Experiment # 3

AIM: Display of an Image, Negative of an Image (Binary & Gray Scale)

Description:

Arithmetic Operations like Addition, Subtraction, and Bitwise Operations (AND, OR, NOT, XOR) can be applied to the input images. These operations can be helpful in enhancing the properties of the input images. The Image arithmetics are important for analyzing the input image properties. The operated images can be further used as an enhanced input image, and many more operations can be applied for clarifying, thresholding, dilating etc of the image.

Addition of Image:

We can add two images by using function **cv2.add()**. This directly adds up image pixels in the two images.

Syntax: cv2.add(img1, img2)
But adding the pixels is not an ideal situation. So, we use cv2.addweighted(). Remember, both images should be of equal size and depth.

Syntax*: cv2.addWeighted(img1, wt1, img2, wt2, gammaValue)*
Parameters*:*
img1*: First Input Image array(Single-channel, 8-bit or floating-point)*
wt1*: Weight of the first input image elements to be applied to the final image*
img2*: Second Input Image array(Single-channel, 8-bit or floating-point)*
wt2*: Weight of the second input image elements to be applied to the final image*
gammaValue*: Measurement of light*

Images used as Input:
Input Image1:

Input Image2:

Below is the code:

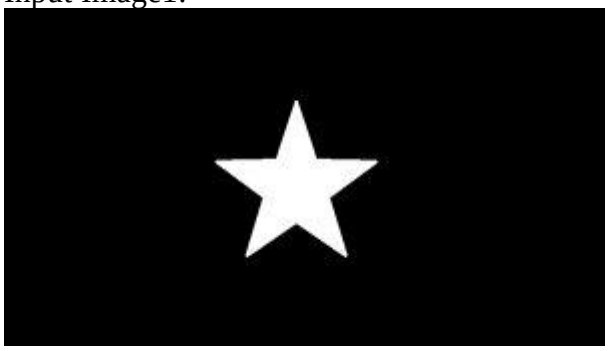
```
# Python program to illustrate
# arithmetic operation of
# addition of two images

# organizing imports
import cv2
import numpy as np

# path to input images are specified and
# images are loaded with imread command
image1 = cv2.imread('input1.jpg')
image2 = cv2.imread('input2.jpg')

# cv2.addWeighted is applied over the
# image inputs with applied parameters
weightedSum = cv2.addWeighted(image1, 0.5, image2, 0.4, 0)

# the window showing output image
# with the weighted sum
cv2.imshow('Weighted Image', weightedSum)

# De-allocate any associated memory usage
if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()
```

Output:

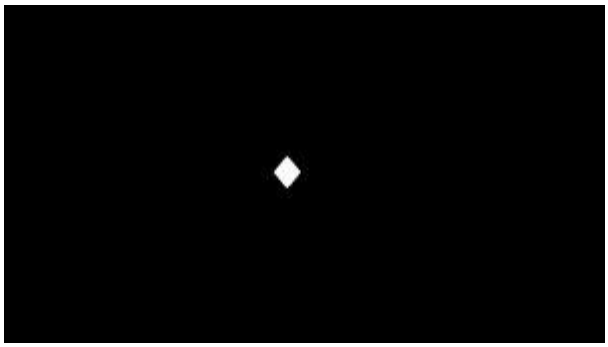
Subtraction of Image:

Just like addition, we can subtract the pixel values in two images and merge them with the help of cv2.subtract(). The images should be of equal size and depth.

