1Aim: To study the basic signals Unit Impulse, Ramp, Unit Step, Exponential, Discrete sine and cosine signals with given sampling frequency.

Code--

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

# Step Signal
def step_fn(shift, t):
    y = []
    for k in t:
        y.append(0 if k < shift else 1)
    return y

n1 = np.arange(-10, 11)
u = step_fn(2, n1)

# Impulse Signal
def impulse_fn(pos, t):
    y = []
    for k in t:
        y.append(1 if k == pos else 0)
    return y

n2 = np.arange(-10, 10)
d = impulse_fn(4, n2)

#  Ramp Signal
```

```
def ramp_fn(t):
   y = []
   for k in t:
      y.append(0 if k < 0 else k)
   return y

n3 = np.arange(-10, 10)
r = ramp_fn(n3)

#  Exponential Positive
def expo_fn(a, t):
   return [np.exp(a*k) for k in t]

n4 = np.arange(-1, 1, 0.1)
exp_pos = expo_fn(2, n4)

#  Exponential Negative
exp_neg = expo_fn(-2, n4)

#  Sine & Cosine
Fs, f, N = 100, 5, 50
n5 = np.arange(N)
t5 = n5 / Fs
cos_dis = np.cos(2*np.pi*f*n5/Fs)
sin_dis = np.sin(2*np.pi*f*n5/Fs)

t_cont = np.linspace(0, N/Fs, 1000)
cos_cont = np.cos(2*np.pi*f*t_cont)
```

```
sin_cont = np.sin(2*np.pi*f*t_cont)



plt.figure(figsize=(14, 12))

plt.subplot(3, 2, 1)
plt.stem(n1, u)
plt.title("Unit Step u[n-2]")
plt.grid(True)

plt.subplot(3, 2, 2)
plt.stem(n2, d)
plt.title("Unit Impulse δ[n-4]")
plt.grid(True)

plt.subplot(3, 2, 3)
plt.stem(n3, r)
plt.title("Unit Ramp r[n]")
plt.grid(True)

plt.subplot(3, 2, 4)
plt.stem(n4, exp_pos)
plt.title("Exponential e^(2n)")
plt.grid(True)

plt.subplot(3, 2, 5)
plt.stem(n4, exp_neg)
plt.title("Exponential e^(-2n)")
```

```
plt.grid(True)


plt.subplot(3, 2, 6)
plt.stem(n5, cos_dis, linefmt="b-", markerfmt="bo", basefmt=" ")
plt.stem(n5, sin_dis, linefmt="r-", markerfmt="ro", basefmt=" ")
plt.title("Discrete Cos & Sin")
plt.legend(["cos", "sin"])
plt.grid(True)

plt.tight_layout()
plt.show()

# continuous vs discrete comparison
plt.figure(figsize=(12,5))
plt.plot(t_cont, cos_cont, 'b-', label="cos cont")
plt.stem(t5, cos_dis, linefmt="k-", markerfmt="ko", basefmt=" ", label="cos disc")
plt.title("Cosine: Continuous vs Discrete")
plt.legend()
plt.grid(True)

