MAKERERE UNIVERSITY

Solution Set 1

Image Processing (MCS 7225)

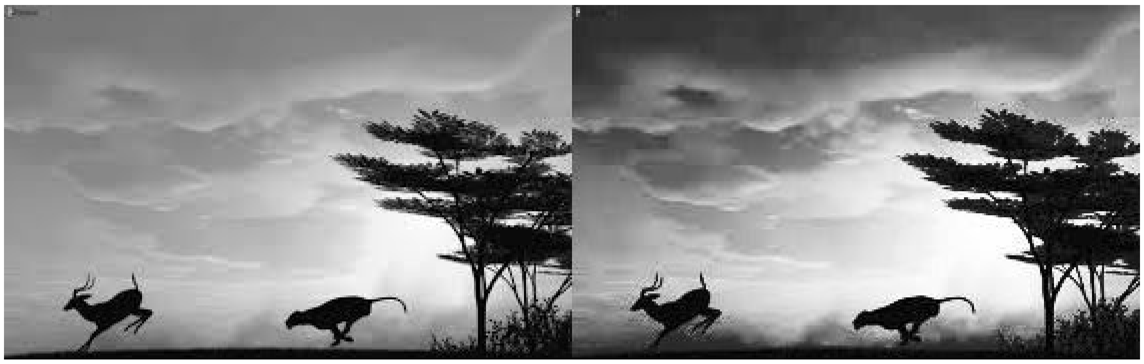
Department of Computer Science

Ibrahim Nurudeen Isa	2016/HD05/348X
Oryem Samuel	2016/HD05/340U
Semakula Abdumajidhu	2016/HD05/342U

2 (a)

Brighter images mostly have all their pixels confined to high values. Boosting the lower intensities of the image and not the higher ones will have pixels from all regions make a good image, this case we can improve contrast of the image using ‘Histogram Equalization’ to stretch our image histogram to either ends.

```
img = cv2.imread('lion.jpg',0)

equ = cv2.equalizeHist(img)

res = np.hstack((img,equ))
```

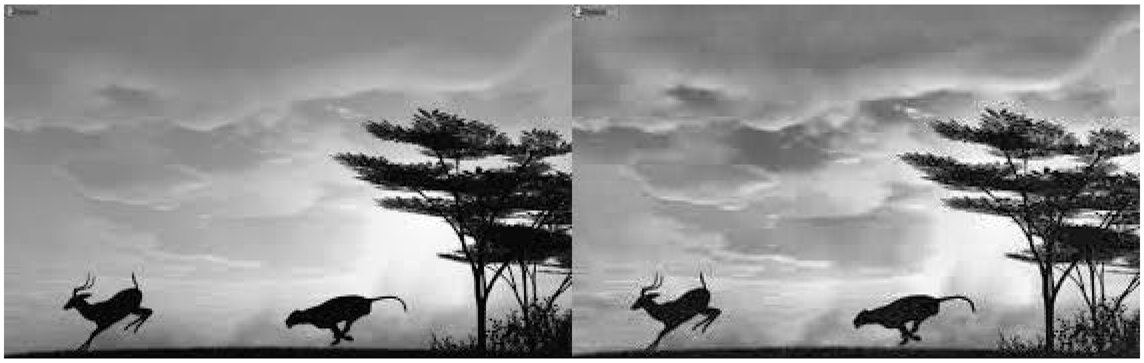


(a) (b)

Some of our high-intensity-regions in the clouds have been restored. Histogram equalization is good when histogram of the image is confined to a particular region. It won't work good in places where there is large intensity variations where histogram covers a large region, ie both bright and dark pixels are present. We still need more corrections on the position of the sun.

