DSP First
Lab 04a: Synthesis of Sinusoidal Signals - Speech Synthesis

Pre-Lab and Warm-Up: You should read at least the Pre-Lab and Warm-up sections of this lab assignment and go over all exercises in the Pre-Lab section before going to your assigned lab session.

Verification: The Warm-up section of each lab must be completed **during your assigned Lab time** and the steps marked *Instructor Verification* must also be signed off **during the lab time**. One of the laboratory instructors must verify the appropriate steps by signing on the **Instructor Verification** line. When you have completed a step that requires verification, simply demonstrate the step to the TA or instructor. Turn in the completed verification sheet to your TA when you leave the lab.

Lab Report: It is only necessary to turn in a report on Section 4 with graphs and explanations. You are asked to **label** the axes of your plots and include a title for every plot. In order to keep track of plots, include your plot *inlined*[1] within your report. If you are unsure about what is expected, ask the TA who will grade your report.

1 Introduction

This lab includes a project on speech synthesis with sinusoids. The speech synthesis will be done with sinusoidal waveforms of the form

$$x(t) = \sum_k A_k \cos(\omega_k t + \phi_k)$$

where each sinusoid will have short duration on the order of the pitch period of the speaker. One objective of this lab is to study how many sinusoids are needed to create a sentence that sounds good. A secondary objective of the lab is the challenge of putting together the short duration sinusoids without introducing artifacts at the transition times. Finally, much of the understanding needed for this lab involves the spectral representation of signals - a topic that underlies this entire course.

2 Pre-Lab

In this lab, the periodic waveforms and speech signals will be created with the intention of playing them out through a speaker. Therefore, it is necessary to take into account the fact that a conversion is needed from the digital samples, which are numbers stored in the computer memory to the actual voltage waveform that will be amplified for the speakers.

2.1 LabVIEW Review

You learned several LabVIEW techniques during the last lab. Here is a list of some of them:
How to use probes.

- How to convert a VI to a sub VI
    - Adding terminals to a sub VI

[1] Remember, if you want to include a waveform plot, right-click and select **Data Operations»Export Simplified Image…**.

    - Adding a sub VI to a block diagram
- How to use a for loop
    - Iterating over an array
    - Using N to specify the number of iterations
    - Using i
    - Using shift registers to carry values from one iteration to another.

2.2 Theory of Sampling

Chapter 4 treats sampling in detail, but this lab is usually done prior to lectures on sampling, so we provide a quick summary of essential facts here. The idealized process of sampling a signal and the subsequent reconstruction of the signal from its samples is depicted in Figure 1.

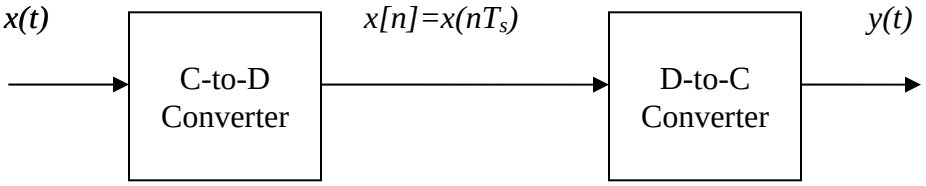


Figure 1: Sampling and reconstruction of a continuous-time Signal.

This figure shows a continuous-time input signal $x(t)$, which is sampled by the continuous-to-discrete (C-to-D) converter to produce a sequence of samples $x[n]=x(nT_s)$, where n is the integer sample index and T_s is the sampling period. The sampling rate is $f_s=1/T_s$ where the units are samples per second. As described in Chapter 4 of the text, the ideal discrete-to-continuous (D-to-C) converter takes the input samples and interpolates a smooth curve between them. The *Sampling Theorem* tells us that if the input signal $x(t)$ is a sum of sine waves, then the output $y(t)$ will be equal to the input $x(t)$ if the sampling rate is more than twice the highest frequency f_{max} in the input, i.e., $f_s > 2\, f_{max}$. In other words, if we *sample fast enough* then there will be no problems synthesizing the continuous audio signals from $x[n]$.

