The Fast Fourier Transform

Sreenath Are

May 9, 2013

"fft does not stand for fast and furious turtles"
– *Remy Lee*
"WHAT THE HECK you can code an FFT is that even possible woah"
– *Alex Chen*
"i dont know why its viewed as too hard its so simple"
– *Nick Haliday*
"stop quoting me"
– *Nick Haliday*

1 Introduction

The *discrete Fourier transform*, or DFT, $\mathbf{X} = \mathcal{F}\{\mathbf{x}\}$ of a sequence of N numbers $x_0, x_1, x_2, \cdots, x_{N-1}$ is defined as

$$X_k = \sum_{n=0}^{N-1} x_n \cdot \omega^{kn}$$

where ω is a principal Nth root of unity, which means that $\omega^{kN} = 1$ if and only if k is an integer. The inverse DFT can be used to reconstruct the original sequence from the transformed sequence.

$$x_k = \mathcal{F}^{-1}\{\mathbf{X}\}_k = \frac{1}{N} \sum_{n=0}^{N-1} X_n \cdot \omega^{-kn}$$

This is most commonly done over the complex numbers, with $\omega = e^{-2\pi i/N}$. In this form, the transform converts the input data into the coefficients of a series of complex sinusoidal functions.

Example

Consider the sequence $\mathbf{x} = \{2, 0, 1, 0\}$. Using $\omega = e^{-2\pi i/4} = -i$, we get

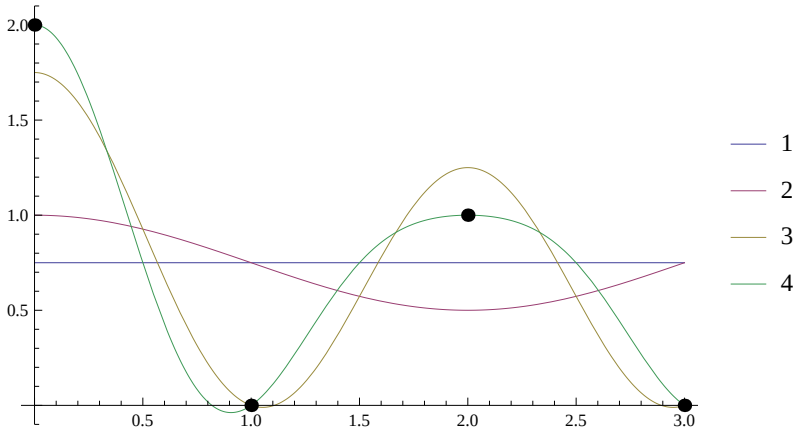
$$\begin{aligned}
X_0 &= 2 \cdot (-i)^0 + 0 \cdot (-i)^0 + 1 \cdot (-i)^0 + 0 \cdot (-i)^0 = 3 \\
X_1 &= 2 \cdot (-i)^0 + 0 \cdot (-i)^1 + 1 \cdot (-i)^2 + 0 \cdot (-i)^3 = 1 \\
X_2 &= 2 \cdot (-i)^0 + 0 \cdot (-i)^2 + 1 \cdot (-i)^4 + 0 \cdot (-i)^6 = 3 \\
X_3 &= 2 \cdot (-i)^0 + 0 \cdot (-i)^3 + 1 \cdot (-i)^6 + 0 \cdot (-i)^9 = 1
\end{aligned}$$

Thus $\mathcal{F}\{2, 0, 1, 0\} = \{3, 1, 3, 1\}$. This takes $O(N^2)$ operations to compute, which can be prohibitively large in some applications. The term *fast Fourier transform* refers to any algorithm that computes the DFT in $O(N \log N)$ time.

2 Applications

Compression

By omitting some parts of the DFT of the sequence, we can decrease the size needed to store it while still being able to recover values close to every point in the original series. For example, the plot below shows the inverse DFTs of $\{3,0,0,0\}$, $\{3,1,0,0\}$, $\{3,1,3,0\}$ and $\{3,1,3,1\}$:



When compressing audio, this allows us to keep more information about the most noticable frequencies, while discarding a lot of unnecessary information about less audible ones.

