lab05_audio_partial

March 16, 2018

1 Lab 5: Pitch Detection in Audio

In this lab, we will use numerical optimization to find the pitch and harmonics in a simple audio signal. In addition to the concepts in the gradient descent demo, you will learn to: * Load, visualize and play audio recordings * Divide audio data into frames * Perform nested minimization

The ML method presented here for pitch detection is actually not a very good one. As we will see, it is highly susceptible to local minima and quite slow. There are several better pitch detection algorithms, mostly using frequency-domain techniques. But, the method here will illustrate non-linear estimation well.

1.1 Reading the Audio File

Python provides a very simple method to read a `wav` file in the `scipy.io.wavefile` package. We first load that along with the other packages.

```
In [1]: from scipy.io.wavfile import read
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

In the github repository, you should find a file, `viola.wav`. Download this file to your local directory. Although the file is included in the github repository, you can find it along with many other audio samples in CCRMA audio website. After you have downloaded the file, you can then read the file with the `read` command. Print the sample rate in Hz, the number of samples in the file and the file length in seconds.

```
In [2]: # Read the file
        sr, y = read('viola.wav')

        # Convert to floating point values so that compuations below do not overflow
        y = y.astype(float)

        # TODO:  Print sample rate, number of samples and file length in seconds.
        print('Sample rate: %dHz' % sr)
        print('The number of samples: %d' % len(y))
        print('The file length: %.4fs' % (len(y)/sr))
```

```
Sample rate: 44100Hz
The number of samples: 299350
The file length: 6.7880s
```

You can then play the file with the following command. You should hear the viola play a sequence of simple notes.

```
In [3]: import IPython.display as ipd
        ipd.Audio(y, rate=sr) # load a NumPy array
```

```
Out[3]: <IPython.lib.display.Audio object>
```

For the analysis below, it will be easier to re-scale the samples so that they have an average squared value of 1. Find the `scale` value in the code below to do this.

```
In [4]: # TODO
        scale = np.sqrt(np.sum((y/np.sqrt(len(y)))**2))
        y = y / scale
```

1.2 Dividing the Audio File into Frames

In audio processing, it is common to divide audio streams into short frames (typically between 10 to 40 ms long). Since frames are often processed with an FFT, the frames are typically a power of two. Analysis is then performed in the frames separately. Given the vector `y`, create a `nfft x nframe` matrix `yframe` where

```
yframe[:,0] = samples y[k], k=0,...,nfft-1
yframe[:,1] = samples y[k], k=nfft,...,2*nfft-1,
yframe[:,2] = samples y[k], k=2*nfft,...,3*nfft-1,
...
```

You can do this with the `reshape` command with `order=F`. Zero pad `y` if the number of samples of `y` is not divisible by `nfft`. Print the total number of frames as well as the length (in milliseconds) of each frame.

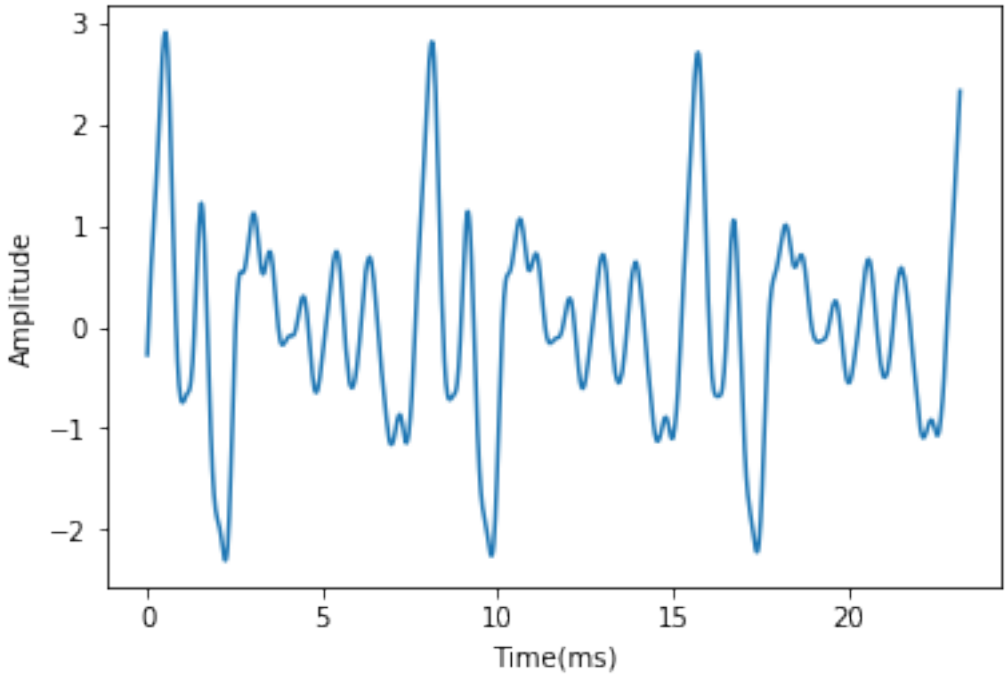
Note that in actual audio processing, the frames are typically overlapping and use careful windowing. But, we will ignore that here for simplicity.

