# EE323 Mini Project Report

Yuncheng Tu

Student ID: 12111008

***Abstract*—This project aims to understand how musical elements such as pitch, rhythm, and timbre can be translated into digital waveform data that can be processed by computers. Key musical concepts like notes, scales, intervals, and durations are represented numerically. Functions are implemented in MATLAB to generate sine waveforms that correspond to musical notes based on their frequency, amplitude envelope, and inclusion of harmonic overtones. This allows digital audio signals to be synthesized from musical scores.**

***Index Terms*—musical signal processing, digital sound synthesis, Fourier analysis, sound timbre, decay function, tone, harmonic waves.**

## I. Introduction

MUSIC incorporates many elements that give structure and expression. To analyze and generate music computationally, these elements must be represented numerically. This work examines how pitch, rhythm, scales and other aspects are encoded in digital musical notation. Functions are developed in MATLAB to convert these numeric representations into audio waveforms. Different methods to model features like amplitude decay and harmonic content are explored to synthesize increasingly realistic instrument tones.

## II. MUSICAL ELEMENTS TRANSLATION TO DATA

### *A. Understanding Musical Components*

Music is composed of elements such as pitch, rhythm, timbre, fundamental frequency, and chords. These elements are translated into data that can be processed by computers to generate a matrix representing a musical score. This matrix has dimensions M * N, where N is the number of channels. The task is to understand how these elements are reflected in our data and use MATLAB to convert a simple musical score into data that can be played as music. Additionally, the aim is to produce music that mimics certain musical instruments.

### *B. Digital Numbered Musical Notation*

#### *1) Pitch in Notation*

Pitch is indicated in numerical scores by the numbers 1 to 7, which correspond to the seven basic pitches in the scale: do, re, mi, fa, sol, la, ti, with rests represented by 0. These pitches are associated with different frequencies, which can be halved or doubled to represent octaves above or below the given note.

The ratio of frequency between these seven notes are given by $1: 2^{\frac{2}{12}}: 2^{\frac{4}{12}}: 2^{\frac{5}{12}}: 2^{\frac{7}{12}}: 2^{\frac{9}{12}}: 2^{\frac{11}{12}}$

#### *2) Tonal Key Signatures*

The key signature of a score provides the frequency for the tonic (the "1" note), which is essential for calculating the frequencies of all other notes. For example, in the key of C (notated as 1=C), the frequencies of the notes 1 to 7 are directly provided. When the key changes, these frequencies can be recalculated based on their relation to the tonic note in the key of C.

#### *3) Note Duration*

Note duration in numerical scores is manipulated through the use of lines and dots to indicate extension or reduction of the note's length. Different symbols are used to double the duration, cut it in half, or add half of the original duration.

### *C. MATLAB Implementation by function **tone2freq***

#### *1) Frequency Calculation Function overview*

The ‘tone2freq’ function in MATLAB converts numerical notes into frequencies. The function takes in four parameters: the numerical note (tone), the tonal scale (scale), octave alterations (noctave), and whether the note is raised or lowered in pitch (rising). It uses a base frequency array for the key of C and adjusts according to the given parameters to calculate the accurate frequency.

Therefore, we are going to implement this formula:

$$\textbf{Frequency} = \textbf{tonal scale} \times 2^{octave} \times 2^{\frac{rising}{12}} \times tone's\ ratio \qquad (1)$$

```
function freq = tone2freq(tone, scale, noctave,
rising)
C_scale_base_freqs = [261.5, 293.5, 329.5, 349,
391.5, 440, 494];
main_f = C_scale_base_freqs(scale);
semitone_map = [0, 2, 4, 5, 7, 9, 11];
semitone_from_main = semitone_map(tone) + rising;
freq = main_f * 2^(semitone_from_main/12) *
2^noctave;
end
```

#### *2) Parameters analysis*

1. Base Frequency identification

For a given scale (denoted as **scale**), the base frequency (**main_f**) is identified from a reference array of frequencies

corresponding to the C major scale (**C_scale_base_freqs**). This reference array is defined as:
**C_scale_base_freqs**=[261.5,293.5,329.5,349,391.5,440,494] which corresponds to the musical notes C, D, E, F, G, A, B, respectively.

2. Semitone Shift Calculation

The numerical tone (**tone**) is used to index into a semitone map (**semitone_map**):
***semitone_map***=[0,2,4,5,7,9,11]

The position of the tone within the scale is given in terms of semitones from the base note of the scale. This value is then adjusted by the **rising** parameter to account for any sharp or flat modifications.

