1 FFT and DFT

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 as well, effectively doing $O(N^2)$ operations. This improvement in efficiency is achieved through realizing properties of the N^{th} roots of unity such that redundant calculations would be eliminated thereby giving us the speed-up we desire. In this section we will first show the significance of Fourier analysis on the N^{th} roots of unity with an application on polynomial multiplication and then justify the theoretical basis of DFT.

1.2 Polynomials

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 next subsection we will introduce the FFT which will complete our search for efficiency. We first start by going over the operations we will cover in this subsection on any degree n polynomial of the general form $P(x) = \sum_{i=0}^{n} p_i x^i$ for scalars $(p_i)_{i=0}^{n}$.

- **Evaluation:** Given a polynomial $P(x)$ we can evaluate it at point x by feeding it the value x after which it will produce a number y which we desired.
- **Addition:** Given two polynomials $A(x) = \sum_{i=0}^{n} a_i x^i$ and $B(x) = \sum_{i=0}^{n} b_i x^i$ we can add them together to obtain a third polynomial $C(x) = \sum_{i=0}^{n} (a_i + b_i) x^i = \sum_{i=0}^{n} c_i x^i$.
- **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 $(c_0, c_1, ..., c_{2n})$ which is exactly the convolution of $(a_0, ..., a_n)$ and $(b_0, ..., b_n)$. 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 a polynomial 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 polynomial, we can write it as a vector in a vector space with the basis $\{1, x, x^2, ..., x^n\}$. 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 $P(x_i) = y_i)$. Say $P(x)$ has degree n, then we require exactly $n + 1$ distinct samples to be able to uniquely determine a $P(x)$ that is coherent with our $n + 1$ 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 of both worlds?" The answer is no, sadly. However, we may build a bridge using FFT connecting the best parts of these representations which will allow for a close-to-ideal solution.

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

Starting 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 $n-1$ multiplications, finally adding them together takes at most $n-1$ additions, so we would end up doing $3n-3$ 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 given only the roots of two polynomials is extremely difficult, and conversion to other forms would be costly as well, 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 each polynomial 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 roots together, which takes $O(n)$ concatenations. With samples it takes only $O(n)$ operations as well, since multiplying the second entries of two samples at the same point in the domain accomplishes polynomial multiplication.

To summarize, we have the following chart that lists the computational complexity we have covered 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 rather than accepting the fate of a 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 $O(n \log_2 n)$ complexity, as we will show in the next subsection.

1.3 A Faster Polynomial Multiplication