

Mathematical Background to the FFT

Fourier Series

If $f(x)$ is a periodic function defined on an interval $[-\pi, \pi]$ we can write $f(x)$ as a superposition of trig functions.

$$f(x) = a_0 + \sum_{n=1}^{\infty} a_n \cos(n x) + \sum_{n=1}^{\infty} b_n \sin(n x) \quad (1)$$

The mathematical trick needed to compute the coefficients is the *orthogonality property* of integrals of trig functions.

$$\int_{-\pi}^{\pi} \sin(m x) \sin(n x) dx = \begin{cases} \pi & \text{if } n = m \\ 0 & \text{otherwise} \end{cases}$$

$$\int_{-\pi}^{\pi} \cos(m x) \cos(n x) dx = \begin{cases} \pi & \text{if } n = m \\ 0 & \text{otherwise} \end{cases}$$

$$\int_{-\pi}^{\pi} \sin(m x) \cos(n x) dx = 0$$

$$\int_{-\pi}^{\pi} \sin(m x) dx = 0$$

$$\int_{-\pi}^{\pi} \cos(m x) dx = 0$$

Multiplying both sides of (1) by factors of $\sin(m x)$ or $\cos(m x)$ and using the orthogonality properties of trig functions makes it possible to compute the coefficients in (1).

$$a_0 = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) dx$$

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(n x) dx$$

$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(n x) dx$$

Complex Fourier Series

If $f(x)$ is a real-valued function the coefficients a_n and b_n will all be real numbers. Another kind of Fourier series comes from using Euler's formula

$$e^{i n x} = \cos(n x) + i \sin(n x)$$

to replace the trig functions in (1) with complex exponentials. This leads to a *complex Fourier series*

$$f(x) = \sum_{n=-\infty}^{\infty} A_n e^{i n x}$$

where we allow the coefficients A_n to be complex-valued. Just as there are orthogonality relations for the trig functions that make it possible to compute the coefficients in (1) the complex exponential also has an orthogonality relation that makes it possible to compute the coefficients A_n .

$$A_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) e^{-i n x} dx \quad (2)$$

The Discrete Fourier Transform

If $f(x)$ is a *discrete function* that takes a series of discrete values over sub-intervals of the real line, the integral (2) reduces to a summation of integrals that look like

$$\int_{x_k}^{x_{k+1}} f_k e^{-i n x} dx$$

These integrals are easy to compute, and essentially convert (2) into a summation formula over the subintervals over which $f(x)$ is constant.

This is the essential idea behind the *Discrete Fourier Transform*. Let $a = (a_0, a_1, \dots, a_{n-1})$ be a sequence of complex numbers. The Discrete Fourier Transform of a is a new sequence of numbers $y = (y_0, y_1, \dots, y_{n-1})$ where

$$y_k = \sum_{j=0}^{n-1} a_j (\omega_n^k)^j \quad (3)$$

Here ω_n is the complex n^{th} root of unity, $\omega_n = e^{2\pi i/n}$.

The Fast Fourier Transform

Formula (3) provides a straight-forward algorithm for computing the discrete Fourier

transform of a sequence of numbers. Since the formula requires that we sum n terms for each of n values of k , the basic algorithm is $O(n^2)$. The Fast Fourier Transform (FFT) replaces the basic calculation (3) with a more efficient algorithm that is $O(n \lg n)$.

The starting point for the FFT is the observation that the summation on the right hand side of (3) looks like a polynomial. Specifically, if we introduce

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

we see that the formula for the DFT can be written

$$y_k = A(\omega_n^k)$$

FFT is based on a divide and conquer strategy. Specifically, we introduce two new functions

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \dots + a_{n-2} x^{n/2-1}$$

$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \dots + a_{n-1} x^{n/2-1}$$

and note that

$$A(x) = A^{[0]}(x^2) + x A^{[1]}(x^2)$$

Where the recursion comes in is when we note that the computation of $A^{[0]}$ and $A^{[1]}$ looks like the computation of a DFT on the two subsequences $(a_0, a_2, a_4, \dots, a_{n-2})$ and $(a_1, a_3, a_5, \dots, a_{n-1})$.

The trick that makes FFT successful is the fact that the sequences $y^{[0]}$ and $y^{[1]}$ returned by $\text{Recursive-FFT}(a^{[0]})$ and $\text{Recursive-FFT}(a^{[1]})$ can be easily combined to produce the desired sequence y . For the details, please refer to the textbook.

Pseudocode for the FFT

```

Recursive-FFT( $a$ )
1.  $n \leftarrow \text{length}[a]$ 
2. if  $n=1$ 
3.     then return  $a$ 
4.  $\omega_n \leftarrow e^{2\pi i/n}$ 
5.  $\omega \leftarrow 1$ 
6.  $a^{[0]} \leftarrow (a_0, a_2, a_4, \dots, a_{n-2})$ 

```

7. $a^{[1]} \leftarrow (a_1, a_3, a_5, \dots, a_{n-1})$
8. $y^{[0]} \leftarrow \text{Recursive-FFT}(a^{[0]})$
9. $y^{[1]} \leftarrow \text{Recursive-FFT}(a^{[1]})$
10. **for** $k \leftarrow 0$ **to** $n/2 - 1$ **do**
11. $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$
12. $y_{k+n/2} \leftarrow y_k^{[0]} - \omega y_k^{[1]}$
13. $\omega \leftarrow \omega \omega_n$
14. **return** y

Computing the inverse DFT

Another useful mathematical accident is that the formula for the inverse DFT looks very similar to (3):

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k (\omega_n^{-j})^k$$

This means that we can use the same algorithm as before with the roles of a and y switched, ω_n replaced by ω_n^{-1} , and an extra multiplication by a factor of $1/n$.

Applications of the FFT - Multiplying Polynomials

The naive algorithm for multiplying two polynomials $A(x)$ and $B(x)$ is to multiply them term by term and add up the products. This naive algorithm has run-time $O(n^2)$. Given that the FFT is a process that evaluates a polynomial at a specific set of points (the n roots of unity $\omega_n^k = e^{2\pi i k/n}$), it may just be possible that the FFT can be used to solve certain problems involving polynomials. Multiplying two polynomials efficiently is one such problem.

Here is the outline of a method that could be used to multiply two n^{th} degree polynomials $A(x)$ and $B(x)$:

1. Evaluate both $A(x)$ and $B(x)$ at a set of sample points x_0, x_1, \dots, x_{2n} to produce points $a_k = A(x_k)$ and $b_k = B(x_k)$.
2. Note that the product polynomial $C(x) = A(x) B(x)$ satisfies the equality $c_k = C(x_k) = A(x_k) B(x_k) = a_k b_k$ at all of the sample points.
3. Use the data from the $2n+1$ pairs (x_k, c_k) to reconstruct the polynomial $C(x)$.

The final step is possible because passing through any $2n+1$ points (x_k, c_k) in the plain with distinct x -coordinates there is a unique $(2n)^{th}$ degree polynomial.

In general, this more complicated method of computing $C(x) = A(x)B(x)$ is no more efficient than the naive method, because step 1 alone takes time $O(n^2)$ and there is no efficient algorithm available for doing step 3.

Although the general method outlined above is not efficient, in one special case it can be made quite efficient. That special case is when the sample points x_k just happen to be the $2n+1$ distinct $(2n+1)^{st}$ roots of unity in the complex plain. In that special case, we do the following.

1. Let a_0, a_1, \dots, a_n be the coefficients of $A(x)$ and b_0, b_1, \dots, b_n the coefficients of $B(x)$. Pad both lists with trailing 0s to make two lists of length $2n+1$.
2. Apply the FFT to lists $a_0, a_1, \dots, a_n, 0, \dots, 0$ and $b_0, b_1, \dots, b_n, 0, \dots, 0$ to generate lists y_0, y_1, \dots, y_{2n} and z_0, z_1, \dots, z_{2n} .
3. Form the product list $w_0 = y_0 z_0, w_1 = y_1 z_1, \dots, w_{2n} = y_{2n} z_{2n}$.
4. Apply the inverse FFT to the list w_0, w_1, \dots, w_{2n} to produce a list c_0, c_1, \dots, c_{2n} .
5. Construct a polynomial $C(x)$ whose coefficients are the numbers in the list c_0, c_1, \dots, c_{2n} .

Since this method is essentially the same as the method outlined above with specially chosen sample points, it also computes the product polynomial $C(x)$ correctly. What is special about this method is that the FFT steps take time $O(n \lg n)$ and dominate the run time of this algorithm. Thus, we have found an $O(n \lg n)$ algorithm to multiply polynomials.

Second Application - Fast Multiplication of Large Integers

The fast multiplication algorithm for polynomials outlined above has a simple extension to another problem, multiplying two n -digit integers quickly. The naive algorithm for multiplying integers is similar to the naive algorithm for multiplying polynomials, and has similar performance - multiplying two large, n -digit integers takes time $O(n^2)$.

With a little bit of cleverness we can bend the fast polynomial multiplication algorithm to our purposes here.

The trick is to replace the numbers A and B with digits $a_{n-1}, a_{n-2}, \dots, a_0$ and $b_{n-1}, b_{n-2}, \dots, b_0$ with *digit polynomials*

$$A(x) = a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \dots + a_0$$

$$B(x) = b_{n-1} x^{n-1} + b_{n-2} x^{n-2} + \dots + b_0$$

The connection between the numbers and their digit polynomials is that $A = A(10)$ and $B = B(10)$. If we form the product polynomial $C(x) = A(x) B(x)$ with the fast polynomial product algorithm described above, we can get $C = C(10) = A(10) B(10) = A B$ in time $O(n \lg n)$.