plt.figure(figsize=(12,5))
plt.plot(t_cont, sin_cont, 'r-', label="sin cont")
plt.stem(t5, sin_dis, linefmt="k-", markerfmt="ko", basefmt=" ", label="sin disc")
plt.title("Sine: Continuous vs Discrete")
plt.legend()
plt.grid(True)
plt.show()
```

2Aim: Represent complex exponentials as a function of real and imaginary part, and impulse and step response of two vectors using MATLAB.

Code:

Complex exponentials as a function of real and imaginary part:

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

# discrete index
k = np.arange(0, 30)
w = 0.2 * np.pi

# complex exponential
sig = np.exp(1j * w * k)
sig_re = sig.real
sig_im = sig.imag

plt.figure(figsize=(8,6))

plt.subplot(2,1,1)
plt.stem(k, sig_re)
plt.title("Real Component → cos(ωk)")
plt.xlabel("k")
plt.ylabel("Value")
plt.grid(True)

plt.subplot(2,1,2)
plt.stem(k, sig_im)
```

```
plt.title("Imag Component → sin(ωk)")
plt.xlabel("k")
plt.ylabel("Value")
plt.grid(True)

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

Impulse and Step response of two vectors

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


h1 = np.array([1, -0.5, 0.25])
h2 = np.array([1, 1.6, 3, 2.5])

N = 20
n = np.arange(N)

# input signals
x_imp = np.zeros(N)
x_imp[0] = 1
x_step = np.ones(N)

# responses for h1
```

```
resp_imp1  = np.convolve(x_imp, h1)[:N]
resp_step1 = np.convolve(x_step, h1)[:N]

# responses for h2
resp_imp2  = np.convolve(x_imp, h2)[:N]
resp_step2 = np.convolve(x_step, h2)[:N]

plt.figure(figsize=(12,8))

plt.subplot(3,2,1)
plt.stem(h1)
plt.title("System h1")
plt.grid(True)

plt.subplot(3,2,2)
plt.stem(h2)
plt.title("System h2")
plt.grid(True)

plt.subplot(3,2,3)
plt.stem(n, resp_imp1)
plt.title("Impulse → h1")
plt.grid(True)

plt.subplot(3,2,4)
plt.stem(n, resp_step1)
plt.title("Step → h1")
plt.grid(True)
```

```
plt.subplot(3,2,5)
plt.stem(n, resp_imp2)
plt.title("Impulse → h2")
plt.grid(True)

plt.subplot(3,2,6)
plt.stem(n, resp_step2)
plt.title("Step → h2")
plt.grid(True)

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

3.Aim : Convolution between two vectors using MATLAB without using conv function, and cross correlation between two vectors using MATLAB.

Code:

Convolution between two vectors

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

x = np.array([1, 2, 3])
h = np.array([0, 1, 0.5])
```

```
N = len(x) + len(h) - 1
y = np.zeros(N)



for n in range(N):
   for k in range(len(x)):
      if 0 <= n-k < len(h):
         y[n] += x[k] * h[n-k]

print("x[n]:", x)
print("h[n]:", h)
print("Convolution y[n]:", y)



plt.figure(figsize=(10,6))

plt.subplot(3,1,1)
plt.stem(x)
plt.title("Signal x[n]")
plt.grid(True)

plt.subplot(3,1,2)
plt.stem(h)
plt.title("Signal h[n]")
plt.grid(True)

plt.subplot(3,1,3)
plt.stem(np.arange(N), y)
```

```
plt.title("Convolution Result y[n]")
plt.grid(True)

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

cross correlation between two vectors

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



x = np.array([1, 2, 3])
y_sig = np.array([0, 1, 0.5])

N_corr = len(x) + len(y_sig) - 1
r = np.zeros(N_corr)



for n in range(-len(y_sig)+1, len(x)):
    acc = 0
    for k in range(len(x)):
        if 0 <= k+n < len(y_sig):
            acc += x[k] * y_sig[k+n]
    r[n + len(y_sig) - 1] = acc

print("x[n]:", x)
print("y[n]:", y_sig)
print("Cross-Correlation r_xy[n]:", r)
```

```
plt.figure(figsize=(10,6))


plt.subplot(3,1,1)
plt.stem(x)
plt.title("Signal x[n]")
plt.grid(True)

plt.subplot(3,1,2)
plt.stem(y_sig)
plt.title("Signal y[n]")
plt.grid(True)

plt.subplot(3,1,3)
lags = np.arange(-len(y_sig)+1, len(x))
plt.stem(lags, r)
plt.title("Cross-Correlation r_xy[n]")
plt.grid(True)

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

4Aim: Compute DFT and IDFT of a given sequence using python.

Code:

```
import math
```

```
import cmath
import matplotlib.pyplot as plt

#find dft
def DFT(x):
    N = len(x)
    X = []
    for k in range(N):
        val = 0
        for n in range(N):
            angle = -2j * math.pi * k * n / N
            val += x[n] * cmath.exp(angle)
        X.append(val)
    return X

#find idft
def IDFT(X):
    N = len(X)
    x = []
    for n in range(N):
        val = 0
        for k in range(N):
            angle = 2j * math.pi * k * n / N
            val += X[k] * cmath.exp(angle)
        x.append(val / N)
    return x

#16input sequence
```

```
x = [1, 2, 3, 4, 2, 1, 0, 1, 2, 3, 4, 3, 2, 1, 0, 1]
N = len(x)

X = DFT(x)
x_reconstructed = IDFT(X)

n = range(N)
k = range(N)
X_mag = [abs(val) for val in X]
X_phase = [cmath.phase(val) for val in X]

plt.figure(figsize=(12, 8))

#Input Sequence
plt.subplot(2, 2, 1)
plt.stem(n, x)
plt.title("Input Sequence")
plt.xlabel("n")
plt.ylabel("Amplitude")

