Problem 1

Spatial Filtering and Binarisation: Apply Gaussian blurring on the image `moon.noisy.png`. Generate a spatial Gaussian filter of size 41 × 41 by generating a filter kernel as:

[G_f(x, y) = \frac{1}{K} \exp\left(\frac{-x^2 + y^2}{2\sigma_g^2}\right),]

where (x, y \in {-20, -19, \cdots, 19, 20}) and (\sigma_g) is the standard deviation of the Gaussian kernel. (K) normalizes the filter such that (\sum_x \sum_y G_f(x, y) = 1). You can use a library function to convolve the input image with the kernel you created to obtain the Gaussian blurred image.

Apply Otsu's Binarization algorithm on the blurred image and note the optimal within-class variance (\sigma_w^{2*}) for a given (\sigma_g) blur parameter.

Plot the histogram and the binarized image for the blurred images and find their corresponding optimal within-class variances (\sigma_w^2) for each (\sigma_g \in {0, 0.1, 0.5, 1, 2.5, 5, 10, 20}) ((0) corresponds to the input image itself). Find the optimal (\sigma_g) that minimizes (\sigma_w^{2*}). Comment on your observations.

In [29]:
```
import numpy as np
import matplotlib.image as mpimg
from scipy.ndimage import convolve
import matplotlib.pyplot as plt
from scipy.ndimage import convolve
import matplotlib.pyplot as plt
from scipy import misc
from matplotlib.image import imread
from math import pi
```

In [30]:
```
def gaussian_kernel(size, sigma):
    """Creates a 2D Gaussian kernel."""
    kernel = np.zeros((size, size))
    center = size // 2

    if sigma == 0:
        kernel[center, center] = 1
        return kernel

    for x in range(size):
        for y in range(size):
            x_dist = (x - center) ** 2
            y_dist = (y - center) ** 2
            kernel[x, y] = np.exp(-(x_dist + y_dist) / (2 * sigma ** 2))

    # Normalize the kernel so that the sum equals 1
    kernel /= np.sum(kernel)
```

