Fourier transform based real-time pitch estimation

Assessing the limits of Fourier transform based real-time pitch estimation and construction of a pitch estimation framework

Luc de Jonckheere

June 13, 2022

Abstract

Short summary. Note that even though we focus on guitar pitch estimation, the methods described in this thesis are general to all instruments and even singing. The only guitar specific thing is the parameter optimization. We focus on guitar as it is not easily replaced by a MIDI variant as is possible with instruments such as a piano.

Contents

1 Introduction

Pitch estimation, which is also referred to as f0 estimation, is an important subtask within the field of Automatic Music Transcription (AMT). The goal of pitch estimation is to estimate the pitch or fundamental frequency f_0 of a given signal. In the context of AMT, pitch estimation is used to determine which notes are played in a given signal.

Real-time pitch estimation is a subproblem where we want to estimate the note associated with the measured pitch while the musician is playing it with minimal latency. This entails we have to use the latest received signal. In contrast to non-real-time methods, we have no knowledge of what may happen ahead of time (we cannot peak-ahead) and signal corresponding to previous notes is mostly irrelevant. This limits the methods we can use to solve this problem.

If pitch estimation can accurately be performed in real-time, it can be used to create a digital (MIDI)

instrument from an acoustic instrument. This digital instrument can then be used as an input for audio synthesizers, allowing musicians to produce sounds from a wide variety of instruments. Furthermore, accurate real-time pitch estimation can be used to automatically correct detuned instruments by pitch shifting the original signal to the closest harmonious note.

The Fourier transform is often used for pitch estimation. The transform decomposes a signal into the frequencies that make up the signal. Predominant frequencies in the signal show up as spectral peaks in the frequency domain. These peaks are important to human perception of melody [12]. Other popular methods used for pitch estimation include non-negative matrix factorization, autocorrelation, statistical model based estimation and hidden Markov model based estimation.

Our research focuses on monophonic pitch estimation. Here, we assume the signal contains at most one note. It is much easier to perform monophonic pitch estimation compared to polyphonic pitch estimation [13], especially when using Fourier transform based methods, as fundamental limits of the Fourier transform inhibit our ability to discern two low pitched notes [6]. Furthermore, hexaphonic guitar pickups are becoming more widespread, which allows us to view the guitar as six monophonic instruments instead of one six-way polyphonic instrument. State of the art commercial guitar synthesizer solutions also use these hexaphonic pick-ups, which indicates the infeasibility of accurate and responsive real-time polyphonic pitch estimation.

This thesis builds upon a preliminary research project [9]. In our research project, we found that Fourier transform based pitch estimation methods might not be well suited for real-time use due to fundamental limitations of the Fourier transform [7]. In this work, we will further research if Fourier transform based methods are viable, as real-time pitch estimation research often relies Fourier transform based methods.

Researching pitch estimation is not accessible, as many other small problems have to be solved in order to produce a working prototype on which experiments can be performed. Furthermore, automated experimentation is a lot of work to implement and as a consequence, pitch estimation research often only includes some informal testing or no experiments at all. To combat these problems, we set out to create a modular pitch estimation framework in which every pitch estimation subproblem is implemented as a separate module which can be replaced. This allows the research of one specific subproblem and the effect of specific combinations of subproblems. In addition, it provides researchers with many tools which aid in developing and fine-tuning the solutions to the different subproblems.

TODO: Rewrite this paragraph. The goal of this thesis is to research the limits of Fourier transform based real-time pitch estimation. To correctly assess the limits, we develop a pitch estimation framework. This framework will focus on extensibility and the ability to perform automated tests. This is important, as much work in this field does not provide its associated source code. This limits the ability to build on other's work and hinders direct comparisons between different methods. Our framework can provide a common ground for the different methods and algorithms to be implemented and compared in. The framework is available at `www.github.com/lucmans/digistring`.

2 Related work

Much research has been performed on Fourier transform based real-time pitch estimation. All research we found relies on obtaining a high resolution frequency domain in which spectral peaks can be isolated and notes can be associated with. These methods are deemed infeasible by some due to low frequency resolution [7].This is especially problematic when adhering to a real-time constraint, as extra short signal frames have to be used. Some papers circumvent this problem by choosing a very high real-time constraint [17, 14], however, this inhibits the use for real-world applications. On top of the latency from the pitch estimation algorithm, conventional operating systems also have a latency when delivering audio samples to your program due to how audio drivers work [24].

A big problem with Fourier based pitch estimation is the occurrence of overtones [25]. Especially octaves are a problem, as the fundamental and all overtones of the higher note overlap with overtones of the lower note. This is referred to as the octave problem [28]. Overtones are periodic in nature, as they diminishingly repeat every multiple of the fundamental frequency. As a consequence, they could also be detected using a subsequent Fourier transform [21] on the frequency domain. However, this does not solve the octave problem.

Many different transform have been researched for pitch estimation, however, Fourier transform remains popular as it is broadly studied and its behavior is well known [23]. Lately, the CQT transform is gaining popularity as it may provide higher resolution

in the frequency domain [33] at the cost of lower computational efficiency [29]. However, the CQT transform is efficiently implemented using Fourier transforms [8] and the main problem with Fourier is the fundamentally low frequency resolution on short frames, so we are left with the same problem. One big advantage is that the frequency bins can perfectly align with the notes of an instrument [10]. However, as described in Section 3.5, overtones are dissonant with respect to our notes and consequently, the CQT bins do not align with the overtones. If a note perfectly aligns with a Fourier bin, all overtones will also align. In order to cover every note, we could instead perform 12 Fourier transform in parallel. This difference is especially important when performing polyphonic pitch estimation.

TODO: Note on problems with other research, such as no experiments, no available source code, assuming normal continuous Fourier behavior instead of DFT, downsampling

TODO: Vergelijk materiaal

3 Preliminaries

In order to effectively research Fourier transform based real-time pitch estimation, it is important to have a thorough understanding of how computers deal with audio and how the Fourier transform is implemented in computers. Furthermore, in order to use the output of the Fourier transform, we need to understand the characteristics of the sound generated by instruments. Moreover, in order to interpret the results of our estimated pitch and reason about the performance of the used algorithms, some music theory is necessary. Lastly, we will discuss the concept of real-time in depth, as there are multiple definition for real-time which often leads to incorrect use of the concept.

3.1 Audio in computers

Audio in computers is represented through a series of equally spaced samples. A sample is the height of the audio wave at a specific point in time. The sample rate determines the number of samples per second used for representing the audio. The sample format determines how the height of the audio wave is represented in a sample. Often used formats are 8/16/24 bit integers and 32 bit IEEE-754 floating point numbers (which we will refer to as floats). The integer samples simply uniformly spread the range of the waveform over range of the integer [32]. Float samples typically take values in $[-1, 1]$. Samples with values outside this range are considered to be clipping. Because float samples have 24 bits precision (23 mantissa bits and a sign bit [5]), 24 bit integer samples can converted lossless to float samples. The 8 exponent bits can be used to scale the samples to a different order, which allows us to accurately describe very soft and loud audio. Float samples have many advantages over integer samples for digital signal processing [32], so we will always use float samples throughout this thesis.

When working with audio input/output in computers, a small latency is always introduced [1]. One part of the latency comes from the used audio hardware and is not configurable. The other part comes from the audio driver's buffers. Audio drivers work on buffer of samples instead of single samples for computational efficiency. A full buffer of data has to be gathered from the audio in, or a full buffer has to be send to the audio out, so the first or last sample respectively is one buffer length behind. The buffer length is calculated by dividing the number of samples in the buffer by the sample rate. In order to minimize latency, the number of samples per buffer has to be minimized and the sample rate has to be maximized. Since these latencies are outside of Digistring's control and can be mostly circumvented by running Digistring's algorithms on specialized hardware, we won't take them into account in this thesis.

3.2 Fourier transform

The Fourier transform is a mathematical transform which transforms a function of time to a complex valued function of frequency and phase. Here, the magnitude represents the amplitude and the argument represents the phase of the corresponding sine wave. The function which maps the frequencies to amplitudes is called the spectrum of the time dependent function. The Fourier transform works on continuous functions and assumes an infinite time interval. Concepts such as continuous and infinite cannot be represented by a computer. Consequently, the discrete Fourier transform (DFT) has to be used for Fourier analyses on computers. The DFT can efficiently be calculated using the fast Fourier transform (FFT) algorithm.

The DFT transforms a finite sequence of equally spaced samples, which we will refer to as a frame, into an equal number of complex values representing the amplitude and phase, which we refer to as bins. Technically, the bins do not form a spectrum as they are not continuous, however, within digital signal processing it is still referred to as the spectrum of the frame.When working with audio, the samples

are real valued, and the DFT output is symmetrical. Because of this, we can discard the second half of the output. In the rest of this thesis, we will only consider the first half of the output. Each bin corresponds to a specific frequency. All other frequencies are spread out over multiple bins. This is called spectral leakage and will be discussed later. Given a frame F, the number of samples in the frame is $n_F = |F|$. Using n_F and sample rate f_{SR}, we can calculate the distance between bins in Hz:

$$\Delta f_{bin} = \frac{f_{\text{SR}}}{n_F}$$

This is also referred to as the frequency resolution. Closely related to the frequency resolution is the frame length, which is calculated as follows:

$$t_F = \frac{n_F}{f_{\text{SR}}} = {\Delta f_{bin}}^{-1}$$

Given a bin number $i \in [0, \lfloor \frac{n_F}{2} \rfloor]$, the frequency of a bin can be calculated as:

$$f_{bin} = \Delta f_{bin} * i$$

The 0 Hz bin corresponds to the so called DC offset. This is the average amplitude of the signal. The frequency of the last bin is also called the Nyquist frequency and is equal to half the sample rate. The Nyquist frequency is important for two reasons [36]. The first relates to the sampling theorem, which states that if a continuous function is sampled at a rate of f_{SR} and only contains frequencies f for which $f \leq \frac{f_{\text{SR}}}{2}$, the samples will completely determine the original function. In other words, the samples perfectly describe the original waveform. Secondly, frequencies f for which $f > \frac{f_{\text{SR}}}{2}$ are spuriously moved into $[0, \frac{f_{\text{SR}}}{2}]$. This implies we cannot simply downsample the input signal for an easy performance gain, as it might introduce noise.

