# Automatic Music Transcription Implementation for Guitar

Shaurya Chopra 170100108
Rishabh Dahale 17D070008

Guide: Prof. Preeti Rao

May 7, 2021


Department of Electrical Engineering
IIT Bombay

# Abstract

This study is an attempt to solve a problem in Automatic Music Transcription (AMT) - namely, the problem of chord detection (a subset of which is, of course, note detection).
We discuss the common approaches used in the literature, and implement the approach of multi-pitch estimation as discussed by the authors of [1], since it works on a straightforward maximum likelihood estimation approach, and offers performance superior to traditional methods. All equations and ideas discussed are adapted from the aforementioned paper. A step-by-step discussion of the ideas, followed by a guide for our implementation is provided. We conclude by providing results of our implementation, run on the Fraunhofer IDMT dataset.
This document is a detailed report encompassing the ideas behind AMT, review of literature, detailed discussion of the method of multipitch estimation, results and future work. The appendix serves as a manual for our Python implementation, with discussion of all the individual modules and how they come together to form the transcription model.

# List of Abbreviations

| | |
|---|---|
| STFT | Short-time Fourier Transform |
| NMF | Non-negative Matrix Factorization |
| HPS | Harmonic Product Spectrum |
| HMM | Hidden Markov Model |
| PCP | Pitch Class Profile |
| N | Polyphony of the audio |
| M | Maximum possible polyphony |
| T | threshold for polyphony estimation |
| $F_0$ | Fundamental Frequency |
| $\theta$ | Set of fundamental frequencies predicted in estimation |
| $C$ | candidate set of fundamental frequencies |
| $M$ | Maximum possible polyphony, or `MAX_POLYPHONY` |
| $f_k$ | frequency(in MIDI) of $K^{th}$ observed peak |
| $h_k$ | harmonic number (approx) of $K^{th}$ observed peak |
| $f_k$ | log-amplitude of $K^{th}$ observed peak |
| $d_k$ | distance(in MIDI) of $K^{th}$ observed peak from the corresponding harmonic |
| $p(.), P(.)$ | probability measure |

# Contents

# Automatic Music Transcription

## 1 Introduction

Automatic Music Transcription (AMT) is the problem of converting acoustic signals into musical notation. It is an interesting and challenging problem from the field of Music Information Retrieval(MIR). Applications of an AMT system include, but aren't limited to an online music education platform, music similarity identification and cover song detection - especially across different languages for the lyrics, and chord-progression based recommender systems, that can automatically generate accompaniment.
A comprehensive and reliable AMT system is yet to be realized, owing to the complicated nature of both musical information itself - pitch, timbre, chords, melodies, onsets and offsets, and even instrument specific information and playing styles - to the difficulty in extracting them from a single acoustic signal. AMT encompasses solving various subproblems - onset and offset detection, multipitch estimation, and multi-class classification. In this work, we shall focus on the task of chord detection for guitar music. We first discuss traditional methods used to approach this problem, and then dive into the method of multipitch estimation, since an estimate of the various fundamental frequencies (corresponding to semitones of an instrument) in each time frame gives us the basic information about the notes (and hence chord) being played at each instant.

## 2 Basics: Notes and Chords

We first review some basics about musical notation, with some emphasis on guitar.

- A note is simply the harmonics of given fundamental frequency $F_0$
- For most instruments, including the guitar, notes are equitempered, i.e. the $F_0$ of notes grows as a geometric progression of common ratio $2^{\frac{1}{12}}$
- Gap between successive notes is called a *semitone*
- Every $12^{th}$ note is labelled as the same note in a different *octave*, since they sound like the same note at a higher pitch
- An octave is thus defined as a set of 12 successive notes, typically starting at the note $C$ in western notation.
- A musical chord is defined as a special set of superimposed notes, i.e. played together

- The succession of notes and chords over time forms the core in a piece of music, just like words of a language

We note that an acoustic music signal is expected to have a very specific harmonic structure, containing fundamentals and overtones of some finite set of notes at each time instant. Thus, one way to think about the transcription problem is finding out exactly which notes and chords are being played at each time instant.

## 2.1 Chords in a Guitar

A guitar has a very unique structure. It comprises of six strings of different thickness, labelled 1-6 with 1 being the thinnest, *tuned* by convention to *E,B,G,D,A,E*. Playing notes is straightforward. We press down on a *fret* and pluck the string; each fret advances the open string note by a semitone.

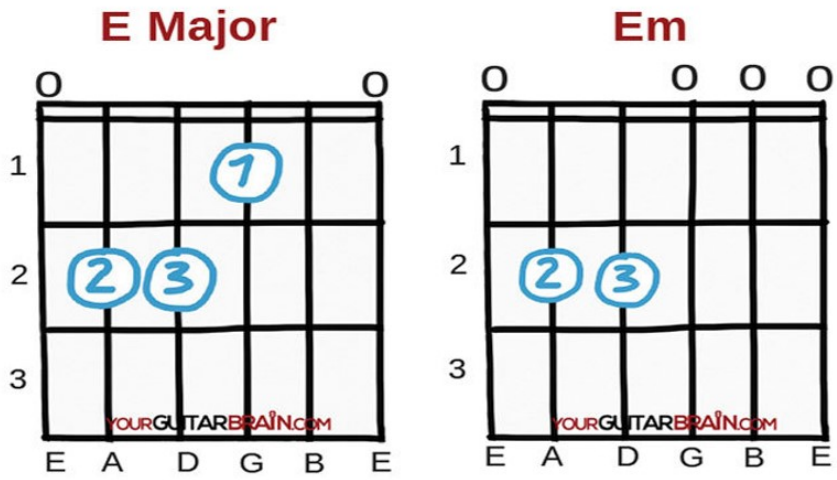



Figure .1: Guitar Basics

Note that standard chords - majors and minors, are a superposition of three notes. However, a guitar chord is always a stroke to all six strings simultaneously, with some notes repeated.

# 3 Literature Review

We now give a brief account of previous, and popular work on this problem, with the goal of transcribing polyphonic music with chords and notes:

1. Non-negative Matrix Factorization (NMF)
2. Harmonic Product Spectrum (HPS)
3. Template-based Matching
4. Hidden Markov Models
5. Multi-pitch Estimation

Although all these classes of methods carry out unique operations, all of them work with the short-time Fourier transform of the signal, a standard approach for audio processing tasks. That is, we chop up acoustic signals into frames with a given window and hop length, and process one at a time. Template matching and HMM are similar in that their approach implicity build on the HPS. The latter, however, along with NMF and Multipitch estimation are data-driven.

## 3.1 Non-negative Matrix Factorization

NMF is a matrix algorithm, fist proposed by the authors of [2], and used for the AMT problem by the authors of [3].
The idea is the following: factor the STFT matrix $X = X(n,k) \in \mathbb{R}^{M\times N}$ as $X \approx WH$ where $W \in \mathbb{R}^{M\times R}$ and $H \in \mathbb{R}^{R\times N}$. $R$ is a hyperparameter. The approximation is in the sense of Frobenius norm.
NMF has the following characteristic: Rows of $H$ summarize rows of $X$ and columns of $W$summarize columns of $X$. Thus, when $X$ is the STFT, columns of $W$ form a "spectral basis" of the notes in the frame, while rows of $H$ contain the temporal information (when the note starts and ends within the frame, for instance).

### Drawbacks

While this is a light, data-driven approach, it has a significant drawback in that it reliably performs transcriptions only for static harmonic profiles (notes playing for a long time).

