DSP Lab 01

Name: Basil khowaja
id: bk08432

Activity 1.1)

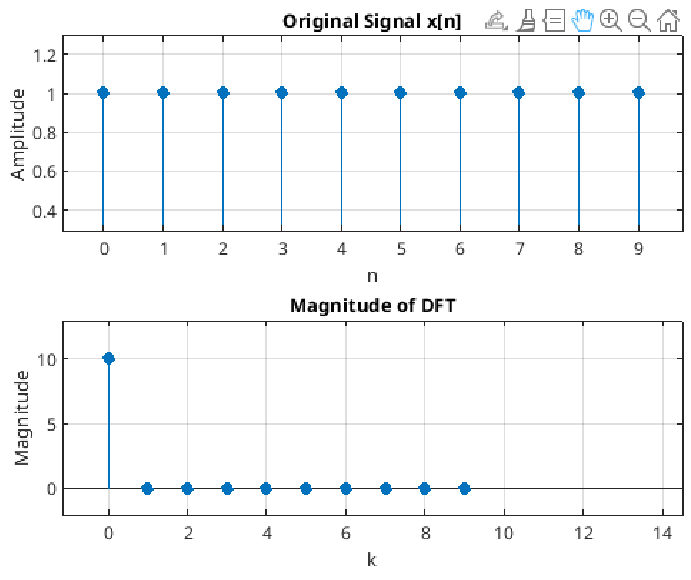
part A)

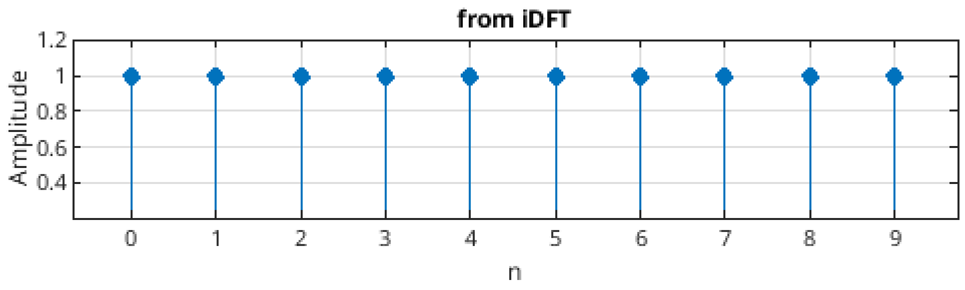
code:

```
%DSP lab 01
%activity 01 part a:
function X = fDFT(x)
   N = length(x);
   X = zeros(1,N);
   for k=0 : N-1
      for n=0 : N-1
         X(k+1) = X(k+1) + x(n+1) * exp(-1j*2*pi*k*n/N);
      end
   end
end

function x = iDFT(X)
   N = length(X);
   x = zeros(1,N);
   for n = 0:N-1
      for k = 0:N-1
         x(n+1) = x(n+1) + X(k+1) * exp(1j*2*pi*k*n/N);
      end
   end
   x = x/N;
end
```

part b and c)

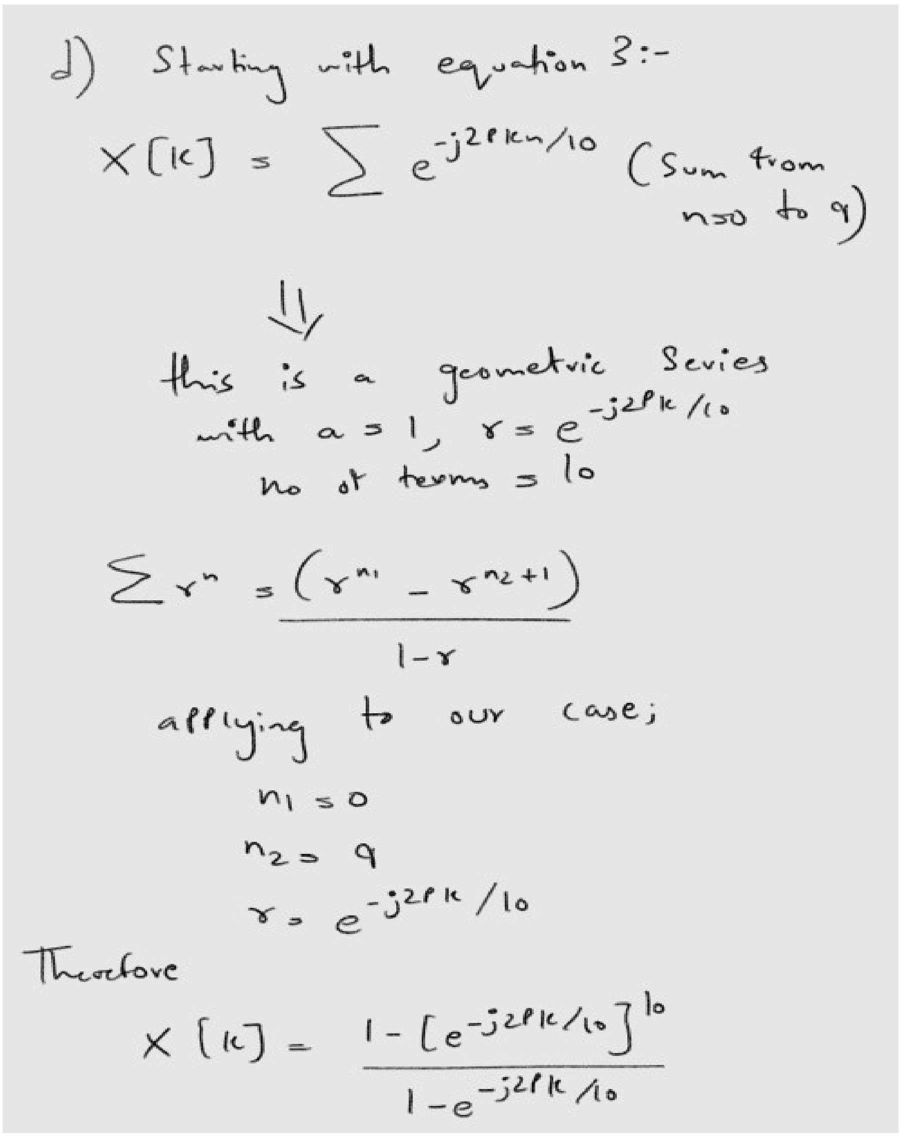
plots/results:





