

**数字媒体技术**

**实验报告**

题 目 声音、图像数据显示与变换

学 院 计算机科学与技术

专 业 软件工程

学 号 1173710204

学 生 陈东鑫

任 课 教 师 刘绍辉

哈尔滨工业大学计算机科学与技术学院

2020.3

**实验二:声音图像数据的数学变换**

**注意：请按照大家阅读文献的格式进行撰写，确保文档格式的规范性！**

1. **实验内容或者文献情况介绍**

能对音频文件进行DFT，DCT和DWT变换

* + 1. 例如，读入音频文件，以1024长度对音频分窗处理，然后对其进行一维的DFT，DCT，DWT处理，然后画出原始音频，以及处理后音频的图形

能对图像进行二维的DFT，DCT，DWT变换

* + 1. 理解FFT的优势：用一维的DFT变换对图像块进行变换，分别显示其幅度图和相位图（注意如何可视化结果，例如，将系数区间归一化到[0,255]，或者系数取对数等）；用FFT变换来对图像块进行变换，看看其速度；
    2. 理解DCT变换的能量聚集特性，变换后，保留左上角k个系数后，再做逆DCT变换，恢复原始图像，比价原始图像与恢复图像的PSNR值和SSIM值(需要查阅PSNR和SSIM的公式并实现)
    3. 理解DWT变换的频率特性

1. **算法简介及其实现细节**

**编程环境：**

Windows 10

Python 3.7.0

**库版本：**

numpy:1.17.4

opencv2:4.2.0

matplotlib:3.0.2

**DFT**：

DFT(FFT)的作用:可以将信号从时域变换到频域，而且时域和频域都是离散的，通俗的说，可以求出一个信号由哪些正弦波叠加而成，求出的结果就是这些正弦波的幅度和相位

实现公式：

上式可转化为：

其逆变换的公式为：

可以转化为：

**DCT**:

离散余弦变换（DCT）是对实信号定义的一种变换，变换后在频域中得到的也是一个实信号，相比DFT而言,DCT可以减少一半以上的计算。DCT还有一个很重要的性质（能量集中特性）：大多书自然信号（声音、图像）的能量都集中在离散余弦变换后的低频部分，因而DCT在（声音、图像）数据压缩中得到了广泛的使用。由于DCT是从DFT推导出来的另一种变换，因此许多DFT的属性在DCT中仍然是保留下来的。

实现公式：

对经过DCT变换之后的数据再做一次DCT变换即可还原

**DWT**：

首先我们定义一些需要用到的信号及滤波器。

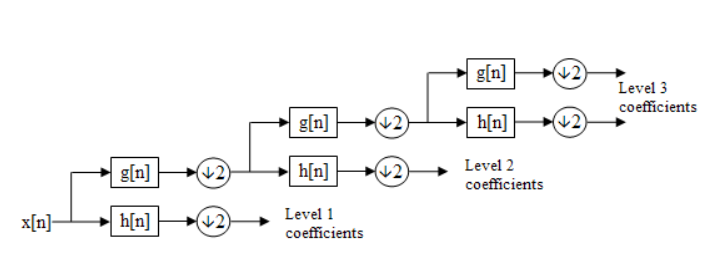
x[n]：离散的输入信号，长度为N。

g[n]：low pass filter低通滤波器，可以将输入信号的高频部份滤掉而输出低频部份。

h[n]：high pass filter高通滤波器，与低通滤波器相反，滤掉低频部份而输出高频部份。

Q：downsampling filter降采样滤波器，如果以x[n]作为输入，则输出y[n]=x[Qn]。此处举例Q=2。

清楚规定以上符号之后，便可以利用阶层架构来介绍如何将一个离散信号作离散小波变换



架构中的第α层

在数字图像处理中，需要将连续的小波及其小波变换离散化。一般计算机实现中使用二进制离散处理，将经过这种离散化的小波及其相应的小波变换成为离散小波变换（简称DWT）。实际上，离散小波变换是对连续小波变换的尺度、位移按照2的幂次进行离散化得到的，所以也称之为二进制小波变换。

虽然经典的傅里叶变换可以反映出信号的整体内涵，但表现形式往往不够直观，并且噪声会使得信号频谱复杂化。在信号处理领域一直都是使用一族带通滤波器将信号分解为不同频率分量，即将信号f(x)送到带通滤波器族Hi(x)中。

小波分解的意义就在于能够在不同尺度上对信号进行分解，而且对不同尺度的选择可以根据不同的目标来确定。

对于许多信号，低频成分相当重要，它常常蕴含着信号的特征，而高频成分则给出信号的细节或差别。人的话音如果去掉高频成分，听起来与以前可能不同，但仍能知道所说的内容；如果去掉足够的低频成分，则听到的是一些没有意义的声音。在小波分析中经常用到近似与细节。近似表示信号的高尺度，即低频信息；细节表示信号的高尺度，即高频信息。因此，原始信号通过两个相互滤波器产生两个信号。

通过不断的分解过程，将近似信号连续分解，就可以将信号分解成许多低分辨率成分。理论上分解可以无限制的进行下去，但事实上，分解可以进行到细节（高频）只包含单个样本为止。因此，在实际应用中，一般依据信号的特征或者合适的标准来选择适当的分解层数。

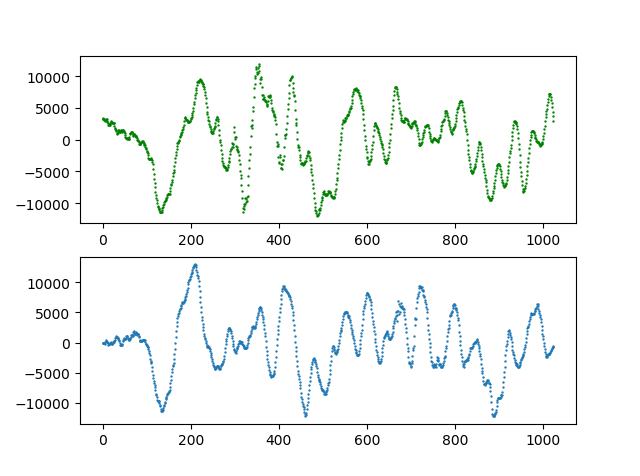
1. **实验设置及结果分析（包括实验数据集）**

**读入音频文件：**

定义一个Wave类，用来存储所读入的音频文件的相关信息，读入音频之后，将各信息打印，并显示每个声道前1024个数据的图像（之后所有的操作都是同时针对两个声道的前1024个数据进行，不再特别说明），相关代码如下：

