Course reader: *The discrete* inverse *Fourier transform*

- What you learned so far is called the *forward* Fourier transform, because that gets you from the time domain to the frequency domain. Getting back from the frequency domain to the time domain is implemented via the *inverse* Fourier transform.

- Here is the procedure to implement the discrete inverse 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. *Notice the lack of minus sine in the exponential compared to the forward Fourier transform!*
  4. Multiply that sine wave by the complex Fourier coefficient that you obtained from the forward Fourier transform. Then sum all sine waves together.

- The presence/absence of the negative sign in the forward/inverse Fourier transform is key. That's what allows the imaginary parts to cancel so a real-valued signal will still be real-valued after the inverse Fourier transform.

- The inverse Fourier transform is a perfect reconstruction of the original time-domain signal, which will be explained in the video called "The perfection of the Fourier transform."

- On the other hand, there are sometimes small computer rounding errors or uncertainties that cause small imaginary components in the reconstruction. That's why the `real` function is occasionally used even when it shouldn't (in theory) be necessary.

- I think of the Fourier transform as creating a bunch of blank templates (the complex sine waves). The goal of the forward Fourier transform is to find the coefficient to make that template best fit the data. And the goal of the inverse Fourier transform is to modulate those templates using that best-fit coefficient.

- The inverse Fourier transform has considerable importance in applied math, including signal processing and data analysis. The idea is that you can compute the Fourier transform of a signal, modify the Fourier coefficients, and then compute the inverse Fourier transform.

- For example, if you have signals mixed together on the same channel and want to isolate one signal, you can take the Fourier transform, attenuate the coefficients of frequencies you are not interested in, then compute the inverse Fourier transform to get back to the time domain.

- You should keep in mind that this spectral signal separation approach is not generally useful for any kinds of mixed signals. It works well only when the mixed signals are clearly separable in the frequency domain. There are many other signals that are not spectrally separable (e.g., people talking at the same time), and narrowband filtering is unlikely to be a good solution.

- Two examples of spectral separability of overlapping signals are (1) radio and (2) electrical line noise ("mains" noise) at 50 or 60 Hz.

Exercises

1. Open a new script in MATLAB or Python and program the loop-based discrete inverse Fourier transform from scratch. Test it by generating a random vector, taking the forward Fourier transform, then the inverse Fourier transform, and then compare the original to the reconstructed signal.

2. In a previous section, I mentioned two normalizations to apply to the output of the Fourier transform when interpreting amplitude or power. Do you still need to normalize when applying the inverse Fourier transform? To find out, generate a signal, take its Fourier transform, and then apply the inverse Fourier transform. Compute the inverse twice: Once without normalizing the Fourier coefficients before computing the inverse, and once with normalizations before computing the inverse.

3. Create a 1-second sine wave at 5 Hz using 100 Hz sampling rate. This signal will be used for the remaining exercises. Compute the Fourier transform and obtain two copies of the Fourier coefficients—one for plotting, and the other for modulating. Then plot the time-domain signal in the top panel and the amplitude spectrum on the bottom panel.

4. Change the phase of the 5 Hz component while preserving the amplitude. To do this, you need to transform the 5 Hz Fourier coefficient to its Euler notation ($ae^{i\theta}$), then change θ and then replace the 5 Hz coefficient in the extra copy of the Fourier coefficients. (Hint: don't forget about the negative frequencies!) Plot the two time-domain signals (original and reconstructed) on top of each other.

5. Comment out the line where you modulate the negative frequency component, and set the phase of $+5$ Hz to be $\pi/2$. What happens to the reconstructed signal and why? Try it again with the phase set to $\pi/4$.

6. Repeat questions 4 and 5 but simulate the signal for 0.9 seconds instead of 1.0 seconds. How are the results different and why?
7. To understand these effects better, make a plot that shows the Fourier coefficients in the complex plane. Show the positive and negative coefficients for the original and modified spectra.

Python Answers

Scroll down for MATLAB solutions.

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

2. The answer is no, you don't apply the normalizations. There are normalizations for the forward and inverse Fourier transforms, and they cancel each other out when done in succession.

