- Assignment 7 -

Signal and Image Processing

Philip Rajani Lassen & Maria Luise da Costa Zemsch

March 24, 2019

Deconvolution and Texture Analysis

---

1 Inverse filtering

1.1

Listing 1: LSI Degredation Implementation

```
def degrade(kernel, image, image_noise):
  f = image
  f_fft = fft2(f)
  k_fft = fft2(kernel, shape = f.shape)
  i_fft = fft2(image_noise, shape = f.shape)
  result = ifft2((f_fft * k_fft) + i_fft)
  return result
```

The function degrade works in the Fourier domain. The convolution is applied in the Fourier domain and then the image of noise is added in the Fourier domain. Where F is the original image, K is the filter and N is the image of noise (all in the Fourier domain).

$$D(x, y) = F(x, y) * K(x, y) + N(x, y)$$

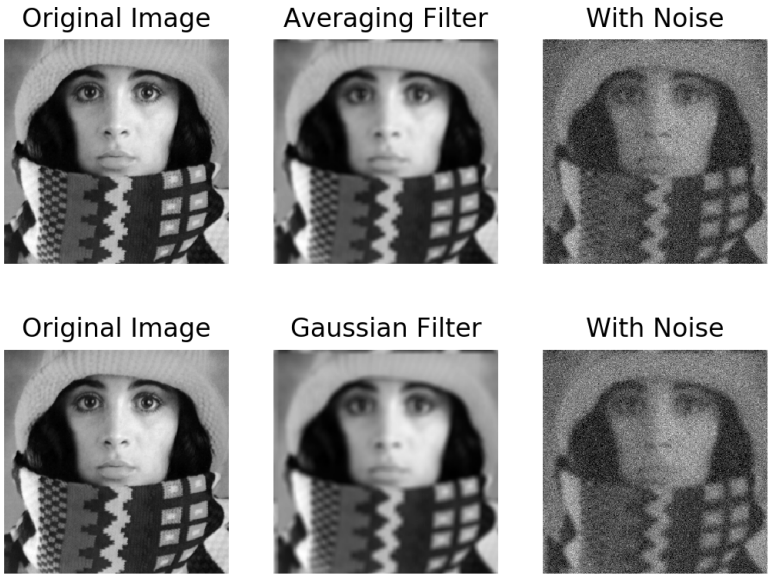

Figure 1: Original Image on the Left. First row is averaging filter , Second row, is Gaussian filter. second column is with ought noise and third column is with noise

Above is a visualization of how the degredation function works on trui.png for both the averaging filter and gaussian blur filter with different levels of noise.

1.2

Listing 2: Wiener Filtering Implementation

```
def inverse_filter(kernel, degraded_image, n):
  g = degraded_image
  h = kernel
  (rows, columns) = g.shape
  (height, width) = kernel.shape
  g_fft = fft2(g)
  h_fft = fft2(kernel, g_fft.shape)
  f_fft = g_fft / h_fft
  f_fft = np.clip(f_fft, n, np.max(f_fft))
  result = ifft2(f_fft)
  return result
```

The code calculates the inverse filter using the equations from the textbook. np.clip is used for thresholding.

