Application of a Machine Learning Model toward a Better Hearing Aid Signal Processing Scheme

Aaron Hoffman

1 Introduction

Hearing aids have functioned on one simple principle since their inception: amplify an incoming sound signal so that is perceived as audible for the patient. More advanced hearing aids implement some signal processing in an attempt to make the user perceive speech as "standing out" against background noise[1], but these methods ignore a more fundamental issue. When the body of an outer hair cell (OHC) rooted in the Basilar membrane (BM) experiences a vibration, the stereo cilia extend and conduct vibration to the Tectorial membrane (TM) which then transmits the vibrational signal to stereocilia extending from the inner hair cells (IHC). Conceptually, the OHCs function as a nonlinear gain medium which can selectively amplify sound signals. A schematic diagram of these features is shown in Figure 1.

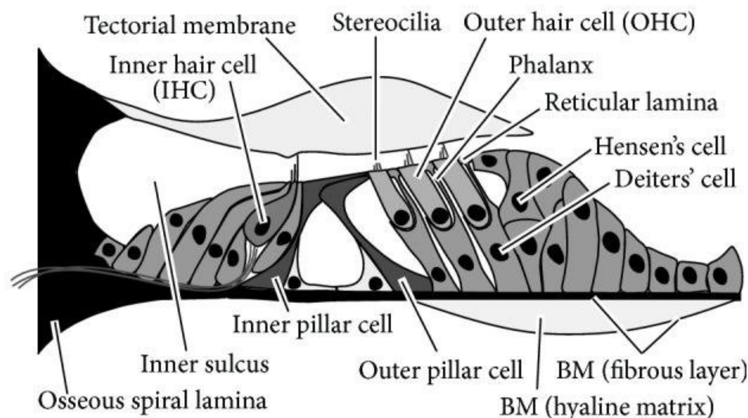

Figure 1: A schematic diagram of the organ of Corti.

The fundamental problem is that when the OHCs are damaged, the loss of functional "gain medium" causes the Basilar membrane and, in turn, the Tectorial membrane to mechanically respond differently to incoming sound waves. An example of this response difference is displayed in Figure 2. A 250 Hz pure sine wave with duration 50 ms was played for both the active and passive (damaged) cochlea models. Notice that the active response is a concentrated, horizontal line in time in a region centered about ≈ 25 mm. In comparison, the passive Basilar membrane response is dramatically broadened in space and centered about a region closer to ≈ 22 mm. The location of response in the cochlea is frequency dependent. Higher frequencies will show a response closer to 0 mm on a plot like those in Figure 2, and lower frequencies will show a response closer to 30 mm. This shows that without a functional OHC "gain medium", an incoming sound signal will produce a response that is broadened in space along the cochlea and shifted to a higher central frequency.

The ultimate goal of this project is to design a sound filtering scheme which will alter the response of a passive cochlea in order to cause it to respond as it would if it were active. In other words: how must the pure 250 Hz tone be filtered such that the right half of Figure 2 matches the left half?

[1] Robyn M. Cox, Jani A. Johnson, and Jingjing Xu (2016). *Impact of Hearing Aid Technology on Outcomes in Daily Life I: The Patients' Perspective.* Vol. 37. 4. NIH Public Access, e224–e237. ISBN: 0000000000000. DOI: 10.1097/AUD.0000000000000277.

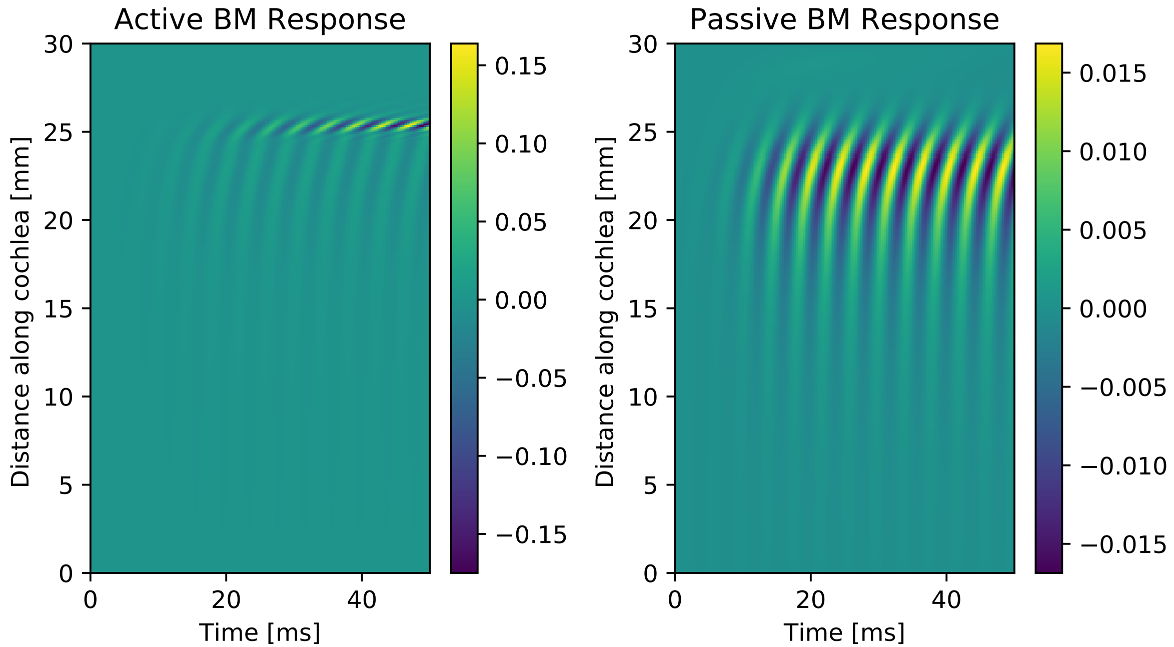

Figure 2: The response of the Basilar membrane in an active cochlea compared to the Basilar membrane response in a passive cochlea. Both the active and passive cochlea models were stimulated with a 250 Hz sine wave with a duration of 50 ms.

