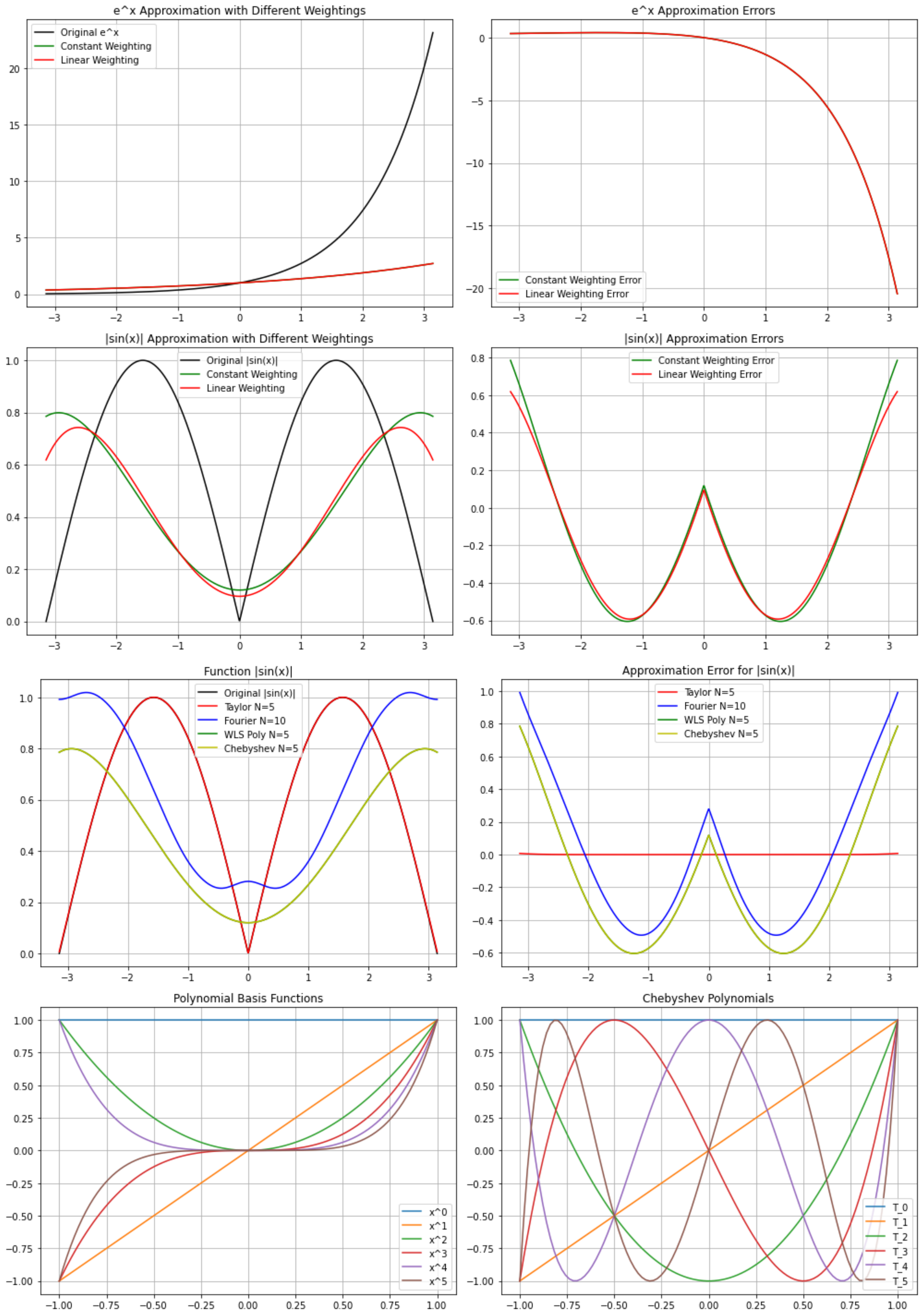
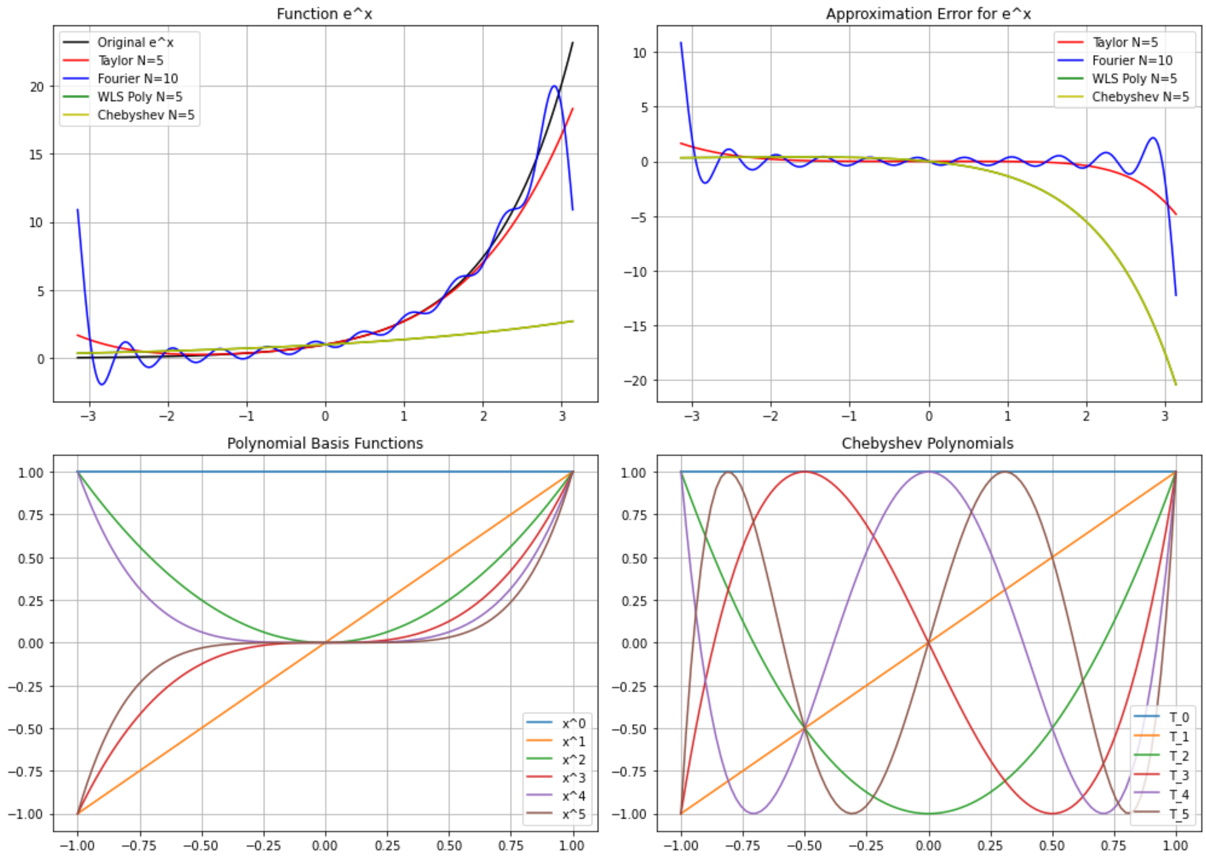
Sprawozdanie Lab4 Paweł Mozgowiec