3. Frequency Calculation

The frequency of the note is calculated using the following formula, which incorporates the semitone shift and octave adjustment (**noctave**):

$$\boldsymbol{freq = mainf \times 2^{(semitone\_from\_main/12)} \times 2^{noctave}}$$

where **semitone_from_main** is the sum of the semitone position from the **semitone_map** and the **rising** adjustment.

4. In this formula:

$2^{(semitone\_from\_main/12)}$ represents the frequency change due to the shift in semitones, utilizing the fact that a semitone step corresponds to the twelfth root of 2.

$2^{noctave}$ accounts for the change in frequency when moving across octaves, doubling or halving the frequency for each octave up or down, respectively.

## III. GENERATING WAVEFORMS

Generating waveforms for specific frequencies is a common practice in audio signal processing. In MATLAB, this is typically achieved using trigonometric functions like the sine function to simulate sound waves. The human audible frequency range spans from 20 Hz to 20 kHz, and sounds outside this range cannot be heard. Therefore, generating waveforms within this audible range is crucial for any auditory application.

### *A. Generating Single Note Waveforms*

#### *1) Code Functionality*

The provided MATLAB function **gen_wave** takes musical note parameters and generates the corresponding sine wave. It relies on the previously discussed **tone2freq** function to convert a musical note into its frequency.

#### *2) Waveform Generation Process*

To generate a waveform, the function calculates the frequency using the **tone2freq** function, then creates a time array that spans the note's duration (**rhythm**). It ensures the number of samples matches the desired note duration by rounding the product of **rhythm** and the sampling frequency **fs** to the nearest integer. Finally, it uses the sine function to produce a waveform that corresponds to the note's frequency over the given time array.

MATLAB Code:

```
function waves = gen_wave(tone, scale, octave,
rising, rhythm, fs)
    f = tone2freq(tone, scale, octave, rising);
    t = linspace(0, rhythm, round(rhythm * fs));
    waves = sin(2 * pi * f * t);
end
```

### *B. "Castle in the Sky" Melody Waveform Generation*

#### *1) output*

To connect individual notes into a complete waveform for a melody, we start by translating each note from the numerical sheet music into a discrete audio signal with a specified pitch and duration. Utilizing a function, `gen_wave`, each note is converted into a waveform segment based on its frequency, duration, and amplitude. These segments are then concatenated in sequence by function '**gen_music**' to form a continuous audio signal that represents the entire piece. After generating the waveform for the full melody, we plot it to visualize the audio signal's shape and examine the output. This process converts the abstract notation of sheet music into a tangible auditory experience, bridging the gap between numeric scores and musical sound.

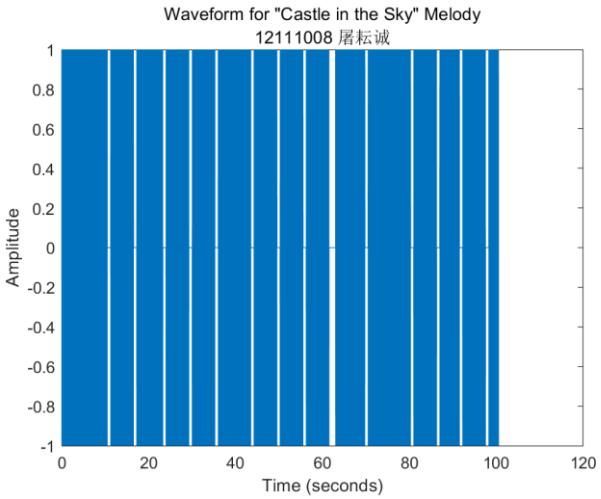



Figure. 1

#### *2) analysis*

Upon the generation and plotting of the entire waveform for the "Castle in the Sky" melody, we observe a distinct pattern corresponding to the rhythmic values and pitches of the piece. Each note vibrates at a fixed amplitude, and the waveform presents a rectangular shape due to the constant amplitude of the notes. This visualization confirms that there is no attenuation between the notes, and the legato line, which should ideally indicate a smooth transition between notes, is severely lacking, resulting in a staccato effect that does not represent the intended musical expression well.