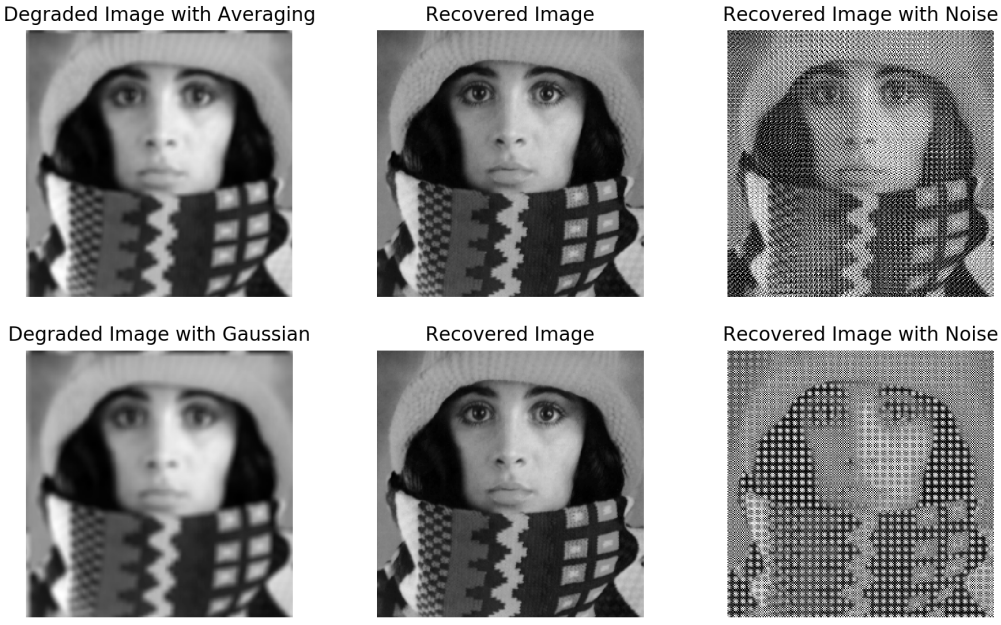

Figure 2: First row is Degraded with Averaging filter, while the second row is degraded with the gaussian filter

The inverse filter does a good job of capturing the original image when the image has been degraded with the averaging filter. However once the noise level get sufficiently high, it struggles to reconstruct the image. With reconstructing the degraded image with a gaussian,

the algorithm is very sensitive to noise.

1.3

Listing 3: Inverse Filtering Implementation

```
def weiner_filter(kernel, degraded_image, K):
  g = degraded_image
  h = kernel
  G = fft2(g)
  H = fft2(kernel, G.shape)
  result = (1 / H) * G * np.square(np.absolute(H)) / (np.square(np.absolute(H)) + K)
  return ifft2(result)
```

The code performs the calculation from the textbook using the closed form equations.

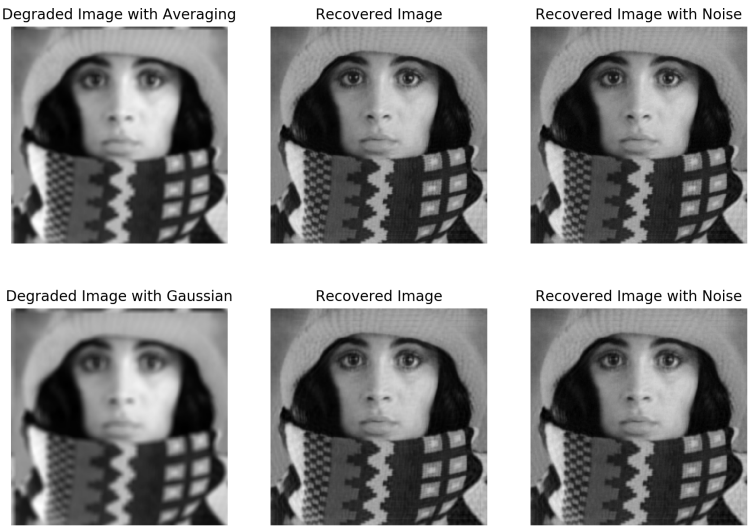

Figure 3: First row is Degraded with Averaging filter, while the second row is degraded with the gaussian filter, and columns 2 and 3 are recovered using the weiner filter

The Weiner filter does a far better job than the inverse filter. It is more robust to noise. The image is also not clouded as it is in the previous inverse filter.

However once noise get sufficiently high we can see that it again struggles to reconstruct the image. The diagram also shows changing the power spectrum and varying K does alter the performance of the wiener filter, and needs to be changed for optimal performance based on the kernel and noise level.

2 Texture analysis

2.1 Intensity histograms and χ^2-distance of texture images

The normalized intensity histogram can be computed with the function `skimage.exposure.histogram`. The result can be seen in figure 4.

