# Understanding the FFT

Neil Jones

October 17, 2005

## Introduction

The FFT seems like a rather mysterious algorithm at first, but when you consider its limitations, it is less so. One reason that many people find it confusing is that textbooks often conflate the application of the FFT with the algorithm itself, for example by trying to explain all at once how to use the FFT to multiply large numbers. This is harder than it needs to be, because the FFT is fundamentally an operation on a single mathematical object, namely, a polynomial.

## Polynomials

Any polynomial can be represented in two ways: as a collection of coefficients, or as a collection of points. For example, the quadratic equation $Q(x) = x^2 - 2x + 1$ can be written $(1, -2, 1)$ in coefficient form, or as the three points $(1, 0); (0, 1); (-1, 4)$. One reason that the point representation is not commonly used to represent a polynomial is that it is quite difficult to calculate the polynomial at an arbitrary other point with this representation. For example, $Q(4.3)$ is difficult to calculate with the points representation—how this can be accomplished is described below—but it is much easier in the coefficients representation. On the other hand, the points representation does allow some operations to be done trivially. Consider the two polynomials $Q(x) = x^2 - 2x + 1$ and $R(x) = 3x^2 - x + 2$. In coefficient form, they are $Q = (1, -2, 1)$ and $R = (3, -1, 2)$ and in points form they are $Q = ((1, 0); (0, 1); (-1, 4))$ and $R = ((1, 4); (0, 2); (-1, 6))$. What is $Q(x)R(x)$? In coefficient form one needs to multiply all the terms and add them together, but in the points representation one can simply multiply the corresponding $y$ values to get the series $((1, 0); (0, 2); (-1, 24))$. Unfortunately, this last representation has only 3 points to represent a 4-th degree equation, but this is a minor technical problem, and it should be clear that the points representation is preferrable when you're multiplying two different polynomials.

A natural question is how one might go about transforming between the two representations for a given polynomial. It's clear how one can convert from a coefficient representation to a point representation: pick a sufficient number of points and carry out the calculations in one of the following ways (the following considers calculating a polynomial $A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_0$ at $x = 4$).

**Naive** Calculate $4^{n-1}$. Multiply by $a_{n-1}$. Calculate $4^{n-2}$. Multiply by $a_{n-2}$, and add to the previous result. Repeat until $a_0$. Calculating $4^n$ is $O(n)$, and there are $O(n)$ such terms, so this algorithm takes $O(n^2)$ time for each evaluation point.

**Methodical, step 1** Create a column vector $P$ such that $P[0] = 4$, and $P[i] = 4P[i-1]$. Create a row vector $a = [a_{n-1}, a_{n-2}, \ldots, a_0]$. Multiply $aP$ to get $A(4)$. This trades $O(n)$ time for $O(n)$ space and takes $O(n)$ time and $O(n)$ space.

**Methodical, step 2** Calculate $4a_{n-1} + a_{n-2}$. Multiply this by 4 and add $a_{n-3}$. Multiply this by 4 and add $a_{n-4}$, etc. This is $O(n)$ time, and $O(1)$ space (aside from the coefficients). This method is also called *Horner's rule* for polynomials.

Thus, for an arbitrary polynomial of degree $n$ and an arbitrary point, it will take $O(n)$ time to evaluate

the polynomial at that point if one assumes that multiplication and addition are $O(1)$ operations.[1] Since a degree-bound $n$ polynomial requires $n$ points to uniquely define it, the conversion of coefficients to *an arbitrary set* of points will require $O(n^2)$ time. This result is true in light of the DFT algorithm discussed below, since the DFT algorithm does not operate on an arbitrary set of points. The DFT takes $O(n \log n)$ time for the same task, but operates on a particular set of points that is determined by the degree of the polynomial.

The reverse conversion of a polynomial from points representation to coefficients representation is somewhat harder than the conversion of coefficients to points. The problem under consideration is to design an algorithm that can take $n$ pairs of points $(x_i, y_i)$ that represent a polynomial $A(x)$ of degree-bound $n$, and return the $n$ coefficients $(a_{n-1}, a_{n-2}, \ldots, a_0)$ such that $A(x_i) = \sum_{j=0}^{n-1} a_j x_i^j$. Conceptually, one could precompute all $n-1$ powers of $x_i$ and consider this problem to be a set of linear equations:

$$\begin{aligned} y_1 &= a_0 + x_1 a_1 + x_1^2 a_2 + x_1^3 a_3 + \cdots + x_1^{n-1} a_{n-1} \\ y_2 &= a_0 + x_2 a_1 + x_2^2 a_2 + x_2^3 a_3 + \cdots + x_2^{n-1} a_{n-1} \\ \vdots &\quad \ddots \\ y_n &= a_0 - x_n a_1 - x_n^2 a_2 + \cdots + x_n^{n-1} a_{n-1} \end{aligned}$$

One way to get back the various $a_i$'s is to solve this linear system of equations, perhaps by inverting the matrix of the $x$'s and applying it to the vector of the $y$'s. This works for any values of $y$'s and $x$'s and will find the coefficients of the polynomial, provided that the equations are consistent. This requires calculating a matrix inverse, which is not terribly efficient (say, $O(n^3)$). The DFT bypasses this matrix inversion only by virtue of the fact that it does not work on an arbitrary set of points.

