Course reader: *The fast Fourier transform*

- The implementation of the Fourier transform you've seen so far is really slow. You don't notice how slow it is for a signal of 100s of points, but in practice, people take the Fourier transform of signals that are billions of points long. Running through the loop can take hours or days of computation time!

- The point is, the loop-based implementation of the Fourier transform is great for learning, **but don't ever use it in practice!**

- The fast Fourier transform (FFT) is much faster, and it's what everyone uses. There are several FFT algorithms, but they generally work by (1) putting all of the complex sine waves into a matrix, (2) decomposing that matrix into a lot of other matrices that are sparse (meaning they contain mostly zeros), and then (3) performing a series of matrix-matrix and matrix-vector multiplications. The FFT algorithm is not so difficult to understand, but it requires some background knowledge about linear algebra, and therefore I won't go into it here. Anyway, the important thing is that once you understand the Fourier transform, you can safely apply the FFT function and get the same results.

- The fast inverse Fourier transform is called the IFFT.

- The two normalization steps you need to apply are the same as for the loop-based Fourier transform.

- There are at least two ways to understand why the Fourier transform is a lossless transform (and thus, why the inverse Fourier transform is a perfect reconstruction of the original measured signal).
  1. *Linear algebra*: When the complex sine waves are columns in a matrix, it is an $N \times N$ matrix with linearly independent columns, meaning it has an inverse. So if the sine waves are in matrix $\mathbf{F}$ and the signal is in row vector $\mathbf{s}$, then $\mathbf{sF} = \mathbf{c}$ gives the Fourier coefficients in row vector $\mathbf{c}$. Once you have the coefficients vector, you can compute $\mathbf{s} = \mathbf{cF^{-1}}$. Essentially, you are moving $\mathbf{s}$ from the standard bases into the "Fourier bases" which are sine waves.
  2. *Statistics*: Think of the time series signal as a dependent variable and the sine waves as independent variables (predictors). The 0 frequency component is the intercept. The collection of independent variables is the model, and the Fourier coefficients are the regression coefficients (beta values). The thing is that the model has N predictors for N data points, and therefore has 0 degrees of freedom. A model with 0 degrees of freedom accounts for 100% of the variance of the data. (Technically, this is really just a reinterpretation of the linear algebra statement, because the equation for the statistics interpretation is the general linear model: $\mathbf{Xs} = \boldsymbol{\beta}$)

Exercises

1. One method of scrambling a signal is to shuffle the phases of the Fourier coefficients and then reconstruct the time-domain data. In this exercise, generate a linear chirp and compute its inverse after shuffling the phases (do not alter the power values). Plot the time domain signals and their spectra. Is the scrambled time series recognizable as having been a chirp?

2. Real data often contain noise. One way to reduce noise is to average over more data; if the noise is random, then it will decrease with increased averaging. This and the next several exercises will explore this idea.

- Generate a signal comprising three sine waves at three frequencies with three different amplitudes. You can pick the parameters, but make sure the signal contains only peaks at the frequencies, and not at neighboring frequencies due to a mismatch in the frequencies, time ranges, and frequency resolution.
- Generate 50 noisy repetitions of the signal by creating a new instance of the signal and adding random noise. It's most convenient if you store the data in a $50 \times time$ matrix. Make sure to use a sufficient amplitude of noise so the signals look noisy.

3. Show three power spectra from these data: (1) average over all repetitions in the *time domain* and then computing one FFT; (2) compute the FFT of each repetition (thus 50 FFTs, note that you can do this by inputting a matrix into the `fft` function, just make sure the FFT is computed along the time dimension!), average the coefficients together and then extract power; (3) same as #2 except first extract power for each repetition and then average the power spectra together. Comment on any differences in the results.

4. Rerun the previous simulation using a lot of noise or a little bit of noise (by adjusting the multiplication of `randn` by a large or small number), and larger or small signal (by multiplying the signal by a large or small number). Obviously, large signal and low noise is the best scenario, but is it better to have large signal but large noise, or small signal and small noise? "Better" is a subjective term, but you can think of it as the visual interpretability of the power spectrum.

