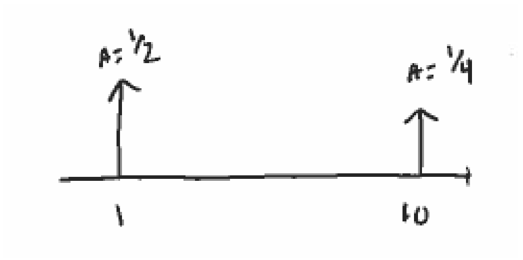
Kyle Flores

Problem 1:

The gunshot sound is close to an ideal impulse, so playing it through the system produces an output similar to the ideal impulse response. By then convolving the ideal impulse response in the time domain against the violin signal, we can apply the impulse response h(t) to the violin signal x(t) to produce h(t)*x(t)=y(t), the output signal which is what the violin would sound like in the gun's room.

As an aside, to achieve a “truer” sound, you may want the impulse response of the room the violin was originally played in to calibrate it out of the final output. In this case, you'd just be dividing the violin sound and the violin room's impulse response.

Problem 2:
If y(t) = 0.5 x(t-1) + 0.25 x(t-10), then h(t) = 0.5 δ(t-1) + 0.25 δ(t-10).



This would sound like an echo because it incorporates lesser weighted copies of the original signal in delay to the output. You would hear the oirignal signal attenuated and delayed 1t late, and then even more attenuated again 9t later.

Problem 3:
a.

3a)

$$C_k = \frac{1}{T}\int_{-T/4}^{T/4} 1 \cdot e^{-J\frac{2\pi}{T}kt}\, dt$$

$$= \frac{1}{T}\cdot\frac{-T}{J2\pi k}\, e^{-J\frac{2\pi}{T}kt}\Big|_{-T/4}^{T/4}$$

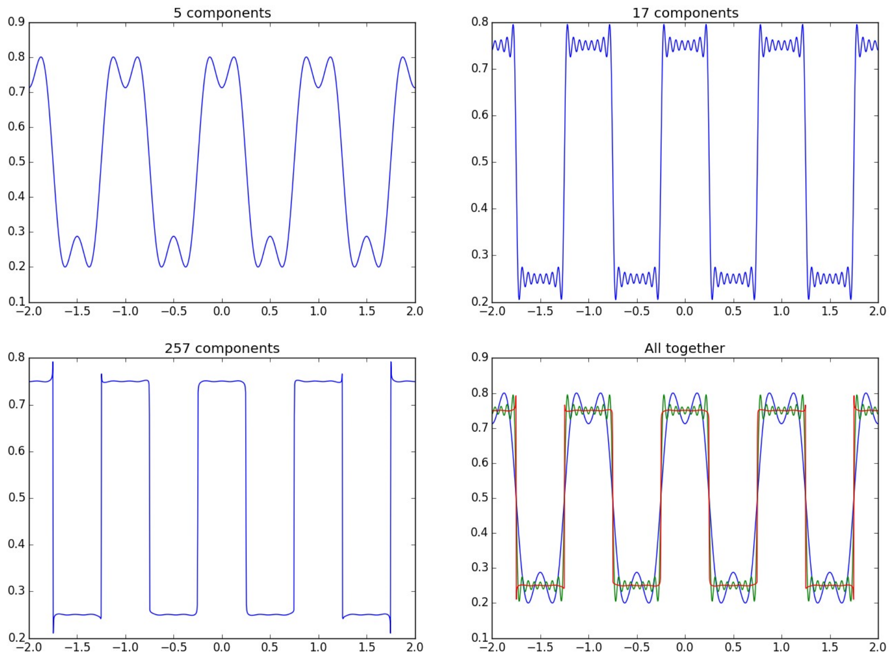
$$= \frac{-1}{J2\pi k}\left(e^{-J\frac{\pi}{2}k\cancel{t}} - e^{J\frac{\pi}{2}k\cancel{t}}\right)$$

$$= \frac{-1}{2J\pi k}\, 2J\sin\left(-\frac{\pi}{2}k\right)$$

$$= \frac{-1}{2}\,\frac{2\sin\left(-\frac{\pi}{2}k\right)}{\pi k}$$

$$= \boxed{\frac{\mathrm{sinc}\left(\frac{k}{2}\right)}{2} = C_k}$$

b.



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

def sq_sum(coefs):
    ks=np.arange(coefs,dtype=np.float)
    C=0.5*(np.sinc(ks/2))

    ts=np.linspace(-2,2,1000,dtype=np.float)
    ys=np.empty_like(ts)
    s=0
    for a in range(len(C)):
        tmp=C[a]*np.exp(-1j*2*np.pi*a*ts)
        s+=tmp

    return (ts,s)

if __name__=='__main__':
    res=sq_sum(5)
    res1=sq_sum(17)
    res2=sq_sum(257)
    plt.close('all')
    f, axarr = plt.subplots(2, 2)
    axarr[0, 0].plot(res[0],res[1])
    axarr[0, 0].set_title('5 components')
    axarr[0, 1].plot(res1[0],res1[1])
    axarr[0, 1].set_title('17 components')
    axarr[1, 0].plot(res2[0],res2[1])
    axarr[1, 0].set_title('257 components')
    axarr[1, 1].plot(res[0],res[1])
    axarr[1, 1].plot(res1[0],res1[1])
    axarr[1, 1].plot(res2[0],res2[1])
    axarr[1, 1].set_title('All together')
    plt.show()
```

c. At the discontinuities, the Fourier series representation cannot produce the immediate sharp change correctly, instead showing a sharp vertical jump just before the wave inverts. This is probably attributable to the idea that we are representing a continuous system with discrete points. Sampling the continuous function is like putting the input through another function (which I think is the Shah function...?) so the output should cannot be the exact result of the continuous integration. Looking at the plots, the extreme jumps on the edges of the discontinuities aren't always visible because actually seeing the jumps is dependent not only on the time steps falling on the increasingly thin spikes, but also on the plotting library actually drawing points in a fine enough resolution.

Problem 4:
a.

4a)

$$C_{k_x} = \frac{1}{T}\int_{-T/2}^{T/2} x(t)\, e^{-j\frac{2\pi}{T}kt}\, dt$$

- - - - - - - -

$t = u - T_1$

$dt = du$

$t = \frac{-T}{2} \rightarrow u = \frac{-T}{2} + T_1$

$t = \frac{T}{2} \rightarrow u = \frac{T}{2} + T_1$

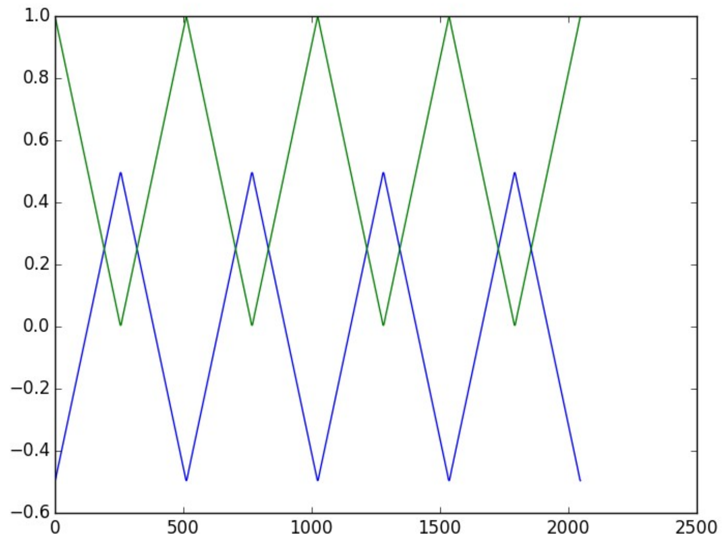
$$C_{k_y} = \frac{1}{T}\int_{\frac{-T}{2}+T_1}^{\frac{T}{2}+T_1} x(u-T_1)\, e^{-j\frac{2\pi}{T}k(u-T_1)}\, du$$

$$C_{k_y} = \frac{e^{2\pi jk\frac{T_1}{T}}}{T}\int_{\frac{-T}{2}}^{\frac{T}{2}} x(t-T_1)\, e^{-\frac{2\pi jk}{T}}\, dt$$

$$\boxed{\cancel{C_{k_y} = e^{2\pi jk T_1/T} C_{k_x}}}$$

$$C_{k_y} = e^{2\pi jk\frac{T_1}{T}} \cdot C_{k_x}$$

b. I found the Fourier coefficients by running the triangle wave code and adding a term to multiply by exp(2*pi*1j*k*T1/T), the complex ratio between each Ck in x and y.



Blue is original wave, green is my converted wave. This transformation requires that T1=2 so that the shift is T/2 (T=4 in the original code). Interestingly, multiplication by complex value that depends on k also biases the signal correctly so that it is between 0 and 1. As I had hoped, it looks like the reference square wave.

CODE (modified parts highlighted in red):

```
def fs_triangle_shift(ts, M=100, T=4, T1=0):
    # computes a fourier series representation of a triangle wave
    # with M terms in the Fourier series approximation
    # if M is odd, terms -(M-1)/2 -> (M-1)/2 are used
    # if M is even terms -M/2 -> M/2-1 are used

    # create an array to store the signal
    x = np.zeros(len(ts))

    # if M is even
    if np.mod(M,2) ==0:
        for k in range(-int(M/2), int(M/2)):
            # if n is odd compute the coefficients
            if np.mod(k, 2)==1:
                Coeff = -2/((np.pi)**2*(k**2))
            if np.mod(k,2)==0:
                Coeff = 0
            if k == 0:
                Coeff = 0.5
            Coeff=Coeff*np.exp(2*np.pi*1j*k*T1/T)
            x = x + Coeff*np.exp(1j*2*np.pi/T*k*ts)

    # if M is odd
    if np.mod(M,2) == 1:
        for k in range(-int((M-1)/2), int((M-1)/2)+1):
           # if n is odd compute the coefficients
            if np.mod(k, 2)==1:
                Coeff = -2/((np.pi)**2*(k**2))
            if np.mod(k,2)==0:
                Coeff = 0
            if k == 0:
                Coeff = 0.5
            Coeff=Coeff*np.exp(2*np.pi*1j*k*T1/T)
            x = x + Coeff*np.exp(1j*2*np.pi/T*k*ts)

    return x

if __name__ == '__main__':
    ts = np.linspace(-8,8,2048)
    fs=fs_triangle(ts)
    fs_shift=fs_triangle_shift(ts,T1=2)
    plt.plot(fs)
    plt.plot(fs_shift)
    plt.show()
```