## 3.2 Harmonic Product Spectrum (HPS)

The HPS measures the maximum coincidence for harmonics for each spectral frames. HPS of a spectrum $X(\omega)$ is simply a superposition (product) of $K$ decimated versions of $X(\omega)$, and can be calculated as:

$$\hat{Y}(\omega) = \prod_{k=1}^{K} |X(k\omega)| \tag{.1}$$

The idea behind HPS is that it only retains harmonic information by collapsing overtones onto the fundamental, getting rid of everything else. Thus, in principle, the HPS is nothing but peaks in the spectrum, at locations of the fundamental frequencies present in the frame. The polyphony, too, is nothing but the number of peaks in the HPS.

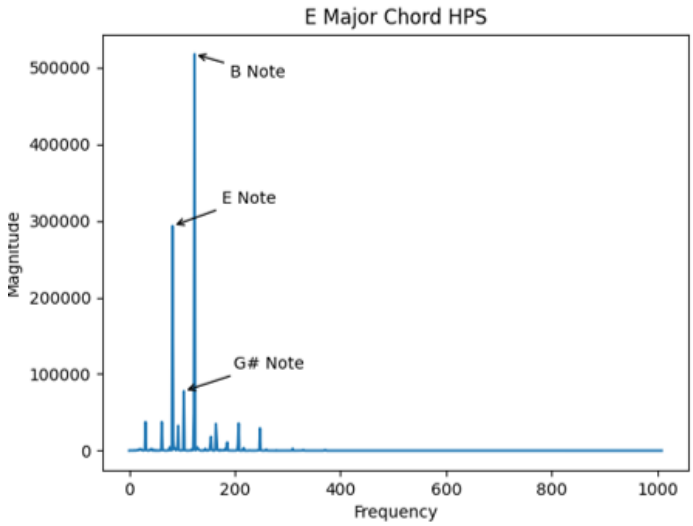



Figure .2: HPS of E Major Chord (E + $G^{\#}$ + B)

For speech related tasks, $k$ is varied from 1 to $K$, the latter being a hyperparameter. In music, as octaves differ in frequency by a factor of 2, harmonics not at the power of 2 should not contribute energy to the base note. Hence the above equation is modified as follows:

$$Y(\omega) = \prod_{k=1}^{K} |X(2^k \omega)| \tag{.2}$$

### Drawbacks

Although this representation may look simple and promising, it has some shortcomings with its representation.

Figure .2 shows the HPS calculated for a frame of E Major chord which consists of notes E, $G^{\#}$ and B. Due to variation in playing the notes, it can be seen that some notes are much more emphasized than others. Moreover, some spurious peaks can be seen which can easily hinder with the transcription task, especially polyphony estimation. In addition, while computing the HPS, the octave information is lost as higher octaves are collapsed into lower ones making the transcription octave free.

## 3.3 Template-based Matching

This class of methods works on the following idea: define a feature vector that succinctly captures all the information relevant to detecting a chord. Once we have an appropriate feature vector, we can define classes corresponding to each type of chord, create pre-defined templates, and then match a probe vector to one of these templates in order to transcribe the music.

### Chromagrams

The most popular feature vector for chord recognition is called a chromagram, also referred to as a Pitch Class Profile (PCP)[4]. It builds on the idea of the harmonic product spectrum.
The representation proposed by the authors of [4] is the *enhanced pitch class profile*, a novel

improvement over the traditional PCP feature vector[5].
This feature vector uses the harmonic product spectrum, and creates (a multiple of) 12 *bins*, each corresponding to a note (if 36 bins, 3 would correspond to a semitone). The energy in $k^{th}$ bin is total energy in $2^{\frac{k}{12}}$ to $2^{\frac{k+1}{12}}$.
Note that this feature vector is identical to the Constant-Q Transform of the spectrum, as explained in [4]. However, we arrive at the same feature vector, building on the HPS for ease of understanding.

**Drawbacks**

Although this is an instrument independent method, and the templates are simply one-hot vectors of the chord with ones at the constituent notes' bins, noise can significantly affect the results.

### 3.4 Hidden Markov Models

These are analogous to classical speech recognition models[6]. The idea[7] is to form a statistical model of a collection of random vectors $O_1, ..., O_T$ representing observations, emitted by states $q_1, q_2, ....$
The states form an underlying (hidden) markov chain. The observations are nothing but PCP vectors - that is, the training input signals chopped up into frames, and converted to PCP vectors. The markov model, with states belonging to chord classes, are trained on these using Expectation-Maximization.
Chord recognition is done using the Viterbi algorithm, as in the case of speech recognition models.

### 3.5 Multi-pitch Estimation

The other main class of algorithms, and the class we focus on, is that of multi-pitch estimation. These attempt to solve the general problem of finding the various pitches present in an audio clip in any given frame, without knowing the structure or even the polyphony (no. of pitches). Once we know each of the fundamental frequencies contributing to the harmonic structure of each frame of a music signal, we can easily identify the precise notes and chords being played.
We now discuss in detail the the algorithm proposed by the authors in [1], which shall be our focus in this work.

## 4 Multipitch Estimation Overview

The main idea behind detecting chords/notes in a musical piece is detecting, at any given moment, exactly which pitches occur. Note that notes are simply sounds from a specific set of fundamental frequencies in a geometric progression of $2^{\frac{1}{12}}$. These notes are given logarithmically increasing numbers, called MIDI numbers, where a spacing of 1 in the MIDI scale is referred to as a *semitone*. Chords are nothing but combinations of notes following certain rules.
Thus, even without knowing explicitly the rules that determine chords, it is in principle sufficient to know the (MIDI numbers of) *fundamental frequency* $F_0$ each note playing at a given time.
We now discuss in detail the multipitch estimation method proposed in [1]

# 5 The STFT

In order to process audio data, we follow the standard approach of computing the *Short-time Fourier Transform* of the audio signal.

The entire signal is divided into time slices of a pre-decided size. For this, the audio signal, sampled at `SAMPLING_RATE` samples/sec, is multiplied with a moving window of fixed width (we call this parameter `WIN_LEN`) and fixed overlap `HOP_LEN`. Then, the magnitude spectrum of each frame is computed, and finally, normalized by energy.

$$\hat{X}(n,k) = \sum_{m=-\infty}^{\infty} x[m]w[n-m]e^{-jk\omega_0 m} \tag{.3}$$

$$X(n,k) = \frac{|\hat{X}(n,k)|}{\sum_n |\hat{X}(n,k)|^2} \tag{.4}$$

where $n$ refers to the frame index/position and $k$ runs over the frequency bins of the STFT.

# 6 Maximum Likelihood Estimation

The authors[1] follow the following approach: set up a probabilistic model parametrized by fundamental frequencies, that shall then be maximized over the set of possible $F_0$ and the fundamental frequencies that maximize this function over a frame, are our *predictions*.

A detailed discussion of this novel likelihood function proposed by the authors is given in the subsequent sections.

## 6.1 Greedy Search

Once the likelihood function is known (trained), instead of a brute force search, predictions are found by a *greedy* iterative strategy over a *candidate set* of fundamental frequencies.

The idea is simple - maximization of likelihood is done over the candidate set, and possible $F_0$'s added one-by-one.

## 6.2 The Candidate Set

The *missing fundamental* situation (audio having overtones of a fundamental frequency but not the fundamental itself) is not considered.

Thus, we restrict our search space to the peaks seen in the frame spectrum. The candidate set $C$ is defined to be those values within `DEVIATION` $\pm 3\%$ of *observed peaks* in the frame spectrum (how these peaks are obtained will be discussed subsequently) with a step size `STEP` of 1%. Note that this is done in order to account for improper tuning of instruments, and its effect aggregated over all observed peaks.

