Real-time guitar transcription using Fourier transform based methods

A pitch estimation framework and overtone sieve algorithm

Luc de Jonckheere

June 18, 2021

Abstract

In this paper, we explore real-time transcription of an electric guitar signal. We focus on Fourier transform based methods, as this transform can be performed very fast. Using the obtained frequency domain, we select significant peaks using a Gaussian envelope. These peaks contain many overtones, which are filtered using our overtone sieve algorithm.

We will develop a basis for monophonic transcription and a proof-of-concept algorithm for polyphonic transcription. The goal of this paper is to give an overview of the problems when trying to solve this problem and to provide an implementation in which different methods for peak picking and overtone sieving can be tested.

1 Introduction

Automatic Music Transcription (AMT) is a field of study which tries to automatically convert acoustic recordings into some form of (digital) music notation [15]. In this paper, we will focus on translating a guitar signal into a set of played notes in real-time. This task is similar to AMT, because both have to solve the problem of pitch estimation and onset/offset detection [16]. Because we try to solve the problem in real-time, we have some extra limitations. We must process the data fast enough such that no significant latency is introduced. Furthermore, we must process the data at least as fast as it is generated, otherwise latency would increase over time. Lastly, we cannot look-ahead, as can be done when transcribing pre-recorded songs.

Single pitch estimation can be accurately performed using Fourier transform based methods [14]. Multi pitch estimation is much harder as harmonic overtones are difficult to differentiate from the fundamental frequency [20]. As the Fourier transform can be performed consistently and efficiently on common hardware [3], it is well suited for real-time transcription.

Pitch estimation can be divided into a few subproblems. If a paper focuses on one of these subproblems, it also has to solve the other subproblems in order to test the system. Most papers do not publish their source code, which makes building upon other's work difficult. It also limits the research of the interaction between different solutions for the different subproblems. Because of this, we provide a framework which can be built upon, as the implementation of a subproblem can easily be interchanged with another. The code can be found at "`github.com/lucmans/dechord`".

The goal of this paper is to give a complete overview of the methods used to solve this problem. This is done by developing a program which can translate a guitar signal to a set of observed notes in real-time. We combine techniques spread over different papers into one project and provide an algorithm which can estimate a set of played notes using the set of observed notes. The code should be publicly available and easily extensible so others can optimize a subproblem without having to solve the other subproblems. Using our program, we can experiment if Fourier transform based methods are feasible for real-time transcription.

The focus of this paper is on building a prototype which can perform monophonic AMT and presenting a first draft of a polyphonic version.

2 Related work

Spectral peak picking based AMT methods are often deemed infeasible for real-time AMT [25], due to the relatively low resolution in the low frequencies when using the Fourier transform [6]. This problem cannot be alleviated by zero-padding the Fourier transform input, as it does not add any information, it merely increases the resolution by interpolation [11][5] and increases the computing time of the transform [21]. However, using quadratic interpolation, we can very accurately interpolate peaks within Fourier frequency bins [12]. Also, other transforms such as the constant Q transform [7] which do provide more resolution in lower frequencies are getting more popular, which will allow us to differentiate two notes on a very small interval.

A big problem in polyphonic pitch estimation is the occurrence of overtones [20]. As these overtones also have a periodic nature, they can be detected using another Fourier transform [18] on the obtained frequency domain. However, this periodicity also be detected with less overhead using note sets, which are explained in Section 4.3. Notes which are at octave intervals have the same periodicity [23], which makes them difficult to distinguish. This is often referred to as the octave problem.

Another problem is choosing the real-time constraint for processing the input. Some papers claim 140 ms is sufficient [10]. This is however too long, considering it would be about a note difference when playing eight notes in 200 BPM. Furthermore, there always is a latency from the operating system due to how audio drivers work [19].

When focussing on transcribing a guitar, we can use physical limitations of the instrument to our advantage [9]. However, by taking these limitation into consideration, the resulting program will be less versatile. For instance, these methods fail when the strings are tuned to different notes.

3 Preliminaries

In this paper, we use the Fourier transform to obtain a frequency domain representation of a signal. This transform takes a frame full of samples and produces a list of amplitudes corresponding to frequency bins. When performing the Fourier transform on a frame, spectral leakage is introduced [17]. This kind of noise is remedied by applying a window function to a frame.

When playing a note on an instrument, multiple frequencies are produced. The most notable is the fundamental frequency. Every note corresponds to one fundamental frequency. Integer multiples of the fundamental frequency resonate and produce harmonic overtones [24]. They are called harmonic overtones, as they are exact integer multiples and do not produce destructive interference among each other. Every integer multiple resonates with the fundamental frequency, however, each subsequent overtone is of lower amplitude.

Generally, instruments do not produce perfect sine waves. The specific distortion of an instrument is called its timbre [13]. Even though the frequency components of these distortions are insignificant compared to the fundamental frequency and its overtones, it still results in extra noise in the signal. When strumming a note on the guitar, there is a strong attack. During this attack, the signal behaves chaotically and results in many fluctuating peaks in the frequency domain. This phenomenon is called a transient.

To specify in what octave a certain note lies, we use scientific note notation. This adds a subscript octave number to a note name. In scientific notation, we start from the note C. In other words, these notes are denoted in increasing pitch: C_3, A_3 and C_4. Note that A_3 is higher pitches than C_3, which is counter intuitive.

Most modern instruments use the twelve-tone equal temperament tuning system (12-TET). This tuning system divides an octave, which the distance between a frequency and its double, into twelve equal parts on the logarithmic scale. In other words, the ratio between two contiguous notes is $\sqrt[12]{2}$. The distance between two notes can further be divided on a logarithmic scale into 100 cents. The number of cents between two frequencies can be calculated as $1200 * \log_2(\frac{f_1}{f_2})$. 12-TET is tuned relative to a standard pitch. In modern music, we define the A_4 to be 440 Hz.