```
In [5]: # Frame size
        nfft = 1024

        # TODO:
        import math
        nframe = math.ceil(len(y) / nfft)
        yframe = np.pad(y, (0, nfft - len(y) % nfft), 'constant').reshape(nfft, nframe, order=
        print('Total number of frames: %d' % nframe)
        print('Length of each frame: %.4fms' % (nfft/sr*1000))
```

```
Total number of frames: 293
Length of each frame: 23.2200ms
```

Let `i0=10` and set `yi=yframe[:,i0]` be the samples of frame `i0`. We will use this frame for most of the rest of the lab. Plot the samples of `yi`. Label the time axis in milliseconds (ms).

```
In [6]: # Get samples from frame 10
        i0 = 10
        yi = yframe[:,i0]

        # TODO:  Plot yi vs. time (in ms)
        plt.plot(np.array(range(len(yi)))/sr*1000, yi)
        plt.xlabel('Time(ms)')
        plt.ylabel('Amplitude')
        plt.show()
```



1.3 Fitting a Multi-Sinusoid

A common model for audio samples, `yi[k]`, containing an instrument playing a single note is the multi-sinusoid model:

```
yi[k] \approx yhati[k] = c + \sum_{j=0}^{nterms-1} a[j]*cos(2*np.pi*k*freq0*(j+1)/sr)
                                                 +  b[j]*sin(2*np.pi*k*freq0*(j+1)/sr),
```

where `sr` is the sample rate. The parameter `freq0` is called the fundamental frequency and the audio signal is modeled as being composed of sinusoids and cosinusoids with frequencies equal to integer multiples of the fundamental. In audio processing, these terms are called *harmonics*. In

analyzing audio signals, a common goal is to determine both the fundamental frequency `freq0` (the pitch of the audio) as well as the coefficients of the harmonics,

`beta = (c, a[0], ..., a[nterms-1], b[0], ..., b[nterms-1]).`

To find the parameters, we will fit the mean squared error loss function:

`mse(freq0,beta) := 1/N * \sum_k (yi[k] - yhati[k])**2,  N = len(yi).`

In practice, a separate model would be fit for each audio frame. But, in this lab, we will mostly look at a single frame.

1.3.1 Nested Minimization

We will perform the minimization of `mse` in a nested manner: First, given a fundamental frequency `freq0`, we minimize over the coefficients `beta`. Call this minimum `mse1`:

`mse1(freq0) := min_beta mse(freq0,beta)`

Importantly, this minimizaiton can be performed by least-squares. Then, we find the fundamental frequency `freq0` by minimizing `mse1`:

`min_{freq0} mse1(freq0)`

We will use gradient-descent minimization with `mse1(freq0)` as the objective function. This form of *nested* minimization is commonly used whenever we can minimize over one set of parameters easily given the other.

