哈爾濱工業大學

# 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！**

## 一、 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

i. 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

ii. 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；

iii. 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)

iv. 理解 DWT 变换的频率特性

## 二、 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：

$$\mathrm{X(k)} = \sum_{n=0}^{N-1} x(n) e^{-i2\pi k \frac{n}{N}}, (k = 0,1,2,\dots,N-1)$$

上式可转化为：

$$\mathrm{X(k)} = \sum_{n=0}^{N-1} x(n)(\cos 2\pi k \frac{n}{N} - i \sin 2\pi k \frac{n}{N}), (k = 0,1,2,\dots,N-1)$$

其逆变换的公式为：

$$\mathrm{x(n)} = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{i2\pi k \frac{n}{N}}$$

可以转化为：

$$\mathrm{x(n)} = \frac{1}{N}\sum_{k=0}^{N-1} X(k)(\cos 2\pi k\frac{n}{N} + i\sin 2\pi k\frac{n}{N})$$

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：

$$\mathrm{F(u)} = \mathrm{c(u)}\sum_{i=0}^{N-1} f(i)\cos\left[\frac{(i+0.5)\pi}{N}u\right]$$

$$\mathrm{c(u)} = \begin{cases}\sqrt{\frac{1}{N}}, u = 0\\ \sqrt{\frac{2}{N}}, u \neq 0\end{cases}$$

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。

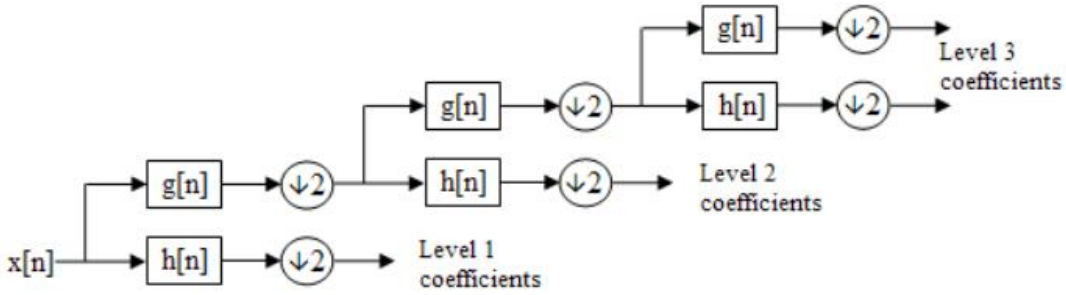
x[n]：离散的输入信号，长度为 N。

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

$$x_{\alpha,L}[n] = \sum_{k=0}^{K-1} x_{\alpha-1,L}[2n-k]g[k]$$

$$x_{\alpha,H}[n] = \sum_{k=0}^{K-1} x_{\alpha-1,L}[2n-k]h[k]$$

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。

## 三、 实验设置及结果分析（包括实验数据集）

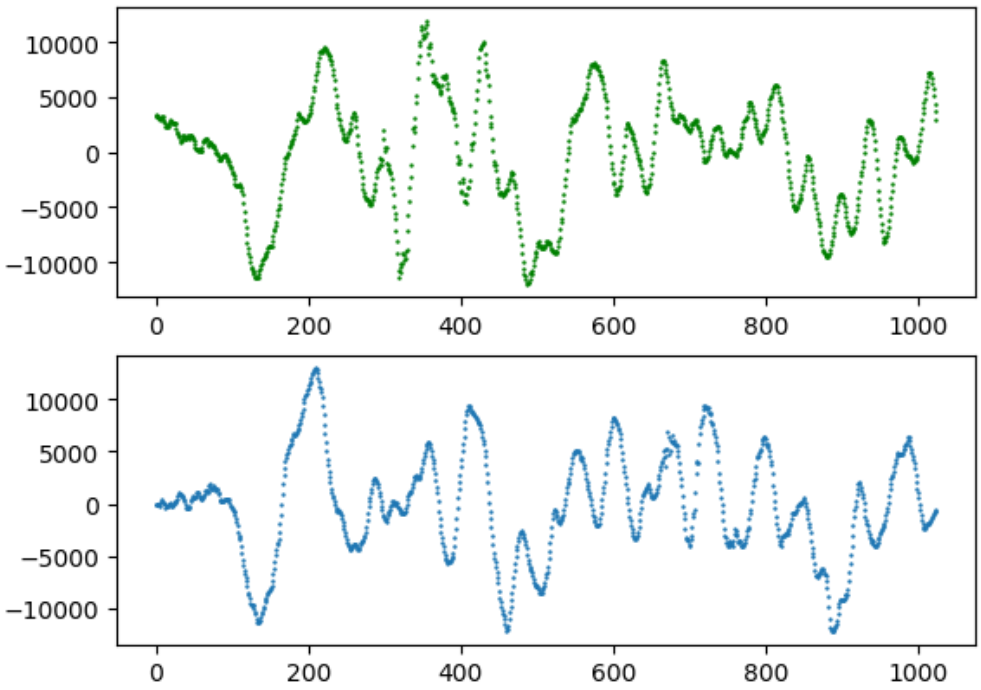
**读入音频文件：**

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:
```

```
    def __init__(self, filePath):
        self.audio = we.open(filePath, 'rb')
        nchannels = self.audio.getnchannels()
        sampwidth = self.audio.getsampwidth()
        self.framerate = self.audio.getframerate()
        self.nframes = self.audio.getnframes()
        comptype = self.audio.getcomptype()
        compname = self.audio.getcompname()
        self.params = self.audio.getparams()
        self.dataWav = self.audio.readframes(self.nframes)
        self.secs = self.nframes / self.framerate
        self.hammingWav = self.hamming()
        self.audio.close()