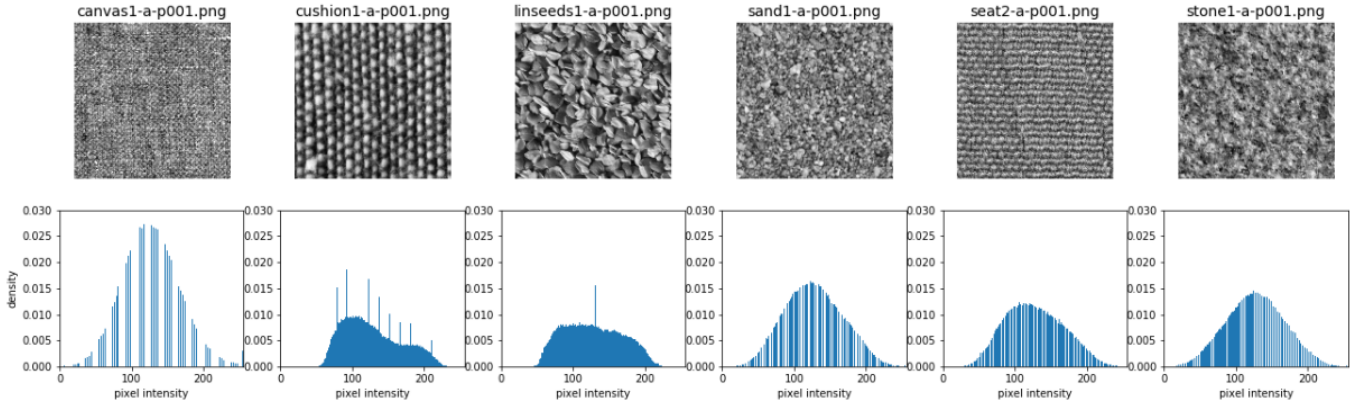

Figure 4: Normalized intensity histograms of six example texture images

The χ^2-distance is defined by the function 1.

$$\chi^2(h,g) = \sum_{i=1}^{n} \frac{(h_i - g_i)^2}{h_i + g_i} \tag{1}$$

To compute the χ^2-distance between each pair of the six histograms we need to use the normalized histograms (function `norm_hist()`).

Listing 4: Normalized histogram

```
def norm_hist(histo):
  summe = sum(histo[0])
  return( histo[0]/summe, histo[1] )
```

Furthermore the method `skimage.exposure.histogram` does not return the zero values. Therefore we still need to fill up the array with the missing values, this is why the method `fill_up_arr()` was implemented. If the method `numpy.histogram()` is used instead of `skimage.exposure.histogram`, the array does not need to be filled up. Thus the χ^2-distance between two histograms can be computed using equation 1. Attention should be paid to the case where h_i and g_i are both zero and the denominator is zero as well. In this special case the output was set to 0.

Listing 5: χ^2-distance of two texture images

```
def chi2_distance(histo1, histo2):
  h = fill_up_arr(histo1, 255); g = fill_up_arr(histo2, 255)
```

```
output = []
for i in range(256):
  if g[i]+h[i] != 0:
    output.append((h[i]-g[i])**2/(h[i]+g[i]))
return(sum(output))
```

In table 1 you can see the result of the comparison of all six histograms. The method shows the highest resemblance between cushion and linseeds (0.07) and the biggest difference between the sand and the canvas (1.17). In this six cases the different textures cannot be distinguished by the χ^2-distance. To mention a negative example: the texton stone and sand, which are optically quite similar, have a χ^2-distance of 0.69, while cushion and linseeds have a smaller distance between them, but differ more optically.

Table 1: χ^2-distance of the six example images seen in figure 4

	canvas1	**cushion1**	**linseeds1**	**sand1**	**seat2**	**stone1**
canvas1	0	0.9725	0.9411	1.1696	0.9849	1.0348
cushion1	0.9725	0	0.0697	0.5517	0.3339	0.4813
linseeds1	0.9411	0.06974	0	0.5349	0.3134	0.4192
sand1	1.1696	0.5517	0.5349	0	0.6590	0.6900
seat2	0.9849	0.3339	0.3134	0.6590	0	0.6070
stone1	1.0348	0.4813	0.4192	0.6900	0.6070	0