2 Methods

2.1 The Cochlear Model

In order to carry out the analysis that will be described in the following sections, a comprehensive data set consisting of active and passive cochlear responses to certain audio signals would be required. Even if such a data set existed, it would be impossible to test the efficacy of a given filtering method as the time and money required to do the requisite number of experiments would be substantial. Instead, I have chosen to use a computational model of Basilar and Tectorial membrane response in the time domain. The model used for this project is based on a Matlab code written by Nobili and Mammano. This can be found at http://147.162.36.50/cochlea/cochleapages/download/cochmodels.htm.

The model first considers the Basilar membrane as a discrete collection of hydrodynamically-coupled oscillators driven by a fluid-pressure input from the Stapes which is proportional to the time-varying amplitude of the incoming sound wave. The equations for the Tectorial and Basilar membranes are coupled in that a displacement of the Basilar membrane will case some displacement in the Tectorial membrane which can again influence the motion of the Basilar membrane. The outer hair cells are modeled as a term which provides a localized oscillation undamping. This undamping term is localized in that each point along the basilar and Tectorial membrane has an undamping coefficient. Because this undamping coefficient is localized in space and the spatial response of the cochlea is dependent on frequency, there is a frequency-dependent relation between the undamping term and the frequency of the incoming sound wave. The only difference between the active cochlea model and the passive cochlea model is that this undamping term is 0 for all space. A plot of the undamping coefficient for the active and passive cochlea is displayed in Figure 3. Remembering that higher frequencies produce a response closer to 0 mm and lower frequencies closer to 30 mm, this figure shows that the undamping coefficient is large for high frequencies and much smaller for low frequencies. It is also possible to look at a cochlea that is ”partially“ passive in that the undamping coefficient is somewhere between that of the active cochlea and passive cochlea.

2.2 Modifications to the cochlear model

I have had to make some significant changes to the original Matlab code to make it suitable for this project. The original Matlab code allowed the user to load in a sound file and then interactively displayed the cochlear response. However, it did not support saving the response data and was very inflexible with the way simulations were set up

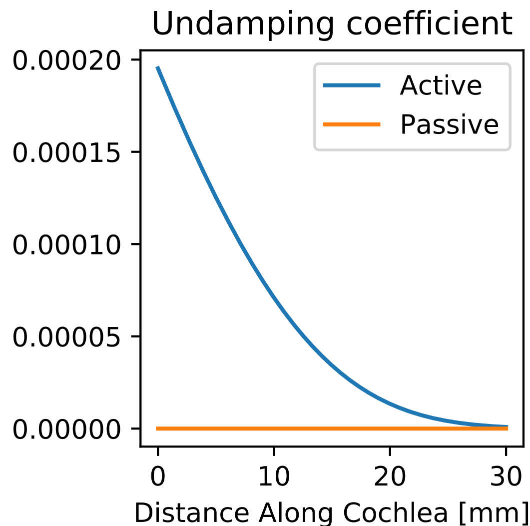

Figure 3: A comparison of the undamping coefficient between the active cochlea and the passive cochlea.

and called. Due to Matlab licensing issues and personal preference, I chose to port the simulation to python. Many of the parameters for the underlying model were calculated once and than saved to MAT files for access later. Basically my python version of the code reads in these pre-generated parameters and then reimplements the motion equations these parameters are used in. I have restructured the code into a more user-friendly, object-oriented format which was essential for my optimization techniques. An example for basic usage of the simulation code is as follows:

```
import pyCochlea
#set up the simulation object
simulationObject = pyCochlea.sim()

#create a 250Hz sine wave with a duration of 50 milliseconds
signal = pyCochlea.sinFunc(250,50e-3)

#initialize the simulation with the signal
simulationObject.initializeSignalFromArray(signal)

#run the active cochlea model
activeResponse = simulationObject.runActive()
activeResponse = simulationObject.runPassive()

#separate the BM and TM responses
BMActive = activeResponse["BMActive"]
TMActive = activeResponse["TMActive"]

BMPassive = activeResponse["BMPassive"]
TMPassive = activeResponse["TMPassive"]
```

2.3 Basis for the filtering mechanism

As seen in Figure 2, the passive response seems to "bleed" into higher frequency regions of the cochlea. This indicates that instead of hearing a 250 Hz sine wave as a pure tone, the passive cochlea would hear this wave as a band of sound between 250 Hz and $\approx$ 900 Hz. This diminished frequency acuity prevents a person from distinguishing different tones and as frequencies blur together sound will become muffled and "muddy". Therefore, the primary goal is to cancel regions where this frequency bleed occurs in order to restore frequency acuity. Conceptually, it seems plausible that if a higher frequency pitch is played simultaneously with the central frequency at some phase difference, the second

