6.300: Signal Processing

Fast Fourier Transform

Agenda for Recitation

- History of the FFT
- Decimation-in-time FFT algorithm
- Finite-dimensional linear algebra and the FFT
- Polynomial multiplication with the FFT

What questions do you have from lecture?

Agenda for Recitation

- History of the FFT
- Decimation-in-time FFT algorithm
- Finite-dimensional linear algebra and the FFT
- Polynomial multiplication with the FFT

History of the FFT

1805: Gauss develops a "method [that] greatly reduces the tediousness of mechanical calculations," which will later come to be known as the **fast Fourier transform**.[1]

1965: Cooley and **Tukey** publish "An Algorithm for the Machine Calculation of Complex Fourier Series." This algorithm — the Cooley-Tukey FFT algorithm — made digital signal processing (DSP) practical.

> **Professor Alan V. Oppenheim:** "You could think of it as the difference between BC and AD. The birth of the FFT was a very significant event."[a]
>
> [a] https://ethw.org/Oral-History:Alan_Oppenheim

[1] M.T. Heideman, D.H. Johnson, C.S. Burrus, and C. Truesdell. "Gauss and the History of the Fast Fourier Transform." 1985.

History of the FFT

The FFT isn't really for pencil-and-paper calculations.

FFT Speedup

The small change in operation count for small N also explains why Gauss was not so excited about the method.

N	DFT	FFT	speed-up
2	4	2	2.0
4	16	8	2.0
8	64	24	2.7
16	256	64	4.0

So, we will try to **understand** conceptually why the FFT is so fast, rather than working through examples (with relatively small N) on the chalkboard.

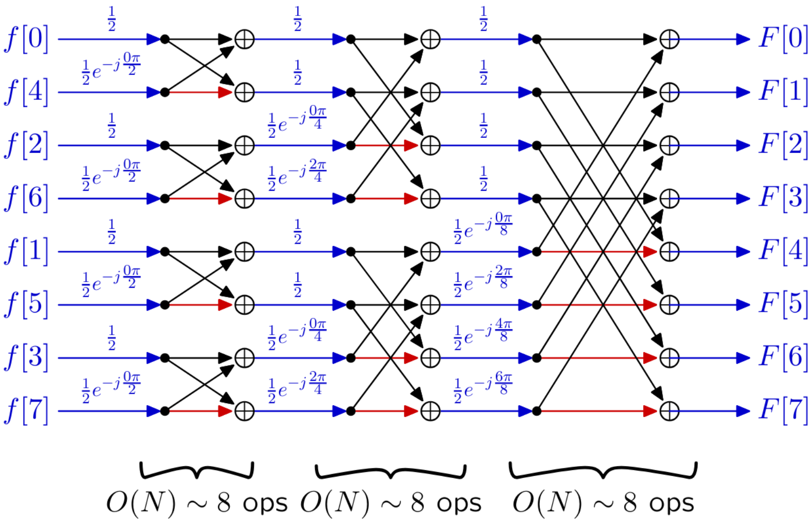
Agenda for Recitation

- History of the FFT
- Decimation-in-time FFT algorithm
- Finite-dimensional linear algebra and the FFT
- Polynomial multiplication with the FFT

Agenda for Recitation

- History of the FFT
- Decimation-in-time FFT algorithm
- Finite-dimensional linear algebra and the FFT
- Polynomial multiplication with the FFT

Decimation in Time



Graphic: Professor Denny Freeman (freeman@mit.edu)

Decimation in Time

The **signal flowgraph** on the previous slide is a graphical representation of the **radix-2 decimation-in-time FFT algorithm**. The big idea is to split an N-point DFT into a linear combination of $N/2$-point DFTs.

Decimation in Time

The **signal flowgraph** on the previous slide is a graphical representation of the **radix-2 decimation-in-time FFT algorithm**. The big idea is to split an N-point DFT into a linear combination of $N/2$-point DFTs.

$$X_N[k] = \frac{1}{2}\left(X_{N/2}^{\text{even}}[k] + r_N^k X_{N/2}^{\text{odd}}[k]\right)$$

Here, $r_N \triangleq e^{-j\frac{2\pi}{N}}$ denotes the **primitive N^{th} root of unity** — also called "**twiddle factor**," which is more fun to say.

Decimation in Time

The **signal flowgraph** on the previous slide is a graphical representation of the **radix-2 decimation-in-time FFT algorithm**. The big idea is to split an N-point DFT into a linear combination of $N/2$-point DFTs.

$$X_N[k] = \frac{1}{2}\left(X_{N/2}^{\text{even}}[k] + r_N^k X_{N/2}^{\text{odd}}[k]\right)$$

Here, $r_N \triangleq e^{-j\frac{2\pi}{N}}$ denotes the **primitive N^{th} root of unity** — also called "**twiddle factor**," which is more fun to say.

- Repeat until $N/2 = 1$, when we can't divide by 2 again.
- The DFT of a 1-point signal is the signal itself.
- "Glue" the 1-point DFTs together.

Decimation in Time

The **signal flowgraph** on the previous slide is a graphical representation of the **radix-2 decimation-in-time FFT algorithm**. The big idea is to split an N-point DFT into a linear combination of $N/2$-point DFTs.

$$X_N[k] = \tfrac{1}{2}\left(X_{N/2}^{\text{even}}[k] + r_N^k X_{N/2}^{\text{odd}}[k]\right)$$

Here, $r_N \triangleq e^{-j\frac{2\pi}{N}}$ denotes the **primitive N^{th} root of unity** — also called "**twiddle factor**," which is more fun to say.

- Repeat until $N/2 = 1$, when we can't divide by 2 again.
- The DFT of a 1-point signal is the signal itself.
- "Glue" the 1-point DFTs together.