## IV. VOLUME FLUCTUATION: APPLICATION OF DECAY FUNCTIONS

### *A. Background*

In the translation from musical notation to digital waveform, the resulting sound often resembles that of an electronic instrument rather than a traditional one, largely due to volume dynamics. Traditional instruments exhibit a natural decay in volume - the sound is loudest at the onset of the note

and diminishes over time as energy dissipates, such as from air friction. This decay causes the amplitude of the waveform to decrease as time progresses. To emulate this acoustic characteristic digitally, envelope decay functions such as linear, square, and exponential decay have been implemented. These functions are not only applied to individual notes but also across sequences of notes to more authentically simulate the production of music.

*B. Methodology*

We utilized MATLAB to generate a pure sine wave representing a single note. Three decay functions were applied:

1. Exponential Decay

$$A(t) = A_0 e^{-\alpha t}$$

where *A*0 is the initial amplitude, $\alpha$ is the decay constant, and *t* is time.

2. Linear Decay

$$A(t) = A_0 \left(1 - \frac{t}{T}\right)$$

A simple linear decrease from the initial amplitude to zero, , where T is the total duration.

3. Square Decay

$$A(t) = A_0 \left(1 - \frac{t}{T}\right)^2$$

```
% 指数衰减
exp_decay = exp(-t/rhythm);
y_exp = y .* exp_decay;
% 线性衰减
lin_decay = linspace(1, 0, length(t));
y_lin = y .* lin_decay;
% 平方衰减
sqr_decay = (1 - (t/rhythm)).^2;
y_sqr = y .* sqr_decay;
```

To visualize the decay functions applied to a single musical note, let's consider an example with the note A4, which has a frequency of 440 Hz. This frequency is within the range of many musical instruments and is often used as a standard for tuning. The simulated waveforms represent the note A4 subjected to different decay envelopes to illustrate how these envelopes influence the timbre and realism of the synthesized sound

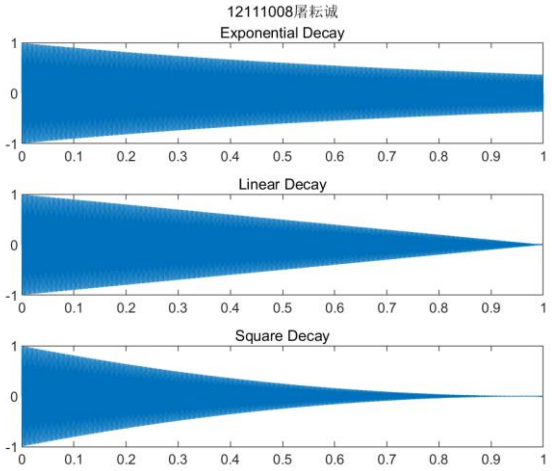



Figure. 2

The plots reveal distinct differences in the decay profiles. The exponential decay provided a natural sounding fade-out, closely resembling the attenuation characteristics of many acoustic instruments. Linear decay offered a more abrupt decrease in amplitude, while square decay presented a gentler fade-out than linear but less natural than exponential.

To simulate a more realistic sound envelope similar to that of traditional musical instruments, the exponential decay function is incorporated into the existing **gen_wave** function in MATLAB. The code snippet below demonstrates the updated function:

```
decay_exp = exp(-3*t/rhythm); % Exponential decay factor
waves = waves .* decay_exp; % Modulation of the sine wave
```

Here, **t** represents the time vector, **rhythm** is the note duration, and **waves** is the generated sine wave. The decay constant (3 in this case) determines the rate of decay, which can be adjusted to match the characteristics of different instruments.

Then the decay function is used to test the new waveform:

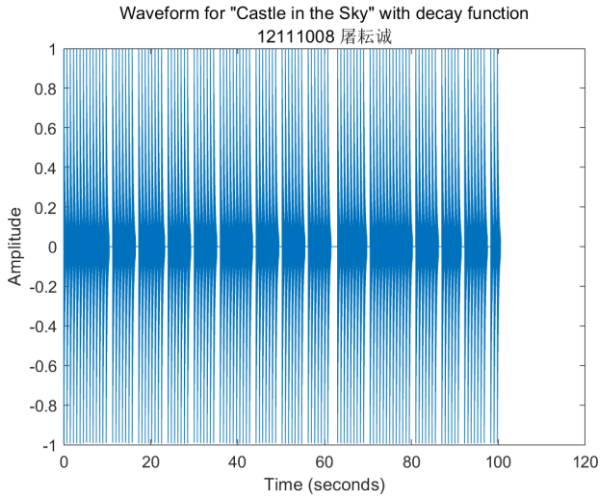



Figure. 3

With the integration of the exponential decay function, the resulting waveform exhibits a reduction in amplitude over time. This behavior mimics the natural decay of sound from a

traditional musical instrument, where the loudest moment occurs at the initial strike or pluck, followed by a gradual decrease due to energy loss through air friction and the inherent damping characteristics of the instrument's materials.

