信号处理原理实验报告

谢兴宇
2017011326

2020 年 1 月

1 实验一

1.1 Goertzel 算法

我实现了二阶 Goertzel 算法，其应用的差分公式为：

$$v_k(n) = x(n) + 2\cos\left(\frac{2\pi k}{N}\right) v_k(n-1) - v_k(n-2) \tag{1}$$

$$y_k(n) = v_k(n) - v_k(n-1)W_N^k \tag{2}$$

边界条件为：

$$v_k(-1) = v_k(-2) = 0, x(N) = 0$$

算法的输出为 $X(k) = y_k(N)$

我部分实现了 MATLAB 标准库中的`goertzel`，限定其参数为`data`和`freq_indices`，其中`data`必须是一个`size`为`(1, N)`的数组，`freq_indices`中的元素必须为 $[0, N)$ 中的整数。

1.2 基于 FFT 的信号分析

基于 MATLAB 标准库中的`fft`函数，我以另一种方式又实现了一个计算频率幅度的函数，其接口与 Goertzel 算法完全一致。

1.3 性能测试

我模拟生成了 16 个手机按键的信号，以 $Fs = 8\,\text{kHz}$ 的抽样频率抽取 $N = 205$ 个样本。对上述两种实现进行性能测试，其本地测试的结果为：

按键	低频（Hz）	高频（Hz）	Goertzel Algorithm		FFT	
			正确性	耗时（µs）	正确性	耗时（µs）
1	697	1209	✓	1944	✓	3551
2	697	1336	✓	1082	✓	567
3	697	1477	✓	1543	✓	1048
A	697	1633	✓	701	✓	670
4	770	1209	✓	2216	✓	3192
5	770	1336	✓	275	✓	215
6	770	1477	✓	141	✓	54
B	770	1633	✓	76	✓	151
7	852	1209	✓	91	✓	217
8	852	1336	✓	112	✓	84
9	852	1477	✓	110	✓	67
C	852	1633	✓	137	✓	74
*	941	1209	✓	172	✓	145
0	941	1336	✓	127	✓	165
#	941	1477	✓	69	✓	55
D	941	1633	✓	83	✓	53

表 1: 对两种 DTMF 识别算法的性能比较

对于生成的模拟信号，两种算法的精确度均为 100%。

从表中可以看到，我所实现的 Goertzel 算法相比于 FFT 算法并无显著优势，我想这是由于我的实现不如标准库实现精细所致。我的算法像 MAT 主体通过一个 for 循环来实现，而 LAB 一样的解释型语言的循环会导致巨大的时间开销。

2 实验二

记 x 的序列长度为 N，y 的序列长度为 M，对四种卷积算法的理论时间复杂度分析结果为：

- 按直接定义计算：$O((N+M)^2)$
- 利用圆卷积 FFT 计算：$O((N+M)\log(N+M))$

- overlap-add：$O((N+M)\log M)$
- overlap-save: $O((N+M)\log M)$

为了测量四种算法的实际运行性能，随机生成长度为 $N = 2^1, 2^2, \cdots, 2^{15}$ 的序列 x，与长度固定为 $M = 2^9$ 的定长序列 h 作卷积，在本地进行测试，对于四种算法的性能测试结果如图所示：

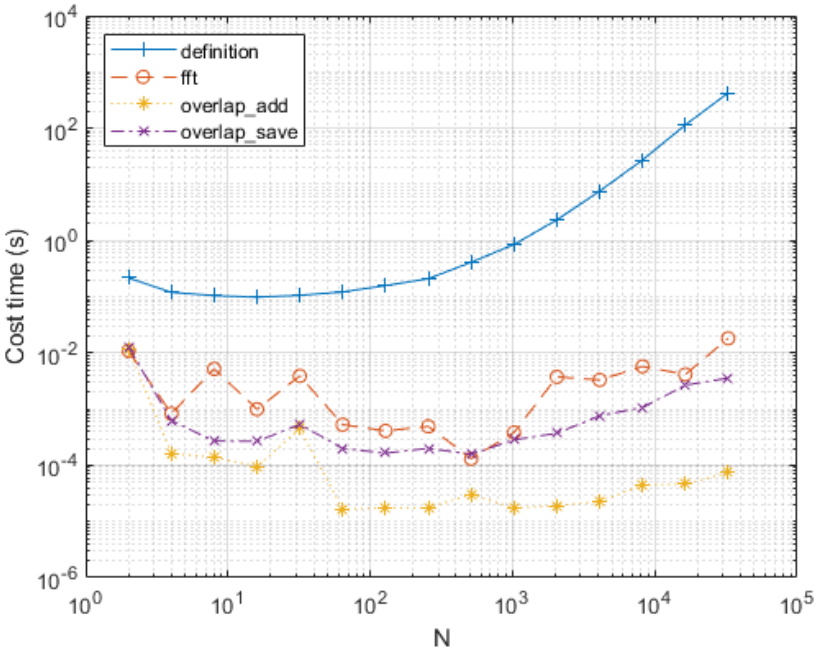

图 1: 对四周线卷积算法的性能测试

这样的结果基本是符合预期的。最快的算法是 overlap_add，其次是 overlap_save，之后是 fft，直接按定义计算是最慢的。之所以，overlap_add 要快于 overlap_save，是由于 overlap_save 在实现上对于分段的长度要比 overlap_add 多一个 2 倍的常数因子。之所以 overlap_save 的运行速度接近 fft，是由于 $M = 512$ 与 N 较为接近所致。

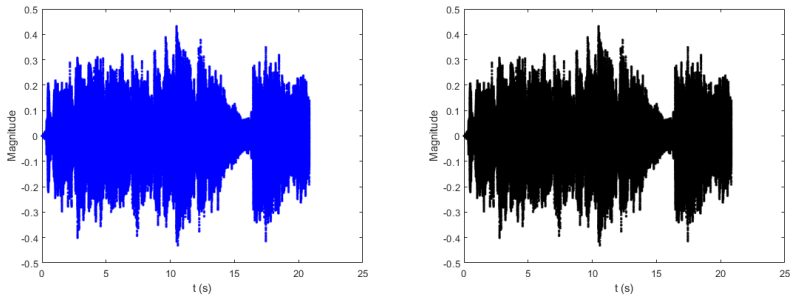

图 2: 时谱图：Let It Go（左）和 Blowin'In The Wind（右）

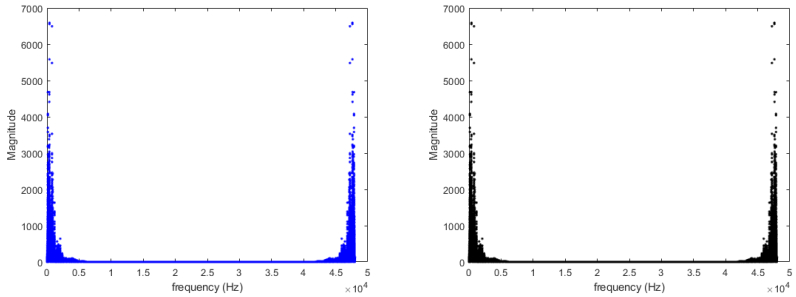

图 3: 频谱图：Let It Go（左）和 Blowin'In The Wind（右）

3 实验三

出于个人兴趣，我尝试将两首我很喜欢的歌曲——电影 *Frozen* 的主题曲 Let It Go 和 Bob Dylan 著名的反战歌曲 Blowin' In The Wind——频分复用之后再解码。

我截取了 40kbps 的 Let It Go（官方原声版）和 Blowin'In the Wind（官方原声版）的前 1M 个抽样，其在时谱图如图2所示，频谱图如图3所示。

对其频分复用进行编码之后的时谱图和频谱图如图4所示，同时将编码之后的录音输出到了 mixed.wav 中。

解码之后，得到的 Let It Go 如图5所示，输出到 decodedLetItGo.wav 中；得到的 Blowin'In The Wind 如图**??**所示，输出到 decodedBlowinInTheWind.wav 中。

实际试听的效果并没有非常良好，处于刚刚可以听懂歌词的地步，两首歌产生了一定的混杂。经过排查，这实际上是由于 FFT 精度的误差导致的。

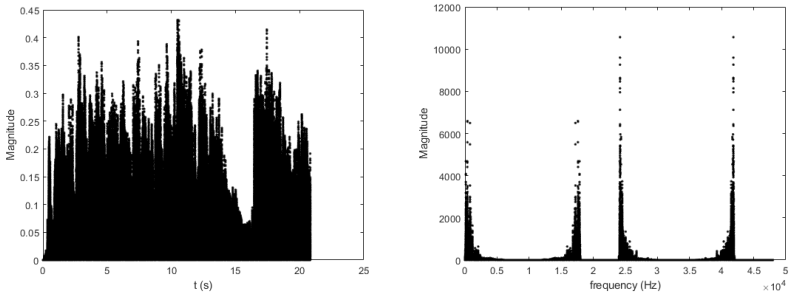

图 4: 频分复用的时谱图（左）和频谱图（右）

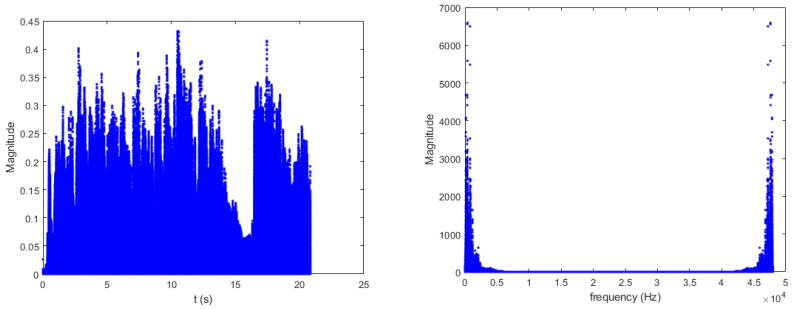

图 5: 解码之后得到的 Let It Go 的时谱图（左）和频谱图（右）

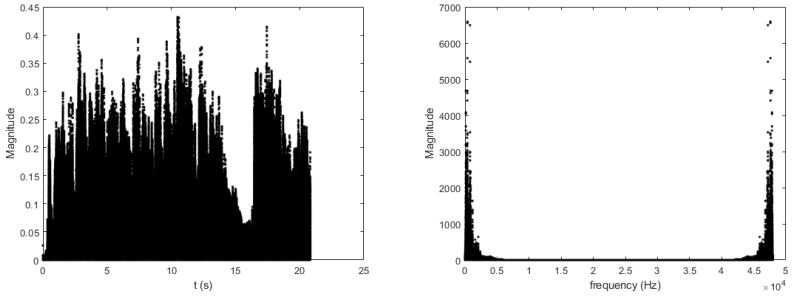

图 6: 解码之后得到的 Let It Go 的时谱图（左）和频谱图（右）

FFT 是十分精妙的算法，但由于其中涉及大量的复数运算，即使是 64 位的双精度浮点数也难免会引入不小的误差。在最终我输出的 Let It Go 的音频中，其最大的绝对值有 0.4319，但其与原音频的方差却达到了 0.8623。FFT 的精度具体有多坏、究竟该如何衡量也一直是我很感兴趣的问题之一，可惜一直没有机会去深入探索。

FFT 的精度问题也并不是无法解决，事实上，我们可以通过 NTT（Number Theoretic Transform）来做到无精度损失，这是一种以数论上的原根取代 FTT 中用到的单位根的方法。但由于 NTT 在处理更大规模和更大值域的数据上尚有一些缺陷，此种方法尚未得到广泛普及，尤其是未在工业界广泛普及，尚缺乏成熟有效的实现。由于我的时间和精力限制，还有一些其他课程的大作业需要完成，虽然我也很想尝试实现一个无损的频分复用，听到一个无损的 Let It Go 和 Blowin'In The Wind，但在 MATLAB 中实现一个有足够效率的 NTT 于我绝非易事，考虑到投入产出比，便不得不暂且搁置了。