The DFT assumes the frame to be periodic. In other words, the frame is regarded as infinitely repeating. This may distort the waveform if the beginning and end of a frame do not align and leads to artifacts in the spectrum. For example, in Figure 1, we take a frame shown by the red lines. The frame is not aligned to a period of the waveform and causes a distortion when repeated as seen in the second graph. Note that only sine waves with frequencies which are a multiple of Δf_{bin} would exactly fit the frame. In our example, the distortion introduces a new high frequency component from suddenly going up and down around the frame border. It also introduces a low frequency component, as the distance between two peaks within a frame is shorter can the distance of two peaks between frames. The introduction of these new frequencies is a form of spectral leakage, as the peak in the continuous spectrum corresponding to the sine frequency leaks into multiple bins. We can smooth the distortion around frame borders by forcing the beginning and end of a frame to align. This is commonly done by applying a window function to the frame, which forces both ends of the frame to zero.

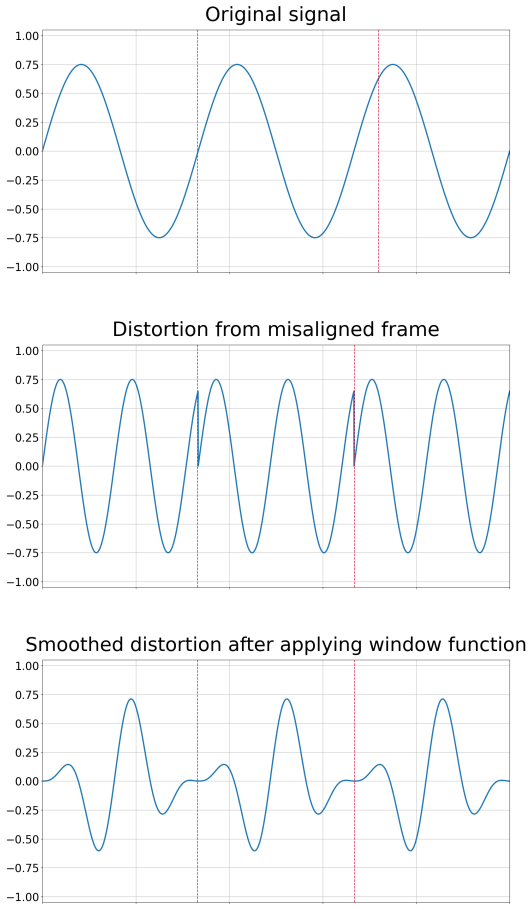

Figure 1: Distortion caused by misaligned framing.

Given signal $s(n)$ and window function $w(n)$, we get the resulting windowed signal $\text{res}(n)$ using:

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

Figure 2 shows the working of a window function on a frame graphically.

The characteristics (TODO: better word) of spectral leakage from framing can be controlled using different window functions. Applying no window function is referred to as using a rectangular window, as all the "wanted" samples from the signal are multiplied by 1 and all other samples by 0, effectively

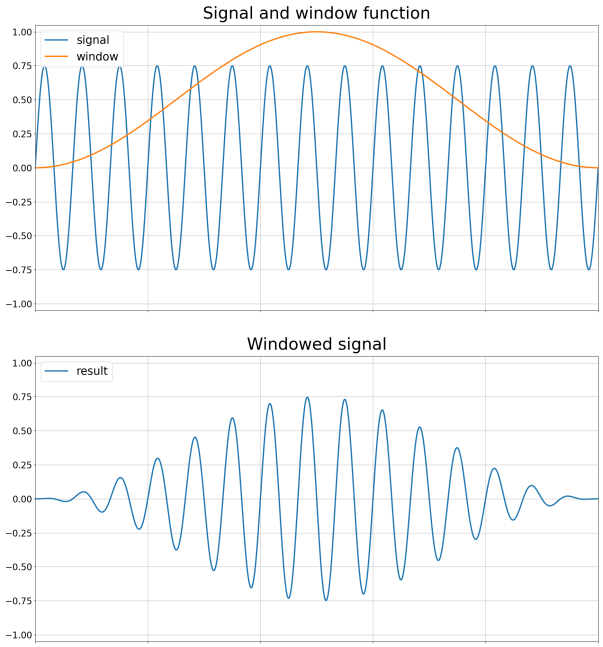

Figure 2: An example of applying a window function to a signal.

framing the signal. We won't discuss individual window function in this section; see Appendix B for an in depth discussion on the different window functions.

The leakage behavior of a window function can be quantified by performing a Fourier transform on the window function. The resulting frequency domain shows the amount of leakage in neighboring bins compared to the power of the center bin. To show the leakage behavior of frequencies which do not align with a bin, we can zero-pad the window function, which causes over-sampling in the frequency domain. Zero-padding is further discussed in Section 4.4. Figure 3 shows the leakage of a rectangular window when a frequency exactly matches the center frequency of a bin and the leakage when a frequency is exactly between bins. Here we can see the leakage spectrum has lobes of leakage. The lobe containing the center bin is called the main lobe and the other lobes are called side lobes. The different window functions trade-off between having a narrow main lobe width and low side lobe levels [18].

The performance of window functions is often described using equivalent noise bandwidth, coherent power gain and scalloping loss [19, 27, 34]. Equivalent noise bandwidth signifies the variation in noise floor compared to using a rectangular window. In other words, when transforming a signal, all the noise in the signal will result in some power in every bin, creating a floor in the spectrum. Window functions raise this noise floor by a consistent amount, which is quantified using equivalent noise bandwidth. Applying a window function causes the overall amplitude of a frame to decrease, which in turn decreases the power of bins. This reduction is called the coherent power gain and signifies the loss of power in the spectrum. As mentioned earlier, when a signal does not fit a frame, it leaks into other bins in the spectrum. Scalloping loss is the amount of decibel lost in the bin containing the main lobe when transforming a frame containing a signal halfway between bins compared to a signal right on a bin. In order to accurately describe the signal power, we would need to correct for these effects. Equivalent noise bandwidth and coherent power gain can be described by a fixed value for each window function, but scalloping loss depends, along with the sample rate and frame size, on the spectral content of the signal, which we have no control over. On top of this, there are several other problems with accurate signal power estimation. Since absolute signal power estimations are not required for the content of this thesis, we will simply rely on relative power estimations.

3.3 Real-time

Real-time is a difficult concept, as it has multiple definitions based on what field of research it is used in. As the formal definitions relate to very different concepts, it usually does not cause any problems. Problems arise when the term is informally used, as the vernacular definitions often miss an important aspect of the formal definitions of real-time, which causes statements made about systems which adhere to these vernacular definitions to be unreliable or useless. Here are a few examples of different definitions (vernacular definitions are marked red):

1. Being synced with actual clock time (or wall time). This is for instance relevant when playing media such as audio and video. When such media is played at an incorrect speed, it could be considered distorted. The hardware which keeps track of the clock time is called a real-time clock.

2. A system must response within a specified time constraint, which is called the real-time constraint or deadline. This constraint is usually a relatively short time. This definition comes from real-time computing and is relevant when making car airbags or airplane control systems. Failing to response within the real-time constraint leads to failures of the system. Real-time systems are often classified into hard, firm and soft

real-time based on the impact of missing the deadline [22].

3. A system which can provide a result or feedback with no noticeable delay after receiving some input. Examples of such systems include graphical user interfaces or instant chatting/calling. There are no hard deadlines which the system has to respond within and the system does not fail if some delay does occur. Only user experience is slightly impacted. In the field of real-time computing, this is often referred to as near real-time.

4. A system which can process data faster than it acquires data. This is technically not real-time, however, it is often used as such in academic literature. It is important for real-time systems to process data faster than it acquires data so it does not lag behind after some time, however, this is an implicit deadline. Not having this deadline explicit may lead to non-sensible expectations of the system.

Even though the first definition is very relevant when working with audio, it is not relevant for us. The audio drivers of operating systems handle all timing for us. We simply have to wait for samples to be recorded and made available to our program. We only have to keep the sampling rate in mind when working with the samples as shown in the previous section.

In order to allow guitarists to use their guitar as a MIDI instrument, our system has to respond within a small time frame. On top of that, if the system fails to respond quickly enough, the usefulness of the result degrades, as timing is very important when playing an instrument. These restrictions would classify our system as a soft real-time system. We choose a real-time of constraint of TODO milliseconds. We elaborate on this choice in Section 4.1.