#DFT Magnitude Spectrum
plt.subplot(2, 2, 2)
plt.stem(k, X_mag)
plt.title("DFT Magnitude Spectrum")
plt.xlabel("k")
plt.ylabel("|X[k]|")

#DFT Phase Spectrum
```

```
plt.subplot(2, 2, 3)
plt.stem(k, X_phase)
plt.title("DFT Phase Spectrum")
plt.xlabel("k")
plt.ylabel("Phase (radians)")

#Reconstructed Sequence (IDFT)
plt.subplot(2, 2, 4)
plt.stem(n, [val.real for val in x_reconstructed])
plt.title("Reconstructed Sequence (IDFT)")
plt.xlabel("n")
plt.ylabel("Amplitude")

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

5. Aim: Linear convolution of two sequences using DFT using python.

Code:

```
import math
import cmath
import matplotlib.pyplot as plt

#find dft
def DFT(x):
    N = len(x)
    X = []
```

```
    for k in range(N):
        val = 0
        for n in range(N):
            angle = -2j * math.pi * k * n / N
            val += x[n] * cmath.exp(angle)
        X.append(val)
    return X

#find idft
def IDFT(X):
    N = len(X)
    x = []
    for n in range(N):
        val = 0
        for k in range(N):
            angle = 2j * math.pi * k * n / N
            val += X[k] * cmath.exp(angle)
        x.append(val / N)
    return x

x1 = [1, 2, 3, 4, 2, 1, 0, 1, 2, 3, 4, 3, 2, 1, 0, 1]
x2 = [2, 1, 0, 1, 3, 2, 1, 0, 2, 1, 3, 4, 2, 1, 0, 1]

N = len(x1) + len(x2) - 1

# Zero padding both sequences
x1 += [0] * (N - len(x1))
x2 += [0] * (N - len(x2))
```

```
X1 = DFT(x1)
X2 = DFT(x2)

# Multiply in f domain
Y = [X1[k] * X2[k] for k in range(N)]

y = IDFT(Y)
y_real = [val.real for val in y]

plt.figure(figsize=(12, 9))

# x1[n]
plt.subplot(3, 1, 1)
plt.stem(range(len(x1)), x1)
plt.title("Input Sequence x1[n]")
plt.xlabel("n")
plt.ylabel("Amplitude")

# x2[n]
plt.subplot(3, 1, 2)
plt.stem(range(len(x2)), x2)
plt.title("Input Sequence x2[n]")
plt.xlabel("n")
plt.ylabel("Amplitude")

# Convolution output
plt.subplot(3, 1, 3)
```

```
plt.stem(range(len(y_real)), y_real)
plt.title("Linear Convolution of x1[n] and x2[n] using DFT")
plt.xlabel("n")
plt.ylabel("Amplitude")

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

6.Aim: Compute z-transform from the given transfer function and its ROC using python. Compute rational z-transform from the given poles and zeros using python.

Code:

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

# X(z) = (1 + 0.5z^-1) / (1 - 0.25z^-1 - 0.125z^-2)
num = [1, 0.5]          # numerator coeff
den = [1, -0.25, -0.125]   # denominator coeff

# find poles and zeros
zeros = np.roots(num)
poles = np.roots(den)
gain = num[0] / den[0]

print("Z-Transform from Transfer Function")
print(f"Numerator Coefficients (N(z)): {num}")
print(f"Denominator Coefficients (D(z)): {den}")
```

```
print(f"Zeros: {zeros}")

print(f"Poles: {poles}")

print(f"Gain: {gain}")


#From given Poles and Zeros

given_zeros = [0.2, -0.9]

given_poles = [0.8, 0.3]

gain = 1.0


# num & den from given poles and zeroes

num_poly = np.poly(given_zeros) * gain

den_poly = np.poly(given_poles)


print("\nRational Z-Transform from Given Poles and Zeros")

print(f"Given Zeros: {given_zeros}")

print(f"Given Poles: {given_poles}")

print(f"Numerator Polynomial (N(z)): {num_poly}")

print(f"Denominator Polynomial (D(z)): {den_poly}")



plt.figure(figsize=(8, 6))

# Unit circle

theta = np.linspace(0, 2*np.pi, 500)

plt.plot(np.cos(theta), np.sin(theta), 'k--', label='Unit Circle')


# Plot poles and zeros

plt.scatter(np.real(zeros), np.imag(zeros), marker='o', color='b', s=100, label='Zeros
(from TF)')
```

```
plt.scatter(np.real(poles), np.imag(poles), marker='x', color='r', s=100, label='Poles (from TF)')

plt.scatter(np.real(given_zeros), np.imag(given_zeros), marker='o', color='g', s=80, label='Given Zeros')

plt.scatter(np.real(given_poles), np.imag(given_poles), marker='x', color='m', s=80, label='Given Poles')


plt.title("Pole–Zero Plot in Z-plane")

plt.xlabel("Real Part")

plt.ylabel("Imaginary Part")

plt.axhline(0, color='black')

plt.axvline(0, color='black')

plt.legend(loc="upper right", bbox_to_anchor=(1, 1))

plt.grid(True)

plt.axis('equal')


max_pole_mag = np.max(np.abs(poles))

print("\nROC for Z-Transform from Transfer Function")

print(f"Poles Magnitudes: {np.abs(poles)}")

print(f"ROC: |z| > {max_pole_mag:.4f} (for causal system)")


max_given_mag = np.max(np.abs(given_poles))

print("\nROC for Rational Z-Transform from Given Poles and Zeros")

print(f"Given Poles Magnitudes: {np.abs(given_poles)}")

print(f"ROC: |z| > {max_given_mag:.4f}  (for causal system)")

plt.show()
```

