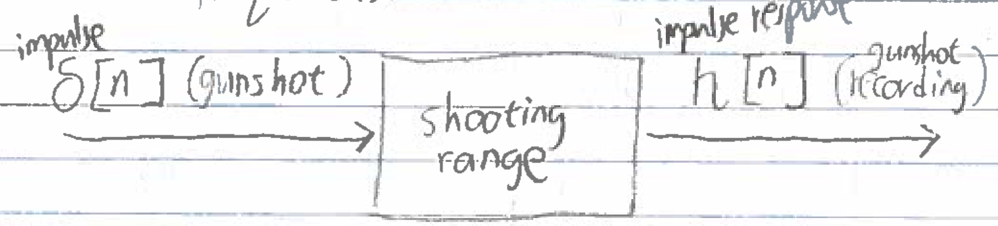
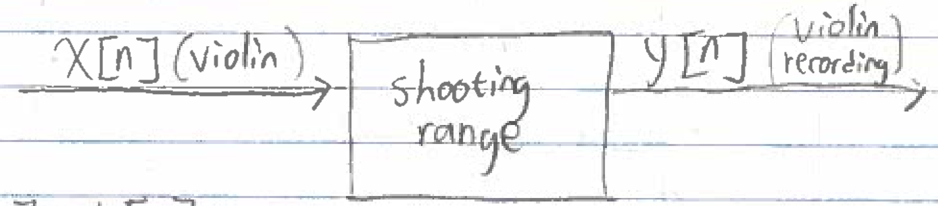
Dennis Chen

PS06

1. An impulse response to an impulse characterizes an LTI system & tells you how the system will amplify or kill off every frequency. So, convolving this response w/ any signal tells you what the output of the system will be when that signal is put in, since the impulse response gave you info about all frequencies.

Basically,

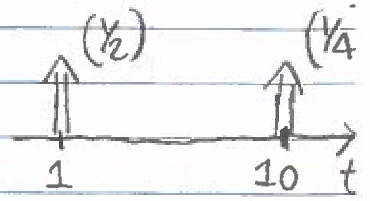


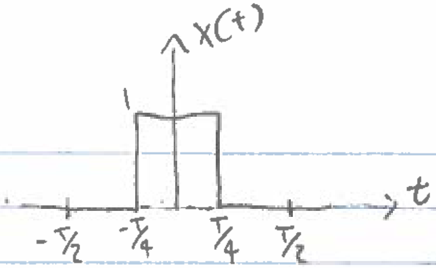


$y[n] = x[n] * h[n]$

2. Echo channel makes sense because the output is the input shifted later in time & scaled to be smaller. So, if you put in a sound, you'd hear that same sound 1 second later at half volume, and then 10 seconds later at a quarter of the original volume. That's just like an echo!

The impulse response: $h(t) = \frac{1}{2}\delta(t-1) + \frac{1}{4}\delta(t-10)$



3a. 

$$x(t) = \begin{cases} 0 \text{ if } -T/2 \le x < -T/4 \\ 1 \text{ if } -T/4 \le x < T/4 \\ 0 \text{ if } T/4 \le x < T/2 \end{cases}$$

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

$$= \cancel{\frac{1}{T}\int_{-T/2}^{-T/4} 0} + \frac{1}{T}\int_{-T/4}^{T/4} 1\, e^{-j\frac{2\pi}{T}kt}\, dt + \cancel{\frac{1}{T}\int_{T/4}^{T/2} 0}$$

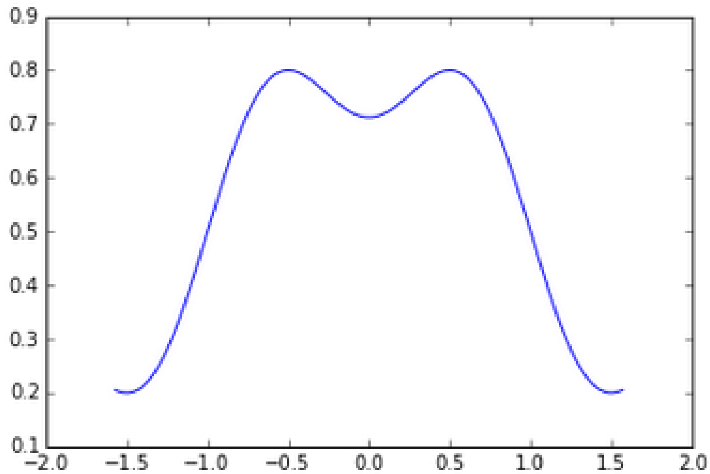
$$= \frac{1}{\cancel{T}} \cdot -\frac{\cancel{T}}{2\pi k j}\left[e^{-\frac{2\pi k t j}{T}}\right]_{-T/4}^{T/4}$$

