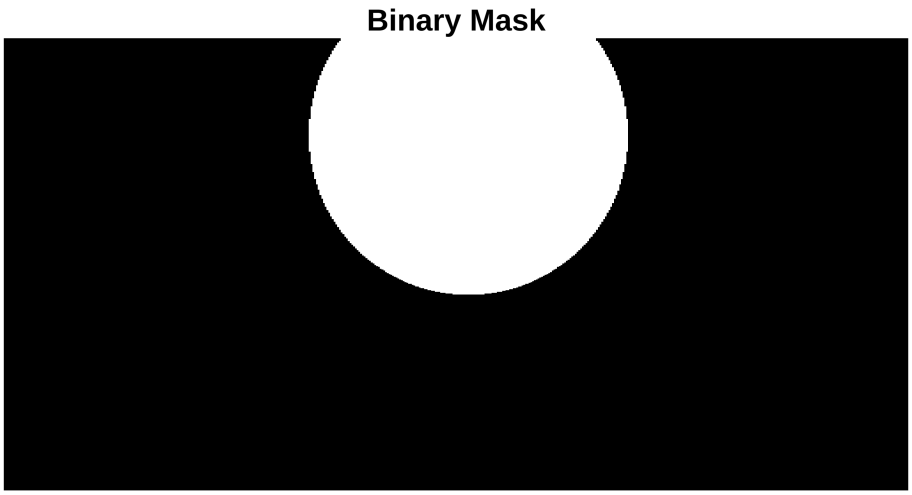
1st question

```
% Load an image
img = imread('ass pic.jpg');
h_im = imshow(img);  % Display the image

% Draw a circular mask on the image
circ = drawcircle('Center', [525, 110], 'Radius', 180);
```

```
BW = createMask(circ);  % Create binary mask

% Display the binary mask
figure;
imshow(BW);
title('Binary Mask');
```



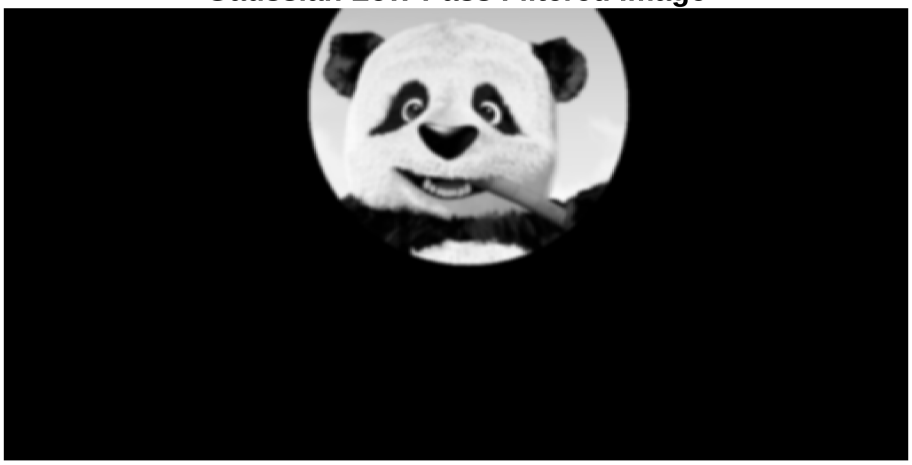

```
% Apply the mask to the image
maskedImg = img;
maskedImg(repmat(~BW, [1, 1, size(img, 3)])) = 0;  % Zero out pixels outside
the mask

% Display the masked image
figure;
imshow(maskedImg);
title('Masked Image');
```


```
% Convert the image to grayscale for filtering
grayImg = rgb2gray(maskedImg);

% Apply Gaussian low-pass filter
gaussianFiltered = imgaussfilt(grayImg, 2);  % Sigma of 2
figure;
imshow(gaussianFiltered);
title('Gaussian Low-Pass Filtered Image');
```