4 Pitch estimation

Automatic music transcription can be separated into two main problems; onset detection and pitch estimation [16]. In this paper, we will focus on pitch estimation. Onset detection is done implicitly, as we always output the set of played notes for the last analysed frame. If a note is added to the set of played notes, it can be considered as an onset. However, actual onset detection can improve the results of pitch estimation. For example, when an onset is detected, we can discard all samples from the transient that arises from this onset.

Pitch detection can be broken down into a few subproblems. First, a frequency domain representation of the signal is obtained (e.g. Fourier transform). Then, significant peaks have to be selected from the frequency domain. Based on the found peaks, we have to determine what notes were actually played on the instrument.

4.1 Transform to frequency domain

To estimate the frequency components of a frame, transforms are used. Most commonly, the Fourier transform is used as it is the most researched transform and much is known about it [16]. Its main downside is that the frequency bins are constant in size, where the distance between notes increases exponentially. Because of this, low notes are hard to discern in the frequency domain where high notes have more resolution than needed. Because of this, other transforms such as the constant Q transform (CQT) are getting more popular. Despite this, we will still use the Fourier transform as very efficient implementations exist for this transform. We need this efficiency to remain real-time.

When using the Fourier transform, a window function has to be applied to a frame to prevent spectral leakage [22]. In this paper, we use the Blackman-Nuttall window. Given a window with N samples, the Blackman-Nuttall window $w(n)$ is defined as:

$$\begin{aligned} w(n) &= 0.3635819 - 0.4891775 * \cos(\tfrac{2\pi n}{N}) \\ &+0.1365995 * \cos(\tfrac{4\pi n}{N}) - 0.0106411 * \cos(\tfrac{6\pi n}{N}) \end{aligned}$$

Then, with the signal $s(n)$ and the window $w(n)$, we can obtain the resulting windowed signal $\mathrm{res}(n)$ using:

$$\mathrm{res}(n) = s(n) * w(n)$$

Figure 1 shows the effect of a window function on a frame graphically.

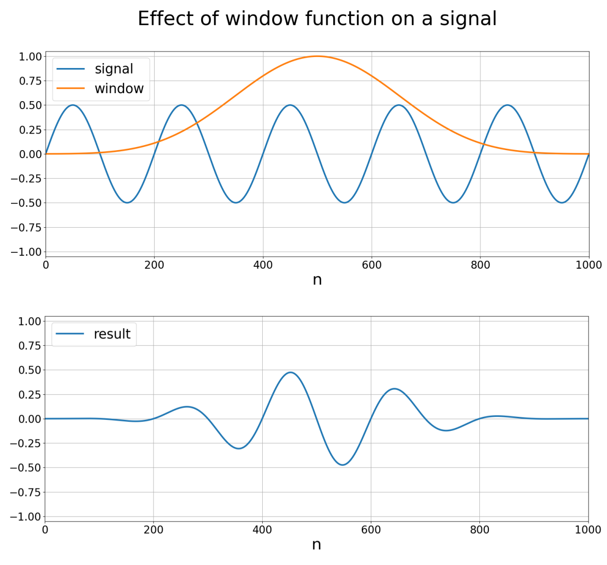


Figure 1: The effect of applying a window function to a signal

The output of a Fourier transform is a list of complex number. The norm of each complex number corresponds to the amplitude of a certain frequency bin. The size of these frequency bins f_{bin} is determined by the sampling rate f_{SR} of the signal (samples per second) and the number of samples in a frame SPF with the following relation:

$$f_{\text{bin}} = \frac{f_{\text{SR}}}{\text{SPF}}$$

Furthermore, the frame length t_{frame} in seconds can be calculated as follows:

$$t_{\text{frame}} = \frac{\text{SPF}}{f_{\text{SR}}}$$

4.2 Peak picking

Peaks in the frequency domain correspond to predominant frequencies in the analysed frame. A peak is defined as a bin which has a higher amplitude than its left and right neighbour.

Not every peak is of interest. Some peaks are generated by noise in the signal. Many of these peaks can be filtered by requiring an arbitrary signal level to be considered a peak. This approach will however result in less sustain.

In order to determine if a peak is significant enough, we first calculate a Gaussian average envelope [8]. For every frequency bin, the Gaussian average is a weighted average of all frequency bins, where the weight is determined by the distance from the considered bin. Given a distance of n bins, the weight is determined by the following Gaussian function:

$$e^{-\pi(\frac{n}{\sigma})^2}$$

Here, σ can arbitrarily be chosen. Higher values for σ make points close by weight more compared to distant points. By using the envelope, we effectively filter small local maxima close to large local maxima.

Every peak which surpass the envelope and is above the arbitrary signal level is deemed significant and used in the next step, the harmonic overtone sieve. We also provided a different variant of the peak picker where only the highest peak is selected for every lobe which surpasses the envelope and a variant which picks all peaks, disregarding the envelope. We also provided a peak picker which picks every peak, which is useful for development purposes.

