Plucked-String Synthesizer

Karplus Strong method

Audio Processing mini-project

Louise Dørr Nielsen 20183839 ldni18@student.aau.dk
Thomas Kim Kroman Kjeldsen 20183944 tkjeld18@student.aau.dk

Aalborg University
The Technical Faculty of IT and Design (TECH)

Contents

Chapter 1
Introduction

This report is part of the fourth semester mini-project for Audio Processing, and prerequisite for attending the exam. The aim of the project is to make a Karplus Strong Pluck-String synthesizer, and make a code that can play different chords depending on which button is pressed. For this propose, the synthesizer have been made a long with a very simple GUI, using TKinter in Python. The GUI contains a button for all Major chords. This report will first go over relevant research for this project, then go through the implementation of the synthesizer and lastly, possible improvements of the prototype will be covered.

Chapter 2
Background Research

Here, the background knowledge used to make the Karplus Strong Synthesizer and the different chords of a guitar, will be covered.

2.1 Plucked-string synthesis

The idea behind a plucked string synthesis is to simulate the sound of a plucked-string instrument from white noise, for example an acoustic guitar. This is done by manipulating and extending the sinusoid of the white noise. The method used for the manipulation is the karplus strong model [1].

2.1.1 White noise

White noise is a group of random sinusoids with different frequencies, thereby creation one sinusoid that is randomised and can appear chaotic when looking at it in the frequency spectrum.
White noise can appear naturally by, for example a sound wave bouncing off walls and the sound waves thereby arriving at a microphone with a delay, causing the received signal to be chaotic [1].

2.2 Karplus strong model

This model is a method of simulating the sound of a string being plucked, which is done by sending random noise through a feedback comb filter, a low-pass filter and an all-pass filter. These filters will first be explained individually, followed by an explanation of how they should be incorporated together to form the karplus strong synthesizer [1].

2.2.1 Feedback comb filter

The feedback comb filter is the most common example of a comb filter, where the output signal is delayed and added back to the input signal. The difference equation for this filter looks like this:

$$y_n = x_n + R^D y_{n-D}$$

Where y_n is the output signal, x_n is the input signal, D is the amount of samples the output should be delayed and R is the filter coefficient. R should be between zero and one, where if it is smaller than one the filter will be decaying and therefore be stable. The filter coefficient should not be larger than one, since this would result in the frequency amplifying itself and the volume getting more and more loud. But if it is equal to one, the filter will create a repeating pulse. A signal flow diagram of the filter can be seen in figure 2.1, where the different components is marked as in the difference equation above. In the figure, it is shown how the output signal is delayed and added back to the input signal [1].

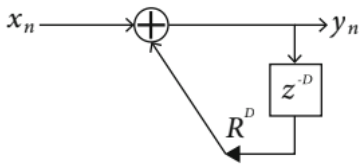

Figure 2.1: Signal flow diagram of a comb filter[1].

The reason why it is called a comb filter, is due to the amplitude spectrum of the filter having a comb-like structure, which can be seen in figure 2.2.

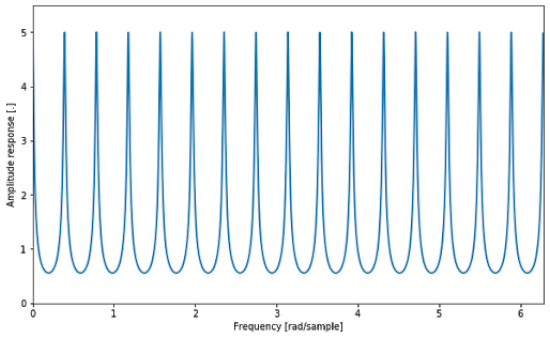

Figure 2.2: Visual representation of a comb filter [4].

2.2.2 All-pass filter and low-pass filter

Different from most other filters, an all-pass filter is a filter designed to change the phase of a signal instead of the amplitude. A low-pass filter is designed to let all frequencies beneath a cutoff frequency through and remove or attenuate the rest. In figure 2.3 a signal flow diagram of a low-pass filter can be seen, in this case the input signal gets halved and a delayed version of the input signal gets halved as well, whereafter they are added together [1].

