Audio Signal Processing Laboratory #1: Physical Modeling of Strings

Intro Video: https://youtu.be/JkuxYmjKsYE

1

What to Submit:

- Your code and Simulink block diagrams. Make sure you are saving all Simulink models to your Current Folder in MATLAB
- A report with answers, figures, and explanations
- Any sound files that are asked for
- When to Hand It In:
  - It is due immediately at the beginning of the day (i.e., 0:00AM) specified on the course calendar. **Late assignments will receive a grade deduction of 20% of the full grade each day.**

2

How to Submit:

- Put all your files in one folder. Compress this folder and name it <firstname>_<lastname>_HW6 .zip , ie “Zhiyao_Duan_HW6.zip”.
- Submit via blackboard to the appropriate entry. You can make multiple attempts but only the last attempt will be graded.
- **New to these labs:** Record a video of yourself demonstrating the applicable parts so they can be graded remotely.

  An example demonstration video is provided here: https://youtu.be/Ji8ov3MhLQQ

If you don’t have a computer that can screen-record or a phone that can take videos, let us know ASAP

3

Problem 0: Object Oriented Programming (OOP) in MATLAB

This problem is **optional**, but may be useful for those who have never done OOP in MATLAB before

Video Tutorial: https://youtu.be/FxMeBtp7zGc

4

In this problem, we will be making a *class* to represent a dog (animal examples are standard in first OOP examples). Then, we will use this class file to create an *instance* of a dog object and use this instance to demonstrate the usefulness of objects in MATLAB.

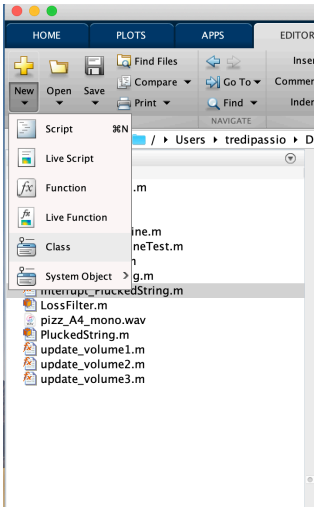


Dogs are a real-world object

Goals: we want to be able to examine a ”dog object” and ascertain its name, age, fur color, and whether it’s awake or asleep. We also want the dog to do certain commands, like sleep, wake up, speak, and change names.

5

Part A) Make a new class file in MATLAB



Use the “New” button in the MATLAB window (available in the Home or Editor tab) and create a new Class file

6

Part B) Name the class "Dog"

```
classdef Dog
    %UNTITLED8 Summary of this class goes here
    %   Detailed explanation goes here

    properties
        name;
    end

    methods
        function obj = untitled8(inputArg1,inputArg2)
            %UNTITLED8 Construct an instance of this class
            %   Detailed explanation goes here
            obj.Property1 = inputArg1 + inputArg2;
        end
```

To name the class, change the line right after the "classdef" to the desired name of your class, in this case, Dog

7

Part C) Make the dog's properties

```
classdef Dog

    properties
        name;
        age;
        fur_color;
        sleep_state;
    end
```

The properties of a class are items that are saved to *instances* of a class. We will talk more about what an *instance* means in a few slides. These properties are items that we can access within a class index, or use them in functions associated with the class, also called *class methods.*

In this simple example, we want each dog to have a name, an age, a fur color, and a sleep state. At this phase, we don't **make these properties equal to anything**, we're just letting MATLAB know we will set them equal to something later on.

8

Part D) Make the *Constructor*

Every class needs what is called a constructor. This function/method will return an object (by convention called obj), which is then called an *instance* of the class.

The constructor should take in several inputs in order to assign values to the classes properties. In this case, we want to **construct a dog** so that the dog has a name, an age, a fur color, and a sleep state.

The name of the constructor is the same as the name of the class, and the return is called obj. We can set class properties with the syntax obj.PROPERTY = …

```
methods
    function obj = Dog(input_name,input_age,input_fur_color,input_sleep_state)
        obj.name = input_name;
        obj.age = input_age;
        obj.fur_color = input_fur_color;
        obj.sleep_state = input_sleep_state;
    end
```

9

Part E) Make the methods

Methods are functions associated with a given class. The first input argument for each method is **reserved** for the keyword ”obj” (this just means don’t change this). If you write a new method, you need to make obj the first argument. Methods *act on an instance of a class*, meaning we can use methods to do calculations based on the object’s properties, or update an objects properties.

The next four slides show four example methods.

10

i) Make a "speak" method

Similar to hello world, we want to make a method that prints out a message in the form: "The dog named <NAME> says bark!". As such, this method doesn't need any input arguments (**except obj**) and doesn't need to return anything. We will access the dogs name with **obj.name**, and use an array to package the dog's message:

```
function [] = speak(obj)
    message = ['The dog named ' obj.name ' says bark!'];
    disp(message);
end
```

11

ii) Make a "go_to_sleep" method

Let's make a method that very simply changes the Dogs sleep_state property to "asleep", regardless of it's current state. Once again, we don't need any inputs except obj or any outputs to this method:

```
function [] = go_to_sleep(obj)
    obj.sleep_state = 'asleep';
end
```

12

iii) Make a "wake_up" method

The opposite of the previous method, this will wake the dog up regardless of its current sleep state:

```
function [] = wake_up(obj)
    obj.sleep_state = 'awake';

end
```

13

iiii) Make a "change_name" method

This time, we will need an input argument other than obj, since we need to update the dog's name to a new name. We still don't need an output though.

```
function [] = change_name(obj,new_name)
    obj.name = new_name;

end
```

14

Part F) Test your class

Run the following lines of code in the Command Window to demonstrate functionality:

```
% Make an instance of a dog using the constructor:
dog1 = Dog('Albert',3,'red','awake');
% print out the dog's properties:
dog1.name
dog1.age
dog1.fur_color
dog1.sleep_state
% Make the dog speak:
dog1.speak()
% [NOTICE, we use (args) to call methods, just like functions
% Make the dog sleep:
dog1.go_to_sleep();
% Make sure it went to sleep:
dog1.sleep_state
% Change the dogs name:
dog1.change_name('Peter');
% Print the dogs new name, make it speak
dog1.name
dog1.speak()
% Make a second dog instance:
dog2 = Dog('Alice',3,'blue','asleep');
% Make the second dog speak
dog2.speak()
```

15

Problem 1: DelayLine and FractionalDelayLine Classes

This problem is **NOT optional**, and involves the implementation of the above classes

Video Tutorial: https://youtu.be/kgGN8Tcfyf8

16

Part A) DelayLine

(1 point)

Fill in blanks in *DelayLine.m*. This class implements a delay line with a fixed delay. It is required that you implement a specialized circular buffer internally for the delay line. In the *process* method/function, each time one sample is processed and the state is updated. A script *delayLineTest.m* can be used to test some basic cases of your implementation. It also shows you the basic usage of the intended interface. Currently, it only checks if a correct delay is produced. Feel free to add test cases to the unit test.

17

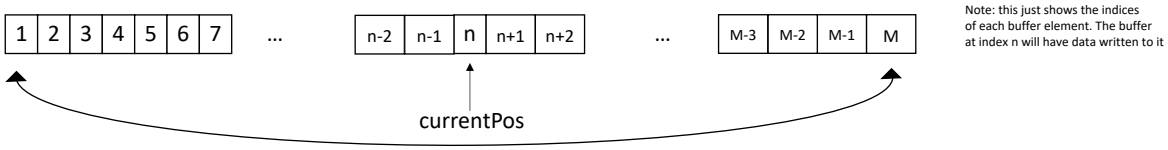
As discussed in lecture, a DelayLine serves to delay an input audio sample by a set number of samples before it is output and played. For instance, a delayline with length M will delay the output by M samples.

Let's say we have an incoming audio string x, indexable by sample number (ie, the first audio sample is x[1], then x[2], … x[n], x[n + 1] …). At current time n, a DelayLine will first output x[n – M], the audio sample M samples ago.

For efficiency reasons, we will only store a total of M samples at a time within the DelayLine object, and then read and write samples into a *circular buffer* so that we can get the proper output. The next slide talks about this implementation.

18

The given properties of the DelayLine are a buffer of size delayLineLength (sometimes called M) which is initialized by ”zeroes” and a currentPos *pointer*. The following are the buffer indices, where n represents an arbitrary buffer position:



Every time time DelayLine “process” method is called on an instance of a DelayLine, the following happens:

1. The buffer element at the current position is set to be the output (notice the output of the process method is called xDelayed)
2. The incoming sample (notice the input argument not named ‘obj’ to the process function is called x) overwrites the buffer element at the current position
3. The currentPos is incremented by 1 circularly, ie, if the currentPos tries to go above M it circles back to the beginning

19

Part B) FractionalDelayLine (1 point)

Fill in blanks in *FractionalDelayLine.m*. This class implements a fractional delay line with a fixed delay. You need to implement the *process* method/function for both the linear interpolation method and the first-order all-pass interpolation method. A script *fractionalDelayLineTest.m* provides a unit test for this class. Feel free to add test cases to the unit test.