## 6.3 Algorithm

Thus, the basic idea is the following: define a likelihood function that depends on where the peaks occur in the frame spectrum. Then greedily iterate over a set of candidate fundamental frequencies and select your candidates. Then, use this likelihood function to figure out the polyphony of this

frame, say N, and then return the N best candidates as the prediction.
A high level pseudocode of the algorithm is given below:

**Algorithm 1** High Level View of Multipitch estimation

```
for energy normalized frame in audio do
    find all peaks with amplitudes in frame spectrum
    C ← candidate F_0
    θ ← ∅
    for N=1 to MaxPolyphony do
        for Each F_0 ∈ C do
            evaluate likelihood on θ ⋃ {F_0}
            F_0^* = F_0 with max likelihood
        end forθ ← θ ⋃ {F_0^*}
    end for
    Estimate Polyphony, N Return first N elements of θ
end for
for each frame of audio do
    refine F_0 estimates using neighbouring frames
end for
```

# 7 The Likelihood Function

We now discuss in detail the novel likelihood function proposed by the authors[1]. Implementation of each of these will be discussed subsequently.
The observed power spectrum is modelled in terms of two components - the *peak region* and the *non-peak region*. The peak region is defined as those intervals of the frequency axis which are within a distance $d$ of an *observed peak* (details of how peaks are observed can be found in 3.2. In the implementation, we call this parameter `DEVIATION` and is set to $\pm 3\%$ (or a musical *quarter-tone*.
The *non-peak region* is defined as the complement of the peak region.
With these definitions, we set up a parameter estimation problem as follows:

$$\hat{\theta} = \arg\max_{\theta \in \Theta} P(O|\theta) \tag{.5}$$

where $\theta = \{F_0^1, ..., F_0^N\}$, the likelihood function $P(O|\theta)$ is broken down as follows:

$$P(O|\theta) = P_{peakregion}(O|\theta) \times P_{non-peakregion}(O|\theta) \tag{.6}$$

We now look at each of these terms individually.

## 7.1 Peak Region Likelihood

The peak region likelihood represents the probability of occurrence of peaks given your current predictions $\theta$. Now, if a peak occurs at location $f_k$ with log-amplitude $a_k$, then

$$P_{peakregion}(O|\theta) = p(f_1, a_1, ..., f_K, a_K|\theta) \approx \prod_{k=1}^{K} p(f_k, a_k|\theta) \tag{.7}$$

assuming conditional independence of peaks given $\theta$.
Ignoring the possibility of *spurious peaks* that is, each observed peak in the dataset is within a musical semitone of a harmonic of the ground truth fundamental frequencies, we can assume that each peak is associated with only one $F_0 \in \theta$. Thus,

$$p(f_k, a_k|\theta) \approx \max_{F_0 \in \theta} p(f_k, a_k|F_0) = p(f_k|F_0) \times p(a_k|f_k, F_0) \tag{.8}$$

Now if $h_k = \left[2^{\frac{f_k - F0}{12}}\right]$ is the harmonic of $F_0$ closest to $f_k$, ([.] denotes rounding to nearest integer; frequencies are always expressed in MIDI) and $d_k = f_k - F_0 - 12log_2 h_k$ is the distance of the observed peak from the nearest harmonic in MIDI, then

$$p(f_k|F_0) = p(d_k|F_0) \approx p(d_k) \tag{.9}$$

where the approximation is justified by the authors by measuring correlation between $d_k$ and $F_0$ and finding it to be small.
$p(d_k)$ is learned from the dataset and modelled as a GMM.
Finally, we write

$$p(a_k|f_k, F_0) = \frac{p(a_k, f_k, h_k)}{p(f_k, h_k)} = \frac{p(a_k, f_k|h_k)\cancel{p(h_k)}}{p(f_k|h_k)\cancel{p(h_k)}} \tag{.10}$$

Note that the authors use models for $p(a_k, f_k, h_k)$ and $p(f_k, h_k)$ directly. However, we break these down even further for the following reason: $a_k$ and $f_k$ are real valued, while $h_k$ is a discrete harmonic number. The authors use the Parzen window method to train the PDFs, while we use GMMs (more on this in the implementation section). Using the latter on both continuous and discrete data will not capture the correlations correctly.
Thus, as for $p(d_k)$, GMMs are trained from the dataset for $p(a_k, f_k|h_k)$ and $p(f_k|h_k)$, that is, one GMM for each possible harmonic $h_k$.

## 7.2 Non-peak Region Likelihood

The non-peak region likelihood captures the probability of not observing a peak in non-peak region, given the set of assumed $F_0$. To model this, we define

$$P_{non-peakregion}(O|\theta) = \prod_{F_0 \in \theta} \prod_{h} 1 - P(e_h = 1|F_0) \tag{.11}$$

where $e_h$ is an indicator function, that takes value 1 when the $h^{th}$ harmonic of $F_0$ is observed in the non-peak region. These probabilites are once again trained over the dataset.
A pictorial summary of the way we break down the likelihood function is as follows (note that the boxes in green are the ones we "train"):

