1. **说5个数字0,1,2,3,4，用短时自相关法计算基音，使用10阶线性预测LPC，计算预测增益**

**解：**

1. **技术思路、主要计算公式**
2. **录音。转为单声道，采样频率8khz，wav**

录音格式为mp3，使用python的pydub库的audiosegment，转化为wav。

指定参数8000hz，1 channel

1. **求基音**

格式转化。

使用f = wave.open以bytes格式读取wav文件，然后使用str\_data = f.readframes(nframes)，将采样点数据从bytes转化为signed int。方便后续处理。

设置阈值±2000，低于阈值的样点认为是噪音，过滤。

上一步已经将样点数据转化为signed int，直接循环，从0-最后一个样点，在±2000间的样点值设为0.

分帧。一帧20ms，20ms\*8000hz=160样点。帧间无重叠。Hamming加窗。预加重。

先计算得出一帧160样点，然后计算原始文件总共分为的帧数frameNum。

然后每160个样点提取一帧，乘以预加重系数，乘以hamming窗函数。

frameNum= int(math.ceil(wlen / step)) #帧数

frameData = np.zeros((frameSize, frameNum)) #frameNum列，frameSize行

hamwin = np.hamming(frameSize)

for i in range(frameNum):

singleFrame = wavData[np.arange(i \* step, min(i \* step + frameSize, wlen))] #取一帧长的数据

singleFrame = np.append(singleFrame[0], singleFrame[:-1] - coeff \* singleFrame[1:]) # 预加重 不含最后一位-不含第一位

frameData[:singleFrame.size, i] = singleFrame

frameData[:, i] = hamwin \* frameData[:, i] # 加窗 第i列为第i帧数据

计算短时自相关函数R(K) ,k在20-100之间，使得R(K)最大的K为基音频率pitch。

IMG_256

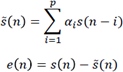
双循环，外层循环k，内层循环每一帧。

对每一帧，按照公式计算短时自相关函数，然后求和。

使得各帧相加，总的自相关函数值最大的k为pitch。

通过后续计算已知读3 的时间为5.8-6.2s，取5.9-6.1,295-305帧做短时能量计算

1. **做10阶线性预测LPC，求预测增益E0/EP**



ai为加权系数，p为阶数。等式左侧s(n)为预测值，右侧s(n)为真实值。E(n)为预测误差。

先对未分帧的数据进行线性预测，预测方法为对于每个采样点i，根据上述公式，根据i-10 -> i-1 10个真实样点预测。得到预测的数据，然后分帧加窗。

此时有了真实的数据分帧加窗后的frameDta和预测信号分帧加窗后的数据LPCData。

计算出每一帧的短时能量，计算方法为每帧样点值平方和。

然后采用原始能量E0/预测能量EP的方法计算预测增益。E0/EP越接近1预测效果越好。

也可以采用先用真实的数据减去预测的数据，得到残差信号，然后对残差信号分帧加窗，再用原始能量E0/残差能量EP，此时E0/EP越大越好。

计算读取的0-4四个数字各选取5帧计算预测增益，然后从296帧开始连续取10帧计算预测增益，然后计算总的原始能量与总的预测能量的预测增益。

1. **核心代码**

import pydub

import pylab as plt

import numpy as np

from pydub import AudioSegment

import wave

import math

import os

import librosa

from scipy.io import wavfile

# mp3转wav 单声道 8000hz

# mp3内容为 人说了0,1,2,3,4 五个数，有间隔

def mp3towav():

# 读取mp3的波形数据

mp3\_version = AudioSegment.from\_file("E:\\REC20191109103531.mp3", format = 'MP3')

# 将读取的波形数据转化为wav

tp = mp3\_version.set\_frame\_rate(8000).set\_channels(1) # 设置声道和采样率

tp.export("E:\\test.wav", format='wav', codec='pcm\_s16le',bitrate="192k") # codec参数是设定16bits pcm编码器, 此参数可以省略

# 用bytes格式读取wav文件，转化为numpy数组。每个采样点为一个 signed int

def initial():

f = wave.open("E:\\test.wav",'rb') #流

#读取格式信息

#声道数, 量化位数（byte单位）, 采样频率, 采样点数

params = f.getparams()

nchannels, sampwidth, framerate, nframes = params[:4]

print("声道数 量化位数(bytes) 采样频率(HZ) 采样点数")

print(nchannels, sampwidth, framerate, nframes)

#读取波形数据

#读取声音数据，传递一个参数指定需要读取的长度（以取样点为单位）

str\_data = f.readframes(nframes)#返回的是二进制数据 type==bytes 如\x00\x00 两个两位16进制数，两个bytes，构成一个采样值。故用np.short