Note: In your implementation, you will need to split a large delay quantity into an integer and a fractional part. Also, for firstOrderAllpass interpolator, the fractional part works better (more flat phase delay) when $\Delta \in [0.3, 1.3]$, therefore, for firstOrderAllpass, adjust the split between integer part and fractional part to make $\Delta \in [0.3, 1.3]$.

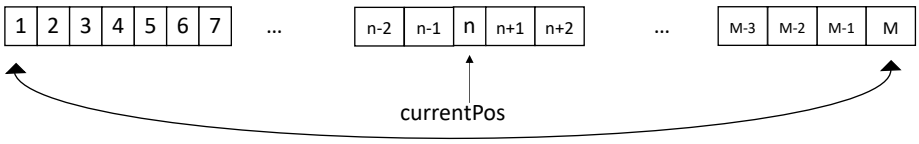
20

The FractionalDelayLine has the same concept as the DelayLine, except with a non-integer number of delay samples. Because we can only store things in integer-length arrays, some interpolation is needed to effectively store fractional data. Notice the ”interptype” property of the FractionalDelayLine class. If this is set to 0 in the classes constructor method, linear interpolation is used. If this is set to 1, all pass interpolation is used.

Notice also that the constructor has been written for you. This means that the ”delta” and “eta” properties have been calculated already, though you should look through the code and lecture notes to make sure you understand these. The next slide talks about the implementation.

21

Linear Interpolation: If the interpType property is equal to 0, we use linear interpolation. Once again, a circular buffer will be used, as shown below:

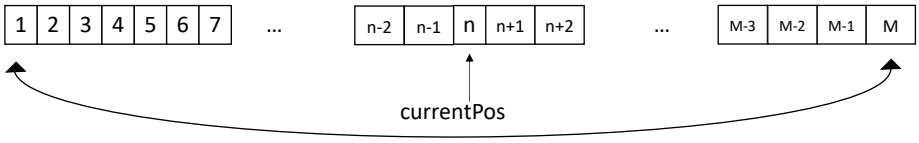


You may have noticed that the length of the buffer is equal to the integer component of the buffer length **plus one**, meaning that this time, the current position of the pointer is pointing to, effectively, x[n-M-1]. Therefore, again using circular shifting, x[n-M -1] and x[n-M] can be obtained from the buffer and the currentPos property. The equation given in the lecture notes can then be used to calculate the delayed output sample y[n]

y[n] = ((1-delta)*x[n-M]) + (delta*x[n-M-1])

22

All Pass Interpolation: If the interpType property is equal to 1, we use all pass interpolation. Once again, a circular buffer will be used, as shown below:



The all pass interpolation also makes use of the previous output of the system, which will be changed each time the process function is called with this interpolation type. We've discussed how to obtain x[n-M-1] and x[n-M], and eta and prevOutput are known are accessible as properties of the class. Therefore, we can compute the delayed output sample y[n] from the equation in the lecture notes:

y[n] = (eta*(x[n-M] – y[n-1])) + x[n-M-1]

Where y[n-1] is the prevOutput property. After the output sample is calculated in the process function, that output sample then becomes the new prevOutput.

23

Problem 1 Deliverables

- Implemented DelayLine.m (1 point)
- Running delayLineTest gives "all tests passed"
- Implemented FractionalDelayLine.m (1 point)
- Running fractionalDelayLineTest gives "all tests passed"
- **In your report:** make sure you answer all questions posed in this PowerPoint, and give an explanation of what you did and why you did it

24

Problem 2: LossFilter class and genPluckShape function

This problem is **NOT optional and** involves implementing the above class and helper function. These are the final building blocks needed for the physical string model

Video Tutorial: https://youtu.be/i8jjZ0mVlJE

25

Part A) LossFilter (0.5 points)