1. **class** Wave:
2. **def** \_\_init\_\_(self, filePath):
3. self.audio = we.open(filePath, 'rb')
4. nchannels = self.audio.getnchannels()
5. sampwidth = self.audio.getsampwidth()
6. self.framerate = self.audio.getframerate()
7. self.nframes = self.audio.getnframes()
8. comptype = self.audio.getcomptype()
9. compname = self.audio.getcompname()
10. self.params = self.audio.getparams()
11. self.dataWav = self.audio.readframes(self.nframes)
12. self.secs = self.nframes / self.framerate
13. self.hammingWav = self.hamming()
14. self.audio.close()
16. **def** getInfo(self):
17. **print**(self.params)
19. **def** drawReg(self):
20. wave = np.fromstring(self.dataWav, dtype=np.short)
21. wave = wave.reshape((-1, 2))
22. wave = wave.T
24. x = np.arange(1024)
26. fig = plt.figure('原始图像前1024个数据')
27. plt.subplot(211)
28. plt.scatter(x, wave[0][:1024], color='green', s=0.5)
29. plt.subplot(212)
30. plt.scatter(x, wave[1][:1024], s=0.5)
31. fig.show()
33. filePath = 'wav/fav.wav'
34. wav = Wave(filePath)
35. wav.getInfo()
36. wav.drawReg()

展示原始数据



**以1024长度对音频分窗处理**：

首先进行分帧操作，每帧长度为1024，每帧起点的间隔为256,该操作由Wave类中的framing函数完成，代码如下：

1. **def** framing(self, wave\_data, wlen=1024, inc=256):
2. signal\_length = self.nframes
3. nf = int(np.ceil((1.0 \* signal\_length - wlen + inc) / inc))
4. pad\_length = int((nf - 1) \* inc + wlen)  #所有帧加起来总的铺平后的长度
5. zeros = np.zeros((pad\_length - signal\_length, ))
7. pad\_signal = np.concatenate((wave\_data, zeros))  #填补后的信号记为pad\_signal
8. tile\_0 = np.tile(np.arange(0, wlen), (nf, 1))
9. tile\_1 = np.tile(np.arange(0, nf \* inc, inc), (wlen, 1)).T
10. indices = tile\_0 + tile\_1
12. indices = np.array(indices, dtype=np.int32)  #将indices转化为矩阵
13. frames = pad\_signal[indices]  #得到帧信号
15. **return** frames

接下来进行加窗操作，使用的是余弦窗函数，即

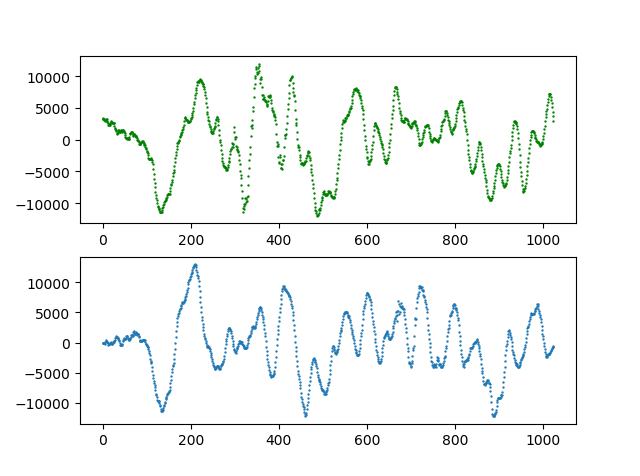
由于分帧时长度为1024，所以这里的M=1024，将每帧中的第n个数据乘以窗函数的第n项，得到加窗后的音频数据，即

同时画出其

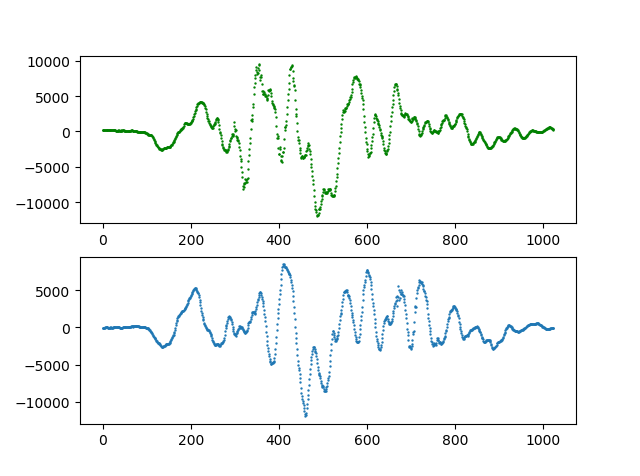
加窗操作的函数代码如下：

1. **def** hamming(self, wlen=1024, inc=256):
2. wave\_data = np.fromstring(self.dataWav, dtype=np.short)
3. wave\_data = wave\_data.reshape((-1, 2))
4. wave\_data = wave\_data.T
6. frame\_0 = self.framing(wave\_data[0].copy())
7. frame\_1 = self.framing(wave\_data[1].copy())
8. frames = np.r\_[frame\_0.copy(), frame\_1.copy()]
10. a = np.r\_[frame\_0[0].reshape((1, -1)), frame\_1[0].reshape((1, -1))]
11. x = np.array(range(wlen))
12. self.draw(a, x, '分帧之后两个声道的第一帧')
14. frames = (frames \* np.hamming(wlen)).reshape(2, -1)
16. self.draw(frames[:, :1024], x, '加窗之后两个声道的第一帧')
18. **return** frames

