Course reader: *The discrete Fourier transform*

- The discrete-time Fourier transform involves computing the dot product between complex sine waves and the signal. There are as many complex sine waves as there are data points. The resulting complex dot products are called "Fourier coefficients," and from that complex number you extract the magnitude (*amplitude*, and you can square it to get *power*) and angle relative to the positive real axis (*phase*).

- Here is the procedure to implement the discrete Fourier transform.
  1. Create a time vector for the sine wave, which goes from 0 to 1-dt (dt=sample time step, e.g., 1 ms if the sampling rate is 1 kHz) in the number of steps corresponding to the number of time points in the signal.
  2. Create a loop over time points.
  3. Inside that loop, create a complex sine wave as $e^{-i2\pi ft}$ where $i = \sqrt{-1}$, f is the frequency and is set by the looping index minus 1 (so the first frequency is 0), and t is the time vector defined previously.
  4. Compute the dot product between that complex sine wave and the signal. Notice that the signal doesn't change inside the loop, only the frequency of the complex sine wave. The resulting complex dot product is the Fourier coefficient for this frequency.

- It's really that simple. The discrete Fourier transform can also be implemented using matrices, but thinking about the procedure as a loop has some conceptual advantages, and requires no linear algebra background knowledge.

- The "raw" frequencies are indices, not meaningful units such as Hz. To convert from indices to frequencies, compute a linearly spaced vector of N/2+1 numbers (where N is the number of time points) between 0 and the Nyquist (1/2 of the data sampling rate).

- The frequencies can be organized into four groups: 0 Hz (often called "DC" for direct current), "positive frequencies" (just above DC and below Nyquist), "Nyquist" (1/2 the sampling rate), and "negative frequencies" (above the Nyquist).

- A real-valued signal has amplitudes split between the positive and negative frequencies. This is a result of the cosine identity, which states that a real-valued cosine can be made from the combination of two complex exponentials: one with a negative exponent and one with a positive exponent ($\cos(k) = .5(e^{ik} + e^{-ik})$).

- This means that to reconstruct the amplitude of the original signal, you double the amplitudes of the positive frequencies and ignore the negative frequencies.

- This is actually one of two normalization factors you need to apply to recover accurate amplitudes from the Fourier transform. The other factor is to divide the Fourier coefficients by N (number of time points). The reason is that the dot product involves point-wise multiplications and sum, and thus *longer* signals have *larger* dot products. Dividing by N scales the coefficients down to account for all the summing.

- The DC component reflects the mean offset (also sometimes called a global offset or shift) of the

entire signal. If the signal fluctuates around zero such that the positive and negative values cancel, then the DC will be zero. Note that because 0 has no corresponding negative frequency, you *don't* double the amplitude of the DC the way you double the positive frequencies.

- These two normalizations are necessary *only if you are interpreting the amplitude/power results of the Fourier transform!* If you are using the Fourier transform as a tool in signal processing (e.g., filtering or convolution) then you do not apply either of these normalizations.

- The relationship between the *amplitude* and the *power* spectrum is that power is amplitude squared. Squaring all of the values has the effect of making the relatively large spectral features even larger. I prefer using the amplitude spectrum for teaching because it is easier to confirm visually that the results of the Fourier transform match the simulated signal.

Exercises

1. By now, you've read the description of how the Fourier transform works and have seen it in code. Open a new script in MATLAB or Python and program the loop-based discrete Fourier transform from scratch. Test it using a simple signal, like a sine wave, where you can confirm the accuracy of your code.

2. What are the four main groups of frequencies resulting from the Fourier transform?

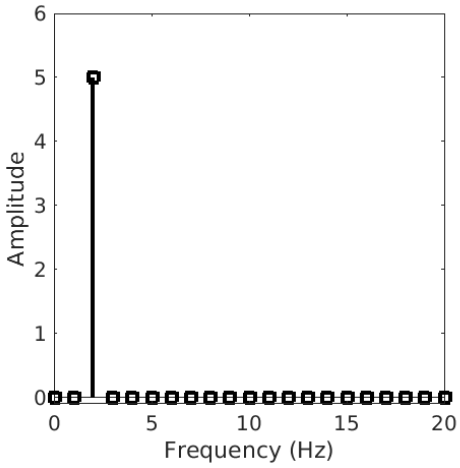
3. If you perform a Fourier transform of a signal that contains 200 time points sampled at 100 Hz, what is the highest frequency (in hertz) that you can reconstruct? What would the highest frequency be if you had 400 time points?

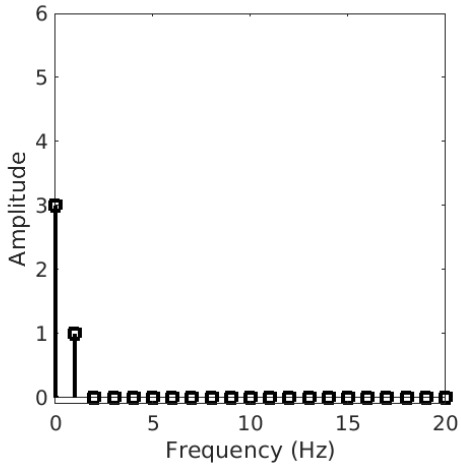
4. What are the two normalizations for the positive-frequency Fourier coefficients for the resulting amplitudes to be in the same scale as the original signal?

5. One of those normalizations involves a division. What is the justification for that normalization?

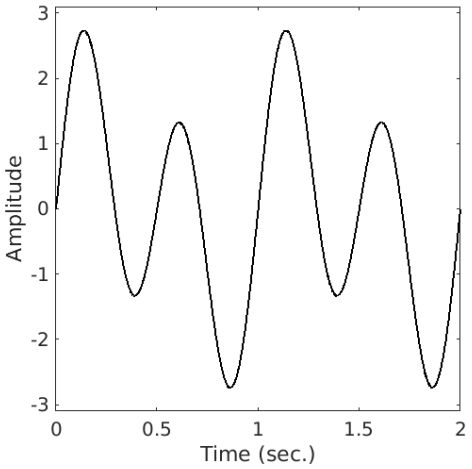
6. You can compute power either as $\|x\|^2$ or as $x\overline{x}$, where x is a Fourier coefficient. Translated into MATLAB code, those two operations are, respectively, `abs(x)^2` and `x*conj(x)` (in Python: `np.abs(x)**2` and `x*np.conj(x)`). I use the former when teaching because of the clear link to geometry. But is one faster than the other? To find out, generate a 10,000-point random signal, compute the Fourier transform, and use these two methods to compute power. Because this is a scientific experiment, you should run the procedure many times on different vectors, and then average the resulting computation times together. Show the resulting average clock times in a bar plot.

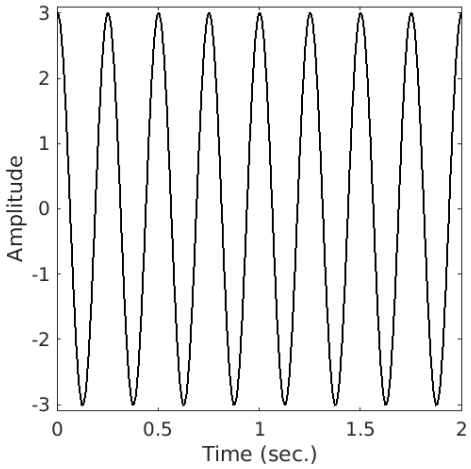
7. Draw time-domain signals that would have the following power spectra (note that without knowing the phase, your answers should focus on frequency and amplitude).





8. Draw amplitude spectra that would correspond to the following time-domain signals.





MATLAB Answers

Scroll down for Python solutions.

1. The MATLAB and Python codes for this section show correct answers.

2. 0 (or DC for direct current), positive (between but not including 0 and Nyquist), Nyquist, and negative.

3. 50 Hz. The answer is always 50 Hz because the highest frequency is determined by the sampling rate, not the number of time points. The frequency resolution increases, though.

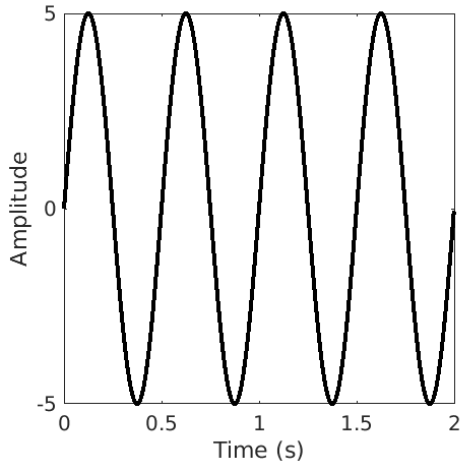
4. Divide by N (the number of sample points) and multiply by 2.

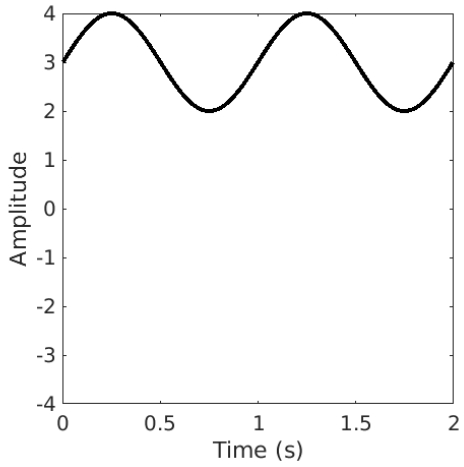
5. The dot product involves a lot of summing. Thus, the longer the signal, the larger the Fourier coefficient. Dividing by N accounts for that growth.

6. I'm kindof cheating in my solution by using the `fft` function, so you can insert the loop-based Fourier transform instead.