Zad1

```python
import numpy as np
import matplotlib.pyplot as plt
from numpy.polynomial import Chebyshev as Cheb

# Ustawienia
x = np.linspace(-np.pi, np.pi, 1000)
functions = {
    'exp(x)': np.exp(x),
    '|sin(x)|': np.abs(np.sin(x))
}

# 1. Aproksymacja szeregami Taylora
def taylor_series_exp(x, n):
    return sum([x**i / np.math.factorial(i) for i in range(n)])

def taylor_series_sin_abs(x, n):
    # rozwinięcie Taylora |sin(x)| nie istnieje globalnie, przybliżymy przez |suma_n sin(x)|
    sin_approx = sum([(-1)**i * x**(2*i+1) / np.math.factorial(2*i+1) for i in range(n)])
    return np.abs(sin_approx)

# 2. Szereg Fouriera
def fourier_series_exp(x, mu, N):
    a0 = 2 * np.sinh(mu * np.pi) / np.pi
    result = a0 * 1/2*mu
    for n in range(1, N + 1):
        an = ((2 * np.sinh(mu * np.pi)) / (np.pi)) * (((-1)**n * (mu * np.cos(n * x) - n * np.sin(n * x)))) / (mu**2 + n**2)
        result += an
    return result

def fourier_series_abs_sin(x, N):
    result = 2 / np.pi
    for n in range(1, N + 1):
        result -= 4 / (np.pi) * ((4 * n**2 - 1)) * np.cos(2 * n * x)
    return result

# 3. WLS z wielomianami standardowymi
def wls_poly_fit(x, y, degree, weights):
    X = np.vander(x, degree + 1, increasing=True)
    W = np.diag(weights)
    coeffs = np.linalg.inv(X.T @ W @ X) @ X.T @ W @ y
    return X @ coeffs

# 4. WLS z wielomianami Czebyszewa
def wls_chebyshev_fit(x, y, degree, weights):
    T = Cheb.fit(x, y, degree, w=weights)
    return T(x)
```

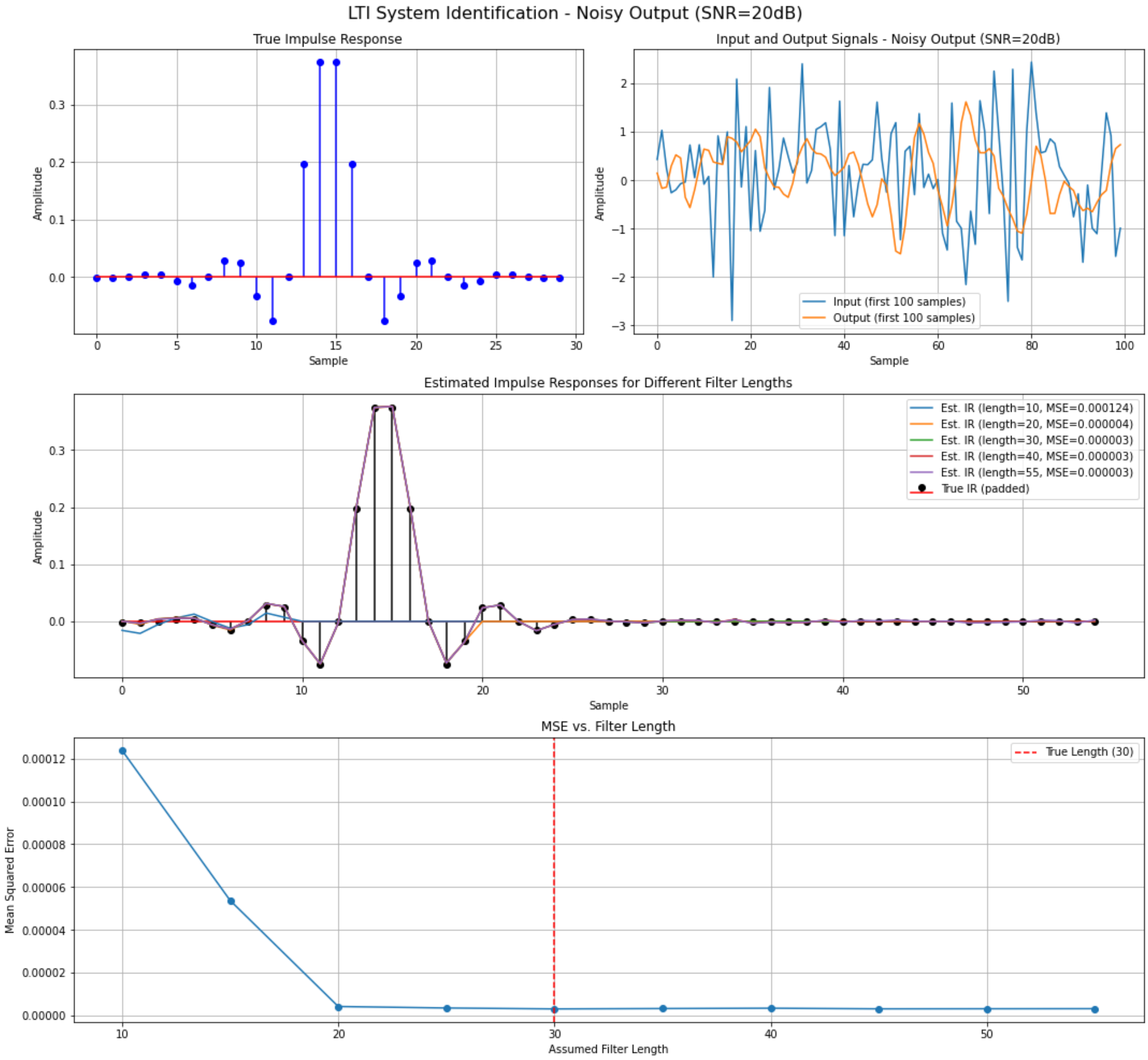

e^x Approximation with Different Weightings
Original e^x
Constant Weighting
Linear Weighting
e^x Approximation Errors
Constant Weighting Error
Linear Weighting Error
|sin(x)| Approximation with Different Weightings
Original |sin(x)|
Constant Weighting
Linear Weighting
|sin(x)| Approximation Errors
Constant Weighting Error
Linear Weighting Error
Function |sin(x)|
Original |sin(x)|
Taylor N=5
Fourier N=10
WLS Poly N=5
Chebyshev N=5
Approximation Error for |sin(x)|
Taylor N=5
Fourier N=10
WLS Poly N=5
Chebyshev N=5
Polynomial Basis Functions
x^0
x^1
x^2
x^3
x^4
x^5
Chebyshev Polynomials
T_0
T_1
T_2
T_3
T_4
T_5


Function e^x
Original e^x
Taylor N=5
Fourier N=10
WLS Poly N=5
Chebyshev N=5
Approximation Error for e^x
Taylor N=5
Fourier N=10
WLS Poly N=5
Chebyshev N=5
Polynomial Basis Functions
x^0
x^1
x^2
x^3
x^4
x^5
Chebyshev Polynomials
T_0
T_1
T_2
T_3
T_4
T_5

