4 The Discrete Fourier Transform

In this section we will look at the Discrete Fourier Transform, which transforms the representation of a function from a sequence of coefficients to a (discrete) sequence of values of the function. Once we understand the transform itself, we will look at how to compute it quickly, and then how this can be used in polynomial multiplication.

4.1 Theory behind the DFT

Definition 4.1. *Let* F *be a field,* $n \in \mathbb{N}$ *and* $\omega \in \mathsf{F}$*. We say* ω *is a primitive nth root of unity (n-PRU) if*

(1) $\omega^n = 1$,

(2) n *is a unit in* F,

(3) $\omega^k - 1$ *is not zero for* $1 \le k < n$.

The last condition implies that $\omega^k \neq 1$, $1 \le k < n$.

Example 4.2. *Suppose* $\mathsf{F} = \mathbb{C}$ *and* $\omega = e^{2\pi i/8}$*. Then* ω *is an 8-PRU.*

Example 4.3. *Consider the "Fermat prime",* $m = 2^4 + 1$*. 3 is a 16-PRU in* $\mathbb{Z}_m$*.*

Example 4.4. *More generally if p is a prime and* 2^k *divides* $p-1$ *then* $\mathbb{Z}_p$ *has a* 2^k*th primitive root of unity. Why?*

Exercise 4.5. *If* ω *is an n-PRU, then show that* ω^{-1} *is also. If n is even, then show that* ω^2 *is a* $\frac{n}{2}$*-PRU.*

Recall the definition of a Vandermonde matrix:

$$\mathrm{VDM}(u_1, \ldots, u_n) = \begin{pmatrix} u_1^0 & u_1^1 & \ldots & u_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ u_n^0 & u_n^1 & \ldots & u_n^{n-1} \end{pmatrix}.$$

Suppose $a = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \in \mathsf{F}[x]$. We see that

$$\text{If } \overline{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \in \mathsf{F}^n \quad \text{then} \quad \mathrm{VDM}(u_1, \ldots, u_n)\overline{a} = \begin{pmatrix} a(u_1) \\ a(u_2) \\ \vdots \\ a(u_{n-1}) \end{pmatrix} \in \mathsf{F}^n.$$

Now build a Vandermonde matrix on roots of unity. Let $V(\omega)$ denote $\mathrm{VDM}(\omega^0, \omega^1, \ldots, \omega^{n-1})$. Then

$$V(\omega) = \mathrm{VDM}(\omega^0, \omega^1, \omega^2, \ldots, \omega^{n-1}) = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{(n-1)} & \ldots & \omega^{(n-1)^2} \end{pmatrix}$$

$$V(\omega^{-1}) = \mathrm{VDM}(1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(n-1)}) = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \cdots & \omega^{-2(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \ldots & \omega^{-(n-1)^2} \end{pmatrix}$$

Theorem 4.6. *Let ω be an n-PRU. Then $V(\omega) \cdot V(\omega^{-1}) = nI$, where I is the $n \times n$ identity matrix.*

Proof. Let

$$\begin{aligned} u = (V(\omega) \cdot V(\omega^{-1}))_{ij} &= \sum_{0 \le k < n} V(\omega)_{ik} V(\omega^{-1})_{kj} \\ &= \sum_{0 \le k < n} \omega^{ik} \omega^{-kj} \\ &= \sum_{0 \le k < n} (\omega^{i-j})^k. \end{aligned}$$

If $i = j$, then $u = \sum_k 1 = n$. If $i \neq j$ then

$$u = \sum_{0 \le k < n} \omega^{(i-j)k} = \frac{\omega^{(i-j)n} - 1}{\omega^{i-j} - 1} = 0,$$

by the previous lemma. □

This makes it particularly easy to compute the inverse of the matrix $V(\omega)$ since

$$(V(\omega))^{-1} = n^{-1} V(\omega^{-1}).$$

Definition 4.7. *Let $\omega \in \mathsf{F}$ be an n-PRU. Then the mapping*

$$\begin{aligned} DFT(\omega): \quad \mathsf{F}^n &\to \mathsf{F}^n \\ \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix} &\mapsto \begin{pmatrix} b_0 \\ \vdots \\ b_{n-1} \end{pmatrix}, \quad b_j = \sum_{0 \le k < n} a_k \omega^{jk}, \\ &\mapsto V(\omega) \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix} \end{aligned}$$

is called the Discrete Fourier Transform. For a polynomial $a = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} \in \mathsf{F}[x]$ *we mean the DFT applied to* $(a_0, \ldots, a_{n-1})$.

