Topic

Investigating the effectiveness of the Fourier analysis in its applications with audio processing.

Research Question

To what extent is the performance of Fourier analysis based automatic music transcription algorithms influenced by time window and activation level?

Vu Cao
Words: 3718

Table of Contents

1. Introduction

The capability of transcribing music audio into music notation is a fascinating example of human intelligence. It involves perception (analysing complex auditory scenes), cognition (recognizing musical objects), knowledge representation (forming musical structures), and inference (testing alternative hypotheses).

This paper seeks to investigate the extent to which Fourier analysis can be used to create algorithms to automatically transcribe music from an audio source. Specifically, how Fourier transform (the decomposition process of a function into its components) can be used to break down a continuous audio source into its resulting frequencies.

This research could prove useful for those wishing to study or enjoy music. For example, the ability to automatically transcribe sheet music from one's favourite song to play along with, dictating improvised musical ideas to a program that automatically transcribes it, and music search through indexing and identifying music from melody, rhythm, or chord progression. It is from this challenge that many have attempted to create software and algorithms to transcribe music automatically, yet all had their limitations, whether that be in accuracy or efficiency (Benetos et al.).

This paper aims to provide an algorithmic approach utilising Fourier transformation to transcribe music from an audio source, and analyze the accuracy and effectiveness of this algorithm. To investigate this technique's effectiveness, the algorithm was tested against different audio files with varying complexity. Different parameters for the algorithm were utilized for the goal of finding an optimal configuration. The resulting data will be analyzed to assess the effectiveness, as well as the limitations of my implementation of the Fourier transform to transcribe music.

2. Background

2.1. Fourier Analysis

Fourier analysis is a field of mathematics that studies the way functions can be represented or approximated as a sum of basic trigonometric functions with definite frequency. (Morin). This field has many scientific applications, including physics, protein structure analysis and signal processing, as expressing a general function as a sum of sinusoidal functions is simpler to work with.

2.2. Discrete Fourier Transform

The Discrete Fourier Transform (DFT) is an important concept in mathematics and is used to perform Fourier analysis in many practical applications. It converts a finite sequence of equispaced samples into another sequence with the same spacing, which represents the

frequencies and amplitudes that make up the signal. The DFT is therefore a frequency domain representation of the original input sequence. For the field of audio processing, performing the Fourier Transform on a sound signal allows it to be decomposed into the sound frequencies and their respective amplitudes.

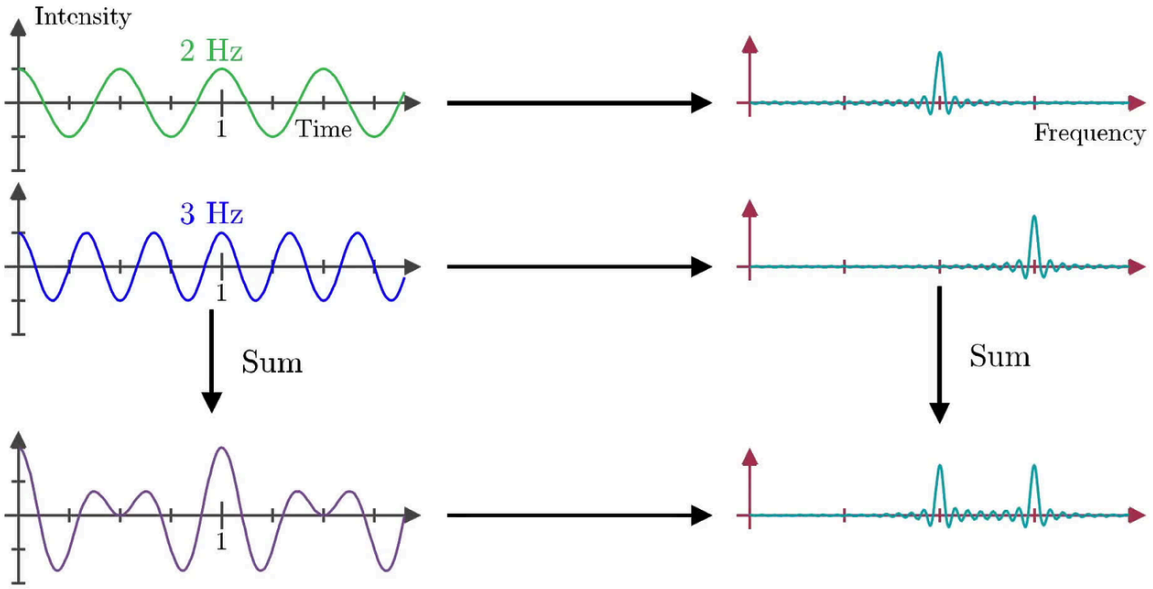

Figure 1: A graphical representation of the DFT being applied to a signal to produce its frequency domain representation. Adapted from (3Blue1Brown)

2.3. Fast Fourier Transform

The DFT deals with a finite amount of data, and therefore it can be implemented through computer algorithms or even dedicated hardware. The Fast Fourier Transform (FFT) refers to any algorithm that computes the DFT of a sequence in an optimized way, as calculating the DFT by its strict definition can be computationally expensive. FFT is almost always used over DFT as it produces the same result with much better runtime, making the terms “DFT” and “FFT” almost interchangeable.

In general, an FFT will break down a DFT matrix into a product of smaller, mostly zero factors, which then the transformation is performed on, rather than computing from a single matrix. As a result it reduces the time complexity from $O(N^2)$ of the definition of the DFT to O(N log N), where N is the datasize. The basic ideas for various FFT methods were popularized in 1965, but some algorithms had been derived as early as 1805 (Heideman et al.).

