Discrete Fourier Transforms and the Fast Fourier Transform (FFT) Algorithm

Paul Heckbert

We start in the continuous world; then we get discrete.

Definition of the Fourier Transform

The Fourier transform (FT) of the function $f(x)$ is the function $F(\omega)$, where:

$$F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-i\omega x}\, dx$$

and the inverse Fourier transform is

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) e^{i\omega x}\, d\omega$$

Recall that $i = \sqrt{-1}$ and $e^{i\theta} = \cos\theta + i \sin\theta$.

Think of it as a transformation into a different set of basis functions. The Fourier transform uses complex exponentials (sinusoids) of various frequencies as its basis functions. (Other transforms, such as Z, Laplace, Cosine, Wavelet, and Hartley, use different basis functions).

A Fourier transform pair is often written $f(x) \leftrightarrow F(\omega)$, or $\mathcal{F}(f(x)) = F(\omega)$ where $\mathcal{F}$ is the Fourier transform operator.

If $f(x)$ is thought of as a signal (i.e. input data) then we call $F(\omega)$ the signal's *spectrum*. If f is thought of as the *impulse response* of a filter (which operates on input data to produce output data) then we call F the filter's *frequency response*. (Occasionally the line between what's signal and what's filter becomes blurry).

Example of a Fourier Transform

In signal processing terms, the ideal low pass filter with cutoff frequency ω_c has a frequency response that is a box:

$$F(\omega) = \begin{cases} 1 & |\omega| \leq \omega_c \\ 0 & |\omega| > \omega_c \end{cases}$$

What is its impulse response?

We know that the impulse response is the inverse Fourier transform of the frequency response, so taking off our signal processing hat and putting on our mathematics hat, all we need to do is evaluate:

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) e^{i\omega x}\, d\omega$$

for this particular $F(\omega)$:

$$\begin{aligned} f(x) &= \frac{1}{2\pi} \int_{-\omega_c}^{\omega_c} e^{i\omega x}\, d\omega \\ &= \frac{1}{2\pi} \frac{e^{i\omega x}}{ix} \bigg|_{\omega=-\omega_c}^{\omega_c} \\ &= \frac{1}{\pi x} \frac{e^{i\omega_c x} - e^{-i\omega_c x}}{2i} \\ &= \frac{\sin \omega_c x}{\pi x} \qquad\qquad \text{since } \sin\theta = \frac{e^{i\theta} - e^{-i\theta}}{2i} \\ &= \frac{\omega_c}{\pi} \text{sinc}(\frac{\omega_c}{\pi} x) \end{aligned}$$

where $\text{sinc(x)} = \sin(\pi\text{x})/(\pi\text{x})$. For antialiasing with unit-spaced samples, you want the cutoff frequency to equal the Nyquist frequency, so $\omega_c = \pi$.

Fourier Transform Properties

Rather than write "the Fourier transform of an X function is a Y function", we write the shorthand: $X \leftrightarrow Y$. If z is a complex number and $z = x + iy$ where x and y are its real and imaginary parts, then the complex conjugate of z is $z^* = x - iy$. A function $f(u)$ is *even* if $f(u) = f(-u)$, it is *odd* if $f(u) = -f(-u)$, it is conjugate symmetric if $f(u) = f^*(-u)$, and it is conjugate antisymmetric if $f(u) = -f^*(-u)$.

discrete $\leftrightarrow$ periodic
periodic $\leftrightarrow$ discrete
discrete, periodic $\leftrightarrow$ discrete, periodic
real $\leftrightarrow$ conjugate symmetric
imaginary $\leftrightarrow$ conjugate antisymmetric
box $\leftrightarrow$ sinc
sinc $\leftrightarrow$ box
Gaussian $\leftrightarrow$ Gaussian
impulse $\leftrightarrow$ constant
impulse train $\leftrightarrow$ impulse train

(can you prove the above?)

When a signal is scaled up spatially, its spectrum is scaled down in frequency, and vice versa: $f(ax) \leftrightarrow F(\omega/a)$ for any real, nonzero a.

Convolution Theorem

The Fourier transform of a convolution of two signals is the product of their Fourier transforms: $f \circledast g \leftrightarrow FG$. Recall that the convolution of signals f and g is $(f \circledast g)(x) = \int_{-\infty}^{+\infty} f(t)g(x-t)\,dt$. So $\int_{-\infty}^{+\infty} f(t)g(x-t)\,dt \leftrightarrow F(\omega)G(\omega)$.

The Fourier transform of a product of two signals is the convolution of their Fourier transforms: $fg \leftrightarrow F \circledast G/2\pi$.

Delta Functions

The (Dirac) delta function $\delta(x)$ is defined such that $\delta(x) = 0$ for all $x \neq 0$, $\int_{-\infty}^{+\infty} \delta(t)\,dt = 1$, and for any $f(x)$:

$$(f \circledast \delta)(x) = \int_{-\infty}^{+\infty} f(t)\delta(x-t)\,dt = f(x)$$

The latter is called the *sifting property* of delta functions. Because convolution with a delta is linear shift-invariant filtering, translating the delta by a will translate the output by a:

$$\Big(f(x) \circledast \delta(x-a)\Big)(x) = f(x-a)$$

Discrete Fourier Transform (DFT)

When a signal is discrete and periodic, we don't need the continuous Fourier transform. Instead we use the discrete Fourier transform, or DFT. Suppose our signal is a_n for $n = 0 \ldots N-1$, and $a_n = a_{n+jN}$ for all n and j. The spectrum of a is:

$$A_k = \sum_{n=0}^{N-1} W_N^{kn} a_n \tag{1}$$

where

$$W_N = e^{-i\frac{2\pi}{N}}$$

and W_N^k for $k = 0 \ldots N-1$ are called the *Nth roots of unity*. They're called this because, in complex arithmetic, $(W_N^k)^N = 1$ for all k. They're vertices of a regular polygon inscribed in the unit circle of the complex plane, with one vertex at $(1, 0)$.

The sequence A_k is the discrete Fourier transform of the sequence a_n. Each is a sequence of N complex numbers.

Two-point DFT (N=2)

$W_2 = e^{-i\pi} = -1$, and

$$A_k = \sum_{n=0}^{1} (-1)^{kn} a_n = (-1)^{k\cdot 0} a_0 + (-1)^{k\cdot 1} a_1 = a_0 + (-1)^k a_1$$

so

$$\begin{aligned} A_0 &= a_0 + a_1 \\ A_1 &= a_0 - a_1 \end{aligned}$$

Four-point DFT (N=4)

$W_4 = e^{-i\pi/2} = -i$, and

$$A_k = \sum_{n=0}^{3} (-i)^{kn} a_n = a_0 + (-i)^k a_1 + (-i)^{2k} a_2 + (-i)^{3k} a_3 = a_0 + (-i)^k a_1 + (-1)^k a_2 + i^k a_3$$

so

$$\begin{aligned} A_0 &= a_0 + a_1 + a_2 + a_3 \\ A_1 &= a_0 - ia_1 - a_2 + ia_3 \\ A_2 &= a_0 - a_1 + a_2 - a_3 \\ A_3 &= a_0 + ia_1 - a_2 - ia_3 \end{aligned}$$

This can also be written as a matrix multiply:

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

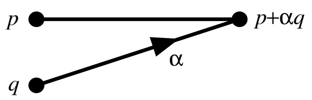
More on this later.

To compute A quickly, we can pre-compute common subexpressions:

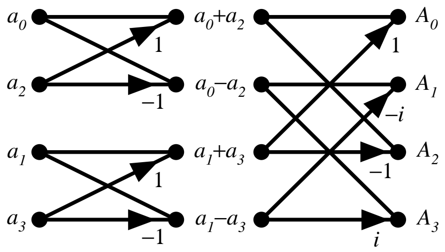
$$\begin{aligned} A_0 &= (a_0 + a_2) + (a_1 + a_3) \\ A_1 &= (a_0 - a_2) - i(a_1 - a_3) \\ A_2 &= (a_0 + a_2) - (a_1 + a_3) \\ A_3 &= (a_0 - a_2) + i(a_1 - a_3) \end{aligned}$$

This saves a lot of adds. (Note that each add and multiply here is a complex (not real) operation.)

If we use the following diagram for a complex multiply and add:



then we can diagram the 4-point DFT like so:



If we carry on to $N = 8$, $N = 16$, and other power-of-two discrete Fourier transforms, we get...

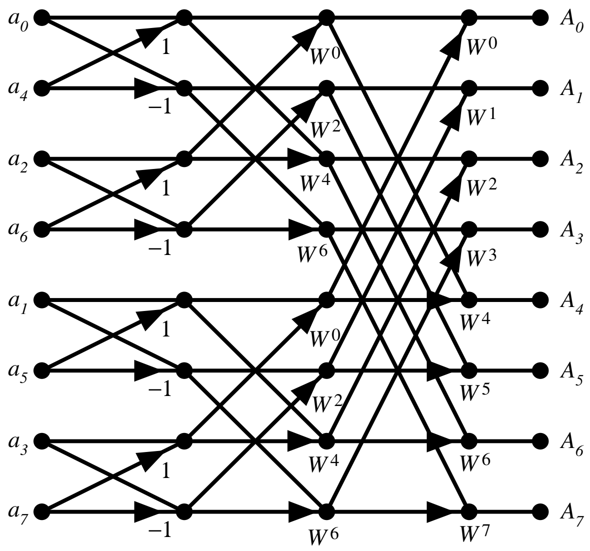
The Fast Fourier Transform (FFT) Algorithm

The FFT is a fast algorithm for computing the DFT. If we take the 2-point DFT and 4-point DFT and generalize them to 8-point, 16-point, ..., 2^r-point, we get the FFT algorithm.

To compute the DFT of an N-point sequence using equation (1) would take $O(N^2)$ multiplies and adds. The FFT algorithm computes the DFT using $O(N \log N)$ multiplies and adds.

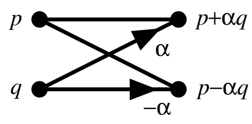
There are many variants of the FFT algorithm. We'll discuss one of them, the "decimation-in-time" FFT algorithm for sequences whose length is a power of two ($N = 2^r$ for some integer r).

Below is a diagram of an 8-point FFT, where $W = W_8 = e^{-i\pi/4} = (1-i)/\sqrt{2}$:



Butterflies and Bit-Reversal

The FFT algorithm decomposes the DFT into $\log_2 N$ stages, each of which consists of $N/2$ *butterfly* computations. Each butterfly takes two complex numbers p and q and computes from them two other numbers, $p + \alpha q$ and $p - \alpha q$, where α is a complex number. Below is a diagram of a butterfly operation:



In the diagram of the 8-point FFT above, note that the inputs aren't in normal order: $a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7$, they're in the bizarre order: $a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7$. Why this sequence?

Below is a table of j and the index of the jth input sample, n_j:

j	0	1	2	3	4	5	6	7
n_j	0	4	2	6	1	5	3	7
j base 2	000	001	010	011	100	101	110	111
n_j base 2	000	100	010	110	001	101	011	111

The pattern is obvious if j and n_j are written in binary (last two rows of the table). Observe that each n_j is the *bit-reversal* of j. The sequence is also related to breadth-first traversal of a binary tree.

It turns out that this FFT algorithm is simplest if the input array is rearranged to be in bit-reversed order. The re-ordering can be done in one pass through the array a:

```
for j = 0 to N-1
    nj = bit_reverse(j)
    if (j<nj) swap a[j] and a[nj]
```

FFT Explained Using Matrix Factorization

The 8-point DFT can be written as a matrix product, where we let $W = W_8 = e^{-i\pi/4} = (1-i)/\sqrt{2}$:

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \\ A_7 \end{pmatrix} = \begin{pmatrix} W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 \\ W^0 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 \\ W^0 & W^2 & W^4 & W^6 & W^0 & W^2 & W^4 & W^6 \\ W^0 & W^3 & W^6 & W^1 & W^4 & W^7 & W^2 & W^5 \\ W^0 & W^4 & W^0 & W^4 & W^0 & W^4 & W^0 & W^4 \\ W^0 & W^5 & W^2 & W^7 & W^4 & W^1 & W^6 & W^3 \\ W^0 & W^6 & W^4 & W^2 & W^0 & W^6 & W^4 & W^2 \\ W^0 & W^7 & W^6 & W^5 & W^4 & W^3 & W^2 & W^1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix}$$

Rearranging so that the input array a is bit-reversed and factoring the 8×8 matrix:

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \\ A_7 \end{pmatrix} = \begin{pmatrix} W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 \\ W^0 & W^4 & W^2 & W^6 & W^1 & W^5 & W^3 & W^7 \\ W^0 & W^0 & W^4 & W^4 & W^2 & W^2 & W^6 & W^6 \\ W^0 & W^4 & W^6 & W^2 & W^3 & W^7 & W^1 & W^5 \\ W^0 & W^0 & W^0 & W^0 & W^4 & W^4 & W^4 & W^4 \\ W^0 & W^4 & W^2 & W^6 & W^5 & W^1 & W^7 & W^3 \\ W^0 & W^0 & W^4 & W^4 & W^6 & W^6 & W^2 & W^2 \\ W^0 & W^4 & W^6 & W^2 & W^7 & W^3 & W^5 & W^1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_4 \\ a_2 \\ a_6 \\ a_1 \\ a_5 \\ a_3 \\ a_7 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & \cdot & \cdot & \cdot & W^0 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & W^1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & W^2 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & W^3 \\ 1 & \cdot & \cdot & \cdot & W^4 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & W^5 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & W^6 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & W^7 \end{pmatrix} \begin{pmatrix} 1 & \cdot & W^0 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & W^2 & \cdot & \cdot & \cdot & \cdot \\ 1 & \cdot & W^4 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & W^6 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & W^0 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & W^2 \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & W^4 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & W^6 \end{pmatrix} \begin{pmatrix} 1 & W^0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & W^4 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & W^0 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & W^4 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & W^0 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & W^4 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & W^0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & W^4 \end{pmatrix} \begin{pmatrix} a_0 \\ a_4 \\ a_2 \\ a_6 \\ a_1 \\ a_5 \\ a_3 \\ a_7 \end{pmatrix}$$

where "·" means 0.

These are sparse matrices (lots of zeros), so multiplying by the dense (no zeros) matrix on top is more expensive than multiplying by the three sparse matrices on the bottom.

For $N = 2^r$, the factorization would involve r matrices of size $N \times N$, each with 2 non-zero entries in each row and column.

How Much Faster is the FFT?

To compute the DFT of an N-point sequence using the definition,

$$A_k = \sum_{n=0}^{N-1} W_N^{kn} a_n,$$

would require N^2 complex multiplies and adds, which works out to $4N^2$ real multiplies and $4N^2$ real adds (you can easily check this, using the definition of complex multiplication).

The basic computational step of the FFT algorithm is a butterfly. Each butterfly computes two complex numbers of the form $p + \alpha q$ and $p - \alpha q$, so it requires one complex multiply $(\alpha \cdot q)$ and two complex adds. This works out to 4 real multiplies and 6 real adds per butterfly.

There are $N/2$ butterflies per stage, and $\log_2 N$ stages, so that means about $4 \cdot N/2 \cdot \log_2 N = 2N \log_2 N$ real multiplies and $3N \log_2 N$ real adds for an N-point FFT. (There are ways to optimize further, but this is the basic FFT algorithm.)

Cost comparison:

N	$r = \log_2 N$	**BRUTE FORCE** $4N^2$	**FFT** $2N \log_2 N$	speedup
2	1	16	4	4
4	2	64	16	4
8	3	256	48	5
1,024	10	4,194,304	20,480	205
65,536	16	$1.7 \cdot 10^{10}$	$2.1 \cdot 10^{6}$	~10^4

The FFT algorithm is a LOT faster for big N.

There are also FFT algorithms for N not a power of two. The algorithms are generally fastest when N has many factors, however.

An excellent book on the FFT is: E. Oran Brigham, *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

Why Would We Want to Compute Fourier Transforms, Anyway?

The FFT algorithm is used for fast convolution (linear, shift-invariant filtering). If $h = f \circledast g$ then convolution of continuous signals involves an integral:
$h(x) = \int_{-\infty}^{+\infty} f(t)g(x-t)\,dt$, but convolution of discrete signals involves a sum: $h[x] = \sum_{t=-\infty}^{\infty} f[t]g[x-t]$. We might think of f as the signal and g as the filter.

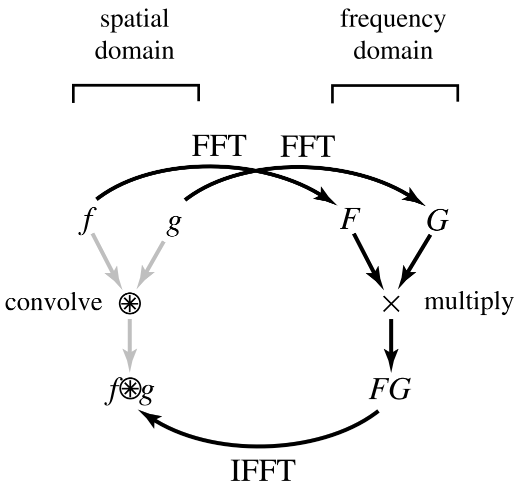
When working with finite sequences, the definition of convolution simplifies if we assume that f and g have the same length N and we regard the signals as being periodic, so that f and g "wrap around". Then we get *circular convolution*:

$$h[x] = \sum_{t=0}^{N-1} f[t]g[x-t \bmod N] \text{ for } x = 0 \ldots N-1$$

The convolution theorem says that the Fourier transform of the convolution of two signals is the product of their Fourier transforms: $f \circledast g \leftrightarrow FG$. The corresponding theorem for discrete signals is that the DFT of the circular convolution of two signals is the product of their DFT's.

Computing the convolution with a straightforward algorithm would require N^2 (real) multiplies and adds – too expensive!

We can do the same computation faster using discrete Fourier transforms. If we compute the DFT of sequence f and the DFT of sequence g, multiply them point-by-point, and then compute the inverse DFT, we'll get the same answer. This is called *Fourier Convolution*:



If we use the FFT algorithm, then the two DFT's and the one inverse DFT have a total cost of $6N \log_2 N$ real multiplies, and the multiplication of transforms in the frequency domain has a negligible cost of $4N$ real multiplies. The straightforward algorithm, on the other hand, required N^2 real multiplies.

Fourier convolution wins big for large N.

Often, circular convolution isn't what you want, but this algorithm can be modified to do standard "linear" convolution by padding the sequences with zeros appropriately.

Fourier Transforms of Images

To compute the Fourier transform of an image, you

- Compute DFT of each row, in place.
- Compute DFT of each column, in place.

Practical issues: You probably want to cyclically translate the picture so that pixel (0,0), which now contains frequency $(\omega_x, \omega_y) = (0, 0)$, moves to the center of the image. And you probably want to display pixel values proportional to log(magnitude) of each complex number (this looks more interesting than just magnitude). For color images, do the above to each of the three channels (R, G, and B) independently.

For an $N \times N$ picture, the cost is proportional to $N^2 \log N$. The straightforward algorithm has a cost proportional to N^4.

FFT's are also used for synthesis of fractal textures and to create images with a given spectrum.

Fourier Transforms and Arithmetic

The FFT is also used for fast extended precision arithmetic, and multiplication of high-degree polynomials, since they also involve convolution. If polynomials p and q have the form: $p(x) = \sum_{n=0}^{N-1} f_n x^n$ and $q(x) = \sum_{n=0}^{N-1} g_n x^n$ then their product is the polynomial

$$\begin{aligned} r(x) = p(x)q(x) &= \Big(\sum_{n=0}^{N-1} f_n x^n\Big)\Big(\sum_{n=0}^{N-1} g_n x^n\Big) \\ &= (f_0 + f_1 x + f_2 x^2 + \cdots)(g_0 + g_1 x + g_2 x^2 + \cdots) \\ &= f_0 g_0 + (f_0 g_1 + f_1 g_0)x + (f_0 g_2 + f_1 g_1 + f_2 g_0)x^2 + \cdots \\ &= \sum_{n=0}^{2N-2} h_n x^n \end{aligned}$$

where $h_n = \sum_{j=0}^{N-1} f_j g_{n-j}$, and $h = f \circledast g$. Thus, computing the product of two polynomials involves the convolution of their coefficient sequences.

Extended precision numbers (numbers with hundreds or thousands of significant figures) are typically stored in the form of a polynomial, where x has a fixed value of perhaps 2^{32} or 10^9. Multiplication of extended precision numbers involves the convolution of high-degree polynomials.

When N is small (< 100, say), then straightforward convolution is reasonable, but for large N, it makes sense to compute convolutions using Fourier convolution.