Other work in real-time pitch estimation often uses the forth definition of real-time. This is problematic when using Fourier transform based methods, as

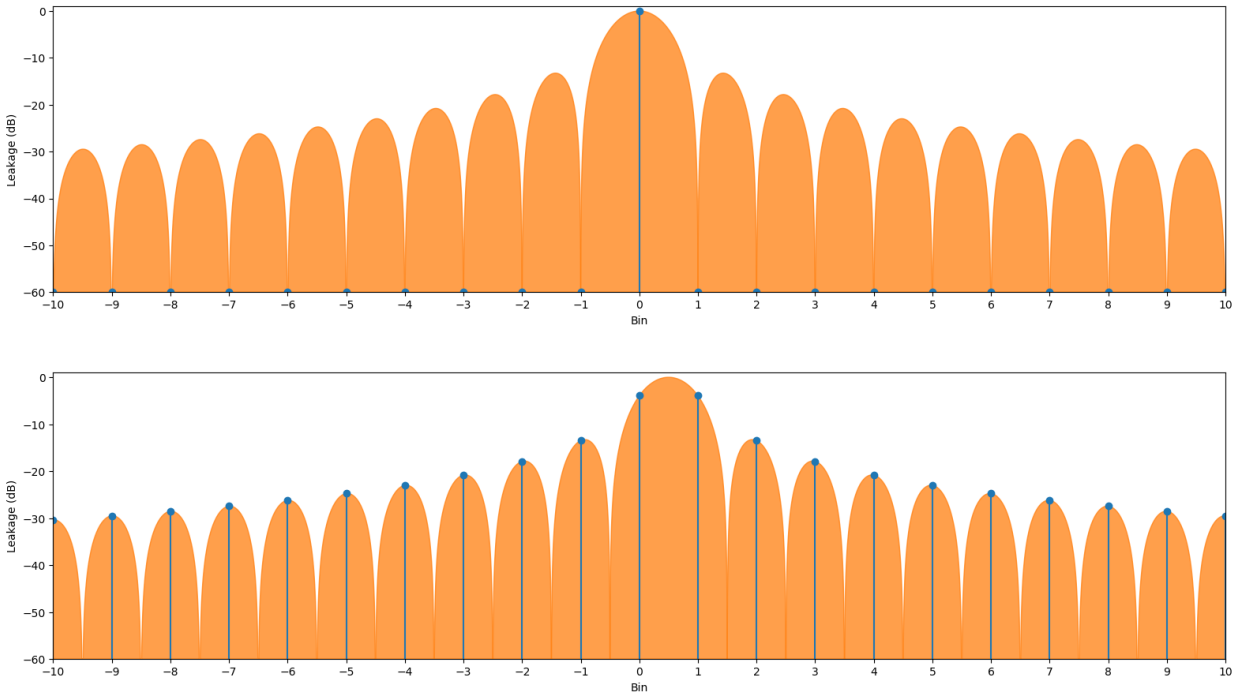

Figure 3: TODO: Better image. Leakage behavior of the rectangular window. The orange lobes represent the frequency response of the rectangular window function; the blue lines represent the signal power measured in each bin given a signal frequency at the center of the main lobe of the frequency response of the window function. In the top plot, the frequency of the signal exactly matches the center frequency of a bin. If we then look at the leakage at every bin position, we see that all signal power goes into the center bin and nothing into any other bin. In the bottom plot, we shifted the frequency response half a bin up to simulate a frequency in the signal half a bin higher than some arbitrary bin. If we now look at the height of the frequency response at every bin position, we can see that the signal power diminishingly leaks into all subsequent bins.

many papers choose large frame sizes to get a high resolution in the frequency domain. For instance, in order to discern the two lowest notes on a guitar which are 4.9 Hz apart, we would need a frame length of 204 milliseconds. This implicit deadline is well over our real-time constraint and would be unplayable for any musician. Other papers we found which do explicitly set a real-time constraint, choose very high constraints from 140 ms [17] up to 360 ms [14]. These constraints were likely chosen with the inherent limits of their solutions in mind. It is very important to set a real-time constraint solely based on the expectation of the systems from an outside perspective. Real-time constraints chosen with the inner working of the system in mind are merely a measure of performance that is hoped to be achieved and claiming a system is real-time based on such constrains is considered fraudulent. We have found no papers Fourier transform based pitch estimation papers which choose a sensible real-time constraint.

3.4 Music theory and notation

In modern western music, we use the twelve-tone equal temperament (12-TET) music system. This system divides an octave, which is the interval between a pitch and another pitch with double the frequency, into twelve equally spaced semitones on the logarithmic scale. The logarithmic scale is used such that the perceived interval between two adjacent notes is constant [26]. As a result, the ratio between two frequencies in an n-semitone interval is $\sqrt[12]{2}^n$ or $2^{\frac{n}{12}}$, invariant to pitch. A semitone can be further divided into 100 logarithmically scaled cents.

Using scientific pitch notation, every note can be uniquely identified by combining the traditional note names A to G (with accidentals such as ♯ and ♭) with an octave number (e.g. $\mathrm{E}_3^\flat$). An octave starts at C, which means the octave number increases between B and C. This counter intuitively implies A_3 is higher than C_3. Note that in 12-TET, $\mathrm{C}^\sharp$ and $\mathrm{D}^\flat$ are enharmonically equivalent. In this thesis, we will always refer to the sharp (♯) note instead of the enharmonically equivalent flat note (♭). The range of a typical electric guitar in standard tuning is from E_2 up to E_6.

The 12-TET music system only describes the relation between two notes in an interval. In order to play with other musicians in harmony, an arbitrary note has to be tuned to a specific frequency. Per ISO 16, the standard tuning frequency of the A_4 is 440 Hz within an accuracy of 0.5 Hz [4]. In this thesis, we will always assume a 12-TET music system with a 440 Hz tuning note.

Using the above information, we can translate frequencies into scientific note names and vice versa. In order to numerically work with note names, we assign a value to each note as shown in Table 1. In order to make calculations easier, we use C_0 as a tuning note instead of A_4. We can calculate the frequency of C_0 using the fact that C_0 is 57 semitones lower than A_4:

name	number	name	number
C	0	$\mathrm{F}^\sharp$	6
$\mathrm{C}^\sharp$	1	G	7
D	2	$\mathrm{G}^\sharp$	8
$\mathrm{D}^\sharp$	3	A	9
E	4	$\mathrm{A}^\sharp$	10
F	5	B	11

Table 1: The number corresponding to each note name

$$f_{C_0} = f_{A_4} * 2^{\frac{-57}{12}} = 440 * 2^{\frac{-57}{12}} \approx 16.352 \text{ Hz}$$

We can calculate the frequency f_{N_O}, where N is the note name which is represented by a numerical value given by Table 1 and O is the octave number using:

$$f_{N_O} = f_{C_0} * 2^O * 2^{\frac{N}{12}} = f_{C_0} * 2^{O+\frac{N}{12}}$$

To calculate the closest note N_O corresponding to a frequency f, we first calculate the number of semitones n_s between the tuning note f_{C_0} and f:

$$n_s = \left\rfloor 12 * {}^2\log \frac{f}{f_{C_0}} \right\rceil$$

Here, $\lfloor \ldots \rceil$ denotes rounding to the nearest integer. By rounding, we find the closest note to f. Now we can calculate N and O as follows:

$$N = n_s \bmod 12$$
$$O = \left\lfloor \frac{n_s}{12} \right\rfloor$$

Note that we assume $a \bmod b$ always return a number c for which $0 \leq c < b$. Some programming languages allow the modulo operator to return a value c for which $-b < c < b$, resulting in $-13 \bmod 10 = -3$ instead of $-13 \bmod 10 = 7$. Furthermore, when using a conversion to an integer instead of a floor, the

octave number is rounded up when the note distance is negative.

In order to calculate the error e (in cents) between the given frequency to the closest tuned note, we first calculate the tuned frequency f_t of the closest note:

$$f_t = f_{C_0} * 2^{\frac{n_s}{12}}$$

Then the error e can be calculated using:

$$e = 1200 * {}^2\log \frac{f}{f_t}$$

In digital music processing, notes are often represented through MIDI note numbers, as it allows programmers to easily refer to notes using integer values. The MIDI standard defines MIDI note number 69 to be the standard tuning frequency A_4. Every semitone up/down respectively increases/decreases the MIDI note number by 1. This makes our tuning note C_0 number 12. According to the MIDI specification note numbers can take a value from 0 to 127, however, this is only relevant when communicating with MIDI devices. The following equations are valid for any note/MIDI note number.

The MIDI note number m corresponding to the note closest to frequency f can be calculated using the semitone distance from a frequency with a known MIDI note number. Let $m(N_O)$ denote the MIDI note number of N_O:

$$m = \left\lfloor 12 * {}^2\log \frac{f}{f_{N_O}} \right\rceil + m(N_O)$$

Conversely, the frequency f of the note corresponding to MIDI number m can be calculated as follows:

$$f = f_{N_O} * 2^{(m - m(N_O))/12}$$

3.5 Physical properties of sound

The perceived loudness of a note over time can be described using an ADSR envelope. The ADSR envelope of a played note is the convex hull of the waveform of the signal, see Figure 4 for an example. This convex hull can be divided into four parts: Attack, Decay, Sustain and Release. When a note is strummed on the guitar, a percussive sound is generated which causes a loud and sharp attack along with the note. This percussive sound quickly decays and only the actually fretted note will sustain. Finally, when the note is released, it dies out quickly.

The percussive sound generated when strumming a note is called a transient. Transients contain a high degree of non-periodic components. Because of this, transients appear very chaotically in the frequency domain and are often considered noise. As

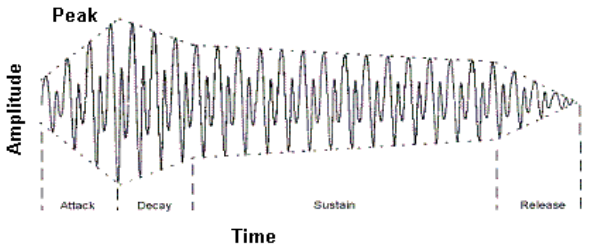

Figure 4: Example of an ADSR envelope (TODO: Better image).

a transient is of high amplitude, it overshadows the note which will eventually sustain. Consequently, we cannot use the samples from a transient for Fourier based pitch estimation. This in turn increases our minimum latency, as we have to wait for samples which do not contain the transient anymore.

When playing a note on an instrument, many sine waves are generated. The most notable frequency is called the fundamental frequency and determines what note is actually played. Integer multiples of the fundamental frequency can resonate and give rise to harmonic overtones [30]. In practice, these overtones are not exact integer multiples due to non-linear effects.

Many other frequencies are generated along with the fundamental and its overtones. The instrument specific pattern of these frequencies, along with the characteristics of the overtones, is called the timbre of the instrument [20]. The timbre is what differentiates the sound of the same note played on two different instruments [26]. Generally, the amplitude of the timbre frequencies is low compared to the fundamental frequency and can be disregarded as noise in the frequency domain. Figure 5 shows the effect of timbre on a waveform.