#print(type(str\_data))

#print(str\_data)

f.close()

#将波形数据转换成numpy 数组

#需要根据声道数和量化单位，将读取的二进制数据转换为一个可以计算的数组 . bytes to signed int

wave\_data = np.fromstring(str\_data,dtype = np.short) #1D数组 np.short. 16-bit signed integer

#print(np.min(wave\_data)) #min -12971 max 8893 采样点数据最大最小值

#print(np.max(wave\_data)) #因为是单声道，所以每个采样点对应bytes就是那个时间的数据;若是双声道,为LRLRLRLR，每个时间点的左右数据;

#time 也是一个数组，与wave\_data配对形成系列点坐标。并画图展示

times=np.arange(0,nframes)/framerate

#展示未加阈值的原始数据

print("original data 未加阈值:")

plt.figure(1)

plt.plot(times,wave\_data)

plt.ylabel("wavData")

plt.xlabel("times(s)")

my\_x\_ticks=np.arange(0,10,0.5)

my\_yticks=np.arange(-15000,10000,1000)

plt.yticks(my\_yticks)

plt.xticks(my\_x\_ticks)

plt.show()

#阈值 设定下限 ±2000 ,低于阈值认为是噪音，删除样点

len = wave\_data.size

for i in range(0,len):

if (wave\_data[i]<2000)and(wave\_data[i]>-2000):

wave\_data[i]=0

#展示加了阈值的数据

print("加了阈值(2000)过滤噪音后的原始数据")

plt.figure(1)

plt.plot(times,wave\_data)

plt.ylabel("wavData")

plt.xlabel("times(s)")

my\_x\_ticks=np.arange(0,10,0.5)

my\_yticks=np.arange(-15000,10000,1000)

plt.yticks(my\_yticks)

plt.xticks(my\_x\_ticks)

plt.show()

return wave\_data,times

#分帧，20ms一帧，20\*8000=160,160个样点一帧。 加窗 hamming加窗 预加重

def enframe(wavData, frameSize, overlap): #overlap 重叠 frameSize 一帧样点数 wavData 用样点表示的音频数据

coeff = 0.97 # 预加重系数

wlen = wavData.size

step = frameSize - overlap # 距离

frameNum= int(math.ceil(wlen / step)) #帧数

frameData = np.zeros((frameSize, frameNum)) #frameNum列，frameSize行

hamwin = np.hamming(frameSize)

for i in range(frameNum):

singleFrame = wavData[np.arange(i \* step, min(i \* step + frameSize, wlen))] #取一帧长的数据

singleFrame = np.append(singleFrame[0], singleFrame[:-1] - coeff \* singleFrame[1:]) # 预加重 不含最后一位-不含第一位

frameData[:singleFrame.size, i] = singleFrame

frameData[:, i] = hamwin \* frameData[:, i] # 加窗 第i列为第i帧数据

return frameData #frameData为按帧存的加好窗的数据

# 计算每一帧能量 短时能量=该段语音取样值的平方和

def energy(frameData):

frameNum = frameData.shape[1]

ener = np.zeros((frameNum, 1))

for i in range(frameNum):

singleframe = frameData[:, i]

ener[i] = sum(singleframe \* singleframe)

return ener

def get\_auto\_corr(k,frameData,framelen,framenum): #对每一帧求短时自相关函数,求和 k为系数。通过后续计算已知读3 的时间为5.8-6.2s，取5.9-6.1,295-305帧做短时能量计算

ret=0

Rn=0

for n in range(0,framenum): #每帧

Rn=0

#for i in range(0,framelen-k):

for n in range(295,305): #295-305帧，5.9-6.1s，读 3 的时间段

#if frameData[i,m]!=0 and frameData[i+k,m]!=0:

Rn=Rn+frameData[i,n]\*frameData[i+k,n]

ret=ret+Rn

print("k :%d, 各帧R(K)之和: %d"%(k,ret))

return ret

#LPC线性预测。p阶 输出LPCData为预测的分好帧加窗的数据

#先直接对原始未分帧数据预测，然后分帧

def LPCAuto(wavData, frameSize, overlap,p,times):

coeff = 0.97 # 预加重系数

wlen = wavData.size

step = frameSize - overlap # 距离

frameNum= int(math.ceil(wlen / step))

hamwin = np.hamming(frameSize)

LPCData = np.zeros((framelen, frameNum))

LPCwavData=np.zeros(wlen)

# LPC 预测，

#加权系数a算法：用指数，对于LPCwavData[i]，wavData[i-1]占比10^10，wavData[i-2]占比10^9，... wavData[i-10]占比10

k=1.0/((pow(10,p+1)-10)\*1.0/9)

#print(k)

for i in range(0,wlen):

cnt=0

temp=0

powera=np.zeros(p,dtype=int)

for j in range(1,p+1):

if (i-j)>=0:

cnt=cnt+1

powera[j-1]=wavData[i-j]

if cnt!=0:

for h in range(0,p):

LPCwavData[i]=powera[h]\*k\*(pow(10,p-h))+LPCwavData[i]

#print(np.min(LPCwavData))

#print(np.min(wavData))

#用图像展示原始数据和LPC预测后的数据,对比

print("原始数据与10阶LPC线性预测数据对比")

plt.figure(1)

plt.title("Compare.wav's Frames")

plt.subplot(2,1,1)

plt.plot(times, wavData, color='green')

plt.ylabel("Origin")

plt.xlabel("times(s)")

my\_x\_ticks=np.arange(0,10,0.5)

my\_yticks=np.arange(-15000,10000,3000)

plt.yticks(my\_yticks)

plt.xticks(my\_x\_ticks)

plt.subplot(2,1,2)

plt.plot(times, LPCwavData)

plt.ylabel("LPC")

plt.xlabel("times(s)")

my\_x\_ticks=np.arange(0,10,0.5)

my\_yticks=np.arange(-15000,10000,3000)

plt.yticks(my\_yticks)

plt.xticks(my\_x\_ticks)

plt.show()

#对LPC预测的数据分帧加窗

for i in range(0,framenum):

singleFrame = LPCwavData[np.arange(i \* step, min(i \* step + frameSize, wlen))]

singleFrame = np.append(singleFrame[0], singleFrame[:-1] - coeff \* singleFrame[1:]) # 预加重

LPCData[:singleFrame.size, i] = singleFrame

LPCData[:, i] = hamwin \* LPCData[:, i] # 加窗 (因为计算原始信号能量时按照加窗后计算，所以预测也按照加窗)

return LPCData

# 计算线性增益

# LPCener为 EP ,ener为E0

#输出 一帧或 几帧之和的E0/EP

#当输入的LPCener为残差信号的能量时为原始能量/残差能量 越大越好

#当LPCener为LPC预测能量时为原始能量/预测能量，越接近1越好

def cal(LPCener,ener):

print("预测增益:")

print("采用原始能量/预测能量，越接近1越好")

#对每个字取0.1s (5帧)，进行预测增益计算 E0/EP

#字 秒(s) 帧

#0 1.1-1.2 50-60

#1, 2.8-2.9, 对应帧为 2800/20=140，3000/20=150 (130-160)帧。

#2 4.2-4.3 210-220

#3 6.0-6.1 300-310

#4 7.6-7.7 380-390

print("读的数字0-4 各取5帧 预测增益")

E0=0 #原始信号能量

for i in range (55,60):

E0=E0+ener[i]

EP=0

for i in range (55,60):

EP=EP+LPCener[i]

if EP==0:

print("0's EP=0 ")

else:

print ("0's E0/EP is %f, E0: %f,EP:%f"%(E0\*1.0/EP,E0\*1.0,EP\*1.0))

E0=0 #原始信号能量

for i in range (140,145):

E0=E0+ener[i]

EP=0

for i in range (140,145):

EP=EP+LPCener[i]

if EP==0:

print("1's EP=0 ")

else:

print ("1's E0/EP is %f, E0: %f,EP:%f"%(E0\*1.0/EP,E0\*1.0,EP\*1.0))

E0=0 #原始信号能量

for i in range (210,215):

E0=E0+ener[i]

EP=0

for i in range (210,215):

EP=EP+LPCener[i]

if EP==0:

print("2's EP=0 ")

else:

print ("2's E0/EP is %f, E0: %f,EP:%f"%(E0\*1.0/EP,E0\*1.0,EP\*1.0))

E0=0 #原始信号能量

for i in range (300,305):

E0=E0+ener[i]

EP=0

for i in range (300,305):

EP=EP+LPCener[i]

if EP==0:

print("3's EP=0 ")

else:

print ("3's E0/EP is %f, E0: %f,EP:%f"%(E0\*1.0/EP,E0\*1.0,EP\*1.0))

E0=0 #原始信号能量

for i in range (380,385):

E0=E0+ener[i]

EP=0

for i in range (380,385):

EP=EP+LPCener[i]

if EP==0:

print("4's EP=0 ")

else:

print ("4's E0/EP is %f, E0: %f,EP:%f"%(E0\*1.0/EP,E0\*1.0,EP\*1.0))

#取10帧，计算各帧E0/EP

etuple=np.zeros(10)

frametuple=np.zeros(10)

print("连续取10帧，计算各帧预测增益")

for i in range(290,310):

if LPCener[i]==0:

print("frame %d'sEP is 0"%(i))

frametuple[i-296]=i

else:

print("frame %d's E0 is %f, EP is %f, E0/EP is %f\n"%(i,ener[i],LPCener[i],ener[i]/LPCener[i]))

frametuple[i-296]=i

etuple[i-296]=ener[i]/LPCener[i]

print("296-306帧预测增益")

plt.figure(1)

plt.plot(frametuple,etuple)

plt.ylabel("E0/EP")

plt.xlabel("framenum")

my\_x\_ticks=np.arange(296,306,5)

my\_yticks=np.arange(0.8,1.4,0.05)

plt.yticks(my\_yticks)

plt.xticks(my\_x\_ticks)

plt.show()

#对于整个语音数据能量，求和，计算E0/EP

EP=0

E0=0

for i in range(0,458):

E0=E0+ener[i]

EP=EP+LPCener[i]

print("整体预测增益")

print("origin sum energy E0 :%f, LPC sum energy EP :%f,E0/EP :%f"%(E0\*1.0,EP\*1.0,(E0\*1.0)/EP))

if \_\_name\_\_ == '\_\_main\_\_':

#初始化得样点数据数组wave\_data 时间数组times

wave\_data,times=initial()

framelen = 160 #1帧160样点

overlap = 0 #相邻帧重叠 0

origin=wave\_data

#分帧加窗预加重

frameData = enframe(origin, framelen,overlap)

print("-" \* 10)

#print(frameData.shape) (160,458) 458\*160=73280 原始音频样点

#计算原始数据分帧后能量

ener=energy(frameData)

framenum=frameData.shape[1] #帧数

#计算自相关函数，返回声调pitch(hz)

pitch=0;

temp=0;

ktuple=np.zeros(105)

rktuple=np.zeros(105)

for k in range(20,101):

a=get\_auto\_corr(k,frameData,framelen,framenum)

ktuple[k]=k

rktuple[k]=a

if temp<=a:

temp=a

pitch=k

print("pitch : %d hz"%(pitch)) # k ==pitch 音调 75hz

print("短时自相关函数，k，R(k)")

plt.figure(1)

plt.plot(ktuple,rktuple)

plt.ylabel("R(K)")

plt.xlabel("K")

my\_x\_ticks=np.arange(0,120,20)

my\_yticks=np.arange(-250000000,250000000,100000000)

plt.yticks(my\_yticks)

plt.xticks(my\_x\_ticks)

plt.show()

#10阶线性预测

p=10

LPCData=LPCAuto(origin,framelen,overlap,p,times)

LPCener=energy(LPCData)

#残差信号

#预测增益 采用原始能量/LPC预测能量，越接近 1 说明预测效果越好

cal(LPCener,ener)

#预测增益 采用原始能量/(LPC预测能量-预测能量)，越大说明预测效果越好

'''

diff=LPCData-frameData

diffener=energy(diff)

cal(diffener,ener)

'''

