6.3000: Signal Processing

Fast Fourier Transform

Quiz 2: November 7, 2-4pm, room 50-340 (Walker).

- Closed book except for two pages of notes (8.5"x11" four sides total).
- No electronic devices. (No headphones, cellphones, calculators, ...)
- Coverage up to and including classes on November 2 and HW 8.

We have posted a practice quiz as a study aid for the upcoming quiz 2.

- Your solutions will not be submitted or counted in your grade.
- Solutions will be posted on Friday.

There is no HW 9.

If you have personal or medical difficulties, please contact S^3 and/or `6.3000-instructors@mit.edu` for accommodations.

We will hold quiz reviews during regular class hours on Thursday.

October 31, 2023

Fast Fourier Transform

The Fast-Fourier Transform (FFT) is an algorithm (actually a family of algorithms) for computing the Discrete Fourier Transform (DFT).

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

– **widely used** in engineering and science
– **elegant mathematics** (as alternative representations for polynomials)
– **elegant computer science** (divide-and-conquer)

It's also interesting from an historical perspective.

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

Historical Perspective

Gauss used the basic idea behind the FFT algorithm in his study of the orbit of the then recently discovered asteroid Pallas.

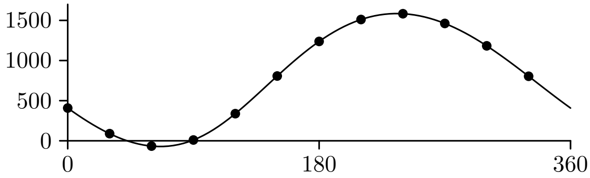
Gauss' data: "declination" X (minutes of arc) v. "ascension" θ (degrees)[2]

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

Gauss used the basic idea behind the FFT algorithm in his study of the orbit of the then recently discovered asteroid Pallas.

Gauss' data: "declination" X (minutes of arc) v. "ascension" θ (degrees)[3]

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 coefficients:

k :	0	1	2	3	4	5	6
a_k :	780.6	-411.0	43.4	-4.3	-1.1	0.3	0.1
b_k :	–	-720.2	-2.2	5.5	-1.0	-0.3	–

3 B. Osgood, "The Fourier Transform and its Applications"

Historical Perspective

In this work, Gauss introduced least-squares curve fitting and efficient computation of Fourier coefficients.

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

Gauss did not even publish the algorithm. The manuscript was written circa 1805 and published posthumously in 1866.

FFT: Divide-and-Conquer

One of the most important features of the FFT algorithm is its modularity at successive scales – what we now call **divide-and-conquer**.

Why is divide-and-conquer good?

FFT: Divide-and-Conquer

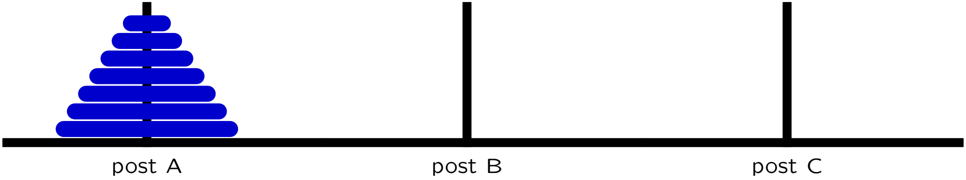
One of the most important features of the FFT algorithm is its modularity at successive scales – what we now call **divide-and-conquer**.

Why is divide-and-conquer good?

- break a problem into sub-problems
  - can lead to expressive algorithms: simple and elegant
- break a problem into sub-problems
  - can speed computations

Towers of Hanoi

Transfer a stack of disks from post A to post B by moving the disks one-at-a-time, without placing any disk on a smaller disk.




```
def Hanoi(n,A,B,C):
  if n==1:
       print 'move top of ' + A + ' to ' + B
  else:
       Hanoi(n-1,A,C,B)
       Hanoi(1,A,B,C)
       Hanoi(n-1,C,B,A)
```

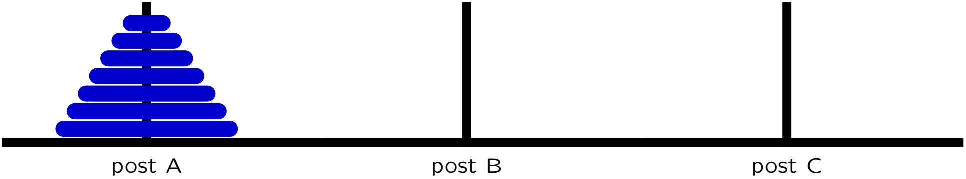
Towers of Hanoi

Details for a tower of height 3.

```
> Hanoi(3,'a','b','c')
entering Hanoi(3,a,b,c)
    entering Hanoi(2,a,c,b)
        entering Hanoi(1,a,b,c)
            move top of a to b
        exiting
        entering Hanoi(1,a,c,b)
            move top of a to c
        exiting
        entering Hanoi(1,b,c,a)
            move top of b to c
        exiting
    exiting
    entering Hanoi(1,a,b,c)
        move top of a to b
    exiting
    entering Hanoi(2,c,b,a)
        entering Hanoi(1,c,a,b)
            move top of c to a
        exiting
        entering Hanoi(1,c,b,a)
            move top of c to b
        exiting
        entering Hanoi(1,a,b,c)
            move top of a to b
        exiting
    exiting
exiting
```

Towers of Hanoi

Transfer a stack of disks from post A to post B by moving the disks one-at-a-time, without placing any disk on a smaller disk.




```
def Hanoi(n,A,B,C):
  if n==1:
       print 'move from ' + A + ' to ' + B
  else:
       Hanoi(n-1,A,C,B)
       Hanoi(1,A,B,C)
       Hanoi(n-1,C,B,A)
```

Breaking into sub-problems leads to an elegant, expressive algorithm.

