

MATLAB 音乐合成

文件结构

```
.
├── code
│   ├── data
│   │   ├── additional # 自定义数据
│   │   │   ├── cruel_angels_action_plan.json
│   │   │   └── the_truth_that_you_leave.json
│   │   └── original # 原有数据
│   │       ├── fmt.wav
│   │       └── Guitar.MAT
│   ├── hw_1_2_1_1.m # 实验代码
│   ├── hw_1_2_1_2.m
│   ├── hw_1_2_1_3.m
│   ├── hw_1_2_1_4.m
│   ├── hw_1_2_1_5.m
│   ├── hw_1_2_2_6.m
│   ├── hw_1_2_2_7.m
│   ├── hw_1_2_2_8.m
│   ├── hw_1_2_2_9.m
│   ├── hw_1_2_3_10.m
│   ├── hw_1_2_3_11.m
│   ├── hw_1_2_3_12.m
│   ├── hw_1_2_3_12.mlapp
│   ├── generate_fft.m # 辅助函数
│   ├── generate_music.m
│   ├── generate_music_with_harmonic.m
│   ├── generate_peak_point.m
│   ├── generate_table.m
│   ├── get_harmonic.m
│   ├── get_harmonic_test.m
│   ├── make_envelope.m
│   ├── make_envelop_tb.m
│   ├── play_music_with_json_and_harmonic.m
│   ├── play_music_with_json.m
│   ├── search_in_standard_table.m
│   ├── search_nearest_tune.m
│   └── test_peak_point.m
└── 实验报告.pdf
```

1.2.1 简单的合成音乐

1.2.1.1 合成《东方红》音乐（基础版）

主要思想

使用不同频率的正弦波生成音乐。

核心代码及说明

本题代码位于 `hw_1_2_1_1.m` 内，生成文件为 `hw_1_2_1_1.wav`。

根据“十二平均律”，我们首先将需要用到的乐音制成表格。值得注意的是，为了方便音乐简谱编写，我们可以根据F调唱名和音名的特点设计乐音表格：

```
% params
freq = 8000;
base_ratio = 2^(1/12);

% generate table
base_freq = [220, 440, 880];
f_tune_base = [-4, -2, 0, 2, 3, 5, 7]; % F G A bB C D E F
freq_list = base_freq'.*(base_ratio.^ f_tune_base);
freq_table = reshape(freq_list, [3, 7]);
```

这样在取用乐音是十分简单，例如，我们需要取用 5 这一音时，只需直接调用 `freq(2,5)`，不容易出错。

在编写音乐时，需要将所有的乐音及其对应的节拍数列作列表：

```
% melody settings
song_list = [
    freq_table(2,5), 1;
    freq_table(2,5), 0.5;
    freq_table(2,6), 0.5;
    freq_table(2,2), 2;
    freq_table(2,1), 1;
    freq_table(2,1), 0.5;
    freq_table(1,6), 0.5;
    freq_table(2,2), 2;
];
```

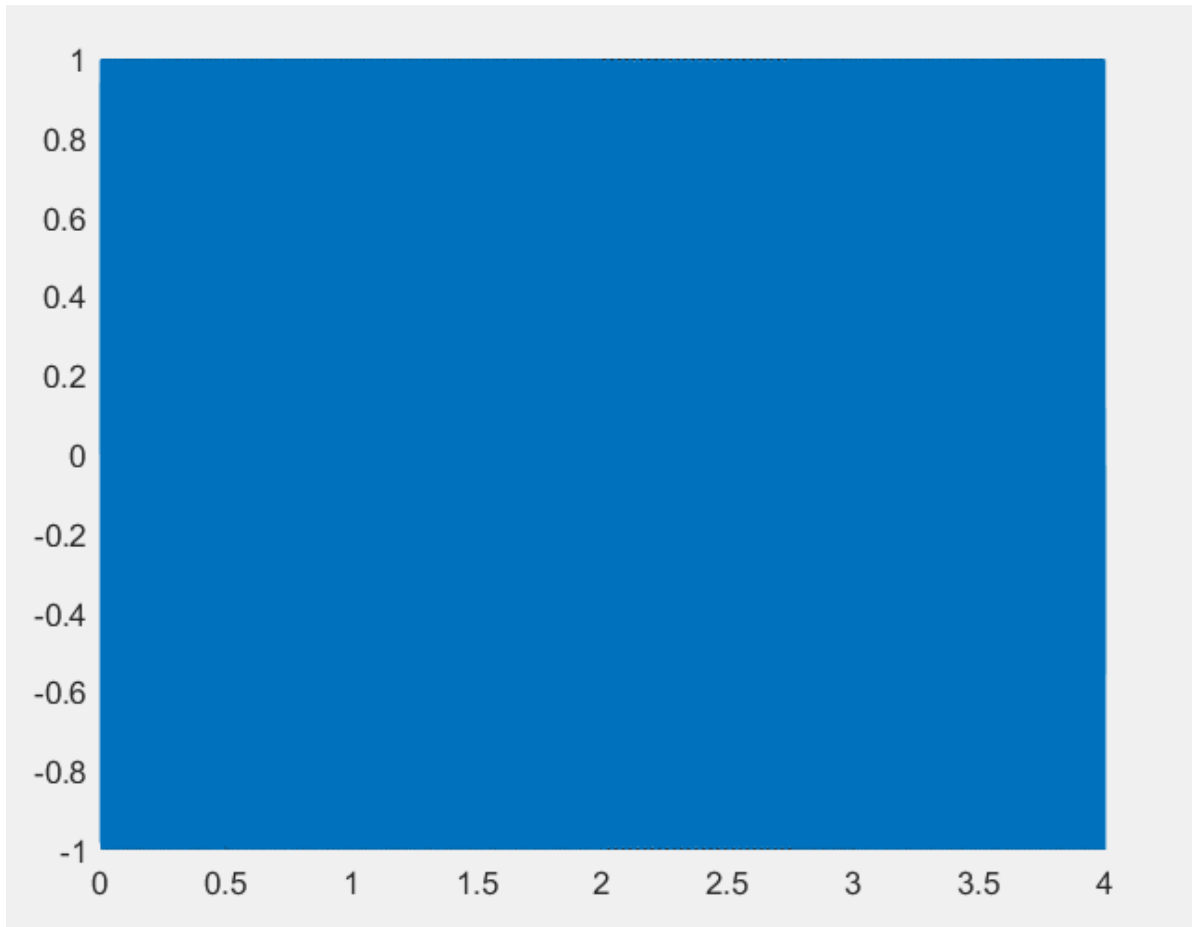
最后，我们需要使用定义好的简谱生成不同频率和长度的正弦波，并依次拼接，生成完整的音乐：

```
% generate the music
beat = 0.5;
song_data = [];
for i = 1:length(song_list)
    single_freq = song_list(i,1);
    single_time_length = song_list(i,2) * beat;
    t = linspace(0, single_time_length, single_time_length * freq);
    single_song_data = sin(2 * pi * single_freq .* t');
    song_data = [song_data; single_song_data];
end
```

绘制生成的波形，并将其播放、保存：

```
% play and save the music
plot([0:length(song_data)-1]/freq, song_data);
sound(song_data, freq);
audiowrite('hw_1_2_1_1.wav', song_data, freq);
```

运行结果



结果分析

由上图可以看到，音乐的幅度始终相等，但这也带来了一个问题：在音乐由一个音跳变到另一个音时，由于相位不连续产生高频分量，导致相邻的乐音之间有着杂声。

1.2.1.2 使用包络抑制噪声

主要思想

指导说明¹中给出了不同乐音之间音量变化的一般规律。但同时说明中也指出，为了确保包络面平滑，推荐使用指数衰减的包络面来处理音乐。我们可以模仿钢琴波形包络来设计包络面：

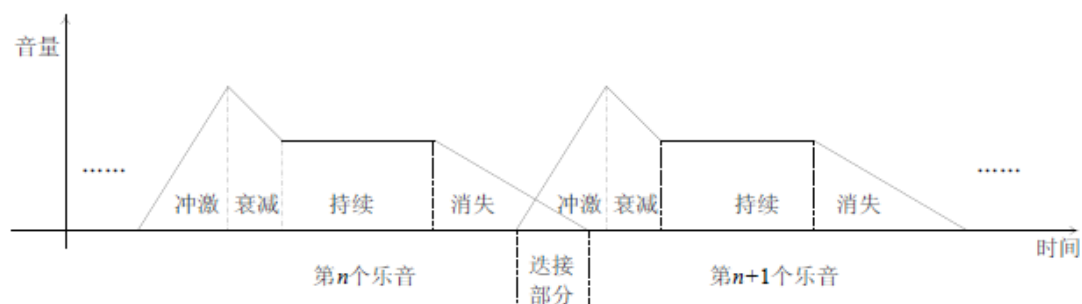
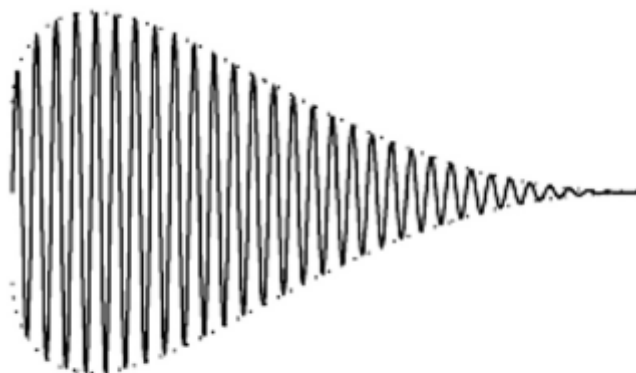


图 1.5: 音量变化



钢琴:

此包络面的形状让我很容易联想到高中导数大题中非常常见的一种函数形式：

$$f(x) = axe^{-bx} \quad (a > 0, b > 0) \quad (1)$$

此函数在 $x = \frac{1}{b}$ 处取得最大值 $\frac{a}{be}$ ，在 $x \rightarrow 0$ 和 $x \rightarrow \infty$ 时 $f(x)$ 均趋于 0，而且在 $\frac{df}{dx} > 0$ 时增长较快，在 $\frac{df}{dx} < 0$ 时衰减较慢，符合我们对于包络面的要求。

当然在实践中，由于信号的起始时间和持续时间不相同，我们需要引入缩放倍数 l 和平移系数 c 对包络面的形状进行调整，即包络面方程变为 $g(x) = f(\frac{x-c}{l})$ 。

核心代码及说明

本题主代码位于 `hw_1_2_1_2.m` 内，辅助函数文件为 `make_envelope.m`，生成文件为 `hw_1_2_1_2.wav`。

我们首先将上述构造包络面的函数封装为一个 MATLAB 函数：

```
function [y] = make_envelope(x, a, b)
    y = a*x.*exp(-b.*x).*(x>0);
end
```

