Here's some background info for you, since today's material was not covered in the book.

When we evaluate real signals in a lab, the computer (oscilloscope, graduate student, etc.) only takes data at discrete times. So rather than a continuous function from $t = -\infty$ to $\infty$, we have a finite list of numbers taken at specific times. We can still write our data as a sum of sines and cosines or complex exponentials, but because we have discrete points, there will be a natural limit to the highest frequency in that sum. To see why, look at the images below.

In the upper left hand image, we see a sine wave (the black line) which is being sampled at a fairly fast rate (the round circles represent data points). If we connect the data points, we have something that oscillates at the frequency of the sine wave. Moving to the upper right-hand image, the sample rate slows down to twice the frequency of the sine wave. This is just fast enough to resolve the sine wave. If we sample any slower - as seen in the lower left-hand image, we get data that appears to be oscillating - but at the wrong frequency! This is known as "aliasing." Our data makes it look like a frequency is there which isn't. If we sample at exactly the frequency of the sine wave (the lower right-hand image), we get something that looks like a zero-frequency wave.

So what did we learn from this?

- First, we learned that if the thing we are measuring has frequency components higher than half the sample rate, we will get "artifacts" due to "aliasing" when we do our transform - we will see power appear at lower frequencies when the power is really at higher frequencies. So if you are doing your Fourier transform to try to see what frequencies are in your signal, you could be fooled.

- Second, we learned that if I take data points spaced by a time interval $\tau$ the maximum frequency component that we can extract from our data is a frequency of $v_{max} = 1/2\tau$. So when I write my data as a sum of sines and cosines or complex exponentials, there will be a finite number of terms in my sum.

The next problem that we have with "real data" is that we only take data for a finite amount of time. As such, we never really know if the wave is truly periodic. So in order to perform a Fourier transform, we have to assume something about what the signal does outside of our sample window. The normal thing to do is to assume that it is periodic.

Despite these limitations, there are "fixes" which allow us to pull a spectrum out of a finite set of discrete data points. But while the spectrum we get is useful, we have to remember that it isn't an "exact" representation.
of the thing that we measured. The Fourier transform will, however, be an exact representation of our data. The problem isn’t with the transform, it’s with our data! So, for whatever data we’ve got, we can exactly represent our incomplete data with a Fourier transform.

If I have \( N \) data points, each taken a time \( \tau \) after the previous point, and if I label each data point as \( f_k \) where \( k = 0, 1, 2, \ldots, N - 1 \), we can express that data set in the form

\[
f_k = \sum_{n=-N/2}^{N/2} c_n e^{2\pi i nk/N}.
\]

Or, we can recognize that \( k = t/\tau \), and that \( \omega_0 = 2\pi/T = 2\pi/N\tau \Rightarrow N = 2\pi/\omega_0\tau \), and write this as

\[
f(t) = \sum_{n=-N/2}^{N/2} c_n e^{in\omega_0 t}.
\]

Of course, this is only valid for the times at which we took the data. At other times, this function will “interpolate.” The expression that tells us what the \( c_n \) coefficients are\(^2\) is

\[
c_n = \frac{1}{N} \sum_{k=0}^{N-1} f_k e^{-2\pi i nk/N}.
\]

This process is known as a “discrete Fourier transform,” or DFT.

1. Use some mathematical package on the computer to do the following. (Hint, save your code, because you’ll reuse it in the next problems. If you do this, the rest of the problems should go very quickly. Also, I recommend that you use MATLAB or GNU Octave, the free program with similar syntax to MATLAB. They are more geared to numerical work, unlike Mathematica and Maple which are more geared to symbolic math. But you can use whatever you’d like.)

   (a) Define an array of numbers \( f_k = \sin(16\pi k/N) \) where \( N = 100 \) and \( k = 0, 1, 2, 3, \ldots, 99 \). Plot \( f_k \) and attach the plot to your homework.

   (b) Since you have a pure sine wave with an amplitude of 1, when you perform the DFT you should get a bunch of \( c_n \) terms that are zero, and only the ones corresponding to the correct frequency will be non-zero. Which \( c_n \) terms should be non-zero?

   (c) Have the computer calculate an array of numbers \( c_n \), and then plot \( |c_n| \) as a function of \( n \). Attach a printout of your code and the plot to your homework.

   (d) Now do the inverse transform, and plot the real and imaginary parts of the resulting \( f_k \) array.

2. Now let \( f_k = \sin(14.7\pi k/N) \).

   (a) Plot \( f_k \).

   (b) Find \( c_n \) and plot \(|c_n|\) vs \( n \). Notice that this time, there is a spread of \( c_n \)’s that are non-zero. Why?

   (c) Do the inverse transform and plot the real and imaginary parts of the resulting \( f_k \) array.

\(^2\)Most people don’t put the \( 1/N \) term in the transform, but instead put it into the inverse transform (and some, for the sake of symmetry, put a \( 1/\sqrt{N} \) in both the transform and the inverse transform). But if you do this, the \( c_n \) terms doesn’t actually tell you the amplitude of a given frequency. I’m not exactly sure why they do this, but I suspect it is because most software that performs Fourier transforms are set up to use integer rather than floating point numbers (for increased speed). If you wait until the end to divide by \( N \), that would reduce round-off error. Since we’re more interested in understanding than computer algorithm speed and accuracy, I’ll stick to the more intuitive form, which is more like the Fourier transforms we’ve seen before. Also note that most DFTs have \( n \) go from 0 to \( N - 1 \), rather than \(-N/2\) to \( N/2 \). This works, and makes the transform and inverse transform look more alike. But once again, for the sake of understanding what is going on, I’ll stick with the latter.
3. Now let’s see what happens when we get frequencies close to half the sample rate.

(a) Let \( f_k = \sin(98\pi k/N) \). Plot \( f_k \) and \( |c_n| \). Note that even though \( f_k \) doesn’t look much like a sine wave, \( c_n \) is still smart enough to find the sine wave and give you the right frequency.

(b) Now let \( f_k = \sin(120\pi k/N) \). Plot \( f_k \) and \( |c_n| \). Note that we get the spikes in \( |c_n| \) at the wrong frequency! This is aliasing, my friend. Also note, however, that the inverse transform is (within round-off error) the same as \( f_k \). The problem is NOT with the transform, but with the data you started with.

4. Now let’s see how Fourier transforms can pull a signal out of a mountain of noise. Let \( f_k = 0.2 \sin(30\pi k/N) + 0.3 \sin(14\pi k/N) + b_k \) where \( b_k \) is just a string of random numbers such that each \( b_k \) has a random value between -0.5 and 0.5.

(a) Plot \( f_k \).

(b) Calculate and plot \( |c_n| \).

5. Now let’s convert from our complex exponential transform back into the sine+cosine transformation.

(a) Let \( f_k = \sin(20\pi k/N) \). Calculate \( c_n \), and plot the real and imaginary parts of \( c_n \) as a function of \( n \).

(b) Now use \( c_n \) to find \( a_n \). Plot the real and imaginary parts of \( a_n \) vs. \( n \). Remember that, although the \( n \) in \( c_n \) goes from \(-N/2\) to \( N/2\), for \( a_n \) and \( b_n \) it goes from 0 to \( N/2\).

(c) Now use \( c_n \) to find \( b_n \). Plot the real and imaginary parts of \( b_n \) vs. \( n \).

6. Repeat problem 5, only now let \( f_k = \cos(20\pi k/N) \).