The Fast Fourier Transform

Andreas Klappenecker

Motivation

There are few algorithms that had more impact on modern society than the fast Fourier transform and its relatives.

The applications of the fast Fourier transform touch nearly every area of science and engineering in some way.

For example, it changed medicine by enabling magnetic resonance imaging. It sparked a revolution in the music industry. It even finds uses in applications such as the fast multiplication of large integers.

A Brief History

- Gauss (1805, 1866). Used the FFT in calculations in astronomy.
- Danielson-Lanczos (1942). Gave an efficient algorithm, but low impact as digital computer were just emerging.
- Cooley-Tukey (1965). Rediscovered and popularized FFT. The importance was immediately recognized, and this is one of the most widely cited papers in science and engineering.

Key Property

The fast Fourier transform allows one to quickly multiply polynomials, that is, given

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

$B(x) = b_0 + b_1 x + \ldots + b_{n-1} x^{n-1}$

calculate $C(x) = A(x)B(x) = \Sigma_{i,j}\ a_i b_j\ x^{i+j} = \Sigma_k\ (\Sigma_i\ a_{k-i}\ b_i\ )\ x^k$

Representations of Polynomials

Coefficient Representation

A polynomial

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

can be represented in various ways. The most common way is to specify its coefficients $(a_0, a_1, \ldots, a_{n-1})$; this is called the coefficient representation.

Operations in C. Representation

Given polynomials in coefficient representation:

$A(x) = a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$ and $B(x) = b_0 + b_1x + \ldots + b_{n-1}x^{n-1}$

Addition in O(n):

$A(x)+B(x) = (a_0 +b_0) + (a_1+b_1)x + \ldots + (a_{n-1}+b_{n-1})x^{n-1}$

Evaluation in O(n) using Horner's scheme:

$A(w) = a_0 + (a_1+ \ldots(a_{n-3} + (a_{n-2} + a_{n-1}w)w)w\ldots)w$

Operations in C. Representation

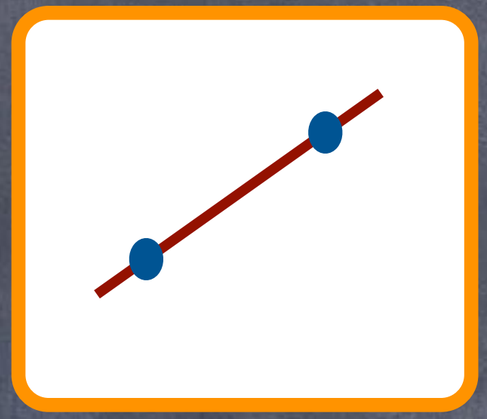
Given polynomials in coefficient representation:

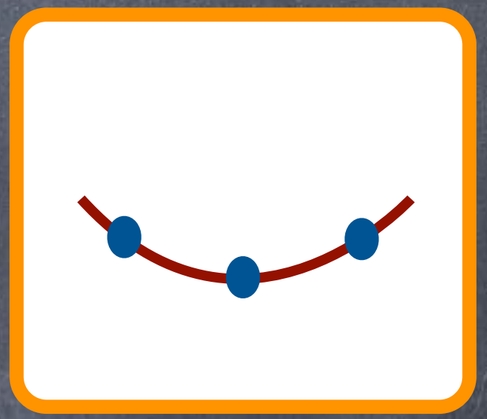
$A(x) = a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$ and $B(x) = b_0 + b_1x + \ldots + b_{n-1}x^{n-1}$

Multiplication in $O(n^2)$:

$A(x)B(x) = \Sigma_{i,j}\ a_ib_j\ x^{i+j} = \Sigma_k\ (\Sigma_i\ a_{k-i}\ b_i\ )\ x^k$

Point-Value Representation





A polynomial

$$A(x) = a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$$

can be understood as a function x -> y = A(x). We can specify a polynomial by n point and value pairs:

$$\{ (x_0, y_0), (x_1, y_1), \ldots , (x_{n-1}, y_{n-1}) \}$$

A polynomial of degree n-1 is uniquely specified by giving n point-value pairs for n distinct points.

