Discrete Fourier Transform
and
Fast Fourier Transform

Bryant W. York
September 2018
DFT vs Continuous Fourier Transform

- As computer scientists we are more interested in the discrete Fourier transform.
- If you have interest in the continuous Fourier transform, see https://www.youtube.com/watch?v=1JnayXHhjlg

\[ F(\nu) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i \nu \cdot t} \, dt \] (forward FT)

\[ f(t) = \int_{-\infty}^{\infty} F(\nu) e^{2\pi i \nu \cdot t} \, d\nu \] (Inverse FT)

\[ \nu = \frac{\omega}{2\pi} \equiv \text{frequency (cycles per second or Hertz)} \]

Any continuous signal in the time domain can be represented as a sum of sinusoids.
Finite Discrete Functions

- Think of a **finite discrete** function which maps \( n \) points \( \{0, 1, \ldots, n - 1\} \) to \( n \) real or complex numbers as a **vector** in an \( n \)-dimensional vector space.

- Example: \( f(0) = 5, f(1) = 17, f(2) = 12 \)

\[
\begin{bmatrix}
  0 \\
  1 \\
  2
\end{bmatrix} \rightarrow \begin{bmatrix}
  5 \\
  17 \\
  12
\end{bmatrix}
\text{ in 3D space}
\]

- Note: We can consider the domain to be the group \( \mathbb{Z}/n\mathbb{Z} \). The range may be any field (most commonly, we will use \( \mathbb{C} \) as the range; i.e. \( f: \mathbb{Z}/n\mathbb{Z} \rightarrow \mathbb{C} \)).

- A finite function is defined only for a finite number of values. A discrete function is not defined for values between the values at which it is defined.
Complex numbers

- $z = a + bi$, where $i^2 = -1$ or $z = re^{i\theta}$
- Complex conjugate $\bar{z} = a - bi$
- Unit complex number $|z\bar{z}| = 1$, $\sqrt{a^2 + b^2} = 1$, $r = 1$
- $e^{i\theta} = \cos \theta + i \sin \theta$

- Complex numbers are:
  - A group with addition
  - A commutative ring with addition and multiplication
  - A field with addition and multiplication
  - An algebraically closed field
  - A commutative algebra
An $nth$ root of unity is a number $\omega$ in some field $F$ such that $\omega^n = 1$ for some integer $n$.

A primitive $nth$ root of unity is a number $\omega$ in some field such that $\omega^n = 1$ for some integer $n$ and $\omega^m \neq 1$ for any integer $m < n$.

The multiplicative inverses of roots of unity are also roots of unity.

The set of all roots of unity form an abelian group under multiplication.

The set of all $nth$ roots of unity form an abelian group under multiplication.
DFT matrix

The $n$-point discrete Fourier transform can be realized as a matrix,

$$\frac{1}{\sqrt{n}} [\omega_{ij}], i = 0, 1, \ldots, n - 1; j = 0, 1, \ldots, n - 1$$

where $\omega$ is a primitive $n$th root of unity

For example: the 4-point DFT matrix is

$$\frac{1}{\sqrt{4}} \begin{bmatrix} \omega^{0 \cdot 0} & \omega^{0 \cdot 1} & \omega^{0 \cdot 2} & \omega^{0 \cdot 3} \\ \omega^{1 \cdot 0} & \omega^{1 \cdot 1} & \omega^{1 \cdot 2} & \omega^{1 \cdot 3} \\ \omega^{2 \cdot 0} & \omega^{2 \cdot 1} & \omega^{2 \cdot 2} & \omega^{2 \cdot 3} \\ \omega^{3 \cdot 0} & \omega^{3 \cdot 1} & \omega^{3 \cdot 2} & \omega^{3 \cdot 3} \end{bmatrix} = \frac{1}{\sqrt{4}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & 1 & \omega \\ 1 & \omega^3 & \omega^2 & \omega \end{bmatrix}$$