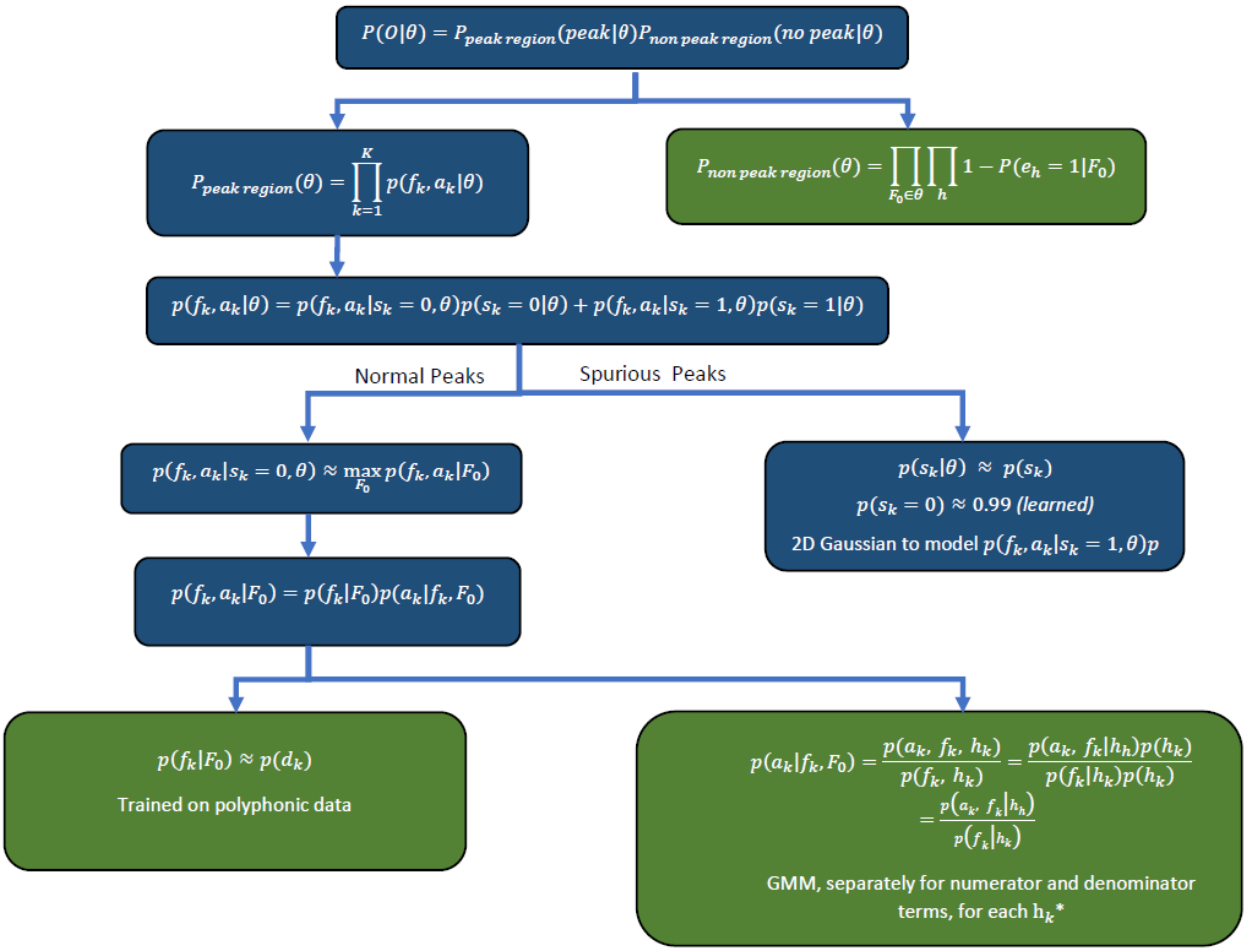



Figure .3: Breakdown of Likelihood Function

We conclude the discussion of the likelihood function by noting that unlike other types of models which rely on carefully chosen heuristics and functions to pick out specific information(for instance, feature-vector based ones) the learned distributions implicitly capture information about the instrument like its hidden envelope, timbre information and so on.

# 8 Polyphony Estimation

The above maximum likelihood estimation problem gives us a set of "predicted" fundamental frequencies that best describe a frame. However, we also need to estimate the polyphony of the frame, namely, how many fundamental frequencies are actually there in the frame.
Polyphony estimation is closely related to *overfitting* - we want the least number of fundamental frequencies that best represent the frame spectrum. In the algorithm discussed so far, however, we have only an upper limit on the size of $\theta$
**Note:** In the subproblem of chord detection of a guitar, although six strings are being played at a time, in standard major and minor chords, you actually only have three "independent" fundamental frequencies, with other strings usually repeating a note. Therefore, we expect the predicted polyphony to be 3, not 6 in guitar chord estimation.

Since adding each $F_0$ to $\theta$ always (in theory) increases likelihood, we set a threshold on the change:

$$N = \min_{1 \leq n \leq M} n \text{s.t.} \Delta(n) \geq T \times \Delta(M) \tag{.12}$$

where $\Delta(n) = ln\mathcal{L}(\hat{\theta}^n) - ln\mathcal{L}(\hat{\theta}^1)$, $\mathcal{L}$ denotes the likelihood function and the number in the superscript denotes the size of $\theta$. $T$ is a learned threshold.

# 9 Results

## 9.1 Performance Metrics

In order to quantify performance, we first note that in the task of multipitch estimation, the following errors are possible [8]:

- **"miss":** an $F_0$ present in the ground truth may be missed altogether
- **"sub":** a ground truth $F_0$ may be substituted for another in the prediction
- **"fa":** (false alarm) an $F_0$ present in the prediction that isn't present in the ground truth

To this end, we define error metrics specifically aimed at identifying these scenarios, in addition to a **total** error metric.
First define

- $N_{ref}(t)$ = number of $F_0$ in ground truth of frame $t$
- $N_{sys}(t)$ = number of $F_0$ in predicted by the system for frame $t$
- $N_{corr}(t)$ = number of $F_0$ correctly identified compared to the ground truth, for frame $t$

With this notation, we define the following error metrics ($T$ is the total number of frames):

$$E_{tot} = \frac{\sum_{t=1}^{T} \max\left(N_{ref}(t), N_{sys}(t)\right) - N_{corr}(t)}{\sum_{t=1}^{T} N_{ref}(t)} \tag{.13}$$

$$E_{sub} = \frac{\sum_{t=1}^{T} \min\left(N_{ref}(t), N_{sys}(t)\right) - N_{corr}(t)}{\sum_{t=1}^{T} N_{ref}(t)} \tag{.14}$$

$$E_{miss} = \frac{\sum_{t=1}^{T} \max\left(N_{ref}(t), 0\right) - N_{sys}(t)}{\sum_{t=1}^{T} N_{ref}(t)} \tag{.15}$$

$$E_{fa} = \frac{\sum_{t=1}^{T} \max\left(0, N_{sys}(t)\right) - N_{ref}(t)}{\sum_{t=1}^{T} N_{ref}(t)} \tag{.16}$$

Finally, we define the usual metrics for precision, recall and accuracy as follows. Let

- $TP(t)$ =no. of true positives (overlap with ground truth) in frame $t$
- $FP(t)$ =no. of false positives (detected, but not in ground truth) in frame $t$
- $FN(t)$ =no. of false negatives (in ground truth, but not detected) in frame $t$

Thus,

$$\text{precision} = \frac{\sum_{t=1}^{T} TP(t)}{\sum_{t=1}^{T} TP(t) + FP(t)} \tag{.17}$$

$$\text{recall} = \frac{\sum_{t=1}^{T} TP(t)}{\sum_{t=1}^{T} TP(t) + FN(t)} \tag{.18}$$

$$\text{accuracy} = \frac{\sum_{t=1}^{T} TP(t)}{\sum_{t=1}^{T} TP(t) + FP(t) + FN(t)} \tag{.19}$$

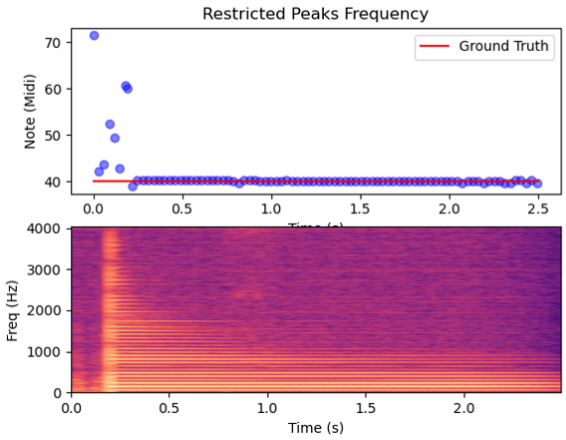
**Octave Folding**

We observed that for a significant proportion of time, the prediction of the note(s) currently playing falls within an "octave" error. That is, the predicted note is exactly an octave (12 MIDI notes) away from the ground truth. This may happen primarily because of the relative strength of harmonic peaks detected, and the greedy search strategy (a higher octave of the same note may satisfactorily account for its peaks observed).
Thus, we tune the above error metrics to ignore octaves (done by taking modulo 12 on the predicted and ground truth MIDI numbers) and recompute those as well. Note that for the purpose of rudimentary chord detection, we do not require octaves, merely the set of notes.

