Scientific Computing and Simulation, Lab 2: The Fast Fourier Transform

In the next couple of weeks I want to look at the uses of Fourier analysis in filtering and reconstructing images - I hope to cover examples from medical and astronomical imaging.

This week as a preliminary we will be looking at the Fast Fourier Transform, and I will ask you to apply this to same image filtering task that took so long to process last week, when we using a naive implementation of the Discrete Fourier Transform.

Implementation of an FFT

Although it shouldn't be too hard to implement a recursive version of the Fast Fourier Transform, based for example on the formula on slide 15 of this week's lecture, it is a little more challenging to implement a fully optimized version of the FFT. In particular, for efficiency, it is preferable to unpick the recursion and produce a purely iterative version of the same mathematics given in the slides.

So instead I will provide you with the full Java code of a class FFT here. Download it and import it into Netbeans or your Java development environment of choice.

The principal method of the class FFT is a static method called fft1d, and the code of that method is reproduced here:

```
public static void fft1d(double [] re, double [] im, int isgn) {

    // One-dimensional FFT, or inverse FFT (in-place algorithm).

    // When this method is called, the arrays re and im should contain
    // the real and imaginary parts of the input data.

    // When this method returns the values in these arrays are
```

```
// are overwritten with the real and imaginary parts of the

// transformed data.


// isgn = +1 or -1 for forward or inverse transform.


// Size of arrays should be a power or two.


final double pi = Math.PI ;


final int N = re.length ;    // im better be the same size


bitReverse(re, im) ;


int ln2     = ilog2(N)    ;    // Base 2 log of the leading dimension.


// Danielson-Lanczos algorithm for FFT.


for(int ilevel = 1 ; ilevel <= ln2 ; ilevel++) {
      int le      = ipow(2,ilevel) ;
      int lev2 = le / 2 ;


      double uRe = 1.0F ;
      double uIm = 0.0F ;


      double wRe = Math.cos(isgn * pi / lev2) ;
      double wIm = Math.sin(isgn * pi / lev2) ;
```

```
            for(int jj = 0 ; jj < lev2 ; jj++) {
                for(int ii = jj ; ii < N ; ii += le) {
                    int jndex = ii + lev2 ;
                    int index = ii ;

                    //tmp           = u * a(jndex) ;
                    double tmpRe = uRe * re [jndex] - uIm * im [jndex] ;
                    double tmpIm = uRe * im [jndex] + uIm * re [jndex] ;

                    //a(jndex) = a(index) - tmp ;
                    re [jndex] = re [index] - tmpRe ;
                    im [jndex] = im [index] - tmpIm ;

                    //a(index) = a(index) + tmp ;
                    re [index] = re [index] + tmpRe ;
                    im [index] = im [index] + tmpIm ;
                }
                //tmp = u * w ;
                double tmpRe = uRe * wRe - uIm * wIm ;
                double tmpIm = uRe * wIm + uIm * wRe ;

                //u     = tmp ;
                uRe     = tmpRe ;
                uIm     = tmpIm ;
            }
        }
    }
```

Barring changes to names of variables, and the need in Java to unbundle complex numbers into real and imaginary parts, this is *almost* identical to pseudocode that at the time of writing is available on Wikipedia here (under the heading "Data reordering, bit reversal, and in-place algorithms" of the article "Cooley–Tukey FFT algorithm").

One minor change is that in our version the nesting of the two innermost loops is inverted. Amongst other things this makes our code slightly more economical in terms of computation of the twiddle factors. A notable feature of these codes is that the number of imaginary exponentials (or trig functions) that have to be computed is quite minimal - one just calculates the $\log_2 N$ imaginary exponentials with arguments minus or plus $2i\pi/N$, $4i\pi/N$, $8i\pi/N$, and so on, up to $i\pi/2$. These are composed together using complex multiplication to get all other twiddle factors.

(By the way I didn't write this Java code myself from scratch. I inheritted a FORTRAN FFT benchmark code quite a few years ago, and transcribed that to Java for some demo. There's nothing wrong with code reuse!)

Another feature of the code here is that it works "in place". The same pair or arrays is used for the inputs or the outputs, and the computed values of the Fourier transform replace the values of the real space data in these arrays as the computation unfolds (or vice versa for an inverse transform).

In place algorithms are popular and valued in large scale computation, because they can be used to save memory. In our setting they mean we do often have to define new arrays before calling the methods, and copy the input data to the new arrays before calling the method.

Lab Work - Using the FFT

Last week we performed a two dimensional Discrete Fourier Transform on some image data, applied a trivial filter in Fourier space, then performed an inverse DFT to recover the filtered image.

This week I want you to do the same, using FFTs.

Above I have given you the code for a *one dimensional FFT*; I will leave it as an exercise to you to complete the two dimensional FFT based on this code.

This should be simpler than it may sound, using the observation on slide 16 of the lecture that you can obtain an FFT of two dimensional data just by performing a one dimensional FFT on all rows of the data, followed by a one dimensional FFT on all columns of the data.

Operationally, thinking of the two dimensional data as a (complex) matrix, it is easier to do the FFT on all all rows, transpose the matrix to swap rows and columns, do FFTs on all *rows* again, then transpose the matrix back.

Like last week, let me give you the skeleton code for a Fourier Transform and its inverse:

```
public class FFTImageFiltering {

    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") ;

        // create array for in-place FFT, and copy original data to it

        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++) {

                CRe [k] [l] = X [k] [l] ;

            }

        }

        fft2d(CRe, CIm, 1) ;    // Fourier transform

        Display2dFT display2 =

                new Display2dFT(CRe, CIm, N, "Discrete FT") ;

        // create array for in-place inverse FFT, and copy FT to it

        double [] [] reconRe = new double [N] [N],

                        reconIm = new double [N] [N] ;

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

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

                reconRe [k] [l] = CRe [k] [l] ;
```

```
                reconIm [k] [l] = CIm [k] [l] ;
            }
        }
        fft2d(reconRe, reconIm, -1) ;    // Inverse Fourier transform
        DisplayDensity display3 =
                new DisplayDensity(reconRe, N, "Reconstructed Image") ;
    }
    ... implementation of fft2d ...
}
```

You "just" need to implement the static method fft2d, which is an in place implementation of a two dimensional FFT. The transform is controlled by the final integer parameter - plus 1 means the forward transform and -1 means the inverse transform.

Don't panic! I have given you the non-trivial code for this, and according to the discussion above the most challenging part remaining is probably implementing an in place matrix transpose!

I will even give you the general structure of that:

```
    static void transpose(double [] [] a) {
        for(int i = 0 ; i < N ; i++) {
            for(int j = 0 ; j < i ; j++) {
                ... swap values in a [i] [j] and a [j] [i] elements ...
            }
        }
    }
```

Now you need to complete implemention of this method:

```
    static void fft2d(double [] [] re, double [] [] im, int isgn) {
        // For simplicity, assume square arrays
        ... fft1d on all rows of re, im ...
```

```
        transpose(re) ;
        transpose(im) ;
        ... fft1d on all rows of re, im ...
        transpose(re) ;
        transpose(im) ;
    }
```

Remember that in Java the *i*th row of a matrix, represented by a double [] [] array a, is just the one dimensional array a [*i*], which has type double []. Use this to separately extract rows of real and imaginary arrays and pass them as arguments to fft1d, in a loop over *i*.

You should notice that the FFT version of the code is dramatically faster than the original naive implementation.

For a fully optimized production quality library for performing Fourier transforms in "the wild", see, for example, The Fastest Fourier Transform in the West.

Exercises

1. Benchmark your new implementation of the two dimensional Fourier Transform and compare it with the naive code you produced before.
2. Complete the exercises suggested last week - experimenting with the effects of different kinds of low pass and high pass filters on the image.
3. (Optional) Experiment with larger images (which you will have to source and load into the program). Please note that the image dimension (N) should be a square number. If it is not the case, the output of fft1d() won't be correct.
4. (Optional) How could you make a parallel version of the two dimensional FTT?