Part 1 - Complex Numbers

1. Define $z = -1 - 2j$ as a complex number. See Appendix A.

2. Find the real and imaginary part: $z_R = -1.0, z_{Im} = -2.0$

3. The magnitude derived from np.abs and formula's being equal is: True.

4. The phase of z from np.phase: -2.0344439357957027, in radians.
   The phase of z from arctan: 1.1071487177940904, in radians.

   These values differ, as arctan spits the value out regardless of quadrant, while np.phase is always in the correct quadrant.

5. (z+z*) / 2 is: (-1+0j)
   (z-z*) / 2 is: -2j

   These relate to the components of z, where the first is z_R, the second is z_{Im}.

Part 2 - Random Numbers and Plotting

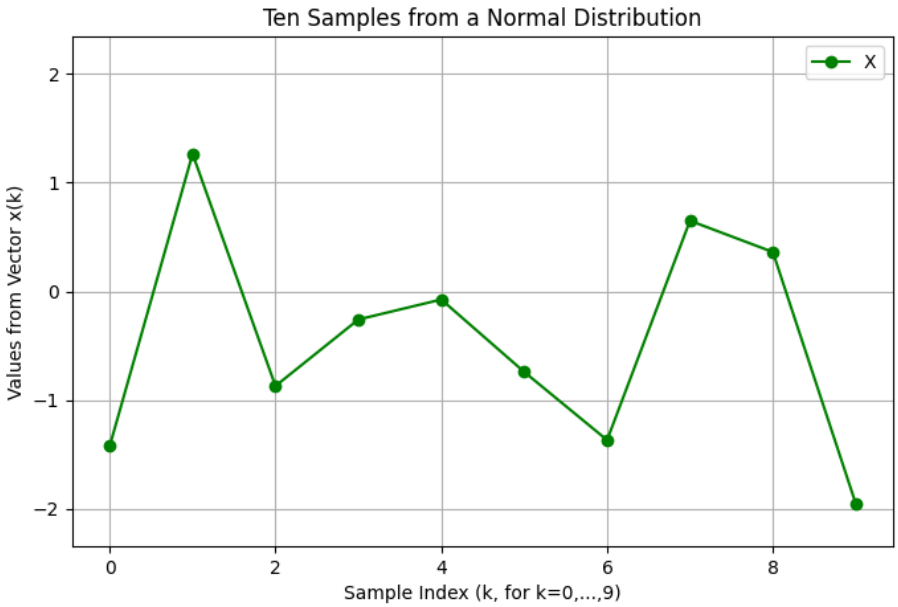

Figure 1: Plot for #3

4.) I primarily learned how to utilize numpy and matplotlib to produce nice looking graphs in Python. More importantly, also how to fill these graphs with vital information to inform the viewer the details about the contents of the PMF and PDF in part 3.

