transcription can be tedious and difficult, and automatic chord recognition systems can help musicians work more quickly and more accurately [4]. Because of its wide range of applications, chord recognition has been the subject of much research in the past few decades, trying to improve the number of chords it can recognize and its accuracy through novel methods and approaches [5].

One of the first ACR systems was developed by Takuya Fujishima in 1999 [5]. 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 different pitch classes. The chroma feature is then compared the templates, one for each of 27 different chords, and the closest match is chosen as the chord label in a process called pattern matching [6].

Since then, chord recognition has evolved greatly. Pattern matching is simple and effective, but it is not the most accurate [7]. Some researchers have found great success using hidden Markov models (HMMs) rather than template matching to perform ACR [8], and others have explored the use of deep learning and neural networks both to detect chords [9] and to extract features from the audio sample [10]. However, both HMMs and deep learning require a significant amount of training data, and good training data may be difficult to come by. Manual transcription is prone to errors due to human subjectivity [5] and as mentioned before, it can be time consuming. Nevertheless, these approaches do show significant accuracy improvements over pattern matching [7], [9]. 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 [11]. 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 [11]. The processor speed is listed as 456 MHz for both the DSP and CPU [11].

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 [11]. 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. 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 the sample 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 [5]. 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 [5]. Part of our motivation for pursuing this project was to increase our skills in real-time digital signal processing and to understand how automatic chord recognition works. 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 [1]. 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 on those features, and then returning the results of that evaluation to the user [6]. 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 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 initial plan was as follows:

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.).

Our actual implementation went as follows:

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 [12]. Referring to this library and another paper, we were able to implement feature extraction on the LCDK [1], [12].

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 [1]. 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.

C. Standard

The most common features used in automatic chord recognition systems are chroma features [13]. First introduced by Fujishima, chroma features represent the pitch content of a music sample. Basic chroma features are calculated by summing the magnitude of the DFT of the music signal over certain frequency bins [6]. These bins correspond roughly to the energy in different pitch classes, such as C or G#. Each chroma vector has twelve elements, one for each note in an octave, corresponding to the twelve pitch classes that the frequency bins are mapped to, and the distribution of energy across the twelve pitch classes is used to determine which chord was played in the music sample [7]. 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 [13]. In particular, 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 template vector for the major C chord would be $\mathbf{v}_C = (1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0)$, where the ones are where a major C should have pitch contributions and the zeros are pitches that should not be present in a major C. In general, the tones that should be present in the ideal chord are one, while all the other tones are zero [13]. To find the matching template, many implementations look for the minimum Euclidean distance between a template and the chroma feature [1], but other approaches such as finding the minimum angle between the two are also used [7]. 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 features, and pattern matching. In this section, we will explain in detail the theory behind each of these steps.

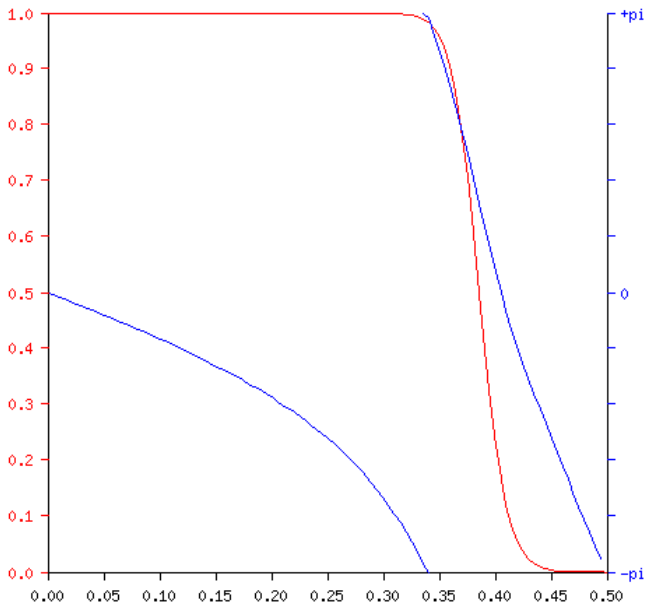

Fig. 1. Frequency response of the low-pass filter used in our chord recognition system. The red curve is the normalized magnitude of the frequency response and the blue curve is the phase. Generated from Tony Fisher's website at https://www-users.cs.york.ac.uk/~fisher/mkfilter/.

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.

We used a 6th order Butterworth low-pass filter with a cutoff frequency at 6000 Hz. 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. The actual transfer function is too large and complex to show here, so we used a program to generate the time-domain filter function. The frequency response is given in Figure 1.

Chroma features are features that describe the pitch content of a sample of music [7]. A chroma feature has 12 elements, one for each note in an octave from C through B. The value of one of these elements corresponds to the energy present in the music signal coming from 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 [1]. The first step is to apply a Hamming window $w(n)$ to the signal $x(n)$, given by

$$w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right) \tag{3}$$

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,

$$X(k) = \sum_{n=0}^{N-1} x_w(n)e^{-\frac{i2\pi kn}{N}} \tag{4}$$

From here, the original chroma feature algorithm would sum the squared magnitudes of the DFT over certain frequency bins to get the chroma values [6]. 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 $f_{C3} = 130.81$ Hz, which is lower C. Then for $n = 0, 1, \ldots, 11$, we calculate

$$f(n) = f_{C3}2^{(n/12)} \tag{5}$$

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 $f_{C3} = 130.81$ Hz to $f_{C5} = 523.25$ 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 [1]. The chroma feature is then given by

$$c_n = \frac{1}{h}\sum_{\phi=1}^{2}\sum_{h=1}^{2}\max_{k_0^{(n,\phi,h)} \le k \le k_1^{(n,\phi,h)}} X(k) \tag{6}$$