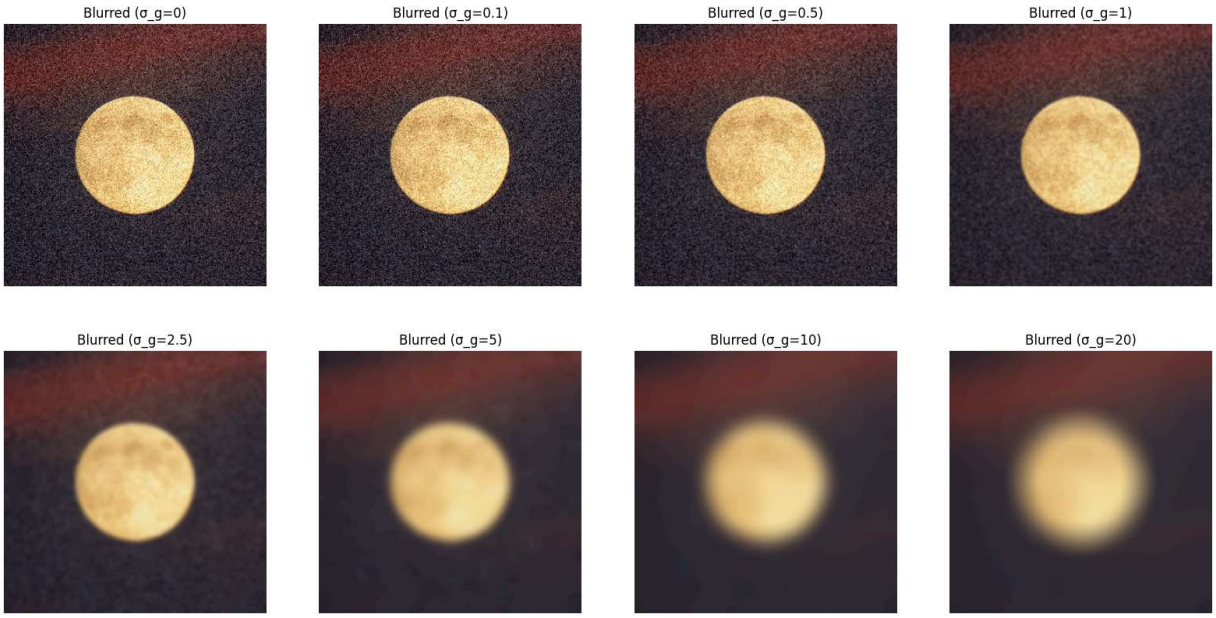
```
    return kernel


def Histogram(img):

    if np.max(img) <= 1:
        img = (255 * img).astype(int)
    # Count occurrences for each intensity level
    img_freq = np.zeros(256, dtype=int)
    for l in range(256):
       img_freq[l] = np.count_nonzero(img == l)

    return img_freq


def Within_class_variance(img):

    # Normalize the image to 0-255 range
    if np.max(img) <= 1:
        img = (255 * img).astype(int)

    var_w_lst = np.zeros(256, dtype = int)
    img_freq = Histogram(img)
    #print(coins_array)
    w_t = np.sum(img_freq)


    for t in range(256):
        mean_0 = 0
        mean_1 = 0
        var_0 = 0
        var_1 = 0
        w_0 = 0
        w_1 = 0
        for i in range(t + 1):
            w_0 += img_freq[i]

        for j in range(t + 1, 256):
            w_1 += img_freq[j]

        if  w_0 > 0:
            for i in range(t + 1):
                mean_0 += i * img_freq[i] / w_0
        if t < 255 and w_1 > 0:
            for j in range(t + 1, 256):
                mean_1 += j * img_freq[j] / w_1


        if  w_0 > 0:
            for i in range(t + 1):
                var_0 += ((i-mean_0)**2) * img_freq[i] / w_0
        if t < 255 and w_1 > 0:
            for j in range(t + 1, 256):
                var_1 += ((j-mean_1)**2) * img_freq[j] / w_1
        var_w = int((w_0*var_0 + w_1*var_1)/w_t)
```