    def getInfo(self):
        print(self.params)

    def drawReg(self):
        wave = np.fromstring(self.dataWav, dtype=np.short)
        wave = wave.reshape((-1, 2))
        wave = wave.T

        x = np.arange(1024)

        fig = plt.figure('原始图像前 1024 个数据')
        plt.subplot(211)
        plt.scatter(x, wave[0][:1024], color='green', s=0.5)
        plt.subplot(212)
        plt.scatter(x, wave[1][:1024], s=0.5)
        fig.show()

filePath = 'wav/fav.wav'
wav = Wave(filePath)
wav.getInfo()
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, ))
6.
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
11.
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
14.
15.     return frames
```

接下来进行加窗操作，使用的是余弦窗函数，即

$$\omega(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{M-1}\right), 0 \le n \le M-1$$

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$\varphi(n)$乘以窗函数的第 n 项，得到加窗后的音频数据，即

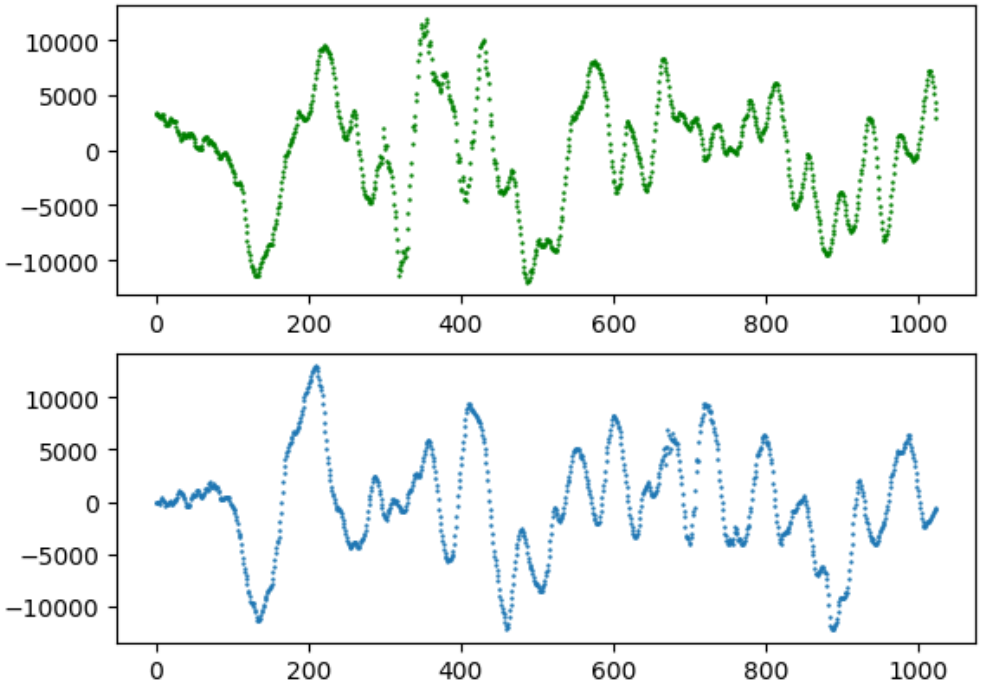
$$\tau(n) = \varphi(n)\omega(n), 0 \le n \le M - 1$$

同时画出其

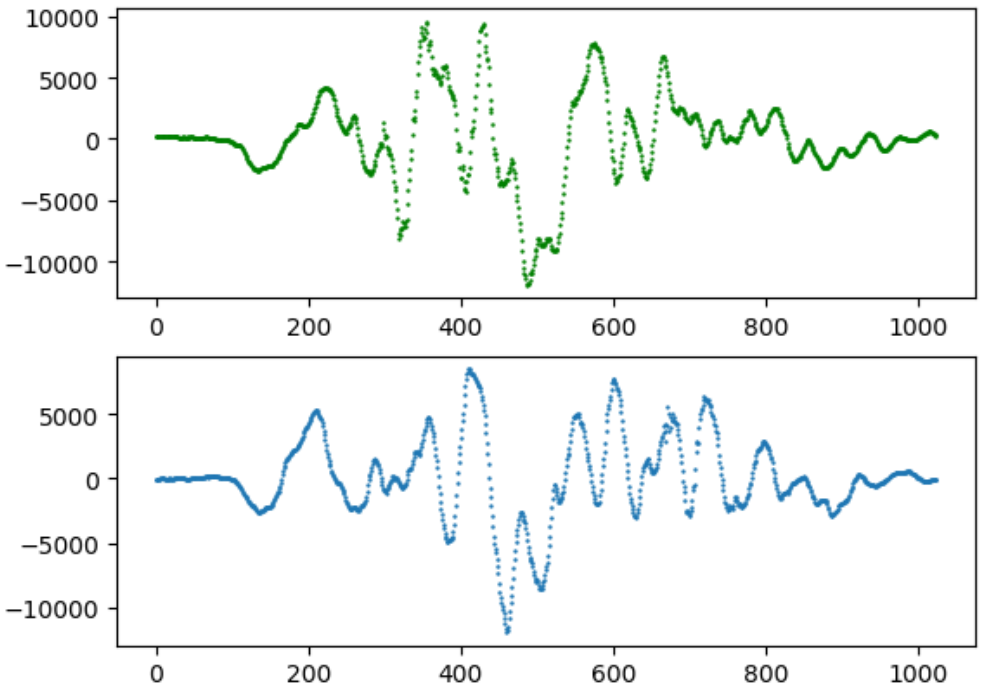
加窗操作的函数代码如下：