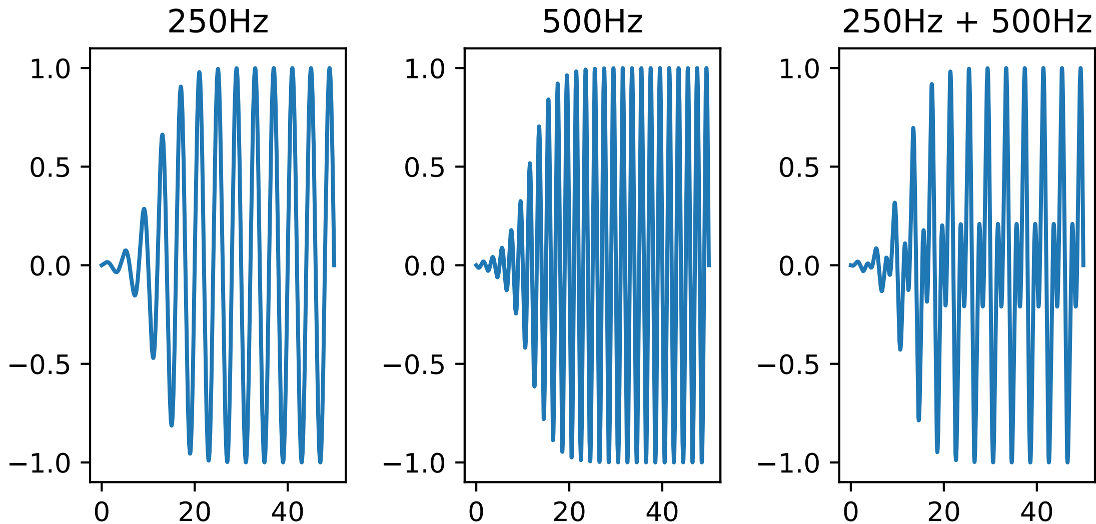

Figure 4: This is a very simple example of one case of Equation 1. In this case, f_c is 250 Hz, and a single extra sine wave at 500 Hz is added with a 90° phase shift. The plot on the far right shows the combination of the two waves.

signal may cancel some of the bleed from the central frequency. In practice, it may take multiple different additional frequencies to optimally cancel the frequency bleeding effect. To this end, a filter which is a linear combination of pure sine waves will be defined as follows:

$$S(t) = A\left(\sin(2\pi f_c t) + \sum_{i}^{N} b_i sin(2\pi f_i t + \phi_i)\right) \tag{1}$$

In Equation 1, f_c is the central frequency. In the example shown in Figure 2, f_c would be 250 Hz. The summation in the above equation is a linear combination of N sine waves with amplitude b_i, frequency f_i, and phase shift ϕ_i. A is a final amplification factor that is designed to make the new signal exhibit the same response magnitude present when the active cochlea is stimulated by an unmodified signal. A basic example of this filter is shown in Figure 4. For a given central frequency f_c, the goal is to find an optimal set of parameters (f_i, ϕ_i, b_i) which makes the passive cochlea response as similar as possible to the active cochlear response to the unfiltered signal. Once the process for finding these filter parameters is reliable, filter parameters can be found for a large number of f_c frequencies and the filter applied to a more complex sound signal.

2.4 Selecting a loss function

Perhaps the most difficult portion of this exercise is to formulate a loss function which sufficiently compares two of the spatio-temporal matrices. To make the comparison easier, some preprocessing can be done on the response generated by the computational model. Note the phase lead in the active response shown in Figure 2 compared to the passive response. The fringed lines lean more to the right than in the passive response. This effect is not considered to be important for the perception of sound. To remove any artifacts this phenomenon may cause while optimizing filter parameters, optimization will be done over the response's envelope. The envelope of the response is calculated by taking the absolute value of the Hilbert transform in time of the response at each point along the membrane. The Hilbert transform is done numerically using the implementation in `scipy.signal`. This preprocessing step preserves important aspect of the matrices while discarding unimportant features.

In order to find the best filter parameters, a metric is needed to define the similarity. I decided to begin with a simple loss function and add complexity as needed. The first attempt was a normal mean square error loss function:

$$\text{MSE} = \frac{1}{NM}\sum_{i,j}^{N,M}(A_{ij} - P_{ij})^2 \tag{2}$$

This function turned out to be an unsatisfactory measure of similarity between my two matrices for a few reasons. Crucially, it gives all regions of the response equal importance. This is undesirable because of the large number of

elements which are either zero or near-zero in both matrices. From empirical observation, using a normal MSE method tends to produce metric values which are quite small due to the small magnitude of the signal. When attempting optimization over this metric with several different methods, the results from the optimizer don't tend to change much if at all from the given initial conditions. Normalizing the data before calculating MSE by dividing both matrices by the maximum value present in the active response did not provide an appreciable difference in optimization performance.

Conveniently, areas with larger response magnitude in the active simulation (left half of Figure 2) are more important. Based on this fact, the second attempt at defining a loss function was to use a weighted version of the MSE calculation. The weighted MSE was calculated using the active response envelope A and passive response envelope P as:

$$\text{WMSE} = \frac{1}{\sum\limits_{i,j}^{N,M} w_{ij}} \sum_{i,j}^{N,M} \left(w_{ij}\left(A_{ij} - P_{ij}\right)\right)^2 \tag{3}$$

This loss function has the advantage that it allows more important regions to be considered more heavily in the comparison metric. The weight matrix for this method was generated based on thresholded values of the active response. An example of this in python is:

```
weights = np.zeros_like(activeEnvelope)
#Thresholds will be relative to the maximum response value.
maximumResponseValue = np.max(np.abs(activeEnvelope))
#Give pixels with zero magnitude a small amount of weight.
weights[np.where(active) >= 0.]                          = 0.05
#Weight based on different thresholds. Stronger responses get stronger weights.
weights[np.where(active) > 0.02*maximumResponseValue] = 0.3
weights[np.where(active) > 0.25*maximumResponseValue] = 1
weights[np.where(active) > 0.4*maximumResponseValue]  = 10
```

An example of the weights generated from the code above is shown in Figure 5. This loss function works reasonably well in some situations, but it incurs a potentially large number of hyper-parameters. To generate the weights, the user needs to decide several factors: how many thresholds to use, what the thresholds should be, and how much weight to assign to each threshold. This method may eventually provide good results but it requires too much heuristic analysis to get the weight generation parameters right.

