Fast Fourier Transform

Dima Kochkov[1]

[1]Department of Physics
University of Illinois at Urbana-Champaign

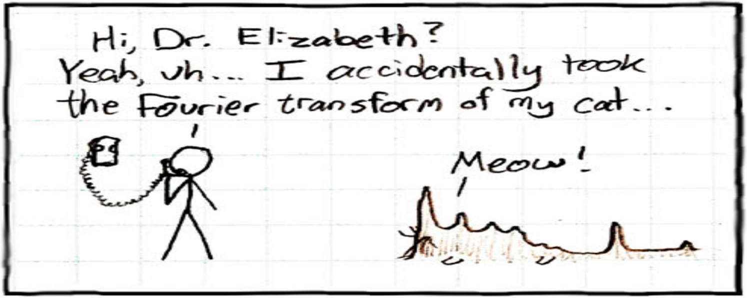
Algorithm interest meeting, 2016

Outline

Outline

Significance

Fast Fourier Transform is considered to be one of the most important algorithms of 20^{th} century. It performs Discrete Fourier Transform in $N \log_2 N$ time in comparison to the naive N^2. This algorithm is accountable for many advances and wide applicability of Fourier transformation in numerics.

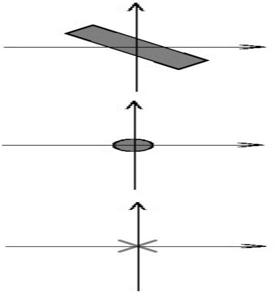


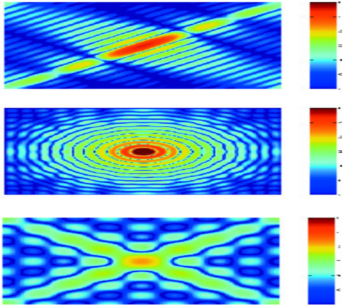
Outline

Where it is used

Fast Fourier Transform is heavily used in many fields, including, but not limited to:

- Data analysis (Fourier components)
- Data compression (Jpeg, MP3, etc)
- Partial differential equation solvers
- Big integer multiplication





Outline

In the literature the term Fast Fourier Transform stands for an algorithm that computes a Discrete Fourier Transform in $N \log_2 N$ time. There exist a variety of different implementations and generalizations. In this talk I will only concentrate on the central idea.

$$FFT(\vec{a}) = \vec{X} \quad : \quad X_j = \sum_{k=0}^{k=N-1} e^{2\pi i \frac{kj}{N}} a_k \tag{1}$$

Which is equivalent to a convolution of a given sequence with a special sequence of elements. In the following sections I will map the problem to a multiplication of polynomials and will closely follow the lecture on FFT by Erik Demaine MIT lecture on FFT

Outline

Polynomials

A polynomial ring in X over a field K is an algebraic structure that consists of polynomials of variable $z \in$ X and coefficients from the field K.

$$P_{N-1}(z) = \sum_{j=0}^{j=N-1} a_j z^j \tag{2}$$

This structure features addition, multiplication and CS-ish addition evaluation operations. In this presentation we will establish a correspondence between vectors and polynomials, as well as different operations on them.

Outline

Representations

There are several reasonable representations that we might consider:

- Coefficients a_j :
  $\tilde{P_{N-1}}(z) = \sum_{j=0}^{j=N-1} a_j z^j$
- Roots z_j :
  $\tilde{P_{N-1}}(z) = a_{N-1}(z - z_{N-1})..(z - z_1)$
- Samples $(z_j, \tilde{P_{N-1}}(z_j))$ for N different points

From now on I will fix the N to be some power of 2 and will use bottom index to indicate different polynomials.

Outline

Summation

Coefficients

$P_a(z) + P_b(z) = P_c(z)$: $c_j = a_j + bj$: add corresponding coefficients N

Roots

Impossible if arbitrary precision is desired

Samples

$P_a(z_j) + P_b(z_j) = P_c(z_j)$: add corresponding values of P N

So far : coefficients - ok, Samples - ok, Roots - Boo!

