SIP A8

A. Holst (wlc376)

April 3, 2023

1 Transformations on points clouds

1.1 Procrustes: Benefits of homogeneous coordinates

Homogeneous coordinates are advantageous for Procrustes because it allows us to concisely but explicitly describe transformations in terms of linear maps, ie. (transformation) matrices. This is also beneficial for implementation as it means alignment can be efficiently implemented with matrix multiplication.

1.2 Procrustes: Degrees of freedom and minimum number of points

There are four degrees of freedom: Two for the separate translations along the x and y axes; one for the rotation; and one for the scaling (since Procrustes uses only uniform scaling).

If $N = 1$, then we *are* able to perfectly align the single point pair, but we will have infinitely many choices for the rotation and scaling factors, since a shape consisting of a single point that has been moved to the origin (0, 0) by standardization is rotationally symmetric and scaling does not transform the origin. Hence we require $N \geq 2$ to uniquely determine the parameters of the transformation.

2 Transformations on images

2.1 Describing TRS with homogeneous coordinates

Please note: As I understand the task, we are to compute the coordinate $(x_0\, y_0)$ such that $\tilde{I}(x,\,y) = I(x_0,\ y_0)$.

To do so, we first compute the inverse map $(\mathbf{TRS})^{-1}$:

$$\begin{aligned}\tilde{I}(x,\,y) = {} & I(x_0,\,y_0)\\ \text{where } \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} & = (\mathbf{TRS})^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.\end{aligned}$$

In order to compute $(x_0,\,y_0)$, we first compute the inverse map $(\mathbf{TRS})^{-1}$:

$$\begin{aligned}(\mathbf{TRS})^{-1} &= \left(\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} \right)^{-1}\\ &= \begin{bmatrix} s\cos\theta & -s\sin\theta & t_x \\ s\sin\theta & s\cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix}^{-1}\\ &= \frac{1}{s} \begin{bmatrix} \cos\theta & \sin\theta & -t_x\cos\theta - t_y\sin\theta \\ -\sin\theta & \cos\theta & t_x\sin\theta - t_y\cos\theta \\ 0 & 0 & s \end{bmatrix}.\end{aligned}$$

We then have:

$$\begin{aligned}\begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} &= (\mathbf{TRS})^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}\\ &= \frac{1}{s} \begin{bmatrix} \cos\theta & \sin\theta & -t_x\cos\theta - t_y\sin\theta \\ -\sin\theta & \cos\theta & t_x\sin\theta - t_y\cos\theta \\ 0 & 0 & s \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}\\ &= \frac{1}{s} \begin{bmatrix} x\cos\theta + y\sin\theta - t_x\cos\theta - t_y\sin\theta \\ -x\sin\theta + y\cos\theta + t_x\sin\theta - t_y\cos\theta \\ s \end{bmatrix}.\end{aligned}$$

Putting it all together, we have:

$$\begin{aligned}\tilde{I}(x,\, y) &= I\left(\frac{\cos\theta(x - t_x) + \sin\theta(y - t_y)}{s},\ \frac{\sin\theta(t_x - x) + \cos\theta(y - t_y)}{s}\right)\\ &= I(x_0,\, y_0).\end{aligned}$$

To use nearest neighbor interpolation, we then round x_0 and y_0 to the nearest integers:

$$\tilde{I}(x,\, y) = I\big(\mathrm{round}(x_0),\, \mathrm{round}(y_0)\big).$$

2.2 Transformations in Python

Listing 1 shows the function I write to perform the transformations using nearest neighbor interpolation. The function assumes `f_inv` is a 3×3 inverse map matrix.

```
def my_warp_NN(img, f_inv, cval = 0):

    # array of homogeneous coordinates into img.
    idx1 = np.c_[[*np.ndindex(img.shape)], [1] * img.size].T

    # backward mapped coordinates with NN-interpolation.
    idx2 = (f_inv @ idx1)[:2].T.round().astype(int)

    # mask valid indices.
    mask = ((idx2 >= 0) & (idx2 < img.shape)).all(1)

    # construct result.
    res = np.ones_like(img) * cval
    res[tuple(idx1[:2, mask])] = img[tuple(idx2[mask].T)]
    return res
```

Listing 1: Image warping using NN-interpolation.

