Supplementary Notes for ELEN 4810 Lecture 9
The Fast Fourier Transform

John Wright
Columbia University
October 19, 2015

Disclaimer: These notes are intended to be an accessible introduction to the subject, with no pretense at completeness. In general, you can find more thorough discussions in Oppenheim’s book. Please let me know if you find any typos.

Reading suggestions: Oppenheim-Schafer Sections 9.1-9.3

Homework: HW4 is due Wednesday, October 21.

In the next two lectures, we will continue to expand our toolbox for computing in frequency domain, by adding two important computational tools. The first, the Fast Fourier Transform (FFT) is a family of algorithms for efficiently computing the Discrete Fourier Transform. These algorithms are of great practical importance, as they make it possible to perform frequency domain analysis of very large signals. With this computational tool in hand, we will then describe the Short-Time Fourier Transform, which makes it possible to produce signal representations which are localized in both time and frequency, enabling us to analyze signals whose frequency content changes over time.

1 Computing the DFT

In this lecture, our goal is to efficiently compute the Discrete Fourier Transform

$$X[k] = \sum_{n=0}^{N-1} x[n] \exp\left(-j \frac{2\pi kn}{N}\right) \quad k = 0, \ldots, N - 1.$$  \hspace{1cm} (1.1)

We denote the output sequence as $X = \text{DFT}_N \{x\}$, where $N$ is the length of the input sequence $x$. There is a very straightforward algorithm for computing the sequence $X$: simply use the formula (1.1)! Computing each entry $X[k]$ requires $N$ complex multiplications and $N - 1$ complex additions. We can compute the entire sequence $X$ by computing each element sequentially, using a total of $N^2$ complex multiplications and $N(N - 1)$ complex additions. Typical general purpose digital computers do not have dedicated hardware for complex arithmetic, and so, complex operations are performed using real arithmetic and additional logic. For example, a complex multiplication $(a + jb)(c + jd)$ can be computed using four real multiplications and two real additions. A complex addition can be computed using two real additions. So, if we want to be extremely precise,
the naive algorithm for (1.1) requires $4N^2$ real multiplications and $2N^2 + 2N(N - 1) = 4N^2 - 2N$
real additions. In analyzing algorithms, this kind of precision is typically overkill – we are happy
to know qualitatively how the computation grows with $N$. Because it requires $4N^2$ multiplications,
the naive algorithm takes time $O(N^2)$.\textsuperscript{1}

2 Fast Fourier Transform Algorithms

Fast Fourier Transform (FFT) algorithms fundamentally improve over the naive algorithm, reducing
the computational cost from $O(N^2)$ to $O(N \log N)$. In addition, the hidden constant in the $O(\cdot)$
notation is not too large, making the FFT (and its variants) the algorithm of choice for computing
the DFT in practice.

The core idea in the FFT algorithm is to reduce the computation of one length-$N$ DFT two the com-
putation of two length-$N/2$ DFT's, plus an additional $O(N)$ computation to combine them together.
This divide and conquer strategy can be iterated to reduce the computation of the DFT to successively
smaller DFT's (length $N/4$, $N/8$, $N/16$, ...), until one finally arrives at a base case (conceptually,
$N = 2$), which can be evaluated by direct computation. For notational simplicity, we describe these
procedures under the assumption that $N$ is a power of 2: $N = 2^p$ for some $p$.

There are two popular strategies for reducing a length-$N$ DFT to two length $N/2$ DFT's, known
as decimation in time and decimation in frequency.

2.1 Decimation in time

The decimation-in-time FFT architecture is a consequence of the following result, which shows that
we can compute DFT of an even-length sequence $x[n]$ by combining the DFT's of the even terms
$x_e[n] = x[2n]$ and odd terms $x_o[n] = x[2n + 1]$:

\textbf{Theorem 2.1.} Let $x[n]$ be a length-$N$ sequence, with $N$ an even integer. Let

\begin{align*}
x_e[n] &= x[2n], \quad n = 0, \ldots, N/2 - 1 \\
x_o[n] &= x[2n + 1], \quad n = 0, \ldots, N/2 - 1.
\end{align*}

Then

\begin{align*}
\text{DFT}_N \{x\} [k] &= \text{DFT}_{N/2} \{x_e\} \left[ k \mod \frac{N}{2} \right] + \exp \left( -j \frac{2\pi k}{N} \right) \text{DFT}_{N/2} \{x_o\} \left[ k \mod \frac{N}{2} \right].
\end{align*}

\textsuperscript{1}The “big-$O$” or Landau notation has the following meaning: we write $f(N) = O(g(N))$ if there exists a constant $C$
independent of $N$ such that for all $N$, $f(N) \leq C g(N)$.\textsuperscript{1}
Proof. We calculate

\[
\text{DFT}_N \{x\}[k] = \sum_{n=0}^{N-1} x[n] \exp \left( -j \frac{2\pi kn}{N} \right) \tag{2.4}
\]

\[
= \sum_{i=0}^{N/2-1} x[2i] \exp \left( -j \frac{2\pi k 	imes 2i}{N} \right) + \sum_{\ell=0}^{N/2-1} x[2\ell + 1] \exp \left( -j \frac{2\pi k \times (2\ell + 1)}{N} \right) \tag{2.5}
\]

\[
= \sum_{i=0}^{N/2-1} x_e[i] \exp \left( -j \frac{2\pi ki}{N/2} \right) + \exp \left( -j \frac{2\pi k}{N} \right) \sum_{\ell=0}^{N/2-1} x_o[i] \exp \left( -j \frac{2\pi ki}{N/2} \right) \tag{2.6}
\]

\[
= \text{DFT}_{N/2} \{x_e\} \left[ k \text{ mod } \frac{N}{2} \right] + \exp \left( -j \frac{2\pi k}{N} \right) \text{DFT}_{N/2} \{x_o\} \left[ k \text{ mod } \frac{N}{2} \right] \tag{2.7}
\]

This result immediately suggests an architecture for calculating the \(N\)-point DFT, which we draw as Figure 1: namely, we compute the \(N/2\)-point DFT of the event terms, the \(N/2\)-point DFT of the odd terms, and then combine them together. Each entry \(X[k]\) of the \(N\)-point DFT of \(x\) can be generated by adding an entry of \(\text{DFT}_{N/2} \{x_e\}\) with an entry of \(\text{DFT}_{N/2} \{x_o\}\), scaled by a complex exponential.

Let \(T_N\) be the number of arithmetic operations required to compute an \(N\)-point DFT. Using the architecture in Figure 1, the computational complexity is at most \(2T_{N/2} + CN\). The first term accounts for the cost of the two \(N/2\)-point DFT’s, while the second term accounts for the cost of combining their outputs to produce \(X[k]\). If we were to simply implement the \(N/2\)-point DFT’s using the naive algorithm, we would not have saved all that much – their cost would be proportional to \(4 \times (N/2)^2\), and so we would save roughly a factor of two. We can try to further improve the complexity by iterating the procedure – expressing each of the \(N/2\)-point DFT’s in terms of two \(N/4\)-point DFT’s, as shown in Figure 2. This leads to a computational cost of at most \(4T_{N/4} + 2CN\). It is clear that if \(N = 2^p\) for some integer \(p\), we can iterate this procedure \(p-1\), arriving at a computational cost of at most \(2^{p-1}T_2 + (p-1)CN\). We can compute the two-point DFT directly, in constant time. The architecture for doing this is drawn in Figure 3. Since \(T_2\) is a constant, \(2^{p-1}\) is bounded by \(N\), and \(p-1\) is bounded by \(\log_2 N\), the overall computational cost of this algorithm is bounded by a constant times \(N \log N\). This is a remarkable improvement over the \(O(N^2)\) algorithms described above.

Several practical simplifications are possible. First, we can note that for \(W_N = \exp \left( -j \frac{2\pi}{N} \right)\), \(W_{N/2} = W_N^2\). So, we can express all of the complex coefficients in Figure 2 as powers of \(W_N\). This yields Figure 4. In Figure 4, we repeatedly perform a basic operation, pictured in Figure 5(left). This operation combines two complex numbers in two different ways, with multiplicative coefficients which are powers of \(W_N\). With a glance at Figure 4 and some mental extrapolation, we can guess that (i) the multiplicative coefficients are always applied to the “bottom” input, and (ii) the multiplicative coefficients are always of the form \(W_N^r\) and \(W_N^{r+N/2} = -W_N^r\). We can exploit this observation to replace one of the multiplications with a subtraction, yielding the basic structure in Figure 5(right). Substituting this transformation into Figure 4 yields a practical FFT architecture. The text describes additional practical considerations, including indexing the input and operating “in place” with limited memory.
Figure 1: **Recursive DFT architecture suggested by Theorem 2.1 (Decimation in Time).** In this diagram, arrows leading into other lines represent additions; a scalar component labeling any line represents a scalar multiplication. So, each of the components $X[k]$ of the output length-$N$ DFT is generated by adding an output of the top length $N/2$ DFT to a scaled output of the bottom length $N/2$ DFT.

Figure 2: **Recursive DFT architecture**, with two levels of recursion, generated by replacing the DFT$_{N/2}$ boxes in Figure 1 with two length $N/4$ DFT’s, using Theorem 1 (Decimation in Time).
Figure 3: The base of the recursion: a two-point DFT, using two complex additions and a single negation.

Figure 4: Recursive DFT architecture, with coefficients as powers of $W_N$.

Figure 5: The basic operation in a decimation-in-time FFT (left), and a slight simplification (right), which uses that $W_N^{r+\frac{N}{2}} = -W_N^r$. 

$W_N = \exp\left(-\frac{2\pi}{N}\right)$
The algorithm described here is a special case of the Cooley-Tukey FFT, published by Cooley and Tukey in 1965. It turned out that this algorithm was actually known to (and used by!) Gauss as early as 1805, for interpolation problems in astronomy. The procedure works as described when \( N = 2^p \) for some integer \( p \). Variants are known for other bases than 2—e.g., \( N = 3^p \). A more general version of the Cooley-Tukey algorithm can be applied to general composite integers \( N \): for \( N = N_1 N_2 \) we can reduce the computation to \( N_1 \) DFT’s of length \( N_2 \). There are also algorithmic variants which apply to \( N \) that are not composite (i.e., prime \( N \)). These work by expressing the length-\( N \) DFT as a convolution, and then computing this convolution in frequency domain using FFT’s of different (composite) length. For example, for prime \( N \) Rader’s algorithm reduces the DFT to 3 DFT’s of (composite) length \( N − 1 \). You can consult, e.g., Section 9.6.2 of the text for more information about these approaches.

2.2 Decimation in frequency

The decimation-in-frequency FFT architecture is inspired by the following observation on the DFT of a length-\( N \) sequence \( x[n] \), for \( N \) an even integer. We assume that \( x[n] \) is defined for \( n = 0, 1, \ldots, N − 1 \); our goal is to compute the discrete Fourier transform \( X[k] = \text{DFT}_N \{ x \} [k] \) for \( k = 0, 1, \ldots, N − 1 \). The decimation-in-frequency architecture results from the observation that we can compute the terms for even \( k \) and the terms for odd \( k \) separately:

**Theorem 2.2.** Let

\[
\begin{align*}
    x_0[n] &= x[n] + x[n + N/2], \\
    x_1[n] &= (x[n] - x[n + N/2]) \exp \left( -j \frac{2\pi n}{N} \right).
\end{align*}
\]

Then

\[
\text{DFT}_N \{ x \} [k] = \begin{cases} 
    \text{DFT}_{N/2} \{ x_0 \} [k/2] & \text{if } k \text{ even} \\
    \text{DFT}_{N/2} \{ x_1 \} \left[ \frac{k-1}{2} \right] & \text{if } k \text{ odd}.
\end{cases}
\]