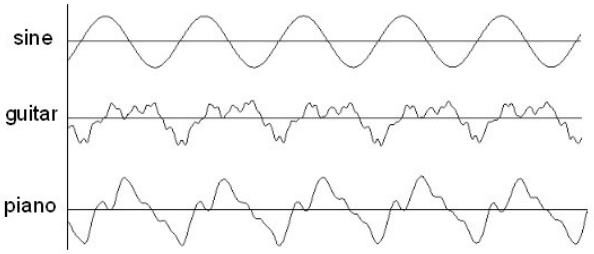

Figure 5: Example of difference in timbre between instruments compared to a sine wave.

In Section 2, we mentioned overtones are dissonant with respect to notes in 12-TET. This is true for all overtones, except for octaves, which are all overtones numbers equal to $2^n - 1$ for every n. In Table 2, we

show an example for the overtones of C_4. Note that the series of errors is always the same, regardless of what the starting note is.

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 2: Example of overtone series from C_4 and the errors compared to the closest note

TODO: Rewrite. As mentioned in Section 2, when using the CQT transform, none of the non-octave overtones coincides with a CQT bin as the CQT bins are exponentially spaced like the notes in a scale. This causes the frequencies of the overtones to spread out over bins, resulting in more noise in the frequency domain. Furthermore, overtones are important for discerning fundamentals from noise generated by transients. By using a Fourier transform tuned to a specific note, all its overtones are also coincide with Fourier bins. By performing a Fourier tuned to every note in the 12 tone scale, we can measure every note and its overtones.

TODO: This thesis mainly focuses on monophonic pitch estimation, as it is much easier to perform. But we can still see how well our strategy works in the polyphonic case. TODO: A big problem in monophonic pitch estimation is the octave problem. Octaves are difficult to discern as the fundamental frequency and overtones of the higher note coincide with the overtones of the lower note. A similar problem arises TODO: The main problem in polyphonic pitch estimation comes from the occurrence of overtones. As mentioned before, notes in octave tend overshadow each other. Furthermore, as shown in Table 3, the overtones of a note can coincide with the... TODO: Overtone overlap and polyphonic difficulty.

n	$f_0^{C_3} * n$	n	$f_0^{E_3} * n$	n	$f_0^{G_3} * n$
1	130.813	1	164.814	1	195.998
2	261.626	2	329.628	2	391.995
3	392.438	3	494.441	3	587.993
4	523.251	4	659.255	4	783.991
5	654.064	5	824.069	5	979.989

Table 3: Overtones of C, E and G

Note	f	Δf
$f_2^{C_3}$	392.438	0.443
$f_1^{G_3}$	391.995	
$f_4^{C_3}$	654.064	5.191
$f_3^{E_3}$	659.255	

Table 4: Colliding overtones

4 Real-time Fourier transform based monophonic pitch estimation

At the heart of our research lies a pitch estimation algorithm. The task of a pitch estimation algorithm is to convert a frame of samples to some representation of the notes contained in the frame. In this thesis, we will focus on spectral analysis methods of pitch estimation. Specifically, we focus on pitch estimation using the spectra obtained from frames using the Fourier transform.

4.1 Real-time constraint

Before we start constructing our pitch estimation system, let us first choose a real-time constraint. The goal of our pitch estimation system is creating a digital representation of the played notes so it can for example be used for software sound synthesis. Consequently, the real-time constraint should reflect the critical latency. This is the latency for which delay between playing a note and receiving the feedback, such as hearing back the synthesized note, becomes problematic. The critical latency is highly subjective and may even differ between playing styles. Note that our real-time constraint only considers pitch estimation latency. If we want to do anything with the pitch estimation results in real-time, such as synthesizing audio, additional latency will be introduced. For this reason, we shouldn't put our real-time constraint right at the critical latency, but leave some headroom for the program using our estimated pitches.

Even though the critical latency is subjective, it is the factor determining if a real-time pitch estimation system is usable in musical context. In order to determine the critical latency empirically, we created a tool called *delayed playback*, which plays back recorded audio with an arbitrary latency. Using *delayed playback*, we found a latency of TODO milliseconds is a reasonable latency at which we can still comfortably play many songs. *Delayed playback* can also be used to verify if a real-time constraint

chosen by someone else is reasonable by your own standards.

The critical latency depends, among other things, on the speed with which notes are played. For instance, let's assume a latency of 50 milliseconds. When playing a slow song consisting of quarter notes at 120 BPM, resulting in 2 notes per second, the latency is only a tenth of the note duration. However, when playing a fast solo consisting of sixteenth notes at 210 BPM, resulting in 12 notes per second, the latency is 70% of the note duration. This means you hear more of the previous note than the current note for the duration with which the current note is played, which makes playing at fast speeds extra difficult.

There is a limit on how fast we can estimate the pitch of a played note. For instance, when we consider transients noise, we have to wait for the transient to pass before we can gather samples containing the note. Then, we need enough samples such that the lowest bin of the DFT doesn't exceed the frequency of the lowest note. In the case of E_2, which is 82.407 Hz, we would need a minimum frame length of $\frac{1}{82.407} * 1000 = 12.135$ milliseconds. Due to spectral leakage, using such small frame lengths is not feasible. However, as outlined in Section 8.1, we can translate this idea to the time domain. In practice, this would result in a minimum latency of 14 to 15 milliseconds.

In order to assess what we can reasonably expect from our system, we measured the latency of a commercial guitar synthesizer solution. This is described in depth in Appendix A. Here, we found the Axon AX 100 MKII at worst has a latency of 15 milliseconds when it is able to guess the note "the first try". Otherwise, the latency may reach up to 40 milliseconds.

TODO: Note on unaccounted latencies (audio driver/hardware latencies). A careful reader may have noticed we actually discussed two different latencies. We mostly reason about the latency of our pitch estimation system. However, when reasoning about the critical latency, latencies from the audio driver and hardware as discussed in Section 3.1 do matter. TODO: Estimate on these latencies. Estimate of latency when running Digistring's algorithms on specialized hardware.

TODO: Our real-time constraint choice.

4.2 Basic algorithm for pitch estimation

Let us start with constructing the most basic Fourier based monophonic pitch estimation algorithm. Given a frame of samples and the sample rate, the algorithm will return the MIDI number of the note closest to the most prominent frequency in the frame. First, we apply a window function to the frame and apply the Fourier transform. Then, we iterate over every output bin and check which bin has the highest magnitude. From the bin number, we can calculate the corresponding frequency and subsequently determine the note closest to this frequency. See Algorithm 1 for a pseudo-code implementation.

To minimize our latency, we want to choose our frame size as small as possible while still being able to discern the two closest notes in frequency a typical guitar can produce. Due to the exponential nature of notes, the lowest two notes are always the closest two. For a typical guitar, these notes are E_2 and F_2, which have a corresponding frequency of 82.407 Hz and 87.307 Hz respectively. This means we need a frequency resolution of at least $87.307 - 82.407 = 4.9$ Hz. This equates to a frame length of $4.9^{-1} = 0.204$ seconds, or 204 milliseconds. The bin centers do not have to exactly align with the notes to be able to discern them, so in practice we could get away with slightly shorter frames. Still, such large frame lengths are problematic for multiple reasons. A frame will contain multiple played notes when a guitarist plays at a moderate tempo (playing eighth notes in 150 BPM corresponds to 200 millisecond notes). Furthermore, if a frame contains the start of note, the whole frame might be useless due to the transient. Lastly, since we have to wait until the audio driver has enough samples ready for us to fill a frame, the first samples from that frame will be 200 milliseconds old. The average latency of processing a sample is 100 milliseconds due to the frame length alone. This is well over our real-time constraint.

We implemented this basic Fourier pitch estimation algorithm in Digistring (`BasicFourierEstimator`). On average, the pitch estimation is performed in 0.13 milliseconds. This means our limiting factor is the long frame lengths, which implies we should look for methods which allow for accurate pitch estimation with lower frequency resolution. Furthermore, especially on low pitched notes, it often picks the note one octave higher than the fundamental frequency. Apart from the octave errors, the pitch estimator does often produce correct results, however, it is completely infeasible for real-time usage due to the low frame rate. As we only produce one note estimate for every frame, we essentially quantize our estimator to eight notes on 150 BPM. Lastly, our basic estimator has no notion of silence. If no note is played in the frame, we essentially perform pitch estimation on white noise, giving

us practically random note guesses.

4.3 Overlapping frames

As seen in the previous section, a low frame rate is problematic, as our note output rate is synced to the frame processing rate, essentially quantizing our note estimation to the frame processing rate. The straightforward solution would be to decrease the frame length, however, the frame length is limited by the minimum frequency resolution we need. Instead, we can increase the frame rate by partially overlapping subsequent frames.

Overlapping frames has several advantages. Apart from the increased frame rate, which in turn decreases quantization error, it may decrease the average latency of processing a sample. Furthermore, as transients are very short in the time domain, having more frames over a certain period of time decreases the relative number of frames containing the transient, which causes more frames to have a useful note estimate, see Figure 6 for a graphical explanation.

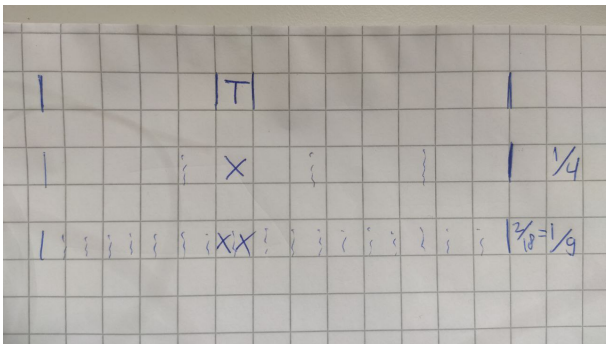

Figure 6: TODO: Correct image showing overlap instead of just shorter frames. Given the signal given on top with a transient at T, relatively less frame estimations are ruined by the transient when using a higher frame rate.