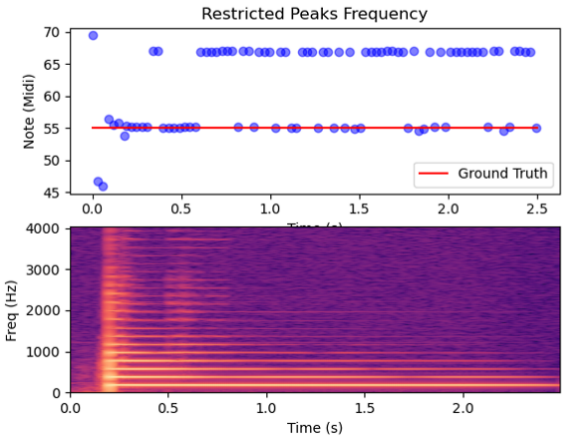
## 9.2 Frame Level Results

In order to get a feel of the predictions, we run this on some test files, and compare with the ground truth. We show the prediction for 25 contiguous frames, and contrast these with the ground truth. The STFT is plotted for reference.

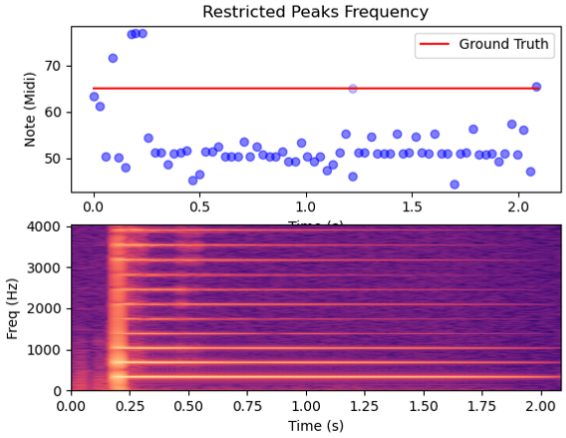
### Monophonic Results



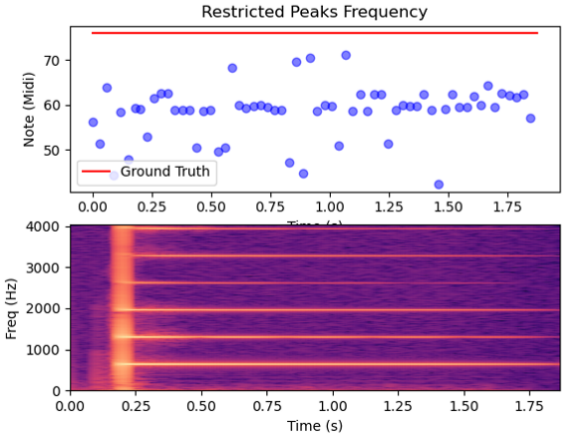

(a) Prediction for MIDI 40




(b) Prediction for MIDI 55




(c) Prediction for MIDI 65



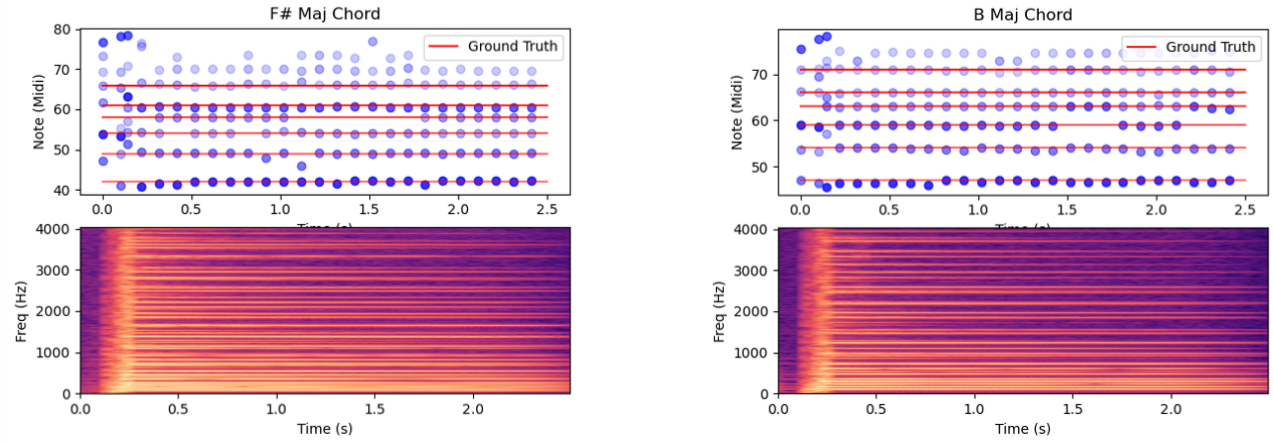

(d) Prediction for MIDI 76

Figure .4: Predictions for Various Monophonic Files

We observe that for the lower MIDI numbers, the prediction overlaps with the ground truth (red line), while for the higher ones, we observe octave errors.

### Polyphonic Results

For polyphonic files, we consider guitar clips of two chords, and plot as before.




(a) Prediction for F Sharp Maj Chord

(b) Prediction for B Maj Chord

Figure .5: Predictions for Polyphonic Files

We observe that the predicted polyphony is actually three, not six, as noted in the previous chapter. The predictions agree with the ground truth in the relevant (i.e. the ones the algorithm picked) octaves.

## 9.3 File Level Results

Since running the prediction algorithm on the entire dataset is computationally very expensive, we first run the algorithm on a single file and observe the values of the previously defined error metrics, with and without octave folding. The file under consideration is `1-E1-Major 00.wav`, which has six notes playing in one stroke of the E major chord, with three detected.

| | w/o octave folding | w/ octave folding |
|---|---|---|
| $E_{tot}$ | 0.81 | 0.39 |
| $E_{sub}$ | 0.27 | 0.21 |
| $E_{miss}$ | 0.53 | 0.17 |
| $E_{fa}$ | -.053 | -0.17 |
| Precision | 0.42 | 0.74 |
| Recall | 0.19 | 0.61 |
| Accuracy | 0.15 | 0.50 |

Table .1: Results on `1-E1-Major 00.wav`

## 9.4 Preliminary Batch Level Results

An initial run on the entire batch of files (on entire dataset 1 and 2) yielded the following results:

| | w/o octave folding | w/ octave folding |
|---|---|---|
| Precision | 0.22 | 0.37 |
| Recall | 0.39 | 0.69 |
| Accuracy | 0.16 | 0.32 |

Table .2: Preliminary Results

We note that the performance improves considerably by considering octave folding.

# 10 Future Work

We conclude by listing out the list of possible future steps:

- Onset detection:
  - combine various state of the art methods in order to distinguish between onset of a note, a chord and the constituent notes of a chord
- Post Processing
  - Use estimates of neighbouring frames to refine the estimates
  - Couple with onset detection to limit predictions to intervals strictly between onsets (and offsets)
- Training and testing for an instrument other than guitar (note that the models themselves do not assume instrument specific information; we may, however see different results particularly in the error metrics)
- Automatic Accompaniment Generation Problem:
  - use the transcription of acoustic signals into chords and notes to feed into a Reinforcement Learning based model, that generates harmonic and rhythmic support

# Bibliography

[1] Z. Duan, B. Pardo, and C. 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.

[2] Daniel Lee and Hyunjune Seung. Algorithms for non-negative matrix factorization. *Adv. Neural Inform. Process. Syst.*, 13, 02 2001.

[3] P. Smaragdis and J.C. Brown. Non-negative matrix factorization for polyphonic music transcription. In *2003 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (IEEE Cat. No.03TH8684)*, pages 177–180, 2003.

[4] Kyogu Lee. Automatic chord recognition from audio using enhanced pitch class profile. In *ICMC*. Citeseer, 2006.

[5] Takuya Fujishima. Realtime chord recognition of musical sound: a system using common lisp music. In *ICMC*, 1999.

[6] Ben Gold, Nelson Morgan, Dan Ellis, and Douglas O'Shaughnessy. Speech and audio signal processing: Processing and perception of speech and music, second edition. *The Journal of the Acoustical Society of America*, 132:1861–2, 09 2012.