### C. Conclusion

Exponential decay is the most effective in simulating the natural attenuation of sound in an acoustic environment, followed by square and linear decay. Further exploration could involve applying these decay functions across a series of notes to assess the impact on more complex musical phrases.

## V. Harmonics and Timbre in Synthesized Music

### A. Background

While a musical score indicates the fundamental frequencies, traditional instruments produce a complex sound that includes a series of harmonics or overtones. These harmonics are integer multiples of the fundamental frequency, created by standing waves within the instrument. The characteristic sound, or timbre, of an instrument is shaped significantly by the relative intensity of these harmonics.

### B. Implementation

The MATLAB function **gen_wave** was enhanced to simulate the harmonic series along with the fundamental frequency. This was achieved by summing sine waves at integer multiples of the fundamental frequency, with each harmonic weighted by a specific factor to reflect its relative strength in the overall sound.

Code:

```
harmonics = [1,0.20,0.15,0.15,0.10];
waves = harmonics(1)*sin(2*pi*f*t);
% Fundamental frequency
for n = 2:length(harmonics)
   waves = waves + harmonics(n)*sin(2*pi*n*f*t);
% Adding the nth harmonic
end
waves = waves/max(waves);
% Normalize the amplitude
```

#### 1) Time Domain

In figure4, when a single note C6 is used for the test waveform, and harmonic components are added, the waveform clearly exhibits an additional peak, indicating a change in shape.

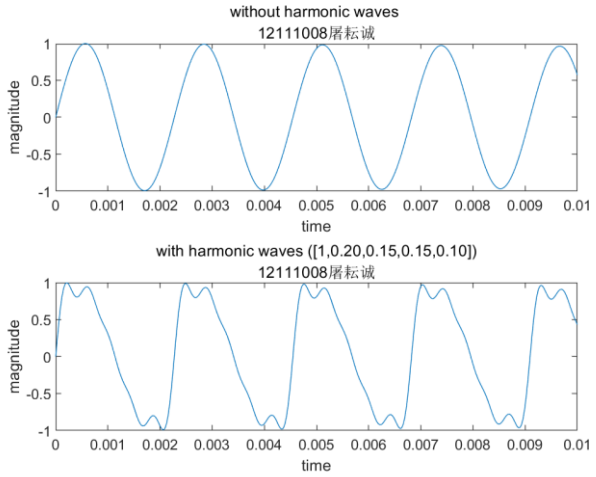


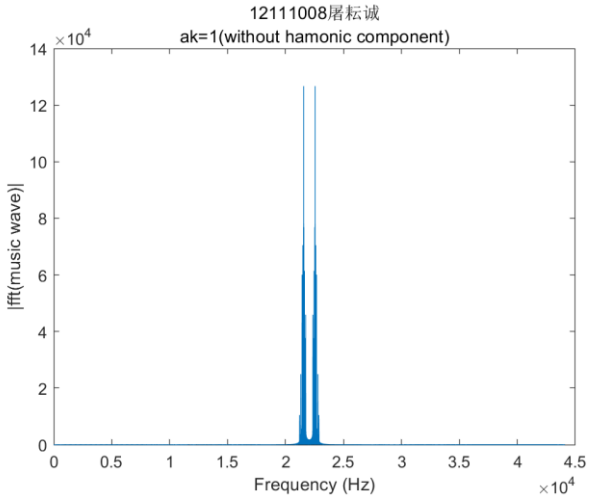

Figure. 4

#### 2) Frequency Analysis of the entire spectrum

First, we can write down the waveform function expressions for each harmonic.

$$y = \sum_{k=1}^{N} ak \sin(2\pi kft)$$

Then, we experiment with different harmonic component ratios to test. From the frequency domain, it can be observed that for discrete real periodic signals, the spectrum exhibits symmetry, and the data is discrete. Furthermore, it can be seen that the peak data closely matches the harmonic energy ratios in the code.