As two subsequent frames share information when overlapping, the estimation from subsequent frames is correlated. This limits the usefulness of overlapping beyond a certain overlap ratio [15]. However, as long as we don't overlap so much that samples are gathered faster than we can process them, overlapping doesn't incur an addition latency on the pitch estimation system.

When performing real-time estimation, instead of overlapping a constant ratio between frames, we can overlap based on the rate at which the audio driver acquires samples and our previous frame processing time. Time spend waiting for samples is effectively wasted time, as that time could've been spent on generating note estimations. Instead, we can always retrieve all currently available samples from the audio driver and fill the rest of the frame with samples from the previous frame. This ensures maximum possible overlap every frame.

4.4 Zero padding

The number of output bins is determined by the number of samples that is transformed. We can artificially increase the number of samples in a transform by appending zeros to the frame. This technique is called zero padding. As only silence is added to the frame, it does not alter the spectrum.

As mentioned in Section 3.2, the frequency resolution of a DFT is $\Delta f_{bin} = \frac{f_{SR}}{n_F}$. Let n_{F_P} be the zero padded frame size. Given that the sample rate is constant, our frequency resolution will increase by a factor of $\frac{n_{F_P}}{n_F}$. In other words, if we zero pad the frame such that it becomes x times larger, our frequency resolution will increase by a factor x.

It is important to note that zero padding does not increase the resolution of the DFT, as no extra information about the original signal is added to the frame. It merely interpolates the coarse spectrum to become more smooth [3]. Two frequencies closer than Δf_{bin} together form one big lobe in the smoothed spectrum [2]. However, the interpolated peaks may have a higher amplitude than the original peaks, thus improving the results of our basic pitch estimation algorithm. See Figure 7 for a graphical example of this.

Zero padding is relatively compute intensive form of interpolation [35]. TODO: For a length-N signal, the computational complexity of the Fast Fourier Transform (FFT) algorithm for calculating the DFT is O(N log N). So, zero padding a signal by a factor of D up to a length N D raises the cost by a factor D(logN D + 1). Zero padding by a factor D increases the cost of the algorithm by a factor that is greater than D—this is very expensive.

4.5 Quadratic interpolation

A less compute intensive method of interpolation is quadratic interpolation, also know as Quadratically Interpolated FFT (QIFFT) [35]. Given a peak bin and its neighbors, QIFFT interpolates the actual peak location by fitting a parabola through these three points. The vertex of the fitted parabola is the interpolated peak location.

The accuracy of interpolation can be improved by performing the interpolation on a logarithmically weighted magnitude spectrum (LQIFFT). In other

input : Frame F and sample rate f_{SR} in Hz
output : MIDI number of estimated note

1 $F_w \leftarrow$ `apply_window_function(`F`)`;
2 $S \leftarrow$ `fourier_transform(`F_w`)`;
3 max_index $\leftarrow 1$;
4 **for** $i \leftarrow 2$ **to** $\left\lfloor \frac{|F|}{2} + 1 \right\rfloor$ **do**
5 | **if** $|S[i]| > |S[\text{max\_index}]|$ **then**
6 | | max_index $\leftarrow i$;
7 | **end**
8 **end**
9 `// Most prominent frequency in frame (assumes arrays start at 0)`
10 $f_e \leftarrow \text{max\_index} * \frac{f_{\mathsf{SR}}}{|F|}$;
11 **return** $\left\lfloor 12 * {}^2\log \frac{f_e}{440} \right\rceil + 69$;

Algorithm 1: Basic Fourier based pitch estimation algorithm. Note that this algorithm assumes arrays to start at 0. The for-loop on line 4 loops from 2, as the first bin (index 0) corresponds to the DC offset and is not relevant. Then, we set the current found max to the second bin (index 1) and check if any subsequent bin is higher. We loop till $\left\lfloor \frac{|F|}{2} + 1 \right\rfloor$, as the second half of the Fourier transform is symmetric due to the purely real input data and can be discarded.

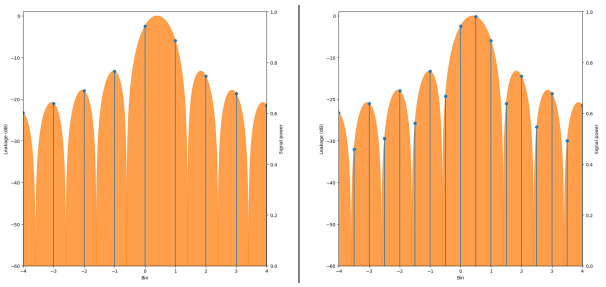

Figure 7: TODO: Better image. Left and right figure show a DFT spectrum without and with zero padding respectively. Zeros are padded such that $n_{F_P} = 2{*}n_F$ holds. The blue lines represent the bin magnitudes. The orange lobes represent the frequency content of the frame if infinite zero padding was applied.

words, the error of interpolation is reduced if the logarithm of the bin power is used. Since a Gaussian curve is a parabola on a logarithmic scale, the interpolation is nearly perfect when using a Gaussian window function on a unweighted spectrum [16]. It isn't perfect, as a true Gaussian window would need infinite long tails [31].

Using an exponentially weighted magnitude spectrum (XQIFFT) may further reduce the interpolation error [35]. However, this requires carefully choosing the exponent with which to weight the bins, as some choices may increase error. As outlined in Section 5.5, we can iteratively approximate the optimal exponent. Note that the optimal value is specific to the exact pitch estimation strategy up to peak interpolation. For example, if the frame size, zero-padding factor or window function changes, the optimal exponent has to be approximated again. In Section 6.1, we empirically derive the errors for the different interpolation methods. The exponent factor cannot be zero.

To perform quadratic interpolation, we calculate a value $p \in [-\frac{1}{2}, \frac{1}{2}]$, which is the offset in bins of the interpolated peak with respect to the peak bin b_j at index j. Using the magnitude of the peak $|b_j|$ and the magnitude of the neighboring bins $|b_{j-1}|$ and $|b_{j+1}|$, we define:

$$\alpha = w(|b_{j-1}|)$$
$$\beta = w(|b_j|)$$
$$\gamma = w(|b_{j+1}|)$$

Here, $w(x)$ is an arbitrary weighting function. In the case of LQIFFT:

$$w(x) = \ln x$$

Or in the case of XQIFFT with exponent ϵ:

$$w(x) = x^\epsilon$$

Then, we can calculate p as follows:

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

The weighted amplitude a_i^w corresponding to the interpolated peak is:

$$a_i^w = \beta - \frac{(\alpha - \gamma) * p}{4}$$

The non-weighted interpolated amplitude is:

$$a_i = w^{-1}(a_i^w)$$

Here, $w^{-1}(x)$ is the inverse of $w(x)$. In the case of LQIFFT:

$$w^{-1}(x) = e^x$$

Or in the case of XQIFFT with exponent ϵ:

$$w^{-1}(x) = x^{\frac{1}{\epsilon}}$$

Given the bin number j of the spectral peak location, the frequency f_i corresponding to the interpolated peak is:

$$f_i = \Delta f_{bin} * (j + p)$$

The Lagrange polynomial describing the interpolation parabola is defined as follows:

$$L(x) = \sum_{n=-1}^{1} w(|b_{j+n}|) \left(\prod_{\substack{m=-1 \\ m \neq n}}^{1} \frac{x - (j+m)}{(j+n) - (j+m)} \right)$$

$$\begin{aligned} &= \alpha * \frac{x-j}{(j-1)-j} * \frac{x-(j+1)}{(j-1)-(j+1)} \\ &\quad + \beta * \frac{x-(j-1)}{j-(j-1)} * \frac{x-(j+1)}{j-(j+1)} \\ &\quad + \gamma * \frac{x-(j-1)}{(j+1)-(j-1)} * \frac{x-j}{(j+1)-j} \end{aligned}$$

$$\begin{aligned} &= \alpha * \frac{1}{2}(x^2 - x - 2jx + j + j^2) \\ &\quad + \beta * (-x^2 + 1 + 2jx - j^2) \\ &\quad + \gamma * \frac{1}{2}(x^2 + x - 2jx - j + j^2) \end{aligned}$$

One often overlooked detail is that the three interpolation points need to be in one spectral lobe, as we approximate the shape of the lobe with a quadratic function. Most window functions have a wide enough main lobe width, causing the three points to always be within the main lobe. In the case of the rectangular window, there are at most two bins within the main lobe. By zero padding, we can increase the number of bins in the spectrum and in turn get more than two bins in the main lobe, see Figure 7. In general, zero-padding decreases the error of quadratic interpolation [35].

4.6 Peak picking

As explained in Section 3.5, overtones are generated along with the fundamental frequency when playing a note on an instrument. On the guitar, the first overtone is often measured as louder than the fundamental frequency, causing our basic pitch algorithm to report the first overtone instead of the fundamental frequency. This is referred to as the octave problem, as the first overtone is an octave from the fundamental frequency.

Instead of looking for the bin with the highest magnitude, we can try to identify all spectral peaks. Then, using these peaks, we can make a better estimate on what note is in played in the frame.

Let us start with the most basic peak picker. Here, we simply return each bin which has two neighboring bins with a lower magnitude. This method does correctly identify all significant spectral peaks, but also finds many irrelevant peaks; especially in noisy areas of the spectrum, as there are many local maxima in random noise. For now, we can eliminate the peaks in noise by requiring a minimum peak power, but if the minimum value is not chosen carefully, the sustain of a note may be cut short. We need some method to select which peaks are significant.

One way to determine if a peak is significant enough is Gaussian average envelope based peak picking [11]. Here, we first calculate a Gaussian weighted average envelope of the spectrum. Only peaks higher than the envelope can be deemed significant. One envelope point is calculated for every spectral bin. For each bin, we calculate the weighted average of all bins, where a bin's weight is determined by number of bins between a bin and the considered bin. Given this distance n, the weight is calculated using a Gaussian function:

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

Here, the parameter σ determines the relative weight of close bins compared to distant bins. Higher values for σ causes nearby bins to have a higher weighting.

