Cochlea.jl:
A Julia-based real-time cochleogram visualizer for the Jupyter notebook

Alex Kell
15F: 18.337/6.338 Final Project

Motivation

If I ask you to describe an image, you would be able to give me a rich description. You could give semantic information (e.g., What objects are in it? Where is this picture taken? What do you think the scene will look like in three seconds? Etc.), low-level information (e.g., Where are the lines and edges? What are the colors? Etc.), and putatively intermediate-level information (Where are the surfaces? What are the textures on the surfaces? What are the parts of the objects and what are the relative positions of those parts? Etc.) People are quite good at describing an image at various levels of description, from relatively high-level semantics to lower-level, nearly pixel-based properties.

I would argue that people are much worse at describing a sound at multiple levels of abstraction. People can often label the *source* of the sound (a person talk or walking, a dog panting, bacon sizzling). But people seem to have a relatively limited vocabulary for describing sound at lower or intermediate levels. People can certainly talk about pitch and loudness, but little beyond that. (It's arguable that trained musicians do have a richer vocabulary, but crucially this expanded vocabulary requires training.) At the very least, *I* don't feel like I have nearly as naturally rich vocabulary for describing sounds as I do for describing images, and as an auditory neuroscientist I want to better understand what *stuff* makes up sounds. So I did this project. I used Julia to create a coarse model of the cochlea and then measured and visualized that model's response to sounds in real-time recorded from a computer's microphone.

The Cochlea & The Cochleogram

The cochlea transduces the mechanical energy of sound vibrations to the electrical energy used in the nervous system. A sound pressure waveform travels down the ear canal, vibrates the tympanic membrane (i.e., the "ear drum"), which in turns vibrates a handful of bones in the middle ear, which in turn vibrates the fluid inside the cochlea, which in turn vibrates the basilar membrane (see Fig 1., left). The physical properties of the basilar membrane vary along its length. Near the base of the cochlea, the basilar membrane is narrow and stiff, while farther towards its apical end it is wider and floppier (see Fig. 1, right). Due to the material properties of the basilar membrane, sounds of different frequencies cause different portions of it to vibrate. Low-frequency sounds vibrate more apical parts of the basilar membrane, while high frequency sounds vibrate more basal parts. A given cochlea neuron responses to vibrations in a small local portion of the basilar membrane and thus cochlear neurons selectively respond to certain frequencies. In essence, the cochlea performs a time-frequency decomposition of a given sound.

The cochleogram is a coarse model of cochlear neurons (Patterson et al., 1987; Slaney, 1995). It is a time-frequency decomposition that is similar to a spectrogram or a short-

time Fourier transform, but is different in a handful of key ways which were inspired by observations about the cochlea. First, the frequency-axis is roughly log-spaced, inspired by the frequency tuning pattern along the basilar membrane. Second, different time-frequency bins can overlap, because the size of the filter bandwidth is determined by inferences about the bandwidth of human cochlear filters (Moore and Glasberg, 1983). Third, the amplitudes of each spectrotemporal bin go through a compressive nonlinearity, because basilar membrane motion appears to have a similar relationship to sound level (Ruggero et al., 1997).

Computationally, the cochleogram consists of three steps:

1. First, the incoming sound is passed through a bank of linear filters. These filters are localized in frequency and their bandwidths are determined by observations of how bandwidth and the center-frequency are related in the cochlea (Moore and Glasberg, 1983). Each filter's shape in the frequency domain is one half cycle of a cosine.

2. Second, the envelope of the time course of these filters is extracted. Specifically, we use the Hilbert transform to compute the analytic representation of each filter's time course. The Hilbert transform generates a complex-valued signal where the real part is the original time course and the imaginary part is that same time course, but shifted in phase by a quarter cycle. For instance, the Hilbert transform of the sine function is the cosine. Once we have the analytic signal, we compute the magnitude to generate the envelope.

