Visual Computing Fundamentals
Image Processing
Assignment 2

Adrian Opheim

November 10, 2017

Task 1: Theory Question

a) The convolution theorem can be seen in Equation 1, where $\mathcal{F}$ is the Fourier transform, $*$ is the convolution operator and $\cdot$ is pointwise multiplication.

i) What does the convolution theorem entail?
The convolution theorem entails that convolution is commutative, because the same result would be obtained if the order of f and g were reversed. The convolution theorem also states that convolution in the frequency domain is analogous to multiplication in the spatial domain. The two are being related by the forward and inverse Fourier transforms, respectively.

$$\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\} \tag{1}$$

ii) What are the main steps an implementation of frequency filtering requires?
The basic filtering equation has the form:

$$g(x, y) = \mathcal{F}^{-1}\left[H(u, v) \cdot F(u, v)\right] \tag{2}$$

where $\mathcal{F}^{-1}$ is the Inverse Discrete Fourier Transform (IDFT), $F(u, v)$ is the Discrete Fourier Transform (DFT) of the input image $f(x, y)$. $H(u, v)$ is a filter function and $g(x, y)$ is the filtered output image. A step-by-step procedure of filtering in the frequency domain is given here: [1]

(i) Given an input image of size $M \times N$, obtain the padding parameters P and Q from $P \geq 2M - 1$ and $Q \geq 2N - 1$. Typically, we select $P = 2M$ and $Q = 2N$.

(ii) Form a padded image, $f_p(x, y)$ of size $P \times Q$ by appending the necessary number of zeros to $f(x, y)$.

(iii) Multiply $f_p(x, y)$ by $(-1)^{x+y}$ to center its transform.

(iv) Compute the DFT, $F(u, v)$, of the image from step 3.

(v) Generate a real, symmetric filter function, $H(u, v)$ of size $P \times Q$ with center at coordinates $(P/2, Q/2)$. Form the product $G(u, v) = H(u, v)F(u, v)$ using array multiplication; that is, $G(i, k) = H(i, k)F(i, k)$

(vi) Obtain the processed image:

$$g_p(x, y) = \left\{\text{real}\left[\mathcal{F}^{-1}\left[G(u, v)\right]\right]\right\}(-1)^{x+y} \tag{3}$$

[1] Taken from page 285 of *Digital Image processing*

where the real part is selected in order to ignore parasitic complex components resulting from computational inaccuracies, and the subscript p indicates that we are dealing with padded arrays.

(vii) Obtain the final processed result $g(x, y)$ by extracting the $M \times N$ region from the top, left quadrant of $g_p(x, y)$.

b) Convolution kernels are typically understood in terms of the frequency domain. In your own words, explain what high- and low-pass filters are.

A low-pass filter is a filter that discards high frequencies obtained from the DFT, while passing low frequencies. A low-pass filter blurs an image.
A high-pass filter has the opposite property: It discards low frequencies and passes high frequencies. A high-pass filter would therefore enhance sharp detail, but cause a reduction in contrast in the image.

c) The Fourier spectrum $|\mathcal{F}\{f\}|$ of tree commonly used convolution kernels can be seen in Figure 1. For each kernel, write down what kind of kernel it is. Explain your reasoning.

(a) (b) (c)

Figure 1: Three different convolution kernels

Kernel (a) is a low-pass filter, probably a Gaussian Lowpass-filter, as we have no reoccurring white circles.
I would assume (b) and (c) are both high-pass filters, because they are both rejecting low frequencies in the middle, and passes high frequencies further away from the origo.

d) Answer the following:

i) *Explain why padding is important when we want to do frequency filtering.*
If the images are not padded, we could expect a wraparound error. Meaning that the filtering will have no effect, as the filter will meet light areas in both the image and in its right neighbour.

ii) *In the context of frequency filtering, we commonly pad images by appending zeros to the rows and columns as seen in Figure 2. Let' s assume we instead pad the images by prepending zeros to the rows and columns. This can be seen in Figure 2 (b). Let the number of zeros we pad by remain the same. Does this change the result of frequency filtering? Explain your reasoning.*
This will not change the result of frequency filtering, as you can avoid the wraparound error by appending or prepending zeros to the image. The number of zeros should satisfy $P \geq A + B - 1$, where P is the total length of functions f and h, composed of A and B samples. If each function has 400 points, so the minimum value that can be used is $P = 799$, which implies that we could append or prepend 399 zeros to each function.

