Simplifying Chord Recognition: A Case Study with NEK

Chukwuemeka Nkama
Lagos, Nigeria
nkcemeka@gmail.com

***Abstract*—Popular educational applications like Scaler 2, Chordie and Midiculous have significantly aided music instructors in teaching complex concepts, with chord recognition being a crucial component. However, their commercial focus often results in proprietary, hidden code, making it challenging for developers to understand the intricacies of chord detection. This paper uses NEK, an open-source MIDI and audio chord recognition plugin, as a case study to explore the technical aspects of chord detection. By examining NEK, we aim to provide insights that will help developers build comparable educational tools.**

NEK available on Github: https://github.com/Nkcemeka/NEK

***Index Terms*—NEK, JUCE, Chroma, MIDI**

I. Introduction

Chord recognition refers to the prediction of time-synchronized chord labels given a music audio recording [1]. It is an important task since chords are a high-level representation of the harmonic structure of a piece of music. Additionally, chords play a useful role in various musical applications including key detection, structural segmentation [2], cover-song identification and automatic-lead sheet creation [1].

Automated chord recognition aims to automatically recognize the chords corresponding to a sequence of audio frames. This is essential since manually annotating chords can be laborious and will often require solid musical expertise [1]; also, different annotators might interpret each section of a score differently.

A typical chord recognition system consists of two main steps: feature extraction and pattern matching. In the feature extraction phase, an audio recording is broken into frames, each of which is transformed into a feature vector [3]. During the pattern matching phase, each feature vector is mapped to a set of predefined chord labels. Extensive research has been conducted on automatic chord recognition with techniques ranging from template matching [3] to more complex methods such as hidden markov models, convolutional neural networks, recurrent neural networks and bi-directional transformers [1].

In this paper, we redefine chord recognition as a process that takes an audio or midi input signal and outputs a corresponding chord label. This means the chord recognition system described here encapsulates both audio and MIDI (a communication protocol) inputs. Given a set of MIDI events, our system should be able to assign chord labels to three or more notes with the same onset time. For an audio signal, we assume that the signal contains a recording of a single polyphonic instrument. The approach presented does not evaluate the complexities involved with multiple instruments playing together at once.

The rest of this paper is structured as follows. Section II details the chord recognition process for both MIDI and audio. Section III presents key insights and finally, Section IV concludes with a summary of our findings and their implications.

II. Approach

To explain how chord recognition works, we will be using the implementation of NEK, an open-source plugin as a basis for discussion. NEK is a music education software plugin that functions as either a MIDI or Audio chord recognition tool [4]. In the case of MIDI, it reads MIDI events and outputs the corresponding chord name. If no chord is detected, it simply outputs the notes being played.

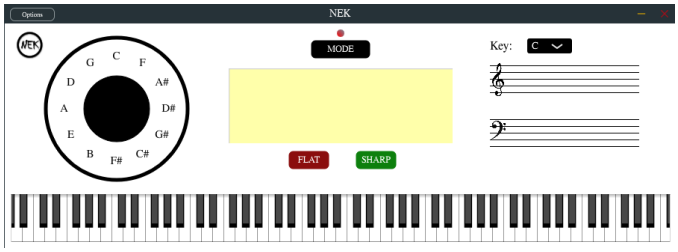

Fig. 1. The NEK plugin interface

For audio, NEK breaks the signal into frames and assigns chord labels to each frame. This is based on an algorithm designed by Adam Stark, which uses a chroma-based method that calculates the energy in the harmonics within a given range [5]. NEK's MIDI functionality can identify a wide array of chord types, including altered chords. However for audio chord recognition, the current implementation is limited to major triads, minor triads, major sevenths, minor sevenths and dominant sevenths. In the following subsections, we will explore how NEK works in both MIDI and audio modes.

A. MIDI Chord Recognition

The MIDI protocol is an ubiquitous tool used to transfer musical information [6]. MIDI stands for Musical Instrument Digital Interface and was invented to allow different instruments communicate electronically.

From a simplified perspective, a MIDI message consists of three bytes: one status byte and two data bytes. The status byte denotes the command type and the channel on which it must be performed. Common commands include *note on*, *note off*, *pitch bend* and various *system messages*. For chord recognition purposes, we are primarily concerned with *note on* and *note off* messages. Fortunately, plugin frameworks like JUCE [7] offer functions that easily allow us to identify a message's type.