Part 3 - Statistic Tools and More Plotting

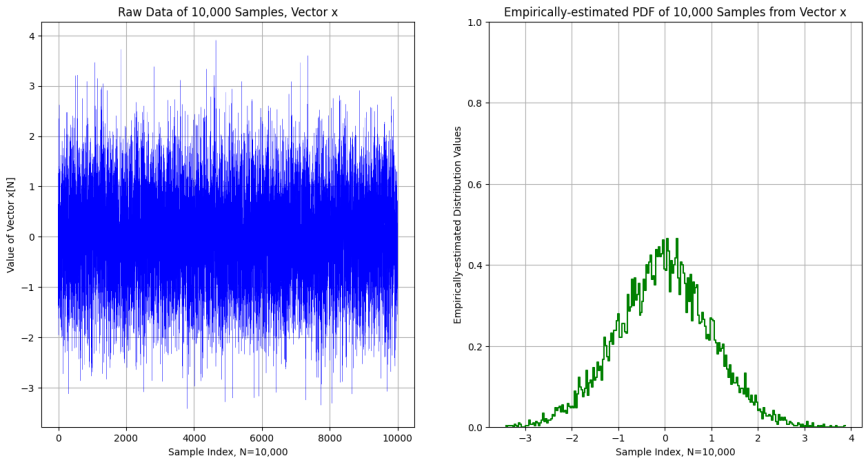

Figure 2: Plot for #1

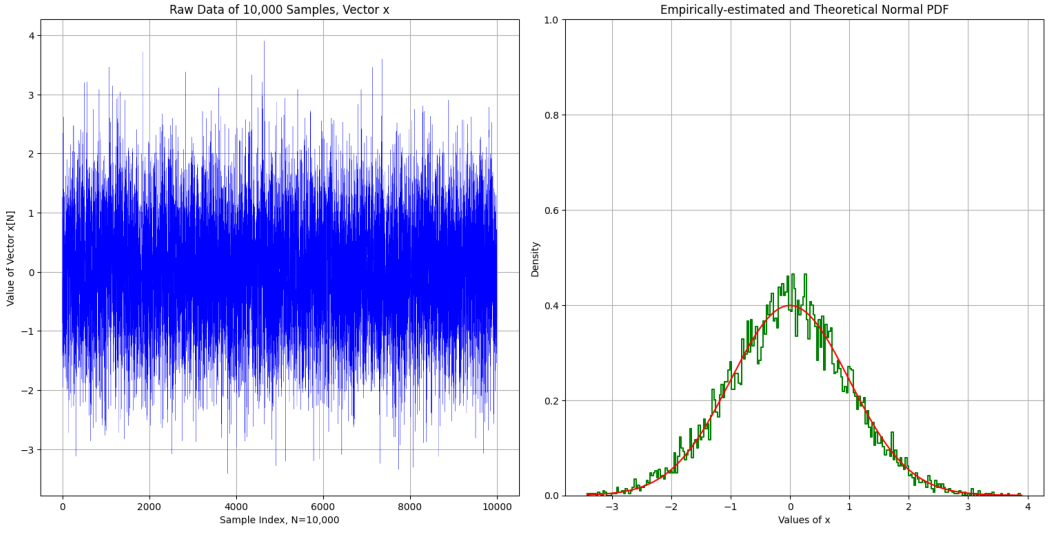

Figure 3: Plot for #2

Part 4 - Sinusoids

1. See function $s(t_n)$ in Appendix D.

2. Three Subplots s_1, s_2, and s_3.

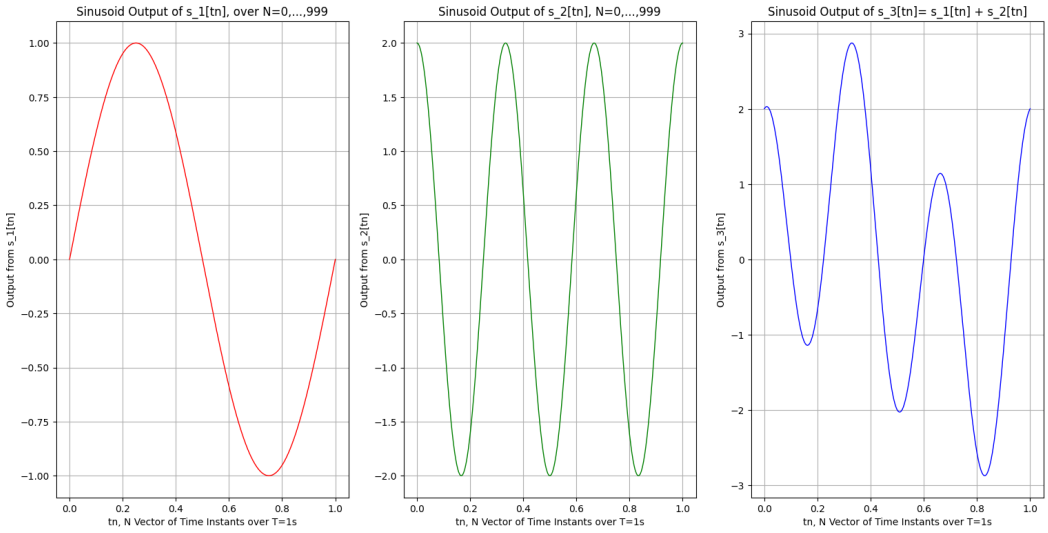

Figure 4: Subplot Exercise

3. *What is the relationship between these three values?*
   It seems as if the Mean Square Value of s_3 is the sum of s_1 and s_2, which follows $s_3 = s_1 + s_2$.

   Is this relationship true for any arbitrary functions $s_1(t)$ and $s_2(t)$
   No, as Mean Square Value is non-linear operation. For example, if $s_1(t)$ and $s_2(t)$ contain any scaling factors that depend on t, then this relationship will fail.