4.3 Harmonic overtone sieve

Given the significant spectral peaks from the previous step, we can try to calculate what notes they represent. First, we get the actual location of a peak with a Fourier bin using quadratic interpolation [12][26]. Here, we calculate a value $p \in [-\frac{1}{2}, \frac{1}{2}]$, which is the location of the actual peak relative to the found peak in number of Fourier bins. In order to calculate p, we need the height of our found peak ($y(0)$) and the height of the two neighbouring bins ($y(-1)$ and $y(1)$). As quadratic interpolation is more accurate in the dB (logarithmic) scale, we define:

$$\begin{aligned} \alpha &= \log y(-1) \\ \beta &= \log y(0) \\ \gamma &= \log y(1) \end{aligned}$$

Then, we can calculate p as follows:

$$p = \frac{1}{2} \cdot \frac{\alpha-\gamma}{\alpha-2\beta+\gamma}$$

Given p, we can also calculate the amplitude a corresponding to the interpolated location:

$$a = \beta - \frac{(\alpha-\gamma)*p}{4}$$

Now, we can calculate the frequency of the peak i by multiplying the Fourier bin size by $i + p$. Using the obtained frequencies and corresponding amplitudes, we construct note sets. A note contains all useful information about the peak for transcription. Other than the interpolated frequency and corresponding amplitude, we calculate the distance to the closest note in cents and we also store the scientific pitch notation. To calculate the distance between our interpolated frequency f_i and the closest note, we first calculate the tuned frequency f_t:

$$f_t = f_{\mathrm{A}_4} * 2^{\mathrm{round}(12*\log_2(\frac{f_i}{f_{\mathrm{A}_4}}))/12}$$

Then, we can calculate the distance d:

$$d = 1200 * \log_2(\frac{f_i}{f_t})$$

Given an array with all the note names (A, A#, B, C, C# etc.), we can calculate the index which contains the name of the note as follows:

$$i = \mathrm{round}(12 * \log_2(\frac{f_i}{f_{\mathrm{A}_4}}) \mod 12)$$

Here, we assume that "$-1 \mod 12 = 11$", which is not the case in all programming languages. To calculate the octave number, we first calculate the pitch of C_1:

$$f_{\mathrm{C}_1} = f_{\mathrm{A}_4} * 2^{\frac{45}{12}}$$

Using this, we can calculate the octave number o:

$$o = \mathrm{floor}(\log_2(\frac{f_i}{f_{\mathrm{C}_1}}) + 1)$$

Note that this equation fails if we calculate the octave number of a C which is tuned too low ($f_i < f_t$). To remedy this, we can simply add a check if this is the case and increment the octave number accordingly.

Unfortunately, a note set does not only contain fundamental frequencies, but also harmonic overtones. This is only a small problem in monophonic transcription, as there is only one fundamental frequency and its overtones. However, in the polyphonic case, it is difficult to determine which peaks are fundamentals and by extension know how many notes are played at the same time.

To determine if notes are fundamentals or overtones, we developed a "harmonic overtone sieve" algorithm. This algorithm iterates over all found peaks and checks if any higher pitched peak is a possible overtone of the considered peak. As the calculated frequency of a peak may not be accurate, we have to set a threshold on the error under which a peak is considered an overtone. Because the distance between notes increases with frequency, we cannot set a constant threshold of error in Hz. Therefore, we calculate the error in cents and choose a threshold for this error. This also allows for equal error tolerance above and below the note, which percentage based methods would not. Given the frequency of the considered peak f_c and the possible overtone f_o, we can calculate the frequency of the closest harmonic overtone f_h:

$$f_h = f_c * \mathrm{round}(\frac{f_o}{f_c})$$

Using the closest harmonic overtone frequency, we can calculate the error from our possible overtone in cents:

$$e = 1200 * \log_2(\frac{f_o}{f_h})$$

T

For every peak, we keep track of two variables. The first is the number of number of overtones we detected for every peak. The second is a boolean which indicates if a peak is a overtone of another peak. For monophonic note detection, we can assume that the peak with the most overtones is the fundamental frequency. For polyphonic note detection, we can assume that every peak which is not an overtone of another lower pitches peak is a fundamental. Another method for polyphonic note detection would be setting a threshold for the number of overtones a fundamental should have. This however gives inconsistent results, because when the same note is played louder, more overtones are detected. Furthermore, the number of overtones decreases while a note sustains.

A guitar has a limited range of fundamental tones which it can produce. Using this information, we can discard all peaks lower than the lowest fundamental of a guitar as noise and mark all peaks higher than the highest fundamental as overtones or noise. These ranges have to be configurable to accommodate different tunings or guitars with extra strings or frets.

Overtones are dissonant relative to our scale. The error of an overtone is calculated as the number of cents from the closest harmonic note. The errors are different for every overtone, but the series of errors is constant for every harmonic overtone series. In Table 1, we provided the errors of the first 6 overtones.

n	f_{overtone}	closest note	f_{note}	error
0	261.626	C_4	261.626	-
1	523.251	C_5	523.251	0
2	784.877	G_5	783.991	1.955
3	1046.502	C_6	1046.502	0
4	1308.128	E_6	1318.510	-13.686
5	1569.753	G_6	1567.982	1.955
6	1831.379	$A_6^{\#}$	1864.655	-31.174

Table 1: Example of overtone series and the errors compared to the closest note

If only harmonic notes were played on a guitar, we could filter all notes with an error higher than an arbitrary threshold. However, as most guitars are not perfectly intonated and strings detune by pressing them down on a fret, this information has to be used with care. Correctly using this information is out of the scope of this paper.

```
input  : Samples from audio driver
output : Estimation of played note(s)
//Obtain frequency domain
frame ← get_frame();
amps_complex ←
    fourier(apply_window(frame));
amps ← calc_norms(amps_complex);
//Peak picking
envelope ← calc_envelope(amps);
peaks ← pick_peaks(envelope);
//Harmonic overtone sieve
noteset ← create_noteset(amps, peaks);
if monophonic then
    print(noteset.get_likeliest_note());
else
    print_array(noteset.get_likely_notes());
end
```

Algorithm 1: Main transcription loop.