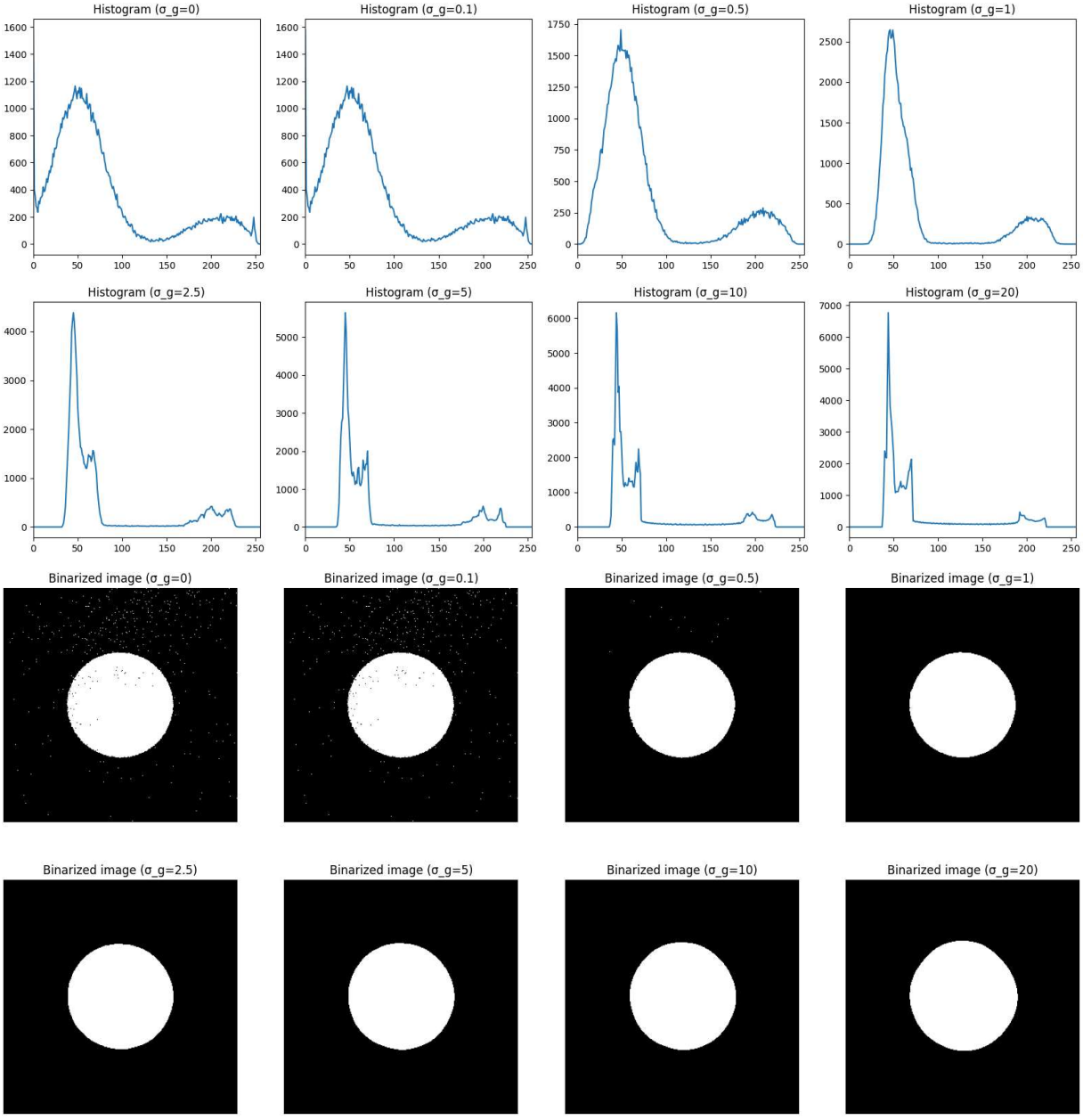
```
        var_w_lst[t] = var_w

    return var_w_lst
```

In [31]:
```
# Load the original image (RGB)
img = mpimg.imread("moon_noisy.png")
print("Image Shape:", img.shape)

# Split image into red, green, and blue channels
red_channel = img[:, :, 0]
green_channel = img[:, :, 1]
blue_channel = img[:, :, 2]

# Create subplots
fig, axs1 = plt.subplots(2, 4, figsize=(20, 10))
fig, axs2 = plt.subplots(2, 4, figsize=(20, 10))
fig, axs3 = plt.subplots(2, 4, figsize=(20, 10))

# Apply Gaussian filters with different variances
n = 0
for variance in [0, 0.1, 0.5, 1, 2.5, 5, 10, 20]:
    kernel_size = 41
    sigma_g = variance
    gaussian_filter = gaussian_kernel(kernel_size, sigma_g)

    # Convolve the filter with each channel
    red_blurred_channel = convolve(red_channel, gaussian_filter)
    green_blurred_channel = convolve(green_channel, gaussian_filter)
    blue_blurred_channel = convolve(blue_channel, gaussian_filter)

    # Stack the channels back into an RGB image
    blurred_img = np.dstack((red_blurred_channel, green_blurred_channel, blue_blurr
    # Convert the blurred image to grayscale
    blurred_img_gray = np.dot(blurred_img, [0.2989, 0.5870, 0.1140])

    #binarize the blurred image
    within_variance = Within_class_variance(blurred_img_gray)
    optimal_within_variance = np.min(within_variance)
    threshold = np.argmin(within_variance)
    print("Threshold: {} \noptimal within-class variances for σ({})=:{}".format(thr
    binarized_img = np.zeros_like(blurred_img_gray)
    for i in range(binarized_img.shape[0]):
        for j in range(binarized_img.shape[1]):
            if blurred_img_gray[i, j] > threshold/255:
                binarized_img[i, j] = 255
            else:
                binarized_img[i, j] = 0

    # Plot the blurred image in axs1
    row = n // 4
    col = n % 4

    axs3[row, col].imshow(binarized_img, cmap='gray')
    axs3[row, col].set_title(f"Binarized image (σ_g={variance})")
    axs3[row, col].axis('off')
    axs1[row, col].imshow(blurred_img)
```

```
    axs1[row, col].set_title(f"Blurred (σ_g={variance})")
    axs1[row, col].axis('off')

    # Calculate the histogram for the grayscale blurred image
    hist_values = Histogram(blurred_img_gray)

    # Plot the histogram in axs2
    axs2[row, col].plot(np.arange(0, 256), hist_values)
    axs2[row, col].set_title(f"Histogram (σ_g={variance})")
    axs2[row, col].set_xlim([0, 255])

    n += 1

# Show the plots
plt.show()
```

```
Image Shape: (300, 300, 3)
Threshold: 124
optimal within-class variances for σ(0)=:705
Threshold: 124
optimal within-class variances for σ(0.1)=:705
Threshold: 120
optimal within-class variances for σ(0.5)=:369
Threshold: 123
optimal within-class variances for σ(1)=:181
Threshold: 125
optimal within-class variances for σ(2.5)=:170
Threshold: 120
optimal within-class variances for σ(5)=:212
Threshold: 120
optimal within-class variances for σ(10)=:278
Threshold: 116
optimal within-class variances for σ(20)=:320
```