此函数需要满足以下条件：

- $f(x)_{max} = 1$
- $f(1) \ll 1$

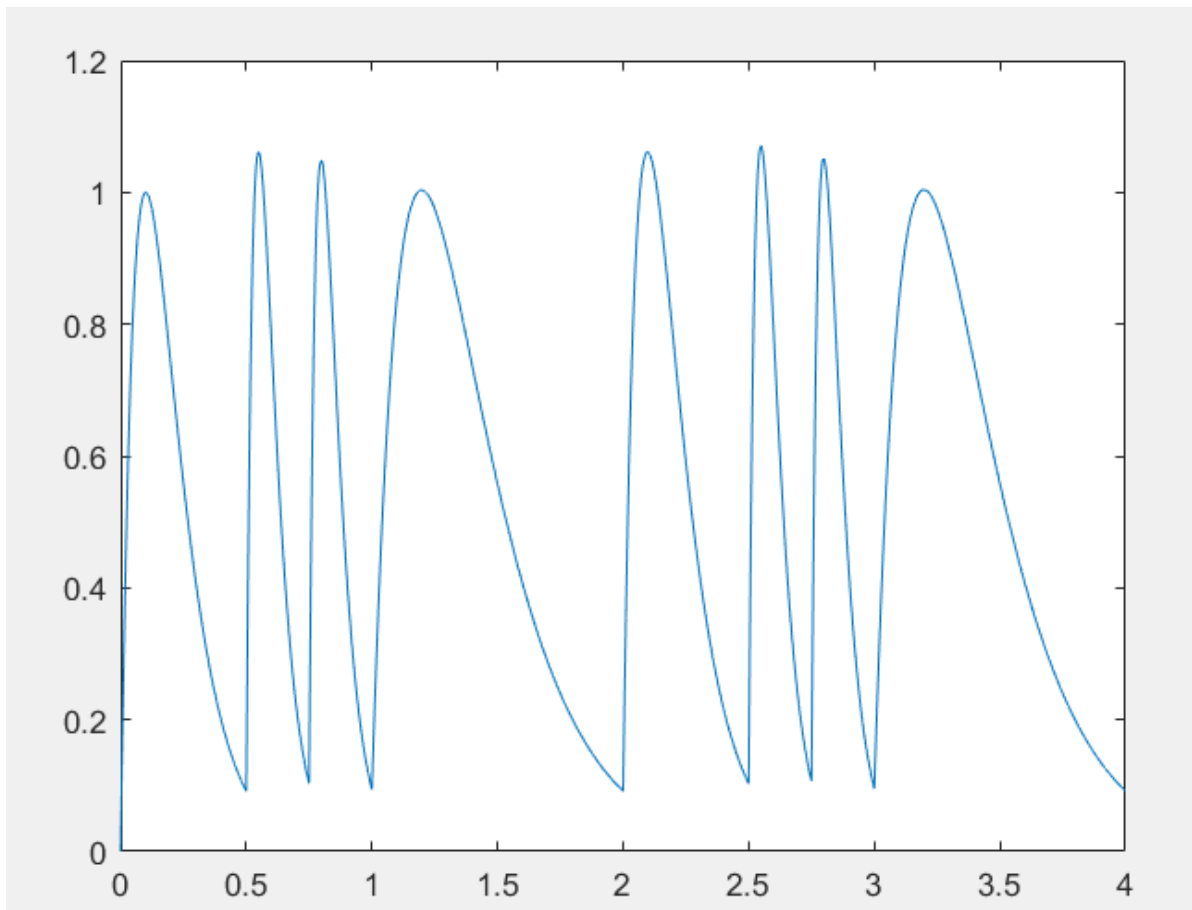
这就要求 $a = be$ ， $ae^{-b} \ll 1$ 。经过调参，计算，我发现当 $a = 5e$ ， $b = 5$ 时，包络面可以起到较好的效果。

在生成乐音时，在题1.2.1.1的基础上，我们首先需要生成不同频率的正弦波，之后根据每一段乐音的长度计算出包络面的缩放尺度 `single_time_length` 和平移距离 `c`，然后将包络面方程和正弦波相乘，最后依次累加即可。

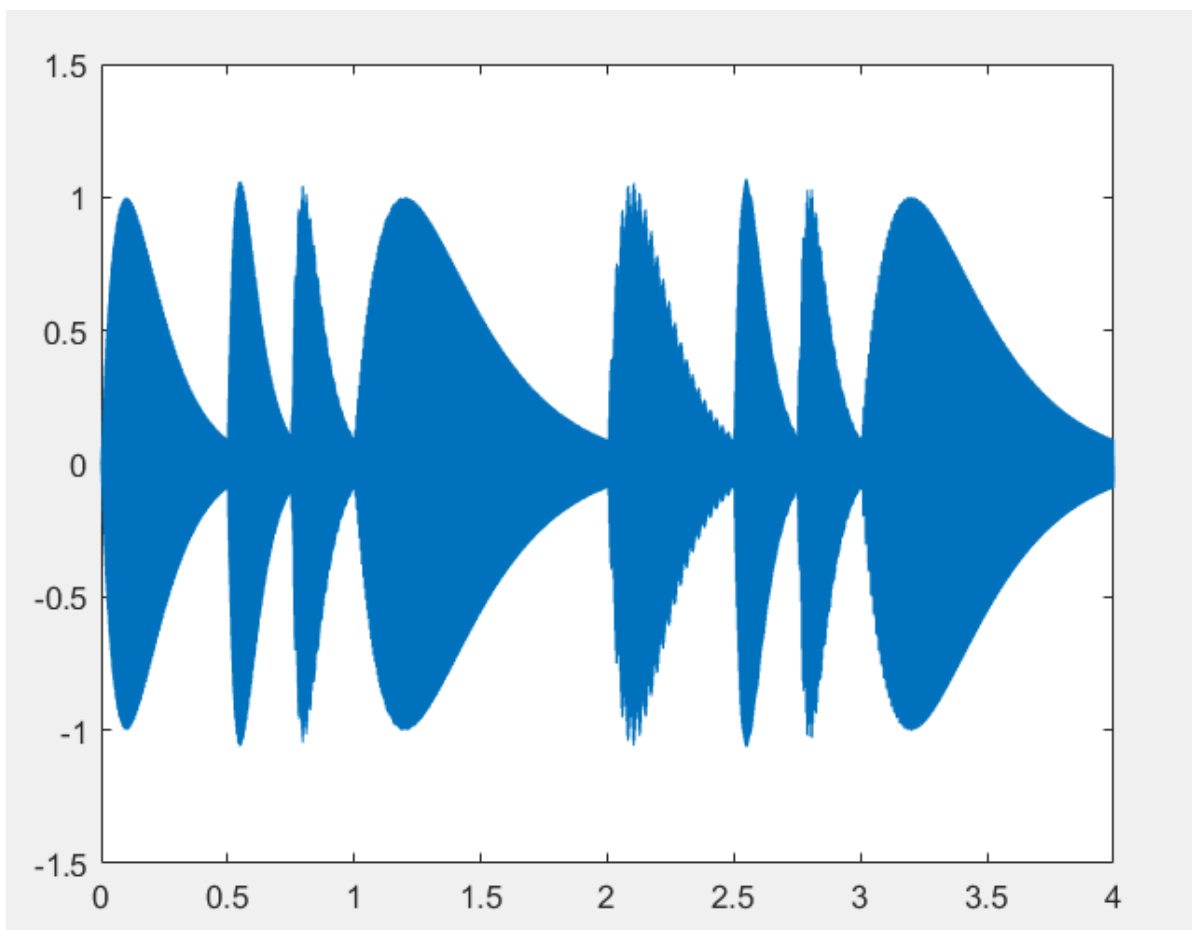
```
for i = 1:1:length(song_list)
    single_freq = song_list(i,1);
    single_time_length = song_list(i,2) * beat;
    if i == 1
        c = 0;
    else
        c = c + song_list(i-1,2) * beat;
    end
    single_song_data = sin(2 * pi * single_freq .* t');
    single_envelope = make_envelope((t'-c)/single_time_length,a,b);
    masked_single_song_data = single_song_data .* single_envelope;
    envelope = envelope + single_envelope;
    song_data = song_data + masked_single_song_data;
end
```

运行结果

包络面方程如下图所示：



音乐波形如下图所示：



与题1.2.1.1相比，生成的音乐要更加悦耳自然，音与音之间的杂音消失。

1.2.1.3 将生成的音乐升高/降低一个八度/半个音阶

主要思想

升高/降低一个八度，即令音乐的频率扩大/缩小一倍，所以只需要在播放时调节采样率为之前的2倍/0.5倍。

升高半个音阶，即令音乐的频率扩大为原来的 $2^{\frac{1}{12}}$ 倍，所以我们需要改变采样率为之前的 $2^{\frac{1}{12}}$ 倍。

核心代码及说明

本题主代码位于 `hw_1_2_1_3.m` 内，辅助函数文件为 `make_envelope.m`，生成文件为

`hw_1_2_1_3_1.wav`（升高一个八度）、`hw_1_2_1_3_2.wav`（降低一个八度）、`hw_1_2_1_3_3.wav`（升高半个音阶）。

值得指出的是，在升高半个音阶时，我们需要使用 `resample` 函数，使用 $2^{\frac{1}{12}}$ 倍的采样率对音乐进行采样，得到新的数据样点，之后按照原采样率进行播放。

```
sound(song_data, freq*2);
audiowrite('hw_1_2_1_3_1.wav', song_data, freq*2);

sound(song_data, freq/2);
audiowrite('hw_1_2_1_3_2.wav', song_data, freq/2);

new_song_data = resample(song_data, freq, round(freq*2^(1/12)));
sound(new_song_data, freq);
audiowrite('hw_1_2_1_3_3.wav', new_song_data, freq);
```

1.2.1.4 增加谐波分量

主要思想

在音乐中增加一些谐波分量，可以使得音乐更加有厚度。此处我们选择基波幅度为1，二次谐波幅度0.2，三次谐波幅度0.3。

核心代码及说明

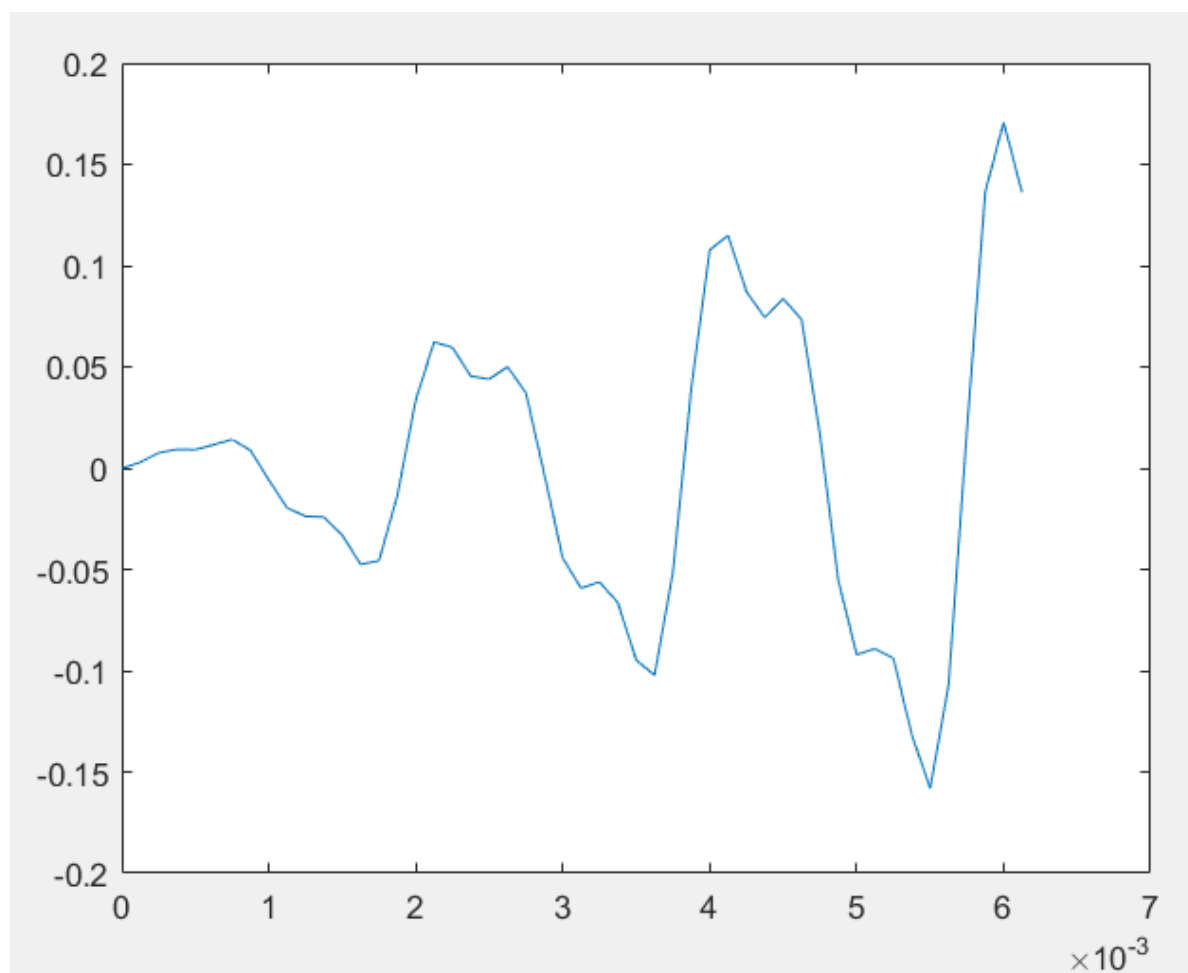
本题主代码位于 `hw_1_2_1_4.m` 内，辅助函数文件为 `make_envelope.m`，生成文件为 `hw_1_2_1_4.wav`。

