Main.py:

Main.py > ...

```python
from imports import *
from Freq_detec import *
from Pitch_change import *
from Epoch_detec import *
from tests import *
from Voice_sep import *


Frame_size = 2048

#reading input audios
F_s, audio_song  = read("test_song_complete.wav")
F_s, audio_user_ = read("test_song_user1.wav")
F_s, audio_voice = read("test_song_clear_voice.wav")

print("Read file complete, F_s is", F_s)

#adjusting the sampling frequency by double sampling the similated user input
audio_user = []
for i in range(len(audio_user_)):
    audio_user.append(audio_user_[i])


audio_user_out = np.array(audio_user,dtype=np.int16)
scipy.io.wavfile.write('output_stage0_user.wav',F_s,audio_user_out)


#stage 1, music and voice seperation
audio_music_sep = music_voice_sep(F_s,audio_song,Frame_size)
audio_voice_sep = audio_song - audio_music_sep

audio_data_v = []
audio_data_m = []
audio_data_u = []



if(len(audio_music_sep.shape) > 1):
    for i in range(len(audio_music_sep)):
        audio_data_v.append((audio_voice_sep[i][0] + audio_voice_sep[i][1] )/2)
        audio_data_m.append((audio_music_sep[i][0] + audio_music_sep[i][1] )/2)


if(len(audio_user_out.shape) > 1):
    for i in range(len(audio_user_out)):
        audio_data_u.append((audio_user_out[i][0] + audio_user_out[i][1] )/2)
print ("Song data processing complete")


audio_data_u = np.array(audio_data_u,dtype=np.int16)

```

```python
audio_data_v_out = np.array(audio_data_v,dtype=np.int16)
scipy.io.wavfile.write('output_stage1_voice.wav',F_s,audio_data_v_out)

audio_data_m_out = np.array(audio_data_m,dtype=np.int16)
scipy.io.wavfile.write('output_stage1_music.wav',F_s,audio_data_m_out)
#stage 1 complete


#stage 2 tempral adjustment to user inputs
numFrames_org = int(len(audio_data_v)/Frame_size)
numFrames_usr = int(len(audio_data_u)/Frame_size)

fraction_co = find_closest_fraction(numFrames_org/numFrames_usr)
#print ('debug the fractions are', fraction_co[0], fraction_co[1])
#print (numFrames_org,numFrames_usr)

audio_user_adj = signal.resample_poly(audio_data_u.astype('float'), fraction_co[0], fraction_co[1])
audio_user_adj_out = np.array(audio_user_adj,dtype=np.int16)
scipy.io.wavfile.write('output_stage2_adjusted.wav',F_s,audio_user_adj_out)

#stage 2 complete


#stage 3 pitch correction to user inputs
notes = []
audio_user_syth =[]

for i in range(numFrames_org):
    frame_ = audio_data_v[i*Frame_size:(i+1)*Frame_size]
    frame = np.array(frame_,dtype=np.int16)
    freq = freq_detect(frame.astype(float),F_s)
    notes.append(freq)
print ("notes generated")



def find_closest_in_vector(vector, value, min_idx, max_idx):

    # Ensure the indices are within the bounds of the vector
    min_idx = max(0, min_idx)
    max_idx = min(len(vector), max_idx)

    # Initialize the minimum difference found and the index of that difference
    min_diff = float(np.inf)
    closest_idx = min_idx

```

```python
    # Iterate through the specified range
    for i in range(min_idx, max_idx):
        diff = abs(vector[i] - value)
        if diff < min_diff:
            min_diff = diff
            closest_idx = i

    return closest_idx

def lab5_pitch_shift(buffer_in, F_S, FREQ_NEW, FRAME_SIZE):
    period_len = F_s/freq_detect(buffer_in,F_s)
    freq = F_S / period_len
    print(f"Frequency target: {FREQ_NEW}")
    if period_len > 0:
        print(f"Frequency detected: {freq}")

        epoch_locations = findEpochLocations(buffer_in, period_len)

        new_epoch_spacing = F_S / FREQ_NEW
        new_epoch_idx = 0
        epoch_mark = 0

        buffer_out = np.zeros_like(buffer_in)

        while (new_epoch_idx < FRAME_SIZE * 2):
            itr = find_closest_in_vector(epoch_locations, new_epoch_idx, epoch_mark, len(epoch_locations
            epoch_mark = itr

            p0 = abs(epoch_locations[itr - 1] - epoch_locations[itr + 1]) // 2

            # Window generation

            window = np.hanning(p0*2)
            windowed_sample = []
            # Window application
            for z in range(2 * int(p0)):
                windowed_sample.append (window[z] * buffer_in[int(epoch_locations[itr] )- int(p0) + z])

            # Sample localization
            sample_addition(buffer_out, windowed_sample, int(new_epoch_idx - p0))
            new_epoch_idx += new_epoch_spacing

        # Final bookkeeping
        new_epoch_idx -= FRAME_SIZE
        if (new_epoch_idx < FRAME_SIZE):
            new_epoch_idx = FRAME_SIZE

    return buffer_out
```

```python
def butter_lowpass(cutoff_freq, sampling_freq, order=5):
    nyquist_freq = 0.5 * sampling_freq
    normal_cutoff = cutoff_freq / nyquist_freq
    b, a = signal.butter(order, normal_cutoff, btype='low', analog=False)
    return b, a

def apply_filter(data, cutoff_freq, sampling_freq, order=5):
    b, a = butter_lowpass(cutoff_freq, sampling_freq, order=order)
    filtered_data = signal.lfilter(b, a, data)
    return filtered_data





#after resampling, there is still a slight mismatch in length from the target and user, here we select
numFrames_usr = int(len(audio_user_adj)/Frame_size)

if (numFrames_org<numFrames_usr):
    numFrames = numFrames_org
else:
    numFrames = numFrames_usr

buffer = np.zeros(3*Frame_size)
epochs1 = [0]
epochs2 = []
epochs3 = []

for k in range(numFrames-1):

    frame = audio_user_adj[k*Frame_size:(k+1)*Frame_size]
    buffer[Frame_size*2:Frame_size*3] = frame
    freq = freq_detect(buffer.astype(float),F_s)
    epochs1 = findEpochLocations(frame.astype(float), F_s/freq)
    target_freq = notes[k]
    epochs = epochs1 + epochs2 + epochs3
    epochs.sort()
    #print(epochs)
    #audio_out_ = lab5_pitch_shift(buffer, F_s, target_freq, Frame_size)
    #audio_out_ = pitch_synth (epochs,F_s, buffer,target_freq)
    audio_out_ = pitch_synth (epochs1,F_s, frame,target_freq)
    for j in range (Frame_size):
        #audio_user_syth.append(audio_out_[Frame_size*2+j])
        audio_user_syth.append(audio_out_[j])

    #print(buffer[Frame_size:Frame_size*2])
    buffer[Frame_size:Frame_size*2] = buffer[Frame_size*2:Frame_size*3]
    buffer[0 : Frame_size] = buffer[Frame_size:Frame_size*2]
    epochs3 = [x + Frame_size for x in epochs2]
    epochs2 = [x + Frame_size for x in epochs1]
```

```python
    #print(epochs)
    #audio_out_ = lab5_pitch_shift(buffer, F_s, target_freq, Frame_size)
    #audio_out_ = pitch_synth (epochs,F_s, buffer,target_freq)
    audio_out_ = pitch_synth (epochs1,F_s, frame,target_freq)
    for j in range (Frame_size):
        #audio_user_syth.append(audio_out_[Frame_size*2+j])
        audio_user_syth.append(audio_out_[j])

    #print(buffer[Frame_size:Frame_size*2])
    buffer[Frame_size:Frame_size*2] = buffer[Frame_size*2:Frame_size*3]
    buffer[0 : Frame_size] = buffer[Frame_size:Frame_size*2]
    epochs3 = [x + Frame_size for x in epochs2]
    epochs2 = [x + Frame_size for x in epochs1]


apply_filter(audio_user_syth, 5000, F_s, order=5)


audio_user_adj_syth_out = np.array(audio_user_syth,dtype=np.int16)
scipy.io.wavfile.write('output_stage3_synthesized.wav',F_s,audio_user_adj_syth_out)


data_fd = np.fft.fft(audio_user_adj_syth_out)
freq_i = np.fft.fftfreq(len(audio_user_adj_syth_out),d=1/F_s)



audio_output = sample_addition(audio_data_m_out,audio_user_adj_syth_out,0)

#audio_output_final = np.array(audio_output,dtype=np.int16)
scipy.io.wavfile.write('output_final_synthesized.wav',F_s,audio_data_m_out)



quit()


audio_data_m_o = np.array(audio_data_m,dtype=np.int16)
scipy.io.wavfile.write('music.wav',F_s,audio_data_m_o)

Voice_output = np.array(Voice_full,dtype=np.int16)
scipy.io.wavfile.write('voice.wav',F_s,Voice_output)

audio_output = np.array(audio_out,dtype=np.int16)
    #audio_output = audio_output.astype(int16)
spwav.write('audio_out.wav',F_s,audio_output)
print ('debug audio_out shape', audio_output.shape)

print ('finished')
```

Imports.py

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.io.wavfile as spwav
import sys
import scipy
import math


from scipy.io.wavfile import read, write
from numpy.fft import fft, ifft
from scipy import signal


def find_closest_fraction(number):
    closest_fraction = None
    min_difference = float('inf')

    for numerator in range(0, 11):
        for denominator in range(1, 11):
            fraction = numerator / denominator
            difference = abs(number - fraction)
            if difference < min_difference:
                min_difference = difference
                closest_fraction = (numerator, denominator)

    return closest_fraction




def get_output(up_,down_,data_i):

    up_ratio = up_
    down_ratio = down_

    output = signal.resample_poly(data_i, up_ratio, down_ratio)
    return output


def generate_sine_wave(frequency, num_samples, sampling_freq):
    t = np.linspace(0, (num_samples-1) / sampling_freq, num_samples)  # Ge
    x = np.sin(2 * np.pi * frequency * t)  # Calculate sine values
    return x


def generate_audio(notes, sampling_freq, duration):
    audio = []
    for freq in notes:
        t = np.linspace(0, duration, int(duration * sampling_freq), endpoi
        note = np.sin(2 * np.pi * freq * t)*1000
        audio.extend(note)
    audio = np.array(audio)
```

```python
def generate_audio(notes, sampling_freq, duration):
    audio = []
    for freq in notes:
        t = np.linspace(0, duration, int(duration * sampling_freq), endpoi
        note = np.sin(2 * np.pi * freq * t)*1000
        audio.extend(note)
    audio = np.array(audio)
    # Scale the audio to be between -1 and 1
    audio /= np.max(np.abs(audio), axis=0)
    return audio
```

Pitch_change.py

Pitch_change.py > find_map

```python
from imports import *

def pitch_synth (epoch_marks_orig,F_s, audio_data,F_new):
    N = len(audio_data)
    new_epoch_spacing = int(F_s//F_new)
    audio_out = np.zeros(N)
    epoch_mark = 0
    itr = 0
    epoch_marks_orig = np.insert(epoch_marks_orig,0,0)
    for i in range(0, N, new_epoch_spacing):
        itr = find_map(i,epoch_marks_orig,epoch_mark,epoch_marks_orig)
        epoch_mark = itr
        epoch_marks_orig = np.append(epoch_marks_orig,len(audio_data)-1000)
        p0 = int(abs((epoch_marks_orig[itr-1])-(epoch_marks_orig[itr+1]))/2)
        epoch_marks_orig = np.delete(epoch_marks_orig,len(epoch_marks_orig)-1)
        window = np.hanning(p0*2)
        left_idx = int(epoch_marks_orig[itr]-p0)
        right_idx = int(epoch_marks_orig[itr]+p0)
        windowed_sample = window_apply(audio_data[left_idx:right_idx] ,window)
        #print(i-p0)
        sample_addition(audio_out,windowed_sample,i-p0)
    return audio_out



def find_map (new_epoch, epoc_org, epoch_mark,epoch_marks_orig):
    itr = epoch_mark
    delta_min = 0
    for k in range (epoch_mark,len(epoc_org)):

        if (k == epoch_mark):
            delta_min = abs(new_epoch - epoch_marks_orig[k])

        else:
            delta_new = abs(new_epoch - epoch_marks_orig[k])
            if (delta_new <= delta_min):
                delta_min = delta_new
                itr = k
            else:
                break
    return itr
```

