Discrete Fourier Transform (DFT)

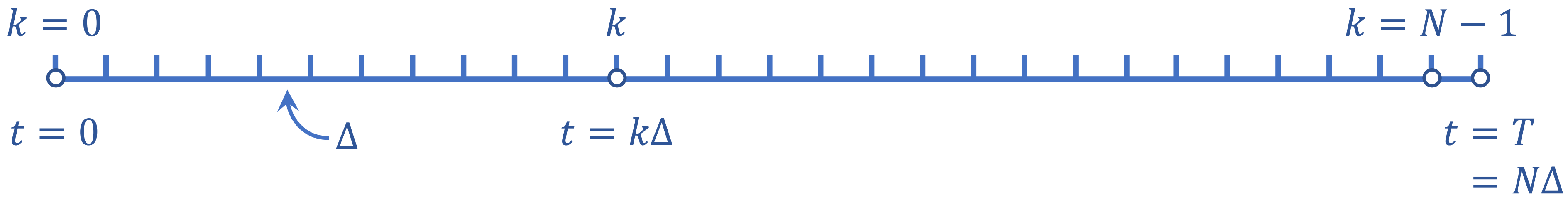
- Continuous FT and inverse (follow NR notation)

$$H(\omega) = \int_{-\infty}^{\infty} dt\ \ h(t)\, e^{i\omega t} \qquad (\dagger)$$

$$h(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} d\omega\ \ H(\omega)\, e^{i\omega t}$$

- Generally don’t have an infinite amount of input data…
- More typically, we sample a data stream at a finite number N of discrete points, with (for convenience) a fixed sampling interval Δ.
- Sampling times are

$$t_k = k\Delta, \quad k = 0, 1, \dots, N-1$$



Discrete Fourier Transform (DFT)

- Discrete version of (†) is

$$H(\omega) = \Delta \sum_{k=0}^{N-1} h(t_k)\, e^{i\omega t_k}$$

- Conventionally drop the Δ and define $h_k = h(t_k)$, so

$$H(\omega) = \sum_{k=0}^{N-1} h_k\, e^{i\omega t_k}$$

- Clearly, only N input data points, so only expect N independent output data values.
- Conventional to sample outputs only at discrete angular frequencies

$$\omega_n = \frac{2\pi n}{T}, \quad n = 0, 1, \dots, N-1$$

or frequencies

$$f_n = \frac{\omega_n}{2\pi} = \frac{n}{T}$$

Discrete Fourier Transform (DFT)

- Defining $H_n = H(\omega_n)$, we have

$$H_n = \sum_{k=0}^{N-1} h_k\, e^{i\omega_n t_k}$$

$$= \sum_{k=0}^{N-1} h_k\, e^{i\frac{2\pi n}{N\Delta}k\Delta}$$

$$= \sum_{k=0}^{N-1} h_k\, e^{2\pi ink/N}$$

Discrete Fourier Transform

- Note: we have largely abstracted away the physical context:

  Matrix operation on vector $\boldsymbol{h} = \{h_k\}$:

$$\boldsymbol{H} = M\boldsymbol{h}$$

$$H_n = \sum_k M_{nk} h_k$$

where

$$M_{nk} = e^{2\pi ink/N}.$$

Discrete Fourier Transform (DFT)

- Discrete orthogonality: recall continuous version

$$\int_0^{2\pi} \left(e^{ipx}\right)^* e^{iqx}\, dx = 2\pi\delta_{pq}$$

- Here,

$$\sum_{k=0}^{N-1} \left(e^{2\pi ipk/N}\right)^* e^{2\pi iqk/N}$$

$$= \sum_{k=0}^{N-1} e^{2\pi i(q-p)k/N}$$

$$= \frac{1-r^N}{1-r}$$

$$= \begin{cases} 0, & r \neq 1,\ p \neq q \\ N, & r = 1,\ p = q \end{cases}$$

$$= N\delta_{pq}$$

$$\sum_{k=0}^{N-1} r^k = \frac{1-r^N}{1-r}$$

$$r = e^{2\pi i(q-p)/N}$$

Discrete Fourier Transform (DFT)

