

HIGH-PERFORMANCE COMPUTING IN APPLIED MATHEMATICS

DISCRETE FOURIER TRANSFORM AND FAST FOURIER TRANSFORM ¹

The discrete Fourier transform (DFT) has several practical applications including:

- signal processing
- partial differential equations
- polynomial multiplication and interpolation

A simple practical example: Consider the Fourier series representation of a continuous periodic function on the interval $[0, 2\pi]$:

$$f(x) = a_0 + \sum_{k=1}^{\infty} (a_k \cos kx + b_k \sin kx)$$

Using the Euler's formulas

$$\cos \theta = \frac{e^{i\theta} + e^{-i\theta}}{2}, \quad \sin \theta = \frac{e^{i\theta} - e^{-i\theta}}{2i}$$

the complex form of the Fourier series may be written

$$f(x) = \sum_{k=-\infty}^{\infty} c_k e^{-ikx}$$

where

$$c_0 = a_0, \quad c_k = \frac{a_k + ib_k}{2} \quad (1)$$

The complex Fourier coefficients are given by the formula:

$$c_k = \frac{1}{2\pi} \int_0^{2\pi} f(x) e^{ikx} dx \quad (2)$$

and if f is a real-valued function then $c_{-k} = \bar{c}_k$. Once the complex coefficients are found, a_k and b_k may be recovered from (1).

If numerical integration with the nodes $h = 2\pi/n$, $x_j = 2j\pi/n$, $j = 0, 1, \dots, n$ is used to evaluate (2) the approximation formula become:

$$c_k = \frac{1}{n} \sum_{j=0}^{n-1} f(x_j) e^{ikx_j} = \frac{1}{n} \sum_{j=0}^{n-1} f(x_j) e^{2\pi i k j / n} = \frac{1}{n} \sum_{j=0}^{n-1} f(x_j) (e^{2\pi i / n})^{kj}, \quad k = 0, 1, \dots \quad (3)$$

Definition: Given a vector $a = (a_0, a_1, \dots, a_{n-1})$ we define the Discrete Fourier Transform of a as the vector $y = (y_0, y_1, \dots, y_{n-1})$ whose components are

$$y \stackrel{def}{=} DFT_n(a); \quad y_k = \sum_{j=0}^{n-1} a_j \omega_n^{kj}, \quad k = 0, 1, \dots, n-1 \quad (4)$$

¹Reference: "Introduction to Algorithms" by T.H. Cormen, C.E. Leiserson and R.L. Rivest. MIT Press, 1990.

where ω_n is the principal n^{th} root of the unity

$$\omega_n = e^{2\pi i/n}$$

All other complex roots of order n of the unity are powers of ω_n

$$\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$$

The problem of evaluating $DFT(a)$ is equivalent to the problem of evaluating the polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \tag{5}$$

at the complex roots of order n of the unity, since

$$y_k = A(\omega_n^k), \quad k = 0, 1, \dots, n-1$$

If we define the DFT matrix as

$$F \in \mathbb{R}^{n \times n}, F_{k,j} = \omega_n^{kj}, \quad 0 \leq k, j \leq n-1 \tag{6}$$

then y may be written as the matrix vector product

$$y = Fa \tag{7}$$

For example, with $n = 4$ the DFT matrix is:

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\ 1 & \omega_4^3 & \omega_4^6 & \omega_4^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

The inverse discrete Fourier transform (IDFT) is

$$DFT^{-1}(y) = F^{-1}a \tag{8}$$

and one notices that IDFT solves the problem of interpolation at the roots of the unity:

Given the data points $(\omega_n^k, y_k), k = 0, 1, \dots, n-1$ find the polynomial of $A(x)$ degree $n-1$ such that $A(\omega_n^k) = y_k, k = 0, 1, \dots, n-1$

Evaluating $DFT(a)$ using standard multiplication has a computational complexity of order $O(n^2)$.

The Fast Fourier Transform (FFT, Cooley-Tukey 1965) provides an algorithm to evaluate DFT with a computational complexity of order $O(n \log n)$ where $\log = \log_2$. As we shall see shortly, IDFT may be then evaluated as well using $O(n \log n)$ operations. The computational savings are significant. For example, if $n = 10^3$, the standard DFT requires $\sim 10^6$ operations, whereas FFT will involve only $\sim 10^4$, thus a

computational saving by a factor of 100.