Part 5 - Moving Average and Convolution

1. See Appendix E for code.

2. Arbitrary test for movingAvg:

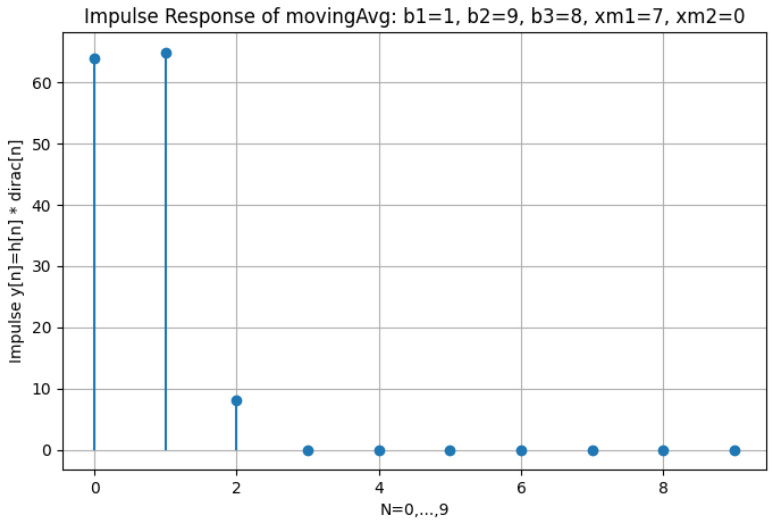

Figure 5: Plot for #2

The impulse response coefficients are directly related to the function inputs I picked, as:
$y[0] = b1 * x[0] + b2 * xm1 + b3 * xm2 = 1 * 1 + 9 * 7 + 8 * 0 = 1 + 63 = 64$
$y[1] = b1 * x[1] + b2 * x[0] + b3 * xm1 = 0 + 9 * 1 + 8 * 7 = 9 + 56 = 65$
$y[3] = b1 * 0 + b2 * 0 + b3 * x[0] = 8 * 1 = 8$

3. Vector x, with multiple responses and plots:

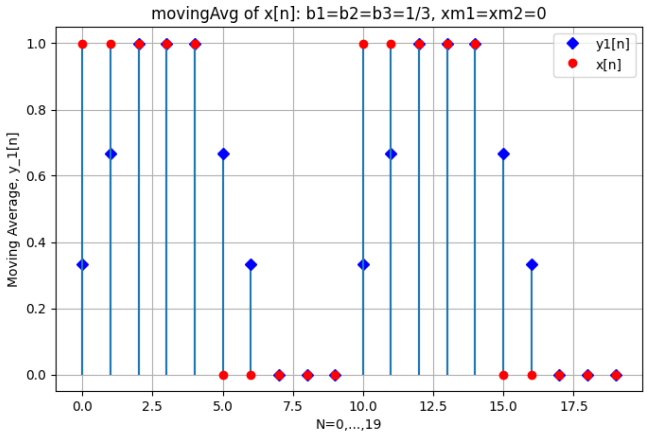

Figure 6: Plot for #3, $y_1[n]$

The input is modified in a way that matches my intuition, the changes in the output are delayed by 3 samples, and as there are five 1's, the average maximum will still be 1.

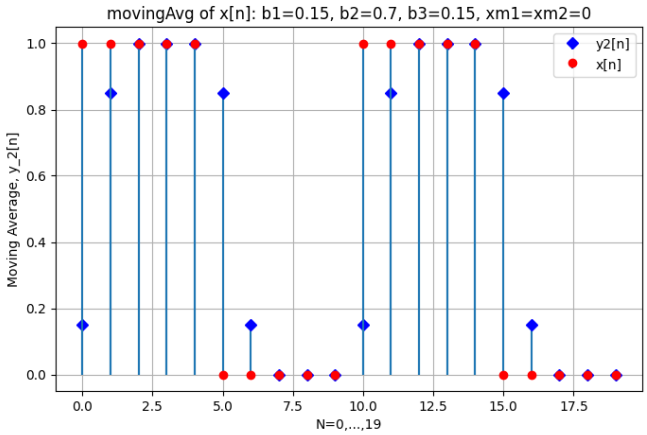

Figure 7: Plot for #3, $y_2[n]$

This plot looks different, as the rate of change in the average is different. This is because the weights in our average are different, but the maximum is still the same as the weights still sum to 1.