The FFT shares this property.

Divide-and-Conquer: Efficiency

Dividing a problem into two sub-problems can speed the computation.

Example: original algorithm requires $O(N^2)$ operations.

Dividing in two halves yields two sub-problems that are each $O((\frac{N}{2})^2)$. Ignoring recombination costs, the total operation count is $O(N^2/2)$, half that of the original problem.

This is how the FFT achieves its speed.

Fast Fourier Transform

How fast is the FFT (relative to the DFT)?

Why is the FFT fast?

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 (naive) 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
```

How many operations are required by this algorithm if $N = 1024$?

1. less than 10,000
2. between 10,000 and 100,000
3. between 100,000 and 1,000,000
4. greater than 1,000,000

How does the number of operations **scale** with N?

Computing the DFT

How many operations are required to compute a DFT of length N?

```
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 n,k pair (of which there are N^2):

- compute the complex exponent (3 multiplies and a divide),
- raise e to the power of that exponent,
- multiply by $f[n]$ and divide by N, and
- add the result to the appropriate $F[k]$.

Total number is $1024 \times 1024 \times 8$: nearly 10 million!

The total number of operations scales as N^2.

Computing the DFT

How many operations are required to compute a DFT of length N?

```
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
```

Empirical results (for my laptop):

N	seconds
1024	0.41
2048	1.67
4096	6.70
8192	27.34

The test signal in lab 8 had $735,000$ samples:

Extrapolating to that length: $221,492$ seconds $=61$ hours (>2.5 days).

Computing the DFT

Much of the direct-form computation is in computing the **kernel functions**.

```
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
```

Computing the DFT

Much of the direct-form computation is in computing the **kernel functions**.

Complex exponentials $e^{j\theta}$ are periodic in θ with period 2π.

N unique values $\rightarrow$ **precompute** all of them!

```
from math import e,pi
def DFTprecompute(f):
    N = len(f)
    bases = [e**(-2j*pi*m/N)/N for m 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
```

N	direct (sec.)	pre-computing
1024	0.41	0.13
2048	1.67	0.54
4096	6.70	2.15
8192	27.34	9.01

Pre-computing kernel functions reduces run-time more than a **factor of 3**.

Check Yourself

The direct-form implementation of the DFT works not only for real-valued input signals but also for complex-valued input signals.

```
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
```

How could we change the algorithm to use fewer computations for **real-valued inputs**? How many fewer computations would then be required?

Check Yourself

The direct-form implementation of the DFT works not only for real-valued input signals but also for complex-valued input signals.

```
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
```

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

$$F[-k] = F^*[k]$$

We can compute $F[k]$ for $0 \leq k < N/2$ using the DFT algorithm and then set $F[-k] = F[N-k] = F^*[k]$ for the remaining values of k.

$\rightarrow$ approximately a **factor of 2 reduction** in operations

Computing the DFT

The optimizations that we have discussed so far reduce computation time by a (roughly) constant factor.

While reducing the computation of the test signal from lab 8 ($N = 735,000$) by a factor of 3 (for example) is good

$221,492\,\text{seconds} = 61\,\text{hours}$ ($> 2.5\,\text{days}$)

$\rightarrow 73,831\,\text{seconds} = 20\,\text{hours}$ (most of one day)

or by a factor of 6 is even better

$\rightarrow 36,916\,\text{seconds} = 10\,\text{hours}$

the resulting computation is still slow.

To reduce the number of computations more drastically, we need to reduce the order from $O(N^2)$ to a lower order – which is what the FFT algorithm does.

FFT Algorithm

Compute contributions of even and odd numbered input samples separately.

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

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

Is that good?

FFT Algorithm

Compute contributions of even and odd numbered input samples separately.

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

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

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

where the additional N comes from “gluing” the two halves together.

FFT Algorithm

Compute contributions of even and odd numbered input samples separately.

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

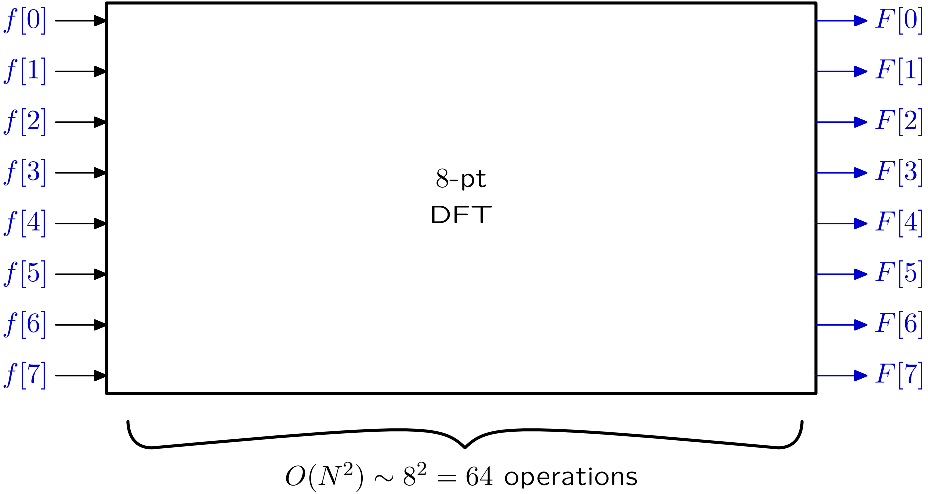
Reducing from N^2 to $\frac{1}{2}N^2$ is good – but it's only a factor of 2.