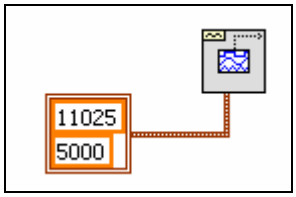
2.3 D-to-A Conversion

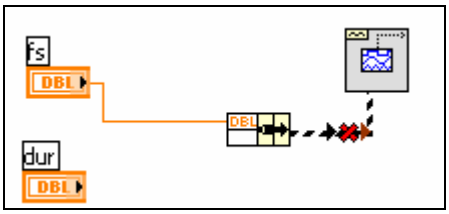
Most computers have a built-in analog-to-digital (A-to-D) converter and a digital-to-analog (D-to-A) converter (usually on the sound card). These hardware systems are physical realizations of the idealized concepts of C-to-D and D-to-C converters respectively, but for purposes of this lab we will assume that the hardware A/D and D/A are perfect realizations.

The digital-to-analog conversion process has a number of aspects, but in its simplest form the only thing we need to worry about at this point is that the time spacing (T_s) between the signal samples must correspond to the rate of the D-to-A hardware that is being used. From LabVIEW, the sound output is done by the **soundcs** VI or the **tripleDisplay** VI which both support a variable D-to-A sampling rate if the hardware on the machine has such capability. A convenient choice for the D-to-A conversion rate is 11025 samples per second[2], so $T_s = 1/11025$ seconds; another common choice is 8000 samples/sec. Both of these rates satisfy the requirement of *sampling fast enough* as explained in the next section. In fact, most piano notes have relatively low frequencies, so an even lower sampling rate could be used. If you are using **soundcs** or **tripleDisplay**, the input will be scaled automatically for the D-to-A converter.

[2] This sampling rate is one quarter of the rate (44,100 Hz) used in audio CD players.

a) The ideal C-to-D converter is, in effect, being implemented whenever we take samples of a continuous-time formula, e.g., $x(t)$ at $t = t_n$. This is done for us in LabVIEW by the **Function Generator VI**. It first makes a vector of times, and then evaluates the formula for the continuous-time signal at the sample times, i.e., $x[n] = x(nT_s)$ if $t_n = nT_s$. This assumes perfect knowledge of the input signal, but we have already been doing it this way in previous labs. To begin, create a vector `x1` of samples of a sinusoidal signal with $A_1=100$, $\omega_1=2\pi(800)$, and $\phi_1=-\pi/3$. Use a sampling rate of 11025 samples/second, and compute a total number of samples equivalent to a time duration of 0.5 seconds. You may find it helpful to recall that in LabVIEW the **sampling info** input lets you set the sampling rate and number of samples.

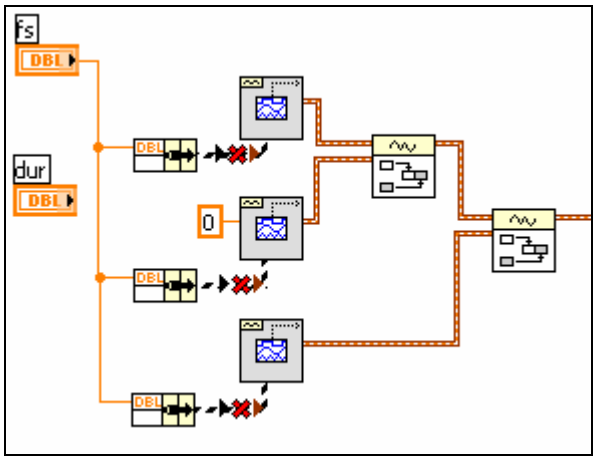




In the left diagram the sampling rate is set to 11025 and there are 5000 samples. The right diagram shows how to use a **Bundle**[3] to access the sampling rate and number of samples separately. You need to figure out how to calculate the number of samples given `fs` and `dur`. You should use the `sin_syn` function from a previous lab for this part. Use **tripleDisplay** to look at and play the resulting vector through the D-to-A converter of the your computer, assuming that the hardware can support the $fs = 11025$ Hz rate. Listen to the output.

b) Now create another vector `x2` of samples of a second sinusoidal signal (0.8 secs. in duration) for the case $A_2=80$, $\omega_2=2\pi(1200)$, and $\phi_2=+\pi/4$. Listen to the signal reconstructed from these samples. How does its sound compare to the signal in part (a)?

c) **Concatenate** the two signals `x1` and `x2` with a short duration of 0.1 seconds of silence in between. You should be able to use something like:



where the right-most two blocks are **Append Waveforms** VIs[4]. Note the middle

[3] Hint: right-click the background of the block diagram and then click **Search**. Type **Bundle** to find it.
[4] You can find this VI by right-clicking on the background of the block diagram view and selecting **Search**.

function generator has amplitude of **0**. Determine the correct value of **dur** to make 0.1 seconds of silence. Listen to this new signal to verify that it is correct.

d) To verify that the concatenation operation was done correctly in the previous part, attach a **tripleDisplay**. This will plot a huge number of points, but it will show the "envelope" of the signal and verify that the amplitude changes from 100 to zero and then to 80 at the correct times.

