Jethro Modequillo, Matthew Dizon May 12, 2023

Exercise 2A: De-convolution using a Wiener Filter

Background

For this problem, we're provided a blurry & noisy image and are expected to deblur the image using the inverse of the blur used. Since this inversion amplifies noise though, we use a Weiner filter to dampen the noise added while still keeping the detail retrieved from the inversion.

Procedure

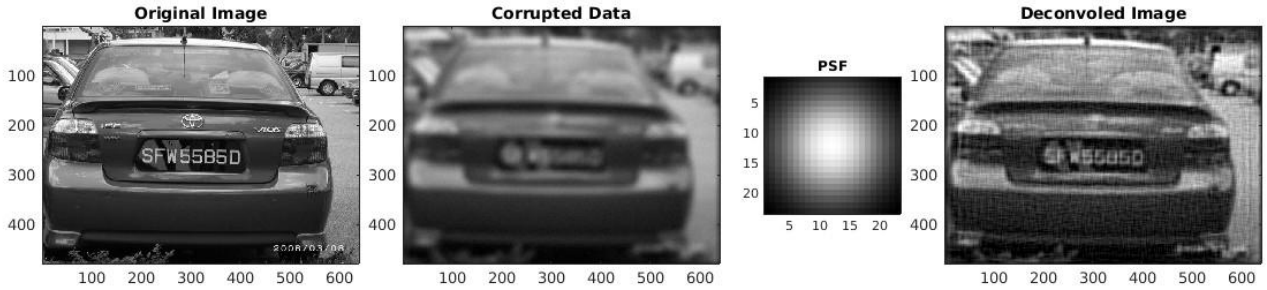
We can create a function that accepts a blurry image, the blurring function used, and a tolerance value for noise. Before we do anything else, we should pad the blur's point-spread function so that it's the same size as the image. This is necessary since we'll be doing cell-by-cell operations between the image and the PSF, so their sizes must be the same.

Afterwards, we can transform both the PSF and image into the fourier space to simplify calculations later. We grab the inverse of the blurry image with $\frac{1}{F_{blur}}$. Since the inversion will greatly amplify noise though, we can dampen the noise that gets through by multiplying the inverse image by a Wiener filter of the form $\frac{|F_{blur}|^2}{|F_{blur}|^2+k}$ where k is our tolerance for noise. We can retrieve more detail at the expense of allowing more noise if we move k closer to 0. Deblurring an image mostly boils down to finding the ideal value for k, assuming the blurring function is correct.

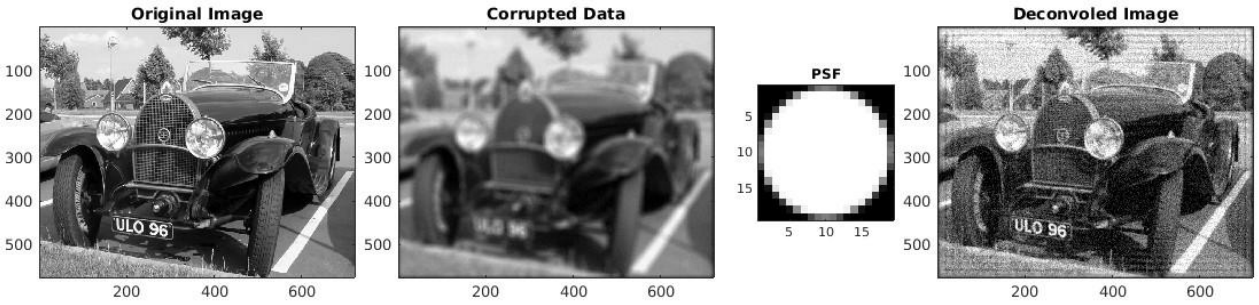
Once we have that, we can apply the filtered inverse of the blurring function to deblur the original image using $F_{image} * F_{inv}$. Since the result is still in the frequency domain though, we reverse our fourier transform using $real(ifftshift(ifft2(F_{result})))$.

Results & Discussion

We can see that our Wiener filter works as expected on blurred images. Inverting the blur image successfully removes most of the blur, while the Wiener filter suppresses most of the noise that comes in from the inversion. Below are sample images of how the filter performs with different blurs and noise amounts.

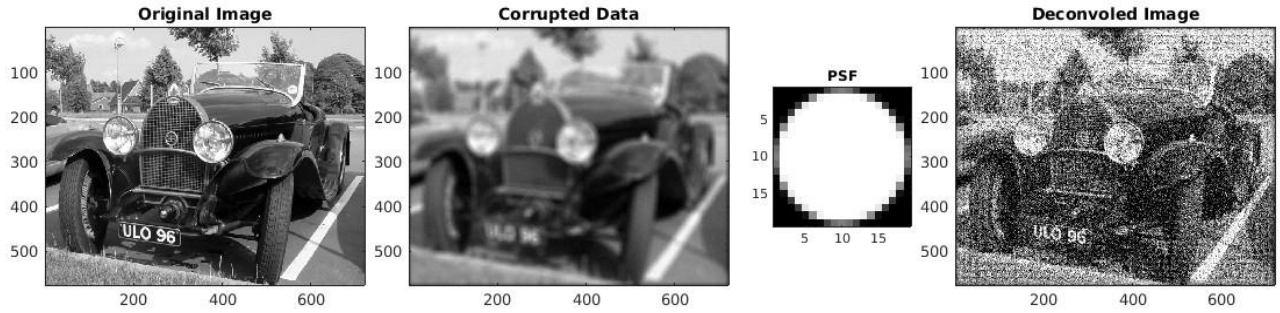


23x23 gaussian blur, k=0.001



Disk blur of radius 9, k=0.001

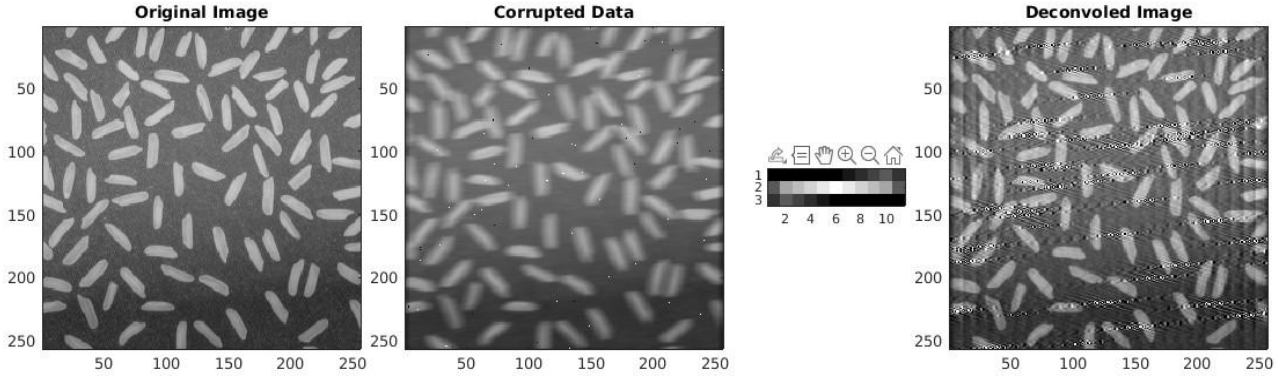
We can see that inverting the PSF is quite good at reconstructing the image. There's some unavoidable ringing at the borders of the image, which is a downside we can't escape because of our assumption that the image repeats at the edges. Despite not knowing the original image though, the Wiener filter uses the inverse image reconstructs details like the license plates of both vehicles (although the second car's license plate is clearer due to high contrast). Although this magnifies much of the noise, the Wiener filter is good at dampening a lot of it. This has to do with careful selection of our threshold *k* though, since bad thresholds will lead to bad images. See the example below for a threshold closer to 0 that allows more noise through. Low values look noisier and eventually look like static, while high values will block too much and eventually darken the image.



Disk blur of radius 9, k=0.0002

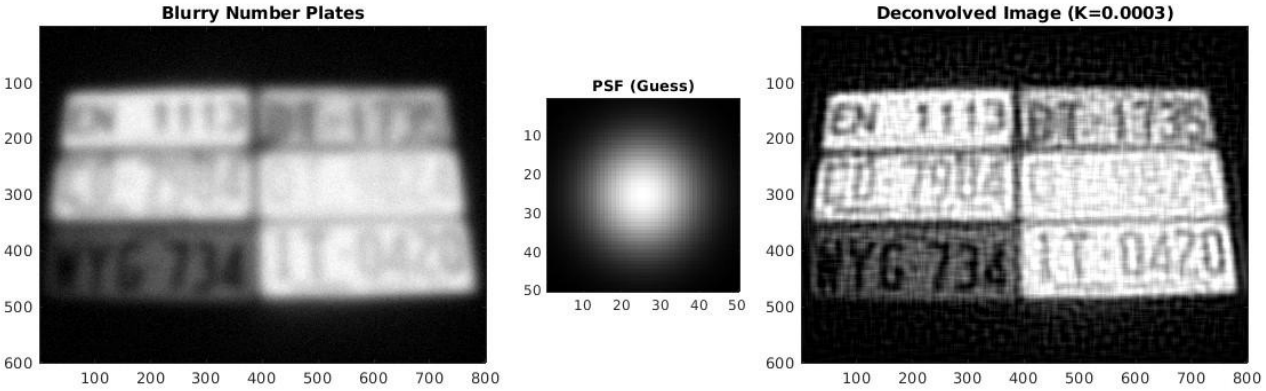
A significant weakness of our implementation is our assumption that noise is even throughout the image. That is, we use a single value *k* instead of getting the noise-to-signal ratio at different locations in the image. This is acceptable with white noise that's distributed evenly, but it's bad for localized artifacts such as salt & pepper noise. See the image below for an

example, where even miniscule amounts of salt & pepper become quite large after inversion. Our current Wiener filter doesn't suppress this noise very well, and a better implementation would suppress noise from the salt & pepper while keeping the signal from regions without the noise.



Rice with motion blur and minimal salt & pepper noise, k=0.001. Note how noise is enlarged

Lastly, we can also apply this deblurring strategy to images with no clear version available. This makes sense since we only need the blurry image, the supposed blur function used, and a tolerance for noise. Below we have one such example, with a guess for a blurring function and a tolerance value we arrived at through trial and error. Still, we can recover the text on the license plates: EN 1113, BT 1735, LD 7984, GT 9874, WYG 734, and IT 0420. Note that since different values can enhance/damage different license plates better, these guesses were made from multiple images and not just the one below.



PSF=('gaussian', 65, 10), K=0.0003

Conclusion

Deblurring can be done quite simply using an inverse of the blurring function used, but this comes with significant damage in the form of noise. With the use of a Wiener filter though, we have finer control over when and where we decide to permit noise in the pursuit of reclaiming more of the deblurred image's detail. Determining a good threshold value should be done case-by-case as each image's blurring, noise, and relevant features are unique.

One clear area for improvement is creating a smarter Wien er filter. This means being more aware of which regions are suspected of carrying more noise, and which ones have a clearer signal. With this additional information, the filter would be more informed for which regions it should pay particular attention to in suppressing noise/allowing signal.

Exercise 2B: Motion Image Compression

Background

Image compression is done to reduce the file size of an image. The motivation for this for file storage and transmission of images. While this process is able to reduce the file size, there is the possibility of removing details when performing lossy compression. However, it is possible to control how much details is lost. As videos are composed of multiple images, it is also possible to perform image compression of each frame of a video. The same benefits would be achieved, reduced file size of the video file and at the same time allowing for efficient video transmission.

Procedure

There are two functions we modified on the Excercise 2B: Motion Image Compression. These functions include djpeg_8x8 and simple_dmpeg. The first function is a jpeg decoder, this also decompresses a jpeg image. And the second function is simple_dmpeg, a video transmission decoder, this is done by processing the individual frame transmission using the djpeg_8x8 function.

The djpeg_8x8 function complements the provided jpeg_8x8 function which provides an encoded compressed jpeg image. The djpeg_8x8 function takes the arguments dc_coeff,ac_coeff, and Q; the first two are generated by the jpeg encoding function. The process of encoding and decoding a jpeg image is similar in a sense that the steps taken are done in a different order. Since the jpeg_8x8 function is provided, the processes in djpeg_8x8 are based on it. Making is as a guide to make djpeg_8x8 functional.

The procedure in djpeg_8x8 are the following, in order: (1) Copy the value of the DC coefficient to the first 8x8 sized tile of the image; (2) Retrieve the AC coefficient for the remaining 8x8 sized tiles of the image in a zig-zag manner; (3) Dequantization, estimate the original values of the image using Q table and Q scale; and lastly (4) Perform inverse discrete cosine transform using the idct function to the image, converting it into the spatial domain from a frequency domain.

The simple_dmpeg function then uses the djepg_8x8 to decode the transmission of a video file. The transmission is generated by a simple_mpeg function which returns an encoded and compressed frame of the video file, also using the jpeg_8x8 function. The process of simple_dmpeg function involves: (1) Iterating through all of the frames 8x8 sized tiles and retrieving the corresponding DC and AC coefficients of that tile; and (2) Decode the 8x8 sized

tile when there is changes observed from the previous image, this is determined if the DC and AC coefficients have zeroes. This is done considering the image transmitted only includes pixels that have changed considering the previous frame.

Results & Discussion

djpeg_8x8

The modified djpeg_8x8 function works when decoding jpeg data from the jpeg_8x8 function, showing an almost visually similar image. However, this would depend on the image and the quality factor (Q ranges from 1 to 99). The results show that there are some details removed from the image as shown as part of jpeg_test with the rms error value.

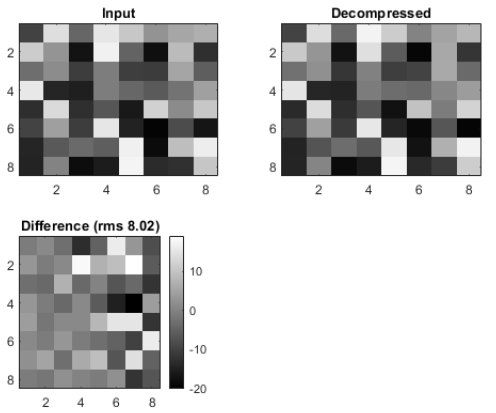


Image with soft edges, Q=80

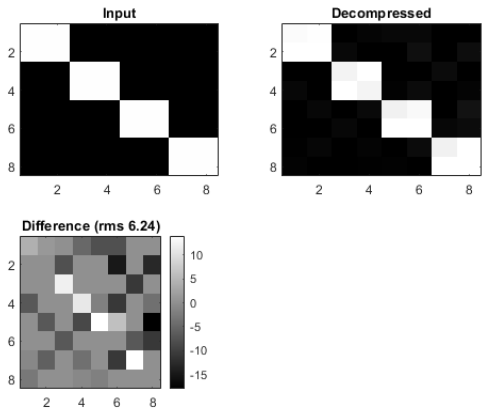


Image with sharp edges, Q=80

On the figures above, working with Q=80. There are missing details on both images as the rms error values are 8.02 and 6.24. The image with soft edges have a greater rms error, meaning that more details are lost. While the image with hard edges relatively lost fewer details. Despite the difference of the rms error, the effect of dequantization (part of image compression) can be observed on the image with sharp edges and not with the image with soft edges, these are called compression artifacts. As there are pixel that have a visible variation from the original image. As such, the Q=80 may be used for images with soft edges, where there may be a balance between compression and image quality.

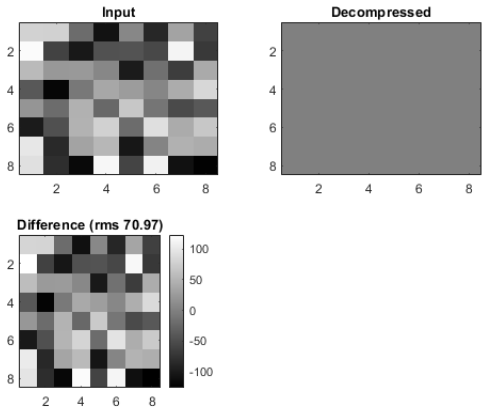


Image with soft edges, Q=1

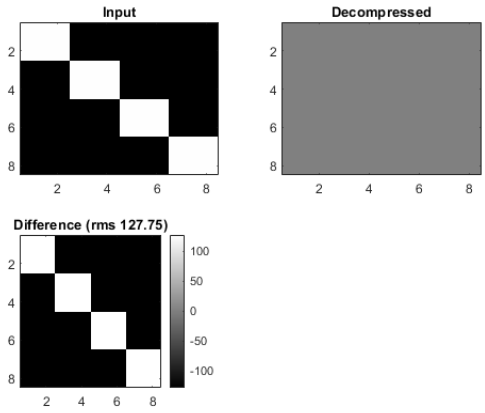


Image with sharp edges, Q=1

When the value of Q=1, there is a great rms error for both images. The great loss of detail has made the images unrecognizable. Compared to the other figures which uses a higher Q value, the rms error of the image with sharp edges is greater than the image with soft edges. It is around Q=10 (not shown in the figures) that the rms error of image with sharp edges is higher. Also, some parts of the image may not be recovered, such that part of the image shows a gray patch, similar to the figure above.

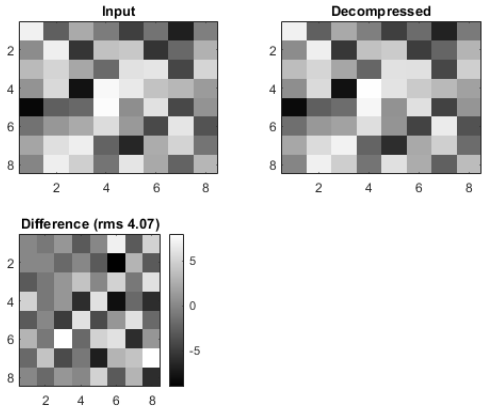


Image with soft edges, Q=90

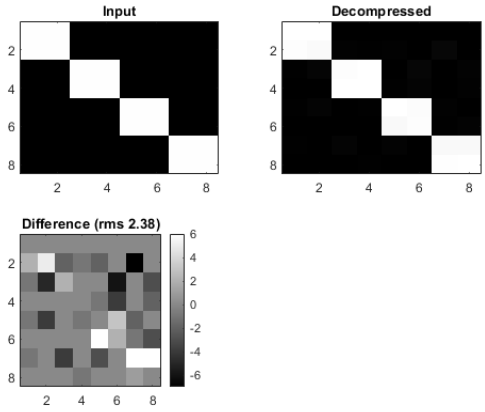
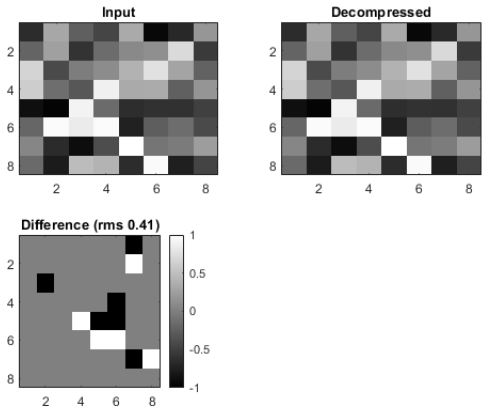


Image with sharp edges, Q=90

The balance between compression and image quality is needed in certain applications. As such the jpeg_8x8 and djpeg_8x8 functions when used with an image with sharp edges may use Q=90 when one wishes to have clear edges on that particular image. The drawback would be a large file size of the jpeg image.



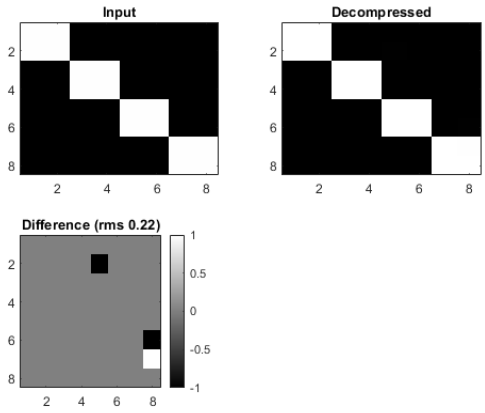


Image with soft edges, Q=99

Image with sharp edges, Q=99

When the value of Q=99, there is a low rms error for both images. As such, there is low detail loss on both images. Since the process involved dequantization, it is unavoidable to have some detail lost during encoding and decoding of the jpeg image. However, with Q=99, the effect of dequantization is only miniscule as the Q value is high.

simple_dmpeg

The modified simple_djmeg function works when decoding a transmission of jpeg data from the simple_mpeg function. All frames in the output (decoded by thesimple_dmpeg function) are similar to the original frames. The difference between the original frame and the reconstructed frame would be the presence of compression artifacts, as a result of the dequantization process in jpeg_8x8 and djpeg_8x8.

Input 1, all of the pixels of the frame were transmitted

For the input 1, all of the pixels were transmitted, considering that there would be no previous frame. All pixels of the first frame were processed using the jpeg_8x8 and djpeg_8x8 as part of the encoding and decoding process respectively. Compression artifacts can be observed in the frame. This allows the transmitted data to have a smaller file size compared to the original frame. As the whole frame file size was reduced 69.73%, for this frame and video. For the following frames, only the pixels which have significantly changed would be transmitted; essentially reducing the whole size of data transmitted. And at every 10 frames, the whole frame would be updated.

Input 2, updated pixels are transmitted and are updated on reconstructed image

For the input 2, only the pixels which have significantly changed are transmitted. And as such, the simple_dmpeg function reconstructed the frame using said data, updating the necessary pixels. With image compression of the frame at the same time excluding unupdated pixels within 10 frames, the transmitted data is reduced to only 3186 bytes for this specific frame.

The following figure shows the lowest and highest possible file size for the provided video file and quality factor. This would be considering the movement of the subject, background, and foreground of the frame. Less data is transmitted if there is little to no movement. Otherwise, more data would be transmitted.

Input 22, lowest file size for transmitted data of 733 bytes

Input 5, highest file size for non-full frame, transmitted data of 4328 bytes

For the final frame of the video, a total of a hundred frames. The overall transmitted and reconstructed video file size resulted in 552853 bytes from the 10368000 bytes of the original video. This resulted in an overall 94.67% reduction in the file size of the video.

Input 100, last frame of the video, resulting in a total of 552853 bytes

Conclusion

Image compression provides the benefits of better file storage and transmission of images. The same could be applied to videos, as these would be simply collection of images played in a sequence. For both image compression and video compression, both would suffer from compression artifacts as the image compression performs dequantization. These could be alleviated by using a higher quality factor, however, the higher this value the lesser the file size reduction is. There should be considerations on the Q value made when performing image compression. The same can be said for video compression. These would be how the image or video would be used, how would one store these images or videos, and lastly the cost of storing or transmitting the image or video.

Exercise 2C

1)

A) Since Jupiter is incredibly distant from us, we don't need to consider 3D perspective from projection—the image is far enough that we can treat it like a flat 2D image. Since our telescope can only move so little while keeping Jupiter in frame, this means images from the telescope can be defined as simple translations. If we're observing over a long time frame like multiple nights though, we may have to change it to isometry since the night sky will slowly rotate around us, rotating Jupiter as well. If we're taking pictures over the course of months or years though, we might even have to step up to similarity/linear-conformal since the orbits of Jupiter and Earth will get closer and farther throughout the year. If the telescope is powerful enough, this might show up as Jupiter changing uniformly in scale.

B) Since we're taking pictures of an object from multiple perspectives, the images can be defined as projective transformations. This makes sense because we're literally projecting a 3D object into a 2D image from different angles, where changing our perspective will distort the object's shape in a way that can't be sufficiently described as affine.

2)

$$T = \begin{bmatrix} 1.7321 & 1 & 10 \\ -1 & 1.7321 & -20 \\ 0 & 0 & 1 \end{bmatrix}$$

This is an affine projection that includes translation, rotation, and scale. It follows this form:

$$\begin{bmatrix} \cos\theta & \sin\theta & t_x \\ -\sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

It's immediately apparent that $t_x = 10$, $t_y = -20$. We can calculate s and θ as follows:

$$\begin{aligned} s\cos\theta &= 1.7321 \\ &= \sqrt{3} \\ s &= \frac{\sqrt{3}}{\cos\theta} \end{aligned}$$

$$\begin{aligned} s\sin\theta &= 1 \\ \sqrt{3}\frac{\sin\theta}{\cos\theta} &= 1 \\ \sqrt{3}\tan\theta &= 1 \\ \theta &= \tan^{-1}\left(\frac{1}{\sqrt{3}}\right) \\ \theta &= 30 \end{aligned}$$

$$\begin{aligned} s &= \frac{\sqrt{3}}{\cos\theta} \\ s &= \frac{\sqrt{3}}{\cos 30} \\ s &= 2 \end{aligned}$$

Translation factors $t_x = 10,\ t_y = -20$

Rotation factor $\theta = 30°$

Scale factor $s = 2$

The transformation matrix will shift the image 10 pixels right, shift 20 pixels down, rotate it 30 degrees, then uniformly increase the scale by 2.

3)

Given the ff:

$x = 20,\ p_x = 20.6,\ y = 10,\ p_y = 10.2$

$I(x, y) = 128,\ I(x + 1,\ y) = 64$

$I(x,\ y + 1) = 64,\ I(x + 1, y + 1) = 32$

We can calculate the bilinear interpolation as follows:

$a = p_x - x = 20.6 - 20 = 0.6$

$b = p_y - y = 10.2 - 10 = 0.2$

$$I(p) = (1 - a)(1 - b)I(x, y) + (1 - a)(b)I(x, y + 1) + a(1 - b)I(x + 1, y) + abI(x + 1, y + 1)$$
$$I(p) = (0.4)(0.8)(128) + (0.4)(0.2)(64) + (0.6)(0.8)(64) + (0.6)(0.2)(32)$$
$$I(p) = 40.96 + 5.12 + 30.72 + 3.84$$
$$I(p) = 80.64$$

With nearest neighbor interpolation, we just get the value at the sub-pixel's floor, which is (20,10) with a value of 128. With bilinear interpolation, we predict the sub-pixel (20.6, 10.2) has the value 80.64.

4)

Spectra 2 corresponds to output 1. The most obvious giveaway for this is the black horizontal line that cuts through the frequency domain of spectra 2. Horizontal lines in a frequency domain indicate the presence of vertical lines in a real image. Removing the horizontal line in the spectra removes vertical lines in the output. This can be seen in the image for output 1, where vertical lines like the building to the right or the man's legs in the bottom-left are completely missing. Spectra 2 also appears to be a mild high-pass filter, since the grey values around the frequency domain are slightly brighter. This explains why output 1 shows strong edges while losing out on the general form of objects.

Meanwhile, spectra 1 appears to be a mild low-pass filter, explaining the loss of detail in output 2. It may also include an inversion that gets the negative of an image's colors—this would explain the color inversion in output 2, but it's hard to see from the frequency domain.

It's also worth noting that both spectra 1 and 2 might include some blurring, like a gaussian filter. Notice how a large white circle dominates the middle of both frequency domains.

In the real image, this comes out as blurry “smudging”, similar to the PSF’s of the gaussian filters in Exercise 2A. This explains why both outputs show a clear loss of detail.