5. Change the simulation generation from question #2 by creating the three sine waves with random phases (between 0 and 2π) on each repetition. Then re-run question 3. What do you notice?

MATLAB Answers

Scroll down for Python solutions.

1. Here's my solution:

```
% first, the basics:
srate= 1000;
t    = 0:1/srate:6;
N    = length(t);
f    = [1 5]; % in hz
ff = linspace(f(1),mean(f),N);
data = sin(2*pi.*ff.*t);

% second, compute Fourier transform
dataX = fft(data);

% third, shuffle phases (here, shifted by 10)
phases = angle([dataX(10:end) dataX(1:9)]);
shuffdata = abs(dataX).*exp(1i*phases);

% fourth, reconstruct the signal
newdata = ifft(shuffdata);

% fifth, plot the results
subplot(211)
plot(t,data), hold on
plot(t,real(newdata),'r')

subplot(212)
hz = linspace(0,srate/2,floor(N/2)+1);
plot(hz,2*abs(dataX(1:length(hz))/N),'o'), hold on
plot(hz,2*abs(shuffdata(1:length(hz))/N),'r')
set(gca,'xlim',[0 20])
```

2. Here is my code.

```
% simulation parameters
nReps = 50;
srate = 1000;
t = 0:1/srate:3-1/srate;
n = length(t);

% amplitude modulators for signal and noise
noiseAmp  = 50; % or try 500
signalAmp = 1;  % or try 10

% signal parameters
a = [2 3 4 2];  % amplitudes
f = [1 3 6 12]; % frequencies

% initialize data matrix
data = zeros(nReps,n);

% create signal
signal = zeros(1,n);
for i=1:length(a)
    signal = signal + a(i)*sin(2*pi*f(i)*t);
end

% modulate amplitude
signal = signal*signalAmp;

% create trials with noise
for repi=1:nReps
    data(repi,:) = signal + noiseAmp*randn(size(signal));
end

% spectrum 1: time-domain averaging
spect1 = 2*abs(fft( mean(data,1) )/n).^2;

% spectrum 2: Fourier coefficient averaging
spect2 = 2*abs( mean(fft(data,[],2)/n ,1)).^2;

% spectrum 3: power averaging
spect3 = mean( 2*abs( fft(data,[],2)/n ).^2,1);

% frequencies
hz = linspace(0,srate/2,floor(n/2)+1);
```

```
% plot
figure(1), clf
plot(hz,spect1(1:length(hz)),'bs-','linew',2,'markersize',10)
hold on
plot(hz,spect2(1:length(hz)),'ro-','linew',2,'markersize',10)
plot(hz,spect3(1:length(hz)),'kp-','linew',2,'markersize',10)

legend({'time ave';'coef ave';'power ave'})
set(gca,'xlim',[0 20])
xlabel('Frequency (Hz)'), ylabel('Power')
```

3. The thing to realize about normally distributed noise is that *it averages to zero only in the time domain*. In the frequency domain, noise has a positive amplitude spectrum. Indeed, everything has a positive amplitude spectrum because amplitude is the length of a line, and thus is a nonnegative quantity. So, in the time domain, you average positive and negative numbers, which tend towards zero; in the frequency domain, you average only positive values, so they necessarily cannot tend towards zero.
On the other hand, you might be surprised to see how much the "best" result here differs from the "best" result for question 5. The point is that in practical applications, there is rarely a single answer that is optimal for all problems. The best solution needs to take the features of the data into consideration.

4. There is no single correct answer, because it depends on the nature of the signal and the type of noise. But if you are working with empirical data, it's important to think about whether you can boost signal or suppress noise.

5. You have to re-create the signal on each repetition, like this:

```
% create trials with noise and repetition-unique signal phase
for repi=1:nReps
    % create signal
    signal = zeros(1,n);
    for i=1:length(a)
        signal = signal + a(i)*sin(2*pi*f(i)*t + rand*2*pi);
    end
    data(repi,:) = signal + noiseAmp*randn(size(signal));
end
```

Python Answers

Note: These answers are identical to those in the previous pages, but with Python code.