We have already seen several instances of reduction by a constant factor.

This reduction is different: it can be applied **recursively**.

Data Paths

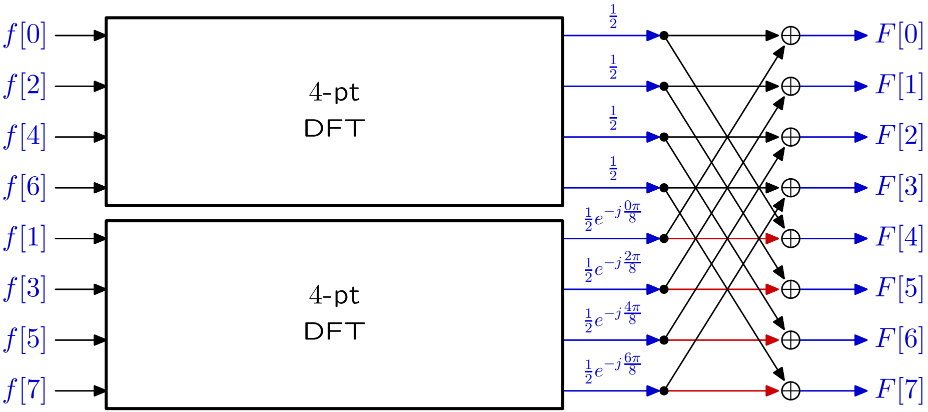
Draw **data paths** to help visualize the FFT algorithm.



Start with an 8-point DFT.

Data Paths

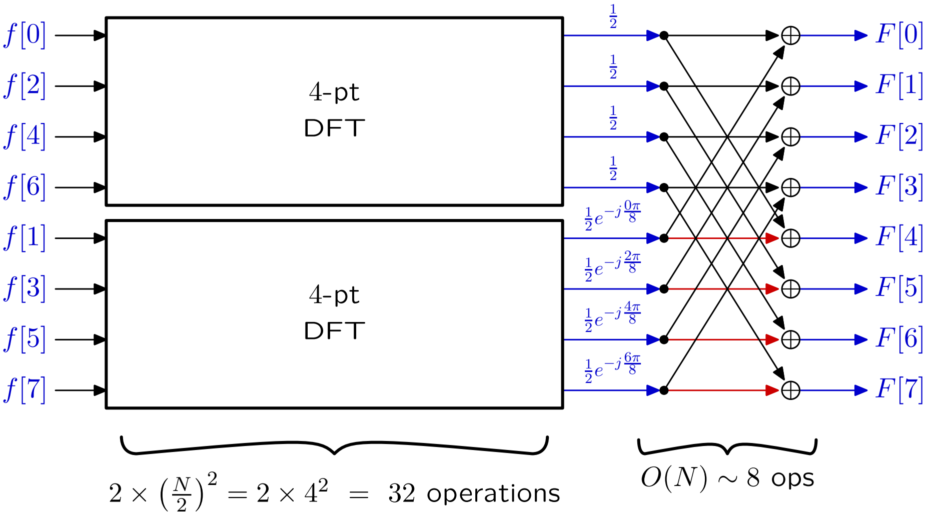
Write the 8-point DFT in terms of the DFTs of even and odd samples.



$$F[k] = \frac{1}{2} \underbrace{\frac{1}{N/2} \sum_{m=0}^{N/2-1} f[2m] e^{-j\frac{2\pi km}{N/2}}}_{\text{DFT of even numbered inputs}} + \frac{1}{2} e^{-j\frac{2\pi k}{N}} \underbrace{\frac{1}{N/2} \sum_{m=0}^{N/2-1} f[2m+1] e^{-j\frac{2\pi km}{N/2}}}_{\text{DFT of odd numbered inputs}}$$

Data Paths

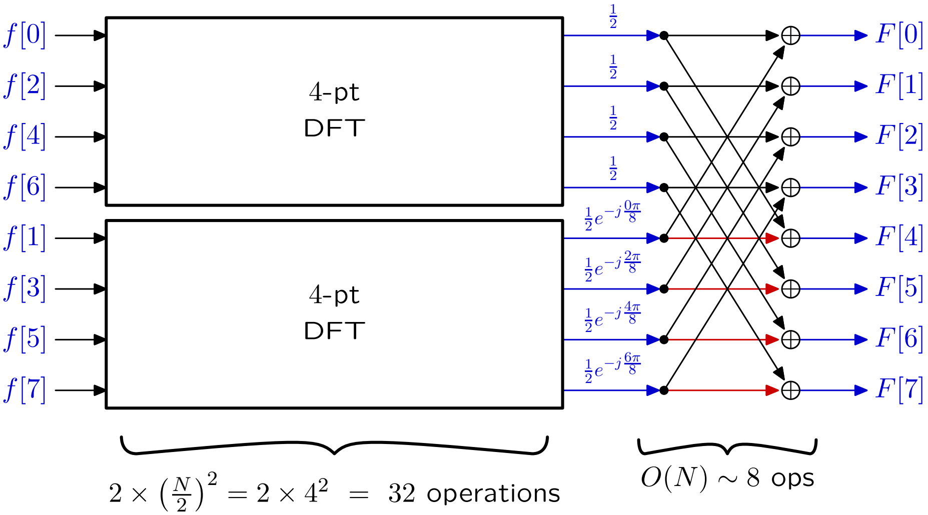
Write the 8-point DFT in terms of the DFTs of even and odd samples.



