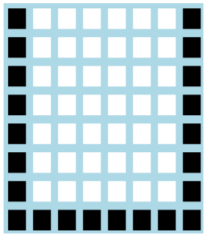
Problem 1

A. A binary image.

Binary images
Enter 0 for black or 1 for white for each pixel.

	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
i=0	0	1	1	1	1	1	1	0
i=1	0	1	1	1	1	1	1	0
i=2	0	1	1	1	1	1	1	0
i=3	0	1	1	1	1	1	1	0
i=4	0	1	1	1	1	1	1	0
i=5	0	1	1	1	1	1	1	0
i=6	0	1	1	1	1	1	1	0
i=7	0	0	0	0	0	0	0	0

Check values and explain any errors



Update Image

Code:

```
# Definition of the image using a 2D Python array
img = [[0, 1, 1, 1, 1, 1, 1, 0], # i=0
       [0, 1, 1, 1, 1, 1, 1, 0], # i=1
       [0, 1, 1, 1, 1, 1, 1, 0], # i=2
       [0, 1, 1, 1, 1, 1, 1, 0], # i=3
       [0, 1, 1, 1, 1, 1, 1, 0], # i=4
       [0, 1, 1, 1, 1, 1, 1, 0], # i=5
       [0, 1, 1, 1, 1, 1, 1, 0], # i=6
       [0, 0, 0, 0, 0, 0, 0, 0]  # i=7
      ]
```

Here, the array is an 8×8 binary image.

0 → black

1 → white

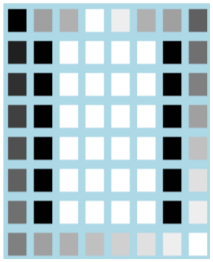
B. A greyscale image.

Grayscale images

Enter #00 for black, #FF for white, and other values for gray (e.g., #0F).

	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
i=0	#00	#A0	#B0	#FF	#EE	#B0	#A0	#60
i=1	#20	#00	#FF	#FF	#FF	#FF	#00	#70
i=2	#30	#00	#FF	#FF	#FF	#FF	#00	#80
i=3	#40	#00	#FF	#FF	#FF	#FF	#00	#A0
i=4	#50	#00	#FF	#FF	#FF	#FF	#00	#C0
i=5	#60	#00	#FF	#FF	#FF	#FF	#00	#E0
i=6	#70	#00	#FF	#FF	#FF	#FF	#00	#EE
i=7	#80	#A0	#B0	#C0	#D0	#E0	#EE	#FF

Check values and explain any errors



Update Image

Code:

```
# Definition of the image using a 2D Python array
img = [[0x00, 0xA0, 0xB0, 0xFF, 0xEE, 0xB0, 0xA0, 0x60], # i=0
       [0x20, 0x00, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x70], # i=1
       [0x30, 0x00, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x80], # i=2
       [0x40, 0x00, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0xA0], # i=3
       [0x50, 0x00, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0xC0], # i=4
       [0x60, 0x00, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0xE0], # i=5
       [0x70, 0x00, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0xEE], # i=6
       [0x80, 0xA0, 0xB0, 0xC0, 0xD0, 0xE0, 0xEE, 0xFF]  # i=7
      ]
```

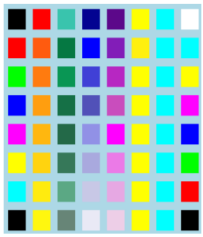
This grayscale image uses 8-bit pixel intensity values written in hexadecimal form, where 0x00 represents black, 0xFF represents white, and intermediate values such as 0x80 represent mid-gray levels.

C. A color image.

Color images
Enter #000000 for black, and #FFFFFF for white.
Enter #FF0000 for red, #00FF00 for green, and #0000FF for blue.

	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
i=0	#000000	#FF0000	#39C4AD	#030391	#5C088A	#FFFF00	#00FFFF	#FFFFFF
i=1	#FF0000	#FF5B0F	#057842	#0000FF	#821DB8	#FFF00F	#00FFFF	#00FFFF
i=2	#00FF00	#FF7B0F	#099654	#4040D6	#B727C2	#FFFF00	#00FFFF	#FFFF00
i=3	#0000FF	#FF9E0F	#157046	#5151B8	#C94DBD	#FFFF00	#00FFFF	#FF00FF
i=4	#FF00FF	#FFB70F	#236948	#9191E6	#FF00FF	#FFFF00	#00FFFF	#0000FF
i=5	#FFFF00	#FFD30F	#367859	#A9A9DE	#EB7AE7	#FFFF00	#00FFFF	#00FF00
i=6	#00FFFF	#FFEB0F	#5BA884	#C8C8E6	#E6A8E3	#FFFF00	#00FFFF	#FF0000
i=7	#000000	#FFF70F	#678577	#E9E9F5	#EDCEE7	#FFFF00	#00FFFF	#000000

Check values and explain any errors



Update Image

Code:

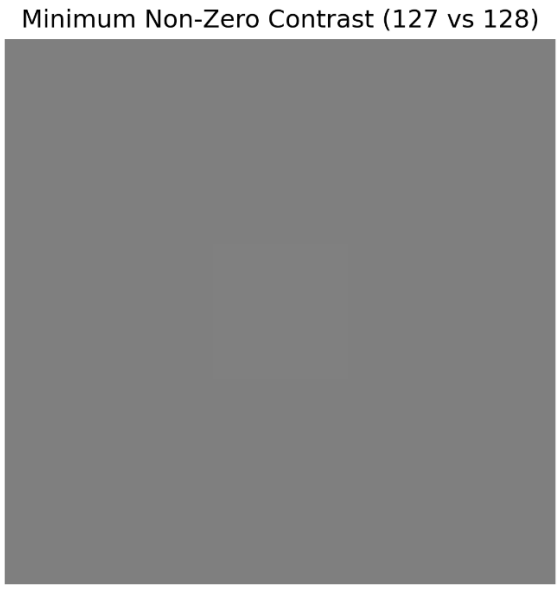
```
# Definition of the image using a 2D Python array

img = [[0x000000, 0xFF0000, 0x39C4AD, 0x030391, 0x5C088A, 0xFFFF00, 0x00FFFF, 0xFFFFFF], # i=0

       [0xFF0000, 0xFF5B0F, 0x057842, 0x0000FF, 0x821DB8, 0xFFF00F, 0x00FFFF, 0x00FFFF], # i=1

       [0x00FF00, 0xFF7B0F, 0x099654, 0x4040D6, 0xB727C2, 0xFFFF00, 0x00FFFF, 0xFFFF00], # i=2

       [0x0000FF, 0xFF9E0F, 0x157046, 0x5151B8, 0xC94DBD, 0xFFFF00, 0x00FFFF, 0xFF00FF], # i=3

       [0xFF00FF, 0xFFB70F, 0x236948, 0x9191E6, 0xFF00FF, 0xFFFF00, 0x00FFFF, 0x0000FF], # i=4

       [0xFFFF00, 0xFFD30F, 0x367859, 0xA9A9DE, 0xEB7AE7, 0xFFFF00, 0x00FFFF, 0x00FF00], # i=5

       [0x00FFFF, 0xFFEB0F, 0x5BA884, 0xC8C8E6, 0xE6A8E3, 0xFFFF00, 0x00FFFF, 0xFF0000], # i=6

       [0x000000, 0xFFF70F, 0x678577, 0xE9E9F5, 0xEDCEE7, 0xFFFF00, 0x00FFFF, 0x000000]  # i=7

    ]
```

In the color image, each pixel is represented by a 24-bit hexadecimal RGB code of the form, where the first two digits control the red intensity, the next two control the green intensity, and the last two control the blue intensity. For example, 0xFF0000 corresponds to pure red (255,0,0), 0x00FFFF corresponds to cyan (0,255,255), and 0xFFFFFF corresponds to white (255,255,255). Intermediate values such as 0x39C4AD produce mixed colors like teal.

Problem 2

In problem 2, all images were generated synthetically as 2D **8-bit grayscale** arrays using NumPy. In an 8-bit grayscale image, pixel intensities are discrete integers in the range. $[0, 255]$. The images were displayed using a fixed grayscale mapping ($vmin = 0$, $vmax = 255$) to ensure consistent brightness and contrast interpretation across figures. Each figure was also saved to disk using plt.savefig().

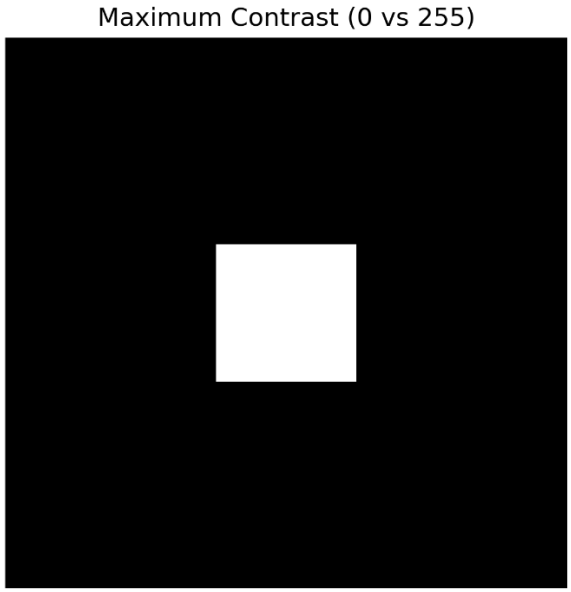
A. Creating an example with a minimum (non-zero) contrast for 8-bit grayscale images.



.

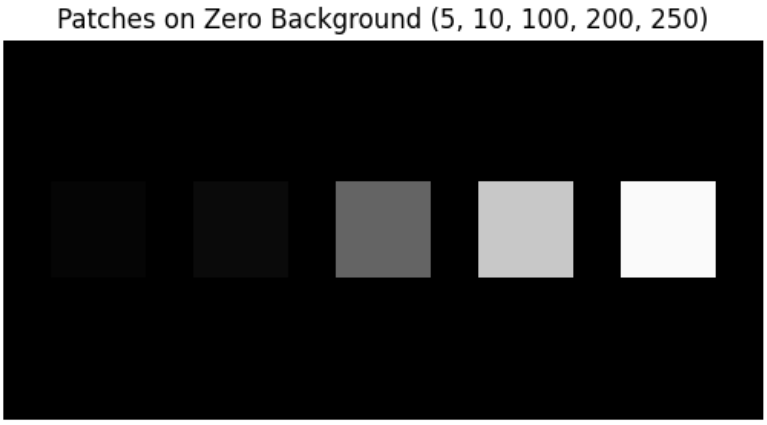
In an 8-bit grayscale image, the intensity values are integers from 0 to 255. The smallest possible non-zero contrast is achieved by a difference of 1 unit. In this image, the background intensity is set at 127, while a square patch of intensity 128 is placed in the center. Due to the minimal difference, the patch is barely visible. This illustrates the threshold for minimum non-zero contrast in 8-bit imaging.

B. Creating an 8-bit grayscale image example with maximum contrast



The maximum contrast in an 8-bit grayscale image is achieved with the lowest and highest possible intensity values: 0 (pure black) and 255 (pure white). This provides the highest possible visual distinction between the two regions.

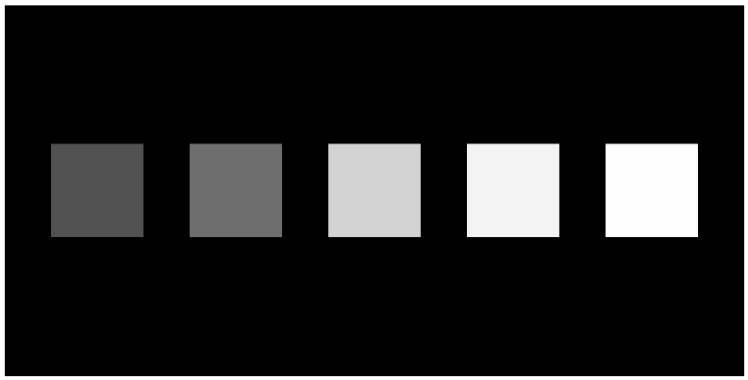
C. Creating an image with zero background and constant patches of 5, 10, 100, 200, and 250.



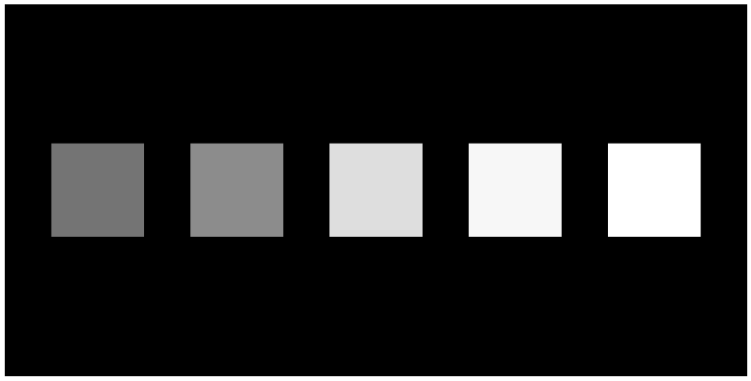
Here, we created an image with a black background (0) and five patches with intensities 5, 10, 100, 200, and 250. This image states how visibility depends on the intensity difference relative to the background. The patches with values of 5 and 10 are challenging to discern against the black background, as the human eye is less sensitive to small absolute differences in low-light conditions. The patch with a value of 100 is moderately bright and easily visible, while the patches with values of 200 and 250 are very bright and stand out significantly.

D. Applying the logarithmic transformation $\log(1 + C \cdot I)$ for different values of C and discussing the visibility of the different patches.

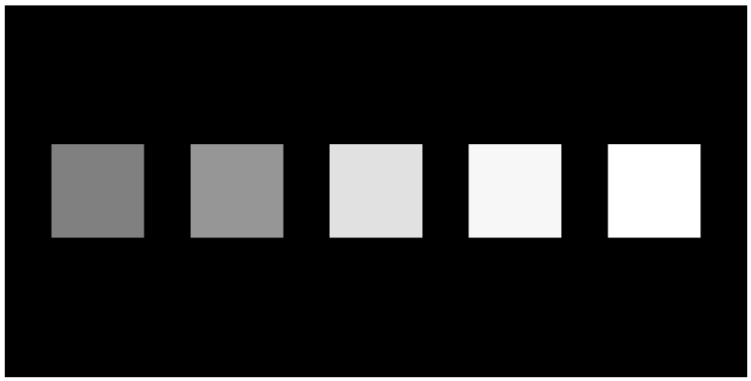
Log Transform: s = log(1 + C·I), C = 1

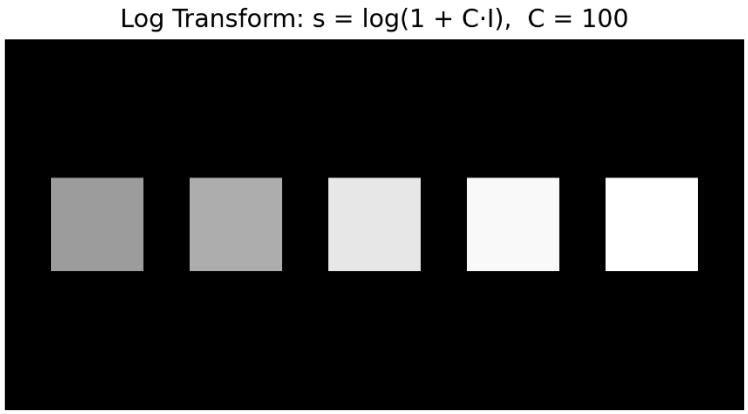


Log Transform: s = log(1 + C·I), C = 5



Log Transform: s = log(1 + C·I), C = 10





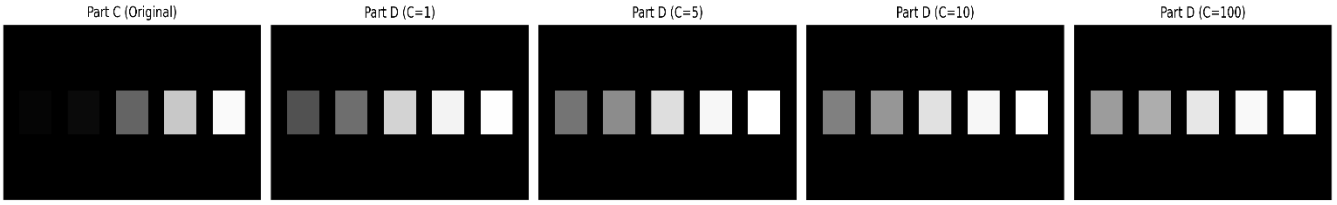
The log transformation is highly effective for images with a high dynamic range, where important details are hidden in the shadows. By increasing C, we effectively stretch the dark regions of the histogram. This makes the patches of 5 and 10 much more visible relative to the background. As C increases (e.g., up to $C = 100$), enhancement of shadow details becomes stronger, but bright values become increasingly similar due to compression. However, high-intensity patches, such as 200 and 250, become compressed, reducing their contrast difference. Thus, the log transform improves the visibility of dark regions while sacrificing contrast in bright regions.

