lab_audio_partial

October 18, 2017

1 Lab 5: Pitch Detection in Audio

James Johnston
EL 9123
Professor Sundeep Rangan
jrj346
N12411794

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 nonlinear 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 [2]: from scipy.io.wavfile import read
        import numpy as np
        import matplotlib.pyplot as plt
        import math
        %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 [3]: # Read the file
        sr, y = read('viola.wav')
        y = y.astype(float)
```

```
nsamp = y.size
flen = nsamp/sr

#Print sample rate
print('sample rate:',sr)

#number of samples
print('sample numebr:', nsamp)

#file length in seconds.
print('file length in sec:',flen)
```

```
sample rate: 44100
sample numebr: 299350
file length in sec: 6.787981859410431
```

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

```
In [4]: import IPython.display as ipd
        ipd.Audio(y, rate=sr) # load a NumPy array
```

```
Out[4]: <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 [5]: scale = np.sqrt(np.mean(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.

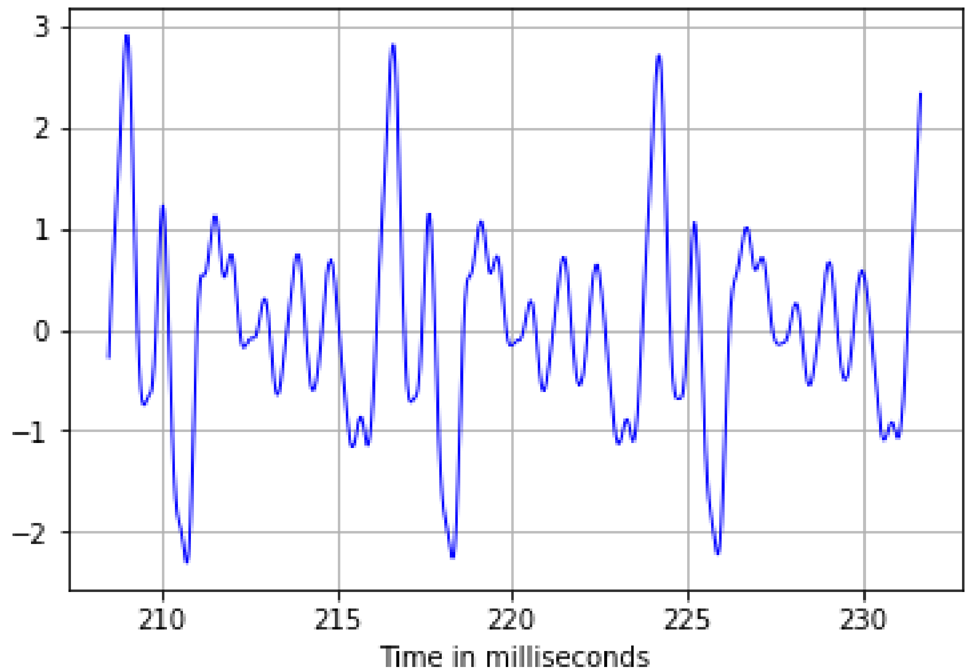
```
In [6]: # Frame size
        nfft = 1024
        nframe = nsamp//nfft + 1
        length = flen*1000 / nframe
        lastNum = nsamp - nfft*(nframe-1) # the number of elements in the last column.
        num_0s = nfft - lastNum
        y_pad = np.lib.pad(y,(0,num_0s),'constant',constant_values=(0))
        yframe = np.reshape(y_pad,[nfft,nframe], order='F')
        print('length of each frame in milliseconds:', length)
        print('total num of frames:', nframe)
```

```
length of each frame in milliseconds: 23.167173581605567
total num of frames: 293
```

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 [7]: # Get samples from frame 10
        i0 = 10
        yi = yframe[:,i0]
        timeStart = (i0-1)*length
        timeEnd = i0*length
        #Plot yi vs. time (in ms)
        time = np.linspace(timeStart, timeEnd, num=nfft, endpoint=True)
        plt.plot(time, yi,'b-',linewidth=1)
        plt.grid()
        plt.xlabel('Time in milliseconds')
```