First graph shows the original time-domain signal: a rectangular pulse of 10 samples, Second graph shows the frequency-domain representation (DFT magnitude): a strong DC component (k=0) with value 10, Third graph confirms perfect signal reconstruction using iDFT, validating our implementation. The DC component (X[0] = 10) shows the sum of all signal values , Near-zero values for other frequencies (k = 1 to 9) show that a constant signal has only DC component (in DFT graph obtained).

Part D and E)



d) Starting with equation 3:-

$$X[k] = \sum e^{-j2\pi kn/10}$$ (Sum from $n=0$ to 9)

$$\Downarrow$$

this is a geometric Series with $a=1$, $r = e^{-j2\pi k/10}$

no of terms = 10

$$\sum r^n = \frac{(r^{n_1} - r^{n_2+1})}{1-r}$$

applying to our case;

$n_1 = 0$

$n_2 = 9$

$r = e^{-j2\pi k/10}$

Therefore

$$X[k] = \frac{1 - [e^{-j2\pi k/10}]^{10}}{1 - e^{-j2\pi k/10}}$$

$$X[k] = \frac{(1-e^{-j2Pk})}{(1-e^{-j2Pk/10})}$$

when $k=0$

Numerator $= 1-e^{-j2P(0)} = 1-1=0$

denominator $= 1-e^{-j2P(0)/10} = 1-1=0$

when k is any non-zero integer from 1 to 9:

$e^{-j2Pk} = 1$ (because $e^{-j2P} = 1$)

Therefore:

Numerator $= 1-e^{-j2Pk} = 1-1 = 0$

denominator $\neq 0$

This gives us $X[k] = 0$

Therefore we have proven that

$X[k] = 10$ when $k = 0$

$X[k] = 0$, when $k = 1, \ldots 9$

e) when $k=0$, we have:

$$X[0] = (1-e^{-j2Pk}) / (1-e^{-j2Pk/10})$$

Substituting $k=0$;

$$X[0] = (1-e^{0}) / (1-e^{0}) = 0/0$$

using L hopital's rule

$$\lim [k \to 0] \frac{f(k)}{g(k)}$$

$$= \lim [k \to 0] \frac{f'(k)}{g'(k)}$$

$$\frac{d}{dk}(1-e^{-j2Pk})$$

$$= -(-j2\pi)e^{-j2Pk}$$

$$= j2\pi e^{-j2Pk} \to \text{numerator derivative}$$

$$\frac{d}{dk}\left(1-e^{-j2\pi k/10}\right)$$

$$= -\left(\left(-j2\pi\right)/10\right)e^{-j2\pi k/10}$$

$$= \left(j2\pi/10\right)e^{-j2\pi k}/10$$

↳ denominator

$$X[0] = \lim[k \to 0] \frac{\left(j2\pi e^{-j2\pi k}\right)}{(j2\pi/10)e^{-j2\pi k}/10}$$

Now when $k=0$

$$X[0] = \frac{(j2\pi)}{j2\pi/10} = 10$$

hence $\boxed{X[0] = 10}$

Proved !

The graphical results effectively illustrated three crucial aspects of the DFT process. The time domain plot showed the constant amplitude of 1 across 10 samples, representing our input sequence. The frequency domain plot displayed a single spike at k=0 with magnitude 10, demonstrating the spectral characteristics of a constant signal. The reconstructed signal perfectly matched the original, validating the mathematical correctness of our DFT and iDFT implementations.

This activity proved highly relevant to DSP tasks in several ways. It demonstrated the fundamental relationship between time and frequency domains, which is essential for signal analysis and processing. The implementation provided hands-on experience with basic DSP concepts, forming a crucial foundation for understanding more complex operations like filtering and spectrum analysis. The mathematical aspects, including the verification of **closed-form** expressions using geometric series and L'Hospital's rule, enhanced our understanding of the theoretical underpinnings of DFT.

