1 FFT with Application on Polynomial Multiplication

1.1 Introduction

The fast Fourier transform (FFT) is a name given to any algorithm that is able to compute the discrete Fourier transform (DFT) of a function over the N^{th} roots of unity in $O(N \log_2 N)$ time with regards to addition and multiplication of complex numbers. It is *fast* in the sense that under general settings to compute the DFT of any given function, one has to compute and multiply with their respective frequencies N Fourier coefficients. With each coefficient being a sum of N products of multiplications, this computation effectively takes $O(N^2)$ operations. This improvement in efficiency is achieved through realizing properties of the N^{th} roots of unity such that redundant calculations would be eliminated and thereby giving us the speed-up we desire. In this section we will first show the significance of a fast polynomial multiplication algorithm and then generalize it to the case of FFT.

1.2 Operations on Polynomials in Different Forms

The main purpose of this subsection is to provide an overview on the pros and cons of doing polynomial operations in different representations and see if we could find the most efficient method of doing these operations. In the following subsections we will devise an algorithm which will complete our search for efficiency. We first start by going over the operations we will cover on any degree $n-1$ polynomial of the general form $P(x) = \sum_{i=0}^{n-1} p_i x^i$ for scalars $(p_i)_{i=0}^{n-1}$.

- **Evaluation:** Given a polynomial $P(x)$ we can evaluate it at point x by feeding it x to receive an output y.
- **Addition:** Given two polynomials $A(x) = \sum_{i=0}^{n-1} a_i x^i$ and $B(x) = \sum_{i=0}^{n-1} b_i x^i$ we can add them together to obtain a third polynomial $C(x) = \sum_{i=0}^{n-1} (a_i + b_i) x^i = \sum_{i=0}^{n-1} c_i x^i$.
- **Multiplication:** Given two polynomials $A(x)$ and $B(x)$ as described above, we can most certainly multiply them for a third polynomial $C(x) = \sum_{i=0}^{2(n-1)} c_i x^i$ where $c_i = \sum_{j=0}^{i} a_j b_{i-j}$. Note that this multiplication defines a vector $(c_0, c_1, ..., c_{2(n-1)})$ which describes exactly the

convolution of $(a_0, ..., a_n)$ and $(b_0, ..., b_n)$ with respect to their inner products. What this implies is that if we can multiply polynomials efficiently, we can convolute vectors efficiently, and thereby process signals efficiently given their vector forms.

To optimize the computational time of these three operations it is helpful to see what different representations of polynomials could provide for our purpose. There are namely three ways of preserving the information contained in a polynomial which are listed below.

- **Coefficient vector:** For any degree $n-1$ polynomial, we can write it as a vector in a vector space with the basis $\{1, x, x^2, ..., x^{n-1}\}$. E.g. the polynomial $P(x) = 1 + 3x + 54x^2$ is equivalent to the vector $(1, 3, 54)$ in a three-dimentional vector space using the aforementioned basis.

- **Roots:** If we know all roots of a polynomial along with a scalar constant surely we can construct the polynomial as a product of $(x - r_i)$'s multiplied by the constant c with each r_i being a root of the polynomial, as guaranteed by the fundamental theorem of algebra. E.g. the polynomial $P(x) = 4x^3 - 24x^2 + 44x - 24$ could be written as $P(x) = 4(x-3)(x-2)(x-1)$.

- **Samples:** A sample of a polynomial $P(x)$ is an ordered pair (x_i, y_i) where $y_i = P(x_i)$. Say $P(x)$ has degree $n-1$, then we require exactly n distinct samples to be able to uniquely determine a $P(x)$ that is coherent with our n samples. This could be accomplished through Lagrange's polynomial interpolation or solving a simple system of equations. E.g. if given three distinct samples (x_i, y_i) we could uniquely determine a polynomial by solving the system

$$\begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

  to obtain $P(x) = c_1 + c_2 x + c_3 x^2$.

With these three different representations we shall see that they each have their own perks and defects as far as our operations are concerned, which will then prompt the question of "is there a middle ground covering the best cases in each form?" The answer is no, sadly. However, we may build a connection using FFT to gain access the best parts of these representations. This will allow for a close-to-ideal solution.

1.3 Runtime Analysis on Polynomial Operations

For the following estimates we will assume our polynomials are of degree $n-1$.

We start with polynomial evaluation. In coefficient form to evalute a polynomial at point x will take at most $O(n)$ calculations using Horner's rule. Simply put, calculate and record each power of x by multiplying x with itself $n-1$ times, then multiplying each power of x with the corresponding coefficient takes another n multiplications, finally adding them together takes at most $n-1$ additions, so we would end up doing $3n-2$ operations which is in the order of linear time $O(n)$. Evaluation with roots is also in linear time, as each $(x-r_i)$ can be calculated with a single addition and multiplying $n-1$ of them together with a constant will take us $n-1$ multiplications. However with samples, to evalute the polynomial we would first have to recover the polynomial through interpolation and then evaluate. The bottleneck here rests on the interpolation step which takes $O(n^2)$ calculations to compute, as can be visualized from the example of solving a system of linear equations given above.