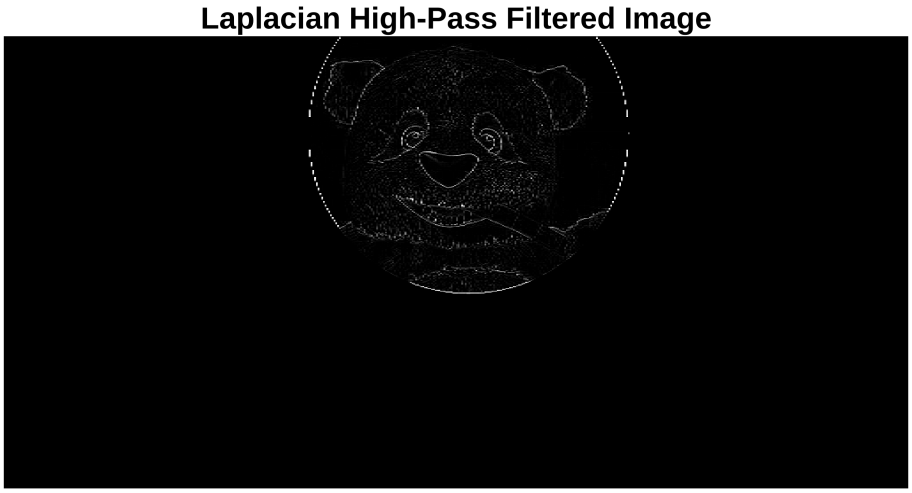
Gaussian Low-Pass Filtered Image



```
% Apply Average low-pass filter
h = fspecial('average', [5 5]);  % 5x5 averaging filter
averageFiltered = imfilter(grayImg, h);
figure;
imshow(averageFiltered);
title('Average Low-Pass Filtered Image');
```

Average Low-Pass Filtered Image

```
% Apply Laplacian high-pass filter
laplacianFiltered = imfilter(grayImg, fspecial('laplacian'));
figure;
imshow(laplacianFiltered, []);
title('Laplacian High-Pass Filtered Image');
```




```
% Apply Prewitt high-pass filter
prewittFiltered = imfilter(grayImg, fspecial('prewitt'));
figure;
imshow(prewittFiltered, []);
title('Prewitt High-Pass Filtered Image');
```

Prewitt High-Pass Filtered Image

2nd question

```
% Load and convert the image to grayscale
img = imread('ass pic.jpg');
if size(img, 3) == 3
    img = rgb2gray(img);  % Convert to grayscale if it's RGB
end

% Normalize the image to [0, 1]
img = double(img) / 255;

% Define a function for Floyd-Steinberg Dithering
function floyd_steinberg_dithered = floydSteinbergDithering(img)
    [rows, cols] = size(img);
    floyd_steinberg_dithered = img;  % Initialize the output image
    for y = 1:rows-1
        for x = 2:cols-1
            oldPixel = floyd_steinberg_dithered(y, x);
            newPixel = round(oldPixel);  % Quantize to 0 or 1
            floyd_steinberg_dithered(y, x) = newPixel;
            quant_error = oldPixel - newPixel;

            % Spread the quantization error
            floyd_steinberg_dithered(y, x+1) = floyd_steinberg_dithered(y,
x+1) + quant_error * 7/16;
```

```
            floyd_steinberg_dithered(y+1, x-1) =
floyd_steinberg_dithered(y+1, x-1) + quant_error * 3/16;
            floyd_steinberg_dithered(y+1, x) = floyd_steinberg_dithered(y+1,
x) + quant_error * 5/16;
            floyd_steinberg_dithered(y+1, x+1) =
floyd_steinberg_dithered(y+1, x+1) + quant_error * 1/16;
        end
    end
end

% Define a function for Jarvis-Judice-Ninke Dithering
function jarvis_judice_ninke_dithered = jarvisJudiceNinkeDithering(img)
    [rows, cols] = size(img);
    jarvis_judice_ninke_dithered = img;  % Initialize the output image

    % JJN kernel (1/48 is the weight factor)
    kernel = [0 0 0 7 5;
              3 5 7 5 3;
              1 3 5 3 1] / 48;

    for y = 1:rows-3
        for x = 3:cols-3
            oldPixel = jarvis_judice_ninke_dithered(y, x);
            newPixel = round(oldPixel);  % Quantize to 0 or 1
            jarvis_judice_ninke_dithered(y, x) = newPixel;
            quant_error = oldPixel - newPixel;

            % Spread the quantization error over the 12 neighbors
            jarvis_judice_ninke_dithered(y:y+2, x-2:x+2) = ...
                jarvis_judice_ninke_dithered(y:y+2, x-2:x+2) + quant_error *
kernel;
        end
    end
end

% Apply Floyd-Steinberg dithering
floyd_dithered_img = floydSteinbergDithering(img);

% Apply Jarvis-Judice-Ninke dithering
jarvis_dithered_img = jarvisJudiceNinkeDithering(img);
```

1

```
% Display the results
figure;
subplot(1, 3, 1);
imshow(img, []);
title('Original Image');
subplot(1, 3, 2);
imshow(floyd_dithered_img, []);
```

```
title('Floyd-Steinberg Dithered Image');
subplot(1, 3, 3);
imshow(jarvis_dithered_img, []);
title('Jarvis-Judice-Ninke Dithered Image');
```

