Fast Fourier Transform and Polynomial Multiplication

(COMS 4770/5770 Notes)

Yan-Bin Jia

Oct 6, 2026

In this lecture we will describe the famous algorithm of fast Fourier transform (FFT), which has revolutionized digital signal processing and in many ways changed our life. It was listed by the Science magazine as one of the ten greatest algorithms in the 20th century. Here we will first introduce the FFT algorithm in the context of polynomial multiplication, and then reveal how it is used for computing the Fourier transform.

Suppose we are given two polynomials:

$$\begin{aligned} p(x) &= a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}, \\ q(x) &= b_0 + b_1 x + \cdots + b_{n-1} x^{n-1}. \end{aligned}$$

Their product is defined by

$$p(x) \cdot q(x) = c_0 + c_1 x + \cdots c_{2n-2} x^{2n-2}$$

where

$$c_i = \sum_{\max\{0, i-(n-1)\} \le k \le \min\{i, n-1\}} a_k b_{i-k}.$$

In computing the product polynomial, every a_i is multiplied with every b_j, for $0 \le i, j \le n-1$. So there are at most n^2 multiplications, given that some of the coefficients may be zero. Obtaining every c_i involves one fewer additions than multiplications. So there are at most $n^2 - 2n + 1$ additions involved. In short, the number of arithmetic operations is $O(n^2)$. This is hardly efficient.

But can we obtain the product more efficiently? The answer is yes, by the use of a well-known method called *fast Fourier transform*, or simply, FFT.

1 Discrete Fourier Transform

Let us start with introducing the discrete Fourier transform (DFT) problem. Denote by ω_n an nth complex root of 1, that is,

$$\omega_n = e^{i\frac{2\pi}{n}}, \tag{1}$$

where $i^2 = -1$. DFT is the mapping between two vectors:

$$\boldsymbol{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \quad \longmapsto \quad \hat{\boldsymbol{a}} = \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \\ \vdots \\ \hat{a}_{n-1} \end{pmatrix} \tag{2}$$

such that

$$\hat{a}_j = \sum_{k=0}^{n-1} a_k \omega_n^{jk}, \qquad j = 0, \ldots, n-1.$$

It can also be written as a matrix equation:

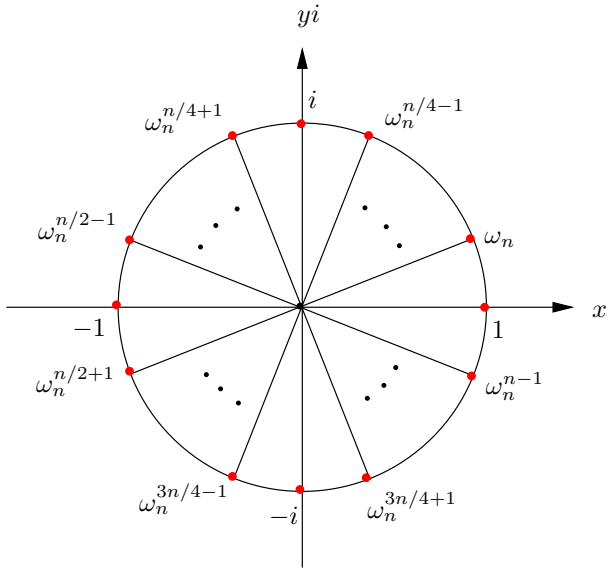
$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \\ \vdots \\ \hat{a}_{n-1} \end{pmatrix}. \tag{3}$$

The matrix above is a Vandermonde matrix and denoted by V_n.

Essentially, DFT evaluates the polynomial

$$p(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$$

at n points $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$; in other words, $\hat{a}_k = p(\omega_n^k)$ for $0 \le k \le n-1$. From now on we assume that n is a power of 2. If not, we can always add in higher order terms with zero coefficients $a_n = a_{n+1} = \cdots = a_{2^{\lceil \log_2 n \rceil} - 1} = 0$. The powers of ω_n are illustrated in the complex plane in the following figure.



