Mansoura University
Faculty of Computers and Information
Department of Computer Science
Second semester 2024-2025

Digital Image Processing

Aya Ayad

OUTLINE

- Image crop
- Image flip
- Image rotation
- plotting
- Image operation
- Addition
- Subtraction

CROP

- Since OpenCV stores images as NumPy arrays, it can leverage array slicing to extract the desired region of interest (ROI).
- **Steps:**

1. **Load the image:** Use cv2.imread to read your image into a NumPy array.
2. **Define crop coordinates:** Specify the starting and ending pixel coordinates for the ROI you want to crop.
3. **Perform slicing:** Use NumPy array slicing syntax on the image array to extract the desired ROI.
4. **image[rows (height), cols (width)]** → image[rows_start:rows_end, cols_start:cols_end]

```
cropped = img[10:200, 100:500]
                    ropped)
```

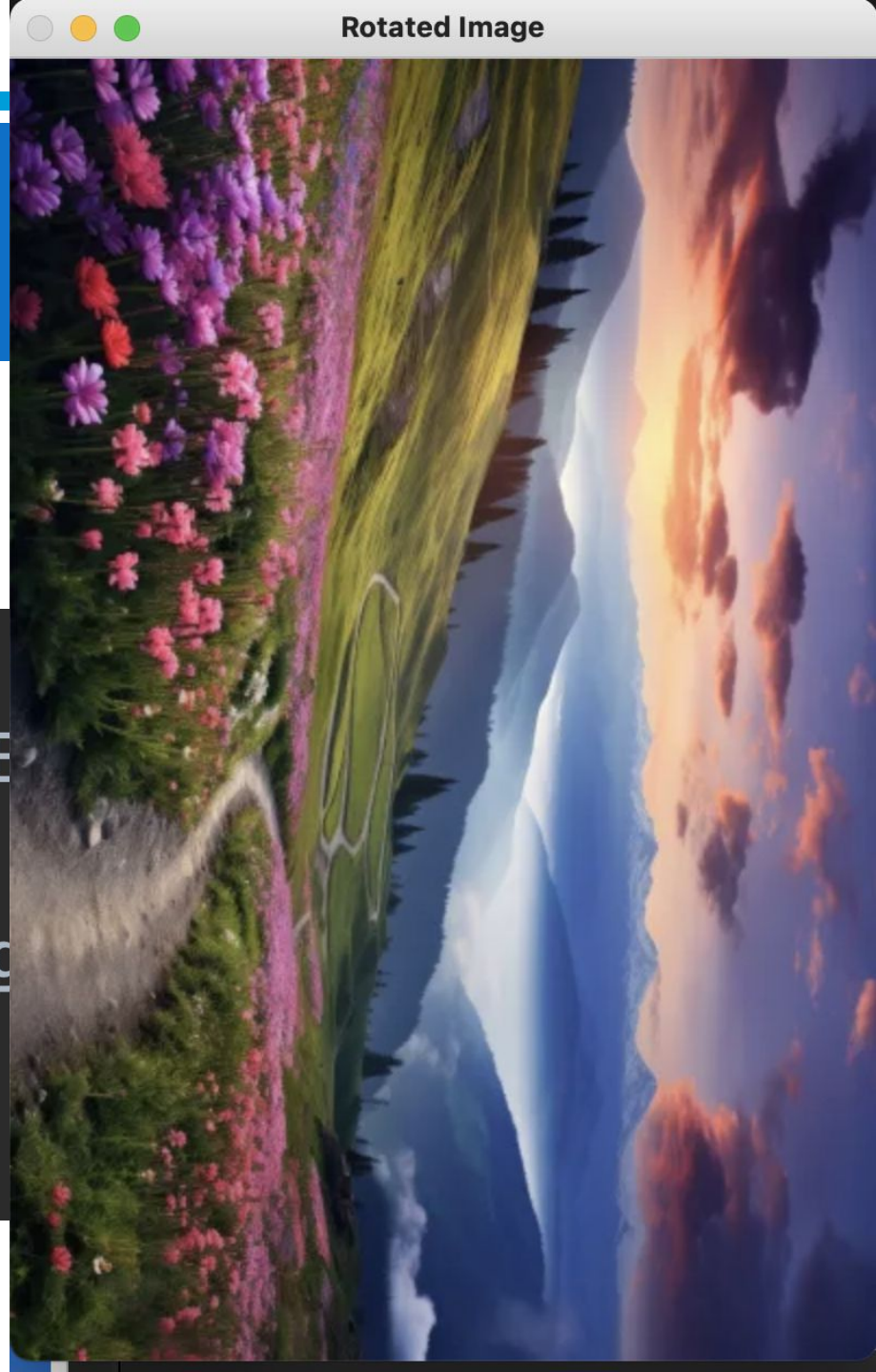
ROTATE

cv2.rotate(src, rotate_code)

- **Parameters:**

1. **src:** The input image to be rotated.
2. **rotate_code:** An integer specifying the rotation direction:
   a) cv2.ROTATE_90_CLOCKWISE: Rotate clockwise by 90 degrees.
   b) cv2.ROTATE_90_COUNTERCLOCKWISE: Rotate counter-clockwise by 90 degrees.(270)
   c) cv2.ROTATE_180: Rotate 180 degrees.

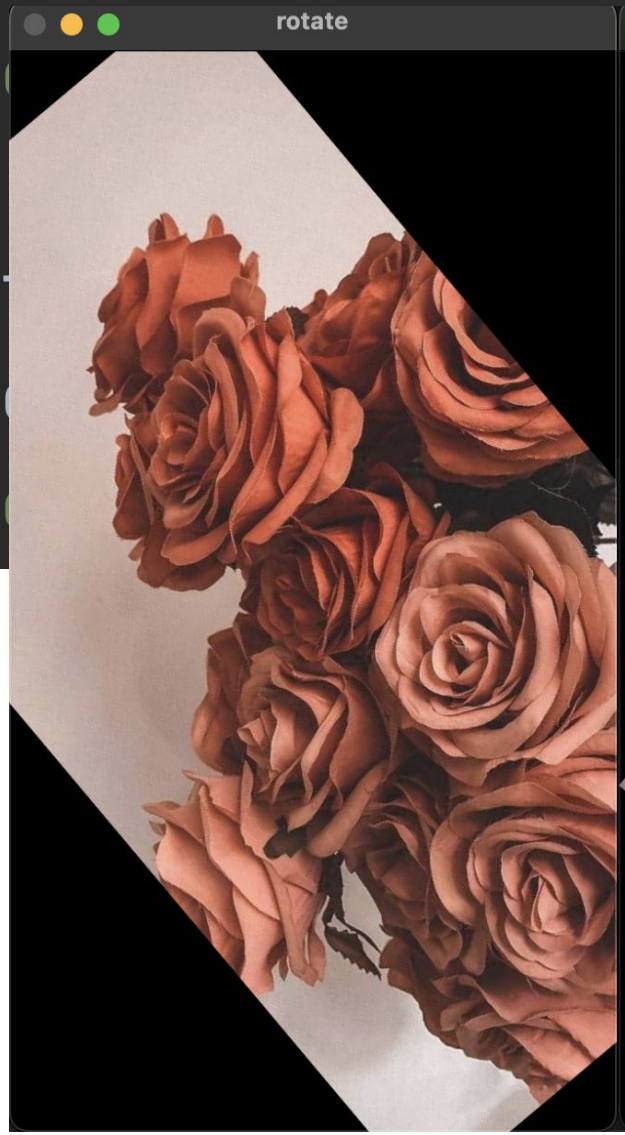
```
rotated_img = cv2.rotate(img,
                         cv2.ROTATE_90_CLOCKWISE

cv2.imshow("Rotated Image", rotated_img
cv2.waitKey(0)
cv2.destroyAllWindows()
```