对原始数据进行分帧之后的图像（由于只看第一帧，所以图像与原始数据相同）：



加汉明窗之后的图像（可以看出，图像整体变小，越靠近边缘变小幅度越大）：

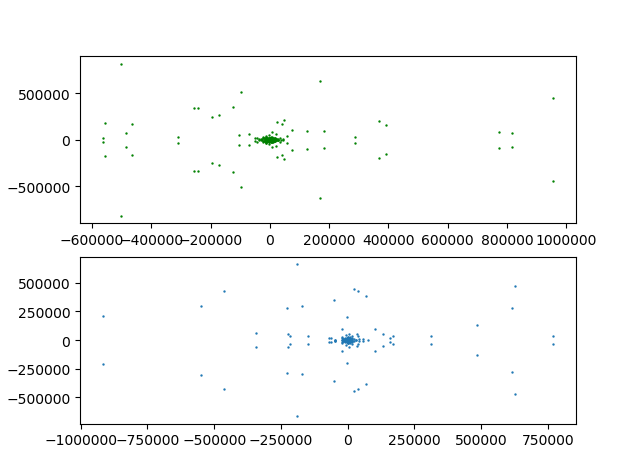


**一维DFT处理**：

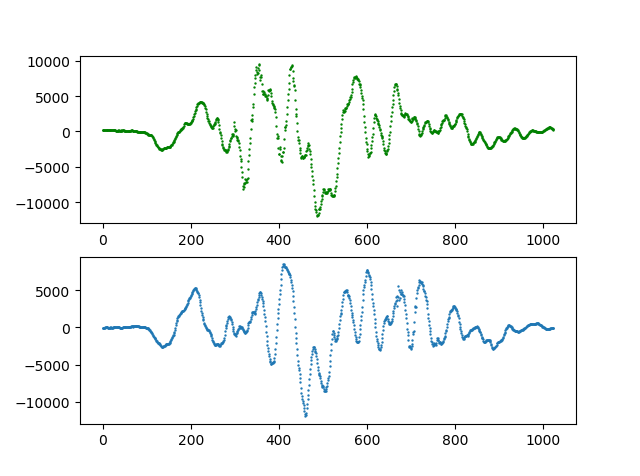
DFT的公式及原理此前已经说明，需要注意的是，由于计算机处理复数不方便，所以分实部和虚部分别计算，并且IDFT在计算之后，需要只保留实部，编写代码如下：

1. **def** mainDFT(self):
2. res\_little = self.hammingWav.copy()[:, :1024]
4. # DFT
5. res\_dft = self.dft(res\_little)
7. fig = plt.figure('经过DFT之后两个声道的第一帧')
8. plt.subplot(211)
9. plt.scatter(np.real(res\_dft)[0],
10. np.imag(res\_dft)[0],
11. color='green',
12. s=0.5)
13. plt.subplot(212)
14. plt.scatter(np.real(res\_dft)[1], np.imag(res\_dft)[1], s=0.5)
15. fig.show()
17. #IDFT
18. res\_idft = self.idft(res\_dft)
19. x = np.array(range(res\_idft.shape[1]))
20. self.draw(res\_idft, x, '经过IDFT之后两个声道的第一帧')
22. **def** dft(self, res):
23. temp = np.zeros\_like(res, dtype='complex')
24. row = temp.shape[0]
25. column = temp.shape[1]
26. **for** j **in** range(row):
27. **for** i **in** range(column):
28. exponent = 2 \* math.pi \* i / column
29. cos = [math.cos(n \* exponent) **for** n **in** range(column)]
30. sin = [math.sin(n \* exponent) \* (1j) **for** n **in** range(column)]
32. cos = np.array(cos).reshape((-1, 1))
33. sin = np.array(sin).reshape((-1, 1))
35. real = np.dot(res[j, :].reshape((1, -1)), cos)[0][0]
36. imag = np.dot(res[j, :].reshape((1, -1)), sin)[0][0]
38. temp[j][i] = real - imag
40. **return** temp.copy()
42. **def** idft(self, res):
43. temp = np.zeros\_like(res, dtype='complex')
44. column = temp.shape[1]
45. row = temp.shape[0]
47. **for** j **in** range(row):
48. **for** i **in** range(column):
49. exponent = 2 \* math.pi \* i / column
51. cos = [math.cos(k \* exponent) **for** k **in** range(column)]
52. sin = [math.sin(k \* exponent) \* (1j) **for** k **in** range(column)]
54. cos = np.array(cos).reshape((1, -1))
55. sin = np.array(sin).reshape((1, -1))
57. real = np.mean(res[j, :].reshape((1, -1)) \* cos)
58. imag = np.mean(res[j, :].reshape((1, -1)) \* sin)
60. temp[j][i] = real + imag
62. temp = np.real(temp)
64. **return** temp.copy()

