GUITAR STRING SIMULATION 2

Electrical Engineering 20N
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

HSIN-I LIU, JONATHAN KOTKER, HOWARD LEI, ANDREW LEE, AND BABAK AYAZIFAR

1 Introduction

In this mini-project, we will bolster the functionality of the guitar string simulation that we created in lab 06, by using several filters in series to construct a system that can delay its input signal by any arbitrary fractional amount. With this in hand, we will be able to construct a model of a guitar string that can vibrate at any real frequency we provide, thus allowing us to produce the guitar version of any song we choose. Along the way, we will learn about, and discover the uses of, systems with linear phase responses [1].

1.1 Mini-Project Goals

- Understand the concept and the applications of linear phase responses.
- Explore relationships between filters and delays, both integer and real.

1.2 Checkoff Points

2 Delay Elements and Linear Phases

Consider a system that delays its input by N samples. In other words, consider the system $\mathsf{D_N}$ with LCCDE $y(n) = x(n - N)$. We know that the frequency response of this system is given by $D_N(\omega) = e^{-i\omega N}$ (Why?) The phase of delay elements is thus a linear function of ω. Since any LTI system is uniquely determined by its frequency response function, we deduce that every system with a frequency response of the form $e^{-i\omega N}$ must be a delay element.

However, in our analysis, we made no assumptions as to what type of number N was: the fact that we were in discrete-time implied that N should be an integer, but this analysis is equally applicable if N were *any* real number. Of course, it does not make much intuitive sense to talk about delaying a discrete-time signal by a non-integer number of samples, so we should expect that, if any system induces a non-integer delay, the output may not look like a shifted version of the input. In fact, we have already seen one such system before; we just called it a low-pass filter. (*Woah*.)

The LCCDE of the particular low-pass filter used in the in-lab section was

$$\forall n \in \mathbb{Z}, \quad y(n) = \frac{1}{2}(x(n) + x(n-1)),$$

also known as the moving average filter. We know that its frequency response is

$$F_L(\omega) = \frac{1}{2}(1 + e^{-i\omega}).$$

Simplifying this further, we see that

$$\begin{aligned} F_L(\omega) &= \frac{1}{2}(1 + e^{-i\omega}) \\ &= e^{-i\omega/2}\left(\frac{1}{2}e^{-i\omega/2} + \frac{1}{2}e^{i\omega/2}\right) \\ &= e^{-i\omega/2}\cos\left(\frac{\omega}{2}\right) \\ &= \cos\left(\frac{\omega}{2}\right)e^{-i\omega/2} \end{aligned}$$

Notice that the phase of a low-pass filter is linear with slope $-1/2$. Based on our prior analysis, we deduce that a low-pass filter 'delays' its input by half a sample. Again, it may not make too much intuitive sense as to what a half-sample delay implies in discrete-time, so we can stipulate that the moving average filter provides a good demonstration of what a half-sample delay does to a signal. The extra factor of $\cos\left(\frac{\omega}{2}\right)$ provides the filtering effect that allows the low-pass filter to attenuate higher frequencies.

3 Fractional Delays

During the in-lab sections of lab 06, we realized that an integer delay could not represent the basic tone A-440. In order to produce this tone, we will look at the slightly more complex all-pass filter to create the fractional delay that we need to produce the tone A-440. For the purposes of this mini-project, we will denote the total delay needed to produce A-440 by D, where D is a real number. The analysis involves a considerable amount of mathematics. If you have been studying for a while, we suggest you take a break and get a cup of hot chocolate, listen to some music, play a short game of your favorite sport, or do all of these simultaneously, before you proceed.