Coclusion

Observations:

As the standard deviation of the Gaussian kernel, (\sigma_g), increases, the blurring effect becomes more due to the influence of neighboring pixels of the center of the kernel. Initially, with lower values of (\sigma_g) (e.g., (\sigma_g = 0) and (\sigma_g = 0.1)), noise in the image is quite visible after binarization. The within-class variance for these cases remains high, around 705.

As (\sigma_g) increases to 0.5 and 1, the Gaussian kernel smooths out more noise, and we observe a sharp reduction in the within-class variance, dropping to 369 for (\sigma_g = 0.5) and 181 for (\sigma_g = 1). The binarized images for these values exhibit far fewer noise

artifacts. This suggests that moderate Gaussian blurring (with \sigma_g) values between 0.5 and 2.5) achieves the optimal balance between noise reduction and detail retention.

For (\sigma_g = 2.5), the within-class variance stabilizes at a low value of 170, further indicating effective noise suppression while maintaining important features. However, as (\sigma_g) continues to increase—reaching 5, 10, and 20—the within-class variance starts to rise again, reaching 320 at (\sigma_g = 20). This is due to the excessive smoothing effect of the larger Gaussian kernel, which results in a loss of finer details, causing some blending of edges and reduced contrast between the background and the foreground object. The blurring at (\sigma_g = 20) begins to act more like an averaging filter, leading to a flattening effect in the binarized images, as indicated by the increased within-class variance.

In summary, while increasing (\sigma_g) reduces noise and improves the quality of the binarized image, there is an optimal range around (\sigma_g = 2.5), where noise is minimized without sacrificing too much detail. Beyond this value, further smoothing results in diminishing returns, with increasing within-class variance and loss of important image details.

In [ ]:

Problem 2

2. Fractional Scaling with Interpolation: (a) Downsample the image 'flowers.png' by 2 and upsample the result by 3 using bilinear interpolation. (b) Upsample the image by 3/2 = 1.5 using bilinear interpolation. Observe what is different in both results and give comments. Hint: Use a zoomed-in patch of the image for a better visual comparison.