In order to introduce FFT we first review some important properties of the roots of order n of the unity.

Lemma 1: For any integers $n \geq 0, k \geq 0$, and $d > 0$

$$\omega_{dn}^{dk} = \omega_n^k \quad (9)$$

Proof:

$$\omega_{dn}^{dk} = e^{\frac{2\pi i}{dn} dk} = e^{\frac{2\pi i}{n} k} = \omega_n^k$$

Lemma 2: For any even integer $n > 0$

$$\omega_n^{n/2} = -1 \quad (10)$$

Proof: Let $n = 2m, m > 0$.

$$\omega_n^{n/2} = e^{\frac{2\pi i}{2m} m} = e^{\pi i} = -1$$

Lemma 3: If $n > 0$ is even, then

$$(\omega_n^k)^2 = \omega_{n/2}^k \quad (11)$$

Proof:

$$(\omega_n^k)^2 = \omega_n^{2k} = \omega_{2n/2}^{2k} \stackrel{\text{lemma 1}}{=} \omega_{n/2}^k$$

Lemma 4: For any integer $n \geq 1$ and any integer $k > 0$ such that $k \bmod n \neq 0$

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0 \quad (12)$$

Proof:

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

since $k \bmod n \neq 0 \Rightarrow \omega_n^k \neq 1$.

Property: For any integers $j, k \geq 0$,

$$\omega_n^j \omega_n^k = \omega_n^{(j+k) \bmod n}, \quad \omega_n^{kj} = \omega_n^{kj \bmod n} \quad (13)$$

Lemma 5: The inverse of the DFT matrix is obtained by conjugating the entries of F and scaling by n :

$$F^{-1} = \frac{1}{n} \bar{F}, \quad F_{k,j}^{-1} = \frac{1}{n} \bar{\omega}_n^{kj}, \quad 0 \leq k, j \leq n-1 \quad (14)$$

where

$$\bar{\omega}_n = e^{-\frac{2\pi i}{n}} = \omega_n^{-1}$$

Proof: We show that $F^{-1}F = I_n$, the identity matrix.

$$(F^{-1}F)_{k,l} = \frac{1}{n} \sum_{j=0}^{n-1} \bar{\omega}_n^{kj} \omega_n^{jl} = \frac{1}{n} \sum_{j=0}^{n-1} \omega_n^{-kj} \omega_n^{jl} = \frac{1}{n} \sum_{j=0}^{n-1} \omega_n^{(l-k)j} = \frac{1}{n} \sum_{j=0}^{n-1} (\omega_n^{(l-k)})^j = \delta_{k,l}$$

where $\delta_{k,l} = 0, k \neq l$ and $\delta_{l,l} = 1$. The last equality in the equation above follows from Lemma 4.

THE FFT ALGORITHM

The FFT algorithm takes advantage of the properties of the complex roots of the unity to calculate DFT(a) using $O(n \log n)$ operations. We assume that n is a power of 2, $n = 2^m$ (radix-2 algorithm). The FFT method is a *divide-and-conquer algorithm* that recursively breaks a DFT of size n into two DFTs of size $n/2$ and an additional $O(n)$ multiplications.

By separating the even-index and the odd-index coefficients, the polynomial (5) may be written

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \tag{15}$$

where

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \dots + a_{n-2}x^{n/2-1} \tag{16}$$

$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \dots + a_{n-1}x^{n/2-1} \tag{17}$$

are polynomials of degree $n/2 - 1$.

The problem of evaluating A at $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$ is thus reduced to

1. Evaluate the polynomials $A^{[0]}$ and $A^{[1]}$ of degree $n/2 - 1$ at $(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2$
2. combine the results according to (15)

A fundamental remark is that $(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2$ are not n distinct values, but, according to Lemma 3, they are the $n/2$ complex roots of order $n/2$ of the unity. Therefore, each subproblem for $A^{[0]}$ and $A^{[1]}$ has exactly the same form as the problem for A but are of half size. The original problem of size n is thus decomposed into two problems each of size $n/2$ and an additional overhead (15) that requires one multiplication and one addition for each value $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$.

recursive function $y = FFT(a, n)$

if $n == 1$ then

$y = a$

 return

end if

$\omega_n = e^{2\pi i/n}$

$\omega = 1$

$m = n/2$

$y_0 = FFT(a(0 : 2 : n - 1), m)$

$y_1 = FFT(a(1 : 2 : n - 1), m)$

for $k = 0 : m - 1$

$y(k) = y_0(k) + \omega * y_1(k)$

! use $t = \omega * y_1(k)$ to multiply only once

$y(k + m) = y_0(k) - \omega * y_1(k)$

! use t from above

$\omega = \omega * \omega_n$

end

The *for loop* of the algorithm performs the evaluation (15) at $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$. Notice that at counter value k , ω has the value ω_n^k such that for $k = 0, 1, \dots, n/2 - 1$ the loop performs

$$y(k) = y_0(k) + \omega_n^k y_1(k) = A^{[0]}(\omega_n^k) + \omega_n^k A^{[1]}(\omega_n^k) = A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) = A(\omega_n^k)$$

$$\begin{aligned} y(k + n/2) &= y_0(k) - \omega_n^k y_1(k) = y_0(k) + \omega_n^{k+(n/2)} y_1(k) = A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) = A(\omega_n^{k+(n/2)}) \end{aligned}$$

Denoting $T(n)$ the computational cost of the problem of size n , the following recursive formula holds:

$$T(n) \leq 2T(n/2) + cn \tag{18}$$

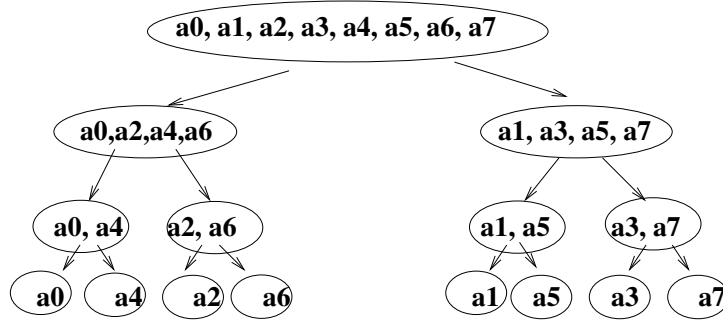
where c is a constant. Recursively,

$$T(n) \leq 2T(n/2) + cn \leq 2^2T(n/2^2) + 2cn/2 + cn \leq \dots \leq 2^m T(n/2^m) + cmn = nT(1) + cn \log n$$

since $n = 2^m \Rightarrow m = \log n$. Therefore, the computational complexity of the algorithm is $O(n \log n)$.

Remark: The inverse DFT is obtained by switching the roles of a and y , setting $\omega_n = e^{-2\pi i/n}$ in the algorithm above, then divide each element by n . The inverse DFT may be thus performed with $O(n \log n)$ computations.

Data structure in the recursive calls



To get the elements of the input vector a into the desired order, a bit-reverse operation may be performed. In binary representation the bit-reverse operation results in

$$0 = 000 \rightarrow 000 = 0$$

$$1 = 001 \rightarrow 100 = 4$$

$$2 = 010 \rightarrow 010 = 2$$

$$3 = 011 \rightarrow 110 = 6$$

$$4 = 100 \rightarrow 001 = 1$$

$$5 = 101 \rightarrow 101 = 5$$

$$6 = 110 \rightarrow 011 = 3$$

$$7 = 111 \rightarrow 111 = 7$$

such that if we bit-reverse-copy a into A , $A[\text{rev}(k)] = a[k]$, then A will hold the elements of a in the desired order. This may be used to implement an iterative FFT algorithm

Iterative-FFT(a)

bit-reverse-copy(a, A)

$n = \text{length}(a)$

for $s = 1 : \log_2(n)$

$$m = 2^s$$

$$\omega_m = e^{2\pi i/m}$$

$$\omega = 1$$

for $j = 0 : m/2 - 1$

for $k = j : m : n - 1$

$$t = \omega * A(k + m/2)$$

$$u = A(k)$$

$$A(k) = u + t$$

$$A(k + m/2) = u - t$$

$$\omega = \omega * \omega_m$$

return A