- Discrete transform:

$$H_n = \sum_{k=0}^{N-1} h_k \, e^{2\pi i n k/N}$$

- Discrete orthogonality:

$$\sum_{k=0}^{N-1} \left(e^{2\pi i p k/N}\right)^* e^{2\pi i q k/N} = N\delta_{pq}$$

- Inverse DFT:

$$h_k = \frac{1}{N}\sum_{n=0}^{N-1} H_n e^{-2\pi i n k/N}$$

or

$$M^{-1} = \frac{1}{N} M^*$$

DFT Conventions

- We chose DFT frequencies with $n = 0, \ldots, N-1$, but the parallel with FT would suggest that we include both positive and negative frequencies as part of the definition.

  i.e. really want $n = -N/2, \ldots, N/2$ in the definition and inverse

- Conventional to define DFT with $n = 0, \ldots, N-1$, but should think of the corresponding frequencies as running from $-N/2$ to $N/2$.

- Why? DFT is <u>periodic</u>

$$H_n = \sum_{k=0}^{N-1} h_k \, e^{2\pi i n k/N}$$

$$H_{n+N} = H_n$$

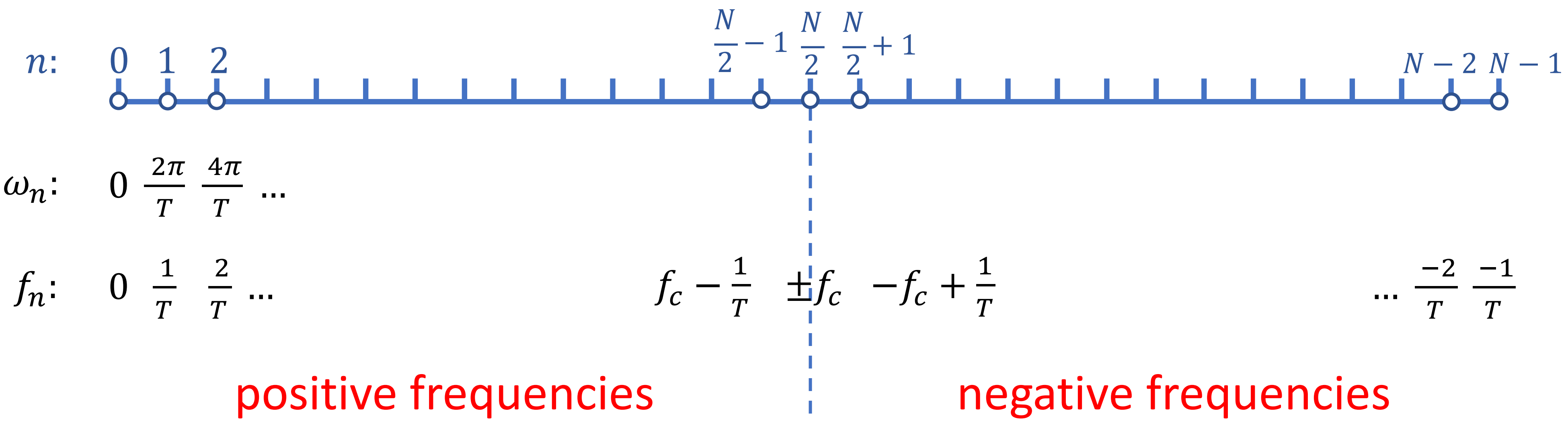
so

$$H_{-N/2} = H_{N/2}$$

DFT Conventions

- Interpretation:

define <u>Nyquist frequency</u> $f_c = \frac{1}{2\Delta} = \frac{N}{2T} = \frac{N/2}{T}$



