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 (./grad_descent.ipynb), 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 (https://en.wikipedia.org/wiki/Pitch_detection_algorithm), mostly using frequency-domain techniques. But, the method here will illustrate non-linear estimation well.

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` (./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 (https://ccrma.stanford.edu/~jos/pasp/Sound_Examples.html). 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')

# TODO:  Print sample rate, number of samples and file length in seconds.
print(sr)
#print(y[3000:3050])
#print(type(sr))
#print(type(y))
print("number of samples: "+str(y.shape[0]))
print("file length in seconds: "+ str(y.shape[0]/sr))
#plt.plot(y)
```

```
44100
number of samples: 299350
file length in seconds: 6.787981859410431
```

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]:

0:00 / 0:06

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 = ...
# y = y / scale

y=y.astype(float)
scale = np.sqrt(np.mean(y**2))
#print(type(y[0].astype(float)))
print(np.mean(y**2))
ys = y / scale
print(scale)
```

```
45668243.5215
6757.828314
```

In [5]:
```
#ys=ys/scale
print(np.mean(ys**2))
```

```
1.0
```

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,
...
```

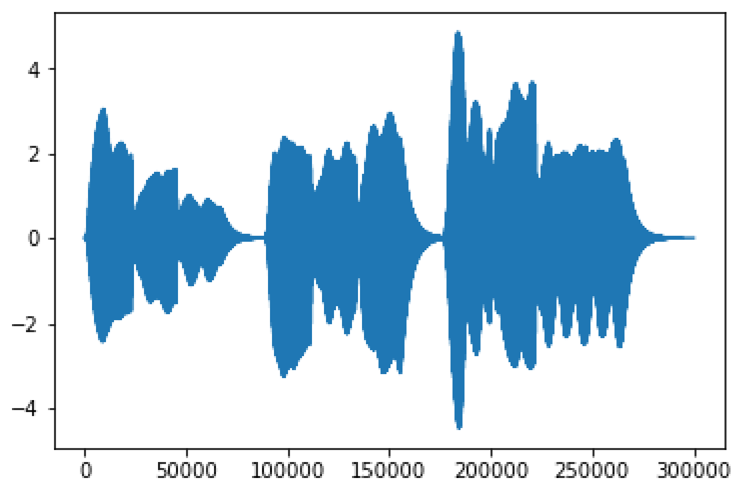
You can do this with the `reshape` command with `order=F`. Zero pad `y` if the number of samples of `y` is not divisible by `nfft`. Print the total number of frames as well as the length (in milliseconds) of each frame.

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

# TODO:
#  nframe = ...
#  yframe = ...
#np.reshape(ys,(1,int(ys.shape[0]/nfft)+1),order='F')
print("length in milliseconds: "+ str(nfft/sr*1000))
nframe = int(ys.shape[0]/nfft)+1
print("total number of frames: "+ str(nframe))
sp=nfft*nframe

ys=np.pad(ys,(0,sp-ys.shape[0]),'constant', constant_values=0)
plt.plot(ys)
yframe=ys.reshape((nfft,nframe),order='F')

#yfram=np.reshape(order='F')
#for i in range(nfram):
#    yfram[:,i]=ys[i*nfft:(i+1)*nfft]
```

```
length in milliseconds: 23.219954648526077
total number of frames: 293
```



In [7]:

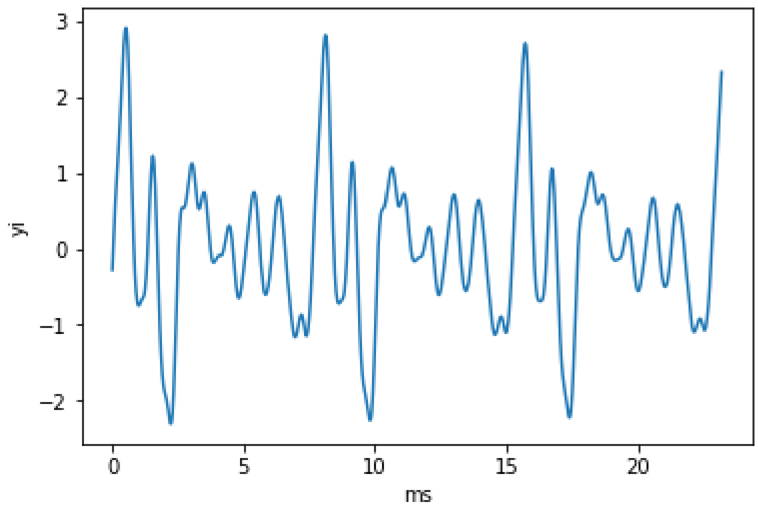
```
print(yframe)
print(ys)
```

```
[[ 0.         -0.00044393  0.13599043 ...,  0.          0.          0.        ]
 [ 0.          0.00147977  0.14575688 ...,  0.          0.          0.        ]
 [ 0.          0.00340346  0.15389559 ...,  0.          0.          0.        ]
 ...,
 [-0.00577108  0.0989963  -0.03270281 ...,  0.          0.          0.        ]
 [-0.00384739  0.11246216 -0.09352117 ...,  0.          0.          0.        ]
 [-0.00192369  0.12474422 -0.13332686 ...,  0.          0.          0.
 ]]
[ 0.  0.  0. ...,  0.  0.  0.]
```

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 [8]:
```
# Get samples from frame 10
i0 = 10
yi = yframe[:,i0]

# TODO:  Plot yi vs. time (in ms)
#t = np.linspace(0, nfft/sr*1000, num=num, endpoint=endpoint)
t = np.arange(0,nfft/sr*1000,1/sr*1000)
plt.plot(t,yi)
plt.xlabel('ms')
plt.ylabel('yi')
```

Out[8]: <matplotlib.text.Text at 0x7f8e240596d8>



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.

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.

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`.

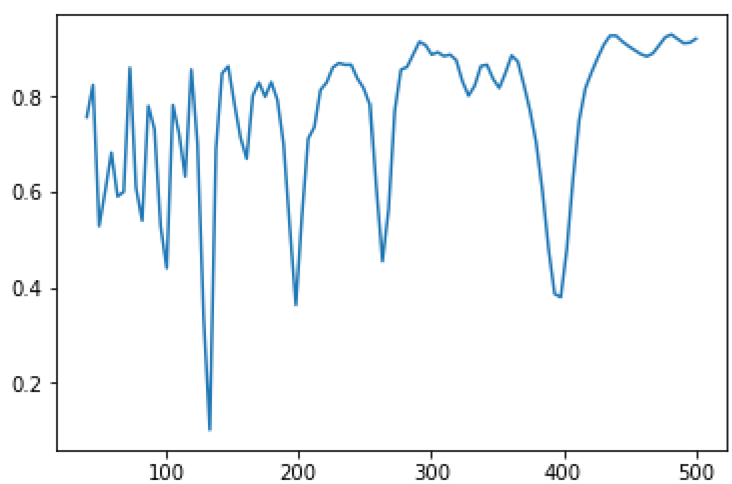
In [9]:
```
class AudioFitFn(object):
    def __init__(self,yi,sr=44100,nterms=8):
        """
        A class for fitting

        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, 
        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

        x1=np.zeros((self.yi.shape[0],self.nterms+self.nterms))
#         x2=[]
        for i in range(0,self.nterms):
            for k in range(self.yi.shape[0]):
                x1[k][i]=(np.cos(2*np.pi*k*freq0*(i+1)/sr))
        for i in range(self.nterms,2*self.nterms):
            for k in range(self.yi.shape[0]):
                x1[k][i]=(np.sin(2*np.pi*k*freq0*(i+1-self.nterms)/sr))

        x1=x1.T
#         print(x1)


        self.A = np.vstack(( np.ones(self.yi.shape[0]) , x1 )).T
#         print(self.A)
#         print(self.yi.shape)
        betahat,_,_,_=np.linalg.lstsq(self.A,self.yi)
#         print(betahat)
#         print(betahat.shape)
        self.yhati=self.A.dot(betahat)
#         print(self.yhati.shape)
```

```
        # mse1 = ...
        mse1 = np.mean((self.yhati-self.yi)**2)
#        print(mse1)
#        print(np.argmin(betahat))


        # 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 [10]:
```
# TODO
audio_fn=AudioFitFn(yi)
test = np.linspace(40, 500, 100)
res = np.zeros(100)
for i in range(len(test)):
    res[i],_=audio_fn.feval(test[i])
plt.plot(test,res)
```

Out[10]:
```
[<matplotlib.lines.Line2D at 0x7f8e1c60bbe0>]
```

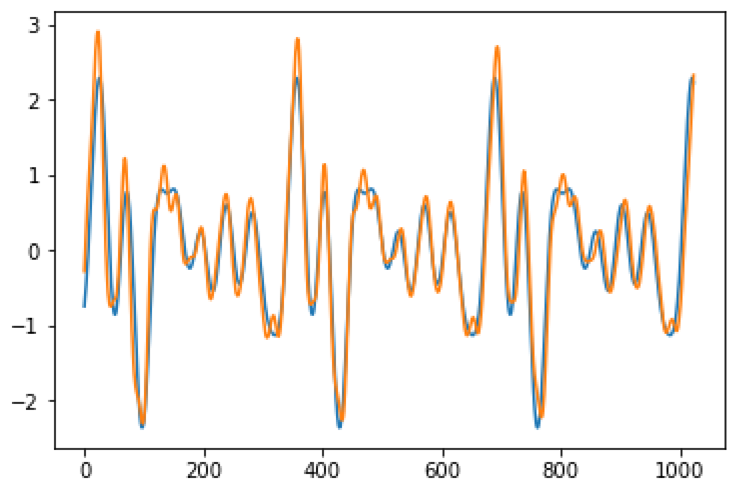


Print the value of `freq0` that achieves the minimum `mse1`. Also, plot the estimated function `audio_fn.yhati` for that along with the original samples `yi`.

In [11]:
```
# TODO
freq0=test[np.argmin(res)]
res0,_=audio_fn.feval(freq0)
print("freq0 = ", freq0, " mse1= ",np.min(res0))

