Computational Methods
CMSC/AMSC/MAPL 460

Fourier transform

Ramani Duraiswami,
Dept. of Computer Science
Fourier Methods

• Last class
  – Introduced the Fourier basis
  – Showed why it might be useful
  – Introduced the notion of a Fourier Matrix
  – Introduced the Danielson Lanczos Lemma
  – Introduced the FFT algorithm

• This class
  – Detailed consideration of the FFT algorithm
  – Inverse FFT
  – Application to polynomial multiplication
Complex Notation

For $f$, periodic with period $p$

Fourier transform $f(t) \rightarrow F[k]$

$$F[k] = \frac{1}{p} \int_{0}^{p} f(t) e^{-2\pi i \frac{k}{p} t} \, dt$$

$$= \frac{1}{p} \int_{0}^{p} f(t) \cos(2\pi \frac{k}{p} t) \, dt$$

$$- \frac{i}{p} \int_{0}^{p} f(t) \sin(2\pi \frac{k}{p} t) \, dt$$

Inverse Fourier transform $F[k] \rightarrow f(t)$

$$f(t) = \sum_{k \in \mathbb{Z}} F[k] e^{2\pi i \frac{k}{p} t}$$
Sampling

Fourier representations work just fine with sampled data.

Simple connection to Fourier of the continuous function it came from.

Familiar example: Digital Audio
DFT

\[ \int_{0}^{p} f(t) \, e^{-2\pi i k t/p} \, dt \] \rightarrow \sum_{n=0}^{N-1} \phi[n] \, e^{-2\pi i k nh/p} \]

\[ f(t) \rightarrow \phi[n] = f(n \, h) \]
DFT and its inverse for periodic discrete data

\[ \Phi[k] = \sum_{n=0}^{N-1} \phi[n] \, e^{-2\pi ikn/p} \quad p = Nh \]

\[ = \sum_{n=0}^{N-1} \phi[n] \, e^{-2\pi ikn/N} \]

This is automatically periodic in \( k \) with period \( N \).

Inverse is like Fourier series, but with only \( p \) terms.
Discrete time Numerical Fourier Analysis

DFT is really just a matrix multiplication!

\[
F[m] = \frac{1}{N} \sum_{k=0}^{N-1} e^{-2\pi i \frac{m}{N} k} f[k]
\]

\[
\begin{pmatrix}
F[0] \\
F[1] \\
F[2] \\
\vdots \\
F[N-1]
\end{pmatrix} =
\begin{pmatrix}
\frac{1}{N} \\
\frac{1}{N} \\
\frac{1}{N} \\
\vdots \\
\frac{1}{N}
\end{pmatrix}
\begin{pmatrix}
e^{-2\pi i \frac{0}{N} k} f[0] \\
e^{-2\pi i \frac{1}{N} k} f[1] \\
e^{-2\pi i \frac{2}{N} k} f[2] \\
\vdots \\
e^{-2\pi i \frac{N-1}{N} k} f[N-1]
\end{pmatrix}
\]
Fourier Matrices

A Fourier matrix of order $n$ is defined as the following

$$F_n = \begin{bmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\
1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\
& \ddots & \ddots & \ddots & \vdots \\
1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \\
\end{bmatrix},$$

where

$$\omega_n = e^{-\frac{2\pi i}{n}},$$

is an $n$th root of unity.
i = \sqrt{-1}  

**Primitive Roots of Unity**

A number \( \omega \) is a *primitive n-th root of unity*, for \( n > 1 \), if
\[
\omega^n = 1
\]
The numbers 1, \( \omega \), \( \omega^2 \), ..., \( \omega^{n-1} \) are all distinct

- Example: The complex number \( e^{2\pi i/n} \) is a primitive n-th root of unity, where

**Check: if properties are satisfied**

1. \( \omega^1 = e^{\frac{2\pi i}{n}} \neq 1 \)
2. \( \omega^n = \left( e^{\frac{2\pi i}{n}} \right)^n = e^{2\pi i} = \cos 2\pi + i \sin 2\pi = 1 \)
3. \( S=\sum_{p=0}^{n-1} \omega^p = \omega^0 + \omega^1 + \omega^2 + \omega^3 + \ldots + \omega^{(n-1)} = 0 \)

\( n \) complex roots of unity equally spaced around the circle of unit radius centered at the origin of the complex plane.
Roots of Unity: Properties

• Property 1: Let \( \omega \) be the principal \( n^{th} \) root of unity. If \( n > 0 \), then \( \omega^{n/2} = -1 \).
  
  – Proof: \( \omega = e^{\frac{2\pi i}{n}} \Rightarrow \omega^{n/2} = e^{\frac{\pi i}{2}} = -1 \). (Euler's formula)
  
  – Reflective Property:
  
  – Corollary: \( \omega^{k+n/2} = -\omega^k \).

• Property 2: Let \( n > 0 \) be even, and let \( \omega \) and \( \nu \) be the principal \( n^{th} \) and \( (n/2)^{th} \) roots of unity. Then \( (\omega^k)^2 = \nu^k \).
  