Consider a filter given by the following difference equation,

$$\forall n \in \mathbb{Z}, \quad y(n) + cy(n-1) = cx(n) + x(n-1), \tag{1}$$

for some constant $0 < c \leq 1$. As we determined in lab 05, the frequency response of this filter is given by

$$F_A(\omega) = \frac{c + e^{-i\omega}}{1 + ce^{-i\omega}}.$$

We could immediately proceed to plot the magnitude and phase of the frequency response $F_A(\omega)$ using `MathScript`, but instead, we will first manipulate the response formula further to get some insight. Being slightly tricky, we will multiply the numerator and the denominator by $e^{i\omega/2}$ to obtain

$$F_A(\omega) = \frac{ce^{i\omega/2} + e^{-i\omega/2}}{e^{i\omega/2} + ce^{-i\omega/2}}.$$

Now, notice that the numerator and the denominator are complex conjugates of one another. In other words, let

$$b(\omega) = ce^{i\omega/2} + e^{-i\omega/2} \tag{2}$$

and notice that

$$F_A(\omega) = \frac{b(\omega)}{b^*(\omega)}.$$

Since the numerator and denominator have the same magnitude, we find that

$$|F_A(\omega)| = 1.$$

The filter is an all-pass filter!

The phase response, however, is more complicated, but more interesting. Notice that

$$\angle F_A(\omega) = \angle b(\omega) - \angle b^*(\omega).$$

Since for any complex number z, $\angle(z^*) = -\angle z$, we have

$$\angle F_A(\omega) = 2\angle b(\omega).$$

Thus, in order to find the phase response, we simply need to determine $\angle b(\omega)$. Plugging Euler's relation into Equation 2, we get

$$b(\omega) = (c+1)\cos\left(\frac{\omega}{2}\right) + i(c-1)\sin\left(\frac{\omega}{2}\right).$$

Since the phase of a complex number z is $\tan^{-1}\left(\frac{\mathsf{Im}\{z\}}{\mathsf{Re}\{z\}}\right)$, we deduce that

$$\angle F_A(\omega) = 2\tan^{-1}\left(\frac{(c-1)\sin(\frac{\omega}{2})}{(c+1)\cos(\frac{\omega}{2})}\right),$$

or alternatively,

$$\angle F_A(\omega) = 2\tan^{-1}\left(\frac{c-1}{c+1}\tan\left(\frac{\omega}{2}\right)\right).$$

In this form, the formula for the phase response yields insight for small ω. In particular, when ω is small (compared to π),

$$\tan\left(\frac{\omega}{2}\right) \approx \frac{\omega}{2},$$

and so

$$\angle F_A(\omega) \approx 2\tan^{-1}\left(\frac{c-1}{c+1}\cdot\frac{\omega}{2}\right).$$

Since $0 < c \leq 1$, the argument to the arctangent is small if ω is small. Hence, for low frequencies,

$$\angle F_A(\omega) \approx 2\left(\frac{c-1}{c+1} \cdot \frac{\omega}{2}\right) = \frac{c-1}{c+1}\omega = -d\omega,$$

where d is defined by

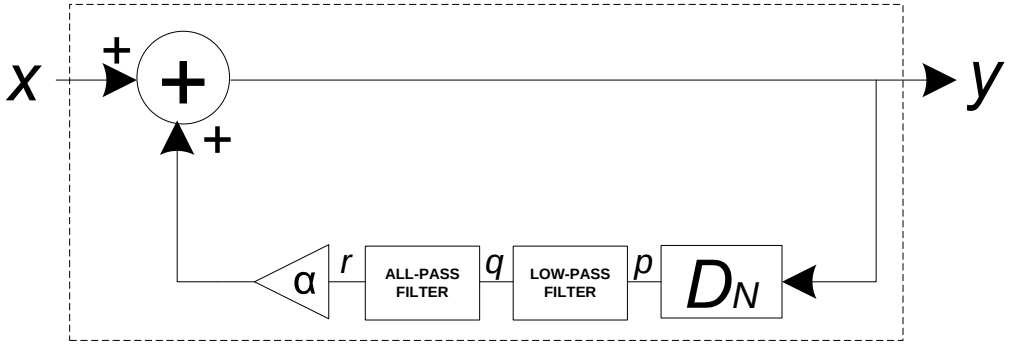
$$d = -\left(\frac{c-1}{c+1}\right). \tag{3}$$

Thus, at low frequencies, this all-pass filter has a linear phase with slope $-d$, which means that at low frequencies, the filter behaves exactly like a delay element. However, unlike the N sample delay, the amount of delay is d, which, depending on c, can be any real number between 0 and 1. Hence, amazingly, the all-pass filter gives us a way to get fractional sample delays in a discrete time system, at least for low frequencies.

4 Finally A-440

In this section, we will use the all-pass filter analyzed in section 3 to finally generate that elusive A-440 tone, which has a fundamental frequency of 440 Hz. We will do this by splicing the filter into the feedback loop of the comb filter, as shown in Figure 1.[1]

Figure 1 Low-Pass Filter and All-Pass Filter embedded into a Comb Filter.



1. Determine the LCCDE that represents the block diagram shown in Figure 1, solely in terms of the input signal $x(n)$ and the output signal $y(n)$.

   You may find the intermediate signals $p(n)$, $q(n)$, and $r(n)$, defined as in Figure 1, useful. Also, note that $x(n) + \alpha r(n) = y(n)$, or that $r(n) = (1/\alpha)(y(n) - x(n))$; this relationship may be useful as well.

2. Recall that the comb filter produces an **integer** delay N, the low-pass filter produces a delay of 0.5, and the all-pass filter produces a **fractional** delay d that **lies between 0 and 1**.

   (a) Given a sampling frequency of **44.1 kHz**, what is the *total* delay D needed to generate the A-440 signal? You will need the relationship between f_0, f_s, and N that you determined in the pre-lab section of lab 06, where f_0 is the fundamental frequency of the signal that we are trying to generate. Note that we are using a different sampling frequency this time!

[1] If you are doing this post-lab on a computer with a Mac operating system, the LabVIEW version for Mac, unfortunately, does not have a `Play Waveform` block of its own. The efforts of Hsin-I, however, have led to a sub-VI that functions very well for this post-lab section, and is available on the course website, as part of the resources for this lab. In contrast to the version for Windows, this block can only support a finite number of sampling frequencies. Use a control or a constant to set the sampling frequency among the ones available; for the purposes of this post-lab, you will use a sampling frequency of **44.1 kHz**.

(b) Determine the best values of N and d that will cause the overall system shown in Figure 1 to generate the A-440 signal. We will do this by recognizing that the total delay of all of the filters is given by $N + d + (1/2)$, with the delay of $1/2$ due to the low-pass filter. This total delay is equal to the delay D that you derived in the previous step. Then, determine values for N and d, remembering the restrictions on N and d.

(c) From the value of d you determined, what is the corresponding value of c? c is the constant used in the LCCDE for the all-pass filter, as seen in Equation 1, and is related to d as shown in Equation 3.

3. Copy the contents of the `Guitar String` folder, created in lab 06, into a new folder called `Guitar A440`. You may need to temporarily convert the `Guitar String` LLB file back into a folder along the way.

4. Create a new VI called `APF for CF` in the `Guitar A440` folder; this VI will be the sub-VI that represents the all-pass filter inserted into the feedback loop of the comb filter. The specifications for this VI are shown in Table 1.

Table 1 Specifications for the `APF for CF` VI.

Terminal	**Type**	**Function**
`Input Signal Sample`	**Scalar** Input	Value of q at a given sample.
`c`	**Scalar** Input	Constant c for the all-pass filter.
`Output Signal Sample`	**Scalar** Output	Value of r at a given sample.

5. Create a descriptive icon to represent your all-pass filter.

6. Create a new VI called `Guitar A440`, using the `Guitar String` VI as a template. Insert the `APF for CF` sub-VI into the `For Loop` representing the comb filter with the low-pass filter, in the `Guitar String` VI, as shown in Figure 1, and make the proper connections. Also, the input `c` should have a numeric control.

7. Run your VI with $\alpha = 0.99$, $p = 22050$ and the values of N and c that you determined in step 2. If done correctly, your VI should now produce a sound of frequency 440 Hz. For comparison, you can go to `http://www.onlinetuningfork.com/` and play the 440 Hz sound (the middle tuning fork).

5 Shall We Play a Song?

You have created a very close approximation to an actual guitar pluck sound, and you have even tuned it to a particular frequency. The next step is thus to generalize your guitar string model to a wider range of frequencies, and from this generalization, create and finally play a simple song in LabVIEW.

Since we have come this far in LabVIEW, the following exercises are not strictly guided, and you can do them in whichever manner you feel most appropriate and most comfortable. However, 10% of your grade for this lab will be based on the style of the block diagrams of the VIs you will be creating in this section, to encourage clean and well-commented VIs. As far as possible, please also use as many sub-VIs as you can, all with appropriate and well-labeled icons.

1. As a first step, we must generalize our guitar string model to handle any (low) frequency that we throw at it. Using the `Guitar A440` VI as a template, create a sub-VI called `Guitar Variable Frequency`, whose specifications are described in Table 2.

   Several points of note:

(a) The output of this subVI will be an **array**, not a waveform, so you need not convert the array output of the `For Loop` block, representing the comb filter, into a waveform. You will perform this step later.

(b) Set α to be 0.99, so that you no longer need an input for α.

(c) Hard-code the `Samping Frequency` to be 44100 samples per second.

(d) The VI is only provided with the frequency of the output signal that it needs to generate. However, the values of N and d can be easily derived from `Frequency` and `Sampling Frequency`: how? You may find the `Round Toward -Infinity` block, located under `Mathematics` → `Numeric`, useful.

(e) Once you obtain the value of d, do not forget to determine the proper value of c, since the all-pass filter needs the constant c, and not d.

(f) If you will be using the `Y[i] = X[i - n] PtByPt` block, this point is especially important. As per the implementation of the block, the information for the delay N is collected and fixed at the beginning of the execution of the whole program, unless other initialization criteria are stated. Since we will be dealing with the same VI multiple times, each time with a different delay, this property is undesirable. To avoid this, set the `initialize` terminal to be true whenever the iteration count `i` is zero; this causes the block to re-initialize with a new value of N every time the program is rerun.

(g) Remove any surrounding `While Loop`s, or else the sub-VI could run forever!

Table 2 Specifications for the `Guitar Variable Frequency` VI.

Terminal	**Type**	**Function**
`Frequency`	**Scalar** Input	Frequency of signal to be generated.
`Samples`	**Scalar** Input	Number of samples to be generated.
`Output Signal`	**Array** Output	Output signal generated based on the input parameters. **If the frequency required is 0 Hz, then the output signal should merely be an array of zeros.**

2. From step 1, you have obtained a sub-VI that can produce an output signal with any (low) frequency you need, for however long you need it to be. In this part of the mini-project, we will provide you with an intuitive framework with which you can play any set of notes in sequence using your `Guitar Variable Frequency` sub-VI.

3. From the course website, download the `MusicFrameworkWithGuitar` LLB, and examine the block diagram of the `Player` VI. The `Player` VI takes as an input a `.lv` file that specifies a set of notes, durations, and loudnesses, and plays them in sequence. Note that it currently uses the `SineNote` sub-VI to generate the sound that corresponds to a particular frequency and duration. The sound that it generates does not sound guitar-like (quite boring, actually): we will fix that.

4. Before plugging in your `Guitar Variable Frequency` sub-VI, we will run `Player.vi` with the `SineNote` sub-VI and a `.lv` file input to make sure that the VI works. Download the file `test.lv` from the course website. The `.lv` file can be read by any simple text-editor, such as `WordPad`. It holds the tones, number of samples, and velocities (loudnesses) of a set of notes, and has the following format:

```
Total_Samples

Tone            Samples  Velocity
Tone            Samples  Velocity
Tone            Samples  Velocity
⋮               ⋮        ⋮
```

The first line specifies the total number of samples included in the entire duration of the set of notes to be played. Each row specifies a separate note, and the notes are read by the `Player` VI from top to bottom in sequence. The tone of each note corresponds to its *relative* position on the chromatic musical scale. The `Player` VI automatically converts the tone to its corresponding frequency.

> A chromatic musical scale is a musical scale where each octave is divided equally into 12 tones. In general, a note is said to be an **octave** below another note with twice its frequency. Here, however, the word 'octave' is used in another sense: as a range of notes between a pair of notes that are separated an octave apart. The same tone from different octaves correspond to either a doubling (for higher tones) or halving (for lower tones) of frequencies. For the `.lv` file format, each tone is represented by an integer, where consecutive integers represent adjacent tones on the chromatic scale. Wikipedia provides supplemental discussion.