In [19]:
```
def downsampling(img, factor):
    size = img.shape

    sampled_img = np.zeros((size[0]//factor, size[1]//factor), dtype=img.dtype)
    for i in range(size[0]//factor):
        for j in range(size[1]//factor):
            sampled_img[i, j] = img[i*factor, j*factor]
    return sampled_img




# write a function to upsample the image by a factor of 3

def upsampling(img, factor):
    size1 = img.shape  # original image size (113, 200)
    upsampled_img = np.zeros((int(size1[0] * factor), int(size1[1] * factor)), dtyp
    size = upsampled_img.shape  # upsampled image size

    for i in range(size[0]):  # Iterate over the rows of upsampled image
```

```
        for j in range(size[1]):  # Iterate over the columns of upsampled image
            a_i = i / factor  # Corresponding row in the original image
            a_j = j / factor  # Corresponding column in the original image

            if a_i == int(a_i) and a_j == int(a_j):
                # Direct copy from the original image
                upsampled_img[i, j] = img[int(a_i), int(a_j)]
            elif a_i != int(a_i) and a_j != int(a_j):
                # Bilinear interpolation
                if int(a_i) + 1 < size1[0] and int(a_j) + 1 < size1[1]:
                    interpolation_x1 = img[int(a_i), int(a_j)+1] + (img[int(a_i)+1,
                    interpolation_x2 = img[int(a_i), int(a_j)] + (img[int(a_i)+1, i
                    interpolation_y1 = interpolation_x2 + (interpolation_x1 - inter
                    upsampled_img[i, j] = interpolation_y1
            elif a_i == int(a_i) and a_j != int(a_j):
                # Horizontal interpolation
                if int(a_j) + 1 < size1[1]:
                    interpolation_y1 = img[int(a_i), int(a_j)] + (img[int(a_i), int
                    upsampled_img[i, j] = interpolation_y1
            elif a_i != int(a_i) and a_j == int(a_j):
                # Vertical interpolation
                if int(a_i) + 1 < size1[0]:
                    interpolation_x1 = img[int(a_i), int(a_j)] + (img[int(a_i)+1, i
                    upsampled_img[i, j] = interpolation_x1

    return upsampled_img


# Load and display the original image
img_1 = mpimg.imread("flowers.png")

downsampled_img = downsampling(img_1, 2)

# Create subplots with 1 row and 2 columns
fig, ax = plt.subplots(1, 2, figsize=(10, 5))

# Display the original image
ax[0].imshow(img_1, cmap='gray')
ax[0].set_title('Original Image')
ax[0].axis('off')  # Hide the axis
print("Original image shape:", img_1.shape)

# Display the downsampled image
downsampled_img = downsampling(img_1, 2)
ax[1].imshow(downsampled_img, cmap='gray')
ax[1].set_title('Downsampled Image')
ax[1].axis('off')  # Hide the axis
print("Downsampled image shape:", downsampled_img.shape)

# Display the images
plt.show()
```

```
Original image shape: (227, 400)
Downsampled image shape: (113, 200)
```

In [20]:
```
# Question 2(a)
down_up_sampled_img = upsampling(downsampled_img, 3)
print("Downsampled followed by upsampled image shape:", down_up_sampled_img.shape)

# Save the downsampled followed by upsampled image
mpimg.imsave("down_up_sampled.png", down_up_sampled_img, cmap='gray')

# Question 2(b)
up_sampled_img = upsampling(img_1, 1.5)
print("Upsampled image shape:", up_sampled_img.shape)

# Save the upsampled image
mpimg.imsave("up_sampled.png", up_sampled_img, cmap='gray')


# Create subplots to plot both the upsampled images
fig, ax = plt.subplots(1, 2, figsize=(10, 5))

# Display the downsampled followed by upsampled image
ax[0].imshow(down_up_sampled_img, cmap='gray')
ax[0].set_title('Downsampled & Upsampled Image')
ax[0].axis('off')  # Hide the axis

# Display the upsampled image
ax[1].imshow(up_sampled_img, cmap='gray')
ax[1].set_title('Upsampled Image')
ax[1].axis('off')  # Hide the axis

# Show the plot
plt.show()
```

```
Downsampled followed by upsampled image shape: (339, 600)
Upsampled image shape: (340, 600)
```

Conclusion

Image 1 is created by first downsampling the original image by a factor of 2, then upsampling it by a factor of 3 using bilinear interpolation. Image 2, on the other hand, is generated by directly upsampling the original image by a factor of 1.5 using bilinear interpolation. Image 2 appears better than Image 1 because, during the downsampling process in Image 1, only half of the original data points are retained, leading to a loss of detail. In contrast, during the upsampling of Image 2, the number of data points remains closer to the original, preserving more image information and resulting in better quality.

