Auto-Transcriber for Electric Guitar

Project Number: **20-1-1-1980**

Project Report

Student: Gersh Yagudaev ID: 913224317

Student: Ethan Daniel ID: 996219598

Supervisor: Omer Tzidki

Project Carried Out at: Tel Aviv University

Contents

Abstract

Throughout the course of this project, our team has constructed a software system capable of performing automatic transcription of electric guitar in real time, representing results using tablature – a guitar specific notation system. The theoretical backbone of this project is in the field of Music Information Retrieval (MIR) which leverages (digital) signal processing insights to extract actionable information from musical signals.

The specific purpose of our project was to implement transcription functionality in *real time*. In our opinion, this use case is underrepresented in industry, and is the most widely marketable fruit of the MIR field. To achieve this purpose, we wrote our program in multiple concurrent processes, effectively pipelining them, thus allowing for a significant increase in performance.

The latest version of our software consists of five processes – retrieval, segmentation, pitch estimation, tab generation and tab output (Figure 1). These constitute various subsystems of the project and are discussed in depth in the Implementation section of this report.

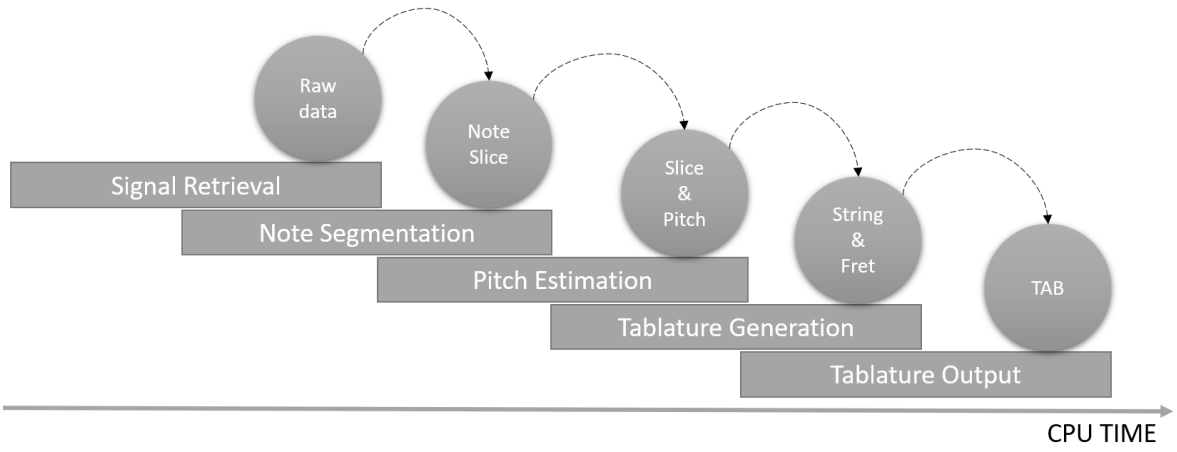


Figure 1 - Block Diagram (Processes)

1 Introduction

The goal of this project was to create a product: something that can generate a tablature based on audio – a so-called “speech-to-text” for music. This was motivated by a void in the market in this space – there are very few products aimed at real time transcription, making the task interesting and providing us with valuable industry insights. On a personal note – one member of our team is a guitarist, and has been unsuccessfully searching for a system similar to what we developed for a considerable amount of time.

During the planning stage, we worked to narrow down the amount of information that we need to retrieve from the signal for a reasonable estimation of the music. We did this to keep the program as “lean” as possible – avoiding future bottlenecks in performance. Upon reviewing available literature and our own experimentation, we determined that only two pieces of information are truly necessary for transcription: note start/stop times and their fundamental frequencies.

Upon investigation of how we might realize the above features in real time, we came up with a program that consists of several subsystems in a pipeline architecture. If all the systems work – the overarching program will work. Thus, our project’s goal simply becomes “implement every subsystem”, making evaluation easy and unambiguous (Table 1).

Table 1 - Project Goals

System	Done	Purpose	Comment
Multiprocessing framework (“Manager”)	YES	Enable launching & communication of multiple concurrent processes	
Real Time Audio Input	YES	Interface with the ADC and retrieve audio samples from the guitar	Preprocess data using “Voicemeeter Banana” as a virtual noise gate
Note Segmentation	YES	Detect when notes start – output a corresponding slice of data	
Pitch Estimation	YES	Estimate the musical note pitch in a data slice	Only monophonic detection
Expression Detection	NO	Identify musical embellishments (vibrato, bend, slide…)	Not implemented due to time concerns, immense difficulty, & low degree of priority for the overall system
Tablature Generation	YES	Deduce string & fret position on the guitar from given onset time & pitch	
Tablature Output	YES	Draw a tablature on screen based on the generator, and display it in real time	

2 Theoretical background

2.1 Musical Background

2.1.1 Tablature

Tablature (tab) is a form of music notation that represents instrument fingering and the order in which notes are triggered. It is commonly used for fretted instruments, such as the guitar. Tablature does not contain rhythmic information, although it can be modified to do so.

Tab is written as a sequence of numbers superimposed on a set of six horizontal lines, one for each string. Thus – the numbers represent the fret, and the line on which the number is drawn represents the string.

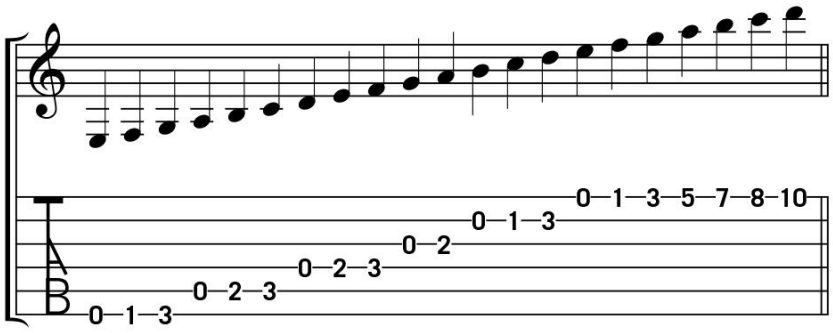


Figure 2 – Staff notation (top) versus Tablature notation (bottom)[1]

2.1.2 Fungible Notes

An interesting property of the electric guitar is that it can produce fungible notes. In other words – the same exact sound can be created by activating notes in different locations: fret E-0 will yield the same sound as B-5. This is diagrammatically represented in Figure 3. Due to this phenomenon – even someone with complete auditory knowledge of a note would still be choosing from multiple equally valid placements on the fret board.

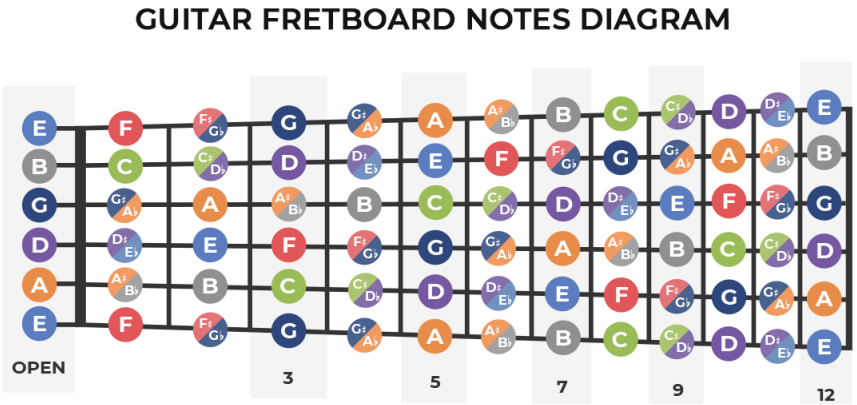


Figure 3 – Note mapping of the guitar (up to fret 12)[2]

[1] [9]

[2] [12]

2.1.3 ADSR Envelope

When a note gets activated in an electric guitar, it goes through four phases before disappearing, known as the ADSR Envelope (**Error! Reference source not found.**):

1. Attack – a rapid rise in energy & amplitude as the string is displaced from equilibrium and then released. This phase is very quick and encompasses a lot of energy. It also contains a lot of spectral "noise" as activation of a note is a non-ideal phenomenon. You could see effects similar to contact bouncing (where a string bounces against the plectrum for a few moments, generating a buzz), or biaxial vibration of the string resulting in collisions with the frets until the vertical component of vibration is attenuated. This translates to a short-term increase in energy across most frequencies.
2. Decay – a rapid damping and regularization (decay of the vertical vibration component and disappearance of attack transients) of the strings oscillations as energy levels in the note decay to a more stable level.
3. Sustain – the main body of the note. The string vibrates at a constant frequency. This is also the longest phase, with higher quality guitars having longer sustain.
4. Release – a reduction in oscillation amplitude as the energy deposited into the string by the musician finally drops below audible levels.

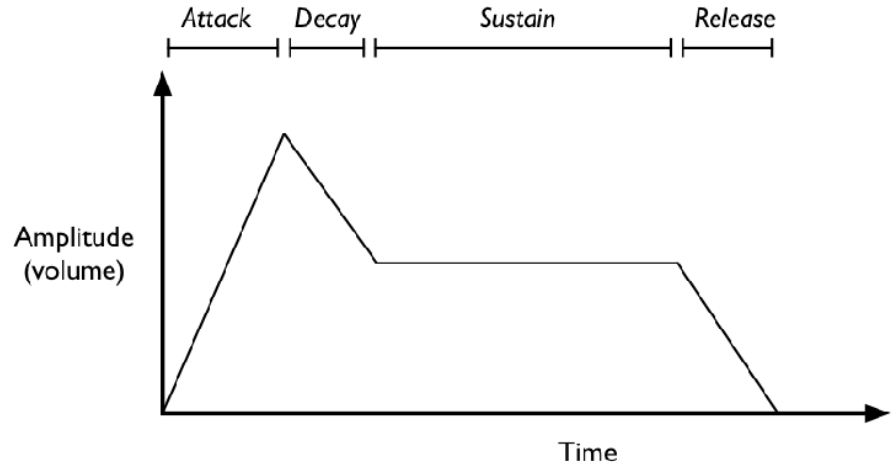


Figure 4 – ADSR Envelope

The above theory is confirmed by our own laboratory experimentation, where we manually built a dataset containing every note on the guitar. We did this by amplifying the output of the guitar using a pre-amp and feeding this signal into an oscilloscope at a sampling rate of 400Hz. This sampling rate is not sufficient for high-quality transcription, but is good enough when analyzing the

general anatomy of the signal. We can observe that the signal does indeed follow the theoretical ADSR Model (Figure 5).

Further – if we look at the spectrogram of the measured signal, we can see the presence of multiple “non-musical” frequencies during the attack phase of the note, as highlighted in Figure 6.