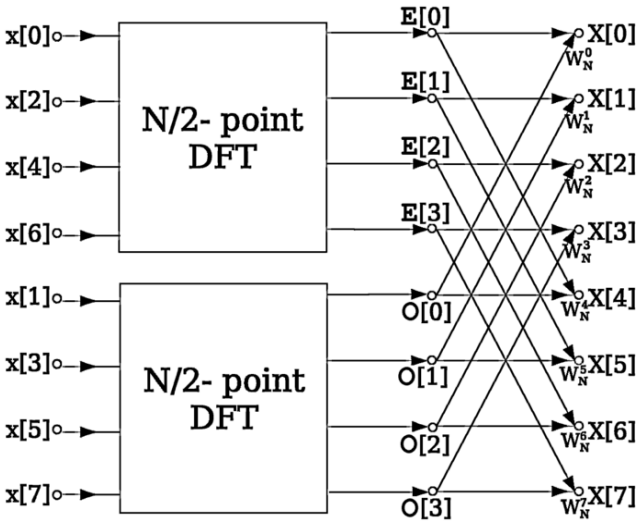

Figure 2: An example FFT algorithm structure, with each step decomposing the DFT into half-size DFTs. This step can be repeated for higher efficiency. (OhArthits)

2.4 Automatic Music Transcription

Automatic music transcription (AMT), is the process of producing sheet music from audio signals through computational algorithms (Benetos et al.). It comprises several subtasks, including monophonic/polyphonic pitch estimation, onset and offset detection, beat and rhythm tracking, and interpretation of expressive timing and dynamics. Because audio signals are often a complex combination of different sound sources (voice, percussion, musical instruments) with varying timbres, AMT is still considered a challenging and open problem, and has been explored since the 1970s. (Gallard and Piszczalski).

3. Methodology

Primary experimental data is the main source of data in this paper. An algorithm utilising the FFT was programmed (code in appendix, adapted from (Jones)), and was tested on a custom dataset, consisting of computer generated audio, and recordings of live samples. Different parameters, including time window and activation level were experimented with, and the resultant processing time and accuracy was recorded. An experimental methodology was chosen for this paper, as this methodology provides ample freedom to manipulate the independent variables.

However, this methodology is subject to limitations. One such limitation is due to the fundamental complexity of AMT, which encompasses many underlying subtasks; and as such the scope of the algorithm discussed in this paper only covers pitch and duration. Time constraints also pose another limitation, as this paper only covers one algorithm for AMT using the FFT, whereas there are many approaches for this problem. One last limitation is the size of the dataset used. Most publicly available datasets are too large for the scope of this paper, and are more fit for machine learning approaches. This paper uses an algorithmic

approach, which requires manual supervision to obtain data. Therefore, a smaller dataset is more optimal, although it may not encompass the full range of possible audio signals.

3.1 Dataset

The dataset in this paper consists of .wav files of computer generated audio and recordings of live samples, as well as their corresponding .midi files that describe them. The computer generated audio was created using MuseScore, an open source music notation software and exporting both .midi and .wav files using the Piano instrument[1]. The live samples are from various performances I as well as my peers have recorded, and I used .midi files generated from their source sheet music. In total, there are 3 of each type, for a total of 16 audio files in the dataset.

List of files in dataset

- Twinkle.wav - Computer generated audio of "Twinkle Twinkle Little Star" (120 bpm) (0:25)
- Mary.wav - Computer generated audio of "Mary had a Little Lamb" (120 bpm) (0:32)
- Canon.wav - Computer generated audio of "Canon in D" by Johann Pachelbel (100 bpm) (4:06)
- ByYourSide.wav - My performance of "By Your Side" by Pedro Silva on piano (88 bpm) (2:57)
- Nocturne.wav - My performance of "Nocturne in E Flat Major" by Frédéric Chopin on piano (60 bpm) (3:40)
- OdeToJoy.wav - My brother's performance of "Ode to Joy" by Ludwig van Beethoven on violin (120 bpm) (0:40)

3.2 The Dependent Variables

The variables being measured in this paper are processing time and accuracy.

Time

The time measured is the time it took the algorithm to process an audio file and output the corresponding .midi transcription. The `time.time()` method from Python's `time` module was used to measure this. Since the algorithm is deterministic, each configuration is ran 3 times, and the time measured is the average.

Accuracy

The accuracy measured is a measure of how close the output .midi file is compared to the source .midi files. To calculate this, pitch and duration were taken into consideration. For each individual note, a correct pitch accounts for 75% accuracy, while duration accounts for 25%. Any excessive note is counted as incorrect.

[1] https://musescore.org/en

3.3 The algorithm programmed

Python was used to create the algorithm and analyze its effectiveness. The first step is to read the audio files, and the `soundfile`[2] module is used to do this. Then, the file is broken into blocks at the specified time window, using `soundfile.blocks()`. The `NumPy`[3] module is used to perform the FFT for each block using `numpy.fft()`. For each frequency peak that passes the user specified activation level, it was mapped to the closest corresponding note on the 88-note Western music system.

Each time block is now associated with some frequencies. For every consecutive block that contains one or more of the same frequencies, it is condensed together into one single note, to establish a sense of rhythm. Finally, the `python3-midi`[4] module was used to write the output to a .midi file, a computer readable format for storing abstract audio information, often used to represent sheet music.

3.4 The Experimental Procedure

For each audio file, a heuristic approach was used. The time window and activation level was adjusted with their accuracy recorded, until an optimal result is obtained. Any patterns appearing in the processing time and accuracy were analyzed.

Time window

Assuming that the audio file is at a constant tempo, and the tempo is known (in terms of beats per minute), the time window can be adjusted accordingly to a high effectiveness. A tempo marking of 120 bpm, means that each second there will be 2 beats. Therefore, if one wishes to perform the FFT for every beat, the time window should be 500ms.

However, music often contains more complex rhythms, which may encompass eighth and sixteenth notes, which are subdivisions of the beat. Therefore, the time window can be adjusted to be 250ms, 125ms, 62.5ms etc…, with the trade-off of higher processing time.