DFT处理之后的图像如下：



IDFT处理之后的图像如下：

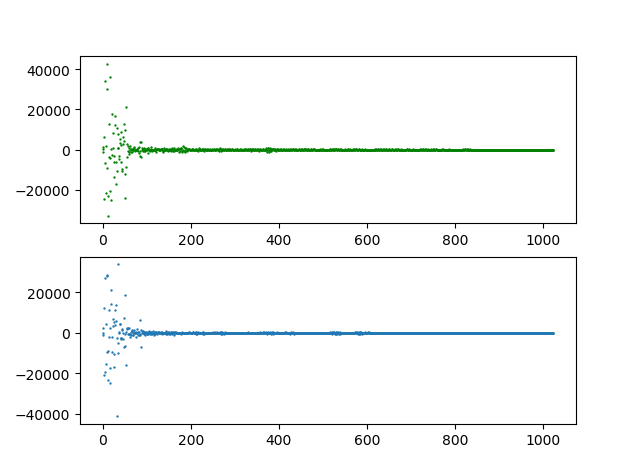


**一维DCT处理：**

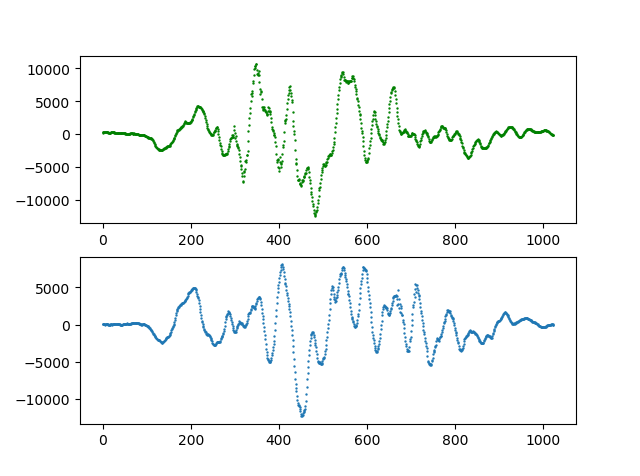
代码如下：

1. **def** mainDCT(self):
2. res\_little = self.hammingWav.copy()[:, :1024]
4. res\_dct = self.dct(res\_little)
5. x = np.array(range(res\_dct.shape[1]))
6. self.draw(res\_dct, x, '经过DCT之后两个声道的第一帧')
8. res\_idct = self.dct(res\_dct)
9. x = np.array(range(res\_dct.shape[1]))
10. self.draw(res\_idct, x, '经过IDCT之后两个声道的第一帧')
12. **def** dct(self, res):
13. column = res.shape[1]
14. row = res.shape[0]
16. c = np.sqrt(2 / column) \* np.ones((1, column))
17. c[0][0] = np.sqrt(1 / column)
19. cos = np.pi \* np.ones((column, column)) / column
20. column\_line = np.linspace(0.5, column - 0.5, column).reshape((-1, 1))
21. row\_line = np.array(range(column)).reshape((1, -1))
22. cos = cos \* column\_line \* row\_line
23. cos = np.cos(cos)
25. temp = np.dot(res, cos)
26. temp = temp \* c
28. **return** temp.copy()

经过DCT处理后的数据，相较于DFT处理之后的数据，没有虚部的数据，且大多分布在0周围：



经过IDCT处理之后的数据，对比IDFT处理之后的数据，有些许的损失：

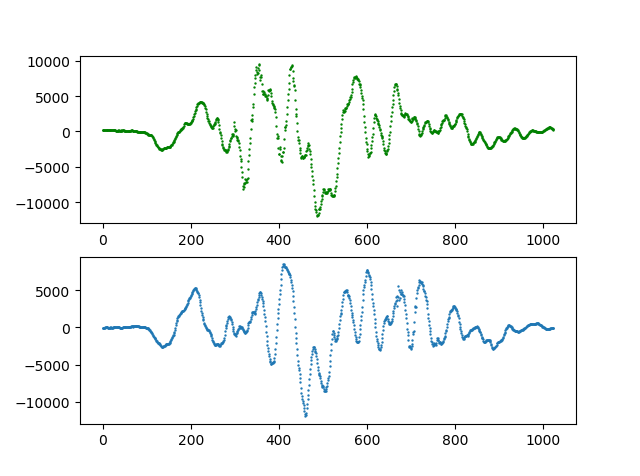


**一维DWT处理：**

代码如下:

1. **def** mainDWT(self):
2. data = self.hammingWav.copy()
4. cA0, cD0 = pywt.dwt(data[0], 'db2', 'smooth')
5. upcoef\_cA0 = pywt.upcoef('a', cA0, 'db2', take=data.shape[1])
6. upcoef\_cD0 = pywt.upcoef('d', cD0, 'db2', take=data.shape[1])
7. res0 = upcoef\_cA0 + upcoef\_cD0
9. cA1, cD1 = pywt.dwt(data[1], 'db2', 'smooth')
10. upcoef\_cA1 = pywt.upcoef('a', cA1, 'db2', take=data.shape[1])
11. upcoef\_cD1 = pywt.upcoef('d', cD1, 'db2', take=data.shape[1])
12. res1 = upcoef\_cA1 + upcoef\_cD1
14. res = np.r\_[res0, res1].reshape((2, -1))
16. self.draw(res[:, :1024], np.array(range(1024)), '经过DWT之后两个声道的第一帧')

处理之后的图像如下：



音频处理部分的完整代码如下：

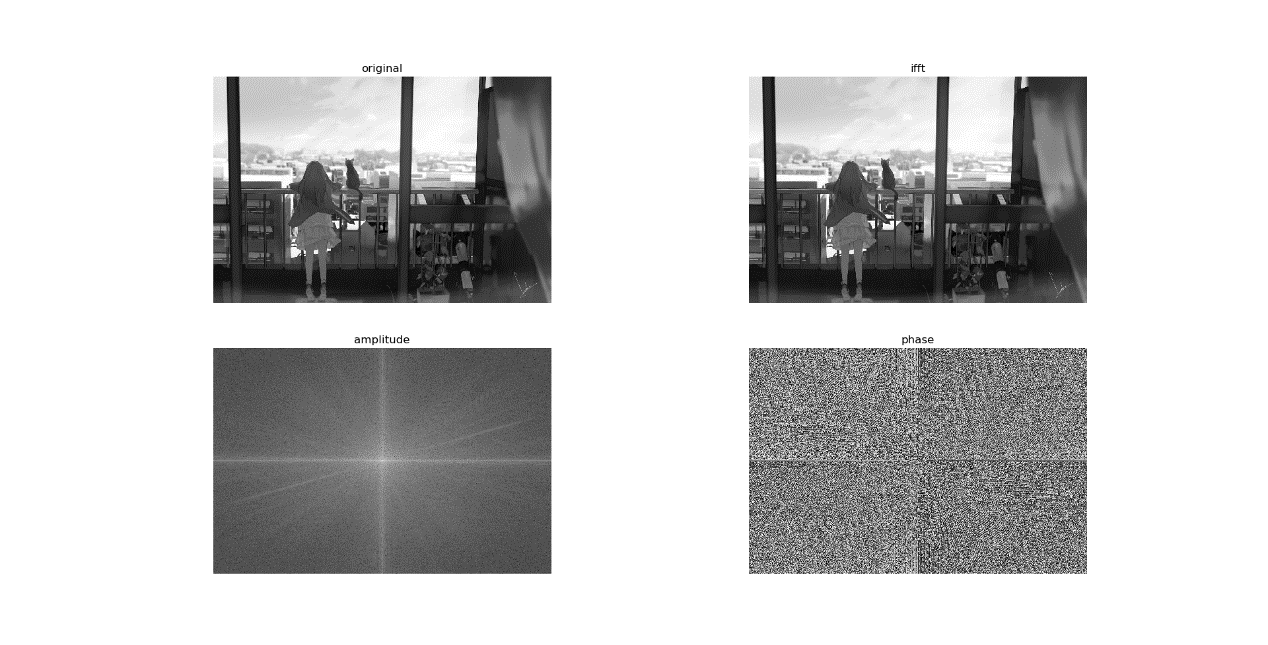
1. **import** wave as we
2. **import** matplotlib
3. matplotlib.use('Qt5Agg')
4. **import** matplotlib.pyplot as plt
5. **import** numpy as np
6. **import** cv2
7. **import** pywt
8. **import** math
10. **class** Wave:
11. **def** \_\_init\_\_(self, filePath):
12. self.audio = we.open(filePath, 'rb')
13. nchannels = self.audio.getnchannels()
14. sampwidth = self.audio.getsampwidth()
15. self.framerate = self.audio.getframerate()
16. self.nframes = self.audio.getnframes()
17. comptype = self.audio.getcomptype()
18. compname = self.audio.getcompname()
19. self.params = self.audio.getparams()
20. self.dataWav = self.audio.readframes(self.nframes)
21. self.secs = self.nframes / self.framerate
22. self.hammingWav = self.hamming()
23. self.audio.close()
25. **def** drawReg(self):
26. wave = np.fromstring(self.dataWav, dtype=np.short)
27. wave = wave.reshape((-1, 2))
28. wave = wave.T
30. x = np.arange(1024)
32. fig = plt.figure('原始图像前1024个数据')
33. plt.subplot(211)
34. plt.scatter(x, wave[0][:1024], color='green', s=0.5)
35. plt.subplot(212)
36. plt.scatter(x, wave[1][:1024], s=0.5)
37. fig.show()
39. **def** draw(self, wave, x, name):
40. fig = plt.figure(name)
41. plt.subplot(211)
42. plt.scatter(x, wave[0][:], color='green', s=0.5)
43. plt.subplot(212)
44. plt.scatter(x, wave[1][:], s=0.5)
45. fig.show()
47. **def** framing(self, wave\_data, wlen=1024, inc=256):
48. signal\_length = self.nframes
49. nf = int(np.ceil((1.0 \* signal\_length - wlen + inc) / inc))
50. pad\_length = int((nf - 1) \* inc + wlen)  #所有帧加起来总的铺平后的长度
51. zeros = np.zeros((pad\_length - signal\_length, ))
53. pad\_signal = np.concatenate((wave\_data, zeros))  #填补后的信号记为pad\_signal
54. tile\_0 = np.tile(np.arange(0, wlen), (nf, 1))
55. tile\_1 = np.tile(np.arange(0, nf \* inc, inc), (wlen, 1)).T
56. indices = tile\_0 + tile\_1
58. indices = np.array(indices, dtype=np.int32)  #将indices转化为矩阵
59. frames = pad\_signal[indices]  #得到帧信号
61. **return** frames
63. **def** hamming(self, wlen=1024, inc=256):
64. wave\_data = np.fromstring(self.dataWav, dtype=np.short)
65. wave\_data = wave\_data.reshape((-1, 2))
66. wave\_data = wave\_data.T
68. frame\_0 = self.framing(wave\_data[0].copy())
69. frame\_1 = self.framing(wave\_data[1].copy())
70. frames = np.r\_[frame\_0.copy(), frame\_1.copy()]
72. a = np.r\_[frame\_0[0].reshape((1, -1)), frame\_1[0].reshape((1, -1))]
73. x = np.array(range(wlen))
74. self.draw(a, x, '分帧之后两个声道的第一帧')
76. frames = (frames \* np.hamming(wlen)).reshape(2, -1)
78. self.draw(frames[:, :1024], x, '加窗之后两个声道的第一帧')
80. **return** frames
82. **def** mainDFT(self):
83. res\_little = self.hammingWav.copy()[:, :1024]
85. # DFT
86. res\_dft = self.dft(res\_little)
88. fig = plt.figure('经过DFT之后两个声道的第一帧')
89. plt.subplot(211)
90. plt.scatter(np.real(res\_dft)[0],
91. np.imag(res\_dft)[0],
92. color='green',
93. s=0.5)
94. plt.subplot(212)
95. plt.scatter(np.real(res\_dft)[1], np.imag(res\_dft)[1], s=0.5)
96. fig.show()
98. #IDFT
99. res\_idft = self.idft(res\_dft)
100. x = np.array(range(res\_idft.shape[1]))
101. self.draw(res\_idft, x, '经过IDFT之后两个声道的第一帧')
103. **def** dft(self, res):
104. temp = np.zeros\_like(res, dtype='complex')
105. row = temp.shape[0]
106. column = temp.shape[1]
107. **for** j **in** range(row):
108. **for** i **in** range(column):
109. exponent = 2 \* math.pi \* i / column
110. cos = [math.cos(n \* exponent) **for** n **in** range(column)]
111. sin = [math.sin(n \* exponent) \* (1j) **for** n **in** range(column)]
113. cos = np.array(cos).reshape((-1, 1))
114. sin = np.array(sin).reshape((-1, 1))
116. real = np.dot(res[j, :].reshape((1, -1)), cos)[0][0]
117. imag = np.dot(res[j, :].reshape((1, -1)), sin)[0][0]
119. temp[j][i] = real - imag
121. **return** temp.copy()
123. **def** idft(self, res):
124. temp = np.zeros\_like(res, dtype='complex')
125. column = temp.shape[1]
126. row = temp.shape[0]
128. **for** j **in** range(row):
129. **for** i **in** range(column):
130. exponent = 2 \* math.pi \* i / column
132. cos = [math.cos(k \* exponent) **for** k **in** range(column)]
133. sin = [math.sin(k \* exponent) \* (1j) **for** k **in** range(column)]
135. cos = np.array(cos).reshape((1, -1))
136. sin = np.array(sin).reshape((1, -1))
138. real = np.mean(res[j, :].reshape((1, -1)) \* cos)
139. imag = np.mean(res[j, :].reshape((1, -1)) \* sin)
141. temp[j][i] = real + imag
143. temp = np.real(temp)
145. **return** temp.copy()
147. **def** mainDCT(self):
148. res\_little = self.hammingWav.copy()[:, :1024]
150. res\_dct = self.dct(res\_little)
151. x = np.array(range(res\_dct.shape[1]))
152. self.draw(res\_dct, x, '经过DCT之后两个声道的第一帧')
154. res\_idct = self.dct(res\_dct)
155. x = np.array(range(res\_dct.shape[1]))
156. self.draw(res\_idct, x, '经过IDCT之后两个声道的第一帧')
158. **def** dct(self, res):
159. column = res.shape[1]
160. row = res.shape[0]
162. c = np.sqrt(2 / column) \* np.ones((1, column))
163. c[0][0] = np.sqrt(1 / column)
165. cos = np.pi \* np.ones((column, column)) / column
166. column\_line = np.linspace(0.5, column - 0.5, column).reshape((-1, 1))
167. row\_line = np.array(range(column)).reshape((1, -1))
168. cos = cos \* column\_line \* row\_line
169. cos = np.cos(cos)
171. temp = np.dot(res, cos)
172. temp = temp \* c
174. **return** temp.copy()
176. **def** mainDWT(self):
177. data = self.hammingWav.copy()
179. cA0, cD0 = pywt.dwt(data[0], 'db2', 'smooth')
180. upcoef\_cA0 = pywt.upcoef('a', cA0, 'db2', take=data.shape[1])
181. upcoef\_cD0 = pywt.upcoef('d', cD0, 'db2', take=data.shape[1])
182. res0 = upcoef\_cA0 + upcoef\_cD0
184. cA1, cD1 = pywt.dwt(data[1], 'db2', 'smooth')
185. upcoef\_cA1 = pywt.upcoef('a', cA1, 'db2', take=data.shape[1])
186. upcoef\_cD1 = pywt.upcoef('d', cD1, 'db2', take=data.shape[1])
187. res1 = upcoef\_cA1 + upcoef\_cD1
189. res = np.r\_[res0, res1].reshape((2, -1))
191. self.draw(res[:, :1024], np.array(range(1024)), '经过DWT之后两个声道的第一帧')
193. **def** getInfo(self):
194. **print**(self.params)
196. filePath = 'wav/fav.wav'
197. wav = Wave(filePath)
199. wav.getInfo()
200. wav.drawReg()
201. wav.mainDFT()
202. wav.mainDCT()
203. wav.mainDWT()

**图像的二维DFT变换**：

代码如下：

1. **def** DFT(self):
2. #快速傅里叶变换算法得到频率分布
3. start = time.time()
4. f = np.fft.fft2(self.img\_gray.copy())
5. end = time.time()
6. **print**('图像大小为:(%d,%d)' %
7. (self.img\_gray.shape[0], self.img\_gray.shape[1]))
8. **print**('FFT耗时为:%dms' % ((end - start) \* 1000))
10. ifft = np.real(np.fft.ifft2(f))
12. #默认结果中心点位置是在左上角,
13. #调用fftshift()函数转移到中间位置
14. fshift = np.fft.fftshift(f)
15. log = np.log(fshift)
17. real = np.real(log)
18. imag = np.imag(log)
20. amplitude = real / np.max(real)
21. phase = imag / np.max(imag)
23. #展示结果
24. #原图，复原图，幅度，相位
25. fig = plt.figure('FFT')
26. plt.subplot(221), plt.imshow(self.img\_gray,
27. 'gray'), plt.title('original')
28. plt.axis('off')
29. plt.subplot(222), plt.imshow(ifft,'gray'),plt.title('ifft')
30. plt.axis('off')
31. plt.subplot(223), plt.imshow(amplitude, 'gray'), plt.title('amplitude')
32. plt.axis('off')
33. plt.subplot(224), plt.imshow(phase, 'gray'), plt.title('phase')
34. plt.axis('off')
35. fig.show()

得到的图像如下：



程序输出了FFT的耗时，如下：

图像大小为:(1292,1929)

