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:

```python
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 \( f[n] e^{\frac{-2j\pi k n}{N}} / N \). **Can this be reduced?**
Computing the DFT

Complex exponentials of the form $e^{j\theta}$ are periodic in $\theta$ with period $2\pi$. Therefore $e^{-j2\pi kn/N}$ aliases to $e^{-j2\pi((kn) \mod N)/N}$.

Only $N$ unique values of complex exponentials are needed. → precompute these values.

```python
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:

<table>
<thead>
<tr>
<th>$N$</th>
<th>direct form</th>
<th>precomputing bases</th>
</tr>
</thead>
<tbody>
<tr>
<td>5120</td>
<td>10.47s</td>
<td>3.67s</td>
</tr>
<tr>
<td>10240</td>
<td>43.86s</td>
<td>15.37s</td>
</tr>
</tbody>
</table>

Run-time still ↑ **quadratically** with $N$: doubling $N$ quadruples run-time.
Computing the DFT

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

```python
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 $0 \leq k \leq N/2$, then fill in $F[-k]$ as $F^*[k]$.

```python
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
```

<table>
<thead>
<tr>
<th>$N$</th>
<th>direct form</th>
<th>precomputing bases</th>
<th>DFTofReal</th>
</tr>
</thead>
<tbody>
<tr>
<td>5120</td>
<td>10.47s</td>
<td>3.67s</td>
<td>1.75s</td>
</tr>
<tr>
<td>10240</td>
<td>43.86s</td>
<td>15.37s</td>
<td>7.25s</td>
</tr>
</tbody>
</table>

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.


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” $\theta$ (degrees)$^2$

\[
\begin{array}{ccccccccccccc}
\theta: & 0 & 30 & 60 & 90 & 120 & 150 & 180 & 210 & 240 & 270 & 300 & 330 \\
X: & 408 & 89 & -66 & 10 & 338 & 807 & 1238 & 1511 & 1583 & 1462 & 1183 & 804 \\
\end{array}
\]

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.


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](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.

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

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

$$= \frac{1}{N} \sum_{m=0}^{N/2-1} f[2m] e^{-j\frac{2\pi k(2m)}{N}} + \frac{1}{N} \sum_{m=0}^{N/2-1} f[2m+1] e^{-j\frac{2\pi k(2m+1)}{N}}$$

$$= \frac{1}{2} \frac{1}{N/2} \sum_{m=0}^{N/2-1} f[2m] e^{-j\frac{2\pi km}{N/2}} + \frac{1}{2} e^{-j\frac{2\pi k}{N}} \frac{1}{N/2} \sum_{m=0}^{N/2-1} f[2m+1] e^{-j\frac{2\pi km}{N/2}}$$

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.

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

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

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

\[
= \frac{1}{2} \frac{1}{N/2} \sum_{m=0}^{N/2-1} f[2m] e^{-j \frac{2\pi k m}{N/2}} + \frac{1}{2} e^{-j \frac{2\pi k}{N}} \frac{1}{N/2} \sum_{m=0}^{N/2-1} f[2m+1] e^{-j \frac{2\pi k m}{N/2}}
\]

DFT of even numbered inputs

DFT of odd numbered inputs

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

\[
N^2 \rightarrow 2 \left( \frac{N}{2} \right)^2 = \frac{1}{2} N^2
\]

plus the overhead needed to “glue” the two halves back together.
Data Paths

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

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.

\[ \ldots \]

$N/2$ 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 $= m \times N = N \log_2(N)$. 
FFT Speedup

Not such a big deal for Gauss.

\[ \theta: \quad 0 \quad 30 \quad 60 \quad 90 \quad 120 \quad 150 \quad 180 \quad 210 \quad 240 \quad 270 \quad 300 \quad 330 \]

\[ X: \quad 408 \quad 89 \quad -66 \quad 10 \quad 338 \quad 807 \quad 1238 \quad 1511 \quad 1583 \quad 1462 \quad 1183 \quad 804 \]

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 $\approx 3$ hours to about a minute (on my laptop)!
### Operation Counts for Direct-Form DFT and FFT algorithms

<table>
<thead>
<tr>
<th>N</th>
<th>DFT</th>
<th>FFT</th>
<th>speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>8</td>
<td>2.0</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>24</td>
<td>2.7</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>64</td>
<td>4.0</td>
</tr>
<tr>
<td>32</td>
<td>1,024</td>
<td>160</td>
<td>6.4</td>
</tr>
<tr>
<td>64</td>
<td>4,096</td>
<td>384</td>
<td>10.7</td>
</tr>
<tr>
<td>128</td>
<td>16,384</td>
<td>896</td>
<td>18.3</td>
</tr>
<tr>
<td>256</td>
<td>65,536</td>
<td>2,048</td>
<td>32.0</td>
</tr>
<tr>
<td>512</td>
<td>262,144</td>
<td>4,608</td>
<td>56.9</td>
</tr>
<tr>
<td>1,024</td>
<td>1,048,576</td>
<td>10,240</td>
<td>102.4</td>
</tr>
<tr>
<td>2,048</td>
<td>4,194,304</td>
<td>22,528</td>
<td>186.2</td>
</tr>
<tr>
<td>4,096</td>
<td>16,777,216</td>
<td>49,152</td>
<td>341.3</td>
</tr>
<tr>
<td>8,192</td>
<td>67,108,864</td>
<td>106,496</td>
<td>630.2</td>
</tr>
<tr>
<td>16,384</td>
<td>268,435,456</td>
<td>229,376</td>
<td>1,170.3</td>
</tr>
</tbody>
</table>
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
Represent each term in the definition of the DFT as a coefficient $f_n$ times a point on the unit circle.

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

**$NF_0 =$**

![Diagram of $NF_0$]

**$NF_1 =$**

![Diagram of $NF_1$]

**$NF_2 =$**

![Diagram of $NF_2$]
Each term can be written as the sum of terms of lower order.

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

\[ NF_1 = f_0 \quad f_1 \quad f_2 \quad f_3 \quad f_4 \quad f_5 \quad f_6 \quad f_7 = f_0 \quad f_2 \quad f_4 \quad f_6 + e^{-j \frac{2\pi}{8}} f_1 \quad f_3 \quad f_5 \quad f_7 \]

The odd terms are just a rotation away from the even terms!
The lower order terms can be reused (by subtracting instead of adding).

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

\[ NF_1 = \]

\[ NF_5 = \]
We can write the $N$-point DFT as a sum of two $N/2$-point DFTs plus glue.

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

\[
NF_0 = f_0, f_1
f_2, f_3
= f_0, f_2 + f_1, f_3
\]

\[
NF_1 = f_2 f_0
f_1 f_3
= f_2 f_0 + e^{-j\pi/2} \times f_3 f_1
\]

\[
NF_2 = f_1, f_3 f_0, f_2
= f_0, f_2 - f_1, f_3
\]

\[
NF_3 = f_2 f_0
f_3 f_1
= f_2 f_0 - e^{-j\pi/2} \times f_3 f_1
\]
The FFT as a Polynomial Representation

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 \( e^{-j \frac{2\pi}{N}} \) 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 \( F[k] \).

---

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