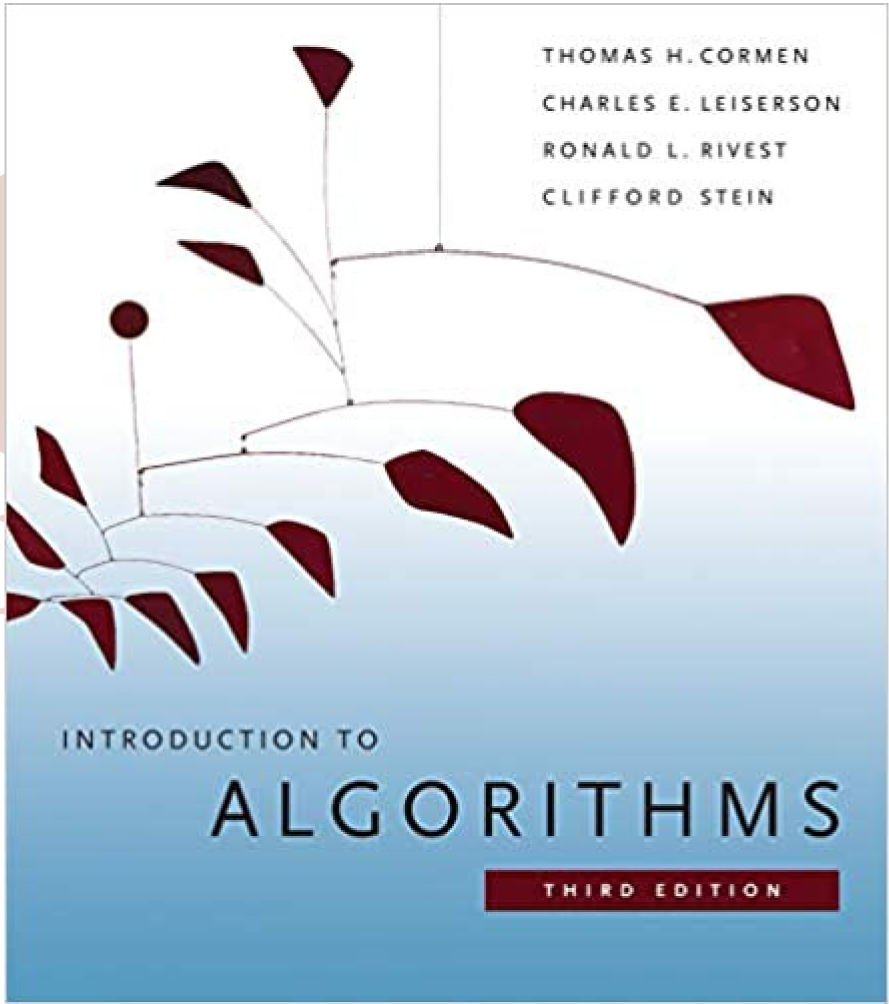
CS 07540 Advanced Design and Analysis of Algorithms

Week 13

- Fourier Transforms
  - Discrete Fourier Transform
  - Inverse Discrete Fourier Transform
  - Fast Fourier Transform



THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN
INTRODUCTION TO
ALGORITHMS
THIRD EDITION

Fourier Analysis

Fourier transforms and related methods are used in virtually all STEM areas:

- circuit design
- crystallography
- imaging

There are obvious application in

- signal processing, signal design, and signal communications

Fourier analysis Wikipedia entry:

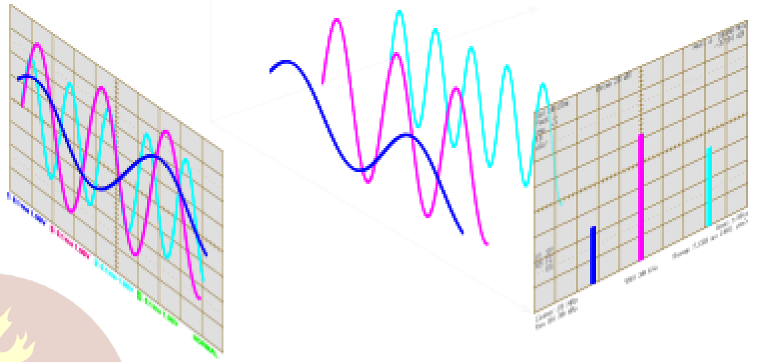
Fourier analysis is the study of the way general functions may be represented or approximated by sums of simpler trigonometric functions. Fourier analysis grew from the study of Fourier series, and is named after Joseph Fourier, who showed that representing a function as a sum of trigonometric functions greatly simplifies the study of heat transfer.

From Fourier Series to Fourier Transforms

Fourier series represent functions as a (possibly infinite) sum of periodic functions (such as sine/cosine functions) under certain conditions. If we allow the period to approach infinity, we are essentially talking about arbitrary functions and their representation as sums of periodic functions. In that case, a discrete set of frequencies in the periodic case becomes a continuum of frequencies in the nonperiodic case called the ***spectrum***, and with it comes the most important principle of Fourier Transforms:

Every signal has a spectrum and is determined by its spectrum. You can analyze the signal either in the time (or spatial) domain or in the frequency domain. The translation is the process of a ***Fourier Transform***.

Fourier Transforms



Signals (sums of sinusoidal waves) can be analyzed in the time domain (as a function of time) or in the frequency domain (its spectrum). Every signal *has* a spectrum and is determined *by* its spectrum. A ***Fourier Transform*** (FT) is transformation of a function from the time domain into the frequency domain. Fourier transform methods are used in virtually all areas of engineering, mathematics, and science – in particular, in electrical engineering (signal design and processing). An example of a transform is the discrete cosine transform (DCT) which is utilized by mp3 and aac audio formats as well as jpeg picture and mpeg movie formats. Note that all these formats are lossy as they are designed to compress data.

The data given in the previous examples (video, audio) is sampled data and not continuous. Therefore in practical applications, the discrete Fourier transform (DFT) and its special form fast Fourier transform (FFT) is of most interest, as well as their inverse operators.

- DFT
    - Sampled signal to spectrum. Receiving.
- IDFT
    - Spectrum to sampled signal. Sending.
    - Can be used to design signals with limited peak-to-average power ratio.

IDFT Application Example

Orthogonal frequency division multiplexing (OFDM) is a modulation technique that is used in many applications. It exhibits robustness against various kinds of interference and also enables multiple access. OFDM uses several signals at equally spaced frequencies starting from a base frequency. Let

- f be the base frequency (and $\omega = 2\pi f$ its angular frequency)
- Δf be the bandwidth for each subchannel
- $f_i = f + i \cdot \Delta f$ be the frequencies of the subchannels

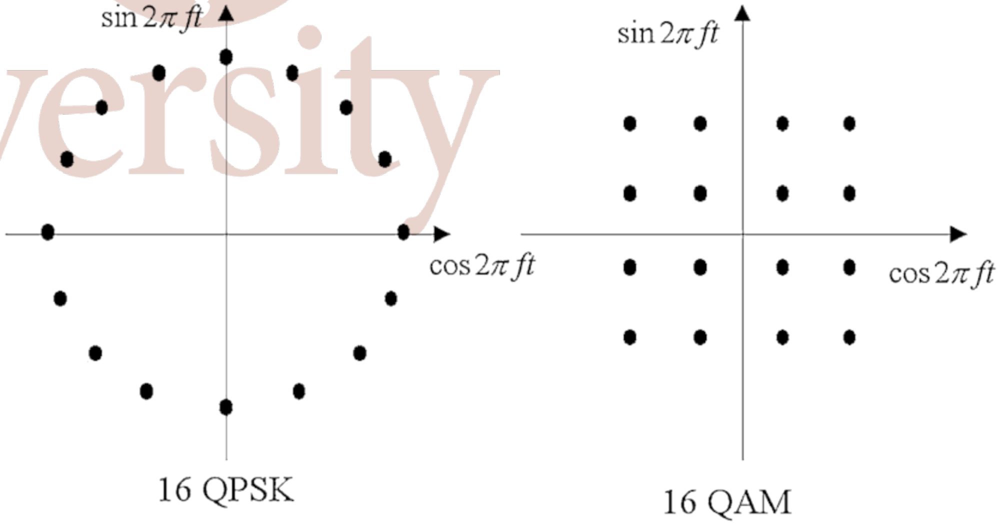
A signal is the sum of the individual (harmonic) signals on each subchannel. Each such signal is uniquely described as a cosine by its amplitude A_k, its frequency f_k and its phase shift φ_k.