The numbers above the blue arrows represent multiplicative constants.
The red arrows represent multiplication by $e^{-j\pi} = -1$.

Data Paths

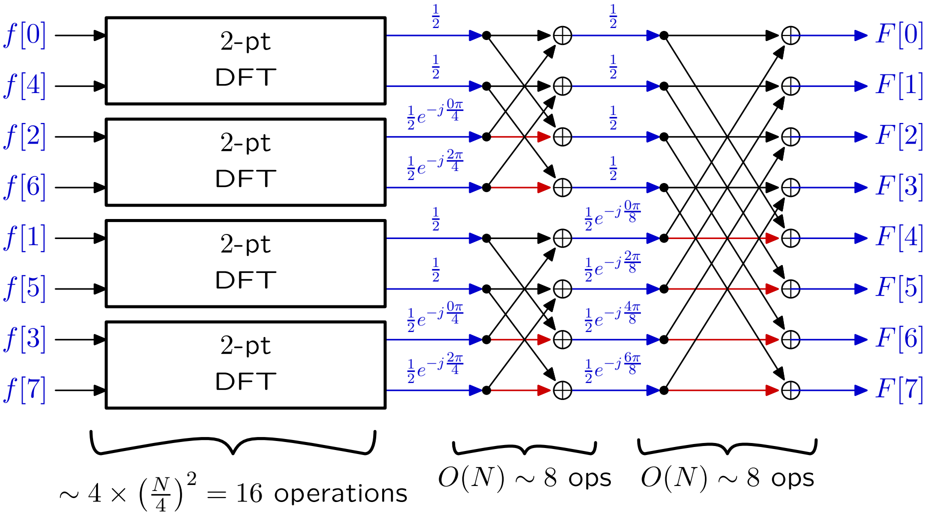
Write the 8-point DFT in terms of the DFTs of even and odd samples.



The number of operations to compute the DFTs is half that of the original. But we have $O(N)$ operations to combine the even and odd results.

Data Paths

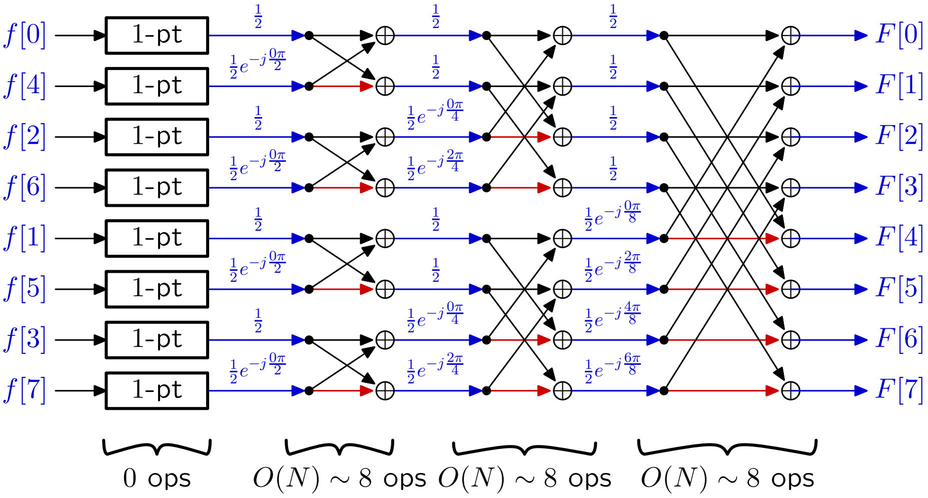
Write the 4-point DFTs in terms of 2-point DFTs.



The number of operations to compute the DFTs is one-fourth that of the original. But we have twice as many operations to combine the parts.

Data Paths

Write the 2-point DFTs in terms of 1-point DFTs.

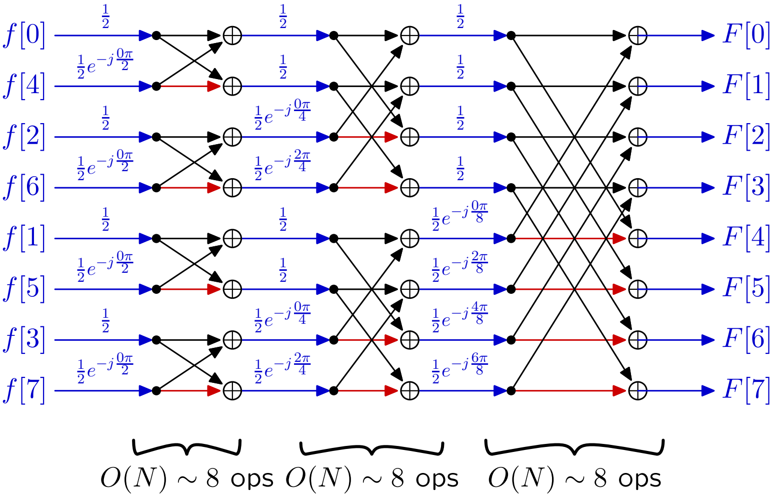


No operations are required to compute the 1-point DFTs.

But we have three times as many operations to combine the parts.

Data Paths

The FFT algorithm reduces the explicit DFTs to length 1.



All that remains to calculate is "glue". There are $\log_2(N)$ stages of glue and each is $O(N)$. So the algorithm is $N\log_2(N)$.

FFT Speedup

The speed of the FFT has had a profound impact on signal processing.