TODO: Min-max peak picking. Peak filtering.

4.7 Note selection from peaks

Due to overtones and errors in peak picking, not every picked peak corresponds to a note in the signal. Therefor, we need an algorithm which can determine which note is likely played given a set of significant peaks.

In our basic pitch estimation algorithm, we implicitly assume the loudest peak as the fundamental frequency of the note contained in the frame. However,

in practice, the first overtone may be louder than the fundamental frequency, causing octave errors. We could select the lowest peak as the fundamental frequency, however, this method is susceptible to low frequency noise.

Instead of selecting a single peak, we can look for a group of peaks which form a valid series of overtones, as played notes always come with overtones. As mentioned in Section 3.5, overtones are not exact integer multiples of the fundamental frequency, so we need to set a threshold on the maximum allowed difference. Because notes are separated exponentially in frequency, our threshold should scale logarithmically. This is achieved by using cents as a measure of error. Then, we can either count the number of overtones for every peak and return the peak with the most overtones, or add the heights of the overtones up and return the peak with the highest overtone power. See Algorithm 2 for a pseudo-code implementation.

Note filtering (only allow note selection in range of instrument).

4.8 High resolution estimator

Base algorithm (apply Fourier, calc norms, pick peaks, find notes in peaks, ...). See Algorithm 3 for an overview.

All the different parameters which have to be configured for good results.

4.9 Tuned Fourier transforms

12 Fouriers for each note and it's overtones.

5 Digistring

Pitch estimation research is not accessible, especially the real-time case. It is not necessarily difficult to implement some pitch estimation algorithm, however, such an implementation would essentially be a black box, as the resulting note estimation does not provide any information on how this estimate came to be. As outlined in Section 4.8 and throughout this section, our pitch estimation algorithm has multiple stages all with multiple parameters which effect the performance of subsequent stages. Since we have extremely limited resolution in the frequency domain in order to minimize latency, it is very important all these parameters can be tuned very carefully. This requires extensive feedback from the estimation process. Furthermore, performing automated experiments is such a hassle, that most other work only includes some informal tests to verify the performance of their pitch estimation systems.

To make pitch estimation research more accessible, we set out to create a pitch estimation framework called Digistring. This framework includes efficient implementations for all concepts described in this thesis, extensive graphical and auditory feedback of the estimation process, the ability to read samples from the audio driver or a file, tools for automated testing/performance measuring and an abstraction for pitch estimators such that all these features work with any arbitrary pitch estimation algorithm. Even though this thesis focuses on monophonic pitch estimation, Digistring supports both monophonic and polyphonic pitch estimation.

Digistring is implemented in C++, as it is a high performance language suited for real-time systems. It uses FFTW3 for efficient Fourier transforms and SDL2 for graphics and audio. Digistring is available at `www.github.com/lucmans/digistring`.

5.1 Digistring overview

Digistring is divided into three main components, shown graphically in Figure 8. The first is the sample getter, which as the name implies, retrieves samples from a sample source. The samples are then given to the pitch estimator, which transform the frame of samples to a note event list. A note events consists of a note and onset/offset information relative to the frame start. Lastly, the constructed note event list can be used by the different output modules. For instance, we can generate a JSON formatted file representing all the note events, which can in turn be used for automated testing. Digistring also features a synthesizer module, allowing us to verify if the results are, musically speaking, satisfactory. Certain small errors may completely negate the usefulness of any application of the pitch estimation algorithm. Furthermore, some technical errors may actually be of musical value, as was the case for guitar distortion, which is now one of the most widely used guitar effects.

5.2 Estimator

The main task of the estimator abstraction is separating pitch estimation from the rest of Digistring, such that the pitch estimation algorithm can easily be replaced. The goal of a pitch estimation algorithm is to convert a frame of sampled audio data to some representation of the notes contained in the frame. Consequently, the interface for an estimator should be a frame of samples as input and note events as output. A note event consists of a note along with

input : Interpolated peaks i_peaks and overtone error threshold overtone_error
output : Likeliest note

```
if |i_peaks| = 0 then
    return;
else if |i_peaks| = 1 then
    return i_peaks[0];
n_harmonics[|i_peaks|] ← [0, ..., 0];
overtone_power[|i_peaks|] ← [0.0, ..., 0.0];
for i ← 0 to |i_peaks| do
    for j ← i + 1 to |i_peaks| do
        peak_freq ← i_peaks[j].freq;
        harm_overtone_freq ← i_peaks[i].freq * ⌊ i_peaks[j].freq / i_peaks[i].freq ⌉;
        cent_error ← 1200.0 * ²log( peak_freq / harm_overtone_freq );
        if cent_error > -overtone_error ∧ cent_error < overtone_error then
            n_harmonics[i]++;
            overtone_power[i] += i_peaks[j].amp;
        end
    end
end
max_idx ← 0;
for i ← 1 to i_peaks do
    // Or use overtone_power[ ] to find maximum overtone power peak
    if n_harmonics[i] > n_harmonics[max_idx] then
        max_idx ← i;
    end
end
Return i_peaks[max_idx];
```

Algorithm 2: Note selection algorithm which selects the peak with the most overtones

input : Frame frame and sample rate sample_rate in Hz
output : Set of notes contained in the the frame

```
windowed ← apply_window_function(frame);
zero_padded ← zero_pad(windowed);
freq_domain ← fourier_transform(windowed);
spectrum ← calc_norms(freq_domain);
peaks ← pick_peaks(spectrum);
interpolated_peaks ← interpolate_peaks(peaks);
notes ← determine_notes(interpolated_peaks);
return notes;
```

Algorithm 3: Overview of a high resolution estimator.
Here, the input frame is overlapped with the previous frame.

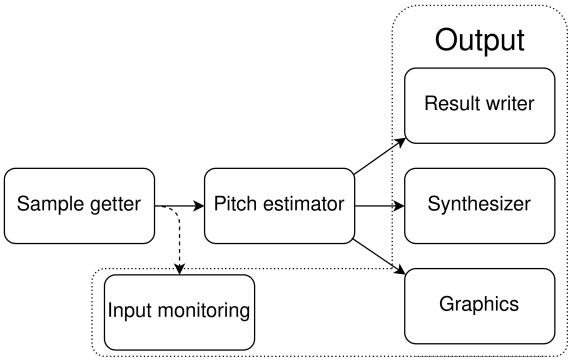

Figure 8: Overview of Digistring.

onset/offset information expressed as number of samples relative to the start of the frame.

Estimators are split up in two phases: the initialization phase and estimation phase. The goal of the initialization phase is alleviating as much work as possible from the estimation phase, which minimizes latency. Examples of initialization phase tasks are precomputing the window function, setting up zero-padding buffers and letting FFTW3 optimize the used FFT algorithms.

When using zero-padding, apply the window function as if the input buffer wasn't zero-padded. In general, any modification to the input signal should never affect the zero-padding. Consequently, the padding only has to be zeroed in initialization phase, as it should never be overwritten.

TODO? As discussed in Section 3.2, we don't perform absolute signal power estimation. Instead, all note amplitudes reported by an estimator are on an arbitrary scale. As a consequence, output modules need to keep track of the maximum reported amplitude to which a specific note's amplitude can be compared to.

5.2.1 HighRes estimator

Details on HighRes estimator implementation.

5.3 Sample getter

As mentioned in Section 3.1, audio samples can be represented in different formats. Floating point samples have many advantages over integer of fixed point samples [32]. Most audio interfaces and audio file formats support 24 bit integer samples or 32 bit floating point samples as the best quality samples. Note that 24 bit integer samples can be converted lossless to 32 bit floating point samples. One could convert to 64 bit floating point samples, which slightly reduces the accumulated floating point rounding errors during digital signal processing. However, the accumulated error is negligible and it would require us to always convert any input samples. Using our tool `float_vs_double`, we found no difference in normal sample processing speed between 32 bit and 64 bit floating point numbers, but this may differ per CPU architecture. When using vector instruction sets such as AVX, the amount or parallelism is limited by the number of bits in the vector registers. As FFTW3 uses these vector instructions, 32 bit floating point number are faster. This can be verified using our `float_vs_double_fftw3` tool. Given these arguments, we decided to use 32 bit floating point samples for all our samples processing in Digistring.

Frame overlapping, as discussed in Section 4.3, is implemented in the sample getter. This way, the pitch estimator doesn't need to know about overlapping. Furthermore, by implementing overlapping in the base class, any newly added sample getter will automatically be able to overlap. There are two different overlapping strategies. The first overlaps two subsequent frames by a constant factor between 0 and 1. The number of samples to overlap is the frame size multiplied by this factor. We clamp the resulting number between 1 and frame size - 1 to assure we always overlap at least one sample or at least get one new sample. The second overlapping strategy is only applicable when reading samples from an audio device. Here, the number of overlapping samples is determined by subtracting the number of samples ready to be read from the audio driver from the frame size. The rest of the frame is filled with samples from the previous frame. To limit the amount of overlap, we allow for a minimum and maximum overlap factor to be set. In live usage, the minimum overlap factor should never be met, as it implies that the estimation system can't keep up with the incoming samples from the audio driver. In Digistring, we refer to this second overlapping strategy as non-blocking overlap, as we try to overlap such that we won't block on retrieving samples from the audio driver.

A simple overlap implementation would save a copy of the entire input buffer, see Figure 9 for a graphical overview of the algorithm. As we know the amount of overlap, or at least an upper bound in the non-blocking case, we can only copy over what we might need next frame. This saved us both time and space, as less data has to be stored and less data has to be copied over. This does however force the sample getter to never request more samples in subsequent "get sample" calls, and in turn, it prevents using a variable overlap strategy. Using our tool

`memcpy_speed`, we measured that the time to copy a big buffer is negligible compared to the processing time of a frame. Because of this, we opted to disable this optimization.