Next, I have implemented a Dice coefficient loss function. The dice coefficient loss seems to be used often in image segmentation and comparison tasks as a measure of the similarity between two images. I have seen this metric used as the amount of overlap between two images. The formulation of dice coefficient loss is:

$$L_D = 1 - \frac{2|A \cdot P|}{|A|^2 + |P|^2} \tag{4}$$

This metric shows quite a bit of promise as it heuristically maintains some of the benefit of the weighted mean square error method without requiring nearly as much user input. I have only recently begun examining optimization performance of this loss function with my data, but preliminary results are promising.

Another loss function possibility I've explored is the log-cosh loss function:

$$L_{LC} = \sum \log\left(\cosh\left(A - P\right)\right) \tag{5}$$

This loss function seems to behave similarly to MSE while being less sensitive to outliers.

My next steps in this area are to continue evaluation of the dice coefficient loss function and log-cosh loss function, and to perhaps try another overlap-based loss function like Jaccard loss.

2.5 Optimization

In my project proposal and midterm report, I discussed using Dakota for optimization. Dakota is a freely-available optimization package from Sandia which has a very wide optimization, parameter study, and experimental design capability. After using this package for some time, it became apparent that similar functionality is available in python. I have shifted to using `scipy.optimize` for loss function minimization. Scipy's optimization package includes several optimization methods. I have tried using several of these optimization methods with some success. My primary

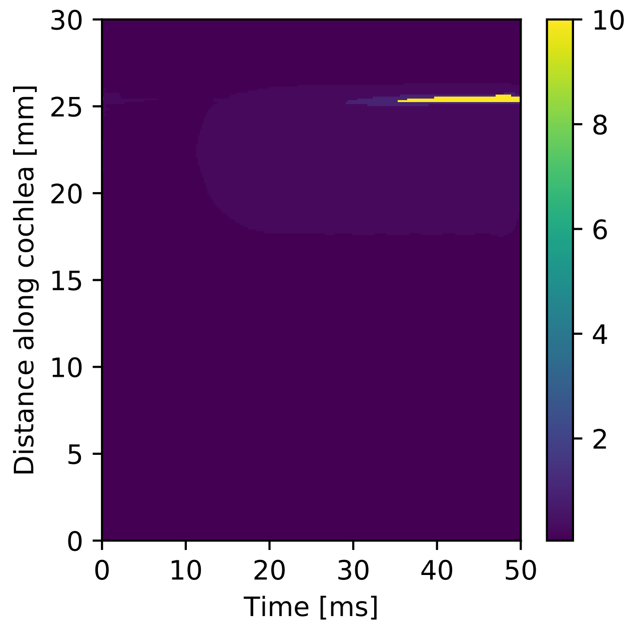

Figure 5: An example of weights generated from the active Basilar membrane response

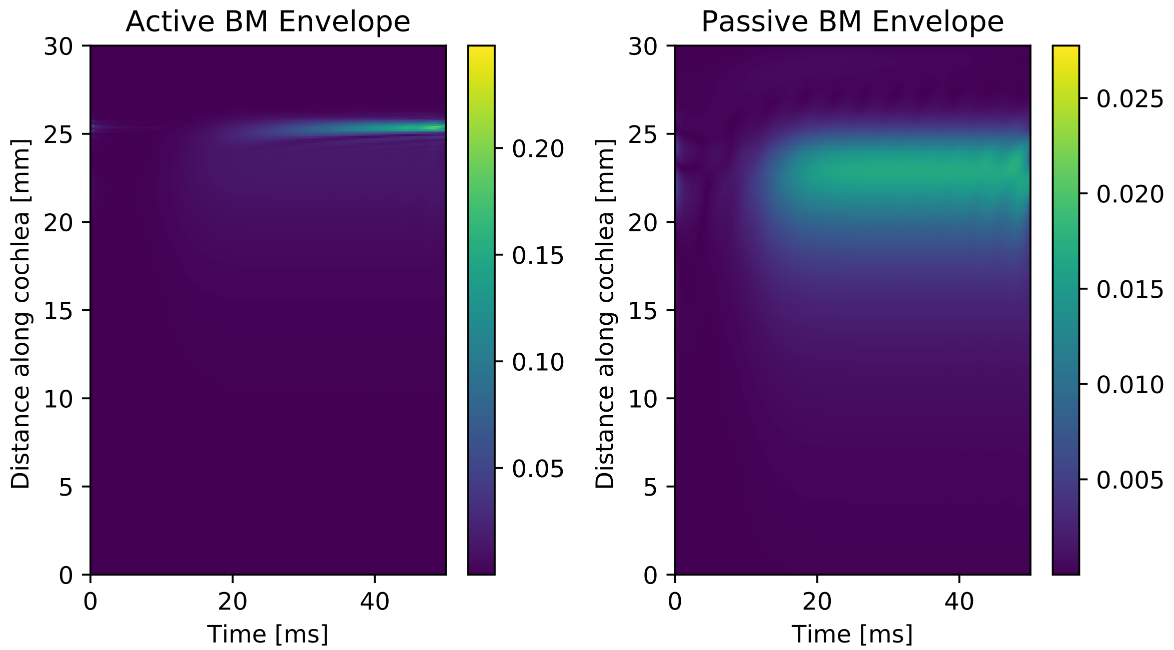

Figure 6: These plots show the envelope of the Basilar membrane response from Figure 2.

choice for optimization has been the conjugate gradient method which is a well-known method for optimization and is very applicable for this situation. I have been using numerical gradients rather than analytical gradients for all of my possible loss functions. I have had to spend a significant amount of time this semester formulating the optimization problem and have not yet been able to give the optimization step the amount of attention that it deserves. As my work on this project will be ongoing beyond the end of this semester, my next primary goal is to formulate analytical gradients for each of my loss functions to avoid any artifacts that may be coming from the numerical gradient calculation in Scipy.

