Lecture 24
Fourier transforms

- Fast Fourier transform (FFT).
- Other FFT methods.
- Real Fourier transforms.
- Fast sine and cosine transforms.

Last class we introduced how to compute a discretized Fourier transform,

$$h_k \equiv h(t_k), \qquad t_k \equiv k\Delta, \qquad k = 0, 1, 2, \ldots, N-1$$

$$f_n \equiv \frac{n}{N\Delta}, \qquad n = -\frac{N}{2}, \ldots, \frac{N}{2}$$

$$H(f_n) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f_n t} dt \approx \sum_{k=0}^{N-1} h_k \, e^{2\pi i f_n t_k} \Delta = \Delta \sum_{k=0}^{N-1} h_k \, e^{2\pi i k n / N}$$

$$H_n \equiv \sum_{k=0}^{N-1} h_k \, e^{2\pi i k n / N} \qquad\qquad H(f_n) \approx \Delta H_n$$

Fast Fourier Transform:

How many operations are needed to compute a Fourier transform? Until the mid 60's the answer went like this: define a complex number $W=e^{2\pi i/N}$. The transform can therefore be written as,

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

Therefore one is multiplying a matrix which has in its n,k component the n times k power of W. This requires $O(N^2)$ products plus the steps needed to construct the matrix.

It therefore came as a surprise when it was noted that one can compute the transform in $N \log_2 N$ operations via the FFT algorithm. This difference can be enormous. For $N=10^6$ it can mean seconds instead of **days** of CPU time on modern computers.

This fact became popularized after a paper by Cooley and Tukey (1967). The fact is that people had been optimizing Fourier transforms since Gauss in 1805. The clearest explanation of FFT is actually in a paper by Danielson and Lanczos (1942!).

The discovery of Danielson an Lanczos was that a Fourier transform of length N can be written as the sum of two Fourier transforms of length N/2. One is formed from the odd numbered points and one from the even numbered points,

$$
\begin{aligned}
F_k &= \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j \\
&= \sum_{j=0}^{N/2-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi ik(2j+1)/N} f_{2j+1} \\
&= \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j+1} \\
&= F_k^e + W^k \; F_k^o
\end{aligned}
$$

The remarkable aspect of this observation is that one can apply it recursively. We can now break each of the last two terms into two Fourier transforms of length N/4, F^{ee}_k, F^{eo}_k, etc.

Although more sophisticated things can be done for other cases, it is evident that this technique works more straightforwardly if N is a power of 2. If your N is not, then you should “pad the function with zeros” (or use other techniques we’ll discuss later) until it is.

The endpoint of the procedure of Danielson and Lanczos is therefore a Fourier transform of length one,

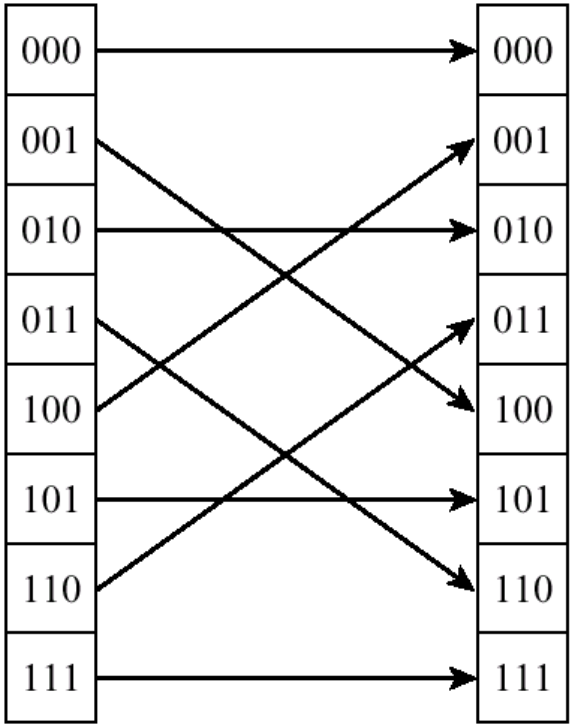
$$F_k^{eoeeoeo\cdots oee} = f_n \qquad \text{for some } n$$

But a Fourier transform of length one is just the identity operation! For each pattern of length $\log_2 N$ of eeoeeeoeoeo…'s we have one of the input numbers f_k.

The next step is to find out which value of n corresponds to each pattern of e's and o's. The answer: reverse the pattern and substitute e=0, o=1 and you will have *in binary*, the number n.