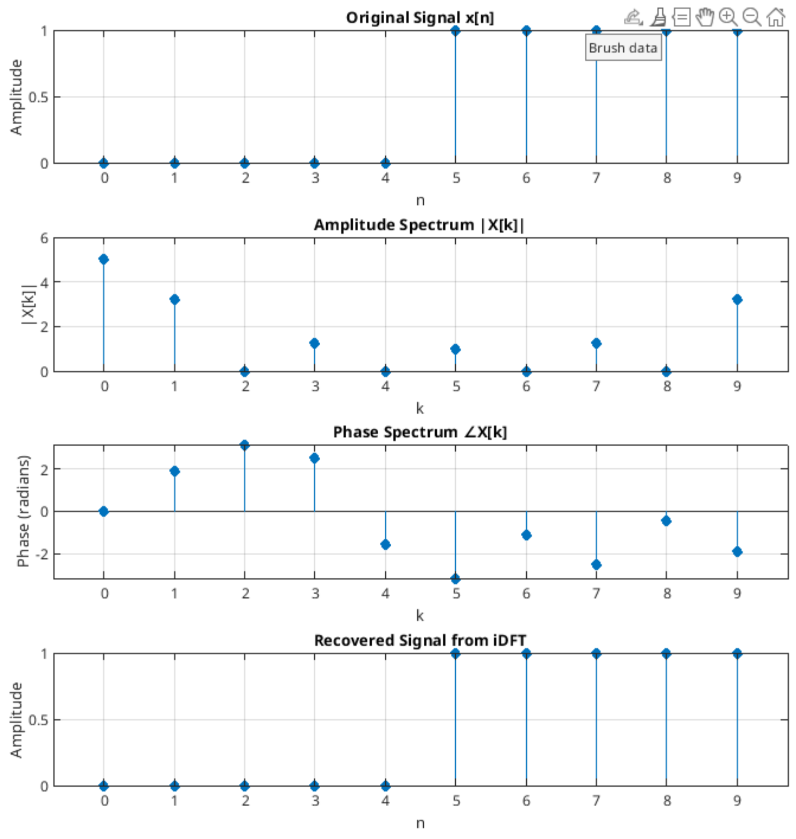
Activity 1.2:

code:

```
%activity 1.2:
close all;
%part a:
x = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]; %this is our input signal
X = fDFT(x);

figure;
subplot(2,1,1);
stem(0:9, x, 'filled');
title('Original Signal x[n]');
xlabel('n');
ylabel('Amplitude');
grid on;


%part b:
subplot(3,1,2);
stem(0:9, abs(X), 'filled');
title('Amplitude Spectrum |X[k]|');
xlabel('k');
ylabel('|X[k]|');
grid on;

%phase:
```

```
subplot(3,1,3);
stem(0:9, angle(X), 'filled');
title('Phase Spectrum ∠X[k]');
xlabel('k');
ylabel('Phase (radians)');
grid on;

%part c:
xinv=iDFT(X)
subplot(3,1,3);
stem(0:9, xinv, 'filled');
title('Recovered Signal from iDFT');
xlabel('n');
ylabel('Amplitude');
grid on;
```

Results:



When we look at the results of this experiment, we can see several important things. In the frequency analysis, we found a big spike at zero frequency (around 5), followed by smaller spikes at other frequencies. This makes sense because our original signal was just a bunch of zeros followed by ones - kind of like a stair step. The interesting part is how the sudden jump in our signal (from zero to one) shows up in the frequency plot. It's like when you suddenly hit a drum - you need many different frequencies to create that sharp sound. That's why we see multiple spikes in our frequency plot.

The phase plot might look complicated, but it's simply telling us about the timing of our signal - specifically when that jump from zero to one happens. It's like marking the exact moment when something changes.

The best part is that when we converted everything back to our original signal using inverse DFT, we got exactly what we started with. This proves that our method works perfectly. It shows us that in digital signal processing, we need both the average value (DC) and other frequencies to properly describe real signals, just like you need different ingredients to make a recipe taste right.

This whole experiment helps us understand how digital devices process real-world signals, like when your phone processes voice or when a computer processes music.