Next, listing 2 shows the functions I use to construct transformation matrices **T**, **R**, and **S**, as well as the combined transformation needed for task 2.2.

```python
def T(t):
    return np.c_[np.eye(3, 2), [*t, 1]]

def S(s):
    return np.diag([s, s, 1])

def R(theta):
    cos_t, sin_t = np.cos(theta), np.sin(theta)
    return np.array([[cos_t, -sin_t, 0], [sin_t, cos_t, 0], [0, 0, 1]])

def task_2_2_transformation(center, t, theta, s):
    Tc = T(center)
    return T(t) @ Tc @ R(theta) @ S(s) @ np.linalg.inv(Tc)
```

Listing 2: Transformation matrices.

Finally, listing 3 shows the code used to generate the example described in the assignment text, while fig. 1 shows the actual result.

```python
img = np.zeros((100, 100))
img[40:60, 40:60] = 1

center = np.asarray(img.shape) // 2
t, theta, s = (10.4, 15.7), np.pi / 10, 2

f   = task_2_2_transformation(center, t, theta, s)
res = my_warp_NN(img, np.linalg.inv(f))
```

Listing 3: Code for generating fig. 1.

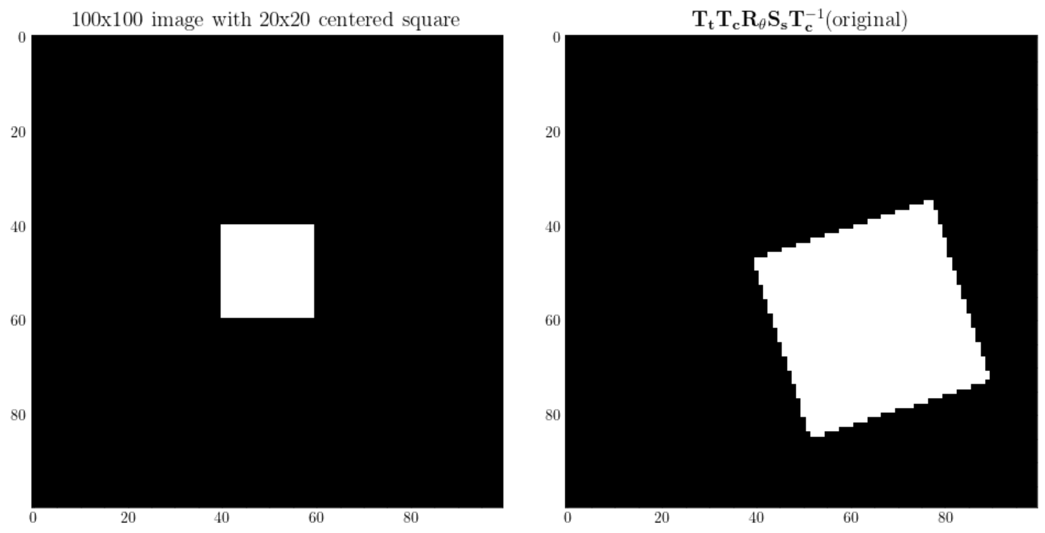
100x100 image with 20x20 centered square
$\mathbf{T_t T_c R_\theta S_s T_c^{-1}}$(original)
0
20
40
60
80
0
20
40
60
80
0
20
40
60
80
0
20
40
60
80

Figure 1: 100x100 black image with centered 20x20 white pixels, and the same image warped using $\mathbf{T_t T_c R_\theta S_s T_c}^{-1}$.

3 Transformations on point clouds (2)

3.1 Aligning the training and testing data

Listing 4 shows the code I use to compute the alignments, and fig. 2 shows the alignments of the first ten training wings using the first training wing as target.

Please note: The assignment text states that "*The landmark points [...] can be visualized as closed curves, which we will do in these questions*", however Figure 1 in the assignment text shows the landmarks plotted with a scatter plot. Hence it is ambiguous as to whether the aligned wings should be plotted using a scatter plot or as a closed curve. I choose to plot the closed curves even if the plots are a little messy.

```python
from scipy.spatial import procrustes
...
def procrustes_many(target_wing, wings):
    return np.array([procrustes(target_wing, wing)[1]
                     for wing in wings])
...
# perform procrustes.
target_wing = train_wings[0]
train_wings_aligned = procrustes_many(target_wing, train_wings)
test_wings_aligned  = procrustes_many(target_wing, test_wings)
```

Listing 4: Code for Procrustes alignment of training and testing data.

