ECE2036 **Fall Semester, 2012**

The Fast Fourier Transform

Assigned: Oct 3, 2012 *Due: Oct 12, 2012, 11:59pm*

Given an array of length N containing complex discrete–time samples of some signal h, the *Discrete Fourier Transform* (*DFT*) H is also an array of complex values of length N, defined as:

$$H[n] = \sum_{k=0}^{N-1} W^{nk} h[k] \qquad \text{where } W = e^{-j2\pi/N} = \cos(2\pi/N) - j\sin(2\pi/N) \qquad \text{where } j = \sqrt{-1} \quad (1)$$

For all equations in this document, we use the following notational conventions. h is the discrete–time sampled signal array. H is the Fourier transform array of h. N is the length of the sample array, and is always assumed to be an even power of 2. n is an index into the h and H arrays, and is always in the range $0 \ldots (N-1)$. k is also an index into h and H, and is the summation variable when needed. j is the square root of negative one.

An important special case of the above equation is the case of a sample of length 1. Since the summation variable k is exactly 0, the W^{nk} term becomes 1, and thus $H[0] = h[0]$. The Fourier transform of a sample set of length 1 is just the original sample set unmodified.

From equation 1, it is clear that to compute the DFT for a sample of length N, it must take N^2 operations, since to compute each element of H requires a summation overall all samples in h. It is easy to see how to implement this algorithm with a simple set of nested `for` loops, computing the value of $H[n]$ for each n by iterting over and summing each $h[h]$ multiplied by the appropriate W^{nk}. It is equally clear that as the input array size gets large, say for example 2^{16}, it would take 2^{32} calculations to compute the transform, which will likely be somewhat time consuming on the computing device.

However, Danielson and Lanczos demonstrated a method that reduces the number of operations from N^2 to $N\log_2(N)$. The insight of Danielson and Lanczos was that the FFT of an array of length N is in fact the sum of two smaller FFT's of length $N/2$, where the first half–length FFT contains only the even numbered samples, and the second contains only the odd numbered samples. The proof of the *Danielson–Lanczos Lemma* is quite simple, as follows:

$$\begin{aligned}
H[n] &= \sum_{k=0}^{N-1} e^{-j2\pi kn/N} h[k] \\
&= \sum_{k=0}^{N/2-1} e^{-j2\pi 2kn/N} h[2k] + \sum_{k=0}^{N/2-1} e^{-j2\pi(2k+1)n/N} h[2k+1] \\
&= \sum_{k=0}^{N/2-1} e^{-j2\pi kn/(N/2)} h[2k] + \sum_{k=0}^{N/2-1} e^{-j2\pi kn/(N/2)} e^{-j2\pi n/N} h[2k+1] \\
&= \sum_{k=0}^{N/2-1} e^{-j2\pi kn/(N/2)} h[2k] + W^n \sum_{k=0}^{N/2-1} e^{-j2\pi kn/(N/2)} h[2k+1] \\
H[n] &= H^e_{n \text{ mod (N/2)}} + W^n H^o_{n \text{ mod (N/2)}}
\end{aligned} \qquad (2)$$

where H^e_n refers to the n^{th} element of the Fourier transform of length $N/2$ formed from the even numbered elements of the original length N samples, and H^o_n refers to the n^{th} element of the odd numbered samples. Thus the Fourier transform of an array of length N can be computed by summing two transforms of length $N/2$. Then each of the $N/2$ transforms can be computed as the sum of two transforms of length $N/4$, which in turn can be computed as the sum of two transforms of length $N/8$. This process can be repeated until we have a transform of length 1, which we already know can trivially be computed.

Using the equation for the Fourier transform and the Danielson–Lanczos lemma, we can design an easy–to–implement and efficient algorithm for computing the Discrete Fourier Transform for a set of signal samples of length $N = 2^m$. This algorithm is known as the Cooley-Tukey algorithm. By using the Cooley-Tukey algorithm, we can reduce the computational complexity of the DFT computation from N^2 operations to $N\log_2(N)$

Step 1. Reordering the initial h array. Consider a simple case of a sample set of length 8, $h[0], h[1] \dots h[7]$. Dividing this into the even samples and odd samples, we get:

$$\begin{aligned} H^e &= h[0] + h[2] + h[4] + h[6] \\ H^o &= h[1] + h[3] + h[5] + h[7] \end{aligned} \tag{3}$$

Further dividing the even set into it's even and odd components, and similarly dividing the odd set into it's even and odd components, we get:

$$\begin{aligned} H^{ee} &= h[0] + h[4] \\ H^{eo} &= h[2] + h[6] \\ H^{oe} &= h[1] + h[5] \\ H^{oo} &= h[3] + h[7] \end{aligned} \tag{4}$$

Finally, dividing each of these into it's even and odd components we get:

$$\begin{aligned} H^{eee} &= h[0] \\ H^{eeo} &= h[4] \\ H^{eoe} &= h[2] \\ H^{eoo} &= h[6] \\ H^{oee} &= h[1] \\ H^{oeo} &= h[5] \\ H^{ooe} &= h[3] \\ H^{ooo} &= h[7] \end{aligned} \tag{5}$$

Since each line of equation 5 is just a Fourier transform of length one, we can trivially compute each of these. The question now becomes is there an easy way to determine which of the $h[n]$ values corresponds to each of the H^{xxx} components. In other words, in equation 5 we determined that H^{eoo} (for example) is equal to $h[6]$, but is there a general way to find this mapping? It turns out that there is, by simply assigning a 0 value to each e (even iteration) and and 1 values to each o (odd iteration), and then interpreting the resulting binary value *in reverse order*. In our example of H^{eoo}, the binary value is 011, which is the value 6 when read from right to left. The first step of the Cooley–Tukey algorithm is to simply transform the original h array from natural ordering ($h[0], h[1], h[2], \dots h[N-1]$) to the bit reversed ordering. For an original vector of length 8, the reverse ordering is the sequence shown in equation 5. For original array lengths other than 8, the reversed orderings are of course different, but always easy to compute.

After reordering the original h array, we can easily compute the four sets of Fourier transforms of length 2, since the required elements are adjacent in the reordered array. Referring to equation 4, we see that the *ee* values are $h[0]$ and $h[4]$, which are the first two elements in the reordered array; the *eo* values are $h[2]$ and $h[6]$ which are the next two, and so on. Similarly we can compute the two sets of Fourier transforms of length 4, since the *e* elements are the first four and the *o* elements are the next four. Thus the bookkeeping needed for determining which elements of the h array are needed at each step is quite simple.

Step 2. Precomputing the W^n values. Recall the definition of the complex *Weight* factor from equation 1 is:

$$\begin{aligned} W &= e^{-j2\pi/N} = \cos(2\pi/N) - j\sin(2\pi/N) \\ W^n &= e^{-j2\pi n/N} = \cos(2\pi n/N) - j\sin(2\pi n/N) \end{aligned} \tag{6}$$

Further recall that we need the value W^n in equation 2 to combine the *even* and *odd* sub–transforms. Since n is the array index in the original h array (of length N), there are exactly N distinct W^n values ($W^0, W^1, \dots W^{N-1}$). Since these are somewhat expensive to compute (two trigonometric functions), we can save some time by precomputing the N distinct weights. As it turns out, we in fact only need to compute the first half of these weights. For any W^n, we can show that:

$$W^{n+N/2} = -W^n \tag{7}$$

The proof of this identity is trivial, and is left as an exercise for the reader. Using this identity, we just need to compute W^n for $n = [0, 1, \dots (N/2-1)]$.

Step 3. Do the transformation. Once the input array is re–ordered in the bit reversed order and the W values are computed, we can easily perform the transform by starting with a two-sample transform of side–by–side elements in the shuffled array, then computing a four–sample transform with four adjacent elements, continuing until we have an N element transform. The problem is complicated by the fact that we want an *in–place* transformation. We *do not* have a separate H array to store the transformed data; rather the original h array is over–written with the computed H elements.

In our example, we would first do a two-sample transform for each of the four sets of two points: {H[0], H[1]}, {H[2], H[3]}, {H[4], H[5]}, {H[6], H[7]},

Note that above we refer to the elements with the symbol H rather than h, since each of the elements are the result of a one–point transform which we already know is simply the point unmodified. For the first two point transform, from equation 2 we can see that the first set would be:

$$\begin{aligned} H[0] &= H^e_{0 \text{ mod (2/2)}} + W^0_2 H^o_{0 \text{ mod (2/2)}} &&= H[0] + W^0_2 H[1] \\ H[1] &= H^e_{1 \text{ mod (2/2)}} + W^1_2 H^o_{1 \text{ mod (2/2)}} &&= H[0] + W^1_2 H[1] \end{aligned} \tag{8}$$

although we have to very careful about the W values above. To clarify these weights, we use a new notational convention W^k_N. The subscript N indicates the number of points in the transform and the superscript is of course the power. Recall that we pre–computed the weights in step 2 above, but these precomputed weights were for a eight point transform (in our example). In this step we are doing a two point transform, which apparently will result in different weight factors (since the definition of W in equation 1 has N in the definition). Given this, it appears that we have to re–compute the weights for each transform size $N = 2, 4, ...$ Luckily, this is not the case. If we start with weights computed for an N point transform we can prove that for a x point transform (for all x where $x^m = N$ for some $m > 0$),

$$W^k_x = W^{kN/x}_N \tag{9}$$

The proof of this is left as an exercise. Returning to our example, since we have pre–computed weights for $N = 8$, we use:

$$\begin{aligned} W^0_2 &= W^{0(8/2)}_8 = W^0_8 \\ W^1_2 &= W^{1(8/2)}_8 = W^4_8 = -W^0_8 \end{aligned} \tag{10}$$

Therefore, assuming we stored our pre–computed weights in array W, for the 2–point transform we end up with:

$$\begin{aligned} H[0] &= H^e_{0 \text{ mod (2/2)}} + W^0_2 H^o_{0 \text{ mod (2/2)}} &&= H[0] + W^0_2 H[1] = H[0] + W[0]H[1] \\ H[1] &= H^e_{1 \text{ mod (2/2)}} + W^1_2 H^o_{1 \text{ mod (2/2)}} &&= H[0] + W^1_2 H[1] = H[0] - W[0]H[1] \end{aligned} \tag{11}$$

We would have similar equations for the other three sets of two–point transforms. Continuing to the four–point transforms, we end up with (for the first set of four for example):

$$\begin{aligned} H[0] &= H^e_{0 \text{ mod (4/2)}} + W^0_4 H^o_{0 \text{ mod (4/2)}} &&= H[0] + W^0_8 H[2] = H[0] + W[0]H[2] \\ H[2] &= H^e_{2 \text{ mod (4/2)}} + W^2_4 H^o_{2 \text{ mod (4/2)}} &&= H[0] + W^4_8 H[2] = H[0] - W[0]H[2] \\ H[1] &= H^e_{1 \text{ mod (4/2)}} + W^1_4 H^o_{1 \text{ mod (4/2)}} &&= H[1] + W^2_8 H[3] = H[1] + W[2]H[3] \\ H[3] &= H^e_{3 \text{ mod (4/2)}} + W^3_4 H^o_{3 \text{ mod (4/2)}} &&= H[1] + W^6_8 H[3] = H[1] - W[2]H[3] \end{aligned} \tag{12}$$

In coding this, we must be careful in that we are using a *transform–in–place*, meaning the output array H is in fact the input array h. Notice that in the above equations, we overwrite $H[0]$ in the first equation, but use it on the right–hand–side in later equations. This means we likely would need one or more temporary variables store the original values inside of this processing loop.

We continue transforming larger and larger sample sets (increasing by a factor of two each time) until our final transform is all N samples and the problem is solved.

Details of the Assignment For this assignment, you need to implement both a *simple* approach, just implementing Equation (1) above with two nested loops. Then implement the more efficient Danielson/Lanczos approach and observe the same answers and a reduced execution time. These are both defined in the skeleton `fft.h` provided.

Resources. There are a number of files that are given to you. These are all on the `jinx-login.cc.gatech.edu` system, and you will copy these to your own directory (instructions as to how to do this are below). In particular see the output operator (<<) for `Complex` objects. This is given to you to insure the outputs are in scientific notation, as several of the results are somewhat small and would print as zero unless we use scientific notation. You should cut-paste the output operator declaration (in `add-to-complex.cc`) to your `complex.h` and the operator implementation to your `complex.cc`.

Copying the Project Skeletons

1. Log into `jinx-login.cc` using `ssh` and your prism log-in name.
2. Copy the files from the ECE2036 user account using the following command:

   ```
   /usr/bin/rsync -avu /nethome/ECE2036/FourierTransform .
   ```

   Be sure to notice the period at the end of the above command.
3. Change your working directory to `FourierTransform`

   ```
   cd FourierTransform
   ```
4. Copy the provided skeleton files as follows:

   ```
   cp fft-skeleton.cc fft.cc
   cp complex-skeleton.h complex.h
   cp complex-skeleton.cc complex.cc
   ```
5. First create your implementation of the `Complex` class by editing `complex.h` and `complex.cc`.
6. Then edit `fft.cc` to implement your transforms.
7. Compile your code using `make` as follows:

   ```
   make
   ```
8. Once you have implemented the transforms, you can test your solution with the provided inputs. Also you can "time" your implementation (find out how long it takes to execute) using the `time` command as follows:

   ```
   time ./fft in_65k.txt simple
   time ./fft in_65k.txt
   ```
9. The second argument (simple) above will use the simple, inefficient implementation. Leaving this off will use the efficient Danielson/Lanczos approach. You can observe the difference in execution time.

Turning in your Project. The system administrator for the jinx cluster has created a script that you are to use to turn in your project. The script is called `riley-turnin` and is found in `/usr/local/bin`, which should be in the search path for everyong. From your **home directory** (not the FourierTransform subdirectory), enter:

```
riley-turnin FourierTransform.
```

This automatically copies everything in your `FourierTransform` directory to a place that I can access (and grade) it.