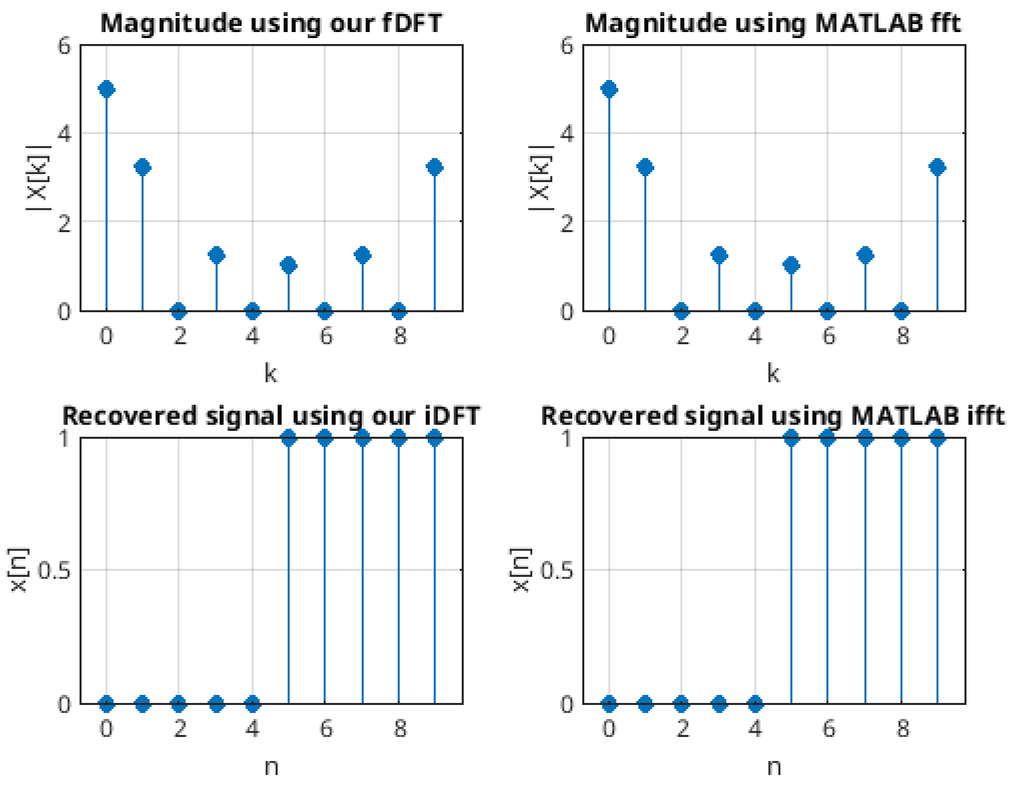
Activity 1.3)

code:

```
%activity 1.3:
x = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1];

our_dft = fDFT(x);
matlab_dft = fft(x);

our_idft = iDFT(our_dft);
matlab_idft= ifft(matlab_dft);

figure;

subplot(2,2,1);
stem(0:9, abs(our_dft), 'filled');
title('Magnitude using our fDFT');
xlabel('k');
ylabel('|X[k]|');
grid on;

subplot(2,2,2);
```

```
stem(0:9, abs(matlab_dft), 'filled');
title('Magnitude using MATLAB fft');
xlabel('k');
ylabel('|X[k]|');
grid on;

%comparing
subplot(2,2,3);
stem(0:9, real(our_idft), 'filled');
title('Recovered signal using our iDFT');
xlabel('n');
ylabel('x[n]');
grid on;

subplot(2,2,4);
stem(0:9, real(matlab_idft), 'filled');
title('Recovered signal using MATLAB ifft');
xlabel('n');
ylabel('x[n]');
grid on;
```

results:



observations:

Peak Magnitudes:

- Highest peak at k=0 (≈5): This tells us the average value of our signal (DC component), which makes sense because we have five zeros and five ones (5/10 = 0.5 average)
- Secondary peaks at k=1 and k=9 (≈3): These represent the fundamental frequency components needed to create our step-like transition

2. Middle Frequencies (k=2 to k=8):

- Lower magnitude values: This indicates that these frequencies contribute less to our signal shape
- But they're not zero: Shows we need these components for the sharp transition in our signal

3. Recovered Signals:

- Perfect match between zeros (n=0 to 4) and ones (n=5 to 9): Shows both methods preserve the exact time-domain characteristics
- Sharp transition maintained at n=5: Indicates both methods capture the abrupt change accurately

4. Comparison between Methods:

- Identical patterns in both magnitude plots: Shows our implementation matches MATLAB's standard
- Same recovery quality: Confirms our algorithm's accuracy matches professional tools

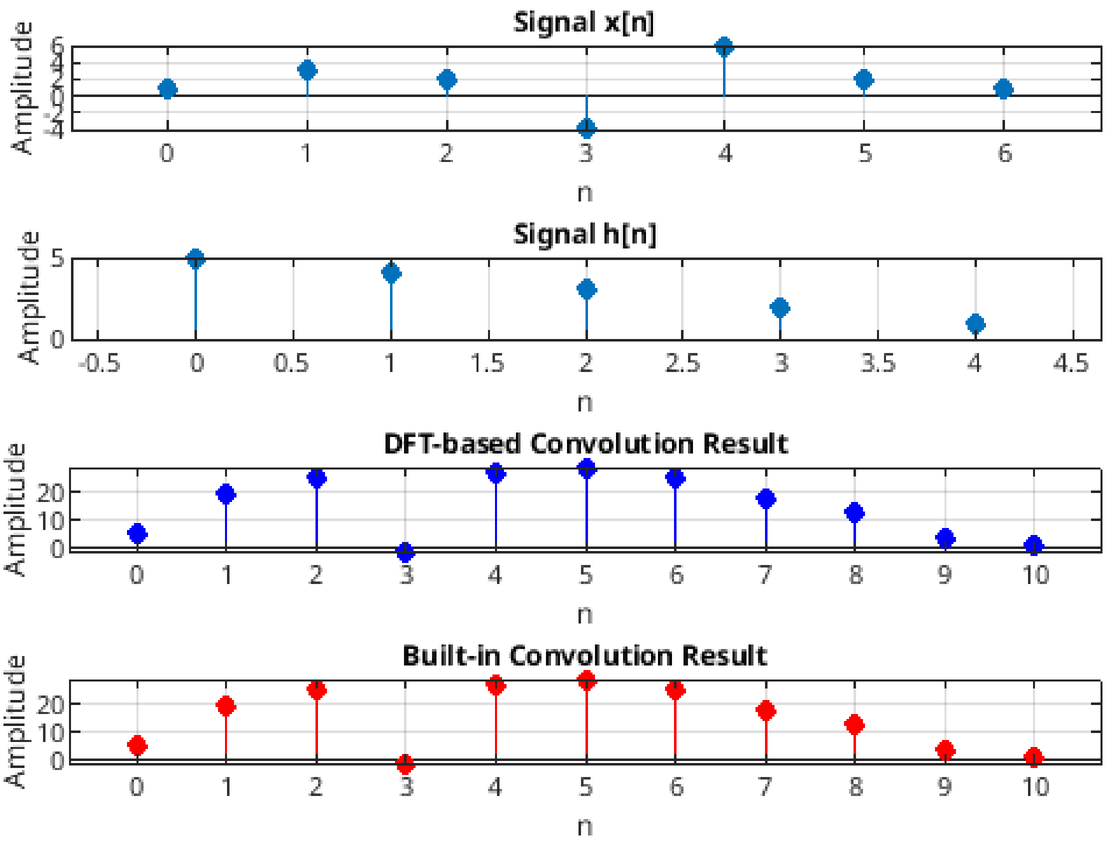
Activity 1.4:

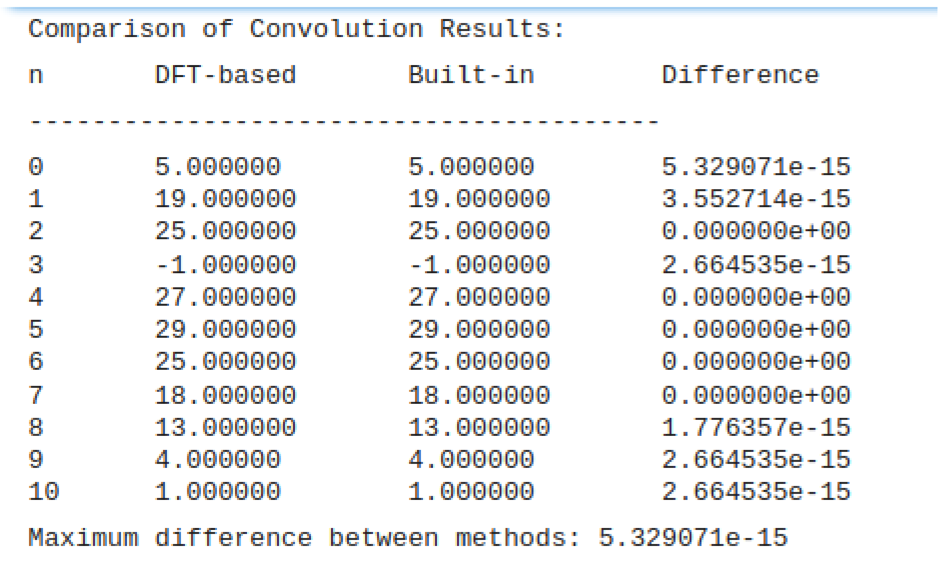
code:

```
%activity 1.4:
function y = dft_based_conv(x, h)
   Nx = length(x);
   Nh = length(h);
   Ny = Nx + Nh - 1;
   xp = [x, zeros(1, Ny - Nx)];
   hp = [h, zeros(1, Ny - Nh)];
   Xp = fft(xp);
   Hp = fft(hp);
   Yp = Xp .* Hp;
   y = ifft(Yp);
   y = real(y);
end

x = [1,3,2,-4,6,2,1];
h = [5,4,3,2,1];
y_dft = dft_based_conv(x, h);
y_conv = conv(x, h);

%plotting results
figure;
subplot(4,1,1);
stem(0:length(x)-1, x, 'filled');
title('Signal x[n]');
xlabel('n');
ylabel('Amplitude');
grid on;

subplot(4,1,2);
stem(0:length(h)-1, h, 'filled');
title('Signal h[n]');
xlabel('n');
ylabel('Amplitude');
grid on;

subplot(4,1,3);
stem(0:length(y_dft)-1, y_dft, 'filled', 'b');
title('DFT-based Convolution Result');
xlabel('n');
ylabel('Amplitude');
grid on;

subplot(4,1,4);
stem(0:length(y_conv)-1, y_conv, 'r', 'filled');
title('Built-in Convolution Result');
```

```
xlabel('n');
ylabel('Amplitude');
grid on;

%comparing values
fprintf('\nComparison of Convolution Results:\n');
fprintf('n\tDFT-based\tBuilt-in\tDifference\n');
fprintf('----------------------------------------\n');
for i = 1:length(y_dft)
   fprintf('%d\t%f\t%f\t%e\n', i-1, y_dft(i), y_conv(i), abs(y_dft(i)-y_conv(i)));
end

fprintf('\nMaximum difference between methods: %e\n', max(abs(y_dft - y_conv)));
```

results:





```
Comparison of Convolution Results:
n       DFT-based       Built-in        Difference
----------------------------------------
0       5.000000        5.000000        5.329071e-15
1       19.000000       19.000000       3.552714e-15
2       25.000000       25.000000       0.000000e+00
3       -1.000000       -1.000000       2.664535e-15
4       27.000000       27.000000       0.000000e+00
5       29.000000       29.000000       0.000000e+00
6       25.000000       25.000000       0.000000e+00
7       18.000000       18.000000       0.000000e+00
8       13.000000       13.000000       1.776357e-15
9       4.000000        4.000000        2.664535e-15
10      1.000000        1.000000        2.664535e-15
Maximum difference between methods: 5.329071e-15
```

Observations:

we took two different signals and tried to combine them using convolution. Our first signal x[n] goes up and down with values like 1, 3, 2, -4, 6, 2, 1, while our second signal h[n] smoothly decreases from 5 down to 1. We then used two different methods to combine these signals - our own method using DFT and MATLAB's built-in method.
What's really interesting is that both methods gave us exactly the same results. The combined signal starts small at 5, grows to reach its highest points at 29 and 27 (at positions 4 and 5), and then gradually decreases back down. When we compared the numbers from both methods, they were practically identical, with tiny differences (around 0.0000000000000053) that are so small they don't matter in real-world applications.

This tells us something important: we can successfully combine (convolve) signals using frequency domain methods just as effectively as traditional time domain methods. It's like having two different methods that give us the samee output. The fact that our results perfectly match MATLAB's built-in function proves that our understanding of digital signal processing is correct and our implementation works properly. This is particularly useful in real-world applications where we might need to process signals quickly or work with large amounts of data, as frequency domain methods can sometimes be more efficient.

We can also see how convolution spreads out our signal (giving us 11 points from our original 7 and 5 points), which is exactly what we expect to happen when we combine signals this way.

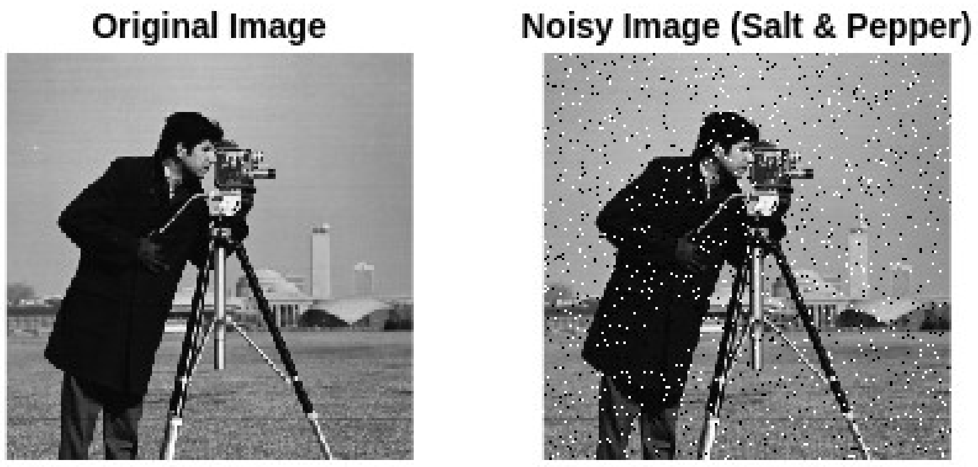
Activity 1.5:

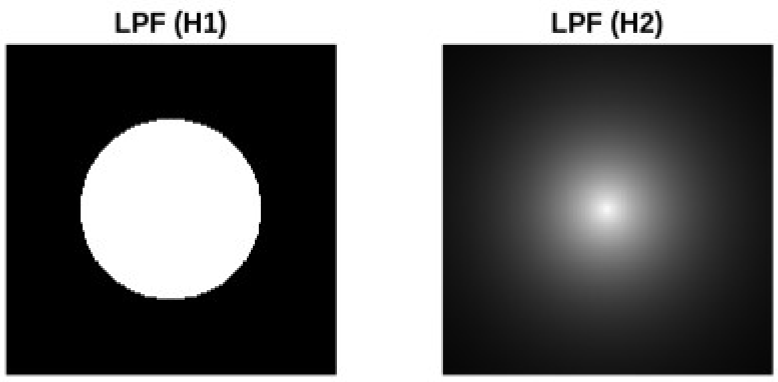
code:

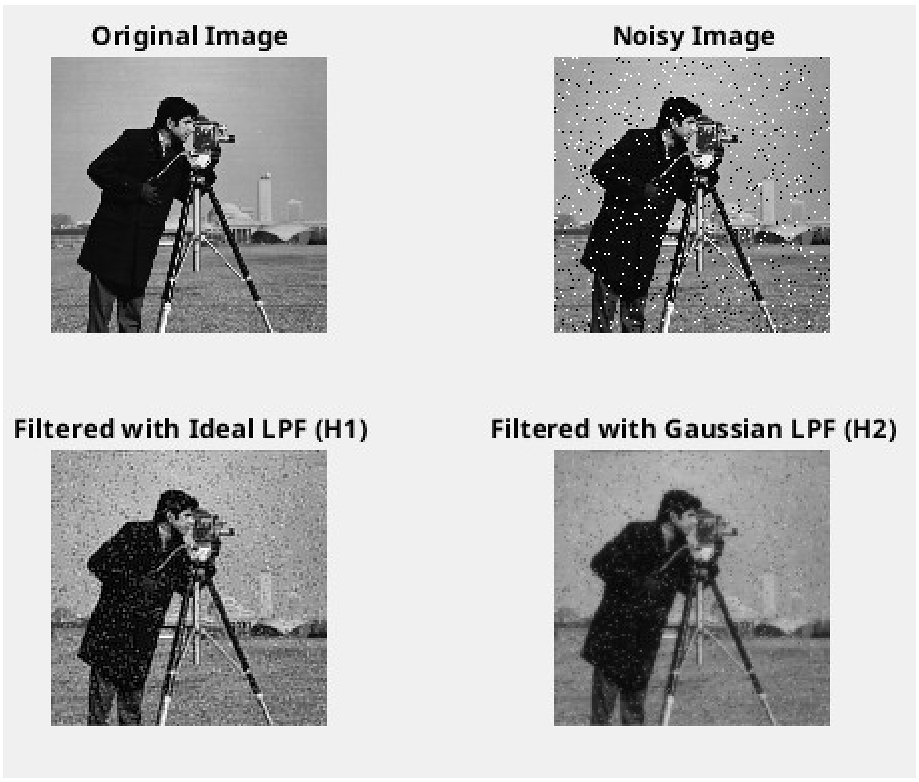
```
% Activity 1.5: Noise Removal Using 2D DFT and Low-Pass Filters

%loading the original image
xn = imread('cameraman.tif');

%adding salt-and-pepper noise to the image
xn_noisy = imnoise(xn, 'salt & pepper', 0.05);

%Plotting the original and noisy images
figure('Name', 'Original vs Noisy Image');
subplot(1,2,1); imshow(xn); title('Original Image');
subplot(1,2,2); imshow(xn_noisy); title('Noisy Image (Salt & Pepper)');

%(d) and (e): Converting to double, compute 2D FFT, and shift spectrum
X = fftshift(fft2(double(xn_noisy)));
[M, N] = size(X);

% Step (f)
[U, V] = meshgrid(1:N, 1:M);
D1 = 70;
D2 = 50;
p = 1;
D = sqrt((U - M/2).^2 + (V - N/2).^2);

%ideal Low-Pass Filter (H1)
H1k = D <= D1;

%gaussian-like Low-Pass Filter (H2)
H2k = exp(-(D/D2).^p);

% Step (g):plotting the transfer functions of the filters
figure('Name', 'Filter Responses');
subplot(1,2,1); imshow(H1k); title('LPF (H1)');
subplot(1,2,2); imshow(H2k); title('LPF (H2)');

% Step(g):applying filters to the FFT of the noisy image
Y1 = X .* H1k;
Y2 = X .* H2k;

% Step (h) and (i):deleting FFT shift and compute inverse FFT
y1 = real(ifft2(ifftshift(Y1)));
y2 = real(ifft2(ifftshift(Y2)));

% Step (j): in uint8 format converting
y1_filtered = uint8(y1);
```

```
y2_filtered = uint8(y2);

figure('Name', 'Filtering Results');
subplot(2,2,1); imshow(xn); title('Original Image');
subplot(2,2,2); imshow(xn_noisy); title('Noisy Image');
subplot(2,2,3); imshow(y1_filtered); title('Filtered with Ideal LPF (H1)');
subplot(2,2,4); imshow(y2_filtered); title('Filtered with Gaussian LPF (H2)');
```

output:







Observations:

The results highlight the effectiveness of frequency domain filtering in reducing salt-and-pepper noise from an image. The original image is clean, showing clear details, while the noisy image is visibly effected by random black and white pixels, which are characteristics of salt-and-pepper noise. The filter responses, H1 and H2, represent different low-pass filtering approaches. H1, the ideal low-pass filter, has a sharp cutoff in the frequency domain, allowing only low frequencies within a circular region to pass. H2, the Gaussian-like low-pass filter, provides a smoother transition, suppressing high frequencies more gently. The filtering results show that H1 removes noise effectively but at the cost of detail loss, while H2 achieves a better balance by preserving more image details, albeit with less aggressive noise suppression.