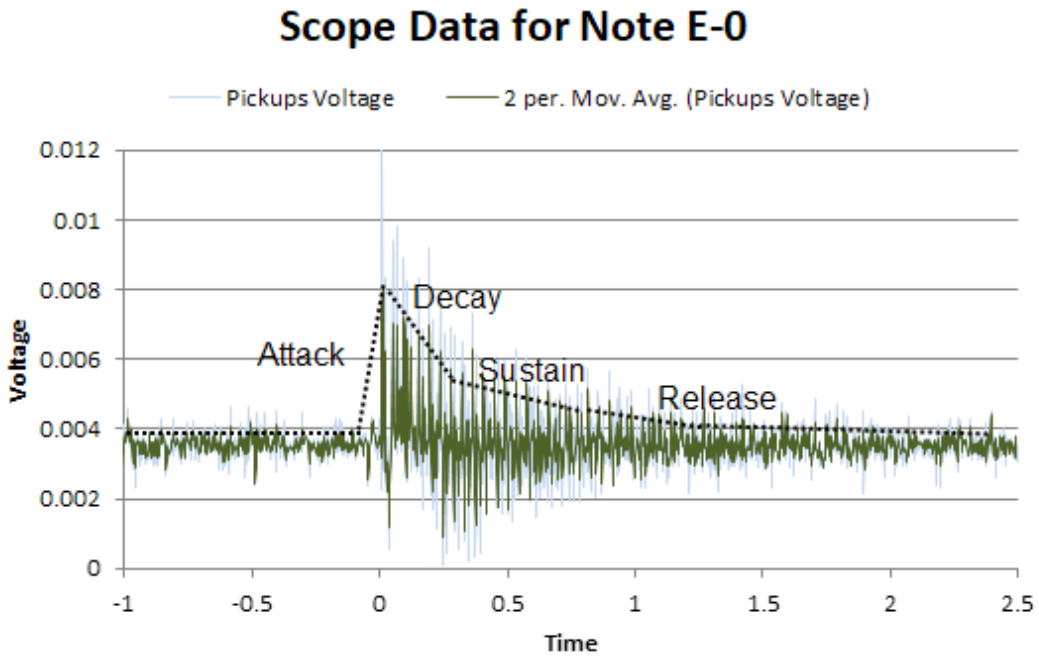


Figure 5 – Experiment Results (Note E-0)

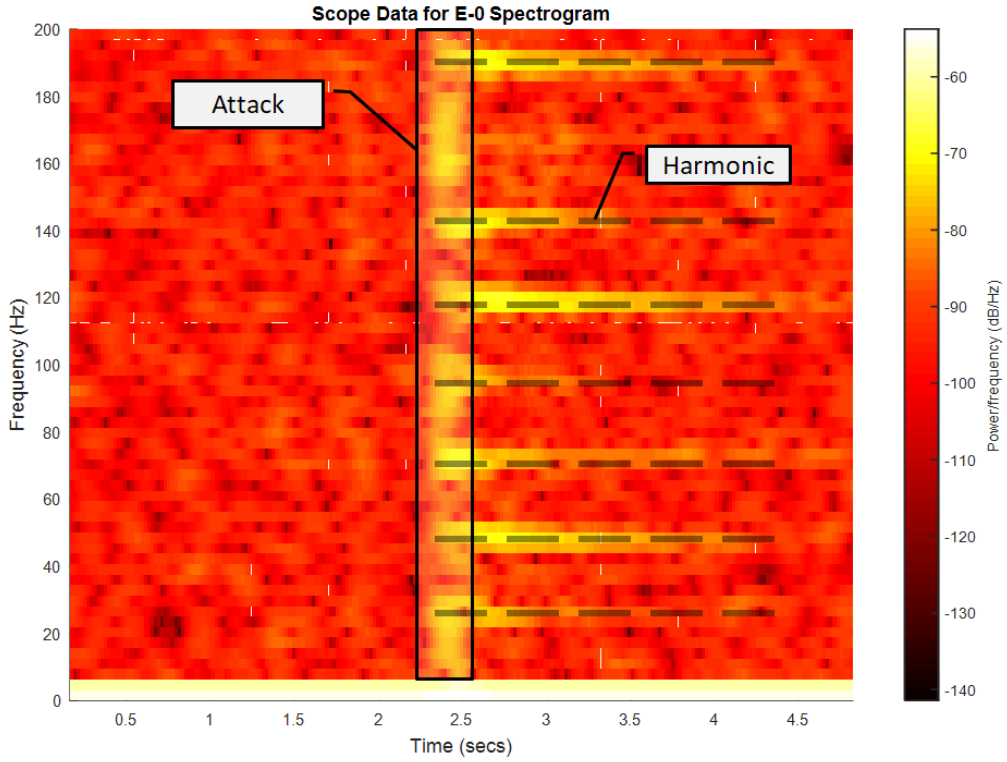


Figure 6 - Spectrogram of Scope Data

2.2 Segmentation Algorithms

2.2.1 Definition & Problem Statement

Segmentation of a music signal is the act of slicing a data stream into blocks containing individual notes. To do this effectively an onset detection algorithm is required. The goal of onset detection is to determine the time-locations of all note events in a signal. To this end, an onset is defined as a single instant chosen to denote the earliest sample of a given note.

Onset detection is an active area of research in MIR. We focus on the Novelty Function (aka Detection Function) approach. The modus operandi of such algorithms is relatively straightforward – one must construct a measure of novelty in the signal (how different is current data compared to the past) and pick the peaks of this function. In reality though, both of these steps are far from trivial.

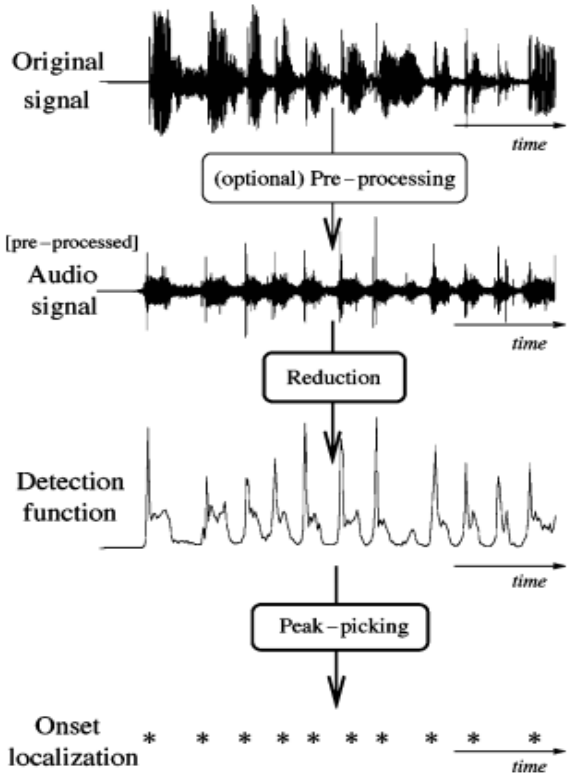


Figure 7 - Flowchart of standard onset detection algorithms[3]

We identified the following Novelty Function approaches in literature:

- Time Domain Energy Novelty
- Spectral Energy Novelty
- Phase Novelty
- Spectral Energy Distribution Novelty (Spectral Flux)

[3] [1]

Most of these novelty functions make use of $X(n,k)$ which is an STFT representation of the signal:

$$X(n,k) = \sum_{m=-\frac{N}{2}}^{\frac{N}{2}-1} x(hn+m)w(m)e^{-\frac{2j\pi mk}{N}}$$

where $w(m)$ is a window function

Our project utilizes the Blackman-Harris window due to its theoretical similarity to the Hamming window, but with a great dynamic range and respectable side lobe rejection (Figure 8).

When determining which window to use, we needed to identify what requirements we have for dynamic range and resolution of the Fourier Transform. A high dynamic range would give us a wider frequency aperture and better sensitivity to partials of different magnitudes, but reduce our ability to resolve frequencies that are near each other.

We expect to work with frequencies in the range of 80Hz to approximately 1800Hz. Since we are assuming monophony, we do not have to worry too much about resolution, since we know that all relevant frequencies will be at integer multiples of the fundamental. However, we still somewhat care about it, since, for example, the 82Hz note would have a 1st harmonic at 164Hz. The spacing between these two partials is 82Hz, which is 4.7% of the proposed bandwidth. It is also important to realize that while we have frequencies predictably spaced out in the band, their amplitudes can occupy a variety of ranges. Each subsequent harmonic typically carries less power than the previous, and yet we need to detect as many as we can. Because of this, we can confidently state that a high dynamic range is necessary for our window.

To summarize - we wanted to use a window with low spectral leakage (to have high dynamic range), and can afford to sacrifice some resolution, but not too much. A standard solution to the above requirement would be the use of a Hamming or a Hann window, which are commonly used for audio applications. However, after some testing we established that the Blackman Harris window is more suitable for our application.

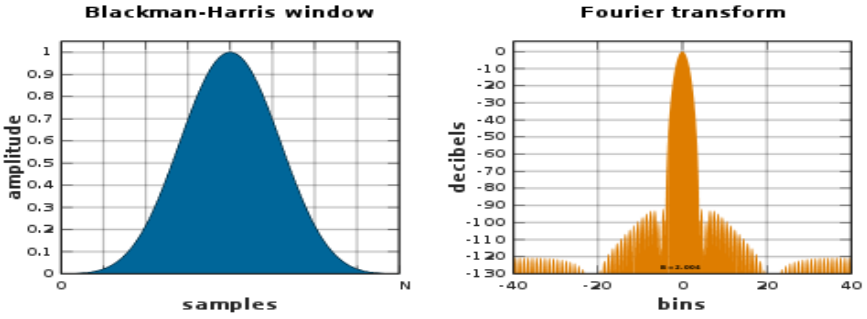


Figure 8 – Blackman-Harris Window

2.2.2 Time Domain Energy Novelty

The simplest approach to finding a Novelty Function would be an Energy method, which was one of the first algorithms developed for onset detection. The theoretical idea here is that the energy of a guitar signal would follow an ADSR envelope, as shown in Figure 4. Therefore, we would expect to see peaks at the transition between Attack and Decay of each note, giving us the onset, conditional on our ability to resolve and detect the peaks. The novelty function computes RMS energy consecutively for each frame of an audio signal according to:

$$\text{For signal } X = \{F_i\}_{i=0}^{i=M} \text{ Where } F_i = \{x_j\}_{j=0+i\cdot N}^{j=N+i\cdot N},\ M = \left\lceil \frac{\text{len}(X)}{N} \right\rceil$$

(The last frame is zero padded to match a static frame length)

$$E_{RMS}^{(k)} = \sqrt{\frac{1}{N}\sum_{j} |F_k|^2}$$

At this stage we have an energy that can be both positive and negative (as the signal oscillates around 0). To turn this into a novelty function we take the first order difference of the RMSE time series, giving us the rate of change of energy through time. The result is indeed a novelty function, but it will have peaks above and below zero. Since we don't care about the peaks below zero, we perform half-wave rectification and end up with an RMSE novelty (Figure 9).

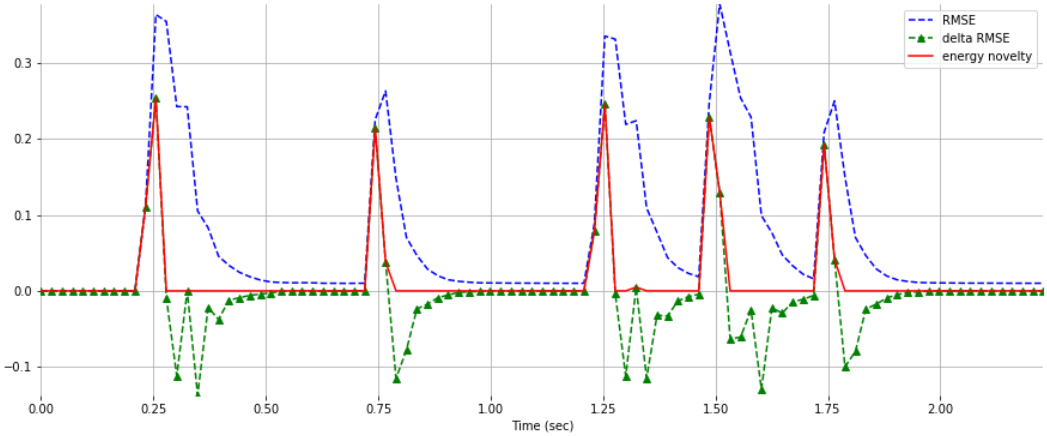


Figure 9 -RMSE Novelty Function[4]

This approach is problematic, and not well suited for our application. What we actually measure here is total energy change over time. This does not necessarily correspond to note onsets. Consider a scenario, where a guitarist continuously activates notes, but does so at identical volume and at a high

[4] [14]

speed. This is called tremolo and is a widely used technique. The energy level would remain relatively constant and the ADSR envelope would not be fully present in the RMSE, thus making the novelty function useless.

At the same time, the RMSE Novelty function is sensitive to energy fluctuations within the same note, which can happen due to physical effects of the guitar (e.g. sympathetic vibrations of other strings causing periodic fluctuations in energy due to constructive/destructive interference), as well as playing techniques employed by the guitarist. In these cases multiple peaks would be present within the same ADSR envelope – making peak picking a very difficult task.