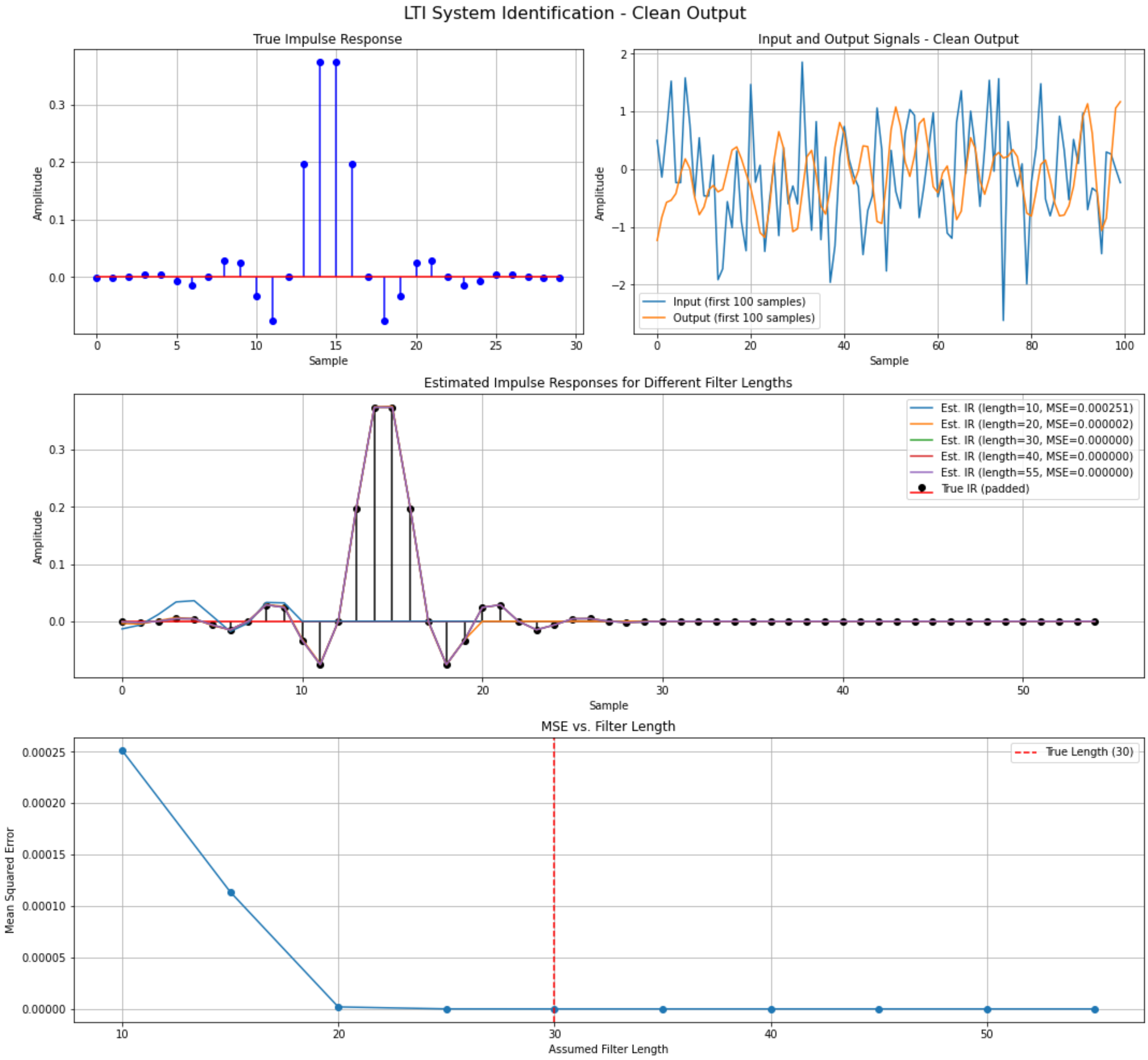
Zad2

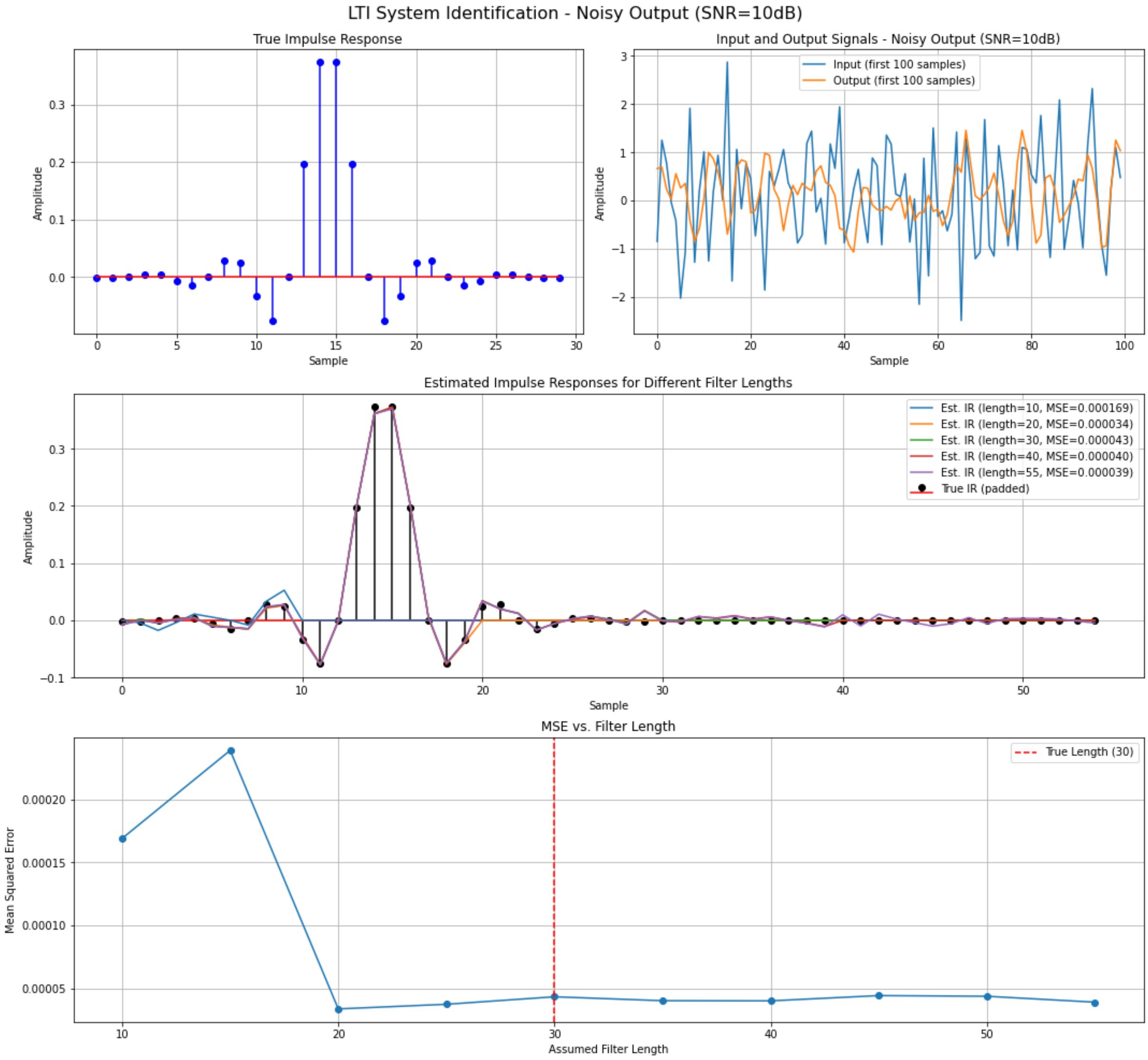
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
import matplotlib.gridspec as gridspec

def generate_gaussian_noise(length, mean=0, std=1.0):
    """Generate Gaussian white noise"""
    return np.random.normal(mean, std, length)

def apply_filter(input_signal, impulse_response):
    """Apply a filter to an input signal using convolution"""
    return np.convolve(input_signal, impulse_response, mode='valid')

def estimate_impulse_response(input_signal, output_signal, filter_length):
    """
    Estimate impulse response using cross-correlation method
    (Wiener-Hopf equation solution using numpy's lstsq)
    """
    # Make sure we have enough input samples for each output sample
    usable_output_length = len(output_signal)