## The $n$-th roots of unity

Some finite sets of numbers are more structured than other sets. For example, the set $\{0, 1, -1\}$ has the property that the product of any two elements is also in the set. On the other hand, the set $\{0, 2, 4, 6\}$ does not have this property. A set of elements is a *group* if it meets two properties:

- if $a$ and $b$ are in the set, then the product $ab$ is also in the set
- for every $a$ in the set, there is an element $b$ such that $ab = 1$

The set of $n$-th roots of unity, or $\{\omega_n^j = e^{\frac{i2\pi j}{n}} : 1 \leq j \leq n\}$, has particularly useful properties in the context of transforming polynomials between coefficient and point representations. The halving lemma states that, if $n$ is even and larger than 0, then the squares of the $n$ complex $n$-th root of unity are the $n/2$ complex $n/2$-th roots of unity.[2] Furthermore, if we use $\omega_n$ to denote the first $n$-th root of unity, then $\omega_n^2$ is the second $n$-th root of unity, and $\omega_n^2$ is easy to calculate.

## The FFT Algorithm

The FFT is an algorithm that can only evaluate a polynomial on the group of the $n$-th roots of unity. The FFT cannot be used to evaluate a polynomial at any other points, but the fact that the roots of unity are so structured allows it to skip several multiplications, leading to an $O(n \log n)$ algorithm to evaluate a polynomial at $O(n)$ points. This is useful in practice because the multiplication of two polynomials does not rely on any particular set of points, but it relies on the fact that there are $n$ points in that set.

Consider expressing a polynomial $A(x)$ as $A(x) = A'(x^2) + xA''(x^2)$, where $A'(x)$ is a polynomial consisting of all the even coefficients of $A$ and $A''$ is the polynomial consisting of all the odd coefficients of $A$. For example, if $A(x) = 5x^6 + 3x^5 - 2x^4 + x^3 - 10x^2 + 7x + 1$, then we can write the coefficients of $A$ as $[5, 3, -2, 1, -10, 7, 1]$ and take the evens $(5, -2, -10, 1)$ and write it as $A'(x) = 5x^3 - 2x^2 - 10x + 1$ and the odds as $A''(x) = 3x^2 + x + 7$. Then $A'(x^2) + xA''(x^2) =$

[1] In this particular context, the multiplication and addition are of small (32 bit) numbers, rather than huge numbers.

[2] The proof of this lemma can be found in Cormen, page 832, but you can rationalize it geometrically by considering where on the unit circle the $n$-th roots lie, and where their squares lie.

$(5x^6 - 2x^4 - 10x^2 + 1) + x(3x^4 + x^2 + 7)$ which gives us back $A(x)$. The halving lemma should give a hint as to why this method of dividing the problem makes the conquering step somewhat easier. The details of the algorithm are in the `RECURSIVE-FFT` code in the text; there, $\omega$ is a "pointer to the current root of unity."

Since the FFT can evaluate a polynomial at $2^k$ points, it should be thought of only as a machine that converts a polynomial from its coefficient representation to the point form of the polynomial in $O(n \log n)$ time. Clearly one will often need to convert the points representation back to the coefficient representation, which we can do with the procedure described above. Here again the $n$-th roots of unity have especially useful properties. Returning to the matrix approach, the FFT has given to us a matrix like

$$\begin{aligned} y_1 &= a_0 + \omega_n a_1 + \omega_n^2 a_2 + \cdots + \omega_n^{n-1} a_{n-1} \\ y_2 &= a_0 + \omega_n^2 a_1 + (\omega_n^2)^2 a_2 + \cdots + (\omega_n^{n-1})^2 a_{n-1} \\ \vdots &\quad \ddots \\ y_n &= a_0 + \omega_n^{n-1} a_1 + (\omega_n^{n-1})^2 a_2 + \cdots (\omega_n^{n-1})^{n-1} a_{n-1} \end{aligned}$$

This matrix equation is really $y = Va$, where $a$ is an $(n \times 1)$ column vector of coefficients, $y$ is an $(n \times 1)$ column vector of values, and $V$ is an $n \times n$ matrix of the roots and their powers. The $(j,k)$'th element of $V$ is $\omega_n^{jk}$, and the inverse of this matrix has a very simple form because of the unique structure of the $n$-th roots of units. The form of the $(j,k)$'th element in the inverse matrix $V^{-1}$ is $1/n\omega_n^{-kj}$, which is proved in theorem 30.7 in Cormen.

By avoiding the explicit computation of $V^{-1}$ we have avoided an $O(n^3)$ step, but the formula $V^{-1}y$ still requires a matrix product at cost $O(n^2)$. Keeping in mind that we know $y$, $V$, and can get $V^{-1}$,

$$\begin{aligned} y &= Va. \\ V^{-1}y &= V^{-1}(Va) \\ &= (V^{-1}V)a \\ &= a. \end{aligned}$$

So

$$a_j = \sum_{k=0}^{n-1} y_k \frac{1}{n} \omega_n^{-kj} = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj}$$

Writing out the sum, we have

$$a_j = \frac{1}{n} \left( y_0 + y_1 \omega_n^{-j} + y_2 (\omega_n^{-j})^2 + \cdots y_{n-1} (\omega_n^{-j})^{n-1} \right)$$

In other words, we now have a polynomial $Y(x)$, such that $a_j = Y(x_j)$: the coefficients of our original polynomial are actually the point values of the polynomial whose coefficients are the point values of the original polynomial. This is surprising, but so far it is not particularly helpful. In order to get the coefficients back we still have to evaluate the polynomial $Y$ at $n$ points, which would ordinarily take $O(n^2)$ time, but notice that the points that $Y$ is evaluated at are the complex roots of unity, but in reverse order. Thus, the FFT can also be applied to solve this problem if we normalize by $1/n$ and change the order in which you apply the roots.

Thus, converting from coefficients to points, then back to coefficients, requires $O(n \log n)$ time, with the caveat that the points are not arbitrary.

# Applications

This would seem to be a somewhat theoretical problem—what use is there to converting between the point representation and the coefficient representation of a polynomial? Already we have seen that multiplying polynomials is easier in point representation than in coefficient representation, and it is also more efficient (when you count up the operations). Other applications spring from the flexibility of the polynomial representation: many objects can be represented as polynomials, and multiplication of those objects often has intrinsic meaning. A few examples follow.

## Multiplication of polynomials

- Take polynomials $a$ and $b$, both of degree-bound $n$
- $\text{points}_a = \text{FFT}(a, 2n)$
- $\text{points}_b = \text{FFT}(b, 2n)$
- $\text{points}_c[i] = \text{points}_a[i]\text{points}_b[i]$
- $c = \text{FFT}^{-1}(\text{points}_c, 2n)$

### Multiplication of large numbers

Consider two $n$-digit numbers. Each number can be thought of as a polynomial evaluated at $x = 10$. Further, the product of the two numbers can be thought of as the product of the polynomials evaluated at $x = 10$. Thus, a recipe for multiplying large numbers is to apply the above polynomial multiplication (in $O(n \log n)$ operations) and then the remaining polynomial coefficients are the digits of the product of the two large numbers.

### Convolution of sequences

Any two ordered sequences of numbers can be convolved (where "convolution" is a well-defined mathematical term). This is often a very useful operation, for example, when dealing with probability distributions.

## Caveats

The operations described above are all reasonable for relatively small values of $n$. Note that the algorithms frequently compute sums of many roots of unity. One problem with the $n$-th roots of unity is that as $n$ gets larger, the spacing of the roots around the unit circle gets smaller, to the point where differences between successive $n$-th roots elude the precision of a physical computer. In this sense the FFT can be considered unstable, introducing gradually larger numerical errors as $n$ increases. There are some techniques to ameliorate this problem in certain contexts (e.g., when convolving certain sequences, only a small portion of that sequence really matters), but it is necessary to keep in mind the relative error of any numerical inaccuracies. This error can sometimes be quantified, albeit very slowly, by using arbitrary precision arithmetic and a non-FFT algorithm that does not suffer from round-off error.

The FFT algorithm provided in the text is fast, but why is it a Fourier Transform? The Fourier Transform is a mathematical operation on continuous functions that converts an arbitrary (nonperiodic) function into a projection onto periodic basis functions, namely, the space spanned by sines and cosines of varying frequencies. The connection between sequences with a discrete number of elements and arbitrary continuous functions is a complex one.

## Sources

This essay was compiled from notes gathered from Professor Impagliazzo's Algorithms class (CSE 202) from 2001 and 2002, from the textbook *Introduction to Algorithms* by Cormen, *et al*, and from *Contemporary Abstract Algebra* by Galian.