# 第二次大作业技术报告

## 环境配置

使用 *python3.8* 以及 *numpy*、*cv2*、*matplotlib* 等基础库

## 实验概述

- 本次实验共分为两个部分：
  - **离散傅里叶变换（DFT）**和**二维快速傅里叶变换（FFT）**的实现以及性质探究
    - 其中包括**频域平移（FFTShift）**和一系列**逆变换（iFFT、iDFT）**的实现
  - **二维离散余弦变换（DCT）**的实现以及性质探究

## 傅里叶变换实验细节

### 离散傅里叶变换（DFT）

- 通过课上的学习，我们可以直接给出离散傅里叶变换及其逆变换的公式，如下所示：

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

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

  这两个公式极其相似，仅仅是差了系数以及幂的正负号，这也预示着实现正逆变换的代码是极其相似的。

- 在实际运算中，我们可以预先将e的幂次那个项计算好，作为一个矩阵。对于任意的输入x(k)，只需要与该矩阵相乘，即可获得变换后的结果。

$$\begin{bmatrix}X(0)\\X(1)\\\vdots\\X(N-1)\end{bmatrix}=\begin{bmatrix}W^0&W^0&W^0&\cdots&W^0\\W^0&W^{1\times1}&W^{2\times1}&\cdots&W^{(N-1)\times1}\\\vdots&\vdots&\vdots&\cdots&\vdots\\W^0&W^{1\times(N-1)}&W^{2\times(N-1)}&\cdots&W^{(N-1)\times(N-1)}\end{bmatrix}\begin{bmatrix}x(0)\\x(1)\\\vdots\\x(N-1)\end{bmatrix}$$

  如下所示：

```
def dft(pic_line):#一维离散傅里叶变换n^2
    pic_line=np.squeeze(pic_line)
    n=pic_line.size
    w=np.array([[np.exp(-1j*2*np.pi*i*k/n) for k in range(0,n) ] for i in
range(0,n)]).T
    #w=np.array([[np.exp(1j*2*np.pi*i*k/n) for k in range(0,n) ] for i in
range(0,n)]).T 逆变换
    return pic_line.dot(w)
```

**二维快速傅里叶变换（FFT）**

- 由于DFT的复杂度显然是O(n^2)，找到一个低复杂度的算法完成计算是很有必要的。于是，基于**分治思想**的快速傅里叶变换就被提出了。他将上述DFT的求和过程运用数学性质拆分成了两部分（奇偶项分离），并且可以继续向下拆分。由此，FFT的算法复杂度就被有效降低到了O(nlogn)的复杂度。其具体原理不再赘述，可参考以下的帖子：[FFT原理](#)。在这里只给出最终的公式。

$$X_N(k)=\begin{cases}X'_{N/2}(k')+W_N^k X''_{N/2}(k'') & k=0\ldots N/2-1\\ X'_{N/2}(k')-W_N^k X''_{N/2}(k'') & k=N/2\ldots N-1\end{cases}$$

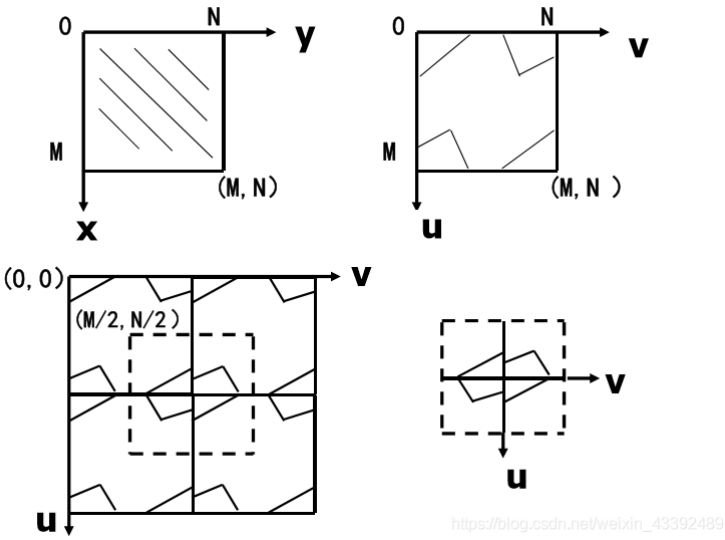
- FFT具体的代码实现采用了递归的思路，这个合并运算单元也被叫做**蝶形算子**。当维度下降到一定程度时(<=8)，便直接采用DFT的公式暴力完成计算。二维的FFT实际上就是先对一个图像的每一行像素做一遍一维的FFT，再对结果的每一列做一遍一维的FFT即可。具体的代码实现如下所示：

```
def Butterfly_operation(seq1,seq2):#蝶形运算 seq1为偶序列，seq2为奇序列
    n=seq1.size
    N=n*2
    seq1=np.squeeze(seq1)#降维
    seq2=np.squeeze(seq2)
    res=np.zeros(2*n,dtype=seq1.dtype)
    for i in range(0,n):
        res[i]+=seq1[i]+np.exp(-1j*2*np.pi*i/N)*seq2[i]
        res[n+i]=seq1[i]-np.exp(-1j*2*np.pi*i/N)*seq2[i]
    return  res

def fft(img):#递归分治实现fft,只对图像的每一行做fft
    n=img.shape[1]
    if(n%2!=0):
        return ValueError("图片大小不是2的次幂")#因为fft分治时每次都分为等大的奇数列和偶数列
    if(n<=8):#n足够小，直接n方对每行求dft
        return np.array([dft(img[i, :]) for i in range(img.shape[0])])
    res_odd=fft(img[:,1::2])
    res_even=fft(img[:,0::2])
    return
 np.array([Butterfly_operation(res_even[i:i+1,:],res_odd[i:i+1,:]) for i in range(0,img.shape[0])])

def TwoD_fft(img):
    return fft(fft(img).T).T
```

- 逆变换的操作与正变换完全一致，也使用了分治、蝶形算子的思路，在此不再赘述。
- 这里有一个细节：
  当我们运算完FFT，生成出频域的结果和图像时，他与我们日常所见的是不一样的。首先，对于原始的频域信息，再转换成灰度图显示的时候，经常用**log函数**来增大对比度，增强美观性。其次，经过搜索发现，在FFT的应用中，都会出现一个**FFTSHIFT**函数，他的目的是将频域沿拓、裁剪（实际上就是裁剪、平移），使得零点在图的中间。具体如下图所示：



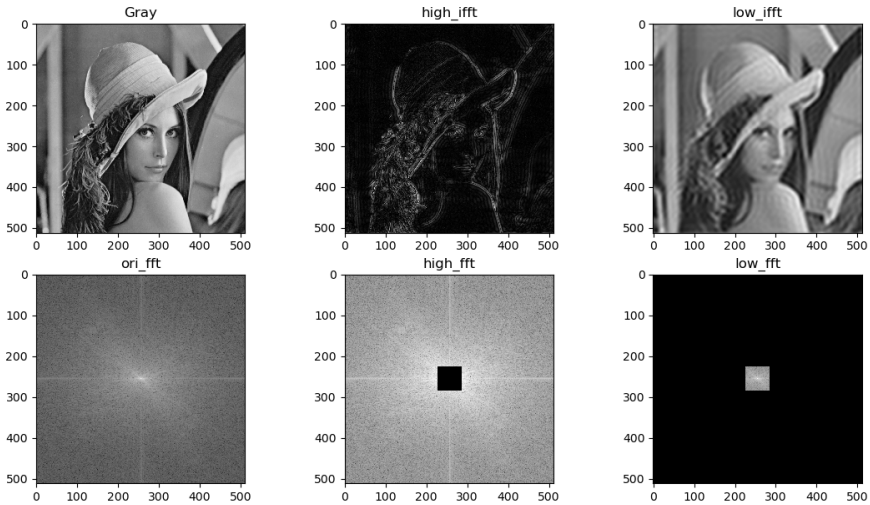
https://blog.csdn.net/weixin_43392489

相应的，我也对应的写了一个 FFT_SHIFT函数完成对应功能

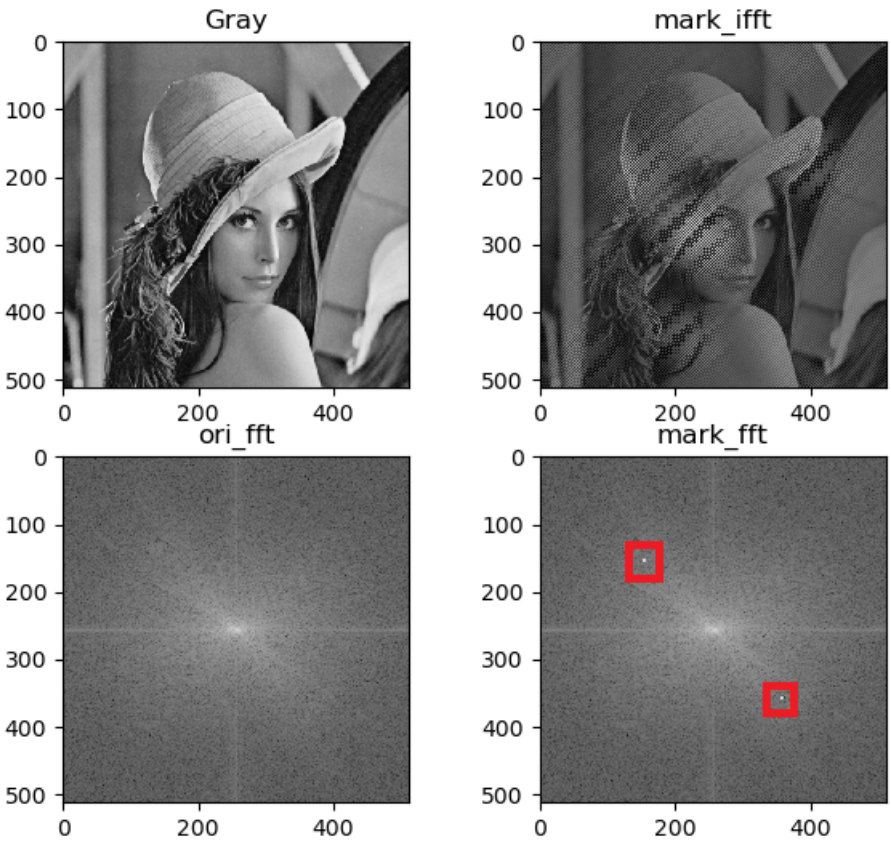
```
def FFT_SHIFT(img):
    M, N = img.shape
    M = int(M / 2)
    N = int(N / 2)
    return np.vstack((np.hstack((img[M:, N:], img[M:, :N])),
np.hstack((img[:M, N:], img[:M, :N]))))
```

**实验探究及结果**

- 实验的流程如下：
  首先将图像读入，使用FFT完成频谱生成并显示，使用iFFT还原图片并检查
  将频谱的**高频**部分滤出，使用iFFT还原图片查看效果
  将频谱的**低频**部分滤出，使用iFFT还原图片查看效果
  在频谱上增加两个**噪声**，使用iFFT还原图片，查看噪声影响
- 实验效果如下所示：




可以看到频谱的信息集中在中间区域，并且呈现中心对称。当去掉低频信息，只保存高频信息后，发现还原的图片正确的出现了边 缘信息。当去掉高频，只保留低频信息后，我们发现图片大体完整，但是变得十分模糊，边缘信息都被丢失。这充分印证了课上理论的正确性。




进一步，当在频域上增加了两点对称的噪声时，经过还原，明显可以看到图片上多了周期性的噪声纹理，这表明我们可以反过来利用傅里叶变换完成图片的去噪等应用。

## 离散余弦变换实验细节

### 离散余弦变换

- 离散余弦变换实际上就是傅里叶变换，但是他利用函数的奇偶性质，对于特定的函数，可以只保留实数项，舍弃虚数项。通过学习，我了解到了离散余弦变换的公式，其推导过程就不在此赘述了，可以参考这个博客：DCT，最终的正、逆变换公式如下所示：

$$X(k) = \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} x[n] a_k cos[k\frac{\pi}{N}(n+\frac{1}{2})]$$

$$x(n) = \sum_{k=0}^{N-1} X(k) a_k cos[k\frac{\pi}{N}(n+\frac{1}{2})]$$

$$\text{其中} a_k \text{在} k=0 \text{时取} \frac{1}{\sqrt{2}}\text{，其他情况取} 1$$

对应的代码如下所示：与DFT实际上较为相似

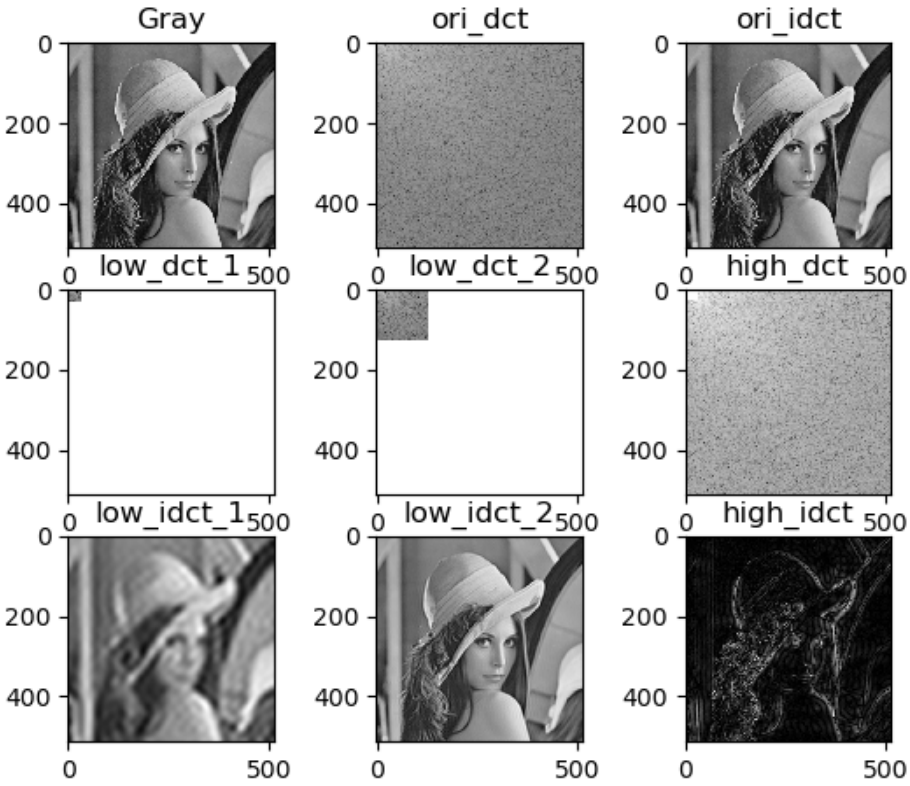
```
def OneD_DCT(pic_line):#一维离散余弦变换n^2
    pic_line=np.squeeze(pic_line)
    n=pic_line.size
    w= np.array([[(math.cos(k*np.pi/n*(i+0.5))) for k in range(0,n)]  for i
in range(0,n)])
    for i in range(0,n):
        w[i,0] = w[i,0] / math.sqrt(2)
    return pic_line.dot(w)
```

二维DCT也与FFT极其类似，先对一个图像的每一行像素做一遍一维的DCT，再对结果的每一列做一遍一维的DCT即可，较为简单。

```
def DCT(img):
    return np.array([OneD_DCT(img[i, :]) for i in range(img.shape[0])])
def TwoD_DCT(img):
    n=img.shape[1]
    return DCT(DCT(img).T).T * 2 / math.sqrt(n*n)
```

**实验探究及结果**

- 实验的流程如下：
  首先将图像读入，使用DCT完成频谱生成并显示，使用iDCT还原图片并检查
  将频谱的**高频**部分滤出，使用iDCT还原图片查看效果
  将频谱的**低频**部分滤出，使用iDCT还原图片查看效果
- 实验效果如下所示：




可以看到，DCT的图像信息集中在左上角，比起FFT更为集中。当我们将其高频部分去除时，图片的边缘信息完全丢失，变得极其模糊。而当我们将其低频部分去除时，只有边缘信息得以保留。

## 心得体会

- 这次实验我从C转到了python环境，一方面是代码可以写的较为简洁，并且不用操心存储等底层技术；另一方面是可以较为轻松的对比、展示图片，实验效果比起上一次实验有了较大的改进。
- 通过这一次的实验，我系统学习了傅里叶变换的推导，明白了信号的处理有多么困难。通过对于滤波的实验，我明白了原来图片可以从波的角度来理解，对于其的更改，操作远不仅仅只有平时的那些方式。另外，这也为我今后的可能的科研打好了基础，学习了基本知识。同时，我也进一步锻炼了递归程序的书写能力，掌握了python图片的操作，了解了原来numpy底下就有fft，ifft等函数的存在。
- 代码已全部开源至Github