7.Aim: Compute partial fraction expansion of rational z-transform using python.

Code:

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

def partial_fraction_manual(num, den):
    num = np.array(num, dtype=float)
    den = np.array(den, dtype=float)

    #find poles
    poles = np.roots(den)

    #do derivative of deno
    d_den = np.polyder(den)

    #calculate residues
    residues = []
    for p in poles:
        N_val = np.polyval(num, p)
        Dp_val = np.polyval(d_den, p)
        A = N_val / Dp_val
        residues.append(A)

    return residues, poles

# X(z) = (1 + 2z^-1 + 3z^-2) / (1 - 0.5z^-1 - 0.25z^-2)
num = [1, 2, 3]
den = [1, -0.5, -0.25]
```

```
residues, poles = partial_fraction_manual(num, den)

print("\nPartial Fraction Expansion Results")
for i in range(len(residues)):
   print(f"Term {i+1}: A{i+1} = {residues[i]:.4f},   Pole p{i+1} = {poles[i]:.4f}")

#roc analysis
magnitudes = np.abs(poles)
max_radius = np.max(magnitudes)
min_radius = np.min(magnitudes)

print("\nROC Conditions")
print(f"Poles Magnitudes: {magnitudes}")
print(f"Right-sided ROC: |z| > {max_radius:.4f}")
print(f"Left-sided  ROC: |z| < {min_radius:.4f}")
print(f"Two-sided  ROC: {min_radius:.4f} < |z| < {max_radius:.4f}")

#reconstruct time-domain sequences
n = np.arange(-10, 20)

x_right = np.zeros_like(n, dtype=float)
x_left = np.zeros_like(n, dtype=float)
x_two = np.zeros_like(n, dtype=float)

for i in range(len(residues)):
   A = residues[i]
   p = poles[i]
```

```
    x_right += A * (p ** n) * (n >= 0)      # Right-sided
    x_left  += -A * (p ** n) * (n < 0)      # Left-sided
    x_two   += A * (p ** n)                 # Two-sided general



plt.figure(figsize=(11, 9))

# Pole-zero plot
plt.subplot(2,2,1)
theta = np.linspace(0,2*np.pi,400)
plt.plot(np.cos(theta), np.sin(theta), 'k--', label="Unit Circle")
plt.scatter(np.real(poles), np.imag(poles), marker='x', color='r', s=120, label="Poles")
plt.scatter(np.real(residues), np.imag(residues), marker='o', color='b', s=100,
label="Residues")
plt.axhline(0, color='black'); plt.axvline(0, color='black')
plt.title("Pole–Zero Plot")
plt.legend(loc="upper right", bbox_to_anchor=(1.3, 1))
plt.grid()

#right sided
plt.subplot(2,2,2)
plt.stem(n, x_right)
plt.title("Right-sided signal (ROC: |z| > max pole)")
plt.xlabel("n"); plt.ylabel("Amplitude")

#left sided
plt.subplot(2,2,3)
plt.stem(n, x_left)
```

```
plt.title("Left-sided signal (ROC: |z| < min pole)")
plt.xlabel("n"); plt.ylabel("Amplitude")

#two sided
plt.subplot(2,2,4)
plt.stem(n, x_two)
plt.title("Two-sided signal (ROC between poles)")
plt.xlabel("n"); plt.ylabel("Amplitude")

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

8.Aim: Design a Type -1 Chebyshev IIR high-pass filter using PYTHON.

