CS364/AM792: Assignment #1

Due on February 26, 2020 at 17:00

Unathi Skosana

Problem 1

Solution

Separating the actors from the green background in the image below, amounted to appropriately thresholding the different color channels. My first few attempts were naively trying to only threshold the green channel. However, this has the effect of also mistaking some parts of the actors for the background, as the actors have colors that are mixtures of green and the other channels.

Figure 1: Greenscreen

Continued tinkering, led me to the following thresholding condition on the red, green and blue channels respectively.

$$r < 10 \cap g > 100 \cap b > 50$$

Using DeMorgan's law and numpy's logical_or function, the following code snippet gives a binary array that's the size of image with 0s for pixels deemed as the background and 1s otherwise.

```python
def threshold_func(image):
    r, g, b = image[:,:,0], image[:,:,1], image[:,:,2]
    return np.array(np.logical_or(r >= 10, g <= 100, b <= 50), dtype="uint8")
```

Problem 1 continued on next page...

Plotting this array as an image produces the approximate outline of the actors as seen below.

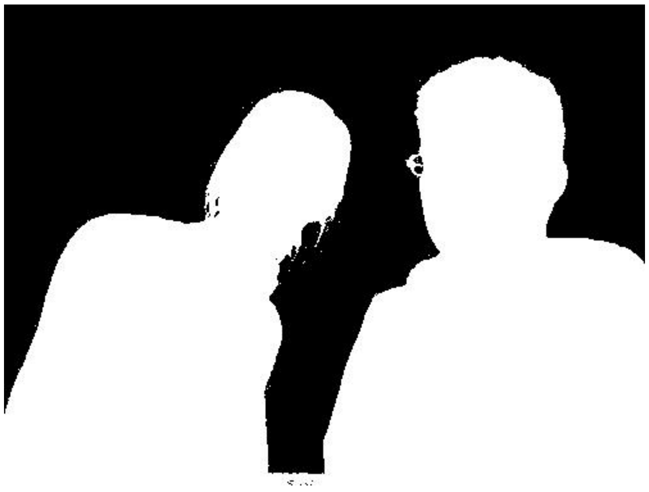


Figure 2: outline

Making use of such an array we can put the actors on any other background.

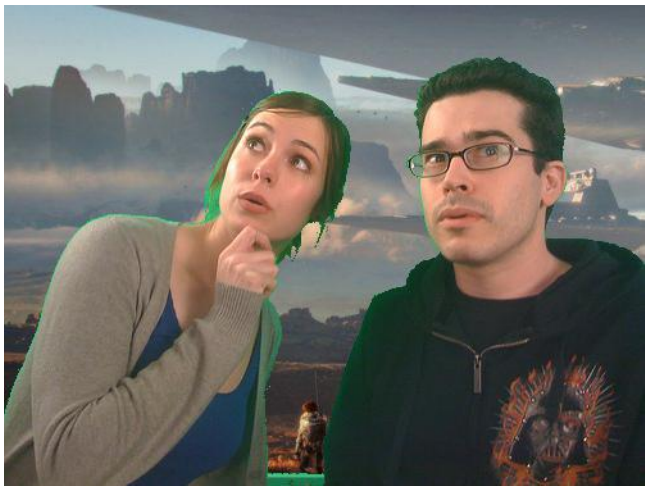


Figure 3: Actors on a new background

j

Problem 2

Solution
The unsharp masking was done on the grayscale image below.

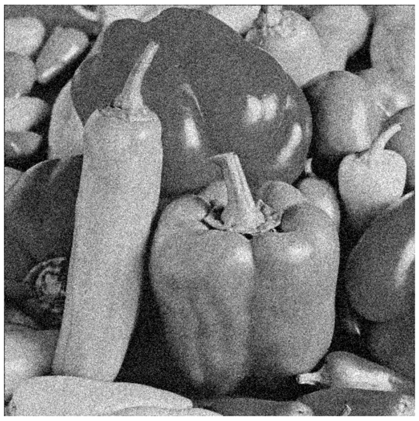


Figure 4: Noisy grayscale image

To obtain the blurred above image, an averaging filter mask of size $2 * h + 1 = 10$ was used. Alternatively one can also use the median filter to achieve roughly the same results.