1.4 Setting Up the Objective Function

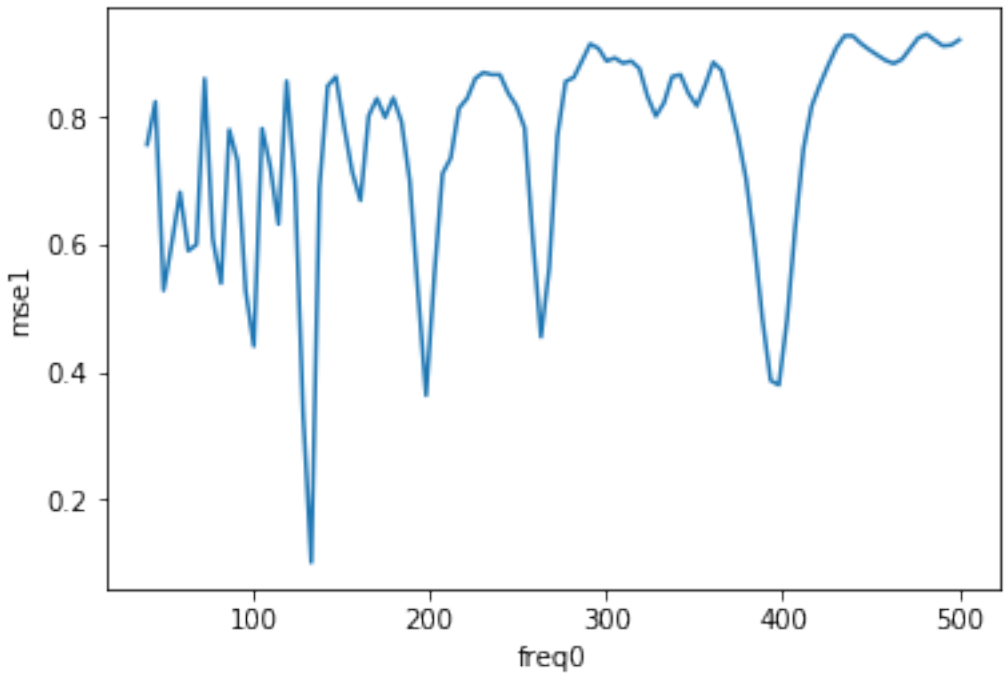
We will use the class `AudioFitFn` below to perform the two-part minimization. Complete the `feval` method in the class. The method should take the argument `freq0` and perform the minimization of the MSE over `beta`. Specifically, fill the code in `feval` to perform the following: * Construct a matrix, `A` such that `yhati = A*beta`.
* Find `betahat` with the `np.linalg.lstsq()` method using the matrix `A` and the samples `self.yi`. This is simpler than constructing a linear regression object.
* Compute and store the estimate `self.yhati = A.dot(betahat)`. * Compute the `mse1`, the minimum MSE, by comparing `self.yhati` and `self.yi`. * For now, set the gradient to `mse1_grad=0`. We will fill this part in later.
* Return `mse1` and `mse1_grad`.

```
In [7]: class AudioFitFn(object):
            def __init__(self,yi,sr=44100,nterms=8):
                """
                A class for fitting

                yi:  One frame of audio
                sr:  Sample rate (in Hz)
                nterms:  Number of harmonics used in the model (default=8)
                """
```

```
        self.yi = yi
        self.sr = sr
        self.nterms = nterms

    def feval(self,freq0):
        """
        Optimization function for audio fitting.  Given a fundamental frequency, freq0
        method performs a least squares fit for the audio sample using the model:

        yhati[k] = c + \sum_{j=0}^{nterms-1} a[j]*cos(2*np.pi*k*freq0*(j+1)/sr)
                                             +  b[j]*sin(2*np.pi*k*freq0*(j+1)/sr)

        The coefficients beta = [c,a[0],...,a[nterms-1],b[0],...,b[nterms-1]]
        are found by least squares.

        Returns:

        mse1:   The MSE of the best least square fit.
        mse1_grad:  The gradient of mse1 wrt to the parameter freq0
        """

        # TODO
        # TODO Write code to find optimal beta to minimize MSE and find minimal MSE wi
        cos_vals = [[np.cos(2*np.pi*k*freq0*(j+1)/self.sr) \
                     for j in range(self.nterms)] for k in range(len(self.yi))]
        sin_vals = [[np.sin(2*np.pi*k*freq0*(j+1)/self.sr) \
                     for j in range(self.nterms)] for k in range(len(self.yi))]
        A = np.column_stack((np.ones(len(self.yi),), cos_vals, sin_vals))
        beta = np.linalg.lstsq(A, self.yi, rcond=None)
        self.yhati = A.dot(beta[0])
        mse1 = np.mean((self.yi-self.yhati)**2)