    # Construct the Toeplitz matrix for the input
    X = np.zeros((usable_output_length, filter_length))

    # Make sure we don't go out of bounds
    for i in range(usable_output_length):
        # Check if we have enough input samples left
        if i + filter_length <= len(input_signal):
            X[i, :] = input_signal[i:i+filter_length]
        else:
            # If we're near the end, adjust the usable output length
            usable_output_length = i
            X = X[:usable_output_length, :]
            break

    # Use only the portion of output that corresponds to complete input segments
    output_used = output_signal[:usable_output_length]

    # Solve the least squares problem
    h_est, residuals, rank, s = np.linalg.lstsq(X, output_used, rcond=None)

    return h_est
```

```python

def run_experiment(true_impulse_response, filter_lengths, input_length=1000, noise_snr=None):
    """
    Run an experiment with different filter lengths
    noise_snr: Signal-to-Noise ratio in dB (None means no added noise)
    """
    # Generate input signal (Gaussian noise)
    # Add extra samples to ensure enough data for filtering with various lengths
    input_signal = generate_gaussian_noise(input_length + len(true_impulse_response) + max(filter_lengths))

    # Apply the true filter to get the output
    clean_output = apply_filter(input_signal, true_impulse_response)

    # Add noise to the output if specified
    if noise_snr is not None:
        # Calculate the power of the signal
        signal_power = np.mean(clean_output**2)

        # Calculate the noise power based on SNR
        noise_power = signal_power / (10**(noise_snr/10))

        # Generate and add noise
        output_noise = generate_gaussian_noise(len(clean_output), std=np.sqrt(noise_power))
        output_signal = clean_output + output_noise
    else:
        output_signal = clean_output

    # Estimate impulse response for each filter length
    results = []
    for length in filter_lengths:
        h_est = estimate_impulse_response(input_signal, output_signal, length)

        # Calculate MSE
        if length <= len(true_impulse_response):
            mse = np.mean((true_impulse_response[:length] - h_est)**2)
        else:
            # Pad true impulse response with zeros for comparison
            h_true_padded = np.pad(true_impulse_response, (0, length - len(true_impulse_response)))
            mse = np.mean((h_true_padded - h_est)**2)

        results.append((length, h_est, mse))

