Problem 1b: Practice on coding + analyzing spectrograms

Pertinent readings for Problem #1b:

The Pokemon theme song:

My recording was taken from an actual recording session by the original guy who sang the title song ! Go to:

https://www.youtube.com/watch?v=fCkeLBGSINs

A brief introduction to vocals and spectrograms:

This is a great webpage !! I would definitely take a look at this first before you tackle the readings in Cohen & Rangayyan below:

http://vocped.ianhowell.net/a-spectrogram-primer-for-singers/

Cohen – Analyzing neural time series data

(Download from Blackboard, under /Resources / Signals & systems texts)

Ch. 11: pages 111 – 127 (A reminder on DFTs)
Ch. 15: pages 195 - 201 (A gentle introduction to spectrograms
Fancy-pants name = short-time Fourier transforms)

Rangayyan – Biomedial signal analysis (2nd 3d) :

(Download from Blackboard, under /Resources / Signals & systems texts)

Ch. 3: pages 113 – 137 (A reminder on DFTs)
Ch. 8: pages 478 - 486 (short-time Fourier transforms

Our main goals: Create spectrograms and do some easy filtering opertaions !

The first of our 2 main objectives for this problem is to generate the "non-overlapping" spectrogram that you see in Figure 1 (and you should check out the fully-annotated, zoomed-in version of the same spectrogram on Blackboard before you start).

$Sound\ file$: $Pokeman.wav$

$Total\ duration\ of\ sound$: $132\ seconds$

$Data\ point\ density$: $f_{sample} = 44{,}100\ Hz$ $\left(\begin{matrix}This\ means\ 1\ second\ worth\ of\ sound \\ is\ equal\ to\ 44{,}100\ data\ points\ !!\end{matrix}\right)$