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	
32,768	1,073,741,824	491,520	2,184.5	
65,536	4,294,967,296	1,048,576	4,096.0	
131,072	17,179,869,184	2,228,224	7,710.1	
262,144	68,719,476,736	4,718,592	14,563.6	
524,288	274,877,906,944	9,961,472	27,594.1	
1,048,576	1,099,511,627,776	20,971,520	52,428.8	$\leftarrow$ lab 8

FFT Speedup

The speed of the FFT has had a profound impact on signal processing **especially multi-dimensional signal processing**!

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

Computing a 2D DFT requires

- one DFT per row + one DFT per column $\sim 2N$ DFTs
- each DFT requires $O(N^2)$ operations
- total is $O(2N \times N^2)$.

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)!

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
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
32,768	1,073,741,824	491,520	2,184.5
65,536	4,294,967,296	1,048,576	4,096.0
131,072	17,179,869,184	2,228,224	7,710.1
262,144	68,719,476,736	4,718,592	14,563.6
524,288	274,877,906,944	9,961,472	27,594.1
1,048,576	1,099,511,627,776	20,971,520	52,428.8

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.

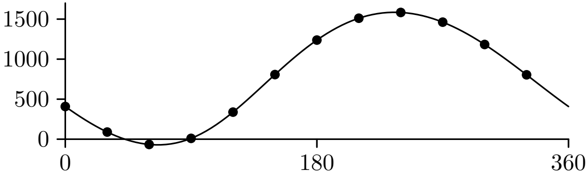
Gauss' data: "declination" X (minutes of arc) v. "ascension" θ (degrees)[4]

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:



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

Gauss

Gauss was more interested in understanding the inherent symmetries and using those to generate a robust solution.

Gauss fitted 12 variables to 12 equations.

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

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

Gauss was more interested in understanding than in operation count.

Python Code

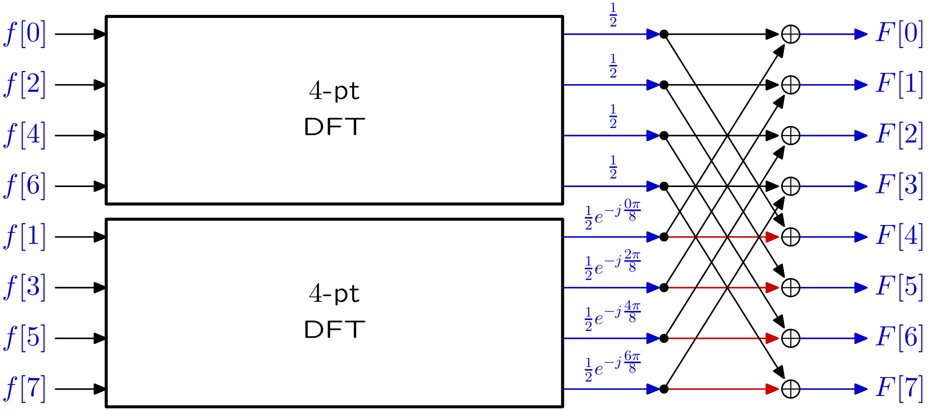
Consider the following code to implement the FFT algorithm.

```
from math import e,pi
def FFT(x):
    N = len(x)
    if N%2 != 0:
        print('N must be even')
        exit(1)
    if N==1:
        return x
    xe = x[::2]
    xo = x[1::2]
    Xe = FFT(xe)
    Xo = FFT(xo)
    X = []
    for k in range(N//2):
        X.append((Xe[k]+e**(-2j*pi*k/N)*Xo[k])/2)
    for k in range(N//2):
        X.append((Xe[k]-e**(-2j*pi*k/N)*Xo[k])/2)
    return X
```

This code implements the decimation-in-time algorithm.

Python Code

The code on the previous slide implements decimation-in-time (below).



$$F[k] = \frac{1}{2} \underbrace{\frac{1}{N/2}\sum_{m=0}^{N/2-1} f[2m]e^{-j\frac{2\pi km}{N/2}}}_{\text{DFT of even numbered inputs}} + \frac{1}{2}e^{-j\frac{2\pi k}{N}} \underbrace{\frac{1}{N/2}\sum_{m=0}^{N/2-1} f[2m+1]e^{-j\frac{2\pi km}{N/2}}}_{\text{DFT of odd numbered inputs}}$$

Python Code

Consider the following code to implement the FFT algorithm.

```
from math import e,pi
def FFT(x):
    N = len(x)
    if N%2 != 0:
        print('N must be even')
        exit(1)
    if N==1:
        return x
    xe = x[::2]
    xo = x[1::2]
    Xe = FFT(xe)
    Xo = FFT(xo)
    X = []
    for k in range(N//2):
        X.append((Xe[k]+e**(-2j*pi*k/N)*Xo[k])/2)
    for k in range(N//2):
        X.append((Xe[k]-e**(-2j*pi*k/N)*Xo[k])/2)
    return X
```

Why are there two `for` loops?

Could we substitute a single loop over all N values?

Python Code

Consider the following code to implement the FFT algorithm.

```
from math import e,pi
def FFT(x):
    N = len(x)
    if N%2 != 0:
        print('N must be even')
        exit(1)
    if N==1:
        return x
    xe = x[::2]
    xo = x[1::2]
    Xe = FFT(xe)
    Xo = FFT(xo)
    X = []
    for k in range(N//2):
        X.append((Xe[k]+e**(-2j*pi*k/N)*Xo[k])/2)
    for k in range(N//2):
        X.append((Xe[k]-e**(-2j*pi*k/N)*Xo[k])/2)
    return X
```

The lengths of the `Xe` and `Xo` lists are just $N/2$.