3rd question

The **Kuwahara filter** is a **non-linear smoothing filter** that preserves edges while reducing noise. It divides the neighborhood of each pixel into four overlapping sub-regions and calculates the mean and variance of the pixel values within each region. The filter then selects the region with the smallest variance and replaces the central pixel with the mean of that region. This process smooths the image while preserving important features, such as edges.

```
function kuwahara_filtered_image = kuwahara_filter(img, window_size)

    % Convert the image to double for calculations
    img = double(img);

    % Get the dimensions of the image
    [rows, cols] = size(img);

    % Initialize the output image
    kuwahara_filtered_image = zeros(rows, cols);
```

```
    % Define half window size
    half_w = (window_size - 1) / 2;

    % Pad the image to avoid boundary issues
    padded_img = padarray(img, [half_w half_w], 'symmetric');

    % Apply the Kuwahara filter to each pixel
    for i = 1:rows
        for j = 1:cols
            % Extract four sub-regions around the pixel
            region_1 = padded_img(i:i+half_w, j:j+half_w);                % 
Top-left
            region_2 = padded_img(i:i+half_w, j+half_w:j+2*half_w);       % 
Top-right
            region_3 = padded_img(i+half_w:i+2*half_w, j:j+half_w);       % 
Bottom-left
            region_4 = padded_img(i+half_w:i+2*half_w, j+half_w:j+2*half_w); 
% Bottom-right

            % Calculate the mean and variance for each region
            means = [mean(region_1(:)), mean(region_2(:)), 
mean(region_3(:)), mean(region_4(:))];
            variances = [var(region_1(:)), var(region_2(:)), 
var(region_3(:)), var(region_4(:))];

            % Find the region with the smallest variance
            [~, min_var_idx] = min(variances);

            % Set the pixel value to the mean of the selected region
            kuwahara_filtered_image(i, j) = means(min_var_idx);
        end
    end

    % Convert the output image back to uint8 for display
    kuwahara_filtered_image = uint8(kuwahara_filtered_image);
end

% Load an image and convert to grayscale if necessary
img = imread('ass pic.jpg');
if size(img, 3) == 3
    img = rgb2gray(img);
end

% Apply Kuwahara filter to the image
window_size = 5;  % Set the window size (typically 3, 5, or 7)
kuwahara_filtered_img = kuwahara_filter(img, window_size);

% Display the original and filtered images
figure;
subplot(1, 2, 1);
```

```
imshow(img);
title('Original Image');
```

.

```
subplot(1, 2, 2);
imshow(kuwahara_filtered_img);
title('Kuwahara Filtered Image');
```

4th question

```
% Read the image (using built-in image or provide a path to your image)
img = imread('ass pic.jpg');  % You can also use your image by replacing this

% Convert the image to grayscale if it's not already
if size(img, 3) == 3
    img = rgb2gray(img);
end

img = double(img);  % Convert to double for processing

% Set the cutoff frequencies as a fraction of the maximum frequency
cutoffs = [0.02, 0.08, 0.16];

% Get image dimensions
[M, N] = size(img);
```

```
% Perform the 2D Fourier Transform
F = fft2(img);
F_shifted = fftshift(F);  % Shift zero-frequency component to the center

% Generate frequency grid
[u, v] = meshgrid(1:N, 1:M);
D0 = sqrt((u - N/2).^2 + (v - M/2).^2);  % Distance from the center

% Butterworth filter parameters
order = 3;  % Order of the filter

% Preallocate storage for filtered images
lowpass_filtered_butter = cell(1, length(cutoffs));
highpass_filtered_butter = cell(1, length(cutoffs));

lowpass_filtered_gaussian = cell(1, length(cutoffs));
highpass_filtered_gaussian = cell(1, length(cutoffs));

% Apply Butterworth filters (low-pass and high-pass) for each cutoff
for i = 1:length(cutoffs)
    cutoff = cutoffs(i) * max(D0(:));  % Calculate cutoff in terms of pixel
distance

    % Butterworth Low-pass filter
    H_low_butter = 1 ./ (1 + (D0 ./ cutoff).^(2 * order));
    F_low_filtered_butter = F_shifted .* H_low_butter;
    lowpass_filtered_butter{i} =
real(ifft2(ifftshift(F_low_filtered_butter)));

    % Butterworth High-pass filter
    H_high_butter = 1 - H_low_butter;
    F_high_filtered_butter = F_shifted .* H_high_butter;
    highpass_filtered_butter{i} =
real(ifft2(ifftshift(F_high_filtered_butter)));

    % Gaussian Low-pass filter
    H_low_gaussian = exp(-(D0.^2) / (2 * (cutoff^2)));
    F_low_filtered_gaussian = F_shifted .* H_low_gaussian;
    lowpass_filtered_gaussian{i} =
real(ifft2(ifftshift(F_low_filtered_gaussian)));

    % Gaussian High-pass filter
    H_high_gaussian = 1 - H_low_gaussian;
    F_high_filtered_gaussian = F_shifted .* H_high_gaussian;
    highpass_filtered_gaussian{i} =
real(ifft2(ifftshift(F_high_filtered_gaussian)));
end

% Display the original and filtered images
```