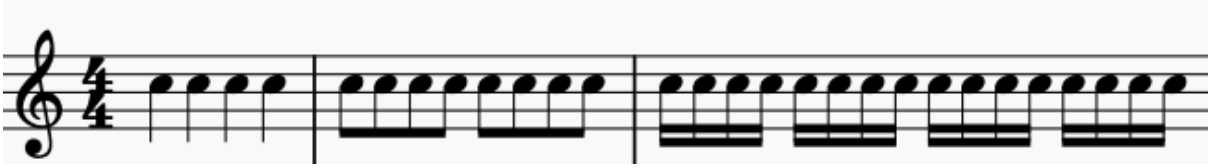

Figure 4: Visual representation of quarter notes, eighth notes, sixteenth notes. There are 4 notes in each measure, with quarter notes taking up one beat each.

[2] https://pypi.org/project/soundfile/
[3] https://pypi.org/project/numpy/
[4] https://pypi.org/project/python3-midi/

Activation level

Activation level is a more sensitive parameter. An activation level of 0 means every peak detected will be transcribed, whereas a value of 1 means only frequencies which reach an arbitrary max amplitude will be transcribed. Each audio file is likely to have a different optimal activation level.

4. Analysis of Data

4.1 Example conversion

The first of the computer generated audio in the dataset is the simple melody of "Twinkle Twinkle Little Star". This serves as a sort of benchmark, as it has a simple rhythm with a monophonic texture.

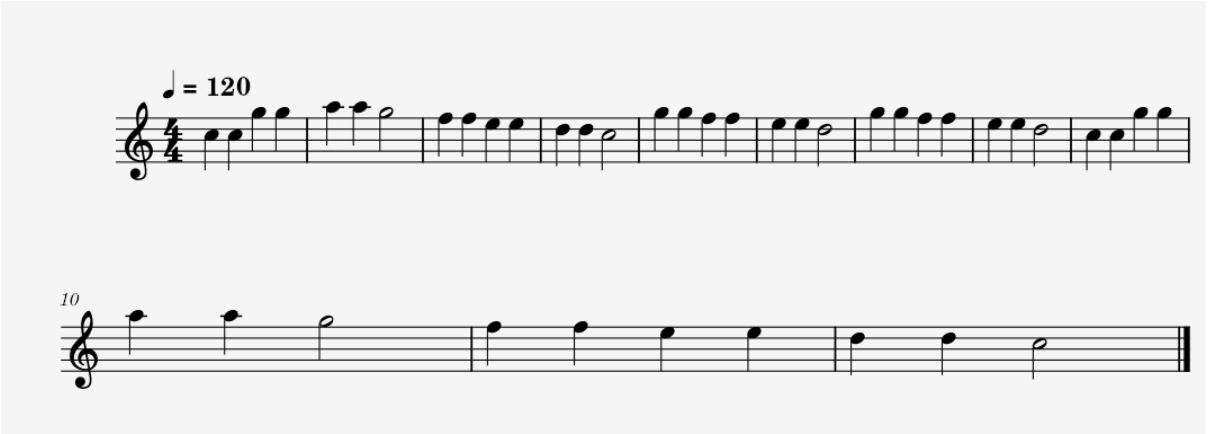

Figure 5: Music sheet representation of "Twinkle Twinkle Little Star"

The following is the resulting .midi file, using a time window of 500ms, and activation level of 0.

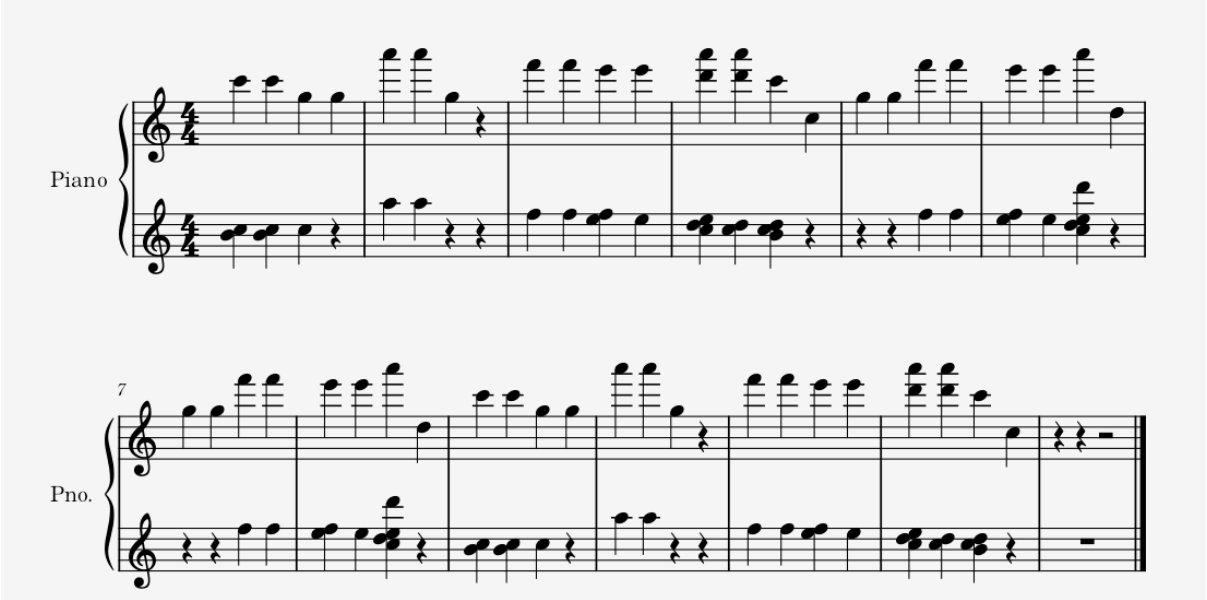

Figure 6: Music sheet representation of transcription with time window 500ms, activation level 0. Processing time: 3.88s

Right away some problems are apparent. One, there appears to be 4 voices here, when the original audio only has 1 voice. This can be explained by a phenomenon called "overtones". An overtone is any resonant frequency above the fundamental frequency of a sound. The sound that is most commonly perceived is called the fundamental tone, which is shown in Figure 5. While the fundamental is usually heard most prominently, overtones are actually present in any pitch except a true sine wave.

