Lecture 34 § 5.7 The Fast Fourier Transform

So far, we've seen the following variants on the Discrete Fourier Transform

	Complex Signal	Real Signal
Standard	$f(x) \approx \sum_{k=0}^{n-1} c_k e^{ikx}$	$f(x) \approx \mathrm{re}\left(\sum_{k=0}^{n-1} c_k e^{ikx}\right)$
Low frequency alternative	$f(x) \approx \sum_{k=-m}^{m^*} c_k e^{ikx}$	$f(x) \approx \mathrm{re}\left(\sum_{k=-m}^{m^*} c_k e^{ikx}\right)$
Noise reducing alternative	$f(x) \approx \sum_{k=-l}^{l^*} c_k e^{ikx}$	$f(x) \approx \mathrm{re}\left(\sum_{k=-l}^{l^*} c_k e^{ikx}\right)$

* (the choice of m or $m-1$ depends on using an even or odd number of sample points)

(where l is much smaller than $n/2$)

Further, ~~because~~ we noted that given n sample points, e^{ikx} will be indistinguishable on the sample points from $e^{i(k+jn)x}$;

$$\omega_k = \omega_{k+n} = \omega_{k-n} = \omega_{k+2n} = \cdots$$

And hence, the coefficients $c_k = \langle \vec{f}, \omega_k \rangle$ can be computed identically for the standard or low frequency case.

How much computation does this use? Computing the averaged dot product $\langle \vec{f}, \omega_k \rangle$ requires n complex arithmetic operations, and we do this for all n coefficients. Thus, there are a total of n^2 complex operations.

The Fast Fourier Transform.

The Fast Fourier Transform (FFT from here on) is a method to greatly reduce the number of computations necessary. It is only applicable for $n = 2^r$ (powers of 2), but this is actually quite reasonable since our computation time will be so much improved that we can just increase our sample rate to the next even power of 2 and still gain quite a bit of computational savings.

Idea We will break down a Fourier transform with $n = 2^r$ sample points into two Fourier transforms requiring 2^{r-1} sample points and combine them. This will lead to an algorithm with $O(r\,n) = O(n \log n)$ operations, quite a savings from n^2!

Let $n = 2^r$. The kth coefficient c_k is computed as

$$c_k = \frac{1}{n}\sum_{j=0}^{n-1} f_j \, e^{-j\frac{2\pi i k}{n}}$$

We split this into two sums

$$c_k = \frac{1}{n}\sum_{\substack{j=0\\ j\text{ even}}}^{n-2} f_j \, e^{-j\frac{2\pi i k}{n}} + \frac{1}{n}\sum_{\substack{j=1\\ j\text{ odd}}}^{n-1} f_j \, e^{-j\frac{2\pi i k}{n}}$$

$$\boxed{2l = j} \qquad\qquad \boxed{2l+1 = j}$$

$$= \frac{1}{n}\sum_{l=0}^{\frac{n}{2}-1} f_{2l}\, e^{-l\frac{2\pi i k}{n/2}} + \frac{e^{\frac{-2\pi i k}{n}}}{n}\sum_{l=0}^{\frac{n}{2}-1} f_{2l+1}\, e^{-l\frac{2\pi i k}{n/2}}$$

$$= \frac{1}{2}\left(\frac{1}{n/2}\sum_{l=0}^{\frac{n}{2}-1} f_{2l}\, e^{-l\frac{2\pi i k}{n/2}} + \frac{e^{\frac{-2\pi i k}{n}}}{n/2}\sum_{l=0}^{\frac{n}{2}-1} f_{2l+1}\, e^{-l\frac{2\pi i k}{n/2}}\right)$$

Setting $\vec{f}^{even} = (f_0, f_2, \ldots, f_{n-2})$, $\vec{f}^{odd} = (f_1, f_3, \ldots, f_{n-1})$

we see that we've written the coefficient c_k in the fourier tranform of $\vec{f}$ as a sum of two fourier coefficients of $\vec{f}^{even}$ and $\vec{f}^{odd}$,

$$c_k = \frac{1}{2}\left(c_k^{even} + e^{\frac{-2\pi i k}{n}} c_k^{odd}\right). \quad \left(\text{recall } \begin{array}{l} c_k^{even} = c_{k-\frac{n}{2}}^{even} \\ c_k^{odd} = c_{k-\frac{n}{2}}^{odd} \end{array} \text{ by aliasing}\right)$$

Thus, we can write the Fourier transform for the vector $\vec{f}$ of length $n = 2^r$ in terms of two fourier transforms of length $\frac{n}{2} = 2^{r-1}$.

Repeating this r times, we eventually arrive at vectors of length 1, for which the fourier coefficient c_0 is the same as the single sample point. This leads to the following recursive algorithm, implemented in Python (with some math symbols for convenience):

= comments

```
fft(r, f):  # r = power of 2, n=2^r; f = data
    if r=0:
        return f
    else:
        n=2^r
        feven = f[0:2^r-1:2]  # Start at f0, continue to end selecting
                              # every other entry, [f0, f2, ..., fn-2]
        fodd = f[1:2^r-1:2]   # start at f1 continue to end selecting
                              # every other entry, [f1, f3, ..., fn-1]
        ceven = fft(r-1, feven)
        codd = fft(r-1, fodd)
        c = [(ceven[k%(n/2)] + e^(2πik/n) codd[k%(n/2)])/2  for k=0,..,n-1]
             # % = mod; so k%(n/2) takes care of the aliasing,
             # c_k = c_{k-n/2} for k ≥ n/2
        return c
```

To see how this works, we perform the algorithm on a set of $n=4$ points.

Let $f = (1, 2, 2, -1)$.

Depth $r = 0$ $\quad c = [1] \quad c = [2] \quad c = [2] \quad c = [-1]$

$r = 1$

~~odd~~ $c_{even} = [1]$

$c_{odd} = [2]$

$$c = [(1 + e^{i\pi \cdot 0} \cdot 2)/2, (1 + e^{i\pi \cdot 1} \cdot 2)/2]$$
$$= [(1 + 1 \cdot 2)/2, (1 + -1 \cdot 2)/2]$$
$$= [3/2, -1/2]$$

$c_{even} = [2]$

$c_{odd} = [-1]$

$$c = [(2 + e^{i\pi \cdot 0} \cdot -1)/2, (2 + e^{i\pi \cdot 1} \cdot -1)/2]$$
$$= [(2 + -1)/2, (2 + -1 \cdot -1)/2]$$
$$= [1/2, 3/2]$$

$r = 2$

$c_{even} = [3/2, -1/2]$

$c_{odd} = [1/2, 3/2]$

$$c = [(3/2 + e^{i\pi/2 \cdot 0} \cdot \tfrac{1}{2})/2, (-\tfrac{1}{2} + e^{i\pi/2 \cdot 1} \cdot 3/2)/2,$$
$$(3/2 + e^{i\pi/2 \cdot 2} \cdot \tfrac{1}{2})/2, (-\tfrac{1}{2} + e^{i\pi/2 \cdot 3} \cdot 3/2)/2]$$

$$= [(3/2 + \tfrac{1}{2})/2, (-\tfrac{1}{2} + i \cdot 3/2)/2,$$
$$(3/2 + -1 \cdot \tfrac{1}{2})/2, (-\tfrac{1}{2} - i \cdot 3/2)/2]$$
$$= [1, -\tfrac{1}{4} + \tfrac{3}{4}i, \tfrac{1}{2}, -\tfrac{1}{4} - \tfrac{3}{4}i]$$