4.2 The Fast Fourier Transform

Evaluating the DFT is simply evaluating the linear map given by the matrix $V(\omega)$. Another view is that it evaluates the polynomial with coefficients $a_0, \ldots, a_{n-1}$ at the powers of ω. Below we discuss an important algorithm that computes the DFT quickly. The relation between the DFT and the Vandermonde matrix shows that the inverse DFT can then also be computed quickly.

Let's look at the problem of computing the DFT. To start with, consider the problem of evaluating a polynomial $f = \sum_k a_k x^k \in \mathsf{F}[x]$ at the points at $+1, -1$.

- Separate the polynomial into its even and odd powered terms.
  Then $f(x) = f_{even}(x^2) + x f_{odd}(x^2)$, where $f_{even} = \sum_k a_{2k} x^k$ and $f_{odd} = \sum_k a_{2k+1} x^k$.
- Now $f(1) = f_{even}(1) + f_{odd}(1)$ and $f(-1) = f_{even}(1) - f_{odd}(1)$.
- The process of evaluating f at $+1, -1$ has been reduced to the problem of evaluating two polynomials of half the degree of f at 1.
- If we could continue this recursively with the square root of -1 etc., we can evaluate the polynomial quickly for more distinct points.

This process was discovered by Cooley and Tukey in 1965. The method became known as the *Fast Fourier Transform*. It is arguably the second most important algorithm in practice. (The most important one is fast sorting.)

Theorem 4.8. *Let n be a power of 2 and $\omega \in \mathsf{F}$ be an n-PRU. Then DFT(ω) can be computed using $O(n \log n)$ field operations in F.*

Proof. We wish to compute

$$\begin{aligned} f(\omega^k) &= \sum_{0 \le j < n} a_j \omega^{kj} \\ &= a_0 + a_2(\omega^{2k})^1 + a_4(\omega^{2k})^2 + a_6(\omega^{2k})^3 + \cdots \\ &\qquad \cdots + \omega^k \left(a_1 + a_3(\omega^{2k})^1 + a_5(\omega^{2k})^2 + \ldots \right) \\ &= f_{even}(\omega^{2k}) + \omega^k f_{odd}(\omega^{2k}). \end{aligned}$$

Again, we split the polynomial $f = \sum a_j x^j$ into polynomials f_{even}, f_{odd} whose coefficients are the odd numbered and even numbered coefficients of a, respectively:

$$f_{even} = \sum_{0 \le j < n/2} a_{2j} x^j \text{ and } f_{odd} = \sum_{0 \le j < n/2} a_{2j+1} x^j.$$

so $f(x) = f_{even}(x^2) + x f_{odd}(x^2)$. We have thus reduced the problem of computing $\mathrm{DFT}(\omega)(f)$ to computing the powers $\omega^2, \cdots, \omega^n$, then $\mathrm{DFT}(\omega^2)(f_{even})$ and $\mathrm{DFT}(\omega^2)(f_{odd})$, plus n field multiplications and n field additions. If $T(n)$ is the cost for size n, we have $T(n) = 2T(n/2) + 3n$. We deduce that $T(n) = O(n \log n)$. □

4.3 Fast polynomial multiplication with the FFT

Theorem 4.9. *Let F be a field with characteristic different from two. Let $n = 2^k$ for some $k \in \mathbb{N}$, and $\omega \in \mathsf{F}$ an n-PRU. Then multiplication of polynomials in $\mathsf{F}[x]$ of degree $n/2$ can be performed using $O(n \log n)$ field operations.*

Proof. Let $a, b \in \mathsf{F}[x]$ have degree $< n/2$. We assume that

$$a = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n/2-1} x^{n/2-1}$$
$$b = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n/2-1} x^{n/2-1}$$