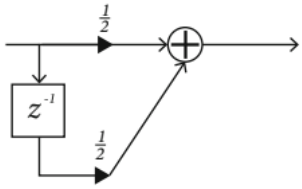

Figure 2.3: Signal flow diagram of a lowpass filter [1].

2.2.3 Combining them in the model

The first step in combining these three filters, is to make sure that the filter coefficient in the comb filter is greater than zero and smaller than one. In that way, the signal that is sent through will attenuate over time. The reason for this, is to make sure that the pitch can be controlled with the delay D in the comb filter:

$$f_0 = \frac{f_s}{D}$$

Where f_0 is the pitch and f_s is the sampling frequency, the reason why we want this is to replicate the effect of sound dying out over time, for example when a string is plucked on a real guitar. What is missing now is the effect seen, when looking at the difference between lower and higher frequencies when a real string is plucked. The higher frequencies die out faster than the lower ones, which is accomplished in the model by introducing a low-pass filter, which would attenuate some of the energy from the higher frequencies when the signal passes through. The filter needed for this is the one seen in figure 2.3, and the difference equation for it looks like this:

$$y_n = \frac{1}{2}x_n + \frac{1}{2}x_{n-1}$$

When this filter is inserted into the comb filter seen in figure 2.1, the result will have a signal flow diagram as the one seen in figure 2.4 [1].

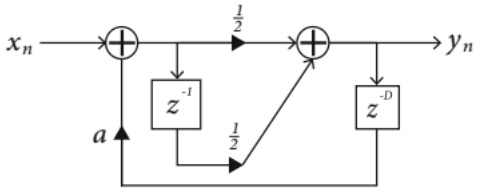

Figure 2.4: Signal flow diagram of a combination of the comb and the low-pass filter[1].

Lastly, in order to be able to control the pitch, the half-sample delay from the low-pass filter must be taken into account. To do this, a type of all-pass filter called a fractional delay filter, will be introduced. This makes sure to minimise the delay in such a way, that the pitch can be controlled with the comb filter delay. The three filters combined can be seen in figure 2.5.

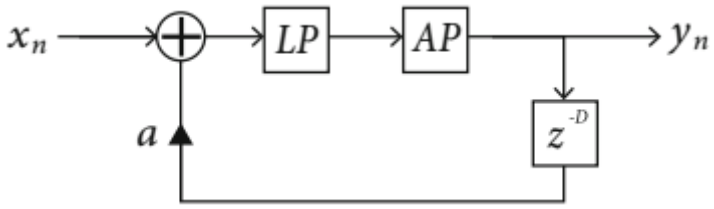

Figure 2.5: Signal flow diagram of the three filters combined[1].

2.3 Guitar

Figure 2.6 shows the layout of an acoustic guitar. This guitar is comprised of six strings attached to the bridge and stretched over the sound hole, along the neck and ends at the tuners. The tuners are used to control the tension of the strings, and this tension, along with density and length of the string, is what makes the guitar able to produce different frequencies. The frequency is given by

$$f_0 = \frac{c}{2L} = \frac{1}{2L}\sqrt{\frac{T}{p}}$$

Where c is the propagation speed of the string in meters per second and L is the length in meters. T is the tension of the string measured in newtons and p is the linear density measured in kg/m.