E. Place the images from parts C and D side-by-side for $\boldsymbol{C = 1, 5, 10, 100}$. Create a visibility table with $\boldsymbol{C}$ values along the rows and the patch values along the columns (5, 10, 100, 200, 25). In the table, provide a ranked score from 0 to 5 as follows:

- **0 means it is non-visible.**
- **1 means it has the worst visibility among all values of $\boldsymbol{C}$.**
- **2 means it has the second-worst visibility among all values of $\boldsymbol{C}$. Similarly for 3 and 4.**
- **5 means it has the best visibility among all values of $\boldsymbol{C}$. Provide a very brief justification of how you chose your scores. You do not need to talk about all of them! Note: Against the zero-background, each patch will have unit local contrast (verify). However, when assessing them, your perception of them will vary based on their relative positions and sizes. Thus, not everyone will have the same values.**

The following figure presents the original patch image from Part C alongside the log-transformed results from Part D for $C = 1,5,10,$and 100. Each patch has a constant intensity value (5, 10, 100, 200, and 250) placed on a zero-valued black background. Using the Michelson contrast definition,

$$C_M = \frac{L_{\max} - L_{\min}}{L_{\max} + L_{\min}},$$



The minimum luminance in each patch neighborhood is always. $L_{\min} = 0$because the background is black, while the maximum luminance is the patch value $L_{\max} > 0$. Therefore, for every patch,

$$C_M = \frac{L_{\max} - 0}{L_{\max} + 0} = 1$$

This confirms that all patches have unit local Michelson contrast regardless of their intensity or the value of C. The contrast table verifies that the Michelson contrast remains equal to 1 for the original image as well as all log-transformed versions.

```
================================================================
                  MICHELSON CONTRAST ANALYSIS
================================================================
Row \ Patch  |    5    |   10    |   100   |   200   |   250
----------------------------------------------------------------
Original     |  1.000  |  1.000  |  1.000  |  1.000  |  1.000  |
C = 1        |  1.000  |  1.000  |  1.000  |  1.000  |  1.000  |
C = 5        |  1.000  |  1.000  |  1.000  |  1.000  |  1.000  |
C = 10       |  1.000  |  1.000  |  1.000  |  1.000  |  1.000  |
C = 100      |  1.000  |  1.000  |  1.000  |  1.000  |  1.000  |
----------------------------------------------------------------

================================================================
            NORMALIZED LOG-TRANSFORMED PIXEL VALUES
================================================================
Row \ Patch  |    5    |   10    |   100   |   200   |   250
----------------------------------------------------------------
C = 1        |  82.0   |  110.0  |  212.0  |  244.0  |  254.0  |
C = 5        |  116.0  |  140.0  |  222.0  |  247.0  |  255.0  |
C = 10       |  128.0  |  150.0  |  225.0  |  247.0  |  255.0  |
C = 100      |  156.0  |  173.0  |  231.0  |  249.0  |  255.0  |
----------------------------------------------------------------
```

However, although the mathematical contrast is identical, the perceived visibility of the patches changes significantly after applying the logarithmic transformation. The normalized log-transformed values show how the logarithmic mapping changes the displayed brightness of each patch for different values of C. As C increases, lower intensity patches such as 5 and 10 are amplified more strongly relative to their original values, so their normalized display levels rise significantly. So, the visibility score also increases from 0 to 4 for patches 5, 10, and 100, as shown in Table 1. Higher intensity patches, such as 200 and 250, increase more slowly because the log function compresses large inputs. Thus, though for larger C, the lowest-intensity patches become

clearer, and the brighter patches appear increasingly similar. That is why the visibility score remains almost the same for patches 200 and 250 with the larger C. Thus, the log transformation improves shadow detail at the expense of contrast separation among high-intensity values.

Table 1: Visibility Score

C/Patch	5	10	100	200	250
Original	0	0	2	4	5
Part D (C=1)	2	2	3	4	5
Part D (C=5)	3	3	3	4	5
Part D (C=10)	3	3	4	4	5
Part D (C=100)	4	4	4	4	4

Code:

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


# for displaying and saving
def im_show(img, title="", filename=None):
    plt.figure()
    plt.imshow(img, cmap="gray", vmin=0, vmax=255)
    plt.title(title)
    plt.axis("off")

    # Save image
    if filename is not None:
        plt.savefig(filename)
        print(f"Saved: {filename}")

    plt.show()

# Part A: Minimum Non-Zero Contrast
rows, cols = 64, 64
```

```
imgA = np.full((rows, cols), 127)
imgA[24:40, 24:40] = 128

im_show(
    imgA,
    "Minimum Non-Zero Contrast (127 vs 128)",
    "MinContrast.png"
)

# Part B: Maximum Contrast
imgB = np.full((rows, cols), 0)
imgB[24:40, 24:40] = 255

im_show(
    imgB,
    "Maximum Contrast (0 vs 255)",
    "MaxContrast.png"
)

# Part C: Constant Patches on Zero Background
imgC = np.zeros((80, 160))

vals = [5, 10, 100, 200, 250]
patch = 20
gap = 10
top = 30

# Creating multiple patches of the same size in a loop
for k, v in enumerate(vals):
    left = gap + k * (patch + gap)
```

```
    imgC[top:top + patch, left:left + patch] = v

im_show(
    imgC,
    "Patches on Zero Background (5, 10, 100, 200, 250)",
    "Patches.png"
)

# Part D: Log Transformation  s = log(1 + C·I)
def log_transform(img, C):
    I = img.astype(np.float32)
    S = np.log1p(C * I)

    # Normalize
    S = 255 * (S - S.min()) / (S.max() - S.min() + 1e-12)
    return S.astype(np.uint8)

C_list = [1, 5, 10, 100]
log_images = {}  # C -> imgD

for C in C_list:
    imgD = log_transform(imgC, C)
    log_images[C] = imgD

    im_show(
        imgD,
        f"Log Transform: s = log(1 + C·I),  C = {C}",
        f"PartD_Log_C{C}.png"
    )
```

```
# Part E
fig, axes = plt.subplots(1, 1 + len(C_list), figsize=(4*(1+len(C_list)), 4))

# original
axes[0].imshow(imgC, cmap="gray", vmin=0, vmax=255)
axes[0].set_title("Part C (Original)")
axes[0].axis("off")

# log-transformed images
for i, C in enumerate(C_list, start=1):
    axes[i].imshow(log_images[C], cmap="gray", vmin=0, vmax=255)
    axes[i].set_title(f"Part D (C={C})")
    axes[i].axis("off")

plt.tight_layout()
plt.savefig("vis_score.png")
print("Saved: vis_score.png")
plt.show()

# Michelson Contrast
def michelson_contrast(patch_value, background=0):
    Lmax = patch_value
    Lmin = background
    if (Lmax + Lmin) == 0:
        return 0
    return (Lmax - Lmin) / (Lmax + Lmin)

# analysis
print("\n" + "="*65)
print(f"{'MICHELSON CONTRAST ANALYSIS':^65}")
```

```
print("="*65)
header = "{:<12} | ".format("Row \\ Patch") + " | ".join(f"{v:^7}" for v in vals)
print(header)
print("-" * len(header))

# Original
row_orig = f"{'Original':<12} | "
for v in vals:
    cval = michelson_contrast(v, background=0)
    row_orig += f"{cval:^7.3f} | "
print(row_orig)

# Log transformed
for C in C_list:
    row_log = f"C = {C:<9} | "
    for v in vals:
        cval = michelson_contrast(v, background=0)
        row_log += f"{cval:^7.3f} | "
    print(row_log)
print("-" * len(header))

# normalized log transformed values
print("\n" + "="*65)
print(f"{'NORMALIZED LOG-TRANSFORMED PIXEL VALUES':^65}")
print("="*65)
header2 = "{:<12} | ".format("Row \\ Patch") + " | ".join(f"{v:^7}" for v in vals)
print(header2)
print("-" * len(header2))

for C in C_list:
```

```
    row_norm = f"C = {C:<9} | "
    imgD = log_images[C]
    for k, v in enumerate(vals):
        left = gap + k * (patch + gap)
        patch_region = imgD[top:top+patch, left:left+patch]
        mean_val = np.mean(patch_region)
        row_norm += f"{mean_val:^7.1f} | "
    print(row_norm)
print("-" * len(header2))
```

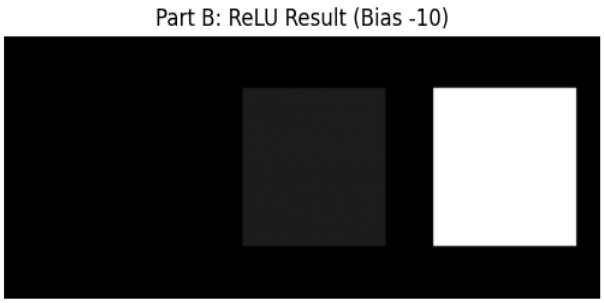
Problem 3

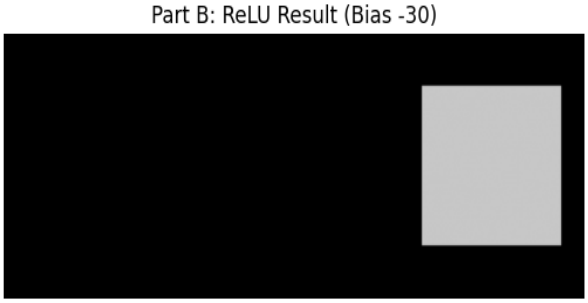
A. Create an image with zero background and the constant patches of 5 and 100.

First, we generated a synthetic image with a black background (value 0) and two constant patches with intensities of 5 and 100. The lower brightness patch (value 5) is barely visible, though. Here, I am also using a **patch of intensity value of 20**, as patch 5 is barely visible.

B. Show how a ReLU with proper bias will eliminate the lower-brightness patch.

The ReLU (Rectified Linear Unit) function is defined as f(x) = max (0, x). By introducing a bias b, the function becomes f(x) = max (0, x + b).





In this experiment, the image contains two constant patches: a low-brightness patch with intensity. 5 , a medium patch with intensity 20 and a brighter patch with intensity 100, both placed on a zero-valued background. By applying a negative bias, the low-intensity patch is shifted into the negative range and is therefore clipped to zero by ReLU. With $b = -10$: $f(5) = \max(0,5 - 10) = 0$, the patch with the value 5 disappears, while $f(100) = \max(0,100 - 10) = 90$, the brighter patch remains visible. With $b = -30$: $f(5) = \max(0,5 - 30) = 0$, and $b = -30$: $f(20) = \max(0, 20 - 30) = 0$, again eliminating the low and medium patch, and $f(100) = \max(0,100 - 30) = 70$, So the brighter patch is still visible but slightly darker. These results show that ReLU with a negative bias acts like a hard threshold: values below $-b$ are suppressed completely, while larger values remain. Increasing the magnitude of the negative bias removes dim regions more aggressively, while reducing the brightness of the remaining visible patch.

C) Sigmoid reduces the lower patch

The sigmoid activation function is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}},$$

and it maps all input values into the bounded range $(0, 1)$. Unlike ReLU, sigmoid does not completely remove low-intensity values, but instead compresses them toward values close to zero. Here, the image contains two constant patches: a dim patch with intensity 5, a medium patch with intensity 20, and a brighter patch with intensity 100, placed on a black background. To control how these intensities are mapped, a bias term b is added: $\sigma(x + b)$. The bias shifts the sigmoid curve so that lower values fall deeper into the dark region of the function.

With $b = -10$, the low-intensity patch 5 is pushed closer to zero as $\sigma(-5) = 0.00669$, becoming much darker and less visible, while the patch with value 20, $\sigma(20 - 10) = 0.99995$, so this is also reduced but still produces a noticeable gray level, and for the high-intensity patch (100), $\sigma(100 - 10) = 0.9999999$, so it remains near the bright end of the sigmoid output. However, with a stronger bias $b = -30$, the separation becomes more pronounced: the lower patches 5 ($\sigma(5 - 30) \approx 0$) and 20 ($\sigma(20 - 30) \approx 0$) are pushed much closer to 0, so they become very dark and difficult to distinguish from the background, while the brighter patch still maps close to 1 and stays clearly visible. These results show that a sigmoid with a negative bias acts like a soft threshold. Instead of eliminating the low-brightness patch completely, it gradually reduces its contribution, while preserving higher intensities. This makes sigmoid useful when we want to suppress dim regions without applying a hard cutoff.

Code:

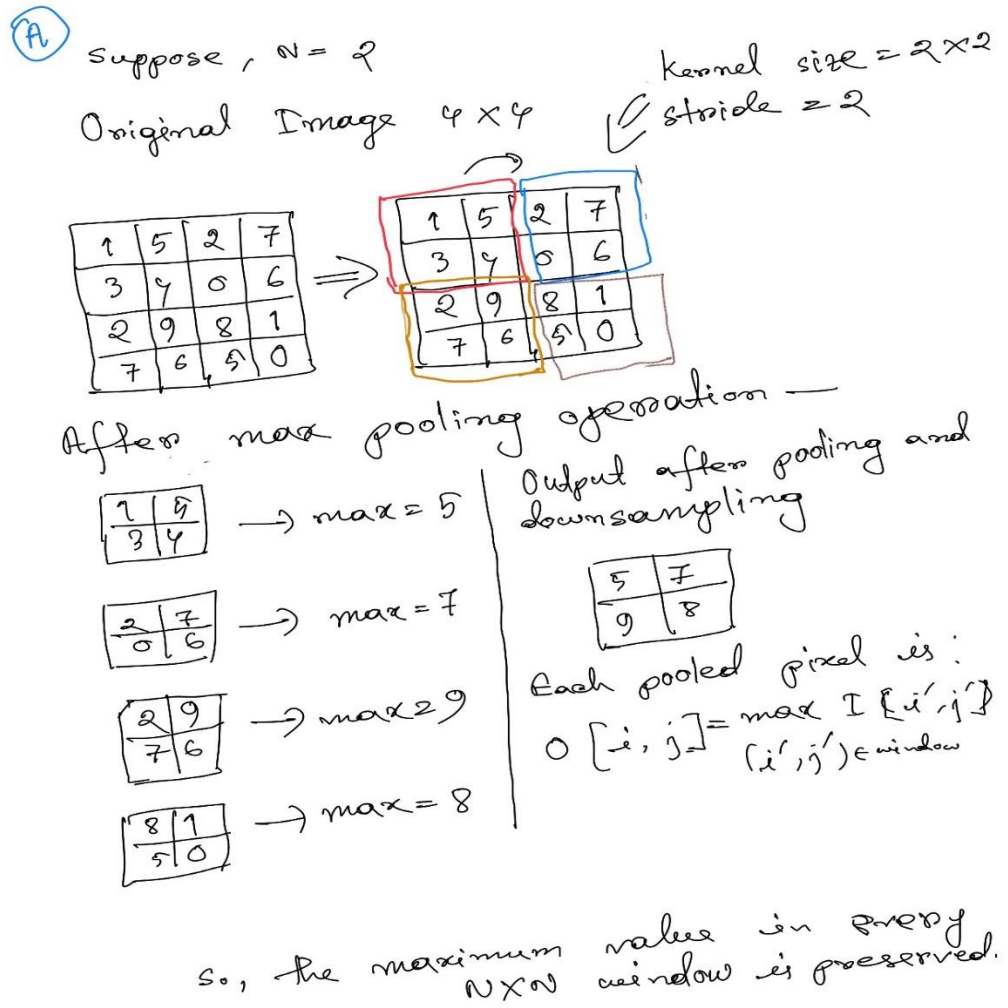
```
import numpy as np

import matplotlib.pyplot as plt


def solve_problem_3():


    # Base Image

    img = np.zeros((100, 250))
```

```
# intensity 5
img[20:80, 20:80] = 5

# intensity 20
img[20:80, 100:160] = 20

# intensity 100
img[20:80, 180:240] = 100

# Save
plt.figure()
plt.imshow(img, cmap="gray", vmin=0, vmax=100)
plt.title("Part A: Original Image (Patches 5, 20 & 100)")
plt.axis("off")
plt.savefig("Problem3_A_Original.png", bbox_inches="tight")
plt.show()
print("Saved: Problem3_A_Original.png")

#  ReLU with Proper Bias
for bias_relu in [10, -10, -30]:
    img_relu = np.maximum(0, img + bias_relu)

    plt.figure()
    plt.imshow(img_relu, cmap="gray", vmin=0, vmax=90)
    plt.title(f"Part B: ReLU Result (Bias {bias_relu})")
    plt.axis("off")

    filename = f"Problem3_B_ReLU_Bias{bias_relu}.png"
    plt.savefig(filename, bbox_inches="tight")
```

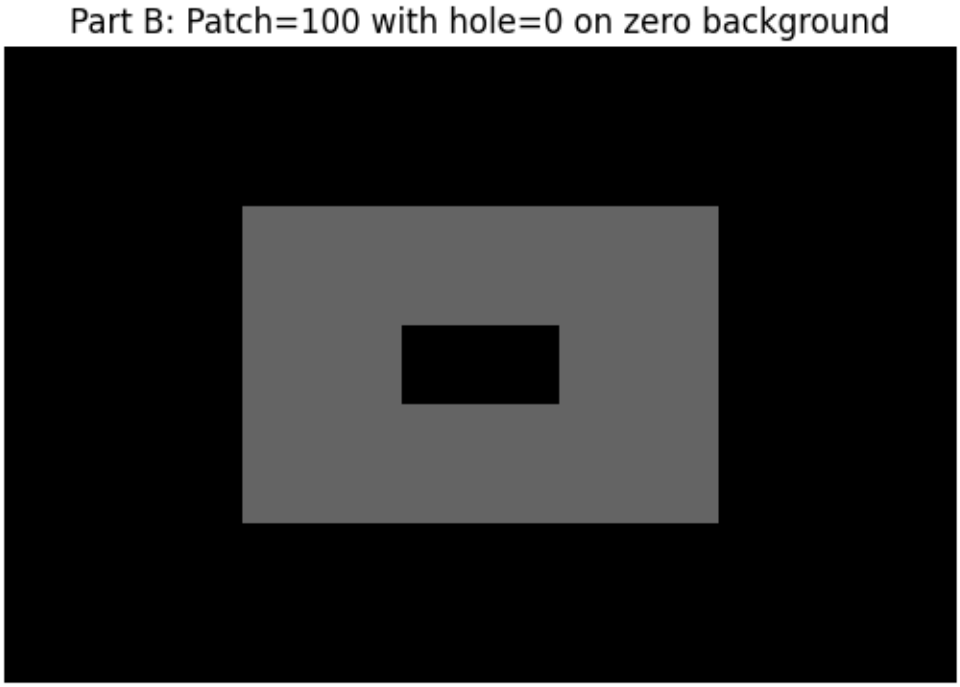
```
        plt.show()
        print("Saved:", filename)


    # Sigmoid with Proper Bias
    for bias_sig in [ -10, -30]:

        img_sigmoid = 1 / (1 + np.exp(-(img + bias_sig)))

        plt.figure()
        plt.imshow(img_sigmoid, cmap="gray", vmin=0, vmax=1)
        plt.title(f"Part C: Sigmoid Result (Bias {bias_sig})")
        plt.axis("off")

        filename = f"Problem3_C_Sigmoid_Bias{bias_sig}.png"
        plt.savefig(filename, bbox_inches="tight")

        plt.show()
        print("Saved:", filename)



if __name__ == "__main__":
    solve_problem_3()
```

Problem 4:

A. Show that max pooling by a NxN window, stride=N, followed by NxN downsampling preserves the maximum value in the original window.



B. Create an image with zero background, a rectangular patch with a value of 100 with a smaller rectangular hole with value=0.



C. Use the rolling-ball analogy from dilation to estimate the output from dilating with a 2x2. Show your work using pencil and paper.

(c) Gray scale dialation with a 2X2 structuring element (Rolling Ball analogy)

Here, we apply grayscale dialation using 2X2 structuring element. Dialation in gray-scale morphology defined as:

$$O[i,j] = \max_{(i',j') \in s+(i,j)} I[i',j']$$

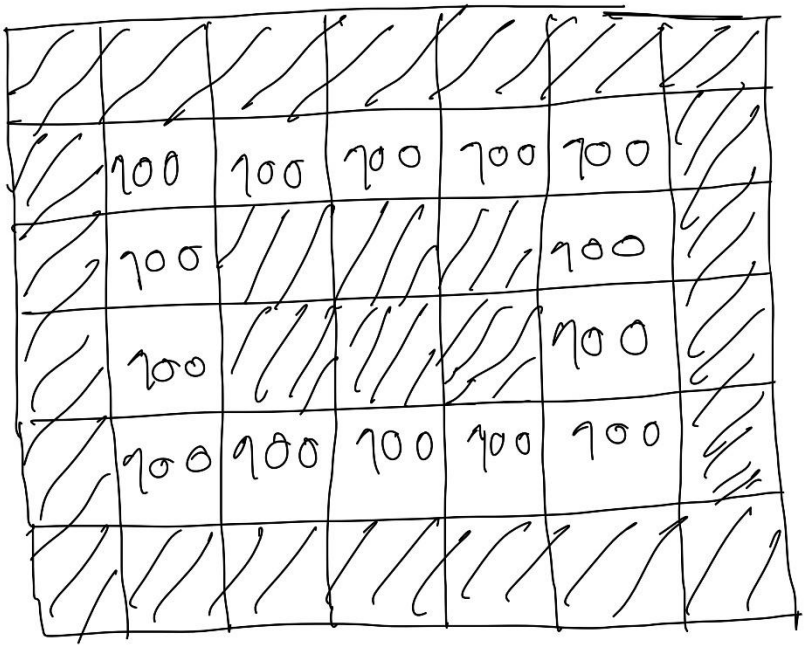
This means that for every pixel location (i,j), the o/p value is the maximum intensity inside the local 2X2 neighbor-hood.

The i/p image contains

→ A large rectangular bright patch of $I = 100$

→ A small rectangular dark hole of $I = 0$

In gray scale dialation stride is always 1.

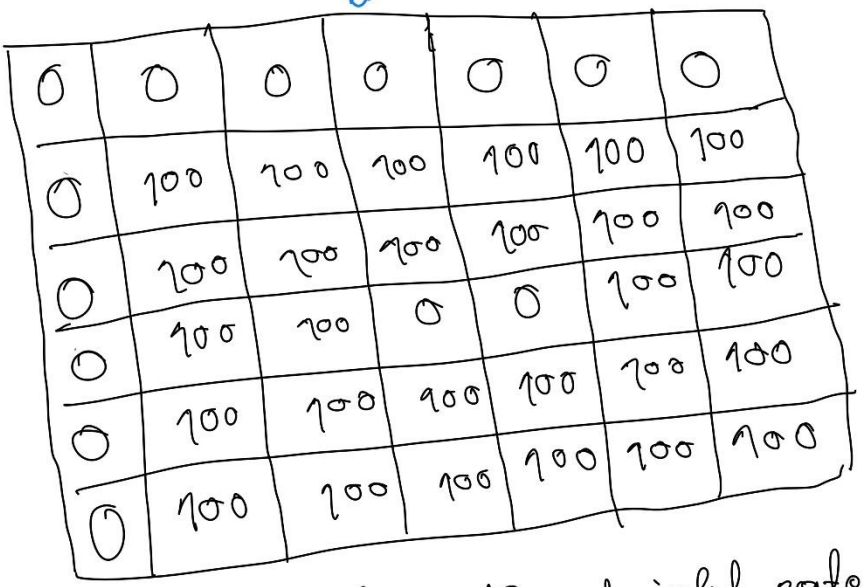


Suppose, this is my input image where there is a black background, a rectangular bright patch of intensity inside and a rectangular hole of intensity 0. So the image becomes like the following:

0	0	0	0	0	0	0
0	100	100	100	100	100	0
0	100	0	0	0	100	0
0	100	0	0	0	100	0
0	100	100	100	100	100	0
0	0	0	0	0	0	0

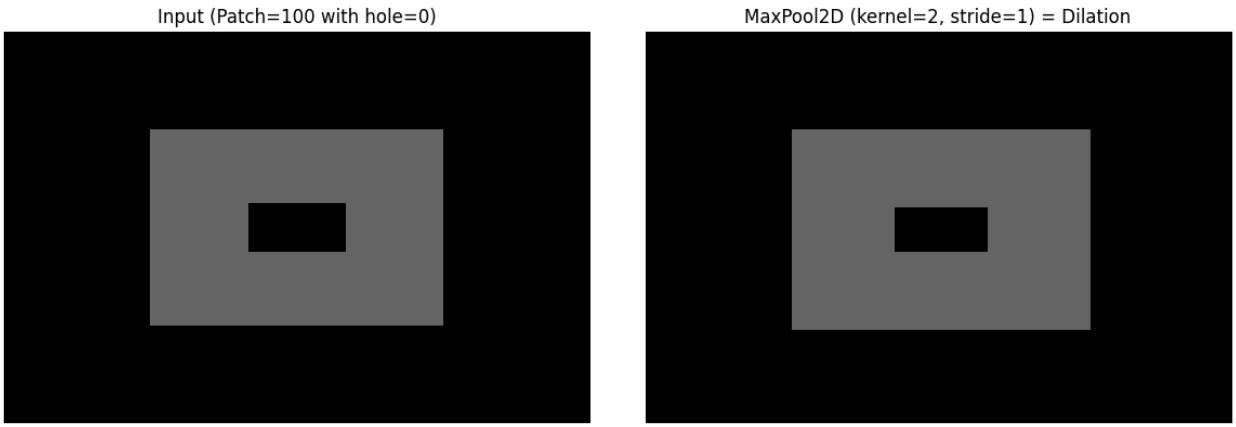
Now for dialation we take kernel size = 2 and stride = 1. Using rolling ball analogy —

If we use padding —



So, after dialation the bright patch expand inward and the hole shrinks.

D. Verify your example using max pooling in Python.




```
Input shape : torch.Size([1, 1, 80, 120])
Output shape: torch.Size([1, 1, 80, 120])
Hole size before dilation: height=10, width=20
Hole size after dilation (k=2): height=9, width=19
```

In this part of the problem, max pooling is utilized in Python to verify the grayscale definition of dilation. The core concept explored is that for grayscale morphology, dilation is equivalent to max pooling without downsampling, which is achieved by setting the stride to 1. The starting image consists of a zero (black) background featuring a rectangular patch with a pixel value of 100. Inside this rectangular patch, a smaller rectangular hole with a pixel value of 0 is placed. A 2x2 kernel with a stride of 1 is applied using the MaxPool2d function. As shown in the "Dilation" output image, the max pooling operation effectively causes the high-intensity pixels (value 100) to expand. This expansion results in the internal "hole" (value 0) becoming smaller, as the max operator selects the surrounding higher value of 100 as the kernel overlaps with the hole's boundaries. The **hole shape of [10*20] before dilation** results in a **hole shape of [9*19] after dilation**. This reduction in height and width by one pixel occurs because the bright pixel of 100 moves inward. So, it can be said that applying a max filter with a stride of 1 correctly performs grayscale dilation, expanding brighter regions and shrinking darker internal features like holes.

E. Demonstrate how you can increase the size of the max pooling to eliminate the smaller rectangular hole. What is the minimum square size of your kernel? Derive a mathematical expression

(E) From the image we generate at part B, we get that —

The rectangular hole has

height = h = 10

width = w = 20

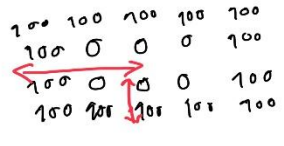
The hole is surrounded by a constant intensity value of 100.

A single grayscale dialation implemented by maxpooling with a square kernel of size k×k (with odel k, same padding) fills a hole pixel if there exists at least one bright pixel within the kernel window centered at the hole pixel.

With a square k×k window, the window reaches ⌊k/2⌋ pixels away from the center in every direction.

For odd k, the radius of a $k \times k$ square kernel is, $r = \frac{k-1}{2}$

This is how many pixels from the centre the dilation can reach all directions.



Now, the hardest point to fill is the center of the hole, because it is the farthest from the boundary values (100).

So, distance from the hole center to
the → top/bottom boundary $\approx \lceil \frac{h}{2} \rceil$
→ left/right " $\approx \lceil \frac{w}{2} \rceil$

The hole becomes 100 as soon as any boundary side 100 lies in the window, so we only need the kernel radius to reach the

nearest boundary direction. Therefore the required radius of the kernel is —

$$r \geq \min\left(\left\lceil \frac{h}{2} \right\rceil, \left\lceil \frac{w}{2} \right\rceil\right)$$

Now, $r \geq \frac{k-1}{2}$

So, $\frac{k-1}{2} \geq \min\left(\left\lceil \frac{h}{2} \right\rceil, \left\lceil \frac{w}{2} \right\rceil\right)$

$$k_{min} = 2 \min\left(\left\lceil \frac{h}{2} \right\rceil, \left\lceil \frac{w}{2} \right\rceil\right) + 1$$

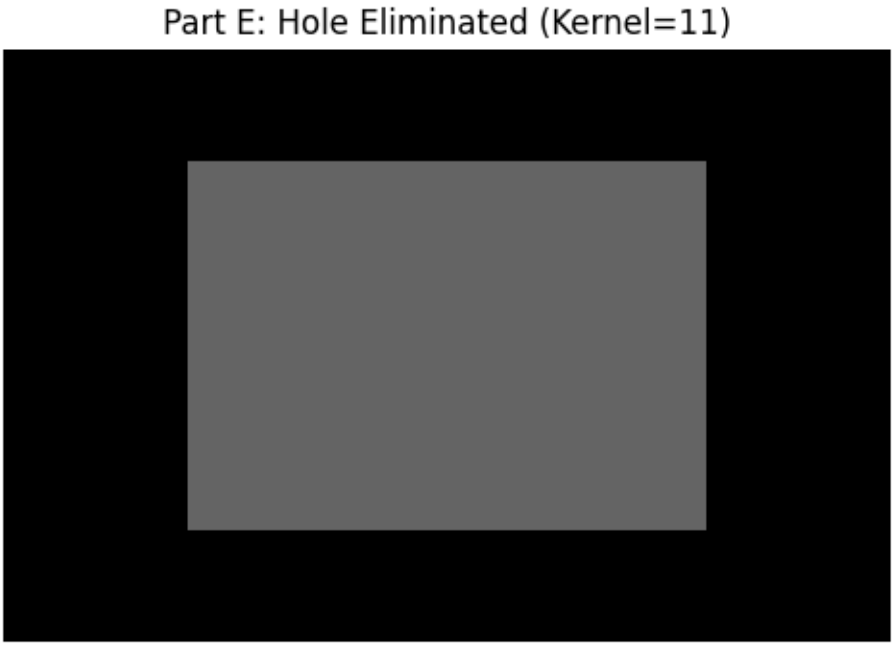
This is the expression for the minimum square maxpool kernel (single steps) the eliminates the rectangular hole. From part B for our hole, $w = 20$

$h = 10$

So, $k_{min} = 2 \min(5, 10) + 1$

$= 2 \times 5 + 1 = 11$

So, the minimum kernel size will be 11 X 11 for eliminating the hole.



F. Suppose that the hole still appears after the max pooling operation. Sketch a diagram to show how it can disappear during the downsampling process.

(F) The hole can still appear after maxpooling when we do max pooling (or dialation) without downsampling and the kernel is not large enough to fully remove the hole. Then the downsampling step can make the remaining hole pixels disappear.

If we do maxpooling with kernel size 2×2 and stride 1. for our input image after 1-step the hole shrinks but it still appears as following →

Input image

100	100	100	100	100
100	0	0	0	100
100	0	0	0	100
100	100	100	100	100

2×2 maxpooling stride = 1

Output image

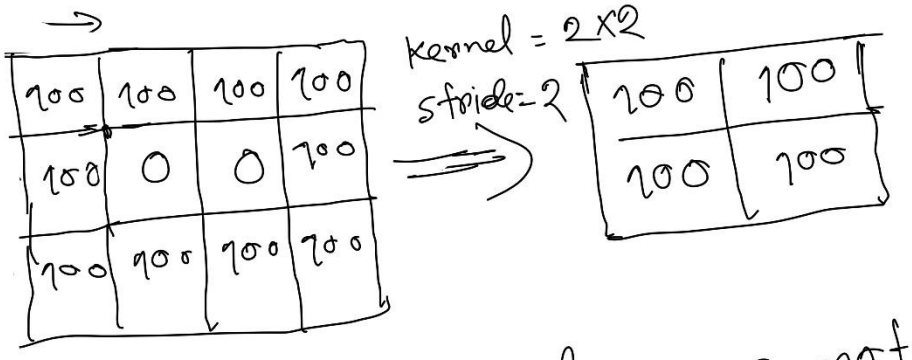
100	100	100	100
100	0	0	100
100	100	100	100

So, there are still two '0' pixels.

From part (A) we get,

maxpooling by N×N window, stride = N, followed by N×N downsampling

For our case, N=2 which means the downsampling keeps only every 2nd pixel. So, downsampling by 2 keeps.



So, the whole pixels are not sampled in this case and discarded. So, during downsampling max pooling, if the window covering the hole contains at least one foreground pixel, the pooled o/p becomes foreground and hole disappears.

G. Suppose that the input image is multiplied by -1. Then apply the max pooling operation without downsampling. What is the relationship of the output image to applying an erosion to the input image?

(G) Let the input grayscale image be I and S be the neighbourhood (kernel) centered at (i,j)

If we multiply the image I with -1, it turns into $\longrightarrow -I$

After applying maxpooling with an $N \times N$ window and stride 1 to that image.

$$\text{Maxpool}(-I, s)[i,j] = \max_{(i',j') \in S+(i,j)} (-I[i',j'])$$

According to algebra, the maximum of a set of negative numbers is the negative of the minimum of those positive numbers.

$$\max\{-a, -b, -c\} = -\min\{a, b, c\}$$

So,

$$\text{Maxpool}(-I, s)[i,j] = -\min_{i',j' \in S} \{I(i',j')\}$$

But erosion is:

$$\text{Erosion}(I, S)[i,j] = \min_{(i',j') \in S+(i,j)} I[i',j']$$

So,

Maxpool $(-I, s)[i, j] = -\text{Erosion}(I, s)[i, j]$

Suppose,

$$I = \begin{Bmatrix} 100, & 0, \\ 50, & 200 \end{Bmatrix}$$

Multiply with -1

$$\{-I\} = \{-100, 0, -50, -200\}$$

So, bright pixels becomes very negative

and dark " " less negative.

So, maxpooling $\{-I\} = \max(-100, 0, -50, -200)$

$= 0$

$= \text{Erosion}$

H. How can you implement erosion using max pooling and multiplication by a constant?

(H) To implement erosion using only Max pooling and constant multiplication we need to —

(a) Multiply the original image I by a constant $c = -1$

(b) Apply max pooling with desired kernel size to an inverted image

(c) Again multiply the maxpooled output with constant -1

So, $\text{erosion}(I, s) = -\text{Maxpool}(-I, s)$

Suppose, $I = \{100, 40, 70, 10\}$

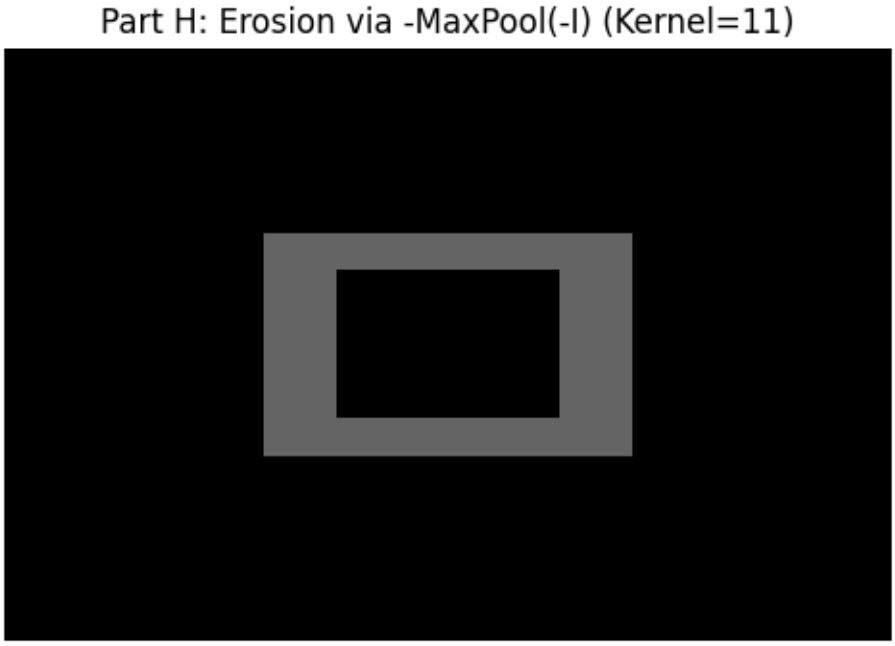
Erosion $I = \min\{100, 40, 70, 10\}$

$= 10$

$\{-I\} = \{-100, -40, -70, -10\}$

$$\text{Maxpool}\{-I\} = \max\{-100, -40, -70, -90\}$$
$$= -10$$
$$= -\text{Erosion } I$$

$$\text{So, Erosion } I = -\text{Maxpool}\{-I\}$$



Code:

```
import numpy as np
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt

# Part B
```

```
def im_show(img, title="", filename=None):
    plt.figure()
    plt.imshow(img, cmap="gray", vmin=0, vmax=255)
    plt.title(title)
    plt.axis("off")
    if filename is not None:
        plt.savefig(filename, bbox_inches="tight")
        print(f"Saved: {filename}")
    plt.show()

rows, cols = 80, 120
img = np.zeros((rows, cols), dtype=np.float32)

# patch value = 100
img[20:60, 30:90] = 100

# rectangle hole
img[35:45, 50:70] = 0 #h=20, w=10

# Original image
im_show(
    img,
    "Part B: Patch=100 with hole=0 on zero background",
    "Problem4_B.png"
)

patch_r0, patch_r1 = 20, 60
patch_c0, patch_c1 = 30, 90

patch_before = img[patch_r0:patch_r1, patch_c0:patch_c1] #hole size before dialation
```

```
mask_before = (patch_before == 0)

rr, cc = np.where(mask_before)
hole_h_before = rr.max() - rr.min() + 1
hole_w_before = cc.max() - cc.min() + 1


# Part D

x = torch.tensor(img).unsqueeze(0).unsqueeze(0)   # shape: (1, 1, H, W)

# pad by 1 pixel on all sides
x_pad = F.pad(x, (1, 1, 1, 1), mode="constant", value=0)

# max pooling (dilation) on padded image
dilated_pad = F.max_pool2d(x_pad, kernel_size=2, stride=1)

H, W = x.shape[-2], x.shape[-1]
dilated = dilated_pad[:, :, 0:H, 0:W]   # now same shape as input

dilated_np = dilated.squeeze().numpy()
patch_after = dilated_np[patch_r0:patch_r1, patch_c0:patch_c1] #hole size before dialation
mask_after = (patch_after == 0)



# Visualization of input vs output

plt.figure(figsize=(12,4))

plt.subplot(1,2,1)
```

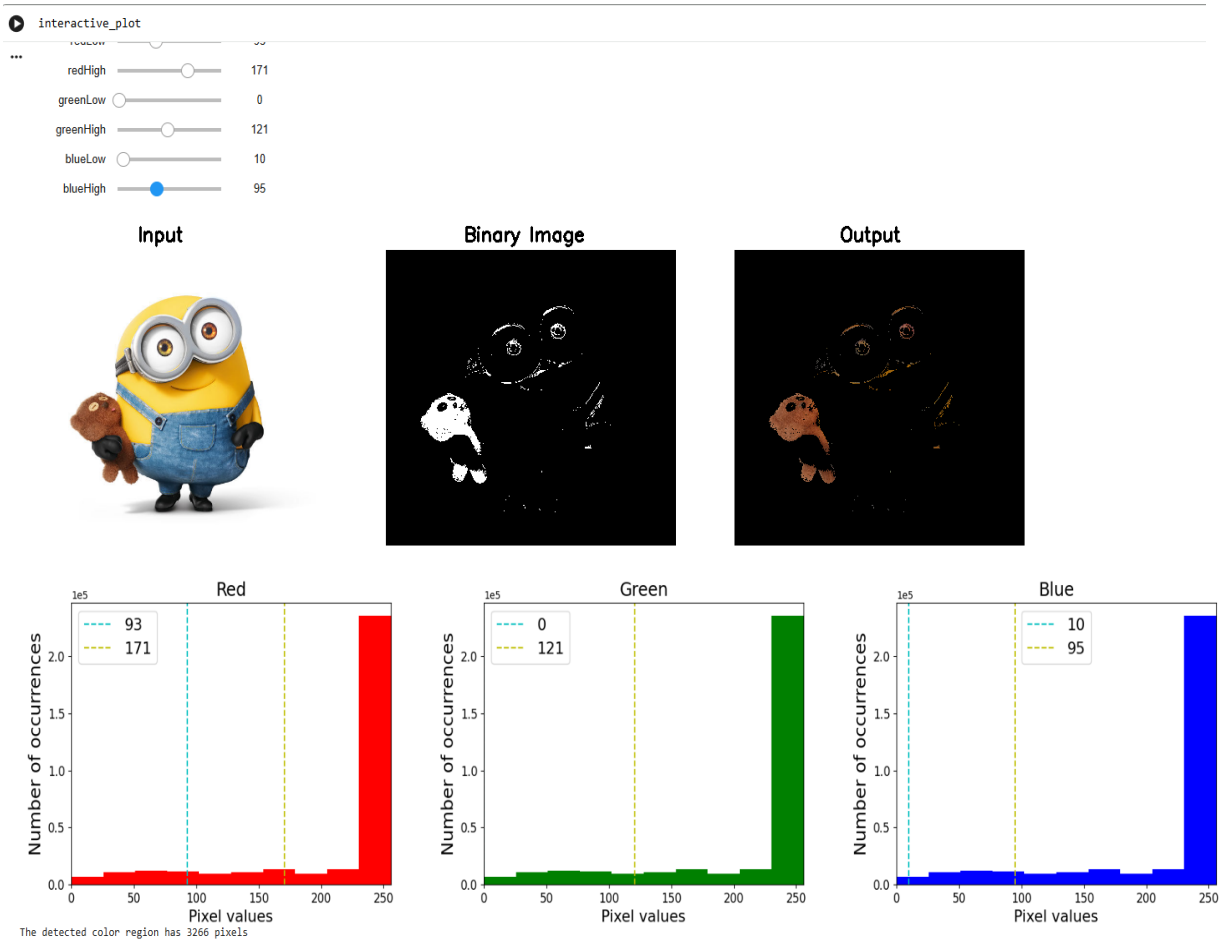
```
plt.imshow(img, cmap="gray", vmin=0, vmax=255)
plt.title("Input (Patch=100 with hole=0)")
plt.axis("off")

plt.subplot(1,2,2)
plt.imshow(dilated_np, cmap="gray", vmin=0, vmax=255)
plt.title("MaxPool2D (kernel=2, stride=1) = Dilation")
plt.axis("off")

plt.tight_layout()
plt.savefig("Problem4_D.png", bbox_inches="tight")
plt.show()

print("Input shape :", x.shape)
print("Output shape:", dilated.shape)
print(f"Hole size before dilation: height={hole_h_before}, width={hole_w_before}")
if np.any(mask_after):
    rr2, cc2 = np.where(mask_after)
    hole_h_after = rr2.max() - rr2.min() + 1
    hole_w_after = cc2.max() - cc2.min() + 1
    print(f"Hole size after dilation (k=2): height={hole_h_after}, width={hole_w_after}")
else:
    print("Hole size after dilation (k=2): eliminated")

# Part E verification

kernel_size_E = 11
p = kernel_size_E // 2  # 5 for k=11

x_pad_E = F.pad(x, (p, p, p, p), mode="constant", value=0)
```

```
dilated_pad_E = F.max_pool2d(x_pad_E, kernel_size=kernel_size_E, stride=1)

H, W = x.shape[-2], x.shape[-1]
dilated_E = dilated_pad_E[:, :, 0:H, 0:W]
dilated_np_E = dilated_E.squeeze().numpy()

#visualization
plt.figure(figsize=(6,4))
plt.imshow(dilated_np_E, cmap="gray", vmin=0, vmax=255)
plt.title(f"Part E: Hole Eliminated (Kernel={kernel_size_E})")
plt.axis("off")
plt.savefig("Problem4_E.png", bbox_inches="tight")
plt.show()

# Part H
# Erode(I) = - MaxPool( -I )
kernel_size_H = 11
pH = kernel_size_H // 2

# converting to torch, applying erosion
x_pad_H = F.pad(-x, (pH, pH, pH, pH), mode="constant", value=0)
eroded_pad = F.max_pool2d(x_pad_H, kernel_size=kernel_size_H, stride=1)
Hh, Wh = x.shape[-2], x.shape[-1]
eroded = -eroded_pad[:, :, 0:Hh, 0:Wh]

eroded_np = eroded.squeeze().numpy()

#visualization
plt.figure(figsize=(6,4))
plt.imshow(eroded_np, cmap="gray", vmin=0, vmax=255)
```

```
plt.title(f"Part H: Erosion via -MaxPool(-I) (Kernel={kernel_size_H})")
plt.axis("off")
plt.savefig("Problem4_H.png", bbox_inches="tight")
plt.show()
```

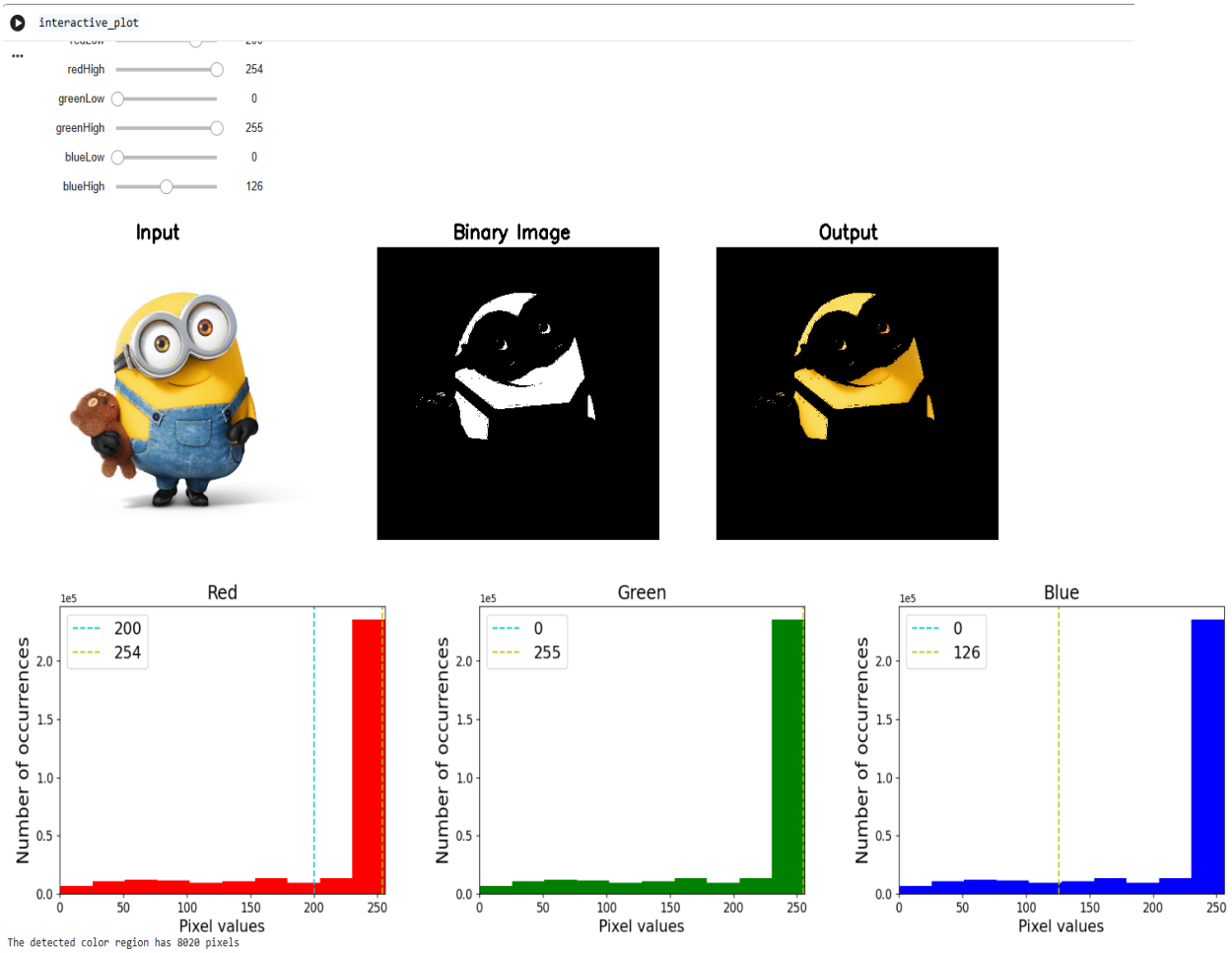
Problem 5:

A. The bear



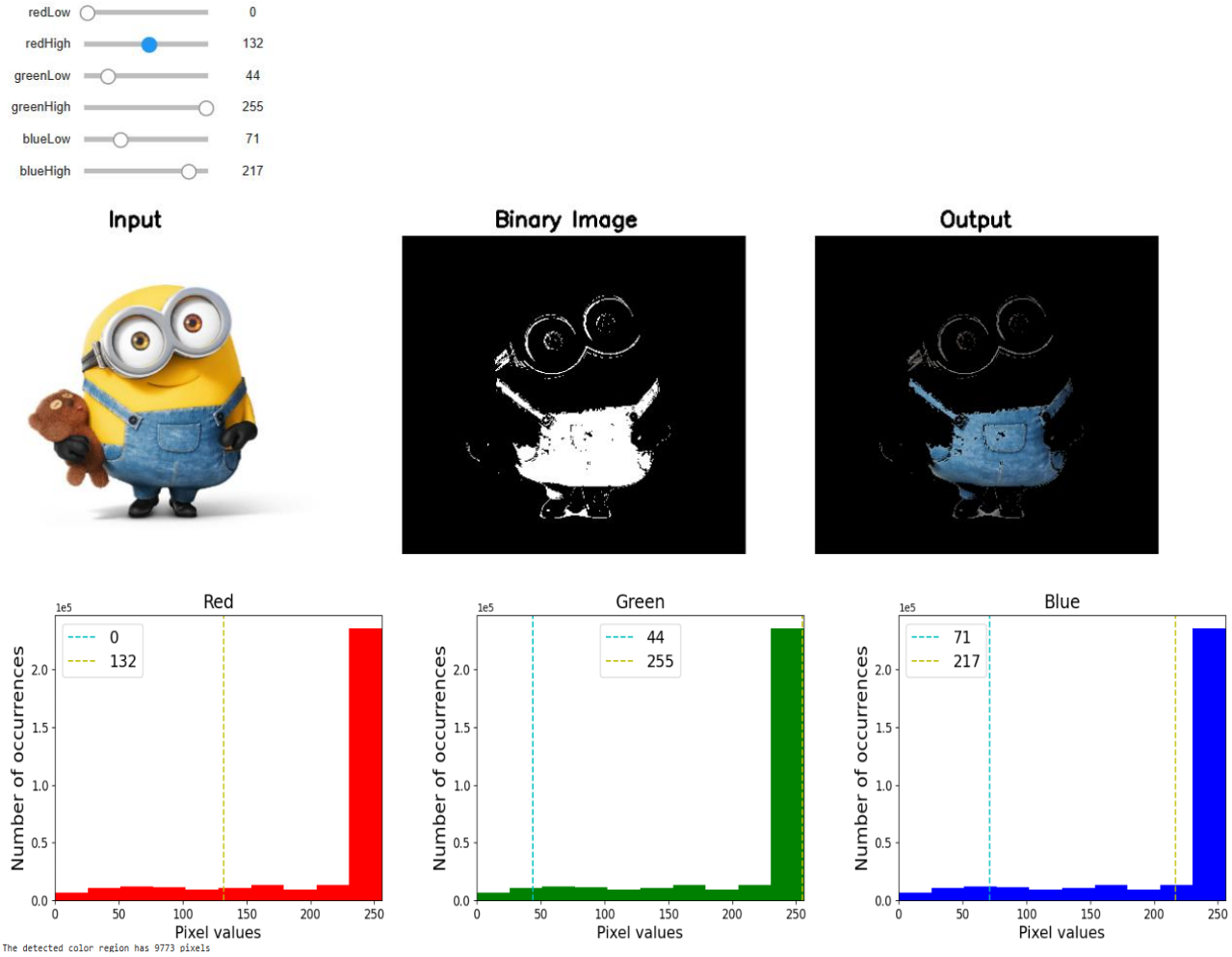
Detecting the bear involves selecting pixels with RGB values within the teddy bear's brown color range, creating a binary mask that isolates it from the image. The bear has a significant red component, with a lower bound of 93 to capture darker tones and an upper bound of 171 to exclude the brighter yellow of the Minion's skin. Brown is a warm color with low green content, capping the green channel at 121 filters out bright yellow pixels. The blue channel remains low since brown has minimal blue, aiding in distinguishing the bear from the Minion's blue overalls. The resulting mask reveals a strong white cluster corresponding to the bear's shape, with 3,266 pixels detected. While the output effectively isolates the bear, some small noise patches from the Minion's goggles appear due to their shadow values falling within the specified RGB ranges.

B. Skin color (yellow-like)



Yellow is a secondary color in the RGB model, created by a combination of high intensities in both the Red and Green channels while maintaining a lower intensity in the Blue channel. For the red channel, the lower bound is set extremely high at 200, capturing only the most vibrant red components. This is essential for yellow, which requires a near-peak red intensity. The green channel is left completely open, covering the entire range from 0 to 255. While yellow requires high green, this broad range ensures that various shaded or shadowed areas of the skin are not accidentally excluded. To maintain the yellow hue, the blue channel must be suppressed. Setting an upper bound of 126 effectively excludes white areas (which have high blue) and the blue denim overalls. The resulting binary image shows a clear, high-confidence mask covering the Minion's head and arms. The segmentation effectively excludes the goggles, overalls, and the brown teddy bear. The detected color region consists of 8,020 pixels. This is significantly more than the 3,266 pixels found when detecting the smaller brown bear. The use of a Sigmoid function, as discussed in related image processing problems, could potentially be used to refine these boundaries by creating a smoother transition for pixel intensities rather than a hard threshold cut-off.

C. Clothes for the minion.



Because the overalls are a distinct blue-denim color, the segmentation must prioritize high blue values while suppressing extremely high red values to avoid the yellow skin or white background. The red intensity upper bound is capped at 132. This relatively low upper bound is critical for excluding the yellow skin and white background, both of which contain very high red components. While denim contains some green, the range is set broadly from 44 up to 255. However, the restriction in the red and blue channels ensures that only "bluer" pixels are selected despite this broad green range. A minimum blue threshold of 71 ensures that dark shadows are included, while allowing values up to 217 to capture the vibrant blue of the denim.

The resulting binary image effectively isolates the overalls, including the shoulder straps and the main body of the garment. The segmentation also picks up the blue-tinted shadows on the minion's boots and the darker reflections in the goggles. The detected color region consists of 9773 pixels. The "Output" image shows that the overalls are mostly captured.

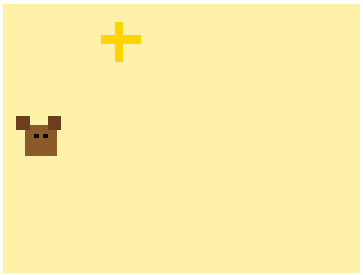
Problem 6

This script generates a short synthetic video sequence composed of two moving objects on a uniform background. Each frame is an 80 × 60image initialized with a constant “butter” color. A simple teddy icon is drawn using small square patches for the head and ears, plus single-pixel eyes. The teddy moves horizontally from left to right while its vertical position follows a sinusoidal trajectory, producing a clear bouncing motion across frames.

A second object—a “banana” represented as a thick plus sign—is animated independently. It moves from the top toward the bottom of the frame, while its horizontal position oscillates with a sine-wave “wiggle,” creating a smooth drifting path. By combining linear motion with periodic (sinusoidal) offsets, the video demonstrates how basic parametric equations can control object motion and produce natural-looking animations.

Overall, the sequence is useful for understanding frame-by-frame video construction, coordinate-based drawing, and motion modeling. Because the shapes and trajectories are controlled and noise-free, it can also serve as a simple testbed for tracking, motion estimation, or basic video understanding experiments.

The video is given following:



download (2).mp4

Code:

```
!wget https://raw.githubusercontent.com/pattichis/AIML/main/AOLME.py

from AOLME import *

import numpy as np

from IPython.display import HTML
```

```
# Video setup
rows, cols = 60, 80
num_frames = 20
fps = 6

# Colors (hex RGB)
butter = "fff2a8"   # butter yellow background
bear   = "8b5a2b"   # brown teddy
bear2  = "6b3f1d"   # darker brown details
banana = "ffd400"   # banana yellow
black  = "000000"   # eyes

# Helpers: draw objects
def draw_teddy(frame, cx, cy):
    """
    Simple teddy icon: head: 7x7 square, ears: two 3x3 squares, eyes: 1-pixel dots
    """
    # head
    y0, y1 = max(0, cy-3), min(rows, cy+4)
    x0, x1 = max(0, cx-3), min(cols, cx+4)
    frame[y0:y1, x0:x1] = bear

    # ears
    frame[max(0, cy-5):min(rows, cy-2), max(0, cx-5):min(cols, cx-2)] = bear2
    frame[max(0, cy-5):min(rows, cy-2), max(0, cx+2):min(cols, cx+5)] = bear2
```

```
    # eyes
    ey = cy-1
    for ex in (cx-1, cx+1):
        if 0 <= ey < rows and 0 <= ex < cols:
            frame[ey, ex] = black


# Banana drawn like a plus sign:vertical stroke + horizontal stroke, thickness controls how fat the plus looks
def draw_banana_plus(frame, cx, cy, arm=4, thickness=2, color=banana):


    # vertical stroke
    y0, y1 = max(0, cy-arm), min(rows, cy+arm+1)
    for dx in range(-(thickness//2), thickness - (thickness//2)):
        x = cx + dx
        if 0 <= x < cols:
            frame[y0:y1, x] = color


    # horizontal stroke
    x0, x1 = max(0, cx-arm), min(cols, cx+arm+1)
    for dy in range(-(thickness//2), thickness - (thickness//2)):
        y = cy + dy
        if 0 <= y < rows:
            frame[y, x0:x1] = color


# Build frames


frame_list = []
```

```
# Teddy moves left to right
teddy_y = rows // 2
teddy_x_start, teddy_x_end = 8, cols - 9

# Banana-plus moves top to bottom with wiggle
banana_y_start, banana_y_end = 8, rows - 9
banana_x_base = cols // 3
wiggle_amp = 8  # how far it wiggles left-right

# teddy bounces vertically like a sine wave
teddy_bounce_amp = 10  # bounce height in pixels

for t in range(num_frames):
    frame = np.full((rows, cols), butter)

    # Teddy position (linear motion and sine-wave bounce)
    teddy_x = int(teddy_x_start + (teddy_x_end - teddy_x_start) * t / (num_frames - 1))
    bounce = int(teddy_bounce_amp * np.sin(2 * np.pi * t / (num_frames - 1)))
    teddy_y_bounce = int(np.clip(teddy_y + bounce, 6, rows - 7))
    draw_teddy(frame, teddy_x, teddy_y_bounce)

    # Banana-plus position (downward motion and sine wiggle)
    banana_y = int(banana_y_start + (banana_y_end - banana_y_start) * t / (num_frames - 1))
    wiggle = int(wiggle_amp * np.sin(2 * np.pi * t / (num_frames - 1)))
    banana_x = int(np.clip(banana_x_base + wiggle, 6, cols - 7))
    draw_banana_plus(frame, banana_x, banana_y, arm=4, thickness=2, color=banana)
```

```
    frame_list.append(frame)

# Play video
play_video = vid_show(frame_list, fps)
HTML(play_video.to_html5_video())

# Save as GIF

import imageio

def hex_to_rgb01(h):
    h = h.lstrip('#')
    return [int(h[i:i+2], 16)/255 for i in (0, 2, 4)]

gif_frames = []

for frame in frame_list:
    rgb = np.zeros((rows, cols, 3), dtype=np.uint8)
    for color in np.unique(frame):
        mask = (frame == color)
        r, g, b = [int(c*255) for c in hex_to_rgb01(color)]
        rgb[mask] = [r, g, b]
    gif_frames.append(rgb)

imageio.mimsave("teddy_banana.gif", gif_frames, duration=1/fps)
print("Saved: teddy_banana.gif")
```

In [ ]:

Problem 2

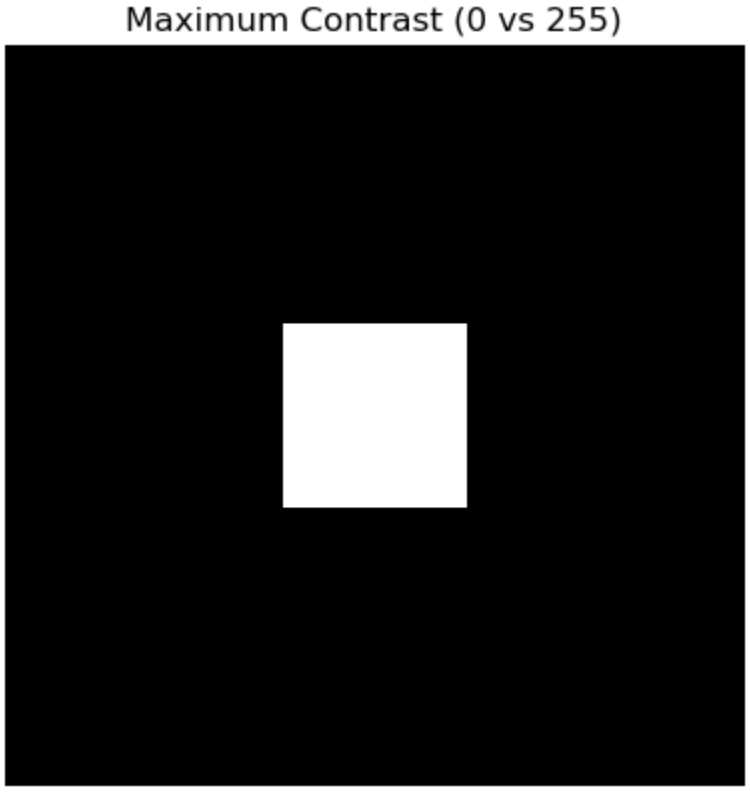
In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

# for displaying and saving
def im_show(img, title="", filename=None):
    plt.figure()
    plt.imshow(img, cmap="gray", vmin=0, vmax=255)
    plt.title(title)
    plt.axis("off")

    # Save image
    if filename is not None:
        plt.savefig(filename)
        print(f"Saved: {filename}")

    plt.show()

# Part A: Minimum Non-Zero Contrast
rows, cols = 64, 64
imgA = np.full((rows, cols), 127)
imgA[24:40, 24:40] = 128

im_show(
    imgA,
    "Minimum Non-Zero Contrast (127 vs 128)",
    "MinContrast.png"
)

# Part B: Maximum Contrast
imgB = np.full((rows, cols), 0)
imgB[24:40, 24:40] = 255

im_show(
    imgB,
    "Maximum Contrast (0 vs 255)",
    "MaxContrast.png"
)

# Part C: Constant Patches on Zero Background
imgC = np.zeros((80, 160))

vals = [5, 10, 100, 200, 250]
patch = 20
gap = 10
top = 30

# Creating multiple patches of the same size in a loop
for k, v in enumerate(vals):
    left = gap + k * (patch + gap)
    imgC[top:top + patch, left:left + patch] = v
```

```
im_show(
    imgC,
    "Patches on Zero Background (5, 10, 100, 200, 250)",
    "Patches.png"
)

# Part D: Log Transformation  s = log(1 + C·I)
def log_transform(img, C):
    I = img.astype(np.float32)
    S = np.log1p(C * I)

    # Normalize
    S = 255 * (S - S.min()) / (S.max() - S.min() + 1e-12)
    return S.astype(np.uint8)

C_list = [1, 5, 10, 100]
log_images = {}  # C -> imgD

for C in C_list:
    imgD = log_transform(imgC, C)
    log_images[C] = imgD

    im_show(
        imgD,
        f"Log Transform: s = log(1 + C·I),  C = {C}",
        f"PartD_Log_C{C}.png"
    )

# Part E
fig, axes = plt.subplots(1, 1 + len(C_list), figsize=(4*(1+len(C_list)), 4))

# original
axes[0].imshow(imgC, cmap="gray", vmin=0, vmax=255)
axes[0].set_title("Part C (Original)")
axes[0].axis("off")

# log-transformed images
for i, C in enumerate(C_list, start=1):
    axes[i].imshow(log_images[C], cmap="gray", vmin=0, vmax=255)
    axes[i].set_title(f"Part D (C={C})")
    axes[i].axis("off")

plt.tight_layout()
plt.savefig("vis_score.png")
print("Saved: vis_score.png")
plt.show()

# Michelson Contrast
def michelson_contrast(patch_value, background=0):
    Lmax = patch_value
    Lmin = background
    if (Lmax + Lmin) == 0:
        return 0
    return (Lmax - Lmin) / (Lmax + Lmin)
```

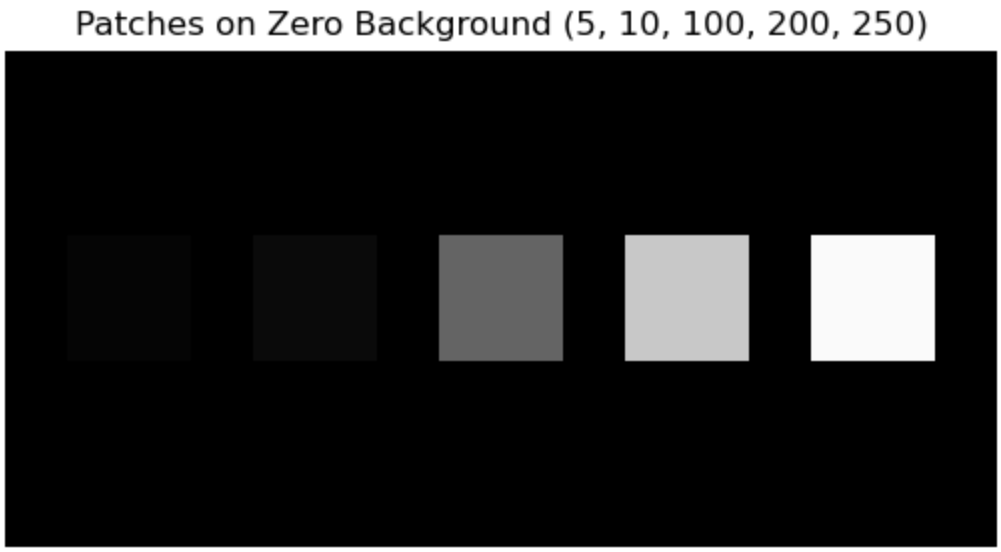
```
# analysis
print("\n" + "="*65)
print(f"{'MICHELSON CONTRAST ANALYSIS':^65}")
print("="*65)
header = "{:<12} | ".format("Row \\ Patch") + " | ".join(f"{v:^7}" for v in vals)

print(header)
print("-" * len(header))

# Original
row_orig = f"{'Original':<12} | "
for v in vals:
    cval = michelson_contrast(v, background=0)
    row_orig += f"{cval:^7.3f} | "
print(row_orig)

# Log transformed
for C in C_list:
    row_log = f"C = {C:<9} | "
    for v in vals:
        cval = michelson_contrast(v, background=0)
        row_log += f"{cval:^7.3f} | "
    print(row_log)
print("-" * len(header))

# normalized log transformed values
print("\n" + "="*65)
print(f"{'NORMALIZED LOG-TRANSFORMED PIXEL VALUES':^65}")
print("="*65)
header2 = "{:<12} | ".format("Row \\ Patch") + " | ".join(f"{v:^7}" for v in vals)

print(header2)
print("-" * len(header2))

for C in C_list:
    row_norm = f"C = {C:<9} | "
    imgD = log_images[C]
    for k, v in enumerate(vals):
        left = gap + k * (patch + gap)
        patch_region = imgD[top:top+patch, left:left+patch]
        mean_val = np.mean(patch_region)
        row_norm += f"{mean_val:^7.1f} | "
    print(row_norm)
print("-" * len(header2))
```

```
Saved: MinContrast.png
```

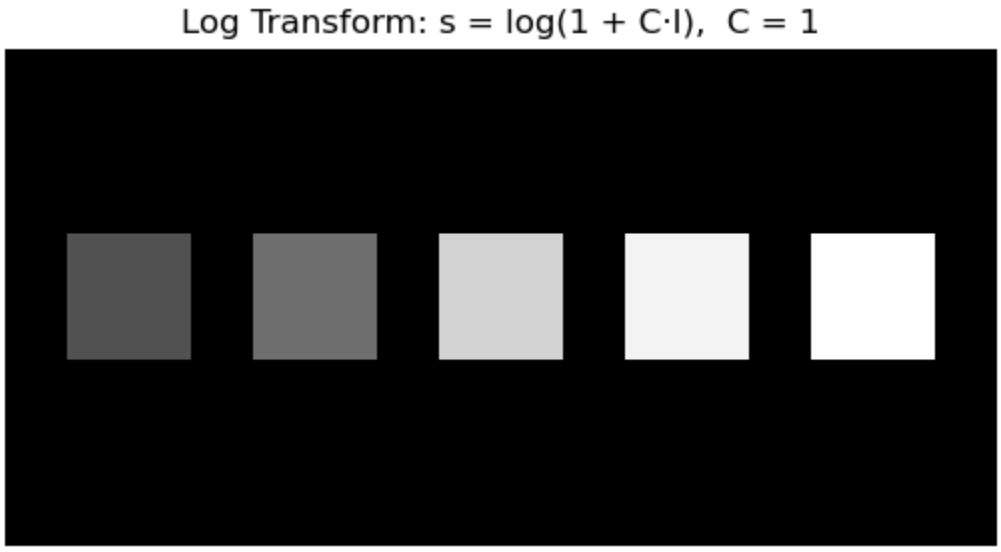

```
Saved: MaxContrast.png
```



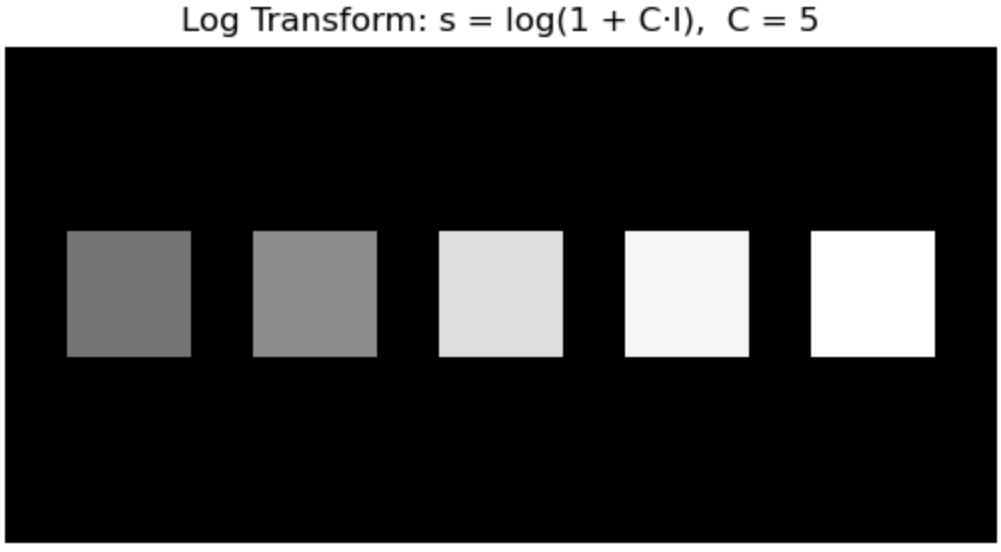

```
Saved: Patches.png
```



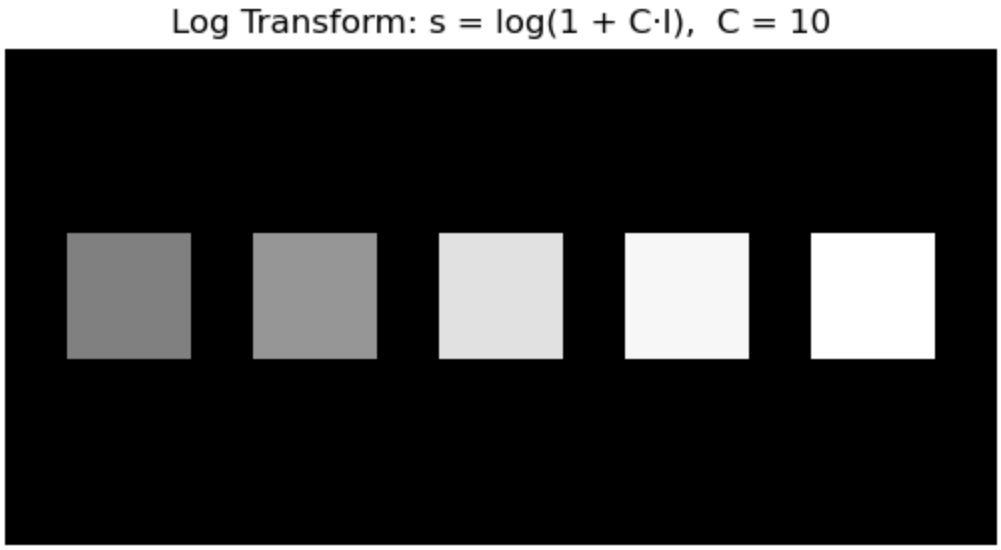

```
Saved: PartD_Log_C1.png
```



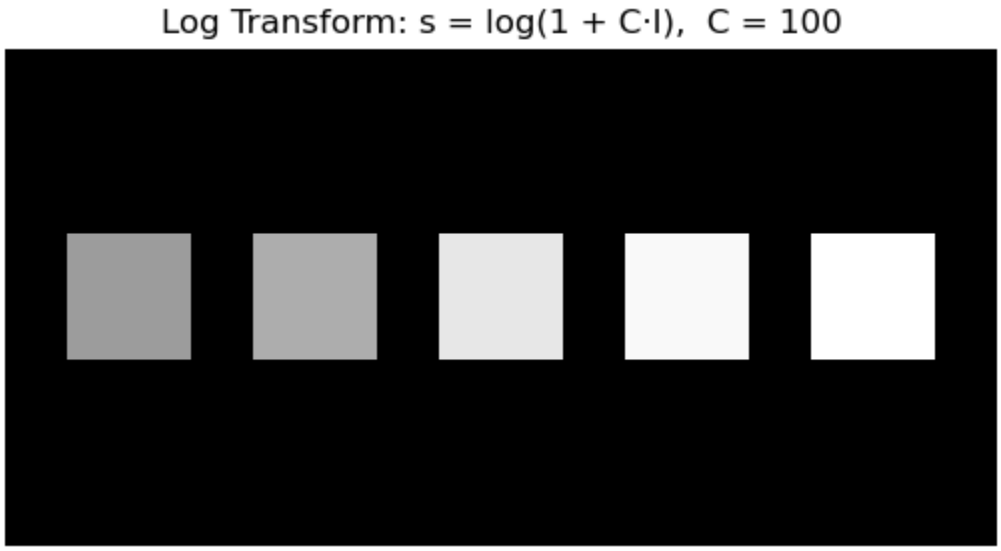

```
Saved: PartD_Log_C5.png
```



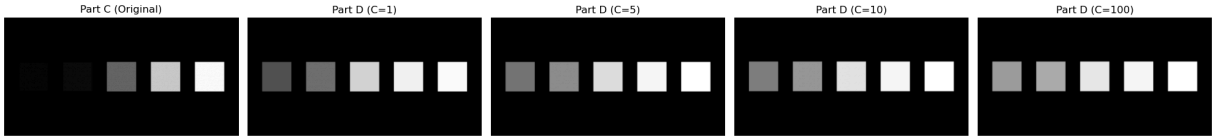

```
Saved: PartD_Log_C10.png
```




```
Saved: PartD_Log_C100.png
```




```
Saved: vis_score.png
```



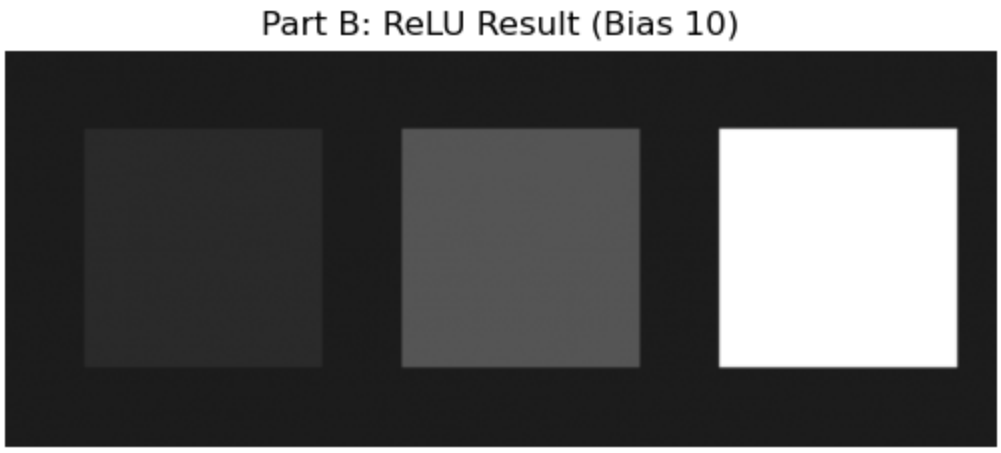
```
================================================================
                  MICHELSON CONTRAST ANALYSIS
================================================================
Row \ Patch  |    5    |   10    |   100   |   200   |   250
----------------------------------------------------------------
Original     |  1.000  |  1.000  |  1.000  |  1.000  |  1.000  |
C = 1        |  1.000  |  1.000  |  1.000  |  1.000  |  1.000  |
C = 5        |  1.000  |  1.000  |  1.000  |  1.000  |  1.000  |
C = 10       |  1.000  |  1.000  |  1.000  |  1.000  |  1.000  |
C = 100      |  1.000  |  1.000  |  1.000  |  1.000  |  1.000  |
----------------------------------------------------------------

================================================================
            NORMALIZED LOG-TRANSFORMED PIXEL VALUES
================================================================
Row \ Patch  |    5    |   10    |   100   |   200   |   250
----------------------------------------------------------------
C = 1        |  82.0   |  110.0  |  212.0  |  244.0  |  254.0  |
C = 5        |  116.0  |  140.0  |  222.0  |  247.0  |  255.0  |
C = 10       |  128.0  |  150.0  |  225.0  |  247.0  |  255.0  |
C = 100      |  156.0  |  173.0  |  231.0  |  249.0  |  255.0  |
----------------------------------------------------------------
```

Problem 3

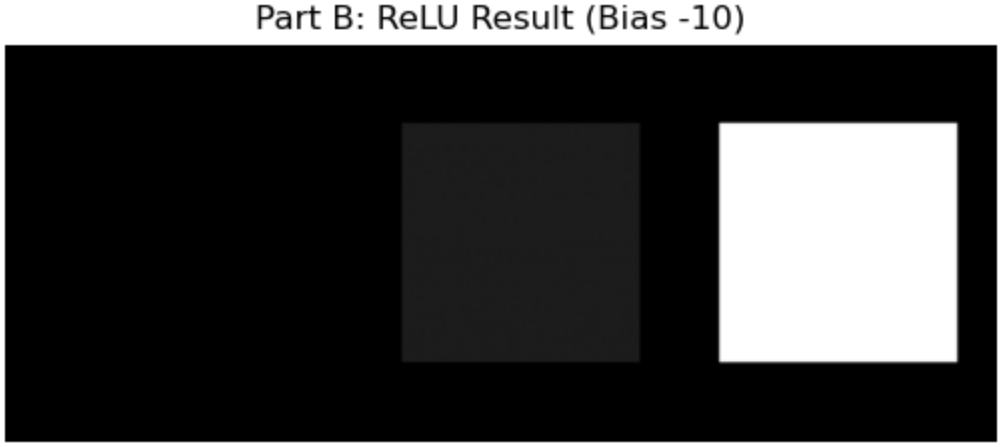
In [3]:
```python
import numpy as np
import matplotlib.pyplot as plt

def solve_problem_3():


    # Base Image
    img = np.zeros((100, 250))


    # intensity 5
    img[20:80, 20:80] = 5

    # intensity 20
    img[20:80, 100:160] = 20

    # intensity 100
    img[20:80, 180:240] = 100

    # Save
    plt.figure()
    plt.imshow(img, cmap="gray", vmin=0, vmax=100)
    plt.title("Part A: Original Image (Patches 5, 20 & 100)")
    plt.axis("off")
    plt.savefig("Problem3_A_Original.png", bbox_inches="tight")
    plt.show()
    print("Saved: Problem3_A_Original.png")
```

```
    #  ReLU with Proper Bias
    for bias_relu in [10, -10, -30]:
        img_relu = np.maximum(0, img + bias_relu)

        plt.figure()
        plt.imshow(img_relu, cmap="gray", vmin=0, vmax=90)
        plt.title(f"Part B: ReLU Result (Bias {bias_relu})")
        plt.axis("off")

        filename = f"Problem3_B_ReLU_Bias{bias_relu}.png"
        plt.savefig(filename, bbox_inches="tight")

        plt.show()
        print("Saved:", filename)



    # Sigmoid with Proper Bias
    for bias_sig in [ -10, -30]:

        img_sigmoid = 1 / (1 + np.exp(-(img + bias_sig)))

        plt.figure()
        plt.imshow(img_sigmoid, cmap="gray", vmin=0, vmax=1)
        plt.title(f"Part C: Sigmoid Result (Bias {bias_sig})")
        plt.axis("off")

        filename = f"Problem3_C_Sigmoid_Bias{bias_sig}.png"
        plt.savefig(filename, bbox_inches="tight")

        plt.show()
        print("Saved:", filename)




if __name__ == "__main__":
    solve_problem_3()
```

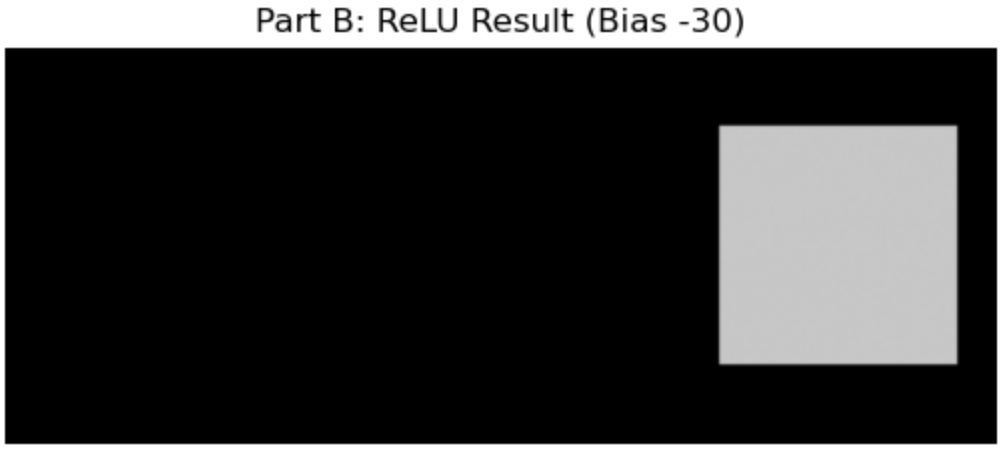

```
Saved: Problem3_A_Original.png
```



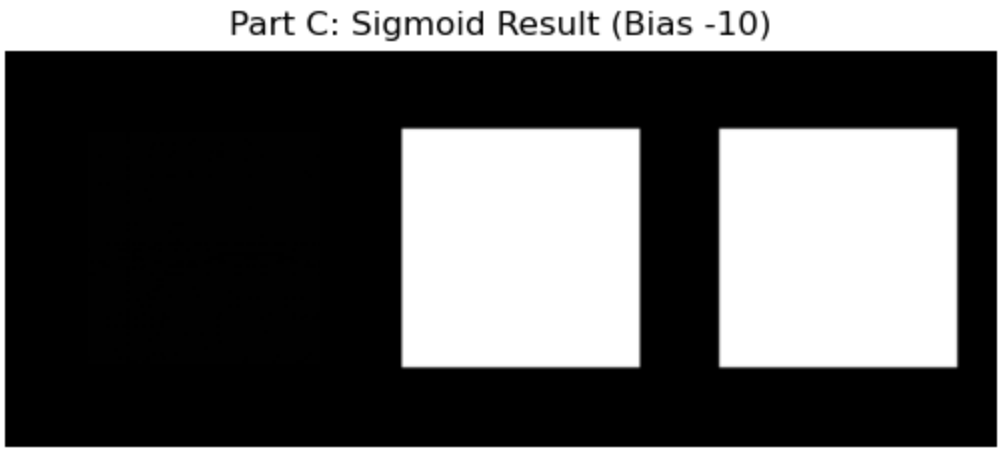

```
Saved: Problem3_B_ReLU_Bias10.png
```



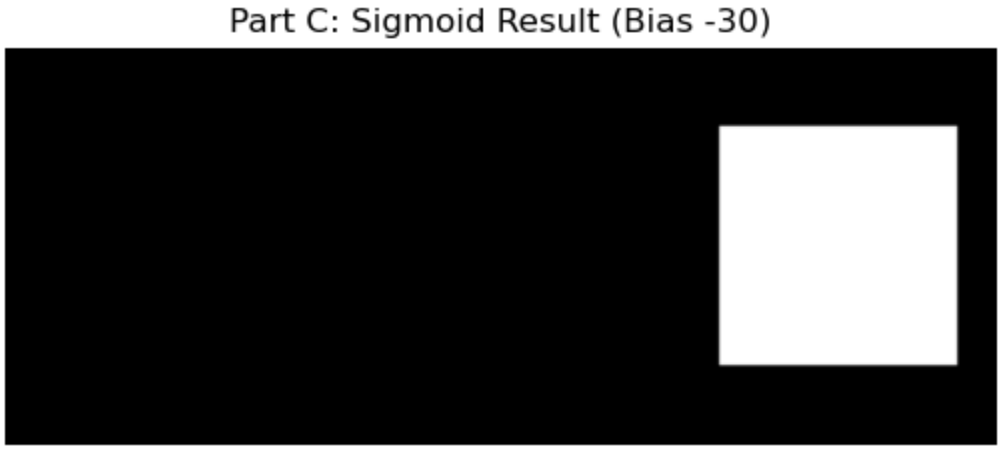

```
Saved: Problem3_B_ReLU_Bias-10.png
```




```
Saved: Problem3_B_ReLU_Bias-30.png
```




```
Saved: Problem3_C_Sigmoid_Bias-10.png
```




```
Saved: Problem3_C_Sigmoid_Bias-30.png
```

Problem 4

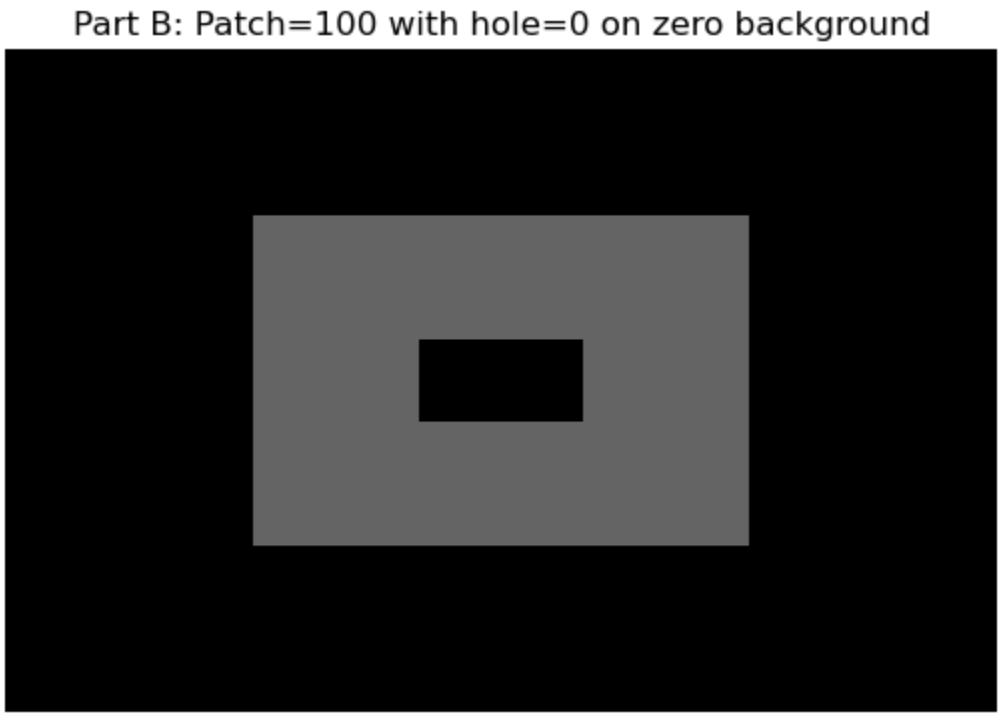
In [4]:
```
import numpy as np
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt


# Part B
def im_show(img, title="", filename=None):
    plt.figure()
    plt.imshow(img, cmap="gray", vmin=0, vmax=255)
    plt.title(title)
    plt.axis("off")
    if filename is not None:
        plt.savefig(filename, bbox_inches="tight")
        print(f"Saved: {filename}")
    plt.show()


rows, cols = 80, 120
img = np.zeros((rows, cols), dtype=np.float32)

# patch value = 100
```