3. Lastly, we downsample the envelopes of the time courses of each filter, simply for ease of plotting.

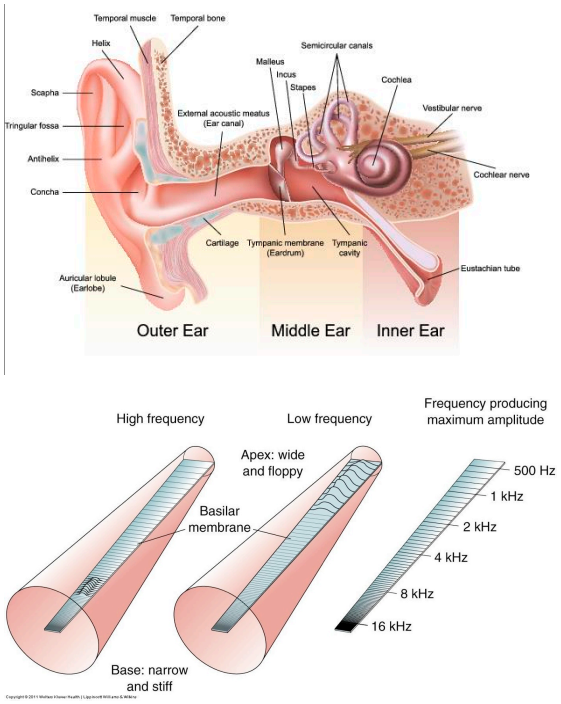

Figure 1. Cochlea schematic. Left: Outer, middle, and inner ear. Note that the sound pressure waveform propagates down the ear canal, vibrates the tympanic membrane (ear drum), then vibrates the various bones of the middle ear, which in turn vibrates the oval window on the cochlea, which in turn vibrates fluid within the cochlea, which in turn vibrates the basilar membrane. **Right: Basilar membrane.** The basilar membrane is curled in a snail-like pattern in the cochlea, and here it is schematically shown flattened. Note that different portions of the membrane have different material properties which in turn affects how the membrane vibrates with respect to different incoming frequencies.

The cochleogram certainly does not recapitulate all we know about cochlear processing (e.g., Dallas, 1992; Heinz et al., 2001; Ruggero et al., 1992, 1997; Russell and Nilsen, 1997; Schwartz and Simoncelli, 2001; Sellick and Russell, 1979). Nonetheless it is a reasonable coarse model of the first stage of auditory processing since it captures a few

key properties of cochlea processing, such as frequency selectivity and loudness compression.

Cochlea.jl overview

Cochlea.jl has three key parts. It interfaces with the microphone, then computes a cochleogram on the input waveform, and finally plots that waveform in a Jupyter notebook. It iterates through this procedure for as long as the user specifies.

Given that this is a real-time visualizer, how high-fidelity of cochleogram you can display is constrained by the speed of the code. Given the importance of efficiency and speed, I followed some good best practices to help the Julia compiler optimize efficiently, including having no globals, making variables constants when I can, and explicitly labeling the inputs to functions.

I also sought to speed up the performance in other ways, as well, and below I below I discuss how I sped up interfacing with the microphone and computing the cochleogram. Plotting was relatively straightforward and there was not much sophisticated there.

Interfacing with the microphone
The code must interface with the microphone. My first-pass implementation of this interface was simply to call a function in python that used PyAudio to fetch data from the microphone and write wav files with the following call:

```
@async run(`python recordWriteWav.py
                   $sd_name
                   $n_seconds_to_record
                   $record_secs_len
                   $sampling_rate`)
```

The script **recordWriteWav.py** fetched the data from the microphone and then wrote it to a wav file. Then I simply had a big while loop that checked whether a new file had been written and read it in if it had, plotted it, and so on.

I figured I could speed up this process by using PyCall directly in Julia, which I ultimately implemented instead. It consisted of a handful of relatively simple calls:

```
p = pyaudio.PyAudio()

stream = p[:open](format=pa_format,
                  channels=pa_channels,
                  rate=pa_sampling_rate,
                  input=true,
                  frames_per_buffer=pa_samples_per_chunk)

wav_raw = pyeval(
         "converter(stream.read(chunk_length),dtype=thisDtype)",
         Array{Int16,1}, # return type
         stream=stream,
         chunk_length=pa_samples_per_chunk,
         converter=np.fromstring,
         thisDtype=np.int16)
```

However, Julia calling Python is not the most elegant (or likely quickest) solution. PortAudio is an excellent lower-level API that's written in C and C++. It offers substantial control over auditory input and output via a simple callback function or a blocking read/write interface. Matlab and Python both have bindings to PortAudio, such as the PyAudio module that I used here. Julia also has a module that was built for this purpose: AudioIO.jl. Unfortunately, this module appears to have been broken by v0.4 and it hasn't been maintained. If this project were longer term, I would want to create Julia bindings for PortAudio.