Directly computing the DFT requires $\mathcal{O}(N \log_2 N)$ operations. The FFT exploits the structure of the DFT to cut down the number of operations to $\mathcal{O}(N \log_2 N)$.

Agenda for Recitation

- History of the FFT
- Decimation-in-time FFT algorithm
- Finite-dimensional linear algebra and the FFT
- Polynomial multiplication with the FFT

Agenda for Recitation

- History of the FFT
- Decimation-in-time FFT algorithm
- Finite-dimensional linear algebra and the FFT
- Polynomial multiplication with the FFT

Signal Processing and Linear Algebra

Signal processing is built on **linear algebra**.

For example, the analysis formulæ are inner products.

$$X(\omega) = \left\langle x(t) \middle| e^{j\omega t} \right\rangle = \int_{-\infty}^{\infty} x(t) e^{-j\omega t} dt$$

Signal Processing and Linear Algebra

Signal processing is built on **linear algebra**.

For example, the analysis formulæ are inner products.

$$X(\omega) = \left\langle x(t) \middle| e^{j\omega t} \right\rangle = \int_{-\infty}^{\infty} x(t) e^{-j\omega t} dt$$

So far, we have been doing linear algebra in an **infinite-dimensional space**, with $t \in \mathbb{R}$ or $n \in \mathbb{Z}$. If we restrict n to lie in $\{0, 1, 2, \ldots, N-1\}$, then we'll be doing linear algebra in a **finite-dimensional space**.

Signal Processing and Linear Algebra

Signal processing is built on **linear algebra**.

For example, the analysis formulæ are inner products.

$$X(\omega) = \left\langle x(t) \middle| e^{j\omega t} \right\rangle = \int_{-\infty}^{\infty} x(t) e^{-j\omega t} dt$$

So far, we have been doing linear algebra in an **infinite-dimensional space**, with $t \in \mathbb{R}$ or $n \in \mathbb{Z}$. If we restrict n to lie in $\{0, 1, 2, \ldots, N-1\}$, then we'll be doing linear algebra in a **finite-dimensional space**.

Irving Kaplansky (1917–2006)

> *[Paul Halmos and I] share a philosophy about linear algebra: We think basis-free, we write basis-free, but when the chips are down we close the office door and compute with matrices like fury.*

This is where **matrices** and **vectors** come into play.

Finite-Dimensional Linear Algebra

Letting $r = e^{-j\frac{2\pi}{N}}$ denote the **primitive N^{th} root of unity**, we can write the **DFT analysis equation** as follows.

$$X[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[n] r^{kn}$$

Express the DFT analysis equation as a matrix-vector multiplication.

Finite-Dimensional Linear Algebra

Letting $r = e^{-j\frac{2\pi}{N}}$ denote the **primitive N^{th} root of unity**, we can write the **DFT analysis equation** as follows.

$$X[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[n] r^{kn}$$

The **DFT** is a **matrix-vector multiplication:** $\widehat{\boldsymbol{x}} = \frac{1}{N}\boldsymbol{F}\boldsymbol{x}$.

$$\underbrace{\begin{bmatrix} X[0] \\ X[1] \\ X[2] \\ \vdots \\ X[N-1] \end{bmatrix}}_{\widehat{\boldsymbol{x}}\text{ (frequency)}} = \frac{1}{N} \underbrace{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & r & r^2 & \cdots & r^{N-1} \\ 1 & r^2 & r^4 & \cdots & r^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & r^{N-1} & r^{2(N-1)} & \cdots & r^{(N-1)^2} \end{bmatrix}}_{N\times N\text{ Fourier matrix } \boldsymbol{F}_N\text{ (DFT)}} \underbrace{\begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ \vdots \\ x[N-1] \end{bmatrix}}_{\boldsymbol{x}\text{ (time)}}$$

Let's examine $\boldsymbol{F}_N$, the **Fourier matrix**.

Fourier Matrix

The $N = 1$ Fourier matrix is hardly a matrix.

$$\boldsymbol{F}_1 = \left[e^{-j\frac{2\pi}{1}0} \right] = \left[1 \right]$$

Fourier Matrix

The $N = 1$ Fourier matrix is hardly a matrix.

$$\boldsymbol{F}_1 = \left[e^{-j\frac{2\pi}{1}0} \right] = \left[1 \right]$$

The $N = 2$ Fourier matrix is simple.

$$\boldsymbol{F}_2 = \begin{bmatrix} e^{-j\frac{2\pi}{2}0} & e^{-j\frac{2\pi}{2}0} \\ e^{-j\frac{2\pi}{2}0} & e^{-j\frac{2\pi}{2}1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Fourier Matrix

The $N = 1$ Fourier matrix is hardly a matrix.

$$\boldsymbol{F}_1 = \left[e^{-j\frac{2\pi}{1}0}\right] = \left[1\right]$$

The $N = 2$ Fourier matrix is simple.

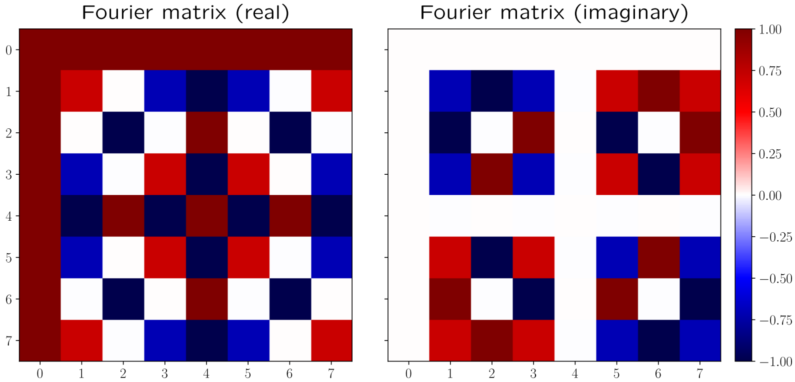
$$\boldsymbol{F}_2 = \begin{bmatrix} e^{-j\frac{2\pi}{2}0} & e^{-j\frac{2\pi}{2}0} \\ e^{-j\frac{2\pi}{2}0} & e^{-j\frac{2\pi}{2}1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The $N = 4$ Fourier matrix is relatively simple, too.

