哈爾濱工業大學

实验报告

实 验（一）

题　　目 DPCM 编解码实验

专　　业 视听觉信号处理

学　　号 1170300511

班　　级 1703105

学　　生 易 亚 玲

指 导 教 师 郑 铁 然

实 验 地 点 G709

实 验 日 期 2019. 10. 28

计算机科学与技术学院

一、 8 比特 DPCM 编解码算法

1.1 简述算法内容

由于 8 位能表示的无符号数的范围是-128 至 127，指数运算后数字非常大（可表示的最大值远大于数据中的最大值，可表示的最小值也远小于数据中的最小值）。所以使用八位编码时不需要考虑上溢和下溢，误差主要发生在进行无符号数编码时舍去的小数部分。

1.1.1 数字的编解码规则

编码规则：若为正数，则直接计算对数的绝对值并转化为 uint8，然后保存这个 uint8；若为负数，则先计算绝对值的对数后取绝对值并转化为 uint8，然后用 255 减去这个 uint8。（用 255 减是因为 uint8 取整是按照靠 0 取整的规则）

解码规则：若解码的数小于 128，说明这是一个正数，则直接进行幂运算得到解码的数据；若解码的数大于等于 128，说明这是一个负数，先用 255 减去这个数，然后幂运算后取相反数得到原数据。

1.1.2 对数据编码写入.dpc 文件：采用‘变压缩边解压’的方式

算法过程：① 处理第一个点：按照编码规则对第一个点进行编码，并写入压缩文件中，然后对第一个压缩数进行解码，并存入解压缩的数组中。

② 处理第 i 个点（i>=1）：将第 i 个数据与第 i-1 个解压缩的数据做差得 d，对 d 进行编码，同时将第 i 个压缩的点进行解码，存入解压缩的数组中，用于下次编码。

1.1.3 读取.dpc 文件中的数据，解压数据

算法过程：以二进制的形式读取文件，每次读取一个字节，struct.unpack 得到对应的十进制数，如果是第一个数，直接解码；如果不是第一个数，解码后与前一个解压缩的数据求和得到这一次的解压缩数据。

1.1.4 写入.pcm 文件

用 wave.open 以二进制写入的形式打开.pcm 文件，配置声道数，量化位数和采样频率，然后将解码得到的列表转化为二进制数据写入.pcm 文件中。

1.1.5 计算信噪比

根据下面的公式求信噪比，要注意的是，数据较大容易溢出，因此需要使用 numpy.longlong 来存储数据，必要时还可以除以数据长度防止溢出。

$$SNR = 10 * \log_{10}\left\{\frac{\sum_{n=0}^{M}(x(n))^2}{\sum_{n=0}^{M}(\overline{x}(n) - x(n))^2}\right\}$$