        # Compute the gradient wrt to freq0
        mse1_grad = 0
        return mse1, mse1_grad
```

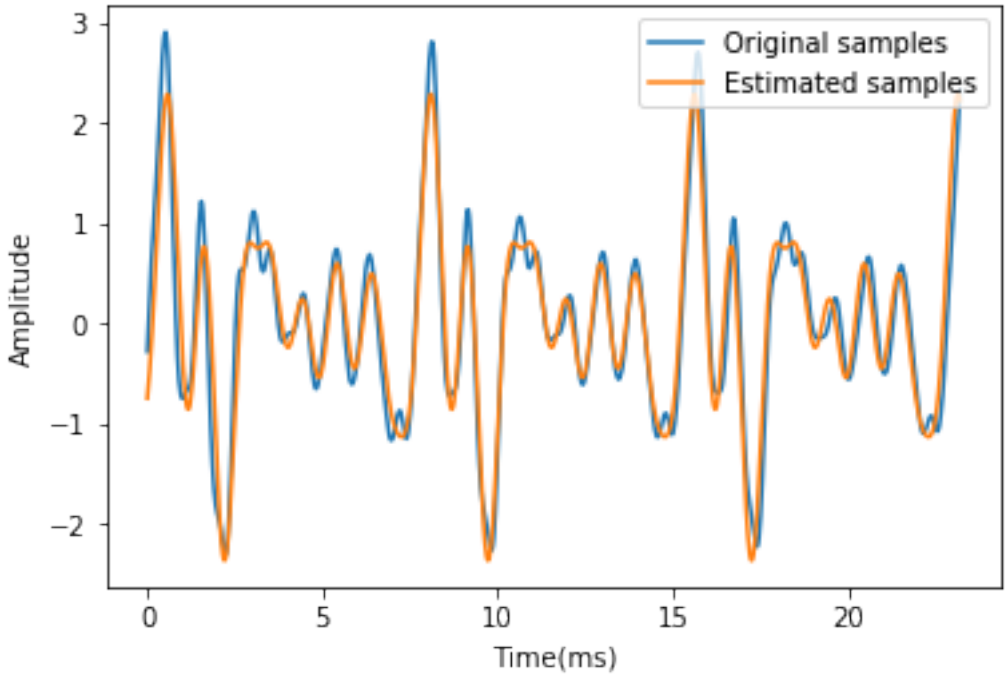
Instantiate an object, `audio_fn` from the class `AudioFitFn` with the samples `yi`. Then, using the `feval` method, compute and plot `mse1` for values `freq0` in the range of 0 to 500 Hz with a step size of 0.5Hz. You should see a minimum around `freq0 = 131` Hz, but there are several other local minima.

```
In [8]: # TODO
        audio_fn = AudioFitFn(yi, sr)
        freq0 = np.linspace(40, 500, 100)
        mse1 = np.array([audio_fn.feval(f)[0] for f in freq0])
        plt.plot(freq0, mse1)
        plt.xlabel('freq0')
        plt.ylabel('mse1')
        plt.show()
```



Determine and print the value of `freq0` that achieves the minimum `mse1`. Also, plot the estimated function `audio_fn.yhati` for that `freq0` along with the original samples `yi`.

```
In [9]: # TODO
        min_freq0 = freq0[np.argmin(mse1)]
        print('freq0 that minimums mse1: %.6fHz' % min_freq0)
        audio_fn.feval(min_freq0)
        plt.plot(np.array(range(len(yi)))/sr*1000, \
                 yi, label='Original samples')
        plt.plot(np.array(range(len(yi)))/sr*1000, \
                 audio_fn.yhati, label='Estimated samples')
        plt.xlabel('Time(ms)')
        plt.ylabel('Amplitude')
        plt.legend()
        plt.show()

freq0 that minimums mse1: 132.929293Hz
```



1.5 Computing the Gradient

The above method found the estimate for `freq0` by performing a search over 100 different frequency values and selecting the frequency value with the lowest MSE. We now see if we can estimate the frequency with gradient descent minimization of the MSE. We first need to modify the `feval` method in the `AudioFitFn` class above to compute the gradient. Some elementary calculus (see the homework), shows that

```
dmse1(freq0)/dfreq0 = dmse(freq0,betahat)/dfreq0
```

So, we just need to evaluate the partial derivative of `mse = np.mean((yi-yhati)**2)` with respect to the parameter `freq0` holding the parameters `beta=betahat`. Modify the `feval` method above to compute the gradient and return the gradient in `mse1_grad`.