(a) (b)

Figure 2: (a) shows appending zeros, (b) shows prepending zeros.

Task 2: Frequency Domain Filtering

a) *Using an already existing FFT implementation, implement a function that takes a greyscale image and an arbitrary linear convolution kernel and performs frequency filtering. Apply the function you just made on an appropriate greyscale image with both a high- and a low-pass convolution kernel. It's up to you to choose which convolution kernels to use.*
The chosen image and the following greyscale conversion, using the luminance-preserving method used in Assignment 1, is shown in Figure 3

(a) Original image (b) Greyscale conversion

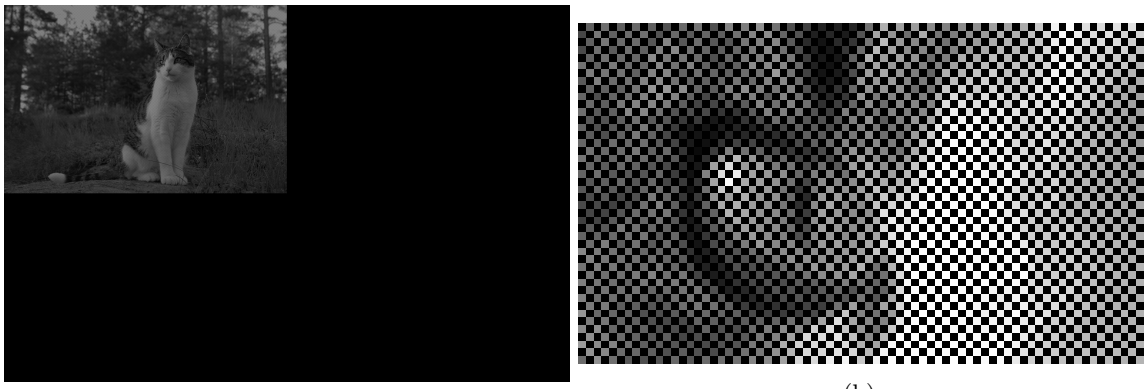
Figure 3: Original image and corresponding greyscale conversion. The image is of size (997×1500) pixels

Step (i) Obtaining padding parameters
As the grayscale image is of dimensions $(M \times N) = (997 \times 1500)$, the padding parameters are set to $P = 2M = 1994$ and $Q = 2N = 3000$.

Step (ii) Padding the image
Forming a padded image of size $P \times Q$, by appending zeros. The corresponding image can be seen in Figure 4

Figure 4: The padded image of size 1994×3000

Step (iii) Centering the image's transform
After centering the image $f_p(x, y)$ with $(-1)^{x+y}$, we get the image shown in Figure 5



(a) (b)

Figure 5: (a) shows the complete, centered image. (b) shows an even further zoom, making it possible to see individual pixels that are changed to alternating black, and original grayscale value. The black pixels are pixels set to a value of zero (black), for visualisation purposes, as the pixel's actual value is the negative of the original pixel value.

Step (iv) Computing the Fourier transform The Fourier transform of the image can be seen in Figure 6

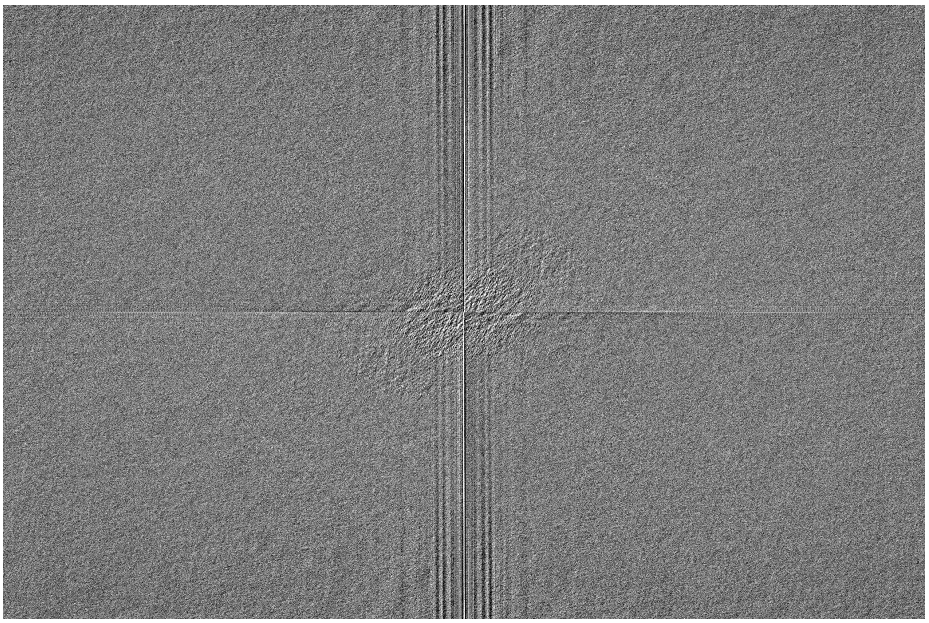
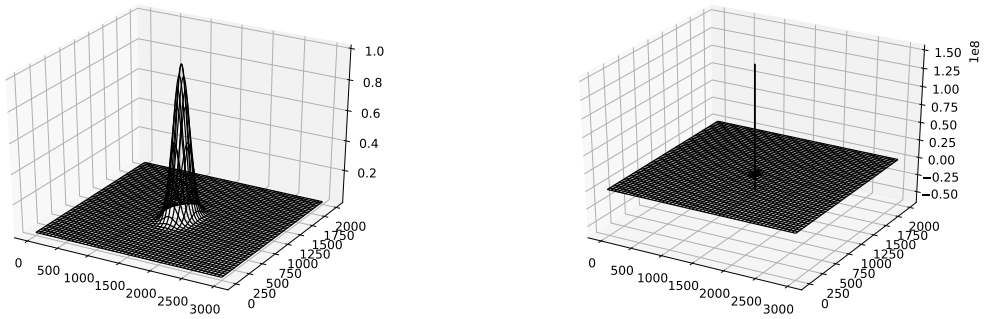


Figure 6: The Fourier transform of the padded image.

Step (v) Generating a filter
A Gaussian lowpass filter of size $(P \times Q)$ is generated. The parameters used are $D_0 = 100$. The generated lowpass filter can be seen in Figure 7 (a), and the product $G(u, v) = H(u, v)F(u, v)$ can be seen in Figure 7 (b).



(a) (b)

Figure 7: (a) shows the generated Gauss lowpass kernel. (b) shows the product $G(u, v) = H(u, v)F(u, v)$.

Step (vi) Obtaining the processed image

The processed image can be seen in Figure 8

Figure 8: The inverse Fourier transform of the padded image.

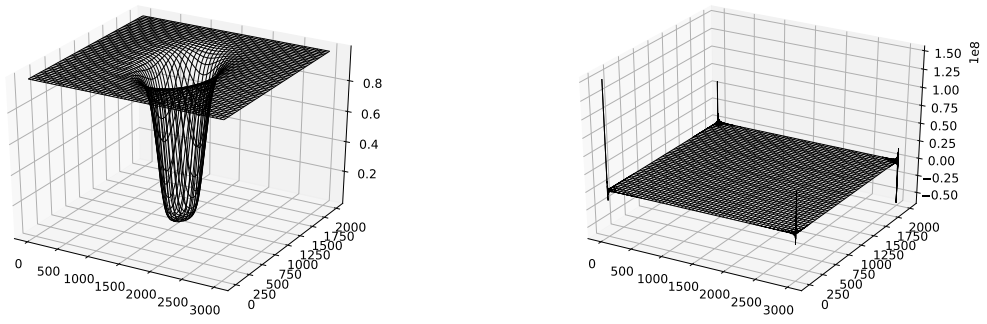
Step (vii) Obtaining the image of size $(M \times N)$
The filtered image can be seen in Figure 9

(a) (b)

Figure 9: (a) shows the processed image after low-pass filtering, while (b) shows the original grayscale image.

Applying a high-pass filter

A Butterworth high-pass filter is also implemented. The high-pass filter is useful for sharpening images. The generated high-pass Butterworth filter is shown in Figure 12



(a) (b)

Figure 10: (a)shows the generated Butterworth filter with $D_0 = 300, n = 3$ (b) shows the spectrum of the Fourier transformed image, $F(x, y)$. Note that this spectrum is not centered, as I did not get the high-pass filter to work with the centered image.

Figure 11 shows the processed image after the high-pass filtering, compared to the original grayscale image.

(a) (b)

Figure 11: (a) shows the processed image with a high-pass Butterworth filter, while (b) shows the original grayscale image.

Task 3: Unsharp Masking

a) *Implement a function that performs unsharp masking on a greyscale image using frequency filtering. Instead of implementing Equation 3 directly you must compute a single kernel, which when used with frequency filtering will yield the desired sharpened image. We recommend that you use Gaussian smoothing when creating the unsharp kernel.*

(a) (b)

Figure 12: (a) shows the original color image used for unsharp masking. (b) shows the corresponding grayscale image, generated using the luminance-preserving method.

I did not completely understand what was being meant by "computing a single kernel". But I have implemented the procedure in the book, going as follows:

$$g(x,y) = f(x,y) + k \cdot g_{\text{mask}}(x,y)$$

where

$$g_{\text{mask}} = f(x,y) - f_{LP}(x,y)$$

and

$$f_{LP}(x,y) = \mathcal{F}^{-1}\left[H_{LP}(u,v)F(u,v)\right]$$

... and this was how far I got before the deadline, unfortunately. Very much time was spent fiddling with NumPy and with exporting figures. I will try to use my time smart for the next assignment ;)

For future reference, my current code is added below. It is very much unpolished.

```
1  import numpy as np       #n-D arrays
2  import matplotlib.pyplot as plt              # For plotting
3  from mpl_toolkits.mplot3d import axes3d # For 3D plotting
4  from PIL import Image
5  import math
6
7
8  def img_save(img, filename):
9      print("Saving...")
10     img.save(filename, "PNG")
11
12 def img_show(img):
13     plt.figure()
14     plt.imshow(img, cmap='gray')      # have to use the gray color map.
15     plt.axis('off')
16     plt.show()
17
18
19 def conv2Grey_lum(img):
20     img_arr = np.array(img) # convert from image object to array
21     r = img_arr[:,:, 0] # equivalent to img[...,0]
```

```
22      g = img_arr[:,:, 1]
23      b = img_arr[:,:, 2]
24
25      grey = np.zeros((len(r), len(r[0])), np.uint8)
26      print("Converting to grey...")
27      for row in range(0, len(r)-1):
28          for col in range(0, len(r[0])-1):
29              grey[row][col] = int(0.2126*r[row][col] + 0.7152*g[row][col]
                    + 0.0722*b[row][col])
30
31      img = Image.fromarray(grey) # converting back to image object
32      return img
33
34
35
36  def freq_filter(img):
37
38      def pad_image(img_arr):
39          print("Padding image...")
40          img_padded = np.zeros((P, Q), dtype = np.uint8)
41          for i in range(0, M):
42              for j in range(0, N):
43                  img_padded[i][j] = img_arr[i][j]
44          return img_padded
45
46      def plot_wireframe(kernel, filename):
47
48          u = np.zeros((P, Q), dtype=np.int)
49          v = np.zeros((P, Q), dtype=np.int)
50          j_counter = 0
51          for i in range(0, P):
52              for j in range(0, Q):
53                  u[i] = np.arange(Q)
54                  v[i].fill(j_counter)
55              j_counter += 1
56
57          #print("u: ", u, "\nv: ", v, "\nkernel: ", kernel)
58          #print("u: ", u.shape, "v: ", v.shape, "kernel: ", kernel.shape)
59          fig = plt.figure()
60          ax = fig.add_subplot(111, projection='3d')
61          lines = ax.plot_wireframe(u, v, kernel)
62          plt.setp(lines, color='black', linewidth=0.5)
63          plt.show()
64          fig.savefig(filename, format='pdf') #saving as pdf
65
66
67      def D(u, v): # Function for calculating the distance between a point
            (u, v) in the frequency domain and the center of the frequency
            rectangle
68          term1 = (u - P/2)*(u - P/2)
69          term2 = (v - Q/2)*(v - Q/2)
70          #print(math.sqrt(term1 + term2))
71          ans = math.sqrt(term1 + term2)
```

```
72          # Not possible to divide by zero:
73          if (ans < 10e-7):
74              return 10e-7
75          else:
76              return ans
77
78      def lowpass_gauss(D0):
79          H = np.zeros((P, Q), dtype=np.float) # H is the kernel --> H(u,
                v)
80          print("Creating Gauss kernel...")
81          for u in range(0, P):
82              for v in range(0, Q):
83                  teller = -(D(u, v) * D(u, v))
84                  nevner = 2 * (D0 * D0)
85                  eksp = teller / nevner
86                  H[u][v] = math.exp(eksp)
87
88          #print(H.shape)
89          #plot_wireframe(H)
90          #print("H: ", H)