  – Proof: \( (\omega^k)^2 = e^{(2k)\frac{2\pi i}{n}} = e^{(k)\frac{2\pi i}{(n/2)}} = \nu^k \).
  
  – Reduction Property: If \( \omega \) is a primitive \((2n)\)-th root of unity, then \( \omega^2 \) is a primitive \(n\)-th root of unity.
• L3: Let $n > 0$ be even. Then, the squares of the $n$ complex $n^{th}$ roots of unity are the $n/2$ complex $(n/2)^{th}$ roots of unity.
  
  – Proof: If we square all of the $n^{th}$ roots of unity, then each $(n/2)^{th}$ root is obtained exactly twice since:
  
  • L1 $\Rightarrow \omega^{k+n/2} = -\omega^k$
  • thus, $(\omega^{k+n/2})^2 = (\omega^k)^2$
  • L2 $\Rightarrow$ both of these $= \nu^k$
  • $\omega^{k+n/2}$ and $\omega^k$ have the same square

• **Inverse Property:** If $\omega$ is a primitive root of unity, then $\omega^{-1} = \omega^{n-1}$
  
  – Proof: $\omega \omega^{n-1} = \omega^n = 1$
Fast Fourier Transform

- Presented by Cooley and Tukey in 1965, but invented several times, including by Gauss (1809) and Danielson & Lanczos (1948)
- Danielson Lanczos lemma

\[
F_k = \sum_{j=0}^{N-1} e^{2\pi i j k/N} f_j
\]

\[
= \sum_{j=0}^{N/2-1} e^{2\pi i k(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi i k(2j+1)/N} f_{2j+1}
\]

\[
= \sum_{j=0}^{N/2-1} e^{2\pi i k j/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi i k j/(N/2)} f_{2j+1}
\]

\[
= F_k^e + W^k F_k^o
\]
• So far we have seen what happens on the right hand side
• How about the left hand side?
• When we split the sums in two we have two sets of sums with \( N/2 \) quantities for \( N \) points.
• So the complexity is \( N^2/2 + N^2/2 = N^2 \)
• So there is no improvement
• Need to reduce the sums on the right hand side as well
  – We need to reduce the number of sums computed from \( 2N \) to a lower number
  – Notice that the values corresponding to \( k \) and \( k+N/2 \) will be the same.
  – The transforms \( F_e^k \) and \( F_o^k \) are periodic in \( k \) with length \( N/2 \).
  – So we need only compute half of them!
FFT

- So DFT of order $N$ can be expressed as sum of two DFTs of order $N/2$ evaluated at $N/2$ points.
- Does this improve the complexity?
- Yes \((N/2)^2+(N/2)^2 = \frac{N^2}{2} < N^2\)
- But we are not done ….
- Can apply the lemma recursively
  \[ F_k^e = F_k^{ee} + W_k^e F_k^{eo}, \quad F_k^o = F_k^{oe} + W_k^o F_k^{oo}, \]
- Finally we have a set of one point transforms
- One point transform is identity
  \[ F_k^{eeoeoeoeoeoeoeoe} = f_n \]
FFT Algorithm

FFT \( (n, a_0, a_1, a_2, \ldots, a_{n-1}) \)

if \( n == 1 \)  // \( n \) is a power of 2
  return \( a_0 \)

\( \omega \leftarrow e^{2\pi i / n} \)

\((e_0, e_1, e_2, \ldots, e_{n/2-1}) \leftarrow \text{FFT}(n/2, a_0, a_2, a_4, \ldots, a_{n-2}) \)

\((d_0, d_1, d_2, \ldots, d_{n/2-1}) \leftarrow \text{FFT}(n/2, a_1, a_3, a_5, \ldots, a_{n-1}) \)

for \( k = 0 \) to \( n/2 - 1 \)
  \( y_k \leftarrow e_k + \omega^k d_k \)
  \( y_{k+n/2} \leftarrow e_k - \omega^k d_k \)

return \((y_0, y_1, y_2, \ldots, y_{n-1})\)

\( T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n) \)
Complexity

- Each $F_k$ is a sum of $\log_2 N$ transforms and (factors)
- There are $N$ $F_k$'s
- So the algorithm is $O(N \log_2 N)$
- This is a recursive algorithm
Scrambled Output of the FFT

"bit-reversed" order
function y = ffttx(x)
%FFTTX  Textbook Fast Finite
Fourier Transform.
% FFTTX(X) computes the same
finite Fourier transform as FFT(X).
% The code uses a recursive divide
and conquer algorithm for
% even order and matrix-vector
multiplication for odd order.
% If length(X) is m*p where m is
odd and p is a power of 2, the
% computational complexity of this
approach is O(m^2)*O(p*log2(p)).

x = x(:);
n = length(x);
omega = exp(-2*pi*i/n);

if rem(n,2) == 0
  % Recursive divide and conquer
  k = (0:n/2-1)';
  w = omega .^ k;
  u = ffttx(x(1:2:n-1));
  v = w.*ffttx(x(2:2:n));
  y = [u+v; u-v];
else
  % The Fourier matrix.
  j = 0:n-1;
  k = j';
  F = omega .^ (k*j);
  y = F*x;
end