We can write the signal in the time domain as a real function as well as the real part of a complex function (here $j = \sqrt{-1}$).

$$
\begin{aligned}
S(t) &= \sum_k A_k \cos(2\pi f_k + \varphi_k) \\
&= \Re\left(\sum_k e^{\ln A_k + j(\omega + 2\pi k \Delta f + \varphi_k)} \right) \\
&= \Re\left(e^{j\omega} \sum_k e^{\ln A_k + j(2\pi k \Delta f + \varphi_k)} \right)
\end{aligned}
$$

A widely used modulation technique to transmit data is Phase-shift keying (PSK). Data is transported by changing (modulating) the phase of a constant frequency. In signal design we may be interested in using only phase shifts which create a limited power envelop. Standard phase-shift keying does not modify amplitudes: $A_k = 1$. The phase shifts φ_k are normalized to represent shifts around the unit circle. Standard shifts are

- binary
  - $\varphi \in \{k \cdot 2\pi j/2 \mid k \in \{0,1\}\}$
- quaternary
  - $\varphi \in \{k \cdot 2\pi j/4 \mid k \in \{0,1,2,3\}\}$



$$|S(t)|^2 = \left(e^{j\omega}\sum_k e^{j(2\pi k\Delta f+\varphi_k)}\right)\left(\overline{e^{j\omega}\sum_k e^{j(2\pi k\Delta f+\varphi_k)}}\right)$$
$$= \sum_{k_1}\sum_{k_2} e^{j\left(2\pi k_1\Delta f+\varphi_{k_1}\right)-j\left(2\pi k_2\Delta f+\varphi_{k_2}\right)}$$
$$= \sum_{u=k_1-k_2} e^{j(2\pi u\Delta f)} \sum_{k=k_1} e^{j(\varphi_k-\varphi_{k-u})}$$

The inner sum is called autocorrelation of the signal and does not depend on time or frequencies. For a given signal it can be calculated in $\Theta(n^2)$.

Each term in the outer sum does not depend on the actual signal, so we can precalculate it for any signal we may want to investigate. The outer sum itself then turns into an inner product (dot-product). If this inner product can be parallelized, we can get the power envelop in $\Theta(n^2 + m)$ where m is the number of samples. The outer sum can be thought of as test-functions applied to the inner sum (the autocorrelation). This is actually the Fourier Transform applied to the autocorrelation function.

The Fourier Transform of the autocorrelation function of a signal (corresponding to a phase-shift keying data sequence in our case) **is the power spectrum of the signal**.

Fast Fourier Transform

A vector $\vec{a}$ of n values can uniquely represent a polynomial of degree $(n-1)$. Recall that a polynomial in the variable x over an algebraic field $\mathbb{F}$ represents a function $A(x) = \sum_{i=0}^{n-1} a_i x^i$ as a formal sum. Conversely, given a polynomial of degree $(n-1)$, we only need n distinct data points to uniquely identify its n coefficients (using a system of linear equations). Finding the coefficients for a polynomial given the values at n points is called ***interpolation***.