What? It is essentially because each subdivision in e and o tests the lowest order bits (in binary) of the number n.

The bit-reversal technique makes constructing the Fourier transform exceedingly simple. Suppose you start with a vector of length N, f_k and bit-reverse the order of its components.

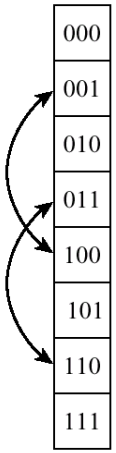


Example of length 8

We now view each component as the length-one Fourier transform of the initial data. We then combine adjacent pairs of produce the length-two transform. We then combine pairs of pairs to form the length-four transform, etc.

At the end of the day we are left with two terms, one per each N/2 transform

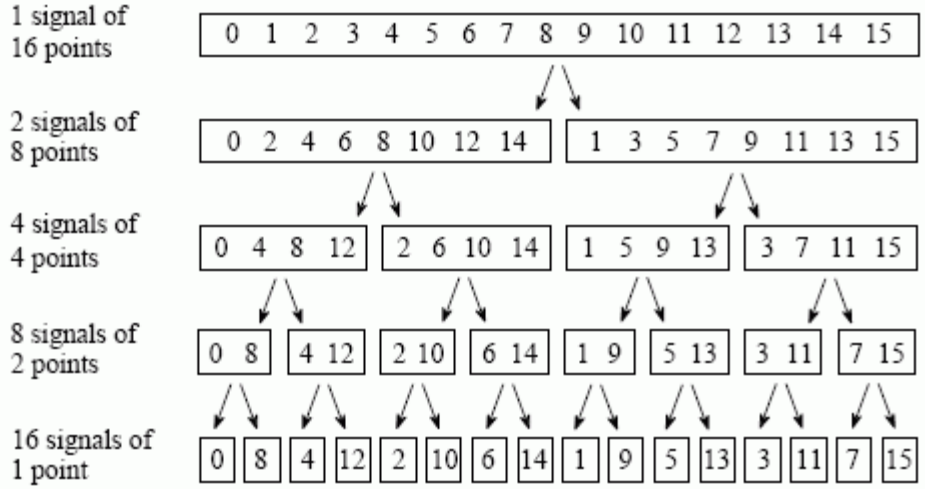
Each combination takes about N operations, and there are $\log_2 N$ combinations, for a total of $N \log_2 N$ operations.



Bit swapping can be done “in place” so it does not take additional storage.

And this is it.

Example



Sample numbers in normal order			Sample numbers after bit reversal	
Decimal	*Binary*		*Decimal*	*Binary*
0	0000		0	0000
1	0001		8	1000
2	0010		4	0100
3	0011		12	1100
4	0100		2	0010
5	0101		10	1010
6	0110	⇨	6	0100
7	0111		14	1110
8	1000		1	0001
9	1001		9	1001
10	1010		5	0101
11	1011		13	1101
12	1100		3	0011
13	1101		11	1011
14	1110		7	0111
15	1111		15	1111

```
subroutine four1(data,nn,isign)
    real*8 wr,wi,wpr,wpi,wtemp,theta
    dimension data(*)
    n=2*nn
    j=1
    do 11 i=1,n,2
     if(j.gt.i)then
       tempr=data(j)
       tempi=data(j+1)
       data(j)=data(i)
       data(j+1)=data(i+1)
       data(i)=tempr
       data(i+1)=tempi
     endif
     m=n/2
1     if ((m.ge.2).and.(j.gt.m)) then
       j=j-m
       m=m/2
     go to 1
     endif
     j=j+m
11   continue
    mmax=2
2    if (n.gt.mmax) then
     istep=2*mmax
     theta=6.28318530717959d0/(isign*mmax)
     wpr=-2.d0*dsin(0.5d0*theta)**2
     wpi=dsin(theta)
     wr=1.d0
     wi=0.d0
     do 13 m=1,mmax,2
      do 12 i=m,n,istep
       j=i+mmax
       tempr=sngl(wr)*data(j)-sngl(wi)*data(j+1)
       tempi=sngl(wr)*data(j+1)+sngl(wi)*data(j)
       data(j)=data(i)-tempr
       data(j+1)=data(i+1)-tempi
       data(i)=data(i)+tempr
       data(i+1)=data(i+1)+tempi
12     continue
       wtemp=wr
       wr=wr*wpr-wi*wpi+wr
       wi=wi*wpr+wtemp*wpi+wi
13     continue
      mmax=istep
    go to 2
    endif
    return
    end
```

Danielson Lanczos formulae

Bit swapping

Therfore the essence of the Danielson-Lanczos FFT implementation is: bit swap and then arrange in pairs.