Convolution

The convolution $\mathbf{a} * \mathbf{b}$ of two sequences $\mathbf{a}$ and $\mathbf{b}$ is defined as

$$(\mathbf{a} * \mathbf{b})_k = \sum_{i=0}^{k} a_i b_{k-i}$$

For example, the convolution of $\{1,1,0,0\}$ with $\{1,1,1,0\}$ is $\{1,2,2,1\}$. The *convolution theorem* (the proof is left as an exercise to the reader) states that

$$\mathcal{F}\{\mathbf{a} * \mathbf{b}\}_k = \mathcal{F}\{\mathbf{a}\}_k \cdot \mathcal{F}\{\mathbf{b}\}_k$$

The convolution takes $O(N^2)$ operations to compute from its definition, but we can use the convolution theorem and a fast Fourier transform to compute it in $O(N \log N)$ operations.

Polynomial Multiplication

The convolution of the series of coefficients of two polyonomials is the series of coefficients of their product; using the example given above, $(1+x)(1+x+x^2) = 1+2x+2x^2+x^3$. This allows us to compute polynomial products quickly using the FFT by padding the coefficient lists with enough 0s that the lists are the same length and large enough to contain the product, taking the DFT of both lists, computing the pointwise product of the two lists, and taking the inverse DFT.

Integer Multiplication

Multiplying integers is very similar to multiplying polynomials. For example, $1243 \cdot 412$ can be thought of as $(1x^3 + 2x^2 + 4x + 3) \cdot (4x^2 + 1x + 2)$ evaluated at $x = 10$. We can multiply these polynomials using the FFT, giving us the polynomial $4x^5 + 9x^4 + 20x^3 + 20x^2 + 11x + 6$. Evaluating this polynomial gives us our

final answer of 512116. This algorithm has an overall runtime of $O(n \log n \log\log n)$ for multiplying two n digit integers. Note that the "normal" method of multiplication has a runtime of $O(n^2)$. This algorithm, or variants of it, are used in Python, Java's BigInteger, and C++'s GMP for numbers larger than about 10000 digits.

3 The Cooley-Tukey Algorithm

Given a DFT of even length, we can express it in terms of two smaller DFTs:

$$\begin{aligned} X_k = \sum_{n=0}^{N-1} x_n \cdot \omega^{kn} &= \sum_{n=0}^{N/2-1} x_{2n} \cdot \omega^{2kn} + \sum_{n=0}^{N/2-1} x_{2n+1} \cdot \omega^{k(2n+1)} \\ &= \sum_{n=0}^{N/2-1} x_{2n} \cdot \omega^{2kn} + \omega^k \sum_{n=0}^{N/2-1} x_{2n+1} \cdot \omega^{2kn} \\ &= E_k + \omega^k O_k \end{aligned}$$

where $\mathbf{E}$ is the DFT of the even-indexed elements of $\mathbf{X}$ and $\mathbf{O}$ is the DFT of the odd-indexed elements of $\mathbf{X}$. We can use this fact to recursively compute a DFT of size $N = 2^k$ in $O(N \log N)$ time. This can be generalized to split any DFT of composite size ab into a DFTs of size b and b DFTs of size a.

4 Number Theoretic Transform

It is often preferable to avoid complex arithmetic when multiplying integers or polynomials with integer coefficients, and this is done by operating instead on the integers mod p, where p is some prime. A primitive nth root of unity exists in this mod if n divides $p-1$. For example, 8 is a primitive 4th root of unity mod 13, as the powers of 8 form the sequence $\{8, 12, 5, 1\}$ when taken mod 13. As an example, the fourier transform of $\mathbf{x} = \{2, 1, 1, 0\}$ with $\omega = 8 \bmod 13$ is

$$\begin{aligned} X_0 &= 2 \cdot 8^0 + 1 \cdot 8^0 + 1 \cdot 8^0 + 0 \cdot 8^0 = 4 \\ X_1 &= 2 \cdot 8^0 + 1 \cdot 8^1 + 1 \cdot 8^2 + 0 \cdot 8^3 = 9 \\ X_2 &= 2 \cdot 8^0 + 1 \cdot 8^2 + 1 \cdot 8^4 + 0 \cdot 8^6 = 2 \\ X_3 &= 2 \cdot 8^0 + 1 \cdot 8^3 + 1 \cdot 8^6 + 0 \cdot 8^9 = 6 \\ \mathbf{X} &= \{4, 9, 2, 6\} \end{aligned}$$

Although the result no longer represents a sinusoidal interpolation of the data, it retains the other nice properties of the DFT without the computational difficulty of dealing with complex numbers to the required accuracy.