    return results, input_signal[:100], output_signal[:100]  # Return just a portion for visualization
```



LTI System Identification - Noisy Output (SNR=20dB)
True Impulse Response
Amplitude
Sample
Input and Output Signals - Noisy Output (SNR=20dB)
Input (first 100 samples)
Output (first 100 samples)
Estimated Impulse Responses for Different Filter Lengths
Est. IR (length=10, MSE=0.000124)
Est. IR (length=20, MSE=0.000004)
Est. IR (length=30, MSE=0.000003)
Est. IR (length=40, MSE=0.000003)
Est. IR (length=55, MSE=0.000003)
True IR (padded)
MSE vs. Filter Length
True Length (30)
Mean Squared Error
Assumed Filter Length


LTI System Identification - Clean Output
True Impulse Response
Amplitude
Sample
Input and Output Signals - Clean Output
Input (first 100 samples)
Output (first 100 samples)
Estimated Impulse Responses for Different Filter Lengths
Est. IR (length=10, MSE=0.000251)
Est. IR (length=20, MSE=0.000002)
Est. IR (length=30, MSE=0.000000)
Est. IR (length=40, MSE=0.000000)
Est. IR (length=55, MSE=0.000000)
True IR (padded)
MSE vs. Filter Length
True Length (30)
Mean Squared Error
Assumed Filter Length

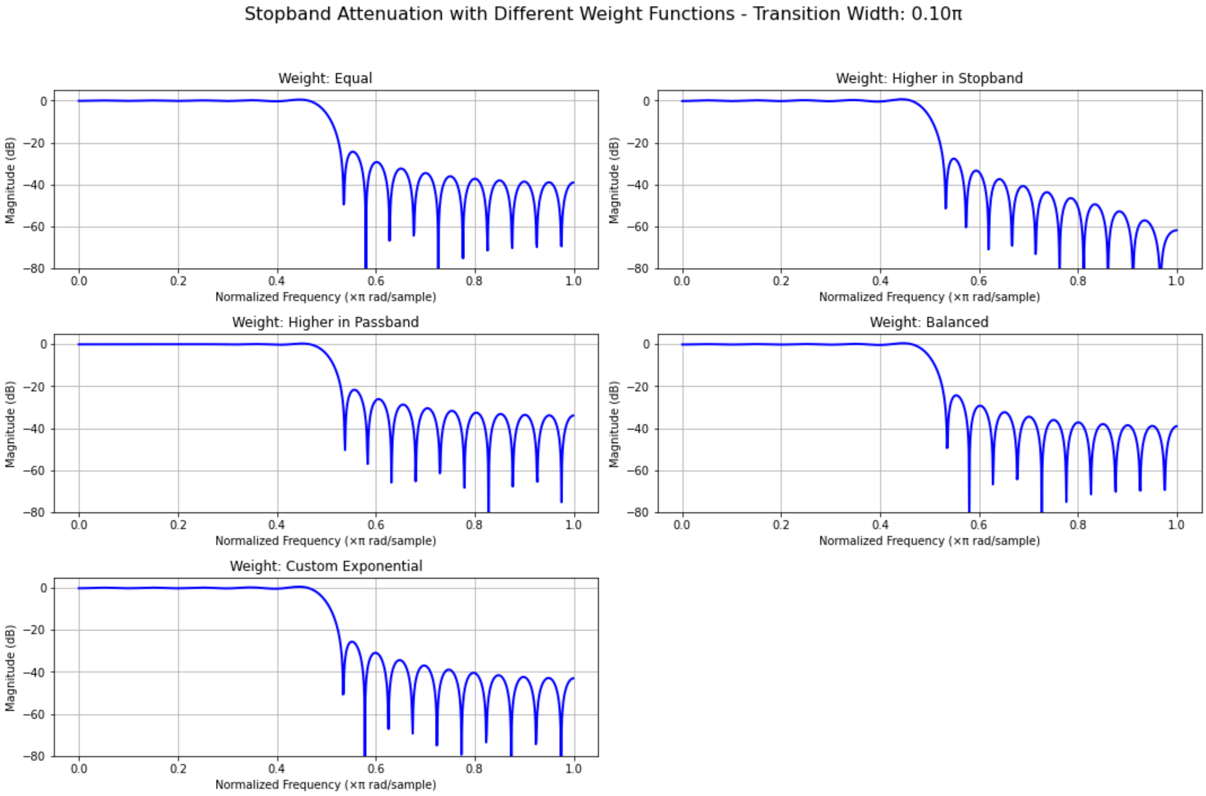


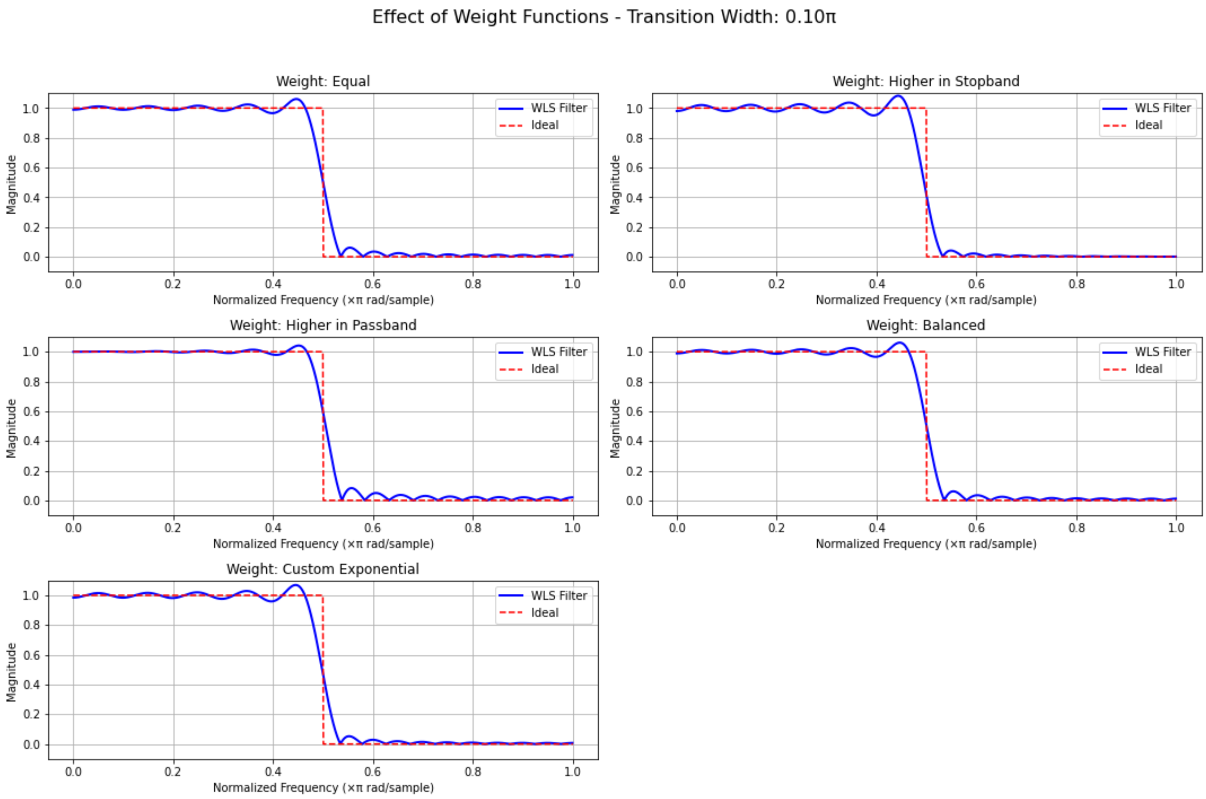
LTI System Identification - Noisy Output (SNR=10dB)
True Impulse Response
Amplitude
Sample
Input and Output Signals - Noisy Output (SNR=10dB)
Input (first 100 samples)
Output (first 100 samples)
Amplitude
Sample
Estimated Impulse Responses for Different Filter Lengths
Est. IR (length=10, MSE=0.000169)
Est. IR (length=20, MSE=0.000034)
Est. IR (length=30, MSE=0.000043)
Est. IR (length=40, MSE=0.000040)
Est. IR (length=55, MSE=0.000039)
True IR (padded)
Amplitude
Sample
MSE vs. Filter Length
True Length (30)
Mean Squared Error
Assumed Filter Length

Zad3

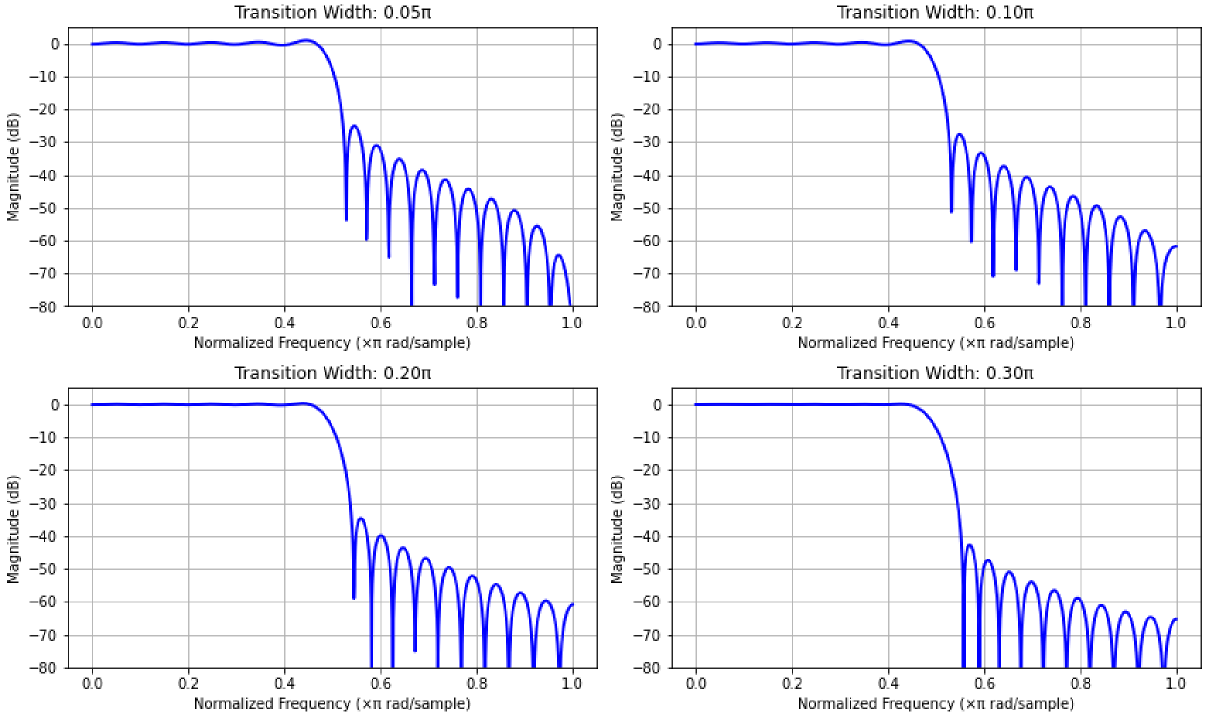
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from matplotlib.gridspec import GridSpec

def ideal_lp(cutoff, N):
    w = np.linspace(0, np.pi, N)
    H_ideal = np.zeros(N)
    H_ideal[w <= cutoff] = 1.0
    return H_ideal, w

def wls_fir_design(N, cutoff, weight_func, transition_width=0.1):
    # Ensure filter length is odd for firls (numtaps = N+1)
    if N % 2 == 0:
        N = N + 1  # Make filter order even so length (N+1) is odd

    # Number of frequency points for design
    num_freq_points = 1024
    w = np.linspace(0, np.pi, num_freq_points)

    # Ideal response
    H_ideal = np.zeros(num_freq_points)
    H_ideal[w <= cutoff] = 1.0

    # Create weight function
    weights = np.ones(num_freq_points)

    # Define transition band
    pass_edge = cutoff - transition_width/2
    stop_edge = cutoff + transition_width/2

    pass_band = w <= pass_edge
    stop_band = w >= stop_edge
    trans_band = ~(pass_band | stop_band)

    # Apply weighting function
    if callable(weight_func):
        weights = weight_func(w, pass_edge, stop_edge)
    elif weight_func == 1:
        # Equal weighting in passband and stopband
        weights[pass_band] = 1.0
        weights[stop_band] = 1.0
        weights[trans_band] = 0.01  # Small weight in transition band
    elif weight_func == 2:
        # Higher weight in stopband
        weights[pass_band] = 1.0
        weights[stop_band] = 10.0
        weights[trans_band] = 0.01
    elif weight_func == 3:
        # Higher weight in passband
        weights[pass_band] = 10.0
        weights[stop_band] = 1.0
        weights[trans_band] = 0.01
    elif weight_func == 4:
        # Custom weights to balance passband and stopband errors
        weights[pass_band] = 5.0
```

```python
        weights[trans_band] = 0.01

    # Design the filter using firls (Least-Squares FIR filter design)
    # Convert to band edges format required by firls
    bands = np.zeros(4)
    bands[0] = 0           # Start of passband
    bands[1] = pass_edge   # End of passband
    bands[2] = stop_edge   # Start of stopband
    bands[3] = np.pi       # End of stopband

    desired = np.array([1, 1, 0, 0])  # Desired response at band edges

    # Weight values at band edges
    weight_values = np.array([np.sqrt(np.mean(weights[pass_band])),
                             np.sqrt(np.mean(weights[stop_band]))])

    # Design the filter
    h = signal.firls(N, bands/np.pi, desired, weight=weight_values)

    # Calculate the frequency response
    w_plot, H = signal.freqz(h, worN=num_freq_points)
    w_plot = w_plot.real
    H = np.abs(H)

    return h, w_plot, H

def custom_weight(w, pass_edge, stop_edge):
    """Custom exponential weighting function"""
    weights = np.ones_like(w)
    pass_band = w <= pass_edge
    stop_band = w >= stop_edge
    trans_band = ~(pass_band | stop_band)

    weights[pass_band] = 5.0
    weights[stop_band] = 5.0 * np.exp((w[stop_band] - stop_edge))
    weights[trans_band] = 0.01

    return weights
```

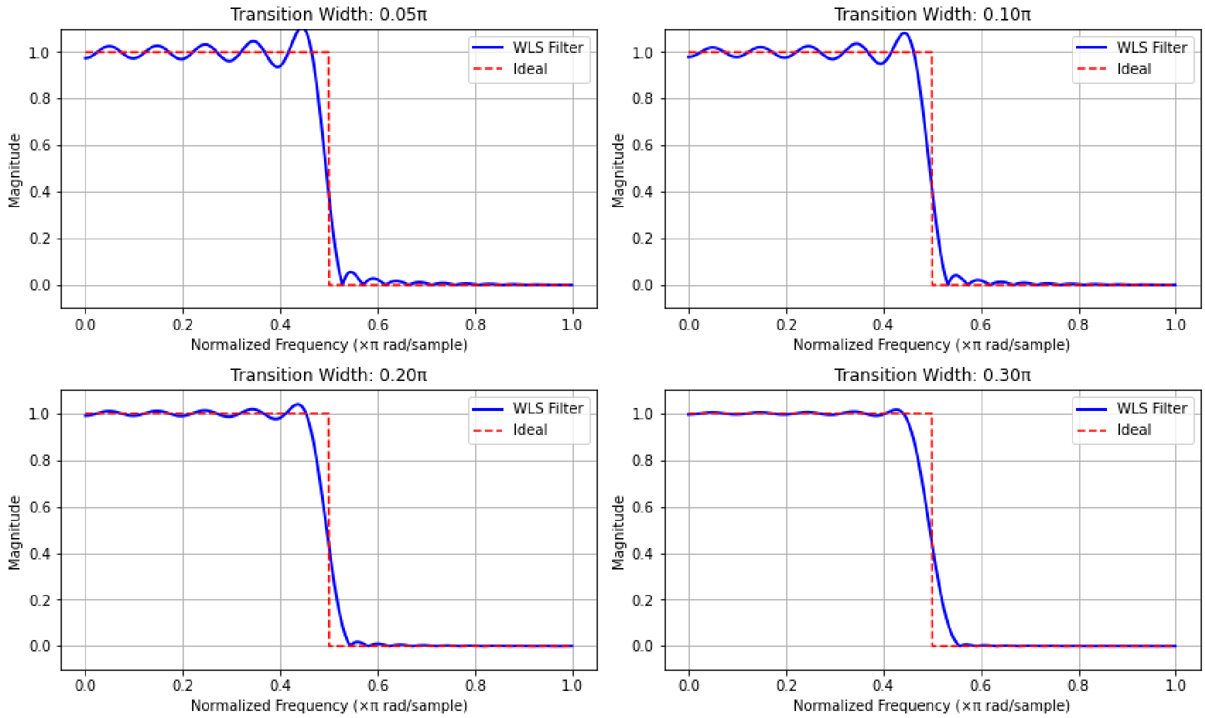

Stopband Attenuation with Different Weight Functions - Transition Width: 0.10π
Weight: Equal
Weight: Higher in Stopband
Weight: Higher in Passband
Weight: Balanced
Weight: Custom Exponential
Magnitude (dB)
Normalized Frequency (×π rad/sample)

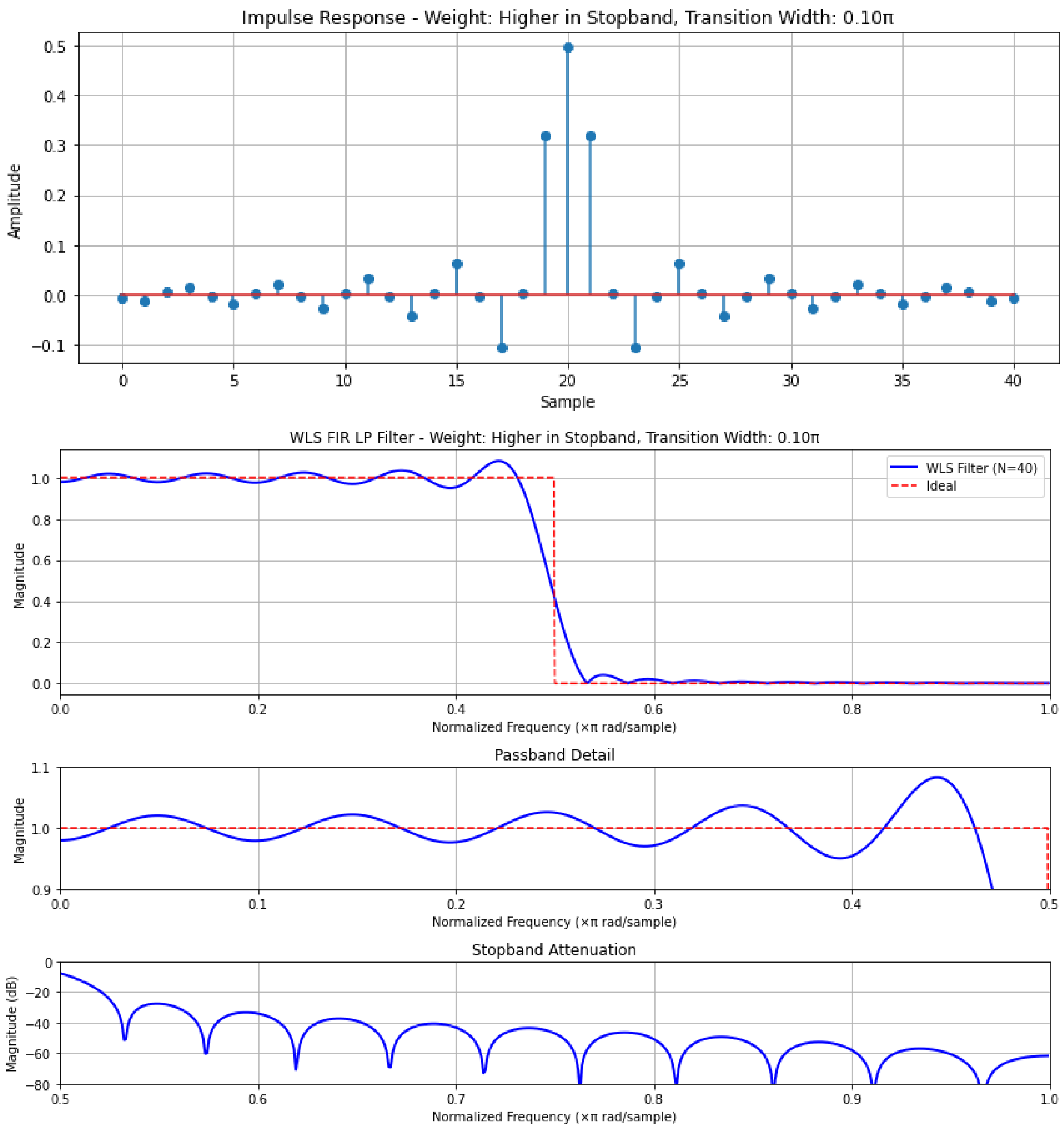

Effect of Weight Functions - Transition Width: 0.10π
Weight: Equal
Weight: Higher in Stopband
Weight: Higher in Passband
Weight: Balanced
Weight: Custom Exponential
WLS Filter
Ideal
Magnitude
Normalized Frequency (×π rad/sample)

Stopband Attenuation vs Transition Width - Weight: Higher in Stopband



Effect of Transition Band Width - Weight: Higher in Stopband

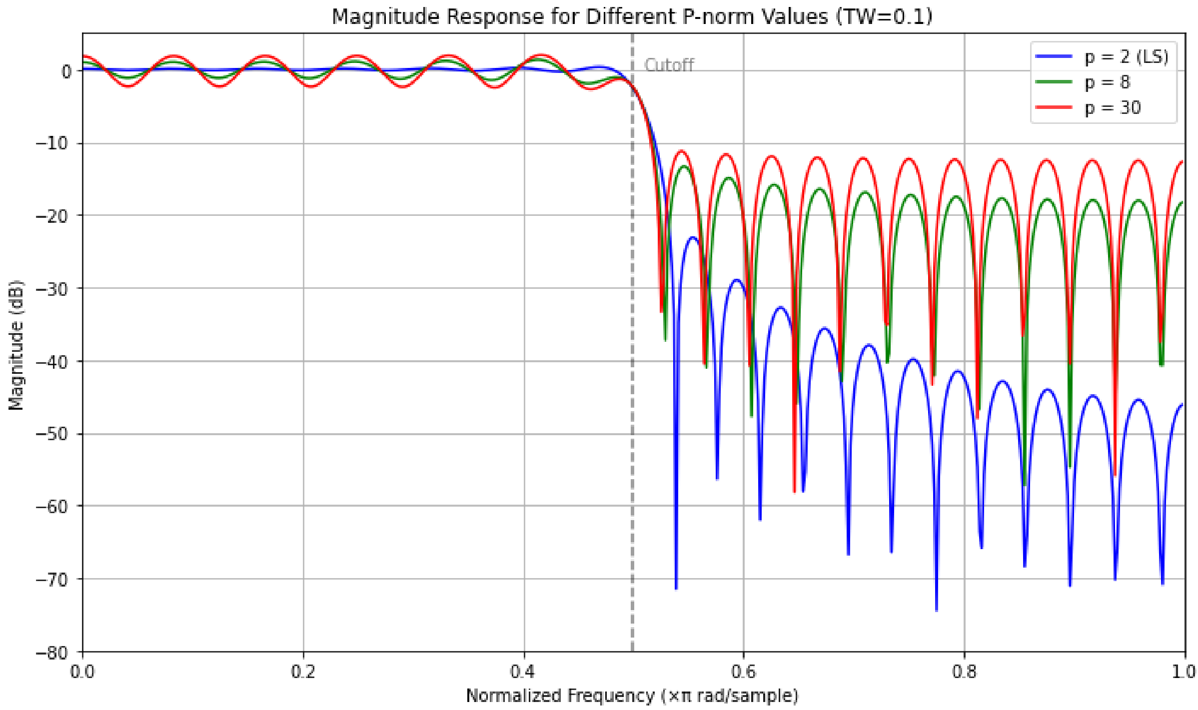


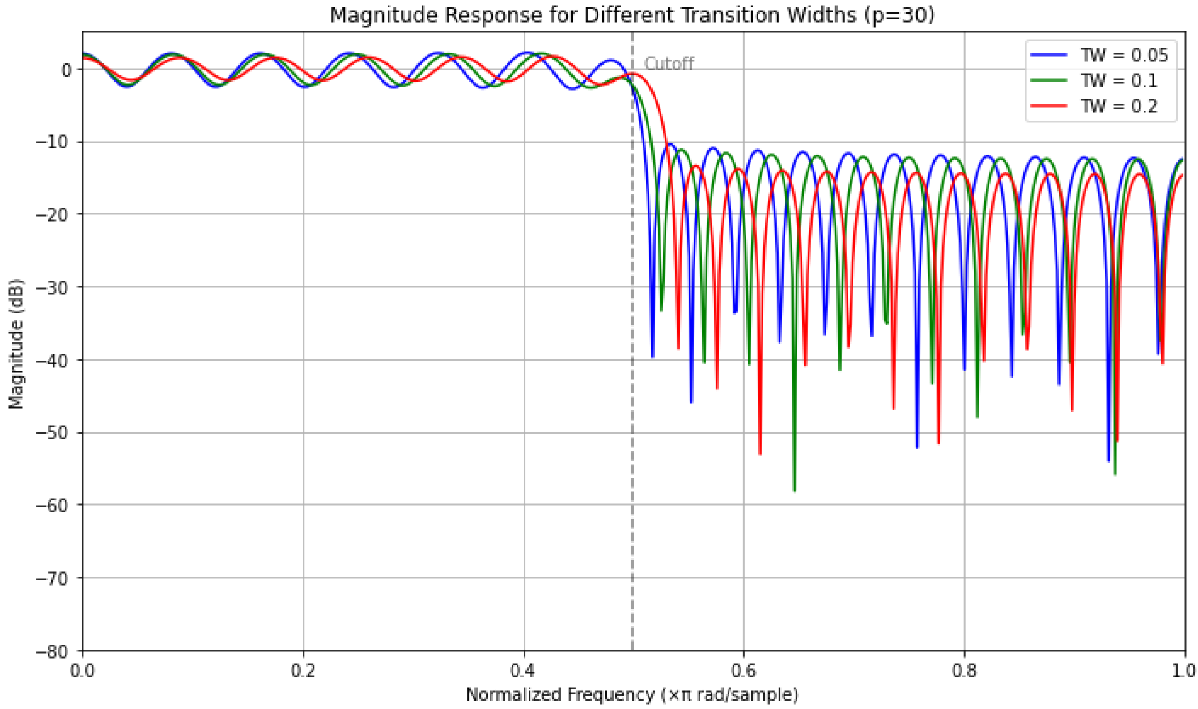

Impulse Response - Weight: Higher in Stopband, Transition Width: 0.10π
Amplitude
Sample
WLS FIR LP Filter - Weight: Higher in Stopband, Transition Width: 0.10π
WLS Filter (N=40)
Ideal
Magnitude
Normalized Frequency (×π rad/sample)
Passband Detail
Magnitude
Normalized Frequency (×π rad/sample)
Stopband Attenuation
Magnitude (dB)
Normalized Frequency (×π rad/sample)