```
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
5.
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()]
9.
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')
13.
14.     frames = (frames * np.hamming(wlen)).reshape(2, -1)
15.
16.     self.draw(frames[:, :1024], x, '加窗之后两个声道的第一帧')
17.
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：**

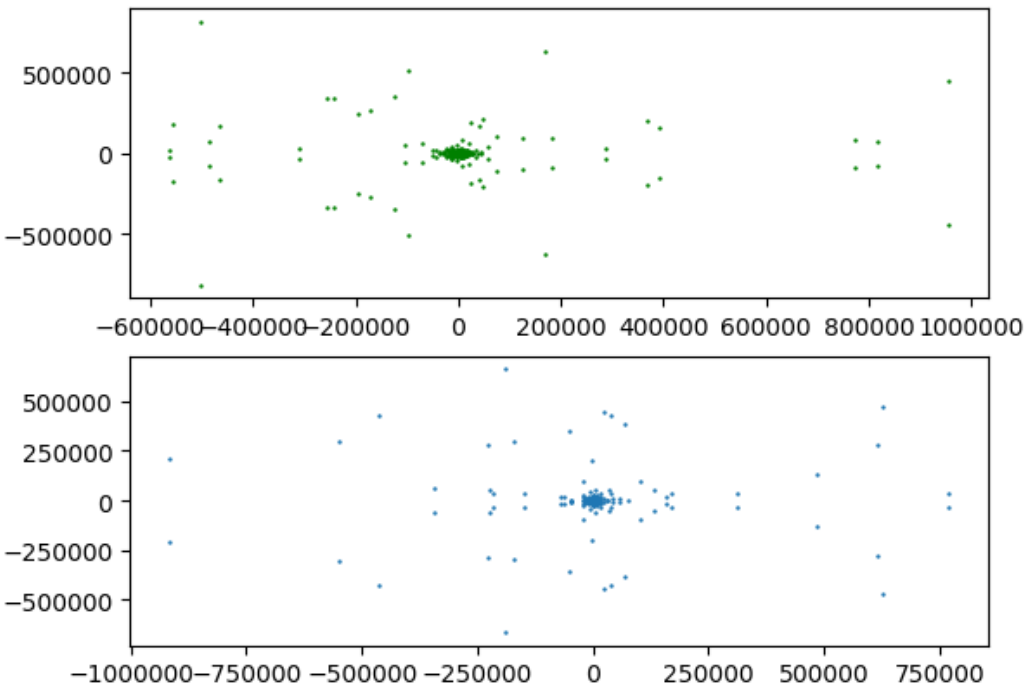
DFT 的公式及原理此前已经说明，需要注意的是，由于计算机处理复数不方便，所以分实部和虚部分别计算，并且 IDFT 在计算之后，需要只保留实部，编写代