It is the combination of overtones and their amplitudes that produce different timbres for different sounds, like instruments or human voice. Luckily, overtones are often much quieter than their fundamentals, so by adjusting the activation level, the overtones can be filtered out.

Below is the resulting .midi file, using a time window of 500ms, and activation level of 0.1.

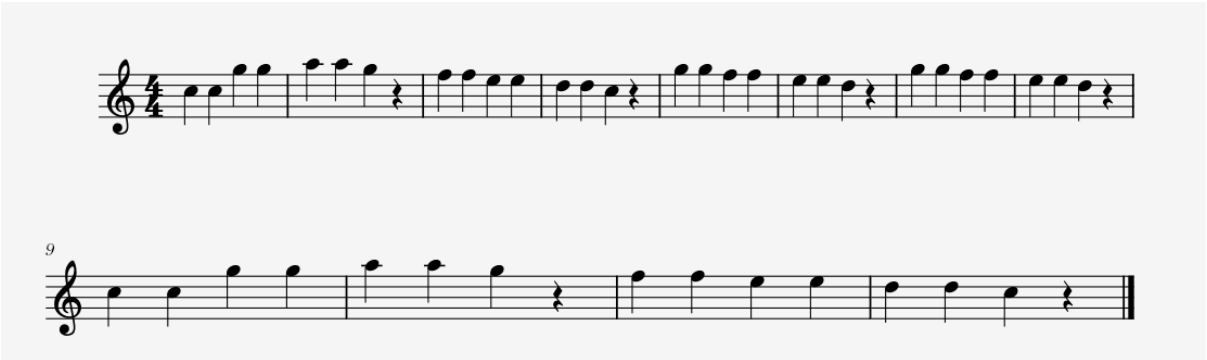

Figure 7: Music sheet representation of transcription with time window 500ms, activation level 0.1. Processing time: 3.88s

This produces a much more optimal result, with an accuracy of 93.75%. It is missing the half notes every second measure, and instead chooses to describe it as a quarter note along with a quarter rest. This is caused by the activation level being too high, as a note fades out after it is played in MuseScore, to mimic the sound of a real piano.

If the activation level is adjusted to 0.05, the following is obtained.

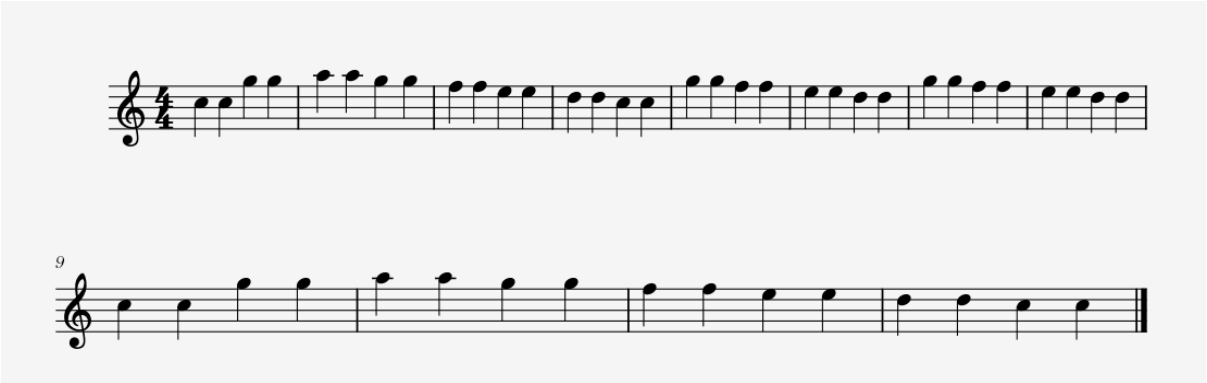

Figure 8: Music sheet representation of transcription with time window 500ms, activation level 0.05. Processing time: 3.96s

Now, instead of a half note, it is two quarter notes. Up until now, the option to condense notes together had been turned off. This was done as by condensing it, consecutive quarter notes with the same pitch will get counted as one half note, even when they are distinct. If this condensation method is applied, the following is obtained.

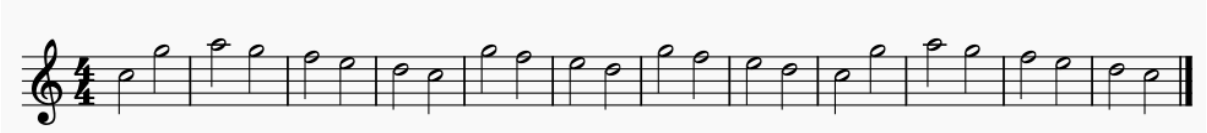

Figure 9: Music sheet representation of transcription with time window 500ms, activation level 0.05, condensed notes. Processing time: 3.94s.

One way to fix this is by performing a “mini-FFT” between two consecutive time windows. If two notes are distinct, the amplitude of the second window should be about the same as the first one. If it is instead a continuous note, the amplitude of the second window will be much smaller, and the two can be combined into the same note safely. A smaller time window may allow this to be more accurate as well.

Applying this change, a perfect result is obtained.