So to solve this problem, **adaptive histogram equalization** is used. In this, our image is divided into small blocks called "tiles". Then each of these blocks are histogram equalized as usual. So in a small area, histogram would confine to a small region (unless there is noise). If noise is there, it will be amplified. To avoid this, **contrast limiting** is applied. If any histogram bin is above the specified contrast limit, those pixels are clipped and distributed uniformly to other bins before applying histogram equalization. After equalization, to remove artifacts in tile borders, bilinear interpolation is applied. Let us try applying the CLAHE (Contrast Limited Adaptive Histogram).

```
# create a CLAHE object (Arguments are optional).

clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))

cl1 = clahe.apply(img)

res = np.hstack((img,cl1))

cv2.imwrite('newlion.jpg',res)
```

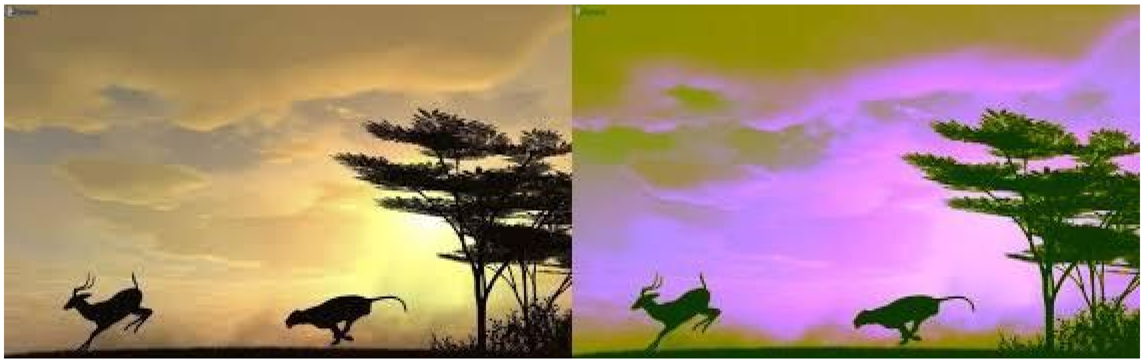


(a) (b)

See the result below and compare it with results above and see (a) the original image and (b) the output. You can easily see much of the image improvement especially when you look at the point of the sun. We have improved other regions and also the sun intensity have been matched compared to the original image but still have more dark regions in the image.

Let us see what our results would look like after applying Histogram Equalisation in RGB. We shall convert our image to YUV then apply our *equalizeHist()* then convert it back to RGB.

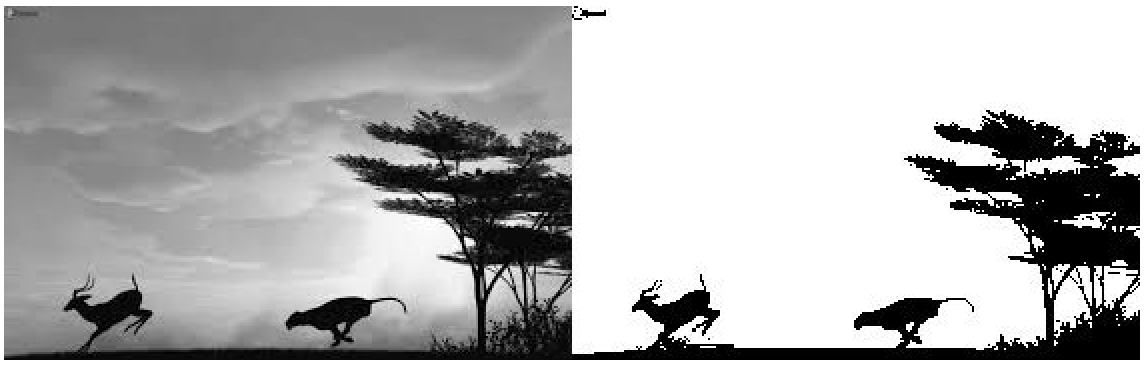
```
img_yuv = cv2.cvtColor(img, cv2.COLOR_BGR2YUV)

# equalize the histogram of the Y channel

img_yuv[:,:,1] = cv2.equalizeHist(img_yuv[:,:,0])

# convert the YUV image back to RGB format

img_output = cv2.cvtColor(img_yuv, cv2.COLOR_YUV2BGR)
```



(a) (b)

We can see our (a) original image and our (b) output where the sun high intensities are less effective but still more needfulness of seeing the dark portions of the image.

Thresholding is a great tool that can give us great looks on this. Let us get to see some of the outputs after applying threshold.

