6.003: Signal Processing

Fast Fourier Transform

December 2, 2021

Computation Speed and the FFT

For the past few weeks we have been using the FFT because it is much faster than direct-form computation of the DFT.

Why is the FFT fast? How fast is it?

Computing the DFT

Direct-form computation of DFT in Python.

$$F[k] = \frac{1}{N}\sum_{n=0}^{N-1} f[n]e^{-j\frac{2\pi kn}{N}}$$

Simple Python implementation:

```
from math import e,pi
def DFT(f):
    N = len(f)
    F = []
    for k in range(N):
        ans = 0
        for n in range(N):
            ans += f[n]*e**(-2j*pi*k*n/N)/N
        F.append(ans)
    return F
```

For each pair of values n and k:

- evaluate the complex exponential,
- multiply by $f[n]$, and
- sum across n for each k

N^2 multiplications of `f[n]*e**(-2j*pi*k*n/N)/N`. **Can this be reduced?**

Computing the DFT

Complex exponentials of the form $e^{j\theta}$ are periodic in θ with period 2π.
Therefore $e^{-j2\pi kn/N}$ aliases to $e^{-j2\pi((kn) \bmod N)/N}$.
Only N unique values of complex exponentials are needed.
$\rightarrow$ precompute these values.

```
from math import e,pi
def DFTprecompute(f):
    N = len(f)
    bases = [e**(-2j*pi*k/N)/N for k in range(N)]
    F = []
    for k in range(N):
        ans = 0
        for n in range(N):
            ans += f[n]*bases[k*n%N]
        F.append(ans)
    return F
```

Run-time is reduced by a factor of 2.85 by precomputing basis functions:

N	direct form	precomputing bases
5120	10.47s	3.67s
10240	43.86s	15.37s

Run-time still $\uparrow$ **quadratically** with N: doubling N quadruples run-time.

Computing the DFT

We are often interested in computing the DFT for a real-valued input $f[n]$.

```
from math import e,pi
def DFTofReal(f):
    N = len(f)
    bases = [e**(-2j*pi*k/N)/N for k in range(N)]
    F = []
    for k in range(N):
        ans = 0
        for n in range(N):
            ans += f[n]*bases[k*n%N]
        F.append(ans)
    return F
```

Can we reduce the operation count by assuming $f[n]$ is real-valued?

Calculating DFTs for Real-Valued Inputs

If $f[n]$ is **real**, then $F[k]$ is **conjugate symmetric**: $F[-k] = F^*[k]$

Method: Compute $F[k]$ for $0 \le k \le N/2$, then fill in $F[-k]$ as $F^*[k]$.

```
from math import e,pi
def DFTofReal(f):
    N = len(f)
    bases = [e**(-2j*pi*n/N)/N for n in range(N)]
    F = []
    for k in range(N//2+1):
        ans = 0
        for n in range(N):
            ans += f[n]*bases[k*n%N]
        F.append(ans)
    for k in range(-N//2+1,0):
        F.append(F[-k].conjugate())
    return F
```

N	direct form	precomputing bases	DFTofReal
5120	10.47s	3.67s	1.75s
10240	43.86s	15.37s	7.25s

DFTofReal needs half as many k's and halves the number of calculations.

These two manipulations reduce operation count by a factor > 6.

Very worth while, but the FFT is **much** better.

Historical Perspective

Both elegant and useful, the FFT algorithm is arguably the most important algorithm in modern signal processing.

Modern interest stems most directly from James Cooley (IBM) and John Tukey (Princeton): "An Algorithm for the Machine Calculation of Complex Fourier Series," published in *Mathematics of Computation* 19: 297-301 (1965).

However there were a number previous, independent discoveries, including Danielson and Lanczos (1942), Runge and König (1924), and most significantly work by Gauss (1805).[1]

[1] http://nonagon.org/ExLibris/gauss-fast-fourier-transform

Gauss

Gauss used the basic idea behind the FFT algorithm in his study of the orbit of the then recently discovered asteroid Pallas. The manuscript was written circa 1805 and published posthumously in 1866.