**Proof.** For \( k = 2i \) an even integer, we calculate

\[
\begin{align*}
    X[k] &= \sum_{n=0}^{N-1} x[n] \exp \left( -j \frac{2\pi kn}{N} \right) \\
    &= \sum_{n=0}^{N/2-1} x[n] \exp \left( -j \frac{2\pi kn}{N} \right) + \sum_{n=N/2}^{N-1} x[n + N/2] \exp \left( -j \frac{2\pi k(n + N/2)}{N} \right) \\
    &= \exp \left( -j \frac{2\pi kn}{N} \right), \text{ since } \exp \left( -j \frac{2\pi kN/2}{N} \right) = 1 \text{ for } k = 2i.
\end{align*}
\]

\[
\begin{align*}
    &= \sum_{n=0}^{N/2-1} (x[n] + x[n + N/2]) \exp \left( -j \frac{2\pi kn}{N} \right) \\
    &= \sum_{n=0}^{N/2-1} x_0[n] \exp \left( -j \frac{2\pi in}{N/2} \right) \\
    &= \text{DFT}_{N/2} \{ x_0 \} [i].
\end{align*}
\]
For $k = 2i + 1$ an odd integer, we similarly calculate

$$X[k] = \sum_{n=0}^{N-1} x[n] \exp\left(-j \frac{2\pi kn}{N}\right)$$

(2.16)

$$= \sum_{n=0}^{N/2-1} x[n] \exp\left(-j \frac{2\pi kn}{N}\right) + x[n + N/2] \exp\left(-j \frac{2\pi k(n + N/2)}{N}\right)$$

(2.17)

$$= \sum_{n=0}^{N/2-1} \exp\left(-j \frac{2\pi n}{N}\right) \exp\left(-j \frac{2\pi n}{N/2}\right)$$

(2.18)

$$= \text{DFT}_{N/2} \{x_1\} \{i\},$$

(2.19)

completing the proof.

Thus, we can compute the DFT of $x$ by first computing the length $N/2$ DFT’s of $x_0$ and $x_1$, and then interleaving the results. Iterating this operation again leads to an $O(N \log N)$ time algorithm, also referred to as the FFT, here computed by *decimation in frequency*. Based on the formula, you can easily draw similar diagrams relating the input and the output, as we did for decimation in time.

Decimation in time and decimation in frequency lead to the same asymptotic complexity $O(N \log N)$. The relative advantages and disadvantages between decimation in time and decimation in frequency tend to be platform specific. Most platforms/languages in which you would think to perform digital signal processing already come with highly optimized implementations of the FFT – as such, you are unlikely to ever implement the FFT, except as a learning exercise. For example, Matlab’s implementation (*fft*) uses a combination of the decimation-in-time (Cooley-Tukey) algorithm, with Radar’s algorithm (alluded to above, and in the text) to handle prime $N$, and optimized implementations of special base cases. For our purposes, the important point is to understand why the DFT can be computed so efficiently.

### 3 Inverse DFT via the FFT

The inverse discrete Fourier transform has a very similar structure to the forward transform. In fact, the inverse DFT can be computed directly using the (forward) FFT and some simple manipulation. Since

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] \exp\left(j \frac{2\pi nk}{N}\right),$$

(3.1)

we have the identity

$$Nx^*\{n\} = \sum_{k=0}^{N-1} X^*\{k\} \exp\left(-j \frac{2\pi nk}{N}\right)$$

(3.2)

$$= \text{DFT}_N \{X^*\} \{n\}.$$
Hence, we can evaluate the inverse DFT as

\[ x[n] = \frac{1}{N} \text{DFT}_N \{ X^* \} [n]^* \].

(3.4)

The DFT in this expression can be computed efficiently via the FFT.

As above, the main purpose of this derivation is to convince you that we can compute the inverse DFT efficiently using a single FFT and some additional minor manipulations. There are a variety of ways of relating \( x[n] \) to a (forward) DFT of a manipulated version of \( X[k] \), which may have practical advantages over the above formula. You can consult the text and its references for more details on these alternatives.