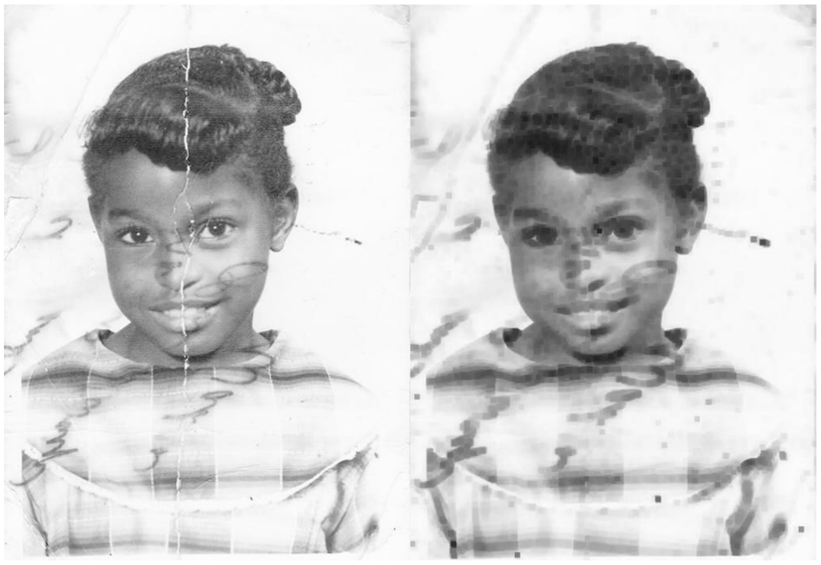


2 (b)

In old, scratched and noisy photos, we can carry out a number of operations to bring back the photo to good prints. Let us apply some of the morphological transformations to our scratched image to improve it.

Erosion helps us erode away the boundaries of foreground object especially white. The kernel slides through the image (as in 2D convolution). A pixel in the original image (either 1 or 0) will be considered 1 only if all the pixels under the kernel is 1, otherwise it is eroded (made to zero)

```
img = cv2.imread('j.png',0)
kernel = np.ones((5,5),np.uint8)
erosion = cv2.erode(img,kernel,iterations = 1)
```

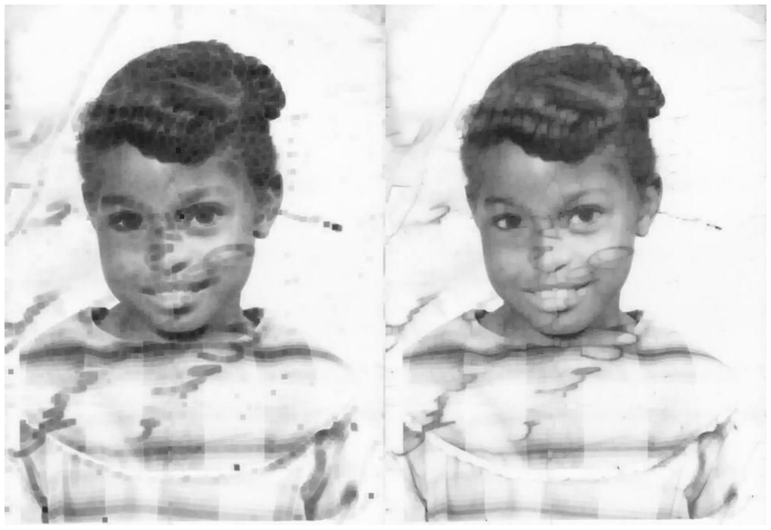


(a) (b)

The (a) original image had a long vertical crack. We manage to get rid of the crack in our (b) output image using the cv2.erode(). This made a good progress but it degrade the image and also added more noise we makes our image not good enough.

We can try to enhance our image by applying dilation. Dilation increases the white region in the image or size of foreground object increases. Because, erosion removes white noises, but it also shrinks our object. So we dilate it. Since noise is gone, they won't come back, but our object area increases. It is also useful in joining broken parts of an object.