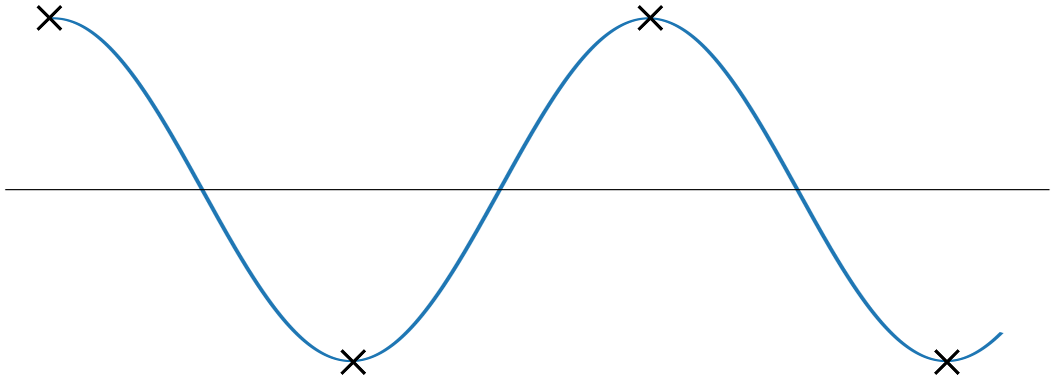
Nyquist Frequency

- Significance of the Nyquist frequency $f_c = \frac{1}{2\Delta}$
- Imagine measuring a cosine wave of frequency f_c at sampling interval Δ

$$h(t) = \cos(2\pi f_c t)$$

$$\Longrightarrow \quad \{h_k\} = \{1, -1,\ 1, -1, \dots\}$$

- Two sample points per period.
- f_c is the <u>maximum</u> frequency resolvable with sampling interval Δ.



DFT Example

- Take $N = 4,\ T = 2\pi,\ \Delta = \pi/2$

$$h(t) = \cos t$$

$$t_k = k\pi/2, \quad k = 0, 1, 2, 3$$

$$\{h_k\} = \{1,\ 0, -1,\ 0\}$$

Frequencies

$$\omega_n = \frac{2\pi n}{T} = n$$

- Then

$$M_{nk} = e^{i\omega_n t_k} = e^{in\pi k/2}$$

$$\Longrightarrow\ \{H_n\} = \{0,\ 2,\ 0,\ 2\}$$

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

DFT Example

- Transform back:

$$\boldsymbol{h} = M^{-1}\boldsymbol{H} = \begin{pmatrix} 1 \\ 0 \\ -1 \\ 0 \end{pmatrix}$$

- Works as matrix operation, but if we interpret in terms of frequencies and times, find

$$h_k = \frac{1}{4}\left[2e^{-i\omega_1 t_k} + 2e^{-i\omega_3 t_k}\right]$$

$3 \rightarrow -1$

$$= \frac{1}{2}\left[e^{-it_k} + e^{it_k}\right]$$

$$= \cos t_k$$

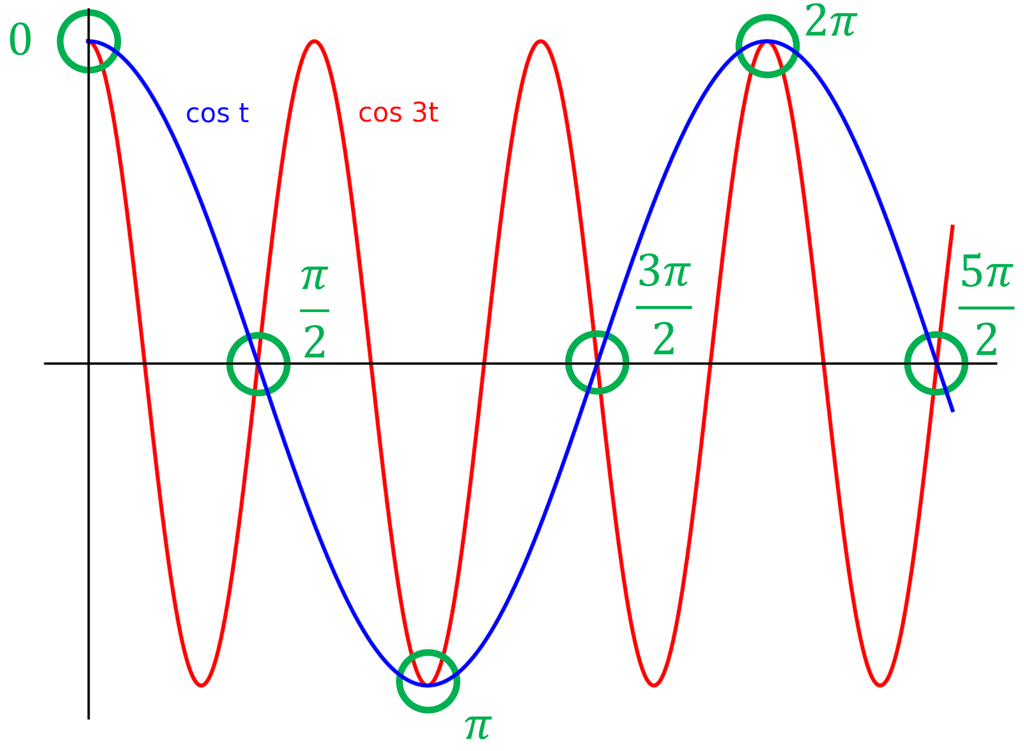
$$M = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