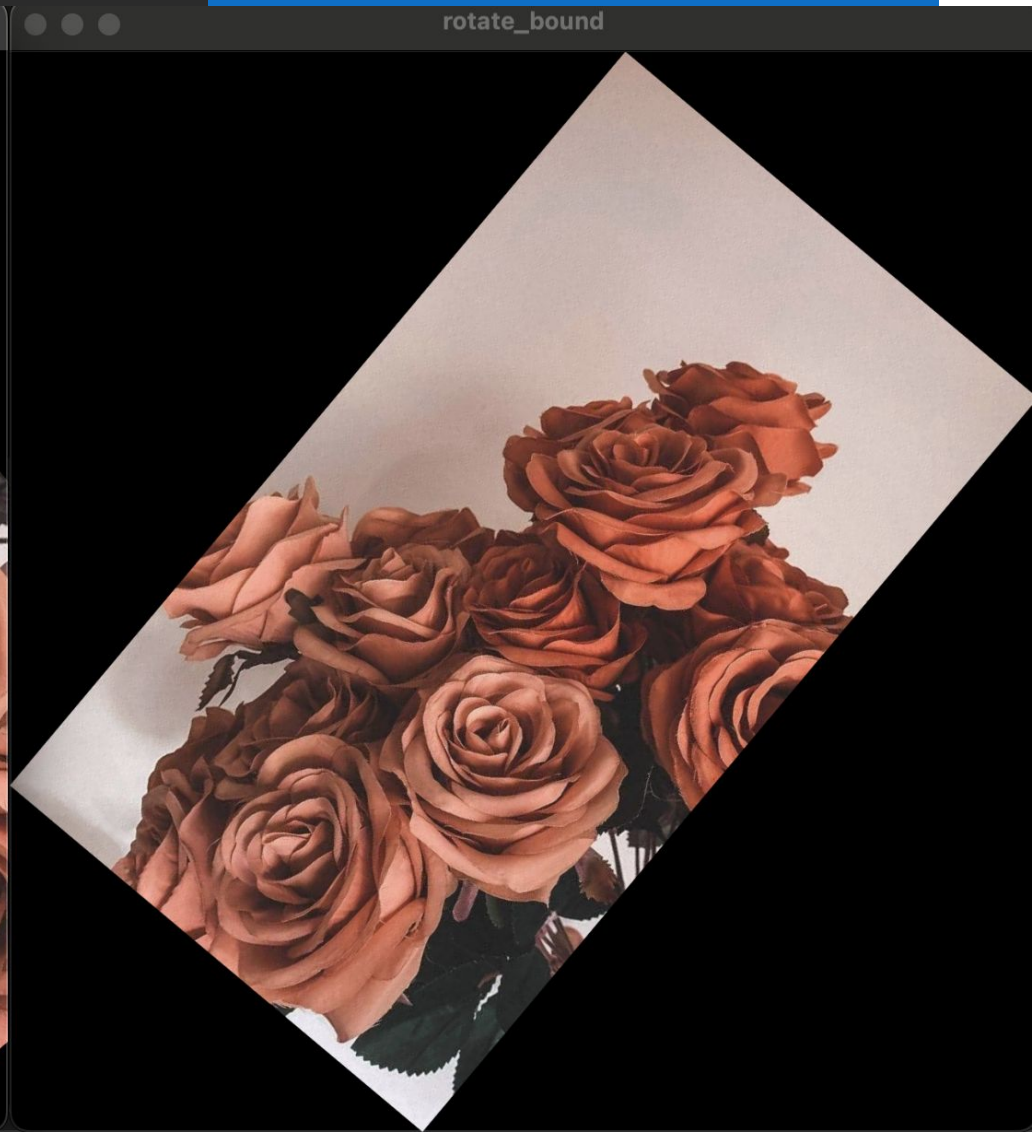


IMUTILS FOR ROTATION

- Since OpenCV has three options for rotations, another library provides more flexibility: the imutils library.
  - Rotated_img=imutils.rotate(source, angle)
- However, imutils.rotate() may crop parts of the image if the rotation extends beyond the original dimensions.
- To prevent cropping, use imutils.rotate_bound(), which adjusts the image size to fit the entire rotated content:
  - Rotated_img = imutils.rotate_bound(source, angle)

```
r1=imutils.rotate(imgg1, 40)
print(r1.shape)
cv2.imshow("r
cv2.waitKey()
r2=imutils.ro
print(r2.shap
cv2.imshow("r
```





FLIP

cv2.flip(src, flip_code)

flip_code: An integer code specifying the flipping direction:

Flipping Horizontally (flip code = 1):

- Swaps the left and right sides of the image. Pixels on the left side of the original image move to the corresponding positions on the right side in the flipped image, and vice versa.

Flipping Vertically (flip code = 0):

- Swaps the top and bottom sides of the image. Pixels on the top of the original image move to the corresponding positions on the bottom in the flipped image, and vice versa.

Mirroring the Image (flip code = -1):

- Combines both horizontal and vertical flips. - Essentially creates a mirrored image where both left-right and top-bottom are swapped.

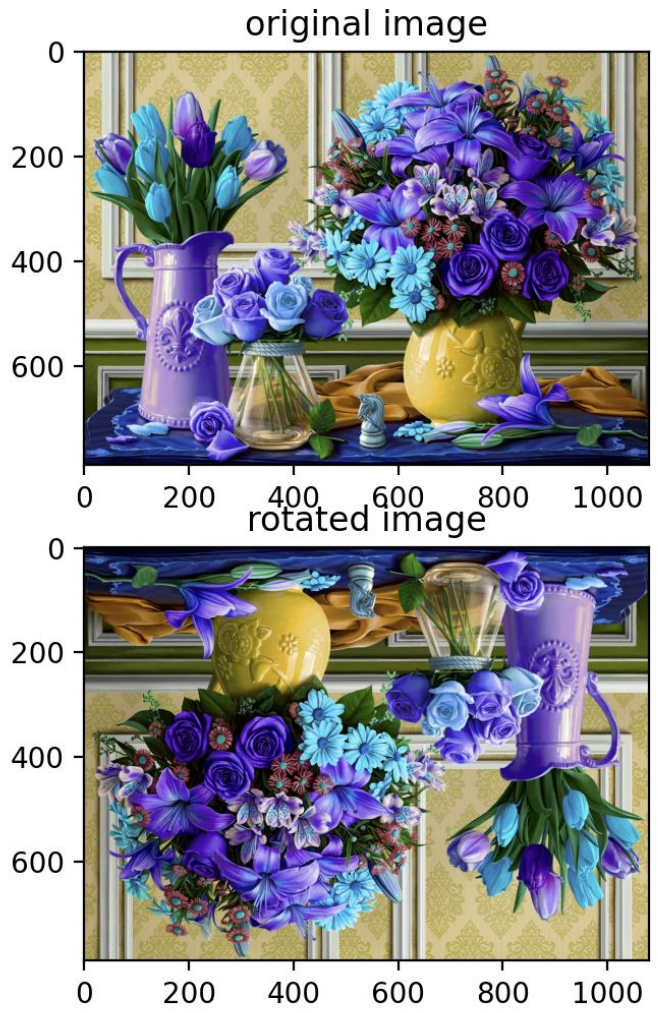
```
imgg1=cv2.imread("flower.png")
# Flip horizontally (left-right)
flipped_horizontally = cv2.flip(imgg1, 1)
# Flip vertically (up-down)
flipped_vertically = cv2.flip(imgg1, 0)
# Flip both horizontally and vertically (180-degree rotation)
flipped_both = cv2.flip(imgg1, -1)
# Show images
cv2.imshow("Original", imgg1)
cv2.waitKey()
cv2.imshow("Flipped Horizontally", flipped_horizontally)
cv2.waitKey()
cv2.imshow("Flipped Vertically", flipped_vertically)
cv2.waitKey()
cv2.imshow("Flipped Both", flipped_both)
cv2.waitKey()
```


Original
Flipped Vertically
Flipped Horizontally
Flipped Both

PLOTTING

- **OpenCV does not provide a built-in option to display multiple images in a single window.** To achieve this, we use **Matplotlib's subplot feature** to arrange multiple images in a grid format.
  - plt.subplot(n_rows, n_columns, position)
    - plt.imshow(image)
      - plt.title(" ")
      - plt.show()
- **Note:** Matplotlib processes images in **RGB mode**, whereas OpenCV uses **BGR mode** by default. If displaying OpenCV images with Matplotlib, convert them from **BGR to RGB**

```
plt.subplot(2, 1, 1)
plt.imshow(original)
plt.title("original image")
plt.subplot(2, 1, 2)
plt.imshow(rot)
plt.title("rotated image")
plt.show()
```