$$\boldsymbol{F}_4 = \begin{bmatrix} e^{-j\frac{2\pi}{4}0} & e^{-j\frac{2\pi}{4}0} & e^{-j\frac{2\pi}{4}0} & e^{-j\frac{2\pi}{4}0} \\ e^{-j\frac{2\pi}{4}0} & e^{-j\frac{2\pi}{4}1} & e^{-j\frac{2\pi}{4}2} & e^{-j\frac{2\pi}{4}3} \\ e^{-j\frac{2\pi}{4}0} & e^{-j\frac{2\pi}{4}2} & e^{-j\frac{2\pi}{4}4} & e^{-j\frac{2\pi}{4}6} \\ e^{-j\frac{2\pi}{4}0} & e^{-j\frac{2\pi}{4}3} & e^{-j\frac{2\pi}{4}6} & e^{-j\frac{2\pi}{4}9} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix}$$

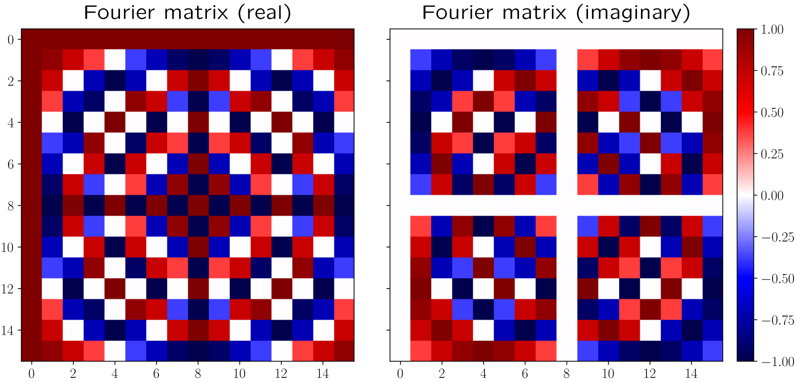
Fourier Matrix

Here's a visualization of the $N = 8$ Fourier matrix.



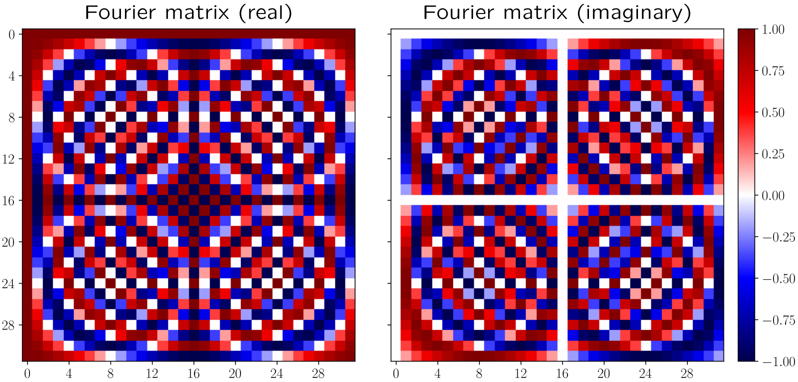
Fourier Matrix

Here's a visualization of the $N = 16$ Fourier matrix.



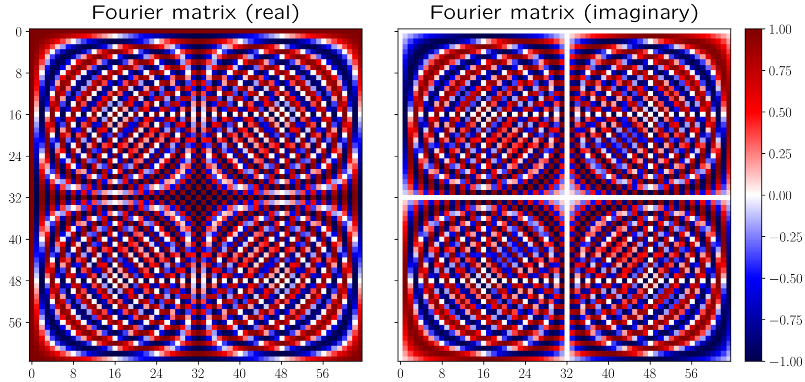
Fourier Matrix

Here's a visualization of the $N = 32$ Fourier matrix.



Fourier Matrix

Here's a visualization of the $N = 64$ Fourier matrix.



Counting Operations: Analysis

It takes $\mathcal{O}(N^2)$ operations to evaluate the sum

$$X[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[n]e^{-jk\frac{2\pi}{N}n}$$

for $k \in \{0, 1, 2, \ldots, N-1\}$.

- N different values of k
- $\mathcal{O}(N)$ operations per k

Counting Operations: Analysis

It takes $\mathcal{O}(N^2)$ operations to evaluate the sum

$$X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{-jk\frac{2\pi}{N}n}$$

for $k \in \{0, 1, 2, \ldots, N-1\}$.

- N different values of k
- $\mathcal{O}(N)$ operations per k

Likewise, the matrix-vector multiplication

$$\widehat{\boldsymbol{x}} = \tfrac{1}{N}\boldsymbol{F}\boldsymbol{x}$$

takes $\mathcal{O}(N^2)$ operations.

Counting Operations: Analysis

It takes $\mathcal{O}(N^2)$ operations to evaluate the sum

$$X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{-jk\frac{2\pi}{N}n}$$

for $k \in \{0, 1, 2, \ldots, N-1\}$.

