Scientific Computing and Simulation, Lab 1: Fourier Tranforms for Image Filtering

This week and the next couple of weeks we will be looking at the uses of Fourier analysis in filtering and reconstructing images - I hope to cover examples from medical and astronomical imaging.

In this first week I will ask you to code up a naive two-dimensional Discrete Fourier Transform based on formulae in the lecture, and apply it to simple filtering of a sample image.

Some Support Classes

To give you a chance of completing this task in the time we have in the lab, I will provide a few support classes. The code in these is mostly fairly short and shouldn't be hard to follow, but it would probably take some hours to code these things from scratch.

First, the class ReadPGM has a static read() method for reading an image file in PGM format. For simplicity it only works with a square image of given size, and puts it's output in a given double [ ] [ ] of greyscale values. See the main method of this class for example usage of the read() method.

The class DisplayDensity displays a grey-scale representation of a "density" given in a double [ ] [ ] array. By default, the smallest value in this array will be scaled to display as black and the largest as white.

The main method of ReadPGM also uses DisplayDensity to display the read image, so you can immediately test both these classes by downloading them and running ReadPGM. You will also need a PGM image to read, and there is one in this file: wolf.pgm. To make thngs easier, ReadPGM by default reads from a file called "wolf.pgm", but you may have to experiment with where you place this file in your folder hierarchy - it depends on the Java development environment. It needs to be in the working directory of the running Java program (alternatively edit the code to give an absolute path name.)

The wolf image is quite small. You will soon discover why!

The final support class I am going to provide you with is Display2dFT. This is designed to represent 2D Discrete Fourier Transforms. The k,l components of these are complex numbers, so the input to this class is two double [ ] [ ] arrays - one of real parts and one of imaginary parts. The actual display is quite hard to interpret because there isn't an obvious way of representing complex numbers as pixels of an image. It uses a colour coding to represent the "direction" of the complex number in the argand diagram. The absolute size of the complex number is displayed through brightness of the colour, using a logarithmic scale, because this seems to work reasonably well for Fourier transforms.

We will illustrate the use of Display2dFT shortly.

Doing a Fourier Transform

To save more time I will give you the skeleton of a program to perform a Discrete Fourier Transform:

```
public class SimpleFT {

    public static int N = 256 ;

    public static void main(String [] args) throws Exception {

        double [] [] X = new double [N] [N] ;
        ReadPGM.read(X, "wolf.pgm", N) ;

        DisplayDensity display =
                new DisplayDensity(X, N, "Original Image") ;

        double [] [] CRe = new double [N] [N], CIm = new double [N] [N] ;

        for(int k = 0 ; k < N ; k++) {
            for(int l = 0 ; l < N ; l++) {
```

```
                    double sumRe = 0, sumIm = 0 ;
                    // Nested for loops performing sum over X elements
                    for(int m = ...) {
                        for(int n = ...) {
                            double arg = ... ;
                            double cos = ... ;
                            double sin = ... ;
                            sumRe += cos * X [m] [n] ;
                            sumIm += ... ;
                        }
                    }
                    CRe [k] [l] = sumRe ;
                    CIm [k] [l] = sumIm ;
                }
                System.out.println("Completed FT line " + k + " out of " + N) ;
            }

            Display2dFT display2 =
                    new Display2dFT(CRe, CIm, N, "Discrete FT") ;
        }
    }
```

Try to fill in the omitted code based on the first equation in slide 34 of this week's lecture. It involves two nested loops over *m* and *n* to do the sums in the equation over all elements of the input pixels in X. Inside the loop you will need to calculate the argument $-2\pi(mk + nl)/N$ of the imaginary exponential, and then use the Java sin and cos functions to calculate the exponential according to the famous formulae on slide 27 (what effect does the minus sign in the exponent have here?) Multiplication of the

resulting complex number by X components is easy, because X is a real number - it just scales the real and imaginary parts independently.

If you get this right, the program is likely to take a few minutes to run - a problem we will fix in later weeks by using the Fast Fourier Transform algorithm.

Inverse Fourier Transform

Next we will confirm that we can recover exactly the original image by inverting the transform on the represented in coefficients C.

Add this skeleton code to the end of the main method above:

```
double [] [] reconstructed = new double [N] [N] ;


for(int m = 0 ; m < N ; m++) {
    for(int n = 0 ; n < N ; n++) {
        double sum ;
        ... nested for loops performing sum over C elements
        reconstructed [m] [n] = sum ;
    }
    System.out.println("Completed inverse FT line " + m + " out of " + N) ;
}


DisplayDensity display3 =
        new DisplayDensity(reconstructed, N, "Reconstructed Image") ;
```

According to the second formula on slide 34, the code in the body of the nested loops over *k* and *l* will have quite a similar structure to the loop you wrote previously except:

- There is no minus sign in the argument of the imaginary exponential.
- In the earlier loop we multiplied the imaginary exponential by a real number to obtain a complex number; here we will be multiplying together *two* complex numbers, though we will ignore the imaginary part of the result. Actually the

imaginary part should be zero, so we only need to calculate the real part.

You can use the formula at the bottom of slide 26 to do this multiplication - the product of complex numbers *a + bi* and *c + di* has a real part of the form *ac - bd*, so a term in the sum will have a structure like cos * CRe - sin * CIm. But you need to fill in arguments and subscripts!

Filtering Images with Fourier Transforms

Many kinds of filter can be efficiently applied to images using Fourier Transformed, including sophisticated filters based on *convolutions*. For the purposes of this lab we consider a very simple kind of "low pass filter", where we simply omit Fourier components with large wave numbers before reconstructing the image.

In between the two code segments you wrote above, try adding these lines:

```
int cutoff = N/8 ;    // for example
for(int k = 0 ; k < N ; k++) {
    int kSigned = k <= N/2 ? k : k - N ;
    for(int l = 0 ; l < N ; l++) {
        int lSigned = l <= N/2 ? l : l - N ;
        if(Math.abs(kSigned) > cutoff || Math.abs(lSigned) > cutoff) {
            CRe [k] [l] = 0 ;
            CIm [k] [l] = 0 ;
        }
    }
}

Display2dFT display2a =
        new Display2dFT(CRe, CIm, N, "Truncated FT") ;
```

For simplicity in the above code, we have applied Display2dFT twice to the same array,

and modified the array in between, which may lead to some odd effects - hopefully they won't be confusing.

You should also try implementing a high pass filter, for example by inverting the test condition in the if statement above.

By now you will see quite obviously why we only considered a small image in this lab, and may be doubtful about the claim that the DFT can be used to filter images "efficiently"! Hopefully these doubts will be resolved when we go on to consider the Fast Fourier Transform.