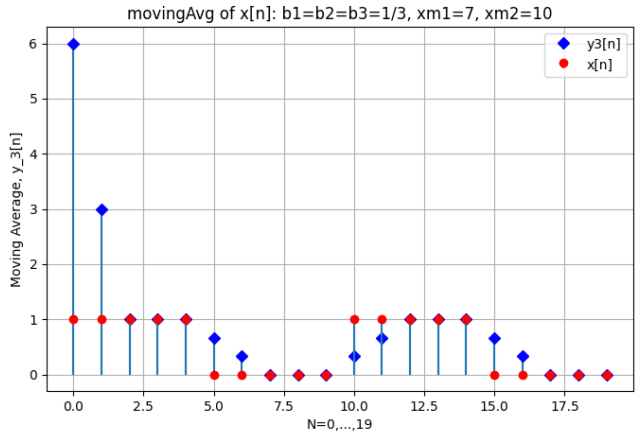

Figure 8: Plot for #3, $y_3[n]$

The difference in the outputs is that the rate of change is different in the two outputs.

4. Comparison between y_1 and y_3

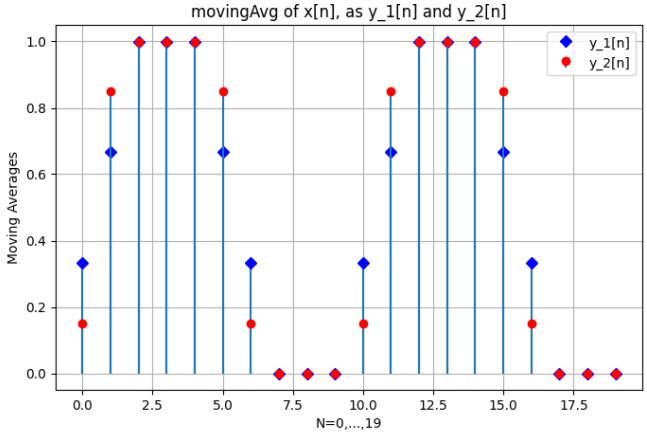

Figure 9: Plot for #3, $y_3[n]$

The number of outputs in y_4 are not the same as y_1 and y_3, as it has $N = 28$ instead of 20. The output values in the same range of $N = 20$

are identical to $y_1[n]$.
I can then make the assumption that the convolution of the impulse response of the movingAvg convolved with $x[n]$ is identical to applying movingAvg directly to $x[n]$.

Appendix

Appendix A - Part 1 Python Code

```
# EE125 Project 1 - Part 1, Complex Numbers
# Sept. 16, 2024

import numpy as np

# 1. Define a complex variable z as z = -1 - 2j
z = complex('-1-2j')

# 2. Find the real and imaginary part of z.
z_real = z.real
z_im   = z.imag
print(f"For z, z_real = {z.real}, z_im = {z_im}")

# 3. Determine the magnitude of z using both ABS and
#    the formula involving R and Im. parts of z. Equal?
z_mag = np.abs(z)
z_mag_formula = np.sqrt(np.pow(z_real, 2) + np.pow(z_im, 2))
print(f"The magnitude derived from np.abs and formula's being equal is:
                    {z_mag == z_mag_formula}")

# 4. Determine phase of z using both arctan and ANGLE function.
z_phase = np.angle(z) # in radians
z_phase_arctan = np.arctan(z_im / z_real)

print(f"The phase of z from np.phase: {z_phase}, which is in radians")
print(f"The phase of z from arctan: {z_phase_arctan}, in radians")
# these values differ, as arctan spits the value out regardless of quadrant,
# while np.phase is always correct quad
```

```
# 5. Calculate (z + z*) / 2 and (z - z*) / 2j
print(f"(z+z*) / 2 is: {(z + z.conjugate()) / 2}")
print(f"(z-z*) / 2 is {(z - z.conjugate()) / 2}")
# these relate to the components of z, where the first is z_real,
# the second is z_im
```