- N different values of k
- $\mathcal{O}(N)$ operations per k

Likewise, the matrix-vector multiplication

$$\widehat{\boldsymbol{x}} = \tfrac{1}{N} \boldsymbol{F} \boldsymbol{x}$$

takes $\mathcal{O}(N^2)$ operations.

Whichever way we look at it, direct computation of the DFT takes $\mathcal{O}(N^2)$ operations.

Counting Operations: Synthesis

Likewise, it takes $\mathcal{O}(N^2)$ operations to evaluate the sum

$$x[n] = \sum_{k=0}^{N-1} X[k] e^{jk\frac{2\pi}{N}n}$$

for $n \in \{0, 1, 2, \ldots, N-1\}$.

Counting Operations: Synthesis

Likewise, it takes $\mathcal{O}(N^2)$ operations to evaluate the sum

$$x[n] = \sum_{k=0}^{N-1} X[k] e^{jk\frac{2\pi}{N}n}$$

for $n \in \{0, 1, 2, \ldots, N-1\}$.

However, in general, it takes $\mathcal{O}(N^3)$ operations to solve a matrix-vector equation like $\widehat{\boldsymbol{x}} = \frac{1}{N}\boldsymbol{F}\boldsymbol{x}$ for $\boldsymbol{x}$. In general, inverting a matrix or solving a system of linear equations is $\mathcal{O}(N^3)$.

$$\boldsymbol{x} = \boldsymbol{F}^{-1}\widehat{\boldsymbol{x}}$$

Counting Operations: Synthesis

Likewise, it takes $\mathcal{O}(N^2)$ operations to evaluate the sum

$$x[n] = \sum_{k=0}^{N-1} X[k] e^{jk\frac{2\pi}{N}n}$$

for $n \in \{0, 1, 2, \ldots, N-1\}$.

However, in general, it takes $\mathcal{O}(N^3)$ operations to solve a matrix-vector equation like $\widehat{\boldsymbol{x}} = \frac{1}{N}\boldsymbol{F}\boldsymbol{x}$ for $\boldsymbol{x}$. In general, inverting a matrix or solving a system of linear equations is $\mathcal{O}(N^3)$.

$$\boldsymbol{x} = \boldsymbol{F}^{-1}\widehat{\boldsymbol{x}}$$

Something seems off: Shouldn't we able be to solve $\widehat{\boldsymbol{x}} = \frac{1}{N}\boldsymbol{F}\boldsymbol{x}$ for $\boldsymbol{x}$ using no more than $\mathcal{O}(N^2)$ operations?

Counting Operations: Synthesis

Indeed, we can solve $\widehat{\boldsymbol{x}} = \frac{1}{N}\boldsymbol{F}\boldsymbol{x}$ for $\boldsymbol{x}$ in $\mathcal{O}(N^2)$ by exploiting the special structure of $\boldsymbol{F}$, the Fourier matrix.

Counting Operations: Synthesis

Indeed, we can solve $\widehat{\boldsymbol{x}} = \frac{1}{N}\boldsymbol{F}\boldsymbol{x}$ for $\boldsymbol{x}$ in $\mathcal{O}(N^2)$ by exploiting the special structure of $\boldsymbol{F}$, the Fourier matrix.

$\boldsymbol{F}^{-1} = \boldsymbol{F}^*$ is simply the **conjugate-transpose** of $\boldsymbol{F}$. This is trivial to compute.

$$\boldsymbol{x} = \boldsymbol{F}^{-1}\widehat{\boldsymbol{x}} = \boldsymbol{F}^*\widehat{\boldsymbol{x}}$$

Counting Operations: Synthesis

Indeed, we can solve $\widehat{\boldsymbol{x}} = \frac{1}{N}\boldsymbol{F}\boldsymbol{x}$ for $\boldsymbol{x}$ in $\mathcal{O}(N^2)$ by exploiting the special structure of $\boldsymbol{F}$, the Fourier matrix.

$\boldsymbol{F}^{-1} = \boldsymbol{F}^*$ is simply the **conjugate-transpose** of $\boldsymbol{F}$. This is trivial to compute.

$$\boldsymbol{x} = \boldsymbol{F}^{-1}\widehat{\boldsymbol{x}} = \boldsymbol{F}^*\widehat{\boldsymbol{x}}$$

The IDFT is a matrix-vector multiplication, which takes $\mathcal{O}(N^2)$ operations — not the $\mathcal{O}(N^3)$ we predicted earlier.

Counting Operations: Synthesis

Indeed, we can solve $\widehat{\boldsymbol{x}} = \frac{1}{N}\boldsymbol{F}\boldsymbol{x}$ for $\boldsymbol{x}$ in $\mathcal{O}(N^2)$ by exploiting the special structure of $\boldsymbol{F}$, the Fourier matrix.

$\boldsymbol{F}^{-1} = \boldsymbol{F}^*$ is simply the **conjugate-transpose** of $\boldsymbol{F}$. This is trivial to compute.

$$\boldsymbol{x} = \boldsymbol{F}^{-1}\widehat{\boldsymbol{x}} = \boldsymbol{F}^*\widehat{\boldsymbol{x}}$$

The IDFT is a matrix-vector multiplication, which takes $\mathcal{O}(N^2)$ operations — not the $\mathcal{O}(N^3)$ we predicted earlier.

So, direct computation of the DFT or IDFT requires $\mathcal{O}(N^2)$ operations. **FFT algorithms** let us do better — $\mathcal{O}(N\log_2 N)$ **operations!**

FFT as a Matrix Factorization

The big idea behind the **matrix factorization** approach is this: Multiplying a vector by a **sparse matrix** (i.e., mostly zeros) requires fewer than $\mathcal{O}(N^2)$ operations — perhaps on the order of $\mathcal{O}(N)$ **operations** instead.

