Fast Fourier Transform (FFT)

Don Johnson

This work is produced by The Connexions Project and licensed under the Creative Commons Attribution License.*

Abstract

The DFT can be reduced from exponential time with the Fast Fourier Transform algorithm.

Fast Fourier Transform (FFT)

One wonders if the DFT can be computed faster: Does another computational procedure — an algorithm — exist that can compute the same quantity, but more efficiently. We could seek methods that reduce the constant of proportionality, but do not change the DFT’s complexity $O(N^2)$. Here, we have something more dramatic in mind: Can the computations be restructured so that a smaller complexity results?

In 1965, IBM researcher Jim Cooley and Princeton faculty member John Tukey developed what is now known as the Fast Fourier Transform (FFT). It is an algorithm for computing that DFT that has order $O(N \log N)$ for certain length inputs. Now when the length of data doubles, the spectral computational time will not quadruple as with the DFT algorithm; instead, it approximately doubles. Later research showed that no algorithm for computing the DFT could have a smaller complexity than the FFT. Surprisingly, historical work has shown that Gauss\(^1\) in the early nineteenth century developed the same algorithm, but did not publish it! After the FFT’s rediscovery, not only was the computation of a signal’s spectrum greatly speeded, but also the added feature of algorithm meant that computations had flexibility not available to analog implementations.

Exercise 1:

Before developing the FFT, let’s try to appreciate the algorithm’s impact. Suppose a short-length transform takes 1 ms. We want to calculate a transform of a signal that is 10 times longer. Compare how much longer a straightforward implementation of the DFT would take in comparison to an FFT, both of which compute exactly the same quantity.

Solution:

If a DFT required 1ms to compute, and signal having ten times the duration would require 100ms to compute. Using the FFT, a 1ms computing time would increase by a factor of about $\log_2 10 = 3.3$, a factor of 30 less than the DFT would have needed.

---

*http://creativecommons.org/licenses/by/1.0

\(^1\)http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Gauss.html

http://cnx.rice.edu/content/m10250/latest/
To derive the FFT, we assume that the signal’s duration is a power of two: \( N = 2^L \). Consider what happens to the even-numbered and odd-numbered elements of the sequence in the DFT calculation.

\[
S(k) = s(0) + s(2)e^{(-i)\frac{2\pi k}{N}} + \cdots + s(N - 2)e^{(-i)\frac{2\pi (N - 2)k}{N}} + s(1)e^{(-i)\frac{2\pi k}{N}} + s(3)e^{(-i)\frac{2\pi (3)k}{N}} + \cdots + s(N - 1)e^{(-i)\frac{2\pi (N - 1)k}{N}}

= \left[ s(0) + s(2)e^{(-i)\frac{2\pi k}{N}} + \cdots + s(N - 2)e^{(-i)\frac{2\pi (N - 2)k}{N}} \right] + \left[ s(1) + s(3)e^{(-i)\frac{2\pi k}{N}} + \cdots + s(N - 1)e^{(-i)\frac{2\pi (N - 1)k}{N}} \right]
\]

Each term in square brackets has the form of a \( \frac{N}{2} \)-length DFT. The first one is a DFT of the even-numbered elements, and the second of the odd-numbered elements. The first DFT is combined with the second multiplied by the complex exponential \( e^{(-i)\frac{2\pi k}{N}} \). The half-length transforms are each evaluated at frequency indices \( k = 0, \ldots, N-1 \). Normally, the number of frequency indices in a DFT calculation range between zero and the transform length minus one. The computational advantage of the FFT comes from recognizing the periodic nature of the discrete Fourier transform. The FFT simply reuses the computations made in the half-length transforms and combines them through additions and the multiplication by \( e^{(-i)\frac{2\pi k}{N}} \), which is not periodic over \( \frac{N}{2} \), to rewrite the length-\( N \) DFT. Figure 1 illustrates this decomposition. As it stands, we now compute two length-\( \frac{N}{2} \) transforms (complexity \( 2O\left(\frac{N^2}{4}\right) \)), multiply one of them by the complex exponential (complexity \( O(1) \)), and add the results (complexity \( O(N) \)). At this point, the total complexity is still dominated by the half-length DFT calculations, but the proportionality coefficient has been reduced.

Now for the fun. Because \( N = 2^L \), each of the half-length transforms can be reduced to two quarter-length transforms, each of these to two eighth-length ones, etc. This decomposition continues until we are left with length-2 transforms. This transform is quite simple, involving only additions. Thus, the first stage of the FFT has \( \frac{N}{2} \) length-2 transforms (see the bottom part of Figure 1). Pairs of these transforms are combined by adding one to the other multiplied by a complex exponential. Each pair requires 4 additions and 4 multiplications, giving a total number of computations equaling \( 8 \cdot \frac{N}{2} = \frac{N}{2} \). This number of computations does not change from stage to stage. Because the number of stages, the number of times the length can be divided by two, equals \( \log_2 N \), the complexity of the FFT is \( O(N\log_2 N) \).

Doing an example will make computational savings more obvious. Let’s look at the details of a length-8 DFT. As shown on Figure 2, we first decompose the DFT into two length-4 DFTs, with the outputs added and subtracted together in pairs. Considering Figure 2 as the frequency index goes from 0 through 7, we recycle values from the length-4 DFTs into the final calculation because of the periodicity of the DFT output. Examining how pairs of outputs are collected together, we create the basic computational element known as a butterfly (Figure 2).

By considering together the computations involving common output frequencies from the two half-length DFTs, we see that the two complex multiplies are related to each other, and we can reduce our computational work even further. By further decomposing the length-4 DFTs into two length-2 DFTs and combining their outputs, we arrive at the diagram summarizing the length-8 fast Fourier transform (Figure 1). Although most of the complex multiplies are quite simple (multiplying by \( e^{(-i\pi)} \) means negating real and imaginary parts), let’s count those for purposes of evaluating the complexity as full complex multiplies. We have \( \frac{N}{2} = 4 \) complex multiplies and \( 2N = 16 \) additions for each stage and \( \log_2 N = 3 \) stages, making the number of basic computations \( \frac{3N}{2} \log_2 N \) as predicted.

Exercise 2:
Figure 1: The initial decomposition of a length-8 DFT into the terms using even- and odd-indexed inputs marks the first phase of developing the FFT algorithm. When these half-length transforms are successively decomposed, we are left with the diagram shown in the bottom panel that depicts the length-8 FFT computation.
Figure 2: The basic computational element of the fast Fourier transform is the butterfly. It takes two complex numbers, represented by \( a \) and \( b \), and forms the quantities shown. Each butterfly requires one complex multiplication and two complex additions.

Note that the ordering of the input sequence in the two parts of Figure 1 aren’t quite the same. Why not? How is the ordering determined?

Solution:

The upper panel has not used the FFT algorithm to compute the length-4 DFTs while the lower one has. The ordering is determined by the algorithm.

Other "fast" algorithms were discovered, all of which make use of how many common factors the transform length \( N \) has. In number theory, the number of prime factors a given integer has measures how composite it is. The numbers 16 and 81 are highly composite (equaling \( 2^4 \) and \( 3^4 \) respectively), the number 18 is less so (\( 2^1 \cdot 3^2 \)), and 17 not at all (it’s prime). In over thirty years of Fourier transform algorithm development, the original Cooley-Tukey algorithm is far and away the most frequently used. It is so computationally efficient that power-of-two transform lengths are frequently used regardless of what the actual length of the data.

Exercise 3:

Suppose the length of the signal were 500? How would you compute the spectrum of this signal using the Cooley-Tukey algorithm? What would the length \( N \) of the transform be?

Solution:

The transform can have any greater than or equal to the actual duration of the signal. We simply “pad” the signal with zero-valued samples until a computationally advantageous signal length results. Recall that the FFT is an algorithm to compute the DFT\(^2\). Extending the length of the signal this way merely means we are sampling the frequency axis more finely than required. To use the Cooley-Tukey algorithm, the length of the resulting zero-padded signal can be 512, 1024, etc. samples long.

\(^2\)http://cnx.rice.edu/content/m10249/latest/

http://cnx.rice.edu/content/m10250/latest/