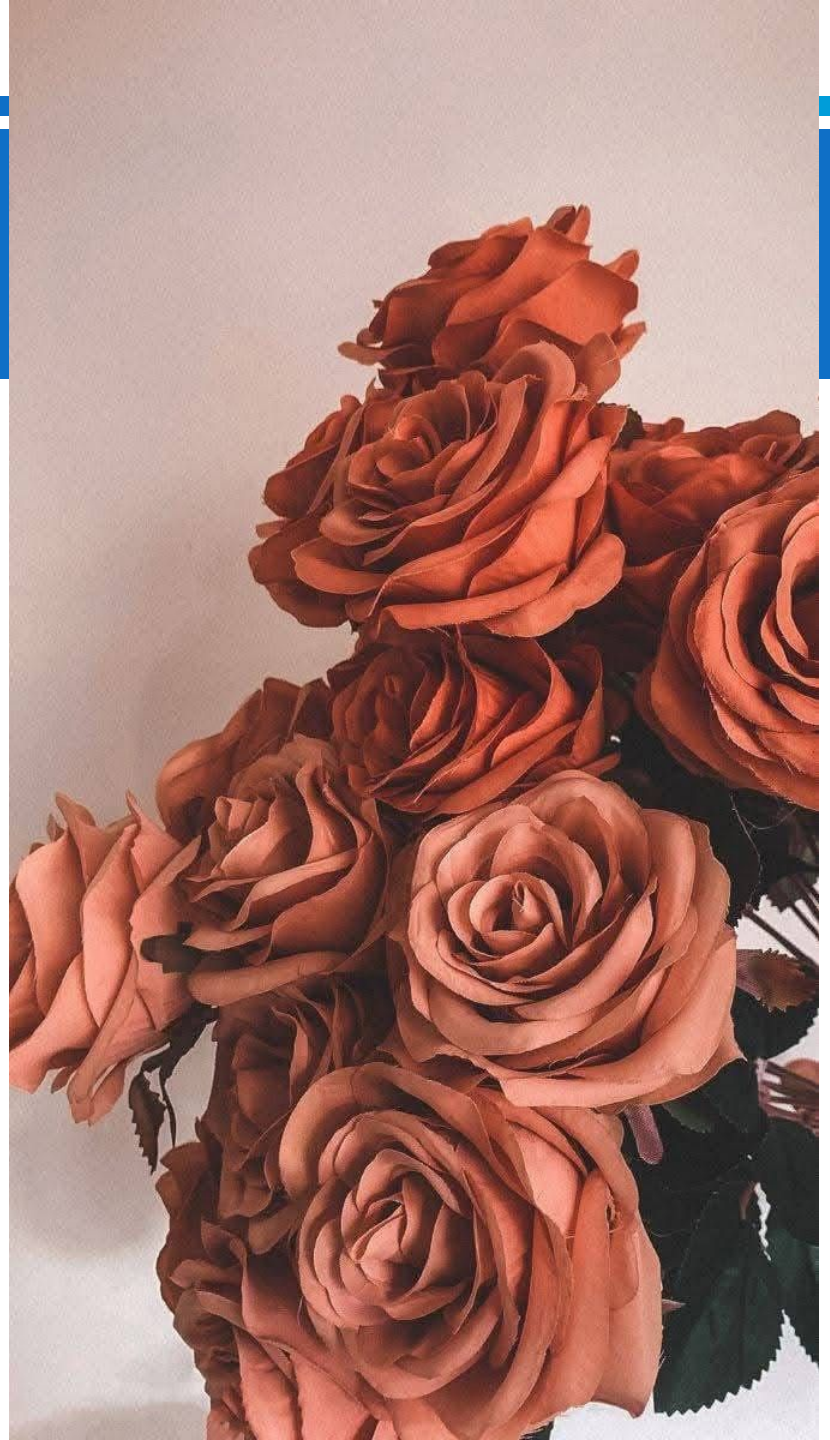


OPEN-CV ARITHMETIC OPERATIONS ON IMAGES

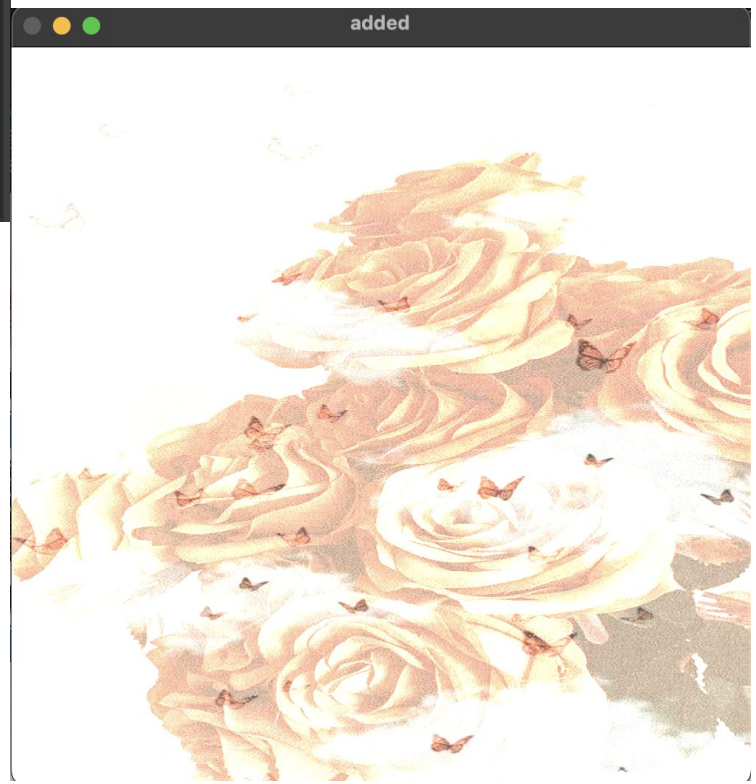
- OpenCV provides various arithmetic operations that allow you to manipulate images by adding, subtracting, multiplying, and blending pixel values.
- These operations are applied **pixel by pixel**, meaning the value of each pixel in the output image is determined solely by the corresponding pixels in the input images.
- As a result, the input images **must typically be of the same size and type**.
- When adding a constant offset to an image, one of the input "images" may be a **scalar (constant) value** instead.

ADDITION

- In its most basic form, the addition operator takes two identically sized images as input and produces a third image of the same size, where each pixel value is the **sum of the corresponding pixel values** from the two input images.
- More advanced implementations allow for the combination of multiple images in a single operation.
- A common variation of this operator enables the **addition of a specified constant** to each pixel.
- Using the function cv2.add(), we can add two images together.
  Unlike NumPy's raw addition (+), which allows overflow, cv2.add() **performs saturated addition**, ensuring that pixel values are clipped within the valid range [0, 255].



```python
imgg1 = cv2.imread("img1.jpg")
imgg2 = cv2.imread("img2.jpg")
img1 = cv2.resize(imgg1, (1000, 100
img2 = cv2.resize(imgg2, (1000, 100
out_add = cv2.add(img1, img2)
cv2.imshow("added", out_add)
cv2.waitKey()
```