```
In [10]: class AudioFitFn(object):
             def __init__(self,yi,sr=44100,nterms=8):
                 """
                 A class for fitting

                 yi:  One frame of audio
                 sr:  Sample rate (in Hz)
                 nterms:  Number of harmonics used in the model (default=8)
                 """
                 self.yi = yi
                 self.sr = sr
```

```
        self.nterms = nterms

    def feval(self,freq0):
        """
        Optimization function for audio fitting.  Given a fundamental frequency, freq
        method performs a least squares fit for the audio sample using the model:

        yhati[k] = c + \sum_{j=0}^{nterms-1} a[j]*cos(2*np.pi*k*freq0*(j+1)/sr)
                                             +  b[j]*sin(2*np.pi*k*freq0*(j+1)/sr)

        The coefficients beta = [c,a[0],...,a[nterms-1],b[0],...,b[nterms-1]]
        are found by least squares.

        Returns:

        mse1:   The MSE of the best least square fit.
        mse1_grad:  The gradient of mse1 wrt to the parameter freq0
        """

        # TODO
        cos_vals = [[np.cos(2*np.pi*k*freq0*(j+1)/self.sr) \
                     for j in range(self.nterms)] for k in range(len(self.yi))]
        sin_vals = [[np.sin(2*np.pi*k*freq0*(j+1)/self.sr) \
                     for j in range(self.nterms)] for k in range(len(self.yi))]
        A = np.column_stack((np.ones(len(self.yi),), cos_vals, sin_vals))
        beta = np.linalg.lstsq(A, self.yi, rcond=None)
        self.yhati = A.dot(beta[0])
        mse1 = np.mean((self.yi-self.yhati)**2)

        # Compute the gradient wrt to freq0

        # TODO
        dcos_vals = [[np.sin(-2*np.pi*k*freq0*(j+1)/self.sr)*(2*np.pi*k*(j+1)/self.sr
                      for j in range(self.nterms)] for k in range(len(self.yi))]
        dsin_vals = [[np.cos(2*np.pi*k*freq0*(j+1)/self.sr)*(2*np.pi*k*(j+1)/self.sr)
                      for j in range(self.nterms)] for k in range(len(self.yi))]
        A_grad = np.column_stack((np.zeros(len(self.yi),), dcos_vals, dsin_vals))
        self.yhati_grad = A_grad.dot(beta[0])
        mse1_grad = np.mean(2*(self.yhati-self.yi)*self.yhati_grad)
        return mse1, mse1_grad
```

Now, test the gradient by taking two close values of `freq0`, say `freq0_0` and `freq0_1` and verifying that first-order approximation holds.

In [11]:
```
# TODO
audio_fn = AudioFitFn(yi, sr)

freq0_0 = min_freq0
```

```
        freq0_1 = freq0_0 + 1e-6

        mse1_0, mse1_grad_0 = audio_fn.feval(freq0_0)
        mse1_1, _           = audio_fn.feval(freq0_1)

        print('LHS = %.12e' % (mse1_0-mse1_1))
        print('RHS = %.12e' % ((mse1_grad_0) * (freq0_0-freq0_1)))
```

```
LHS = -1.045718676929e-07
RHS = -1.045718388664e-07
```

1.6 Run the Optimizer

We cut and paste the optimizer from the gradient descent demo.