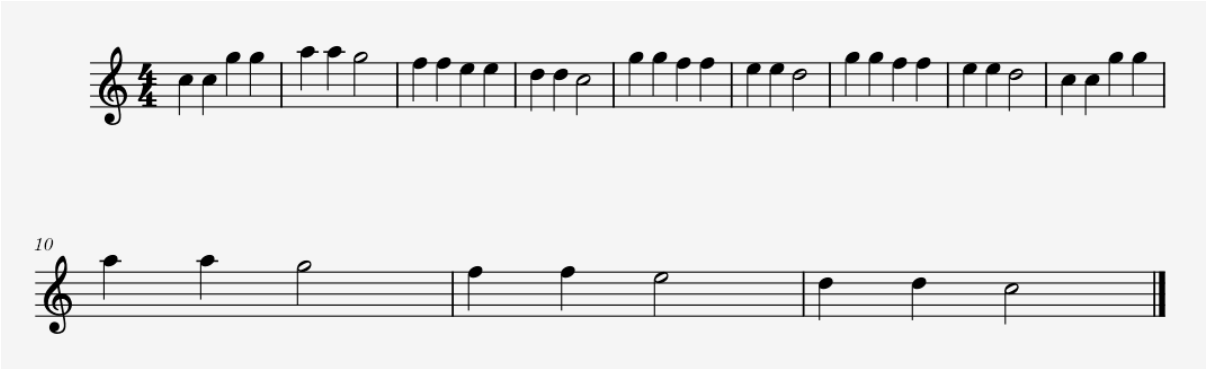

Figure 10: Music sheet representation of transcription with time window 250ms, activation level 0.05, smart condensed notes. Processing time: 7.33s

4.2 Tabular Data Presentation

Twinkle Twinkle Little Star

Time window (ms)	Activation level	Processing time (s)	Accuracy
500	0	3.88	17.93%
500	0.1	3.58	93.75%
500	0.05	3.96	75.00%
250	0.05	4.73	100%[5]
250	0.5	4.35	3.58%

Mary had a Little Lamb

Time window (ms)	Activation level	Processing time (s)	Accuracy
500	0	4.96	21.55%
500	0.1	4.95	81.74%
250	0.1	5.89	83.71%
250	0.05	5.93	97.83%
250	0.5	5.63	2.21%

[5] The first three rows of data do not use smart condensation, for demonstration purposes. Every other data obtained onwards utilizes smart condensation.

Canon in D

Time window (ms)	Activation level	Processing time (s)	Accuracy
600	0	45.53	15.38%
600	0.1	45.49	61.74%
300	0.1	47.03	65.69%
300	0.05	47.93	73.16%
300	0.3	47.96	15.64%

By Your Side

Time window (ms)	Activation level	Processing time (s)	Accuracy
340.91	0	24.83	5.38%
340.91	0.05	25.05	30.12%
170.45	0.05	26.10	26.01%
170.45	0.02	26.39	33.70%
170.45	0.2	26.15	16.50%

Nocturne in E flat major

Time window (ms)	Activation level	Processing time (s)	Accuracy
125	0	34.45	5.48%
125	0.02	37.27	10.14%
62.25	0.01	39.56	46.21%
62.25	0.01	39.39	41.30%
62.25	0.4	38.75	6.79%

Ode to Joy

Time window (ms)	Activation level	Processing time (s)	Accuracy
250	0	6.38	17.53%
250	0.1	6.12	70.79%
125	0.1	6.85	79.16%
125	0.05	6.76	75.97%
125	0.5	6.79	32.83%

4.3 Data Analysis

Possible correlations

There appears to be a correlation between time window and processing time. Both units are a measure of time. However, each file has a different length. Therefore, processing time per second will be used instead to provide a more accurate comparison. Below is a scatter plot comparing the time window and processing time.

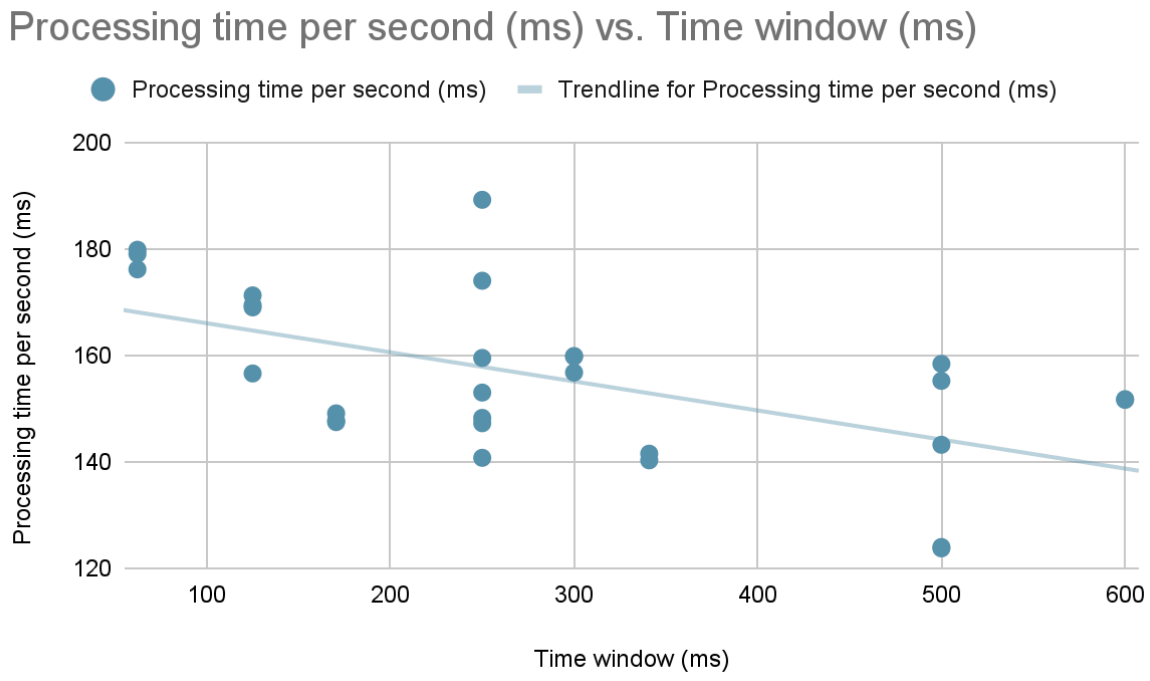

Figure 11: A scatter plot comparing time window and processing time per second

From this graph, it can be seen that the time window is inversely proportional to processing time. As the time window increases, processing time decreases. Since higher time windows mean less blocks are produced, therefore less FFTs are performed. However, one

would expect that the relationship would be more drastic. For example, if the timing window was doubled, the processing time should be halved, and vice versa.