1. **运行结果**

**环境：**

主机配置: CPU :intel core i5-7300 2.50GHZ RAM :8.0GB

运行环境：win10 64位操作系统

开发环境：python3.7 anaconda Jupyter Notebook

执行时间：2019/11/9

输入：E:\\REC20191109103531.mp3

输出：E:\\test.wav 以及展示结果的图片，数据

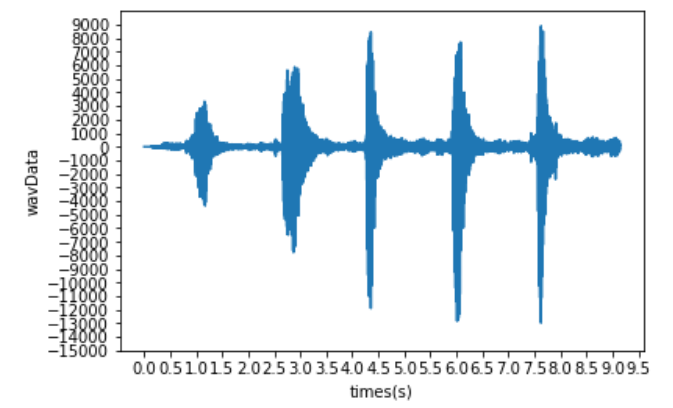
**结果：**

转化为wav格式后的文件格式：

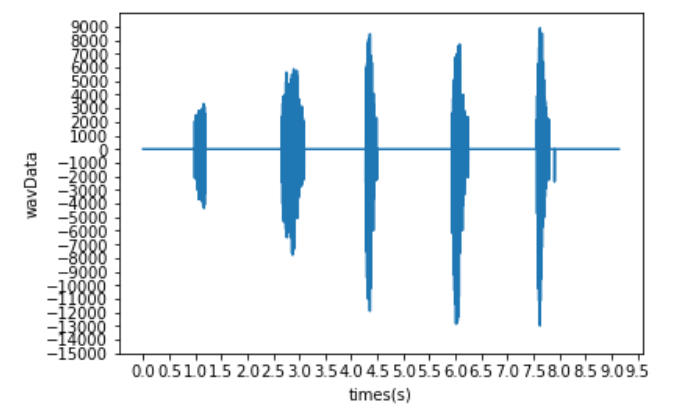
声道数 量化位数(bytes) 采样频率(HZ) 采样点数

1 2 8000 73152

采样点转化为signed int 后，未加阈值的波形



增加阈值为±2000后的波形



短时自相关函数：

k :20, 各帧R(K)之和: -232599517

k :21, 各帧R(K)之和: -81484761

k :22, 各帧R(K)之和: -105120094

k :23, 各帧R(K)之和: -100358

......

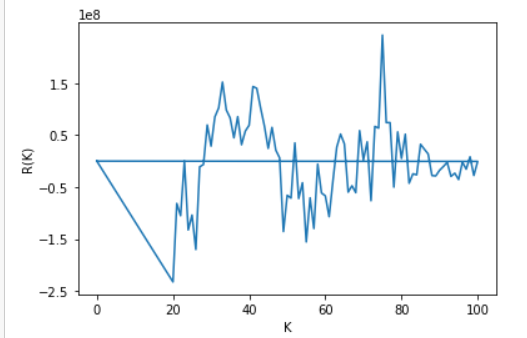
k :98, 各帧R(K)之和: 8278260

k :99, 各帧R(K)之和: -27782807

k :100, 各帧R(K)之和: -555528

折线图：y轴1e8 为步长：

峰值位于k=75处. 故pitch=75hz。



预测增益：E0/EP (原始能量/预测能量)：

越接近1越好：

读取的0-4四个数字各选取5帧计算预测增益：

0's E0/EP is 1.230508, E0: 150114926.173627,EP:121994234.264523

1's E0/EP is 1.169531, E0: 290491729.506734,EP:248383194.537866

2's E0/EP is 1.176957, E0: 133242076.009148,EP:113208996.335206

3's E0/EP is 1.169590, E0: 777897393.244453,EP:665102391.744366

4's E0/EP is 1.118544, E0: 439886259.122752,EP:393266826.524070

290帧开始连续取10帧计算预测增益：

frame 296's E0 is 35138.498979, EP is 29036.864055, E0/EP is 1.210134

frame 297's E0 is 50344103.676155, EP is 43180325.546875, E0/EP is 1.165904

frame 298's E0 is 138640533.695729, EP is 110198599.269078, E0/EP is 1.258097

frame 299's E0 is 185276238.601114, EP is 153114565.537040, E0/EP is 1.210050

frame 300's E0 is 236878113.921633, EP is 194573495.525355, E0/EP is 1.217422

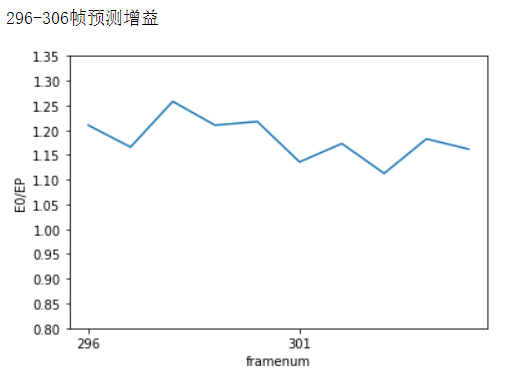
frame 301's E0 is 174224138.371681, EP is 153388627.486270, E0/EP is 1.135835

frame 302's E0 is 186439754.897978, EP is 158935124.730747, E0/EP is 1.173056

frame 303's E0 is 107158245.025065, EP is 96304426.458342, E0/EP is 1.112703

frame 304's E0 is 73197141.028095, EP is 61900717.543654, E0/EP is 1.182493

frame 305's E0 is 29861207.478541, EP is 25703754.631482, E0/EP is 1.161745



整体预测增益：

origin sum energy E0 :4718027734.396956,

LPC sum energy EP :4003251459.794050

E0/EP :1.178549