```
Out[7]: <matplotlib.text.Text at 0x10cffda90>
```



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 [8]: 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
        """

        size = yi.shape[0]
        X = np.zeros([size,2*self.nterms])

        for j in range(self.nterms*2):
            for k in range(size):
                if j<self.nterms:
                    # a[0] to a[7]
                    X[k,j] = math.cos(2*np.pi*k*freq0*(j+1)/self.sr)
                else:
                    #b[0] to b[7]
                    X[k,j] = math.sin(2*np.pi*k*freq0*(j-self.nterms+1)/self.sr)

        ones = np.ones((size,1))
        A = np.hstack((ones, X))

        # Find betahat
        betahat = np.linalg.lstsq(A,self.yi)[0]
        self.yhati = A.dot(betahat)

        # mse1
        mse1 = np.mean((self.yi-self.yhati)**2)
        # Compute the gradient wrt to freq0
        mse1_grad = 0
        return mse1, mse1_grad
```

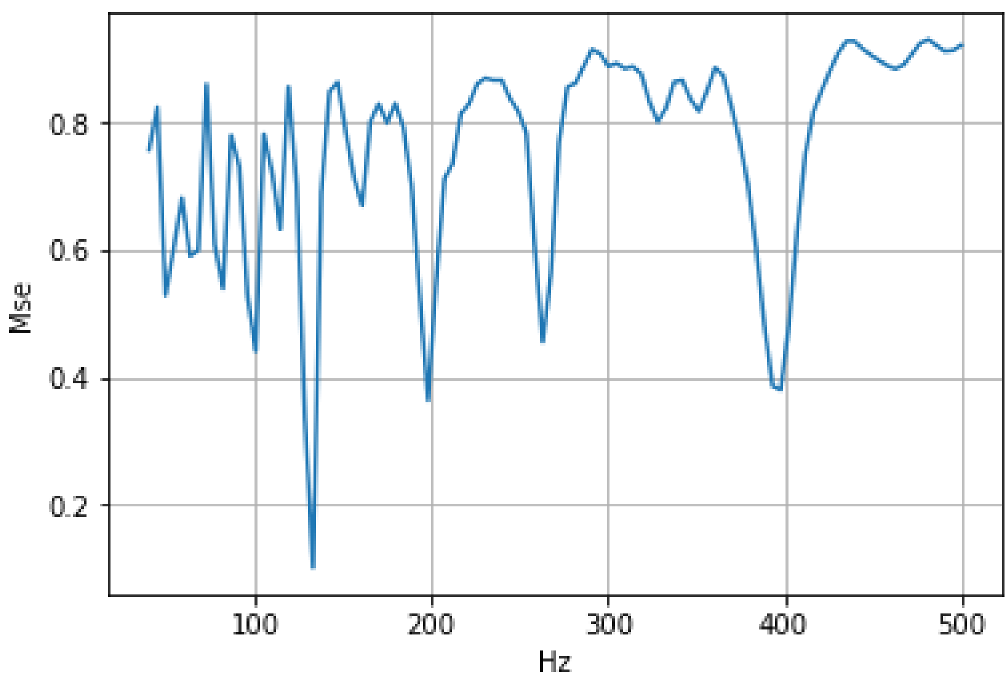
Instatiate an object, `audio_fn` from the class `AudioFitFn` with the samples `yi`. Then, using the `feval` method, compute and plot `mse1` for 100 values `freq0` in the range of 40 to 500 Hz. You should see a minimum around `freq0 = 130` Hz, but there are several other local minima.

```
In [9]: n = 100
        audio_fn = AudioFitFn(yi)
        freq0 = np.linspace(40,500,n) #n = 100
        mse = []
        for i in range(n):
            mse1,mse1_grad = audio_fn.feval(freq0[i])
            mse = np.append(mse,mse1)
```

```
#plot
plt.plot(freq0, mse)
plt.grid()
plt.xlabel('Hz')
plt.ylabel('Mse')
```

```
132.929292929
```

```
Out[9]: <matplotlib.text.Text at 0x10db0be10>
```

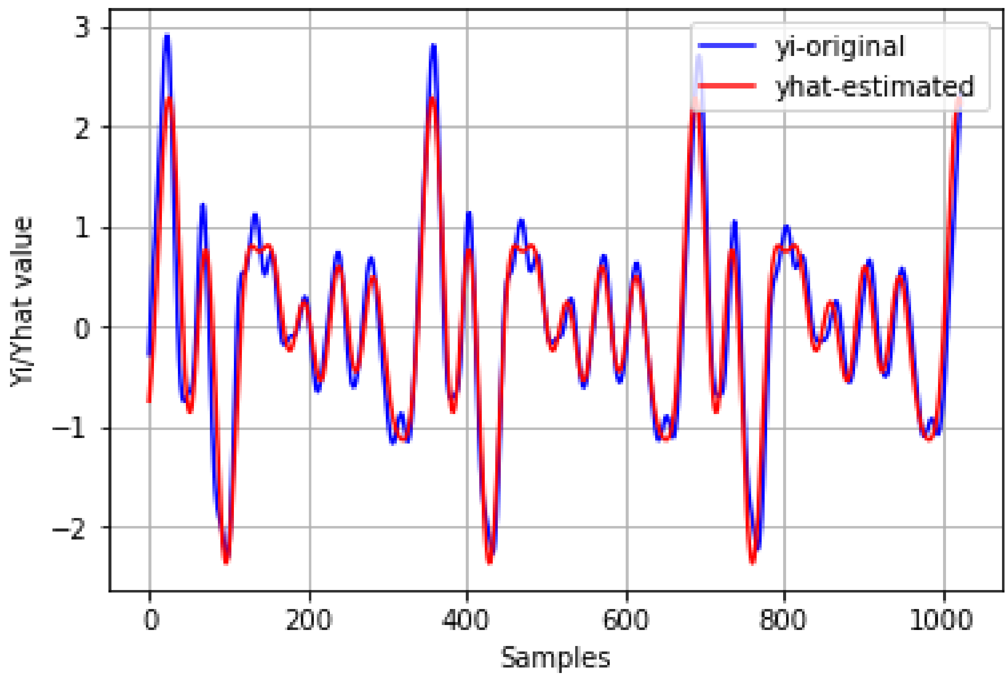


Print the value of freq0 that achieves the minimum mse1. Also, plot the estimated function audio_fn.yhati for that along with the original samples yi.

```
In [16]: imin1 = np.argmin(mse) # is this supposed to be mse, collection of appended msel or i
         print(" value of freq0 that achieves the minimum mse1",freq0[imin1])
         x_ = np.arange(0,1024)
         audio_fn.feval(freq0[imin1])
         yhat = audio_fn.yhati

         plt.plot(x_, yi,'b-') # original samples

         plt.plot(x_, yhat,'r-') # estimated function

         plt.legend(['yi-original','yhat-estimated'],loc='upper right')
```

```
plt.xlabel('Samples')
plt.ylabel('Yi/Yhat value')
plt.grid()
```

```
value of freq0 that achieves the minimum mse1 132.929292929
```



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`.

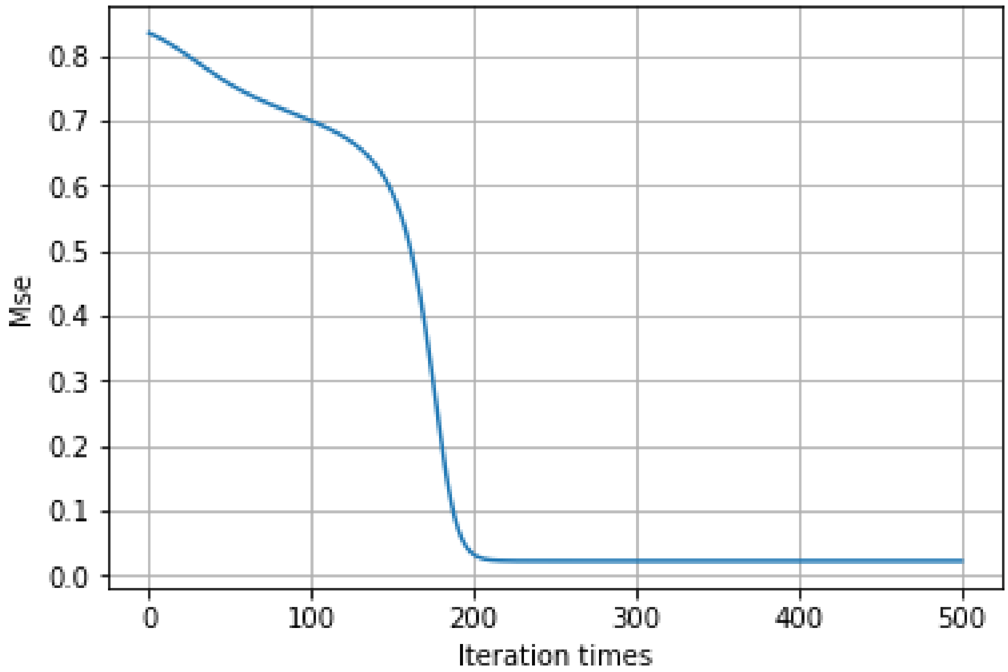
Then, test the gradient by taking two close values of `freq0`, say `freq0_0` and `freq0_1` and verifying that first-order approximation holds.

```
In [12]: class Audio(object):
             def __init__(self,yi,sr=44100,nterms=8):
```

```
        self.yi = yi
        self.sr = sr
        self.nterms = nterms
        self.nsamp = self.yi.shape[0]
    def feval(self,freq0):
        # Construct matrix A
        size = self.yi.shape[0]
        X = np.zeros([size,2*self.nterms])
        for j in range(self.nterms*2):
            for k in range(size):
                if j<self.nterms:
                    # a[0] to a[7]

                    X[k,j] = math.cos(2*np.pi*k*freq0*(j+1)/self.sr)
                else:
                    #b[0] to b[7]
                    X[k,j] = math.sin(2*np.pi*k*freq0*(j-self.nterms+1)/self.sr)
        A = np.column_stack((np.ones(size,), X))
        # Find betahat
        betahat = np.linalg.lstsq(A,self.yi)[0]
        yhati = A.dot(betahat)
        # mse1
        mse1 = np.mean((yhati-self.yi)**2)
        # Compute the gradient wrt to freq0
        mse1_grad = self.grad(freq0, betahat, yhati)
        return mse1, mse1_grad
    # grad method to calculate the partial derivative of mse respect to freq0
    def grad(self,f,beta,yhati):
        tmp = 0
        for k in range(self.nsamp):
            dmse_df = self.cal_df(beta,f,k)
            tmp = tmp + (self.yi[k]-yhati[k])*dmse_df
        grad = (2/self.nsamp)*tmp
        return grad
    # cal_df method helps grad divide calculation into small parts
    def cal_df(self,beta,f,k):
        summ = 0
        tmp1 = 0
        tmp2 = 0
        for j in range(self.nterms):
            coef1 = 2*np.pi*k*(j+1)/self.sr
            aj = beta[j+1]
            tmp1 = tmp1 + aj*math.sin(coef1*f)*coef1
        for j in range(self.nterms):
            coef2 = 2*np.pi*k*(j+1)/self.sr
            bj = beta[j+1+self.nterms]
            tmp2 = tmp2 + bj*math.cos(coef2*f)*coef2
        summ = tmp1-tmp2
```

```
        return summ

# take the start point and iteration times to do the iteration.
def iterate(startFreq,itr):
    lastF = startFreq
    a = Audio(yi)
    mseL = []
    mse, mse_grad = a.feval(lastF)
    mseL = np.append(mseL,mse)
    for i in range(itr):
        newF = lastF - 1*mse_grad
        mse,mse_grad = a.feval(newF)
        lastF = newF
        mseL = np.append(mseL,mse)
    return newF, mseL, itr
f0,mse,itr_times = iterate(120,500)
print('Frequence converge to:',f0,'Hz')
x = np.linspace(0,itr_times,itr_times+1)
plt.plot(x,mse)
plt.xlabel('Iteration times')
plt.ylabel('Mse')
plt.grid()
```

```
Frequence converge to: 131.528923318 Hz
```



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

```
In [13]: # Test
         import random

         # take a random initial point
         freq0_0 = np.random.randint(40,500)

         step = 1e-6
         freq0_1 = freq0_0 + step*np.random.randint(40,500)
         a_test = Audio(yi)
         mse0,mse_grad0 = a_test.feval(freq0_0)
         mse1,mse_grad1 = a_test.feval(freq0_1)
         dmse = mse_grad0*(freq0_1 - freq0_0)
         print('Actual mse1-mse0:',mse1-mse0)
         print('Predicted mse1-mse0:',dmse)
```

```
Actual mse1-mse0: 1.64482951992e-06
Predicted mse1-mse0: 1.64470212736e-06
```

1.6 Run the Optimizer

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

```
In [34]: 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))################ not a matrix need to use np.

    # 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.

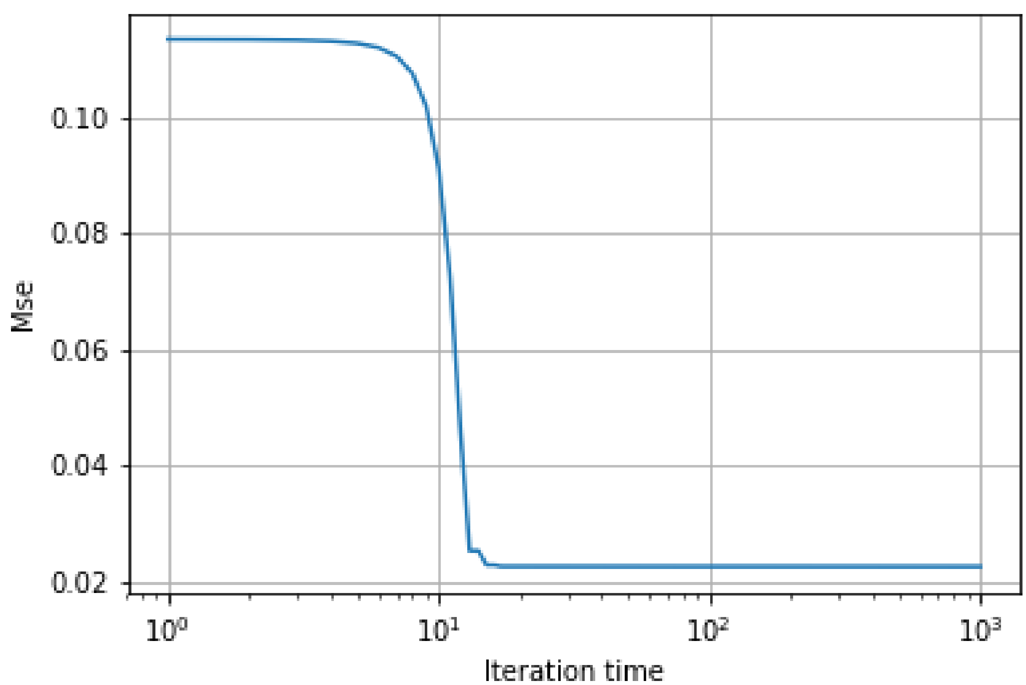
```
In [35]: au = Audio(yi)
         print(au.feval)

         opt = grad_opt_adapt(Audio(yi).feval, 130)
```

```
freq = opt[0]
midi_num = 12*math.log2(freq/440)+69
print('Final frequency estimate is:',freq)
print('The midi_num is:',midi_num)
```

```
<bound method Audio.feval of <__main__.Audio object at 0x110040ef0>>
Final frequency estimate is: 131.528923319
The midi_num is: 48.094518725781484
```

Plot the MSE as a function of the iteration.

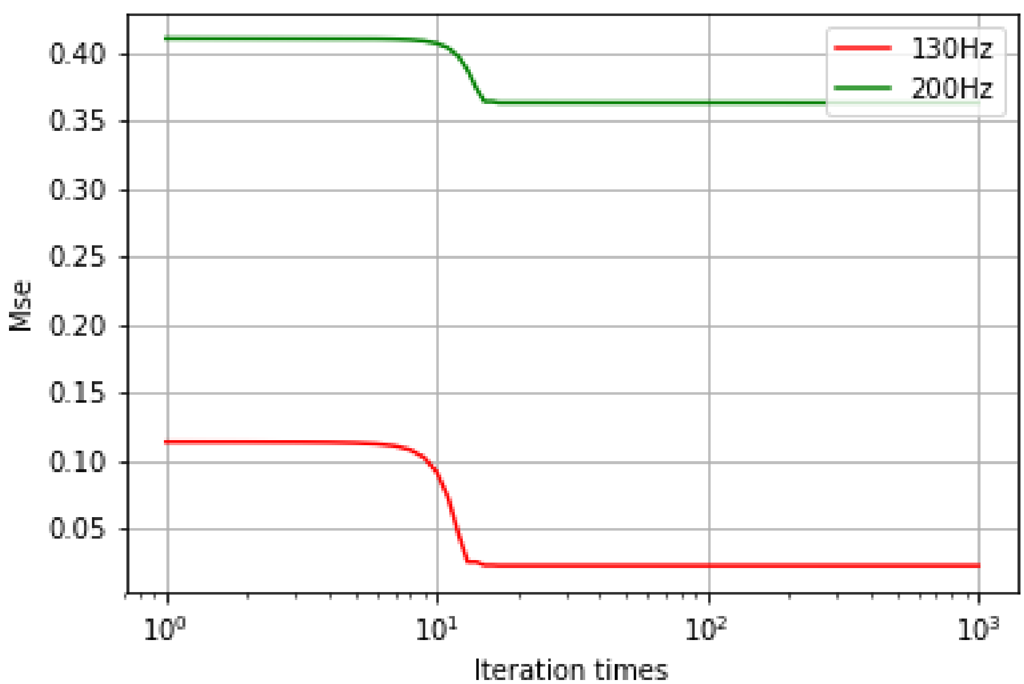
```
In [38]: mse_130 = opt[2]['f']
         n = mse_130.size
         x_130 = np.arange(n)
         plt.semilogx(x_130, mse_130)
         plt.xlabel('Iteration time')
         plt.ylabel('Mse')
         plt.grid()
```



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. You will see that that the optimizer does not obtain the minimum MSE since it gets stuck at a local minima. This is the main reason this form of pitch detection is not used -- it requires a very good initial condition.

```
In [40]: au_200 = Audio(yi)
         opt_200 = grad_opt_adapt(au_200.feval, 200)

         mse_200 = opt_200[2]['f']
         n = mse_200.size

         x_200 = np.arange(n)
         plt.semilogx(x_130, mse_130, 'r-')
         plt.semilogx(x_200, mse_200, 'g-')
         plt.xlabel('Iteration times')
         plt.ylabel('Mse')
         plt.legend(['130Hz','200Hz'],loc='upper right')
         plt.grid()

         print('Final estimated frequency:',opt_200[0])
```

```
Final estimated frequency: 197.872343473
```



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.

```
In [ ]:
```