One explanation for this, is that while less FFTs are performed, the computation complexity of each FFTs is increased. The standard sample rate of a .wav file is 44.1 kHz, or 44100 data points per second. For a timing window of 500ms, that is the equivalent of performing the FFT every 22050 data points; and for a timing window of 250ms, that is performing the FFT every 11025 data points. Another cause is the condensation technique applied, as it performs a check of a small time window between two consecutive time windows, which drastically increases processing time for lower timing windows.

The existence of an inverse relationship between the time window and processing time implies that it is more ideal to choose a larger time window to optimize processing timer, as performing a small amount of large-sized FFTs is more efficient than a large amount of small-sized FFTs. However, details such as rhythm are lost when using a large time window. Therefore, a balance must be found.

With a sufficiently small enough time window, no details will be lost. Because of the condensation technique, the correlation between the time window and accuracy is negligible as the timing window approaches 0. This is due to there being simply no more things to detect past the smallest subdivision of a beat present in an audio file. If a song consists fully of quarter notes, there is no need to change the timing window to fit an eighth or sixteenth notes pattern.

Activation level has no effect on time, as it only affects what gets recorded from the FFT calculations. Below is a scatter plot comparing the activation level and accuracy.

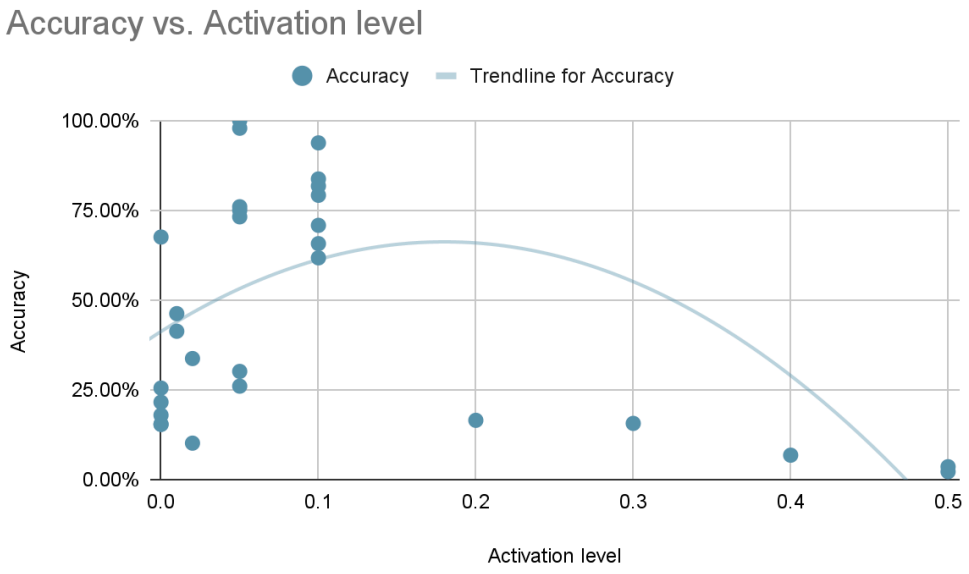

Figure 11: A scatter plot comparing activation level and accuracy

At an activation level of 0, accuracy is very low, as unnecessary artifacts from noise and overtones are added. However, high activation levels (greater than 0.2) also produce low accuracy, as details can be missed. This varies between different audio files, as they have different volumes, as well as varying instruments and timbres. The optimal activation level for most audio files after manual testing were around 0.02-0.1, however the graph suggests that the best activation level is around 0.2. The reason for this may be due to the lack of variety in the dataset; as 5 of those consisted solely of piano; 3 of which were computer generated.

5. Further Research Opportunities

5.1 Larger datasets

This paper only utilized a small dataset consisting of 6 audio files. This is fine for a proof of concept analysis, such as this paper, but does not provide diversity and variety in audio information. As such, the analysis may be limited to only the files in the dataset, as the sample size is not sufficient to generalize to any audio files.

Some other limitations of the dataset are a lack of polyphonic textures, lack of variety in instruments, and lack of tempo changes.

5.2 Automated testing

A severe limitation of this paper was the lack of automation. This means that using a large dataset is virtually impossible. By transcribing each audio file in the dataset automatically through a set of specified parameters, more results could be obtained for analysis, as well as reducing the amount of trial and error with manual human input.

5.3 Machine learning

Following from 5.1, a predictive machine learning model can determine the optimal configuration of time window and activation level to obtain the optimal results given the data from automated tests. Given that machine learning models are virtually capable of anything, perhaps there will exist such a program that can transcribe music better than any humans.

Another alternative, could be to use machine learning to transcribe music altogether, replacing the need for algorithms utilizing FFTs. There exists publicly available datasets with MIDI annotated audio files, which is suitable for training. Those datasets would have a wider variety of instrumentation, rhythm and style as well, which can provide a lot more information for the model to learn about what truly is music, instead of simply analyzing sound waves.

6. Conclusion

In this paper, the relationship between the performance of a Fourier analysis based AMT technique and time window and activation level. Logical and mathematical explanations for the patterns observed were also provided.

The results show that by increasing the time window, the processing time is decreased. This pattern is observed across all points of data, and has a very strong correlation value. However, increasing the time window comes at the cost of accuracy. As the time window increases, certain details, such as rhythm, phrasing and dynamics will be missed. By decreasing the time window, accuracy is improved to a certain point; this being the timing of the shortest sub division, after which decreasing the time window provides diminishing returns.

It also showed that the optimal activation level is around 0.2. This is not a very strong correlation however, as some files had an optimal activation value between 0.05-0.1. This mostly comes down to the quality of the file; volume, background noise, and timbres are all qualities that can affect the optimal activation level. In general, values that differ too much from this will produce a transcription that is either too noisy, or not enough information at all. However, this may prove to be different for different audio files, especially in larger datasets.

Hopefully this paper will prove useful as a proof of concept for solving a complex problem in computer science and mathematics, which has great significance and potential for recreational and educational usage.

7. Works Cited

3Blue1Brown. “But What Is the Fourier Transform? A Visual Introduction.” *YouTube*, 27 Jan. 2018, www.youtube.com/watch?v=spUNpyF58BY&vl=en.