The fast Fourier transform algorithm cleverly makes use of the following properties about ω_n:

$$\begin{aligned} \omega_n^n &= 1, \\ \omega_n^{n+k} &= \omega^k, \\ \omega_n^{\frac{n}{2}} &= -1, \\ \omega_n^{\frac{n}{2}+k} &= -\omega_n^k. \end{aligned}$$

It uses a divide-and-conquer strategy. More specifically, it divides $p(x)$ into two polynomials $p_0(x)$ and $p_1(x)$, both of degree $\frac{n}{2}-1$; namely,

$$\begin{aligned} p_0(x) &= a_0 + a_2 x + \cdots + a_{n-2} x^{\frac{n}{2}-1}, \\ p_1(x) &= a_1 + a_3 x + \cdots + a_{n-1} x^{\frac{n}{2}-1}. \end{aligned}$$

Hence

$$p(x) = p_0(x^2) + x p_1(x^2). \tag{4}$$

In this way the problem of evaluating $p(x)$ at $\omega_n^0, \ldots, \omega_n^{n-1}$ breaks down into two steps:

1. evaluating $p_0(x)$ and $p_1(x)$ at $(\omega_n^0)^2$, $(\omega_n^1)^2$, …, $(\omega_n^{n-1})^2$,
2. combining the resulting according to (4).

Note that the list $(\omega_n^0)^2$, $(\omega_n^1)^2$, …, $(\omega_n^{n-1})^2$ consists of only $\frac{n}{2}$ complex roots of unity, i.e., $\omega_n^0, \omega_n^2, \ldots, \omega_n^{n-2}$. So the subproblems of evaluating $p_0(x)$ and $p_1(x)$ have exactly the same form as the original problem of evaluating $p(x)$, only at half the size. This decomposition forms the basis for the recursive FFT algorithm presented below.

```
RECURSIVE-DFT(a, n)
1   if n = 1
2       then return a
3   ω_n ← e^{i 2π/n}
4   ω ← 1
5   a^[0] ← (a_0, a_2, ..., a_{n-2})
6   a^[1] ← (a_1, a_3, ..., a_{n-1})
7   â^[0] ← RECURSIVE-DFT(a^[0], n/2)
8   â^[1] ← RECURSIVE-DFT(a^[1], n/2)
9   for k = 0 to n/2 − 1 do
10      â_k ← â_k^[0] + ω â_k^[1]
11      â_{k+n/2} ← â_k^[0] − ω â_k^[1]
12      ω ← ω ω_n
13  return (â_0, â_1, ..., â_{n-1})
```

To verify the correctness, we here understand line 11 in the procedure RECURSIVE-DFT:

$$\hat{a}_{k+\frac{n}{2}} = \hat{a}_k^{[0]} - \omega \hat{a}_k^{[1]}.$$

At the kth iteration of the **for** loop of lines 9–12, $\omega = \omega_n^k$. We have

$$\begin{aligned} \hat{a}_{k+\frac{n}{2}} &= \hat{a}_k^{[0]} - \omega_n^k \hat{a}_k^{[1]} \\ &= \hat{a}_k^{[0]} + \omega_n^{k+\frac{n}{2}} \hat{a}_k^{[1]} \\ &= p_0\left(\omega_n^{2k}\right) + \omega_n^{k+\frac{n}{2}} p_1\left(\omega_n^{2k}\right) \\ &= p_0\left(\omega_n^{2k+n}\right) + \omega_n^{k+\frac{n}{2}} p_1\left(\omega_n^{2k+n}\right) \\ &= p\left(\omega_n^{k+\frac{n}{2}}\right), \qquad \text{from (4).} \end{aligned}$$

Let $T(n)$ be the running time of RECURSIVE-DFT. Steps 1–6 take time $\Theta(n)$. Steps 7 and 8 each takes time $T(\frac{n}{2})$. Steps 9–13 take time $\Theta(n)$. So we end up with the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n),$$

which has the solution

$$T(n) = \Theta(n \log_2 n).$$

2 Inverse DFT

Suppose we need to compute the inverse Fourier transform given by

$$\boldsymbol{a} = V_n^{-1}\hat{\boldsymbol{a}}. \tag{5}$$

Namely, we would like to determine the coefficients of the polynomial $p(x) = a_0 + \cdots + a_{n-1}x^{n-1}$ given its values at $\omega_n^0, \ldots, \omega_n^{n-1}$. Can we do it with the same efficiency, that is, in time $\Theta(n \log n)$? The answer is yes. To see why, note that the Vandermonde matrix V_n has inverse

$$V_n^{-1} = \frac{1}{n}\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \cdots & \omega_n^{-(n-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \cdots & \omega_n^{-(n-1)^2} \end{pmatrix} \tag{6}$$

To verify the above, make use of the equation $\sum_{j=0}^{n-1}(\omega_n^k)^j = 0$ for non-negative integer k not divisible by n.

Based on the above observation, we can still apply RECURSIVE-DFT by replacing $\boldsymbol{a}$ with $\hat{\boldsymbol{a}}$, $\hat{\boldsymbol{a}}$ with $\boldsymbol{a}$, ω_n with ω_n^{-1} (that is, ω_n^{n-1}), and scaling the result by $\frac{1}{n}$.