Problem 3

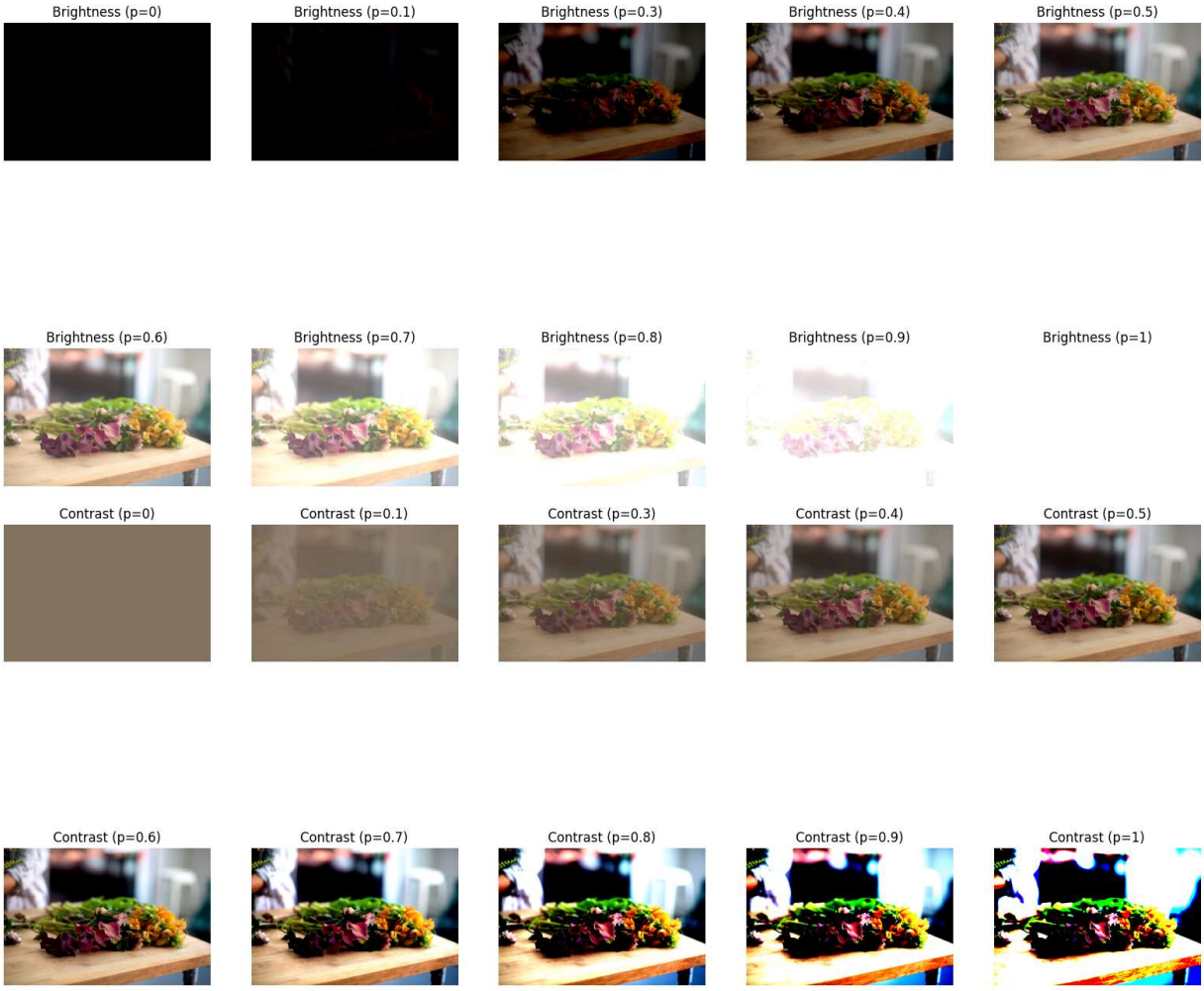
Photoshop Feature: Watch the video explaining the Brightness/Contrast feature in Adobe Photoshop ('photoshop feature.mp4'). Implement this based on your knowledge of pointwise operations applied to images. Implement two functions brightnessAdjust(img,p) and contrastAdjust(img,p), that output the respective adjusted images where p controls the respective adjustments. In particular, for brightnessAdjust, p=0 should output a black image, p=1, a white image, and p=0.5 should output the input image itself without any adjustment. Output for p values between 0.5 to 0 should gradually change from the input to a black image. Similarly, for p values between 0.5 to 1, the output should gradually from the input to a white image. For contrastAdjust, p=0.5 should not alter the input image, p=0 should output a grey image, and p=1 should output a black and white image (would look like a binarized image). For all other intermediate values of p, the behaviour should gradually change from the behaviour at p=0.5 to either p=0 or p=1 depending on the value of p.