There are several variations that can be done. The first one is to do the arrangement in pairs first and then bit-swap.

Another possibility, if one wants to go to Fourier space and then return, and for certain operations that don't require to know where one is in frequency space, one can arrange in pairs, operate and rearrange in pairs again without bit swapping. This is hardly useful: swapping consumes little time and few operations are doable this way.

Other algorithms use 4 or 8 terms instead of pairs and then use highly optimized routines for handling the four or eight point transforms.

For N that are not powers of 2, some algorithms factorize with the smallest prime. If N is prime then they do a slow N^2 transform.

The routine we just discussed was for transforming a complex functions. In many occasions one is faced with transforming real data. Of course, one can use the routine and simply fill with zeros the complex part.

One possibility is to use the routine to transform simultaneously two functions. It is common that one needs to transform two functions, for instance, when one computes a correlation.

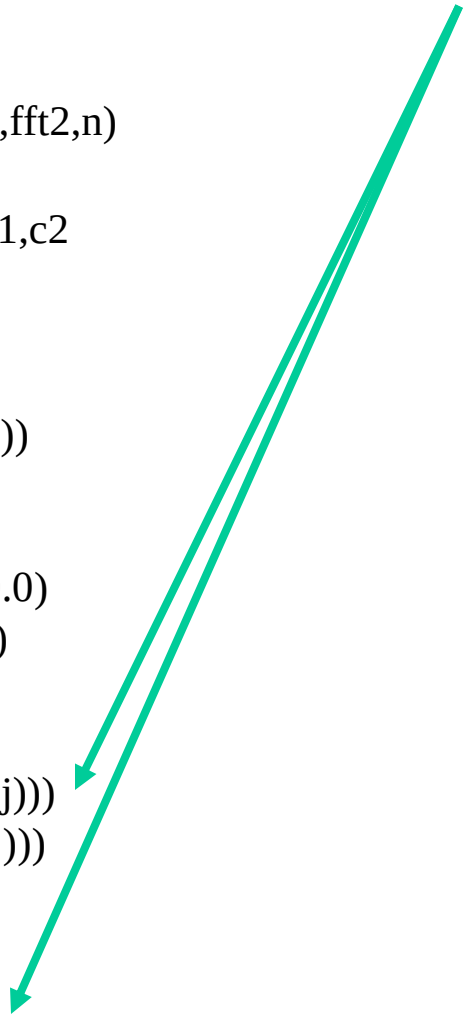
Another possibility is to pack the function cleverly in such a way that it occupies half the space and use the complex transformation routine.

To transform two real functions simultaneously we use the fact that a purely real and purely imaginary function's FT satisfy,

$$F_{N-n} = (F_n)^* \qquad G_{N-n} = -(G_n)^*$$

We use the symmetries to “unpack” the resulting FT into two transforms,

```
      subroutine twofft(data1,data2,fft1,fft2,n)
          dimension data1(n),data2(n)
          complex fft1(n),fft2(n),h1,h2,c1,c2
          c1=cmplx(0.5,0.0)
          c2=cmplx(0.0,-0.5)
          do 11 j=1,n
            fft1(j)=cmplx(data1(j),data2(j))
     11   continue
          call four1(fft1,n,1)
          fft2(1)=cmplx(aimag(fft1(1)),0.0)
          fft1(1)=cmplx(real(fft1(1)),0.0)
          n2=n+2
          do 12 j=2,n/2+1
            h1=c1*(fft1(j)+conjg(fft1(n2-j)))
            h2=c2*(fft1(j)-conjg(fft1(n2-j)))
            fft1(j)=h1
            fft1(n2-j)=conjg(h1)
            fft2(j)=h2
            fft2(n2-j)=conjg(h2)
     12   continue
          return
          end
```



If one wishes to transform one single real function as a complex function of half the length, one simply treats the even and odd points separately as components of a complex vector,

$$h_j = f_{2j} + i f_{2j+1}, \quad j = 0, \ldots, N/2 - 1$$

$$F_n^e = \sum_{k=0}^{N/2-1} f_{2k}\, e^{2\pi i k n/(N/2)}$$

$$F_n^o = \sum_{k=0}^{N/2-1} f_{2k+1}\, e^{2\pi i k n/(N/2)}$$

Repackaging works as in the FFT formula,

$$F_n = F_n^e + e^{2\pi i n/N} F_n^o \qquad n = 0, \ldots, N-1$$

Fast sine and cosine transforms:

As we discussed in previous lectures, one use of Fourier transforms is to solve differential equations. In such cases one needs to take care of the boundary conditions. A common type of boundary condition is the function vanishing at the boundary or having a constant value. In such cases, sine and cosine transforms are handy.

$$F_k = \sum_{j=1}^{N-1} f_j \sin(\pi jk/N)$$

At a first glance, this looks like the real part of the ordinary Fourier transform. But it is not. There is a factor 1/2 missing in the argument of the sine. The sine transform uses only sines to expand the function. The usual transform uses both sines and cosines, but only half of each.

The strategy to construct fast sine transforms will be to extend the function to be transformed outside its domain of definition in such a way as to make it an odd function.

So given a function f_j with j=0..N-1, we extend it past N to 2N in such a way that f_N=0 and with,

$$f_{2N-j} \equiv -f_j \qquad j = 0, \ldots, N-1$$

Consider the FFT of this extended function, $F_k = \sum_{j=0}^{2N-1} f_j e^{2\pi i j k/(2N)}$

The second half of the sum can be rewritten as,

$$\sum_{j=N}^{2N-1} f_j e^{2\pi i j k/(2N)} = \sum_{j'=1}^{N} f_{2N-j'} e^{2\pi i (2N-j')k/(2N)}$$

$$= -\sum_{j'=0}^{N-1} f_{j'} e^{-2\pi i j' k/(2N)}$$

$$F_k = \sum_{j=0}^{N-1} f_j \left[e^{2\pi i j k/(2N)} - e^{-2\pi i j k/(2N)} \right]$$

So,

$$= 2i \sum_{j=0}^{N-1} f_j \sin(\pi j k/N)$$

The obvious disadvantage of this method is a factor of 2 inefficiency, which appears in the result which contains half the vector with zeros. In higher dimensions this is intolerable.

To achieve sine transforms more efficiently, we define an auxiliary vector,

$$y_0 = 0$$

$$y_j = \sin(j\pi/N)(f_j + f_{N-j}) + \frac{1}{2}(f_j - f_{N-j}) \qquad j = 1, \ldots, N-1$$

And notice that the first term is symmetric and the second antisymmetric around j=N/2. Then the real and imaginary part of the Fourier transform are given by,

$$\begin{aligned} R_k &= \sum_{j=0}^{N-1} y_j \cos(2\pi jk/N) \\ &= \sum_{j=1}^{N-1} (f_j + f_{N-j}) \sin(j\pi/N) \cos(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} 2 f_j \sin(j\pi/N) \cos(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} f_j \left[\sin\frac{(2k+1)j\pi}{N} - \sin\frac{(2k-1)j\pi}{N} \right] \\ &= F_{2k+1} - F_{2k-1} \end{aligned}$$

$$\begin{aligned} I_k &= \sum_{j=0}^{N-1} y_j \sin(2\pi jk/N) \\ &= \sum_{j=1}^{N-1} (f_j - f_{N-j}) \frac{1}{2} \sin(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} f_j \sin(2\pi jk/N) \\ &= F_{2k} \end{aligned}$$

We can conclude:

$$F_{2k} = I_k \qquad F_{2k+1} = F_{2k-1} + R_k \qquad k = 0, \ldots, (N/2-1)$$

FFT in more than one dimension:

Naively, one can compute successive Fourier transforms just as one would compute successive integrals in higher dimensions.

$$H(n_1, n_2) \equiv \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} \exp(2\pi i k_2 n_2/N_2)\ \exp(2\pi i k_1 n_1/N_1)\ h(k_1, k_2)$$

$$\begin{aligned} H(n_1, n_2) &= \text{FFT-on-index-1}\,(\text{FFT-on-index-2}\,[h(k_1, k_2)]) \\ &= \text{FFT-on-index-2}\,(\text{FFT-on-index-1}\,[h(k_1, k_2)]) \end{aligned}$$

This is harder than it sounds (to do efficiently). Since the FFT routines require the data in forms of vectors, one needs to read them out of the array defining $h(k_1,k_2)$. One finds oneself copying several times the same information.

There are many tricks to handle this and other aspects (memory efficiency). Most are too detailed and uninspired to discuss in class. Some are language dependent.

Summary

- FFT considerably speeds up the computation of Fourier transform. It is based on a neat trick.
- Sine transforms are handled by modifying the function.
- FFT in higher dimensions: the devil is in the details.