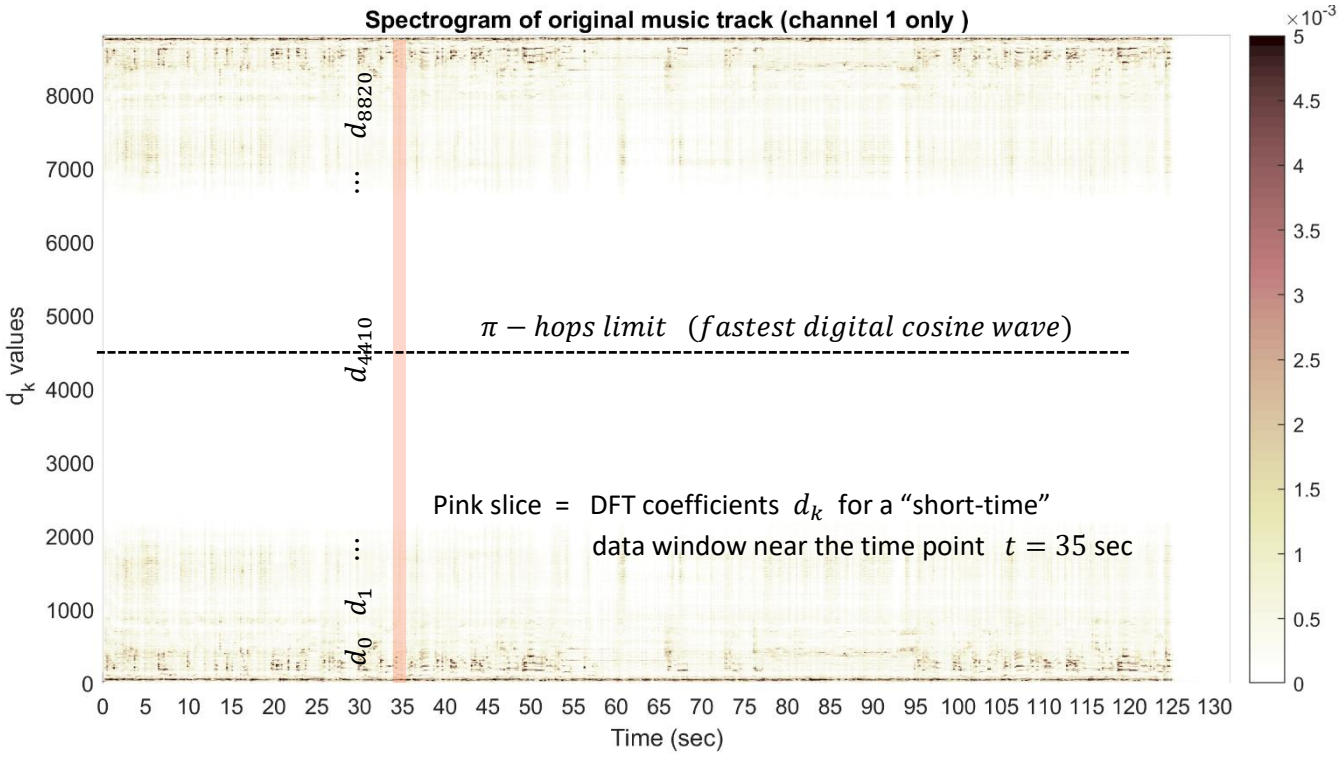

Figure 1: A spectrogram of *pokemon.wav*, where the frequency content (vertical axis) within lots of "short-time" window slices our data are plotted agains the time point of interest.

Part A: Loading your audio file + separating out the left / right channels

1. Using the *audioread* function, load *Pokemon.wav* into matlab. Make sure your sampling frequency is 44.1 kHz before you start anything else !

$$f_{sample} = 44.1\ kHz$$

2. Separate the left vs. right channels into 2 separate column vectors. Let's suppose we call them:

ch1_raw = one channel
ch2_raw = the other channel

Part B: Code your spectrogram using a window length of L

For this part, just focus on your channel 1 data first !! We only need the spectrogram for one of the 2 available channels.

1. Using a window length of:

$$L = \frac{1}{5} f_{sample} = 8820\ points = 0.2\ second\ worth\ of\ data\ per\ window$$

a) First, add the appropriate # of *ghost-zero nodes* at the end of the audio file for our choice of L

b) Create a *non-overlapping* spectrogram using our window length L. Store your spectrogram data in a giant matrix (you don't have to echo it), and it should be ready to-be plotted with the *pcolor* command !

** Caution: When you're taking windowed slices of data and calculating their corresponding discrete Fourier coefficients d_k , remember the syntax for matlab's *fft* command (and note that the total number of data points in your slice is now equal to L):

$$d = \frac{1}{L} \left(\overline{F}\right)^T x_{slice\ data} \xrightarrow{matlab} d = (1/L) .* fft(x)$$

Hint: As a reminder, see Figure 2 for a synopsis of what your tasks are ! If you're not sure, then go back and read:

a) Cohen – Ch. 15 (the indicated pages on the 1st page of this PDF), and
b) Re-read the overview file for Problem 1 again =)

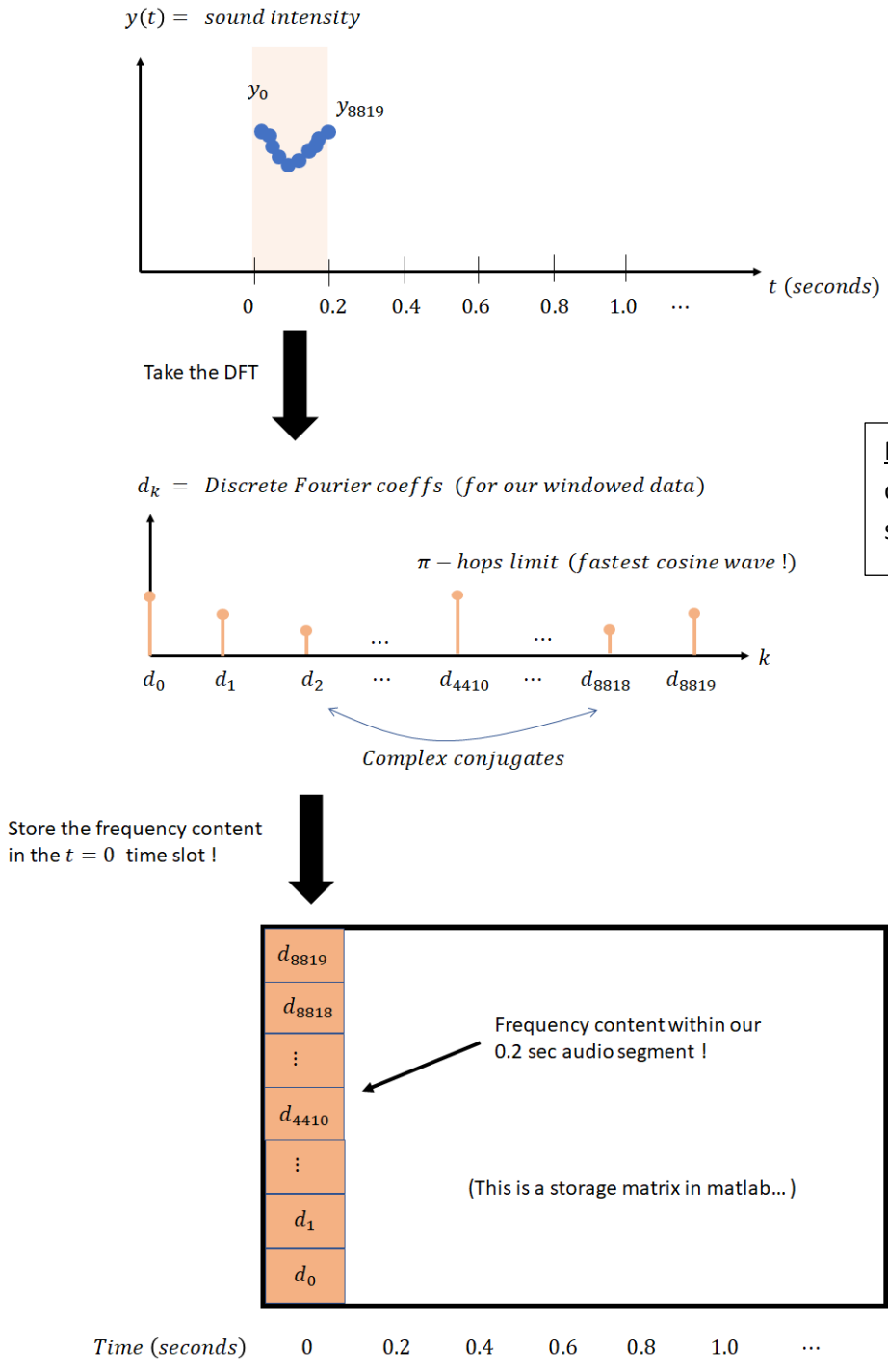

Figure2: A quick reminder of how to construct a non-overlapping spectrogram.

2. Using matlab's *pcolor* command, plot your spectrogram.

Suppose you named your giant spectrogram matrix called "my_spectro_matrix." You would plot your spectrogram by invoking something like:

```
original_spectrogram_plothandle = pcolor( T,  K,   abs( my_spectro_matrix)  )
set(original_spectrogram_plothandle, 'EdgeColor', 'none');    % -- Disable the black edges !
coloarbar
 caxis( [ 0  0.005 ]) ;        %  Set your colorbar limits like this (because there is 1 dominant
                               %  Fourier coefficient that "outshines" all the others ones, and if
                               %  you don't  apply these limits, you won't see 99% of the dk's…
```

such that:

a) You are plotting the magnitude $|d_k|$ for all of your complex Fourier coefficients
b) Your horizontal axis labels are in time (seconds)
c) Your vertical axis labels are the k-index values for the discrete Fourier coefficients d_k
d) Title your plot so that I know it's the spectrogram of the original channel 1's data
e) Add a colobar that will indicate the values of the d_k