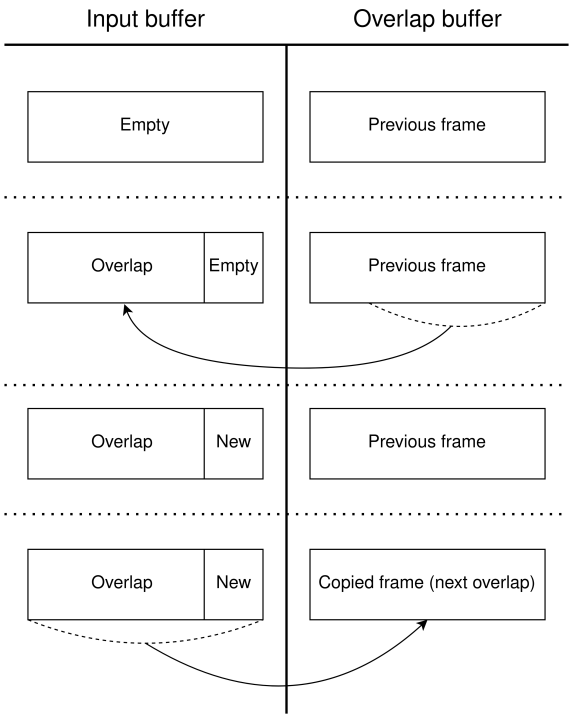

Figure 9: Graphical overview of overlap copy-pasting. The sample getter start with an empty input buffer and the previous frame stored in the overlap buffer. First, we copy the relevant part of the overlap frame to the start of the input buffer. The rest of the input buffer is filled with new samples. Lastly, we copy the entire frame to the overlap buffer for the next get sample call.

Since estimators are agnostic to overlapping, the note events reported by the estimator will overlap if the input frames overlap. Since the output modules assume every given note event list to directly follow the preceding one, we have to adjust the note events to only reflect the newly acquired samples. Moreover, it is possible for an estimator to find a note in the beginning of a frame which wasn't found when the note was at the end of the frame. In other words, it may find a note event completely in the past which wasn't found when the relevant samples were the present. We simply discard these past note events, as past information is irrelevant in real time pitch estimation.

When reading samples from an audio device, there might not be enough samples ready to fill a frame. This might cause partial reads on non-blocking audio APIs, which are solved using a basic read loop. Here, we simply loop the read call until the frame is filled, which causes high CPU usage. However, using the sample rate, we can calculate how long we have to wait until enough samples are ready and sleep for this duration. This reduces the CPU usage of Digistring, which makes Digistring run significantly more energy efficient. However, this does require the operating system to guarantee a timely return from the sleep call, which in Linux is only the case when using the real-time kernel. Furthermore, modern CPUs scale down their performance when they aren't fully utilized and it may take some time before the CPU is back in full performance mode, which causes additional latency after sleeping. As the optimization doesn't improve estimation accuracy or latency, we disabled it in Digistring.

We have implemented four different sample getters in Digistring. The most important one is `audio_in`, which retrieves samples from the audio driver. This sample getter allows us to play live with Digistring. For performing automated experiments and to aid with parameter fine tuning, we implemented `audio_file`. Currently, it only supports reading from `.wav` files. We also implemented `sine_generator` and `note_generator`, which generate sine waves tuned to specific frequencies or notes respectively. These are useful for debugging and testing.

5.4 Real time sound synthesis

To verify if the results of an estimator are musically sound, we implemented a synthesizer interface in Digistring. Given a list of note events, a sample buffer and its size, the synthesizer will fill the buffer with samples such that, when played back, the resulting sound correctly represents the note events. Generating the samples has to be done carefully, as discontinuities in the samples cause audible plops in the resulting audio.

There are two cases where discontinuities may arise when synthesizing sine waves. The first is within a buffer. If we for example want to generate a one hertz sine wave for 1.25 seconds, the wave would end at its highest point and go back to silence the next sample, causing a large discontinuity. We can circumvent this by altering sine wave in such a way that it does reach a zero crossing before stopping. For instance, we can keep generating samples until the waveform reaches a zero crossing, which causes temporal distortion (distortions in time) in the synthesized sound. In the case of sine waves, this quantizes note lengths

to half the frequency, as sine waves have two equally spaced zero crossings. We can also stop the sine wave early if the previous zero crossing is closer than the next. Using this technique, the worst case temporal distortion on a guitar would be 3 milliseconds, as the E_2 has a zero crossing every $\frac{1000}{82.41}/2 \approx 6$ milliseconds. We can also alter the end of the sine wave to force the end to zero, by either amplitude modulating the end of the wave or by adding samples which quickly go back to zero. These method causes spectral distortions (distortions in frequency).

The second case where discontinuities may arise is when a note is not finished at the end of a frame. If the note ended and the next frame contains silence at the start, the signal should go back to 0 as described in the previous paragraph. If the note sustains or another note is played at the start of the next frame, the synthesizer has to continue from this point. Because of this, a synthesizer needs some inter-frame communication.

Our sine synthesizer algorithm starts by zeroing the synth buffer. We first try to zero notes which were not zero at the end of the previous frame. Then, we iterate over every note event and add the generated sine wave samples corresponding to the note event to the synth buffer in the right place. We make sure both ends of the note event samples are at zero by temporally shifting the offset of the note. We only shift the offset, as the onset is more important for conveying rhythm. For every note event that could not reach zero at the end of the synth buffer, we save the note and the phase corresponding to the last generated sample. By saving the phase of the last sample of a note, we can continue at the correct position in the corresponding sine wave in the next frame. This algorithm will always ensure continuity, even if there is polyphony in the note events.

There is one small caveat with this algorithm in the polyphonic case. For instance, if we generate two overlapping sine waves of the same frequency, but one sine temporally shifted half a wavelength, the two waves will cancel each other out. If they are generated a full wavelength apart, they will amplify each other maximally and might cause clipping. Mixing multiple audio channels is out of the scope of this thesis, so we simply ignore this problem. In practice, these cases rarely show up.

Currently, we have implemented TODO synths.

5.5 Optimize XQIFFT exponent

As discussed in Section 4.5, when performing XQIFFT, we need to carefully choose the exponent with which to weight the bins. To test the error of XQIFFT for some exponent ϵ, we generate many different sine waves and perform pitch estimation as we normally would. As we are transforming pure sines, we can simply pick the loudest bin as our estimated pitch. After finding this bin, we can XQIFFT the true peak location and return the found pitch. Since we know the original frequency of the generated sine, we can calculate a measure of error. For instance, we can minimize the mean error over all frequencies or minimize the maximum error. We chose to use the mean squared error for a combination of the two. See Algorithm 4 for an example of such an algorithm. Here, we generate the sines exponentially.

TODO: The error may vary slightly depending on the starting phase of the sine. To circumvent this, we generate multiple sine waves of different phase for every frequency.

As error increases for exponents further away from the optimum, we can iteratively approximate the optimal exponent. See Algorithm 5 for a pseudo-code implementation of such an algorithm. Here, we start our search at some arbitrary range. Then, we calculate the error of some points in this range. These points can be calculated in parallel for an enormous speed-up. If the minimum value is at either end of the range, we have to search further in that direction. We always overlap two points with neighboring ranges, in case the end of the range is the minimum value for that search resolution. Otherwise, we update the search range to the two values surrounding the minimum point and repeat the optimization process.

5.6 Experimentation and estimation feedback tools

TODO: Experimentation is difficult... Many ways to report note estimations and all datasets have different annotation standard (solved using intermediate representation). Automatically calculates commonly used performance measures. Ambiguity of performance measure (all are both good and bad).

JSON output and `generate_report` tool.

Performance (speed) measuring.

Estimator graphics. Estimator can optionally define an Estimator graphics object, which holds information regarding the last pitch estimated frame. This information can be rendered using visualizers. Currently, we implemented four visualizers (add screenshots).

Slowdown mode.

Synthesizer, especially combined with stereo split input monitoring/synthesizer output. MIDI output version of Digistring.

input : Error ϵ and pitch estimation algorithm `pitch_estimation_algorithm()`
reps_per_freq and n_freqs to control precision of error estimation
output : Mean squared error of all tested sines

```
// Return invalid error on invalid exponent
if ϵ = 0 then
    return -1.0;
// Repetitions per frequency
reps_per_freq ← 5;
// Generate list of frequencies to test
n_freqs ← 1200;
frequencies[n_freqs];
for i ← 0 to n_freqs do
    frequencies[i] ← 110.0 * 2^(i/n_freqs);
end
total_squared_error ← 0.0;
foreach f ∈ frequencies do
    last_phase ← 0.0;
    for r ← 0 to reps_per_freq do
        // Generate sine wave
        phase_offset ← last_phase * (sample_rate / f);
        for i ← 0 to |input_buffer| do
            input_buffer[i] ← sin(2.0 * π * (i + phase_offset) * f)/sample_rate;
        end
        last_phase ← last_phase + (f / (sample_rate / |input_buffer|)) mod 1.0;
        detected_freq ← pitch_estimation_algorithm(input_buffer, ϵ);
        hz_error ← detected_freq − f;
        squared_error ← hz_error * hz_error;
        total_squared_error ← total_squared_error + squared_error;
    end
end
mean_squared_error ← total_squared_error / (reps_per_freq * n_freqs);
return mean_squared_error;
```

Algorithm 4: Calculates the mean squared error of XQIFFT given exponent ϵ and a pitch estimation algorithm

input : XQIFFT error estimation algorithm `estimate_xqifft_error(`ϵ`)`
output : Approximation of optimal exponent ϵ

```
1  // Start at some range
2  min_range ← -2.0;
3  max_range ← 2.0;
4  steps ← 8;
5  step_size ← (max_range − min_range) / steps;
6  exponents[steps] ← make_range(min_range, max_range, step_size);
7  while ¬quit do
8  |  for i ← 0 to steps do   // All iterations can be done in parallel
9  |  |  errors[i] ← estimate_xqifft_error(exponents[i]);
10 |  end
11 |  min_idx ← 0;
12 |  for i ← 1 to steps do
13 |  |  if exponents[i] < exponents[min_idx] then
14 |  |  |  min_idx ← i;
15 |  end
16 |  if min_idx = 0 then
17 |  |  // Lowest value is at the start, so optimum is lower than current range
18 |  |  min_range ← min_range − (step_size ∗ (steps − 2));
19 |  |  max_range ← max_range − (step_size ∗ (steps − 2));
20 |  else if min_idx = steps − 1 then
21 |  |  // Lowest value is at the end, so optimum is higher than current range
22 |  |  min_range ← min_range + (step_size ∗ (steps − 2));
23 |  |  max_range ← max_range + (step_size ∗ (steps − 2));
24 |  else
25 |  |  // Optimum is between min_idx − 1 and min_idx + 1
26 |  |  min_range ← exponents[min_idx − 1];
27 |  |  max_range ← exponents[min_idx + 1];
28 |  |  step_size ← (max_range − min_range) / steps;
29 |  end
30 |  exponents[steps] ← make_range(min_range, max_range, step_size);
31 end
32 return exponents[min_idx];
```

Algorithm 5: Iterative algorithm approximating optimal choice of ϵ to be used by XQIFFT

6 Experiments

Note that the parameters were empirically optimized with informal experiments. Datasets. Actual experiments.