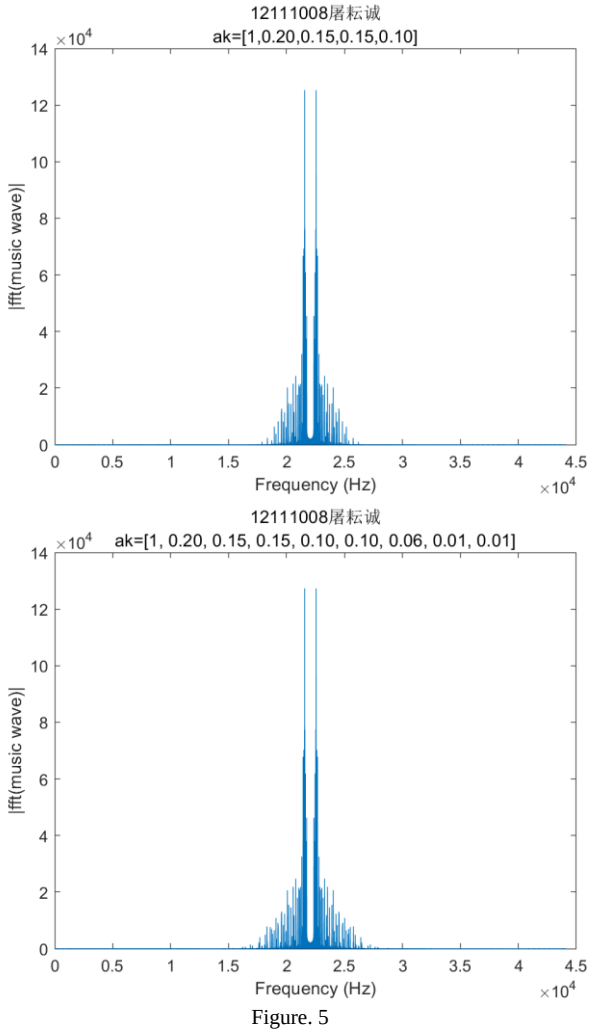



Figure. 5

From figure5, it can be seen that the energy is concentrated around the frequency of 2πkf_base. Additionally, as the number of harmonic components increases, the frequency components also gradually increase.

In general, lower notes on a piano tend to have more pronounced harmonic content compared to higher notes, and the energy of harmonics decreases as the harmonic number increases. For the purpose of this example, let's assume a set of simplified harmonic ratios for piano notes. These values are not exact, but they serve as a basic starting point for simulation. In our final test, we adopted a harmonic component ratio of ak = [1, 0.20, 0.15, 0.15, 0.10, 0.10, 0.06, 0.01, 0.01], resulting in a better sound quality. It provides a more faithful representation of piano timbre with a clear, crisp, and elegantly pleasing tone.

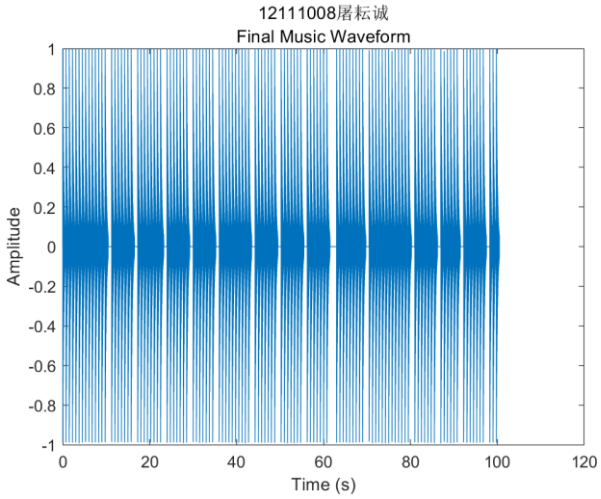



Figure. 6

## VI. Conclusion

By implementing amplitude decay envelopes and incorporating harmonic overtones, the digitally synthesized tones begin to accurately mimic the complex, evolving sounds produced by acoustic instruments. Specifically, we improved the sound quality by modifying the decay function to exponential and adjusting the harmonic component ratios to better simulate the timbre of piano tones. These techniques allowed us to successfully synthesize the melody of "Castle in the Sky" from its numerical score representation. The methods demonstrated in this project bridge the domains of abstract musical notation and perceived auditory experience, enabling algorithmic generation and processing of music.