FFT as a Matrix Factorization

The big idea behind the **matrix factorization** approach is this: Multiplying a vector by a **sparse matrix** (i.e., mostly zeros) requires fewer than $\mathcal{O}(N^2)$ operations — perhaps on the order of $\mathcal{O}(N)$ **operations** instead.

FFT: Factor $\boldsymbol{F}_N$ into $\log_2 N$ sparse matrices, where multiplying by each sparse matrix requires $\mathcal{O}(N)$ operations. The total cost is $\mathcal{O}(N \log_2 N)$.

FFT as a Matrix Factorization

The big idea behind the **matrix factorization** approach is this: Multiplying a vector by a **sparse matrix** (i.e., mostly zeros) requires fewer than $\mathcal{O}(N^2)$ operations — perhaps on the order of $\mathcal{O}(N)$ **operations** instead.

FFT: Factor $\boldsymbol{F}_N$ into $\log_2 N$ sparse matrices, where multiplying by each sparse matrix requires $\mathcal{O}(N)$ operations. The total cost is $\mathcal{O}(N \log_2 N)$.

$$\boldsymbol{F}_N = \boldsymbol{F}_{\log_2 N} \cdots \boldsymbol{F}_2 \boldsymbol{F}_1 \boldsymbol{P}_N$$

- $\boldsymbol{F}_{\log_2 N}$, . . . , $\boldsymbol{F}_2$, $\boldsymbol{F}_1$ are sparse matrices.
- $\boldsymbol{P}_N$ is a permutation matrix.

FFT as a Matrix Factorization

Recursive block-matrix factorization:

$$\boldsymbol{F}_N = \begin{bmatrix} \boldsymbol{I}_{N/2} & \boldsymbol{D}_{N/2} \\ \boldsymbol{I}_{N/2} & -\boldsymbol{D}_{N/2} \end{bmatrix} \begin{bmatrix} \boldsymbol{F}_{N/2} & \\ & \boldsymbol{F}_{N/2} \end{bmatrix} \begin{bmatrix} \text{even-odd} \\ \text{permutation} \end{bmatrix}$$

FFT as a Matrix Factorization

Recursive block-matrix factorization:

$$\boldsymbol{F}_N = \begin{bmatrix} \boldsymbol{I}_{N/2} & \boldsymbol{D}_{N/2} \\ \boldsymbol{I}_{N/2} & -\boldsymbol{D}_{N/2} \end{bmatrix} \begin{bmatrix} \boldsymbol{F}_{N/2} & \\ & \boldsymbol{F}_{N/2} \end{bmatrix} \begin{bmatrix} \text{even-odd} \\ \text{permutation} \end{bmatrix}$$

Example with $N = 4$:

$$\boldsymbol{F}_4 = \begin{bmatrix} 1 & & 1 & \\ & 1 & & j \\ 1 & & -1 & \\ & 1 & & -j \end{bmatrix} \begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & & 1 & \\ & 1 & & \\ & & & 1 \end{bmatrix}$$

FFT as a Matrix Factorization

Recursive block-matrix factorization:

$$\boldsymbol{F}_N = \begin{bmatrix} \boldsymbol{I}_{N/2} & \boldsymbol{D}_{N/2} \\ \boldsymbol{I}_{N/2} & -\boldsymbol{D}_{N/2} \end{bmatrix} \begin{bmatrix} \boldsymbol{F}_{N/2} & \\ & \boldsymbol{F}_{N/2} \end{bmatrix} \begin{bmatrix} \text{even-odd} \\ \text{permutation} \end{bmatrix}$$

Example with $N = 4$:

$$\boldsymbol{F}_4 = \begin{bmatrix} 1 & & 1 & \\ & 1 & & j \\ 1 & & -1 & \\ & 1 & & -j \end{bmatrix} \begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & & 1 & \\ & 1 & & \\ & & & 1 \end{bmatrix}$$

FFT algorithms let us compute (circular) convolutions in $\mathcal{O}(N \log_2 N)$ operations, rather than $\mathcal{O}(N^2)$ operations! Let's investigate how.

Fast Circular Convolution with the FFT

Let

$$\boldsymbol{x} = \left[x[0], x[1], x[2], x[3], \ldots, x[N-1]\right]^\top$$

and

$$\boldsymbol{h} = \left[h[0], h[1], h[2], h[3], \ldots, h[N-1]\right]^\top$$

hold the samples of $x[n]$ and $h[n]$, respectively. (Assume we zero-padded them to the same length if necessary.)

Fast Circular Convolution with the FFT

Let

$$\boldsymbol{x} = \big[x[0], x[1], x[2], x[3], \ldots, x[N-1]\big]^\top$$

and

$$\boldsymbol{h} = \big[h[0], h[1], h[2], h[3], \ldots, h[N-1]\big]^\top$$

hold the samples of $x[n]$ and $h[n]$, respectively. (Assume we zero-padded them to the same length if necessary.)

Circular convolution is a matrix-vector multiplication.

$$(\boldsymbol{x} \circledast \boldsymbol{h}) = \begin{bmatrix} h[0] & h[N-1] & h[N-2] & \cdots & h[1] \\ h[1] & h[0] & h[N-1] & \cdots & h[2] \\ h[2] & h[1] & h[0] & \cdots & h[3] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h[N-1] & h[N-2] & h[N-3] & \cdots & h[0] \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ \vdots \\ x[N-1] \end{bmatrix}$$

Fast Circular Convolution with the FFT

Circular convolution is a matrix-vector multiplication.

$$(\boldsymbol{x} \circledast \boldsymbol{h}) = \underbrace{\begin{bmatrix} h[0] & h[N-1] & h[N-2] & \cdots & h[1] \\ h[1] & h[0] & h[N-1] & \cdots & h[2] \\ h[2] & h[1] & h[0] & \cdots & h[3] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h[N-1] & h[N-2] & h[N-3] & \cdots & h[0] \end{bmatrix}}_{\boldsymbol{H} = \mathrm{circulant}(\boldsymbol{h})} \underbrace{\begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ \vdots \\ x[N-1] \end{bmatrix}}_{\boldsymbol{x}}$$

The matrix $\boldsymbol{H} = \mathrm{circulant}(\boldsymbol{h})$ has a very special structure: It is a **circulant matrix!** Each column (or row) is a circular shift of another column (or row) in the matrix.

