

9 Fast Fourier Transforms

9.1 References

- Wilkinson and Allen, chapter 11
- Quinn, chapter 8
- Sedgewick, chapter 36
- Knuth
- Aho, Hopcroft, and Ullman
- many other books

9.2 Background

- Theorem: Any oscillatory squiggle can be written as a sum of sine waves
- That is, any function with $f(t)$ a function of time t “in the time domain” can also be written “in the frequency domain” as

$$f(\theta) = \sum_{n \in \mathbb{N}} a_n \exp(2\pi i \theta / n)$$

- Going from $f(t)$ to get the coefficients a_n for $f(\theta)$ and back again is done by the *Fourier transform* and its inverse

9.3 Polynomial Multiplication

- Polynomial multiplication is actually the same problem
- Let $f(x) = \sum a_i x^i$ and $g(x) = \sum b_i x^i$
- Then $h(x) = f(x) \cdot g(x) = \sum c_i x^i$ where

$$c_0 = a_0 b_0$$

$$c_1 = a_0 b_1 + a_1 b_0$$

$$c_2 = a_0 b_2 + a_1 b_1 + a_2 b_0$$

$$c_3 = a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0$$

$$c_4 = a_0 b_4 + a_1 b_3 + a_2 b_2 + a_3 b_1 + a_4 b_0$$

- The coefficients c_i are *convolution products*
- In general, then, to multiply two polys of degrees n and m would take nm multiplies of all coeffs against all other coeffs (and then some adds that we don't count because addition is so much cheaper than multiplication)
- But maybe we can do it faster
 1. Evaluate the polys f and g at $2n + 1$ points (WLOG $n > m$)
 2. Multiply the $2n + 1$ values together
 3. We know that the poly passing through the points is unique and must be the product h of f and g
 4. Interpolate to determine the unique h
- This sounds like *more* work – How can this be faster?

- We will interpolate at roots of unity and use the convenient fact that sums of such things collapse to zero and thus won't need to be computed

9.4 The Cooley-Tukey Fast Fourier Transform

- An Important Aside On Hygiene and Cooking:

In the fall of the year we start thinking about the great American Thanksgiving holiday, when roasted turkey appears at the dinner in nearly every home. This is usually a big family holiday, and very often people eat so much and are having such an enjoyable time with their extended families who have travelled for the holiday that they forget to put away the leftovers properly.

But this is a bad thing. A turkey is a fairly large bird, and it can start to spoil rapidly if not properly refrigerated. One of the first signs of spoilage is that the leftover turkey will grow moldy and take on a fuzzy appearance as if it had fur.

The Cooley-Tukey Fast Fourier Transform

- An Important Aside On Hygiene and Cooking:

In the fall of the year we start thinking about the great American Thanksgiving holiday, when roasted turkey appears at the dinner in nearly every home. This is usually a big family holiday, and very often people eat so much and are having such an enjoyable time with their extended families who have travelled for the holiday that they forget to put away the leftovers properly.

But this is a bad thing. A turkey is a fairly large bird, and it can start to spoil rapidly if not properly refrigerated. One of the first signs of spoilage is that the leftover turkey will grow moldy and take on a fuzzy appearance as if it had fur.

The moral is this: Life is **exactly the opposite** of computing. Put the turkey in the refrigerator quickly. That is, use the Cooler Turkey Algorithm to avoid the Faster Furrier Transform.

The Fourier Transform

- We'll use an 8-point FFT as an example
- Let ω be a primitive 8th root of unity, that is, $\omega = \exp(2\pi i/8)$, so $\omega^8 = 1$
- Consider the matrix

$$F = (\omega^{ij}) = \begin{pmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ \omega^0 & \omega^4 & \omega^0 & \omega^4 & \omega^0 & \omega^4 & \omega^0 & \omega^4 \\ \omega^0 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega^1 & \omega^6 & \omega^3 \\ \omega^0 & \omega^6 & \omega^4 & \omega^2 & \omega^0 & \omega^6 & \omega^4 & \omega^2 \\ \omega^0 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & -1 & -\omega^1 & -\omega^2 & -\omega^3 \\ 1 & \omega^2 & -1 & -\omega^2 & 1 & \omega^2 & -1 & -\omega^2 \\ 1 & \omega^3 & -\omega^2 & \omega^1 & -1 & -\omega^3 & \omega^2 & -\omega^1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -\omega^1 & \omega^2 & -\omega^3 & -1 & \omega^1 & -\omega^2 & \omega^3 \\ 1 & -\omega^2 & -1 & \omega^2 & 1 & -\omega^2 & -1 & \omega^2 \\ 1 & -\omega^3 & -\omega^2 & -\omega^1 & -1 & \omega^3 & \omega^2 & \omega^1 \end{pmatrix}$$

- We note that the inverse $F^{-1} = (\omega^{-ij})$

- Then if $f(x) = a_0 + a_1x + a_2x^2 + \dots + a_7x^7$ the *discrete Fourier transform* of f is

$$\begin{pmatrix} f(\omega^0) \\ f(\omega^1) \\ f(\omega^2) \\ f(\omega^3) \\ f(\omega^4) \\ f(\omega^5) \\ f(\omega^6) \\ f(\omega^7) \end{pmatrix} = F \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix}$$

- The Fourier transform takes coefficients to points
- The inverse transform takes points to coefficients
- Transform is a column with entries

$$f(\omega^k) = \sum_{j=0}^{n-1} a_j \omega^{jk}$$

The Cooley-Tukey Fast Fourier Transform

- (From Quinn, pp. 206ff)
- Now, the *fast* part of the FFT
- We don't really have to do the n^2 mults of the matrix mult
- We can use the structure of the roots of unity to do the Fourier transform in $O(n \lg n)$ time
- To evaluate a poly $f(x)$ of degree n at the n -th roots of unity, let

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

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

First poly has even and second poly has odd powers of x

- Now we have

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

so to evaluate $f(x)$ at the n -th roots of unity we need to evaluate $f^{[0]}(x)$ and $f^{[1]}(x)$ at

$$(\omega^0)^2, (\omega^1)^2, (\omega^2)^2, \dots, (\omega^{n-1})^2$$

and then in n mults we get $f(x)$ at all n -th roots

- Theorem: If n is positive and even, the squares of the n -th roots of unity are identical to the $n/2$ -th roots of unity
- Apply the reduction recursively: To compute the n -th order FT, we need to evaluate two polys at the $n/2$ -th roots of unity and then do n mults.

- That is, to compute the n -th order FT, we need to do two $n/2$ -th order FTs and then do n mults.
- Recursively,

$$\begin{aligned}T(n) &= 2T\left(\frac{n}{2}\right) + n \\&= 4T\left(\frac{n}{4}\right) + n + n/2 \\&= \dots \\&= n \lg n + n(1 + 1/2 + 1/4 + \dots) \\&= n \lg n + 2n \\&= O(n \lg n)\end{aligned}$$

- Another formulation (Wilkinson, pp. 356ff)
- Start with

$$\begin{aligned}
 F_k &= f(\omega^k) = \sum_{j=0}^{n-1} a_j \omega^{jk} \\
 &= \sum_{j=0}^{(n/2)-1} a_{2j} \omega^{2jk} + \sum_{j=0}^{(n/2)-1} a_{2j+1} \omega^{(2j+1)k} \\
 &= \sum_{j=0}^{(n/2)-1} a_{2j} \omega^{2jk} + \omega^k \sum_{j=0}^{(n/2)-1} a_{2j+1} \omega^{2jk} \\
 &= F_k^{even} + \omega^k F_k^{odd}
 \end{aligned}$$

- Now, for $0 \leq k \leq n/2 - 1$ we use the antisymmetry of the roots of unity to write

$$\begin{aligned}
 F_k &= F_k^{even} + \omega^k F_k^{odd} \\
 F_{k+n/2} &= F_k^{even} - \omega^k F_k^{odd}
 \end{aligned}$$

since $\omega^{k+n/2} = -\omega^k$

- So it's a plus and minus of the smaller transforms
- Wilkinson, figures 11-28 and 11-30
- Now, how do we implement this so as to be efficient on parallel machines?
- Quinn, pp 207ff