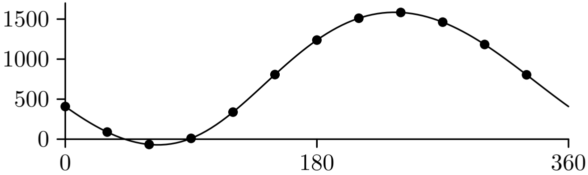
Gauss' data: "declination" X (minutes of arc) v. "ascension" θ (degrees)[2]

θ:	0	30	60	90	120	150	180	210	240	270	300	330
X:	408	89	−66	10	338	807	1238	1511	1583	1462	1183	804

Fitting function:

$$X = f(\theta) = a_0 + \sum_{k=1}^{5} \left[a_k \cos\left(\frac{2\pi k\theta}{360}\right) + b_k \sin\left(\frac{2\pi k\theta}{360}\right)\right] + a_6 \cos\left(\frac{12\pi\theta}{360}\right)$$

Resulting fit:



[2] B. Osgood, "The Fourier Transform and its Applications"

Historical Perspective

Both elegant and useful, the FFT algorithm is arguably the most important algorithm in modern signal processing.

Modern interest stems most directly from James Cooley (IBM) and John Tukey (Princeton): "An Algorithm for the Machine Calculation of Complex Fourier Series," published in *Mathematics of Computation* 19: 297-301 (1965).

However there were a number previous, independent discoveries, including Danielson and Lanczos (1942), Runge and König (1924), and most significantly work by Gauss (1805).[3]

While you might imagine that Gauss was interested in minimizing computation (since it was done by hand), he was more interested in understanding the inherent symmetries and using those to generate a robust solution.

[3] http://nonagon.org/ExLibris/gauss-fast-fourier-transform

FFT Algorithm

Start with the definition of an N-point DFT, where N is an even number.

$$
\begin{aligned}
F[k] &= \frac{1}{N}\sum_{n=0}^{N-1} f[n]e^{-j\frac{2\pi kn}{N}} \\
&= \frac{1}{N}\sum_{\substack{n=0\\ n\text{ even}}}^{N-1} f[n]e^{-j\frac{2\pi kn}{N}} + \frac{1}{N}\sum_{\substack{n=0\\ n\text{ odd}}}^{N-1} f[n]e^{-j\frac{2\pi kn}{N}} \\
&= \frac{1}{N}\sum_{m=0}^{N/2-1} f[2m]e^{-j\frac{2\pi k(2m)}{N}} + \frac{1}{N}\sum_{m=0}^{N/2-1} f[2m+1]e^{-j\frac{2\pi k(2m+1)}{N}} \\
&= \frac{1}{2}\underbrace{\frac{1}{N/2}\sum_{m=0}^{N/2-1} f[2m]e^{-j\frac{2\pi km}{N/2}}}_{\text{DFT of even numbered inputs}} + \frac{1}{2}e^{-j\frac{2\pi k}{N}}\underbrace{\frac{1}{N/2}\sum_{m=0}^{N/2-1} f[2m+1]e^{-j\frac{2\pi km}{N/2}}}_{\text{DFT of odd numbered inputs}}
\end{aligned}
$$

This refactorization reduces an N-point DFT to two $N/2$-point DFTs.

Is that good?

FFT Algorithm

Start with the definition of an N-point DFT, where N is an even number.