```python
def window_apply (a,b):
    output = []

    for j in range(len(a)):
        result = a[j]*b[j]
        output.append(result)


    return output

def sample_addition(a,b,start):
    for x in range(len(b)-1):
        if (start+x >= len(a)):
            break
        if (start+x >=0):
            a[start+x]+=b[x]
    return
```

Epoch_detec,py

```python
from imports import *

def findEpochLocations(audio_data,periodlen):
    epoch_location =[]
    min_idx = int(0)
    max_idx = int(0)

    #print ('debug audio_data is', audio_data[:20])
    #print ('debug periodlen is', periodlen)
    largestPeak = findMaxArrayIdx(audio_data,0,len(audio_data))
    epoch_location.append(largestPeak)
    #print(largestPeak)
    epochCandidateIdx = epoch_location[0] + periodlen

    while (epochCandidateIdx < len(audio_data)):
        epoch_location.append(epochCandidateIdx)
        epochCandidateIdx += periodlen


    epochCandidateIdx = epoch_location[0] - periodlen
    while (epochCandidateIdx > 0):
        epoch_location.append(epochCandidateIdx)
        epochCandidateIdx -= periodlen


    epoch_location.sort()
    for i in range (len(epoch_location)-1):
        min_idx = int(epoch_location[i] - periodlen/3)
        max_idx = int(epoch_location[i] + periodlen/3)
        peakoffset = findMaxArrayIdx(audio_data,min_idx,max_idx)
        offset = -(epoch_location[i] - peakoffset)
        epoch_location[i] = peakoffset
        if (i < len(epoch_location)-1):
            epoch_location[i+1] += offset
        if (i>0):
            delta = epoch_location[i] - epoch_location[i-1]
            if (delta < periodlen/2.5):
                epoch_location[i] = 9999999

    epoch_location.sort()
    epochs_clean_up(epoch_location,len(audio_data))

    return epoch_location

```

```python


def findMaxArrayIdx(array, min_idx,max_idx):
    ret_idx = min_idx
    for i in range(min_idx,max_idx):
        if (array[i] > array[ret_idx]):
            ret_idx = i
    return ret_idx


def epochs_clean_up (array,frame_length):
    if(array[0] < 0):
        array.pop(0)

    i = len(array)-1

    while (i>=0):
        if(array[i] >= frame_length):
            array.pop(i)
            i-=1
        else:
            break
        #print (i)
    return
```

Voice_sep.py

```python
# input x(signal), N(Number of samples in each Hamming window for STFT)
#output
import numpy as np;
import scipy
import math


def music_voice_sep(F_s,x,N):

    Music_full = np.zeros(x.shape)

    for channel in range(x.shape[1]):
        current_channel_data = x[:, channel]
        processed_channel_data = TempX(current_channel_data, F_s,N)
        _, Music = scipy.signal.istft(processed_channel_data, fs=F_s, window='hamming', nperseg=N, nover
        if len(Music) != x.shape[0]:
            Music = np.resize(Music, x.shape[0])
        Music_full[:, channel] = Music

    return Music_full




def TempX(x, F_s,N):
    # Constants
    #N = 1024  # Assuming N is defined outside, used for STFT computation

    def autoc(frame):
        fft_frame = np.fft.fft(frame)
        power_spectrum = fft_frame * np.conj(fft_frame)
        autoc = np.abs(np.fft.ifft(power_spectrum))
        return autoc

    # Perform STFT
    f, t, X = scipy.signal.stft(x, fs=F_s, window='hamming', nperseg=N, noverlap=N//2)

    # Compute magnitude spectrogram and square it
    m = len(t)
    n = N / 2 + 1
    V = np.abs(X)
    V_squared = V**2
    B = np.zeros_like(V_squared)