3 Fast Multiplication of Two Polynomials

Let us now go back to the two polynomials at the beginning:

$$\begin{aligned} p(x) &= a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}, \\ q(x) &= b_0 + b_1 x + \cdots + b_{n-1}x^{n-1}. \end{aligned}$$

Their product

$$(p \cdot q)(x) = p(x) \cdot q(x) = c_0 + c_1 x + \cdots c_{2n-2}x^{2n-2}$$

can be computed by combining FFT with interpolation. The computation takes time $\Theta(n \log n)$ and consists of the following three steps:

1. Evaluate $p(x)$ and $q(x)$ at $2n$ points $\omega_{2n}^0, \ldots, \omega_{2n}^{2n-1}$ using DFT. This step takes time $\Theta(n \log n)$.

2. Obtain the values of $p(x)q(x)$ at these $2n$ points through pointwise multiplication

$$\begin{aligned}(p \cdot q)(\omega_{2n}^0) &= p(\omega_{2n}^0) \cdot q(\omega_{2n}^0),\\ (p \cdot q)(\omega_{2n}^1) &= p(\omega_{2n}^1) \cdot q(\omega_{2n}^1),\\ &\vdots\\ (p \cdot q)(\omega_{2n}^{2n-1}) &= p(\omega_{2n}^{2n-1}) \cdot q(\omega_{2n}^{2n-1}).\end{aligned}$$

   This step takes time $\Theta(n)$.

3. Interpolate the polynomial $p \cdot q$ at the product values using inverse DFT to obtain coefficients $c_0, c_1, \ldots, c_{2n-2}$. This last step requires time $\Theta(n \log n)$.

We can also use FFT to compute the *convolution* of two vectors

$$a = (a_0, \ldots, a_{n-1}) \qquad \text{and} \qquad b = (b_0, \ldots, b_{n-1}),$$

which is defined as a vector $c = (c_0, \ldots, c_{n-1})$ where

$$c_j = \sum_{k=0}^{j} a_k b_{j-k}, \qquad j = 0, \ldots, n-1.$$

The running time is again $\Theta(n \log n)$.

4 Fourier Transform and Coefficients

A large class of important computational problems falls under the category of "Fourier transform methods" or "spectral methods". For some of these problems, the Fourier transform, named in honor of Jean-Baptiste Joseph Fourier (1768–1830), is simply an efficient computational tool for data manipulation. For other problems, the Fourier transform is itself of intrinsic interest. Fourier methods have revolutionized fields of science and engineering, from radio astronomy to medical imaging, from seismology to spectroscopy. The wide application of Fourier methods is credited principally to the development of the fast Fourier transform (FFT). The most direct applications of the FFT are to the convolution or deconvolution of data, correlation and autocorrelation, optimal filtering, power spectrum estimation, and the computation of Fourier integrals.

The Fourier transform has many applications in computer vision. It is a main tool for analysis, enhancement, reconstruction, and compression of images. By representing a noisy image in the frequency domain, it can be smoothed by removing high or low frequencies that correspond to the noise components, and then converting the result back. In the analysis of, say, a medical image, the Fourier transform can be used to select significant frequencies.

A physical process can be described either in the *time domain*, by the values of some quantity h as a function of time t, or else in the *frequency domain*, where the process is specified by giving its amplitude H as a function of frequency ξ with $-\infty < \xi < \infty$. For many purposes it is useful to think of $h(t)$ and $H(\xi)$ as being two different representations of the same function. These two representations are related to each other by the Fourier transform equations,

$$\begin{aligned}H(\xi) &= \int_{-\infty}^{\infty} h(t) e^{2\pi i \xi t}\, dt,\\ h(t) &= \int_{-\infty}^{\infty} H(\xi) e^{-2\pi i \xi t}\, d\xi,\end{aligned}$$

where $i^2 = -1$.

The *Fourier series* for a function $f(x)$ is given as

$$f(x) \approx \sum_{j=-\infty}^{\infty} \hat{f}(j) e^{ijx}, \tag{7}$$

where

$$\hat{f}(j) = \frac{1}{2\pi} \int_0^{2\pi} f(x) e^{-ijx} dx.$$

are the *Fourier coefficients*. From the above definition we easily see that $\hat{f}(-j) = \overline{\hat{f}(j)}$, the complex conjugate of $\hat{f}(j)$. In (7), "$\approx$" means that the Fourier series converges to $f(x)$ under rather mild conditions. For example, the series converges uniformly if $f(x)$ is continuous and $f'(x)$ is piecewise continuous.