One important note is that I am not scaling the parameters that I'm optimizing over. I expect that doing so would give some benefit in convergence speed and accuracy. However, because these parameters are being directly used in a simulation for each optimization step, there is some difficulty in making sure that parameters are scaled for optimization, un-scaled for model evaluation, and then re-scaled again. This is one direction that I will take in the coming weeks.

Recall the filter expression:

$$S(t) = A\left(sin(2\pi f_c t) + \sum_i^N b_i \sin(2\pi f_i t + \phi_i))\right)$$

One modification needs to be made before optimizing over this equation. As discussed earlier, the passive cochlea shows some shift toward higher-frequency regions of the cochlea. I will add a parameter to compensate for this:

$$S(t) = A\left(sin(2\pi(f_c + \Delta f)) + \sum_i^N b_i \sin(2\pi f_i t + \phi_i)\right)$$

Because N may become large, it's advantageous to describe a surrogate function to describe the amplitudes of the higher-frequency additions b_i. The goal of these additions is to cancel the effects of the passive cochlear response broadening. As seen on the right half of Figure 2, the region of high frequency response "bleed" decreases in intensity as it gets further away from the central frequency f_c. Because of this, I have added a functional form for the amplitudes b_i:

$$b_i(f_i; \tau) = \exp\left(-\frac{(f_i - f_c)}{\tau}\right) \tag{6}$$

A similar problem exists for the phase shift of each frequency ϕ_i. In the coming weeks I will either formulate a similar surrogate function for these values or allow my optimizer to optimize a ϕ_i value for each frequency f_i. For now, the same value for ϕ_i is used for all frequencies.

I have implemented a python code which accepts parameters for the smallest and largest values that f_i should take, the number of frequencies to generate N, the phase shift to apply to these frequencies phi_i, as well as the central frequency f_c and frequency shift Δf. I am optimizing the loss function according to the following model parameters: f_{i-max}, f_{i-min}, A, Δf, and τ.

Ideally, the result of this optimization would be a signal carrying the central frequency which has been shifted such that the response is at the proper place on the cochlea spatially along with a complex of higher-frequency signals which will cancel out the frequency "bleed" effect seen in Figures 2 and 6.

3 Results

In practice, it is useful to consider a cochlea that is only partially passive rather than an entirely passive cochlea. For this case, the undamping parameter displayed in Figure 3 is multiplied by a constant to lower it compared to the active cochlea undamping parameter without making it completely zero. This is representative of a case where the OHCs acting as a gain medium aren't producing as much gain as they should be, but are not totally dead. An example of such an undamping parameter is shown in Figure 7. I have implemented this functionality by adding a `runPartiallyPassive` class to the pyCochlea simulation class. This method accepts a single parameter which is the multiplier used for the undamping coefficient. A usage example is below:

```
#set up and initialize the simulation object
simObj = pyCochlea.sim()
signal = pyCochlea.sinFunc(250,50e-3)
simObj.initializeSignalFromArray(signal)

#run the partially passive simulation with an undamping multiplier of 0.5 and
    extract the BM Partially Passive response
BMPPassive = simObj.runPartiallyPassive(0.5)["BMPPassive"]
```

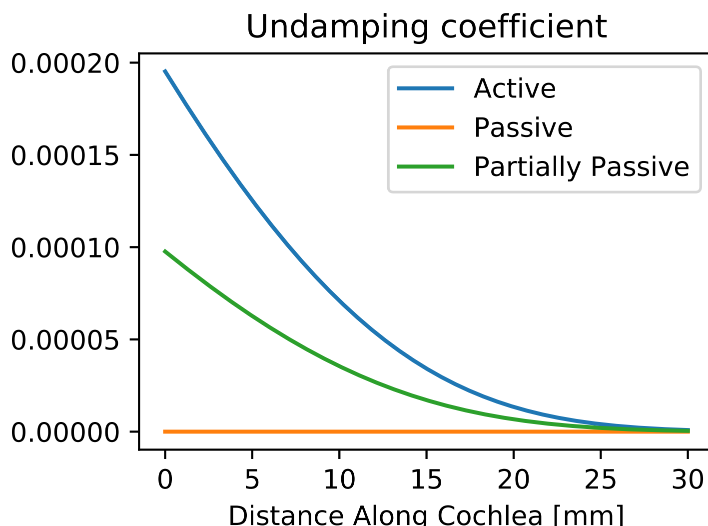

Figure 7: Undamping coefficient for a partially passive cochlea compared to those for an active cochlea and fully passive cochlea.

For a given optimization attempt, I will create a class object in python which initializes the required active response envelope and the passive response to an unfiltered signal for comparison. This class is responsible for setting up the simulation object used for each iteration of the optimizer as well as calculating and returning the value of the loss function to the optimizer. An example of such a script is included as an appendix to this document. Although I haven't yet gotten the result I'm looking for, I have found some interesting results.

Figure 8 shows one such interesting result. In this image, it's clear that there is some potential for this model to eliminate the frequency bleed effect in the passive cochlea. The large intensity spike is an artifact that may be caused by the way in which the tone complex is generated. More exploration needs to be done to find the source of this artifact and then to find a way to eliminate it.

Figure 9 is another result which shows some promise that this method may be able to reduce the magnitude of frequency bleed while maintaining an acceptable response magnitude in the signal region.