Appendix B - Part 2 Python Code

```
# EE125 Project 1 - Part 2, Random Numbers and Plotting
# Sept. 16, 2024

import numpy as np
import matplotlib.pyplot as plt

# 1. Set the seed of the RNG to 12345
seed = 12345
rng = np.random.default_rng(seed)

# 2. Create vector x, containing N=10,000 samples
#    drawing from a Normal prob. dist. w/mean u=0 variance=1
N = 10000
mean = 0
variance = 1
x = rng.normal(loc=mean, scale=variance, size=N)

# 3. Plot the vector x, with the first 10 samples associated
#    with sample index k=0
k = np.arange(10) # k=0,..,9
x_sampled = x[:10] # first 10 samples

plt.figure(figsize=(8,5)) # set size of plot
plt.plot(k, x_sampled, marker='o', linestyle='-', color='green',
         label='First 10 Samples')

#    Label x and y axis w/text, adding title
#    "Ten Samples from a Normal Distribution"
plt.xlabel('Sample Index (k, for k=0,...,9)')
```

```
plt.ylabel('Values from Vector x(k)')
plt.title('Ten Samples from a Normal Distribution')

#    Adjust y axis to range [-R, R], wh. R=1.2*max{|x(k)|}
#    for k=0,...,9
R = 1.2 * max(abs(x_sampled))
plt.ylim(-R, R)

#    Add a grid to the plot.
plt.grid(True)

#    Add a legend in default location w/string "X ~ N(0,1)" to describe
#    entries of x are on a norm. dist.
plt.legend("X ~ N(0,1)")

# Show the plot
plt.show()

# 4. What did you learn from this plotting exercise?
#    I primarily learned how to utilize numpy and matplotlib to produce
#    nice looking graphs in Python.
#    More importantly, also how to fill these graphs with vital
#    information to inform the viewer the details about the contents
#    of the PMF and PDF.
```

Appendix C - Part 3 Python Code

```
# EE125 Project 1 - Part 3, Statistic Tools and More Plotting
# Sept. 16, 2024

import numpy as np
import matplotlib.pyplot as plt

# Raw data from part 2:
# 1. Set the seed of the RNG to 12345
seed = 12345
rng = np.random.default_rng(seed)

# 2. Create vector x, containing N=10,000 samples
```

```
#    drawing from a Normal prob. dist. w/mean u=0 variance=1
N = 10000
mean = 0
variance = 1
x = rng.normal(loc=mean, scale=variance, size=N)

# Part 3 Questions
# 1. Plot all raw data w/empirically-estimated PDF in two subplots
#    in a new figure
# Create 2 subplots within new figure, in a 1 row and 2 column pattern
figure, (subplot1, subplot2) = plt.subplots(1, 2, figsize=(15, 10))

# Plot all N samples connected by a line
subplot1.plot(np.arange(N), x, linestyle='-', lw='.2', color='blue',
              label='x Raw Data')

# Label all elements of axis appropriately, and include title.
subplot1.set_xlabel('Sample Index, N=10,000')
subplot1.set_ylabel('Value of Vector x[N]')
subplot1.set_title('Raw Data of 10,000 Samples, Vector x')

# Adjust y-axis limits to range [-R, R], where R=6variance
R = 6*variance
plt.ylim(-R, R)

# Add grid to plot
subplot1.grid(True)

# In the second column, plot the PDF for the data.
# Create histogram, density=True to normalize data
hist_data, bin_edges = np.histogram(x, bins=250, density=True)
subplot2.step(bin_edges[:-1], hist_data, color='g',
              label='Normalized Histogram')

# # Label all elements of axis appropriately, and include title.
# subplot2.set_xlabel('Sample Index, N=10,000')
# subplot2.set_ylabel('Empirically-estimated Distribution Values')
# subplot2.set_title('Empirically-estimated PDF of
```

```
                    # 10,000 Samples from Vector x')
# plt.ylim(0, 1)
#
# # Add grid to plot
# subplot2.grid(True)
# plt.show()

# 2. Plot theoetical probability distribution p(x) ontop of subplot 2.
# p(x) = 1 / (variance * np.sqrt(2*pi)) * e^(-.5((x-mean)/variance)^2)
# here, mean = 0, variance = 1
# p(x) = (1 / np.sqrt(2*pi)) * np.exp(-0.5 * x ^2)
linspace_x = np.linspace(np.min(x), np.max(x), N)
p_x = (1 / np.sqrt(2 * np.pi)) * np.exp(-0.5 * np.pow(linspace_x, 2))

# Plot ontop
subplot2.plot(linspace_x, p_x, color='r', label='Theoretical Normal PDF')
# Label all elements of axis appropriately, and include title.
subplot2.set_xlabel('Values of x')
subplot2.set_ylabel('Density')
subplot2.set_title('Empirically-estimated and Theoretical Normal PDF')
plt.ylim(0, 1)

# Add grid to plot
subplot2.grid(True)
plt.tight_layout()
plt.show()
```