#print(audio_fn.A[1].shape)
#plt.plot(audio_fn.A[:,9])
plt.plot(audio_fn.yhati,label="yhati")
plt.plot(yi,label="yi")
#plt.legend('yhati', loc='upper right')
```

```
freq0 =  132.929292929  mse1=  0.100453382721
```

Out[11]:
```
[<matplotlib.lines.Line2D at 0x7f8e1c64b358>]
```



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]:
```
# TODO
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, t
        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

        x1=np.zeros((self.yi.shape[0],self.nterms+self.nterms))
#        x2=[]
        for i in range(0,self.nterms):
            for k in range(self.yi.shape[0]):
                x1[k][i]=(np.cos(2*np.pi*k*freq0*(i+1)/sr))
        for i in range(self.nterms,2*self.nterms):
            for k in range(self.yi.shape[0]):
                x1[k][i]=(np.sin(2*np.pi*k*freq0*(i+1-self.nterms)/sr))

        x1=x1.T
#        print(x1)


        self.A = np.vstack(( np.ones(self.yi.shape[0]) , x1 )).T
#        print(self.A)
#        print(self.yi.shape)
        betahat,_,_,_=np.linalg.lstsq(self.A,self.yi)
#        print(betahat)
#        print(betahat.shape)
        self.yhati=self.A.dot(betahat)
#        print(self.yhati.shape)
```

```
        # mse1 = ...
        mse1 = np.mean((self.yhati-self.yi)**2)
#        print(mse1)
#        print(np.argmin(betahat))


        # Compute the gradient wrt to freq0



        x2=np.zeros((self.yi.shape[0],self.nterms+self.nterms))
        for i in range(0,self.nterms):
            for k in range(self.yi.shape[0]):
                x2[k][i]=(2*np.pi*k*(i+1)/sr*(-np.sin(2*np.pi*k*freq0*(i+1)/sr)))
        for i in range(self.nterms,2*self.nterms):
            for k in range(self.yi.shape[0]):
                x2[k][i]=(2*np.pi*k*(i+1-self.nterms)/sr*(np.cos(2*np.pi*k*freq0*

        x2=x2.T
#        print(x1)


        self.A1 = np.vstack(( np.ones(self.yi.shape[0]) , x2 )).T
        df_df=self.A1.dot(betahat)

        mse1_grad = np.mean(2*(self.yhati-self.yi)*df_df)
        return mse1, mse1_grad
```

In [13]:

```
audio_fn=AudioFitFn(yi)
_,res1=audio_fn.feval(132)
#print(res1.shape)
#plt.plot(audio_fn.A1[:,16])
```

In [14]:
```
###### Take a random initial point
#p = X.shape[1]+1

freq0_0=freq0

# Perturb the point
step = 1e-6
freq0_1 = freq0 + step
#*np.random.randn(p)

# Measure the function and gradient at w0 and w1
f0, fgrad0 = audio_fn.feval(freq0_0)
f1, fgrad1 = audio_fn.feval(freq0_1)

# Predict the amount the function should have changed based on the gradient
df_est = fgrad0*(freq0_1-freq0_0)

# Print the two values to see if they are close
print("Actual f1-f0    = %12.4e" % (f1-f0))
print("estimate gradient = %12.4e"% (df_est))
#plt.plot(df_est)
#plt.plot(fgrad0)
```