The first `for` loop implements the "glue" for the first half of the output, the second `for` loop implements the glue for the results for the second half.

Check Yourself

We can make minor changes to this FFT algorithm to compute the iDFT.

```
from math import e,pi
def FFT(x):
    N = len(x)
    if N%2 != 0:
        print('N must be even')
        exit(1)
    if N==1:
        return x
    xe = x[::2]
    xo = x[1::2]
    Xe = FFT(xe)
    Xo = FFT(xo)
    X = []
    for k in range(N//2):
        X.append((Xe[k]+e**(-2j*pi*k/N)*Xo[k])/2)
    for k in range(N//2):
        X.append((Xe[k]-e**(-2j*pi*k/N)*Xo[k])/2)
    return X
```

Determine the changes that are needed.

Check Yourself

We can make minor changes to this FFT algorithm to compute the iDFT.

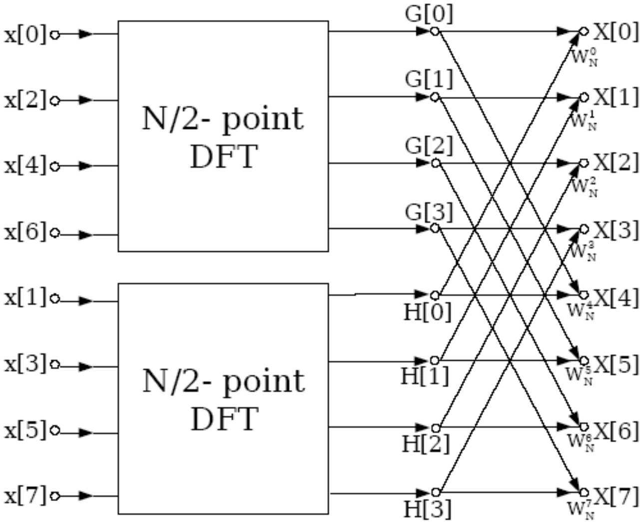
```
from math import e,pi
def FFT(x):
    N = len(x)
    if N%2 != 0:
        print('N must be even')
        exit(1)
    if N==1:
        return x
    xe = x[::2]
    xo = x[1::2]
    Xe = FFT(xe)
    Xo = FFT(xo)
    X = []
    for k in range(N//2):
        X.append((Xe[k]+e**(-2j*pi*k/N)*Xo[k])/2) --> X.append((Xe[k]+e**(2j*pi*k/N)*Xo[k]))
    for k in range(N//2):
        X.append((Xe[k]-e**(-2j*pi*k/N)*Xo[k])/2) --> X.append((Xe[k]-e**(2j*pi*k/N)*Xo[k]))
    return X
```

1. negate the complex exponents
2. remove the divisions by 2

Decimation in Time

There are many different "FFT" algorithms.

We have been looking at a "decimation in time" algorithm.

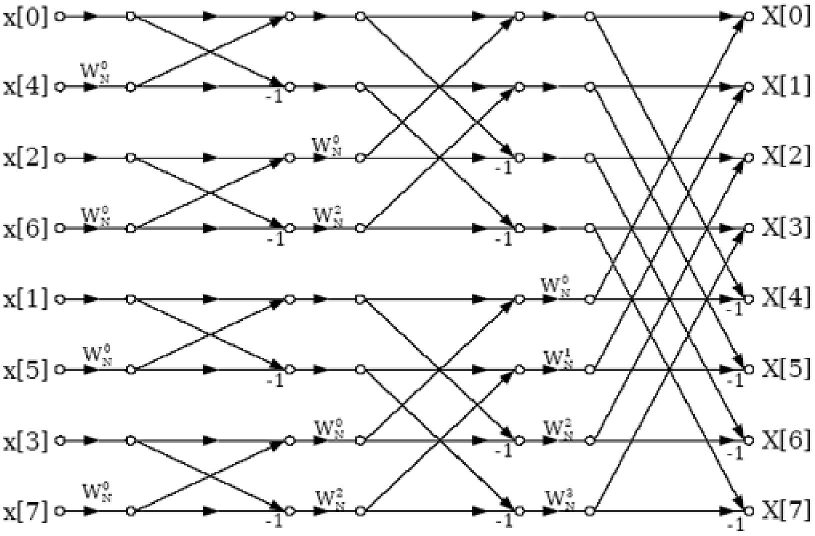


https://cnx.org/contents/qAa9OhlP@2.44:zmcmahhR@7/Decimation-in-time-DIT-Radix-2-FFT

Decimation in Time

There are many different "FFT" algorithms.

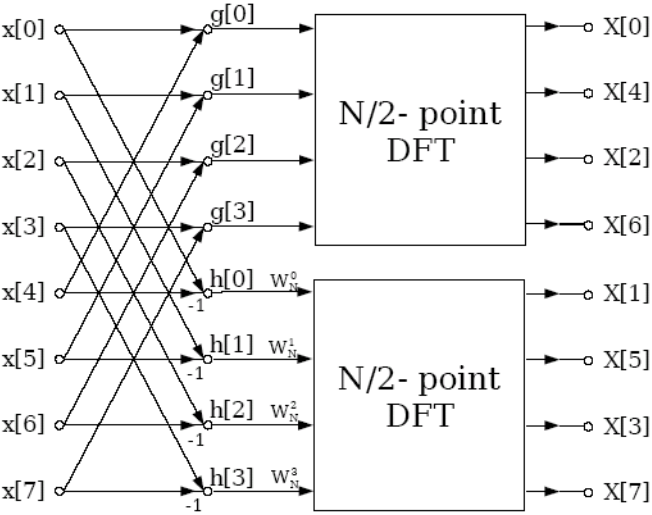
After recursive application of "decimation in time."



Decimation in Frequency

There are many different "FFT" algorithms.

Here is a "decimation in frequency" algorithm.

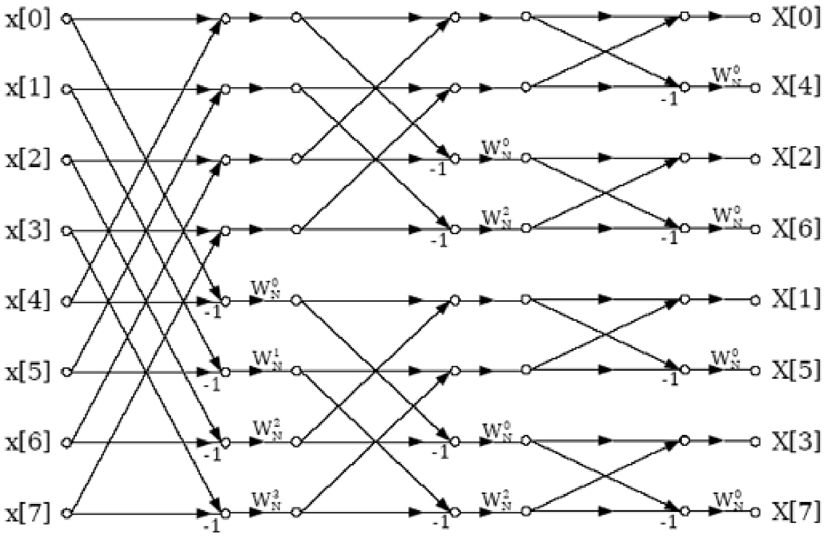


https://cnx.org/contents/qAa9OhlP@2.44:XaYDVUAS@6/Decimation-in-Frequency-DIF-Radix-2-FFT

Decimation in Frequency

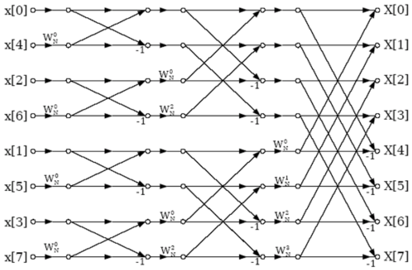
There are many different "FFT" algorithms.

After recursive application of "decimation in frequency."

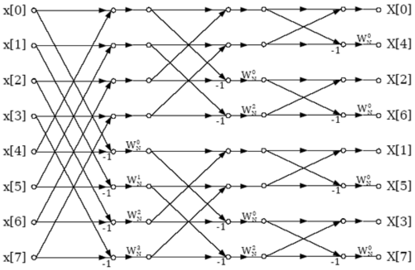


Scrambled Inputs / Outputs

Decimation in time: inputs are provided in a "scrambled" order.



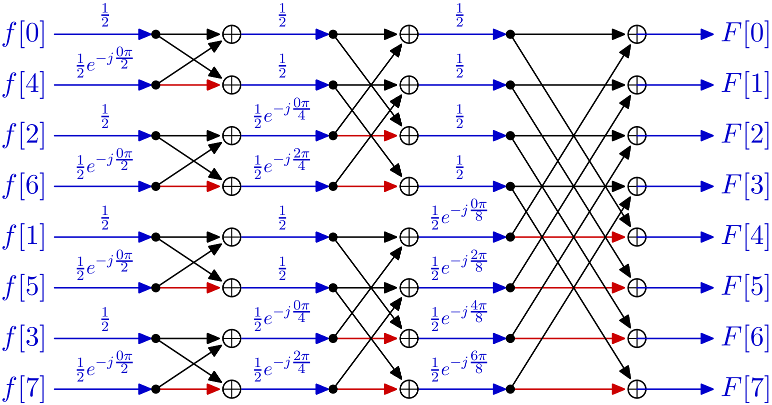
Decimation in frequency: outputs are generated in a "scrambled" order.



What is the resulting pattern?

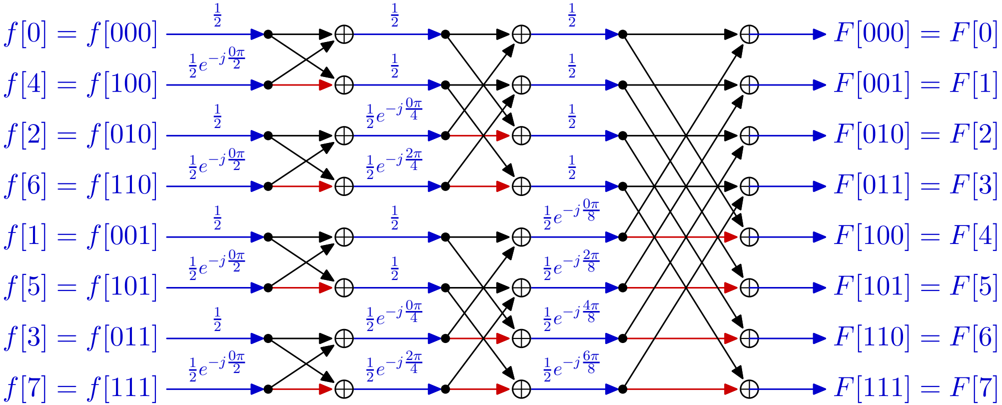
Scrambled Inputs

Decimation in time.



Scrambled Inputs

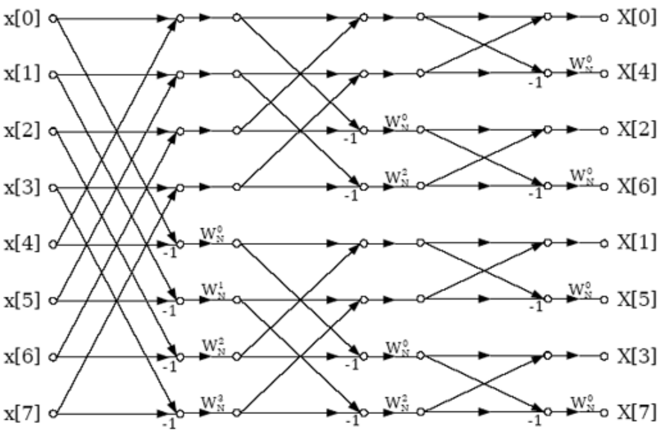
Decimation in time.



The input samples are in **bit-reversed** order.

Scrambled Outputs

Decimation in frequency is similar.



But now the output order is bit-reversed.

Other FFT Algorithms

A variety of other FFT algorithms have been developed to optimize computation.

- to avoid bit-reversal
- in-place algorithms
- generalizing for lengths N not equal to a power of 2.

FFTs With Other Radices

What if N is not a power of 2?

Factor N, and apply an algorithm tailored to each factor.

Example: radix 3

$$
\begin{aligned}
F[k] &= \frac{1}{N}\sum_{n=0}^{N-1} f[n]e^{-j\frac{2\pi kn}{N}} \\
&= \frac{1}{N}\sum_{m=0}^{N/3-1} f[3m]e^{-j\frac{2\pi k(3m)}{N}} + \frac{1}{N}\sum_{m=0}^{N/3-1} f[3m+1]e^{-j\frac{2\pi k(3m+1)}{N}} + \frac{1}{N}\sum_{m=0}^{N/3-1} f[3m+2]e^{-j\frac{2\pi k(3m+2)}{N}} \\
&= \frac{1}{3}\frac{1}{N/3}\sum_{m=0}^{N/3-1} f[3m]e^{-j\frac{2\pi km}{N/3}} \\
&\qquad + \frac{1}{3}\frac{1}{N/3}e^{-j2\pi k/N}\sum_{m=0}^{N/3-1} f[3m+1]e^{-j\frac{2\pi km}{N/3}} \\
&\qquad\qquad + \frac{1}{3}\frac{1}{N/3}e^{-j4\pi k/N}\sum_{m=0}^{N/3-1} f[3m+2]e^{-j\frac{2\pi km}{N/3}} \\
&= \frac{1}{3}\text{DFT(block 0)} + \frac{1}{3}e^{-j\frac{2\pi k}{N}}\text{DFT(block 1)} + \frac{1}{3}e^{-j\frac{4\pi k}{N}}\text{DFT(block 2)}
\end{aligned}
$$

Fast Fourier Transform

The Fast-Fourier Transform (FFT) is an algorithm (actually a family of algorithms) for computing the Discrete Fourier Transform (DFT).

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

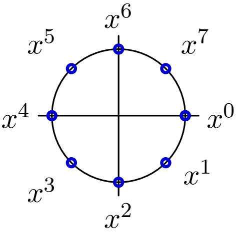
– **widely used** in engineering and science
– **elegant mathematics** (as alternative representations for polynomials)
– **elegant computer science** (divide-and-conquer)

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 values of an underlying frequency representation $F'(\cdot)$ at points x^k in the complex plane, where $x = e^{-j2\pi/N}$.



$$F[k] = F'(x^k) = \frac{1}{N} \sum_{n=0}^{N-1} f[n](x^k)^n$$

$F'(x^k)$ can be computed as a polynomial in x^k with coefficients $f[n]$. Evaluating the polynomial yields the frequency representation $F'(\cdot)$ and sampling $F'(\cdot)$ at powers of the N^{th} root of unity provides the DFT.

---

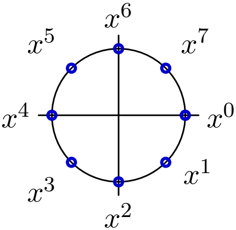
5 `https://www.youtube.com/watch?v=h7ap07q16V0` and Prof. Erik Demaine in 6.046 `https://www.youtube.com/watch?v=iTMn0Kt18tg`

The FFT as a Polynomial Representation

Separating **even** and **odd** powers of n to make two polynomials reduces the number of computations.

Values of the **even** polynomial will be symmetric about $x = 0$, so the values for $k = N/2$ to $N-1$ can be inferred from those for $k = 0$ to $N/2-1$.

Values of the **odd** polynomial will be anti-symmetric about $x = 0$, so the values for $k = N/2$ to $N-1$ can also be inferred from those for $k = 0$ to $N/2-1$.



$$F[k] = F'(x^k) = \frac{1}{N}\sum_{n=0}^{N-1} f[n](x^k)^n$$

This halves the number of computations required. But we can do better.

Recursively apply this decomposition on the even and odd parts $\rightarrow$ **FFT**.

Fast Fourier Transform

The Fast-Fourier Transform (FFT) is an algorithm (actually a family of algorithms) for computing the Discrete Fourier Transform (DFT).

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

– **widely used** in engineering and science
– **elegant mathematics** (as alternative representations for polynomials)
– **elegant computer science** (divide-and-conquer)