Code:

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

sample_rate = 8000
pass_edge = 3900
stop_edge = 1800
pass_ripple = 0.4
stop_atten = 30

omega_p_d = 2 * np.pi * pass_edge / sample_rate
omega_s_d = 2 * np.pi * stop_edge / sample_rate
```

```
warp_p = np.tan(omega_p_d / 2)
warp_s = np.tan(omega_s_d / 2)

ratio_ws = warp_p / warp_s
g_s = 10**(stop_atten / 10) - 1
g_p = 10**(pass_ripple / 10) - 1

order_exact = np.arccosh(np.sqrt(g_s / g_p)) / np.arccosh(ratio_ws)
order_N = int(np.ceil(order_exact))
print(f"Calculated Order N: {order_N}")

eps_val = np.sqrt(g_p)
alpha_val = np.arcsinh(1 / eps_val) / order_N

proto_poles = []
for idx in range(1, order_N + 1):
    angle_val = (2 * idx - 1) * np.pi / (2 * order_N)
    real_comp = -np.sinh(alpha_val) * np.sin(angle_val)
    imag_comp = np.cosh(alpha_val) * np.cos(angle_val)
    proto_poles.append(real_comp + 1j * imag_comp)
proto_poles = np.array(proto_poles)

hp_poles = warp_p / proto_poles
zplane_poles = (1 + hp_poles) / (1 - hp_poles)
zplane_zeros = np.ones(order_N)

coef_b = np.poly(zplane_zeros)
coef_a = np.poly(zplane_poles)
```

```
h_nyq = np.polyval(coef_b, -1) / np.polyval(coef_a, -1)
gain_adj = 1 / np.abs(h_nyq)
coef_b *= gain_adj
coef_b = np.real(coef_b)
coef_a = np.real(coef_a)
total_pts = 60
input_sig = np.zeros(total_pts)
input_sig[0] = 1
output_sig = np.zeros(total_pts)

b_norm_new = coef_b / coef_a[0]
a_norm_new = coef_a / coef_a[0]

Nb = len(b_norm_new)
Na = len(a_norm_new)

for n_idx in range(total_pts):
  for kk in range(Nb):
    if n_idx - kk >= 0:
      output_sig[n_idx] += b_norm_new[kk] * input_sig[n_idx - kk]
  for kk in range(1, Na):
    if n_idx - kk >= 0:
      output_sig[n_idx] -= a_norm_new[kk] * output_sig[n_idx - kk]

time_ms = np.arange(total_pts) / sample_rate * 1000
ww = np.linspace(0, np.pi, 1024)
zv = np.exp(1j * ww)
```

```
Hresp = np.polyval(coef_b, zv) / np.polyval(coef_a, zv)
Hmag = np.maximum(np.abs(Hresp), 1e-12)
freq_vals = ww * sample_rate / (2 * np.pi)

plt.figure(figsize=(12, 10))

plt.subplot(2, 2, 1)
plt.plot(freq_vals, 20 * np.log10(Hmag))
plt.title(f"Magnitude Response (N={order_N})")
plt.ylabel("Magnitude (dB)")
plt.grid(True)
plt.ylim(-stop_atten - 20, 5)

plt.subplot(2, 2, 2)
plt.plot(freq_vals, np.unwrap(np.angle(Hresp)))
plt.title("Phase Response")
plt.ylabel("Phase (rad)")
plt.grid(True)

plt.subplot(2, 2, 3)
plt.plot(np.real(zplane_poles), np.imag(zplane_poles), 'x', color='red', label='Poles')
plt.plot(np.real(zplane_zeros), np.imag(zplane_zeros), 'o', color='blue', label='Zeros')
circle_uc = plt.Circle((0, 0), 1, fill=False, color='black', linestyle='--')
plt.gca().add_artist(circle_uc)
plt.gca().set_aspect('equal')
radius_max = max(1.0, np.max(np.abs(zplane_poles)))
plt.xlim(-radius_max-0.2, radius_max+0.2)
plt.ylim(-radius_max-0.2, radius_max+0.2)
```

```
plt.grid(True)
plt.legend()
plt.title("Pole-Zero Plot")

plt.subplot(2, 2, 4)
plt.stem(time_ms, output_sig, basefmt=" ")
plt.title("Impulse Response (Manual)")
plt.xlabel("Time (ms)")
plt.ylabel("Amplitude")
plt.grid(True)

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