Operations in PV Representation

Suppose that we are given two polynomials in PV representation:

$A(x)$: $\{ (x_0, y_0), (x_1, y_1), \ldots , (x_{n-1}, y_{n-1}) \}$

$B(x)$: $\{ (x_0, z_0), (x_1, z_1), \ldots , (x_{n-1}, z_{n-1}) \}$

Addition in $O(n)$:

$A(x) + B(x)$: $\{ (x_0, y_0+z_0), (x_1, y_1+z_1), \ldots , (x_{n-1}, y_{n-1}+z_{n-1}) \}$

Multiplication in $O(n)$, but need at least $2n-1$ distinct points:

$A(x)B(x)$: $\{ (x_0, y_0z_0), (x_1, y_1z_1), \ldots , (x_{2n-2}, y_{2n-2}z_{2n-2}) \}$

Operations in PV Representation

Evaluate. We can evaluate a polynomial in PV representation using some interpolation formula. Given

$A(x)$: $\{ (x_0, y_0), (x_1, y_1), \ldots , (x_{n-1}, y_{n-1}) \}$

one can evaluate at a point w e.g. using Lagrange's interpolation formula

$$A(w) = \Sigma_k \; y_k \; \Pi_{j:\, k \neq j} \; (w - x_j)/(x_k - x_j)$$

However, evaluation uses $O(n^2)$ operations.

Tradeoffs

Representation	Multiply	Evaluate
Coefficient	$O(n^2)$	$O(n)$
Point-value	$O(n)$	$O(n^2)$

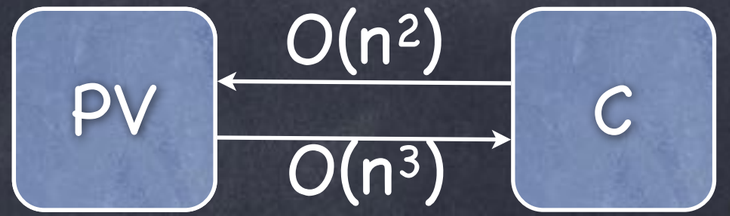
Converting between the Representations

For a polynomial $A(x) = a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$ of degree n-1, a conversion from coefficient representation to p.v. representation at n distinct points $x_0, x_1, \ldots, x_{n-1}$ can be done as follows:

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

The Vandermonde matrix is invertible iff the x_i's are distinct.

Drawback: Conversions are not fast!



The Fast Fourier Transform II

Andreas Klappenecker

Divide-and-Conquer

Motivation

We can speed up the evaluation by choosing suitable points $x_0,\ldots,x_{n-1}$ that have sufficient structure so that we can reuse computational results.

Divide

We can divide the polynomial A(x) by splitting it into its even and odd powers:

$$A(x) = A_{even}(x^2) + x\, A_{odd}(x^2)$$

For example, if $A(x) = a_0+a_1x+\ldots+a_7x^7$ then

$A_{even}(x) = a_0+a_2x+a_4x^2+a_6x^3$ and $A_{odd}(x) = a_1+a_3x+a_5x^2+a_7x^3$

Key Idea

Divide $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$ into even and odd powers:

- $A_{even}(x) = a_0 + a_2x + a_4x^2 + a_6x^3$ and
- $A_{odd}(x) = a_1 + a_3x + a_5x^2 + a_7x^3$.

Evaluate at two points w and -w by evaluating two smaller polys at w^2

- $A(w) = A_{even}(w^2) + w\, A_{odd}(w^2)$.
- $A(-w) = A_{even}(w^2) - w\, A_{odd}(w^2)$.