5 Implementation

The program developed for this paper is written in C++ (11) and compiled with g++ (10.2.0). It uses SDL (2.0.14) for audio input/output and spectrum visualisation. FFTW (3.3.9) is used for efficient Fourier transforms. Its code can be found at "`github.com/lucmans/dechord`".

By default, the program assumes an A_4 of 440 Hz. Only 12 tone equal temperament (12-TET) tuning is currently supported. Our framework can be changed to other TET tunings, but non TET tuning is not easily supported.

An pseudo-code overview of the transcription process is given in Algorithm 1. We first obtain a frame of samples from the audio driver. After applying a window function to it, we perform the Fourier transform on it. To get the amplitudes of every frequency bin, we have to calculate the norm of the output of the Fourier transform. We then calculate the Gaussian envelope of the frequency domain and find the location of these peaks. Using the peak locations, we can interpolate the actual frequency corresponding to each peak and build a set of notes from these peaks. Using our harmonic overtone sieve algorithm, we can perform monophonic or polyphonic transcription.

5.1 Transform to frequency domain

As mentioned before, the resolution of the Fourier transform depends two parameters, the sampling rate and the number of samples in a frame. When using a higher sampling rate we increase the resolution of the Fourier transform. This also increases the computation power required to process one second of samples. We are limited to a sampling rate of 192 kHz, as this is the maximum for most audio interfaces and audio drivers. To allow guitarists to play mildly fast, we have to limit our frame length. If the frame size if too large, multiple separate notes may be included in one transform, which would make it seem like the notes were played at the same time. Moreover, frame sizes which are a power of two can be transformed more efficiently, which further restricts are frame size choice. For example, if we choose a frame size of $2^{14} = 16384$ samples at a sampling rate of 192 kHz, the frame length is 85.33 ms (11.72 frames per second). The corresponding Fourier bin size is 11.72 Hz. At first glance, such a large bin size seems like a big problem, as the smallest interval that can be played on a guitar ($G\#_2$-A_2) is about 6 Hz. However, the overtones of these notes will have more Hz between them, which makes to possible to differentiate the two notes.

Before applying the transform, a window function is applied to minimize the artifacts produced by the transform. The following windows are also provided: Hamming window, Hann window, Blackman window, Nuttall window, Blackman-Harris window, Blackman-Nuttall window, Flat top window. Other window functions can easily be added.

Because our input data is purely real, we know the output of the Fourier transform will be symmetric. For this reason, we can omit calculation half of the output. This results in an almost two times speed-up [2].

5.2 Peak picking

Three different peak picking algorithms have been implemented. The main algorithm is based on the Gaussian envelope. First, the relevant points on the Gaussian function are calculated and stored in an array. Normally, at least as many values have to be precomputed as there are frequency bins. However, because values further in the tail are close to zero, we can omit calculating these values and subsequently skip calculation involving them in the Gaussian average, which serves as a slight optimization. Algorithm 2 shows how the values on the Gaussian function are computed given a `kernel_width`, which is the number of values we precompute.

```
input  : Compile time configuration
output : Gaussian weights
1  //kernel_width has to be odd
2  mid ← floor(kernel_width / 2);
3  for $i$ ← 0 to kernel_width do
4      gaussian[$i$] ← $e^{-\pi*(\frac{i-\text{mid}}{\sigma*\text{mid}})^2}$;
5  end
```

Algorithm 2: Calculate the Gaussian weights based on compile-time configuration.

Using our previously computed values, we can calculate the Gaussian average envelope as shown in Algorithm 3. The envelope is calculated for every Fourier bin i by calculating the weighted average of `kernel_width` bins around i. As mentioned before, higher values of σ result in points closer to i to be weighted higher. In line 7, we use the `min` and `max` function to prevent array accesses outside of array bounds. Using this envelope, we can select all significant peaks as shown in Algorithm 4. By commenting line 4, we obtain the algorithm which selects all peaks. Line 5 checks if a peak is loud enough to be considered a peak, which prevents finding peaks in noise.

The other method based on the Gaussian average envelope only select one peak every time a lobe surpasses the envelope. A possible implementation is given in Algorithm 5. Here, we keep track track of the highest peak found so far. Every time the amplitude of a bin dips below the envelope, we select the highest found peak and reset the currently highest found peak. This results in always picking the highest peak from a single lobe which surpasses the envelope. We split the if statement on line 7 into two check to prevent an invalid array access when `highest` is -1.

5.3 Harmonic overtone sieve

An overview of the monophonic is given in Algorithm 6. Here, we iterate over all found peaks and check if every higher peak can be an overtone of the considered peak. After counting the number of overtones for every peak, we can pick the peak with the most overtones.

Polyphonic note detection can use an algorithm similar to the monophonic algorithm. As the number of played notes is unknown in the polyphonic case, we have to determine if a note is a fundamental or overtone. As mentioned before, we cannot just set a threshold for the number of overtones a fundamental should have. To solve this problem, we present a harmonic overtone sieve algorithm. A pseudo-code implementation of this algorithm is shown in Algorithm 7. Instead of directly looking for fundamentals, we try to sieve away all harmonics until we are left with fundamentals. We start at the lowest peak end mark all its possible overtones. If there are still unmarked peaks left, they are likely from another note and we sieve its overtones away. We can repeat this process until no more peaks can be sieved. Lastly, we can remove all peaks which are outside of the frequency range of a guitar. If two peaks are not distinguishable in the root notes but are distinguishable in the overtones, this algorithm will report the first distinguishable overtone as the root note. It is possible to calculate what the root note should have been, but that is outside the scope of this paper and will be further discussed in Section 8.