码如下：

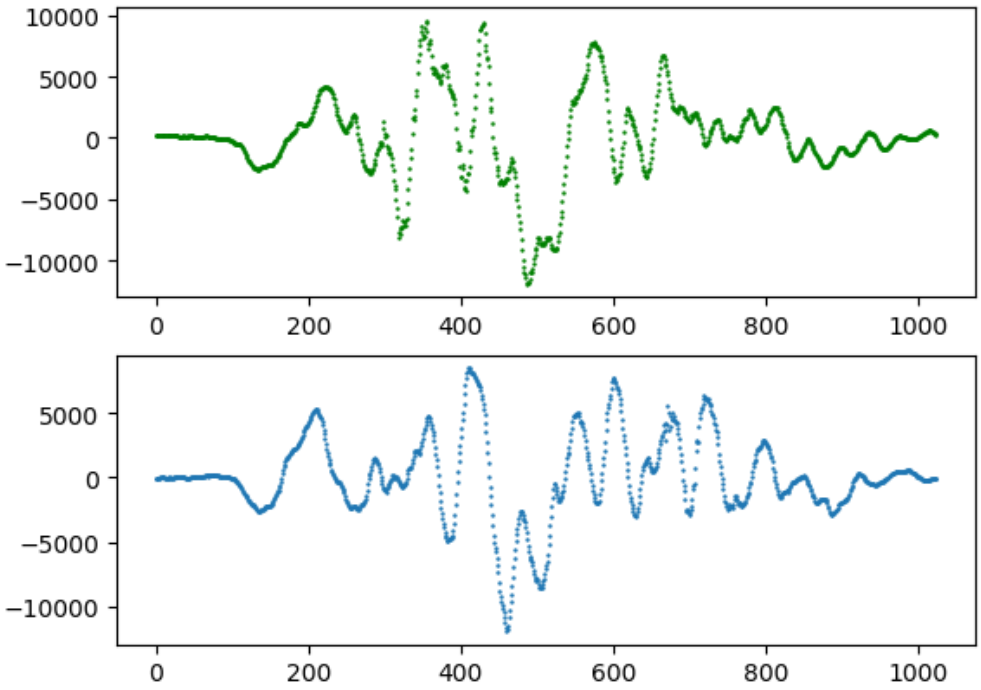
```
def mainDFT(self):
    res_little = self.hammingWav.copy()[:, :1024]

    # DFT
    res_dft = self.dft(res_little)

    fig = plt.figure('经过 DFT 之后两个声道的第一帧')
    plt.subplot(211)
    plt.scatter(np.real(res_dft)[0],
                np.imag(res_dft)[0],
                color='green',
                s=0.5)
    plt.subplot(212)
    plt.scatter(np.real(res_dft)[1], np.imag(res_dft)[1], s=0.5)
    fig.show()

    #IDFT
    res_idft = self.idft(res_dft)
    x = np.array(range(res_idft.shape[1]))
    self.draw(res_idft, x, '经过 IDFT 之后两个声道的第一帧')

def dft(self, res):
    temp = np.zeros_like(res, dtype='complex')
    row = temp.shape[0]
    column = temp.shape[1]
    for j in range(row):
        for i in range(column):
            exponent = 2 * math.pi * i / column
            cos = [math.cos(n * exponent) for n in range(column)]
            sin = [math.sin(n * exponent) * (1j) for n in range(column)]

            cos = np.array(cos).reshape((-1, 1))
            sin = np.array(sin).reshape((-1, 1))

            real = np.dot(res[j, :].reshape((1, -1)), cos)[0][0]
            imag = np.dot(res[j, :].reshape((1, -1)), sin)[0][0]

            temp[j][i] = real - imag

    return temp.copy()

def idft(self, res):
    temp = np.zeros_like(res, dtype='complex')
```

```
    column = temp.shape[1]
    row = temp.shape[0]

    for j in range(row):
        for i in range(column):
            exponent = 2 * math.pi * i / column

            cos = [math.cos(k * exponent) for k in range(column)]
            sin = [math.sin(k * exponent) * (1j) for k in range(column)]

            cos = np.array(cos).reshape((1, -1))
            sin = np.array(sin).reshape((1, -1))

            real = np.mean(res[j, :].reshape((1, -1)) * cos)
            imag = np.mean(res[j, :].reshape((1, -1)) * sin)

            temp[j][i] = real + imag

    temp = np.real(temp)

    return temp.copy()
```