Zad4

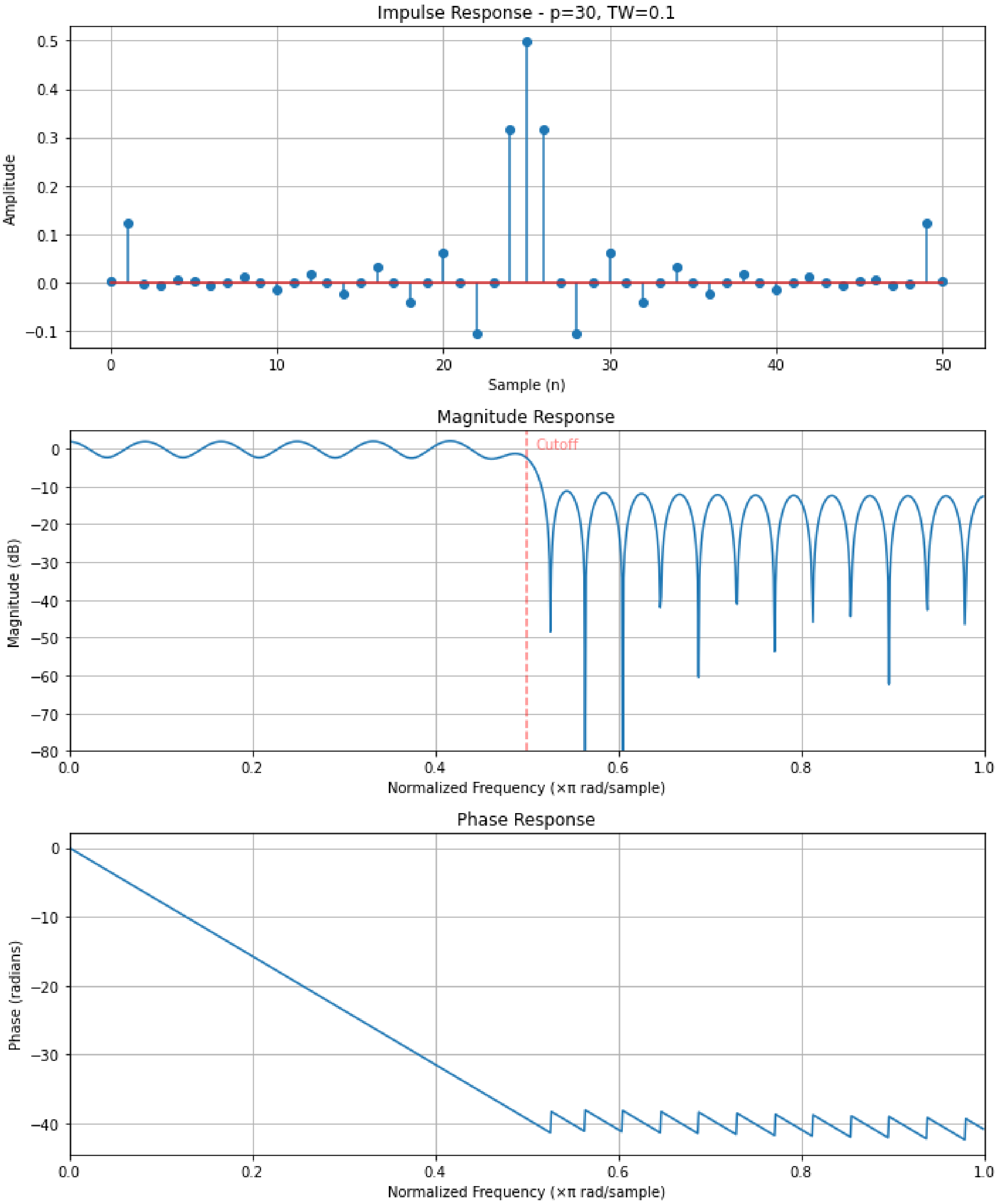
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal

def irwls(A, x, p=2, gamma=1.5, max_iter=200, stopeps=1e-7):
    pk = 2  # starting value of p

    # Find an initial LS solution
    c = np.linalg.lstsq(A, x, rcond=None)[0]
    xhat = A @ c

    for k in range(max_iter):
        pk = min(p, gamma * pk)  # p for this iteration
        e = x - xhat  # estimation error
        s = np.abs(e) ** ((pk - 2) / 2)  # new weights

        # Handle zero weights to avoid division by zero
        mask = s == 0
        if np.any(mask):
            s[mask] = 1e-10

        WA = np.diag(s) @ A  # weighted matrix
        weighted_x = s * x   # weighted vector

        # Weighted least-squares solution
        chat = np.linalg.lstsq(WA, weighted_x, rcond=None)[0]

        lambda_val = 1 / (pk - 1)
        cnew = lambda_val * chat + (1 - lambda_val) * c

        if np.linalg.norm(c - cnew) < stopeps:
            c = cnew
            print(f"Converged at iteration {k}, p={pk}")
            break

        c = cnew
        xhat = A @ c

    return c, s

def design_p_norm_fir(N, cutoff, p=2, trans_width=0.1):
    if N % 2 == 1:
        N += 1  # Ensure N is even for Type I filter

    # Number of frequency points
    num_freq = 512

    # Frequency grid from 0 to π
    omega = np.linspace(0, np.pi, num_freq)

    # Define desired frequency response (ideal low-pass)
    D = np.zeros(num_freq)

    # Passband
    D[omega <= cutoff - trans_width/2] = 1

    # Transition band - linearly decreasing
    trans_indices = np.logical_and(omega > cutoff - trans_width/2, omega < cutoff + trans_width/2)
    trans_omega = omega[trans_indices]
    D[trans_indices] = 0.5 + 0.5 * np.cos(np.pi * (trans_omega - cutoff) / trans_width)

    # Stopband
    D[omega >= cutoff + trans_width/2] = 0

    # Create the A matrix for the frequency response
    L = (N // 2) + 1  # Number of unique coefficients for linear-phase
    A = np.zeros((num_freq, L))

    for i in range(num_freq):
        for n in range(L):
            if n == 0:
                A[i, n] = 1
            else:
                A[i, n] = 2 * np.cos(omega[i] * n)

    # Solve using IRLS
    c, weights = irwls(A, D, p)
```

```python
        c, weights = irwls(A, D, p)

        # Construct the filter coefficients
        h = np.zeros(N+1)
        h[N//2] = c[0]  # Center coefficient

        for n in range(1, L):
            h[N//2 + n] = c[n]
            h[N//2 - n] = c[n]  # Symmetry for linear phase

        return h, D, omega, weights

    def compute_frequency_response(h, num_points=512):
        """
        Compute frequency response of the filter
        """
        w, H = signal.freqz(h, worN=num_points)
        return w, H
```





Magnitude Response for Different Transition Widths (p=30)
TW = 0.05
TW = 0.1
TW = 0.2
Cutoff
Magnitude (dB)
0
−10
−20
−30
−40
−50
−60
−70
−80
0.0
0.2
0.4
0.6
0.8
1.0
Normalized Frequency (×π rad/sample)


Impulse Response - p=30, TW=0.1
Amplitude
Sample (n)
Magnitude Response
Cutoff
Magnitude (dB)
Normalized Frequency (×π rad/sample)
Phase Response
Phase (radians)
Normalized Frequency (×π rad/sample)