Appendix D - Part 4 Python Code

```
# EE125 Project 1 - Part 4, Sinusoids
# Sept. 16, 2024

import numpy as np
import matplotlib.pyplot as plt

# 1. Create a function that calculates the output
#    s(t_n) = Asin(2*pi*f_0*t_n + phi), given A, f_0, phi, and
#    vector of time instants t_n as inputs
```

```
def s_calculate(A, f_0, phi, t_n):
    # f_0 in Hz, phi is phase in radians, t_n is vector of time instants in N
    # if t_n is not type(np.linspace):
    #     print("t_n is not np.linspace")
    #     exit
    t_n = np.asarray(t_n)
    test = A * np.sin(2 * np.pi * f_0 * t_n + phi)
    return test

# 2. Create 3 subplots 1st subplot having s_1(t) = sin( 2 pi t), plotting
#    over 1 second
# Ensure spacing between samples is sufficient to produce a smooth,
# continuous plot.
# A = 1, f_0 = 1, phi = 0
N = 1000 # Number of samples
T = 1 # T is s
tn = np.linspace(0, T, N-1) # N Equispaced samples over T
s_1 = s_calculate(1, 1, 0, tn)

figure, (subplot1, subplot2, subplot3) = plt.subplots(1, 3, figsize=(15,10))
subplot1.plot(tn, s_1, linestyle='-', lw='1', color='red', label='s_1 sinusoid')

subplot1.set_xlabel('tn, N Vector of Time Instants over T=1s')
subplot1.set_ylabel('Output from s_1[tn]')
subplot1.set_title('Sinusoid Output of s_1[tn], over N=0,...,999')
subplot1.grid(True)

# Second subplot: s_2(t) = 2sin(6 pi t + pi / 2)
# A = 2, f_0 = 3, phi = pi/2
s_2 = s_calculate(2, 3, np.pi/2, tn)

subplot2.plot(tn, s_2, linestyle='-', lw='1', color='g', label='s_2 sinusoid')

subplot2.set_xlabel('tn, N Vector of Time Instants over T=1s')
subplot2.set_ylabel('Output from s_2[tn]')
subplot2.set_title('Sinusoid Output of s_2[tn], N=0,...,999')
subplot2.grid(True)
```

```
# Third subplot: s_3(t) = s_1(t) + s_2(t)
s_3 = s_1 + s_2

subplot3.plot(tn, s_3, linestyle='-', lw='1', color='b', label='s_3 sinusoid')

subplot3.set_xlabel('tn, N Vector of Time Instants over T=1s')
subplot3.set_ylabel('Output from s_3[tn]')
subplot3.set_title('Sinusoid Output of s_3[tn]= s_1[tn] + s_2[tn]')
subplot3.grid(True)

plt.tight_layout()
plt.show()

# 3. Calculate mean-square value of s_1(t), s_2(t), and s_1(t) + s_2(t)
s_1_meansquareval = np.mean(s_1 ** 2)
s_2_meansquareval = np.mean(s_2 ** 2)
s_3_meansquareval = np.mean(s_3 ** 2)

print(f"Mean Square Value, s_1: {s_1_meansquareval}, s_2: {s_2_meansquareval},
      s_3: {s_3_meansquareval}")
# Mean Square Value, s_1: 0.4994994994994996, s_2: 2.002002002002002,
# s_3: 2.5015015015015014
print(f"Sum of s_1 and s_2 mean square value:
      {s_1_meansquareval + s_2_meansquareval}")
# Sum of s_1 and s_2 mean square value: 2.501501501501502

# What is the relationship between these three values?
# It seems as if the Mean Square Value of s_3 is the sum of s_1 and s_2,
# which follows s_3 = s_1 + s_2

# Is this relationship true for any arbitrary functions s_1(t) and s_2(t)
# No, as Mean Square Value is non-linear operation. For example,
# if s_1(t) and s_2(t) contain any scaling factors that depend on t,
# then this relationship will fail.
```