Complete the class *LossFilter.m*, which implements the simplest loss filter:

$$H_{loss}(z) = g\frac{1+a}{1+az^{-1}},$$

where ($0 < g < 1$) and ($-1 < a \leq 0$). Every time the *process* method/function is called, one sample is processed and the previous output (zm) is updated. Implimentation is disscussed in the next slide

26

In order to analyze the z-domain transfer function in the sample domain, it is pertinent to remember that a transfer function H(z) has a physical meaning of the output Y divided by the input X, or H(z) = Y(z) / X(z). Therefore, we can re-write the equation from the previous page as follows:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{g*(1+a)}{1 + a*z^{-1}}$$

This is then a separable equation, that we can split into the following:

$$Y(z) * (1 + a*z^{-1}) = X(z) * g*(1+a)$$

We can then transform this back to the sample-domain, remembering that z^{-1} in the z-domain is a delay of 1 sample in the sample-domain.

$$y[n] + a*y[n-1] = g*(1+a)*x[n]$$

Solve this for y[n], and recall that y[n-1] is the previous output and x[n] is the current input

27

It is noted that for sample-by-sample real-time time-domain processing, we need to convert z-domain transfer functions to time-domain difference equations. Using filter functions requires frames of audio to be analyzed instead of samples, and can slow down overall implementations

28

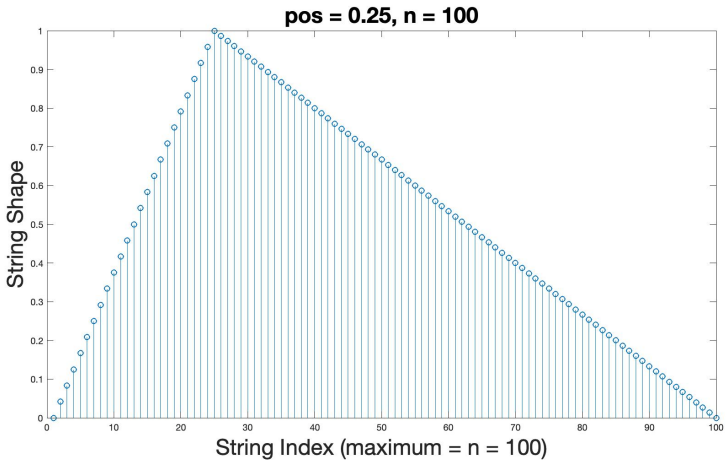
Part B) genPluckShape

(0.5 points)

Complete the function *genPluckShape.m*, which generates the plucking shape given its position ($pos \in [0,1]$) and the length (in samples) of the shape. For simplicity, the plucking position is rounded to integer samples in this function

29

Hint: To figure out what to put in each array index, use the (rounded) pluck point and an arbitrary pluck height of 1 to get the slope of the string shape in the two regions. An example of a 0.25 pluck point is shown below



30

Problem 2 Deliverables

- Implemented LossFilter.m (0.5 points)
- Implemented genPluckShape.m (0.5 points)
  - Include stem plots of a few 'pos' and 'n' variations in your report to verify accuracy of implementation
- **In your report:** make sure you answer all questions posed in this PowerPoint, and give an explanation of what you did and why you did it

31

Problem 3: PluckedString Class

This problem is **NOT optional and** involves using the previous helper functions and classes to make an overall PluckedString Class

Video Tutorial: https://youtu.be/yheuRy5VX4k

32

Part A) PluckedString (2 points)

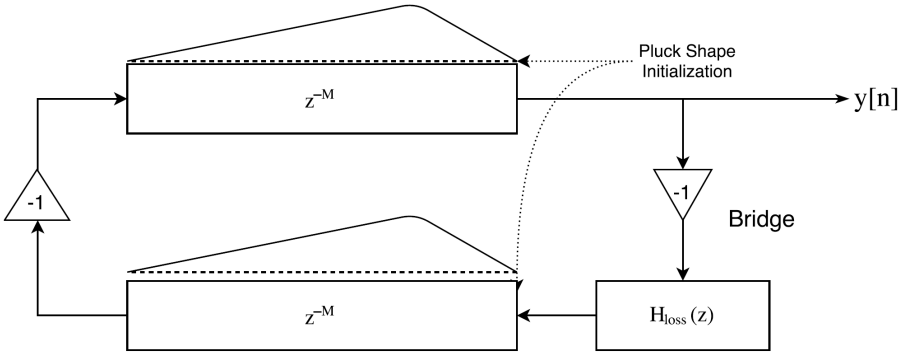
Fill in blanks in *PluckedString.m*. This class implements a physical model of a plucked string as shown in the block diagram on the next page.