Example

Choose $w^2=1$, so that its square root $w=1$ or $w=-1$. Then

- $A(1) = A_{even}(1) + A_{odd}(1)$.
- $A(-1) = A_{even}(1) - A_{odd}(1)$.

So to evaluate A(x) at 1 and -1, we only need to evaluate two polynomials $A_{even}(1)$ and $A_{odd}(1)$.

Benefit: Evaluation of both $A_{even}(x)$ and $A_{odd}(x)$ has the same complexity as a single evaluation of A(x).

The FFT Trick

Our goal is to repeatedly use the same trick, so we need to use square roots of 1 (so 1 and -1) and square roots of -1 (so i and -i).

- $A(1) = A_{even}(1) + 1\,A_{odd}(1)$
- $A(-1) = A_{even}(1) - 1\,A_{odd}(1)$
- $A(i) = A_{even}(-1) + i\,A_{odd}(-1)$
- $A(-i) = A_{even}(-1) - i\,A_{odd}(-1)$

- $A_{even}(1) = A_{even,even}(1) + A_{even,odd}(1)$
- $A_{even}(-1) = A_{even,even}(1) - A_{even,odd}(1)$

- $A_{odd}(1) = A_{odd,even}(1) + A_{odd,odd}(1)$
- $A_{odd}(-1) = A_{odd,even}(1) - A_{odd,odd}(1)$

Roots of Unity

The evaluation at $n=2^k$ different points can be accomplished by repeatedly taking the square roots, starting with 1.

$\{1\} \rightarrow \{1, -1\} \rightarrow \{1, -1, i, -i\} \rightarrow \ldots \rightarrow \{ 1, \omega, \omega^2, \ldots \omega^{n-1} \}$

where ω is a primitive n-th root of unity, that is,

$\omega^n=1$ and $\omega^m \neq 1$ for $1 <= m < n$. We can choose $\omega = \exp(2\pi i/n)$.

Roots of Unity

The set of n-th roots of unity is explicitly given by

$$\{\omega^k = \exp(2\pi i\, k/n) \mid 0 \leq k \leq n-1\}\,.$$

Each number in this set has absolute value 1, since

$$\begin{aligned} |\exp(2\pi ik/n)|^2 &= \exp(2\pi ik/n)\exp(-2\pi ik/n) \\ &= \exp(0) = 1. \end{aligned}$$

Geometric Interpretation

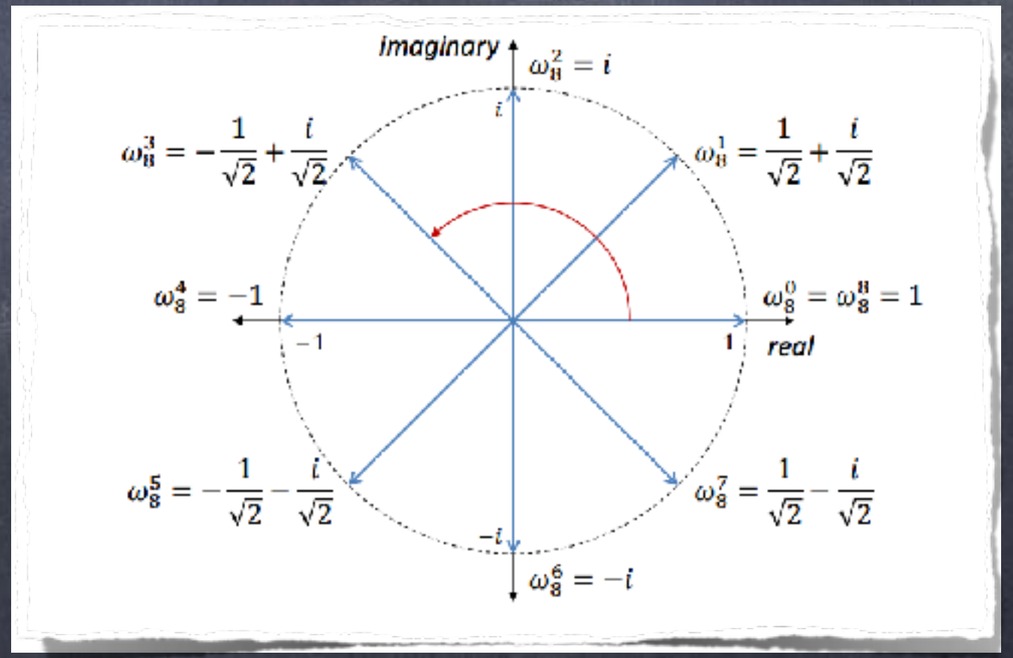
We can write the n-th roots of unity in the form