The product $c = ab$ has degree less than n, and hence is uniquely determined by its values at n evaluations. Let

$$\overline{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n/2-1} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \mathsf{F}^n \qquad \overline{b} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n/2-1} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Then

$$\mathrm{DFT}(\omega)(\overline{a}) = V(\omega)\overline{a} = \begin{pmatrix} a(\omega^0) \\ a(\omega^1) \\ \vdots \\ a(\omega^{2n-1}) \end{pmatrix} \qquad \mathrm{DFT}(\omega)(\overline{b}) = V(\omega)\overline{b} = \begin{pmatrix} b(\omega^0) \\ b(\omega^1) \\ \vdots \\ b(\omega^{2n-1}) \end{pmatrix}$$

We can compute $c(\omega^i) = a(\omega^i)b(\omega^i)$, for $0 \le i < n$ with n multiplications in F.

Let $w = \begin{pmatrix} c(\omega^0) \\ c(\omega^1) \\ \vdots \\ c(\omega^{2n-1}). \end{pmatrix}$. Then $w = \mathrm{DFT}(\omega)(c) = \mathrm{DFT}(\omega)(a) \cdot \mathrm{DFT}(\omega)(b)$,

where the multiplication $(\cdot)$ is the component-wise product. We can then compute

$$c = \mathrm{DFT}(\omega)^{-1}(w) = \mathrm{DFT}(\omega)^{-1}(\mathrm{DFT}(\omega)(c)) = (1/n)V(\omega^{-1})w.$$

This looks a lot like the non-scalar algorithm we described earlier. To conclude, the steps to compute the multiplication are now:

(1) compute $u = \mathrm{DFT}(\omega)(a) = V(\omega)\,\overline{a}$,

(2) compute $v = \mathrm{DFT}(\omega)(b) = V(\omega)\,\overline{b}$,

(3) compute $w = u \cdot v$,

(4) compute $c = \mathrm{DFT}(\omega^{-1})(w) = (1/n)V(\omega^{-1})w$.

Steps 1, 2 and 4 take $O(n\log n)$ field operations, and Step 3 takes $O(n)$ field operations. Thus computing the product takes $O(n\log n)$ time. □

To multiply two arbitrary polynomials of degree less than m, we only need a 2^k-PRU, where $2^k > 2m$.

Definition 4.10. *We say that a field* F *supports the FFT if* F *has a* 2^ℓ*-PRU for any* $\ell \in \mathbb{N}$.

Of course we can perform FFT-based multiplication of polynomials over a field so long as the field has a sufficient large 2^ℓth primitive root of unity.

Theorem 4.11. *If* F *supports the FFT, then polynomials in* $\mathsf{F}[x]$ *of degree at most n can be multiplied with* $O(n\log n)$ *field operations.*

4.4 One prime FFT integer multiplication

Multiplying multi-precision integers is closely related to multiplying polynomials. In fact, we use polynomial multiplication to do integer multiplication. We assume that the numbers involved actually do have a reasonable upper bound. We'll start with a simple version which we'll call "one-prime" FFT integer multiplication. This will allow us to multiply numbers up to with less than 49128 bits. Slightly fancier versions allow us to go far beyond this.

Assume that our numbers are written in base 2^{24}. That is

$$a = (-1)^{s_a}\left(a_0 + a_1\cdot 2^{24} + a_2\cdot 2^{24\cdot 2} + \cdots + a_{\ell-1}\cdot 2^{24\cdot(\ell-1)}\right)$$
$$b = (-1)^{s_b}\left(b_0 + b_1\cdot 2^{24} + b_2\cdot 2^{24\cdot 2} + \cdots + b_{\ell-1}\cdot 2^{24\cdot(\ell-1)}\right),$$

where $s_a, s_b \in \{0,1\}$ are the sign bits. Let

$$A = a_0 + a_1 x + a_2\cdot x^2 + \cdots + a_{\ell-1}\cdot x^{\ell-1}$$
$$B = b_0 + b_1 x + b_2\cdot x^2 + \cdots + b_{\ell-1}\cdot x^{\ell-1}$$