Polynomial addition is straight forward in coefficient form, since it can be computed by simply adding the two coefficient vectors together which would take $O(n)$ computations. Addition with only the roots of two polynomials is extremely difficult, and conversion to other forms would be impossible if we want to end up with the roots of the sum, so we do not consider it as a feasible approach in this paper. Addition with samples, however, takes only linear time. The sum of two polynomials evaluated at one point is simply the sum of the values of both polynomials evaluated at that point. Hence for n samples, performing addition takes $O(n)$ calculations.

Multiplication with coefficients would take $O(n^2)$ computations as it can be written as a double summation given earlier. With roots, one simply concatenate the $(x-r_i)$'s together, which takes $O(n)$ concatenations. With samples it takes only $O(n)$ operations as well, since multiplying the second entry of both samples at the same point accomplishes polynomial multiplication.

To summarize, we have the following chart that lists the computational complexities we have considered thus far.

	Coefficients	Roots	Samples
Evaluation	$O(n)$	$O(n)$	$O(n^2)$
Addition	$O(n)$	N/A	$O(n)$
Multiplication	$O(n^2)$	$O(n)$	$O(n)$

As the chart shows, no representation is perfect and there is a price to be paid no matter which representation one adopts. However, what if there is a method of efficiently converting between representations such that it would make sense to convert and perform the opertaion rather than accepting the fate of an undesirable $O(n^2)$ complexity? Indeed, if conversion could cost significantly less, we would be able to save quite some time. Luckily, it is possible to convert between coefficient form and sample form efficiently in $O(n \log_2 n)$ time, as we will show in the following subsections.

1.4 From Coefficients to Samples

Our objective in this section is to find an algorithm which can convert coefficient vectors into a set of samples efficiently so that we may employ this conversion as an escape route from the horrid $O(n^2)$ runtime it requries to perform polynomial multiplication.

To obtain a sufficient amount of samples from a polynomial, we evaluate our polynomial of interest $P(x) = \sum_{i=0}^{n-1} a_i x^i$ at some x_k in our domain for n such distinct x_k's. This ensures that the least degree polynomial we construct from the samples obtained matches $P(x)$. The following matrix multiplication accurately describes our sampling of polynomial at n distinct points:

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

If we could perform (1) efficiently and invert it efficiently as well, we could gain access to the ideal $O(n)$ running time it takes to perform polynomial multiplication with samples. Hence our goal is to figure out a way to compute (1) as efficiently as possible.

1.5 A Divide-and-Conquer Approach

The usual methods of performing matrix multiplication is to find the dot product of $\vec{a}$ with every row of the matrix it is being multiplied by. However, as each dot product takes $O(n)$ time to compute and we need to compute n of such dot products, the overall running time would be $O(n^2)$ which renders the conversion meaningless for an efficient polynomial multiplication. So at this point we will introduce a divide-and-conquer approach of computing (1) in hopes of a better running time.

The philosophy of divide-and-conquer algorithms is to break a problem down into smaller subproblems, and then break those subproblems down recursively in the same fashion until it is feasible to compute the subproblems without much effort. This approach aims to avoid direct computation of a large problem and instead shift the amount work that has to be done onto the division and combining of subproblems.

We will achieve the ability to recursively break the computation of the dot product between $\vec{a}$ and a row $\vec{x}$ in the matrix by defining the following:

$$\begin{aligned} P_{\text{even}}(x) &= \sum_{k=0}^{\lfloor (n-1)/2 \rfloor} a_{2k} x^k \\ P_{\text{odd}}(x) &= \sum_{k=0}^{\lfloor (n-1)/2 \rfloor} a_{2k+1} x^k \end{aligned}$$

which effectively splits $P(x)$ into two polynomials each half the size of $P(x)$ with only odd or even coefficients. However the powers of x in each sum does not match the coefficients they are multiplied with, so in order to write $P(x)$ as a combination of these two subproblems, we would have to write $P(x)$ as

$$P(x) = P_{\text{even}}(x^2) + x \cdot P_{\text{odd}}(x^2). \tag{2}$$

Recursively apply this method to P_even and P_odd and so on will allow us to compute only bite-size problems which are easily solvable.

Now it is important to note that we want to assume n is a power of two, as the smallest subproblem we want to handle is evaluation of monomials. The assumption does very little to hurt the generality of this algorithm as one can always fill a coefficient vector with trailing zeroes up to the closest power of two and be happy that the algorithm applies well to it.