在1.2.1.2的基础上，我们需要在原有波形的基础上添加谐波：

```
single_song_data = sin(2 * pi * single_freq .* t') + 0.2 * sin(2 * pi * 2 *  
single_freq .* t') + 0.3 * sin(2 * pi * 3 * single_freq .* t');
```

运行结果

波形的放大截图如下图所示：



可以看到，谐波分量添加成功，生成音乐类似于风琴声。

1.2.1.5 自选音乐合成

主要思想

为了可以更加自由地配置音乐地播放选项，也为之后编写GUI程序做铺垫，我们可以将生成音乐波形的若干算法封装成为若干函数。播放音乐时，有若干参数可以调节：

- 调型：通过调型指定基波频率，从而得到不同的乐音表以供取用
- 采样率：调节采样率
- 节拍数：一拍所对应的秒数
- 包络面参数： a 、 b
- 谐波分量幅度： λ_1 、 λ_2

核心代码

本题主代码位于 `hw_1_2_1_5.m` 内，辅助函数文件为 `make_envelope.m`，`generate_table.m`，生成文件为 `hw_1_2_1_5.wav`。

定义函数 `generate_table.m`，根据选择的调型生成不同的频率表：

```
function [freq_table] = generate_table(major)
base_ratio = 2^(1/12);
base_freq = [110, 220, 440, 880];
C_tune_base = [3, 5, 7, 8, 10, 12, 14];

% ref: https://zhuanlan.zhihu.com/p/36706001
switch (major)
    case('C')
        ...
    otherwise
        ME = MException('major not supported. Supported majors: A-G (flat/sharp)');
        throw(ME);
end
freq_list = base_freq'.*(base_ratio.^ tune_base);
freq_table = reshape(freq_list, [4, 7]);
end
```

定义函数 `generate_music.m`，根据输入的音乐数组生成音乐：

```
function [t,song_data] = generate_music(song_list, beat, a, b, lambda_1, lambda_2, freq)
total_time_length = sum(song_list(:,2)) * beat;
t = linspace(0, total_time_length, total_time_length * freq);
song_data = 0;
envelope = 0;

for i = 1:length(song_list)
    single_freq = song_list(i,1);
    single_time_length = song_list(i,2) * beat;
    if i == 1
        c = 0;
    else
        c = c + song_list(i-1,2) * beat;
    end
end
```



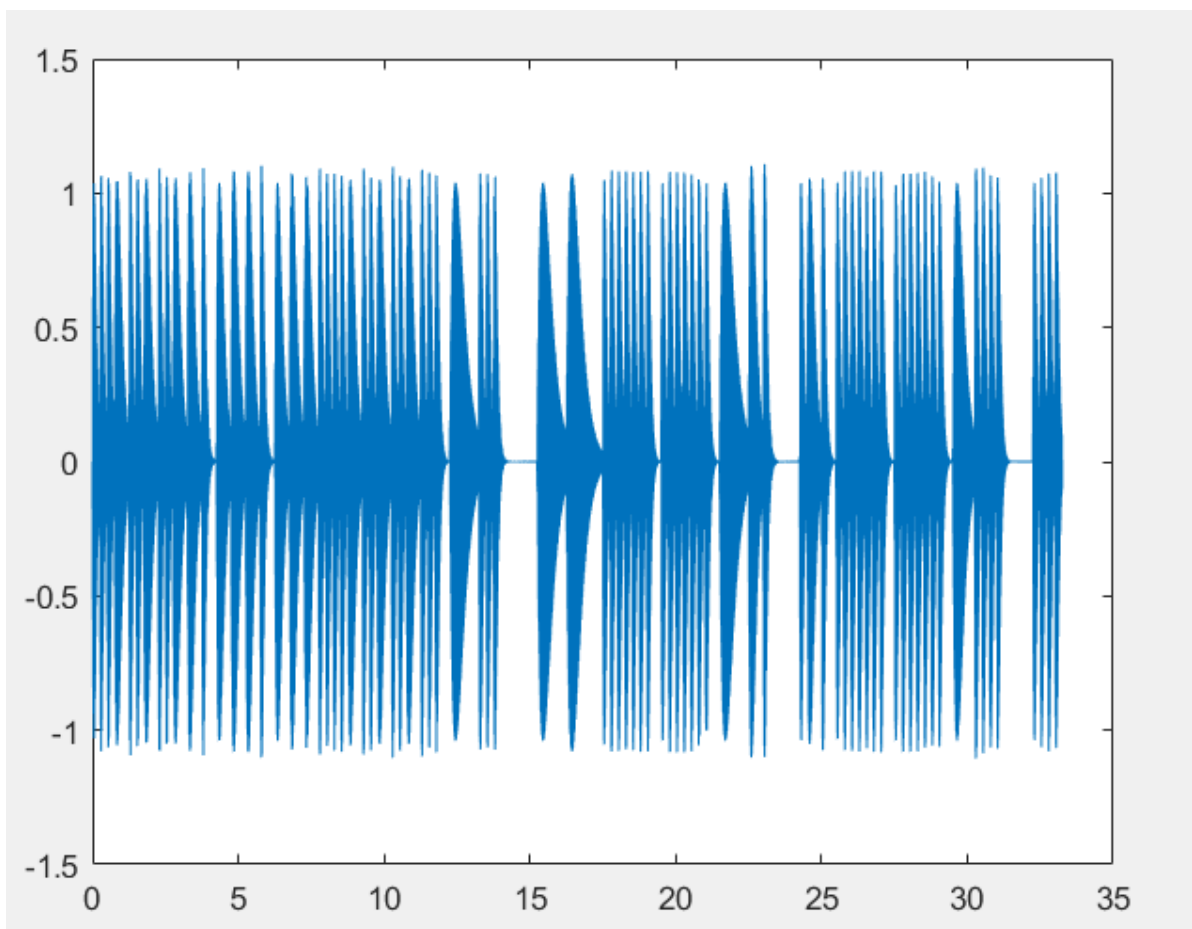
```

end
single_song_data = sin(2 * pi * single_freq .* t') + lambda_1 * sin(2 * pi *
2 * single_freq .* t') + lambda_2 * sin(2 * pi * 3 * single_freq .* t');
single_envelope = make_envelope((t'-c)/single_time_length,a,b);
masked_single_song_data = single_song_data .* single_envelope;
envelope = envelope + single_envelope;
song_data = song_data + masked_single_song_data;
end
end

```

自选音乐为经典钢琴曲《The Truth That You Leave》的一个片段。此音乐为降B大调，每小节4拍，一个4分音符唱1拍。直接调用上述函数即可播放音乐。播放参数为：beat = 0.5, a = 5e, b = 5, lambda_1 = 0.2, lambda_2 = 0.1。

运行结果



最后生成的音乐与原曲较为相近。

1.2.2 用傅里叶级数分析音乐

1.2.2.1 播放 fmt.wav

本题主代码位于 hw_1_2_2_6.m 内。

播放音乐的代码如下所示：

```

[data, freq] = audioread('data/original/fmt.wav');
sound(data, freq);

```

1.2.2.2 wave2proc和realwave

核心代码及说明

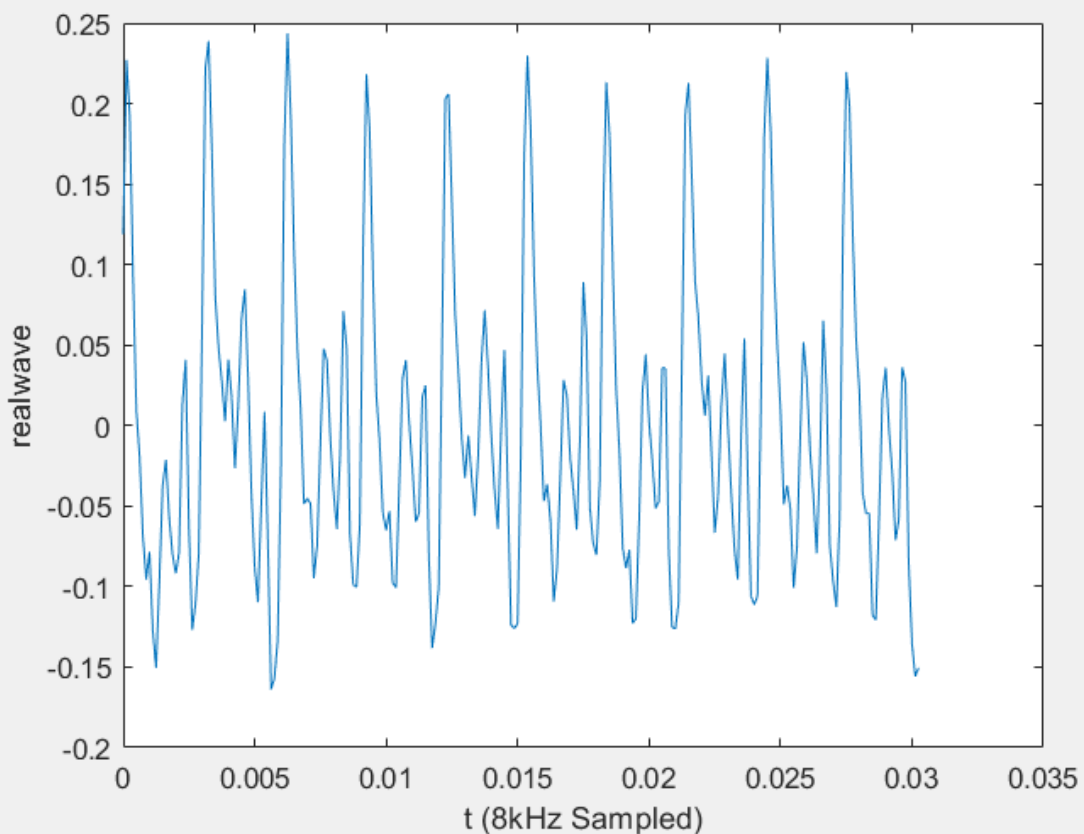
本题主代码位于 `hw_1_2_2_7.m` 内。

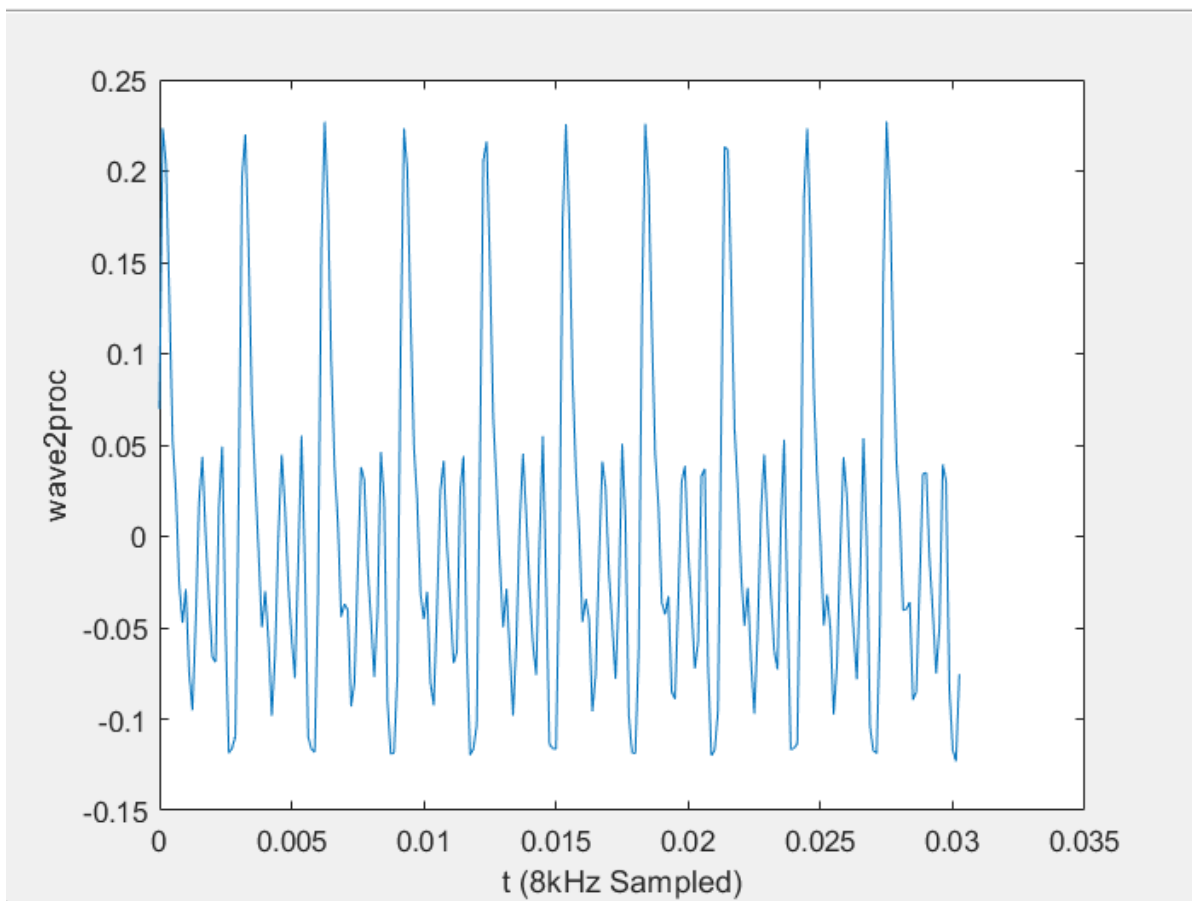
首先，我们绘制出两段音乐的波形：

```
load('data/original/Guitar.MAT');
freq = 8000;
len_realwave = length(realwave);
len_wave2proc = length(wave2proc);
t_realwave = linspace(0, (len_realwave-1)/freq, len_realwave);
t_wave2proc = linspace(0, (len_wave2proc-1)/freq, len_wave2proc);

figure(1);
plot(t_realwave, realwave);
xlabel('t (8kHz Sampled)');
ylabel('realwave');

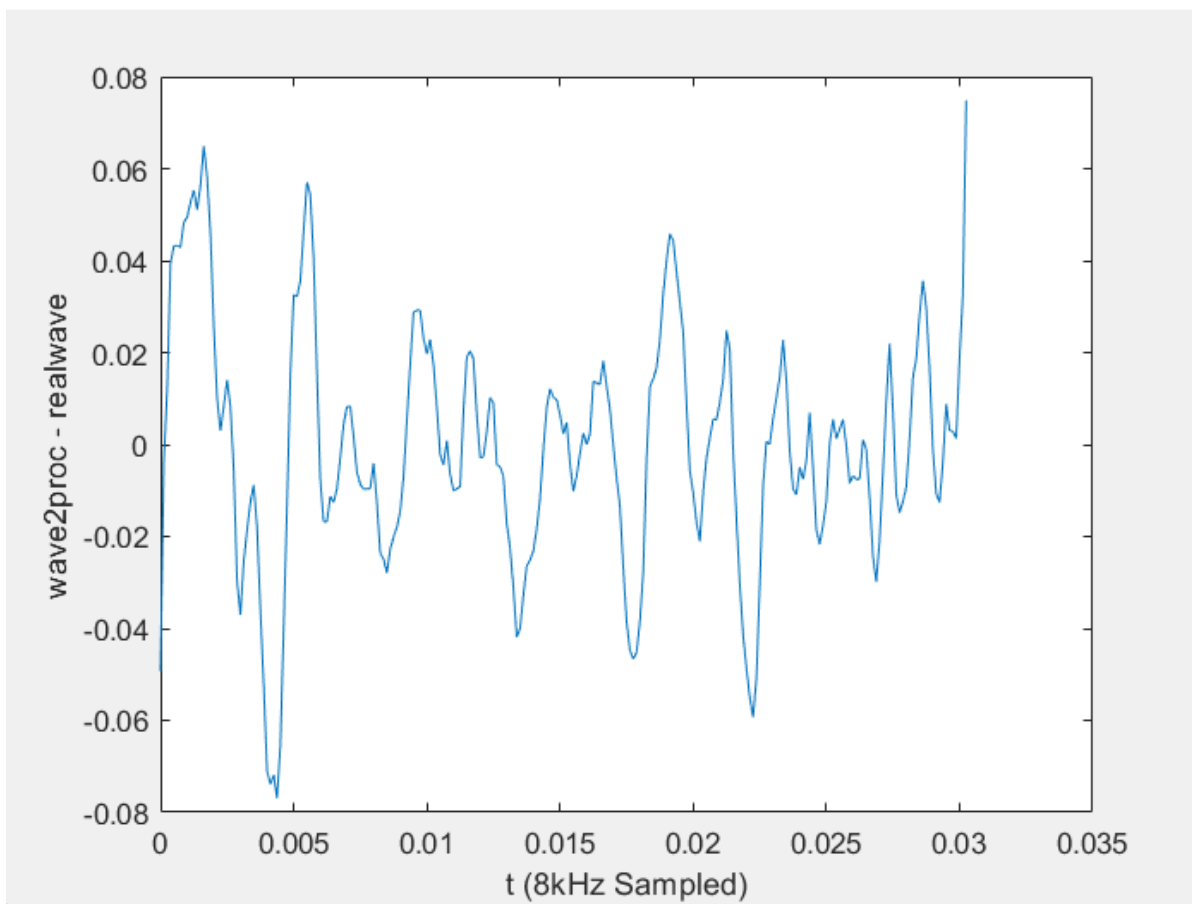
figure(2);
plot(t_wave2proc, wave2proc);
xlabel('t (8kHz Sampled)');
ylabel('wave2proc');
```





可以看出，wav2proc的波形较为规律，而realwave的波形不太规律。为了从wav2proc得到realwave，我们需要对音乐进行去噪。

尽管realwave的波形形状不规则，但我们可以大致看出其波形具有一定的周期性，并在0.03s内重复了10个周期。将realwave和wav2proc作差，绘制其图像，发现差值的积分值接近于0：



由此我们可以尝试：将不同周期的数据求和并计算其平均值。我们首先使用 `resample` 函数对数据进行重采样，以使得采样点总数为10的倍数：

```
realwave_resample = resample(realwave, 50, 1);
```

之后我们对重采样之后的数据点等距离划分为10份，并对这10份数据求平均：

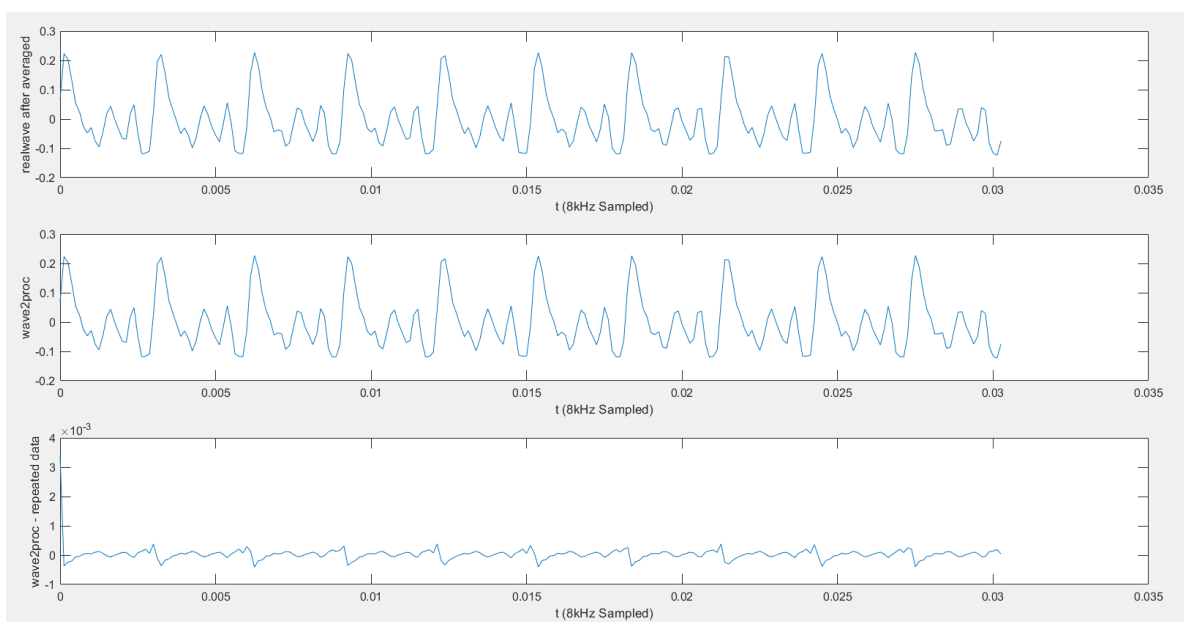
```
% resample
realwave_resample = resample(realwave, 50, 1);
len_realwave_resample = length(realwave_resample);
single_period_data = 0;
for i = 0:1:9
    single_period_data = single_period_data + realwave_resample(i*
(len_realwave_resample/10)+1:(i+1)*(len_realwave_resample/10));
end
single_period_data = single_period_data / 10;
```

最后，将得到的数据重复10个周期，并进行重采样以恢复之前的采样点数：

```
repeated_data = repmat(single_period_data, [10, 1]);
repeated_data = resample(repeated_data, 1, 50);
```

运行结果

将平均后的波形、wav2proc及其差值绘制于同一个表中，发现经过平均之后噪声被明显削弱。



1.2.2.8 使用Fourier变换分析谐波

根据指导书中的提示，分别取用1个周期、10个周期、100个周期对信号进行快速傅里叶变换：

核心代码及说明

本题主代码位于 `hw_1_2_2_8.m` 内，辅助函数为 `generate_fft.m`。

将计算FFT的代码封装为函数 `generate_fft.m`，输入信号和采样频率，返回频域和单侧幅值频谱。

```
function [f,P1] = generate_fft(signal, freq)

len = length(signal);
fft_result = fft(signal);
P2 = abs(fft_result);
P1 = P2(1:len/2+1);
P1(2:end-1) = 2*P1(2:end-1);
f = freq*(0:(len/2))/len;

end
```

之后选取1, 10, 100个周期对信号进行快速傅里叶变换：

```
[f1, p1] = generate_fft(wave2proc(1:round(len_wave2proc/10)), freq);
[f2, p2] = generate_fft(wave2proc, freq);
[f3, p3] = generate_fft(repmat(wave2proc,[10,1]), freq);
```

由于采样点个数为 $n = 243$ ，采样频率为 $f = 8000Hz$ ，周期数为 $N = 10$ ，因此基频频率为：

$$f_0 = \frac{fN}{n} = 329.2Hz \quad (2)$$

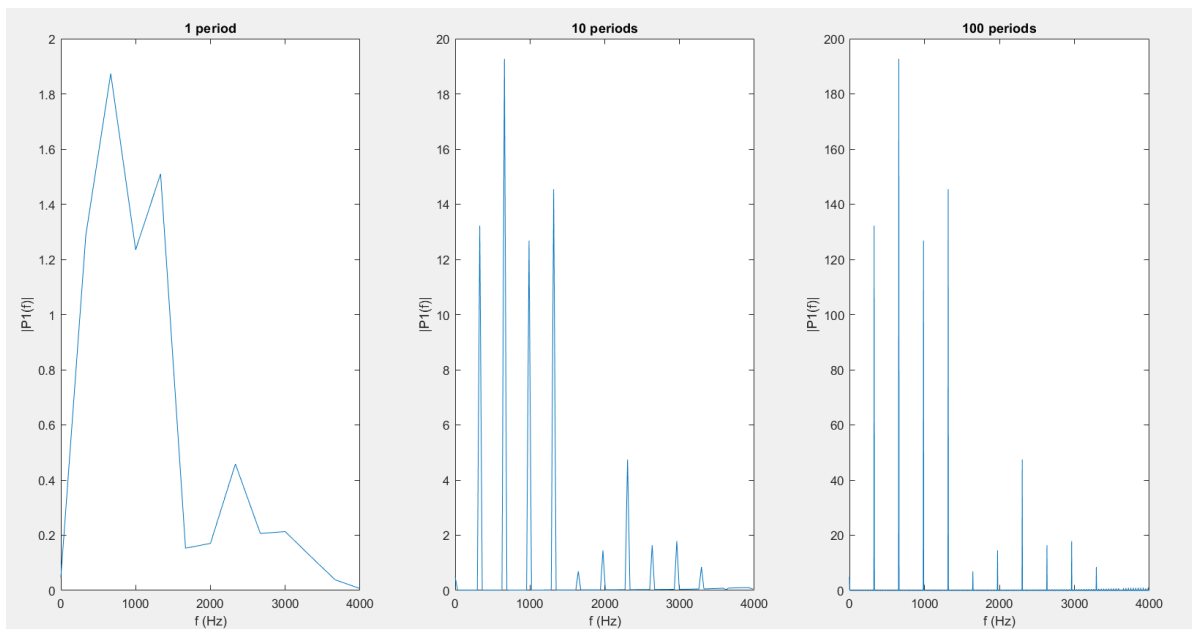
查表可知，与该基频分量最相近的音调是E0（329.63Hz）。

根据奈奎斯特采样定律，在该采样率下谐波的最高频率不超过4000Hz，因此最多有12个谐波分量。

为获得各个谐波分量的幅度，我们需要搜索单侧幅值频谱中的峰值。每一个峰值一定出现在 nf_0 ($n = 1, \dots, 11$)附近，因此，我们可以先粗略定位每一个峰值所在的位置，之后在其附近取一个较小的区间，寻找该区间的最大值。最后对各个峰值除以基频分量，得到归一化之后的值。

```
f0 = round(freq*10/len_wave2proc);
result = [];
len_interval = 2;
len_pruned = round(length(f3)*(f0*12/4000));
len_per_peak = round(len_pruned/12);
for i = len_per_peak:len_per_peak:11*len_per_peak
    max_val = max(p3((i-len_interval):(i+len_interval)));
    result = [result, max_val];
end
result = result/result(1);
for i = 1:length(result)
    disp([i, result(i)]);
end
```

运行结果



可以看到，随着周期数的增大，频谱分布越来越清晰。从图中可以直接观察到，谐波分量个数为10。

依次打印出各谐波分量与基频的比值：

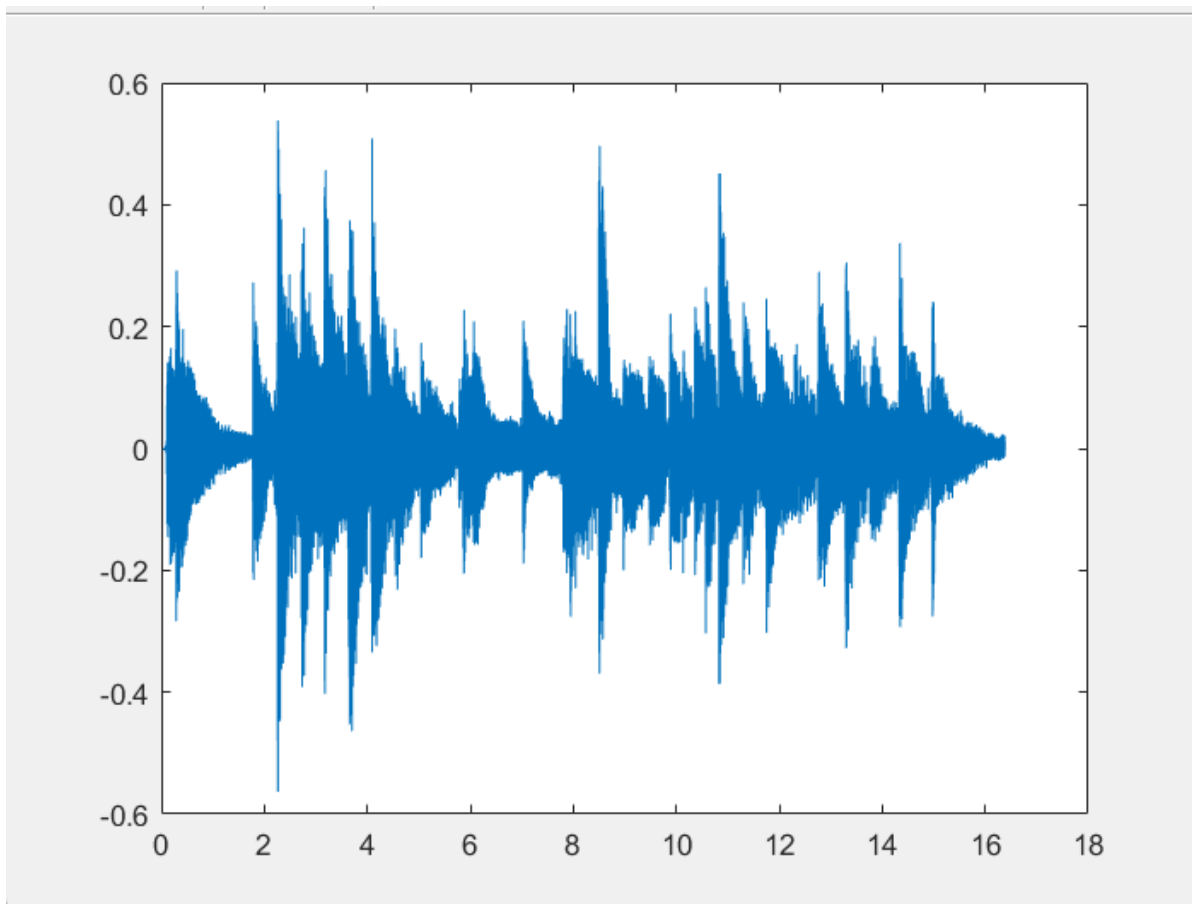
基频倍数	峰值强度
1	1
2	1.4572
3	0.9587
4	1.0999
5	0.0523
6	0.1099
7	0.3589
8	0.1240
9	0.1351
10	0.0643
11	0.0019

1.2.2.9 分析乐曲的音调和节拍

核心代码及说明

本题主代码位于 `hw_1_2_2_9.m` 内，辅助函数为 `generate_fft.m`，`generate_peak_point.m`，`search_nearest_tune.m`，`search_in_standard_table.m`。

我们首先观察音乐的波形：



要想提取出音乐的节拍，就需要提取出音乐的节奏点。若做粗略分析，我们可以粗略地认为节奏点与极值点重合。

此处的方法参考了²中的方法。提取节奏点的方法大致分为5步：

1. 幅度平方求能量：

$$y_1(n) = x^2(n) \quad (3)$$

```
% step 1
energy = wav_data.^2;
```

2. 加窗平滑得到包络：

$$y_2(n) = \sum_{i=0}^{M-1} w(i)y_1(n-i) \quad (4)$$

将此式子的形式加以改写，可以得到一个类似于卷积和的函数。可选的窗函数有：`boxcar`（矩形窗）、`hanning`（汉宁窗）、`hamming`（海明窗）、`blackman`（布拉克曼窗），其中窗的长度可以调整。

```
% step 2
switch (window_name)
case('boxcar')
    y_2 = conv(y_1, boxcar(window_length));
case('hanning')
    y_2 = conv(y_1, hanning(window_length));
case('hamming')
    y_2 = conv(y_1, hamming(window_length));
case('blackman')
```

```

        y_2 = conv(y_1, blackman(window_length));
    otherwise
        ME = MException('window not supported');
        throw(ME);
    end

    y_2 = y_2(window_length:end);

```

3. 做差分得到能量变化点：

$$y_3(n) = y_2(n) - y_2(n-1) \quad (5)$$

```

% step 3
y_3 = diff(y_2);
y_3 = [y_3;0];

```

4. 半波整流：

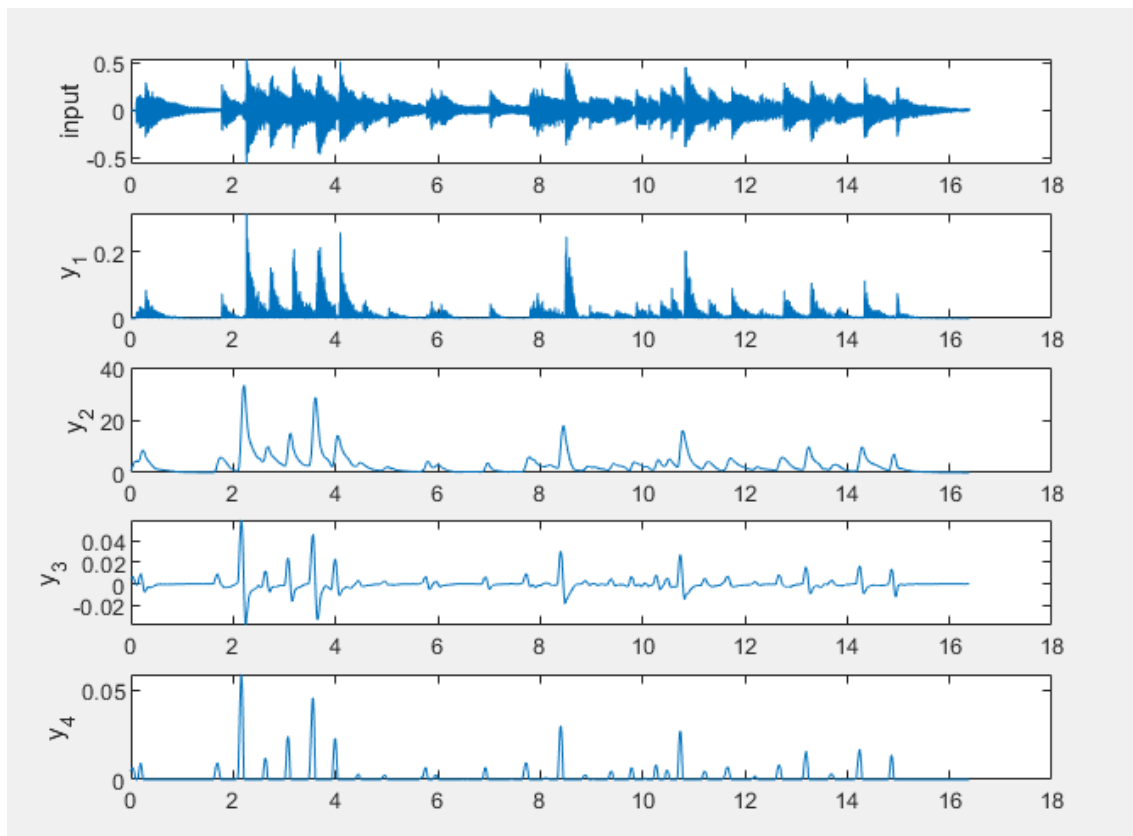
$$y_4(n) = \max\{y_3(n), 0\} \quad (6)$$

```

% step 4
y_4 = max(y_3,0);

```

由于前4步的做法较为固定，我们可以首先将前4步变换的图像绘制出来进行调参。实验结果表明，使用 hanning 窗、窗宽为数据点长度的1/100时，可以取得最好效果。



5. 自动选峰

首先观察 y_4 图像可知，除了极少数毛刺外，图像中的峰值点基本对应节奏的起始点。接下来需要解决两个问题：

- 如何去除毛刺的影响
- 如何筛选得到峰值

针对第一个问题，我先手动从MATLAB绘制的图像中，手动分辨出毛刺点和真正的节奏起始点，发现：毛刺点的峰值大多在0.001以下，而真正的节奏起始点峰值大多在0.001以上。所以，我们可以将 y_4 的数据点手动减去一个固定的常量，再经过半波整流以消除毛刺得到 y_{4minus} （此处半波整流的思想很像深度学习中的 ReLU 函数）。

筛选峰值的一个关键是寻找 y_4 中冲激所在的区间。我们可以用以下这种很简单的方法锚定每一个区间：

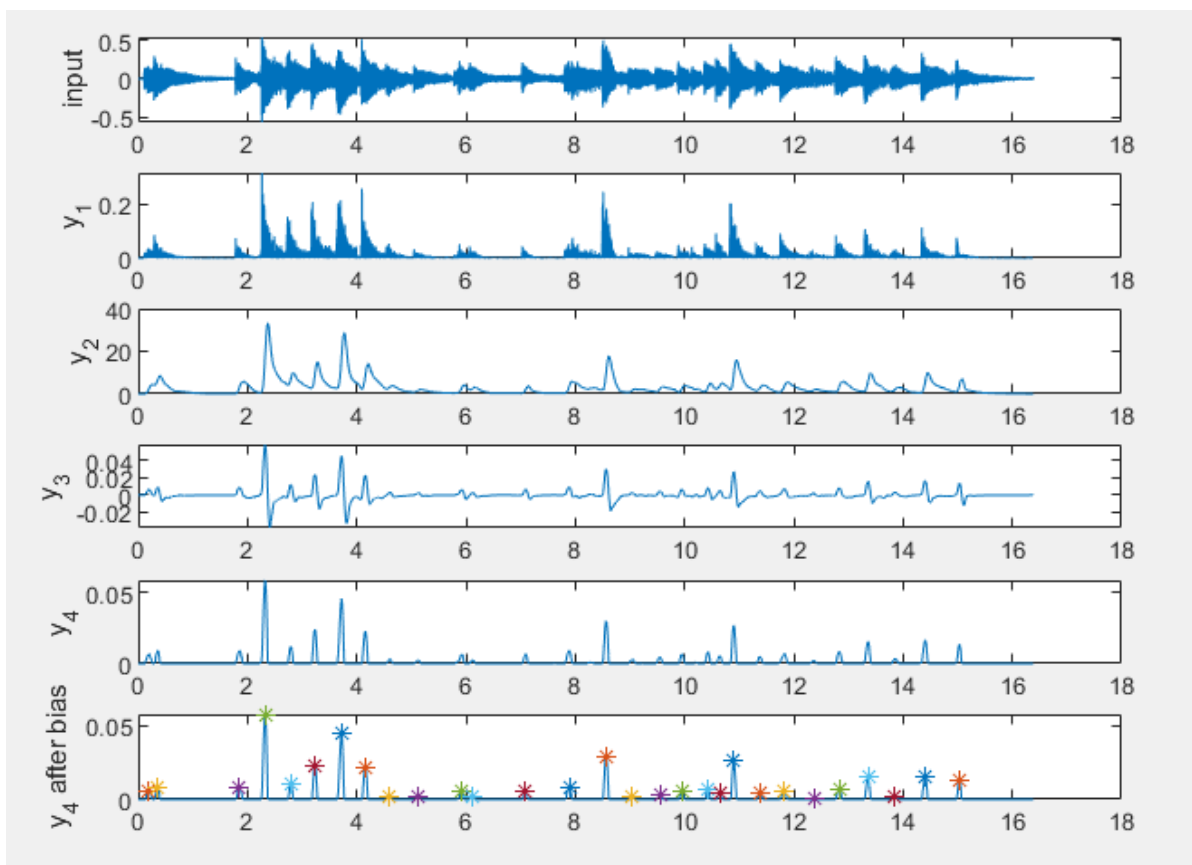
- 1). 构造一个数组 `interval_array`，第一行和第二行分别记录区间的起始点和结束点。
- 2). 初始 `index` 为1，若经过整流后的 `y_4[i] == 0` 且 `y_4[i+1] > 0`，则可以认为 `i` 为区间的起始点，将其放入 `interval_array(index, 1)` 中；若 `y_4[j-1] > 0` 且 `y_4[j] == 0`，则可以认为 `j` 为区间的结束点，将其放入 `interval_array(index, 2)` 中，之后 `index` 自增1。这样就可以做到每一个起始点和每一个结束点严格匹配。
- 3). 在每一个小区间内筛选最大值及其下标即可。

此处有两点需要补充说明：

1. 由于 `y_4` 经过偏置和整流，所以 `y_4[i] == 0` 且 `y_4[i+1] > 0` 这一条件足够精确，不必担心毛刺的影响。
2. 若出现起始点和终止点未能匹配的情况，程序将自动舍弃这一区间（但这种情况只发生在 `y_4[0] != 0` 或 `y_4[end] != 0` 的时）

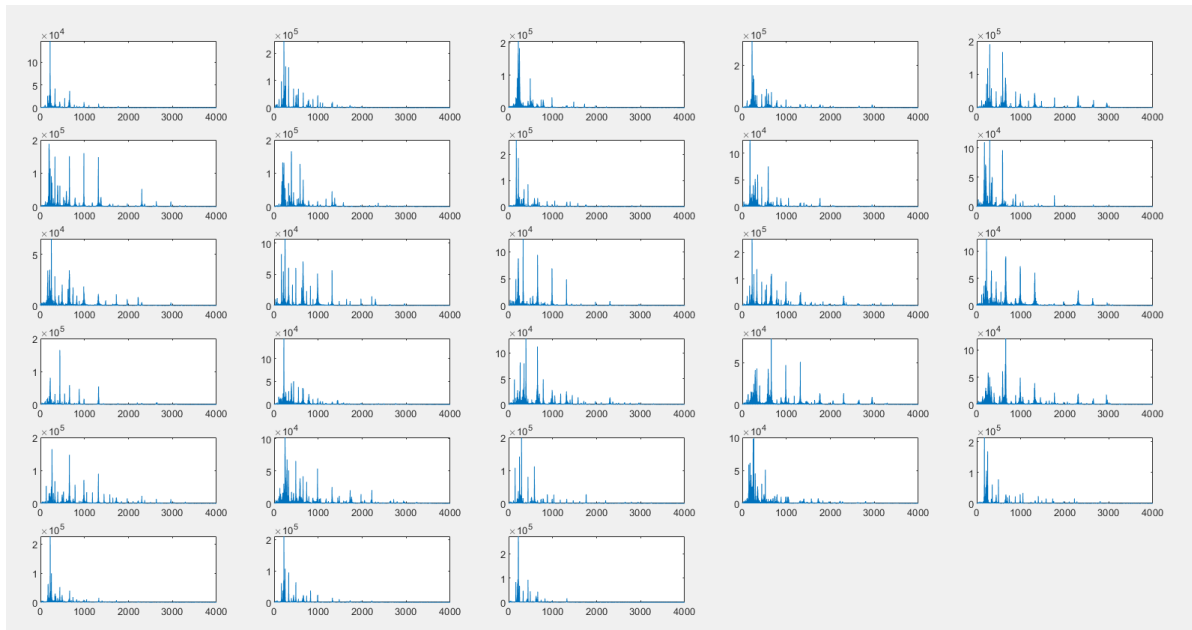
```
% step 5
interval_array = [];
interval_array(1,1) = 0;
peak_idx = [];
index = 1;
for i = 1:length(y_4_minus)-1
    if(y_4_minus(i) == 0 && y_4_minus(i+1) > 0)
        interval_array(index, 1) = i;
    end
    if(y_4_minus(i) > 0 && y_4_minus(i+1) == 0)
        interval_array(index, 2) = i + 1;
        [~, idx] = max(y_4_minus(interval_array(index, 1):interval_array(index, 2)));
        if(interval_array(index, 1) ~= 0)
            idx = idx + interval_array(index, 1);
            peak_idx = [peak_idx, idx];
        end
        index = index + 1;
    end
end
```

效果如下图所示：



可以看到，程序确实正确地找到了峰值点。经过统计，本段音乐含有29个峰值点。

为了提取出更加丰富的信息，我们可以在频域上对音乐进行分析。借鉴1.2.2.8中的思想，我们可以将每一个音乐片段重复若干次（此处为100次）后再进行fft，得到更加精确的图像：



通过观察可以初步得知：

1. 频谱中强度最大的频点为基频的整数倍，且一般较为靠近基频。
2. 非最大值频点的基波分量相比于最大值频点一般不会太弱。
3. 高频频点普遍较弱。

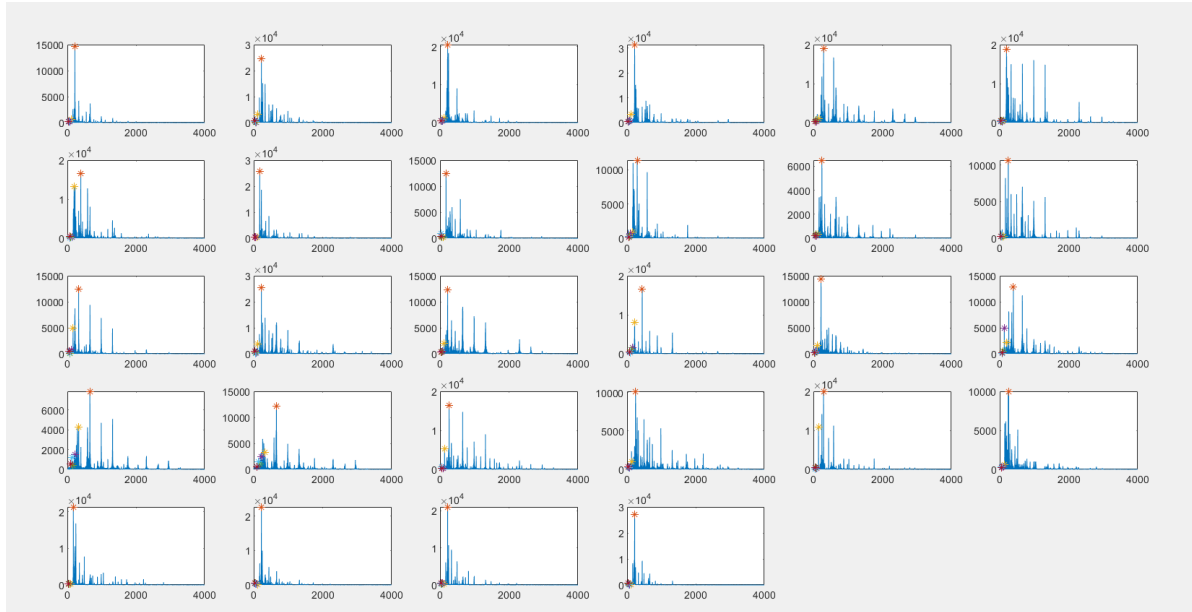
对此，我们可以构思出以下的搜索方案：

1. 选出最大频点 f_{max} 。

2. 指定搜索区间长度参数 $interval$, 在区间

$[index(\frac{f_{max}}{n}) - interval, index(\frac{f_{max}}{n}) + interval]$, $(0 < n < N, n \in \mathbb{Z})$ 中搜索最大值及其对应的下标, 即为可能的基频及其幅度。其中, $interval$ 作为超参可以调整, N 一般不需要取太大, 此处取5。

但这样会带来一个问题: 一般来说, 基频的幅度不会太弱, 我们如何确定基频的阈值? 对此, 我们可以先进行绘图, 绘制出每一个可能的基频频点所在的位置:



由上图可以初步判断, 绝大多数音乐片段的强度最大的频点就是基频, 只有极少数音乐片段的基频为强度最大频点的 $\frac{1}{3}$ 或 $\frac{1}{2}$, 且其强度一般超过最大强度的 $\frac{1}{5}$ 。小于此值的, 我们可以认为并不是基频, 而是受噪声干扰所致。于是我们规定: 基频强度需要大于最大基频强度的 $\frac{1}{5}$ (当然, 根据上图的结果, 我们可以大胆地取 $N = 3$)。

这样就得到了完整的搜索代码:

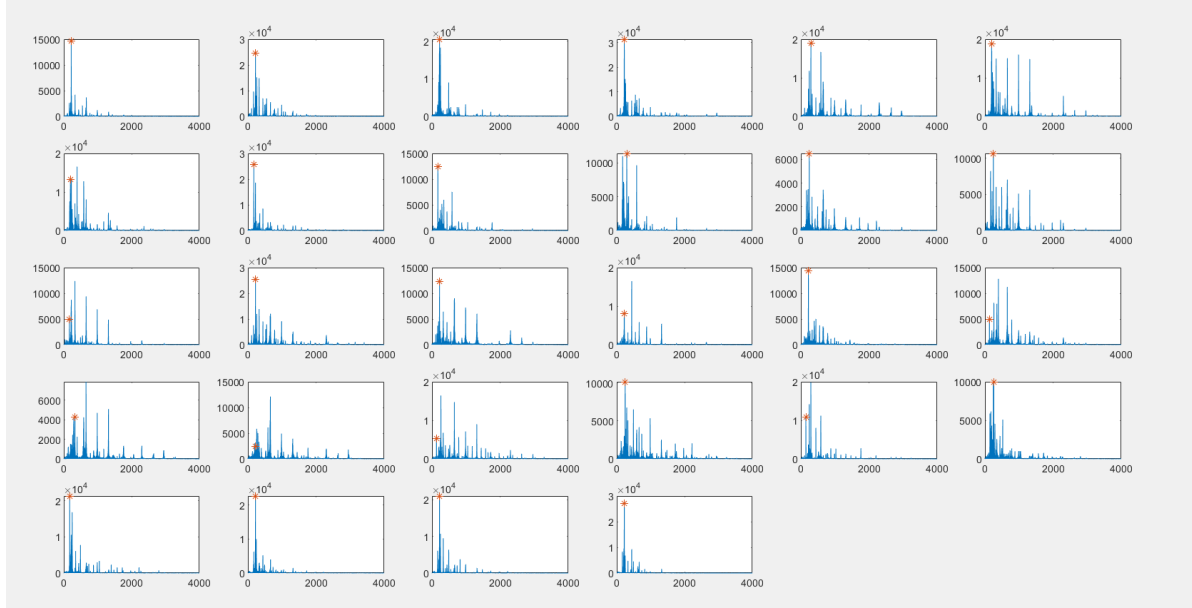
```
% 29 max vals
figure(2);
interval_len = 200;
for i = 1:1:28
    subplot(5,6,i);
    idx_1 = pos(i);
    idx_2 = pos(i+1);
    data_repeat = repmat(data(idx_1:idx_2), [100, 1]);
    [f,p] = generate_fft(data_repeat, freq);
    [max_p, idx] = max(p); % max value
    max_f = f(idx); % max freq
    plot(f,p);
    max_p_base_final = 0;
    idx_base_final = 0;
    idx_final = 0;
    for n = 1:6
        pos_2 = round(idx/n);
        [max_p_base, idx_base] = max(p(pos_2-interval_len:pos_2+interval_len));
        if(max_p_base >= 0.2*max_p)
            max_p_base_final = max_p_base;
            idx_base_final = idx_base;
            idx_final = pos_2-interval_len+idx_base;
        end
    end
end
```

```

end
hold on;
plot(f(idx_final), max_p_base_final, '*');
disp([f(idx_final), max_p_base_final]);
end

```

进一步搜索后结果如下：

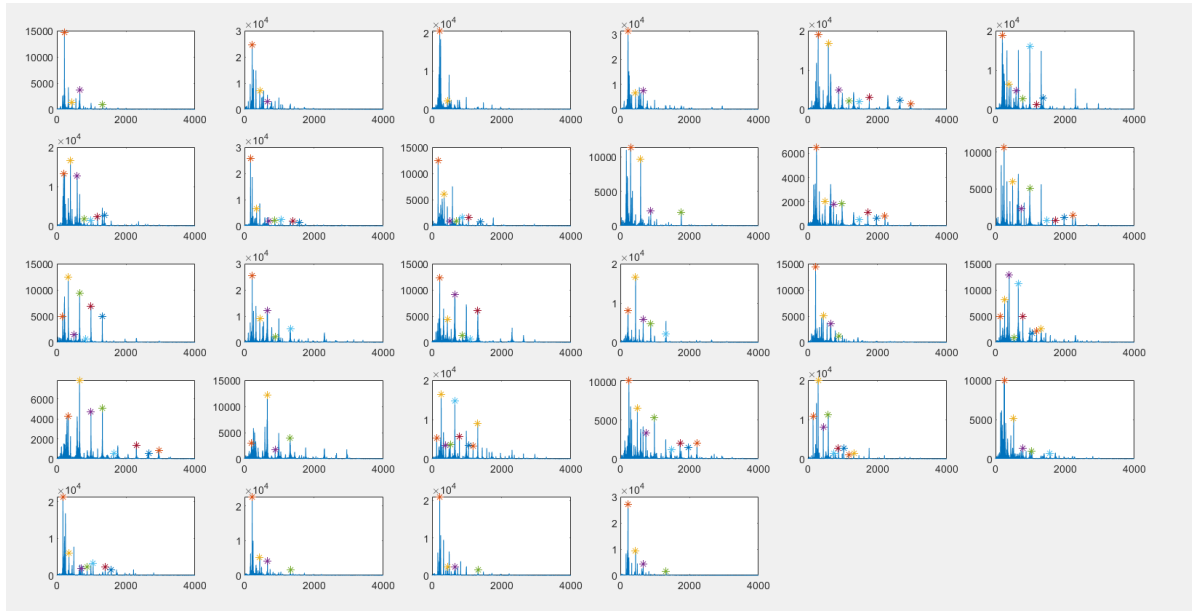


大致观察可以看出，我们基本正确搜索到了基频的位置，但也有例外，例如第17张图。推测此图未能正确搜索到基频的原因是受到噪音干扰较为严重。将所有基频（单位：Hz）排列如下：

131.0320	131.5106	146.8929	165.1932	173.5934	174.1377	175.0914	194.4676
196.2391	215.6198	219.1549	220.7839	221.0627	221.3405	221.4746	221.6044
221.7650	221.9033	222.1051	222.1548	222.6798	246.7975	247.0573	247.2737
261.5434	294.8908	295.1097	330.5106				

在此基频的基础上可以寻找谐波分量。设基频为 f_0 ，指定搜索区间长度参数 $interval$ ，在区间 $[index(nf_{max}) - interval, index(nf_{max}) + interval]$, $(0 < n < N, n \in \mathbb{Z})$ 内搜索最大值，即为谐波分量。同时我们规定，谐波分量若小于基频分量的 $\frac{1}{20}$ ，则可以忽略。此处可以取 $N = 15$ 。

搜索谐波后的结果如下：



在搜索的同时，我们会将谐波归一化之后的能量值（即以基波分量为1）保存起来：

1	2	3	4	5	6	7	8	9	10
1	0.089783	0.25008	0	0	0.059755	0	0	0	0
1	0.069983	0	0	0	0	0	0	0	0
1	0.10227	0	0	0	0	0	0	0	0
1	0.20411	0.23246	0	0	0	0	0	0	0
1	0.878	0.25346	0.11127	0.10593	0.15894	0	0	0.12033	0.076877
1	0.33779	0.24791	0.14314	0.85085	0.064427	0.15018	0	0	0
1	1.2519	0.96542	0.13752	0	0	0	0	0	0
1	0.25966	0	0.068627	0.079852	0.088492	0	0.075817	0.052002	0
1	0.47963	0.068977	0.073548	0.12551	0	0	0	0	0
1	0.78282	0	0	0	0	0	0	0	0
1	0.31236	0.27168	0.28552	0	0.07653	0.16899	0.0948	0.12594	0
1	0.56312	0.15363	0.41242	0	0	0	0	0	0
1	2.5139	0.29192	1.9052	0.13043	0.5834	0	0.13044	0	0
1	0.35503	0.47404	0	0	0.054892	0	0	0	0
1	0.35812	0.73619	0.097983	0	0.20416	0	0	0	0
1	2.0427	0.27124	0.11764	0	0	0	0	0	0
1	0.35112	0.22993	0.081802	0	0	0	0	0	0
1	1.6746	2.6323	0.19528	2.3096	1.002	0	0.35685	0.42808	0.51896
1	1.8337	1.0939	1.1833	0.12362	0	0.31193	0.12977	0	0
1	0	4.8934	0	0	0.56024	0	0	0	0
1	3.1081	0.66568	0.67343	2.7848	1.063	0	0.64384	0.62771	1.6996
1	0.64172	0.32575	0.52687	0	0.11382	0.1982	0.13947	0.20192	0
1	1.8426	0.74483	1.0389	0.09915	0.25355	0.2446	0.10166	0.12259	0
1	0.51389	0.13623	0.098714	0	0.072065	0	0	0	0
1	0.28612	0	0.091263	0.10529	0.15699	0	0.1092	0	0
1	0.23052	0.17642	0	0	0	0	0	0	0
1	0.1093	0.10755	0	0	0	0	0	0	0
1	0.34067	0.086018	0	0	0	0	0	0	0

当然，这些基频不能直接用于计算。为了将特征规范化，我们需要使用“十二平均律”生成的音调表格，在其中寻找出与上述数据中最相近的值。这一代码在：

```
base_ratio = 2^(1/12);
search_table = [];
for i = 0:48
    search_table = [search_table, 110*base_ratio^i];
end
disp(search_table - raw_freq);
diff_abs = abs(search_table - raw_freq);
[~, idx] = min(diff_abs);
standard_freq = search_table(idx);
```

规范化之后的基频（单位：Hz）排列如下：

```
130.8128 130.8128 146.8324 164.8138 174.6141 174.6141 174.6141 195.9977
195.9977 220.0000 220.0000 220.0000 220.0000 220.0000 220.0000 220.0000
220.0000 220.0000 220.0000 220.0000 220.0000 246.9417 246.9417 246.9417
261.6256 293.6648 293.6648 329.6276
```

将上文中计算得到的基频映射为标准频率，并将这些基频所对应的基频及各次谐波分量进行平均：

代码如下：

```
standard_freq_list = [];
for base_freq = base_freq_list
    standard_freq_list = [standard_freq_list,
search_in_standard_table(base_freq)];
end

disp(standard_freq_list);

averaged_harmonic_wave_data = containers.Map;
averaged_harmonic_wave_data_num = containers.Map;
count = 1;
for standard_freq = standard_freq_list
    struct_idx = num2str(standard_freq);
    if(isfield(averaged_harmonic_wave_data, struct_idx))
        averaged_harmonic_wave_data(struct_idx) =
averaged_harmonic_wave_data(struct_idx) + harmonic_wave_weight(count,:);
        averaged_harmonic_wave_data_num(struct_idx) =
averaged_harmonic_wave_data_num(struct_idx) + 1;
    else
        averaged_harmonic_wave_data(struct_idx) = harmonic_wave_weight(count,:);
        averaged_harmonic_wave_data_num(struct_idx) = 1;
    end
    count = count + 1;
end

averaged_harmonic_wave_data_result = [];
for key = averaged_harmonic_wave_data.keys
    disp([str2double(key{1}), averaged_harmonic_wave_data(key{1}) /
averaged_harmonic_wave_data_num(key{1})]);
    averaged_harmonic_wave_data_result = [averaged_harmonic_wave_data_result;
[str2double(key{1}), averaged_harmonic_wave_data(key{1}) /
averaged_harmonic_wave_data_num(key{1})]];
end

csvwrite('averaged_harmonic_wave_data_result.csv',
averaged_harmonic_wave_data_result);
```

可以得到如下表格：

频率(Hz)	1	2	3	4	5	6	7	8	9	10
130.81	1	3.1081	0.66568	0.67343	2.7848	1.063	0	0.64384	0.62771	1.6996
146.83	1	1.8426	0.74483	1.0389	0.09915	0.25355	0.2446	0.10166	0.12259	0
164.81	1	2.5139	0.29192	1.9052	0.13043	0.5834	0	0.13044	0	0
174.61	1	0.28612	0	0.091263	0.10529	0.15699	0	0.1092	0	0

频率(Hz)	1	2	3	4	5	6	7	8	9	10
196	1	1.2519	0.96542	0.13752	0	0	0	0	0	0
220	1	0.34067	0.086018	0	0	0	0	0	0	0
246.94	1	0.64172	0.32575	0.52687	0	0.11382	0.1982	0.13947	0.20192	0
261.63	1	0.51389	0.13623	0.098714	0	0.072065	0	0	0	0
293.66	1	0.78282	0	0	0	0	0	0	0	0
329.63	1	1.8337	1.0939	1.1833	0.12362	0	0.31193	0.12977	0	0