5.4 Latency

As stated before, our goal is to perform the transcription in real-time. We choose a constraint of 6 ms. Most musicians work with a driver latency of 9 to 11 ms. On a good computer, this latency can be reduced to 3-5 ms. We will use these 6 ms of latency difference to perform our calculations in. Note that when playing more notes per second, latency becomes more of a problem. On slower pieces, we can get away with much larger latencies. In order to test the impact of extra latency for yourself, we developed a tool which plays back the incoming audio signal with a user specifiable delay. This tool can be found under the "latency" branch in our GitHub repository. Note that this tool adds a latency on top of the audio driver latency like our program would. The total latency is much larger than the displayed latency.

The latency of our system results from two different latencies, the time to fill a frame with samples and the frame processing time. Because our algorithm works on a "per frame" basis instead of a "per sample" basis,

```
input   : Norms of the Fourier transform output
output : Signal envelope
gaussian ← calc_gaussian();
mid ← floor(kernel_width / 2) ;  //kernel_width has to be odd
// Only use first half because of symmetry
for $i \leftarrow 0$ to (frame_size/2) + 1 do
    sum ← 0;
    weights ← 0;
    for $j \leftarrow$ max(-mid, $-i$) to min(mid, (frame_size/2) + 1) do
        sum ← sum + (norms[$i + j$] * gaussian[$j$ + mid]);
        weights ← weights + gaussian[$j$ + mid];
    end
    envelope[$i$] ← sum/ weights;
end
```

Algorithm 3: Calculate the envelope.

```
input   : Norms of the Fourier transform output and signal envelope
output : List of peak locations
peaks ← {};
for $i \leftarrow 1$ to frame_size/2 do
    if norms [$i - 1$] < norms [$i$] ∧ norms [$i$] > norms [$i + 1$]
            ∧ norms [$i$] > envelope [$i$]
            ∧ norms [$i$] > threshold then
        peaks ← peaks ∪ {$i$};
    end
end
return peaks;
```

Algorithm 4: Peak picking using the Gaussian envelope, where all peaks are picked which surpass envelope.

```
input   : Norms of the Fourier transform output and signal envelope
output : List of peak locations
peaks ← {};
highest ← −1;
for $i \leftarrow 1$ to frame_size/2 do
    if norms [$i - 1$] < norms [$i$] ∧ norms [$i$] > norms [$i + 1$]
            ∧ norms [$i$] > envelope [$i$]
            ∧ norms [$i$] > threshold then
        if highest = −1 then
            highest ← $i$;
        else if norms [$highest$] < norms [$i$] then
            highest ← $i$;
    else if norms [$i$] <= envelope [$i$] then
        peaks ← peaks ∪ {$i$};
        highest ← -1;
end
return peaks;
```

Algorithm 5: Peak picking using the Gaussian envelope, where only one peak is picked every time a lobe surpasses the envelope.

```
input   : Set of notes (sorted on frequency)
output  : Likeliest played note
n_notes ← notes.size();
if n_notes == 0 then
    return;
end
//Array is initialized with zeros
n_harmonics [n_notes] ← {} ;
for $i ← 0$ to n_notes do
    for $j ← i + 1$ to n_notes do
        harmonic_overtone ← notes[$i$] * round(notes[$j$].frequency / notes[$i$].frequency);
        error ← 1200 * log2(notes[$j$].frequency / harmonic_overtone);
        if error > -errorthres ∧ error < errorthres then
            n_harmonics[$i$] ← n_harmonics[$i$] + 1;
        end
    end
end
maxidx ← 0;
for $i ← 0$ to n_notes do
    if n_harmonics[$i$] > n_harmonics[max_idx] then
        max_idx ← $i$;
    end
end
return notes[max_idx];
```

Algorithm 6: Monophonic transcription.

```
input  : Set of notes (sorted on frequency)
output : Set of likely notes
output ← {};
n_notes ← notes.size();
if n_notes == 0 then
    return;
end
//Array is initialized with false
explained [n_notes] ← {} ;
for $i \leftarrow 0$ to n_notes do
    for $j \leftarrow i + 1$ to n_notes do
        harmonic_overtone ← notes[$i$] * round(notes[$j$].frequency / notes[$i$].frequency);
        error ← 1200 * log2(notes[$j$].frequency / harmonic_overtone);
        if error > -errorthres ∧ error < errorthres then
            explained[$i$] ← true;
        end
    end
end
for $i \leftarrow 0$ to n_notes do
    if ¬explained[$i$] then
        output ← output ∪ notes[$i$];
    end
end
return output;
```

Algorithm 7: Polyphonic transcription.

we have to wait until the audio driver has fetched enough samples for us to fill a frame. The time it takes to start analysing a frame after receiving the first sample of this frame can be calculated by dividing the number of samples per frame by the sample rate. This is equal to the length of a frame. In practical terms, it is the maximum latency on detecting a newly played note. The average latency is obtained by dividing this number by two.

We use a frame size of 16384 samples and a sample rate of 192 kHz, which equates to a frame length of 85.3 ms. This is well over our real-time constraint. Unfortunately, we cannot change this value too much. The sample rate is already at the maximum of most audio interfaces. We also cannot lower the number of samples per frame while using the Fourier transform, as it would further reduce the frequency resolution. However, as will be explained in Section 8, this might not be a problem, as we can reduce this latency by using partially overlapping frames.

We can control the second kind of latency by optimizing our algorithms. In Section 6, we will measure the processing time of the different algorithms.