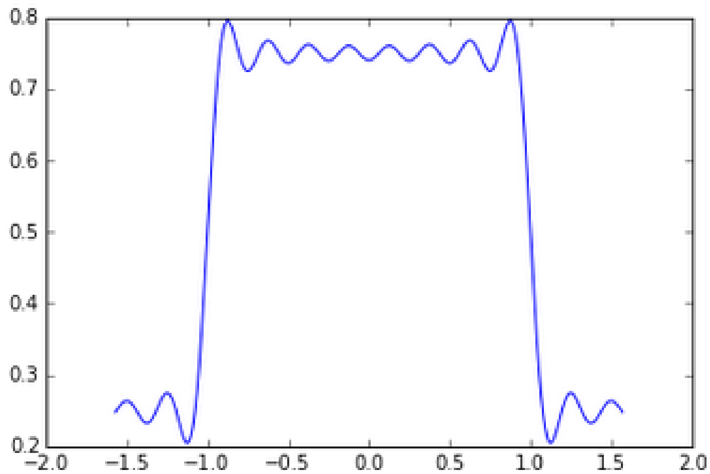
$$= -\frac{1}{\pi k} \cdot \underbrace{\frac{1}{2j}\left[e^{\overset{\theta}{\frac{-\pi k j}{2}}} - e^{\overset{-\theta}{\frac{\pi k j}{2}}}\right]}$$

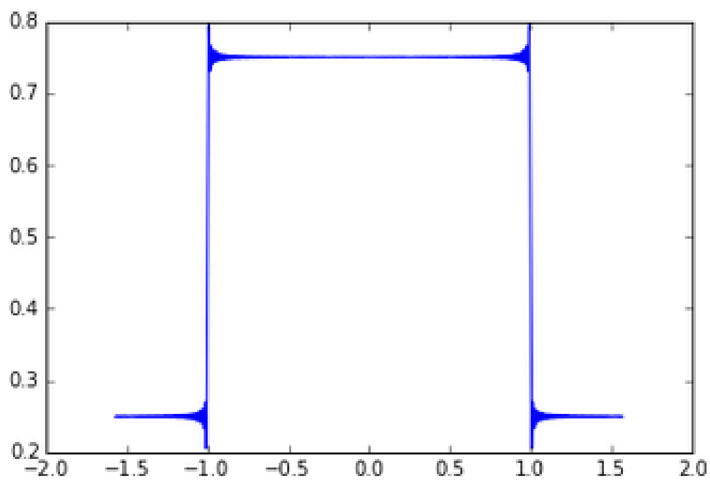
$$= \frac{-1}{\pi k} \cdot \sin\left(\frac{-\pi k}{2}\right) = \frac{\sin\left(\frac{-\pi k}{2}\right)}{\frac{-\pi k}{2} \cdot 2} = \frac{1}{2}\,\frac{\sin\left(\frac{-\pi k}{2}\right)}{\frac{-\pi k}{2}}$$

$$C_k = \boxed{\begin{array}{ll} 0 & \text{if } k \text{ is even} \\ \frac{1}{\pi k} & \text{if } k \text{ is odd } \& \ k \text{ in } (1, 5, 9, \ldots) \\ -\frac{1}{\pi k} & \text{if } k \text{ is odd } \& \ k \text{ in } (3, 7, 11, \ldots) \end{array}}$$

In [24]:
```python
def eval_square_fourier(x, T, num_terms):
    """evaluates fourier series for square wave at x, where the approximation uses the num_terms specified. and has period T"'
    y = np.zeros(len(x),dtype=complex)
    for k in range(num_terms):
        c_k = .5 * np.sinc(-k/(2.0))
        y += c_k * np.exp(i*PI2/T*k*x)
    return y

x = np.linspace(-math.pi/2, math.pi/2, 1000)
y = eval_square_fourier(x, 4, 5)
plt.plot(x,y)
plt.show()

y = eval_square_fourier(x, 4, 17)
plt.plot(x,y)
plt.show()

y = eval_square_fourier(x, 4, 257)
plt.plot(x,y)
plt.show()
```





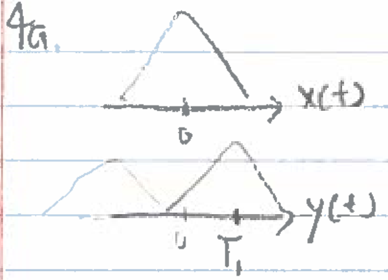


3c. The approximation at the corners has a really high frequency content & does not look like it is getting much smoother. This makes sense, given that

$$\int_{-T/2}^{T/2} |x(t) - \tilde{x}_K(t)|^2 \, dt \to 0 \quad \text{as } K \to \infty$$

To approximate the discontinuity, we need infinitely high frequencies, but K is finite, & we only get higher frequency components by increasing K. So, the fourier series tries really hard to approximate the discontinuity but can't b/c K is finite & we can only get the really high freq components needed by increasing K.

4a.



$$C_{kx} = \frac{-1}{\pi k} \cdot \frac{1}{2j} \Big[$$

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

$$\int_{-T/2}^{T/2} x(t-T_1) \cdots = \int_{-T/2-T_1}^{T/2-T_1}$$

$$y(t) = x(t-T_1)$$

$$x(t) = \sum_{-\infty}^{\infty} C_k e^{j\frac{2\pi}{T}kt}$$

$$\boxed{C_{ky} = C_k \cdot e^{-j\frac{2\pi}{T}kT_1}}$$

$$y(t) = \sum_{-\infty}^{\infty} C_k e^{j\frac{2\pi}{T}kt} \cdot e^{-j\frac{2\pi}{T}kT_1}$$

In [19]:
```
def fs_triangle(ts, M=3, T=4):
    # 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 n == 0:
                Coeff = 0.5
            T1 = T/2
            Coeff = Coeff * np.exp(-2*np.pi*k*T1/T*1j)
            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
            T1 = T/2
            Coeff = Coeff * np.exp(-2*np.pi*k*T1/T*1j)
            x = x + Coeff*np.exp(1j*2*np.pi/T*k*ts)

    return x
```