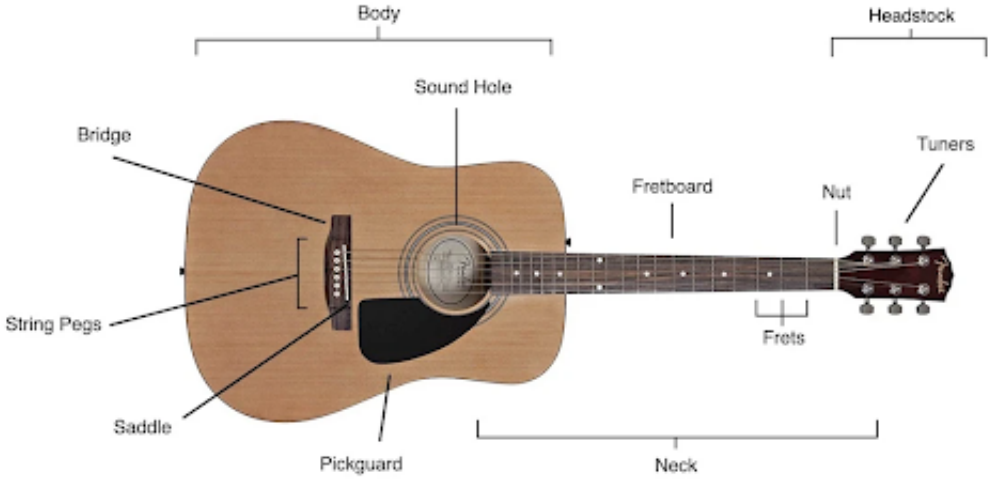

Figure 2.6: Illustration of an acoustic guitar, with names of the different aspects of it. [3]

Along the neck several frets are placed, and by pressing down a string between two frets, the length is shortened of that string and the octaves of that note is thereby changed. A note on a guitar is produced when a string is plucked over the sound hole, and if several strings are pressed down at the same time a chord is produced. The guitar produce the sound by the vibrations from the plucked-string, and these are echoed inside the body of the guitar which makes the sound wave arrive at the ear with different delays.

2.3.1 Frequencies

Every note on a musical instrument have a specific frequency, and different octaves. An octave is the doubling of a frequency, meaning that one octave above a frequency of 100hz is 200hz and one octave below is 50hz

On a guitar the six strings is noted by letters from A - G, and in this project only the fourth octave is used, table 2.1 shows the frequencies of the fourth octave spectrum and table 2.2 shows the chords used in the program.

C_4	261.63
$C_4^\# / D_4^b$	277.18
D_4	293.66
$D_4^\# / E_4^b$	311.13
E_4	329.63
F_4	349.23
$F_4^\# / G_4^b$	369.99
G_4	392.00
$G_4^\# / A_4^b$	415.30
A_4	440.00
$A_4^\# / B_4^b$	466.16
B_4	493.88

Table 2.1: Table of frequencies of notes[1]

C Major	C, E, G
D Major	D, F#, A
E Major	E, G#, B
F Major	F, A, C
A Major	A, C#, E
B Major	B, D#, F#

Table 2.2: Tables of chords used in the program[2]

Chapter 3
Implementation

This chapter contains an explanation of the important aspects of the code used to create the Karplus Strong Synthesizer.

3.1 Noise sample

In order to make the pluck-string synthesizer, firstly, a burst of white noise had to be made, as seen in figure 3.1. This was done by setting the amount of samples to contain noise, and the total amount of samples in the audio. An input signal were then created with the first noise amount of samples being randomised and the rest of the samples being zeros.

```
15  fs = 44100  # Samples per second
16
17  length = 1  # Length of audio file in seconds
18
19  totalSamples = fs * length  # Total amount of samples in audio file
20  totalSampleArray = np.int(totalSamples)
21
22  noise = 200
23
24  inputSignal = np.r_[np.random.randn(noise), np.zeros(totalSampleArray - noise)]
```

Figure 3.1: Line 15-24 of the program code, variables and white noise input signal

3.2 Delays and coefficients

Figure 3.2 shows the function called when clicking a button in the GUI. This function takes three frequencies as the formal parameters and calculates the amount of samples per period. The amount of samples per period is used to calculate the comb delay, so the overlap between the periods is as small as possible. This delay is then used to make the allpass filter coefficient which is always between zero and one. The comb filter coefficient is always 0.99.