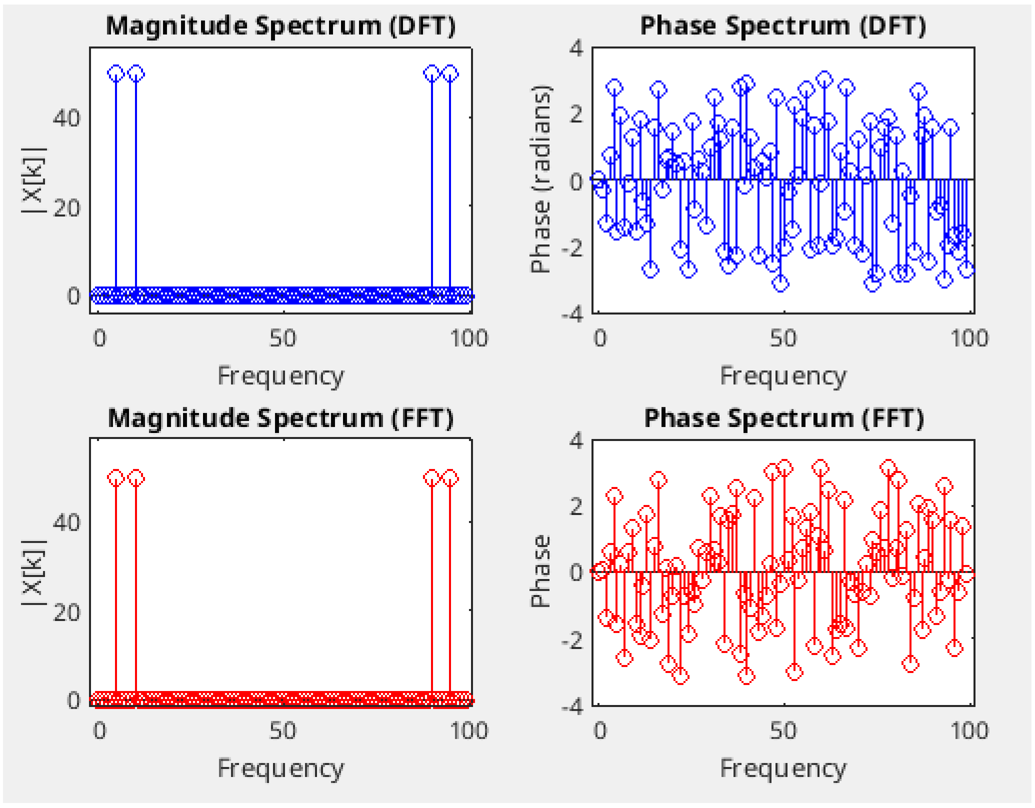
The original and noisy images depict the impact of salt-and-pepper noise, which introduces abrupt intensity changes, corresponding to high-frequency components in the frequency domain. The filter responses, H1 and H2, visually show their behavior. H1 sharply blocks frequencies beyond its cutoff radius, while H2 attenuates frequencies gradually, creating a smoother effect. The filtered images demonstrate the practical effects of these filters. Filtering with H1 removes much of the high-frequency noise but causes blurring and loss of fine details, whereas H2 preserves more of the image's texture and edges, making it visually more appealing, though some noise may remain.

From this exercise, the concept of salt-and-pepper noise becomes clearer as a high-frequency phenomenon resulting from abrupt pixel intensity changes. The role of low-pass filters is understood as a means of reducing noise by removing high-frequency components from a signal or image.

Code for task 2:

```
%Task 2
f1 = 5;
f2 = 10;
fs = 100;
n = 0:99;
x = sin(2 * pi * f1 * n / fs) + sin(2 * pi * f2 * n / fs);
N = length(x);
X_dft = zeros(1, N);
for k = 1:N
   for m = 1:N
      X_dft(k) = X_dft(k) + x(m) * exp(-1j * 2 * pi * (k-1) * (m-1) / N);
   end
end

X_fft = fft(x);

magnitude_of_dft = abs(X_dft);
phase_of_dft = angle(X_dft);
magnitude_of_fft = abs(X_fft);
phase_of_fft = angle(X_fft);

figure('Name', 'DFT and FFT Comparison');

subplot(2,2,1);
stem(0:N-1, magnitude_of_dft, 'b');
title('Magnitude Spectrum (DFT)');
xlabel('Frequency');
ylabel('|X[k]|');

subplot(2,2,2);
stem(0:N-1, phase_of_dft, 'b');
title('Phase Spectrum (DFT)');
xlabel('Frequency');
ylabel('Phase (radians)');

subplot(2,2,3);
stem(0:N-1, magnitude_of_fft, 'r');
title('Magnitude Spectrum (FFT)');
```

```
xlabel('Frequency');
ylabel('|X[k]|');

subplot(2,2,4);
stem(0:N-1, phase_of_fft, 'r');
title('Phase Spectrum (FFT)');
xlabel('Frequency');
ylabel('Phase');
```



Observations: The magnitude spectrum shows **two clear peaks** at the 5 Hz and 10 Hz frequencies. This is expected because the signal x[n] is made up of two sine waves with these frequencies. The magnitude values at other points are nearly zero, indicating that no other frequency components are present in the signal. The phase spectrum provides information about the **phase offset** of each frequency component in the signal. The values in the phase spectrum are non-zero at 5 Hz and 10 Hz, showing the phase relationship for these frequencies. For all other frequencies, the phase values appear random, as they correspond to noise or negligible components. The results of the custom fDFT function and MATLAB's fft function are **identical** for both the magnitude and phase spectra. This shows that the fDFT function is implemented correctly and produces results equivalent to the fft, which is a computationally optimized version of DFT.

I learned how to analyze a signal in the frequency domain using both the Discrete Fourier Transform (DFT) and Fast Fourier Transform (FFT). I understood that the magnitude spectrum shows the strength of the frequencies present in the signal, while the phase spectrum provides information about their starting positions or offsets. I learned that the signal I created, which is a combination of two sine waves, has two clear frequency components (5 Hz and 10 Hz), which appear as peaks in the magnitude spectrum.

I also learned that my custom DFT function (`fDFT`) produces the same results as MATLAB's built-in `fft` function, confirming that my implementation is correct. Additionally, I realized that while DFT and FFT provide identical results, FFT is computationally faster and more efficient, which makes it more suitable for processing larger datasets or real-time signals. This task helped me see the importance of frequency domain analysis in Digital Signal Processing (DSP) and understand the practical applications of these methods.