2.4 Debugging Skills

Testing and debugging code is a big part of any programming job, as you know if you have been staying up late on the first labs. Almost any modern programming environment provides a *debugger* so that break-points can be set and variables examined in the middle of program execution. Of course, many programmers insist on using the old-fashioned method of inserting print statements in the middle of their code (which is rather hard in a graphical language like LabVIEW). This is akin to riding a tricycle to commute around Atlanta.

Do the exercise in the **Debugging** folder to learn some simple, yet powerful techniques.

2.5 Synthesizing a Signal from Sections

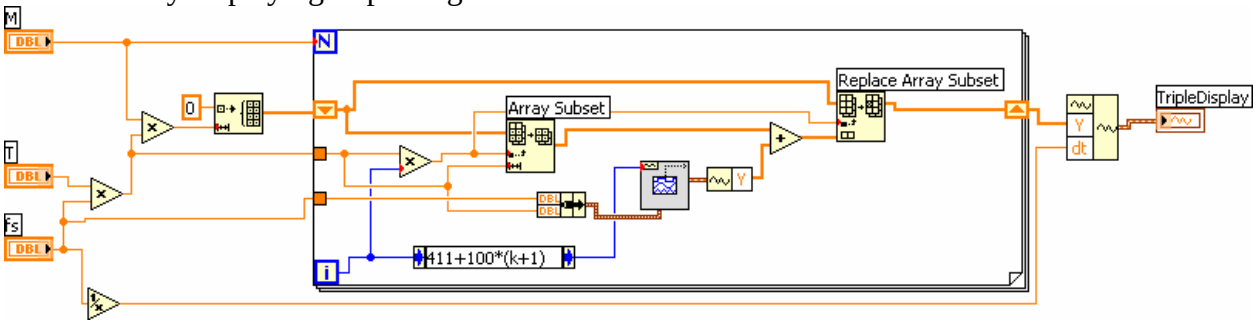
In speech synthesis, we will create the overall signal one section at a time. One way to do this is to add together a number of short signals; a mathematical notation for adding short signals that are time-shifted is given by

$$x(t) = \sum_{k=0}^{K-1} x_k(t - kT)$$

where each signal $x_k(t)$ is shifted by the amount T. If each signal has a duration of T, then the shifted signals $x_k(t-kT)$ do not overlap, and then we would actually be creating $x(t)$ by *concatenating* the sections $x_k(t)$ one after the other, so the total duration of $x(t)$ would be KT. Consider the case where

$$x_k(t) = \cos(2\pi(411+100k)t) \text{ for } 0 \le t \le T$$

In order to concatenate six sinusoids each with a duration of $T = 0.3$ secs, run the LabVIEW code below to make the signal which will have a duration of 1.8 s. The code can be found in `FirstExample.vi` Then the changing frequency content (vs. time) of the synthesized signal can be verified by displaying a spectrogram.



One problem with this synthesis by concatenation is that the transition from one section to the next might not be smooth. Examine a plot of $x(t)$ to see the jumps at multiples of T.

3 Warm-up

We can join signal segments together smoothly if we use "windowing."

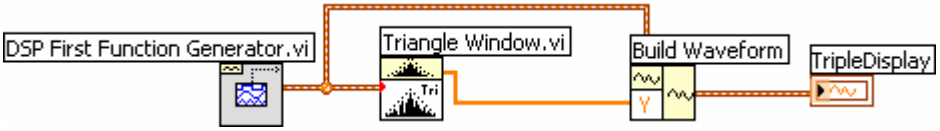
3.1 Triangular Window

Sometimes it is necessary to modify the values of a signal to taper the ends. This can be accomplished with what is called a *window function*. One of the simplest window functions is the *triangular window* defined[5] for duration T as

$$\omega_{\Delta}(t) = \begin{cases} t/T & 0 \le t < T \\ 2 - t/T & T \le t \le 2T \\ 0 & elsewhere \end{cases}$$

(a) Draw a sketch (by hand) of $\omega_{\Delta}(t)$ for the case $T=50$ millisec.

(b) LabVIEW has a vi that applies a triangle window to an array. Build the following to test it. Save it as `triwin.vi`. Hint: Use *Search* to find `Triangle Window.vi`.



Notice that `Triangle Window` takes a Waveform and an input and outputs and array. The `Build Waveform` vi is used to convert the array back to a waveform.

(c) Test your `triwin` vi by making a LabVIEW plot of a triangular window for the case $T=50$ millisec, using a sampling rate of 8000 samples/sec. How long is the window in number of samples?

Instructor Verification (separate page)

3.2 Overlapped Signal Segments

In Section 2.3, you created a long signal by concatenating short segments. A second method of forming the long signal is to use overlapped short segments. Here we will study how to extract such overlapped segments from a long signal. Suppose that we start with a long signal $y(t)$ and we extract short segments from $y(t)$ in the following manner:

$$y_k(t) = \begin{cases} y(t + kT) & 0 \le t < 2T \\ 0 & elsewhere \end{cases} \qquad (3)$$

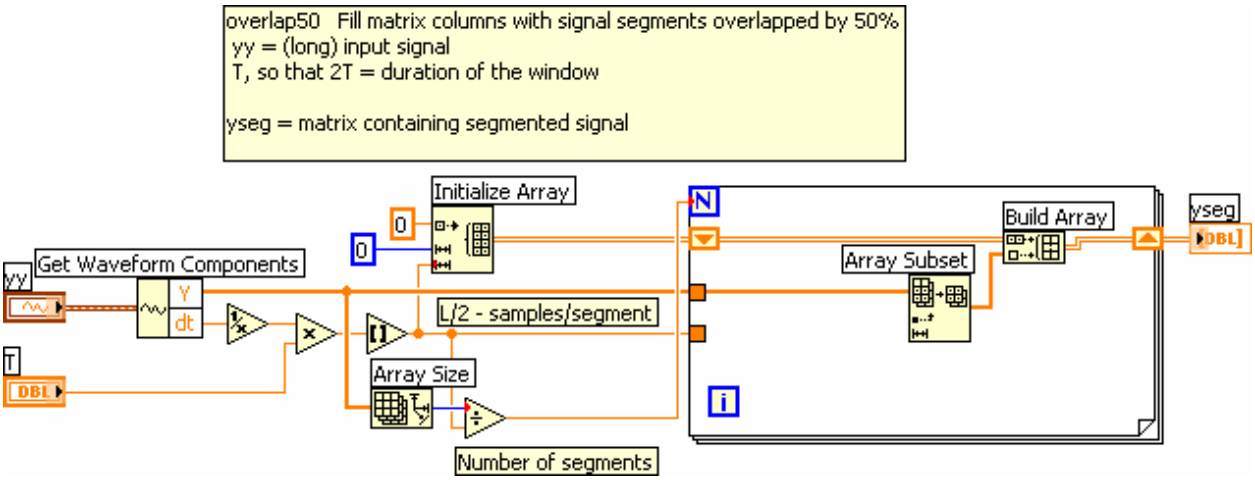
In other words, the k^{th} segment, $y_k(t)$, starts at $t=kT$ and ends at $t=(k+2)T$. Furthermore, successive signal segments, such as $y_k(t)$and $y_{k+1}(t)$, have 50% overlap.

(a) ***Number of Segments:*** If the duration of $y(t)$ is 1.5 sec. and $T = 50$ msec., how many segments would be produced by overlap method in (3)?

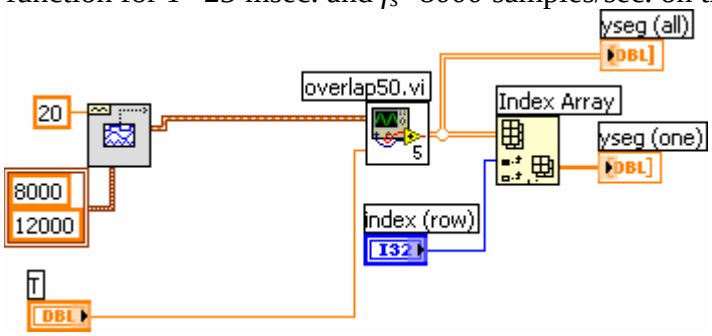
(b) ***Segment Length:*** If we are using LabVIEW to represent the signal $y(t)$, then we would sample $y(t)$ at a rate f_s to produce a vector containing the samples, i.e., $y[n] = y(n/f_s)$. For example, if $f_s = 8000$ samples/sec and the duration of $y(t)$ is 1.5 sec., then the entire "y" vector would contain 12000 samples. How many samples are contained in each segment created by the overlap method in (3) if $T=50$ msecs. and $f_s = 8000$ Hz?

(c) Write a short LabVIEW function that will perform the segmentation according to (3). The function's output should be a matrix whose column length is the number of samples in one segment, and whose row length is the number of segments. Here is a template:

[5] The subscript Δ in $\omega_{\Delta}(t)$ is meant to convey the shape of the triangular window.



(d) Test your function for $T=25$ msec. and $f_s=8000$ samples/sec. on the following signal:



Explain why every other column of the *yseg* matrix is identical for this test case. Use the LabVIEW vi `Index Array` to plot one row at a time.

Instructor Verification (separate page)

3.3 Overlapped Windows

The triangular window has the following interesting property: when you add shifted triangular windows, the result is one, except for the ends. This property can be stated mathematically as

$$\sum_{k=0}^{K-1} \omega_{\Delta}(t-kT) = \begin{cases} t/T & 0 \le t < T \\ 1 & T \le t \le KT \\ K+1-t/T & KT < t < (K+1)T \\ 0 & (K+1)T \le t \end{cases} \qquad (4)$$

The important line in this equation is the second one which says that the sum equals one in the intervals where the triangles overlap, see Fig. 1.

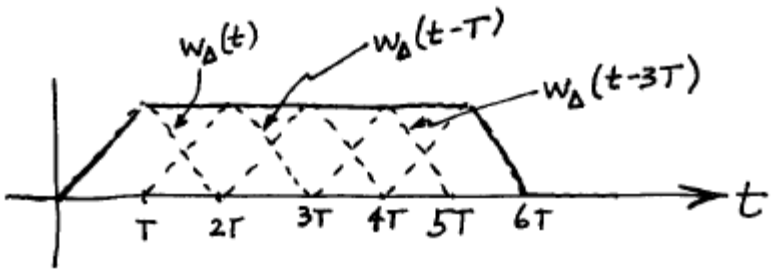


Figure 1: Sum of shifted and overlapped triangular windows. The window length must be even for this property to hold on the time-sampled window.

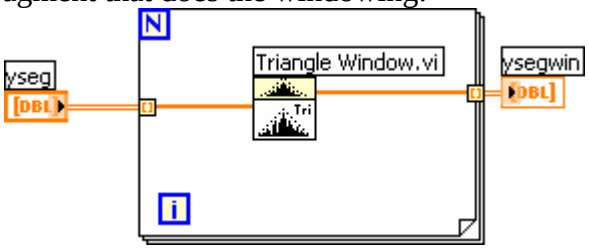
3.4 Add Overlapped and Windowed Segments

We can use the overlap property of the triangular window (eq. (4) in Section 3.3) to add back together the segments from the 50% overlap method of eq. (3). First of all, we would apply the triangular window to each segment, i.e., $\omega_\Delta(t)y_k(t)$, and then we would add all of the segments together with the correct shift.

$$\sum_{k=0}^{K} \omega_\Delta(t-kT)\,y_k(t-kT)$$

The result will be equal to *y(t)* except for the regions at the ends.

(a) Here is a code fragment that does the windowing:



(b) Write a `for` loop that will add the windowed segments back together to form `yy` over most of the time interval, except for the first and last *T*secs.

(c) Now test the entire process with the signal 440 Hz cosine with a segment duration of 10 millisec., 50% overlap, and f_s = 8000 Hz. Perform the three processing steps as:
(1) break it into segments (refer to Section 3.2(c)),
(2) window each segment with a triangular window (part (a) above), and
(3) add the segments back together (part (b) above).

Show that the result is a 440-Hz cosine, except for the first 5 ms and the last 5 ms.

4 Lab: Synthesis of Speech with Sinusoids

Speech signals are often “quasi-periodic” especially in vowel regions. Thus it is reasonable to expect that speech utterances might be synthesized from a few sinusoids. On the other hand, fricatives such as /s/ and /sh/ sounds are not sinusoidal, so there are probably regions where the sinusoidal synthesis would do a poor job. Fortunately, the *intelligibility* of an utterance depends mostly on the vowels and less on the fricatives.

Speech can be synthesized by adding together a bunch of short-duration sinusoids. Thus, an important factor in the sinusoidal synthesis of speech is the *frame length*, which is the time interval during which one set of sinusoids is used. From frame to frame the sinusoids can change. In speech, there are two time durations that would be relevant to picking the frame length: the

speaker's pitch period and the articulation rate of humans in general. The *pitch period* varies with individuals and with sex—adult males generally have a lower pitch than adult females and hence the pitch period is longer for an adult male speaker. The *articulation rate* is a measure of how fast a speaker can form different sounds and is dictated by how fast the muscles in the vocal tract can move to form different sounds. For example, try saying the alphabet (A-B-C-D-E-F) as fast as you can. It is generally accepted that the individual sounds can change no faster than every 40–50 millisec. Taken together the pitch frequency and articulation rate determine how often we should try changing the sinusoids for the speech synthesis. We will use a frame length around 10 ms. which is close to the pitch period of most speakers.
In the process of actually synthesizing the speech, keep in mind the following general ideas:

(a) Determine a sampling frequency that will be used to play out the sound through the D-to-A system of the computer. This will dictate the time $T_s = 1/f_s$ between samples of the sinusoids.
(b) The total time duration needed for each sinusoid is fixed by the frame length.
(c) An analysis function *sigAnalyze.vi* is provided to extract sinusoidal components from a signal. It's block diagram is locked so you can run it, but not look inside it.
(d) Synthesize the speech waveform as a combination of overlapped (or concatenated) sinusoids, and play it out through the computer's built-in speaker or headphones.

Include a spectrogram image of a portion of each synthesized utterance—probably about 1 or 2 seconds — so that you can illustrate the fact that you have used the correct number of sinusoids. In effect, you can use the spectrogram to confirm the correctness of your synthesis. The window length in the spectrogram might have to be adjusted, but start with an initial value of 256 for the window length in the tripleDisplay, and then increase it. *Note:* by default, the spectrogram will scale the frequency axis to run from zero to half the sampling frequency, so it might be useful to "zoom in" on the region where the frequencies are.

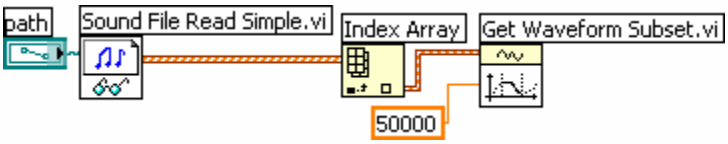
Data Format for Speech Signals
Any speech signal can be encoded with the LABVIEW `sigAnalyze VI` which has the following calling format:

```
Input:
  signal in = input signal waveform (not an array)
  numSines = # of sinusoids to find (only the positive freqs are counted)
  T = duration of overlap between frame in secs.  Total frame size is 2T.
Output:
  Camps = array of complex amps (number of frames by numSines)
  Freqs = array of freqs, one for each complex amp
```

The output of `sigAnalyze` gives the frequencies and complex amplitudes needed in each frame; the inputs are the analysis parameters such as frame duration, and number of sinusoids. In the complex amplitude and frequency arrays, the value of `Freqs(j,n)` is the n^{th} frequency (in Hz) in the j^{th} frame of the signal; the corresponding complex amplitude is `Camps(j,n)`.

There are at least three ways to get a speech signal into LABVIEW, depending on the format:

1. Use the `Sound File Read Simple VI` to load the speech data in from a `WAV` file. This VI produces an array of waves. If you aren't doing stereo, use the following to get the first waveform:



2. The Get Waveform Subset allows you to process just part of the file.
3. Use a VI with data already in it. For example, place the `sigMystery` VI on your diagram.
4. Use the `Acquire Sound` VI to record directly from a microphone into LabVIEW.

In addition, you can write a `WAV` file from LabVIEW with the `Sound File Write Simnple` VI. Be careful that you scale the signal's amplitude to be between 0 and 1.

4.1 Synthesis

In this lab, you will use the given VI `sigAnalyze` to create a set of complex amplitudes and frequencies of sinusoids for each small frame of various signals, and then write a function to re-synthesize the speech signal from a limited number of sinusoidal components.

(a) Write a `sigSynth` VI that will take the outputs from `sigAnalyze` and produce an output signal in which the frames can be overlapped. *Note:* If the signal was analyzed with an overlap in `sigAnalyze`, then you must use a segmenting strategy like `overlap50` inside of `sigSynth`.

(b) Apply `sigAnalyze` to a recording[6] of your own voice which is sampled at $f_s = 8000$ Hz. Use a frame duration of 10 ms, a 5 ms overlap, and extract 8 sinusoidal frequency components (per frame).[7] One possible utterance is the vowels, i.e., "A-E-I-O-U".

(c) Synthesize the speech using all 8 frequencies components obtained in the analysis of part (b). Compare spectrograms of the original and the resynthesized signal, and comment on the differences that you hear in the sounds.

(d) Now, extract 4 frequency components and synthesize the speech (these will be the four largest ones). Compare spectrograms of the original and the resynthesized signal, and comment on the differences that you hear in the sounds.

4.2 Synthesize Music

The per-frame sinusoidal synthesis should work on signals other than speech, e.g., music, singing, etc. However, since the frame duration parameter depends on the type of signal, it is unlikely that the value picked for speech signals will be the best choice for a signal like music.

(a) Apply the `sigAnalysis` and `sigSynth` functions to the recording of the piano piece Fúr Elise which can be found in the file `FurElise.wav`. Use the same

---

[6] LABVIEW (version 7) has a function called `audiorecorder` that can acquire sound from a microphone via a sound card. Consult the `help` on `audiorecorder` and read the examples. Make sure to save the data as "double" instead of "int16."

[7] 4The file `catdogs.wav` is also available, but you are not required to process it for your lab report.

parameters as before: a frame duration of 10 ms, and overlap of 5 ms, and keep 8 sinusoidal components from the analysis. Compare spectrograms of the original and the resynthesized signal, and comment on the differences that you hear in the sounds.

(b) Now, try to get a better result by changing the number of sinusoids. Determine how many are needed to get a result that sounds very close to the original. There is probably not one answer for this number, because it depends on your individual perception of the distortions in the resynthesized signal.

(c) Now, try to improve the result by changing the frame duration while also reducing the number of sinusoids from the amount found in the previous part. For example, consider the possibility that if you make the frame duration shorter, then you might be able to reduce the number of sinusoids.

4.3 Mystery Signal

One more data set, `sigMystery.mat`, is included as a challenge for synthesis. This is a `.MAT` file containing the arrays `Camps` and `Freqs` extracted from a speech signal. There are connectors for T and dt, and a 50% overlap was used. You should run your synthesis algorithm with the objective of trying to understand what is being uttered.

4.4 Testing

Bring your working synthesis program to the lab when your report is due. A couple of test signals will be run to verify that you have a successful synthesis program. The format of the test signals will the same as `sigMystery.vi`, so make sure that your program can handle such inputs quickly and easily. In addition, you will be asked some questions about the inner workings of your LABVIEW synthesis function(s).

4.5 Lab Report Suggestions

Your lab report should include spectrograms of the original and the synthesized signals. The write-up should point out features in the spectrograms that indicate the differences between the eight-sinusoid and four-sinusoid cases for your voice. In addition, you should make subplots of the original signal and the resynthesized signal versus time, and identify where they are different by marking such features on the plots. The comparison will be easier if you scale both the original and the resynthesized signal to have the a maximum value of one.

Lab 04a
Instructor Verification Sheet

For each verification, be prepared to explain your answer and respond to other related questions that the lab TA's or professors might ask. Turn this page in at the end of your lab period.

Name: _____ Date of Lab: _____

Part 3.1 Make a triangular window M-file `triwin.m`:

Verified: _____ Date/Time: _____

Part 3.2 Test overlapped signal segments:

Verified: _____ Date/Time: _____

Part 3.3 Overlapped triangular windows:

Verified: _____ Date/Time: _____

Speech Evaluation Criteria

Are the sentences intelligible All _____ Most _____ Only one_____

Does the music (Fúr Elise) sound good?

Mystery sentence correct?

Test sentence correct?

Overall Impression:_____

Good: Good sound quality. Works well for both 4 and 8 sinusoids.

OK: Basic sinusoidal synthesis, but not smooth at the boundaries. Possible problem with windowing.

Poor: Synthesis does not work properly. Poor sound quality.