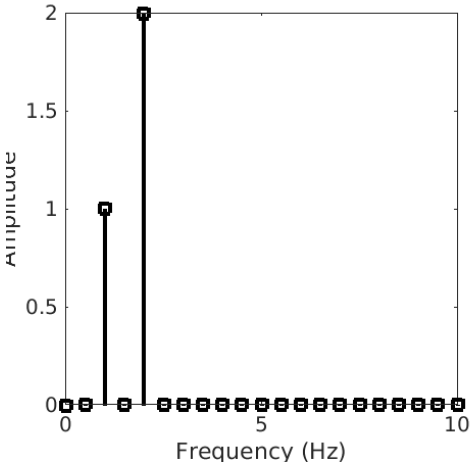
```
tim = zeros(2,100);

for i=1:100
   sx = fft(randn(10000,1));
   tic; powr=abs(sx).^2;   tim(1,i)=toc;
   tic; powr=sx.*conj(sx); tim(2,i)=toc;
end

bar(mean(tim,2)), hold on
errorbar(mean(tim,2),std(tim,[],2),'o')
ylabel('Computation time (s)')
set(gca,'xlim',[0 3],'xticklabel',{'abs(sx)^2';'sx.*conj(sx)'})
```

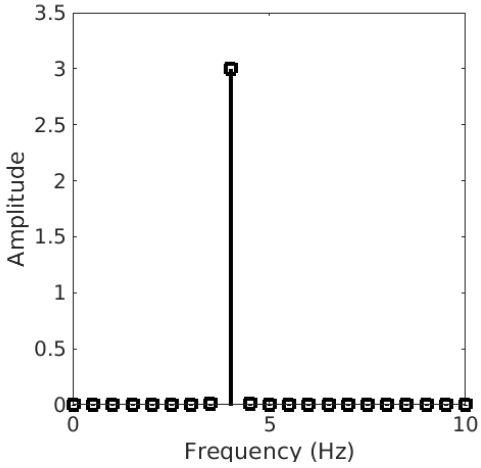
7. The question left some details out, like how long to make the signal, the phase, etc. But your answer should look a bit like these.





8. Less ambiguity for this problem.





Python Answers

These answers are identical to the previous pages but with Python code.

1. The Python codes for this section show correct answers.

2. 0 (or DC for direct current), positive (between but not including 0 and Nyquist), Nyquist, and negative.

3. 50 Hz. The answer is always 50 Hz because the highest frequency is determined by the sampling rate, not the number of time points. The frequency resolution increases, though.

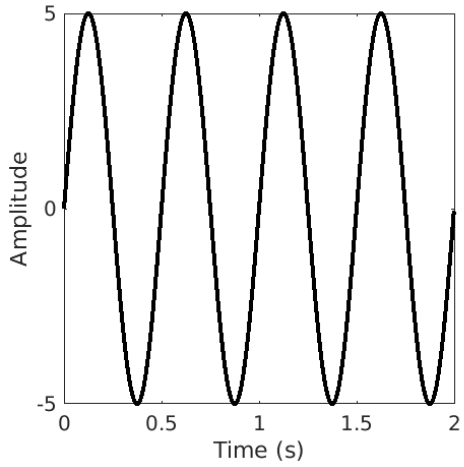
4. Divide by N (the number of sample points) and multiply by 2.

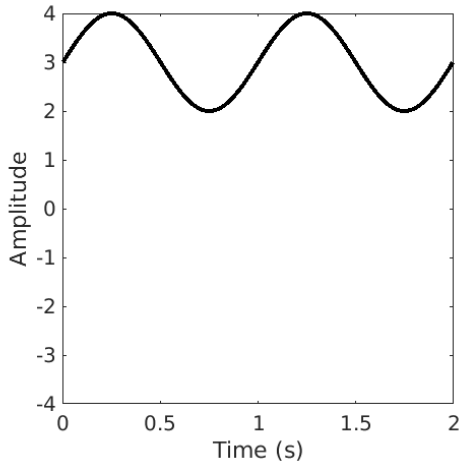
5. The dot product involves a lot of summing. Thus, the longer the signal, the larger the Fourier coefficient. Dividing by N accounts for that growth.

6. I'm kindof cheating in my solution by using the `fft` function, so you can insert the loop-based Fourier transform instead.

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

tim = np.zeros((2,100))

for i in range(100):
   sx = np.fft.fft(np.random.randn(10000,1))

   st = time.time()
   powr = np.abs(sx)**2
   tim[0,i] = time.time() - st

   st = time.time()
   powr = sx*np.conj(sx)
   tim[1,i] = time.time() - st


plt.bar([0,1],np.mean(tim,1))
plt.errorbar([0,1],np.mean(tim,1),np.std(tim,1),marker='o',linestyle='none')
plt.ylabel('Computation time (s)')
plt.xticks([0,1],labels=('abs(sx)**2','sx*np.conj(sx)'))
plt.show()
```

7. The question left some details out, like how long to make the signal, the phase, etc. But your answer should look a bit like these.





8. Less ambiguity for this problem.