```
Actual f1-f0    =   1.0457e-07
estimate gradient =   1.0457e-07
```

Run the Optimizer

We cut and paste the optimizer from the gradient descent demo (./grad_descent.ipynb).

In [15]:
```
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 = 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 (https://newt.phys.unsw.edu.au/jw/notes.html) 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 [16]:
```
# TODO
nit = 1000
lr_init=1e-3
f0_init=130
feval = audio_fn.feval

w0,f0,hist = grad_opt_adapt(feval,f0_init,lr_init=lr_init)
```

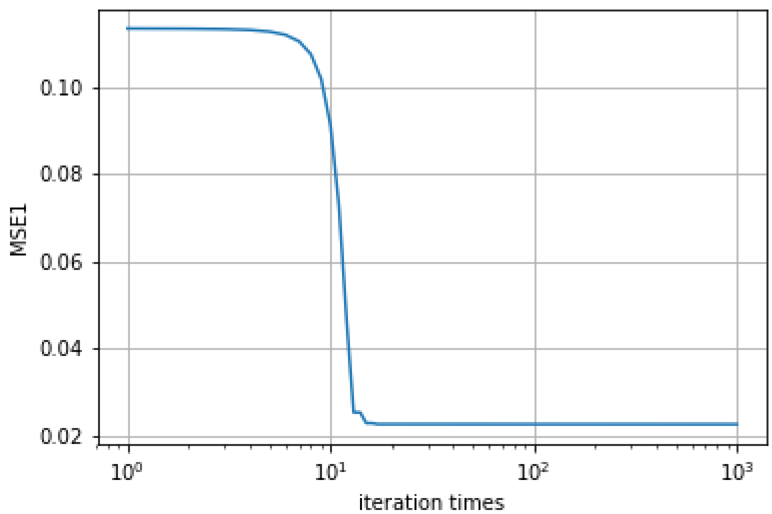
In [17]:
```
midi_num = 12*np.log2(w0/440) + 69
print(midi_num)
```

```
48.0945187258
```

Plot the MSE as a function of the iteration.

In [18]:
```
#TODO
print(w0)
print(f0)
nit = 1000
t = np.arange(nit)
plt.semilogx(t, hist['f'])
plt.xlabel('iteration times')
plt.ylabel('MSE1')
plt.grid()
```

```
131.528923319
0.022564366787
```



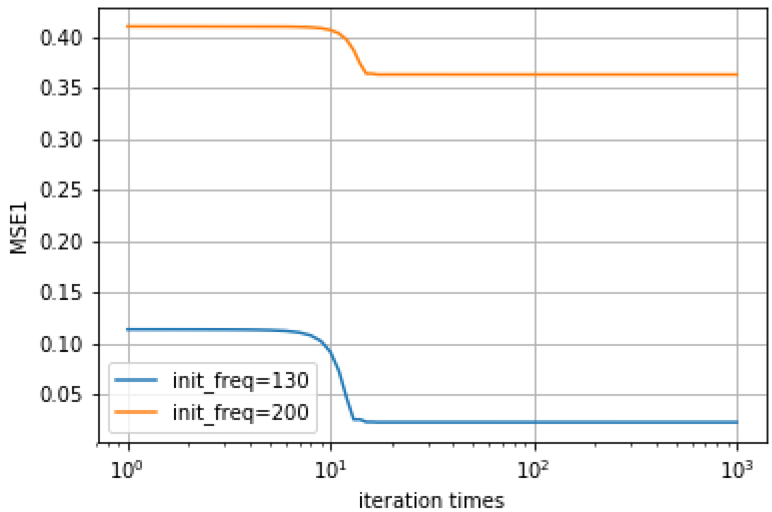
Now, repeat with an initial frequency of 200 Hz. Print the final estimated frequency. Also plot the MSE per iteration on the same graph as the MSE per iteration with the initial condition = 130 Hz. You will see that that the optimizer does not obtain the minimum MSE since it gets stuck at a local minima. This is the main reason this form of pitch detection is not used -- it requires a very good initial condition.

In [19]:
```
# TODO

f0_init=200

w1,f1,hist1 = grad_opt_adapt(feval,f0_init,lr_init=lr_init)
```

In [20]:
```
print(w0)
print(f0)
plt.semilogx(t, hist['f'])
plt.semilogx(t, hist1['f'])
plt.grid()
plt.legend(['init_freq=130', 'init_freq=200'])
plt.xlabel('iteration times')
plt.ylabel('MSE1')
```

```
131.528923319
0.022564366787
```

Out[20]: `<matplotlib.text.Text at 0x7f8e2490ca90>`



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 (https://librosa.github.io/librosa/).

In [ ]:
```

```