```
In [12]: def grad_opt_adapt(feval, winit, nit=1000, lr_init=1e-3):
             """
             Gradient descent optimization with adaptive step size

             feval:  A function that returns f, fgrad, the objective
                     function and its gradient
             winit:  Initial estimate
             nit:    Number of iterations
             lr:     Initial learning rate

             Returns:
             w:   Final estimate for the optimal
             f0:  Function at the optimal
             """

             # Set initial point
             w0 = winit
             f0, fgrad0 = feval(w0)
             lr = lr_init

             # Create history dictionary for tracking progress per iteration.
             # This isn't necessary if you just want the final answer, but it
             # is useful for debugging
             hist = {'lr': [], 'w': [], 'f': []}

             for it in range(nit):

                 # Take a gradient step
                 w1 = w0 - lr*fgrad0

                 # Evaluate the test point by computing the objective function, f1,
                 # at the test point and the predicted decrease, df_est
```

```
            f1, fgrad1 = feval(w1)
            df_est = np.dot(fgrad0, (w1-w0))

            # Check if test point passes the Armijo rule
            alpha = 0.5
            if (f1-f0 < alpha*df_est) and (f1 < f0):
                # If descent is sufficient, accept the point and increase the
                # learning rate
                lr = lr*2
                f0 = f1
                fgrad0 = fgrad1
                w0 = w1
            else:
                # Otherwise, decrease the learning rate
                lr = lr/2

            # Save history
            hist['f'].append(f0)
            hist['lr'].append(lr)
            hist['w'].append(w0)

        # Convert to numpy arrays
        for elem in ('f', 'lr', 'w'):
            hist[elem] = np.array(hist[elem])
        return w0, f0, hist
```

Now, run the optimizer with the feval function with a starting estimate for freq0 = 130 Hz. Use `lr_init=1e-3` and `f0_init=130`. Print the final frequency estimate. Also, print the midi number of the estimated frequency:

```
midi_num = 12*log2(freq/440 Hz) + 69
```

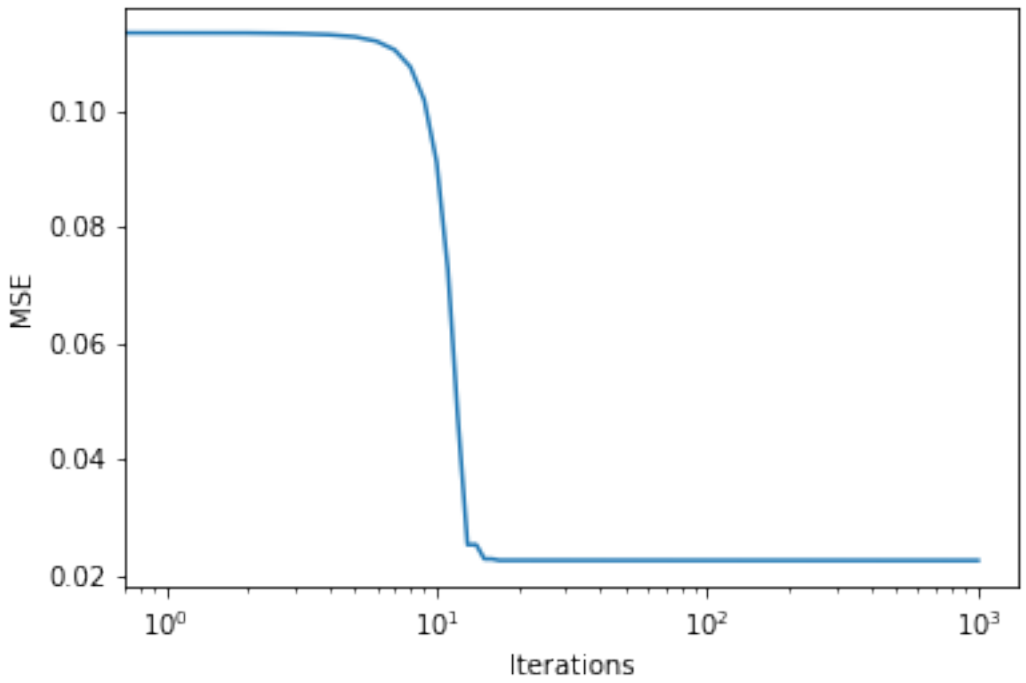
If the note was exactly a musical note, `midi_num` should be an integer. But you will see that the frequency does not exactly lie on a note since the pitch in a viola bends around the note.

```
In [13]: # TODO
         lr_init = 1e-3
         f0_init = 130
         freq0_min, mse_min, hist = grad_opt_adapt(audio_fn.feval, winit=f0_init, lr_init=lr_i

         midi_num = 12 * np.log2(freq0_min / 440) + 69

         print("f0_min=%f midi=%f" % (freq0_min, midi_num))
```

```
f0_min=131.528923 midi=48.094519
```

Plot the MSE as a function of the iteration. You should tell whether you gradient algorithm has converged. If not, you may need to adjust the number of iterations.