2.2 MR8 filter bank

2.2.1 Creating the filter bank

The MR8 filter bank consists of 8 filters including the the Gaussian filter, the Laplacian of Gaussian filter and the oriented Gaussian derivative filters up to order 2. The derivative filters are used for 3 different scales, thus we get in total 8 response images. The filter response image is the filter bank applied to an image, which is given by equation 1 to 4 in the assignment sheet. The Gaussian kernels for the convolution are implemented in a separate method `Gaussian(x, y, n, m, sigma)`, which takes the size of the kernel in the x- and y-direction `x, y`, the standard derivation `sigma` and the order of the derivative `n` in the x-direction and `m` in the y-direction. It only works for the needed orders of derivatives.

Listing 6: χ^2-distance of two texture images

```
def Gaussian(x, y, n, m, sigma):
  x_axis = np.arange(-x//2 +1, x//2 +1); y_axis = np.arange(-y//2 +1, y//2 +1)
  xx, yy = np.meshgrid(x_axis, y_axis)
  gauss = 1/(2.*np.pi*sigma**2)*np.exp(-(xx**2 + yy**2) / (2. * sigma**2))
  if n==0 and m==0:
    gauss = gauss
```

```
  elif n==1 and m==0:
    gauss = -xx/sigma**2 * gauss
  elif n==0 and m==1:
    gauss = -yy/sigma**2 * gauss
  elif n==2 and m==0:
    gauss = 1/sigma**2 * (xx**2/sigma**2 -1) * gauss
  elif n==0 and m==2:
    gauss = 1/sigma**2 * (yy**2/sigma**2 -1) * gauss
  elif n==1 and m==1:
    gauss = xx/sigma**2 * yy/sigma**2 * gauss
  else:
    gauss = None
  return(gauss)
```

To compute the Gaussian response images $L(x, y; \sigma)$, we only convolved the image with the Gaussian kernel (equation 1 in the assignment sheet). For the Laplacian $\nabla^2 L(x, y; \sigma)$ (equation 2 in the assignment sheet), we added the second order derivative for x and y. The convolution with a Gaussian kernel is implemented in the function `Lxy(img, x, y, n, m, sigma)`. Both were used with $\sigma = 10$ *pixels*.

Listing 7: $L_{x^n,y^m}(x, y, \sigma)$

```
def Lxy(img, x, y, n, m, sigma):
  kernel = Gaussian(x, y, n, m, sigma)
  response_img = ifft2(fft2(img) * fft2(kernel, img.shape))
  return np.real(response_img)
```

The oriented 1st and 2nd order derivatives are computed by the functions `Lv(img, x, y, sigma, theta)` and `Lvv(img, x, y, sigma, theta)`, where `theta` represents the rotation angle. All of them were applied with three σ values: $\sigma = 1, 2, 4$ *pixels*

Listing 8: $L_v(x, y, \sigma)$ and $L_{vv}(x, y, \sigma)$

```
def Lv(img, x, y, sigma, theta):
  response_img = np.cos(theta) * Lxy(img, x, y, 1, 0, sigma)
  + np.sin(theta) * Lxy(img, x, y, 0, 1, sigma)
  return response_img

def Lvv(img, x, y, sigma, theta):
  response_img = np.cos(theta)**2 * Lxy(img, x, y, 2, 0, sigma)
  + np.sin(theta)**2 * Lxy(img, x, y, 0, 2, sigma)
  + 2 * np.cos(theta) * np.sin(theta) * Lxy(img, x, y, 1, 1, sigma)
  return response_img
```

To get the response image we still need to get the maximum pixel value for every angle. This is done with the functions `max_Lv(img, x, y, sigma, theta_arr)` and `max_Lvv(img, x, y, sigma, theta_arr)`, where instead of one `theta` value you give the method the array of values.