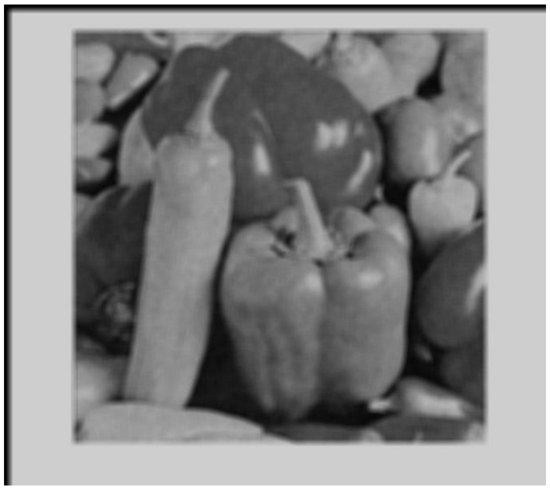


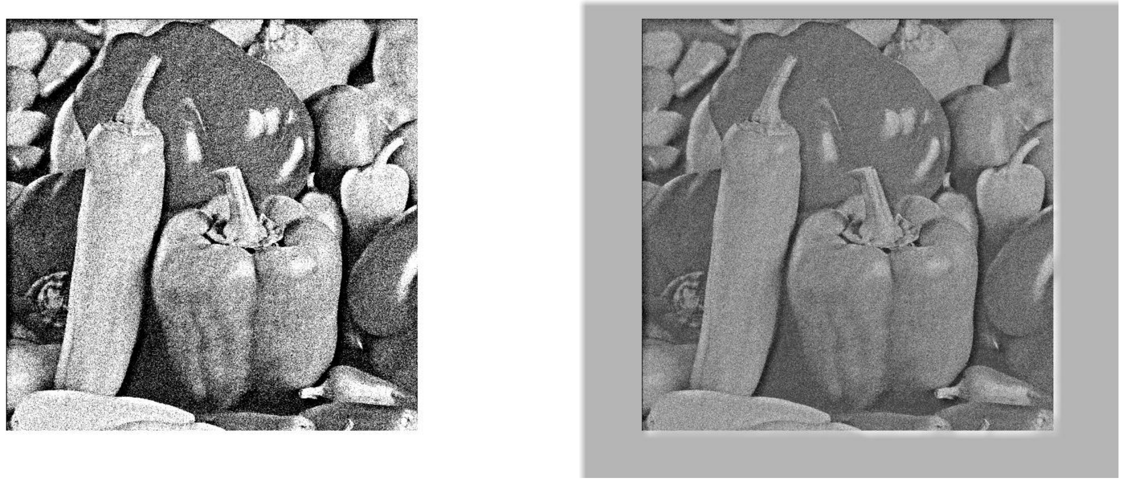
Figure 5: Blurred grayscale image

NB: The image was originally padded with white pixels.

Problem 2 continued on next page...

The unsharp masking algorithm produces output with pixel values $p \notin (0, 255)$ reason being that for some pixel values the expression `output = 2 * input − blurred` can produce values outside the range $(0, 255)$.

Simply truncating the out-of-range pixel values produces the results in Fig. 6a. We get the contrast/sharpness at the expense of having a grainy image. Where else normalizing the pixel values so that they fall in the range $(0, 255)$ has the effect of lowering the contrast/sharpness of the image, undoing the sharpening.



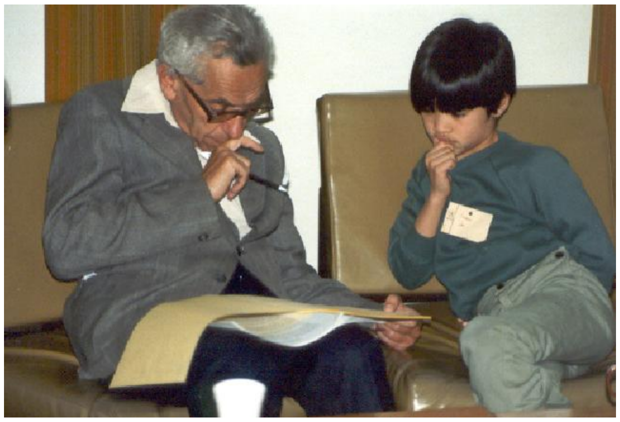
(a) Clipping values outside $(0, 255)$ **(b)** Normalizing pixels