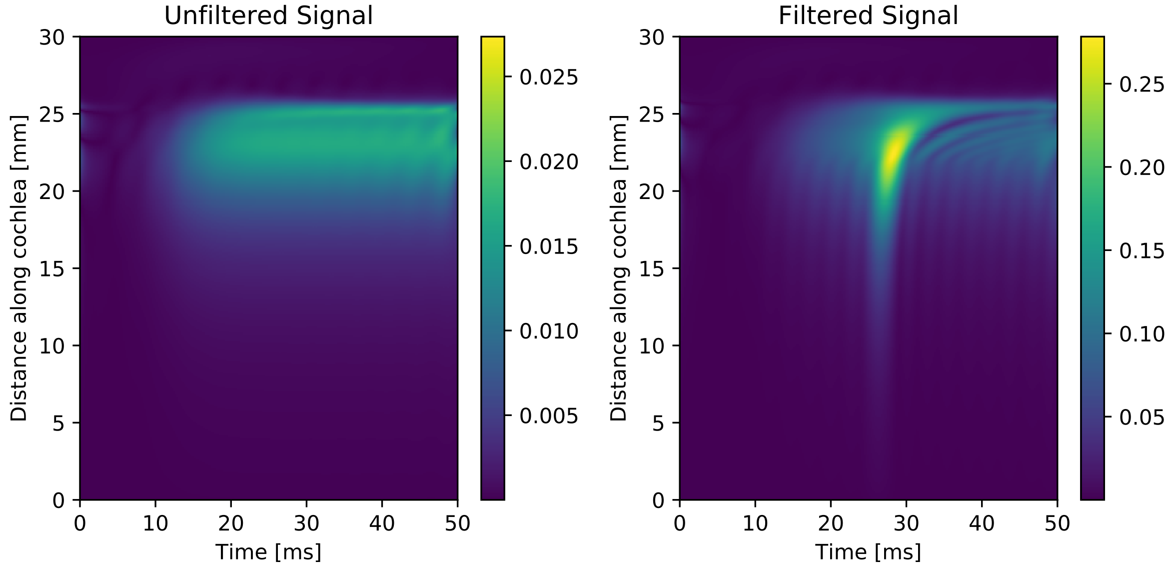

Figure 8: The figure on the left hand side is the envelope of the partially passive Basilar membrane (0.5) response to an unfiltered signal. The right hand side is the same Basilar membrane's response to a filtered signal. This result shows promise that this method will be able to cancel the "frequency bleed" in the passive cochlea model. The large intensity spike in this image is an unfortunate artifact. The optimized parameters are: $A = 3.55$, $\tau = 132.24$, $\Delta f = -16.4$, $f_{i-min} = 229.5$, $f_{i-max} = 571.6$. 10 frequencies are used in the tone complex.

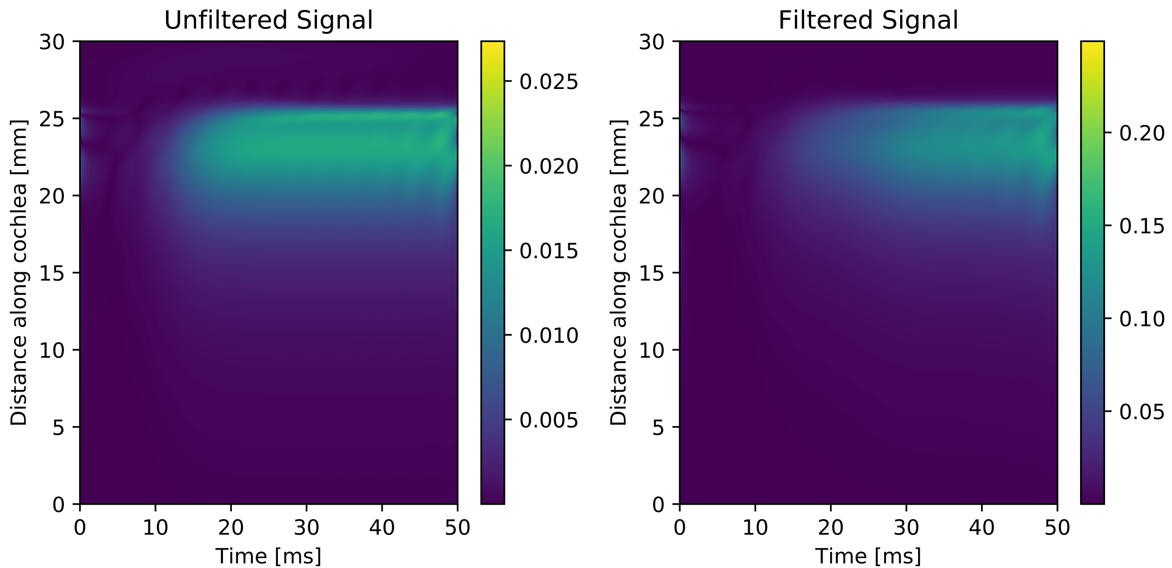

Figure 9: The plot on the left is the partially passive BM response to an unfiltered signal, and the figure on the right is the response to a filtered signal. This result shows that for short time at the beginning of the response, there is some reduction in the frequency bleed while maintaining reasonable signal in the signal region. This result is not yet optimal, but it is a promising preliminary result. The optimized parameters are: $A = 6.12$, $\tau = 16.45$, $\Delta f = -18.96$, $f_{i-min} = 238.8$, and $f_{i-max} = 510.14$.

4 Discussion

The results so far are promising, but there is a large number of other things that I will attempt in the coming weeks. As mentioned in the section on optimization, I would like to try scaling the parameters that are optimized over. Some care needs to be taken to ensure this is done in a way that does not take away from the physical meaning of the model parameters when used in the simulation.

Another factor that may be quite important is that when multiple frequencies are used in the filtering expression, the total magnitude of the signal may grow to a degree that the amplification parameter A is relatively meaningless in the context of comparing it between different optimization runs. The total signal should be normalized such that the maximum amplitude of the resulting waveform is unity before the amplification parameter is applied. As the amplitudes b_i are less than one by definition from Equation 6, these amplitudes will be relative to the signal carrying the central frequency and their proportions will be maintained as the whole signal is normalized. Doing this will make the amplification parameter A more meaningful.

Also as discussed in the optimization section, I would like to implement the use of analytical gradients for all loss functions to increase optimization convergence speed and accuracy. This will not be a difficult addition to make, but I have not had time to do so as of the writing of this report because of time spent on formulating the optimization problem.