    # Autocorrelation row by row
    num_rows = V_squared.shape[0]
    for i in range(num_rows):
        B[i] = autoc(V_squared[i])
    '''
```

```
    # Calculate bear spectrum
    b = np.sum(B, axis=0) / n
    b = b / b[0]

    # Valid part of b
    b_valid = b[0:3*len(b)//4]
    l = len(b_valid)

    # Initialize J array
    J = np.zeros(l // 3)

    # Calculate p using the described algorithm
    for j in range(1, l // 3 + 1):
        if (j >=2):
            delta1 = 2
        else:
            delta1 = j
        delta2 = math.floor(3 * j / 4)
        I = 0
        for i in range(j, l, j):
            h1 = np.argmax(b[i-delta1: i+delta1+1]) + max(0, i-delta1)
            h2 = np.argmax(b[i-delta2:  i+delta2+1]) + max(0, i-delta2)
            sum_ = np.sum(b[i-delta2: i+delta2+1])
            if h1 == h2:
                I += b[h1] - sum_ / ((2 * delta2) + 1)
        J[j-1] = I / math.floor(l / j)

    # Repeating period
    p = np.argmax(J) + 1

    # Repeating segment model
    r = V.shape[1] // p
    S = np.zeros((int(n), int(p * r)))

    for i in range(int(n)):
        for l in range(p):
            values_at_l = [V[i, l + k * p] for k in range(r-1)]
            S[i, l] = np.median(values_at_l)

    # Compute repeating spectrogram model
    W = np.zeros(V.shape)
    for i in range(int(n)):
        for l in range(p):
            for k in range(r):
                idx = l + k * p
                if idx < V.shape[1]:
                    W[i, idx] = np.minimum(S[i, l], V[i, idx])

    #!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
    #!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    # Calculating soft mask, check how they do it I think some thing is wrong here
    M = np.zeros((int(n), m))
    for i in range(int(n-1)):
        for j in range(m-1):
            M[i, j] = W[i, j] / V[i, j] if V[i, j] != 0 else 0

    # Ensure mask values are within [0, 1], I deleted this part to see the difference

    # Apply mask to STFT
    TempX = np.multiply(M, X)

    return TempX
```

Freq_detec.py

```
from imports import *



def getEnergy(frame):
    E = int(0)
    #print (type(threshold))
    for i in range (len(frame)):
        #print (E)
        E = E+(frame[i]*frame[i])
    #print (E)
    return int(E)

def cycle (a,b):
    if (a<0):
        return a+b
    else:
        return a


def get_autocor(frame,E):
    R = []
    for i in range (len(frame)):
        Rl = 0
        for k in range (len(frame)):
            itr = cycle (k-i,len(frame))
            Rl += frame[k] * frame[itr]
        R.append(Rl/E)
    return R

def peak_detection(frame):
    peaks = []
    N = len(frame)
    a = 25
    for i in range(a,N-a):
        if frame[i]>frame[i-a]:
            if frame[i]>=frame[i+a]:
                position = i
                peaks.append(position)
    return peaks

def get_autocor_(frame,E):
    N = np.fft.fft(frame)
    N_ = np.conjugate(N)
    output = np.fft.ifft(N*N_)/E
    return output

```

```python

def peak_select(st_pt,sp_pt,peaks):
    for i in range (len(peaks)):
        if (peaks[i] < st_pt):
            if(peaks[i]>sp_pt):
                return peaks[i]
    #print (peaks)
    #print ("Fs =")
    return 60



def freq_detect(frame, Fs):
    FRAME_SIZE = len(frame)
    threshold = (1800000000/2048)*FRAME_SIZE

    freq = 60

    E = getEnergy(frame)

    #print( 'debug E type is', type(E))

    #print('debug threshold type is', type(threshold))
    if (E<threshold):
        return freq

    R = get_autocor_(frame,E)

    st_pt =  int(Fs/60)
    sp_pt =  int(Fs/270)

    peaks = peak_detection(R)
    freq = Fs/peak_select(st_pt,sp_pt,peaks)

    return freq

```