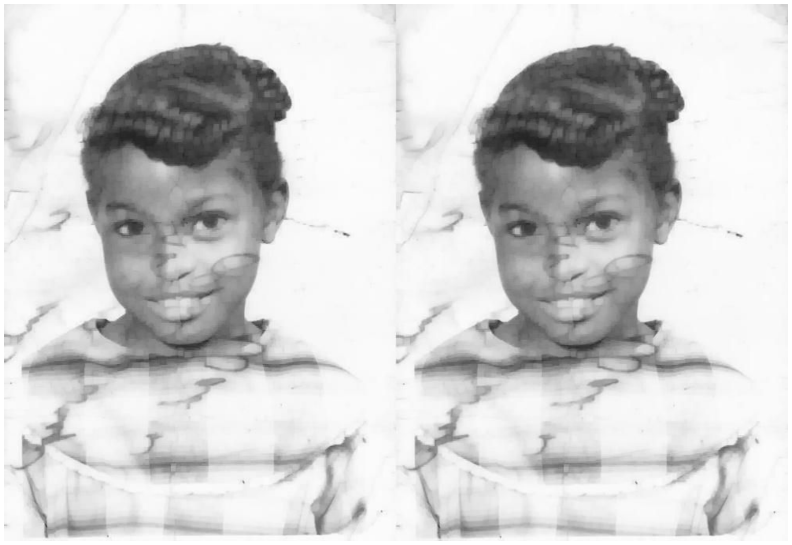
dilation = cv2.dilate(erosion,kernel,iterations = 1)



(a) (b)

Our (a) input image is much-more noisy but after dilation, we can see our (b) improved. We can further improve this by applying opening. Opening is another way of getting rid of noise, this can be an addition to were dilation stopped.

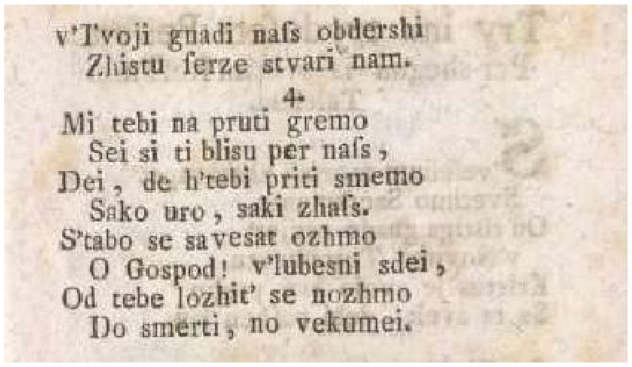
opening = cv2.morphologyEx(dilation, cv2.MORPH_OPEN, kernel)



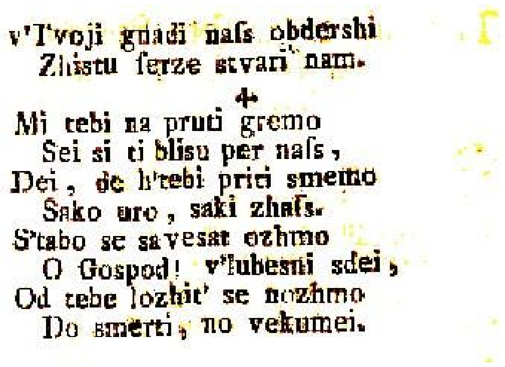
2 (c)

Noise reduction in image pixels is very important especially when preparing a document for character recognition. Techniques such as thresholding can help us improve our image with appropriate noise reduction within an image. At first we apply *cv2.threshold()*.

```
 th, dst = cv2.threshold(src, thresh, maxValue, cv2.THRESH_BINARY);
th, dsts = cv2.threshold(src, thresh2, maxValue, cv2.THRESH_BINARY);
improved = np.hstack((src,dst)) #stacking images side-by-side
improvedmore = np.hstack((src,dsts))
```



v'Tvoji gnadi naſs obdershi
Zhistu ſerze stvari nam.
4.
Mi tebi na pruti gremo
Sei si ti blisu per naſs,
Dei, de h'tebi priti smemo
Sako uro, saki zhaſs.
S'tabo se savesat ozhmo
O Gospod! v'lubesni sdei,
Od tebe lozhit' se nozhmo
Do smerti, no vekumei.