where $\omega = e^{2\pi i/4} = e^{\pi i/2}$
Transformation by matrix multiply

- DFT maps one function into another

Example: Given $f: \mathbb{Z}/4\mathbb{Z} \to \mathbb{C}$ defined by

$f_0 = f(0) = 7; f_1 = f(1) = i; f_2 = f(2) = 2 + 3i; f_3 = f(3) = 4$

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & \omega & \omega^2 & \omega^3 \\
\frac{1}{2} & \omega^2 & \omega & \omega^2 \\
1 & \omega^3 & \omega^2 & \omega
\end{bmatrix}
\begin{bmatrix}
f_0 \\
f_1 \\
f_2 \\
f_3
\end{bmatrix}
= 
\begin{bmatrix}
g_0 \\
g_1 \\
g_2 \\
g_3
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & e^{\pi i/2} & e^{\pi i} & e^{3\pi i/2} \\
\frac{1}{2} & e^{\pi i} & e^{\pi i/2} & e^{\pi i} \\
1 & e^{3\pi i/2} & e^{\pi i} & e^{\pi i/2}
\end{bmatrix}
\begin{bmatrix}
7 \\
i \\
2 + 3i \\
4
\end{bmatrix}
= 
\frac{1}{2}
\begin{bmatrix}
13 + 4i \\
4 - 7i \\
5 + 2i \\
6 + i
\end{bmatrix}
= 
\frac{1}{2}
\begin{bmatrix}
6.5 + 2i \\
2 - 3.5i \\
2.5 + i \\
3 + .5i
\end{bmatrix}
\]
Fast Fourier Transform
Fast Fourier Transform (FFT)

- FFT is an algorithm that computes the discrete Fourier transform (DFT) of a discrete finite function or its inverse (IDFT).
- Fourier Analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain.
- An FFT rapidly computes the DFT by factorizing the DFT matrix into a product of sparse (mostly zero) factors.
- As a result it reduces the complexity of computing the DFT from $O(N^2)$ to $O(N \log N)$.
- There are many different algorithms for FFT, all with $O(N \log N)$ running time for all positive integers $N$, even prime $N$.
- Many FFT algorithms only depend on the fact that $e^{2\pi i/N}$ is a primitive $N^{th}$ root of unity.
- Since the IDFT is the same as the DFT, but with the opposite sign in the exponent and a $1/N$ factor, any FFT can be easily adapted.
The Cooley-Tukey FFT (1965)

- Known to Gauss in 1805 and re-discovered several times
- A divide and conquer algorithm that recursively breaks down a DFT of any composite size $N = N_1 N_2$ into two smaller DFTs of sizes $N_1$ and $N_2$, along with $O(N)$ multiplications by complex roots of unity, traditionally called \textit{twiddle factors}.
- Cooley Tukey is limited to sizes $N = 2^n$, i.e. powers of 2.
8-Point Radix-2 FFT
Butterfly (FFT) Network

<table>
<thead>
<tr>
<th>000</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fast Fourier Transform (FFT)

Pseudocode

In pseudocode, the below procedure could be written:[8]

\[
\begin{align*}
X_0, \ldots, N-1 & \leftarrow \text{ditfft2}(x, N, s) : \\
& \text{if } N = 1 \text{ then} \\
& \quad X_0 \leftarrow x_0 \\
& \text{else} \\
& \quad X_0, \ldots, N/2-1 \leftarrow \text{ditfft2}(x, N/2, 2s) \\
& \quad X_{N/2}, \ldots, N-1 \leftarrow \text{ditfft2}(x+s, N/2, 2s) \\
& \quad \text{for } k = 0 \text{ to } N/2-1 \\
& \quad \quad t \leftarrow X_k \\
& \quad \quad X_k \leftarrow t + \exp(-2\pi i k/N) X_{k+N/2} \\
& \quad \quad X_{k+N/2} \leftarrow t - \exp(-2\pi i k/N) X_{k+N/2} \\
& \quad \text{endfor} \\
& \text{endif}
\end{align*}
\]

DFT of \((x_0, x_5, x_{2s}, \ldots, x_{(N-1)s})\):

- trivial size-1 DFT base case
- DFT of \((x_0, x_{2s}, x_{4s}, \ldots)\)
- DFT of \((x_5, x_{5+2s}, x_{5+4s}, \ldots)\)
- combine DFTs of two halves into full DFT:
.fft.cpp
*
* This is a KISS implementation of
* the Cooley-Tukey recursive FFT algorithm.
* This works, and is visibly clear about what is happening where.
*
* To compile this with the GNU/GCC compiler:
* g++ -o fft fft.cpp -lm
*
* To run the compiled version from a *nix command line:
* ./fft
*
*
#include <complex>
#include <cstdio>