DFT 处理之后的图像如下：




IDFT 处理之后的图像如下：



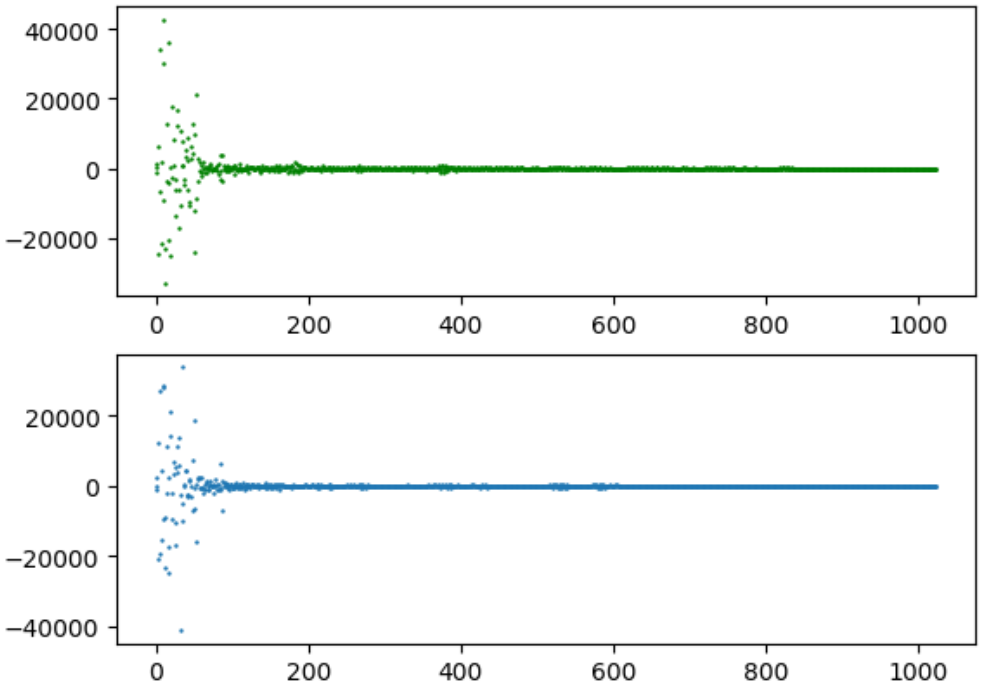

**一维 DCT 处理：**

代码如下：

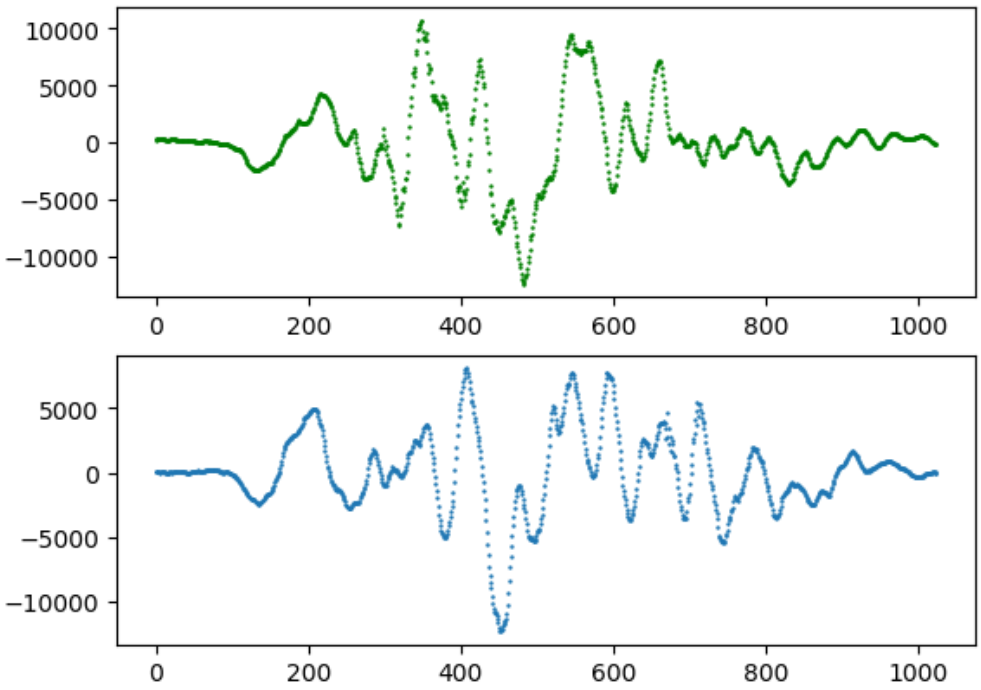
```
1.  def mainDCT(self):
2.      res_little = self.hammingWav.copy()[:, :1024]
3.
4.      res_dct = self.dct(res_little)
5.      x = np.array(range(res_dct.shape[1]))
6.      self.draw(res_dct, x, '经过 DCT 之后两个声道的第一帧')
7.
8.      res_idct = self.dct(res_dct)
9.      x = np.array(range(res_dct.shape[1]))
10.     self.draw(res_idct, x, '经过 IDCT 之后两个声道的第一帧')
11.
12. def dct(self, res):
13.     column = res.shape[1]
14.     row = res.shape[0]
15.
16.     c = np.sqrt(2 / column) * np.ones((1, column))
17.     c[0][0] = np.sqrt(1 / column)
18.
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))
```

```
    cos = cos * column_line * row_line
    cos = np.cos(cos)

    temp = np.dot(res, cos)
    temp = temp * c

    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：

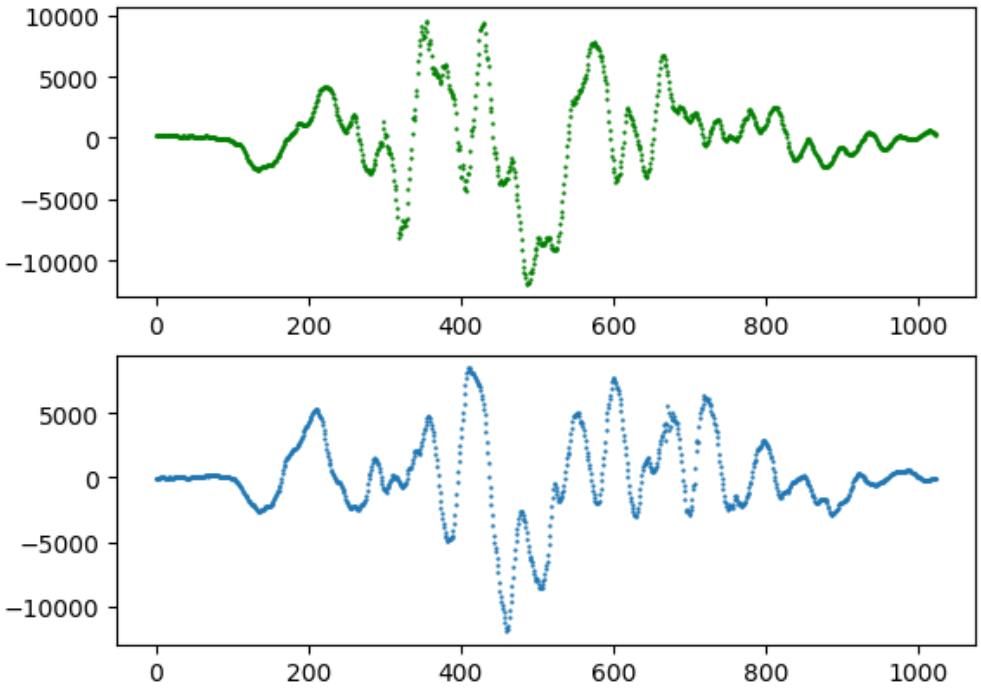



**一维 DWT 处理：**

代码如下：