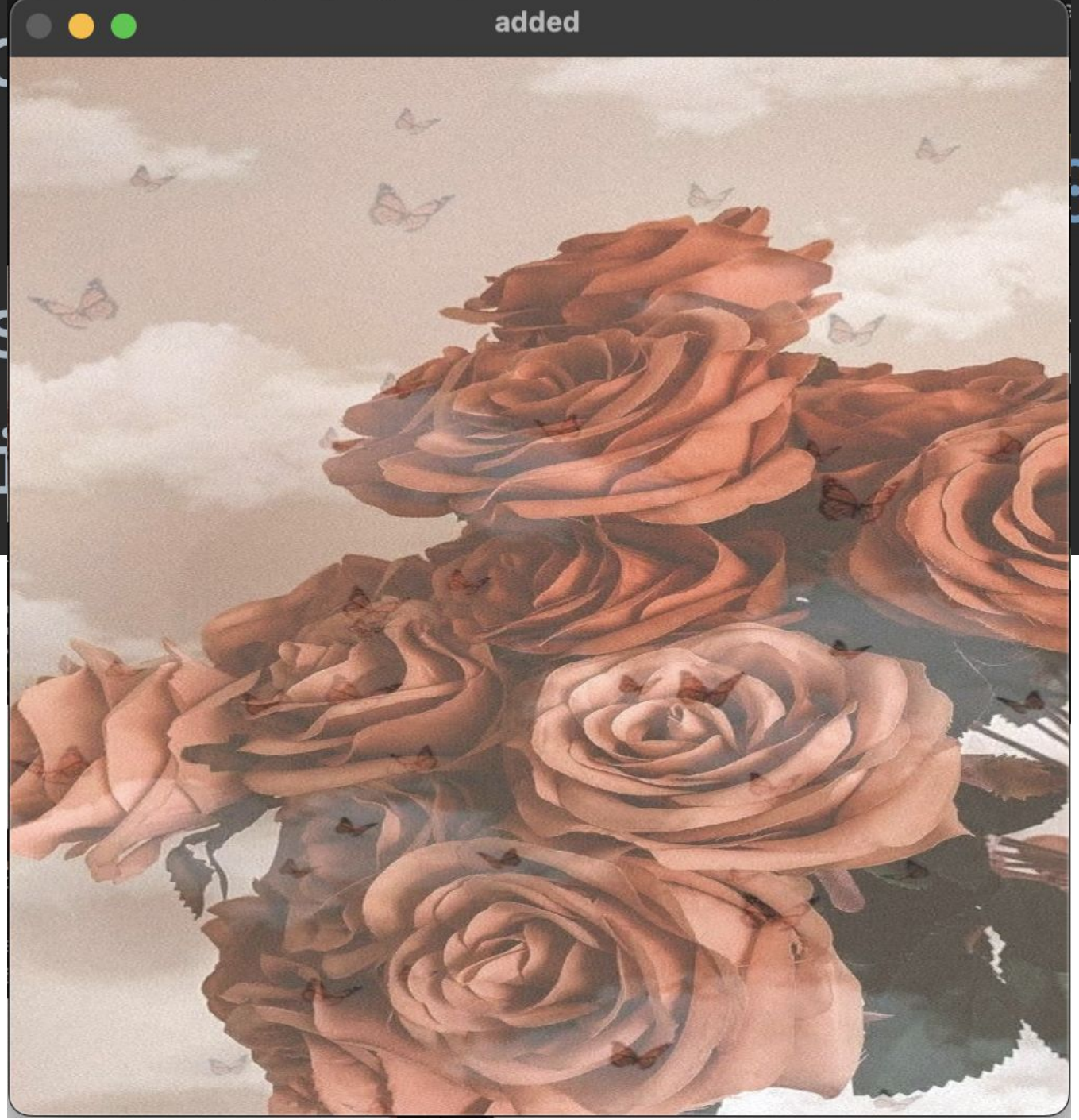


ADDITION(*ADDWEIGHTED*)

- However, directly adding pixel values is not always ideal. Instead, we use cv2.addWeighted(), which blends two images by assigning specific weights to each.
- **Note:** Both input images must have the same shape and number of color channels.
  - cv2.addWeighted(image1, weight1, image2, weight2, gammaValue)
- **Parameters:**

1. **image1** – First input image (array).
2. **weight1** – Weight applied to the first image.
3. **image2** – Second input image (array).
4. **weight2** – Weight applied to the second image.
5. **gammaValue** – Scalar added to each sum, often used for brightness adjustment.

```
out = c                                    0.4,
                                           0.2)

cv2.ims
cv2.wai
```



SUBTRACTION

- The pixel subtraction operator takes two images as input and produces a third image, where each pixel value is the result of subtracting the corresponding pixel value of the second image from the first image.
- It is also common to use a single image as input and subtract a constant value from all pixel intensities.
- Most implementations clamp negative pixel values to zero.
  - ***cv2.subtract(image1, image2)***

```
sub = cv                                    2)
cv2.ims
cv2.wai
```