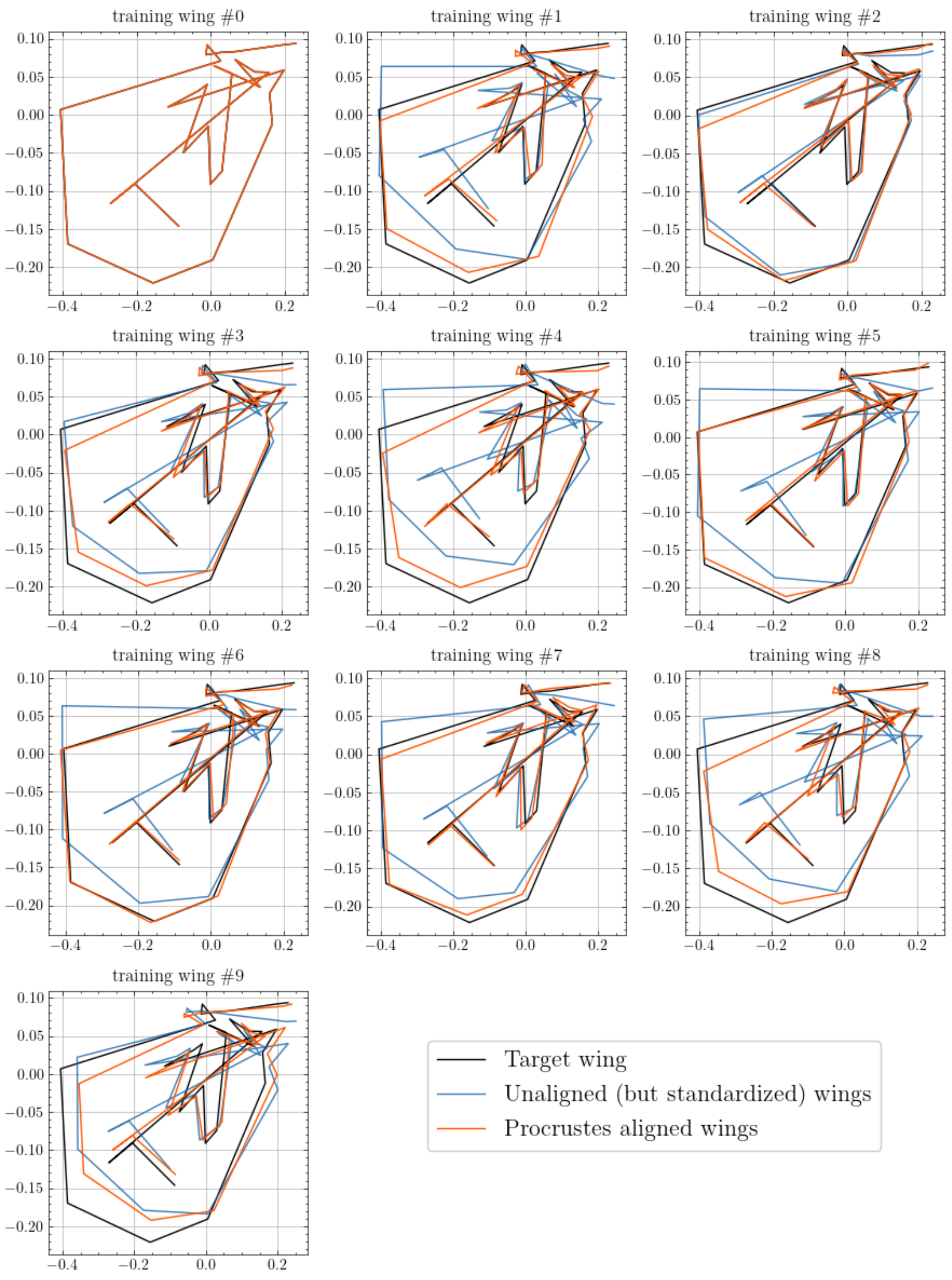

Figure 2: Procrustes alignments of the first 10 training wings using the first training wing as target.

3.2 Alignment and classification

I choose to simply use a k-NN classifier to perform the classifications, and I use the implementation from `sklearn` with uniformly weighted neighbors.