In [27]:
```
# Function to display images using matplotlib
def display_images(original, adjusted, title1="Original", title2="Adjusted"):
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))

    axes[0].imshow(original)
    axes[0].set_title(title1)
    axes[0].axis('off')

    axes[1].imshow(adjusted)
    axes[1].set_title(title2)
    axes[1].axis('off')

    plt.show()


# Adjust brightness
def brightnessAdjust(fname, p):
    # Read the image
    img = imread(fname)
```

```
    # p: brightness factor (0 <= p <= 1)
    if p < 0 or p > 1:
        raise ValueError("p must be between 0 and 1")

    # Adjust brightness
    else:
        return np.clip(2*(p-0.5)*255 +img, 0, 255).astype(np.uint8)



def contrastAdjust(img, p):
    # Ensure p is between 0 and 1
    if p < 0 or p > 1:
        raise ValueError("p must be between 0 and 1")

    # Calculate the mean for each channel (R, G, B) separately
    mean = np.mean(img, axis=(0, 1), keepdims=True)
    theta = pi/4 + pi/2*(p-0.5)
    p_slope = np.tan(theta)
    y = p_slope * (img - mean) + mean
    adjusted_img = np.clip(y, 0, 255).astype(np.uint8)

    return adjusted_img
```

In [28]:

```
# Load an image file
original_image = imread('brightness_contrast.jpg')

# Create two subplots for brightness and contrast adjustments
fig1, axs1 = plt.subplots(2, 5, figsize=(20, 10))  # For brightness adjustments
fig2, axs2 = plt.subplots(2, 5, figsize=(20, 10))  # For contrast adjustments

n = 0

for p in [0, 0.1, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]:
    # Adjust brightness and contrast
    brightness_img = brightnessAdjust('brightness_contrast.jpg', p)
    contrast_img = contrastAdjust(original_image, p)  # Pass the original image arr

    row = n // 5  # Adjust row and col calculation based on 5 columns
    col = n % 5

    # Display brightness-adjusted images
    axs1[row, col].imshow(brightness_img)
    axs1[row, col].set_title(f"Brightness (p={p})")
    axs1[row, col].axis('off')

    # Display contrast-adjusted images
    axs2[row, col].imshow(contrast_img)
    axs2[row, col].set_title(f"Contrast (p={p})")
    axs2[row, col].axis('off')

    n += 1  # Increment counter

# Show the plots
plt.show()
```



Conclusion

Case 1 (p > 0.5): As p increases, during the brightness operation, the brightness of each pixel increases linearly with its original intensity. When p = 1, the brightness of every pixel reaches the maximum value of 255. In terms of contrast, brighter pixels become even brighter, while darker pixels get darker. When p = 1 for the contrast operation, all pixel intensities greater than the mean are set to 255, while those below the mean are set to 0, creating a binary, high-contrast image.

Case 2 (0 ≤ p < 0.5): As p decreases, during the brightness operation, the brightness of each pixel decreases linearly with its original intensity. When p = 0, the brightness of every pixel becomes 0 (completely dark). In terms of contrast, the intensity of brighter pixels decreases, while the intensity of darker pixels increases. When p = 0 for the contrast operation, all pixel intensities converge to the mean value, resulting in a flat, uniform image with no contrast.

In [ ]: