Computer vision 792/492 Assignment 1

David Nicolay (26296918) 6 August 2025

Problem 1: Colour thresholding

For this problem, the source image is first converted (Figure 1) from RGB to RGBA. This adds an *alpha* channel which is used to represent the level of transparency. The lower threshold is defined as: red 0, green 150 and blue 0. The upper threshold is defined as: red 100, green 255 and blue 255. This gives us a green-ish region. A binary mask (Figure 2) is created by finding all the pixels with RGB values within these defined bounds (thresholds). The background is then removed by setting the alpha channel of each of the pixels aligning with the mask to 0 which is transparent (Figure 3). The actors are placed on the new background by simply choosing the pixel from the background-removed image or the new background based on the mask.

The actors are successfully extracted from the source image. I did tune the thresholds to ensure that the green screen is removed as accurately as possible. There is some colour bleed on the actors, which result in part of the actors being removed if the thresholds are made less strict. The black bars in Figure 4 are simply due to the new background being a slightly different resolution. Naively adding the background directly to the RGBA image led to strange bright colours. I fixed this by using `np.where` to assign the background or the green screen image based on the mask.

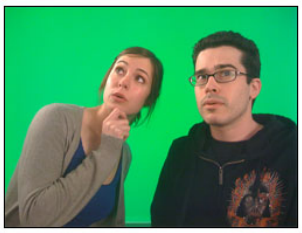


Figure 1: Original image

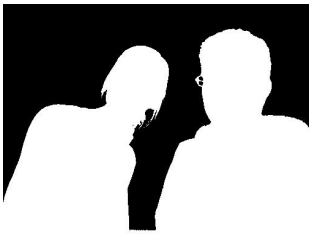


Figure 2: Image mask

Figure 3: Background removed

Figure 4: Actors replaced

Problem 2: Image denoising

I achieved "salt and pepper" image degradation by first calculating the number of pixels and multiplying this by the parameter `d` to obtain the number of pixels to select. A random choice of indices is made, picking only the specified number of pixels previously obtained. This list of indices is simply cut in half and the first half used for salt (white) pixels and the second half for pepper (black) pixels. From here, I simply create two boolean masks of the length of the total number of pixels, setting the indices of salt and pepper pixels to true in each separately. The masks are then reshaped to the same dimensions as the image. The pixels at the salt mask indices are set to 255 and the pixels at the pepper indices are set to 0.
The colour image (Figure 5) is corrupted by running each colour channel separately through the image degradation function I previously described. It can be observed in Figure 6 that, as expected, the noise manifests as all sorts of bright colours. Since each channel has noise applied separately, some pixels will have noise on multiple channels. For example, if red is set 255, green set to 0 and blue remains unchanged at 100, what results is a bright pinkish purple pixel. I also experimented with more severe noise and the result can be observed in Figure 8 where the original image is barely visible. At first when experimenting, when I set d=1 I could still see the image. I noticed this was because in my

single-channel noise function I was choosing noise pixels for salt and then choosing noise pixels for pepper separately. This meant some pixels would be selected for both and the desired density not achieved.

I implemented the median filter by looping through each pixel in the image and building a neighbourhood of the filter. The distance each side of each pixel I need to use in the neighbourhood is determined by the filter size divided by 2 and the result truncated. So for each pixel the function builds the neighbourhood by first checking if the neighbourhood pixel is in bounds of the image. Then if it is, it adds the neighbourhood pixel to a list. Once all the neighbourhood pixels have been accounted for, the median values of each of the colour channels are calculated, and the final pixel set to those values. This process is repeated for each pixel. Experimenting with various filter sizes I observed that with a smaller filter the picture appears a higher resolution (observed in Figure 7), but still contains some stray noise pixels. With a larger filter however, the image appears a lower resolution, but few to no stray noise pixels at all as observed in Figure 8. Applying a median filter to a highly degraded image results in noise pixels remaining even if the filter size is increased. Interestingly, in my testing with a noise density of 0.7 and a median filter of 7, a surprisingly large amount of the image can be recovered, albeit with some noise remaining.

