Name: **Group 9**

**Alamay, Carl Justine S.**

**Del Rosario, Lance Victor**

Section: **S11B**

Laboratory Activity Title: “Music Synthesis”

Files Used:

**Instructions:** , for each of the sections indicated, please provide your corresponding answers to each of the questions indicated in the lab activity section. Please provide relevant information, i.e. **source code, graphs, explanations/answers to the question/s.**

**4 Lab Exercise: Synthesis of Musical Notes**

load bach\_fugue.mat

**Command to load bach\_fugue\_short into the terminal.**

Loading back\_fugue.mat is necessary for the completion of this lab because it contains important functions that would allow debugging and musical synthesis of the different notes to be simplified. It also contains theVoices which are structures that contain information or data about certain melodies, and whose function will be used in the completion of the lab exercise.

bpm = 120;

fs = 11025;

qtr\_per\_second = bpm/60;

seconds\_per\_qtr = 1/qtr\_per\_second;

seconds\_per\_pulse = seconds\_per\_qtr/4;

dur\_each\_note1 = theVoices(1).durations \* seconds\_per\_pulse;

dur\_each\_note2 = theVoices(2).durations \* seconds\_per\_pulse;

dur\_each\_note3 = theVoices(2).durations \* seconds\_per\_pulse;

x1\_len = ceil(sum(dur\_each\_note1)+1)\*fs;

x2\_len = ceil(sum(dur\_each\_note2)+1)\*fs;

x3\_len = ceil(sum(dur\_each\_note3)+1)\*fs;

xx\_len = 0;

if (x1\_len > x2\_len)

xx\_len = x1\_len;

end

if (x3\_len > xx\_len)

xx\_len = x3\_len;

end

n1 = ceil(theVoices(1).startPulses(end)\*seconds\_per\_pulse\*fs);

durae = dur\_each\_note1(end);

xx\_len = ceil(n1 + durae\*fs);

%xx\_len = ceil(sum(max)+1)\*fs; %xx\_len we got da total length of melody in s, then multiple to how many samples needed each sec

xx = zeros(1, xx\_len); %the length of

x = 1\*exp(j\*0);

for jj = 1:length(theVoices)

dur\_each\_note = theVoices(jj).durations \* seconds\_per\_pulse;

x1\_len = ceil(sum(dur\_each\_note)+1)\*fs;

x1 = zeros(1, xx\_len);

n1 = theVoices(jj).startPulses(1)\*seconds\_per\_pulse\*fs;

for kk = 1:length(theVoices(jj).noteNumbers)

keynum = theVoices(jj).noteNumbers(kk);

durae = dur\_each\_note(kk);

note = key2note(x, keynum, durae);

%creating the Envelope

Env\_length = length(note);

Action = round(0.1\*Env\_length);

Drop = round(0.1\*Env\_length);

Sustain = round(0.6\*Env\_length);

Release = Env\_length - Action - Drop - Sustain;

Env\_Action = linspace(0,1, Action);

Env\_Drop = linspace (1,0.8, Drop);

Env\_Sustain = linspace (0.8, 0.6, Sustain);

Env\_Release = linspace (0.6,0, Release);

Env = [Env\_Action Env\_Drop Env\_Sustain Env\_Release];

Env = Env(1:length(note));

note = Env.\*note;

n2 = n1 - 1 + length(note);

x1(n1:n2) = x1(n1:n2) + note; %<=== Insert the note

n1 = theVoices(jj).startPulses(kk)\*seconds\_per\_pulse\*fs;

end

for ii = 1:length(x1)

xx(ii) = xx(ii) + x1(ii);

end

end

**Script to generate the melody of Fugue #2.**

The entire script generates a musical melody with a specified tempo, which will then apply an amplitude envelope to control the structure and shape of every note. For every note in the melody, key2note is called to generate a specific tone, and an envelope is applied to control the magnitude over time which also features the action, drop, sustain, and release phases. These notes are then modified and are inserted into the signal, which is played by the soundsc() function in the end.

bpm = 120;

fs = 11025;

qtr\_per\_second = bpm/60;

seconds\_per\_qtr = 1/qtr\_per\_second;

seconds\_per\_pulse = seconds\_per\_qtr/4;

dur\_each\_note1 = theVoices(1).durations \* seconds\_per\_pulse;

dur\_each\_note2 = theVoices(2).durations \* seconds\_per\_pulse;

dur\_each\_note3 = theVoices(2).durations \* seconds\_per\_pulse;

**Lines of code that define the tempo of the melody.**

The initial bmp is set to 120 with the sampling frequency being set to the standard value of 11025. The bmp is then applied to calculate quarter notes per second, and the value from the latter variable will then be used to calculate the seconds per quarter note, and finally the seconds per beat variable will then be used to calculate the duration of each pulse. The duration of each note is then calculated by the duration stored in theVoices and duration of each pulse.