Once the method discussed in this report is reliable for a single frequency f_c, it will be trivial to automate the training of filter parameters for a broad range of frequencies. When applied as an audio filtering mechanism in a hearing aid, this has the potential to dramatically improve a hearing-impaired patient's frequency acuity and therefore their ability to understand speech.

5 Closing remarks

In closing, I would like to mention a few things related more to my experience in this course than the subject of this report. My undergraduate degree is in Physics from a different university. As a second-semester Biomedical Engineering Master's student, I decided to enroll in this class because the application of machine learning methods to biomedical data sets is a research area that I am very much interested in. Before taking this course I had taken no courses in probability theory, and I had never trained a machine learning model. In combination with that, I had very limited knowledge about the biology of hearing before beginning this project. Over the course of this project I've learned a lot about the biology of hearing and even more about how machine learning models may possibly be applied to some biomedical data sets. I have also gained some understanding and appreciation for the mathematics that make sparse learning models work. Although I feel that I have much left to learn, I have made significant progress from where I was when I started this class. This course has reaffirmed my interest in the application of machine learning methods to biomedical data sets and I am excited to apply the knowledge I gained in this course to other biomedical problems in the future.

A Code

This project includes too much code to include all of it in this document. I have included some relevent scripts in this appendix. The original Matlab and my modified Python versions of the cochlear model are available under the `finalProject/code` folder of my GitHub repository.

A.1 Optimization script

This script is an example of how I am carrying out the optimization for this project. This file is also available on the GitHub repository.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import numpy as np
from scipy.optimize import minimize
import matplotlib.pyplot as plt
import pyCochlea

class optClass():
    def __init__(self):
        #The central frequency and maximum simulation time are defined here
        self.centralF = 250
        self.tMax = 50e-3

        #initialize a simulation object to get active cochlea results
        activeSim = pyCochlea.sim()

        #Initialize a pure sine wave at the centralF for the duration of the
           simulation
        activeSig = pyCochlea.sinFunc(self.centralF,self.tMax)

        #initialize the simulation with this signal
        activeSim.initializeSignalFromArray(activeSig)

        #Calculate the active response, extract the BM response, make the envelope,
           and
        #calculate the maximum value for the envelope.
        self.activeResp = np.array(activeSim.runActive()["BMActive"])[:,:,0].T
        self.activeEnvelope = pyCochlea.makeEnvelope(self.activeResp)
        self.activeEnvMax = np.max(self.activeEnvelope)

        #Initialize some sane defaults for thresholds used to generate the weights
           matrix
        #for the weighted MSE loss function
        self.weightsParams = [[0.01,0.1],
                [0.15,0.25],
                [0.3,1],
                [0.5,3]]
        #Generate the weights matrix
        self.weights = pyCochlea.genWeights(self.activeEnvelope,self.weightsParams)

        #Sum the weights. Used when calculating WMSE. Doing it here prevents it
           being done
        #for every iteration
        self.weightsSum = np.sum(self.weights)

        #Calculate the weighted active envelope
```

```
        self.weightedActiveEnvelope = self.weights*self.activeEnvelope

        #If this is set to True, a plot will be produced each time the
            singleIteration function is run.
        #Leave it set to False while optimizing, but it's useful to set this to True
             to see optimization results
        self.plot = False

        #Calculate the passive response to an unfiltered signal. Useful for
            comparison to passive cochlear response
        #to filtered signal
        passiveSim = pyCochlea.sim()
        passiveSim.initializeSignalFromArray(activeSig)
        #Note using a partially passive model with a multiplier of 0.5
        passiveResp = np.array(passiveSim.runPartiallyPassive(0.5)["BMPPassive"])
            [:,:,0].T
        self.passiveEnvelope = pyCochlea.makeEnvelope(passiveResp)

    def singleIteration(self,x):
        #scipy.optimize expects all parameters in a single vector. Split them out
            for convenience
        ampl = x[0]
        sigma = x[1]
        df = x[2]
        start = x[3]
        stop = x[4]

        #initialize an object for the passive cochlea simulation
        passiveSim = pyCochlea.sim()

        #Initialize an object for creating the tone complex. This will return the
            tone complex added to the
        #signal carrying the central frequency
        passiveCplxgen = pyCochlea.toneComplex(self.centralF+df,sigma,start,stop,10,
            self.tMax,phase=90)
        #Apply the amplification factor to the tone complex
        passiveCplx = ampl*passiveCplxgen.makeSoundComplex()

        #Initialize the model with the signal and extract the BM Partially Passive
            response
        passiveSim.initializeSignalFromArray(passiveCplx)
        passiveResp = np.array(passiveSim.runPartiallyPassive(0.5)["BMPPassive"])
            [:,:,0].T
        passiveEnvelope = pyCochlea.makeEnvelope(passiveResp)

        #The next few lines allow me to change which loss function is being used.
            Uncomment only one of these lines
        #to switch.