I perform a simple parameter search to find the optimal value for k for both the unaligned and aligned training data sets. The parameter search uses a randomly generated 20% training/validation split.

I find $k = 3$ and $k = 1$ to be optimal for the unaligned and aligned training sets, respectively, and the prediction accuracies obtained were $\simeq 0.519$ and $\simeq 0.937$ for the unaligned and aligned testing data, respectively.

To reproduce the results as well as the parameter search, please run ***`task_3_2()`*** *in the attached* ***`task_3.py`****.*

As expected, the classifier performed better on the aligned data, mispredicting only roughly 1 out of 20 cases, as opposed to the unaligned data where we saw mispredictions in roughly half of cases.

3.3 Loss of information in Procrustes

When we perform Procrustes, the standardization preserves only the handedness, relative position of landmarks, etc., but removes information about the relative sizes of the wings between different species of flies and different samples in the dataset. This means that predictions are made based on shape and relative distance between pairs of landmark points.

When it comes to classification, this is only a big problem if there exists two or more species of flies for which the wings are similar in all but the size – in other words, if the distinguishing feature happens to be the size, not the shape.

To examine whether this *could* be a potential source of error for this particular data set, we can examine the mean size of the wing samples across each species. Figure fig. 3 shows the mean wing size for each species, where wing size is quantified by measuring the length of the closed curve given by the points in a wing sample.

From the plot we see that three of the 15 species are outliers when it comes to the mean wing size – namely the albizona, tecticauda, and warreni species – and that 30 out of the total 261 samples belong to these classes. This number is not so high that I expect it to outweigh the benefits in accuracy obtained from Procrustes.

However, please note that two species for which the mean wing size is very different may still be easily distinguishable from one another after standardization if, as explained, they are also different in features/landmarks.

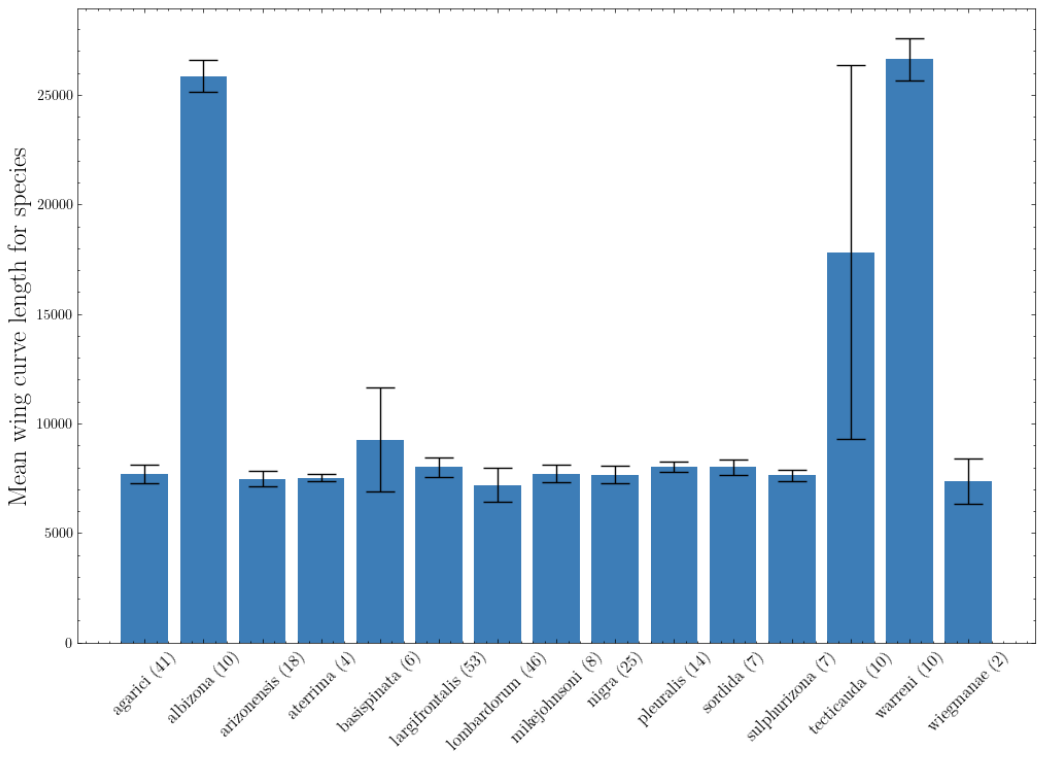

Figure 3: Mean wing curve length for each species measured on the combined training and test sets. The number in parentheses after each species name is the number of samples belonging to that species.