这样，我们就得到了每一个音符的各次谐波分量。为了便于复用，我们将其封装为一个函数 `get_harmonic.m`。

1.2.3 基于傅里叶级数的合成音乐

1.2.3.10 基础傅里叶级数合成音乐

核心代码及思想

本题主代码位于 `hw_1_2_3_10.m` 内，辅助函数为 `generate_music.m`。

我们对函数 `generate_music.m` 做适当改写，以使其可以传入谐波权重向量：

```
count = 1;
for lambda = lambdas
    single_song_data = lambda * sin(2 * pi * count * single_freq .* t');
    count = count + 1;
end
```

约定谐波权重向量为：

```
lambda = [1, 1.4572, 0.9587, 1.0999, 0.0523, 0.1099, 0.3589, 0.1240, 0.1351,
0.0643, 0.0019];
```

将音乐合成后进行播放，可以发现音色确实与吉他相似，但总体效果不是特别好。

1.2.3.11 利用每个音调对应的傅里叶级数合成音乐

首先我们打印出《东方红》一曲中的频率列表：

```
523.2511 523.2511 587.3295 391.9954 349.2282 349.2282 293.6648 391.9954
```

与1.2.2.9中生成的谐波分量列表相比，有两个存在的问题：

1. 《东方红》的音调较高
2. 即使将东方红的音调调低，其音调也未必可以与列表一一对应。

对此，提出两个解决方案：

1. 将《东方红》降调播放，使其音调范围能和谐波分量列表的范围相对应。

将《东方红》的频率变为之前的 $\frac{1}{\sqrt{2}}$ ，频率列表如下：

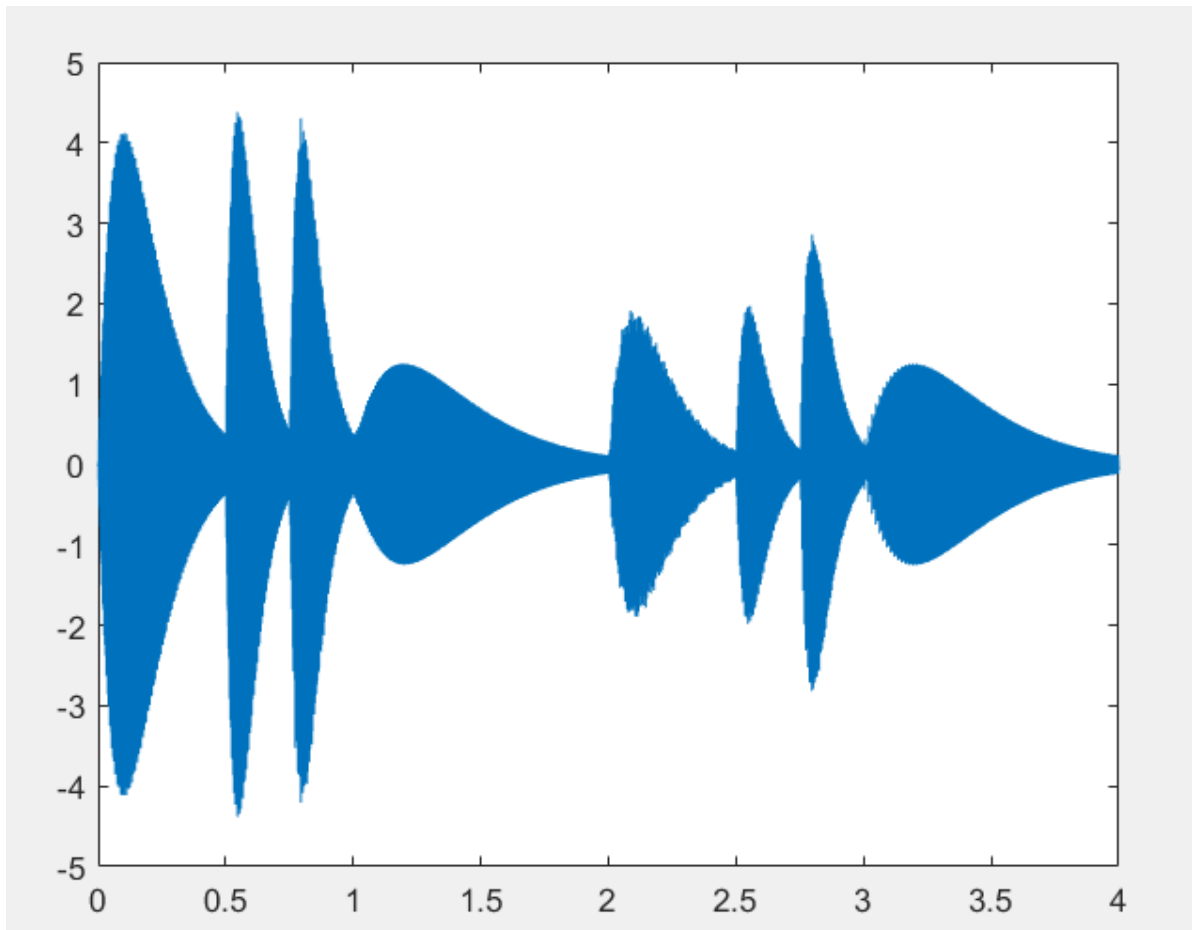
```
261.6256 261.6256 293.6648 195.9977 174.6141 174.6141 146.8324 195.9977
```

可以看到其范围基本与谐波列表相对应。

2. 若在列表中找不到相应的音调，则选择谐波分量列表中权重相对距离最近的一个作为谐波权重。

```
function harmonic_weight = search_nearest_tune(full_harmonic_data, freq)
    freq_list = full_harmonic_data(:,1);
    [~, idx] = min(abs(freq_list-freq));
    harmonic_weight = full_harmonic_data(idx,2:end);
end
```

合成后的音乐如下：



可以听出，音乐比之前更像吉他声。

1.2.3.12 图形界面

在正式进行图形界面的编写之前，我们首先需要定义一种规范的音乐元数据格式（metadata）。其应该包含以下内容：

- beat：一拍所对应的秒数
- major：音乐所用的调型
- freq：音乐的采样频率
- data：定义音乐的每一段音符及其持续时间
- lambda(optional)：定义各次谐波的权重

在经过思考之后，我认为可以用 .json 文件组织 metadata。一个示例

the_truth_that_you_leave.json 如下：


```
{
  "major": "B flat",
  "beat": 0.5,
  "freq": 8000,
  "lambda": [1, 0.2, 0.3],
  "data":
  [
    [2, 5, 0.5],
    [1, 6, 0.5],
    [2, 1, 0.5],
    // ...
  ]
}
```

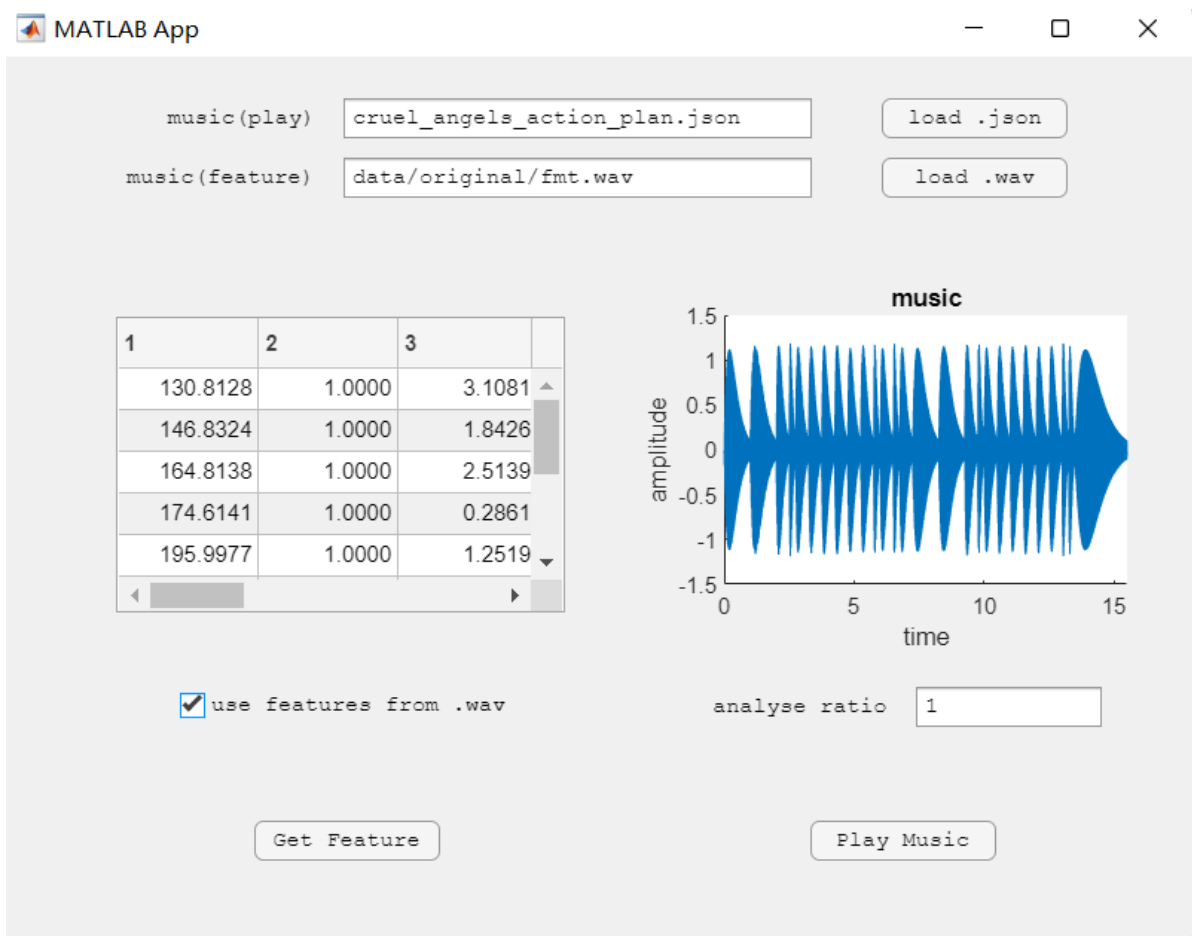
我们需要使用MATLAB中jsonlab库来解析json文件，其安装和使用方法可以参考³。

这样，图形界面大致就可以分为两个功能：

1. 根据音乐转译而来的 .json 文件播放音乐。
2. 解析一段 .wav 文件的谐波分量，这一谐波分量可以用于合成音乐。

在 additional/ 文件夹下有两个示例 .json 文件：the_truth_that_you_leave.json（你离开的事实）和 cruel_angels_action_plan.json（残酷天使的行动纲领）。

图形界面设计如下：



使用方法：

1. 在 music(play) 文本框中输入json文件的相对路径，然后点击 load .json 按键。若路径存在，将会显示success，否则显示fail。

2. 点击 `Play Music`，音乐将会被播放，并显示出波形。
3. 在 `music(feature)` 文本框中输入wav文件的相对路径，然后点击 `load .wav` 按键。若路径存在，将会显示success，否则显示fail。
4. 点击 `Play Music`，谐波特征将会被提取，并显示各频率谐波分量。
5. 选中 `use feature from .wav`，使用wav文件的谐波特征播放音乐。

总结


这次实验让我收获颇多。

首先“音乐合成”这一主题十分有趣，它不仅让我更加深入了解了乐理知识，也让我对音乐产生了更加浓厚的兴趣；其次，本次实验让我对MATLAB有了更加深层次的理解。

但这次实验暴露了我的很多问题：在解决问题的过程当中，需要用到很多启发式算法，而我对这些算法的掌握程度并不够；此外，由于我平时更喜欢使用Python，因此我花了不少时间习惯MATLAB的一些特殊语法，在尝试的过程中也走了不少弯路。

总而言之，感谢老师和助教的辛勤付出，让我深入领会了MATLAB和音乐的魅力！

参考文献

1. 谷源涛.信号与系统——MATLAB综合实验.北京:高等教育出版社 
2. <https://github.com/zhangzw16/Project-for-Signals-and-Systems-2021> 
3. <https://www.bilibili.com/read/cv5076013/> 