        #Weighted MSE
        # loss = 1/self.weightsSum*(np.sum((self.weightedActiveEnvelope-self.weights
            *passiveEnvelope)**2))

        #dice loss
        loss = self.dice_loss(self.activeEnvelope,passiveEnvelope)

        #log-cosh loss
```

```
    # loss = self.logcosh(self.activeEnvelope,passiveEnvelope)

    #Weighted MSE with the active and passive envelopes normalized such that the
        active envelope has a peak intensity of 1
    # loss = 1/self.weightedActiveEnvelope.size*(np.sum((self.activeEnvelope/np.
       max(np.abs(self.activeEnvelope))-self.passiveEnvelope/np.max(np.abs(self
       .activeEnvelope)))**2))

    #Create a plot if the plot flag is enabled
    if(self.plot):
        plt.figure(figsize=(8,4))
        plt.subplot(121)
        plt.imshow(self.passiveEnvelope,origin='lower',aspect='auto',extent
           =[0,50,0,30])
        plt.xlabel("Time [ms]")
        plt.ylabel("Distance along cochlea [mm]")
        plt.colorbar()
        plt.title("Unfiltered Signal")
        plt.subplot(122)
        plt.imshow(passiveEnvelope,origin='lower',aspect='auto',extent
           =[0,50,0,30])
        plt.xlabel("Time [ms]")
        plt.ylabel("Distance along cochlea [mm]")
        plt.colorbar()
        # plt.clim([0,0.2])
        plt.title("Filtered Signal")
        plt.tight_layout()
        plt.show()

    #return the loss function value to the optimizer
    return loss

#If the user chooses to select a different set of weight parameters after this
   class is initialized,
#this function must be used to make all of the relevent updates
def setWeightsParams(self,params):
    self.weightsParams = params
    self.weights = pyCochlea.genWeights(self.activeEnvelope,self.weightsParams)
    self.weightedActiveEnvelope = self.weights*self.activeEnvelope
    self.weightsSum = np.sum(self.weights)

#Calculate the dice coefficient loss
def dice_loss(self,y_true, y_pred):
    y_true_f = y_true.flatten()
    y_pred_f = y_pred.flatten()

    intersection = np.sum(y_true_f*y_pred_f)
    return 1-(2.* intersection)/(np.sum(y_true*y_true_f)+np.sum(y_pred_f*
       y_pred_f))


#calculate the logcosh loss
def logcosh(self,true, pred):
    loss = np.log(np.cosh(pred - true))
    return np.sum(loss)
```

```
if __name__ == '__main__':
    cochOpt = optClass()

    #An example of setting weight generation parameters
    # weightParams = [[0.00,0.05],[0.02,1],[0.09,4],[0.4,14]]
    # cochOpt.setWeightsParams(weightParams)

    #set some initial conditions
    x0 = [6,100,-5,250,500]

    # x0 = [  4.13474444,  11.42116341, -13.2452323 , 234.28161424, 500]

    #Use the scipy.optimize.minimize function to optimize over the initial
       conditions
    # optimizer = minimize(a.singleIteration,x0,method='cg',options={'maxiter':
       1000})
    optimizer = minimize(a.singleIteration,x0)

    #After a run, print output from the optimizer. This object could also be pickled
        and
    #saved for later
    print(optimizer)

    #After optimization, set the plot flag to true
    cochOpt.plot = True

    #Run a single iteration with the final results from the optimizer and plot
    #the results
    cochOpt.singleIteration(optimizer.x)
```

A.2 Sound complex generation

A second relevent piece of code is the one which I'm using to generate the sound complex. I have included it in this appendix, but it is also available under the `finalProject/code/pyCochlea` folder of my GitHub repository.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import numpy as np

'''
This class implements a complex of overtones that should be added to a central
    frequency.
It will produce a tone complex where the amplitude of the tones are controlled by a
    gaussian
or decaying exponential function.
'''

class toneComplex:
    def __init__(self,centralF,tau,start,stop,N,tMax,dt=(1/44.1e3),phase=90):
        #Define all filter parameters
        self.centralF = centralF
        self.tau    = tau
        self.N         = N

        self.start    = start
        self.stop     = stop
        df            = (stop-start)/(N-1)
        #create a vector of frequencies which will be in the tone complex
        self.fVec     = np.array([start+i*df for i in range(N)])

        #Set up a phase vector. The phase for all tones in the complex are the same
            for now
        #(except the central frequency which has phase 0)
        self.phaseVec = np.zeros(N+1)
        self.phaseVec[1:] = phase

        #Create a data structure which will hold all the signals that are generated.
             Useful for analysis later
        tN = int(np.ceil(tMax/dt)+1)
        self.timeVec = np.array([i*dt for i in range(tN)])
        self.signalVec = np.zeros([N+1,tN])
        self.signalVec[0] = self.makeSingleSineWave(centralF,phi=0)

        #This is a functional form with a sloping start which prevents some
        #adverse effects in the model.
        to = 500/44.1e3
        a = 200
        self.form = 0.5*(np.tanh(a*(self.timeVec-to))+1)

    '''
    Inputs:
    - f: frequency for which to get intensity multiplier

    outputs:
    - relAmpl: amplitude relative to the central frequency

    There are two possibilities in this script: a decaying exponential, and a
```

```
        gaussian. The decaying exponential seems
    to work better.
    '''
    def intensityCurve(self,f):
        centralF = self.centralF
        tau = self.tau
        # return np.exp(-(f-centralF)**2/(tau**2))
        return np.exp(-(f-centralF)/tau)

    '''
    This equation generates a single sine wave based on input parameters. This will
        be run in a loop.
    '''
    def makeSingleSineWave(self,f,phi=0):
        ampl = self.intensityCurve(f)
        phi = np.deg2rad(phi)

        sineWave = ampl*np.sin(2*np.pi*f*self.timeVec+phi)
        return sineWave

    def makeSoundComplex(self):
        #Loop through the frequency vector to build the signals that will be
            combined into the final signal
        for i,f in enumerate(self.fVec):
            self.signalVec[i+1] = self.makeSingleSineWave(f,self.phaseVec[i+1])

        #Combine the signals, multiply it by the gently sloping form, and return the
             final signal.
        self.soundComplex = self.form*np.sum(self.signalVec,axis=0)
        return self.soundComplex

'''
A brief usage example of the overtone complex generator.
'''
if __name__ == '__main__':
    a = toneComplex(centralF = 250,tau=120,start=250,stop=400,N=100,tMax=50e-3,phase
        =89)
    signal = a.makeSoundComplex()

    import matplotlib.pyplot as plt
    plt.plot(a.timeVec, signal)
    plt.show()
```