The partial sum

$$\sum_{j=-n}^{n} \hat{f}(j) e^{ijx}$$

of the Fourier series for $f(x)$ is the best approximation to $f(x)$ by any trigonometric polynomial of order n, that is, any function in the form:

$$p(x) = \frac{a_0}{2} + \sum_{j=1}^{n} (a_j \cos jx + b_j \sin jx),$$

where $a_0, \ldots, a_n$ and $b_1, \ldots, b_n$ are real or complex numbers. Furthermore, it can be shown that *Parseval's relation*

$$\sum_{j=-\infty}^{\infty} |\hat{f}(j)|^2 = \frac{1}{2\pi} \int_0^{2\pi} |f(x)|^2 \, dx$$

holds.

The Fourier coefficients $\hat{f}(j)$ can help us "understand" the function $f(x)$. Suppose $f(x)$ is a *real* function with period 2π. It can be viewed as the motion of a point at time x on a line. Substitute the *polar form*

$$\hat{f}(j) = |\hat{f}(j)| e^{i\theta_j}$$

into the Fourier series (7) and use the fact that $\hat{f}(j)$ and $\hat{f}(-j)$ are complex conjugates:

$$f(x) \approx |\hat{f}(0)| \cos\theta_0 + 2 \sum_{j=1}^{\infty} |\hat{f}(j)| \cos(\theta_j + jx).$$

Thus we have obtained a representation of the periodic motion as a superposition of simple harmonic oscillations. The jth such motion (with $j > 0$), $2|\hat{f}(j)| \cos(\theta_j + jx)$, has

amplitude:	$2\lvert\hat{f}(j)\rvert$,
frequency:	$\frac{j}{2\pi}$,
angular frequency:	j,
period or *wavelength*:	$\frac{2\pi}{j}$,
phase angle:	θ_j.

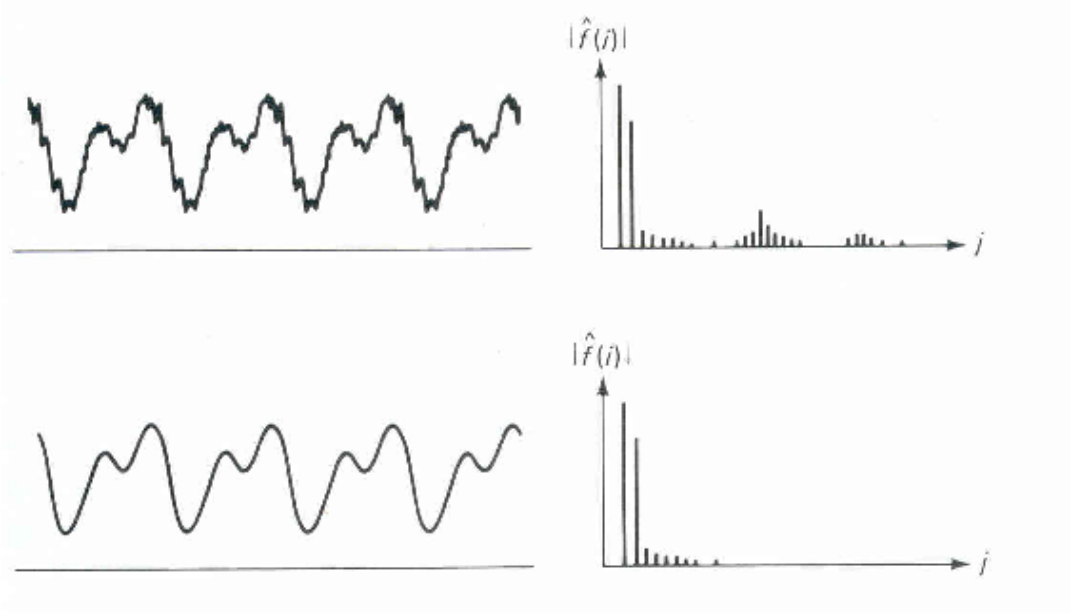


Figure 1: Smoothing (figure from [1, p. 271]).

The number $|\hat{f}(j)|$ measures the strength of the presence of a simple harmonic motion of frequency $\frac{j}{2\pi}$ in the total motion. It can be shown that

$$\hat{f}(j) = O\left(|j|^{-l-1}\right), \tag{8}$$

when the lth derivative of $f(x)$ exists and is piecewise continuous. The sequence $|\hat{f}(0)|, |\hat{f}(1)|, \ldots$ is called the *spectrum* of $f(x)$ over which the "total energy" $\|f\|_2^2$ is distributed. A "noisy" function will have sizeable $|\hat{f}(j)|$ for large j. For a "smooth" function, the spectrum will decrease rapidly as j increases.

The method of *smoothing* often consists in generating the Fourier coefficients of $f(x)$ from data, *filtering* these coefficients to suppress high frequencies (which usually correspond to noise), and then reconstructing the function as a Fourier series with "purified" or "filtered" coefficients. Figure 1 shows two 2π-periodic functions and their power spetrums. The second function is obtained from the first by filtering out its higher frequencies.

5 Computation of Fourier Coefficients Using FFT

We are interested in the frequencies present in $f(x)$ and their strength (or magnitude). Since it is generally difficult or impossible to compute the Fourier coefficients $\hat{f}(j)$, $-\infty < j < \infty$, exactly, we use their discrete approximations that result from *sampling* f at the points $x_k = \frac{2\pi k}{n}$ for $k = 0, \ldots, n-1$. They are

$$\hat{f}_n(j) = \frac{1}{n} \sum_{k=0}^{n-1} f(x_k) e^{-ijx_k}. \tag{9}$$

However, due to the aliasing effect in sampling [4, pp. 502], $\hat{f}_n(j)$, is good as an approximation to $f(j)$ only for $-\frac{n}{2} < j \leq \frac{n}{2}$. Thus, for very large n we want to be able to calculate $\hat{f}_n(j)$, for $0 \leq |j| \leq \frac{n}{2}$, or equivalently, for $0 \leq j \leq n-1$, from $f(x_0), \ldots, f(x_{n-1})$, where $x_k = \frac{2\pi k}{n}$. A straightforward calculation would take time $O(n^2)$.

A significant improvement can be achieved by reducing the above problem to one similar to the inverse discrete Fourier transform (IDFT) given in (5). The mapping here has the following form:

$$\boldsymbol{z} = \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_{n-1} \end{pmatrix} \quad \longmapsto \quad \hat{\boldsymbol{z}} = \begin{pmatrix} \hat{z}_0 \\ \hat{z}_1 \\ \vdots \\ \hat{z}_{n-1} \end{pmatrix}$$

such that

$$\hat{z}_j = \sum_{k=0}^{n-1} z_k \omega_n^{-jk}, \qquad j = 0, \ldots, n-1,$$

where ω_n is an nth root of 1 defined in (1). The mapping can be written as a matrix equation:

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \cdots & \omega_n^{-(n-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \cdots & \omega_n^{-(n-1)^2} \end{pmatrix} \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_{n-1} \end{pmatrix} = nV_n^{-1} \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_{n-1} \end{pmatrix} = \begin{pmatrix} \hat{z}_0 \\ \hat{z}_1 \\ \vdots \\ \hat{z}_{n-1} \end{pmatrix}.$$

If we take $z_j = f(x_j)$, $0 \leq j \leq n-1$, then

$$\hat{f}_n(j) = \frac{1}{n} \hat{z}_j, \qquad j = 0, 1, \ldots, n-1.$$

To verify, by definition (9) we have

$$\begin{aligned} \hat{f}_n(j) &= \frac{1}{n} \sum_{k=0}^{n-1} f(x_k) e^{-ijx_k} \\ &= \frac{1}{n} \sum_{k=0}^{n-1} f(x_k) e^{-ij\frac{2\pi k}{n}} \\ &= \frac{1}{n} \sum_{k=0}^{n-1} f(x_k) \left(\omega_n^{-j}\right)^k \\ &= \frac{1}{n} \hat{z}_j. \end{aligned}$$

Fast Fourier transform allows us to compute the discrete Fourier coefficients in time $O(n \log n)$. With $n = 10^6$, for example, the improvement is from roughly two weeks of CPU time to 30 seconds!

6 History of FFT

Modern FFT is widely credited to the paper [3] by Cooley and Tukey. But the algorithm had been discovered independently by a few individuals in the past. Only the appearance of digital

computers and the wide application of signal processing made people realize the importance of fast computation of large Fourier series. An incomplete list of pioneers includes

- Gauss (1805) — the earliest known origin of the FFT algorithm.
- Runge and König (1924) — the doubling algorithm.
- Danielson and Lanczos (1942) — divide-and-conquer on DFTs.
- Rudnick (1960s) — the first computer program implementation with $O(n \log n)$ time.

References

[1] S. D. Conte and C de Boor. *Elementary Numerical Analysis: An Algorithmic Approach*. McGraw-Hill, Inc., 3rd edition, 1980.

[2] T. H. Cormen *et al. Introduction to Algorithms*. McGraw-Hill, Inc., 2nd edition, 2001.

[3] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297-301, 1965.

[4] W. H. Press, *et al. Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 2002.