Multiplying (correlating) two polynomials $A(x)$ and $B(x)$ of degree $(n_A - 1)$ and $(n_B - 1)$ involves $\Theta(n_A n_B)$ multiplications. The resulting polynomial will have degree $n_A + n_B - 2$, so $(n_A + n_B - 1)$ data points would suffice to identify its coefficients… that is much less than $\Theta(n_A n_B)$ operations suggest. Maybe Fourier transform ideas can help…

Multiplication is considerably more time-consuming than addition. The cost of adding two n-bit numbers is $\Theta(n)$ while straightforward multiplication of two such numbers is $\Theta(n^2)$. If (and that is a big if) taking logarithms and inverse logarithms was “cheap”, then we could replace multiplication by a sequence of

1. Logarithms
2. Addition
3. Inverse logarithm

using $\log(nm) = \log n + \log m$:

We would get

$$nm = \text{inverse}\log(\log(nm)) = \text{inverse}\log(\log n + \log m)$$

The straightforward method for multiplying polynomials takes $\Theta(n^2)$ time if we use the coefficient representation of a polynomial since we essentially gather all contributions to a particular exponent.

However, we can also represent a polynomial in point-value form which lets us recover a polynomial using Lagrange polynomials interpolation. If we evaluate $A(x)$ and $B(x)$ at the same points $x_0, x_1, \ldots, x_{n-1}$ then we only need n additions to evaluate $C(x) = A(x) + B(x)$ at these points. That is no faster than using the coefficient representation. Now suppose we use point-value representation with $2n$ points (essentially oversampling the original polynomials to the Nyquist rate of the product). Then we can evaluate $C(x) = A(x)B(x)$ at these $2n$ points to get its point-value representation in $\Theta(n)$ time! That is better than $\Theta(n^2)$. But is it worth the transformation to a different representation?

$$\begin{tikzcd}[column sep=huge, row sep=huge]
\boxed{\begin{array}{l} A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} \\ B(x) = b_0 + b_1 x + \cdots + b_{n-1} x^{n-1} \end{array}} \arrow[r, dashed, "\text{ordinary multiplication}", "\text{Time } \Theta(n^2)"'] \arrow[d, "\text{evaluation}"', "\text{Time?}"] & \boxed{C(x) = c_0 + c_1 x + \cdots + c_{2n-1} x^{2n-1}} \\
\boxed{\begin{array}{c} A(x_0), B(x_0) \\ A(x_1), B(x_1) \\ \vdots \\ A(x_{2n-1}), B(x_{2n-1}) \end{array}} \arrow[r, "\text{pointwise multiplication}", "\text{Time } \Theta(n)"'] & \boxed{\begin{array}{c} C(x_0) \\ C(x_1) \\ \vdots \\ C(x_{2n-1}) \end{array}} \arrow[u, "\text{interpolation}", "\text{Time?}"']
\end{tikzcd}$$

Coefficient Form to Point-Value Conversion

For us to use the linear-time multiplication method for polynomials in point-value form to expedite polynomial multiplication in coefficient form, we need fast conversions.

- Coefficient form → Point-value form
  - Evaluation
- Point-value form → Coefficient form
  - Interpolation

Choosing the evaluation points carefully will be key to converting between representations in only $\Theta(n \log n)$. These special points will be complex roots of unity.

Polynomial Multiplication with FFT

1. *Double Degree*: Create coefficient representations of $A(x)$ and $B(x)$ as degree $2n-1$ polynomials (padding higher order powers with zeros) evaluated at the $(2n)$th roots of unity.
2. *Evaluation*: Use the FFT to compute point-value representations of $A(x)$ and $B(x)$ at roots of unity of order $2n$. The result is two length $2n$ vectors.
3. *Pointwise Multiplication*: Compute a point-value representation for the product $C(x) = A(x)B(x)$. The result is a length $2n$ vector containing the values of $C(x)$ at the $(2n)$th roots of unity.
4. *Interpolation*: Create the coefficient representation of the polynomial $C(x)$ using the inverse DFT.