Figure 6: Unsharp masking results

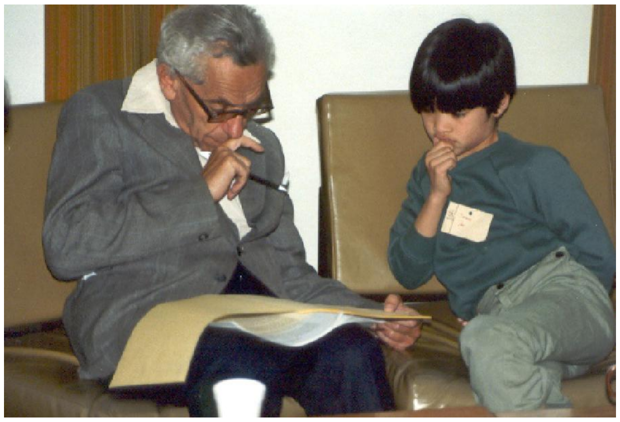
Problem 3

Solution

The nearest neighbor and bilinear interpolation methods were carried out at scales $1.25, 1.75, 2.65, 4.0$. Below are the results


(a) Nearest neightbor interpolationg at the scale of 1.25

(b) Bilinear interpolation at the scale of 1.25


(c) Nearest neightbor interpolationg at the scale of 1.75

(d) Bilinear interpolation at the scale of 1.75

Problem 3 continued on next page...

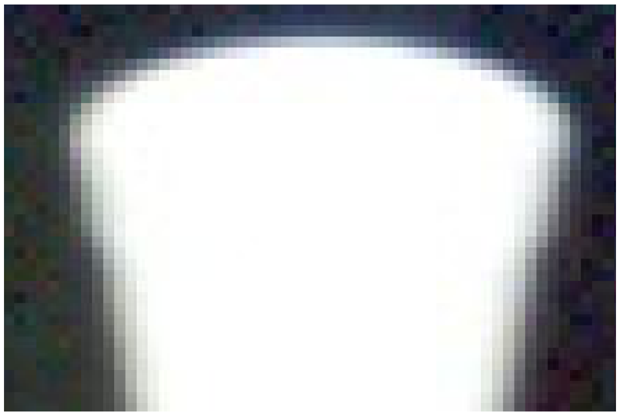
(a) Nearest neightbor interpolationg at the scale of 2.50

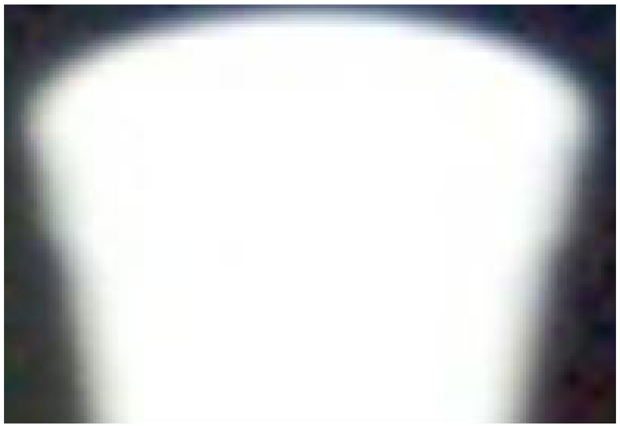
(b) Bilinear interpolation at the scale of 2.50

(c) Nearest neightbor interpolationg at the scale of 4.0

(d) Bilinear interpolation at the scale of 4.0

Inspecting zoomed-in images at the scale of 4.00 of the two methods, we can difference between nearest neighbor and bilinear interpolation, the former provides a 'blocky' pixel resolution while the former is gives a smoother resolution.


(a) Nearest neightbor interpolationg at the scale of 4.00 zoomedin on the cup


(b) Bilinear interpolation at the scale of 4.00 zoomed in on the cup

Problem 4

Solution

(a)

My chosen method was skimage's ORB module, which combines some techniques from FAST and BRIEF feature detectors. ORB has been shown to be two orders of magnitude better than SIFT in computational time complexity. Furthermore ORB is rotationally invariant. Getting skimage to perform the two tasks is simple as using the already provided API to detect and extract keypoints and descriptors. (Sources in the code)

(b)

The default parameters of the ORB module work reasonably well. On the results the number of keypoints to be identified was 300, a considerably large fraction of these keypoints was matched.

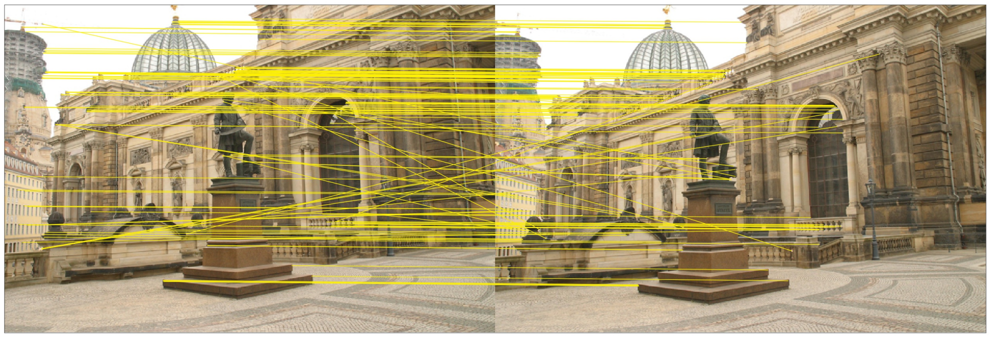


Figure 10: Feature matching using ORB

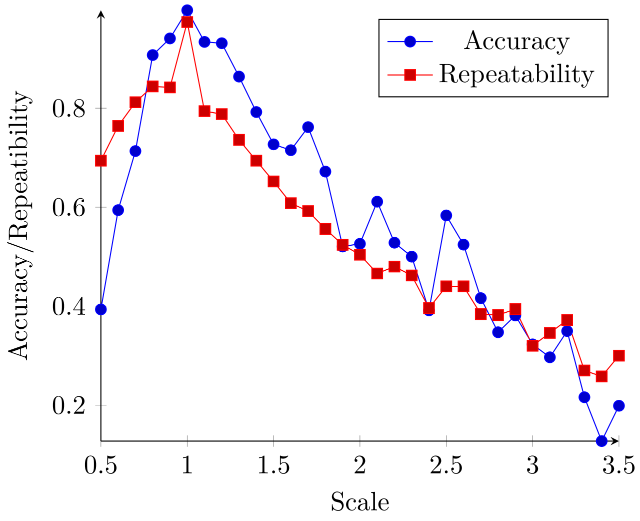
Problem 5

Solution

To perform these two tests, I had first had to scale the images for the desired scales and save the images to disk to reading them later. The grayscale image below was used. The desired scales were in the range $(0.5, 3.5)$ with increments of 0.1. Using the feature matching functions from the previous question, the number of keypoints was set to 500.

Figure 11: Claude Shannon

The following plots of accuracy and repeatability were produced.



A few typical results are presented at the end. The results seem to suggest that the ORB feature matching algorithm is not scale invariant. Both accuracy and repeatability decrease rapidly on either side of the line scale = 0.1

Problem 5 continued on next page...

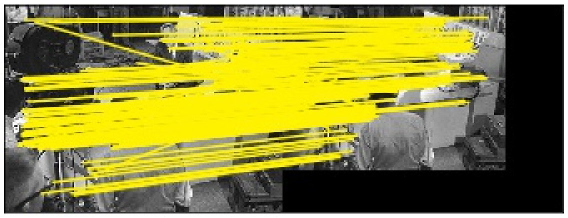


Figure 12: Matches at scale = 0.8

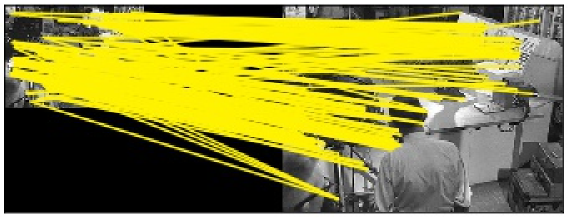


Figure 13: Matches at scale = 2.0

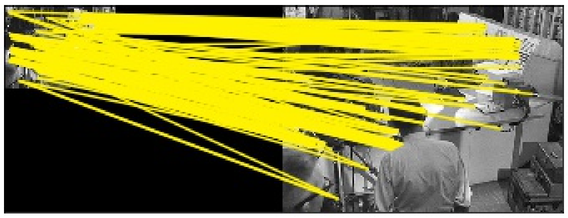


Figure 14: Matches at scale = 2.5

Problem 5 continued on next page...