#define M_PI 3.14159265358979323846 // Pi constant with double precision
FFT in C++ from Wikipedia

/* N must be a power-of-2, or bad things will happen. Currently no check for this condition. N input samples in X[] are FFT'd and results left in X[]. Because of Nyquist theorem, N samples means only first N/2 FFT results in X[] are the answer.(upper half of X[] is a reflection with no new information). */

void fft2 (complex<double>* X, int N) {
    if(N < 2) {
        // bottom of recursion.
        // Do nothing here, because already X[0] = x[0]
    } else {
        separate(X,N); // all evens to lower half, all odds to upper half
        fft2(X, N/2); // recurse even items
        fft2(X+N/2, N/2); // recurse odd items
        // combine results of two half recursions
        for(int k=0; k<N/2; k++) {
            complex<double> e = X[k    ]; // even
            complex<double> o = X[k+N/2]; // odd
            // w is the "twiddle-factor"
            complex<double> w = exp( complex<double>(0,-2.*M_PI*k/N) );
            X[k    ] = e + w * o;
            X[k+N/2] = e - w * o;
        }
    }
}
// simple test program
int main () {
  const int nSamples = 64;
  double nSeconds = 1.0;                      // total time for sampling
  double sampleRate = nSamples / nSeconds;    // n Hz = n / second
  double freqResolution = sampleRate / nSamples; // freq step in FFT

  result
  complex<double> x[nSamples];                // storage for sample data
  complex<double> X[nSamples];                // storage for FFT answer
  const int nFreqs = 5;
  double freq[nFreqs] = { 2, 5, 11, 17, 29 }; // known freqs for testing

  // generate samples for testing
  for(int i=0; i<nSamples; i++) {
    x[i] = complex<double>(0.,0.);
    // sum several known sinusoids into x[]
    for(int j=0; j<nFreqs; j++)
      x[i] += sin( 2*M_PI*freq[j]*i/nSamples );
    X[i] = x[i];        // copy into X[] for FFT work & result
  }
}
// compute fft for this data
fft2(X,nSamples);

printf(" n\tx[]\tX[]\tf\n"); // header line
// loop to print values
for(int i=0; i<nSamples; i++) {
    printf("% 3d\t%+.3f\t%+.3f\t%g\n", i, x[i].real(), abs(X[i]), i*freqResolution );
}

Convolution

- The convolution of $f$ and $g$ is
  \[(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau = \int_{-\infty}^{\infty} f(t - \tau)g(\tau)d\tau\]

- Convolution Theorem
  - $\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}$
  - $\mathcal{F}\{f \cdot g\} = \mathcal{F}\{f\} * \mathcal{F}\{g\}$
  - $f * g = \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\}$
  - $f \cdot g = \mathcal{F}^{-1}\{\mathcal{F}\{f\} * \mathcal{F}\{g\}\}$
Convolution Theorem for Discrete Periodic Signals

- Given two discrete and periodic signals, \( x[m], y[m] \), \( m = 0, 1, \ldots, N - 1 \), then the discrete convolution is:

\[
 x[m] * y[m] = \sum_{n=0}^{N-1} x[n]y[m - n]
\]

- Convolution Theorem

\[
 F(x[m] * y[m]) = X[n]Y[n] = F(x[m])F(y[m])
\]

Where \( X[n] = \frac{1}{T} \sum_{k=0}^{N-1} x[k]e^{-2\pi ink/N} \) (\( n = 0, 1, \ldots, N - 1 \))

and \( Y[n] = \frac{1}{T} \sum_{k=0}^{N-1} y[k]e^{-2\pi ink/N} \) (\( n = 0, 1, \ldots, N - 1 \))
Videos

- Fourier Transform
  - https://www.youtube.com/watch?v=1JnayXHhjlg
- DFT
  - https://www.youtube.com/watch?v=mkGsMWi_j4Q
  - https://www.youtube.com/watch?v=r18Gi8lSkfM&t=44s
- FFT
  - https://www.youtube.com/watch?v=htCj9exbGo0
- Convolution Theorem for FT
  - https://www.youtube.com/watch?v=N-zd-T17uiE
- Discrete Convolution and Polynomial Multiplication
  - https://www.youtube.com/watch?v=T-OwClOlbm0
  - https://www.youtube.com/watch?v=f_M8nyYOPzU
  - https://www.dailymotion.com/video/x560jbg
End