$$
\begin{aligned}
F[k] &= \frac{1}{N}\sum_{n=0}^{N-1} f[n]e^{-j\frac{2\pi kn}{N}} \\
&= \frac{1}{N}\sum_{\substack{n=0\\ n \text{ even}}}^{N-1} f[n]e^{-j\frac{2\pi kn}{N}} + \frac{1}{N}\sum_{\substack{n=0\\ n \text{ odd}}}^{N-1} f[n]e^{-j\frac{2\pi kn}{N}} \\
&= \frac{1}{N}\sum_{m=0}^{N/2-1} f[2m]e^{-j\frac{2\pi k(2m)}{N}} + \frac{1}{N}\sum_{m=0}^{N/2-1} f[2m+1]e^{-j\frac{2\pi k(2m+1)}{N}} \\
&= \frac{1}{2}\underbrace{\frac{1}{N/2}\sum_{m=0}^{N/2-1} f[2m]e^{-j\frac{2\pi km}{N/2}}}_{\text{DFT of even numbered inputs}} + \frac{1}{2}e^{-j\frac{2\pi k}{N}}\underbrace{\frac{1}{N/2}\sum_{m=0}^{N/2-1} f[2m+1]e^{-j\frac{2\pi km}{N/2}}}_{\text{DFT of odd numbered inputs}}
\end{aligned}
$$

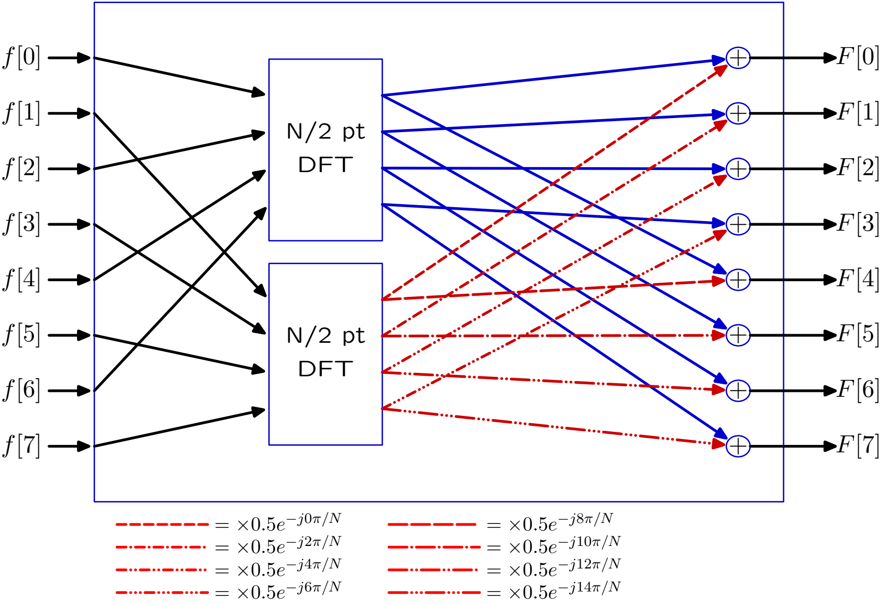
This refactorization reduces an N-point DFT to two $N/2$-point DFTs.

$N^2 \to 2\left(\frac{N}{2}\right)^2 = \frac{1}{2}N^2$

plus the overhead needed to "glue" the two halves back together.

Data Paths

Glue: blue lines represent $\times 0.5$; red lines represent $\times 0.5 e^{-j2\pi k/N}$.



Precompute glue factors, then the glue is N multiply-and-add steps.

Recursive Application

Start with an N-point DFT where $N = 2^m$ for an integer $m = \log_2(N)$.

One N-point DFT $\rightarrow$ two $N/2$-point DFTs plus glue.
Two $N/2$-point DFTs $\rightarrow$ four $N/4$-point DFTs plus glue.
Four $N/4$-point DFTs $\rightarrow$ eight $N/8$-point DFTs plus glue.
Eight $N/8$-point DFTs $\rightarrow$ sixteen $N/16$-point DFTs plus glue.
...
$N/2$ 2-point DFTs $\rightarrow$ N 1-point DFTs plus glue.

Taking a 1-point DFT requires no computations: $F[0] = \sum_{n=0}^{0} f[0] = f[0]$.

The glue includes N multiply-and-add's at each step.
Total number of multiply-and-add's = $m * N = N \log_2(N)$.

FFT Speedup

Not such a big deal for Gauss.

θ:	0	30	60	90	120	150	180	210	240	270	300	330
X:	408	89	-66	10	338	807	1238	1511	1583	1462	1183	804

Fitting 12 variables to 12 equations:

Speedup would be $\frac{12\times 12}{12\times \log_2(12)} \approx 3.3$.

Gauss' motivation was to reduce complexity.

FFT Speedup

Makes an enormous difference for modern signal processing tasks.

Consider processing 1080p video images (1920×1080) pixels.

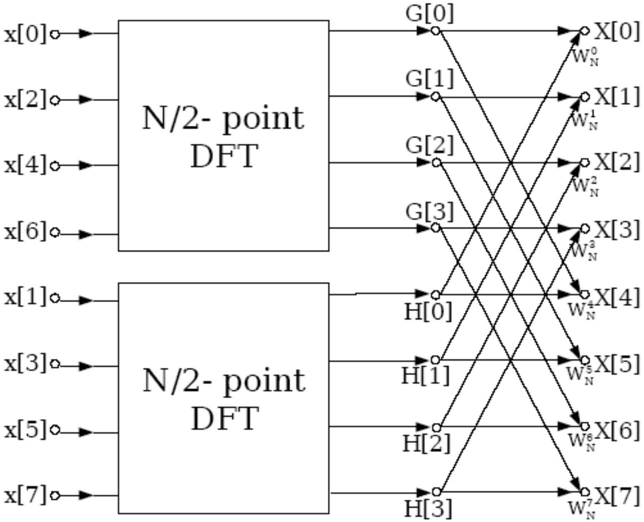
Computing a 2D DFT requires $O(2N\times N^2)$ multiplies.
Using the FFT reduces this to $O(2N\times N\log_2(N))$: faster by $\approx 175\times$.

FFT reduces times from ≈ 3 hours to about a minute (on my laptop)!

Operation Counts for Direct-Form DFT and FFT algorithms

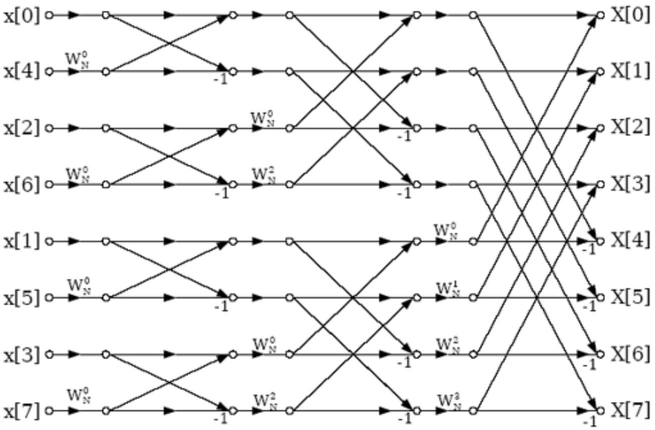
N	DFT	FFT	speed-up
2	4	2	2.0
4	16	8	2.0
8	64	24	2.7
16	256	64	4.0
32	1,024	160	6.4
64	4,096	384	10.7
128	16,384	896	18.3
256	65,536	2,048	32.0
512	262,144	4,608	56.9
1,024	1,048,576	10,240	102.4
2,048	4,194,304	22,528	186.2
4,096	16,777,216	49,152	341.3
8,192	67,108,864	106,496	630.2
16,384	268,435,456	229,376	1,170.3

Decimation in Time

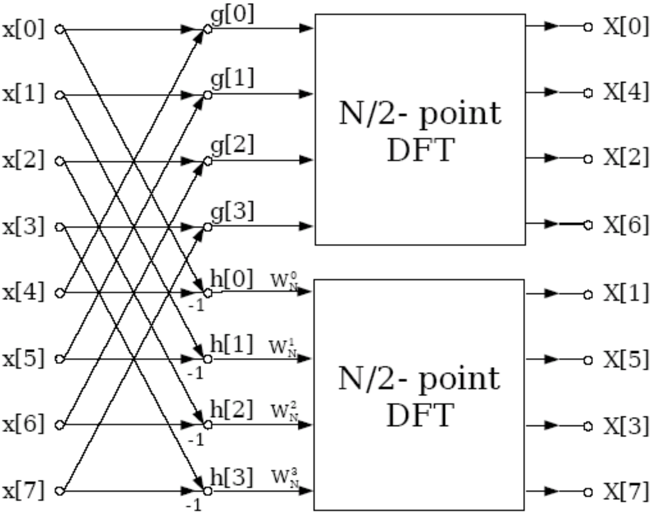


https://cnx.org/contents/qAa9OhlP@2.44:zmcmahhR@7/Decimation-in-time-DIT-Radix-2-FFT

Decimation in Time

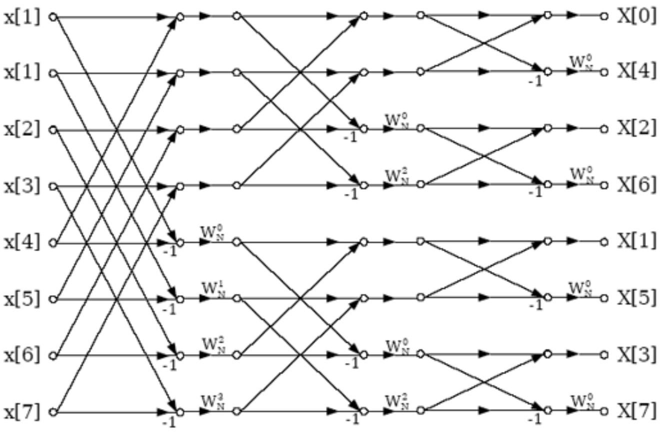


Decimation in Frequency



https://cnx.org/contents/qAa9OhlP@2.44:XaYDVUAS@6/Decimation-in-Frequency-DIF-Radix-2-FFT

Decimation in Frequency

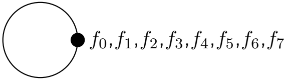


"A Visual Way to Teach the Fast Fourier Transform"[4]

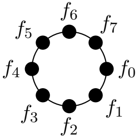
Represent each term in the definition of the DFT as a coefficient f_n times a point on the unit circle.

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n \, e^{-j \frac{2\pi}{N} kn}$$

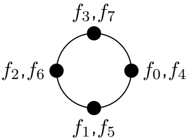
$NF_0 =$



$NF_1 =$



$NF_2 =$

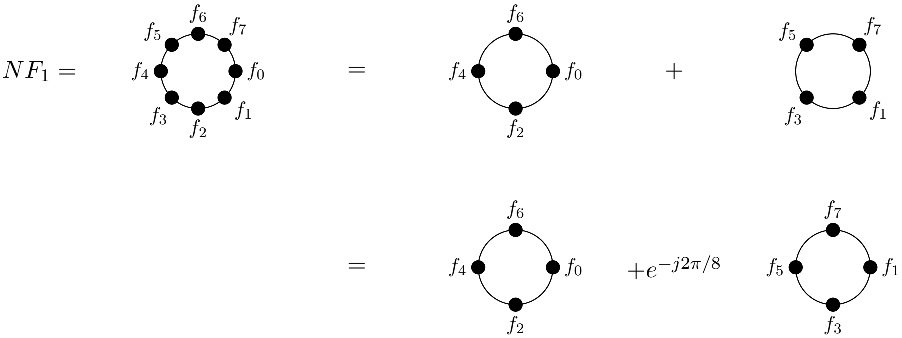


[4] Jithin D. George, SIAM News, November 1, 2018

“A Visual Way to Teach the Fast Fourier Transform”

Each term can be written as the sum of terms of lower order.

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n \, e^{-j \frac{2\pi}{N} kn}$$

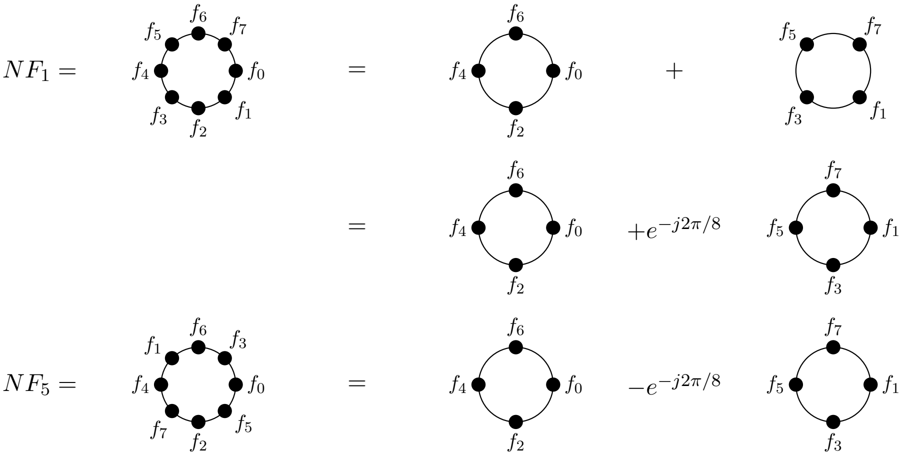


The odd terms are just a rotation away from the even terms!

“A Visual Way to Teach the Fast Fourier Transform”

The lower order terms can be reused (by subtracting instead of adding.

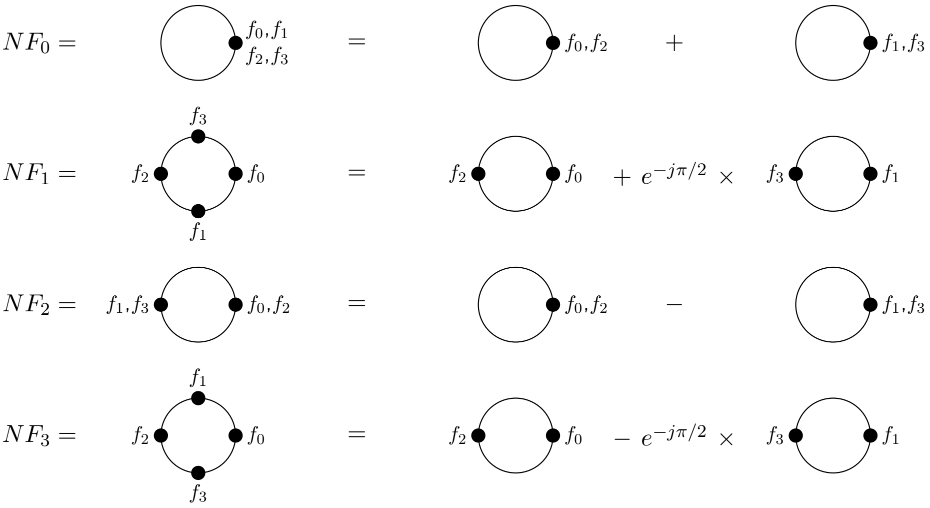
$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n \, e^{-j\frac{2\pi}{N}kn}$$



“A Visual Way to Teach the Fast Fourier Transform”

We can write the N-point DFT as a sum of two $N/2$-point DFTs plus glue.

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n \, e^{-j \frac{2\pi}{N} kn}$$



The FFT as a Polynomial Representation[5]

Think about the DFT

$$F[k] = \frac{1}{N}\sum_{n=0}^{N-1} f[n]e^{-j\frac{2\pi kn}{N}}$$

as a polynomial representation problem.

The polynomial coefficients are $f[n]$. The polynomial maps those coefficients onto the complex plane, and the values that result at the points $e^{-j2\pi/N}$ describe the frequency representation.

Computing the DFT and iDFT are then equivalent to finding alternative representations for a polynomial, as coefficients $f[n]$ or as values in the complex plane as $F[k]$.

[5] `https://www.youtube.com/watch?v=h7ap07q16V0` and Prof. Erik Demaine in 6.046 `https://www.youtube.com/watch?v=iTMn0Kt18tg`