Appendix E - Part 5 Python Code

```
# EE125 Project 1 - Part 5, Moving Average and Convolution
```

```python
# Sept. 16, 2024

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


# 1. Implement movingAverage(), a three point moving average equation
# x is a vector with N elements
# b1, b2, b3 are float weights
# xm1 and xm2 are x[-1] and x[-2] respectively
# movingAvg returns a vector with N elements as well
def movingAvg(x, b1, b2, b3, xm1, xm2):
    N = len(x)
    y = np.zeros(N) # Assuming x is a numpy array, zero out all elements

    # For initial cases, n = 0, use xm1 = x[-1] and xm2 = x[-2]
    y[0] = b1 * x[0] + b2 * xm1 + b3 * xm2

    if N > 1:
        y[1] = b1 * x[1] + b2 * x[0] + b3 * xm1

    for i in range(2, N):
        y[i] = b1 * x[i] + b2 * x[i - 1] + b3 * x[i - 2]

    return y

# Validate that code is working correctly
# y_filt = scipy.signal.lfilter([b1,b2,b3],1,x,zi=zi)

print(f"Test 1: {movingAvg([1, 1, 1, 0, 0], 1, 1, 1, 0, 0)}")
# Test 1: [1. 2. 3. 2. 1.]

print(f"Test 2: {movingAvg([1, 1, 1, 0, 0], 1, 1, 1, 1, 1)}")
# Test 2: [3. 3. 3. 2. 1.]

print(f"Test 3: {movingAvg([1, 2, 3, 0, 0], 4, 5, 6, 7, 8)}")
# Test 3: [87. 55. 28. 27. 18.]
```

```
# All tests seem to follow the example solutions!

# 2. Pick some arbitrary b1, b2, b3, and compute the impulse response
#    using movingAvg
N_2 = 10
dirac_delta = np.zeros(N_2)
dirac_delta[0] = 1

impulse_response = movingAvg(dirac_delta, 1, 9, 8, 7, 0)
print(f"Test for #2.), b1=1, b2=9, b3=8, xm1=7, xm2=0 :
        {movingAvg(dirac_delta, 1, 9, 8, 7, 0)}")

# plt.figure(figsize=(8,5)) # set size of plot
# plt.stem(impulse_response, label='Impulse Response', basefmt=" ")
# plt.xlabel(f'N=0,...,{N_2 - 1}')
# plt.ylabel('Impulse y[n]=h[n] * dirac[n]')
# plt.title('Impulse Response of movingAvg: b1=1, b2=9, b3=8, xm1=7, xm2=0')
# plt.grid(True)
# plt.show()

# The impulse response coefficients are directly related to the function inputs
# I picked, as
# y[0] = b1 * x[0] + b2 * xm1 + b3 * xm2 = 1 * 1 + 9 * 7 + 8 * 0 = 1 + 63 = 64
# y[1] = b1 * x[1] + b2 * x[0] + b3 * xm1 = 0 + 9 * 1 + 8 * 7 = 9 + 56 = 65
# y[3] = b1 * 0 + b2 * 0 + b3 * x[0] = 8 * 1 = 8

# 3. Create vector x, repeating five 1s, then five 0s, two times so
#    total vector length = 20
# Calculate y[n] as y1, from moving average system for vector x,
# where b1=b2=b3=1/3, xm1=xm2=0 (rest)
pattern = [1] * 5 + [0] * 5
x_fives = np.tile(pattern, 2)
print(f"x_fives: {x_fives}")

# Plot the input x[n] and output y1 on the same plot
y_1 = movingAvg(x_fives, 1/3, 1/3, 1/3, 0, 0)
# plt.figure(figsize=(8,5)) # set size of plot
# plt.stem(y_1, markerfmt='bD', label='y1[n]', basefmt=" ")
```

```
# plt.stem(x_fives, markerfmt='r', label='x[n]', basefmt=" ")
# plt.legend()
# plt.xlabel(f'N=0,...,19')
# plt.ylabel('Moving Average, y_1[n]')
# plt.title('movingAvg of x[n]: b1=b2=b3=1/3, xm1=xm2=0')
# plt.grid(True)
# plt.show()

# Add an appropriate legend to label the two signals
# Is the input modified in a way that matches your intuition?
# Describe how the output relates to the input.
# Yes, the input is modified in a way that matches my intuition, the changes in
# the output are delayed by 3 samples,
# and as there are five 1's, the average maximum will still be 1.

# Calculate y2 on x, now b1=0.15, b2=0.7, b3=0.15, same xm's.
# Plot x[n] and y2 on same plot, with appropriate legend.
y_2 = movingAvg(x_fives, 0.15, 0.7, 0.15, 0, 0)