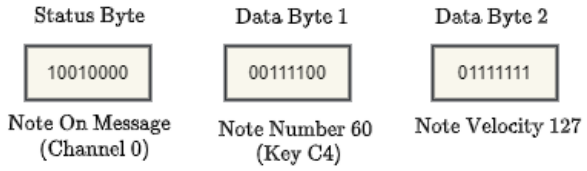

Fig. 2. Bytes in a Midi Message

Figure 2 shows an example of a typical Midi Message. The first four bits in the status byte indicate that it is a Note On message with the latter referring to the channel number. The two data bytes, on the other hand, tell us that the note is C4 and that it has a note velocity of 127 (the highest possible "loudness" a note can have).

Using Figure 3 as a guide, the MIDI chord recognition process in NEK can be easily exemplified. First, the chord recognition system receives MIDI input from an external source. This input can come from a keyboard, a mouse, or even MIDI events within a DAW.

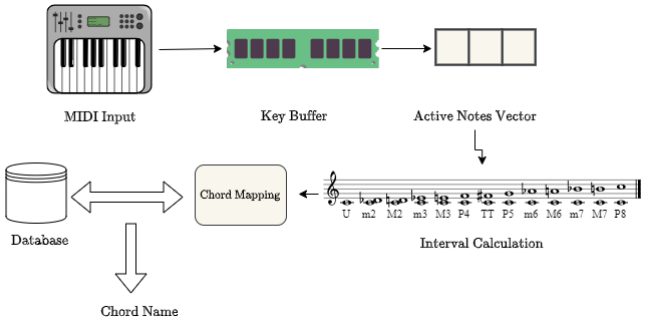

Fig. 3. MIDI Chord Recognition Schematic

The information from these messages is stored in a data structure known as the *Key Buffer*. This is a contiguous memory space (an array or vector) of 128 elements, each representing one of the 128 possible MIDI notes (sufficient to cover the range of notes on a standard digital piano). Each element in the buffer is a boolean that denotes whether a specific note is currently active (on) or inactive (off).

If a MIDI message specifies a *note On*, the corresponding position in the *Key Buffer* will be set to *true*. If the message indicates a *note Off*, the corresponding position in the buffer will be set to *false*. Using Figure 2 as an example, it means the 60th position in the buffer, which is key C4, will be set to *true*.

With the information stored in the *Key Buffer*, we can store all active notes in a vector. The first note in the vector is treated as the root note, and intervals between the root and the other notes in the vector are calculated. Based on these interval relationships, NEK searches its chord database for a match. If a match is found, it outputs the corresponding chord name. If no match is found, the system selects the next active note as the root and repeats the process until all active notes have been evaluated. The pseudocode for this algorithm is shown in Algorithm 1, with further details to be discussed in section III. It should be noted that tools like Chordie and Midiculous perform MIDI chord recognition. Unlike Scaler 2, they lack the ability to detect chords from an audio signal.

B. Audio Chord Recognition

NEK's audio chord recognition system implements Adam Stark's algorithm which was developed to improve on pre-existing frame-based classifiers [5]. While Stark's method is a good starting point, it is not the most advanced option available today. Techniques such as Hidden Markov Models (HMMs) or Deep Learning models can potentially offer better performance. Nevertheless, Stark's algorithm is highly accessible, particularly since it is open-source and available under the GPL license.

The algorithm operates by first extracting a chroma vector, which is a 12-dimensional vector, with each element corresponding to the energy present in a particular pitch class (see Figure 4). These pitch classes are: [C, C#/Db, D, D#,Eb, E, F, F#/Gb, G, G#/Ab, A, A#/Bb, B].

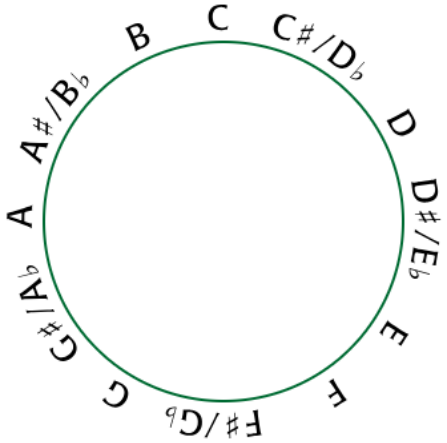

Fig. 4. A Diagram showing the 12 Pitch Classes

The method for calculating the energy in each pitch class varies, and Stark's approach is given by the following equation

[5]:

$$C(n) = \Sigma_{\phi=1}^{2}\Sigma_{h=1}^{2}(\max_{k_0^{(n,\phi,h)} \leq k \leq k_1^{(n,\phi,h)}} X(k))(\frac{1}{h}) \quad (1)$$

Here $n = 0, 1, ..., P-1$ where $P = 12$ is the number of pitch classes in an octave and ϕ is the octave under consideration. The harmonic number is denoted by h and $r = 2$ is the number of bins on either side of a given frequency to search for a maximum peak. The variables $k_0^{(n,\phi,h)} = k'^{(n,\phi,h)} - (r \cdot h)$ and $k_1^{(n,\phi,h)} = k'^{(n,\phi,h)} + (r \cdot h)$ define the range of frequency bins to search given that

$$k'^{(n,\phi,h)} = \text{round}(\frac{f(n) \cdot \phi \cdot h}{(\frac{f_s}{L})}) \quad (2)$$

where f_s is the sampling frequency and L is the frame size. For a more in-depth mathematical explanation, Stark's original work [5] provides comprehensive details.