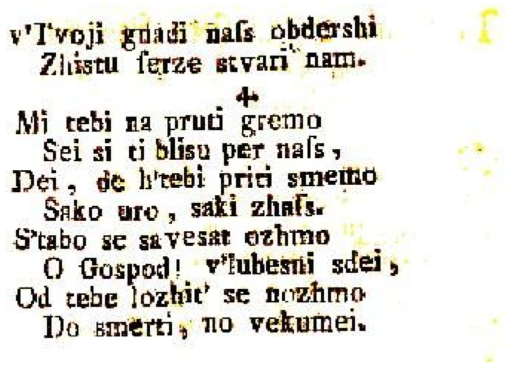


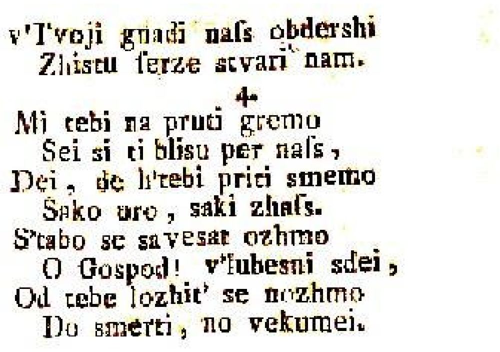
v'Tvoji gnadi naſs obdershi
Zhistu ſerze stvari nam.
4.
Mi tebi na pruti gremo
Sei si ti blisu per naſs,
Dei, de h'tebi priti smemo
Sako uro, saki zhaſs.
S'tabo se savesat ozhmo
O Gospod! v'lubesni sdei,
Od tebe lozhit' se nozhmo
Do smerti, no vekumei.

(a) (b)

As we can see, after applying threshold on our (a) original image. We have done much of noise reduction as seen in our (b) output image. We set our threshold to be 155.

Let’s compare our improvement on the output as a result of varying threshold values. Our main aim is to get a better output that could be converted to text.

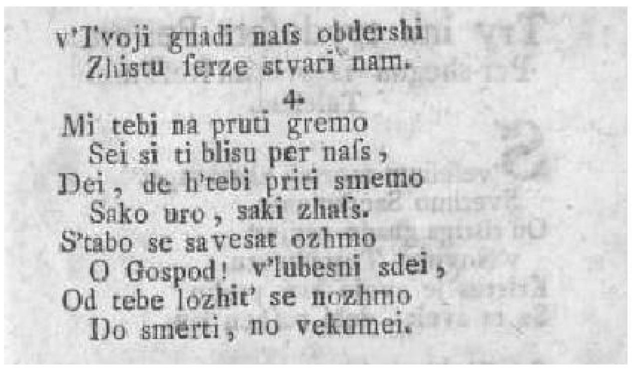

v'Tvoji gnadi naſs obdershi
Zhistu ſerze stvari nam.
4.
Mi tebi na pruti gremo
Sei si ti blisu per naſs,
Dei, de h'tebi priti smemo
Sako uro, saki zhaſs.
S'tabo se savesat ozhmo
O Gospod! v'lubesni sdei,
Od tebe lozhit' se nozhmo
Do smerti, no vekumei.

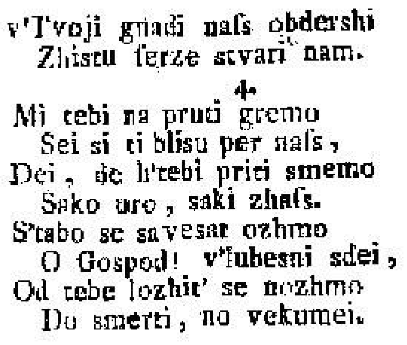

v'Tvoji gnadi naſs obdershi
Zhistu ſerze stvari nam.
4.
Mi tebi na pruti gremo
Sei si ti blisu per naſs,
Dei, de h'tebi priti smemo
Sako uro, saki zhaſs.
S'tabo se savesat ozhmo
O Gospod! v'lubesni sdei,
Od tebe lozhit' se nozhmo
Do smerti, no vekumei.