6 Experiments

We will perform several experiments to test the performance of various aspects of our system. During development, an electric guitar with active EMG humbucker pickups was used. To convert the analog signal to a sampled digital signal, a Behringer UMC404HD was used. This audio interface has a dynamic range of 100 dB. As is the case for most audio interfaces, this is much less than the dynamic range of a sample (144 dB for 24 bit integer). We opted not to amplify the signal in software, as it will distort it; especially if not done carefully. The tests were performed on an AMD 3700X CPU.

We constructed our own dataset to evaluate the performance. Our dataset consists of scales and intervals played on a guitar. Scales allow us to measure the monophonic performance. As scales are easy to play, multiple recordings should have the same results. This allows us to isolate and measure other variables, such as the effect of tempo or playing the scale higher/lower. Recordings of different intervals allow us to measure the polyphonic performance. By measuring all two note intervals, we can identify any problems with specific intervals. As the distance in frequency in an interval increases when the same interval is played at higher notes, we record the intervals at multiple root notes.

Note that when repeating our experiments, your results may vary slightly. This is because the results of the Fourier transform from FFTW are not consistent [1]. This is especially noticeable in inconsistent latencies.

TODO: List of parameters!

6.1 Monophonic performance: scales

To test the monophonic performance of our system, we first measure the performance of scales at different pitch ranges. The scales are played at a speed of two notes per second. This equates to playing fourth notes on 120 BPM. We play the scales at a relatively slow speed, such that this cannot become a limiting factor. The slower speed also helps to see if mistakes are the result of transients, as these mistakes should vanish as the note persists. We only play the C major scale, but change the starting note (mode). We play the full vertical scale from this starting note [4]. Each position has 17 notes, except for F_2, which has 18 notes. In Table 2, we summarize the results of the experiment. We denote the different starting positions as n_{start}. For every starting position, we denote the accuracy (a), the accuracy without transient errors (a_{trans}), the accuracy without octave errors a_{oct} and the adjusted accuracy without both of these errors (a_{adj}). The accuracy is calculated by dividing the number of correctly determined frames by the total number of frames with notes. The only frames without notes are positioned at the start and end of a file. We omitted these frames from the results, as they are always correctly determined to be empty frames. Erroneous notes due to transients usually have a lower amplitude where octave errors usually have a higher amplitude compared the amplitude of the base note. This allows is to distinguish between them. This information cannot be used to detect transients in real-time, as we compare the amplitude of the erroneous note to amplitude of the correct note in the following frames.

n_{start}	a	a_{trans}	a_{oct}	a_{adj}
E_2	91.4%	95.2%	96.2%	100%
F_2	92.3%	96.2%	96.2%	100%
G_2	92.0%	95.0%	97.0%	100%
A_2	93.0%	94.0%	99.0%	100%
B_2	92.8%	96.9%	95.9%	100%
C_3	92.8%	92.8%	100%	100%
D_3	96.0%	98.0%	98.0%	100%
E_3	96.7%	96.7%	100%	100%

Table 2: Accuracy of monophonic f_0 estimation on C major scale at two notes per second for different starting positions. a_{trans} is the accuracy without transient errors and a_{oct} is without octave errors. a_{adj} is the accuracy without both of these errors.

We can see that the transient errors occur about 96% of the time. There are two outliers at C_3 and D_3, which are the cause of inconsistent quality in the recording of the scales. This shows us that the guitarist has to play carefully for the best results. Furthermore, we noticed that the first played note almost always has a transient error. This could be because we start playing from a silent signal instead of changing from a previous note. Also, the transient errors show up more often between lower frequency notes. Almost all octave errors are at note changes, so better transient filtering will also solve these cases. Other than these errors, the system seems to work perfectly.

6.2 Polyphonic performance: intervals

To test the polyphonic performance of our system, we let it determine the played notes in an interval. We play every interval within an octave to test if there are any problems with separating the notes in specific intervals. For instance, some interference may cause oscillating peaks in the frequency domain. Again, we try all the intervals at different root notes, as the pitch of the root note may make a big difference in performance. As we mentioned before, when two peaks are too close together, we cannot distinguish them. Often, the second note is distinguished an octave higher. We set the overtone error threshold to 10 cents. This is a relatively small value, but this allows us to see small problems in our algorithms better. Furthermore, as shown in Table 1, we should not use a constant value, but use a different value for every n-th overtone. In Tables 3–5 is a summary of the results. Here, we list the number of semitones n between the two notes in an interval, if both notes were correctly found. If not, we denote if the second note was detected an octave higher. This often happens, because we cannot distinguish two notes in small intervals at low pitches. The last column shows if there were other notes reported which were not transients.

In the tables, we can see that the algorithm performs well on higher pitched intervals. On medium pitched intervals, the small intervals become more of a problem, but most intervals are detected correctly. Transients are however visible for longer in the small intervals. Big problems arise when we look at the lower pitched intervals. There is much noise in the results and many slightly inharmonic overtones are reported. These could be filtered out by increasing the overtone error threshold, but that does not solve the overtones being inharmonic. The inharmonic overtones are less of a problem for larger intervals.

n	both	octave	other
1	false	true	1-3 slightly inharmonic overtones
2	false	false	-
3	false	true	C_4 and G_4 (+ random at start)
4	true	-	Sometimes octave instead and 1-2 inharmonics at start
5	true	-	1 slightly inharmonic overtone
6	true	-	1-3 slightly inharmonic overtones
7	true	-	no
8	true	-	no
9	true	-	occasional inharmonic overtone
10	true	-	1-2 slightly inharmonic overtones
11	true	-	1-2 slightly inharmonic overtones

Table 3: Performance of distinguishing notes in an interval with root note A_2. Given an n semitone interval, we list if we find both notes and if not, if one of the notes was found an octave higher. We also list the other reported notes.

n	both	octave	other
1	false	true	about 4 slightly inharmonic overtones
2	false	true	about 4 slightly inharmonic overtones
3	true	-	$C\#_6$ or E_6 in first few frames
4	true	-	no
5	true	-	no
6	true	-	no
7	true	-	E_4 often filtered
8	true	-	random note 3 times
9	true	-	no
10	true	-	no
11	true	-	no

Table 4: Performance of distinguishing notes in an interval with root note A_3.