1. Here's my solution:

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

# first, the basics:
srate= 1000
t    = np.arange(0,6,1/srate)
N    = len(t)
f    = [1,5] # in hz
ff   = np.linspace(f[0],np.mean(f),N)
data = np.sin(2*np.pi*ff*t)

# second, compute Fourier transform
dataX = np.fft.fft(data)

# third, shuffle phases (here, shifted by 10)
phases = np.angle(np.concatenate((dataX[10:],dataX[:10])))
shuffdata = np.abs(dataX) * np.exp(1j*phases)

# fourth, reconstruct the signal
newdata = np.fft.ifft(shuffdata)

# fifth, plot the results
_,axs = plt.subplots(2,1)
axs[0].plot(t,data)
axs[0].plot(t,np.real(newdata),'r')
axs[0].set_xlabel('Time (s)')

hz = np.linspace(0,srate/2,int(np.floor(N/2)+1))
axs[1].plot(hz,2*np.abs(dataX[:len(hz)]/N),'o')
axs[1].plot(hz,2*np.abs(shuffdata[:len(hz)]/N),'r')
axs[1].set_xlim([0,20])
axs[1].set_xlabel('Frequency (Hz)')

plt.tight_layout()
plt.show()
```

2. Here is my code.

```
# simulation parameters
nReps = 50
srate = 1000
t = np.arange(0,3-1/srate,1/srate)
n = len(t)

# amplitude modulators for signal and noise
noiseAmp  = 50  # or try 500
signalAmp =  1  # or try 10

# signal parameters
a = [2, 3, 4, 2]  # amplitudes
f = [1, 3, 6, 12] # frequencies

# initialize data matrix
data = np.zeros((nReps,n))

# create signal
signal = np.zeros(n)
for i in range(len(a)):
    signal = signal + a[i] * np.sin(2*np.pi*f[i]*t)


# modulate amplitude
signal = signal*signalAmp

# create trials with noise
for repi in range(nReps):
    data[repi,:] = signal + noiseAmp*np.random.randn(len(signal))


# spectrum 1: time-domain averaging
spect1 = 2*np.abs(np.fft.fft( np.mean(data,axis=0) )/n)**2

# spectrum 2: Fourier coefficient averaging
spect2 = 2*np.abs( np.mean(np.fft.fft(data,axis=1)/n ,axis=0))**2

# spectrum 3: power averaging
spect3 = np.mean( 2*np.abs( np.fft.fft(data,axis=1)/n )**2,axis=0)

# frequencies
hz = np.linspace(0,srate/2,int(np.floor(n/2)+1))
```

```
# plot
plt.plot(hz,spect1[:len(hz)],'bs-',markersize=10)
plt.plot(hz,spect2[:len(hz)],'ro-',markersize=10)
plt.plot(hz,spect3[:len(hz)],'kp-',markersize=10)

plt.legend(['time ave','coef ave','power ave'])
plt.xlim([0,20])
plt.xlabel('Frequency (Hz)')
plt.ylabel('Power')
plt.show()
```

3. The thing to realize about normally distributed noise is that *it averages to zero only in the time domain*. In the frequency domain, noise has a positive amplitude spectrum. Indeed, everything has a positive amplitude spectrum because amplitude is the length of a line, and thus is a nonnegative quantity. So, in the time domain, you average positive and negative numbers, which tend towards zero; in the frequency domain, you average only positive values, so they necessarily cannot tend towards zero.
On the other hand, you might be surprised to see how much the "best" result here differs from the "best" result for question 5. The point is that in practical applications, there is rarely a single answer that is optimal for all problems. The best solution needs to take the features of the data into consideration.

4. There is no single correct answer, because it depends on the nature of the signal and the type of noise. But if you are working with empirical data, it's important to think about whether you can boost signal or suppress noise.

5. You have to re-create the signal on each repetition, like this:

```
# create trials with noise and repetition-unique signal phase
for repi in range(nReps):
    # create signal
    signal = np.zeros(n)
    for i in range(len(a)):
        signal = signal + a[i]*np.sin(2*np.pi*f[i]*t + np.random.rand()*2*np.pi)

    data[repi,:] = signal + noiseAmp*np.random.randn(len(signal))
```