3. This code is for questions 3-6. I'm cheating a bit here by using the `fft` function instead of the loop. I'll talk more about the FFT in the next section, but basically the output of the `fft` and `ifft` functions are identical to the Fourier coefficients and reconstructed time-domain signal from the loop-based Fourier transform.

```
# simulation
srate = 100
time  = np.arange(0,1-1/srate,1/srate)
frex  = 5 # Hz

# create signal and its FFT (and an extra copy)
signal  = np.sin(2*np.pi*frex*time)
sX = np.fft.fft(signal)
sXmod = np.fft.fft(signal)

# frequency vector
hz = np.linspace(0,srate/2,int(len(time)/2+2))

# find the coefficient for target frequency
fidx = np.argmin(np.abs(hz-frex))

# this is the new phase value
newphase = np.pi/2

# modulate positive and negative frequency components
sXmod[fidx] = np.abs(sX[fidx]) * np.exp(1j*newphase)
sXmod[-fidx] = np.abs(sX[-fidx]) * np.exp(-1j*newphase)

# new signal from inverse FFT
newsig = np.real( np.fft.ifft(sXmod) )

# and plot
_,axs = plt.subplots(2,1)
axs[0].plot(time,signal,label='orig')
axs[0].plot(time,newsig,'o',label='phase-modulated')
axs[0].set_xlabel('Time (s)')
axs[0].legend()
```

```
axs[1].stem(hz,2*np.abs(sX[:len(hz)]/len(time)),use_line_collection=True)
axs[1].set_xlabel('Frequency (Hz)')
axs[1].set_xlim([0,10])

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

4. The magnitude is the same; only the phase changes. The tricky part here is that you need to set $+\theta$ for the positive frequencies and $-\theta$ for the negative frequencies (see the code on the previous page).

5. The flat line for $\theta = \pi/2$ is because the negative 5 Hz component is also $\pi/2$, and these subtract away to give zero at the 5 Hz component. Same story for the $\theta = \pi/4$ case, except the positive and negative coefficients only partly cancel, which is why the amplitude is decreased.

6. An integer number of 5 Hz cycles cannot fit into 900 ms, which causes an edge artifact, and that means there needs to be nonzero energy at frequencies other than 5 Hz to represent the signal. In addition, with this sampling rate an number of time points, the Fourier transform cannot resolve exactly 5.0 Hz. So you *are* successfully changing the phase at some frequency (4.44 Hz in my code above), but you haven't changed the phases at the neighboring frequencies. This is a good illustration of the difference between the Fourier transform "in theory" vs. "in practice."

7. Add the following code. Then try running it to plot the coefficients each time you change the Fourier coefficients.

```
# normalize coefficients
sX = sX/len(time)
sXmod = sXmod/len(time)

# plot original coefficients
plt.plot(np.real(sX[fidx]),np.imag(sX[fidx]),'ro',
         markerfacecolor='r',markersize=12,label='Orig +ve')
plt.plot(np.real(sX[-fidx]),np.imag(sX[-fidx]),'rs',
         markerfacecolor='r',markersize=12,label='Orig -ve')

# plot modulated coefficients
plt.plot(np.real(sXmod[fidx]),np.imag(sXmod[fidx]),'ko',
         markerfacecolor='k',markersize=12,label='Mod +ve')
plt.plot(np.real(sXmod[-fidx]),np.imag(sXmod[-fidx]),'ks',
         markerfacecolor='k',markersize=12,label='Mod -ve')
plt.legend()

# make the plot a bit nicer
plt.axis('square')
plt.axis([-1,1,-1,1])
plt.grid('on')
plt.plot([-1,1],[0,0],'k')
```

```
plt.plot([0,0],[-1,1],'k')
plt.xlabel('Real')
plt.ylabel('Imag')
plt.title('Fourier coefficients in the complex plane')

plt.show()
```

MATLAB Answers

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

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

2. The answer is no, you don't apply the normalizations. There are normalizations for the forward and inverse Fourier transforms, and they cancel each other out when done in succession.

3. This code is for questions 3-6. I'm cheating a bit here by using the `fft` function instead of the loop. I'll talk more about the FFT in the next section, but basically the output of the `fft` and `ifft` functions are identical to the Fourier coefficients and reconstructed time-domain signal from the loop-based Fourier transform.

4. The magnitude is the same; only the phase changes. The tricky part here is that you need to set $+\theta$ for the positive frequencies and $-\theta$ for the negative frequencies (see the code on the previous page).

5. The flat line for $\theta = \pi/2$ is because the negative 5 Hz component is also $\pi/2$, and these subtract away to give zero at the 5 Hz component. Same story for the $\theta = \pi/4$ case, except the positive and negative coefficients only partly cancel, which is why the amplitude is decreased.

6. An integer number of 5 Hz cycles cannot fit into 900 ms, which causes an edge artifact, and that means there needs to be nonzero energy at frequencies other than 5 Hz to represent the signal. In addition, with this sampling rate an number of time points, the Fourier transform cannot resolve exactly 5.0 Hz. So you *are* successfully changing the phase at some frequency (4.44 Hz in my code above), but you haven't changed the phases at the neighboring frequencies. This is a good illustration of the difference between the Fourier transform "in theory" vs. "in practice."

7. Add the following code. Then try running it to plot the coefficients each time you change the Fourier coefficients.

```
% normalize coefficients
sX = sX/length(time);
sXmod = sXmod/length(time);

% plot original coefficients
figure(2), clf
plot(real(sX(fidx)),imag(sX(fidx)),'ro','markerfacecolor','r','markersize',12)
hold on
plot(real(sX(end-fidx+2)),imag(sX(end-fidx+2)),'rs','markerfacecolor','r',...
    'markersize',12)

% plot modulated coefficients
plot(real(sXmod(fidx)),imag(sXmod(fidx)),'ko','markerfacecolor','k'...
    ,'markersize',12)
```

```
hold on
plot(real(sXmod(end-fidx+2)),imag(sXmod(end-fidx+2)),'ks','markerfacecolor',...
    'k','markersize',12)
legend({'Orig +ve';'Orig -ve';'Mod +ve';'Mod -ve'})

% make the plot a bit nicer
axis([-1 1 -1 1]), grid on
plot(get(gca,'xlim'),[0 0],'k')
plot([0 0],get(gca,'xlim'),'k')
axis square
xlabel('Real'), ylabel('Imag')
title('Fourier coefficients in the complex plane')
```