Benetos, Emmanouil, et al. “Automatic Music Transcription: An Overview.” *IEEE Signal Processing Magazine*, vol. 36, Jan. 2019, pp. 20–30, https://doi.org/10.1109/MSP.2018.2869928.

Heideman, Michael T., et al. *Gauss and the History of The*. Oct. 1984, www.cis.rit.edu/class/simg716/Gauss_History_FFT.pdf.

Jones, Neil F. “Audio-To-Midi.” *GitHub*, 11 Dec. 2022, github.com/NFJones/audio-to-midi.

Morin, David. *Fourier Analysis*. 28 Nov. 2009,

scholar.harvard.edu/files/david-morin/files/waves_fourier.pdf.

OhArthits. “English: Decimation in Time of a Length-N DFT into Two Length-N/2 DFTs

Followed by a Combining Stage.” *Wikimedia Commons*, 4 Jan. 2010,

commons.wikimedia.org/wiki/File:DIT-FFT-butterfly.png.

Piszczalski, Martin, and Bernard A. Galler. “Automatic Music Transcription.” *Computer

Music Journal, vol. 1, no. 4, 1977, pp. 24–31, www.jstor.org/stable/40731297.

8. Appendix

The appendix contains Python code used to process and analyze the data shown in this paper. The program ran using Python 3.9.10, on a Windows 10 laptop with an Intel i7-6700HQ Processor.

8.1 Code for music transcription

```
import logging

from collections import namedtuple
from functools import lru_cache
from operator import attrgetter

import numpy
import soundfile
import midi_writer, notes

class Note:
    __slots__ = ["pitch", "velocity", "count"]

    def __init__(self, pitch, velocity, count=0):
        self.pitch = pitch
        self.velocity = velocity
        self.count = count


class Converter:
    def __init__(
        self,
        infile=None,
        outfile=None,
        time_window=None,
        activation_level=None,
        condense=None,
        condense_max=False,
        max_note_length=0,
```

```
    transpose=0,
    pitch_set=None,
    pitch_range=None,
    note_count=None,
    progress=None,
    bpm=60,
):

    if infile:
        self.info = soundfile.info(infile)
    else:
        raise RuntimeError("No input provided.")

    self.infile = infile
    self.outfile = outfile
    self.time_window = time_window
    self.condense = condense
    self.condense_max = condense_max
    self.max_note_length = max_note_length
    self.transpose = transpose
    self.pitch_set = pitch_set
    self.pitch_range = pitch_range or [0, 127]
    self.note_count = note_count
    self.progress = progress
    self.bpm = bpm

    self.activation_level = int(127 * activation_level) or 1
    self.block_size = self._time_window_to_block_size(
        self.time_window, self.info.samplerate
    )

    steps = self.info.frames // self.block_size
    self.total = steps
    self.current = 0

    self._determine_ranges()

def _determine_ranges(self):
    self.notes = notes.generate()
    self.max_freq = min(self.notes[127][-1], self.info.samplerate / 2)
    self.min_freq = max(self.notes[0][-1], 1000 / self.time_window)
    self.bins = self.block_size // 2
    self.frequencies = numpy.fft.fftfreq(self.bins, 1 / self.info.samplerate)[
        : self.bins // 2
    ]

    for i, f in enumerate(self.frequencies):
        if f >= self.min_freq:
            self.min_bin = i
            break
    else:
```

```
            self.min_bin = 0
        for i, f in enumerate(self.frequencies):
            if f >= self.max_freq:
                self.max_bin = i
                break
        else:
            self.max_bin = len(self.frequencies)

    def _increment_progress(self):
        if self.progress:
            self.current += 1
            self.progress.update(self.current, self.total)

    @staticmethod
    def _time_window_to_block_size(time_window, rate):
        # rate/1000(samples/ms) * time_window(ms) = block_size(samples)
        rate_per_ms = rate / 1000
        block_size = rate_per_ms * time_window

        return int(block_size)

    def _freqs_to_midi(self, freqs):
        notes = [None for _ in range(128)]
        for pitch, velocity in freqs:
            if not (self.pitch_range[0] <= pitch <= self.pitch_range[1]):
                continue
            velocity = min(int(127 * (velocity / self.bins)), 127)

            if velocity > self.activation_level:
                if not notes[pitch]:
                    notes[pitch] = Note(pitch, velocity)
                else:
                    notes[pitch].velocity = int(
                        ((notes[pitch].velocity * notes[pitch].count) + velocity)
                        / (notes[pitch].count + 1)
                    )
                    notes[pitch].count += 1

        notes = [note for note in notes if note]

        if self.note_count > 0:
            max_count = min(len(notes), self.note_count)
            notes = sorted(notes, key=attrgetter("velocity"))[::-1][:max_count]

        return notes

    def _snap_to_key(self, pitch):
        if self.pitch_set:
            mod = pitch % 12
            pitch = (12 * (pitch // 12)) + min(
                self.pitch_set, key=lambda x: abs(x - mod)
```

```
        )
    return pitch

@lru_cache(None)
def _freq_to_pitch(self, freq):
    for pitch, freq_range in self.notes.items():
        # Find the freq's equivalence class, adding the amplitudes.
        if freq_range[0] <= freq <= freq_range[2]:
            return self._snap_to_key(pitch) + self.transpose
    raise RuntimeError("Unmappable frequency: {}".format(freq[0]))

def _reduce_freqs(self, freqs):
    reduced_freqs = []
    for freq in freqs:
        reduced_freqs.append((self._freq_to_pitch(freq[0]), freq[1]))

    return reduced_freqs

def _samples_to_freqs(self, samples):
    amplitudes = numpy.fft.fft(samples)
    freqs = []

    for index in range(self.min_bin, self.max_bin):
        # frequency, amplitude
        freqs.append(
            [
                self.frequencies[index],
                numpy.sqrt(
                    numpy.float_power(amplitudes[index].real, 2)
                    + numpy.float_power(amplitudes[index].imag, 2)
                ),
            ]
        )

    return self._reduce_freqs(freqs)

def _block_to_notes(self, block):
    channels = [[] for _ in range(self.info.channels)]
    notes = [None for _ in range(self.info.channels)]

    for sample in block:
        for channel in range(self.info.channels):
            channels[channel].append(sample[channel])

    for channel, samples in enumerate(channels):
        freqs = self._samples_to_freqs(samples)
        notes[channel] = self._freqs_to_midi(freqs)

    return notes

def convert(self):
```

```
        logging.info(str(self.info))
        logging.info("window: {} ms".format(self.time_window))
        logging.info(
            "frequencies: min = {} Hz, max = {} Hz".format(self.min_freq,
self.max_freq)
        )

        with midi_writer.MidiWriter(
            outfile=self.outfile,
            channels=self.info.channels,
            time_window=self.time_window,
            bpm=self.bpm,
            condense=self.condense,
            condense_max=self.condense_max,
            max_note_length=self.max_note_length,
        ) as writer:
            for block in soundfile.blocks(
                self.infile,
                blocksize=self.block_size,
                always_2d=True,
            ):
                if len(block) != self.block_size:
                    filler = numpy.array(
                        [
                            numpy.array([0.0 for _ in range(self.info.channels)])
                            for _ in range(self.block_size - len(block))
                        ]
                    )
                    block = numpy.append(block, filler, axis=0)
                notes = self._block_to_notes(block)
                writer.add_notes(notes)
                self._increment_progress()
```