Then $ab = (-1)^{s_a}A(2^{24})\cdot(-1)^{s_b}B(2^{24}) = (-1)^{s_a+s_b}(AB)(2^{24})$. We start by computing the product $A\cdot B \in \mathbb{Z}[x]$ and then evaluate to get $(A\cdot B)(2^{24}) = \pm ab$. How big are the coefficients of $A\cdot B \in \mathbb{Z}[x]$?

$$A\cdot B = \sum_{0\le k\le 2\ell-2} x^k \sum_{\substack{i+j=k\\ 0\le i,j<\ell}} a_i b_j$$

We know that $0 \le a_i < 2^{24}$ and $0 \le b_i < 2^{24}$. Thus $0 \le a_i b_j < 2^{48}$ and the largest coefficient of $A \cdot B$ is less than $\ell \cdot 2^{48}$. Let's assume that p is a prime greater than 2^{57}. Then any integer less than 2^{57} is uniquely represented modulo p (that is, if we know c mod p and we know $0 \le c < p$ then we know c). So long as $\ell \cdot 2^{48} < 2^{57}$, we can recover $A \cdot B \in \mathbb{Z}[x]$ from $A \cdot B$ mod $p \in \mathbb{Z}_p[x]$. This is guaranteed if $\ell < 2^{57-48} = 2^9 = 2048$. Thus we can handle integers with 2048 digits base 2^{24}, or about 49128 binary digits.

We then choose a special p which allows us to do the FFT quickly. Choose the smallest k such that $p = k \cdot 2^{57} + 1$ is a prime. We find $k = 29$ works, and so $p = 4179340454199820289$, and $\omega = 21$ is a 2^{57}-PRU. Now we can compute $C = AB$ with $O(\ell \log \ell)$ operations modulo p (which are basic machine operations since $p < 2^{62}$). After we have computed the product $A \cdot B$, evaluating $C(2^{24})$ is easy.

4.5 Other results

Again the close relation between integers and polynomials leads us to ask whether these results can be extended to integer multiplication. In fact, the following result introduced the FFT into computer algebra.

Theorem 4.12. *(Schönhage and Strassen 1971) Integer multiplication can be performed using $O(n(\log n)(\operatorname{loglog} n))$ bit operations.*

The method uses the FFT. The extra factor $\operatorname{loglog} n$ is caused by the fact that the PRUs are not available in the ring of integers, but "virtual PRUs" have to be "constructed" within the algorithm. Schönhage and Strassen also applied their method to polynomial multiplication in time $O(n(\log n)(\operatorname{loglog} n))$, without our assumption that "F supports the FFT". Schönhage (1974) solved the additional complication that occurs in characteristic two. The most general result is:

Theorem 4.13. *(Cantor and Kaltofen (1991)) Over any ring R, polynomials of degree bounded by n can be multiplied using $O(n(\log n)(\operatorname{loglog} n))$ operations.*

Again, their technique involves adding "virtual" roots of unity to R, if they are not already present. A notable feature of Cantor and Kaltofen's result is its generality: in the theorem "any ring" means the ring just needs to support the operations $\{+, -, \times\}$. The ring does not need to be to be commutative and does not even need to have a multiplicative identity element (there is a technique to embed the ring R into a larger ring that does have an identity).

The result of Theorem 4.12 stood for 36 years. In 2007, Fürer gave an algorithm that replaced the $\operatorname{loglog} n$ factor with $2^{\log^* n}$, where $\log^* n$ is the *iterated logarithm*. (You are encouraged to look up the definition. Here, we only mention that $2^{\log^* n}$ is essentially a constant even for astronomically large values of n.) Fürer's breakthrough inspired subsequent research. The following result confirms a longstanding conjecture about the upper bound for the cost of integer multiplication, in particular that the additional $\operatorname{loglog} n$ factor in Schönhage and Strassen's algorithm is not required.

Theorem 4.14. *(Harvey and van der Hoeven 2021) Integer multiplication can be performed using $O(n \log n)$ bit operations.*