With this recursion in hand, we want to analyze the resulting runtime. Let T measure the runtime of computing $P(x)$ for every x in the set X of points we want to sample our polynomial at, then we have the following recurrence equation relating the runtime of a problem to the runtime of its subproblems:

$$\begin{aligned} T(n, |X|) &= 2T(\frac{n}{2}, |X|) + O(|X|) \\ &= 2\left[2T(\frac{n}{4}, |X|) + O(|X|)\right] + O(|X|) \\ &\vdots \\ &= 2^{\log_2 n} T(1, |X|) + \sum_{l=0}^{(\log_2 n)-1} 2^l O(|X|) \\ &= n^2 + O(|X| \log_2 n) \\ &= O(n^2) \end{aligned}$$

where n is the degree of the polynomial concerned in each stage. In the very first equality our problem on $P(x)$ gets divided into problems on $P_{even}(x)$ and $P_{odd}(x)$ at the cost of $O(|X|)$ time used to split and combine the subproblems back into the original one on $P(x)$ for every x in X. By unrolling this recurrence equation we obtain the last equality.

Although we ended with yet another $O(n^2)$ algorithm, an important insight we gained here is that we only have $O(n^2)$ runtime as a result of needing to evaluate monomials for every x in X. This is the case as in general the size of X does not reduce when one square each element in X; we only need to evaluate subproblems at x^2 for each layer of recursion we add as a result of how (2) was constructed.

Now that we have identified the root cause of the congestion in this algorithm, we look for a set which reduces in size as we square each element. Since we only need n distinct samples of a polynomial to preserve its data with no other requirements, we are justified to evaluate $P(x)$ over any set X given the size of it is n.

1.6 The Nth Roots of Unity

To find a set which reduces in size as each element is squared, we start by taking the square-root of numbers as the square-root fucntion returns two

outputs which fit our purpose for every input we provide. Working reversely in this fashion, we acquire the progression below:

$$\{1\}$$
$$\{1, -1\}$$
$$\{1, -1, i, -i\}$$
$$\{1, -1, i, -i, \tfrac{\sqrt{2}}{2}(i+1), \tfrac{-\sqrt{2}}{2}(i+1), \tfrac{\sqrt{2}}{2}(i-1), \tfrac{-\sqrt{2}}{2}(i-1)\}.$$

This could be continued for as long as we would like, however at this point it is enough to highlight the important feature of these sets that is *every number listed above lies on the unit circle* on the complex plane. This property holds for as many square-roots as we would like to take on any number above.

To provide an explaination, we note that the last set above is simply the set

$$\{e^{2\pi i \frac{k}{8}}, k = 0, 1, ..., 7\} \tag{3}$$

of complex exponentials. From here we see that squaring any element in (3) simply doubles the value of k. Squaring the element four times will eventually yield

$$e^{2\pi i \frac{8k}{8}} = (e^{2\pi i \frac{8}{8}})^k = (1)^k$$

which is a result of Euler's formula. Geometrically we are spinning the point on the complex plane by its angle from $(1, 0)$ and doubling the magnitude we spin it by in every subsequent spin until it eventually rests on $(1, 0)$.

In general, the set

$$\{e^{2\pi i \frac{k}{n}}, k = 0, 1, ..., n-1\} \tag{4}$$

where n is a power of two reduces to $\{1\}$ when we take the n^{th} power of each element, or equivalently, square each element $\log_2 n$ times. Such set is what is known as the n^{th} roots of unity as the n^{th} power of any element evaluates to unity.

By sampling our polynomial over this specific set, we immediately see that every resulting subproblem, i.e. monomial, only has to be evaluated at $x = 1$ instead of n distinct values.

1.7 Completing the Algorithm

Revising (3) to take into account our choice of X, we have that

$$\begin{aligned} T(n,|X|) &= 2^{\log_2 n} T(1,|X^n|) + \sum_{l=0}^{(\log_2 n)-1} 2^l O(|X|) \\ &= n \cdot T(1,1) + \log_2 n O(|X|) \\ &= n + O(n \log_2 n) \\ &= O(n \log_2 n) \end{aligned}$$

which is exactly the efficient conversion from coefficient vectors into samples we were looking for.

This algorithm will enable us to convert two polynomials $A(x)$ and $B(x)$ from coefficient vectors into samples in $2O(n \log_2 n)$ time and perform multiplication in $O(n)$ time.

To convert the product polynomial $C(x)$ back into coefficient form, we shall prove that the inverse operation of

$$\begin{pmatrix} 1 & (e^{2\pi i \frac{0}{N}})^1 & (e^{2\pi i \frac{0}{N}})^2 & \cdots & (e^{2\pi i \frac{0}{N}})^{N-1} \\ 1 & (e^{2\pi i \frac{1}{N}})^1 & (e^{2\pi i \frac{1}{N}})^1 & \cdots & (e^{2\pi i \frac{1}{N}})^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (e^{2\pi i \frac{N-1}{N}})^1 & (e^{2\pi i \frac{N-1}{N}})^2 & \cdots & (e^{2\pi i \frac{N-1}{N}})^{N-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{pmatrix},$$

which we denote as $V\vec{a} = \vec{y}$ (V for Vandermonde), is simply

$$\frac{1}{n}\bar{V}\vec{y} = \vec{a}.$$

Or equivalently, we show that $V^{-1} = \frac{1}{n}\bar{V}$.

Proof. To show that $V^{-1} = \frac{1}{n}\bar{V}$, it is the same as showing that $V\bar{V} = nI$ where I is the identity matrix. We proceed simply by working out the product of matrices.

Let $M = V\bar{V}$, then

$$(m)_{a,b} = \sum_{k=0}^{n-1} v_{a,k}\bar{v}_{k,b} = \sum_{k=0}^{n-1} e^{2\pi i \frac{ak}{n}} e^{-2\pi i \frac{kb}{n}} = \sum_{k=0}^{n-1} e^{2\pi i \frac{ak-kb}{n}}.$$

If $a = b$, we have that $(m)_{a,b}$ is simply a sum of n e^0's, and this gives us that the diagonal entries of M must be n.

If $a \neq b$, we would have a power series to evaluate:

$$\begin{aligned}(m)_{a,b} &= \sum_{k=0}^{n-1} \left(e^{2\pi i \frac{a-b}{n}}\right)^k \\ &= \frac{\left(e^{2\pi i \frac{a-b}{n}}\right)^n - 1}{e^{2\pi i \frac{a-b}{n}} - 1} \\ &= 0.\end{aligned}$$

Now we know that any non-diagonal entries of M is 0, hence $M = nI$ as desired. □

We could apply the same divide-and-conquer technique to compute $\frac{1}{n}\bar{V}\vec{y} = \vec{a}$ and obtain our polynomial $C(x)$ as a vector of coefficients.

In summary, we pay $2O(n \log_2 n)$ to convert $A(x)$ and $B(x)$ into samples and perform multiplication in $O(n)$, then pay $O(n \log_2 n)$ time again to have our polynomial $C(x)$ in coefficient form. Here we assume $n-1$ is the degree of $C(x)$ as it takes n samples of $C(x)$ to accurately reconstruct a degree $\deg(A) + \deg(B)$ polynomial from them.

What this gives us is a way of convoluting two signals or waves represented by polynomial curves in a *fast* manner. Coupled with the Stone-Weierstrass Theorem which shows any continuous function can be approximated by polynomials, we now have a powerful tool in our hands to do greater things.

1.8 The Fast Fourier Transform

The discrete Fourier transform as described in [2] is given by

$$F(k) = \sum_{r=0}^{n-1} A_r e^{2\pi i \frac{k}{n} r}$$

$$\text{where } A_r = \frac{1}{N} \sum_{k=0}^{n-1} F(k) e^{-2\pi i \frac{k}{n} r}.$$

This correspond exactly to the equation $\frac{1}{n}\bar{V}\vec{y} = \vec{a}$ in the previous subsection if we let $\vec{a} = (F(0), F(1), \cdots, F(k)) = \vec{F}$. Following this trend, the

inverse opertaion would simply be $V\vec{F} = \vec{y}$. We have already shown that this transformation could be done in $O(n \log_2 n)$ time, hence the algorithm we described above is the FFT itself.

In the case of converting representations of polynomials, the coefficient form corresponds exactly to the frequency space one would conventionally denote in Fourier analysis, as a coefficient vector tells us exactly how much of each power of x a given polynomial has. Analogously with samples, they are in the time space. Once this realization has been made, it is immediately clear that we have used exactly the convolution theorem for inverse Fourier transform:

$$\begin{aligned} & \mathscr{F}^{-1}\{A(x) * B(x)\} = \mathscr{F}^{-1}\{A(x)\} \cdot \mathscr{F}^{-1}\{B(x)\} \\ \text{or} \quad & A(x) * B(x) = \mathscr{F}\{\mathscr{F}^{-1}\{A(x)\} \cdot \mathscr{F}^{-1}\{B(x)\}\}. \end{aligned}$$

This concludes the section which has set out to demonstrate the Radix-2 form of FFT, an algorithm first used by C.F. Gauss without recognition and then discovered again by James Cooley and John Tukey in 1965.

References

[1] Demaine, Erik D. "3. Divide & Conquer: FFT." YouTube. MIT OpenCourseWare, 04 Mar. 2016. Web. 07 Mar. 2017.

[2] Stein, Elias M., and Rami Shakarchi. "Chapter 7 Finite Fourier Analysis." Fourier Analysis: An Introduction. Princeton: Princeton UP, 2003. N. pag. Print.