Note that the duration of each note is determined by the number of samples of that note, along with the preset sampling frequency of 44100 samples per second. Hence, 44100 samples of one note would last exactly 1 second.

5. If the total number of samples of all the notes exceeds the total samples specified on the top line, an additional track will be formed starting from the note where the number of samples exceeds the total samples specified. This can be repeated an infinite number of times, so as many tracks as needed can be included. Note that in `test.lv` there are three tracks, as the total number of samples corresponding to all the notes (1920000) is three times the total samples specified on the top line (640000). Hence, the set of notes comprising the first 640000 samples comprise the first track, the set of notes comprising the next 640000 samples comprise the second track, and so on. The `Player` VI will generate music containing all tracks simultaneously.

6. In `Player` VI, click on the folder icon corresponding to the `Open .lv file` dialog box to load `test.lv`. Click `Synthesize!` to generate the audio, and then click the `Play` button once the `Ready` indicator is turned on. You should hear the music corresponding to the sequence of notes specified in `test.lv`. What song do you hear?

7. Convert the `MusicFrameworkWithGuitar` LLB into a directory, and add your `Guitar Variable Frequency` sub-VI into the directory. In the block diagram of the `Player` VI, replace the `SineNote` sub-VI with your `Guitar Variable Frequency` subVI, noting that both VIs should have the same input and output terminals.

8. Run the `Player` VI just as you did before. If your `Guitar Variable Frequency` sub-VI implementation is correct, you should hear the same set of notes, but sounding as if generated by a guitar.

9. Congratulations! You have just created a music-playing framework that allows you to play a given set of notes, simulating a musical instrument whose sound you have digitally re-created, merely by employing principles of signal processing and design.

6 Generate Your Own Song

Now that you have become familiar with the framework for inputting and playing a set of notes corresponding to a `.lv` file, generate a simple song using a separate `.lv` file, and play it using `Player.vi`.

It can be as simple or complicated as you want it to be, with either one or multiple tracks. However, it should be something that is easily recognizable with a certain title. We recommend that the length of the song be less than four minutes, or else your computer may run out of memory before it can finish synthesizing the song. Andrew Lee, a former student of the class, has designed a MIDI to LV converter, which you may find useful. It converts pre-existing MIDI files into their LV equivalents. It is located at `http://inst.eecs.berkeley.edu/~alee/EE20/Instructions.html`.

When you are done, convert the `MusicFrameworkWithGuitar` directory back into an LLB file. Be sure to submit both the LLB and the `.lv` file that you created for this mini-project.

7 Submission Rules

1. Submit your files *no later than* 10 minutes after the beginning of the lab session the week of April 18th, 2011.
2. Late submissions will *not* be accepted, except under unusual circumstances.
3. These exercises are recommended to be done *in groups of two*. Only one person need submit the required files, however.

8 Submission Instructions

1. Log on to bSpace and click on the `Assignments` tab.
2. Locate the assignment for `Mini-Project 2` corresponding to your section.
3. Attach the following files to the assignment:
   (a) A text file called `PARTNERS.txt`, containing a list of the students who worked together.
   (b) Your answer to step 1 of section 4 **as a comment** in the `Guitar A440` VI.
   (c) The `Guitar A440` LLB.
   (d) The `Music Framework With Guitar` LLB that includes the `Guitar Variable Frequency` sub-VI.
   (e) The `.lv` file that you created in step 6.

9 Acknowledgments

Special thanks go out to Vinay Raj Hampapur, Miklos Christine, Sarah Wodin-Schwartz, and Andrew Lee for providing suggestions, ideas, and fixes to this mini-project guide. We especially thank Andrew Lee for also providing the `MusicFrameworkWithGuitar` LabVIEW framework. This mini-project guide was based on, although substantially modified from, the "Plucked string instrument" laboratory exercise as presented in the book `Structure and Interpretation of Signals and Systems`, written by Edward A. Lee and Pravin Varaiya (ISBN 0201745518). The A440 sound sample was obtained from the `A440` entry on Wikipedia, under the Creative Commons license.

References

[1] K. Karplus and A. Strong. Digital Synthesis of Plucked-String and Drum Timbres. *Computer Music Journal*, 7(2):43–55, Summer 1983.