6.1 QIFFT errors

Errors of the different interpolation methods.

7 Conclusions

What we did in this thesis. Reflection on the performance of the system. Final reference to the source code.

8 Future work

What could still be improved/further researched.
Overtone dissonance.

8.1 Waveform packets

Low frequency tones from a guitar come in "packets". By detecting the distance between two packets, we can estimate the frequency.

A Measuring the latency of the AXON AX 100 mkII

Lekker meten en weten.

B Effect of different window functions

Plots met effect van verschillende window functions. Rectangular narrowest main lobe and least ENBW. The Hann window is an all round performing window and as a result is often used. The Welch window is a window with a very narrow center lobe. The Dolph–Chebyshev window has little and very evenly spread overall leakage (optimal filter). Parzen window (and b-spline windows per extension) have much space between lobes (lobe density.

References

[1] Digital audio latency explained. `https://www.presonus.com/learn/technical-articles/Digital-Audio-Latency-Explained` Last accessed on 06-04-2022.

[2] Webpage on the limits of zero-padding. `https://dspillustrations.com/pages/posts/misc/spectral-leakage-zero-padding-and-frequency-resolution.html#Frequency-Resolution` Last accessed on 20-10-2021.

[3] Webpage with interactive tool that shows the effect of zero-padding. `https://jackschaedler.github.io/circles-sines-signals/zeropadding.html` Last accessed on 20-10-2021.

[4] Iso 16:1975. acoustics — standard tuning frequency (standard musical pitch). 1975.

[5] IEEE 754-2008 - IEEE standard for floating-point arithmetics. 2008. `https://ieeexplore.ieee.org/document/4610935`.

[6] Eric J. Anderson. Limitations of short-time fourier transforms in polyphonic pitch recognition. Ph.d. qualifying project report, University of Washington, Department of Computer Science and Engineering, 1997.

[7] Eric J. Anderson. Limitations of short-time Fourier transforms in polyphonic pitch recognition. *Technical report, Department of Computer Science and Engineering, University of Washington*, 1997.

[8] Judith Brown and Miller Puckette. An efficient algorithm for the calculation of a constant Q transform. *Journal of the Acoustical Society of America*, 92:2698–2701, 11 1992.

[9] Luc de Jonckheere. Real-time guitar transcription using Fourier transform based methods; a pitch estimation framework and overtone sieve algorithm. 2021.

[10] Robert Dobre and Cristian Negrescu. Automatic music transcription software based on constant Q transform. *8th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–4, 2016.

[11] 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.

[12] Zhiyao Duan, Bryan Pardo, and Changshui Zhang. Multiple fundamental frequency estimation by modeling spectral peaks and non-peak regions. *IEEE Transactions on Audio, Speech, and Language Processing*, 18(8):2121–2133, 2010.

[13] Paolo Nesi Fabrizio Argenti and Gianni Pantaleo. *Automatic Music Transcription: From Monophonic to Polyphonic*, pages 27–46. Springer Berlin Heidelberg, 2011.

[14] Xander Fiss. Real-time software electric guitar audio transcription. Master's thesis, Rochester Institute of Technology, 2011.

[15] A. Rudiger G. Heinzel and R. Schilling. Spectrum and spectral density estimation by the discrete fourier transform (dft), including a comprehensive list of window functions and some new flat-top windows. 2002.

[16] M. Gasior and J. L. Gonzalez. Improving fft frequency measurement resolution by parabolic and gaussian spectrum interpolation. *AIP Conference Proceedings*, 732(1):276–285, 2004.

[17] 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)*, pages 1–6, 2018.

[18] F.J. Harris. On the use of windows for harmonic analysis with the discrete fourier transform. *Proceedings of the IEEE*, 66(1):51–83, 1978.

[19] Wang Hongwei. Evaluation of various window functions using multi-instrument. 2021.

[20] Kristoffer Jensen. Timbre models of musical sounds. 2021. PhD thesis, University of Copenhagen.

[21] 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.

[22] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer US, 1997.

[23] 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.

[24] Michael F. Zbyszyński Matthew Wright, Ryan J. Cassidy. Audio and gesture latency measurements on Linux and OSX. *In Proceedings of the ICMC*, pages 423–429, 2004.

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

[26] Michael J. O'Donnell. Digital sound modeling: Perceptual foundations of sound. `http://people.cs.uchicago.edu/~odonnell/Scholar/Work_in_progress/Digital_Sound_Modelling/lectnotes/node4.html` Last accessed on 04-10-2021.

[27] Stefan Scholl. Exact signal measurements using fft analysis. 2016.

[28] A. Schutz and D. Slock. Periodic signal modeling for the octave problem in music transcription. *in Proceedings of the 16th International Conference on Digital Signal Processing (DSP'09)*, pages 1–6, 2009.

[29] Christian Schörkhuber and Anssi Klapuri. Constant-q transform toolbox for music processing. *Proc. 7th Sound and Music Computing Conf.*, 2010.

[30] 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.

[31] Julius O. Smith. *Physical Audio Signal Processing.* `http://ccrma.stanford.edu/~jos/pasp/`, Last accessed on 18-05-2022. Online book, 2010 edition.

[32] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing.* California Technical Publishing, 1997.

[33] Gino Angelo Velasco, Nicki Holighaus, Monika Doerfler, and Thomas Grill. Constructing an invertible constant-q transform with nonstationary gabor frames. *Proceedings of the 14th International Conference on Digital Audio Effects, DAFx 2011*, 2011.

[34] Mathuranathan Viswanathan. Window function – figure of merits. 2020. `https://www.gaussianwaves.com/2020/09/window-function-figure-of-merits/` Last accessed on 13-05-2022.

[35] Kurt James Werner. The XQIFFT: Increasing the accuracy of quadratic interpolation of spectral peaks via exponential magnitude spectrum weighting. *Proceedings of the International Computer Music Conference*, pages 326–333, 2015.

[36] William T. Vetterling William H. Press, Saul A. Teukolsky and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1986.

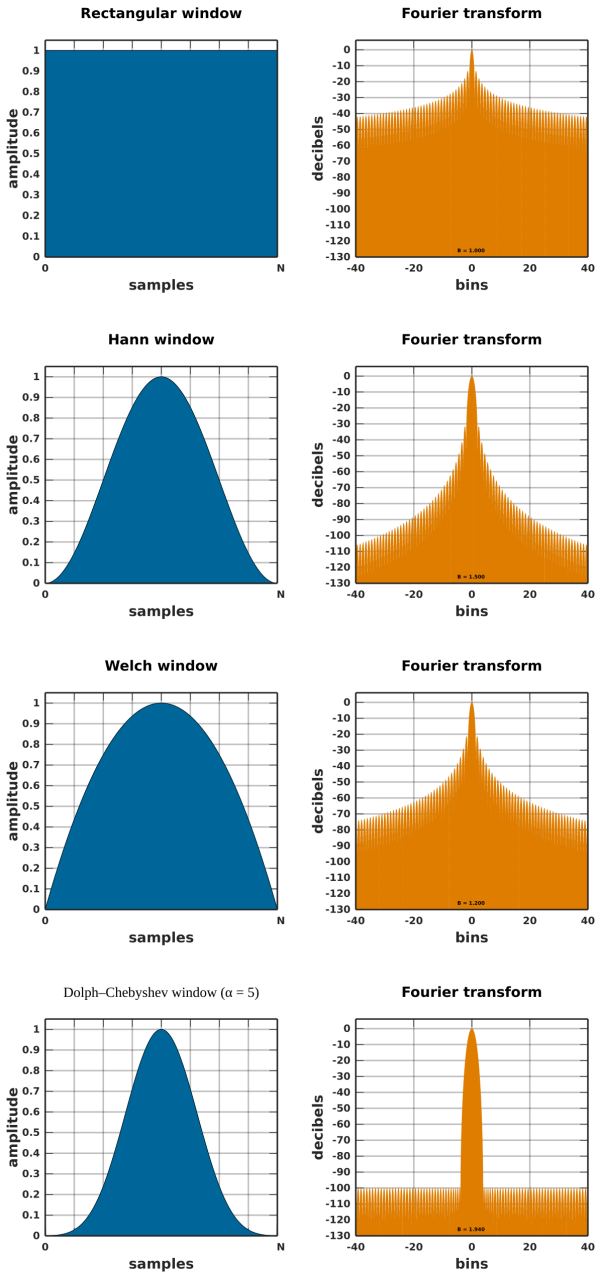

Figure 10: Examples of the spectral leakage of some window functions.