Syntax: cv2.subtract(src1, src2)
Images used as Input:
Input Image1:



Input Image2:



```
# Python program to illustrate
# arithmetic operation of
# subtraction of pixels of two images

# organizing imports
import cv2
import numpy as np

# path to input images are specified and
# images are loaded with imread command
image1 = cv2.imread('input1.jpg')
image2 = cv2.imread('input2.jpg')

# cv2.subtract is applied over the
# image inputs with applied parameters
sub = cv2.subtract(image1, image2)
```

```
# the window showing output image
# with the subtracted image
cv2.imshow('Subtracted Image', sub)

# De-allocate any associated memory usage
if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()
```

II)

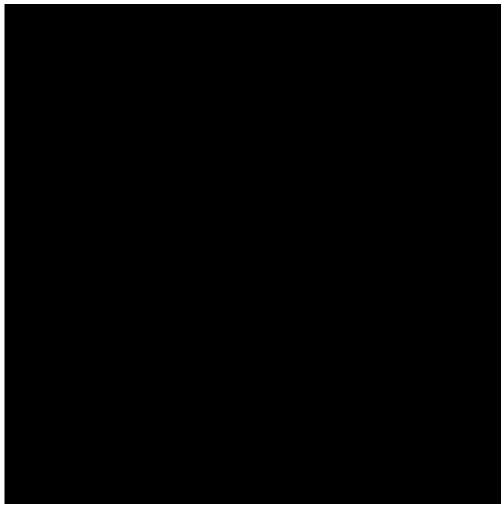
Bitwise operations are used in image manipulation and used for extracting essential parts in the image. In this article, Bitwise operations used are :

1. **AND**
2. **OR**
3. **XOR**
4. **NOT**

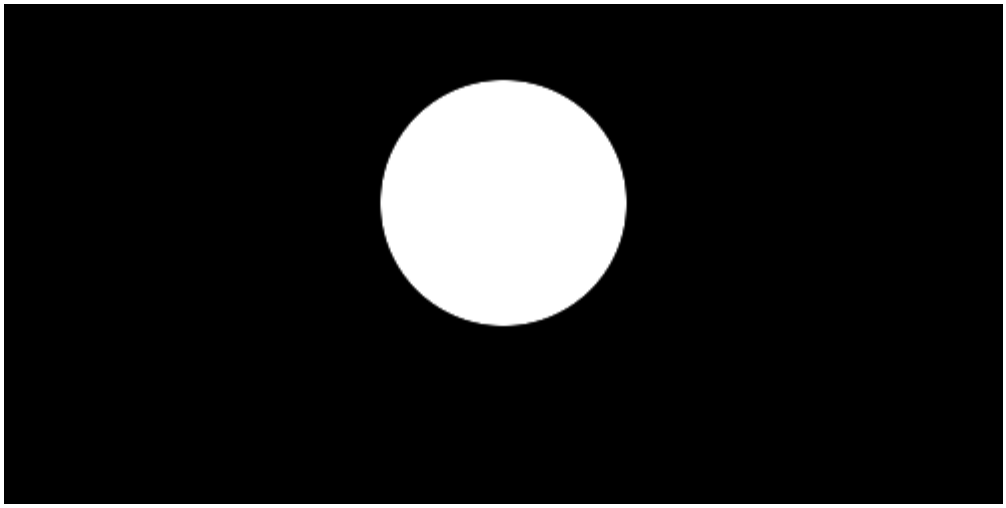
Also, Bitwise operations helps in image masking. Image creation can be enabled with the help of these operations. These operations can be helpful in enhancing the properties of the input images.

NOTE: The Bitwise operations should be applied on input images of same dimensions

Input Image 1:



Input Image 2:



Bitwise AND operation on Image:

Bit-wise conjunction of input array elements.

Syntax: *cv2.bitwise_and (source1, source2, destination, mask)*
Parameters:
source1: *First Input Image array(Single-channel, 8-bit or floating-point)*
source2: *Second Input Image array(Single-channel, 8-bit or floating-point)*
dest: *Output array (Similar to the dimensions and type of Input image array)*
mask: *Operation mask, Input / output 8-bit single-channel mask*

```
# Python program to illustrate
# arithmetic operation of
# bitwise AND of two images

# organizing imports
import cv2
import numpy as np

# path to input images are specified and
# images are loaded with imread command
img1 = cv2.imread('input1.png')
img2 = cv2.imread('input2.png')

# cv2.bitwise_and is applied over the
# image inputs with applied parameters
dest_and = cv2.bitwise_and(img2, img1, mask = None)

# the window showing output image
# with the Bitwise AND operation
```

```
# on the input images
cv2.imshow('Bitwise And', dest_and)

# De-allocate any associated memory usage
if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()
```

Bitwise OR operation on Image:

Bit-wise disjunction of input array elements.