Generating the cochleogram

I outlined the steps of cochleogram generation above, and I originally implemented those steps in a relatively straightforward way:

```
# perform convolution of time domain by performing
# point-wise multiplication in the frequency domain
F = fft(wav)
subbands_fourier = broadcast(.*, F, fft_filts);
subbands = real(ifft(subbands_fourier,1)) # back in time domain

# now get the envelopes of  the signal by computing the magnitude
# of the analytic signal
analytic_subbands = hilbert(subbands)
subbands_envs = abs(analytic_subbands)
```

When I profiled the code, I saw that, perhaps unsurprisingly, these steps were the key limiting factor in the speed at which I could iterate through and generate cochleograms. I made a few key changes. First, my waveform is a real-valued array so I ought to use rfft instead of fft, which roughly halved the amount of compute time. Second, the standard algorithm for the Hilbert transform (and the one Julia uses) transforms the signal to the frequency domain and brings it back with an ifft. Therefore, I have two unnecessary matrix multiplies here, which I was able to get rid of. The more optimized code below:

```
# get subbands in the fourier domain
F_r = rfft(wav)
subbands_fourier_r = broadcast(.*,
                               F_r,
                               fft_filts[1:Int(n/2)+1,:])

subbands_fourier_r_full = zeros(Complex128,
                                n,
                                size(subbands_fourier_r,2))
subbands_fourier_r_full[1:n2+1,:] = subbands_fourier_r

# manually compute the Hilbert transform in the frequency domain
# w/o going back to the time domain
h = zeros(n)
h[ [1,n2+1] ] = 1
h[2:n2] = 2 # implicitly have the zeros for the rest
analytic_subbands_F = broadcast(.*, subbands_fourier_r_full, h)

# bring this back to the time domain and get the envelopes
analytic_subbands_r = ifft(analytic_subbands_F,1)
```

```
subbands_envs = abs(analytic_subbands_r)
```

Miscellaneous tricks to speed things up

I also played with a variety of tricks outside of the scope of Julia to speed things up. Given that I was downsampling the envelopes and only measuring the frequency responses up to 7 kHz, I altered my computer's microphone settings to reduce the fidelity of the input. Given that these options were relatively limited (lowest possibility was 32 kHz), I also immediately decimated the signal when I read it from the microphone buffer.

I also played with the "nice" value for the process that was running the notebook and the Julia kernel, to make buffer overflows less likely. E.g., from a shell call like: **`sudo renice -n 10 -p $pid`**. Where **`$pid`** was the process ID. It's unclear to what extent playing with the nice actually affected performance.

Example call

In an Jupyter/IJulia notebook you can simply import this script and generate a real time cochleogram. See example below:

```
include("Cochlea.jl")
plotRealTimeCgram( 100, # n_filters
                   400, # subbands_ds_factor
                   1.0,  # record_secs_len
                   16, 8) # fig_x, fig_y
```

Figure 2 shows a plot generated from this code.