The three frequencies with three different coefficients and delays, are then passed individually into the karplus strong function and added together to make a chord.

```
def clicked(freq1, freq2, freq3):
    period1 = fs / freq1  # Samples per period
    period2 = fs / freq2  # Samples per period
    period3 = fs / freq3  # Samples per period

    filterCoefficient = 0.99    # Comb filter coefficient

    combDelay1 = np.int(np.floor(period1 - 0.5))                    # Comb Delay
    fracDelay1 = period1 - combDelay1 - 0.5                         # Fractional delay used to calculate all pass coefficient
    allPassFilterCoefficient1 = (1 - fracDelay1) / (1 + fracDelay1)  # All pass coefficient based on the fractional delay

    combDelay2 = np.int(np.floor(period2 - 0.5))                    # Comb Delay
    fracDelay2 = period2 - combDelay2 - 0.5                         # Fractional delay used to calculate all pass coefficient
    allPassFilterCoefficient2 = (1 - fracDelay2) / (1 + fracDelay2)  # All pass coefficient based on the fractional delay

    combDelay3 = np.int(np.floor(period3 - 0.5))                    # Comb Delay
    fracDelay3 = period3 - combDelay3 - 0.5                         # Fractional delay used to calculate all pass coefficient
    allPassFilterCoefficient3 = (1 - fracDelay3) / (1 + fracDelay3)  # All pass coefficient based on the fractional delay

    # Creates an output signal as a note based on the Karplus Strong method
    outputSignal = karplusStrong(inputSignal, combDelay1, filterCoefficient, allPassFilterCoefficient1, totalSampleArray)
    # Adds another note
    outputSignal += karplusStrong(inputSignal, combDelay2, filterCoefficient, allPassFilterCoefficient2, totalSampleArray)
    # Adds another note
    outputSignal += karplusStrong(inputSignal, combDelay3, filterCoefficient, allPassFilterCoefficient3, totalSampleArray)
    # The three notes combined makes a chord
    generateNoiseFile(outputSignal, fs)  # Generate sound file
```

Figure 3.2: Clicked function in the code that show how the coefficients and delays are calculated.

3.3 Karplus Strong

Figure 3.3 shows the karplus strong algorithm used in the program. First four arrays is initialised as these are used inside the for loop. The for loop goes through the input signal and applies the comb, lowpass and allpass filters to index i and returning the new manipulated output signal, which is then played back to the user afterwards.

```
def karplusStrong(signal, delay, coefficient, allPassCoefficient, sampleSize):
    sampling = np.arange(sampleSize)
    output = np.zeros(sampleSize)
    lowPass = np.zeros(1)
    allPass = np.zeros(2)

    for i in sampling:
        if i < delay:
            tempOutput = signal[i]
        else:
            tempOutput = signal[i] + coefficient * output[i - delay]
        allPassInput = 0.5 * tempOutput + 0.5 * lowPass
        lowPass = tempOutput

        output[i] = allPassCoefficient * (allPassInput - allPass[1]) + allPass[0]
        allPass[0] = allPassInput
        allPass[1] = output[i]

    return output
```

Figure 3.3: KarplusStrong function of the code, that shows the implementation of the comb, the lowpass and the allpass filter.

3.4 GUI

The GUI is a very simplistic layout made with TKinter, with six buttons that make the user able to choose which Major chord they want to play.

This is done by creating a window, naming it, and specifying six buttons with text corresponding to which chord will be played. When clicking a button it calls the clicked() function and specifies which three frequencies it should use, and those frequencies corresponds with the notes from section 2.3.1.

Figure 3.4 shows an image of the GUI. It should be noted that the GUI is made to give an overview of the notes to be played, with focus on functionality.

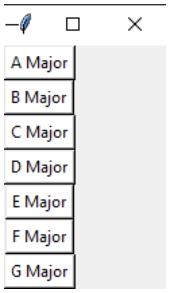

Figure 3.4: Image of the GUI in run-time.

Chapter 4
Possible improvements

While the Karplus Strong Synthesizer described in this report does work, it is not guaranteed to be in tune of what is be expected from a guitar. This is most likely due to the input noise signal being randomised every time the program runs. Also if the user wants to tune the guitar by changing the different variables, like the comb delay, they need to do it in the code, as this can not be done during run-time through the GUI.

Creating a tuner through the GUI, as well as making the user able to change coefficients, would make the program more flexible as it would thereby be possible to change the noise into different types of plucked-string instruments. Choosing which notes to combine into chords would also help improve the program. All of these possible improvements would however require extensive work and is out of scope for this mini-project.

Bibliography

[1] Mads G. Christensen. *Introduction to Audio Processing*. Springer, 2019. ISBN: 978-3-030-11781-8.

[2] *Guitar Chords*. English. June 2020. URL: https://www.guitar-chord.org/chords-by-notes.html (visited on 06/05/2020).

[3] *Guitarboat.com*. English. June 2020. URL: https://guitarboat.com/acoustic-guitars/learn-about-acoustic-guitars/ (visited on 06/05/2020).

[4] Jesper Kjær Nielsen. *Lecture 5: Comb filters and periodic signals*. Jupyter Notebook. 20. Feb. 2019.