Fast Circular Convolution with the FFT

LTI Systems

$$e^{jk\frac{2\pi}{N}n} \rightarrow \boxed{\text{LTI}} \rightarrow H[k]e^{jk\frac{2\pi}{N}n}$$

Recall: Exponentials are eigenfunctions of LTI systems.

- Eigenfunctions: $\phi_k[n] = e^{jk\frac{2\pi}{N}n}$
- Eigenvalues: $\lambda_k = H[k]$

Fast Circular Convolution with the FFT

LTI Systems

$$e^{jk\frac{2\pi}{N}n} \rightarrow \boxed{\text{LTI}} \rightarrow H[k]e^{jk\frac{2\pi}{N}n}$$

Recall: Exponentials are eigenfunctions of LTI systems.

- Eigenfunctions: $\phi_k[n] = e^{jk\frac{2\pi}{N}n}$
- Eigenvalues: $\lambda_k = H[k]$

Arrange samples of the eigenfunctions into vectors.

$$\boldsymbol{\phi}_k = \begin{bmatrix} \phi_k[0] \\ \phi_k[1] \\ \phi_k[2] \\ \vdots \\ \phi_k[N-1] \end{bmatrix} = \begin{bmatrix} 1 \\ e^{jk\frac{2\pi}{N}} \\ e^{jk\frac{2\pi}{N}2} \\ \vdots \\ e^{jk\frac{2\pi}{N}(N-1)} \end{bmatrix} \quad \text{for } k \in \{0, 1, 2, \ldots, N-1\}$$

Fast Circular Convolution with the FFT

Eigenpairs $(\boldsymbol{\phi}_k, \lambda_k)$ satisfy the eigenequation.

$$\boldsymbol{H}\boldsymbol{\phi}_k = \lambda_k \boldsymbol{\phi}_k$$

Fast Circular Convolution with the FFT

Eigenpairs $(\boldsymbol{\phi}_k, \lambda_k)$ satisfy the eigenequation.

$$\boldsymbol{H}\boldsymbol{\phi}_k = \lambda_k \boldsymbol{\phi}_k$$

Write out the eigenequation for $k \in \{0, \ldots, N-1\}$.

$$\boldsymbol{H} \underbrace{\begin{bmatrix} | & | & & | \\ \boldsymbol{\phi}_0 & \boldsymbol{\phi}_1 & \cdots & \boldsymbol{\phi}_{N-1} \\ | & | & & | \end{bmatrix}}_{\text{eigenvector matrix } \boldsymbol{\Phi}} = \underbrace{\begin{bmatrix} | & | & & | \\ \boldsymbol{\phi}_0 & \boldsymbol{\phi}_1 & \cdots & \boldsymbol{\phi}_{N-1} \\ | & | & & | \end{bmatrix}}_{\text{eigenvector matrix } \boldsymbol{\Phi}} \underbrace{\begin{bmatrix} \lambda_0 & & \\ & \ddots & \\ & & \lambda_{N-1} \end{bmatrix}}_{\text{eigenvalue matrix } \boldsymbol{\Lambda}}$$

Fast Circular Convolution with the FFT

Eigenpairs (ϕ_k, λ_k) satisfy the eigenequation.

$$\boldsymbol{H}\boldsymbol{\phi}_k = \lambda_k \boldsymbol{\phi}_k$$

Write out the eigenequation for $k \in \{0, \ldots, N-1\}$.

$$\boldsymbol{H} \underbrace{\begin{bmatrix} | & | & & | \\ \boldsymbol{\phi}_0 & \boldsymbol{\phi}_1 & \cdots & \boldsymbol{\phi}_{N-1} \\ | & | & & | \end{bmatrix}}_{\text{eigenvector matrix } \boldsymbol{\Phi}} = \underbrace{\begin{bmatrix} | & | & & | \\ \boldsymbol{\phi}_0 & \boldsymbol{\phi}_1 & \cdots & \boldsymbol{\phi}_{N-1} \\ | & | & & | \end{bmatrix}}_{\text{eigenvector matrix } \boldsymbol{\Phi}} \underbrace{\begin{bmatrix} \lambda_0 & & \\ & \ddots & \\ & & \lambda_{N-1} \end{bmatrix}}_{\text{eigenvalue matrix } \boldsymbol{\Lambda}}$$

This yields the **eigenvalue decomposition** of $\boldsymbol{H}$.

$$\boldsymbol{H}\boldsymbol{\Phi} = \boldsymbol{\Phi}\boldsymbol{\Lambda} \iff \boldsymbol{H} = \boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^{-1} = \boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^*$$

$\boldsymbol{\Phi} = \boldsymbol{F}^{-1}$ is the (inverse) Fourier matrix. $\boldsymbol{\Phi}^{-1} = \boldsymbol{\Phi}^* = \boldsymbol{F}$.