Listing 9: maximal value of $L_v(x, y, \sigma)$ and $L_{vv}(x, y, \sigma)$

```
def max_Lv(img, x, y, sigma, theta_arr):
  Lv_images = np.zeros((img.shape[0], img.shape[1], len(theta_arr)))
  for t in range(len(theta_arr)):
    Lv_images[:, :, t] = Lv(img, x, y, sigma, theta_arr[t])
  Lv_max = np.max(Lv_images, axis = 2)
  return Lv_max

def max_Lvv(img, x, y, sigma, theta_arr):
  Lvv_images = np.zeros((img.shape[0], img.shape[1], len(theta_arr)))
  for t in range(len(theta_arr)):
    Lvv_images[:, :, t] = Lvv(img, x, y, sigma, theta_arr[t])
  Lvv_max = np.max(Lvv_images, axis = 2)
  return Lvv_max
```

These methods were applied to the example image `linseeds1-a-p001.png` for the angles $\theta = 0, \pi/6, 2\pi/6, 3\pi/6, 4\pi/6,$ and $5\pi/6$ return the 8 filter response images (see figure 5). The kernel size was chosen depending on σ, which is proportional to the radius. The general rule of the size of a gaussian kernel is $N = 3\sigma$. As we are using the derivatives we need a kernel twice as large as before: $N = 6\sigma$.

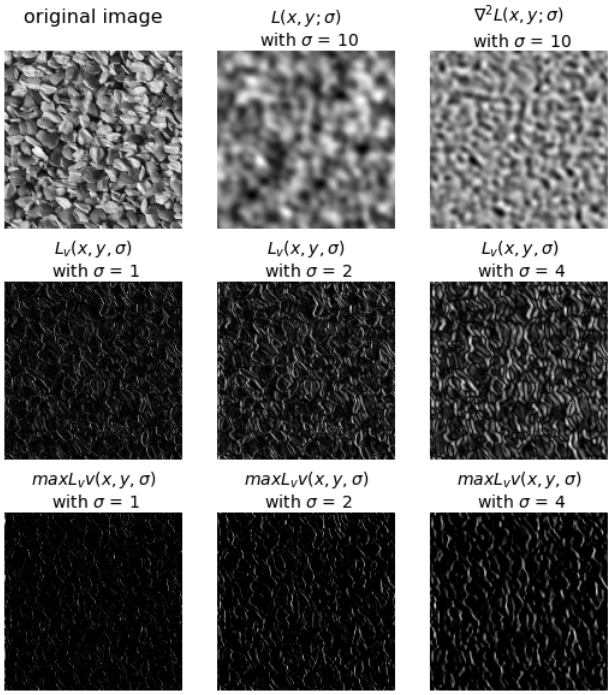

Figure 5: 8 response images of the example image `linseeds1-a-p001.png` for the MR8 filter bank

2.2.2 Learning a set of textons

As a classifier we used the K-means algorithm with K = 60 (`sklearn.cluster.KMeans`). To train our classifier we needed to reshape the data. We looked at each pixel of each image separately, while they where shaped as an array of the length of 8 (one value for each MR8 response image). Thus we flatten each image and put them in an array one after another.

Listing 10: Reshaping the training data and train the classifier

```
X = np.zeros((6*576*576, 8)) #shape training data: [1990656, 8]
for i in range(8):
  for n in range(6):
    X_img = textures_MR8[n, :, :, i].flatten()
    number = len(X_img)
    X[n*number : n*number+number, i] = X_img
kmeans = KMeans(n_clusters=60).fit(X)
```

Now the training data has the shape of $[6 * width * height,\ 8] = [1990656,\ 8]$.

2.2.3 texton histograms

Because the K-Mean is an unsupervised classifier we already get the labels of each pixel due to the training process. To create the histograms we split the label array in 6 pieces (each array resembles one texton class) and plotted them (see figure 6).