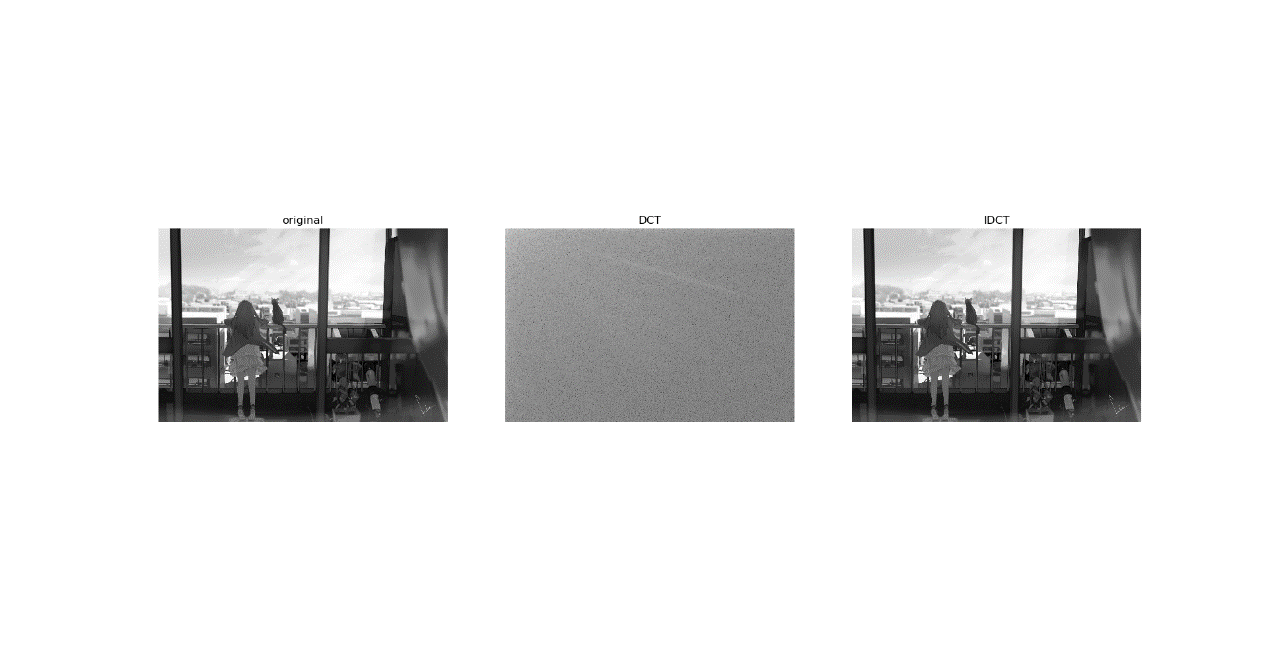
FFT耗时为:543ms

**图像二维DCT变换**：

包括其psnr与ssim计算，代码如下：

1. **def** DCT(self):
2. #         b = self.img[:][:][0]
3. #         g = self.img[:][:][1]
4. #         r = self.img[:][:][2]
5. res = np.float32(self.img\_gray.copy())
6. img\_dct = cv2.dct(res)
7. img\_idct = cv2.idct(img\_dct)
8. img\_dct = np.log(abs(img\_dct))
10. fig = plt.figure('DCT')
11. plt.subplot(131), plt.imshow(self.img\_gray,
12. 'gray'), plt.title('original')
13. plt.axis('off')
14. plt.subplot(132), plt.imshow(img\_dct, 'gray'), plt.title('DCT')
15. plt.axis('off')
16. plt.subplot(133), plt.imshow(img\_idct, 'gray'), plt.title('IDCT')
17. plt.axis('off')
18. fig.show()
20. PSNR = self.psnr(res.copy(), img\_idct.copy())
21. **print**('DCT与IDCT之后原始图像与恢复图像 PSNR =', PSNR)
23. SSIM = self.ssim(res.copy(), img\_idct.copy())
24. **print**('DCT与IDCT之后原始图像与恢复图像 SSIM =', SSIM)
26. **def** psnr(self, origin, target):
27. MSE = np.mean((origin - target)\*\*2)
28. MAX = 255
29. PSNR = 10 \* math.log((MAX\*\*2) / MSE, 10)
30. **return** PSNR
32. **def** ssim(self, origin, target):
33. mu\_x = np.mean(origin)
34. mu\_y = np.mean(target)
35. sigma\_x = np.std(origin)
36. sigma\_y = np.std(target)
37. sigma\_xy = np.cov(origin, target)
39. k1 = 0.01
40. k2 = 0.03
41. L = 255
43. c1 = (k1 \* L)\*\*2
44. c2 = (k2 \* L)\*\*2
46. up = (2 \* mu\_x \* mu\_y + c1) \* (2 \* sigma\_xy + c2)
47. down = (mu\_x\*\*2 + mu\_y\*\*2 + c1) \* (sigma\_x\*\*2 + sigma\_y\*\*2 + c2)
49. SSIM = np.mean(up / down)
51. **return** SSIM

所得的图像如下：



程序计算了其psnr和ssim值：

DCT与IDCT之后原始图像与恢复图像 PSNR = 131.8887089186267

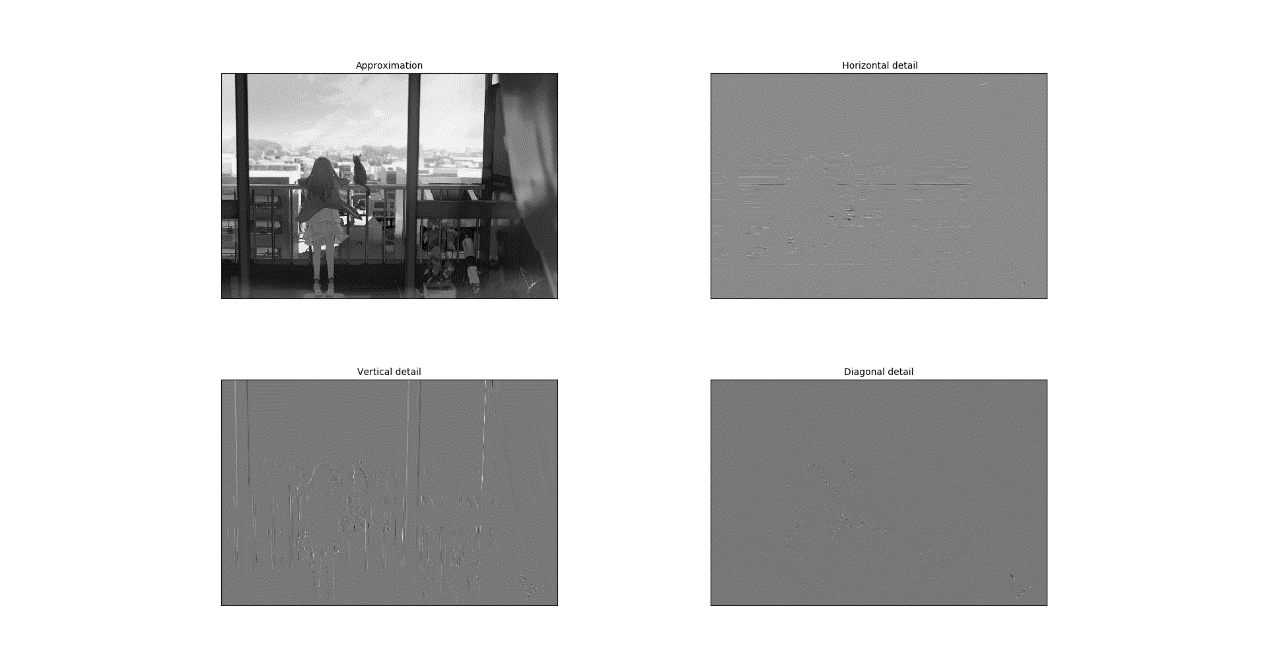
DCT与IDCT之后原始图像与恢复图像 SSIM = 0.17763278573476893