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

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

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

where $r = 2$ is the number of bins to search over for each harmonic and

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

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 $\mathbf{v}_C = (1,0,0,0,1,0,0,1,0,0,0,0)$ for major C [7]. 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

$$\text{cossim}(\mathbf{c}, \mathbf{v}) = \frac{\langle \mathbf{c}, \mathbf{v} \rangle}{||\mathbf{c}|| \cdot ||\mathbf{v}||} \tag{7}$$

and choose the template vector with the maximum cosine similarity [7].

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. 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. 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. The 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:

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 how long the program runs for. 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.

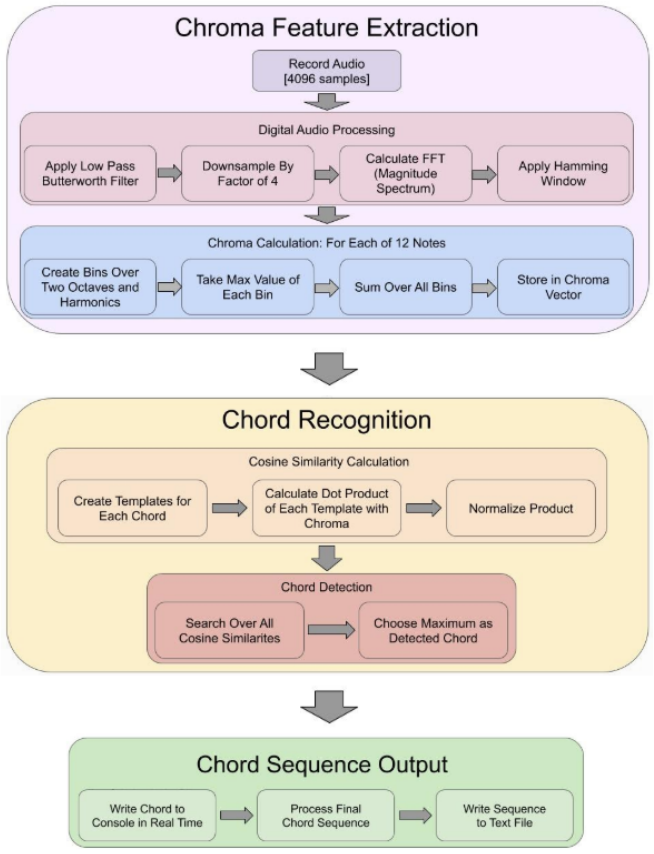

Fig. 2. Block diagram of the system. There are three main steps: chroma feature extraction, chord recognition, and chord sequence output.

Once all the parameters are set to your liking, 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 has recognized to console. The most recent message will be the chord that was just played and recorded. 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.

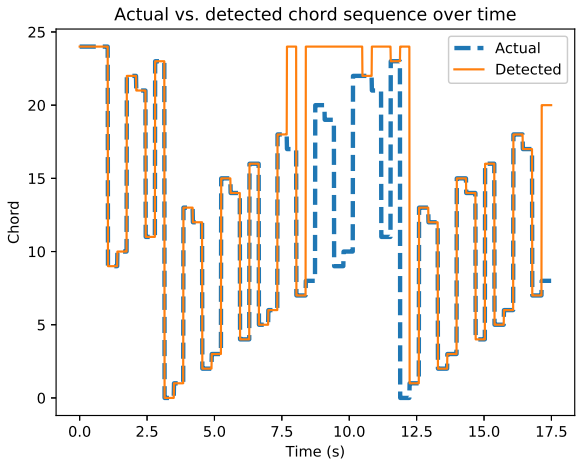

Fig. 3. Graph of the performance of the system for a naturally ordered chord progression.

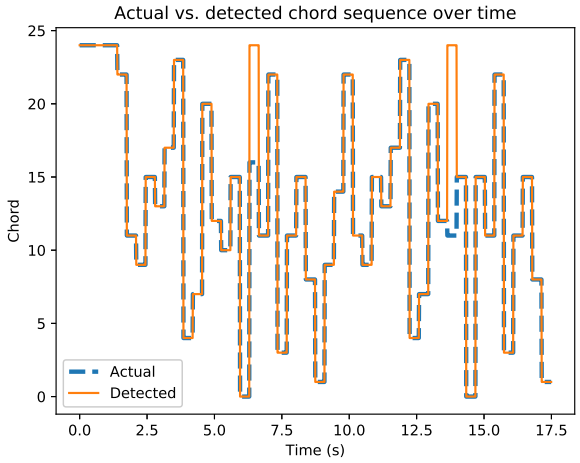

Fig. 4. Graph of the performance of the system for a randomly ordered chord progression.

IV. Results

A. Description

We graphed the performance of our chord recognition system over two different chord sequences, one ordered naturally in Figure 3 and one ordered randomly in Figure 4. We follow the graph convention described by [14], 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

References

[1] 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.

[2] 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.

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

[4] 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.

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

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

[7] 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.

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

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

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

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

[12] 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

[13] 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.

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

```
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.