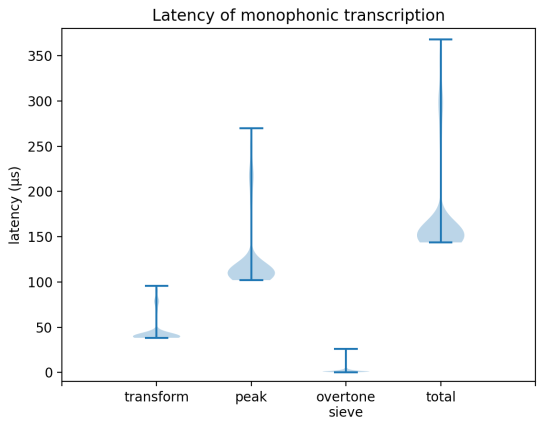
n	both	octave	other
1	false	true	occasional E_6
2	true	-	no
3	true	-	no
4	true	-	no
5	true	-	no
6	true	-	no
7	true	-	no
8	true	-	no
9	true	-	no
10	true	-	no
11	true	-	no

Table 5: Performance of distinguishing notes in an interval with root note A_4.

6.3 Latency

The processing time of our system is very important, as it will directly impact the latency. We will only measure the time between receiving a frame from the audio driver until the answer is computed, as the time spent waiting on enough samples is fully determined by the number of samples in a frame and the sampling rate. The total processing time and the processing time of the individual steps and the total system can be seen in Figure 2. Here, we only measure the latency on frames which contain peaks.

The total latency is never higher than 360 microseconds. Given that a frame is 85 milliseconds, we are at least 235 times real-time. This is well within out real-time constraint if we disregard the latency to fill a frame. Surprisingly, the peak picking algorithm is more computationally intensive than the transform. Most of the time, the latency is on the lower end of the distribution. The outliers are mostly a result of transients. During transients, there are a lot more peaks to process, which slows down the different algorithms. This is especially the case for polyphonic transcription, as we have to take all the peaks into account. In the polyphonic dataset, transients contain more peaks as multiple strings are strummed. This results in a second lobe in the mid segment of the latency distribution.



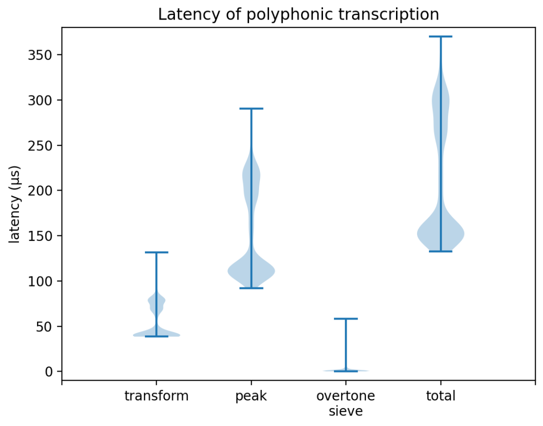


Figure 2: Processing time of the different algorithms. The bar shows the fastest and slowest measured time. The lobe shows the relative distribution of the measurements.

7 Conclusions

In this paper, we presented and implemented a Fourier transform based AMT system and tested if these methods are feasible for real-time transcription. We divided the task in three components which can all be separately interchanged to facility specialized research in one of the components. The components are: obtaining the frequency domain representation, spectral peak picking and note selection. As shown in Section 6.3, these tasks can be performed well within our real-time constraint of 6 ms. However, when using Fourier transform based methods, we need large frame sizes to get the required accuracy to discern lower pitched notes. Given the maximum sampling rate of widespread audio interfaces, it takes too long to fill a frame. The current implementation requires frame sizes of 16384 samples for the necessary resolution, which equates to frame lengths of 85 ms. As described in Section 8, we might still be able to use Fourier transform based methods for real-time transcription. However, other transforms such as CQT or wavelet transform might be better suited for real-time transcription, as we can still spare much computation time to lower the frame-wait time.

Even though we do not meet our real-time goal of 6 ms, we do have a high monophonic accuracy of 90%. Almost all of the mistakes the cause of transients and fall between two notes. The other occasional mistake is an octave error, which is a difficult problem to solve. If we account for these two kinds of errors, our system has a 100%. Our proof-of-concept harmonic sieve algorithm is capable of filtering many harmonic overtones and is able to distinguish between all intervals of more than 3 semitones. It is also able to isolate the two played notes on higher pitched intervals, however, there still is much noise on lower pitched intervals.

Our peak picking algorithm works fine for monophonic transcription, however, it does not work too well for polyphonic transcription. For instance, peaks in close distance to each other are easily missed. It is also compute intensive when compared to the other components and has the largest outliers. Even though the outlier are relatively very severe, they are all within 200 microseconds and might not be noticeable.

All code related to this project can be found at "`github.com/lucmans/dechord`".

8 Future work

Our program serves as a good starting point for further research. However, there are still some problems with our current implementation. Solving these is outside of the scope of this paper and will be the topic of future work.

The first big problem is transients. They will appear as many random peaks in the frequency domain. Many of these peaks will be outside of the fundamental frequency range of a guitar, but not all of them, so discarding all high peaks does not solve the problem. Usually, during transients, there any significantly more peaks and even multiple peaks within one note. If this can pattern can consistently be detected, samples can be discarded until the transient is filtered.