x1\_len = ceil(sum(dur\_each\_note1)+1)\*fs;

x2\_len = ceil(sum(dur\_each\_note2)+1)\*fs;

x3\_len = ceil(sum(dur\_each\_note3)+1)\*fs;

xx\_len = 0;

if (x1\_len > x2\_len)

xx\_len = x1\_len;

end

if (x3\_len > xx\_len)

xx\_len = x3\_len;

end

**Lines of code that determines the length of the final audio signal.**

The snippet calculates the total length, in samples, needed for each voice, and then determines the overall length required among all the voices. The lengths of each voice is calculated by getting the ceiling of the sum of all the durations of the notes in each voice, adding 1 second, and then multiplying by the sampling frequency.

n1 = ceil(theVoices(1).startPulses(end)\*seconds\_per\_pulse\*fs);

durae = dur\_each\_note1(end);

xx\_len = ceil(n1 + durae\*fs);

xx = zeros(1, xx\_len);

x = 1\*exp(j\*0);

**Lines of code that prepare the necessary variables for the final audio signal.**

This code block calculates the starting index for the last note of the first voice, ensuring it aligns with a whole sample. It then determines the duration of the last note. The total length of the final audio signal is adjusted to cover the entire duration of the last note. xx is initialized as a zero-filled vector with a length of xx\_len. Additionally, a complex number x is created with a value of 1 multiplied by Euler's number raised to the power of zero imaginary unit for further use.

for jj = 1:length(theVoices)

dur\_each\_note = theVoices(jj).durations \* seconds\_per\_pulse;

x1\_len = ceil(sum(dur\_each\_note)+1)\*fs;

x1 = zeros(1, xx\_len);

n1 = theVoices(jj).startPulses(1)\*seconds\_per\_pulse\*fs;

**Lines of code that prepare necessary parameters for notes processing.**

This block initializes variables specific to the current voice. It calculates the durations of notes in the current voice, determines the length of the signal for the current voice, initializes a zero-filled vector x1 with a length of xx\_len, and sets the starting index for the first note of the current voice.

for kk = 1:length(theVoices(jj).noteNumbers)

keynum = theVoices(jj).noteNumbers(kk);

durae = dur\_each\_note(kk);

note = key2note(x, keynum, durae);

**Lines of code that process each note in the current voice.**

These lines of code iterates through each note in the current musical voice. For each note, it extracts the musical note number and duration. Then, using the function key2note, it creates the musical waveform corresponding to that specific note, considering its pitch and duration. This process essentially translates the musical sheet into a series of sound waves, forming the individual notes that collectively make up the melody of the entire composition.

Env\_length = length(note);

Action = round(0.1\*Env\_length);

Drop = round(0.1\*Env\_length);

Sustain = round(0.6\*Env\_length);

Release = Env\_length - Action - Drop - Sustain;

**Lines of code that define the envelope and the duration of each of its phases.**

The amplitude envelope that will shape the volume of the musical note is calculated with the appropriate parameters. The variables Action, Drop, and Sustain correspond to the different phases of the envelope and are calculated based on the length of the overall envelope. The Release variable will then ensure that the sum of all phases add up to the envelope’s total length. The aforementioned phases create dynamic and expressive sounds or tones to the melody by controlling how the amplitude of the note changes over a certain period.

    Env\_Action = linspace(0,1, Action);

    Env\_Drop = linspace (1,0.8, Drop);

    Env\_Sustain = linspace (0.8, 0.6, Sustain);

    Env\_Release = linspace (0.6,0, Release);

    Env = [Env\_Action Env\_Drop Env\_Sustain Env\_Release];

    Env = Env(1:length(note));

**Lines of code to create the actual envelope of the note.**

These lines of code construct the actual envelope of the musical note, which determines the volume change over time. The Action phase gradually increases from 0 to 1, the Drop phase gradually decreases from 1 to 0.8, the Sustain phase gradually decreases from 0.8 to 0.6, and the Release phase gradually decreases from 0.6 to 0. The different phases are then concatenated to create the complete envelope, whose length is then trimmed to match the length of the musical note.

note = Env.\*note;

n2 = n1 - 1 + length(note);

x1(n1:n2) = x1(n1:n2) + note;

n1 = theVoices(jj).startPulses(kk)\*seconds\_per\_pulse\*fs;

**Lines of code that apply the envelope to a specific note.**

The element-wise multiplication seen in the note variable scales each sample of the musical note by the corresponding value in the envelope, which basically shapes the overall volume of the note. The end position of the note in the signal is then calculated and is then added to the overall musical composition stored in the xx variable. The starting point of the note is then updated to match the predefined pulses in theVoices.

for ii = 1:length(x1)

xx(ii) = xx(ii) + x1(ii);

**Lines of code that combine every modified signal into one.**

This process effectively combines the contributions of all notes in the current musical voice to create a composite waveform representing the complete audio signal for that voice. The loop iterates through each element of x1 and updates the corresponding elements in xx, leading to the formation of the complete audio signal that encompasses the entire composition, incorporating the characteristics of all individual notes and their envelopes.

A blue and yellow graph

Description automatically generated

**Spectrogram of Fugue #2.**