To illustrate the above concerns, we have tested the RMSE novelty algorithm and our results confirm its inapplicability to our project (Figure 10). We can observe that the envelope contains significant ripple, with the same note producing multiple peaks. Further – we can see "false onsets" in peaks at magnitudes comparable to those of "real peaks" which all but guarantees a heightened rate of false positives if this algorithm were to be employed.

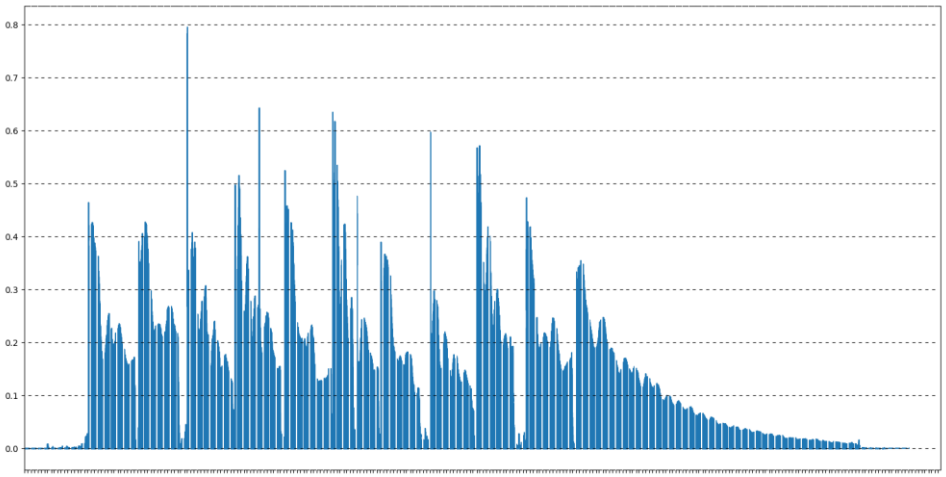


Figure 10 - RMSE Novelty Function Test

2.2.3 Phase Based Novelty

This approach uses the fact that the rate at which the phase in an STFT bin changes is a rough estimation of the instantaneous frequency of that component. We can calculate this change by finding the first order difference in the phase $\angle X(n,k)$. A note onset would imply a change in the instantaneous frequency across multiple frequency bins and thus can be detected by a phase measure.

Let $X(n,k)$ be an STFT of some signal $x[n]$. Then:

$$\mathrm{X}(n,k) = |X(n,k)| \cdot e^{j\varphi(n,k)} \text{ Where } \varphi(n,k) = \angle X(n,k) \in (\pi,\pi]$$

Thus the instantaneous frequency: $\varphi'(n,k) = \varphi(n,k) - \varphi(n-1,k) \in (-\pi,\pi)$

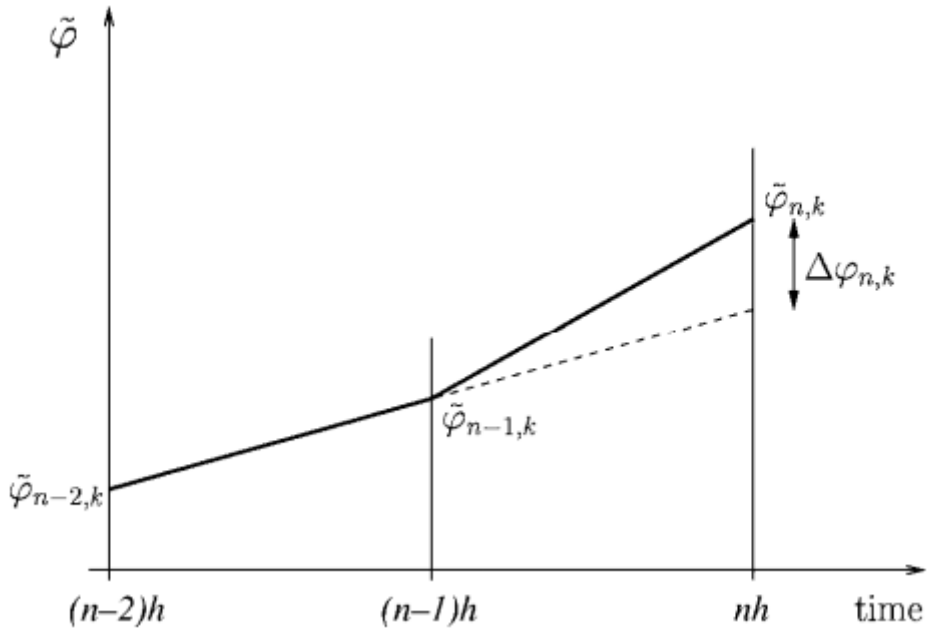


Figure 11 – "Instantaneous frequency over adjacent frames. For a stationary sinusoid this should stay constant (dotted line)"[5]

This instantaneous frequency is our indicator. To calculate the novelty of the indicator we take its first order difference, which corresponds to a second order difference of the phase:

$$\varphi''(n,k) = \varphi'(n,k) - \varphi'(n-1,k) \in (-\pi, \pi]$$

Now to transform this measure into a proper novelty function we must look at the absolute change in novelty across all frequency bins. In other words, we sum the novelty magnitude at each time step across frequency bins:

$$D(n) = \frac{1}{N}\sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} |\varphi''(n,k)|$$

This approach performs poorly, especially in the non-ideal context of a real time input. There is a lot of noise in our signal, partly due to EMI, but also as a result of the guitarist's movements (simply touching a string would generate a signal, even if no note is played). Since phase is independent of magnitude, these "insignificant" components would be weighted equally with the valuable high-energy components, resulting in a low quality result. This is solved by introducing an energy-based weight into the equation, at the expense of greater computational spending.

[5] [1]

2.2.4 Spectral Flux Novelty

A more fruitful approach is to calculate the Spectral Flux Novelty Function, which builds upon the principles of our previous discussion, while resolving problems which made other algorithms undesirable for our application. This approach is similar to the RMSE Energy method, but accounts for novelty across multiple frequency bins:

1. Compute log-amplitude spectrogram using STFT: $X(n,k)$
2. Compute 1st order difference for each frequency bin, then half-wave rectification to obtain $\Phi(n,k) = H(|X(n,k)| - |X(n-1,k)|)$ where $H(x) = \frac{x+|x|}{2}$ is the half-wave rectification function.
3. For each frame perform vertical summation (sum contents of all the frequency bins), to get detection function $D(n) = \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \Phi(n,k)$

This algorithm is superior to the energy function since it looks at novelty distribution across frequency. This exploits a spectral feature of the signal: an attack corresponds to a wider distribution of energy across frequency (Figure 6). As a result – this algorithm is able to reject intra-note energy fluctuations, and can perform well under high speed conditions. It is also superior to the phase approach due to better accuracy[6].

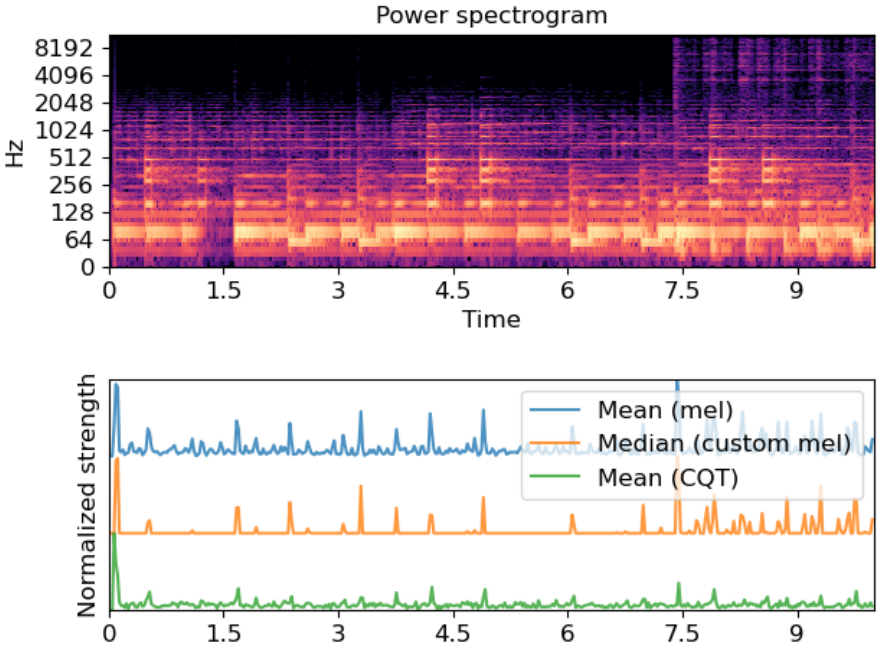


Figure 12 - Spectral Flux Novelty (Librosa)[7]

[6] [2]

[7] https://librosa.org/doc/0.7.0/generated/librosa.onset.onset_strength.html

It also worth noting that this algorithm can be tuned significantly by altering the parameters of the STFT (hop length, frame width, window type, etc) as well as defining alternate spectrogram measures.

2.2.5 Peak Picking

The general approach in literature is to implement an algorithm that finds local maxima in the novelty function under various constraints. Thresholding is common, and probability based approaches are also used. Depending on how these parameters are selected, peak picking performance can change drastically.

There are two main non-probabilistic approaches to peak picking. The first method selects local maxima above a threshold. This requires access to past and future samples, making it unsuitable for real-time processing. The second method attempts to maintain a real-time threshold, updating it when it is exceeded[8]. This approach, despite being suited for speed, has a very high false-positive error rate, rendering it almost unusable for our application.

We utilize a tunable method that does not require knowledge of the future in order to function, but can still benefit if it is provided. We can therefore give it a small amount of future data while still considering it real time. This is because the added delay is tiny - 22ms for 1k samples of "look-ahead" at our current sampling rate.

Let $D(n)$ be the onset detection novelty function. A sample i is selected as a peak if it satisfies three conditions:

1. $D(n) = \max(D(n - \omega_1 : n + \omega_2))$,
2. $D(n) \geq \text{avg}(n - \omega_3 : n + \omega_4) + \delta$
3. $n - n_{\text{prev. onset}} > \omega_5$

Where ω_i are user determined tuning parameters and δ is a threshold.

2.3 Pitch Detection Algorithms

The goal of pitch detection is to identify the value of the musical note present in a sound. In the context of music, this implies detection of the fundamental frequency of the signal. The sound produced by a guitar consists of many frequencies at integer multiples of each other, such that the set of frequencies present in a note is:

---

[8] [2]

$$S_i = \{f_0, 2f_0, 3f_0, \dots, Nf_0\}$$

Humans only perceive f_0, while all the other partials determine timbre – the distinguishing sound of the instrument. In this project we restricted ourselves to exclusively monophonic pitch detection, meaning that we assume that only one note is sounding at any given time.

2.3.1 Autocorrelation Method

Autocorrelation is a function that is typically used for identifying repetitive patterns in a signal. Therefore, when it is applied on a short timescale it can estimate the fundamental frequency of a sound. It is given by:

$$r(k) = \sum_n x(n)x(n-k).$$

The algorithm works to find a value of k such that the autocorrelation is maximized. All autocorrelation does is shift a copy of the signal by k and check how similar the result is to the source signal. The closer k gets to the fundamental period, the more similar the shifted and un-shifted signals become, and thus their product increases. As a result, extracting pitch from autocorrelation could be as simple as $\mathrm{argmax}\big(r(k)\big)$.

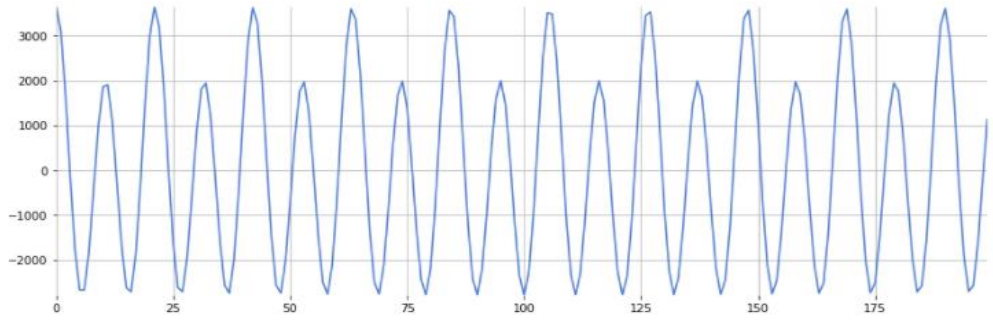


Figure 13 - Example Autocorrelation Function[9]

This algorithm has good speed, scaling approximately with $N \times \log(N)$ where N is the window length[10]. It requires a stable pitch for optimal performance, and this is not always guaranteed with an electric guitar, as the guitarist can bend the strings thus altering pitch without an onset.

The algorithm compensates for the steep stability requirements with a very good ability to detect weak fundamentals, which can frequently occur during standard guitar playing. This method also has an issue with octave errors which occur when it mistakes the $2f_0$ harmonic of a signal with the fundamental.

---

[9] [15]

[10] [3]

2.3.2 Harmonic Product Spectrum Method

Unlike autocorrelation, this algorithm operates in the frequency domain. It is based on the understanding that a note's spectrum will contain harmonics at integer multiples of the fundamental.

If we downsample the spectrum of a note by an integer, the new spectrum would still have a peak at the fundamental frequency since the harmonic at $N \cdot f_0$ gets shifted to f_0 when downsampled by N.Therefore, we can generate a set of downsampled instances of the original spectra, each with a different integer downsampling factor. Each of these would contain a peak at the fundamental, but not necessarily at its integer multiples. If we then proceed to multiply or sum these spectra, we expect to end up with a global maximum at the fundamental frequency, which we then peak-pick, thus getting f_0

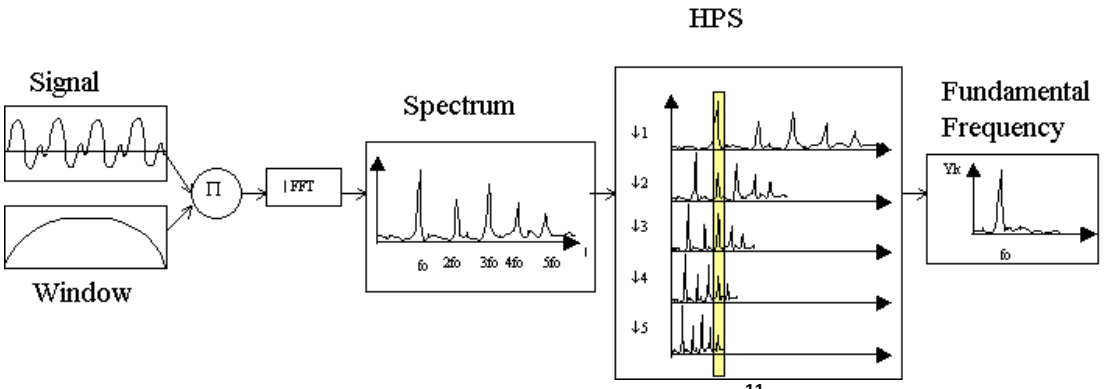


Figure 14 - HPS Block Diagram[11]

[11] [5]

3 Implementation

As mentioned in the Introduction, our implementation of consists of multiple subsystems, outlined in Figure 15. The global flow of our software is:

1. Analog guitar signal enters ADC, gets forwarded to amplifier and the PC.
2. PC receives input, sends it to a virtual noise gate.
3. Audio Input process polls a virtual audio device created by VoiceMeeter for samples using PortAudio.
4. When a full, legitimate frame is collected, it is sent to segmentation.
5. When Segmentation detects onsets, generate a slice containing the note.
6. Send the slice to Pitch Detection, estimate the fundamental frequency.
7. Send an object containing the segment, onset time and pitch to Tablature Generator, identify string & fret for the note.
8. Forward the string & fret information to Tablature Display, which outputs a real time tab using matplotlib.

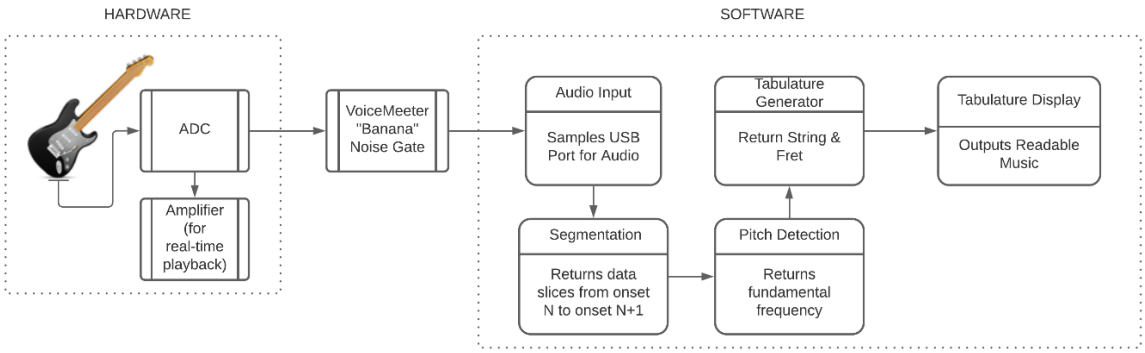


Figure 15 - Overall System Block Diagram

3.1 Hardware Description

We use a commercial audio interface (ADC and Preamp) to enable data collection for our software. The device we are using is an Irig HD2, which can output over USB. An audio interface is necessary to run our software.

This project has been designed specifically for the Fender Squire Stratocaster guitar with the Humbucker pickup selected. Our experimentation shows that other configurations will function, and some will even outperform the default in certain cases: If the environment is particularly noisy and the signal is dirty as a result – it is worthwhile to switch to the neck pickup and set the ‘tone’ knob all the way down to 1. The tone knob controls a variable LPF, and when aggressively tuned it can filter out the noise.

We also utilize an amplifier as part of our project – to be able to hear the music. Despite our ability to perform real time transcription – we are unable to

have real time playback, due to much stricter latency requirements. A delay of 0.1s on transcription is completely acceptable, but a delay of 0.1s with audio is not. Therefore – if a guitarist wants to hear their music while it is being transcribed, we advise a parallel connection to a real amplifier.

3.2 Software Description

Given the modular structure of our project, we believe it makes the most sense to discuss each system independently. Thus, this chapter will be divided into individual sections for each subsystem or relevant structure, taking a deep dive at how they are implemented.

3.2.1 Tools & Libraries

The entire program is written entirely in Python 3. We utilize a range of freely available libraries in our code, with the most important ones being:

- *Numpy, Scipy*: Used for efficient array handling and various mathematical operations
- *Matplotlib*: Used to enable output & plotting
- *Collections.Deque*: Used to access a deque data structure, which is an array optimized for appending and popping from either end
- *SortedContainers.SortedDict*: Used to gain access to a sorted dictionary data structure to efficiently store information about possible notes
- *SoundDevice*: A python wrapper for PortAudio, enables audio input
- Multiprocessing: A built-in python library to support spawning processes and enabling their communication
- *PeakUtils*: A toolbox of functions related to the detection of peaks on 1D data
- *Librosa*: A package for music and audio analysis. Provides building blocks for MIR operations.

3.2.2 Note Object

To standardize the exchange of data between processes, we created a class called *note_object*. This is a container that can hold a note's onset time, pitch, corresponding samples, string and fret. It also provides a sorted note dictionary (sortedDict). This dictionary is used by a built-in function to round the estimated frequency (returned by pitch estimation) to the nearest legal musical frequency. For example, note E-2 has a fundamental frequency of 82.41 Hz, but pitch detection can return values such as 80Hz or 85Hz. The closest-note method will look at the dictionary of legal notes and return the note that matches most closely to the estimated fundamental.

3.2.3 Main Process

This process is straightforward – we define five sub-processes: sender, segmenter, pitch detector, tabulator, and plotter. We then generate four

bidirectional pipe objects to form connections: sender-segmenter, segmenter-pitch detector, pitch detector-tabulator, tabulator-plotter. Each of these pipes is a duplex, even though our information only travels one way. This choice was made based on performance – a duplex pipe is faster (no need to check information direction). Once all these sub-processes are defined, Main starts and joins them all, so that the program will not terminate if any sub-process is running.

3.2.4 Audio Input Process

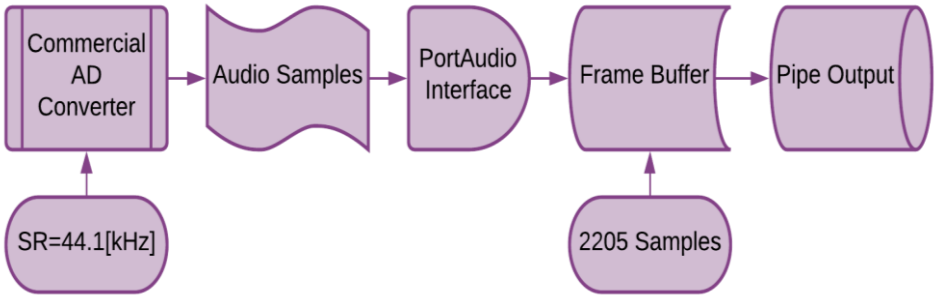


Figure 16 - Audio Input Block Diagram

This process utilizes SoundDevice to collect audio data. It is a Python module that provides bindings to PortAudio – an open-source cross platform audio I/O library.

This method of audio I/O utilizes the callback paradigm, which feeds a block of code (the callback function) as an argument to a stream object. This stream object proceeds to call the function repeatedly subject to certain parameters, to obtain an overall sampling rate of 44.1[kHz]. Our stream is bidirectional (input and output), with a single channel, a block size of 2205 samples, and a float32 data type. The benefit of this approach is that the audio retrieval can happen asynchronously to the rest of the code within the same process. We do not utilize the output side of the stream, but keeping it present does not incur a significant performance cost while enabling forwarding of the data to other programs if desired.

On every call of the callback function, we append a frame of audio (2205 samples) to the right edge of a deque object. At the same time, we run an infinite while loop from inside the stream object, which continuously tries to pop an element from the left side of the deque. This element would be the oldest unprocessed frame. If the pop is successful, that means that we indeed have frames to process. If the pop fails, it would be because of an index error (trying to pop the 0th element of an empty deque). In this case, we catch the exception and do nothing, since the failure means that we are

processing frames faster than receive them. This is not a problem, and we simply wait for the deque to fill up again on the next cycle.

Once a frame has been popped from the deque, we check whether it contains valid data. This is performed by doing a simple summation over the entire frame and checking whether the sum is above a certain threshold. The purpose of this is to detect silences. A period of silence would still contain variations both in frequency and in energy. We do not want to process this because it contains no useful information. An alternative to this decision would be to instead send a zero-filled frame. This would enable us to have absolute, instead of relative onset precision, but we do not require such functionality.

Once frame validity is confirmed, we place it into a pipe which leads to the segmentation process. We set up the pipe such that until the current object in the pipe is received by the target, the pipe cannot be filled with another object, and the process will wait for the pipe's "input slot" to free up.

This approach to sending data has a danger – what happens if the destination process, for some reason, takes a long time to receive the information, and the source process is trying to send several more frames? This would result in dropped frames. Because of this we store frames in a deque, as mentioned earlier, so that they can accumulate and be processed when possible.

3.2.5 Segmentation Process

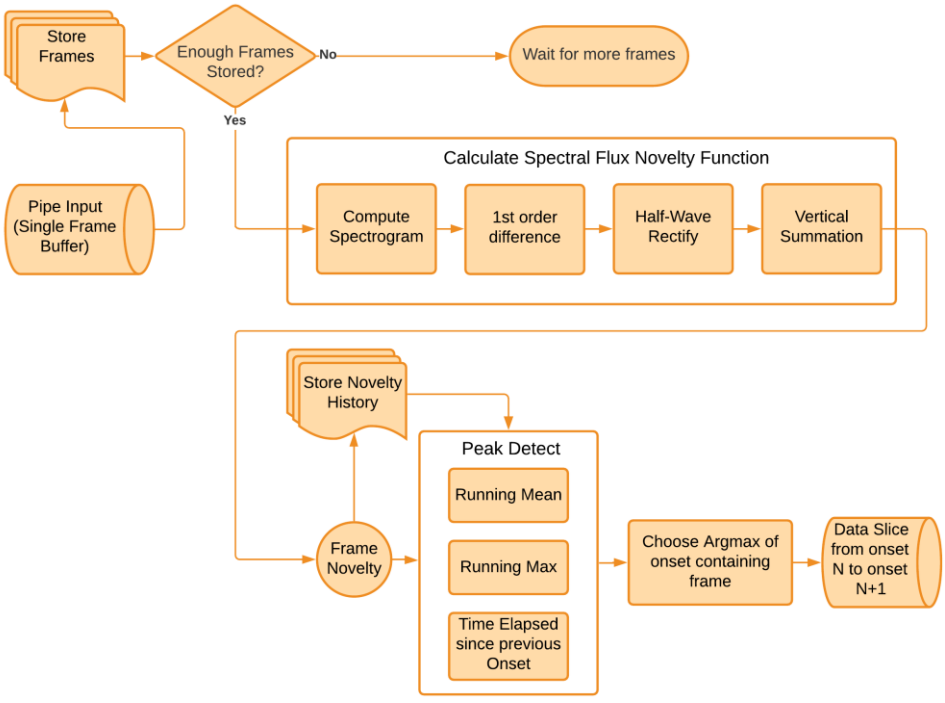


Figure 17 - Segmentation Block Diagram

The segmentation process is the most computationally expensive process in our program, aside from drawing the output with matplotlib. It is also the most important – as erroneous segmentation will result in bad data being forwarded to the pitch estimator.

This process runs inside an infinite while loop, repeating a specified sequence of actions indefinitely. It begins by polling the input pipe for an object. We pass the argument 'None' to the polling function, which means that the poll will not return until an object is present. Because of this decision, the segmenter will not 'run ahead' of the input and will activate when an input is provided.

Once an input is received from the pipe, the data is formatted (flattened into a 1d time series) and appended to a storage array. This storage array will contain all samples received by the process since the program started. This is beneficial because our onset detection algorithm improves when having access to past samples (due to a larger FFT), and the cost of doing this is low. Our data format is a single precision floating point, which occupies 32 bits per sample. Therefore, even an hour-long recording would occupy $\left(\frac{32}{8}\right) \cdot 44100 \cdot 60 \cdot 60 = 635{,}040{,}000\ [B] = 605\ [MB]$ of memory. This is completely acceptable as 8GB of RAM is standard nowadays. Further – this is for an hour of continuous operation, which is a very long time. Should the need for more efficient memory management arise – the fix is simple – set a limit on how much memory the recording can occupy and discard oldest samples whenever the limit is exceeded. As a side note – storing the entire input history this way gives us the ability to build longer-term analysis functions on top of the existing platform to further improve accuracy in the future.

After the input side of the current cycle is complete, and our recording array is updated, we check whether the array contains enough information to attempt onset detection. This is important, because the less data segmentation sees the more sensitive it becomes to noise and other signal imperfections. It is unreasonable to run onset detection until a critical mass of data is reached. In our case, we empirically determined that a history of at least 20 frames is sufficient for reasonably good onset detection at a sampling rate of 44.1 [kHz].

While the above approach solves the problem of data underflow, it introduces a problem in the form of computational bottlenecking. We must operate on an array of a constantly increasing size and constantly check whether its size is above a certain threshold. Fortunately – this was not too much of an issue, and we brought performance in line with expectation by updating the recording every 2 frames, instead of every frame. Further – since we modify the history array with an *append*, there is no performance cost for this large array, since append runs in O(1) time.

Now that we are confident in the data, we can proceed to onset detection. We utilize Librosa's *onset.onset_detect* function, which is an efficient implementation of the spectral flux novelty approach. Internally, *onset.onset_detect* utilizes two functions: *onset.onset_strength* and *util.peak_pick*. Both are highly tunable operations. This implementation is based on "Evaluating the Online Capabilities of Onset Detection Methods." Published in ISMIR in 2012.

Onset_strength computes a spectral flux novelty function exactly as in the algorithm described in 2.2.4. Our hop-length is set to 256, while Librosa recommends 512. Our sampling rate is also double that of Librosa's native rate, and therefore our hop length hops 25% as far as Librosa's native rate in units of time, resulting in better precision. This is especially important because our window has considerable attenuation at the edge of the band, so we want to increase the amount of overlap in each STFT frame to avoid losing information.

Peack_Pick attempts to identify the positions of the onsets present in the novelty function, using an algorithm described in 2.2.5. This algorithm has many parameters that need to be selected for good operation. They are presented in Table 2**Error! Reference source not found.**.

Table 2 - Peak Picking Parameters

Parameter	**Default Value**	**Our Value**	**Function**
Pre_max	$0.03\left\lfloor\frac{SR}{L_{hop}}\right\rfloor = 5$	100	# of past samples for Max calculation
Post_max	1	100	# of future samples for Max calculation
Pre_avg	$0.1\left\lfloor\frac{SR}{L_{hop}}\right\rfloor = 16$	22050	# of past samples for Avg calculation
Post_avg	$0.1\left\lfloor\frac{SR}{L_{hop}}\right\rfloor + 1 = 17$	220	# of future samples for Avg Calculation
Delta	0.07	0.15	Threshold offset for Avg
Wait	$0.03\left\lfloor\frac{SR}{L_{hop}}\right\rfloor = 5$	10	Minimal distance between peaks
Backtrack	False	True	Return location of peak or location of the leftmost edge of the peak

Librosa determined their default values based on large scale hyper parameter optimization on a public static dataset, which has clean and high quality recordings. As a result, their parameters are not tuned for the inherent variability, inconsistency and plain dirt that is present in real time signals.

Our choice of parameters is drastically different from the defaults. We chose to have a range of 200 samples for evaluating the maximum, and to look at a symmetrical aperture around the sample of interest. Librosa's defaults look at only the last 5 samples, and the future 1 sample, which makes it very sensitive. Our choice reduces the sensitivity to variation in the signal, helping to select a true maximum over a longer period of time.

We also decided to use the last 22050 samples and the future 220 samples for calculation of the average. This corresponds to around 0.5[s] of data incorporated in the mean. While this does introduce a computational cost, we found that it is worthwhile due to much better precision and a lower rate of false negatives. If we used a smaller dataset, then in cases where many notes occur in quick succession, the average would become inflated and subsequent onsets would be missed. When we have a long window in which to calculate the average, we get a more stable number, and thus obtain more reliable detection.

Following in the spirit of making the detection "tighter" – we set a delta threshold of 0.15, which means that the algorithm needs to be very confident in the onset before it is reported.

We also set Backtrack=True, because we did not want to discard the onset itself from the note segment when sending it to the following processes. By backtracking to the nearest local minima to the left of the onset we end up with a more precise time-localization of the onset, and a more complete slice.

After an onset is determined, we store it. We then wait until a new onset is found later in time. We then generate a slice of the recording array indexed by $[\mathrm{Onset}_N : \mathrm{Onset}_{N+1}]$, which contains the samples of a note belonging to Onset N. We then create an instance of *note_object*, into which we store the note slice and the onset time.

Once a note object is created, it is ready to be sent down the pipe to pitch estimation. However, this has the same synchronicity issue as our audio-input process, where the segmenter might be done before pitch detector is finished with the previous note, resulting in lost information or even race conditions between two consecutive notes. To solve this, we again construct an intermediary queue-like structure. We add pending notes to the right of the structure, and pop them from the left whenever pitch detector is ready to receive input. The overall flow and performance of this process is demonstrated in Figure 18.

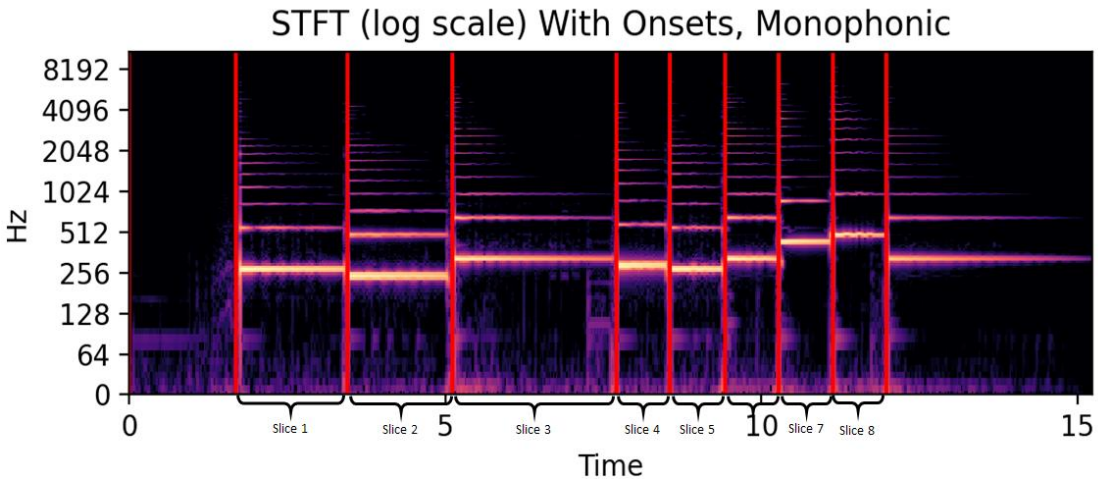
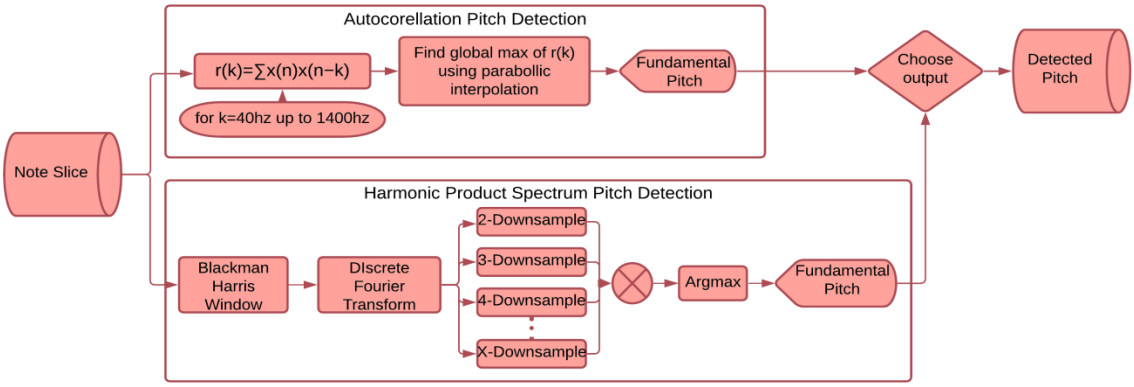


Figure 18 - Segmentation Test (Pseudo Real Time)

3.2.6 Pitch Detection Process



19 - Pitch Selection Block Diagram

Our pitch detection process uses two independent detection algorithms, combined with our custom logic for choosing the best output. We use the Autocorrelation and the Harmonic Product Spectrum algorithms, whose theoretical justifications are explained in 2.3.

The process starts as all the others – by polling the pipe for an input. Once this input is received, it calls both pitch detection algorithms and stores their results. Then, it uses our custom logic to select the output.

3.2.6.1 Autocorrelation

For the Autocorrelation method, we define a function that receives a note segment, sampling rate as well as boundaries of a region within which we permit the search for fundamental frequencies. The boundaries are important, because they prevent detection of fundamentals in components existing outside the band, such as noise or even the sound of fingers moving across the strings.