Fast Circular Convolution with the FFT

All circulant matrices have a full set of orthogonal eigenvectors — the Fourier basis vectors!

$$\boldsymbol{\phi}_k = \begin{bmatrix} 1 \\ e^{jk\frac{2\pi}{N}} \\ e^{jk\frac{2\pi}{N}2} \\ \vdots \\ e^{jk\frac{2\pi}{N}(N-1)} \end{bmatrix} \quad \text{for } k \in \{0, 1, 2, \ldots, N-1\}$$

Fast Circular Convolution with the FFT

All circulant matrices have a full set of orthogonal eigenvectors — the Fourier basis vectors!

$$\phi_k = \begin{bmatrix} 1 \\ e^{jk\frac{2\pi}{N}} \\ e^{jk\frac{2\pi}{N}2} \\ \vdots \\ e^{jk\frac{2\pi}{N}(N-1)} \end{bmatrix} \quad \text{for } k \in \{0, 1, 2, \ldots, N-1\}$$

The eigenvalues come from the analysis equation.

$$\lambda_k = H[k] = \frac{1}{N}\left(h[0] + h[1]e^{-jk\frac{2\pi}{N}} + \cdots + h[N-1]e^{-jk\frac{2\pi}{N}(N-1)}\right)$$

Fast Circular Convolution with the FFT

Recall that circular convolution is a matrix-vector multiplication: $\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h}) = \boldsymbol{H}\boldsymbol{x}$, where $\boldsymbol{H} = \mathrm{circulant}(\boldsymbol{h})$ is a circulant matrix.

Fast Circular Convolution with the FFT

Recall that circular convolution is a matrix-vector multiplication: $\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h}) = \boldsymbol{H}\boldsymbol{x}$, where $\boldsymbol{H} = \text{circulant}(\boldsymbol{h})$ is a circulant matrix.

Substitute the eigenvalue decomposition for $\boldsymbol{H}$.

$$\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h}) = \boldsymbol{H}\boldsymbol{x} = (\boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^*)\boldsymbol{x}$$

Fast Circular Convolution with the FFT

Recall that circular convolution is a matrix-vector multiplication: $\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h}) = \boldsymbol{H}\boldsymbol{x}$, where $\boldsymbol{H} = \mathrm{circulant}(\boldsymbol{h})$ is a circulant matrix.

Substitute the eigenvalue decomposition for $\boldsymbol{H}$.

$$\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h}) = \boldsymbol{H}\boldsymbol{x} = (\boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^*)\boldsymbol{x}$$

The DFT $\widehat{\boldsymbol{x}} = \boldsymbol{\Phi}^*\boldsymbol{x}$ takes $\mathcal{O}(N \log_2 N)$ operations.

Fast Circular Convolution with the FFT

Recall that circular convolution is a matrix-vector multiplication: $\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h}) = \boldsymbol{H}\boldsymbol{x}$, where $\boldsymbol{H} = \mathrm{circulant}(\boldsymbol{h})$ is a circulant matrix.

Substitute the eigenvalue decomposition for $\boldsymbol{H}$.

$$\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h}) = \boldsymbol{H}\boldsymbol{x} = (\boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^*)\boldsymbol{x}$$

The DFT $\widehat{\boldsymbol{x}} = \boldsymbol{\Phi}^*\boldsymbol{x}$ takes $\mathcal{O}(N \log_2 N)$ operations.

Multiplying $\widehat{\boldsymbol{x}}$ by $\boldsymbol{\Lambda}$ (diagonal matrix of eigenvalues, i.e., the $H[k]$ terms) takes only $\mathcal{O}(N)$ operations: $\widehat{\boldsymbol{y}} = \boldsymbol{\Lambda}\widehat{\boldsymbol{x}}$.

Fast Circular Convolution with the FFT

Recall that circular convolution is a matrix-vector multiplication: $\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h}) = \boldsymbol{H}\boldsymbol{x}$, where $\boldsymbol{H} = \mathrm{circulant}(\boldsymbol{h})$ is a circulant matrix.

Substitute the eigenvalue decomposition for $\boldsymbol{H}$.

$$\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h}) = \boldsymbol{H}\boldsymbol{x} = (\boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^*)\boldsymbol{x}$$

The DFT $\widehat{\boldsymbol{x}} = \boldsymbol{\Phi}^*\boldsymbol{x}$ takes $\mathcal{O}(N \log_2 N)$ operations.

Multiplying $\widehat{\boldsymbol{x}}$ by $\boldsymbol{\Lambda}$ (diagonal matrix of eigenvalues, i.e., the $H[k]$ terms) takes only $\mathcal{O}(N)$ operations: $\widehat{\boldsymbol{y}} = \boldsymbol{\Lambda}\widehat{\boldsymbol{x}}$.

Finally, compute the IDFT: $\boldsymbol{y} = \boldsymbol{\Phi}\widehat{\boldsymbol{y}}$. This is $\mathcal{O}(N \log_2 N)$.

Fast Circular Convolution with the FFT

Recall that circular convolution is a matrix-vector multiplication: $\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h}) = \boldsymbol{H}\boldsymbol{x}$, where $\boldsymbol{H} = \mathrm{circulant}(\boldsymbol{h})$ is a circulant matrix.

Substitute the eigenvalue decomposition for $\boldsymbol{H}$.

$$\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h}) = \boldsymbol{H}\boldsymbol{x} = (\boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^*)\boldsymbol{x}$$

The DFT $\widehat{\boldsymbol{x}} = \boldsymbol{\Phi}^*\boldsymbol{x}$ takes $\mathcal{O}(N \log_2 N)$ operations.

Multiplying $\widehat{\boldsymbol{x}}$ by $\boldsymbol{\Lambda}$ (diagonal matrix of eigenvalues, i.e., the $H[k]$ terms) takes only $\mathcal{O}(N)$ operations: $\widehat{\boldsymbol{y}} = \boldsymbol{\Lambda}\widehat{\boldsymbol{x}}$.

Finally, compute the IDFT: $\boldsymbol{y} = \boldsymbol{\Phi}\widehat{\boldsymbol{y}}$. This is $\mathcal{O}(N \log_2 N)$.

Cost: $\mathcal{O}(N \log_2 N) + \mathcal{O}(N) + \mathcal{O}(N \log_2 N) \implies \mathcal{O}(N \log_2 N)$

Fast Circular Convolution with the FFT

Summary

Circulant matrices have a nice eigenvalue decomposition.

- Transform: $\widehat{\boldsymbol{x}} = \boldsymbol{\Phi}^* \boldsymbol{x}$. $\mathcal{O}(N \log_2 N)$
- Multiply by eigenvalues: $\widehat{\boldsymbol{y}} = \boldsymbol{\Lambda} \widehat{\boldsymbol{x}}$. $\mathcal{O}(N)$
- Inverse transform: $\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h}) = \boldsymbol{\Phi} \widehat{\boldsymbol{y}}$. $\mathcal{O}(N \log_2 N)$

$$\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h}) = \underbrace{(\boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^*)}_{\boldsymbol{H}} \boldsymbol{x}$$

FFT algorithms let us compute $\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h})$ in $\mathcal{O}(N \log_2 N)$ operations, rather than $\mathcal{O}(N^2)$ operations!

If $x[n]$ is length L and $h[n]$ is length M, we can **zero-pad** both to length $N = L + M - 1$ so that $(x \circledast h)[n] = (x * h)[n]$ for $n \in \{0, 1, 2, \ldots, N-1\}$.

Agenda for Recitation

- History of the FFT
- Decimation-in-time FFT algorithm
- Finite-dimensional linear algebra and the FFT
- Polynomial multiplication with the FFT

Agenda for Recitation

- History of the FFT
- Decimation-in-time FFT algorithm
- Finite-dimensional linear algebra and the FFT
- Polynomial multiplication with the FFT

Polynomial Multiplication with the FFT

Let

$$p(x) = P[0] + P[1]x + P[2]x^2 + P[3]x^3 + P[4]x^4 + \cdots$$

and

$$q(x) = Q[0] + Q[1]x + Q[2]x^2 + Q[3]x^3 + Q[4]x^4 + \cdots$$

denote polynomials.

Polynomial Multiplication with the FFT

Let

$$p(x) = P[0] + P[1]x + P[2]x^2 + P[3]x^3 + P[4]x^4 + \cdots$$

and

$$q(x) = Q[0] + Q[1]x + Q[2]x^2 + Q[3]x^3 + Q[4]x^4 + \cdots$$

denote polynomials. The coefficients for $p(x)q(x)$ are given by the **convolution** of the coefficients $P[k]$ and $Q[k]$.

$$(P * Q)[k] = \sum_{m=-\infty}^{\infty} P[m]Q[k-m] = \sum_{m=-\infty}^{\infty} Q[m]P[k-m]$$

Polynomial Multiplication with the FFT

Let

$$p(x) = P[0] + P[1]x + P[2]x^2 + P[3]x^3 + P[4]x^4 + \cdots$$

and

$$q(x) = Q[0] + Q[1]x + Q[2]x^2 + Q[3]x^3 + Q[4]x^4 + \cdots$$

denote polynomials. The coefficients for $p(x)q(x)$ are given by the **convolution** of the coefficients $P[k]$ and $Q[k]$.

$$(P * Q)[k] = \sum_{m=-\infty}^{\infty} P[m]\,Q[k-m] = \sum_{m=-\infty}^{\infty} Q[m]P[k-m]$$

Notice that multiplication in one domain corresponds to convolution in the other domain: $p(x)q(x) \iff (P * Q)[k]$.

Polynomial Multiplication with the FFT

Let

$$p(x) = P[0] + P[1]x + P[2]x^2 + P[3]x^3 + P[4]x^4 + \cdots$$

and

$$q(x) = Q[0] + Q[1]x + Q[2]x^2 + Q[3]x^3 + Q[4]x^4 + \cdots$$

denote polynomials. The coefficients for $p(x)q(x)$ are given by the **convolution** of the coefficients $P[k]$ and $Q[k]$.

$$(P * Q)[k] = \sum_{m=-\infty}^{\infty} P[m]Q[k-m] = \sum_{m=-\infty}^{\infty} Q[m]P[k-m]$$

Notice that multiplication in one domain corresponds to convolution in the other domain: $p(x)q(x) \iff (P * Q)[k]$.

What if $x = e^{j\frac{2\pi}{N}n}$? Looks familiar . . .

$$p'[n] \triangleq p\big(e^{j\frac{2\pi}{N}n}\big) = P[0] + P[1]e^{j\frac{2\pi}{N}n} + P[2]e^{j2\frac{2\pi}{N}n} + \cdots$$

Lessons Learned

The **fast Fourier transform (FFT)** refers to a family of algorithms for efficiently computing the **DFT**.

FFT matrix factorization: Factor the $N \times N$ Fourier matrix $\boldsymbol{F}_N$ into $\log_2 N$ sparse matrices, where multiplying by each sparse matrix requires $\mathcal{O}(N)$ operations. The total cost is $\mathcal{O}(N \log_2 N)$.

$$\boldsymbol{F}_N = \boldsymbol{F}_{\log_2 N} \cdots \boldsymbol{F}_2 \boldsymbol{F}_1 \boldsymbol{P}_N$$

- $\boldsymbol{F}_{\log_2 N}$, . . . , $\boldsymbol{F}_2$, $\boldsymbol{F}_1$ are sparse matrices.
- $\boldsymbol{P}_N$ is a permutation matrix.

Circulant matrices have a nice eigenvalue decomposition. FFT algorithms let us compute $\boldsymbol{y} = (\boldsymbol{x} \circledast \boldsymbol{h})$ in $\mathcal{O}(N \log_2 N)$ operations, rather than $\mathcal{O}(N^2)$ operations!

Question of the Day

Directly computing the DFT takes $\mathcal{O}(N^2)$ operations. An FFT reduces that to $\mathcal{O}(N \log_2 N)$ operations. Explain how to count the operations to get these totals.