8.2 Code for writing to a midi file

```
from collections import defaultdict

import python3_midi as midi


class NoteState:
    __slots__ = ["is_active", "event_pos", "count"]

    def __init__(self, is_active=False, event_pos=None, count=0):
        self.is_active = is_active
        self.event_pos = event_pos
        self.count = count


class MidiWriter:
    def __init__(
```

```
        self,
        outfile,
        channels,
        time_window,
        bpm=60,
        condense=False,
        condense_max=False,
        max_note_length=0,
    ):
        self.outfile = outfile
        self.condense = condense
        self.condense_max = condense_max
        self.max_note_length = max_note_length
        self.channels = channels
        self.time_window = time_window
        self.bpm = bpm
        self.note_state = [defaultdict(lambda: NoteState()) for _ in
range(channels)]

        bps = self.bpm / 60
        self.ms_per_beat = int((1.0 / bps) * 1000)
        self.tick_increment = int(time_window)
        self.skip_count = 1
        self._need_increment = False

    def __enter__(self):
        self.stream = midi.FileStream(self.outfile)
        self.stream.start_pattern(
            format=1,
            tick_relative=False,
            resolution=self.ms_per_beat,
            tracks=[],
        )
        self.stream.start_track(
            events=[
                midi.TimeSignatureEvent(
                    tick=0,
                    numerator=1,
                    denominator=4,
                    metronome=int(self.ms_per_beat / self.time_window),
                    thirtyseconds=32,
                )
            ],
            tick_relative=False,
        )
        return self

    def __exit__(self, type, value, traceback):
        self._terminate_notes()
        self.stream.add_event(midi.EndOfTrackEvent(tick=1))
        self.stream.end_track()
```

```
    self.stream.end_pattern()
    self.stream.close()

def _skip(self):
    self.skip_count += 1

def _reset_skip(self):
    self.skip_count = 1

@property
def tick(self):
    ret = 0
    if self._need_increment:
        self._need_increment = False
        ret = self.tick_increment * self.skip_count
        self._reset_skip()
    return ret

def _note_on(self, channel, pitch, velocity):
    pos = self.stream.add_event(
        midi.NoteOnEvent(
        #delete if not work, tried velocity = 100
        #tick=self.tick, channel=channel, pitch=pitch, velocity=velocity
            tick=self.tick, channel=channel, pitch=pitch, velocity=100
        )
    )
    self.note_state[channel][pitch] = NoteState(
        True,
        pos,
        1,
    )

def _note_off(self, channel, pitch):
    self.note_state[channel][pitch] = NoteState()
    self.stream.add_event(
        midi.NoteOffEvent(
            tick=self.tick,
            channel=channel,
            pitch=pitch,
        )
    )

def add_notes(self, notes):
    """
    notes is a list of midi notes to add at the current
        time step.

    Adds each note in the list to the current time step
        with the volume, track and channel specified.
    """
    self._need_increment = True
```

```
        if not self.condense:
            self._terminate_notes()

        for channel, notes in enumerate(notes):
            new_notes = set()
            stale_notes = []
            for note in notes:
                note_state = self.note_state[channel][note.pitch]
                new_notes.add(note.pitch)
                if (not self.condense) or (self.condense and not
note_state.is_active):
                    self._note_on(channel, note.pitch, note.velocity)
                elif self.condense and note_state.is_active:
                    event = self.stream.get_event(
                        midi.NoteOnEvent, note_state.event_pos
                    )
                    old_velocity = event.data[1]
                    if self.condense_max:
                        new_velocity = max(note.velocity, old_velocity)
                    else:
                        count = note_state.count
                        note_state.count += 1
                        new_velocity = ((old_velocity * count) + note.velocity) //
(
                            note_state.count
                        )
                    if old_velocity != event.data[1]:
                        event.data[1] = new_velocity
                        self.stream.set_event(event, note_state.event_pos)

            if self.condense:
                active_notes = [
                    note
                    for note in self.note_state[channel]
                    if self.note_state[channel][note].is_active
                ]
                for note in active_notes:
                    if (
                        note not in new_notes
                        or self.note_state[channel][note].count >
self.max_note_length
                    ):
                        stale_notes.append(note)

                for note in stale_notes:
                    self._note_off(channel, note)

        if self._need_increment:
            self._skip()

    def _terminate_notes(self):
```

```
for channel in range(self.channels):
    for note, note_state in self.note_state[channel].items():
        if note_state.is_active:
            self._note_off(channel, note)
```