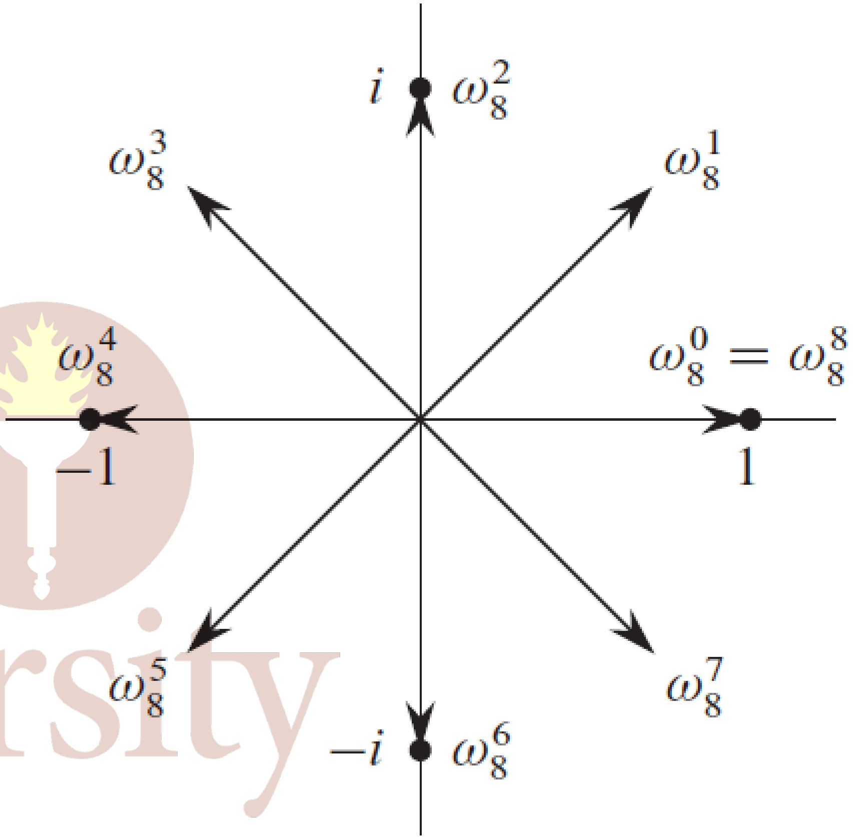
Steps (1) and (3) take time $\Theta(n)$, and steps (2) and (4) take time $\Theta(n \log n)$.

DFT and FFT

A ***complex n-th root of unity*** is a complex number ω such that

$$\omega^n = 1$$

Each n-th root of unity is of form $e^{2\pi jk/n}$ for some $k = 0,1,\ldots,n-1$. Roots of unity add up to 0 (unless $n = 1$).



Let $A(x) = \sum_{i=0}^{n-1} a_i x^i$. Then the vector $(y_0, \ldots, y_{n-1})$ with entries

$$y_k = A(\omega_n^k) = \sum_{i=0}^{n-1} a_i\, \omega_n^{ki}$$

is called ***discrete Fourier transform*** of coefficient vector $(a_0, \ldots, a_{n-1})$.

$$y = \mathrm{DFT}_n(\vec{a})$$

For the following considerations, we assume that n is a power of 2 so we can "half" polynomials into "even" and "odd" polynomials recursively ($n = 2^{\log n}$).

$$A(x) = \sum_{i=0}^{\frac{n}{2}-1} a_{2i} x^{2i} + x \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} x^{2i}$$