The Fourier transform requires too many samples in a frame to remain within the real-time constraint while still providing the necessary frequency resolution in the fundamental frequencies. It might be possible to alleviate this problem with other transforms, such as the constant-Q transform or the wavelet transform, but those transforms may have to solve other problems. As mentioned before, the low resolution does not have to be a problem, as less resolution is needed when discerning higher pitched peaks due to the exponential nature of octaves. Currently, we do not detect enough overtones, but some testing showed us we can detect many overtones if we amplify the signal. This has to be done carefully, as artifacts may be introduced in the signal. Optimally amplifying of a recording is trivial, however, it is trickier in real-time. Using amplified signals would require recalibration of the peak picking parameters or even require a new algorithm in general.

The current implementation has no onset detection. We stated before that onset detection may help to filter out transients. If we can detect the transient in the time domain, we can discard all samples containing the transient. This may however be very difficult to do in real-time, because when we detect a transient in a transformed frame, we are too late to act on it due to the large frame times. We can circumvent this by transforming partially overlapping frames. This could be done in different threads to limit the extra computational pressure which could impact our latency. However, as we are about 240 times real-time, we could likely perform it single threaded. Now, if we have a bad frame, we have to wait a full frame (85 ms) for the next estimation. However, if we use n partially overlapping frames, we can try another estimation in $85/n$ ms, given enough computation power. This will increase the responsiveness of our system greatly.

It is virtually impossible to know on what string a certain note was played. Furthermore, when playing an octave, the overtones of the lower pitched note align perfectly with the higher note and its overtones. If the strings were deliberately detuned to orthogonal frequencies, we can easily distinguish between the played strings. This does however create more peaks in the frequency domain. This trade-off is worth further research.

References

[1] FFTW page on non-determinism of the Fourier transform.
`http://fftw.org/faq/section3.html#nondeterministic`
Last accessed on 02-04-2021.

[2] FFTW3 page about two times speed-up by omitting complex part.
`http://www.fftw.org/fftw3_doc/One_002dDimensional-DFTs-of-Real-Data.html`
Last accessed on 12-04-2021.

[3] FFTW3 page on the speed of transformations.
`http://www.fftw.org/speed/`
Last accessed on 12-05-2021.

[4] Full vertical scale positions.
`https://www.all-guitar-chords.com/`

`scales`
Last accessed on 05-04-2021.

[5] Webpage with interactive tool that shows the effect of zero-padding.
`https://jackschaedler.github.io/circles-sines-signals/zeropadding.html`
Last accessed on 01-04-2021.

[6] Eric J. Anderson. Limitations of short-time fourier transforms in polyphonic pitch recognition. 1997.

[7] Z. Ding and L. Dai. A study of constant Q transform in music signal analysis. 2005.

[8] Z. Duan, Y. Zhang, C. Zhang, and Z. Shi. Unsupervised single-channel music source separation by average harmonic structure modeling. *IEEE Transactions on Audio, Speech, and Language Processing*, 16(4):766–778, 2008.

[9] X. Fiss and A. Kwasinski. Automatic real-time electric guitar audio transcription. 2011.

[10] T. A. Goodman and I. Batten. Real-time polyphonic pitch detection on acoustic musical signals. *2018 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, 2018.

[11] F. Guichard and F. Malgouyres. Total variation based interpolation. 1998.

[12] Julius Orion Smith III. Spectral audio signal processing. 2013.
`https://www.dsprelated.com/freebooks/sasp/Quadratic_Interpolation_Spectral_Peaks.html`
Last accessed on 26-03-2021.

[13] Kristoffer Jensen. Timbre models of musical sounds.

[14] S.S. Limaye K.A. Akant, R. Pande. Accurate monophonic pitch tracking algorithm for QBH and microtone research. *The Pacific Journal of Science and Technology*, 11(2):342–352, 2010.

[15] Emmanouil Benetos, Simon Dixon, Dimitrios Giannoulis, Holger Kirchhoff, Anssi Klapuri. Automatic music transcription: challenges and future directions. *Journal of Intelligent Information Systems*, 41:407–434, 2013.

[16] Tiago Fernandes Tavares, Jayme Garcia Arnal Barbedo, Romis Attux, Amauri Lopes. Survey on automatic transcription of music. *Journal of the Brazilian Computer Society*, 19:589–604, 2013.

[17] Douglas A. Lyon. The discrete fourier transform, part 4: Spectral leakage. *Journal of Object Technology*, 8(7):23–34, 2009.

[18] Sylvain Marchand. An efficient pitch-tracking algorithm using a combination of fourier transforms. 2008.

[19] Michael F. Zbyszyński Matthew Wright, Ryan J. Cassidy. Audio and gesture latency measurements on Linux and OSX. 2004.

[20] James A. Moorer. On the transcription of musical sound by computer. *Computer Music Journal*, 1(4):32–38, 1977.

[21] Meinard Müller. *Fundamentals of Music Processing.* 2015.

[22] K. M. M. Prabhu. *Window Functions and Their Applications in Signal Processing.* 2014.

[23] A. Schutz and D. Slock. Periodic signal modeling for the octave problem in music transcription. 2009.

[24] R. S. Shankland and J. W. Coltman. The departure of the overtones of a vibrating wire from a true harmonic series. *The Journal of the Acoustical Society of America*, 10(3):161 ff, 1939.

[25] Adrian von dem Knesebeck, Udo Zölzer. Comparison of pitch trackers for real-time guitar effects. 2010.

[26] Kurt James Werner. The XQIFFT: Increasing the accuracy of quadratic interpolation of spectral peaks via exponential magnitude spectrum weighting. 2015.