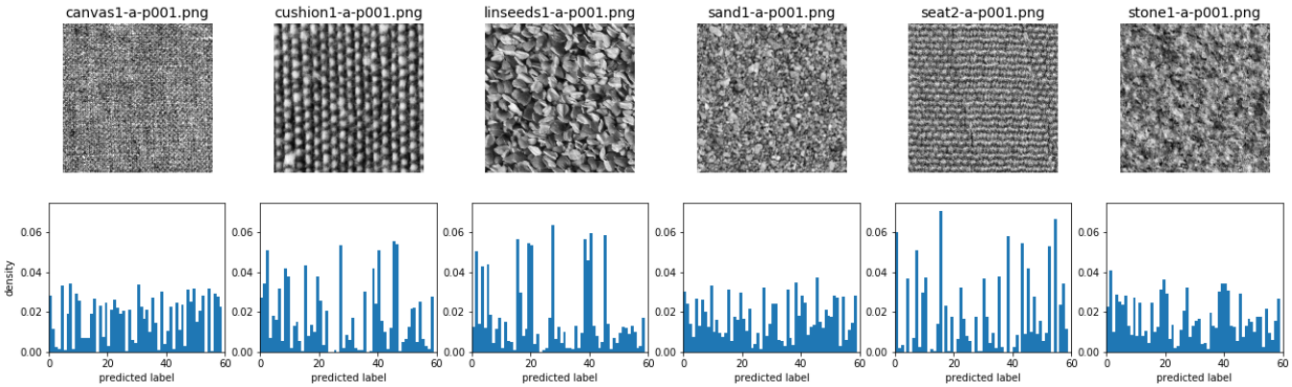

Figure 6: Texton histology diagrams for each of the six texton images. The histograms show the frequency of each predicted label by the K-Means algorithm.

The method from section 2.1 `chi2_distance()` was used to compute the χ^2-distances between each texton histogram, which are listed in table 2.

Table 2: χ^2-distance of the six example images seen in figure 4

	canvas1	**cushion1**	**linseeds1**	**sand1**	**seat2**	**stone1**
canvas1	0	0.6421	0.7821	0.2199	0.2390	0.3979
cushion1	0.6421	0	0.2322	0.2221	0.7433	0.2914
linseeds1	0.7821	0.2322	0	0.3435	0.9107	0.2076
sand1	0.2199	0.2221	0.3435	0	0.4255	0.1029
seat2	0.2390	0.7433	0.9107	0.4255	0	0.6507
stone1	0.3979	0.2914	0.2076	0.1029	0.6507	0

The method shows a highest resemblance between stone and sand (0.10) and the biggest difference between the linseeds and the seat (0.91). In general the output reflect really good the differences between each class. Thus the χ^2-distance can be used as a method for differ textons, if you used it on the texton histology diagrams and not on the intensity histograms.

2.2.4 Test the Texton histogram prediction

To test if this procedure is capable of predicting the correct texton class of an image we computed the texton histogram of the image `cushion1-a-p012.png` the same way as

before (predict the labels with K-Means and plot the texton histology histogram). In figure 7 you can see the image and his texton histogram.

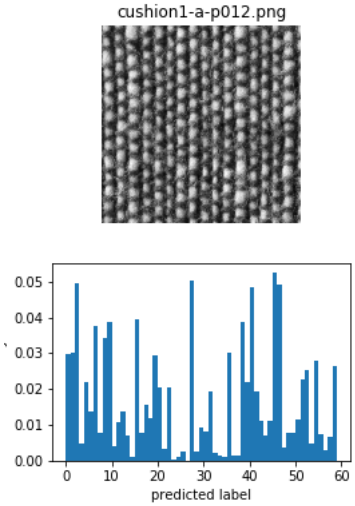

Figure 7: Texton histology diagrams for the `cushion1-a-p012.png`. The histograms show the frequency of each predicted label by the K-Means algorithm. It is used to predict one of the texon classes shown in figure 6

The closest matching texton histogram among the six shown in figure 6 can be found by computing the χ^2-distance. The result is shown in table 3.

Table 3: χ^2-distance of the six example images seen in figure 4

	canvas1	**cushion1**	**linseeds1**	**sand1**	**seat2**	**stone1**
cushion1_test	0.5282	0.0228	0.2539	0.1471	0.6357	0.2266

Thus we get the highest resemblance with the cushion1 class, which should be the case. Seat1 and canvas1 are the two classes which are the furthest from cushion1 with respect to the χ^2-distance. This behaviour can also be seen in table 2, where the class cushion has the least resemblance with those two.