This function finds the autocorrelation of the note segment using *librosa.autocorellate,* which computes a bounded version of the metric and returns it as an array. Once this is calculated, we account for the frequency bounds provided as arguments by finding two parameters:

$$I_{\min} = \frac{\text{sampling rate}}{f_{\max}} \; ; \; I_{\max} = \frac{\text{sampling rate}}{f_{\min}}$$

We then proceed to null all members of the autocorrelation array that do not fall within the index range $[I_{\min} : I_{\max}]$. After this step, our array only contains components within the specified range of frequencies, and the only thing left is to find the maximum value.

Originally, we used the trivial solution: argmax, to find the maximum. However, we discovered that this is rather imprecise, and frequently causes octave errors, where the 1st harmonic is detected instead of the fundamental. To resolve this, we replaced argmax with *peakutils.peak.indexes*, which finds the numeric index of the peaks by taking the first order difference of the signal, then applying horizontal and vertical thresholding. The resulting output is an array containing indexes of the peak samples, similarly to argmax.

This performs better – the rate of octave errors is reduced, but the absolute precision is still poor. To resolve the issue, we employ quadratic interpolation around the detected maxima. This receives a vector $F[n]$ and a specific location $n_i = x$. The interpolator tries to fit a parabola to $F[n]$ and then returns the location of the peak of the parabola. Mathematically, the resulting fundamental frequency looks like:

$$f_0 = \frac{1}{\text{SR}} \cdot \left(\frac{1}{2} \cdot \frac{F[x-1] - F[x+1]}{F[x-1] - 2F[x] + F[x+1]} + x \right)$$

3.2.6.2 Harmonic Product Spectrum

The implementation of this algorithm was relatively straightforward. We receive three parameters: signal, sampling rate, HPS depth. The first two parameters are self-explanatory, and HPS depth determines how many spectra we will use in the HPS. For example, for an HPS depth of 3, we will calculate spectra downsampled by 1, 2, 3.

Upon receiving the signal, we window it with a Blackman-Harris of length equal to the input length. After the windowing, we compute the real FFT of the signal. Once the FFT is ready, we go into a for loop for x in range(2, HPS Depth). On each iteration of the loop, we downsample by an additional integer

and attempt to peak-detect using parabolic interpolation. We append this detected frequency to a list, and once we have reached the HPS depth we average all the detected frequencies.

This is a slight deviation from the standard HPS algorithm, where the pitch detection only happens once. This is because we operate with a small depth (depth=4), so errors are possible, but not in every iteration. Therefore by taking the average we are able to reduce the effect of one-off mistakes in the algorithm.

3.2.6.3 Output Selection

We created a custom set of logic to choose from two available outputs. The general intention here is that these two algorithms operate in different domains, and situations are likely where one of them makes an error and the other does not.

We designed the selection algorithm to check whether any algorithm output a clearly wrong answer that is outside the range of playable notes. If one of them did, then the other one must be selected. However, if both results are plausible, we check whether the answers are actually the same (i.e do the algorithms agree with each other). If they do – then we average the results. However, if the outputs do not agree, then one of them made a mistake. The most common mistake would be an octave error, which means that we detect $2f_0$ instead of f_0. Therefore we check if the division between either combination of outputs returns a value close to 2. It will never return exactly 2 because the detection is not exact, but we use the *isclose* function to make an estimate. If indeed one of the outputs is double the other, then we know an octave error occurred and choose the smaller value.

Once octave errors have been checked, we also look at the difference between the two outputs. Based on our experience, we observe that given that an error occurred, the probability that the error is an overshoot is greater than the probability of an undershoot. Thus, if the difference between the outputs is greater than either one of the outputs, then we choose the smaller value.

Finally, we check how many selections we made. If the above logic determined that it wants to select more than one option (the options are HPS, AUTO, or average of the two), then we give priority to the average. Otherwise we output the selected result. A diagram of this process is given in Figure 20.

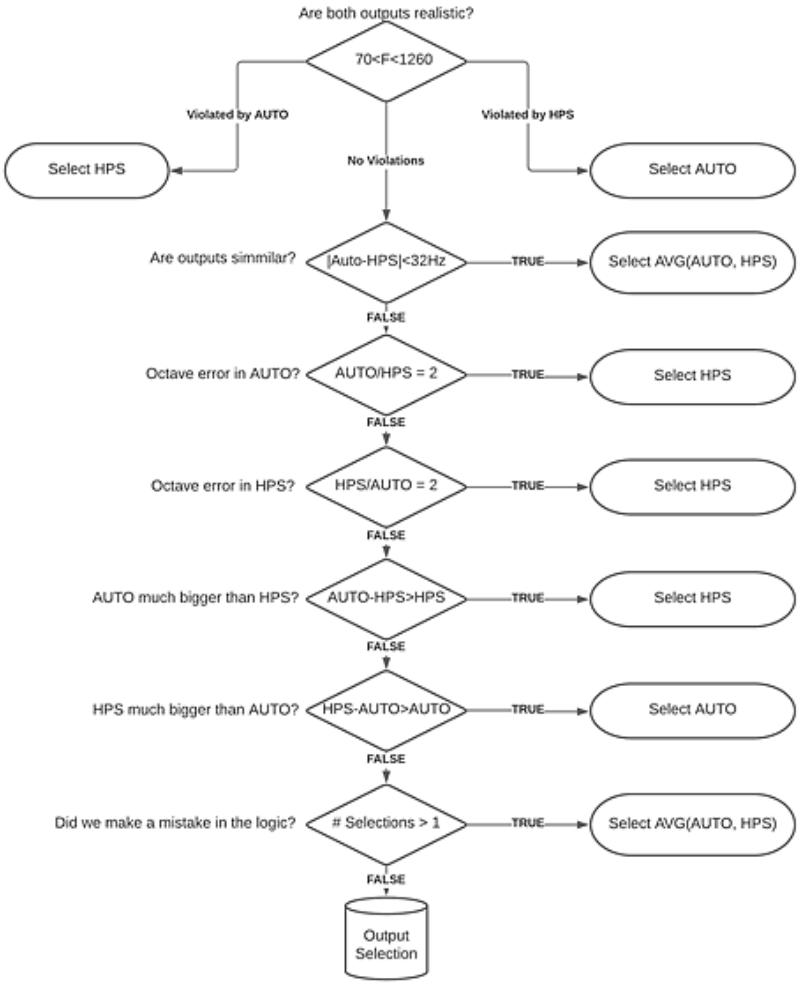


Figure 20 - Decision Tree for Pitch Detection Output Selection

3.2.7 Tablature Generation Process

As we discussed in 2.1.2, multiple locations on the guitar yield the same note. This makes the task of mapping given notes to the fret board quite difficult. Since no optimal solution exists, we chose to design & implement our own heuristic algorithm.

If we recall Figure 3, we can notice that any position on the guitar can be denoted with two numbers – a string number and a fret number. We can think of this as a Cartesian coordinate system, such that any point on the guitar is given by $p_k = (\text{string}_i, \text{fret}_j)$. We make the decision to limit our tablature to 20 frets only. Guitars have different scale lengths, and thus different numbers of frets, but the vast majority of music is played under fret 20. Based on the above, we observe that there are 120 possible locations.

When a real person writes a tab, they consider not only the musical accuracy, but also the ease of playing. This means trying to group notes as tightly

together as possible. In a mathematical context this would imply minimizing the distance between notes' locations. Another consideration is to prioritize selecting frets that are below 12, as playing higher up the neck is more difficult. Finally, one must also consider that horizontal distance (measured in frets) is more important than vertical distance (measured in strings), since it requires a larger physical movement from the guitarist.

Given a note, we define a set containing all the possible coordinates where it could be placed:

$$P_n = \{(s_1, f_1), (s_2, f_2), \dots (s_i, f_i)\}$$

Suppose two notes are available with their corresponding location sets:

$$\exists\ P_n, P_{n+1}$$

Define a weighted distance measure between any two points on the guitar:

$$\rho_1 = (s_1, f_1), \rho_2 = (s_2, f_2):$$

$$D = \sqrt{\text{Fret Weight} \cdot |f_2 - f_1|^2 + \text{String Weight} \cdot |s_2 - s_1|^2} \cdot \text{Playability Penalty}$$

Where,

$$\text{Fret Weight} = 1\ (\text{user defined})\ ; \text{String Weight} = 0.9 \cdot (\text{user defined})$$

$$\text{Playability Penalty} = \frac{f_1 + f_2 - 24}{24} + 1$$

$$\text{Further, IF } f_1 = 0 \text{ OR } f_2 = 0\text{: } D = \frac{D}{2}$$

What we have here is a modified Pythagorean distance formula, that applies a weighting to the x and y axes, such that fret distance has a larger contribution than string distance. It then applies a penalty based on how far, on average, do the two points exceed fret 12. Finally, if either note is located at fret 0 (open string) – we apply a very strong reduction to the distance (so as to incentivize selecting such points).

Considering the above, we designed our tablature generator to follow the following algorithm:

1. Receive note object with onset time and estimated pitch
2. Call the note object's built-in *closest_note* method to round estimated pitch to a legal note
3. Store this note object in history, and then sort history by onset time
4. Check if history contains at least two notes. If not – do not continue
5. Pop two notes from history
6. Identify all the possible coordinate pairs where either note could be placed.
7. Select a coordinate pair with the smallest Distance between them
8. Lock the selected coordinates in history, send them to tablature plotter

3.2.8 Tablature Output Process

In the plotting process we piece the notes together to generate a guitar tab along with a csv file that stores the tablature information. The plotter process

takes the *note_object* as an input and creates a real time scrolling plot using Matplotlib.

The process starts with an axis object containing 6 horizontal lines representing strings. Once it receives a note (with string & fret identified), the process writes it to a csv file and appends the fret on to its appropriate sting's position. Every note has an index which indicates its place in time. This index is not an absolute time location, but rather an indicator of order. After a note is added to the plot, the graph scrolls forward in time to fit in into the displayed window, thus achieving a real time scrolling effect. Every time we shift the window, we refresh the plot, using fig.canvas.flush_events(), so that Matplotlib displays the most updated window. As the notes transition in time, we refresh the plot several times to ensure a smooth animation of the moving plot.

The CSV log is implemented by initializing an empty file. As the process receives notes, it appends the notes attributes (index, fret, and string) to the CSV file. However, every time we add a note, the program must open the file, append the notes attributes, and then close the file. We repeat this sequence for every cycle, because if we do not close the CSV file each time we add a note, the file is not updated. We could instead collect the data and only write it on exit, but this is not necessary since performance is not a concern for this process.

Efficiency of this process is within our computational budget. The slowest leg here is refreshing the matplotlib figure, as the library is not optimized for real time use. This can be resolved by using blitting, but we did not invest effort into this as the plotter is the last process and cannot slow down the preceding analysis.

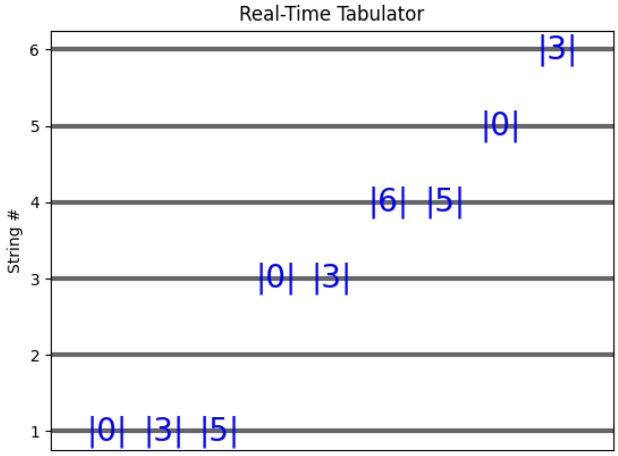


Figure 21 - Plotter Output (Real Time Input)

4 Analysis of results

Overall – our results are good. We are able to perform real time transcription of music and generate accurate tabs. We do run into some unforeseen limitations, the biggest of which is an upper bound on the playing speed. We observe that our program does not like to work with fast music, and operates best when it is given $> 1[s]$ between successive notes. The faster the music becomes, the greater the likelihood of false negative onset detection, which translates to completely missing the note.

