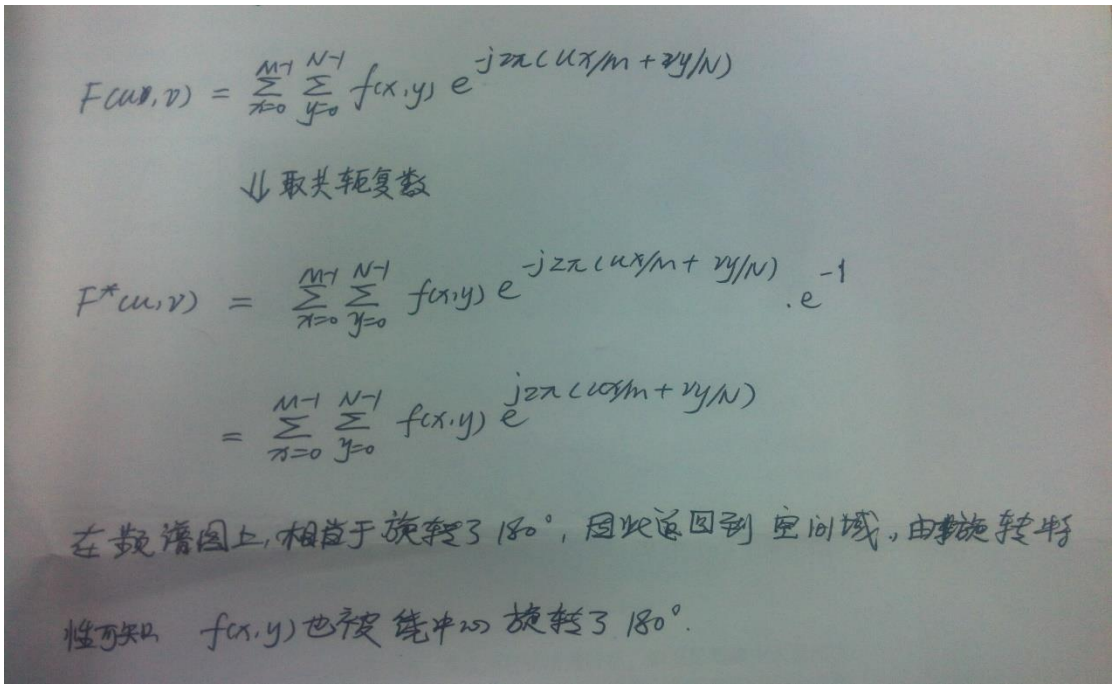
HW3: Filtering in the Frequency Domain Report

1 Exercises

1.1 Rotation (10 Points)

步骤的第三步，取 DFT 的共轭复数，其实就相当于在利用以下公式计算傅里叶变换时，多乘一个分量。如下所示：



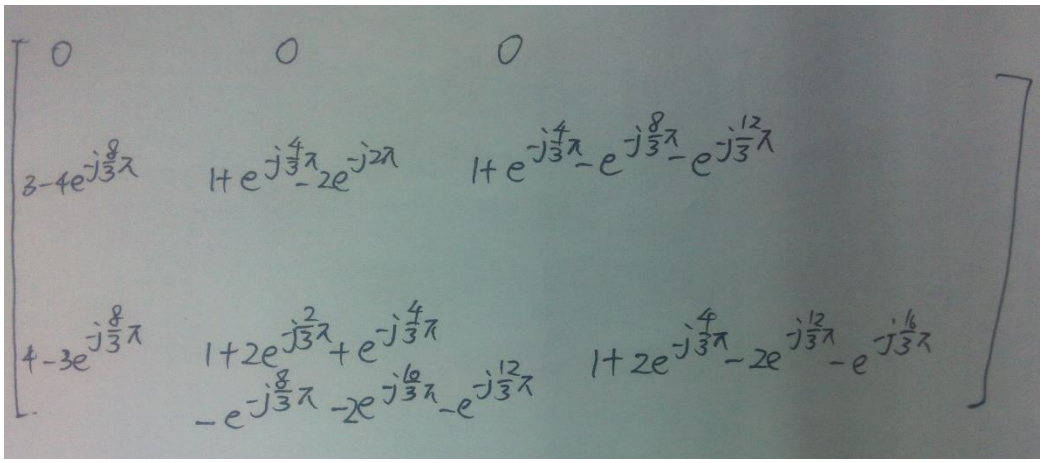
1.2 Fourier Spectrum (10 Points)

在空间域的像素图像周围补 0，一方面增加了低频部分的分量，使得频谱图像中中间部分的亮色有所增大。同时，当 u 或者 v 某个轴为 0 的时候，图像的 0 值变多，会导致落在轴上或者接近轴的地方的频率的值增大。因此，在 u，v 轴上，标定后会有白色的界限效果。如以下变换算法所示。

```
public static Complex computeDFT(int[][] grayImage, int u, int v) {
  int M = grayImage.length;
  int N = grayImage[0].length;
  Complex c = new Complex(0, 0);
  for (int x = 0; x < M; x++) {
    for (int y = 0; y < N; y++) {
      double a = (double)(u * x) / (double)M;
      double b = (double)(v * y) / (double)N;
      Complex temp = new Complex(0, -2 * Math.PI * (a + b));
      c = c.plus(temp.exp().times(grayImage[x][y]));
    }
  }

  return c;
}
```

1.3 Lowpass and Highpass (5 * 2 = 10 Points)

1.



2

上面得到的在频率域下面的 H(u, v)是没有经过中心化的，对比数据，可以知道从中心向 x，y 轴方向递增，因此，改滤波器是高通滤波器。

1.4 Exchangeability (10 Points)

由于高通滤波需要对图像在空间域进行卷积运算，均衡化处理需要改变和处理位置相关的像素，因此类似于矩阵乘法运算的不可交换性，这两个操作也是不可交换的。

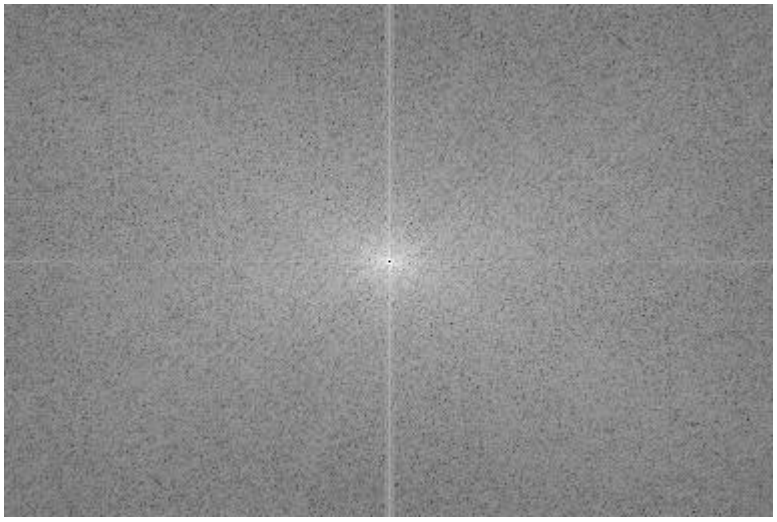
2 Programming Tasks

2.1 Pre-requirement

学号：13331270，选择的图片是：70.png，如下所示：

2.2 Fourier Transform (30 Points)

1. DFT



2. IDFT

3. 解释

DFT 依照以下公式进行变换：

二维离散傅立叶变换（DFT）

$$F(u,v) = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-j2\pi(ux/M+vy/N)}$$

因此可以构造出如下的转换函数：

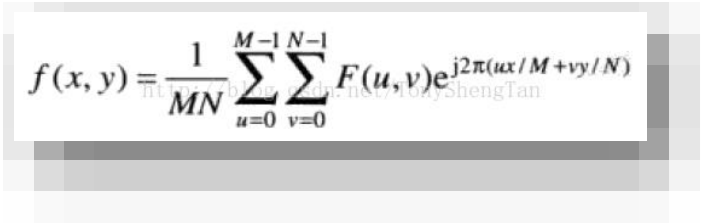
```
public static Complex computeDFT(int[][] grayImage, int u, int v) {
  int M = grayImage.length;
  int N = grayImage[0].length;
  Complex c = new Complex(0, 0);
  for (int x = 0; x < M; x++) {
    for (int y = 0; y < N; y++) {
      double a = (double)(u * x) / (double)M;
      double b = (double)(v * y) / (double)N;
      Complex temp = new Complex(0, -2 * Math.PI * (a + b));
      c = c.plus(temp.exp().times(grayImage[x][y]));
    }
  }

  return c;
}
```

代码简单地进行逐一的变换，所需要的操作是$(M*N)^2$ 因此效率较低。

要得到频谱图像，比较关键一点的操作是对傅里叶变换的数据进行缩小范围和标定的操作。我用到的是取 log 的方法缩小数据范围，同时进行 0-255 范围的量化。如下所示：

第一步：对数据进行取模操作
第二步：对取模数据进行 log 操作
第三步：对第二步的数据量化到 0-255 区域

同理，IDFT 的变换公式如下：



$$f(x,y)=\frac{1}{MN}\sum_{u=0}^{M-1}\sum_{v=0}^{N-1}F(u,v)e^{j2\pi(ux/M+vy/N)}$$

进行类似的编码，可以得到：

```
public static Complex computeIDFT(Complex[][] g, int x, int y) {
  int height = g.length;
  int width = g[0].length;
  Complex c = new Complex(0, 0);

  for (int u = 0; u < height; u++) {
    for (int v = 0; v < width; v++) {
      double a = (double)(u * x) / (double)height;
      double b = (double)(v * y) / (double)width;
      Complex temp = new Complex(0, 2 * Math.PI * (a + b));
      c = c.plus(temp.exp().times(g[u][v]));
    }
  }

  return c.times(1d / (double)(height * width));
}
```

同样也是简单地对公式的转换操作。

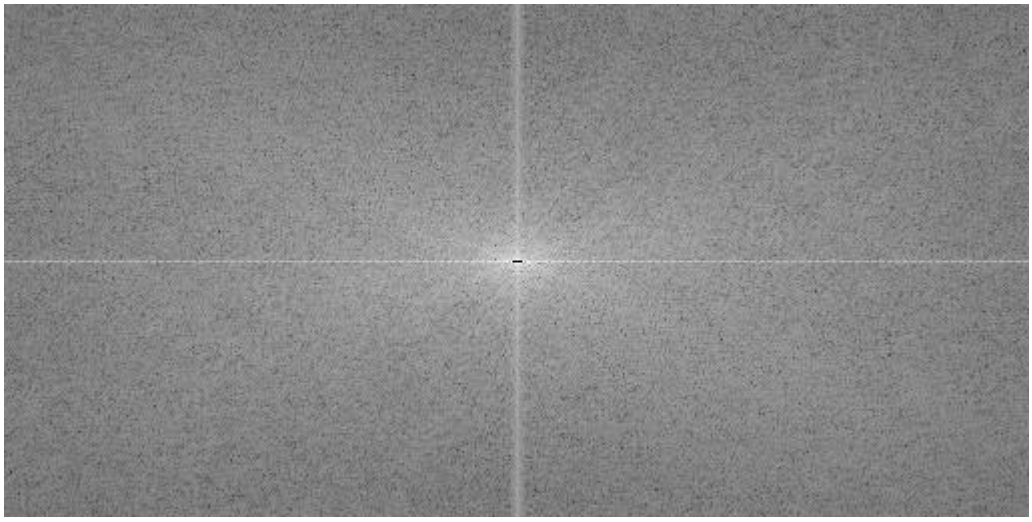
而进行 IDFT 操作后，需要注意的一点是，需要对所得到的数据进行取实部的操作，这是因为忽略由于计算不准确导致的计算分量。

```
// fifth: 取实部
for (int x = 0; x < height; x++) {
  for (int y = 0; y < width; y++) {
    last[x][y] = (int)destComplexs[x][y].getR();
  }
}
```

最后，如果图像在处理前有用 0 填充，需要进行裁剪的操作。

2.3 Bonus: Fast Fourier Transform (50 Points)

1. FFT



2. IFFT

3. 解释

对比 DFT，利用 FFT 进行傅里叶变换操作的时候，带来的速度上的提升确实是能够明显感受得到的。

FFT：

FFT（快速傅里叶变换）本身就是离散傅里叶变换（Discrete Fourier Transform）的快速算法，使算法复杂度由原本的 O(N^2) 变为 O(NlogN)。

将 x(n)分解为偶数与奇数的两个序列之和，即

$$x(n)=x_1(n)+x_2(n)$$

x1(n)和 x2(n)的长度都是 N / 2，x1(n)是偶数序列，x2(n)是奇数序列，则

$$X(k)=\sum_{n=0}^{\frac{N}{2}-1}x_1(n)W_N^{2kn}+\sum_{n=0}^{\frac{N}{2}}x_2(n)W_N^{(2n+1)k} \quad (k=0, 1, \cdots, N-1)$$

所以

$$X(k)=\sum_{n=0}^{\frac{N}{2}-1}x_1(n)W_N^{2kn}+W_N^k\sum_{n=0}^{\frac{N}{2}-1}x_2(n)W_N^{2kn} \quad (k=0, 1, \cdots, N-1)$$

由于 $W_N^{2kn}=e^{-j\frac{2\pi}{N}2kn}=e^{-j\frac{2\pi}{N/2}kn}=W_{N/2}^{kn}$ ，则

$$X(k)=\sum_{n=0}^{\frac{N}{2}-1}x_1(n)W_{N/2}^{kn}+W_N^k\sum_{n=0}^{\frac{N}{2}-1}x_2(n)W_{N/2}^{kn}=X_1(k)+W_N^kX_2(k) \quad (k=0, 1, \cdots, N-1)$$

其中 X1(k)和 X2(k)分别为 x1(n)和 x2(n)的 N / 2 点 DFT。由于 X1(k)和 X2(k)均以 N / 2 为周期，且 WN k+N/2=-WN k，所以 X(k)又可表示为：

$$X(k)=X_1(k)+W_N^kX_2(k) \quad (k=0, 1, \cdots, N/2-1)$$

$$X(k+\frac{N}{2})=X_1(k)-W_N^kX_2(k) \quad (k=0, 1, \cdots, N/2-1)$$

上式的运算可以用图 2 表示，根据其形状称之为蝶形运算。依此类推，经过 m-1 次分解，最后将 N 点 DFT 分解为 N / 2 个两点 DFT。图 3 为 8 点 FFT 的分解流程。

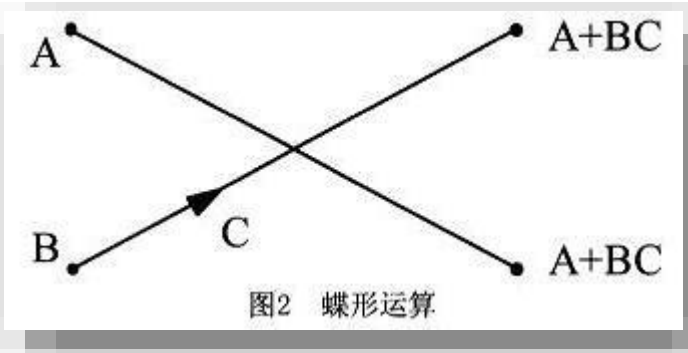


图2　蝶形运算

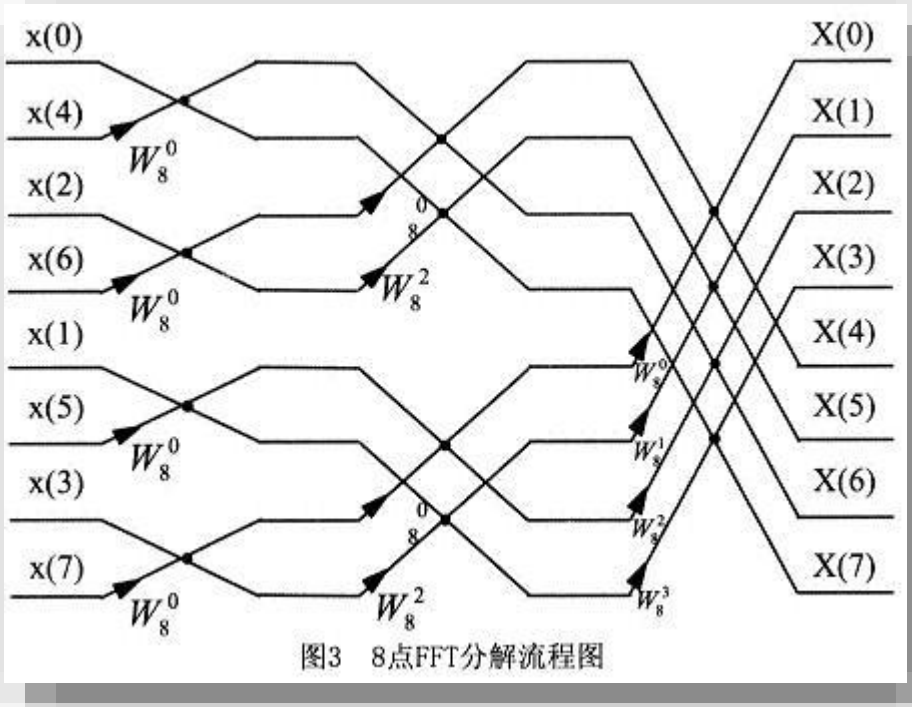
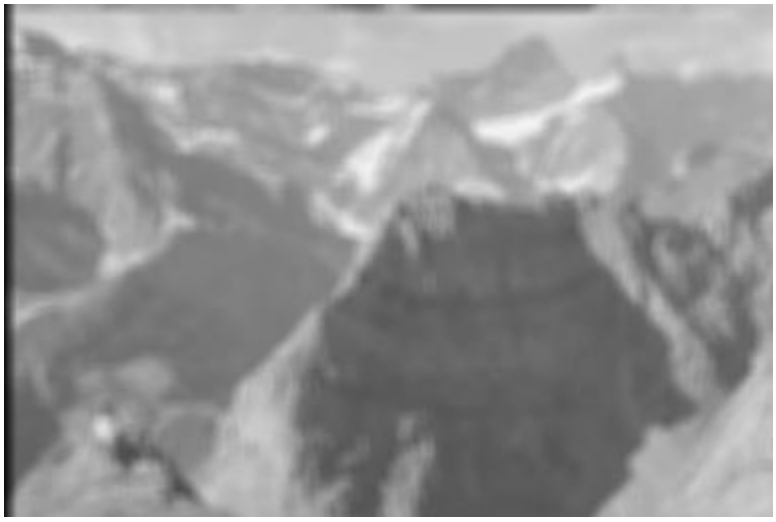


图3　8点FFT分解流程图

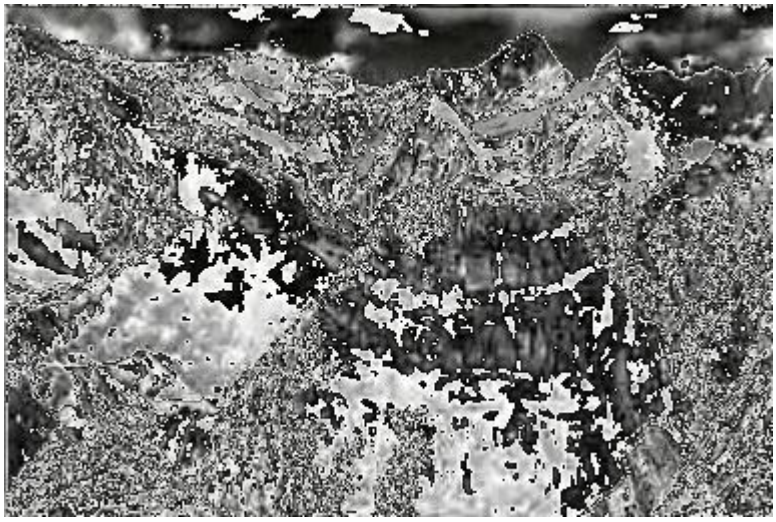
FFT 算法的原理是通过许多小的更加容易进行的变换去实现大规模的变换，降低了运算要求，提高了与运算速度。FFT 不是 DFT 的近似运算，它们完全是等效的。

2.4 Filtering in the Frequency Domain (30 Points)

1. 7*7 均值滤波



2. 拉普拉斯滤波



3. 解释

滤波处理步骤：

1） 取得给定图像的空间域信息 f(x, y)，得到 2 的幂次方的填充参数 height，width

2） 得到填充数组

3） $(-1)^{x+y}$ 乘以 f(x, y)进行中心化

4） 为了提高速度，计算图像的 FFT，得到 F(u, v)

5） 得到处理后的图像

$G(x, y) = \{real[f^{-1}(H(u, v)*F(u, v))]\}*(-1)^{x+y}$

6） 提取图像大小的区域，得到最终结果 g(x, y)

首先，获得填充参数如以下算法所示：

```
// 根据图像的长获得2的整数次幂
public static int get2PowerEdge(int e) {
  if (e == 1) return 1;
  int cur = 1;
  while(true) {
    if (e > cur && e <= 2 * cur) return 2*cur;
    else cur *= 2;
  }
}
```

第二，是滤波器 H(u, v)的获取：从空间模板获得频率域的滤波器：

```
public static Complex[][] getLaplaceFilter(int height, int width) {
  // 构造原始滤波函数
  Complex[][] filter = new Complex[height][width];

  //下面这个是拉普拉斯滤波
  filter[0][0] = new Complex(0, 0);
  filter[0][1] = new Complex(-1, 0);
  filter[0][2] = new Complex(0, 0);
  filter[1][0] = new Complex(-1, 0);
  filter[1][1] = new Complex(4, 0);
  filter[1][2] = new Complex(-1, 0);
  filter[2][0] = new Complex(0, 0);
  filter[2][1] = new Complex(-1, 0);
  filter[2][2] = new Complex(0, 0);

  for (int x = 0; x < height; x++)
    for (int y = 0; y < width; y++)
      if (x < 3 && y < 3) {}
      else filter[x][y] = new Complex(0, 0);

  filter = FFT.fft_2d(filter);

  return filter;
}
```

将得到的滤波器和原图像的傅里叶变换进行点乘，就可以完成类似于空间域滤波的效果。

```
public static Complex[][] performFilter(Complex[][] fourierComplexs, Complex[][] filter) {
  int height = fourierComplexs.length;
  int width = fourierComplexs[0].length;

  Complex[][] filteredImage = new Complex[height][width];
  for (int x = 0; x < height; x++) {
    for (int y = 0; y < width; y++) {
      filteredImage[x][y] = filter[x][y].times(fourierComplexs[x][y]);
    }
  }

  return filteredImage;
}
```

最后将处理结果进行 ifft，取实部后进行图像裁剪即可：

```
// 快速一维傅里叶逆变换
public static Complex[] ifft_1d(Complex[] x) {
  int N = x.length;
  Complex[] y = new Complex[N];

  // take conjugate
  for (int i = 0; i < N; i++) {
    y[i] = x[i].conjugate();
  }

  // compute forward fft
  y = fft_1d(y);

  // take conguate again
  for (int i = 0; i < N; i++) {
    y[i] = y[i].conjugate();
  }

  // divide by N
  for (int i = 0; i < N; i++) {
    y[i] = y[i].times(1.0/N);
  }

  return y;
}
```