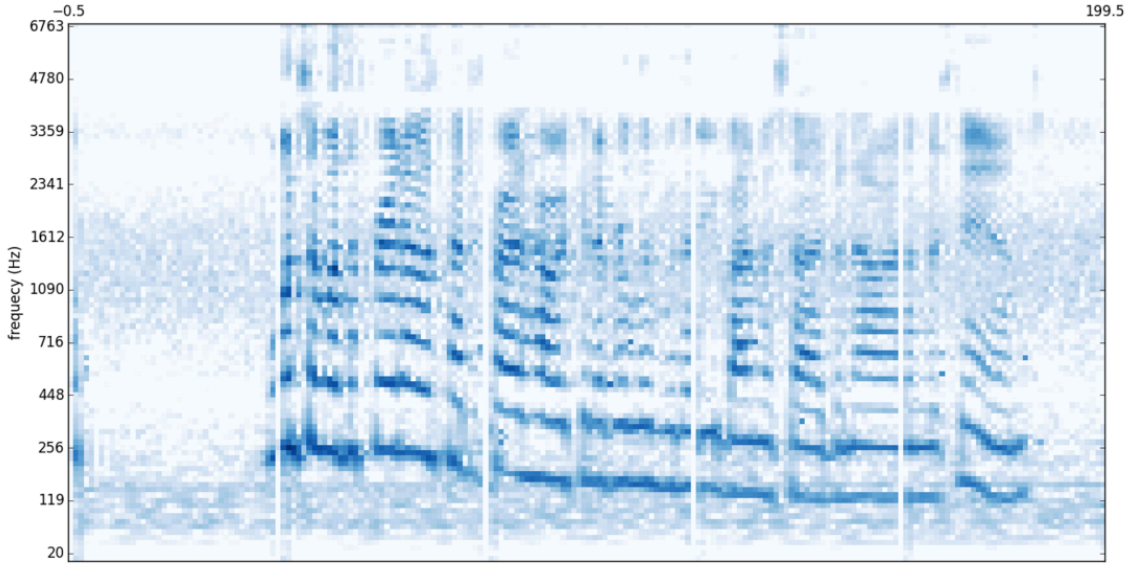

Figure 2. Example cochleogram generated from the code in real time. This is me talking to in a relatively dry background. Y axis indicates frequency; x axis indicates time; the color of shading is amplitude of response.

Conclusions

Here I implemented a real-time cochleogram visualizer in Julia that interfaces with the IJulia/Jupyter notebook. In my efforts to optimize the speed of my code, I think the biggest boon was from an algorithmic side – i.e., realizing that I can use rfft and implement the Hilbert transform myself and skip a couple large matrix multiplies. In any case, this was a fun and useful project to get more familiar with Julia. Thanks for a good class.

References

Dallas, P. (1992). The active cochlea. J. Neurosci. *2*, 4575–4585.

Heinz, M.G., Zhang, X., Bruce, I.C., and Carney, L.H. (2001). Auditory nerve model for predicting performance limits of normal and impaired listeners. Acoust. Res. Lett. Online *2*, 91–96.

Moore, B.C.J., and Glasberg, B.R. (1983). Suggested formulae for calculating auditory-filter bandwidths and excitation patterns. J. Acoust. Soc. Am. *74*, 750–753.

Patterson, R.D., Nimmo-Smith, I., Holdsworth, J., and Rice, P. (1987). An efficient auditory filterbank based on the gammatone function. In A Meeting of the IOC Speech Group on Auditory Modelling at RSRE,.

Ruggero, M.A., Robles, L., and Rich, N.C. (1992). Two-tone suppression in the basilar membrane of the cochlea: Mechanical basis of auditory-nerve rate suppression. J. Neurophysiol. *68*, 1087–1099.

Ruggero, M.A., Rich, N.C., Recio, A., Narayan, S.S., and Robles, L. (1997). Basilar-membrane responses to tones at the base of the chinchilla cochlea. J. Acoust. Soc. Am. *101*, 2151–2163.

Russell, I.J., and Nilsen, K.E. (1997). The location of the cochlear amplifier: spatial representation of a single tone on the guinea pig basilar membrane. Proc. Natl. Acad. Sci. *94*, 2660–2664.

Schwartz, O., and Simoncelli, E.P. (2001). Natural signal statistics and sensory gain control. Nat. Neurosci. *4*, 819–825.

Sellick, P.M., and Russell, I.J. (1979). Two-tone suppression in cochlear hair cells. Hear. Res. *1*, 227–236.

Slaney, M. (1995). Pattern playback from 1950 to 1995. In Systems, Man and Cybernetics, 1995. Intelligent Systems for the 21st Century., IEEE International Conference on, (IEEE), pp. 3519–3524.