To quantitatively evaluate our work, we replace the audio input process with a pseudo-real-time input. This loads a known audio file into memory, and then steps through it, sending frames of 2205 samples down the pipeline. We log the output, and compare the results against known truth. Please note that our software does not attempt to detect the exact location of an onset in time, as it is not necessary for transcription. We are only interested in preserving the order of the notes, which is guaranteed and does not require testing. The data is presented in Table 3, and the log output is provided in Appendix A

.

Table 3 – Results (Pseudo Real Time)

Ground Truth				Software Output					
	Onset Time	ΔOnset	Frequency	Detected (Y/N)	Frequency AUTO	Frequency HPS	Selected Frequency	Rounded Frequency	Error %
0	-		-	Y	1400	5	5	82.41	-
1	1.75	1.75	277	Y	283	285	284	277	0
2	3.44	1.69	246.94	Y	270	179	179	174.61	29
3	5.13	1.69	329.63	Y	329.78	330	329.89	329.63	0
4	7.7	2.57	296.93	Y	296.86	297	296.93	296.93	0
5	8.55	0.85	277.18	Y	284.98	280	282.49	277.18	0
6	9.42	0.87	329.63	Y	321.06	307	314.03	311.13	5.6
7	10.278	0.858	440	Y	425	442	433.83	440	0
8	11.14	0.862	493.88	N	-	-	-	-	-
9	12	0.86	329.63	N	-	-	-	-	-

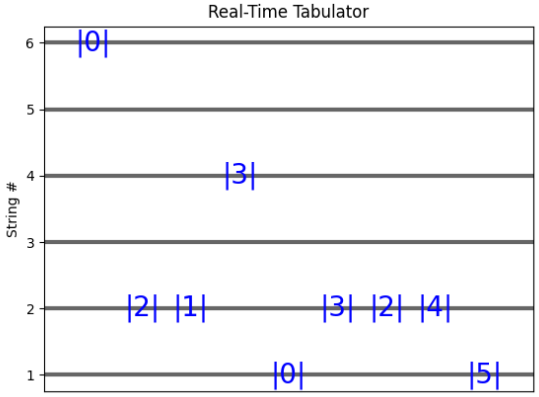


Figure 22 - Testing Result (Pseudo Real Time)

Looking at our results, we can see several problems. First, we observe a false positive at the beginning of transcription. This false positive is found to have a frequency of 5Hz, which suggests a very weak noise component. This is further confirmed by the fact that this onset's index disappears in later calls to onset detection. Since the detected frequency is 5Hz, the closest legal note is E-0, i.e 82.41 Hz, which is plotted as a |0| in Figure 22.

The other problem we have is erroneous selection of pitch detection algorithm results. This happened for note 2, where we chose HPS's output, which had an error of 27.9% when rounded to the nearest legal note. The output of the Autocorrelation algorithm would have had 12% error when rounded to the nearest legal note. In this situation, we should have chosen autocorrelation, but did not. In any case – the autocorrelation output would have resulted in the wrong fret being plotted on the tablature, but unlike the current result, it would be very close to the true coordinates.

A similar situation happened with note 6, where we averaged AUTO and HPS, resulting in a rounded frequency of 311.13, which had a 5.6% error. If we chose only AUTO, then it would round to 329.63, which is the true note that was played.

Finally, we can observe two false negative onsets for notes 8 and 9. The cause of this problem is due to our design of segmentation. As we mentioned in 3.2.5, we define a note slice as the samples indexed from onset N to onset N+1. We can do this, because we are assuming a continuous real time stream of data, so an infinite number of notes existing in the future. In the pseudo real time case, however, the data feed stops. Therefore, for the last onset, it is impossible to generate a note slice since there is no future onset N+1. Therefore, the last note is never displayed.

Why are we missing two onsets and not one, though? This happened because the last true note is very soft in the recording, but our onset detection is tuned for the specific signal generated by our equipment, which is more aggressive. As a result, we have a false-negative onset detection on note 9, which is onset N+1 for note 8, which means that note 8 can never be displayed. Eliminating this issue for the pseudo real time test would require re-tuning of the onset detection parameters. This would be a big change, and the result would not be representative of our project.

It is also worth noting that the performance of our software depends on the conditions. For example, during the project exhibition we observed a higher-than-normal rate of false-positive onsets. This was unexpected, as in our prior testing this issue did not occur. We later discovered that the issue stemmed

from having an external monitor connected to the computer over HDMI. This caused a clicking noise to appear across our guitar signal and be picked up by the segmenter. We speculate that this is a ground loop problem, as providing a path to ground through a capacitor (i.e holding the cable with our hands while standing on the ground) gets rid of the noise.

Now that we have discussed the errors and the reasons behind them, let us evaluate the overall performance of our program. We discard the last note from our analysis because its error is caused by pseudo real time operation and not a mistake in the algorithms.

Table 4 -System Accuracy

Algorithm	**Accuracy**
Onset Detection	78%
Autocorrelation	97.1%
HPS	94.4%
Selected Frequency	95.1%
Selected Frequency (Ignoring outlier Note2)	98.5%

Therefore, the total expected accuracy is $0.78 \cdot 0.951 = 0.742$. We think that with a some optimizations this number can be increased significantly which we will discuss in section 5.

Overall we are very satisfied with the results. The reported recognition rate of our segmentation algorithm (Spectral Flux) in literature is about 74%[12], whereas we achieve 78%. This should be taken with a grain of salt, as our testing sample size is small, but it is still encouraging. In pitch detection, our accuracy is very close to 100%, and the occasional grave errors can be solved with a more developed output selection method.

[12] [2]

5 Conclusions and further work

We embarked on this project with the goal of creating a real time transcription system for electric guitar. We believe that this was successful – our system is able to turn audio into tablature, although not perfectly.

A lot of improvement can be made by optimizing our existing feature set. For example, we established that our system struggles with correctly selecting the "trusted output" in pitch detection, such as in note #2 in Table 3. Modifying the heuristic decision making algorithm (Figure 20) will likely resolve the issue. One modification could be to add a probability based factor, by using the fact that a sequence of notes is likely to belong to the same musical scale. Given a small number of previous notes, we can create several bins of possible future notes, with each bin having a corresponding probability. When receiving the outputs of our pitch algorithms, we can match them to a bin, and thus distinguish the result that is most likely true. Such a fix could theoretically resolve the error in note 2, thus bringing our total pitch accuracy to 98.5% (Table 4).

Another significant point of improvement is onset detection, which is currently our biggest bottleneck, with 78% recognition accuracy. Given the immense importance of tuning for this process, we believe it would be worthwhile to rigorously optimize the parameters using an optimization algorithm, similarly to how Librosa used hyperparameter optimization to choose their defaults. However, instead of using well-recorded data, we could use a dataset of unprocessed high noise data, either by obtaining it online or recording it ourselves. Approaches such as particle swarm optimization might yield considerably improved parameters.

Onset detection can also be improved by implementing better pre-processing. We could employ a noise rejection technique, where we require the guitarist to provide several seconds of silence at the start of the program. Then we can calculate the spectrum of this supposed silence, giving us a description of the inherent noise in the signal. We can then construct a filter to get rid of these spectral components, thus giving a more clear spectrogram, and therefore more precise onset detection.

Further – we could alter the characteristics of onset detection. Specifically – we can replace our STFT with a Mel-Spectrogram, which scales the frequency axis nonlinearly, to conform with human perception of sound, which is logarithmic (Figure 23). By doing this, we introduce an innate "priority" to changes in frequency bands that would be more perceptible to humans, and a penalty to those that we would tend to miss. The potential benefits of this approach are clearly visible in Figure 12, where the Mel-spectrogram novelty

produces cleaner peaks compared to the Constant-Q transform (and we know that constant-Q is comparable to the standard STFT). We can also utilize the *np.median* measure instead of the *np.mean* as it provides a cleaner novelty function.

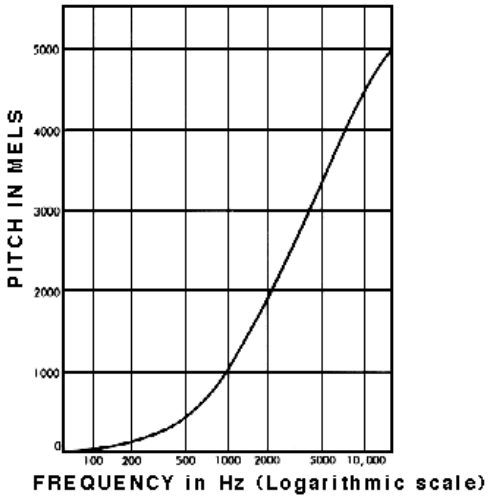


Figure 23 - Mel Scale[13]

On the implementation side, we could definitely improve memory management and computational efficiency. Our program currently uses around 1 [GB] of RAM, and reaches 40% CPU utilization on an Intel Core I7-6700 CPU. Memory use can be drastically reduced by moving some common variables outside of the process scope and into the global scope, so that we do not need to store several copies. Further – we can lower memory consumption by putting an upper bound on the size of our “history arrays”. Regarding performance – a significant improvement would come from replacing *matplotlib* with a more optimized solution, such as *pyqtgraph*, or switching entirely to a text based output. Another possible optimization is to modify the onset detection algorithm to avoid re-calculation of the STFT at every call, since only the last few frames of the data are changing. Finally, we could significantly boost performance by reducing the sampling rate to 22.05 [kHz]. This is still enough to resolve the harmonics, but with half as many samples, meaning smaller FFT, smaller arrays, less algorithm iterations (as a frame now contains more information), which will result in better speed. Finally, a migration of our code to a lower level language, such as C, would produce gains in performance as we circumvent Python’s interpreter overhead.

Even if all the aforementioned improvements are implemented perfectly, and we miraculously obtain 100% accuracy in both onset and pitch detection, our program would still not be competitive with a human transcriber in terms of

[13] [13]

transcription quality. This is because we operate under some limiting assumptions which do not hold in the context of advanced music.

In order to create a truly marketable product, several other features must be implemented. On the MIR side, we want to see added functionality in embellishment detection and polyphonic pitch detection. From a software perspective – the program needs a user interface, start/pause/stop functionality, exporting capabilities to standardized guitar tab formats (such as *.gpx), and presets for different guitar configurations.

We experimented with polyphonic pitch detection already, creating our own algorithm, which worked, albeit with a very low accuracy (around 60%). This algorithm created a list of frequencies present in a note slice by analyzing the spectrogram. It used the understanding that any given note creates a corresponding frequency set $S_i = \{f_i, 2if_i\ 3f_i, 4f_i \dots Nf_i\}$. If we have multiple notes, then the spectrogram would contain all the individual sets, such that $S_{total} = \{S_0,\ S_1,\ S_2 \dots S_k\}_{k>0}^{k\leq 6}$. Suppose we are able to correctly identify one of the many fundamental frequencies present. By knowing the frequency, we know the corresponding set S_i. We can then remove the set S_i from S_{total}, leaving us with a reduced collection of frequencies. We then repeat the process until $S_{total} = \emptyset$, thus obtaining a list of fundamentals. Unfortunately, we were not satisfied with the accuracy of this algorithm, and decided not to include it in the project. Polyphonic pitch detection is extremely difficult, and time considerations prevented us from further developing this aspect. We believe that the above algorithm can be improved significantly, thus adding a polyphonic detection feature to our software.

In addition to the above algorithm, which is theoretically similar to the known iterative subtraction approach, we can also try other methods. A very popular one is the Klapuri algorithm, which was developed by the CTO of Yousician – a company utilizing MIR to provide a guitar training app. Other options include Non-Negative Matrix Factorization (NMF) and frequency domain filter bank (raking)[14].