The properties have been provided, as well as some of the constructor. Finish the constructor (where comments of "FILL IN" are left for you) and write the process function, which will utilized the process functions of the previous classes. We also want this model to reset itself every few seconds, you will also need to complete the reinitialization (reinit) method.

33

Items in **Bold** are class properties

delay_fwd: A FractionalDelayLine who's buffer is initialized to a **string_shape** generated by genPluckShape



delay_rev: A FractionalDelayLine who's buffer is initialized to the *reverse, or fliplr()* of the **string_shape**

loss_filt: a LossFilter specified by some **a** and **g** which are inputs to the constructor

34

The PluckedString is a powerful class as it ties several helper classes together. As such, it has several properties, which are reproduced below. Read and understand them:

```
properties
    % Top level blocks:
    delay_fwd;  % fractional delayline containing string shape
    delay_rev;  % fractional delayline containing flipped string shape
    loss_filt;  % loss filter in plucked string model

    % Loss filter items:
    a;
    g;

    % String model items:
    string_frequency;       % frequency of string we're modeling
    fs;                     % Sampling frequency for playback

    % String shape items:
    pos;                    % position of pluck point
    string_shape;           % shape of string returned by genPluckShape
    M;                      % length of fractional delay lines

    % Simulation items
    counter;                % number of samples processed
    duration_seconds;       % desired duration of playback in seconds
    duration_samples;       % desired duration of playback in samples
    prev_output;            % previous output of the string model
end
```

35

The Constructor: The inputs to the constructor are provided in the given code to be the frequency of the string we're modeling (f_in), the sampling frequency for playback (fs_in), the a_in and g_in of the LossFilter, the dur_in which tells the string how long to play in seconds before reinitializing, and the position of the pluck pos_in.

Most of the items you need to fill in are procedural, and will test your understanding of OOP. The first item, however, requires some thinking. Based on the sampling frequency, how long should the fwd and rev buffers be such that the total trip taken by a sample around the block diagram gives you the appropriate string frequency?

36

The Re-init: Every time the process functions runs, you will increment the counter property by one. At the end of the process function, you will check to see if it is time to re-init (which will occur when the counter is equal to the duration of the playback in samples). If it's time to re-initialize, this function is called to reset the forward and reverse delay lines and the loss filter, as well as the counter. Use your knowledge of OOP to reset the parameters listed in this function to exactly how they were after calling the constructor

37

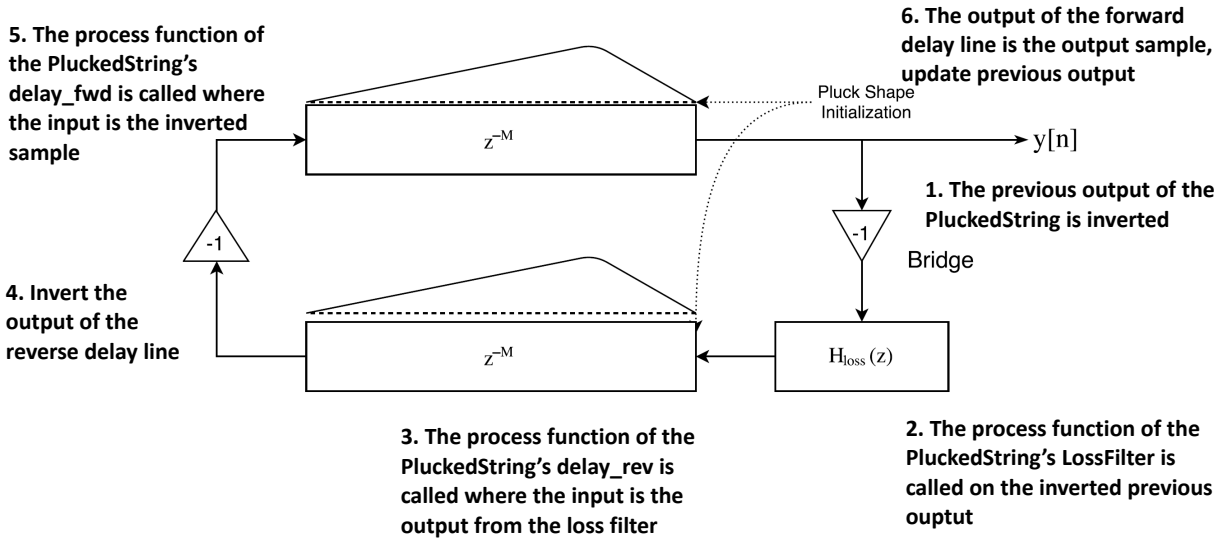
The process function: Two important things to note before implementing this function:

1. There is no *overall input sample.* Because this is a physical model of a string, we just want to playback a sample generated by the model every 1/44100th of a second (assuming the fs = 44100 Hz, otherwise, every 1/fs seconds). We will do this outside of real time in MATLAB and in real-time in Simulink

2. The process function that's been provided is started but incomplete. The steps are broken out into different lines of code, such that a running 'output' value is overwritten at each step. This is one strategy of sequential programming languages like MATLAB

38

Implementation: Simply walk through the block diagram, reproduced below:



39

A framework for doing this process function is provided, simply implement the steps as they are shown in the code. (Hint: remember to re-initialize if the counter reaches the playback duration! Should the duration here be in seconds or samples?)

```
%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Your codes Here %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%

output = obj.prev_output;
output = -1 * output;       % step 1
output = % FILL IN          % step 2
output = % FILL IN          % step 3
output = % FILL IN          % step 4
output = % FILL IN          % step 5
xDelayed = output;          % step 6
obj.prev_output = output;

% Update the counter, and check to make sure
obj.counter = obj.counter + 1;
if obj.counter == % FILL IN, hint, what property should we
                  % check to see if the counter has reached?
    obj.re_init();
end
```

40

Part B) 6 Second Recording of String (1 point)

Fill in the blanks in *pluckedStringTest.m* to generate a 6-second long A4 (440 Hz) tone with a sampling rate of 44.1 kHz and a singular pluck duration of 2 seconds (you should hear three plucks in your output audio). Save it as note_440hz.wav. You can pick any a and g for the loss filter, and you are encouraged to try different combinations of $a, g, and\ pos$. Comment on these variations briefly in your report.

At this point, as an optional exercise, you can add some random noise to the initialization pluck shape within the PluckedString class to achieve timbral variations

41

Problem 3 Deliverables

- Implemented PluckedString.m (2 points)
  - Correctly Implemented constructor
  - Correctly Implemented reinit method
  - Correctly Implemented process method
- The audio file 'note_440hz.wav' sounds like a guitar that is being plucked three times (1 point)
- **In your report:** make sure you answer all questions posed in this PowerPoint, and give an explanation of what you did and why you did it

42

Problem 4: Realtime Implementation in Simulink

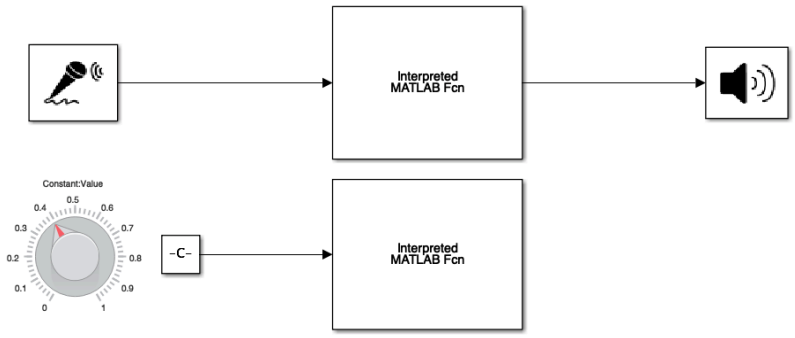
This problem is **NOT optional and** involves implementing three PluckedStrings in real-time in Simulink

Video Tutorial: https://youtu.be/onzsQVRB6gY

43

Intro to Simulink

Double-click the starter Simulink file *ThreeStrings_Simulink.slx*, which shows a framework for implementing one string in Simulink. You should see the following:



44

Hint: double click each block to see what it does and what parameters it takes

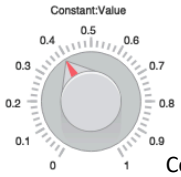
Audio Device Reader block, reads audio continuously from the selected audio device

MATLAB function from editor, in this case, *update_PluckedString1* that will be written in Part A

Audio Device Writer, writes audio so you can hear it to the selected audio device. For our purposes, this should be your default speakers



Knob block that controls the volume of string 1





Constant block, that is paired to the knobs output



MATLAB function from editor, in this case, *update_volume1*

45

The Problem: We want to make three strings that all play simultaneously in real time in the Simulink platform, each of which having its own volume control. The first string's framework is given in the provided Simulink file. The student must, in the same file, implement the second and third strings. Several functions containing **global variables** will be written to achieve this, explained in the following slides

46

The General Idea: Because the Audio Device Reader reads **frames** of audio in the desired sampling rate, the input to the MATLAB FCN block is time-synchronized audio chunk. Thus, the MATLAB FCN, which is run every time an entire audio frame is ready for processing, acts on real-time input audio. As long as the algorithm is fast and efficient, it should be able to play real-time audio out. This is very similar to an embedded program generating an **interrupt** every time an audio sample or frame is ready.

Important: for this lab, we are not using any input audio in the processing, we are merely using the Audio Device Reader to generate these **interrupts,** so that our algorithm runs in real time at 44100 Hz. As such, **the Device parameter in the block is irrelevant (use whatever is default), ensure the sampling rate is 44100 Hz, make the samples per frame 1024, and make the Number of Channels 1 (mono).**

47

Defining the Strings: In Part A, you will write an initialization function in MATLAB to control global variables that can be used in both Simulink and in MATLAB after they are defined.

Controlling Volume: You will control the volume of each string using a Knob block in Simulink. This knob gets connected to a constant block (see the provided Simulink file), which is then used as an input to a volume update MATLAB FCN. Once again, this executes every time a volume update is made using the knob (also similar to an **interrupt**), and this MATLAB FCN updates a global variable defining the volume of the string.

48

Part A) The Init Function

Fill in the blanks in *ThreeStrings_init.m* relating to the first string. You will uncomment the rest of the code later. This function serves to create global variables that you will **"pass by reference"** to your MATLAB FCN blocks, ie they will be linked to each function as a global variable. This syntax is provided, but make sure you look at it and understand how it works.

```
% Define all global variables:
global string1;
global string2;
global string3;

global volume1;
global volume2;
global volume3;

% Define standard variables:
fs = 44100;
dur = 2;

% Define paramaters for the first string:
a1 = % FILL IN            % (-1 0]
g1 = % FILL IN           % [ 0 1]
f1 = % FILL IN
pos1 = %FILL IN
string1 = PluckedString(f1,fs,dur,pos1,a1,g1);
volume1 = 1; % Default
```

49

Part B) The update_PluckedString1 Function

Fill in the blanks in *update_PluckedString1.m*. This function takes in an audio **frame** from the Audio Device Reader within Simulink, and then processes that audio frame sample by sample with the string1.process() function. Remember, we're not using the audio frame directly, we're using it merely to get synchronized output audio in the desired sampling rate.

```
function [out_frame] = update_PluckedString1(in_frame)
% This function takes in an audio frame, and processes each sample in the
% frame. It then outputs a frame to be heard by passing it into the Audio
% Device Writer

% Pass global variables "by reference"
global string1;
global volume1;

% Determine the length of the input frame:
num_samples = length(in_frame);

% initalize the output frame
out_frame = zeros(size(in_frame));

% Process each sample in the input frame:
for idx = 1:num_samples

    % FILL IN
    % hint: what global variables do we have? How do we process samples?
    % hint: out_frame(idx) = volume1 * ??

end
```

50

Part C) The update_volume1 Function

Fill in the blanks in *update_volume1.m*. This function takes in a new volume value to save to the global variable volume1.

```
function [] = update_volume1(new_volume)

% pass in volume 1 by reference
global volume1;

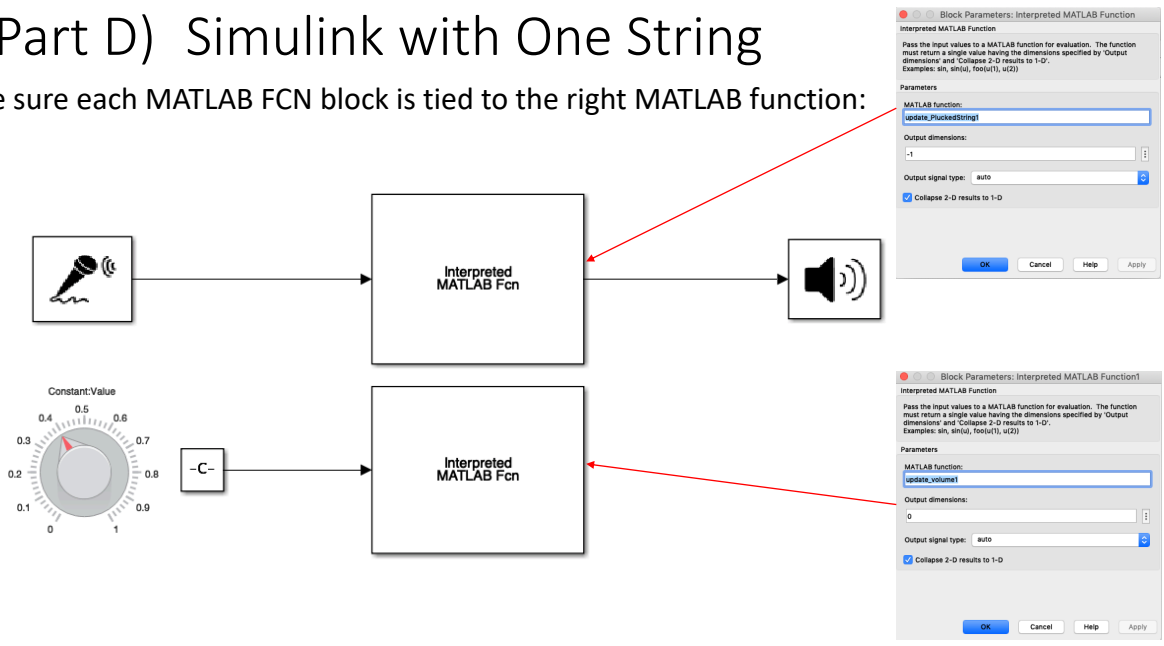
% update the global variable with the new value for volume1:
volume1 = % FILL IN


end
```

51

Part D) Simulink with One String

Make sure each MATLAB FCN block is tied to the right MATLAB function:

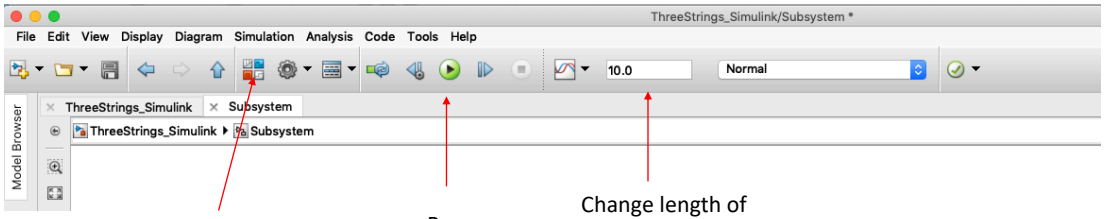


52

Part D) Simulink with One String (continued)

Start by running, **in the MATLAB Editor**, the ThreeStrings_init script that you've made

Then, click on the green Run button on the top of **Simulink's editor**. You should hear 10 seconds of a guitar playing, plucking every two seconds. Make sure your volume knob also works as expected. If this runs, you have completed the real-time implementation of a singular string.



Block Library
Used in Part E

Run

Change length of
Simulink run time
(default 10 seconds)

53

Part E) Simulink with Three Strings (4 points)

Go back and finish the *ThreeStrings_init* function to include strings 2 and 3. Ensure all three strings are at different frequencies. Fill in the blanks in *update_PluckedString2, update_PluckedString3, update_volume2,*and *update_volume3* to account for the three strings.

Expand the Simulink diagram in *ThreeStrings_Simulink.slx* to include a framework for the second and third strings. Figure out how to split the AudioDeviceReaders input into 3 arrows (don't make 3 Readers) to input into the corresponding MATLAB FCN blocks.

In the final step, instead of having all three strings output to individual Audio Device Writer, use an **Add block** from the Block Library to sum all three string outputs together to mono before playback into a singular Writer.

54

Problem 4 Deliverables

- Implemented real-time three string system (4 Points)
  - A video demonstration of your system is needed to grade this part, see the example in the "How to Submit" slide, reproduced here: https://youtu.be/Ji8ov3MhLQQ
  - Partial credit can be given for 1-2 strings
- **In your report:** make sure you answer all questions posed in this PowerPoint, and give an explanation of what you did and why you did it

55