Hint: To control your horizontal and vertical-axis labels, you have to create 2 meshgrid variables T (time) and K (the # of k's you need for all of your d_k coefficients). You did a very similar thing in your giant G-matrix homework, where I asked you to plot both a pcolor and a contour / quiver plot with custom-made axes !! =)

Part C: Low-pass filtering on both your data

Now, you can either study your own spectrogram (or taking a peek at my annotated spectrogram of *Pokemon.wav* on Blackboard). You notice that at around 90 - 95 seconds into the song, there is a series of high-frequency guitar riffs (especially near the 95 second mark).

Perform a low-pass filtering operation on both channels 1 and 2 such that the guitar riff @ around 95 sec is no longer audible.. To actually make sure you don't hear the guitar when you play back your filtered data, you should do the following:

1) For your spectrogram matrix, attenuate all necessary Fourier coefficients d_k down to zero such that you can no longer hear the guitar riff. Then, make a *pcolor* plot of your post-filtered spectrogram (this plot will also serve as a self-check debugging step !!). Make sure you caxis limits are the same as before: caxis([0 0.005])

Hint: You want to identify where the π-hops d_k value is, and then, you should perform symmetric attenuations of the Fourier coefficients across both sides of that π-hops d_k value.

2) Now, you have to reverse-engineer your post-filtered sound file by using your filtered d_k coefficients at each time slice. To do this, you need to:

a) Take the d_k coefficients for each time slice (ie. Take the appropriate column vector in your spectrogram matrix)

b) Then, back-build the audio (per slice) by using the *inverse fast Fourier transform* command:

$$y_{slice\ data} = F\ d \xrightarrow{matlab\ command} y = L .* ifft(d)$$

c) Once you have your "newly-filtered" window slice, you can recursively "stitch" successive window slices together by horizontal concatenation of the slices ! The whole process is essentially performing everything in Figure 2 in reverse, and Figure 3 is a synopsis of what should happen:

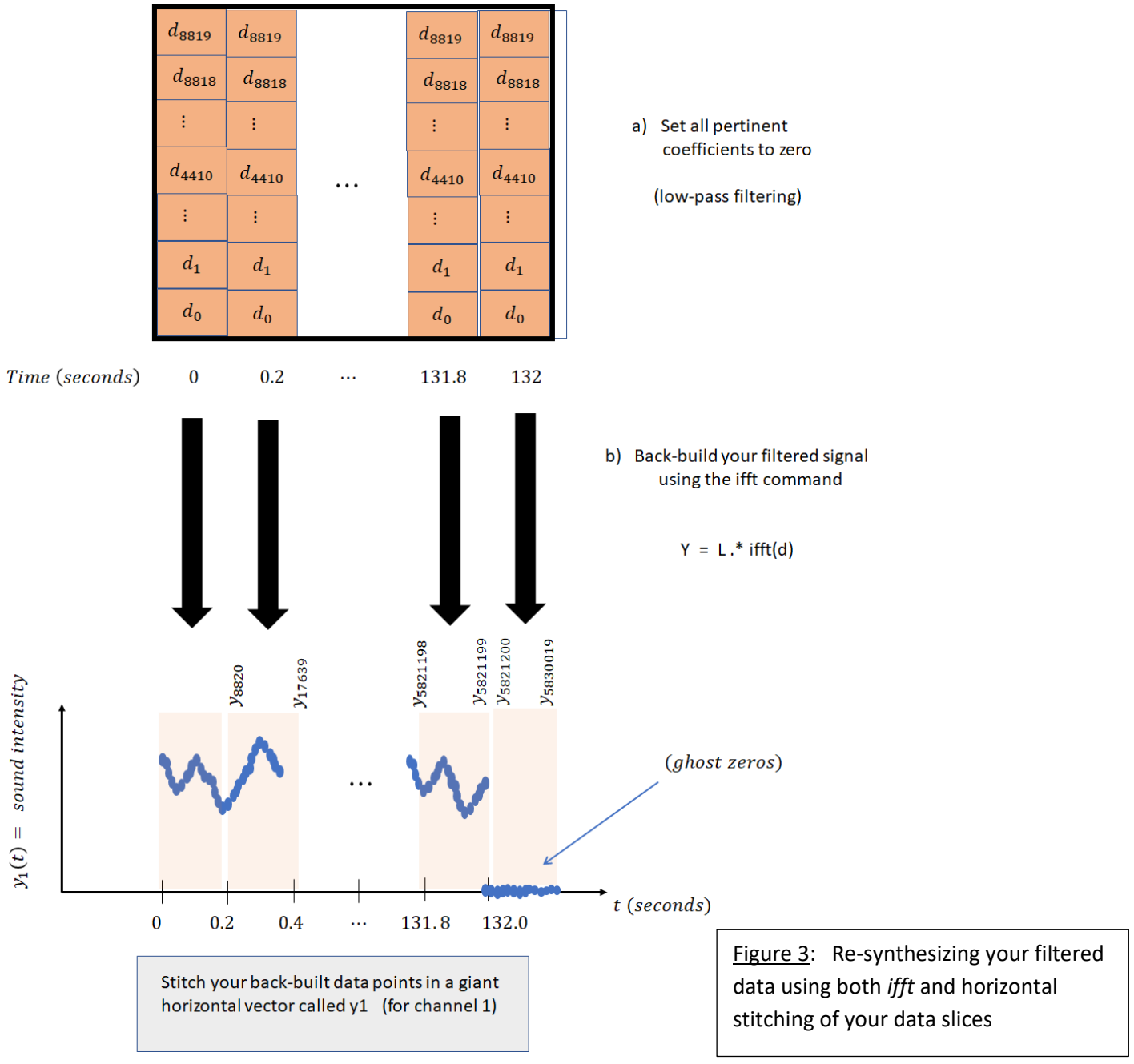

Figure 3: Re-synthesizing your filtered data using both *ifft* and horizontal stitching of your data slices

d) In the end, you should have 1 giant horizontal vector containing our low-pass filtered data for Channel 1.

$$y1 \quad = \quad \begin{bmatrix} LPF \\ channel\ 1 \\ data \\ \\ (left\ speaker) \end{bmatrix}$$

Part D: Write your output audio file…. And play your low-pass filtered song !

Since we're lazy and we don't have a $10k multi-channel surround-sound system, we can just copy the filtered channel 1 data into an empty matrix y….. and make an identical copy of it as the 2nd column vector of y.

$$y = \left[\begin{array}{c} LPF \\ channel\ 1 \\ data \\ \\ (left\ speaker) \end{array}\middle|\begin{array}{c} LPF \\ channel\ 1 \\ data \\ \\ (right\ speaker) \end{array}\right]$$

Note: You have to convert your "y1" into a column vector before inserting it into "y" !!

1. Using the *audiowrite* command, write your low-pass filtered song "y" as:

Pokemon_LPF_noGuitar.wav

2. Use your computer and play your wave file !! Theoretically, if you low-pass-filtered enough, you should barely be able to hear the high guitar notes….. and at the same time, the voice components should still sound decent. You may have to play around with your Fourier coefficient attenuations to get good results !

★★★★★

** *HINT*: When you are invoking the *audiowrite* function, if you get an error that says something like:

"Cannot perform function - values needs to be real, floating, int8, etc…."

99% of the time, this means there was an error your low-pass filter step (ie. You've inadvertently deleted some $d_k's$ in a *non-symmetric fashion with respect to the π-hops d_k* , and as a result, when you back-build your audio data using the *ifft* command, the resulting audio data are complex numbers instead of real numbers !!!)