(a) (b)

Our (a) input image has some noise. Let us compare the two images above. Our (b) output shows some more good improvements.

What would our final product look like?


v'Tvoji gnadi naſs obdershi
Zhistu ſerze stvari nam.
4.
Mi tebi na pruti gremo
Sei si ti blisu per naſs,
Dei, de h'tebi priti smemo
Sako uro, saki zhaſs.
S'tabo se savesat ozhmo
O Gospod! v'lubesni sdei,
Od tebe lozhit' se nozhmo
Do smerti, no vekumei.


v'Tvoji gnadi naſs obdershi
Zhistu ſerze stvari nam.
4.
Mi tebi na pruti gremo
Sei si ti blisu per naſs,
Dei, de h'tebi priti smemo
Sako uro, saki zhaſs.
S'tabo se savesat ozhmo
O Gospod! v'lubesni sdei,
Od tebe lozhit' se nozhmo
Do smerti, no vekumei.

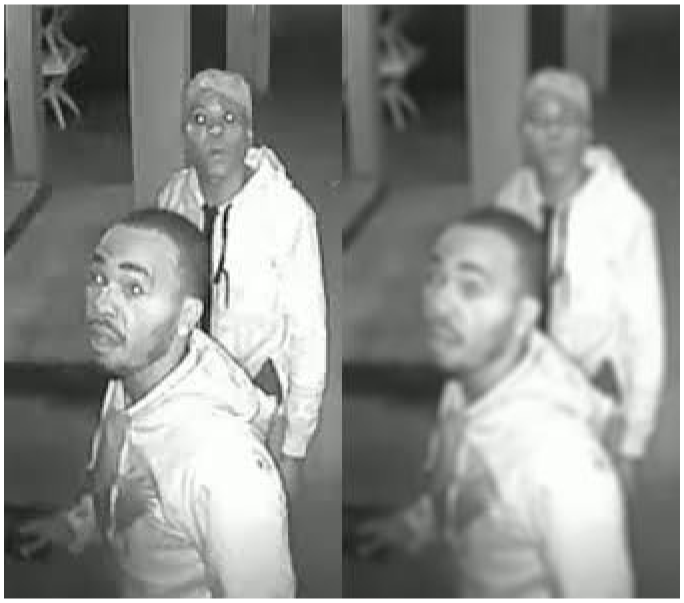
(a) (b)

(a) Is our original image and (b) is the final output as shown above.

2 (d)

We can make suspect images more clear by smoothing and applying custom-made filters to images. At first, we convolve the image with a low-pass filter kernel. It simply takes the average of all the pixels under kernel area and replaces the central element with this average. This is done by the function ***cv2.blur()*** or ***cv2.boxFilter()***.

blur = cv2.blur(img,(5,5))


(a) (b)

As we observe from the results above, we specify the width and height of kernel. A 3x3 normalized box filter. Our (a) input / original image and (b) output image is shown above. Our suspect image is smoothed but too much blur.

We can more improve this, lets try implementing Gaussian filtering. We implement a Gaussian kernel Instead of a box filter consisting of equal filter coefficients. It is done with the function, **cv2.GaussianBlur()**.

gausian = cv2.GaussianBlur(img,(5,5),0)

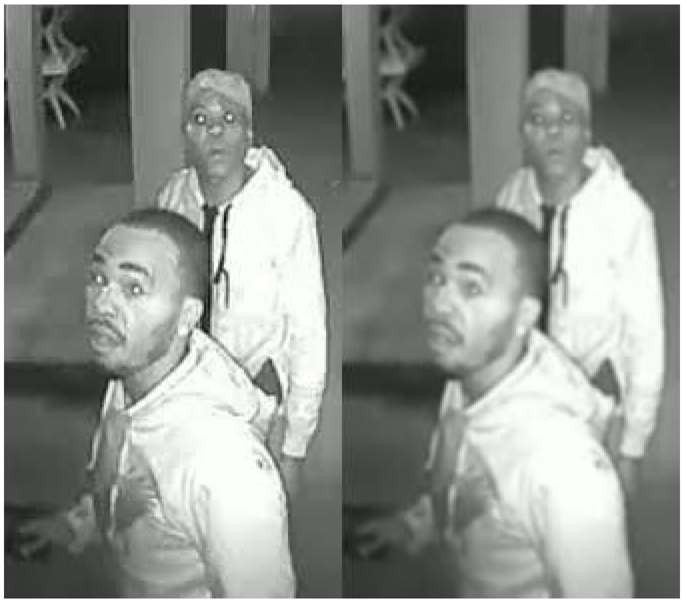
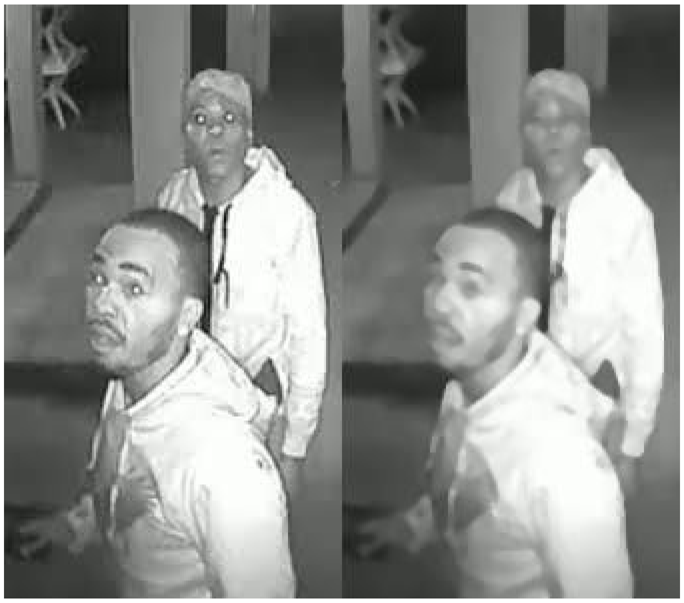

(a) (b)

Image (a) being our input, we show some improvements in our (b) output as compared from the previous blur.

When we further implement Median Blur, the function **cv2.medianBlur()** computes the median of all the pixels under the kernel window and the central pixel is replaced with this median value.

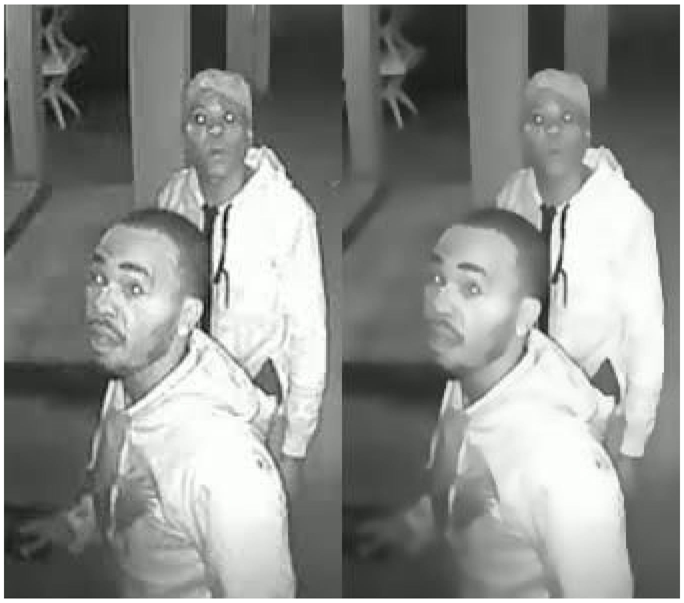
```
median = cv2.medianBlur(img,5)
```


(a) (b)

(a) Is our input image. After applying the median blur, we see (b) our output looking better.

Finally, I want us to witness the powers of bilateral filtering. The bilateral filter uses a Gaussian filter in the space domain, but it also uses one more (multiplicative) Gaussian filter component which is a function of pixel intensity differences.

```
bilateral = cv2.bilateralFilter(img,9,75,75)
```



(a) (b)

Considering all our previous filters, our (b) result shows a desirably better output.