91          #img = Image.fromarray(np.real(H))
92          #img = img.convert('RGB')
93          #img_save(img, "./img_proc/lowpass_gauss.png")
94          return H
95
96
97      def highpass_butterworth(D0, n):
98          H = np.zeros((P, Q), dtype=np.float) # H is the kernel --> H(u,
                v)
99          print("Creating high-pass Butterworth kernel...")
100         for u in range(0, P):
101             for v in range(0, Q):
102                 teller = 1
103                 nevner = D0 / (D(u, v))
104                 nevner = math.pow(nevner, 2*n)
105                 nevner = 1 + nevner
106                 H[u][v] = teller / nevner
107
108         plot_wireframe(H, './plots/highpass_butterworth.pdf')
109         return H
110
111
112
113
114
115     def center_transform(img):
116         print("Centering the image...")
117         #print("Image before centering: ", img)
118         centered = np.zeros((P, Q), dtype=np.float) # Have to allow
                negative values
119         for x in range(0, P):
120             for y in range(0, Q):
121                 eksp = x + y
```

```
122                 centered[x][y] = img[x][y] * math.pow(-1, eksp)
123         #print("Image after centering: ", centered)
124         return centered
125
126     def inv_fourier(G):
127         print("Inverse Fourier transform...")
128         g_p = np.fft.ifft2(G) # performing inverse ourier transform...
129         g_p = np.real(g_p) # extracting the real part...
130         g_p = center_transform(g_p) # Centering
131         g_p = g_p.astype(np.uint8) # Converting back to uint8 type
132         return g_p
133
134     def plot_fourier(F):
135         print(F)
136         # Extracting the real part of the transform:
137         F = np.fft.fftshift(F)
138         F_log = np.log2(1 + np.abs(F)**2)
139
140         print(F_log)
141         print("max: ", np.max(F_log))
142
143         img = Image.fromarray(F_log)
144         img = img.convert('RGB')
145         img_show(img)
146
147
148
149
150
151     img_arr = np.array(img)
152     img_arr = img_arr[:, :, 0] # Removing color bands that are still
            there.
153
154     # Getting image parameters: Original image of size (M, N)
155     M = len(img_arr)
156     N = len(img_arr[0])
157     # Size of the filter kernel:
158     P = 2*M
159     Q = 2*N
160     print("Dimensions of the image: \nM: ", M, "N: ", N, "P: ", P, "Q: "
            , Q)
161
162     # Padding the image:
163     img_padded = pad_image(img_arr)
164
165     # Centering its transform:
166     img_centered = center_transform(img_padded)
167
168
169
170     # Computing the Fourier transform:
171     print("Computing Fourier transform...")
172     img_fourier = np.fft.fft2(img_padded)
```

```
173     plot_fourier(img_fourier)
174
175
176     # Creating a kernel:
177     kernel = lowpass_gauss(1e2)
178     #kernel = highpass_butterworth(3e2, 3)
179
180     # Multiplying element-wise kernel with Fourier transformed image:
181     print("Multiplying kernel with Fourier image...")
182     img_G = np.multiply(kernel, img_fourier)
183     plot_fourier(img_G)
184
185
186     # Obtaining the finished processed image:
187     img_proc = inv_fourier(img_G)
188
189
190     # Clipping out the original (M, N) sized picture:
191     img_proc = img_proc[0:M, 0:N]
192     img = Image.fromarray(img_proc)
193     img = img.convert('RGB')
194     img_show(img)
195
196     # Subtracting the low-pass image from the original grayscale:
197     g_mask = np.subtract(img_arr, img_proc)
198
199
200     # Multiplying the mask with a k value
201     g_mask = np.multiply(1, g_mask)
202
203     g = np.add(img_arr, g_mask)
204     img = Image.fromarray(g)
205     img_save(img, "./img_proc_2/img_g_k1.png")
206
207
208     return img
209
210
211 def main():
212     '''
213     # Creating the greyscale image:
214     img = Image.open("./images/DSC_0784.JPG")
215     img_grey = conv2Grey_lum(img)
216     img_save(img_grey, "./img_proc_2/img_grey.png")
217     '''
218
219     img = Image.open("./img_proc/img_sushi_grey_mini.png")
220     #img = Image.open("./img_proc_2/img_grey_mini.png")
221     #img = Image.open("./images/noise-a.tiff")
222
223
224     ## Filtering in the frequency domain: ##
225     img_freq_filter = freq_filter(img)
```

```
226
227
228
229
230
231 if __name__ == "__main__":
232         main()
```