Outline

Multiplication

Coefficients

$P_a(z) * P_b(z) = P_c(z) : \; c_j = \sum_{k=0}^{k=N} a_k * bj - k \quad N^2$

Roots

Concatenate the list of roots, update the leading coefficient N

Samples

$P_a(z_j) * P_b(z_j) = P_c(z_j)$: multiply corresponding values of P N

So far : coefficients - nah, Samples - alpha?, Roots - still boo!

Outline

Evaluation

Coefficients

$P_c(z)$: add contributions from different powers, N

Roots

Evaluate multiplication of $\prod_j (z - z_j)$ N

Samples

Have to solve for coefficients system of N linear equations N^2

No representation is perfect :-(

Outline

Evaluation at N points

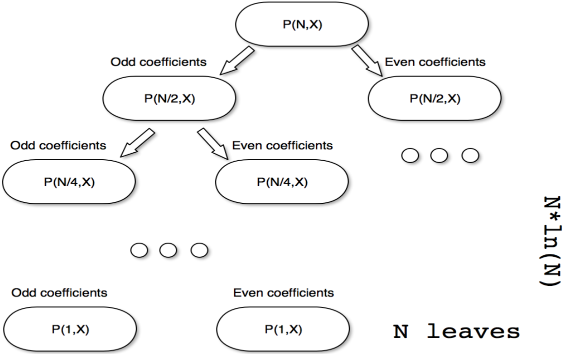
The algorithm gets its boost by jumping back and forth between representations. We will concentrate on the transformation from the **coefficient** representation to the **samples** representation, since the inverse is similar.
To do that we need to evaluate P at N different points. $N * N$, can we do better?

Outline

Split even and odd powers

We can rewrite evaluation $P(z)$ as a sum of 2 values that can be computed recursively:

$$P(z) = \sum_{j=0}^{j=N-1} a_j z^j = \sum_{j=0}^{j=\frac{N-1}{2}} a_{2j} z^{2j} + z \sum_{j=0}^{j=\frac{N-1}{2}} a_{2j+1} z^{2j} = P_a(z^2) + zP_b(z^2) \tag{3}$$



Magic!

Now we will use the freedom of choosing sample points to make this algorithm work! Consider the set of roots of unity.

$$[e^{i\frac{\pi}{2}}, e^{i\pi}, e^{i\frac{3\pi}{2}}, e^{2\pi}]^2 \rightarrow [-1, 1] \tag{4}$$

The number of points on which one needs to evaluate consecutive polynomials shrinks by a factor of 2 with every recursion call. That insures the $N \log_2(N)$ performance.

Algorithm

RECURSIVE-FFT(a)

INPUT: $A(x)$ in the coefficient vector $a = (a_0, a_1, \cdots, a_{n-1})$

OUTPUT: the DFT of a

1. $n \leftarrow length[a]$
2. **if** $(n = 1)$
3. **then return** a ∇*base case*
4. $w_n \leftarrow e^{2\pi i/n}$
5. $w \leftarrow 1$
6. $a' \leftarrow (a_0, a_2, \cdots, a_{n-2})$
7. $a'' \leftarrow (a_1, a_3, \cdots, a_{n-1})$
8. $y' \leftarrow$ **RECURSIVE-FFT**(a')
9. $y'' \leftarrow$ **RECURSIVE-FFT**(a'')
10. **for**$k \leftarrow 0$ **to** $n/2 - 1$
11. $y_k \leftarrow y'_k + w y''_k$
12. $y_{k+(n/2)} \leftarrow y'_k - w y''_k$
13. $w \leftarrow w w_n$
14. **return** y

courses, books

MIT video lecture on FFT by Erik Demaine

lecture on FFT, complexity analysis