Embellishment detection is another exciting route of expansion. The purpose of this system would be to detect the ways in which a guitarist “beatifies” the music. This implies recognition of techniques such as vibrato, bending or sliding. All of these techniques create a non-flat pitch contour within a corresponding note slice. For example, perfect vibrato would look like a

[14] [6]

sinusoid[15] in the frequency domain, while a bend would resemble a smoothed trapezoid, and sliding would look like a staircase.

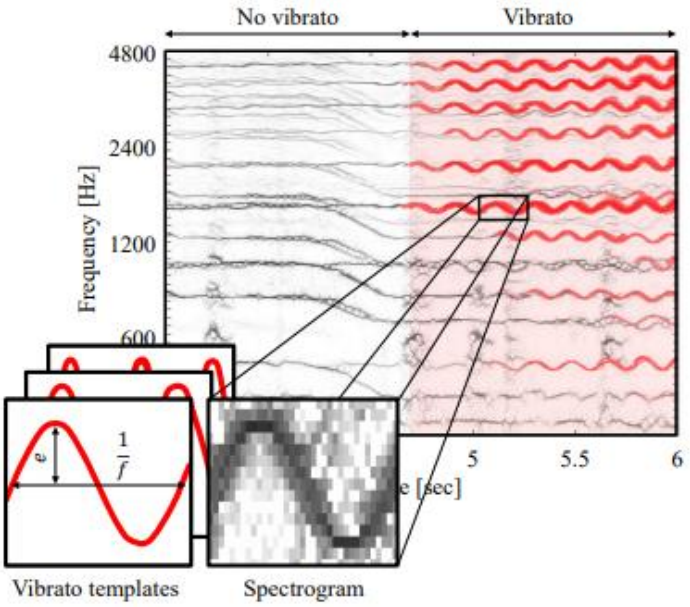


Figure 24 – Vibrato Spectrogram

There exist some approaches to detect embellishments, but the literature is far from complete on this topic. We think a promising idea is to construct three LSTM Recurrent Neural Networks, each responsible for detecting a single embellishment. Again, we experimented with this, but abandoned the idea early on due to a high time cost.

Overall – we are satisfied with what we have achieved, and are excited about the future potential of our project. We firmly believe that automatic transcription is the future. It has the potential to significantly disrupt the established status-quo within the market, while being extremely interesting and yielding insights into related fields such as speech processing or even music generation. We look forward to expanding our knowledge in this field.

[15] [4]

6 References

6.1 Papers:

[1] Bello, Juan Pablo, et al. “A Tutorial on Onset Detection in Music Signals.” *University of Montreal*, IEEE Transactions on Speech and Audio Processing, 2005, www.iro.umontreal.ca/~pift6080/H09/documents/papers/bello_onset_tutorial.pdf.

[2] Bock, Sebastian, et al. “Evaluating the Online Capabilities of Onset Detection Methods.” *ISMIR*, Department of Computational Perception, Johannes Kepler University, 2012, ismir2012.ismir.net/event/papers/049_ISMIR_2012.pdf.

[3] Budhiono, Sofia Shieldy, and Dewa Atmaja Darmawan. “Pitch Transcription of Solo Instrument Tones Using the Autocorrelation Method.” *Jurnal Elektronik Ilmu Komputer Udayana*, vol. 8, no. 3, Feb. 2020, pp. 347–355., ojs.unud.ac.id/index.php/JLK/article/download/53172/33352/.

[4] Driedger, Jonathan, et al. “Template Based Vibrato Analysis of Music Signals.” *Audio Labs*, International Audio Laboratories Erlangen, 2016, www.audiolabs-erlangen.de/content/05-fau/professor/00-mueller/03-publications/2016_DriedgerBEM_VibratoDetection_ISMIR.pdf.

[5] Emerald, Aasha Diana. “Comparative Study of Pitch Estimation Using Harmonic Product Spectrum Derived from DFT, DCT, HAAR and KL Transforms.” *Academic Publications*, International Journal of Pure and Applied Mathematics, 2017, acadpubl.eu/jsi/2017-115-6-7/articles/6/55.pdf.

[6] Goodman, Thomas A, and Ian Batten. “Real-Time Polyphonic Pitch Detection on Acoustic Musical Signals.” *IEEE Xplore*, IEEE, 2018, ieeexplore.ieee.org/document/8642626.

[7] Nonlinear Modeling in Time Domain Numerical Analysis of Stringed Instrument Dynamics - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Attack-Decay-Sustain-Release-ADSR-stages-of-sound-projection-in-time_fig6_310842067

[8] Zhou, Ruohua, and Josh D Reiss. “Music Onset Detection .” *Queen Mary University of London*, Center for Digital Music, 2010, eecs.qmul.ac.uk/~josh/documents/2010/Zhou%20Reiss%20-%20Music%20Onset%20Detection%202010.pdf.

6.2 Websites:

[9] “Acoustic Guitar Notation Guide.” *Acoustic Guitar*, 25 Apr. 2013, acousticguitar.com/acoustic-guitar-notation-guide/.

[10] Barna, Steve, and Matt M. “Whats the Optimal Window Function to Use for Analyzing Real-Time Data Samples?” *Signal Processing Stack Exchange*, dsp.stackexchange.com/questions/586/whats-the-optimal-window-function-to-use-for-analyzing-real-time-data-samples.

[11] “Blackman-Harris Window Family: Spectral Audio Signal Processing.” *DSP Related*, www.dsprelated.com/freebooks/sasp/Blackman_Harris_Window_Family.html.

[12] “Guitar Fretboard: 3 Tips for Learning the Fretboard.” *Yousician*, 21 Apr. 2021, yousician.com/blog/guitar-fretboard-learning-guide.

[13] “MEL.” Edited by Barry Truax, *Handbook for Acoustic Ecology*, Simon Fraser University, 1999, www.sfu.ca/sonic-studio-webdav/handbook/Mel.html.

[14] Tjoa, Steve. “Novelty Functions.” *Music Information Retrieval*, musicinformationretrieval.com/novelty_functions.html.

[15] Tjoa, Steve. “Autocorrelation.” *Music Information Retrieval*, musicinformationretrieval.com/autocorrelation.html.

Appendix A. **Pseudo-Real-Time Test Log**

```
THIS IS A PSEUDO-REAL TIME TEST
DISREGARD PRINTED ONSET TIMES - THEY ARE FOR DEBUGGING USE AND DO NOT
REPRESENT ACTUAL ONSETS
BEGIN INPUT:
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
will have 309 buffers total
**********ONSET DETECTION**********
WE HAVE 3 NEW ELEMENTS
DETECTED ONSETS AT: [2816, 23296, 41216] and in halfsr:
OF WHICH NEW: [ 2816 23296 41216]
**********ONSET DETECTION**********
WE HAVE 1 NEW ELEMENTS
DETECTED ONSETS AT: [2816, 23296, 41216, 66048] and in halfsr:
OF WHICH NEW: [66048]
MSG.SEG LENGTH 24832
**********PITCH DETECTION**********
F-AUTO 1400.3766106961218
F-HPS 5
FREQUENCY CHOSEN: 5
COMPLETE OBJECT HAS:
 ONSET TIME: 2.1192815750638883e-05 FREQUENCY: 5 estimated note 82.41
**********ONSET DETECTION**********
WE HAVE -2 NEW ELEMENTS
DETECTED ONSETS AT: [41472, 76288] and in halfsr:
OF WHICH NEW: []
**********ONSET DETECTION**********
WE HAVE -1 NEW ELEMENTS
DETECTED ONSETS AT: [81408] and in halfsr:
OF WHICH NEW: []
**********ONSET DETECTION**********
WE HAVE 1 NEW ELEMENTS
DETECTED ONSETS AT: [92416, 110080] and in halfsr:
OF WHICH NEW: [110080]
MSG.SEG LENGTH 17664
**********PITCH DETECTION**********
F-AUTO 283.2291313429084
F-HPS 285
FREQUENCY CHOSEN: 284.1145656714542
COMPLETE OBJECT HAS:
 ONSET TIME: 4.75192949439791e-05 FREQUENCY: 284.1145656714542 estimated
note 277.18
**********ONSET DETECTION**********
WE HAVE 1 NEW ELEMENTS
DETECTED ONSETS AT: [92416, 110080, 178432] and in halfsr:
OF WHICH NEW: [178432]
MSG.SEG LENGTH 68352
**********PITCH DETECTION**********
F-AUTO 260.8758292910687
F-HPS 266
FREQUENCY CHOSEN: 263.43791464553436
COMPLETE OBJECT HAS:
 ONSET TIME: 5.6601930265681486e-05 FREQUENCY: 263.43791464553436 estimated
note 261.63
**********ONSET DETECTION**********
WE HAVE 1 NEW ELEMENTS
DETECTED ONSETS AT: [92416, 110080, 189440, 224768] and in halfsr:
```

```
OF WHICH NEW: [224768]
MSG.SEG LENGTH 35328
**********PITCH DETECTION**********
F-AUTO 270.5391078980445
F-HPS 179
FREQUENCY CHOSEN: 179
COMPLETE OBJECT HAS:
 ONSET TIME: 9.740797301535883e-05 FREQUENCY: 179 estimated note 174.61
**********ONSET DETECTION**********
WE HAVE 1 NEW ELEMENTS
DETECTED ONSETS AT: [92416, 110080, 189440, 235776, 306432] and in halfsr:
OF WHICH NEW: [306432]
MSG.SEG LENGTH 70656
**********PITCH DETECTION**********
F-AUTO 329.7889913618215
F-HPS 330
FREQUENCY CHOSEN: 329.8944956809107
COMPLETE OBJECT HAS:
 ONSET TIME: 0.00012123343668533173 FREQUENCY: 329.8944956809107 estimated
note 329.63
**********ONSET DETECTION**********
WE HAVE 1 NEW ELEMENTS
DETECTED ONSETS AT: [92416, 110080, 189440, 235776, 327936, 350464] and in
halfsr:
OF WHICH NEW: [350464]
MSG.SEG LENGTH 22528
**********PITCH DETECTION**********
F-AUTO 296.86301465502606
F-HPS 297
FREQUENCY CHOSEN: 296.93150732751303
COMPLETE OBJECT HAS:
 ONSET TIME: 0.00016862109923334413 FREQUENCY: 296.93150732751303 estimated
note 293.66
**********ONSET DETECTION**********
WE HAVE 1 NEW ELEMENTS
DETECTED ONSETS AT: [92416, 110080, 189440, 235776, 327936, 350464, 378880]
and in halfsr:
OF WHICH NEW: [378880]
MSG.SEG LENGTH 28416
**********PITCH DETECTION**********
F-AUTO 284.98669655084433
F-HPS 280
FREQUENCY CHOSEN: 282.49334827542214
COMPLETE OBJECT HAS:
 ONSET TIME: 0.00018020475007841382 FREQUENCY: 282.49334827542214 estimated
note 277.18
**********ONSET DETECTION**********
WE HAVE 1 NEW ELEMENTS
DETECTED ONSETS AT: [92416, 110080, 189440, 235776, 327936, 350464, 390144,
405504] and in halfsr:
OF WHICH NEW: [405504]
MSG.SEG LENGTH 15360
**********PITCH DETECTION**********
F-AUTO 321.0612947714348
F-HPS 307
FREQUENCY CHOSEN: 314.03064738571743
COMPLETE OBJECT HAS:
 ONSET TIME: 0.00020060777145325252 FREQUENCY: 314.03064738571743 estimated
note 311.13
```

```
**********ONSET DETECTION**********
WE HAVE 1 NEW ELEMENTS
DETECTED ONSETS AT: [92416, 110080, 189440, 235776, 327936, 350464, 390144,
412160, 453888] and in halfsr:
OF WHICH NEW: [453888]
MSG.SEG LENGTH 41728
**********PITCH DETECTION**********
F-AUTO 425.6627394901316
F-HPS 442
FREQUENCY CHOSEN: 433.8313697450658
COMPLETE OBJECT HAS:
 ONSET TIME: 0.0002119281575063888 FREQUENCY: 433.8313697450658 estimated
note 440
LAST FRAME HAS BEEN SENT
************************************
```