1.2 解码信号的信噪比

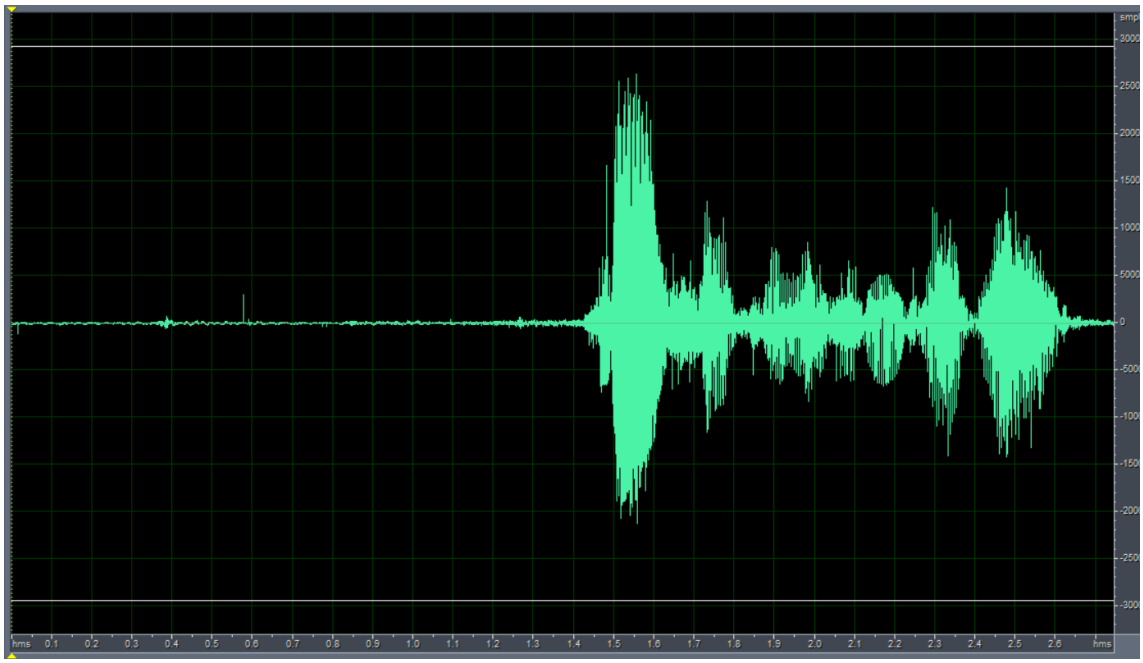
改进前：

SNR = 14.118156458630907

改进后：

SNR = 17.285728068341 65

解码后的波形图如下



二、4 比特 DPCM 编解码算法

2.1 你所采用的量化因子

量化因子：20，

2.2 拷贝你的算法，加上适当的注释说明

```
# 4-bits 编码
def encode_four_bits(self, a):
    num = 0
    n = len(self.data)
    # 先编码第一个数
    # 小于 0
    if self.data[0] < 0:
        if - self.data[0] < a:  # 如果小于量化因子，直接编码为-1
            num = 15
        else:
            num = 15 - (np.uint8(int(0.5 + abs(np.log(- self.data[0] / a))) << 4) >> 4)  # 当数据为负时，符号位设置为 1
        if num < 8:  # 下溢，编码为 8
            self.encode_four.append(8)
        else:  # 正常
            self.encode_four.append(num)
        self.four_decode.append(-a * np.exp((15 - self.encode_four[0])))  # 负数,乘以量化因子 a
    # 大于等于 0
    else:
        if self.data[0] < a:  # 小于量化因子，直接编码为 1
            num = 0
        else:
```

```
            num = np.uint8(int(0.5 +
abs(np.log(self.data[0] / a))) << 4) >> 4
        if num > 7:  # 上溢，编码为 7
            self.encode_four.append(7)
        else:  # 正常
            self.encode_four.append(num)
        self.four_decode.append(a *
np.exp(self.encode_eight[0]))  # 正数解码，乘以量化
因子 a

    # 编码剩余的点
    for i in range(1, n):
        num = 0
        # 与解码后的数据相减的差
        d = self.data[i] - self.four_decode[i - 1]
        if d < 0:  # 差为负
            if -d < a:  # 小于量化因子
                num = 15
            else:
                num = 15 - (np.uint8(int(0.5 +
abs(np.log(-d / a))) << 4) >> 4)  # 当数据为负时，
符号位设置为 1
            if num < 8:  # 下溢
                self.encode_four.append(8)
            else:
                self.encode_four.append(num)

self.four_decode.append(self.four_decode[i - 1]
- a * np.exp((15 - self.encode_four[i])))  # 负数，
乘以量化因子 a
        else:  # 差非负
            if d < a:  # 小于量化因子
                num = 0
            else:
                num = np.uint8(int(0.5 +
abs(np.log(d / a + 1e-5))) << 4) >> 4
            if num >= 8:  # 上溢
                self.encode_four.append(7)
```

```
            else:
                self.encode_four.append(num)

self.four_decode.append(self.four_decode[i - 1]
+ a * np.exp(self.encode_four[i]))

    f = open('1_4bit.dpc', 'wb')  # 打开写入文件
    # 防止奇数个时溢出，故减一
    length = len(self.encode_four)
    for i in range(0, length - 1, 2):
        # 两个一起写
        x = (self.encode_four[i] << 4) +
self.encode_four[i + 1]
        f.write(np.uint8(x))
    if length % 2 == 1:
        x = self.encode_four[length - 1] << 4
        f.write(np.uint8(x))
    f.close()

# 4-bits 解码
def decode_four_bits(self, a):
    f = open('1_4bit.dpc', 'rb')
    cnt = -1
    while True:
        ff = f.read(1)
        if not ff:
            break
        ff = struct.unpack('B', ff)
        ff = ff[0]
        # 第一个数的解析
        if len(self.decode_four) == 0:
            r = 0
            x1 = ff >> 4  # 前 4-bit
            x2 = ff & 15  # 后 4-bit
            if x1 < 8:  # 符号位为 0，则为正数
                r = a * np.exp(x1)
            else:
                r = -a * np.exp(15 - int(x1))
            self.decode_four.append(r)
```

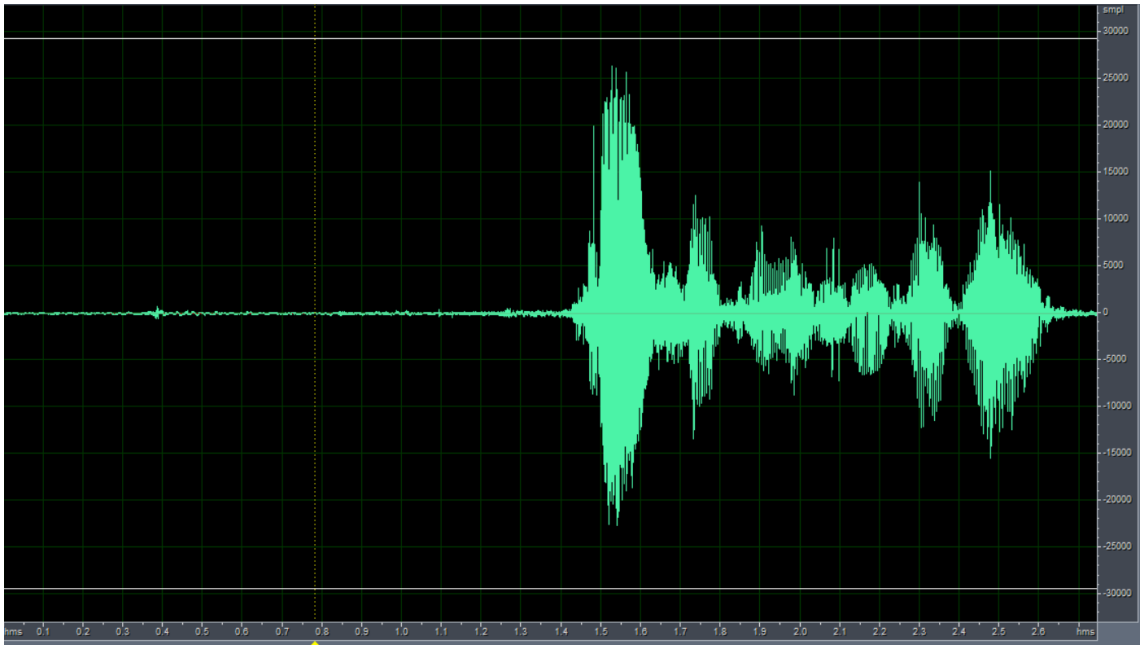
```
            if x2 < 8:  # 符号位为0，则为正数
                r = self.decode_four[0] + a *
np.exp(x2)
            else:
                r = self.decode_four[0] - a *
np.exp(15 - int(x2))
            self.decode_four.append(r)
            cnt += 2

        else:
            x1 = ff >> 4  # 前4-bit
            x2 = ff & 15  # 后4-bit
            if x1 < 8:  # 符号位为0，则为正数
                r = self.decode_four[cnt] + a *
np.exp(x1)
            else:
                r = self.decode_four[cnt] - a *
np.exp(15 - int(x1))
            self.decode_four.append(r)
            cnt += 1
            if x2 < 8:  # 符号位为0，则为正数
                r = self.decode_four[cnt] + a *
np.exp(x2)
            else:
                r = self.decode_four[cnt] - a *
np.exp(15 - int(x2))
            self.decode_four.append(r)
            cnt += 1

    f.close()
    # 计算信噪比
    print(cal_snr(self.data, self.decode_four))
    # 写入文件
    f = wave.open('1_4bit.pcm', 'wb')
    f.setnchannels(1)  # 配置声道数
    f.setsampwidth(2)  # 配置量化位数
    f.setframerate(16000)  # 配置取样频率

f.writeframes(np.array(self.decode_four).astype
```

```
(np.short).tostring())  # 转换为二进制数据写入文件
    f.close()
```

2.3 解码信号的信噪比

```
17.236000611804943
```

解码后的波形图如下：



三、改进策略

3.1 你提出了什么样的改进策略，效果如何

改进 1：8-bit 编码时对无符号数的小数部分的舍弃，不是直接舍弃，而是采用四舍五入的方式，能够有效减少因为小数部分舍弃带来的误差。由于 8-bit 与对数的联合使用效果原本就挺好，所以没有带来特别大的改变，但是信噪比提升了 3

改进 2：4-bit 当绝对值小于量化因子的全部取 1 或者-1，这个方法在 4-bit 中表现非常好，信噪比有质的提升。

改进 3：4-bit 采用量化因子和对数的联合使用，信噪比甚至比 8-bit 结合对数使用的信噪比更高。

改进 4：可以将绝对值小于量化因子一半的直接编码为 0

四、 简述你对量化误差的理解

4.1 什么是量化误差？

量化误差即量化过程中产生的误差，常常发生在连续信号转化为数字信号的过程中，或者将信号压缩存储，如本实验中将数据用 4-bit 的方式存储。

4.2 为什会编码器中会有一个解码器

使用解码器能够时刻跟踪上一个数据解码后的数据是多少，这样能够保证前面的误差不会一直累计，从而对后面的点的采样产生巨大的影响。

五、 总结

5.1 请总结本次实验的收获

对数据的压缩存储有了更新的认识，也掌握了一些简单的压缩方法以及减少误差的方法。

5.2 请给出对本次实验内容的建议

无