FFT Explanation (Fast Fourier Transform)

Daniel Sudzilouski
Olin College of Engineering, Needham MA

Overview

Suppose we wanted to multiply two polynomials together of equal order N.
We can naively find $C(x) = A(x) * B(x)$ by foiling but if the order of A(x) and B(x) is N, then this process would taken $O(N^2)$ run-time.

From fundamental properties of algebra, we know that A(x) and B(x) evaluated at $N + 1$ distinct points uniquely characterize the polynomials. Furthermore, if

$$\begin{aligned} A(x_1) &= y_{a1} \\ B(x_1) &= y_{b1} \\ C(x_1) &= y_{a1} * y_{b1} \end{aligned}$$

This means that if we can pick N+1 unique points and evaluate A(x) and B(x) at those points, we can evaluate C(x) at those points in additional O(N) time. Since we have evaluated C(x) at N+1 distinct points, we have also uniquely characterized C(x).

We refer to the evaluation of C(x) at N+1 points as FFT which can be accomplished in $O(N * log(N))$ runtime. IFFT is the process of interpolating the evaluation of C(x) back to the coefficients of the polynomial representing it which can also be done in $O(N * log(N))$.

N Roots of Unity

The core of FFT relies on utilizing properties of solutions to the equation of type:

$$Z^N = 1 \quad Z \in \mathbb{C}, N \in \mathbb{N}$$

When $N = 2$, for examples, the two roots of unity are $Z = 1$ and $Z = -1$. To find solutions for an arbitrary N, we can use properties of the exponential function $exp(x)$.

$$exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!}...$$

A useful property of exp(x) for complex numbers is eulers identity which can derived by refactoring the exp(x) series into the taylor series for sin(x) and cos(x).

$$exp(i * x) = \sum_{n=0}^{\infty} \frac{(ix)^n}{n!} = 1 + ix + \frac{(ix)^2}{2!} + \frac{(ix)^3}{3!}...$$

$$exp(i * x) = (1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!}...) + i(x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}...)$$

$$e^{ix} = cos(x) + i * sin(x)$$

Often times this is written as e^{ix} as typical exponential properties from the reals follow from showing that $exp(a + b) = exp(a) * exp(b), a \in \mathbb{C}, b \in \mathbb{C}$.

To solve the originally mentioned equation $Z^N = 1$ we can observe that for $k \in \mathbb{N}$:

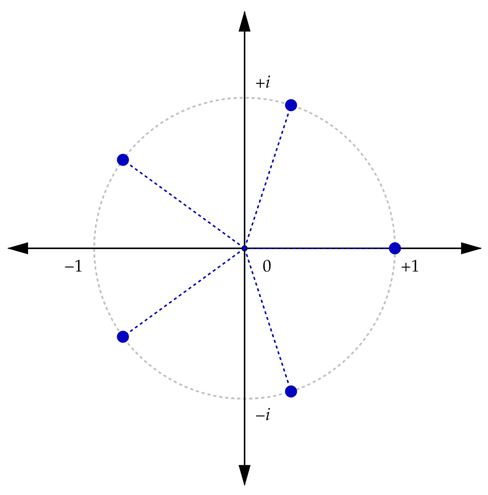
$$(exp(\frac{2\pi ki}{N}))^N = exp(\frac{2\pi ki}{N}N) = exp(2\pi ki)$$

$$exp(2\pi ki) = cos(2\pi ki) + sin(2\pi ki) = 1$$

Conventionally the N roots of unity are written in the form:

$$[w^0, w^1, w^2..w^{n-1}], w = exp(\frac{-2\pi i}{n})$$

From the solution of the roots of unity, we see that they lie on unit circle in the complex plane. For example when N=5, the roots are shown below:



wikipedia.org/wiki/Root_of_unity#/media/File:One5Root.svg

References

[1] Khan Academy. *Euler's formula & Euler's identity*. URL: https://www.khanacademy.org/math/ap-calculus-bc/bc-series-new/bc-10-14/v/euler-s-formula-and-euler-s%20identity.

[2] MIT. *Lecture 3: Divide & Conquer: FFT*. URL: https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2015/resources/lecture-3-divide-conquer-fft/.

[3] Reducible. *The Fast Fourier Transform (FFT): Most Ingenious Algorithm Ever?* URL: https://www.youtube.com/watch?v=h7ap07q16V0.

[4] wikipedia. *Roots of unity*. URL: https://en.wikipedia.org/wiki/Root_of_unity.

FFT

Let's factor an example A(x) polynomial into even and odd powers:

$$A(x) = (a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0)$$

$$A(x) = (a_4x^4 + a_2x^2 + a_0) + (a_3x^3 + a_1x^1)$$

$$A(x) = (a_4(x^2)^2 + a_2(x^2)^1 + a_0) + x(a_3(x^2)^1 + a_1(x^2)^0)$$

$$A(x) = Pe(x^2) + xPo(x^2)$$

$$A(-x) = Pe(x^2) - xPo(x^2)$$

If our input points are positive/negative paired $[\pm x_0, \pm x_1, \pm x_2, ... \pm x_{n-1}]$, we only need to evaluate half the points and then trivially can reconstruct the evaluation for the other pair of points.

If we can always square half the points (one from each pairing) and create a new positive/negative pairing $[\pm x'_0 \pm x'_1, \pm x'_2, ... \pm x'_{(n/2)-1}]$, we can write a divide and conquer algorithm that runs in $O(NlogN)$ time.

Plugging in the n roots of unity where $n = 2^k, k \in \mathbb{N}$ as our points, we see that this preserves the required invariants. The points $[w^0, w^1, ...w^{(n/2)-1}]$ are $\pm$ paired with the points $[w^{(n/2)}, w^{(n/2)+1}, ...w^{(n-1)}]$ by the symmetry of construction. Formally:

$$w^k = -w^{(k+\frac{n}{2})} \mod n$$

Squaring the first $\frac{n}{2}$ points we can see that we have constructed the roots of unity for $n' = \frac{n}{2}$:

$$[(w^0)^2, (w^1)^2, ...(w^{(n/2)-1})^2]$$

$$[(exp(0(\frac{-2\pi i}{n}))^2, (exp(1(\frac{-2\pi i}{n}))^2, ...(exp((\frac{n}{2} - 1)(\frac{-2\pi i}{n}))^2]$$

$$[(exp(0(\frac{-2\pi i}{n/2})), (exp(1(\frac{-2\pi i}{n/2})), ...(exp((\frac{n}{2} - 1)(\frac{-2\pi i}{n/2}))]$$

$$[(exp(0(\frac{-2\pi i}{n'})), (exp(1(\frac{-2\pi i}{n'})), ...(exp((\frac{n}{2} - 1)(\frac{-2\pi i}{n'}))]$$

IFFT

The operation accomplished under the FFT section is equivalent to applying the Fourier matrix onto the vector representing the polynomial coefficients:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & w^1 & w^2 & ... & w^{n-1} \\ 1 & w^2 & w^4 & ... & w^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & w^{n-1} & w^{2(n-1)} & ... & w^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

You can verify that the inverse of the Fourier matrix is as follows:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & w^{-1} & w^{-2} & ... & w^{-(n-1)} \\ 1 & w^{-2} & w^{-4} & ... & w^{-2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & w^{-1(n-1)} & w^{-2(n-1)} & ... & w^{-(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

The similarity between the Fourier matrix F_n and it's inverse $(F_n)^{-1}$ allows reusing the exact same FFT logic to perform IFFT. The only change you need to make is to redefine w' as w^{-1} and scale the resulting matrix by $\frac{1}{n}$.

Implementation C++20

```
using cd = complex<double>;
vector<cd> fft(vector<cd> polynomial, bool inverse=false) {
    if(polynomial.size() <= 1) return polynomial;
    while(polynomial.size() & (polynomial.size()-1))
      polynomial.push_back(0);
    int n = polynomial.size();
    vector<cd> even, odd, result(polynomial.size());
    for(int i=0;i<n;i++) {
        if(i%2 == 0) even.push_back(polynomial[i]);
        else odd.push_back(polynomial[i]);
    }
    even = fft(even,inverse);
    odd = fft(odd,inverse);
    for(int j=0;j<n/2;j++) {
        cd wn = exp(cd(0,(inverse?-1:1)*2*numbers::pi*j/n));
        result[j] = (even[j] + wn*odd[j]) / (inverse?cd(2):cd(1));
        result[j+n/2] = (even[j] - wn*odd[j]) / (inverse?cd(2):cd(1));
    }
    return result;
}
```