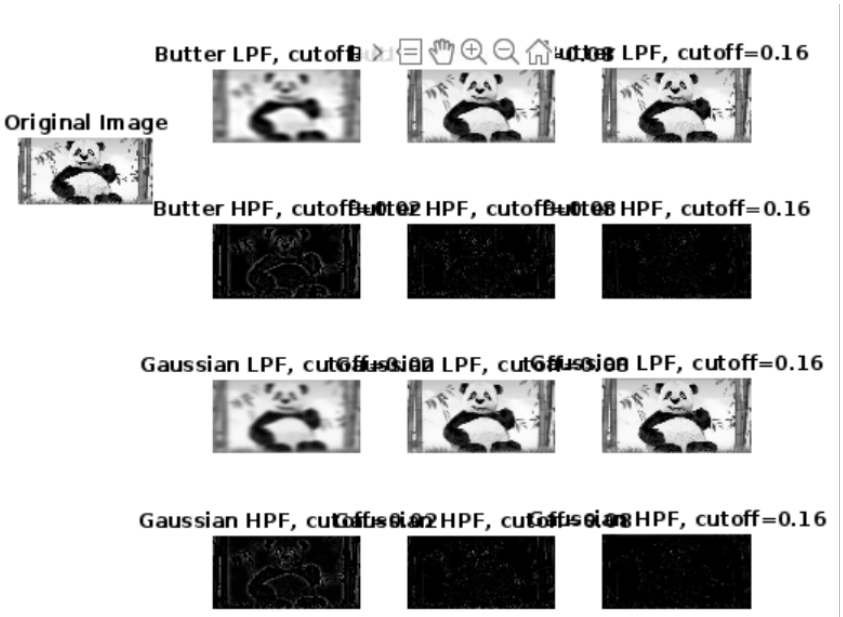
```
figure;
subplot(2, length(cutoffs) + 1, 1);
imshow(uint8(img));
title('Original Image');

for i = 1:length(cutoffs)
    % Butterworth Low-pass filtered image
    subplot(4, length(cutoffs) + 1, i + 1);
    imshow(uint8(lowpass_filtered_butter{i}));
    title(sprintf('Butter LPF, cutoff=%.2f', cutoffs(i)));

    % Butterworth High-pass filtered image
    subplot(4, length(cutoffs) + 1, i + 1 + length(cutoffs) + 1);
    imshow(uint8(highpass_filtered_butter{i}));
    title(sprintf('Butter HPF, cutoff=%.2f', cutoffs(i)));

    % Gaussian Low-pass filtered image
    subplot(4, length(cutoffs) + 1, i + 1 + 2 * (length(cutoffs) + 1));
    imshow(uint8(lowpass_filtered_gaussian{i}));
    title(sprintf('Gaussian LPF, cutoff=%.2f', cutoffs(i)));

    % Gaussian High-pass filtered image
    subplot(4, length(cutoffs) + 1, i + 1 + 3 * (length(cutoffs) + 1));
    imshow(uint8(highpass_filtered_gaussian{i}));
    title(sprintf('Gaussian HPF, cutoff=%.2f', cutoffs(i)));
end
```



5th question

```
% Read the image
img = imread('ass pic.jpg');
if size(img, 3) == 3
    img = rgb2gray(img);  % Convert to grayscale if the image is in color
end

%Normalize
img_normalized = double(img) / 255;
% We want 32 levels, so the size of the "shrunken" version should correspond
to 32 levels
quantized_img = imresize(img_normalized, [32, 32], 'nearest');
quantized_img_resized = imresize(quantized_img, size(img), 'nearest');

%  256 grayscale levels
quantized_img_resized = uint8(quantized_img_resized * 255);

%  Display images
figure;
subplot(1, 2, 1);
imshow(img);
```

```
title('Original Image');

subplot(1, 2, 1);
imshow(quantized_img_resized);
title('Quantized Image (32 Levels)');
```

Step 1: Read the image and convert to grayscale if needed

Step 2: Normalize the image (convert to double precision)

Step 3: Quantize the image using imresize

Step 4: Resize back to the original size

Step 5: Map back to 256 grayscale levels

Step 6: Display the original and quantized images