$$M^{-1} = \frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}$$

$$\boldsymbol{H} = \begin{pmatrix} 0 \\ 2 \\ 0 \\ 2 \end{pmatrix}$$

Aliasing

- In the previous example, can/must replace $\omega_3 = 3$ by $\omega_3 = -1$ because, at this sampling resolution, $\cos t$ and $\cos 3t$ look the same.



- This effect is called aliasing.

Aliasing

- Can only "see" frequencies within the bandwidth defined by $[-f_c, f_c]$.
- Because of aliasing, a high-frequency signal outside the range can masquerade as a lower-frequency signal inside.
- We have no control over the frequency spectrum of the signal, so must find ways to deal with the effects of aliasing.
- Signals $e^{2\pi i f_1 t_k}$ and $e^{2\pi i f_2 t_k}$ look the same at <u>all</u> $t_k = k\Delta$ iff f_1 and f_2 differ by a multiple of $1/\Delta$:

$$e^{2\pi i f_1 t_k} = e^{2\pi i f_2 t_k} \Longrightarrow 2\pi i (f_1 - f_2)\, k\Delta = 2\pi i m \quad (m \text{ integer})$$

$$\Longrightarrow (f_1 - f_2)\, k\Delta = m$$

- Clearly the most restrictive case is $k = 1$

$$(f_1 - f_2)\, \Delta = m$$
$$\Longrightarrow (f_1 - f_2)\, k\Delta = km$$

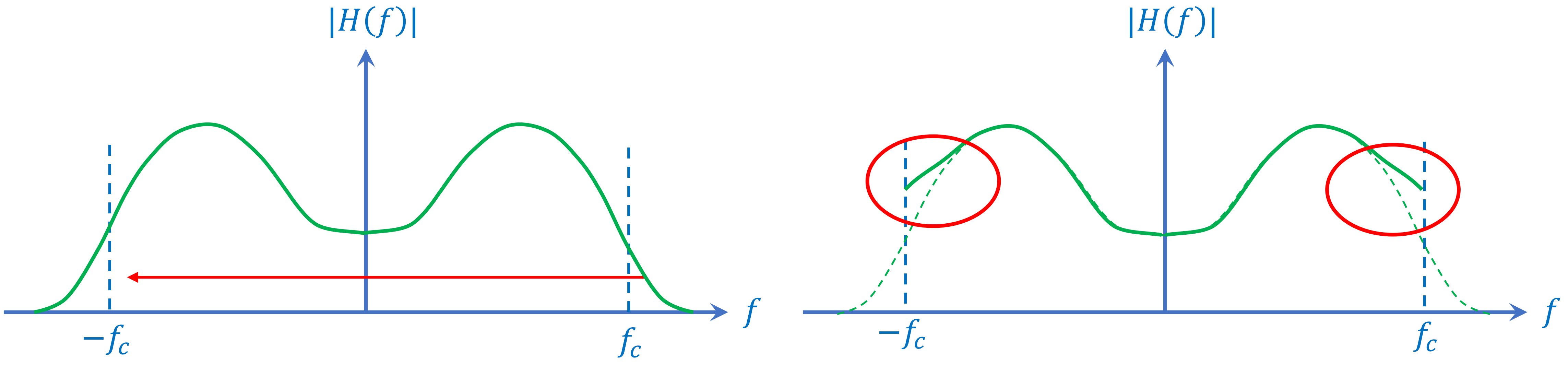
$$\Longrightarrow \quad f_1 - f_2 = \frac{m}{\Delta} = 2mf_c$$

Aliasing

- Because of aliasing, a high-frequency signal outside the range $[-f_c, f_c]$ cannot be distinguished from a lower-frequency one inside:

$$f_2 = f_c + \varepsilon \quad \text{looks like } f_1 = f_2 - 2f_c = -f_c + \varepsilon$$

- “Contaminates” the transform with spurious signal.



Fast Fourier Transform (FFT)

- Computation of DFT is expensive if N is large (may be billions or more)

  $$H_n = \sum_{k=0}^{N-1} h_k\, e^{2\pi i n k/N}$$

  Each sum involves $\mathcal{O}(N)$ operations, and there are N sums

  $\Longrightarrow \quad \mathcal{O}(N^2)$ operations in total.

  Prohibitively expensive for large datasets.

- Fast Fourier transform (FFT) is an algorithm for determining a DFT exactly in $\mathcal{O}(N \log N)$ operations.

- Cooley & Tukey (1965)

  Danielson & Lanczos (1942)

  $\vdots$

  Gauss (1805)

Fast Fourier Transform (FFT)

- Danielson – Lanczos lemma:

  A DFT of length N can be written as a sum of two DFTs, each of length $N/2$

  – one from the even-numbered points

  – one from the odd-numbered points

Notation:

$$H_n = \sum_{k=0}^{N-1} h_k\, \omega_N^{nk}$$

$$\omega_N = e^{2\pi i/N}$$

$=$ principal N-th root of 1

- Let

$$\begin{aligned} H_n &= \sum_{k=0}^{N-1} h_k\, e^{2\pi ink/N} \\ &= \sum_{l=0}^{N/2-1} h_{2l}\, e^{2\pi in(2l)/N} + \sum_{l=0}^{N/2-1} h_{2l+1}\, e^{2\pi in(2l+1)/N} \\ &= \sum_{l=0}^{N/2-1} h_{2l}\, e^{2\pi inl/(N/2)} + e^{2\pi in/N} \sum_{l=0}^{N/2-1} h_{2l+1}\, e^{2\pi inl/(N/2)} \\ &= H_n^e + \omega_N^n H_n^o \end{aligned}$$

- Both new sums are of length $N/2$, periodic in n, period $N/2$.

Fast Fourier Transform (FFT)

- For the Danielson – Lanczos lemma to work, require N even
  - going to apply it recursively, so need N to be a power of 2: $N = 2^m$
- Repeat the process:

$$H_n^e = H_n^{ee} + \omega_{N/2}^n H_n^{eo}$$

$$H_n^o = H_n^{oe} + \omega_{N/2}^n H_n^{oo}$$

  - four sums, each of length $N/4$, periodic in n, period $N/4$
  - each original h_k appears in exactly one of them
- Repeat again, and continue m times until until $N/2^m = 1$.
  - N sums $H_n^{eoe\dots oe}$, each of length 1
  - $H_n^{eoe\dots oe} = h_k$, for some specific k, independent of n
  - number of $eoe \dots oe$ exponents $= m = \log_2 N$

Fast Fourier Transform (FFT)

- We have decomposed the DFT calculation into a collection of sums that reduce trivially to a single member of the original input dataset

  $$H_n^{eoe\ldots oe} = h_k$$

  Question: How does k relate to $eoe \ldots oe$?

- Method of creation gives a clue.

  H^e consists of even terms $\Longrightarrow$ last bit in binary representation of k is 0

  H^o consists of odd terms $\Longrightarrow$ last bit in binary representation of k is 1

  H^{ee} consists of even terms from the even sequence

  $\Longrightarrow$ last 2 bits in binary representation of k are 0

  H^{eo} consists of odd terms from the even sequence

  $\Longrightarrow$ last bit in binary representation of k is 0, previous is 1

"etc."

Fast Fourier Transform (FFT)

- By construction, the $eoe \ldots oe$ in

$$H_n^{eoe \ldots oe} = h_k$$

  is just the binary representation of k, written backwards!

- Rule for determining k is simple:
  1. <u>Reverse</u> the $eoe \ldots oe$ sequence
  2. Replace e by 0, o by 1
  3. Interpret the result in binary, $= k$

- e.g. $N = 8$

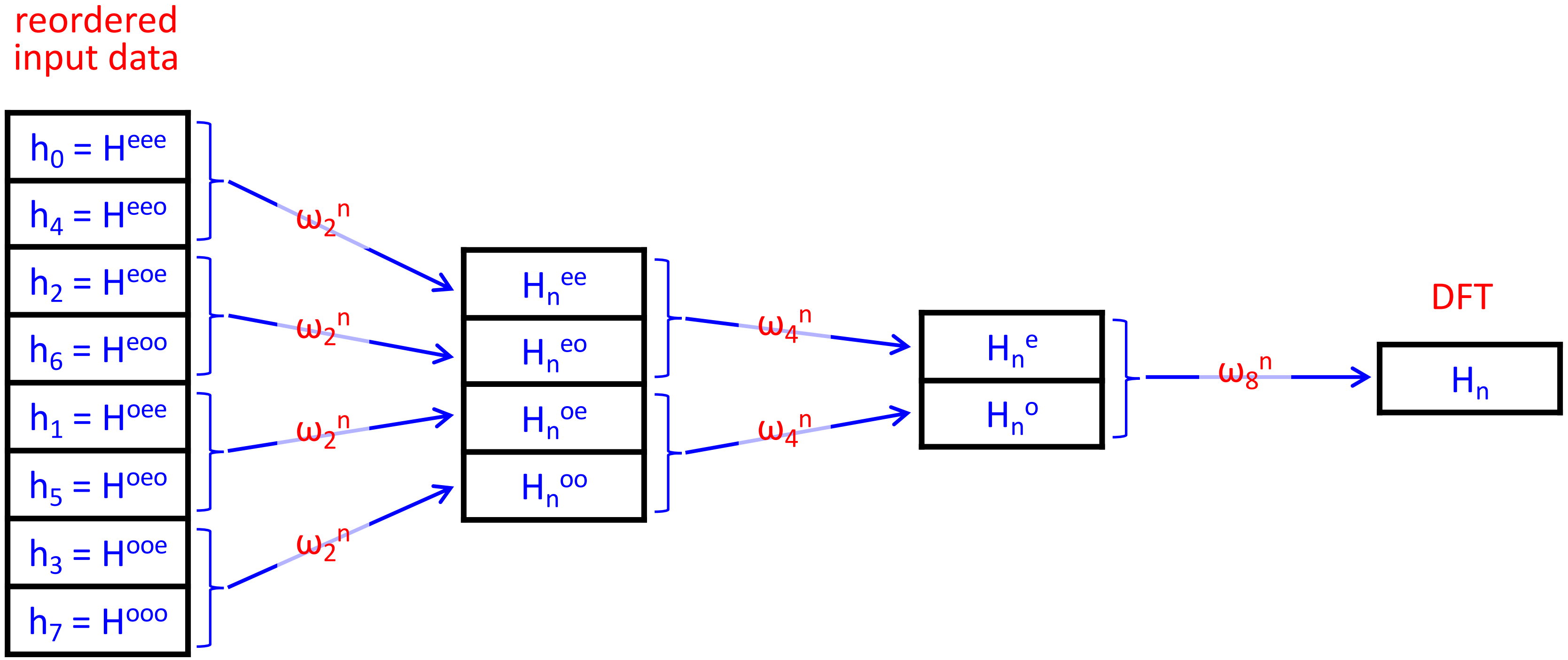
H:	0	1	2	3	4	5	6	7
H^x:	0	2 ^{e}	4	6	1	3 ^{o}	5	7
H^{xx}:	0 ^{ee}	4	2 ^{eo}	6	1 ^{oe}	5	3 ^{oo}	7
	eee	eeo	eoe	eoo	oee	oeo	ooe	ooo

Fast Fourier Transform (FFT)

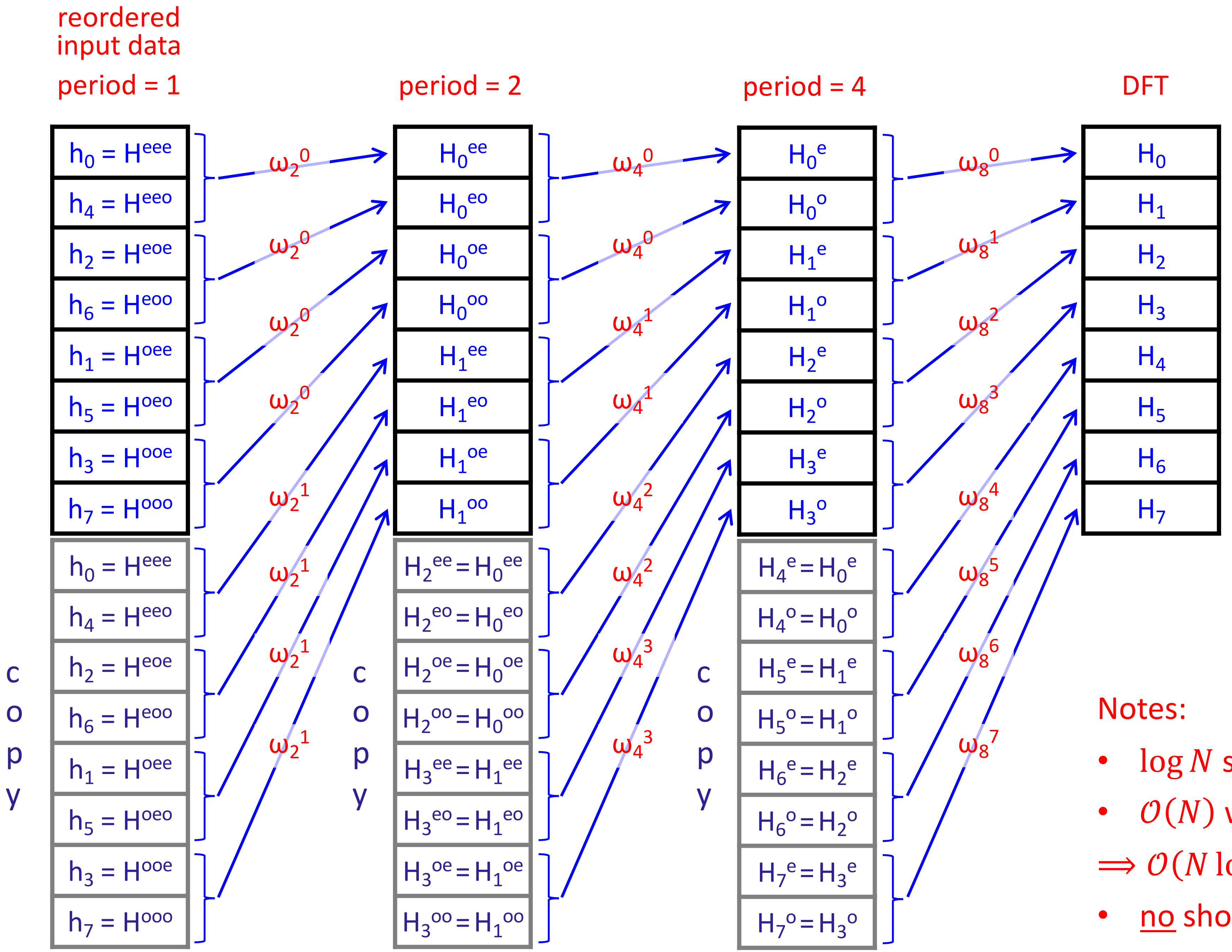
- Going in the other direction:

	reverse		binary		evaluate	
eee	$\rightarrow$	eee	$\rightarrow$	000	$\rightarrow$	0
eeo	$\rightarrow$	oee	$\rightarrow$	100	$\rightarrow$	4
eoe	$\rightarrow$	eoe	$\rightarrow$	010	$\rightarrow$	2
eoo	$\rightarrow$	ooe	$\rightarrow$	110	$\rightarrow$	6
oee	$\rightarrow$	eeo	$\rightarrow$	001	$\rightarrow$	1
oeo	$\rightarrow$	oeo	$\rightarrow$	101	$\rightarrow$	5
ooe	$\rightarrow$	eoo	$\rightarrow$	011	$\rightarrow$	3
ooo	$\rightarrow$	ooo	$\rightarrow$	111	$\rightarrow$	7

- Rules:
  - create a bit-reordered sequence
  - recursively combine adjacent pairs to get to next level



reordered
input data
h0 = Heee
h4 = Heeo
h2 = Heoe
h6 = Heoo
h1 = Hoee
h5 = Hoeo
h3 = Hooe
h7 = Hooo
ω2n
ω2n
ω2n
ω2n
Hnee
Hneo
Hnoe
Hnoo
ω4n
ω4n
Hne
Hno
ω8n
DFT
Hn



Notes:

- $\log N$ stages
- $\mathcal{O}(N)$ work per stage

$\Longrightarrow \mathcal{O}(N \log_2 N)$ complexity

- <u>no</u> short cuts!

Numerical Recipes in C

```c
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void four1(float data[], unsigned long nn, int isign)
{
    unsigned long n,mmax,m,j,istep,i;
    double wtemp,wr,wpr,wpi,wi,theta;
    float tempr,tempi;

    n = nn << 1;
    j = 1;
    for (i = 1; i < n; i += 2) {
        if (j > i) {
            SWAP(data[j],data[i]);
            SWAP(data[j+1],data[i+1]);
        }
        m = n >> 1;
        while (m >= 2 && j > m) {
            j -= m;
            m >>= 1;
        }
        j += m;
    }
```

bit reversal

Numerical
Recipes in C

```
mmax = 2;
while (n > mmax) {
    istep = mmax << 1;
    theta = isign*(6.28318530717959/mmax);
    wtemp = sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi = sin(theta);
    wr = 1.0;
    wi = 0.0;
    for (m = 1; m < mmax; m += 2) {
        for (i = m; i <= n; i += istep) {
            j = i+mmax;
            tempr = wr*data[j]-wi*data[j+1];
            tempi = wr*data[j+1]+wi*data[j];
            data[j] = data[i]-tempr;
            data[j+1] = data[i+1]-tempi;
            data[i] += tempr;
            data[i+1] += tempi;
        }
        wr = (wtemp=wr)*wpr-wi*wpi+wr;
        wi = wi*wpr+wtemp*wpi+wi;
    }
    mmax = istep;
}
```

combine in place

Python

```
import numpy as np
import matplotlib.pyplot as plt

def normalize(a):
    sum = np.sum(np.abs(a)**2)
    return a/np.sqrt(sum)

N = 256
W = 8

x = np.linspace(0, N, N)
h = np.exp(-((x-N/2.)/W)**2)

H = np.fft.fft(h)

plt.plot(normalize(h), c='b')
plt.plot(normalize(np.real(H)),  c='r')
plt.show()
```

do the FFT!

Examples

- FFT Gaussian demo

  Expect transform of a Gaussian to be a Gaussian, but see oscillations too.

  Why?

- DFT $\sim \int_0^T h(t)\, e^{-i\omega t}\, dt$, not $\int_{-\infty}^{\infty} h(t)\, e^{-i\omega t}\, dt$

  so if $h(t) = e^{-(t-T/2)^2/a^2}$

$$\Longrightarrow \text{ DFT } \sim \int_0^T e^{-(t-T/2)^2/a^2} e^{-i\omega t}\, dt$$

$\tau = t - T/2$

$$= \int_{-T/2}^{T/2} e^{-\tau^2/a^2} e^{-i\omega(\tau+T/2)}\, d\tau$$

$$= \left\downarrow e^{-i\omega T/2}\right. \underbrace{\int_{-T/2}^{T/2} e^{-\tau^2/a^2 - i\omega\tau}\, d\tau}_{\text{expected result}}$$

For $\omega = \omega_n = 2\pi n/T$, $e^{-i\omega T/2} = e^{-n\pi i} = (-1)^n$