```
1.  def mainDWT(self):
2.      data = self.hammingWav.copy()
3.
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
8.
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
13.
14.     res = np.r_[res0, res1].reshape((2, -1))
15.
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：

```
import wave as we
import matplotlib
matplotlib.use('Qt5Agg')
import matplotlib.pyplot as plt
import numpy as np
import cv2
import pywt
import math

class Wave:
    def __init__(self, filePath):
        self.audio = we.open(filePath, 'rb')
        nchannels = self.audio.getnchannels()
        sampwidth = self.audio.getsampwidth()
        self.framerate = self.audio.getframerate()
        self.nframes = self.audio.getnframes()
        comptype = self.audio.getcomptype()
        compname = self.audio.getcompname()
        self.params = self.audio.getparams()
        self.dataWav = self.audio.readframes(self.nframes)
        self.secs = self.nframes / self.framerate
        self.hammingWav = self.hamming()
        self.audio.close()
```

```
24.
25.     def drawReg(self):
26.         wave = np.fromstring(self.dataWav, dtype=np.short)
27.         wave = wave.reshape((-1, 2))
28.         wave = wave.T
29.
30.         x = np.arange(1024)
31.
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()
38.
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()
46.
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, ))
52.
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
57.
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
60.
61.         return frames
62.
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))
```

```
        wave_data = wave_data.T

        frame_0 = self.framing(wave_data[0].copy())
        frame_1 = self.framing(wave_data[1].copy())
        frames = np.r_[frame_0.copy(), frame_1.copy()]

        a = np.r_[frame_0[0].reshape((1, -1)), frame_1[0].reshape((1, -
    1))]
        x = np.array(range(wlen))
        self.draw(a, x, '分帧之后两个声道的第一帧')

        frames = (frames * np.hamming(wlen)).reshape(2, -1)

        self.draw(frames[:, :1024], x, '加窗之后两个声道的第一帧')

        return frames

    def mainDFT(self):
        res_little = self.hammingWav.copy()[:, :1024]

        # DFT
        res_dft = self.dft(res_little)

        fig = plt.figure('经过 DFT 之后两个声道的第一帧')
        plt.subplot(211)
        plt.scatter(np.real(res_dft)[0],
                    np.imag(res_dft)[0],
                    color='green',
                    s=0.5)
        plt.subplot(212)
        plt.scatter(np.real(res_dft)[1], np.imag(res_dft)[1], s=0.5)
        fig.show()

        #IDFT
        res_idft = self.idft(res_dft)
         x = np.array(range(res_idft.shape[1]))
         self.draw(res_idft, x, '经过 IDFT 之后两个声道的第一帧')

     def dft(self, res):
         temp = np.zeros_like(res, dtype='complex')
         row = temp.shape[0]
         column = temp.shape[1]
         for j in range(row):
             for i in range(column):
```

```
                exponent = 2 * math.pi * i / column
                cos = [math.cos(n * exponent) for n in range(column)]
                sin = [math.sin(n * exponent) * (1j) for n in range(column)
    ]

                cos = np.array(cos).reshape((-1, 1))
                sin = np.array(sin).reshape((-1, 1))

                real = np.dot(res[j, :].reshape((1, -1)), cos)[0][0]
                imag = np.dot(res[j, :].reshape((1, -1)), sin)[0][0]

                temp[j][i] = real - imag

        return temp.copy()

    def idft(self, res):
        temp = np.zeros_like(res, dtype='complex')
        column = temp.shape[1]
        row = temp.shape[0]

        for j in range(row):
            for i in range(column):
                exponent = 2 * math.pi * i / column

                cos = [math.cos(k * exponent) for k in range(column)]
                sin = [math.sin(k * exponent) * (1j) for k in range(column)
    ]

                cos = np.array(cos).reshape((1, -1))
                sin = np.array(sin).reshape((1, -1))

                real = np.mean(res[j, :].reshape((1, -1)) * cos)
                imag = np.mean(res[j, :].reshape((1, -1)) * sin)

                temp[j][i] = real + imag

        temp = np.real(temp)

        return temp.copy()

    def mainDCT(self):
        res_little = self.hammingWav.copy()[:, :1024]

        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')
153.
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')
157.
158.     def dct(self, res):
159.         column = res.shape[1]
160.         row = res.shape[0]
161.
162.         c = np.sqrt(2 / column) * np.ones((1, column))
163.         c[0][0] = np.sqrt(1 / column)
164.
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)
170.
171.         temp = np.dot(res, cos)
172.         temp = temp * c
173.
174.         return temp.copy()
175.
176.     def mainDWT(self):
177.         data = self.hammingWav.copy()
178.
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
183.
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
188.
189.         res = np.r_[res0, res1].reshape((2, -1))
190.
191.         self.draw(res[:, :1024], np.array(range(1024)), '经过 DWT 之后两个声
    道的第一帧')
192.
```

```
    def getInfo(self):
        print(self.params)

filePath = 'wav/fav.wav'
wav = Wave(filePath)

wav.getInfo()
wav.drawReg()
wav.mainDFT()
wav.mainDCT()
wav.mainDWT()
```

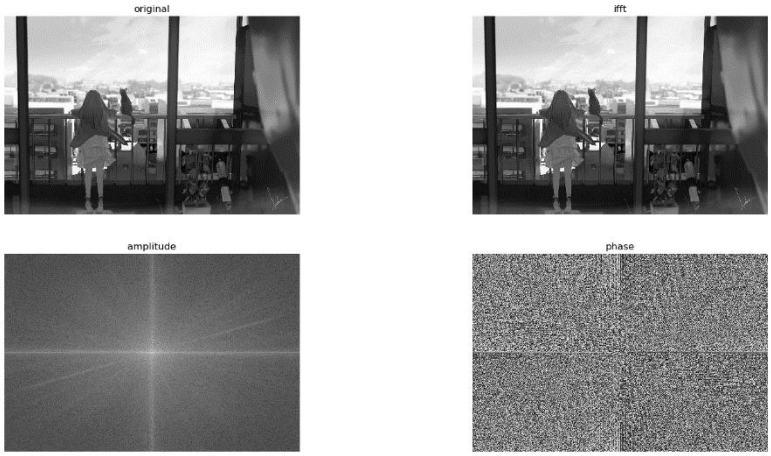
**图像的二维 DFT 变换：**

代码如下：

```
def DFT(self):
    #快速傅里叶变换算法得到频率分布
    start = time.time()
    f = np.fft.fft2(self.img_gray.copy())
    end = time.time()
    print('图像大小为:(%d,%d)' %
          (self.img_gray.shape[0], self.img_gray.shape[1]))
    print('FFT 耗时为:%dms' % ((end - start) * 1000))

    ifft = np.real(np.fft.ifft2(f))

    #默认结果中心点位置是在左上角,
    #调用 fftshift()函数转移到中间位置
    fshift = np.fft.fftshift(f)
    log = np.log(fshift)

    real = np.real(log)
    imag = np.imag(log)

    amplitude = real / np.max(real)
    phase = imag / np.max(imag)

    #展示结果
    #原图，复原图，幅度，相位
    fig = plt.figure('FFT')
    plt.subplot(221), plt.imshow(self.img_gray,
                                 'gray'), plt.title('original')
    plt.axis('off')
    plt.subplot(222), plt.imshow(ifft,'gray'),plt.title('ifft')
```

```
    plt.axis('off')
    plt.subplot(223), plt.imshow(amplitude, 'gray'), plt.title('amplitude')

    plt.axis('off')
    plt.subplot(224), plt.imshow(phase, 'gray'), plt.title('phase')
    plt.axis('off')
    fig.show()
```

得到的图像如下：



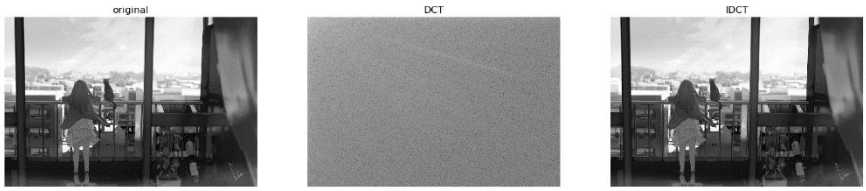

程序输出了 FFT 的耗时，如下：
图像大小为:(1292,1929)
FFT 耗时为:543ms

**图像二维 DCT 变换：**
包括其 psnr 与 ssim 计算，代码如下：

```
def DCT(self):
    #         b = self.img[:][:][0]
    #         g = self.img[:][:][1]
    #         r = self.img[:][:][2]
    res = np.float32(self.img_gray.copy())
    img_dct = cv2.dct(res)
    img_idct = cv2.idct(img_dct)
    img_dct = np.log(abs(img_dct))

    fig = plt.figure('DCT')
    plt.subplot(131), plt.imshow(self.img_gray,
                                 'gray'), plt.title('original')
    plt.axis('off')
    plt.subplot(132), plt.imshow(img_dct, 'gray'), plt.title('DCT')
    plt.axis('off')
```

```
    plt.subplot(133), plt.imshow(img_idct, 'gray'), plt.title('IDCT')
    plt.axis('off')
    fig.show()

    PSNR = self.psnr(res.copy(), img_idct.copy())
    print('DCT 与 IDCT 之后原始图像与恢复图像 PSNR =', PSNR)

    SSIM = self.ssim(res.copy(), img_idct.copy())
    print('DCT 与 IDCT 之后原始图像与恢复图像 SSIM =', SSIM)

def psnr(self, origin, target):
    MSE = np.mean((origin - target)**2)
    MAX = 255
    PSNR = 10 * math.log((MAX**2) / MSE, 10)
    return PSNR

def ssim(self, origin, target):
    mu_x = np.mean(origin)
    mu_y = np.mean(target)
    sigma_x = np.std(origin)
    sigma_y = np.std(target)
    sigma_xy = np.cov(origin, target)

    k1 = 0.01
    k2 = 0.03
    L = 255

    c1 = (k1 * L)**2
    c2 = (k2 * L)**2

    up = (2 * mu_x * mu_y + c1) * (2 * sigma_xy + c2)
    down = (mu_x**2 + mu_y**2 + c1) * (sigma_x**2 + sigma_y**2 + c2)

    SSIM = np.mean(up / down)

    return SSIM
```

所得的图像如下：

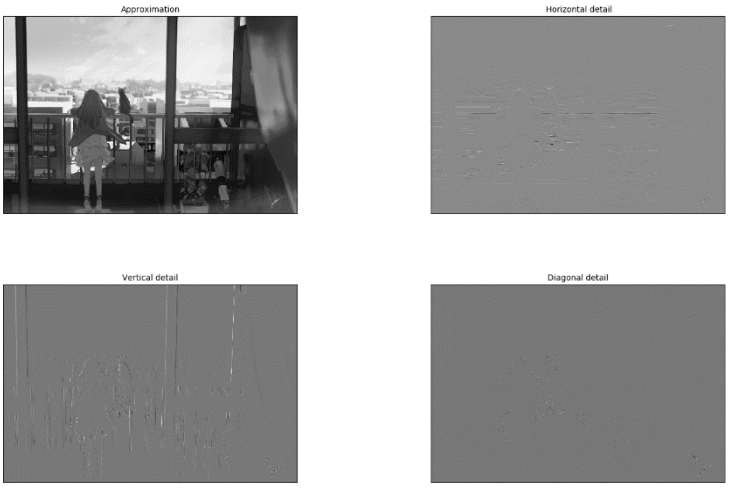



程序计算了其 psnr 和 ssim 值：
DCT 与 IDCT 之后原始图像与恢复图像 PSNR = 131.8887089186267
DCT 与 IDCT 之后原始图像与恢复图像 SSIM = 0.17763278573476893

**图像二维 DWT 变换：**
代码如下：

```
def DWT(self):
    res = self.img_gray.copy()

    titles = [
        'Approximation', ' Horizontal detail', 'Vertical detail',
        'Diagonal detail'
    ]
    coeffs2 = pywt.dwt2(res, 'bior1.3')
    LL, (LH, HL, HH) = coeffs2
    fig = plt.figure('DWT', figsize=(12, 3))
    for i, a in enumerate([LL, LH, HL, HH]):
        ax = fig.add_subplot(2, 2, i + 1)
        ax.imshow(a, interpolation="nearest", cmap=plt.cm.gray)
        ax.set_title(titles[i], fontsize=10)
        ax.set_xticks([])
        ax.set_yticks([])

    fig.tight_layout()
    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
9.
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)
15.
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
21.
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)
28.
29.         k1 = 0.01
```

```
        k2 = 0.03
        L = 255

        c1 = (k1 * L)**2
        c2 = (k2 * L)**2

        up = (2 * mu_x * mu_y + c1) * (2 * sigma_xy + c2)
        down = (mu_x**2 + mu_y**2 + c1) * (sigma_x**2 + sigma_y**2 + c2)

        SSIM = np.mean(up / down)

        return SSIM

    def DFT(self):
        #快速傅里叶变换算法得到频率分布
        start = time.time()
        f = np.fft.fft2(self.img_gray.copy())
        end = time.time()
        print('图像大小为:(%d,%d)' %
              (self.img_gray.shape[0], self.img_gray.shape[1]))
        print('FFT 耗时为:%dms' % ((end - start) * 1000))

        ifft = np.real(np.fft.ifft2(f))

        #默认结果中心点位置是在左上角,
        #调用 fftshift()函数转移到中间位置
        fshift = np.fft.fftshift(f)
        log = np.log(fshift)

        real = np.real(log)
        imag = np.imag(log)

        amplitude = real / np.max(real)
        phase = imag / np.max(imag)

        #展示结果
        #原图，复原图，幅度，相位
        fig = plt.figure('FFT')
        plt.subplot(221), plt.imshow(self.img_gray,
                                     'gray'), plt.title('original')
        plt.axis('off')
        plt.subplot(222), plt.imshow(ifft,'gray'),plt.title('ifft')
        plt.axis('off')
```

```
        plt.subplot(223), plt.imshow(amplitude, 'gray'), plt.title('amplitud
    e')
        plt.axis('off')
        plt.subplot(224), plt.imshow(phase, 'gray'), plt.title('phase')
        plt.axis('off')
        fig.show()

    def DCT(self):
        #         b = self.img[:][:][0]
        #         g = self.img[:][:][1]
        #         r = self.img[:][:][2]
        res = np.float32(self.img_gray.copy())
        img_dct = cv2.dct(res)
        img_idct = cv2.idct(img_dct)
        img_dct = np.log(abs(img_dct))

        fig = plt.figure('DCT')
        plt.subplot(131), plt.imshow(self.img_gray,
                                     'gray'), plt.title('original')
        plt.axis('off')
        plt.subplot(132), plt.imshow(img_dct, 'gray'), plt.title('DCT')
        plt.axis('off')
        plt.subplot(133), plt.imshow(img_idct, 'gray'), plt.title('IDCT')
        plt.axis('off')
        fig.show()

        PSNR = self.psnr(res.copy(), img_idct.copy())
        print('DCT 与 IDCT 之后原始图像与恢复图像 PSNR =', PSNR)

         SSIM = self.ssim(res.copy(), img_idct.copy())
         print('DCT 与 IDCT 之后原始图像与恢复图像 SSIM =', SSIM)

    def DWT(self):
        res = self.img_gray.copy()

        titles = [
            'Approximation', ' Horizontal detail', 'Vertical detail',
            'Diagonal detail'
        ]
        coeffs2 = pywt.dwt2(res, 'bior1.3')
        LL, (LH, HL, HH) = coeffs2
        fig = plt.figure('DWT', figsize=(12, 3))
        for i, a in enumerate([LL, LH, HL, HH]):
            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([])
120.
121.         fig.tight_layout()
122.         fig.show()
123.
124. filePath = 'bmp/fav.bmp'
125. bitmap = Bitmap(filePath)
126. bitmap.DFT()
127. bitmap.DCT()
128. bitmap.DWT()
```

## 四、 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。