# plt.figure(figsize=(8,5)) # set size of plot
# plt.stem(y_2, markerfmt='bD', label='y2[n]', basefmt=" ")
# plt.stem(x_fives, markerfmt='r', label='x[n]', basefmt=" ")
# plt.legend()
# plt.xlabel(f'N=0,...,19')
# plt.ylabel('Moving Average, y_2[n]')
# plt.title('movingAvg of x[n]: b1=0.15, b2=0.7, b3=0.15, xm1=xm2=0')
# plt.grid(True)
# plt.show()

# How does this output look different than the previous plot? Why does it
# look different?
# This plot looks different, as the rate of change in the average is different.
# our average are different, but the maximum is still the same as
# the weights still sum to 1.

# Calculate y3 on x, now b1=b2=b3=1/3, but xm1=7, xm2=10
y_3 = movingAvg(x_fives, 1/3, 1/3, 1/3, 7, 10)
# plt.figure(figsize=(8,5)) # set size of plot
# plt.stem(y_3, markerfmt='bD', label='y3[n]', basefmt=" ")
```

```
# plt.stem(x_fives, markerfmt='r', label='x[n]', basefmt=" ")
# plt.legend()
# plt.xlabel(f'N=0,...,19')
# plt.ylabel('Moving Average, y_3[n]')
# plt.title('movingAvg of x[n]: b1=b2=b3=1/3, xm1=7, xm2=10')
# plt.grid(True)
# plt.show()

# Plot y1 and y2 on the same plot, with appropriate legend.
# plt.figure(figsize=(8,5)) # set size of plot
# plt.stem(y_1, markerfmt='bD', label='y_1[n]', basefmt=" ")
# plt.stem(y_2, markerfmt='r', label='y_2[n]', basefmt=" ")
# plt.legend()
# plt.xlabel(f'N=0,...,19')
# plt.ylabel('Moving Averages')
# plt.title('movingAvg of x[n], as y_1[n] and y_2[n]')
# plt.grid(True)
# plt.show()

# Describe the differences you see in the two outputs.
# The difference in the outputs is that the rate of change is
# different in the two outputs.

# 4. The moving average filter is non-recursive, an FIR, so the
#    convolution can be used.
# For b1=b2=b3=1/3, create vector h for h[n], impulse response
N_4 = 20
dirac_delta_h = np.zeros(N_4)
dirac_delta_h[0] = 1

impulse_response_h = movingAvg(dirac_delta, 1/3, 1/3, 1/3, 0, 0)

# Calculate the convolution of h[n] and x[n] from the last step (fives)
# and store as y4
y_4 = np.convolve(impulse_response_h, x_fives)
plt.stem(y_1, markerfmt='rD', label='y_1[n]', basefmt=" ")
plt.stem(y_3, markerfmt='g', label='y_3[n]', basefmt=" ")
plt.stem(y_4, markerfmt='bx', label='y_4[n]', basefmt=" ")
```

```
plt.legend()
plt.xlabel(f'N')
plt.ylabel('y_1[n], y_3[n], and y_4[n]')
plt.title('movingAvg y_1[n], y_3[n] of x[n], vs. Convolution of movingAvg
          and x[n]')
plt.grid(True)
plt.show()

# Compared to y1 and y3, are the number of outputs in y4 the same? How are
# the values related?
# What can you then say about the assumptions the convolution function makes
# about the initial system conditions?
# The number of outputs in y4 are not the same as y1 and y3, as it has N=28
# instead of 20.
# The output values in the same range of N=20 are identical to y_1[n].
# I can then make the assumption that the convolution of the impulse response
# of the movingAvg convolved with x[n] is identical to applying movingAvg
# directly to x[n].
```