Syntax: *cv2.bitwise_or(source1, source2, destination, mask)*
Parameters:
source1: *First Input Image array(Single-channel, 8-bit or floating-point)*
source2: *Second Input Image array(Single-channel, 8-bit or floating-point)*
dest: *Output array (Similar to the dimensions and type of Input image array)*
mask: *Operation mask, Input / output 8-bit single-channel mask*

```
# Python program to illustrate
# arithmetic operation of
# bitwise OR of two images

# organizing imports
import cv2
import numpy as np

# path to input images are specified and
# images are loaded with imread command
img1 = cv2.imread('input1.png')
img2 = cv2.imread('input2.png')

# cv2.bitwise_or is applied over the
# image inputs with applied parameters
dest_or = cv2.bitwise_or(img2, img1, mask = None)

# the window showing output image
# with the Bitwise OR operation
# on the input images
cv2.imshow('Bitwise OR', dest_or)

# De-allocate any associated memory usage
if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()
```

Bitwise XOR operation on Image:

Bit-wise exclusive-OR operation on input array elements.

Syntax: *cv2.bitwise_xor(source1, source2, destination, mask)*
Parameters:
source1: *First Input Image array(Single-channel, 8-bit or floating-point)*
source2: *Second Input Image array(Single-channel, 8-bit or floating-point)*
dest: *Output array (Similar to the dimensions and type of Input image array)*
mask: *Operation mask, Input / output 8-bit single-channel mask*

```
 # Python program to illustrate
# arithmetic operation of
# bitwise XOR of two images

# organizing imports
import cv2
import numpy as np

# path to input images are specified and
# images are loaded with imread command
img1 = cv2.imread('input1.png')
img2 = cv2.imread('input2.png')

# cv2.bitwise_xor is applied over the
# image inputs with applied parameters
dest_xor = cv2.bitwise_xor(img1, img2, mask = None)

# the window showing output image
# with the Bitwise XOR operation
# on the input images
cv2.imshow('Bitwise XOR', dest_xor)

# De-allocate any associated memory usage
if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()
```

Bitwise NOT operation on Image:

Inversion of input array elements.

Syntax: *cv2.bitwise_not(source, destination, mask)*
Parameters:
source: *Input Image array(Single-channel, 8-bit or floating-point)*
dest: *Output array (Similar to the dimensions and type of Input image array)*
mask: *Operation mask, Input / output 8-bit single-channel mask*

```
# Python program to illustrate
# arithmetic operation of
# bitwise NOT on input image

# organizing imports
import cv2
import numpy as np

# path to input images are specified and
# images are loaded with imread command
img1 = cv2.imread('input1.png')
img2 = cv2.imread('input2.png')

# cv2.bitwise_not is applied over the
# image input with applied parameters
dest_not1 = cv2.bitwise_not(img1, mask = None)
dest_not2 = cv2.bitwise_not(img2, mask = None)

# the windows showing output image
# with the Bitwise NOT operation
# on the 1st and 2nd input image
cv2.imshow('Bitwise NOT on image 1', dest_not1)
cv2.imshow('Bitwise NOT on image 2', dest_not2)

# De-allocate any associated memory usage
if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()
```

Tasks

1. Implement all above operation with the images indicated and produce outputs.

II. Apply following operation on any two I = a I1(x,y) + (1-a) I2, where 0<a<1. Choose any three values of a.

III. Use logic operation to mask object. Show one example.