Last time we introduced the discrete Fourier Transform. This operation is one of the main workhorses used in physics. Unfortunately, as currently formulated, the Fourier transform is very computationally expensive.

The wide use of Fourier techniques in the physical sciences has much to do with the development of the *fast Fourier transform*. We begin the development of this operation now. Keep in mind the fast Fourier transform is **not** different from the discrete Fourier transform. Instead it is computationally efficient way of computing the discrete Fourier transform.

(1) In the lecture we've shown that we can break up the discrete Fourier transform as

$$g(n\Delta\omega) = g_{even}(n\Delta\omega) + e^{-i2\pi n/N} g_{odd}(n\Delta\omega)$$

where $g_{even}(n\Delta\omega)$ is the sum of the even m terms and $g_{odd}(n\Delta\omega)$ is the sum of the odd m terms. Determine how many calculations are now required to compute the discrete Fourier transform.

$2 \times \left(\frac{N}{2}\right)^2 < N^2$

(2) Suppose that g_{even} and g_{odd} contain an even number of terms. Can you break up each sum as was done (1) again? If so, is there an advantage? That is, will computing the discrete Fourier transform require less operations?

split into even and odd terms

(3) What is the necessary condition on the length of the original signal in order to be able to continue splitting the sums as done in (1)

Has to be a length that is a power of 2

(4) Before we begin fully implementing a code to compute the fast Fourier transform, we need to introduce the concept of recursive programming. Sometimes a function refers to itself in order to perform the calculation. For example,

$$n! = n(n-1)!, \quad 0! = 1.$$

That is, in order to evaluate the factorial function, one needs to refer back to the factorial function. In order to calculate $n!$, we need $(n-1)!$, but to get $(n-1)!$, we need $(n-2)!$ and so on.

`MATLAB` allows one function to call other functions, we've used that often already. However, `MATLAB` also allows a function to call itself. This is called *recursive programming* and can be very useful. Recursive programming works by defining a termination step, then having the function call itself repeatedly until that step is reached. See if you can write a recursive function in `MATLAB` that computes the factorial of an inputted integer n.

$W_N^n = e^{-i 2\pi n/N}$

(5) Compute the fast Fourier transform for the sampled function,

$$f(0) = 0;\ f(1) = 1;\ f(2) = 2;\ f(3) = 3. \quad N = 4$$

$T = \Delta t$

Do just the 0 and first frequencies. Assume a $\Delta t = 1$.

$\Delta\omega \sim \frac{1}{T}$

$g(n\Delta\omega) = f(0\Delta t) + W_2^n f(2\Delta t) + W_4^n f(1\Delta t) + W_4^n W_2^n f(3\Delta t)$

0 Freq → will always be the sum of the data

Freq	FFT
0	6
1/3	-2+2i
2/3	-2
3/3	-2-2i

(6) One of the more confusing aspects of the discrete/fast Fourier transform is that on the first $N/2$ frequencies correspond to positive frequencies, the next $N/2$ frequencies are actually *negative frequencies*. To see why this is, we'll work with a signal consisting of $N = 8$ data points. Recall that the discrete Fourier transform is,

$$g(n\Delta\omega) = \sum_{m=0}^{N-1} f(m\Delta t) e^{-i2\pi mn/N}.$$

Use a general function, $f(t)$ sampled so that the signal is represented by $f(m\Delta t) \equiv f(m)$. Write out each term in the sum (again, $N = 8$). What do you notice for $m = 4$ and above.

equation 5.73

(7) Both the *FFT* and *DFT* are often misused because the novice user does not pay attention to important details about sampling. For example, download the data file `periodic.dat` from *Teams*. Plot the data. Now suppose that this is the "real signal" but your sampling rate is such that only every 8th point is sampled. Plot this signal and discuss at your table what has happened. Is there a general way to get around this issue.

(8) Start problem 6. Use a sampling frequency of $\pi/4$.