**图像二维DWT变换**：

代码如下：

1. **def** DWT(self):
2. res = self.img\_gray.copy()
4. titles = [
5. 'Approximation', ' Horizontal detail', 'Vertical detail',
6. 'Diagonal detail'
7. ]
8. coeffs2 = pywt.dwt2(res, 'bior1.3')
9. LL, (LH, HL, HH) = coeffs2
10. fig = plt.figure('DWT', figsize=(12, 3))
11. **for** i, a **in** enumerate([LL, LH, HL, HH]):
12. ax = fig.add\_subplot(2, 2, i + 1)
13. ax.imshow(a, interpolation="nearest", cmap=plt.cm.gray)
14. ax.set\_title(titles[i], fontsize=10)
15. ax.set\_xticks([])
16. ax.set\_yticks([])
18. fig.tight\_layout()
19. fig.show()

得到图像如下：



图像处理部分的完整代码如下：

1. **import** cv2
2. **import** numpy as np
3. **import** matplotlib
4. matplotlib.use('Qt5Agg')
5. **import** matplotlib.pyplot as plt
6. **import** time
7. **import** math
8. **import** pywt
10. **class** Bitmap:
11. **def** \_\_init\_\_(self, filePath):
12. #读取图像
13. self.img\_gray = cv2.imread(filePath, 0)
14. self.img = cv2.imread(filePath)
16. **def** psnr(self, origin, target):
17. MSE = np.mean((origin - target)\*\*2)
18. MAX = 255
19. PSNR = 10 \* math.log((MAX\*\*2) / MSE, 10)
20. **return** PSNR
22. **def** ssim(self, origin, target):
23. mu\_x = np.mean(origin)
24. mu\_y = np.mean(target)
25. sigma\_x = np.std(origin)
26. sigma\_y = np.std(target)
27. sigma\_xy = np.cov(origin, target)
29. k1 = 0.01
30. k2 = 0.03
31. L = 255
33. c1 = (k1 \* L)\*\*2
34. c2 = (k2 \* L)\*\*2
36. up = (2 \* mu\_x \* mu\_y + c1) \* (2 \* sigma\_xy + c2)
37. down = (mu\_x\*\*2 + mu\_y\*\*2 + c1) \* (sigma\_x\*\*2 + sigma\_y\*\*2 + c2)
39. SSIM = np.mean(up / down)
41. **return** SSIM
43. **def** DFT(self):
44. #快速傅里叶变换算法得到频率分布
45. start = time.time()
46. f = np.fft.fft2(self.img\_gray.copy())
47. end = time.time()
48. **print**('图像大小为:(%d,%d)' %
49. (self.img\_gray.shape[0], self.img\_gray.shape[1]))
50. **print**('FFT耗时为:%dms' % ((end - start) \* 1000))
52. ifft = np.real(np.fft.ifft2(f))
54. #默认结果中心点位置是在左上角,
55. #调用fftshift()函数转移到中间位置
56. fshift = np.fft.fftshift(f)
57. log = np.log(fshift)
59. real = np.real(log)
60. imag = np.imag(log)
62. amplitude = real / np.max(real)
63. phase = imag / np.max(imag)
65. #展示结果
66. #原图，复原图，幅度，相位
67. fig = plt.figure('FFT')
68. plt.subplot(221), plt.imshow(self.img\_gray,
69. 'gray'), plt.title('original')
70. plt.axis('off')
71. plt.subplot(222), plt.imshow(ifft,'gray'),plt.title('ifft')
72. plt.axis('off')
73. plt.subplot(223), plt.imshow(amplitude, 'gray'), plt.title('amplitude')
74. plt.axis('off')
75. plt.subplot(224), plt.imshow(phase, 'gray'), plt.title('phase')
76. plt.axis('off')
77. fig.show()
79. **def** DCT(self):
80. #         b = self.img[:][:][0]
81. #         g = self.img[:][:][1]
82. #         r = self.img[:][:][2]
83. res = np.float32(self.img\_gray.copy())
84. img\_dct = cv2.dct(res)
85. img\_idct = cv2.idct(img\_dct)
86. img\_dct = np.log(abs(img\_dct))
88. fig = plt.figure('DCT')
89. plt.subplot(131), plt.imshow(self.img\_gray,
90. 'gray'), plt.title('original')
91. plt.axis('off')
92. plt.subplot(132), plt.imshow(img\_dct, 'gray'), plt.title('DCT')
93. plt.axis('off')
94. plt.subplot(133), plt.imshow(img\_idct, 'gray'), plt.title('IDCT')
95. plt.axis('off')
96. fig.show()
98. PSNR = self.psnr(res.copy(), img\_idct.copy())
99. **print**('DCT与IDCT之后原始图像与恢复图像 PSNR =', PSNR)
101. SSIM = self.ssim(res.copy(), img\_idct.copy())
102. **print**('DCT与IDCT之后原始图像与恢复图像 SSIM =', SSIM)
104. **def** DWT(self):
105. res = self.img\_gray.copy()
107. titles = [
108. 'Approximation', ' Horizontal detail', 'Vertical detail',
109. 'Diagonal detail'
110. ]
111. coeffs2 = pywt.dwt2(res, 'bior1.3')
112. LL, (LH, HL, HH) = coeffs2
113. fig = plt.figure('DWT', figsize=(12, 3))
114. **for** i, a **in** enumerate([LL, LH, HL, HH]):
115. ax = fig.add\_subplot(2, 2, i + 1)
116. ax.imshow(a, interpolation="nearest", cmap=plt.cm.gray)
117. ax.set\_title(titles[i], fontsize=10)
118. ax.set\_xticks([])
119. ax.set\_yticks([])
121. fig.tight\_layout()
122. fig.show()
124. filePath = 'bmp/fav.bmp'
125. bitmap = Bitmap(filePath)
126. bitmap.DFT()
127. bitmap.DCT()
128. bitmap.DWT()
129. **结论**

DCT是离散傅里叶变换；DFT是离散余弦变百换；DWT是离散小波变换。

三者都将空域的图像数据信息转换到频域中，即分离出图像的低频度到高频成分。

区别：

1.DCT与DFT转换后的域仅包含频域成分，就叫频域；DWT转换后的域不仅有频域成分，还具有空域成分，因此叫小波域。

2.DCT的频域的低频成分在DCT系数图的中间，高频成分在四周，离系数图中心越远，频率越高；DFT与DWT的频域的低频成分在系数图左内上方，越往右下方频率越高。

3.DWT的小波域含有不同分辨率的子带，越往左上角分辨率越小，每个子带都包含图像的空域成分。