Once the chroma vector is computed for each audio frame, the next step is classification. The goal is to map the chroma vector to a corresponding chord label. For example, if the vector indicates a C Major chord, we predict its label as C Major. Adam Stark's method does this by classifying the chroma vector based on the residual energy in the chromagram [5]. Essentially, a bit mask is used to mask out notes (or pitch classes) hypothesized to be in the chord and then the residual energy outside of the mask is then minimized. The equation for calculating this residual energy is:

$$\delta_i = \frac{\sqrt{\Sigma_{n=0}^{P-1}\hat{T}_i(n)(C(n))^2}}{(P - N_i)(\beta)} \quad (3)$$

where C is the chroma vector, $\hat{T}_i(n) = 1 - T_i(n)$ with T_i as the ith bit mask, N_i is the number of notes in the ith bit mask, β is the bias factor (more on this in section III) and $P = 12$, the number of notes in an octave. The chord that minimises δ_i is chosen as the chord label for the given frame. It should be noted that one does not need to have a thorough understanding of these equations. However, having an idea of what they mean can help in effectively fine-tuning the implementation, if necessary.

III. Discussion

A. Discussion on the MIDI Recognition Algorithm

Algorithm 1 outlines the MIDI chord recognition algorithm used for identifying chords from a set of MIDI events, given the following input parameters: *noteBuffer*, *keyBuffer*, *intervalMap*, *chordMap*, *noteMap*, *noteMapF*, *reverseMap*, *flats* and *rootNote*.

The **keyBuffer** is a vector representing the status of all MIDI notes (128 of them); **noteBuffer** condenses the information in the keyBuffer into 12 pitch classes, representing the active notes in the current context; **intervalMap** maps distances between notes to interval names; **chordMap** maps interval sequences to chord names; **reverseMap** maps intervals to their reversed intervals (for example, the reverse interval of a minor 2nd is a major 7th); **flats** is a boolean indicating whether or not to use the flat accidental when naming a chord and **rootNote** is an integer for storing the rootNote index. This index can be a value from 0 to 11, which represents each pitch class.

Algorithm 1: MIDI Chord Recognition Algorithm

Require: noteBuffer, keyBuffer, intervalMap, chordMap, noteMap, noteMapF, reverseMap, flats, rootNote
Ensure: Chord name as a string

```
Initialize noteBuffer with zeros
for i = 0 to 127 do
    midiNum ← i mod 12
    noteBuffer[midiNum] ← noteBuffer[midiNum] + keyBuffer[i]
end for
Initialize empty list notePos
for i = 0 to 11 do
    if noteBuffer[i] is true then
        Append i to notePos
    end if
end for
rootNote ← −1
for i = 0 to size of notePos do
    intervals ← empty string
    root ← notePos[i]
    for j = i + 1 to i (looping through notePos) do
        if j == size of notePos then
            j = 0
        end if
        distance ← dist(root, notePos[j])
        if j < i then
            revInterval ← reverseMap[intervalMap[distance]]
            intervals ← intervals + revInterval
        else
            intervals ← intervals + intervalMap[distance]
        end if
    end for
    chordName ← chordMap[intervals]
    if chordName is not empty then
        rootNote ← root
        if flats is true then
            return noteMapF[root] + chordName
        else
            return noteMap[root] + chordName
        end if
    end if
end for
return empty string
```

Here is an itemized overview of what the algorithm does:

- The algorithm starts off by initializing the noteBuffer to zeros. This helps prevent unexpected behavior when holding down certain notes and adding others on top.
- Active note states or pitch classes in noteBuffer are

collated in a vector called **notePos**.

- Assuming each note in **notePos** as the potential root, intervals are calculated between this root and other active notes.
- An interval sequence string is then constructed and passed to chordMap. If a match is found, rootNote is set and the chord name is returned, prefixed by the letter name of rootNote. If a match is not found, a new root is assumed and the process repeats till all notes are exhausted.

Now it should be noted that the above algorithm automatically takes care of chord inversions. Since the information in keyBuffer has been condensed to noteBuffer, it means a *wrapping around effect* occurs because noteBuffer has a size of 12. For example, if the chord F-A-C-E is played, it would appear in the noteBuffer as C-E-F-A which is technically an inverted F Major 7. To account for inversions, the intervallic distance from the root note is calculated in the order from left to right. If the note considered is of a lower index than the root, the intervals are reversed. This *handy trick* helps us to account for all possible inversions of any chord without having to indicate them in the chord database. If no chord is detected, the algorithm returns an empty string; however, NEK, returns the notes being played [4].

B. Discussion on the Audio Chord Recognition Process

The audio chord recognition system implemented by NEK offers a way for any developer to quickly setup an audio-based chord detection system. To begin, you simply need to integrate the C++ code from Adam Stark's code repository into your chosen audio framework [8]. This is not hard to setup in JUCE; see NEK's implementation for this [4]. [8] indicates you might need to build certain FFT (Fast Fourier Transform) libraries, but the repository includes KissFFT header files, which can be integrated easily, avoiding the need to build a library from scratch.

Despite the overall effectiveness of this algorithm, it is prone to occasional misclassification due to the presence of ghost notes [5]. The default β (bias) value in the original implementation is set at 1.06, which is also used by NEK. This value helps mitigate the presence of ghost notes, as increasing the bias reduces the residual energy. However, this does not fully solve the problem and additional approaches like reducing the energy in the seventh chroma bin [5] or even in the fourth chroma bin can be considered [9]. NEK uses the original implementation but altering these parameters must be done carefully, as incorrect adjustments can lead to a lot of wrong chord labels. Another useful tip is setting a minimum RMS amplitude threshold to filter out low-energy signals. NEK uses a value of 0.001, which helps in smoothening the classification process by ignoring low-energy frames that would cause noise in the predictions.

IV. Conclusion

In this short paper, we have explored a chord recognition system that works with both MIDI and audio inputs, using NEK as a reference.

The MIDI chord recognition algorithm leverages the intervallic distance between pitch classes to assign chord labels to a set of MIDI events. In contrast, the audio chord detection implementation analyzes an audio signal and assigns chord labels to each frame based on a chroma vector, which reflects the harmonic energy corresponding to all pitch classes. These approaches are not only effective but offer a solid foundation for developers interested in delving deeper into the complexities of chord recognition.

Acknowledgment

A special thanks to the Audio Developer Conference for their initiatives in fostering the growth of developers in the audio industry. Both this work and NEK are direct outcomes of its mentorship program. Also, a big thank you to Adam Stark; his work on real-time chord detection was highly instrumental to the development of NEK.

References

[1] J. Park, K. Choi, S. Jeon, D. Kim, and J. Park, *A bi-directional transformer for musical chord recognition*, Jul. 5, 2019. DOI: 10.48550/arXiv.1907.02698. arXiv: 1907.02698. [Online]. Available: http://arxiv.org/abs/1907.02698 (visited on 10/19/2024).

[2] X. Zhou and A. Lerch, *Chord Detection Using Deep Learning*. Jan. 1, 2015.

[3] M. Müller, *Fundamentals of Music Processing: Audio, Analysis, Algorithms, Applications*. Cham: Springer International Publishing, 2015, ISBN: 978-3-319-21944-8 978-3-319-21945-5. DOI: 10.1007/978-3-319-21945-5. [Online]. Available: https://link.springer.com/10.1007/978-3-319-21945-5 (visited on 10/19/2024).

[4] N. Chukwuemeka, *Nkcemeka/NEK*, original-date: 2024-08-28T17:57:02Z, Oct. 17, 2024. [Online]. Available: https://github.com/Nkcemeka/NEK (visited on 10/19/2024).

[5] A. M. Stark and M. D. Plumbley, "Real-time chord recognition for live performance," presented at the International Conference on Mathematics and Computing, Aug. 16, 2009. [Online]. Available: https://www.semanticscholar.org/paper/Real-Time-Chord-Recognition-for-Live-Performance-Stark-Plumbley/472f436ad317b23d7bd1ad53fcb4f23e8f0997a7 (visited on 10/19/2024).

[6] "Essentials of the MIDI protocol." (), [Online]. Available: https://ccrma.stanford.edu/~craig/articles/linuxmidi/misc/essenmidi.html (visited on 10/14/2024).

[7] JUCE. "Home," JUCE. (), [Online]. Available: https://juce.com/ (visited on 10/19/2024).

[8] A. Stark, *Adamstark/chord-detector-and-chromagram*, original-date: 2014-11-21T13:59:22Z, Oct. 10, 2024. [Online]. Available: https://github.com/adamstark/Chord-Detector-and-Chromagram (visited on 10/20/2024).

[9] R. B. Dannenberg, *Rbdannenberg/arco*, original-date: 2022-04-17T22:47:18Z, Oct. 18, 2024. [Online]. Available: https://github.com/rbdannenberg/arco (visited on 10/20/2024).