```
In [14]: #TODO
         plt.semilogx(hist['f'])
         plt.xlabel('Iterations')
         plt.ylabel('MSE')
         plt.show()
```



Compare your solution of f0_min with the one obtained using exhaustive search. You may notice a slight difference. Explain why and also which solution is likely to be correct?

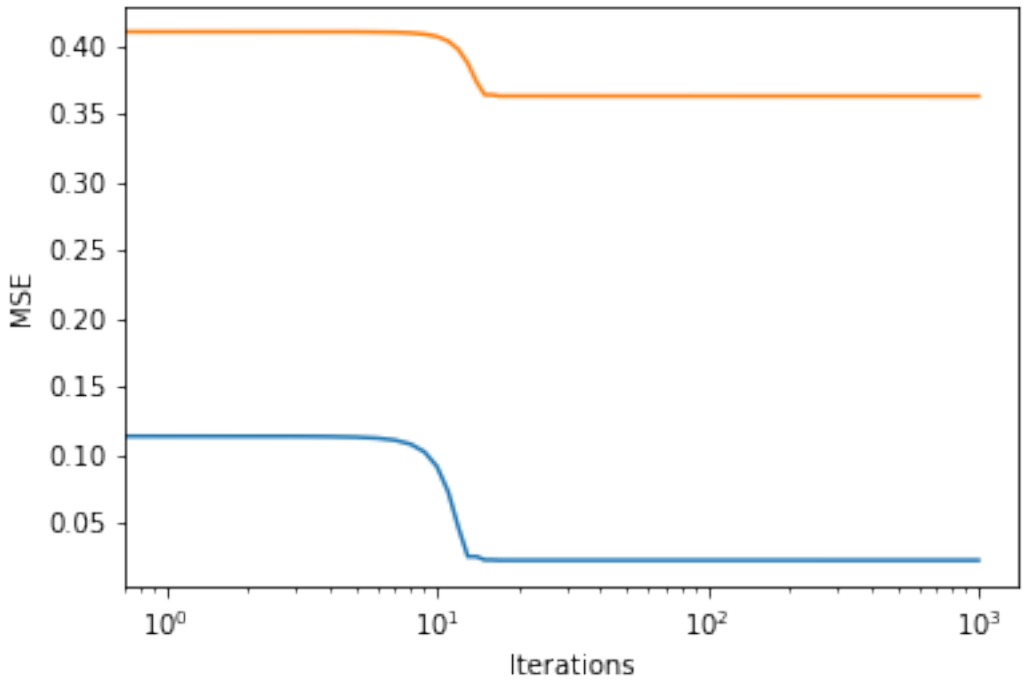
Answer:

This is because the samples of frequency are discrete. That is, if the number of freq0 is infinity, and the step size is extremely near 0, we should get the same results with these two method. However, in this case, the latter one is more likely to be correct, since it has a small step size, which corresponds to a higher accuracy.

Now, repeat with an initial frequency of 200 Hz. Print the final estimated frequency. Also plot the MSE per iteration on the same graph as the MSE per iteration with the initial condition = 130 Hz.

```
In [15]: # TODO
         freq0_min1, mse_min1, hist1 = grad_opt_adapt(audio_fn.feval, winit=200, lr_init=1e-3)
         plt.semilogx(hist['f'])
         plt.semilogx(hist1['f'])
         plt.xlabel('Iterations')
         plt.ylabel('MSE')
         print('Final estimated frequency: %f' % freq0_min1)
```

```
Final estimated frequency: 197.872343
```



Did you get the same solution as when you used an initial solution of 130 Hz? Why?
Answer:
No. The optimizer does not reach the real minimum MSE, since the initial frequency led to another local minimum. So a good initial value is required if a good result is expected.

1.7 More Fun

While the above method does not work very well, there are many good approaches. For one thing, we can obtain a good initial condition using an FFT of the frame. The FFT is used in many pitch detection methods. More difficult problems include multi-tone detection, chord detection and instrument separation. A useful python library that contains all sorts of interesting audio analysis tools in the librosa package.