4 Scale-space feature detectors

Listing 5 shows my function for computing (single-scale) Harris responses using the formula for R given in the assignment text. The function uses `gaussian_filter` from `scipy.ndimage` which allows for easy convolution with (partial derivatives of) Gaussian kernels. Using this, the function is a quite straight-forward implementation of the formulas given in equation (2) and (3).

```python
from skimage.util import img_as_float
from scipy.ndimage import gaussian_filter
...
def harris_response(img, sigma, alpha = 0.05, k = 1):
    img = img_as_float(img)

    Lx  = gaussian_filter(img, sigma, order = [1, 0])
    Ly  = gaussian_filter(img, sigma, order = [0, 1])
    Axx = gaussian_filter(Lx ** 2, k * sigma)
    Axy = gaussian_filter(Lx * Ly, k * sigma)
    Ayy = gaussian_filter(Ly ** 2, k * sigma)

    det_A   = Axx * Ayy - Axy ** 2
    trace_A = Axx + Ayy
    R = sigma ** 4 * (det_A - alpha * trace_A ** 2)
    return R
```

Listing 5: Code for (single-scale) Harris responses using the formula for R given in the assignment text.

Next, listing 6 shows the wrapper code I use to compute multi-scale corner detection. It is essentially just wrappers for the Harris response function in listing 5. Note that I use `peak_local_max` to compute local maxima, and that I simply use the default `min_distance = 1`.

```python
def multi_scale_harris_response(img, sigmas, alpha = 0.05, k = 1):
    return np.array([harris_response(img, sigma, alpha, k)
                     for sigma in sigmas])

def multi_scale_corner_detection(img,
                                 sigmas = np.logspace(0, 5, 30, base = 2),
                                 num_peaks = 350,
                                 alpha = 0.05,
                                 k = 1):
    responses = multi_scale_harris_response(img, sigmas, alpha, k)
    peaks = peak_local_max(responses, num_peaks = num_peaks).astype(float)
    peaks[:, 0] = sigmas[peaks[:, 0].astype(int)]
    return peaks
```

Listing 6: Code for multi scale corner detection.

Running corner detection

As I understand the assignment we only need to test using a single k, but multiple values for α. Hence I choose $k = 1.5$, since I found it to be optimal in most cases. In addition, I try various values of α between 0.005 and 1, including 0.05 as requested. For the range of σ-values, I use simply the formula suggested in the assignment text (`np.logspace(0, 5, 30, base = 2)`).

Figure 4 shows the result of running multi-scale corner detection for these parameters. Interestingly, the detection more or less completely fails for $\alpha \geq 0.5$, while for the four smallest α values, the results are to some degree identical: A lot of the major corners are properly detected, while some of the smaller edges – especially along some of the various window panes – are missing.

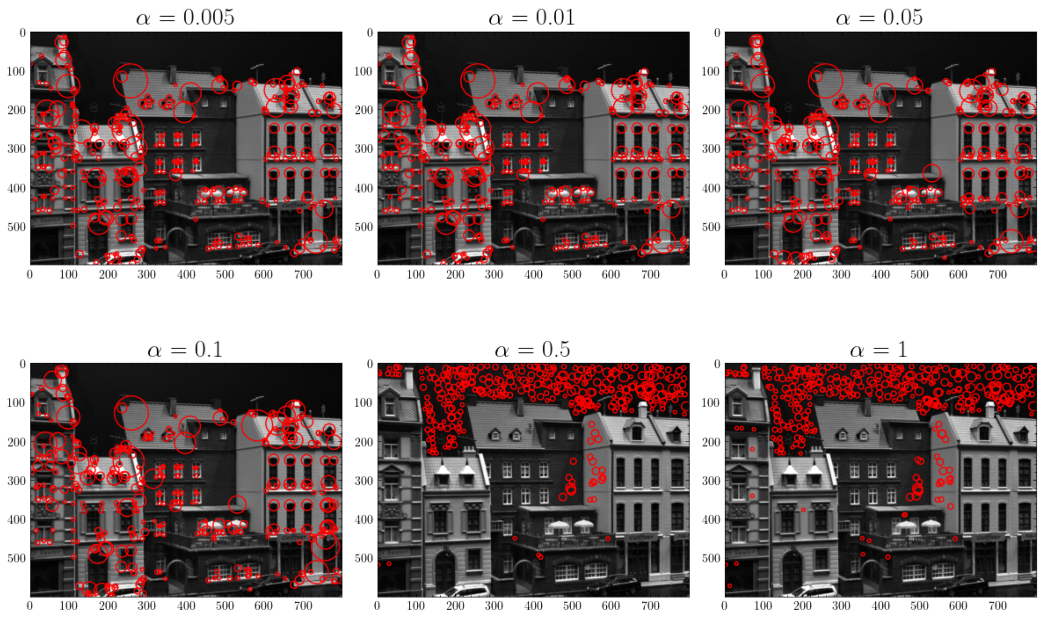

Figure 4: Multi-scale corner detection for $k = 1.5$ and varying α, with (up to) 350 local maxima for each case.

5 Histogram based segmentation

5.1 Three problems with intensity-based segmentation

Problem #1: global thresholding assumes that background and foreground objects are clearly distinguished by pixel intensities. An example of this is fig. 5, where the foreground object (the arm/hand, and possibly its shadow) is not clearly contrasting with the background, since the arm has pixels with intensities between ~ 50 to ~ 205, whereas the table background is mostly in the range ~ 120 to ~ 180.

In fact, the shading along the right edge of the arm, which we would expect to be part of the foreground object, is darker than most of the background, and hence this would be very difficult to segment correctly.

I attempt to segment the image using first a thresholding band in the range 50..90. This only succeeds at segmenting the shaded part of the arm. Adding another band in the range 105..120 segments a bit more of the arm and now also the shadow on the table, which may or may not be desirable. Finally, I add a third band in the range 185..205 in an attempt to better segment the light part of the arm, but this adds a lot of unwanted noise to the segmentation.

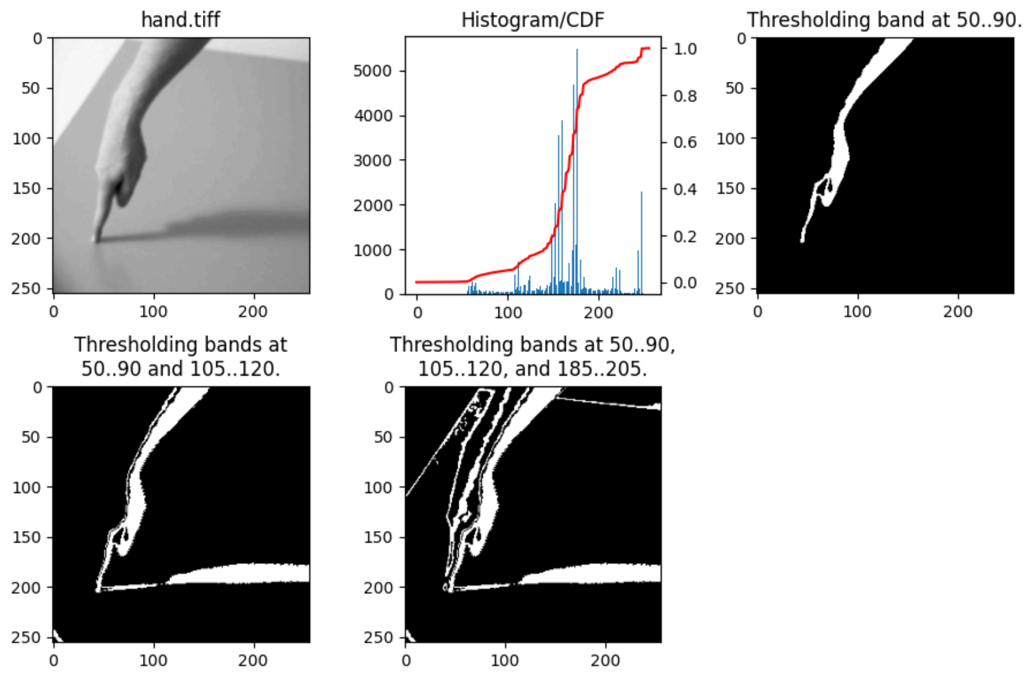

Figure 5: `hand.tiff` – example illustrating problem #1.

Problem #2: intensity-based segmentation can run into trouble with noise in images, since (white) noise is likely to appear in the histogram as foreground object pixels due to the high intensity pixels. Figure 6 shows an example of this. In addition, noise which appears in multiple different intensities can obscure the "valleys" in the histogram, making it (especially) difficult to determine thresholding values/bands.

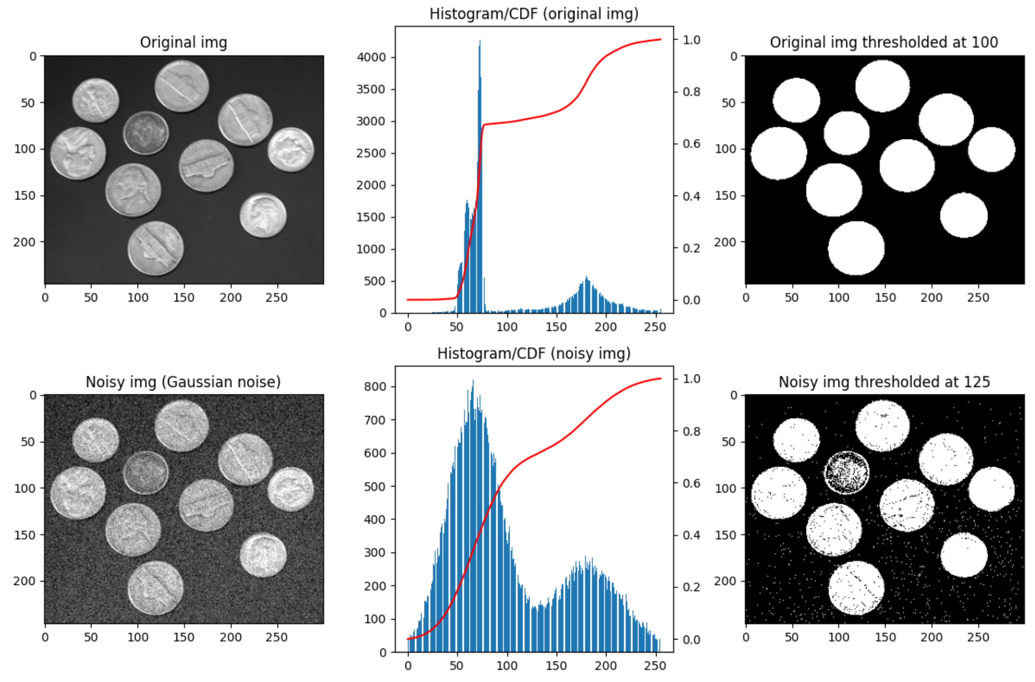

Figure 6: Threshold segmentation of `coins.png` with and without Gaussian noise – example of problem #2.

Problem #3 is segmentation tasks with multiple objects in different intensities and the manual work usually associated with them. In these cases we might see a necessity for multiple thresholding bands since a single catch-all threshold value that works for all foreground objects might accept unwanted parts of the background. Adding additional thresholding bands is in itself not hard, but it can be very tedious process and can eliminate the benefits of automatic segmentation.

Whenever we add a separate thresholding band, we add another dimension to the parameter search for the optimal thresholding values, and each time we add a thresholding band we possibly let some of the background bleed through (similar to what we saw in fig. 5).

5.2 Edge-based vs intensity-based segmentation

First of all, edge-based segmentation can in most cases solve problems #1 and #, since edge-based segmentation is (to some degree) ignorant of similarity between pixels inside a foreground object and the background, so long as there is a clearly defined edge surrounding the object. In fig. 5 we saw how the soft color gradient on the arm/hand in the x-direction caused trouble for thresholding, but since there are clearly defined edge between the arm and background, edge-based segmentation might succeed here. A similar thing goes for multiple foreground objects in different intensities, so long as they have clearly defined edges.

With respect to noisy images, edge-based segmentation has its own problems, since noise can obscure edges and even create spurious “extra” edges.

An example of where edge-based segmentation might perform badly are cases where the background itself has sharp edges, such as a tiled floor, and if the back- and foreground object happen to be clearly distinguished in intensity, then thresholding might give an even better result than edge-based.