$$\omega^k = \exp(2\pi i\, k/n) = \cos(2\pi k/n) + i \sin(2\pi k/n),$$

so they lie on the unit circle.

For example,

$$\begin{aligned}\omega_8 &= \cos(2\pi/8) + i \sin(2\pi/8) \\ &= \cos(\pi/4) + i \sin(\pi/4) \\ &= \frac{1}{\sqrt{2}} + i\frac{1}{\sqrt{2}}.\end{aligned}$$



Properties of Roots of Unity

Suppose that n is even.

An n-th root of unity ω satisfies $\omega^{n/2} = -1$.

Indeed, $(\omega^{n/2})^2 = \omega^n = 1$, so $\omega^{n/2}$ must be equal to 1 or -1. However, 1 is impossible, as ω is a primitive n-th root of unity, hence the claim.

Consequence: $\omega^{n/2+j} = -\omega^j$ for all j.

Notice that $\{ (\omega^k)^2 : 0 <= k < n \} =: \{ \nu^m : 0 <= m < n/2 \}$ is the set of all n/2-th roots of unity.

Fast Fourier Transform

Evaluate a degree n-1 polynomial $A(x) = a_0 + \ldots + a_{n-1}\, x^{n-1}$ at its n^{th} roots of unity: $\omega^0, \omega^1, \ldots, \omega^{n-1}$.

Divide. Divide the polynomial into even and odd powers.

$$A(x) = A_{even}(x^2) + x\, A_{odd}(x^2).$$

Conquer. Evaluate $A_{even}(x)$ and $A_{odd}(x)$ at the $\frac{1}{2}n^{th}$ roots of unity: $\nu^0, \nu^1, \ldots, \nu^{n/2-1}$.

Combine.

$$A(\omega^k) = A_{even}(\nu^k) + \omega^k\, A_{odd}(\nu^k), \quad 0 \le k < n/2$$

$$A(\omega^{k+n/2}) = A_{even}(\nu^k) - \omega^k\, A_{odd}(\nu^k), \quad 0 \le k < n/2$$

FFT Algorithm

```
FFT(n, a_0,a_1,…,a_n-1) {
    if (n == 1) return a_0

    (e_0,e_1,…,e_n/2-1) ← FFT(n/2, a_0,a_2,a_4,…,a_n-2)
    (d_0,d_1,…,d_n/2-1) ← FFT(n/2, a_1,a_3,a_5,…,a_n-1)

    for k = 0 to n/2 - 1 {
        ω^k ← e^2πik/n
        y_k     ← e_k + ω^k d_k
        y_k+n/2 ← e_k - ω^k d_k
    }

    return (y_0,y_1,…,y_n-1)
}
```

Running Time of FFT

We have

$$T(n) = 2T(n/2) + \Theta(n)$$

Therefore, the running time is $T(n) = \Theta(n \log n)$.

Summary

- The FFT evaluates a polynomial of degree n-1 at n-th roots of unity in O(n log n) steps.
- Inverse of FFT just as fast.
- Can multiply two polynomials of degree n-1 in O(n log n) time using FFT of length 2n.
- Find more details in CLRS or Kleinberg/Tardos.