Hence $A(x) = E(x^2) + x \cdot O(x^2)$ and we almost exclusively work with $n/2$-th root of unities! Each of the problems $E(x^2)$ and $O(x^2)$ is of the same form as the original $A(x)$. We have divided an n-element DFT_n computation into two $n/2$-element $\mathrm{DFT}_{n/2}$ computations. After $\log n$ subdivisions, we will have degree 1 polynomials. This is the basis for the recursive FFT algorithm of complexity $\Theta(n \log n)$.

```
RECURSIVE-FFT($a$)
$n = a.length$                         // $n$ is a power of 2
if $n == 1$
    return $a$
$\omega_n = e^{2\pi i/n}$
$\omega = 1$
$a^{[0]} = (a_0, a_2, \ldots, a_{n-2})$
$a^{[1]} = (a_1, a_3, \ldots, a_{n-1})$
$y^{[0]} =$ RECURSIVE-FFT($a^{[0]}$)
$y^{[1]} =$ RECURSIVE-FFT($a^{[1]}$)
for $k = 0$ to $n/2 - 1$
    $y_k = y_k^{[0]} + \omega\, y_k^{[1]}$
    $y_{k+(n/2)} = y_k^{[0]} - \omega\, y_k^{[1]}$
    $\omega = \omega\, \omega_n$
return $y$                             // $y$ is assumed to be a column vector
```

```
import numpy as np

if __name__ == '__main__':
    A=[9,-10,7,6,0,0,0,0]
    B=[-5,4,0,-2,0,0,0,0]
    dftA = np.fft.rfft(A)
    dftB = np.fft.rfft(B)
    C = np.fft.irfft(dftA*dftB)
    print(C)
```

```
[-45.  86. -75. -20.  44. -14. -12.   0.]
```

Here **`rfft/irfft`** tells NumPy that we use real-valued functions.

$$\begin{aligned} A(x) &= 6x^3 + 7x^2 - 10x + 9 \\ B(x) &= -2x^3 + 4x - 5 \\ A(x)B(x) &= -12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45 \end{aligned}$$

```python
import numpy as np

if __name__ == '__main__':
    A=[9,-10,7,6,0,0,0,0]
    B=[-5,4,0,-2,0,0,0,0]
    dftA = np.fft.fft(A)
    dftB = np.fft.fft(B)
    C = np.fft.ifft(dftA*dftB)
    print(C)
```

```
[-45.+0.j  86.+0.j -75.+0.j -20.+0.j  44.+0.j -14.+0.j -12.+0.j   0.+0.j]
```

To complete the whole algorithm, we need to interpolate the product from the point-value representation at the roots of unity. This is done with a Vandermonde matrix multiplication where $v_{ij} = \omega_n^{ji}$.

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

In the special case of roots of unity, the inverse matrix V^{-1} is formed with entries $w_{ij} = \frac{1}{n}\,\omega^{-ji}$.

Then

$$a_i = \frac{1}{n}\sum_{k=0}^{n-1} y_k\, \omega_n^{-ki}$$

We note the symmetry in inverse FFT (above) and FFT (below):

$$y_k = \sum_{i=0}^{n-1} a_i\, \omega_n^{ki}$$

$$\begin{tikzcd}
\boxed{\begin{array}{l} A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} \\ B(x) = b_0 + b_1 x + \cdots + b_{n-1} x^{n-1} \end{array}} \arrow[rr, dashed, "\text{ordinary multiplication}", "\text{Time } \Theta(n^2)"'] \arrow[d, "\text{Time } \Theta(n \log n)", "\text{forward FFT}"'] & & \boxed{C(x) = c_0 + c_1 x + \cdots + c_{2n-1} x^{2n-1}} \\
\boxed{\begin{array}{c} A(\omega_{2n}^{0}), B(\omega_{2n}^{0}) \\ A(\omega_{2n}^{1}), B(\omega_{2n}^{1}) \\ \vdots \\ A(\omega_{2n}^{2n-1}), B(\omega_{2n}^{2n-1}) \end{array}} \arrow[rr, "\text{pointwise multiplication}", "\text{Time } \Theta(n)"'] & & \boxed{\begin{array}{c} C(\omega_{2n}^{0}) \\ C(\omega_{2n}^{1}) \\ \vdots \\ C(\omega_{2n}^{2n-1}) \end{array}} \arrow[u, "\text{inverse FFT}", "\text{Time } \Theta(n \log n)"']
\end{tikzcd}$$

FFT Summary

- Runtime complexity of polynomial multiplication with DFFT is $O(n\log(n))$.