[7] A. Sheh and D. Ellis. Chord segmentation and recognition using em-trained hidden markov models. In *ISMIR*, 2003.

[8] M. Bay, A. F. Ehmann, and J. S. Downie. Evaluation of multiple-f0 estimation and tracking systems. 2009.

# Appendix A : Manual

# Manual

We provide a step-by-step discussion of each of the modules implemented along with a brief discussion of the associated ideas. We implement in Python. Note that files referred to are denoted in typewriter font - for instance `foo.py`, and so are global parameters like `SAMPLING_RATE` defined in `config.json`.

## 1 Dataset Details

We use the Fraunhofer IDMT dataset. We use IDMT-SMT-GUITAR_V2 which consists of 4 sub-datasets, of which we use Datasets 1 and 2.
The samples are stored as WAV files, with annotation provided in XML format with the following *note-event parameters* (the audio file is broken down into events of individual notes being played :

1. **pitch**: MIDI pitch value of the note
2. **onsetSec**: onset time of the note in seconds
3. **offsetSec**: offset time of the note in seconds
4. fretNumber: fret number of the guitar for the note. (0 for open string)
5. stringNumber: string number of guitar for note. 1 stands for lowest string
6. excitationStyle: plucking style of the note (one of 3)
7. expressionStyle: expression style used (one of six)

### 1.1 Dataset 1

Dataset 1 consists of single note and chord recordings. Each recording contains one pluck. Single notes are played from the $0^{th}$ two the $12^{th}$ fret. These add up to 312 files. In addition, it has two chords, A and E, both major and minor of each at different barrés. This accounts for 44 files. Note that unless stated otherwise, *a chord will play 6 notes simultaneously.*

| | |
|---|---|
| No. of Single Note files | 312 |
| No. of Chord Files | 44 |
| Total Duration (minutes) | ≈14 |

Table A.1: Fraunhofer IDMT Dataset 1 Details

## 1.2 Dataset 2

Dataset 2 consists of 12 licks (7 monophonic, 5 polyphonic) sampled at 44.1kHz (a lick is a stock pattern/tune; durations were of the order of 10 seconds). They were recorded by the same guitarist, playing three different guitars with different excitation and expression styles at standard tuning. The dataset also contains some *initialization files*, which are prototype versions for prototype versions for each playing technique as well as a plain version of every note on the fretboard for each guitar and each string from fret zero (empty string) to fret 20

### Guitars Used

The guitars used are the Fender Stratocaster ("FS"), Gibson Les Paul ("LP") and Aristides 010 ("AR"), all at standard tuning (E2, A2, D3, G3, B3, E4).

### Excitation Styles

There are three *excitation/plucking styles* - "picked", "fingered" and "muted".

### Expression Styles

In addition, we have six *expression styles* - normal(no), vibrato(v), slide(s), bending(b), harmonics(h), dead notes(dn).

### Lage

Finally, we have another parameter, *lage*, the German word for position, which denotes the possibility of playing the same pitch at 'x' higher possible fret position (we know that in a guitar, multiple string and fret pairs may map to the same MIDI note).
The table A.1 below (provided by the creators of the dataset) summarizes all the different combinations possible.
In total, we have about 61 minutes of audio in the dataset. In large part, the dataset contains notes in the MIDI number range of 39 to 80, since these are the more "frequently used" notes of a guitar.

| | PS ES | PS ES | PS ES | PS ES (Lage) | PS ES (Lage) | PS ES (Lage) | PS ES |
|---|---|---|---|---|---|---|---|
| **Lick 1** *monophon* | fingered | picked | palm muted | fingered *no* (1 Lage) | fingered *v, s, dn* | palm muted *v, s, dn* | picked *v, s, dn* |
| **Lick 2** *monophon* | | | | picked *no* (1 Lage) | fingered *v, s, h* | palm muted *v, s, h* | picked *v, s, h* |
| **Lick 3** *monophon* | | | | palm muted *no* (1 Lage) | fingered *s, b, dn* | palm muted *s, b, dn* | picked *s, b, dn* |
| **Lick 4** *monophon* | | | | fingered *no* (2 Lagen) | fingered *s, b, h* | palm muted *s, b, h* | picked *s, b, h* |
| **Lick 5** *monophon* | | | | picked *no* (2 Lagen) | fingered *b, v, dn* | palm muted *b, v, dn* | picked *b, v, dn* |
| **Lick 6** *monophon* | | | | palm muted *no* (2 Lagen) | fingered *b, v, h* | palm muted *b, v, h* | picked *b, v, h* |
| **Lick 7** *polyphon* | *no* | *no* | *no* | fingered *no* (1 Lage) | picked *no* (1 Lage) | palm muted *no* (1 Lage) | |
| **Lick 8** *polyphon* | | | | | | | |
| **Lick 9** *polyphon* | | | | fingered *no* (1 Lage) | picked *no* (1 Lage) | palm muted *no* (1 Lage) | |
| **Lick 10** *polyphon* | | | | | | | |
| **Lick 11** *monophon* | | | | fingered *no, b, v, h, s, dn* | palm muted *no, b, v, h, s, dn* | picked *no, b, v, h, s, dn* | |
| **Lick 12** *polyphon* | | | | | | | |

Figure A.1: Fraunhofer IDMT Dataset 2 Details

## 1.3 Annotation Mapping

Out of the aforementioned parameters, for our task of multipitch estimation, the most relevant features have been highlighted. Since we work on a frame-level analysis, we *map* the relevant ground truth data as follows (refer to `transcribe.py`). `idmt_processing.py`:

1. create an"entry" for each frame (boundaries governed by `WINDOW_LEN`, `HOP_LEN` and `SAMPLING_RATE`)

2. find the notes currently being played in that frame as follows: if the time instant of the frame's center $\in$ [offsetSec, onsetSec] of a note, then that note is said to be currently playing

3. map to each frame an ordered list of notes, ordered in reverse chronological order w.r.t. onsetSec (the assumption being that the note played more recently is the *dominant* one. Results will depend heavily on this, especially in scenarios where estimated polyphony is less than actual polyphony)

These mappings are stored as CSV files, as shown in the figure. CSV files like this may be generated for any dataset, using the `ConvertXml2csv` function in `idmt_processing.py`.
From this point on, "ground truth" shall refer to the corresponding CSV file of an audio clip.

| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| 1 | win_center_index (s) | win_center_index | start_index | end_index | notes (midi) | notes | chord | chord_type |
| 68 | 0.383129252 | 16896 | 14848 | 18944 | [64, 59, 56, 52, 47, 40] | ['E4', 'B3', 'G#3', 'E3', 'B2', 'E2'] | E | MAJ |
| 69 | 0.38893424 | 17152 | 15104 | 19200 | [64, 59, 56, 52, 47, 40] | ['E4', 'B3', 'G#3', 'E3', 'B2', 'E2'] | E | MAJ |
| 70 | 0.394739229 | 17408 | 15360 | 19456 | [64, 59, 56, 52, 47, 40] | ['E4', 'B3', 'G#3', 'E3', 'B2', 'E2'] | E | MAJ |
| 71 | 0.400544218 | 17664 | 15616 | 19712 | [64, 59, 56, 52, 47, 40] | ['E4', 'B3', 'G#3', 'E3', 'B2', 'E2'] | E | MAJ |
| 72 | 0.406349206 | 17920 | 15872 | 19968 | [64, 59, 56, 52, 47, 40] | ['E4', 'B3', 'G#3', 'E3', 'B2', 'E2'] | E | MAJ |
| 73 | 0.412154195 | 18176 | 16128 | 20224 | [64, 59, 56, 52, 47, 40] | ['E4', 'B3', 'G#3', 'E3', 'B2', 'E2'] | E | MAJ |
| 74 | 0.417959184 | 18432 | 16384 | 20480 | [64, 59, 56, 52, 47, 40] | ['E4', 'B3', 'G#3', 'E3', 'B2', 'E2'] | E | MAJ |
| 75 | 0.423764172 | 18688 | 16640 | 20736 | [64, 59, 56, 52, 47, 40] | ['E4', 'B3', 'G#3', 'E3', 'B2', 'E2'] | E | MAJ |
| 76 | 0.429569161 | 18944 | 16896 | 20992 | [64, 59, 56, 52, 47, 40] | ['E4', 'B3', 'G#3', 'E3', 'B2', 'E2'] | E | MAJ |
| 77 | 0.43537415 | 19200 | 17152 | 21248 | [64, 59, 56, 52, 47, 40] | ['E4', 'B3', 'G#3', 'E3', 'B2', 'E2'] | E | MAJ |
| 78 | 0.441179138 | 19456 | 17408 | 21504 | [64, 59, 56, 52, 47, 40] | ['E4', 'B3', 'G#3', 'E3', 'B2', 'E2'] | E | MAJ |

Figure A.2: csv Annotation file

# 2 Code Structure

The following figure summarizes the structure of the (Python) code:

```
multi_pitch_method
├── training
│   ├── __init__.py
│   ├── extract.py
│   └── train.py
├── utils
│   ├── __init__.py
│   ├── config_reader.py
│   ├── frame_model.py
│   ├── peak_detection.py
│   └── utils.py
├── __init__.py
├── load_models.py
└── transcribe.py
├── complete_evaluation.py
├── config.json
├── evauation.py
├── extract_training_data.py
├── idmt_processing.py
├── main.py
├── README.md
├── requirements.txt
└── train.py
```

The `utils` subdirectory contains utilities, that is functionalities common to training and testing. The `training` subdirectory contains the files for training, while the remainder of the `multi_pitch_method` directory contains other files required to run the main wrapper for testing, `main.py`
We now discuss each of these modules in detail. In principle, one should be able to replace each of these modules, keeping inputs and outputs same.

# 3 Common Utilities

## 3.1 Silence Detection

Before normalizing the energy of each frame to send for further processing (whether train or test), we remove all the so called *silence* frames - frames that are less than 1% in energy of the frame with the maximum energy in the clip, and therefore are not of use as far as acquiring training data or prediction is concerned. Refer to `transcribe.py`.

## 3.2 Peak Detection

Detecting the positions $f_k$ (and hence log-amplitudes $a_k$) of peaks in a frame spectrum is one of the key tasks involved in both training the GMMs and testing a probe audio frame. We use the following algorithm to detect peaks (refer to the `GetFramePeaks` function in `utils/peak_detection.py`):

1. Compute the power spectrum of the frame, and restrict it to an upper limit of `MAX_FREQ` (of, say, 5000Hz) above which there would be no "meaningful" peaks of standard octaves' notes
2. Compute a *smoothed power spectrum* by running a Gaussian kernel of standard deviation `STD_SMOOTHING`
3. Peaks must satisfy the constraint that the difference between the power spectrum and the smoothened spectrum must be greater than a threshold `MAGNITUDE_THRESHOLD`
4. The peaks (both the location $f_k$ and the magnitude $a_k$) are now *refined* as follows:
    - From the points in the above step, only those which are actually local maxima are retained
    - The peaks are then refined by quadratic interpolation to account for errors due to discretization in the DFT

# 4 Training

## 4.1 Extracting Data

As discussed in section 7 our trained models consist of probability density functions for $d_k$, $a_k$, $f_k|h_k$ and $f_k|h_k$, as well as a probability mass function for $e_h = 1|F_0$. We discuss these next. The relevant data is first extracted from the dataset in `extract_training_data.py` and then stored as GMMs in `train.py`.

### 4.2 Modelling $p(d_k)$

$d_k$ is nothing but the deviation of each observed peak from the corresponding nearest harmonic. Since we know $d_k$ to be (virtually) independent of the fundamental frequency, we consider the "nearest" harmonic to be of that fundamental frequency (which is one of a finite set known from the ground truth) for which the deviation itself is least.
This approach is applied on the entire dataset as follows:
Initialize an empty "histogram" for $d_k$. Then, for each frame in each file,

1. find all the peaks (3.2)
2. attribute to each peak the deviation $d_k$ from the nearest harmonic
3. update the histogram

Finally, the histogram itself is modelled as a GMM of 80 kernels with a full covariance matrix (using the `scikit-learn` library) and stored as a pickle file(the number of kernels was found by looking at the AIC/BIC curves).

### 4.3 Modelling $p(a_k, f_k, h_k)$ and $p(f_k, h_k)$

As discussed in 7, we need PDFs for $a_k, f_k|h_k$ and $f_k|h_k$. The following process is followed over the entire dataset:

1. find the peaks, denoted by the pairs $(a_k, f_k)$ and the corresponding $h_k$ (again viewed as the closest harmonic of one of the ground truth fundamentals)
2. form the "histogram" for both $a_k, f_k$ and $f_k$ for each $h_k$, for a predecided range (say 20) of $h_k$ values
3. store each of these as GMMs as before

### 4.4 Non-Peak Region Modelling

To model the non-peak region, we require the probabilities $P(e_h = 1|F_0)$. These are obtained from the dataset as follows: count the number of occurrences of the $h^{th}$ harmonic of the ground $F_0$, which are observed within a tolerance of 3% of the theoretical position $f_k$ of the harmonic. Divide this by the actual number of occurrences of the $h^{th}$ harmonic. Refer to `extract_training_data.py`.

## 5 Testing

We now look at how, on a frame level, the operations for multipitch estimation are implemented.

### 5.1 Candidate Generation

In order to find the fundamental frequencies of the sources in a frame, we first need to generate the candidate set. We saw how candidates are defined in 6.2. In an interval of size `DEVIATION` around each detected peak, with step size `STEP`, in a frequency range `MIN_FREQ` to `MAX_FREQ` (defined in MIDI).

This is exactly what the function `GetCandidates` in `utils/utils.py` does. In addition, it also returns an `inverse_map`, which maps each candidate to the corresponding detected peak. This is an optimization - once we select a candidate that accounts for a given peak (or set of peaks) we need not keep the other candidates around.

## 5.2 The Frame Class

It was observed empirically that the most expensive step is evaluation of the likelihood function. Thus, it is this computation in particular that we would like to do only when absolutely necessary. It is for this reason that we define a frame *class*.
In section 6.3, we saw the high level pseudocode for the algorithm. Observe that the likelihood function must be evaluated for each candidate $F_0$. However, we can save a lot of computation by doing some of the evaluation at the beginning. That is, computations common to candidates are done at the beginning of the frame, and hence associated with the frame by using a class (refer to `utils/frame_model.py`.

## 5.3 Calculating Likelihoods

Recall that we have GMMs for each $h_k$, for both the tuple of each peak $a_k, f_k$ as well as $f_k$. Note that only $h_k$ depends on the candidate frequency.
Thus, instead of re-evaluating the GMM for each candidate for each peak, we evaluate the GMM for each $h_k$ for each peak. It was observed empirically (code profiling via `cProfile`) that the latter approach saves a lot of computation.
Once we have these evaluations, we simply evaluate the likelihood function over all candidates, and thus "pick" the best ones.

## 5.4 Polyphony Estimation

At this point, we have for our frame a list of possible $F_0$'s, from which to pick our final prediction. In order to pick the $N$ best ones, we need to know what this $N$, i.e. the polyphony of the frame is. We have already seen in section 8 how to pick this $N$. This is exactly what the function `PolyphonyProcessing` in `transcribe.py` does. It evaluates $N$ and then spits out the $N$ length `predictions` from the `possible_f0s`.

## 5.5 Putting it all together

The sequence of operations for testing a probe audio clip can be easily followed by looking at `transcribe.py` and the function of the same name.
The models are first "loaded" for use. See `/utils/load_models.py`
Next, we compute the STFT of the audio clip using the given configuration, carry out the computation of `possible_f0s` by making use of the `Frame` class on each frame Fourier transform, and finally run `PolyphonyProcessing` to return the final predictions.
Finally, `main.py` serves as a wrapper around this, with command line arguments to load the audio clip, models and configuration file.

# 6 Running the Code

## 6.1 Environment

All the required files can be run in a simple Python environment. We use the following requirements for the Python environment:

- Python Version==3.7
- librosa==0.8.0
- numpy==1.19.2
- matplotlib==3.3.3
- pandas==1.1.5
- tqdm==4.54.1
- numba==0.53.0
- scipy==1.5.2
- scikit-learn==0.24.0

## 6.2 Configuration File

All the parameters, paths for training and testing of models are present in `config.json` file. The information of the parameters used in these files is shown in the table below

| **Parameter** | **Use** |
|---|---|
| MAX_POLYPHONY | The maximum polyphony till which the greedy algorithm is used |
| T | The threshold for polyphony estimation |
| SAMPLING_RATE | The sampling rate to which the audios are resampled |
| HOP_LEN | The hop length to be used for analysis in ms |
| WINDOW_LEN | Window length to be used for analysis in ms |
| skip | Number of frames to skip between predictions |
| nfft | Number of samples in FFT |
| window | Window function to be used for analysis |
| models/peak_region/path dk_model | Path to the dk GMM model |
| models/peak_region/path ak_fk_hk_model | Path to the folder containing all the (ak, fk, hk) models |
| models/peak_region/path fk_hk_model | Path to the folder containing all the (fk, hk) models |
| models/non_peak_region/path non_peak_count | Path to the numpy file containing the frequency of missed peaks in non peak region |
| models/non_peak_region/path non_peak_freq | Path to the numpy file containing the frequency of fundamental frequencies |
| peak_detection/MAX_FREQ | Maximum frequency till which peaks needs to be detected |
| peak_detection/MAGNITUDE_THRESHOLD | The threshold above which the peaks in power spectrum should lie when compared to smoothened power spectrum |
| peak_detection/STD_SMOOTHING | The standard deviation used in gaussian filer for smoothing the power spectrum |
| candidates/MIN_FREQ | Minimum frequency (hz) of a candidate |
| candidate/MAX_FREQ | Maximum frequency (hz) of a candidate |
| candidate/DEVIATION | Deviation (in %) around the peak location in which to search for candidate |
| candidate/STEP | Steps size (in %) between candidate frequencies around peak |
| raw_data | Path to the raw data files for training |

## 6.3 Heuristics for Parameter Tuning

- Silence Detection: a threshold of 1% was set to ignore silence frames, That is, frames with energy 1% or less of the peak energy frame in the audio file are ignored
- `peak_detection/MAX_FREQ`: peak detection was carried out only up to 5000 Hz, since it was observed that beyond this number, noise dominates the higher harmonics of the notes
- Max. number of harmonics (hardcoded): kept to 20, since for most instruments, harmonics beyond 20 will be masked by noise
- Threshold $T$ for Polyphony estimation: we observe the "diminishing returns" behaviour of the log likelihood function, and set $T$ such that beyond $T \times \Delta(M)$ not much change happens, as shown below.

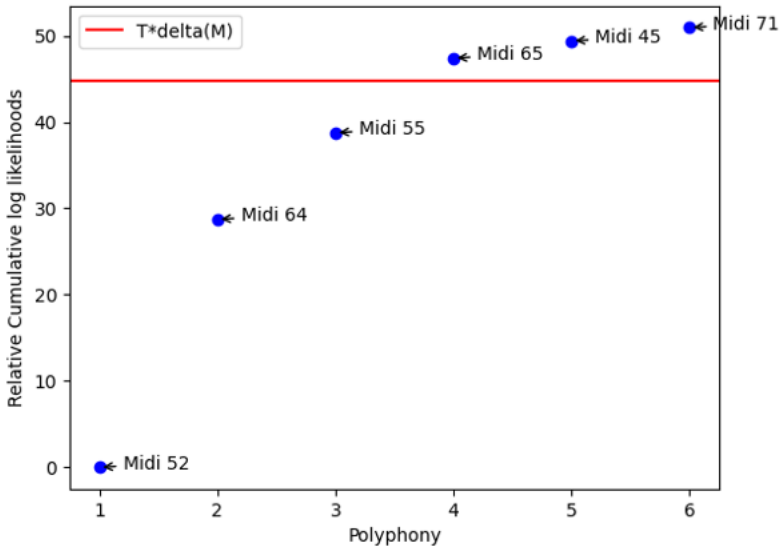



Figure A.3: Method for Estimating $T$

- `SAMPLING_RATE`: to get a better frequency resolution in STFT
- `candidates/MIN_FREQ` and `candidates/MAX_FREQ`: it was observed in the dataset that most observations lie in the MIDI range [39,80]
- `MAX_POLYPHONY`: kept to 5 since for major and minor chords, expect polyphony of 3 (as discussed in octave folding). In principle, any finite number bigger than 3 suffices
- `WINDOW_LEN` and `HOP_LEN`: to balance time-frequency resolution trade-off (dataset dependent). Found by observation in Praat or any other audio analysis tool

## 6.4 Training the Models

### Extracting Data for Model Training

The next step in the training process if the extraction of training data. This is done by `ExtractFromDataset` function in `extract_training_data.py` file. This file takes 2 inputs: `config.json` and path to dataset, and saves $(a_k, f_k, h_K)$, $d_k$, and the count of harmonic peaks in non peak region, frequency of fundamental frequencies. The locations of these files is shown in the table below.

| Data | Location |
|---|---|
| $a_k, f_k, h_k$ | `config['raw_data']/ak_fk_hk.npy` |
| $d_k$ | `config['raw_data']/dk.npy` |
| count of harmonic peaks in non peak region | `config['models']["non_peak_region"] ['path non_peak_count']` |
| frequency of fundamental frequencies | `config['models']["non_peak_region"] ['path non_peak_freq']` |

### GMM Training

The main script for training the GMMs is present in `train.py`. This takes the path of raw data from the config file (as in the above table), trains the GMMs and stores the trained GMMs in the path given in config file. This is done by the `TrainPeakRegion` function present in the `multi_pitch_method` folder to keep consistency in training when dataset changes.

## 6.5 Testing on an Audio File

For this, we need only run the wrapper `main.py` with the relevant arguments:

- `audio_file`: Path to the audio file to transcribe (frame by frame transcription)
- `config_file`: Path to the config.json file
- `profiling_out`: Path to the output file of code profiling. If not supplied, code profiling will not be done
- `plot_file`: Path to the save the plot of the prediction. If not supplied, no plot will be saved
- `output`: Path to which output must be saved. It will be saved in pickle format