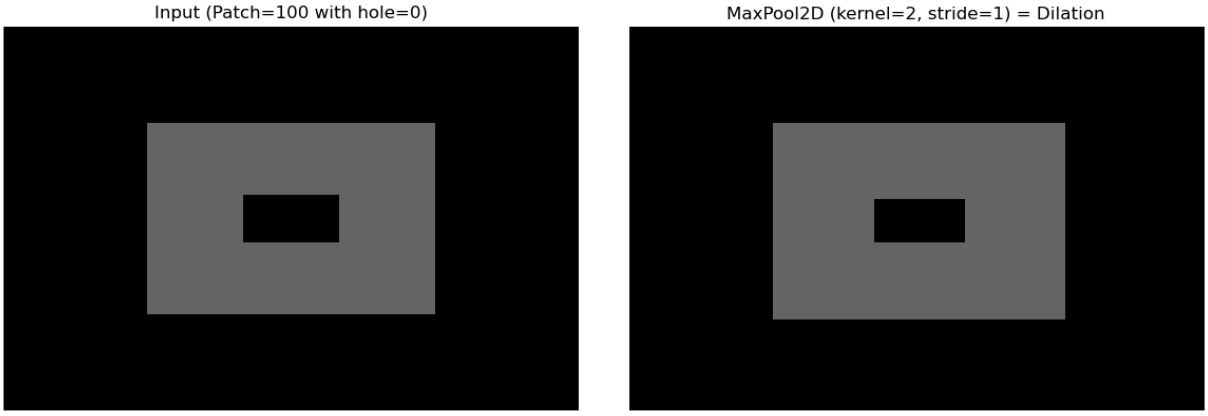
```
img[20:60, 30:90] = 100

# rectangle hole
img[35:45, 50:70] = 0 #h=20, w=10

# Original image
im_show(
    img,
    "Part B: Patch=100 with hole=0 on zero background",
    "Problem4_B.png"
)


patch_r0, patch_r1 = 20, 60
patch_c0, patch_c1 = 30, 90

patch_before = img[patch_r0:patch_r1, patch_c0:patch_c1] #hole size before dialatio
mask_before = (patch_before == 0)

rr, cc = np.where(mask_before)
hole_h_before = rr.max() - rr.min() + 1
hole_w_before = cc.max() - cc.min() + 1



# Part D

x = torch.tensor(img).unsqueeze(0).unsqueeze(0)   # shape: (1, 1, H, W)


# pad by 1 pixel on all sides
x_pad = F.pad(x, (1, 1, 1, 1), mode="constant", value=0)

# max pooling (dilation) on padded image
dilated_pad = F.max_pool2d(x_pad, kernel_size=2, stride=1)


H, W = x.shape[-2], x.shape[-1]
dilated = dilated_pad[:, :, 0:H, 0:W]   # now same shape as input

dilated_np = dilated.squeeze().numpy()
patch_after = dilated_np[patch_r0:patch_r1, patch_c0:patch_c1] #hole size before di
mask_after = (patch_after == 0)




# Visualization of input vs output

plt.figure(figsize=(12,4))

plt.subplot(1,2,1)
plt.imshow(img, cmap="gray", vmin=0, vmax=255)
plt.title("Input (Patch=100 with hole=0)")
plt.axis("off")
```

```
plt.subplot(1,2,2)
plt.imshow(dilated_np, cmap="gray", vmin=0, vmax=255)
plt.title("MaxPool2D (kernel=2, stride=1) = Dilation")
plt.axis("off")

plt.tight_layout()
plt.savefig("Problem4_D.png", bbox_inches="tight")
plt.show()

print("Input shape :", x.shape)
print("Output shape:", dilated.shape)
print(f"Hole size before dilation: height={hole_h_before}, width={hole_w_before}")
if np.any(mask_after):
    rr2, cc2 = np.where(mask_after)
    hole_h_after = rr2.max() - rr2.min() + 1
    hole_w_after = cc2.max() - cc2.min() + 1
    print(f"Hole size after dilation (k=2): height={hole_h_after}, width={hole_w_af
else:
    print("Hole size after dilation (k=2): eliminated")


# Part E verification

kernel_size_E = 11
p = kernel_size_E // 2  # 5 for k=11

x_pad_E = F.pad(x, (p, p, p, p), mode="constant", value=0)
dilated_pad_E = F.max_pool2d(x_pad_E, kernel_size=kernel_size_E, stride=1)

H, W = x.shape[-2], x.shape[-1]
dilated_E = dilated_pad_E[:, :, 0:H, 0:W]
dilated_np_E = dilated_E.squeeze().numpy()

#visualization
plt.figure(figsize=(6,4))
plt.imshow(dilated_np_E, cmap="gray", vmin=0, vmax=255)
plt.title(f"Part E: Hole Eliminated (Kernel={kernel_size_E})")
plt.axis("off")
plt.savefig("Problem4_E.png", bbox_inches="tight")
plt.show()

# Part H
# Erode(I) = - MaxPool( -I )
kernel_size_H = 11
pH = kernel_size_H // 2

# converting to torch, applying erosion
x_pad_H = F.pad(-x, (pH, pH, pH, pH), mode="constant", value=0)
eroded_pad = F.max_pool2d(x_pad_H, kernel_size=kernel_size_H, stride=1)
Hh, Wh = x.shape[-2], x.shape[-1]
eroded = -eroded_pad[:, :, 0:Hh, 0:Wh]

eroded_np = eroded.squeeze().numpy()

#visualization
plt.figure(figsize=(6,4))
```

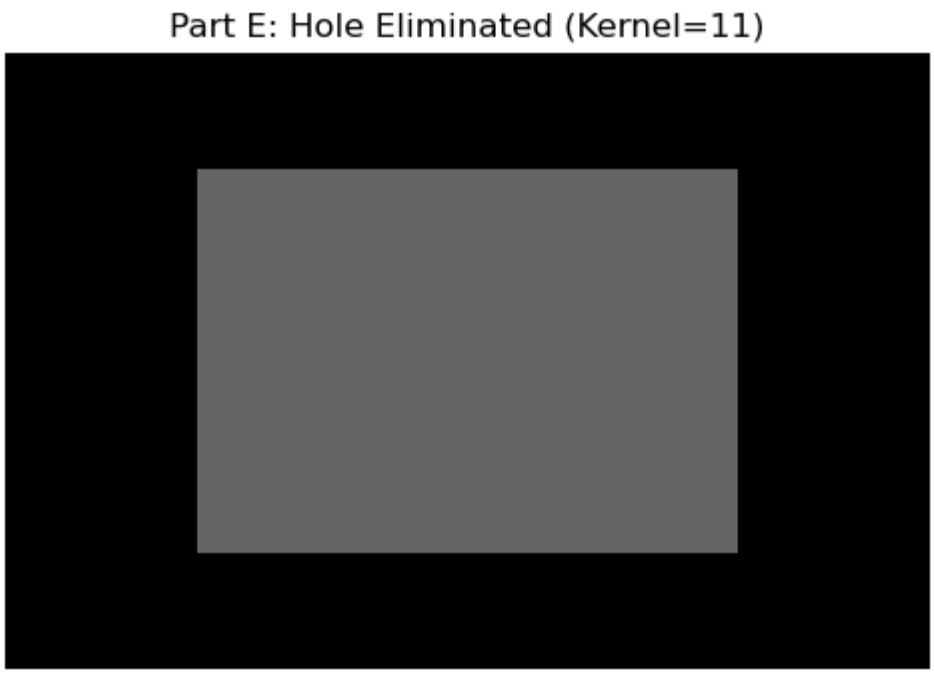
```
plt.imshow(eroded_np, cmap="gray", vmin=0, vmax=255)
plt.title(f"Part H: Erosion via -MaxPool(-I) (Kernel={kernel_size_H})")
plt.axis("off")
plt.savefig("Problem4_H.png", bbox_inches="tight")
plt.show()
```

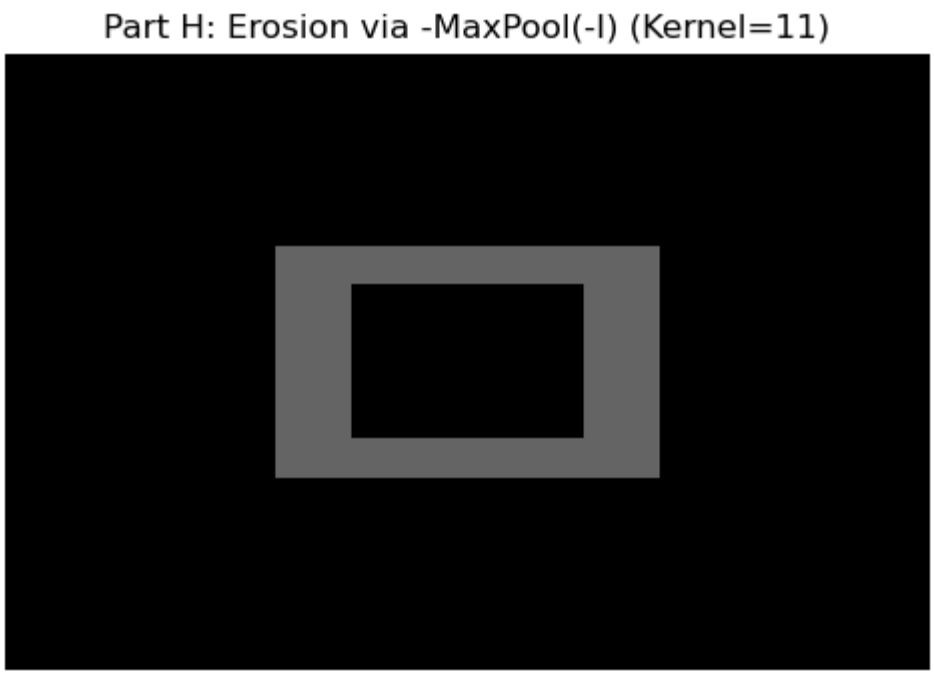
```
Saved: Problem4_B.png
```






```
Input shape : torch.Size([1, 1, 80, 120])
Output shape: torch.Size([1, 1, 80, 120])
Hole size before dilation: height=10, width=20
Hole size after dilation (k=2): height=9, width=19
```





Problem 6

In [14]:
```
import pathlib

p = pathlib.Path("AOLME.py")
txt = p.read_text(encoding="utf-8")

txt = txt.replace(
    "from google.colab.patches import cv2_imshow",
    "try:\n    from google.colab.patches import cv2_imshow  # Colab only\nexcept Ex
)
```

```
p.write_text(txt, encoding="utf-8")
print("Patched AOLME.py to work outside Colab.")
```

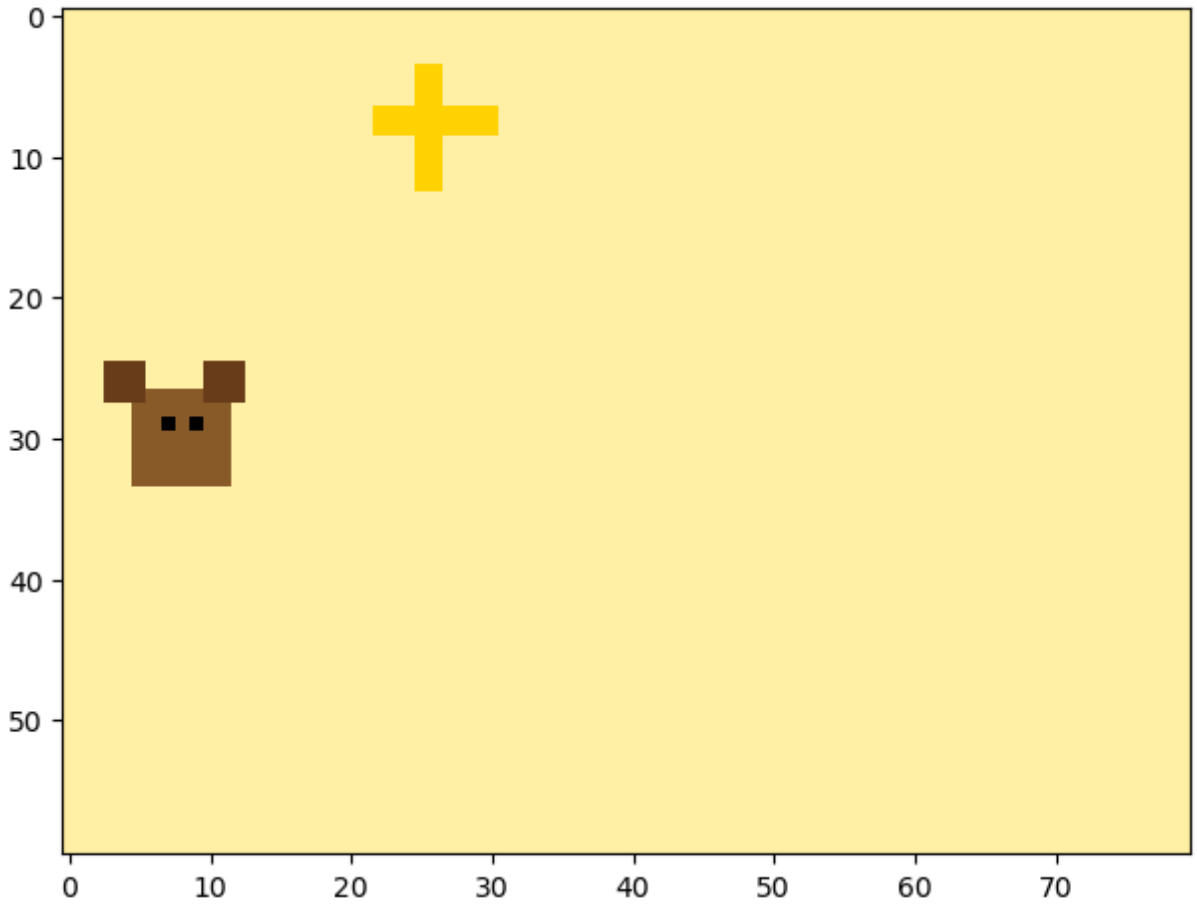
```
Patched AOLME.py to work outside Colab.
```

In [18]:
```
from AOLME import *
!pip install imageio
```

```
Collecting imageio
  Downloading imageio-2.37.2-py3-none-any.whl.metadata (9.7 kB)
Requirement already satisfied: numpy in c:\users\upama\appdata\local\anaconda3\envs
\pytorch\lib\site-packages (from imageio) (1.26.4)
Requirement already satisfied: pillow>=8.3.2 in c:\users\upama\appdata\local\anacond
a3\envs\pytorch\lib\site-packages (from imageio) (12.0.0)
Downloading imageio-2.37.2-py3-none-any.whl (317 kB)
Installing collected packages: imageio
Successfully installed imageio-2.37.2
```

In [20]:
```
from AOLME import *
import numpy as np
from IPython.display import HTML


# Video setup

rows, cols = 60, 80
num_frames = 20
fps = 6

# Colors (hex RGB)
butter = "fff2a8"   # butter yellow background
bear   = "8b5a2b"   # brown teddy
bear2  = "6b3f1d"   # darker brown details
banana = "ffd400"   # banana yellow
black  = "000000"   # eyes


# Helpers: draw objects

def draw_teddy(frame, cx, cy):
    """
    Simple teddy icon: head: 7x7 square, ears: two 3x3 squares, eyes: 1-pixel dots
    """
    # head
    y0, y1 = max(0, cy-3), min(rows, cy+4)
    x0, x1 = max(0, cx-3), min(cols, cx+4)
    frame[y0:y1, x0:x1] = bear

    # ears
    frame[max(0, cy-5):min(rows, cy-2), max(0, cx-5):min(cols, cx-2)] = bear2
    frame[max(0, cy-5):min(rows, cy-2), max(0, cx+2):min(cols, cx+5)] = bear2

    # eyes
    ey = cy-1
    for ex in (cx-1, cx+1):
        if 0 <= ey < rows and 0 <= ex < cols:
            frame[ey, ex] = black
```

```
# Banana drawn like a plus sign:vertical stroke + horizontal stroke, thickness cont
def draw_banana_plus(frame, cx, cy, arm=4, thickness=2, color=banana):

    # vertical stroke
    y0, y1 = max(0, cy-arm), min(rows, cy+arm+1)
    for dx in range(-(thickness//2), thickness - (thickness//2)):
        x = cx + dx
        if 0 <= x < cols:
            frame[y0:y1, x] = color

    # horizontal stroke
    x0, x1 = max(0, cx-arm), min(cols, cx+arm+1)
    for dy in range(-(thickness//2), thickness - (thickness//2)):
        y = cy + dy
        if 0 <= y < rows:
            frame[y, x0:x1] = color

# Build frames

frame_list = []

# Teddy moves left to right
teddy_y = rows // 2
teddy_x_start, teddy_x_end = 8, cols - 9

# Banana-plus moves top to bottom with wiggle
banana_y_start, banana_y_end = 8, rows - 9
banana_x_base = cols // 3
wiggle_amp = 8  # how far it wiggles left-right

# teddy bounces vertically like a sine wave
teddy_bounce_amp = 10  # bounce height in pixels

for t in range(num_frames):
    frame = np.full((rows, cols), butter)

    # Teddy position (linear motion and sine-wave bounce)
    teddy_x = int(teddy_x_start + (teddy_x_end - teddy_x_start) * t / (num_frames -
    bounce = int(teddy_bounce_amp * np.sin(2 * np.pi * t / (num_frames - 1)))
    teddy_y_bounce = int(np.clip(teddy_y + bounce, 6, rows - 7))
    draw_teddy(frame, teddy_x, teddy_y_bounce)

    # Banana-plus position (downward motion and sine wiggle)
    banana_y = int(banana_y_start + (banana_y_end - banana_y_start) * t / (num_fram
    wiggle = int(wiggle_amp * np.sin(2 * np.pi * t / (num_frames - 1)))
    banana_x = int(np.clip(banana_x_base + wiggle, 6, cols - 7))
    draw_banana_plus(frame, banana_x, banana_y, arm=4, thickness=2, color=banana)

    frame_list.append(frame)


# Play video

play_video = vid_show(frame_list, fps)
HTML(play_video.to_html5_video())
```


0
10
20
30
40
50
0
10
20
30
40
50
60
70

Out[20]:

<Figure size 640x480 with 0 Axes>

In [ ]: