Converting audio to MIDI files

Urša Kumelj
Laboratory assignment for Computer based sound production
Fakulteta za računalništvo in informatiko
Univerza v Ljubljani

***Abstract*—This paper addresses the challenge of transforming audio samples into MIDI data. Initially, we focus on converting a single-instrument audio sample using appropriate techniques and analyzing the accuracy of the conversion. Subsequently, we extend our approach to converting an entire song into a MIDI file, evaluating the accuracy of this process. Finally, we present a brief showcasing of the constructed MIDI file.**

I. Introduction

MIDI (Musical Instrument Digital Interface) is a technical standard that describes a communication protocol, digital interface, and electrical connectors that connect a wide variety of electronic musical instruments, computers, and related audio devices for playing, editing, and recording music. It was created in the early 1980s to address the need for a standardized way for electronic musical instruments and computers to communicate with each other. Before MIDI, different manufacturers used proprietary protocols, making it difficult for devices to work together. Visualizing MIDI files provides a clear and intuitive way to understand and analyze musical compositions, making it easier to identify patterns, dynamics, and relationships between notes. This visualization helps musicians and producers quickly spot errors, enhance their arrangements, and gain insights into the structure and complexity of the music. MIDI data is stenographic representation of music recorded in numerical form. In this paper, we are concerned with the processing necessary to extract musical information from audio signals to automate the creation of a MIDI representation from a music signal. Since MIDI messages specify performance events such as notes, their onset times, durations, and pitches, it is essential to estimate these parameters accurately from the audio signals.

II. Related work and goal

Our goal is to convert an audio file into a MIDI file. Existing approaches, including our method, perform well in detecting single notes from single-instrument audio using harmonic analysis and pitch detection aided by a dictionary of harmonic spectra. However, these methods struggle with the complexity of full songs, where overlapping frequencies and noise reduce accuracy. To address this, we will utilize the YIN algorithm to enhance pitch detection and apply it to the analysis of complete musical pieces.

III. Methods

We will explain how our work was done, including details about the two approaches we used.

A. First approach

As outlined in the goals section, the initial step in our approach was to develop a Python function that extracts musical notes from an audio file using the Fast Fourier Transform (FFT). This method allows us to analyze the frequency content of the audio signal and estimate the pitch of the sound. The basic principle behind this approach involves transforming the audio signal from the time domain to the frequency domain using FFT, which helps identify the dominant frequencies in the signal, typically corresponding to musical notes.

In my initial implementation, I wrote a function that processes an audio file using FFT to detect the peak frequency in the signal. The approach worked as follows:

1) Reading the Audio File: I used Python's wave module to read the audio file and extract the raw audio data. The file's sample rate, sample width, and number of channels were determined to correctly interpret the audio signal.
2) Fourier Transform: The Fast Fourier Transform was applied to the signal, converting the time-domain audio signal into a frequency-domain representation. The magnitude of each frequency component was then computed, which showed how much energy was present at each frequency.
3) Peak Frequency Detection: The frequency with the highest magnitude (peak) was selected as the fundamental frequency of the sound. This frequency is then compared with a dictionary of musical note frequencies to find the closest match.
4) Converting musical note to MIDI number: Using Python librosa's implemented function note_to_midi for converting closest matched note to MIDI number (detailed explanation on how this function works is below).

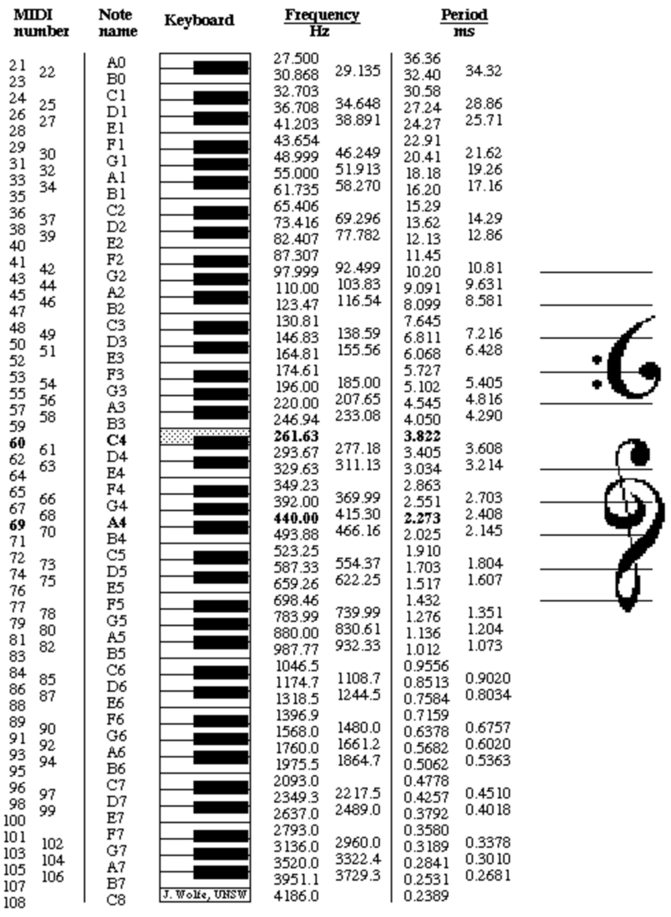
To convert the detected musical note frequencies to their corresponding MIDI numbers, we use the relationship between note frequencies and MIDI numbers (see Figure 1). This relationship is based on the concept of equal temperament tuning, where an octave is divided into 12 semitones. Each semitone has a fixed frequency ratio of approximately 1.059 (which is $2^{\frac{1}{12}}$). The reference note A4 is commonly set at 440 Hz, which corresponds to MIDI number 69. The general formula to convert from frequency f_m to MIDI number m is:

$$m = 12 \times \log_2\left(\frac{f_m}{440}\right) + 69$$

This formula calculates the number of semitones above or below A4, and then adjusts the MIDI number accordingly.

Conversely, to convert a MIDI number m back to a frequency f_m, the formula is:

$$f_m = 2^{\frac{(m-69)}{12}} \times 440$$


J. Wolfe, UNSW

MIDI number	Note name	Frequency Hz	Period ms
21	A0	27.500	36.36
22		29.135	34.32
23	B0	30.868	32.40
24	C1	32.703	30.58
25		34.648	28.86
26	D1	36.708	27.24
27		38.891	25.71
28	E1	41.203	24.27
29	F1	43.654	22.91
30		46.249	21.62
31	G1	48.999	20.41
32		51.913	19.26
33	A1	55.000	18.18
34		58.270	17.16
35	B1	61.735	16.20
36	C2	65.406	15.29
37		69.296	14.29
38	D2	73.416	13.62
39		77.782	12.86
40	E2	82.407	12.13
41	F2	87.307	11.45
42		92.499	10.81
43	G2	97.999	10.20
44		103.83	9.631
45	A2	110.00	9.091
46		116.54	8.581
47	B2	123.47	8.099
48	C3	130.81	7.645
49		138.59	7.216
50	D3	146.83	6.811
51		155.56	6.428
52	E3	164.81	6.068
53	F3	174.61	5.727
54		185.00	5.405
55	G3	196.00	5.102
56		207.65	4.816
57	A3	220.00	4.545
58		233.08	4.290
59	B3	246.94	4.050
60	**C4**	**261.63**	**3.822**
61		277.18	3.608
62	D4	293.67	3.405
63		311.13	3.214
64	E4	329.63	3.034
65	F4	349.23	2.863
66		369.99	2.703
67	G4	392.00	2.551
68		415.30	2.408
69	**A4**	**440.00**	**2.273**
70		466.16	2.145
71	B4	493.88	2.025
72	C5	523.25	1.910
73		554.37	1.804
74	D5	587.33	1.703
75		622.25	1.607
76	E5	659.26	1.517
77	F5	698.46	1.432
78		739.99	1.351
79	G5	783.99	1.276
80		830.61	1.204
81	A5	880.00	1.136
82		932.33	1.073
83	B5	987.77	1.012
84	C6	1046.5	0.9556
85		1108.7	0.9020
86	D6	1174.7	0.8513
87		1244.5	0.8034
88	E6	1318.5	0.7584
89	F6	1396.9	0.7159
90		1480.0	0.6757
91	G6	1568.0	0.6378
92		1661.2	0.6020
93	A6	1760.0	0.5682
94		1864.7	0.5363
95	B6	1975.5	0.5062
96	C7	2093.0	0.4778
97		2217.5	0.4510
98	D7	2349.3	0.4257
99		2489.0	0.4018
100	E7	2637.0	0.3792
101	F7	2793.0	0.3580
102		2960.0	0.3378
103	G7	3136.0	0.3189
104		3322.4	0.3010
105	A7	3520.0	0.2841
106		3729.3	0.2681
107	B7	3951.1	0.2531
108	C8	4186.0	0.2389

Fig. 1. Frequency, Note names and MIDI Number Conversion

B. Second approach

With this approach we observed YIN algorithm. The YIN algorithm is a well-known pitch detection method that is designed to estimate the fundamental frequency (pitch) of a sound signal with high precision. Unlike FFT, the YIN algorithm specifically targets the pitch of a monophonic signal. It is also better suited for noisy signals, because it is more resilient to slight variations and distortions in the signal compared to FFT.

1) YIN algorithm: The YIN algorithm works in a series of steps to accurately estimate the pitch of a signal:

1) Autocorrelation: The process starts by calculating the autocorrelation function, which measures how similar the signal is to itself at different time lags. This is done by calculating the following function for each lag τ:

$$r(\tau) = \sum_{t=0}^{T-\tau} x(t) \cdot x(t+\tau)$$

where $r(\tau)$ is the autocorrelation at lag τ, and $x(t)$ is the audio signal at time t. The signal is divided into short windows, and the autocorrelation is computed for each window.

2) Difference Function: The difference function $d(\tau)$ is then defined as:

$$d(\tau) = \sum_{t=0}^{T-\tau} (x(t) - x(t+\tau))^2$$

This function highlights the differences between the signal and its shifted versions, helping to identify potential pitch periods. The algorithm searches for the values of $d(\tau)$ where the function reaches its minimum, corresponding to multiples of the signal's period.

3) Cumulative Mean Normalized Difference (CMND) Function: The difference function is normalized by the cumulative mean to smooth the curve and reduce noise. The CMND function $d_2(\tau)$ is defined as:

$$d_2(\tau) = \frac{d(\tau)}{\frac{1}{\tau}\sum_{j=1}^{\tau} d(j)}$$

Here, $d(\tau)$ is the difference function, and the denominator normalizes the difference by averaging previous values, making it easier to identify the period.

4) Absolute Threshold: The algorithm applies an absolute threshold to the CMND function to find the smallest value of τ where the function dips below this threshold. This threshold determines the period of the signal, with τ being the estimated period corresponding to the fundamental frequency F_0.

5) Parabolic Interpolation: To improve the precision of the period estimate, the algorithm fits a parabola to the local minimum of the CMND function and its neighboring points. The abscissa of the fitted parabola provides a more accurate estimate of the period.

6) Best Local Estimate: Finally, the algorithm refines the pitch estimate by repeating the process around the detected local minimum to ensure the most accurate estimate of the pitch period.

These steps combine to give an accurate estimate of the fundamental frequency F_0, or pitch, of the signal. The first five steps are sufficient for pitch detection, which is the core of our transcription system.

In my program I used librosa (a popular Python library for audio analysis) to apply the YIN algorithm to extract the fundamental frequencies from the input audio. The following key steps were involved in this implementation:

1) Loading the Audio: I used librosa.load to load the audio file into a waveform.

2) Pitch Estimation Using YIN: I applied the YIN algorithm using librosa.pyin to estimate the fundamental frequencies (f_0). This function calculates the pitch of the audio signal over time. The pitch values were constrained to lie between the frequencies corresponding to the lowest piano note (A0, 27.5 Hz) and the highest

piano note (C8, 4186 Hz) to ensure that the frequencies corresponded to real musical notes.

3) Mapping Frequencies to MIDI: Once the fundamental frequencies were extracted, I converted them into MIDI note numbers using librosa.hz_to_midi. This function maps the detected frequencies to the corresponding MIDI note number, which ranges from 0 (C-1) to 127 (G9).
4) Creating a Piano Roll: A piano roll was constructed to represent the temporal evolution of the detected notes. In this representation, each row corresponds to a time frame, and each column represents a MIDI note. A value of 1 indicates that the note is active during that time frame.
5) Converting to MIDI: Finally, I used pretty_midi, a Python library for working with MIDI files, to convert the piano roll into actual MIDI notes. Each MIDI note was created with a specified start time, end time, and velocity. I also set a minimum note duration to avoid the creation of very short notes.

IV. Results

If you plan to test the program using Audacity or GarageBand, you may encounter misleading results where the MIDI notes appear incorrectly displayed in these programs. However, this is not an error in the program itself. To ensure the notes are displayed correctly:

- Add 12 to every note for Audacity.
- Add 24 to every note for GarageBand.

The initial approach, as mentioned earlier, works effectively only when the audio file contains a single, isolated note. In our testing, the program achieved 100% accuracy with the notes A3, F4, A5, G#5, and E4. However, challenges arose when attempting to extend the note detection function to an entire song. This difficulty stems from the inability to sufficiently eliminate background noise or overlapping sounds. While the program performs exceptionally well for clear, isolated tones, it struggles with more complex audio containing multiple elements.

For the second approach, we created a table where the first column represents the note from the original song, and the second column indicates whether the note was correctly detected by our program. The Happy Birthday song achieved an accuracy of 79%. Initially, the Twinkle Twinkle Little Star song scored 61%, primarily due to the program's failure to detect repeated notes. To address this, we played the song twice as long, which improved the accuracy to 85%.

In our program, we set a minimum note duration of 0.2 seconds to filter out background noise. While adjusting this duration did not improve the results.

Original Song Note Name	Expected Match
C5	Not detected
C5	Detected
D5	Detected
C5	Detected
F5	Detected
E5	Detected
C5	Not detected
C5	Detected
D5	Detected
G5	Detected
F5	Detected
C5	Not detected
C5	Detected
C6	Not detected
A5	Detected
F5	Detected
E5	Detected
D5	Detected
A#5	Detected
A#5	Not detected
A5	Deteceted
F5	Deteceted
G5	Deteceted
F5	Deteceted

TABLE I
COMPARISON OF NOTES IN HAPPY BIRTHDAY SONG

Original Song Note Name	Expected Match (shorter duration notes	Expected Match (longer duration of notes
C4	Not detected	Detected
C4	Detected	Detected
G4	Detected	Detected
G4	Not detected	Detected
A4	Detected	Not detected
A4	Not detected	Detected
G4	Detected	Detected
F4	Detected	Not detected
F4	Not detected	Detected
E4	Detected	Detected
E4	Not detected	Detected
D4	Detected	Detected
D4	Detected	Detected
C4	Detected	Detected
G4	Detected	Not detected
G4	Not detected	Detected
F4	Detected	Detected
F4	Not detected	Detected
E4	Detected	Detected
E4	Not detected	Detected
D4	Detected	Detected

TABLE II
COMPARISON OF NOTES IN HALF OF THE TWINKLE TWINKLE LITTLE STAR SONG

V. Conclusion

The second approach, utilizing the YIN algorithm, proves to be more effective than the first approach based on Fast Fourier Transform (FFT) for audio-to-MIDI conversion. The YIN algorithm offers higher precision in pitch detection and is more resilient to noise and distortions, making it better suited for real-world audio signals. This approach provides more accurate results for clear monophonic signals, achieving an 85% accuracy rate when applied to a song like ”Twinkle Twinkle Little Star.”

However, while this method works well for simple audio with distinct, non-overlapping notes, there is room for improvement when dealing with more complex signals, such as overlapping tones or multi-instrument tracks. In order to enhance the program's ability to handle such situations, further refinements are necessary. These could include better noise filtering, multi-pitch detection, and source separation techniques. With these improvements, the program could more accurately convert polyphonic music and audio recordings involving multiple instruments into MIDI representations.

Thus, while the second approach provides a more robust foundation for audio-to-MIDI conversion, continued development will be required to handle the challenges posed by overlapping notes and multiple instruments.

References

[1] N. J. Sieger and A. H. Tewfik, *Audio coding for conversion to MIDI*, Minneapolis, USA: University of Minessota, 2015.

[2] Gao Qiaozhan, *Pitch detection based monofonic piano transcription*, New York, USA: University of Rochester, 1997.

[3] Tomohiro Ichita, Seisuke Kyochi, Keisuke Imoto *Audio Source Separation Based on Nonnegative Matrix Factorization with Graph Harmonic Structure*, Fukuoka, Japan: The University of Kitakyushu, 2018.

[4] Wolfe Joe *Note names, MIDI numbers and frequencies*, Sydney, Australia: The University New South Wales, 2005.