Figure 5: Original colour image

Figure 6: Salt and pepper noise (colour) d=0.3

Figure 7: After median filter size 3

Figure 8: After median filter size 5

Problem 3: Image sharpening

Unsharp masking was applied to the image in Figure 9a, resulting in the sharpened version shown in Figure 9b. To achieve this I first padded the image with a 2 zeros all around so that I could use a 5×5 averaging filter to blur the the image. The result is Figure 9c. The mask is then created by subtracting the blurred image from the original image. To display the mask (Figure 9d) it was shifted and scaled to the range $[0, 255]$ along with being converted to `uint8`. The sharpened image is created by simply adding the mask to the original image using $k = 1$ in Equation 1.

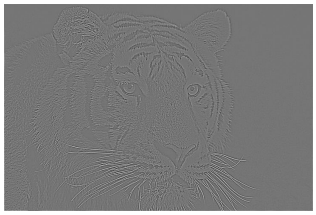
$$\texttt{sharpened\_image} = \texttt{original\_image} + k \times \texttt{mask} \tag{1}$$

I experimented with a 3×3 filter where the sharpening effects were less noticeable and 7×7 filter which was similar to the 5×5 filter. It is also important to note that the mask is NOT scaled before adding it to the original image, this results in a low-contrast image instead of a sharpened image. I simply created a copy of the mask and scaled it for display purposes.

(a) Original image

(b) Sharpened image

(c) Blurred image



(d) Mask

Figure 9: Different components of the unsharp masking process.

Problem 4: Bilinear interpolation

For the nearest neighbour (NN) interpolation, my function begins by calculating the new height and weight of the image by simply multiplying the old value by s (scaling factor) and truncating the result to an integer. The x scale and the y scale are calculated by dividing the old width and height by the new width and heigh respectively. I create a new empty scaled image with the new image dimensions and appropriate number of colour channels (in my colour examples this would be 3). The core of the algorithm is the calculation of the 1D arrays of input coordinates for x and y. Equation 2 and Equation 3 are used in this calculation where j is the column index in the new image and i is the row index in the new image. Once these 1D arrays are created, I simply loop through each of the pixels in the image (using a nested for loop going through y_{old} and x_{old}) to copy across the pixels into the new image.

$$x_{\text{old}}(j) = \min\left(\text{width}_{\text{original}} - 1,\ \max\left(0,\ \left\lfloor \frac{j \cdot \text{width}_{\text{original}}}{\text{width}_{\text{new}}} + 0.5 \right\rfloor\right)\right) \tag{2}$$

$$y_{\text{old}}(i) = \min\left(\text{height}_{\text{original}} - 1,\ \max\left(0,\ \left\lfloor \frac{i \cdot \text{height}_{\text{original}}}{\text{height}_{\text{new}}} + 0.5 \right\rfloor\right)\right) \tag{3}$$

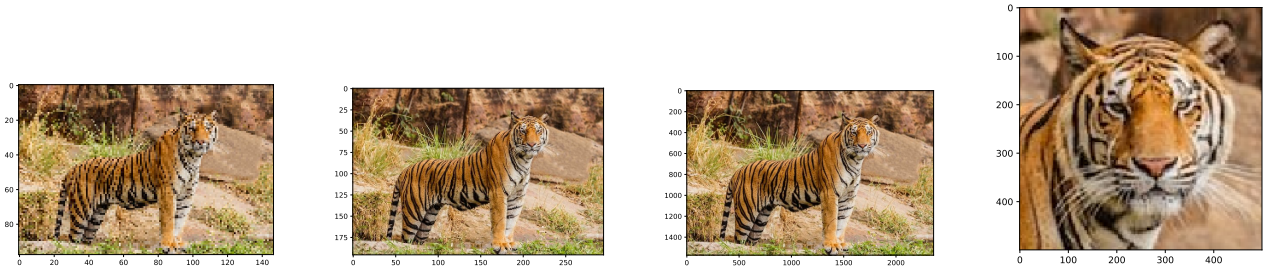
The bilinear (BL) interpolation begins the same as the NN function by calculating the new height and width of the image along with the x and y scale. The empty new image is created with the new height and width. The function now iterates over every pixel in the new image and computes the colour by interpolating between the four closest pixels in the original image. This is achieved by first finding the floating-point coordinate in the original image, then finding the four surrounding pixels. It then calculates the fractional distances to the neighbours and interpolates the pixel by blending the four neighbouring pixels, weighted by their distance to the target point. I referred to Gonzalez and Woods (2002) for these algorithms.

Observe that downscaling the images using both NN and BL functions yield similar results (Figure 10a, Figure 10e, Figure 10b and Figure 10f). Upscaling on the other hand pronounces the differences between NN and BL interpolation as the scale is increased (Figure 10c and Figure 10g). Consider Figure 10d and Figure 10h. The NN interpolation preserves the boundaries and the image tends to be more blocky or pixelated. The BL interpolation creates a smoother, but more blurred image.

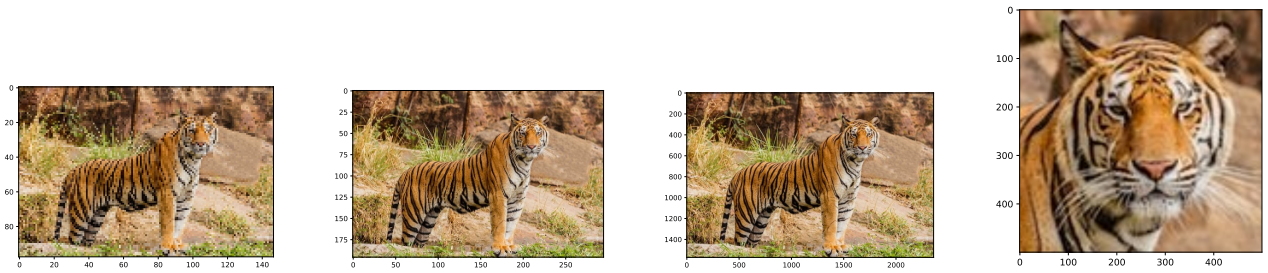
Problem 5: Image feature detection and matching

I use the SIFT (Scale-Invariant Feature Transform by Lowe (2004)) image feature detector and matcher from OpenCV (Bradski (2000)) to build the two functions. The first function that obtains the feature coordinates and the corresponding descriptor vectors first creates a SIFT object using OpenCV's `SIFT_create` function with optional parameters. It then calls the `detectAndCompute` function on the SIFT object with the image as parameter. This function detects the key points in the image using the SIFT algorithm and simultaneously computes the 128-dimensional descriptor vectors that characterise each key point's local neighbourhood. SIFT detects distinctive key points in images by finding local extrema in a scale-space pyramid constructed from Difference of Gaussians, ensuring the features are invariant to scale and rotation changes. The key points are represented as OpenCV KeyPoint objects containing spatial coordinates and other properties, from which I extract the (x, y) pixel coordinates by accessing the `pt` attribute and converting to integers. Finally, the function returns a list of tuples pairing each coordinate with its corresponding descriptor vector along with the key points.

The function to match two sets of descriptors from two images takes the two feature tuple pairs and extracts the descriptors. It then creates a OpenCV `BFMatcher` (Brute-Force Matcher) object and calls the `knnMatch` function on the two descriptors and $k = 2$. It finds the 2 best matches in second image's descriptors for each descriptor in the first, setting up to later decide which ones are actually



(a) NN, $s = 0.25$ (b) NN, $s = 0.5$ (c) NN, $s = 4.0$ (d) NN, $s = 4.0$ (zoom)



(e) BL, $s = 0.25$ (f) BL, $s = 0.5$ (g) BL, $s = 4.0$ (h) BL, $s = 4.0$ (zoom)

Figure 10: Comparison of Nearest Neighbour (NN) and Bilinear (BL) interpolation across different scaling factors.

good matches using Lowe's ratio test. This ratio test filters out ambiguous matches by keeping a match only if its distance is significantly lower ($< 0.75\times$) than the second-best match's distance. After some considerable time experimenting with parameter values, I progressed from Figure 11 to Figure 12. Although not perfect, Figure 12 exhibits good results with a higher number of distinct, well-aligned feature matches across both images. This demonstrates that my SIFT matching approach is effective at capturing robust correspondences.

Figure 11: Feature matching using SIFT parameters: `nfeatures = 500`, `contrastThreshold = 0.03`, `edgeThreshold = 10`, `sigma = 1.6`.

Figure 12: Feature matching using SIFT parameters: `nfeatures = 1500`, `contrastThreshold = 0.04`, `edgeThreshold = 10`, `sigma = 1.6`.

Problem 6: Feature matching accuracy against scale change

For this problem I employ my BL scaling function from question 4 to scale the image. Thereafter, I convert the images to be compatible with OpenCV by converting the type to `uint8` and the colour from RGB to BGR. Now I call my function 1 from problem 5 to obtain the coordinates and descriptors of the features. Using these descriptors I match the features between the images using function 2 from problem 5. Now I evaluate the accuracy by iterating over each match (returned by function 2) and getting the coordinates of each matched key point. Then I scale the key point from image A and measure the Euclidean distance between actual and expected point and check whether the distance is within a specified tolerance, in my implementation 2 pixels. Finally I calculate the accuracy as a percentage; the images in Figure 13 with a scaling factor of 2 resulted in an accuracy of 90.83%. My function repeats this process with scaling factors starting at 0.1 to 3 in steps of 0.1 and records the resulting accuracies.

The general trend in the accuracy is that it increases as the scaling factor approaches one (where obviously the accuracy is 100%) indicated in Figure 14. Thereafter, the accuracy slowly decreases as the scale factor increases. This makes sense as as the scale increases to one, more and more of the image is preserved. As the scale increases past one, less and less of the original image is preserved as the scaling interpolation is required to fill in more and more gaps between pixels.

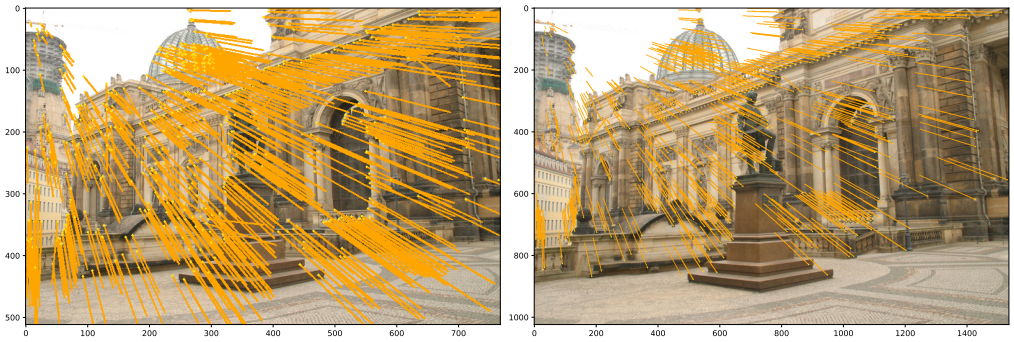


Figure 13: Feature matching using SIFT and a scale of 2

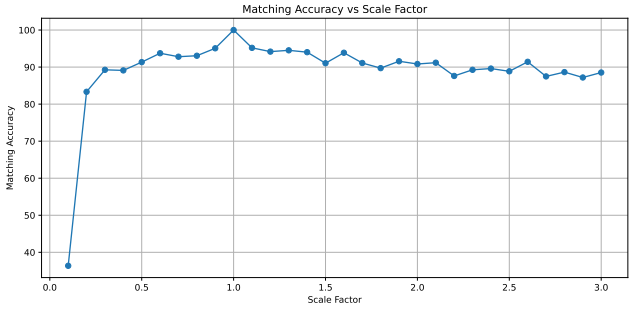


Figure 14: Feature matching using SIFT and a scale of 2

References

G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

Rafael C Gonzalez and Richard E Woods. *Digital Image Processing*. Prentice Hall, 2002.

David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.