Take home exam FYS-2010

Candidate number: 11

UiT - Norges Arktiske Universitet

March 8, 2023

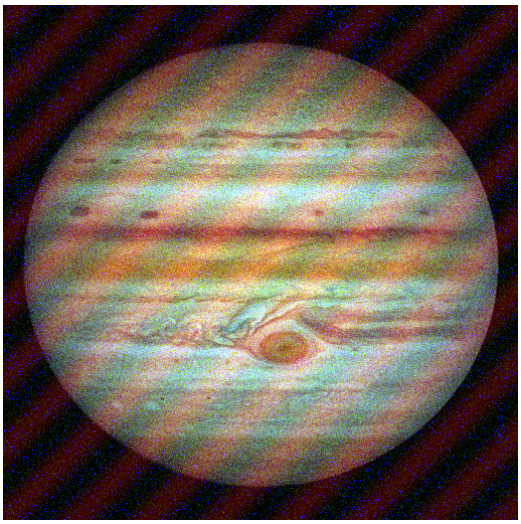
Note that all the code for tasks 1 and 4 is added to the end of the document, as well as being in separate files to this document.

Task 1

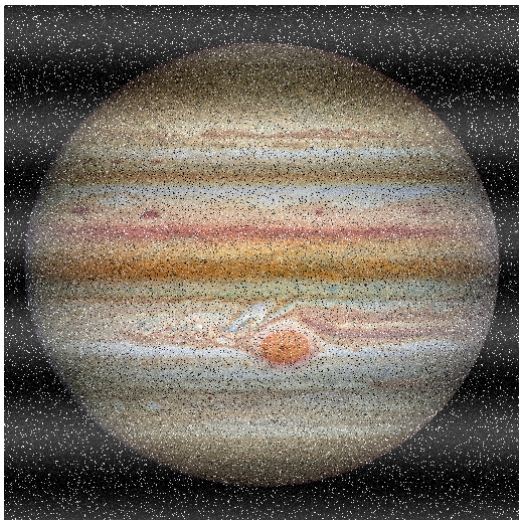
a) Explain the general process of image restoration. How can an image become degraded, and what are we trying to achieve through image restoration?

Image restoration is the process of restoring or 'fixing' an image that has been subject to degradation. There are lots of different ways an image could be degraded, examples are: salt and pepper noise, which are some randomly distributed pixels on the image which have intensity 255 (salt) or 0 (pepper). We can also have periodic noise, which appears as periodic bands across the image. There are also many many other types of noise, these are just two examples. So degradation is basically just an error that has affected the image and distorted it. When we restore an image we try to remove this degradation. Usually we like to draw a distinction between image restoration and image enhancement. Image restoration, as mentioned, is the process of removing the degradation of an image. Image enhancement on the other hand is just subjectively altering an images appearance, we use image enhancement for many reasons, sometimes its just to make the image look nice, but sometimes we might only be interested in certain forms of information 'hidden' in the image. Then we might enhance it so we can make these qualities of the image more prominent, examples here might be that we are interested in high frequency components, maybe just the vertical high frequency components, or the contrasts of the image. What these examples of image enhancement have in common is that they are subjective, I might think very high contrast and high saturation in an image looks nice, and someone else might think it looks bad. And same with making certain features more prominent, its subjective what features one is after. Image restoration on the other hand is objective, the image has been degraded in a certain manner, and we can use mathematics and statistics to model this degradation, and then try to remove it. A purely objective process. The usual source of degradation in an image is interference during the transmission and/or the acquisition of the image. And there might be different types of sources for degradation, and they correspond to different types of noise. Thermal effects on the sensor in a camera could cause some noise, electric interference during acquisition could cause periodic noise.

b) Take a look at the images, or analyse them in any way you see fit. Describe the types of noise you see, and if possible provide indicators for the noise being what you see. Have the images been degraded in the same way? If not, how are they different? Be precise, use figures and plots to make your point.



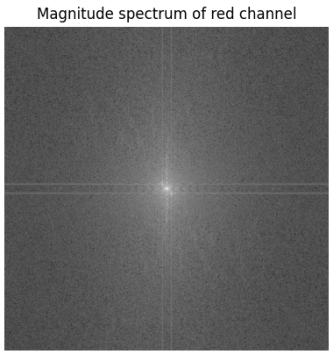
(a) Jupiter1.png



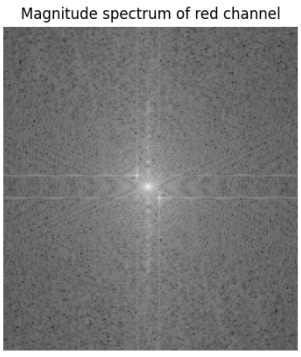
(b) Jupiter2.png

Jupiter 1

If we look at jupiter1.png the first thing we see are the red bands diagonally across the image, we can easily see that this is sinusoidal interference in the red channel. But we should also prove that this is true, the easiest way to do this is to plot the spectrum of the red channel of the image.



(a) Spectrum of the red channel of jupiter1.png



(b) Zoomed in view of 2a

Here we see exactly what we would expect, the two small impulses in the spectrum corresponds to the sinusoidal interference in the image. So we have successfully identified the periodic noise in the red channel. But we also see some noise in the green and blue channels. At first sight both look like salt and pepper noise, but we need to investigate closer. We can do this by plotting the color histogram of the image.

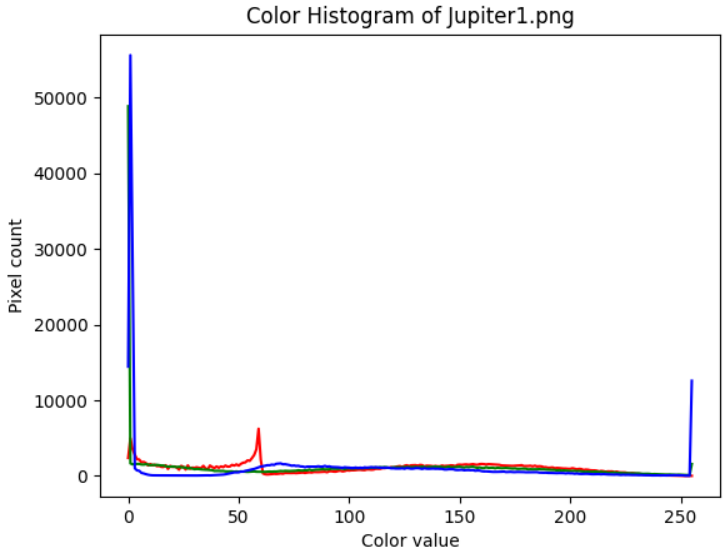
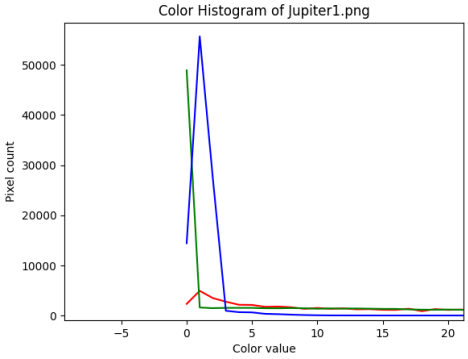
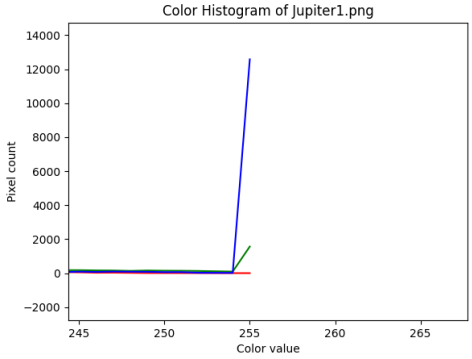


Figure 3: Color histogram of Jupiter1.png



(a) Jupiter1.png color histogram start



(b) Jupiter1.png color histogram end

Here we can see more clearly what kind of noise we have, if we take a look at the beginning and the end we can see very clearly what noise we are dealing with. These 'spikes' are characteristic of salt and pepper noise in the blue channel and pepper noise in the green channel. Here we can also

see a small 'peak' in the red channel closer to the middle of the intensity range, this corresponds to the periodic noise mentioned earlier.

Jupiter 2

In jupiter2.png it is actually quite easy to identify the type of noise we have. We can see periodic noise going horizontally across the image, like we had in jupiter1, only here they are white, meaning all color channels are affected equally. We can also see some salt and pepper noise over the image, this too affecting all channels equally. Of course we still want some proof, or indicators that this is correct, which we can get from the spectrum, and the histogram of the image.

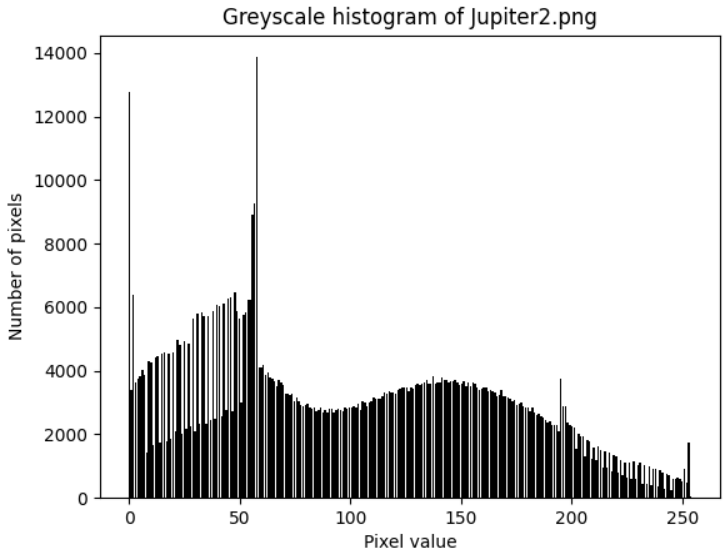


Figure 5: Histogram of Jupiter2.png

Here we have the histogram of jupiter2, what we see is the expected, a peak at 0 and a peak at 255, indicating that there is salt and pepper noise. There are also some other spikes, these correspond to the periodic noise.

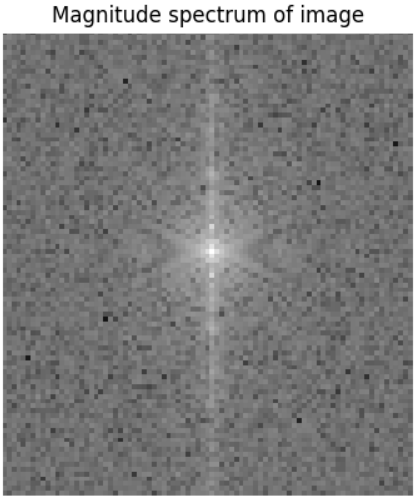


Figure 6: Spectrum of jupiter2

Figure 6 shows the frequency spectrum of jupiter2 (zoomed in to the area of interest) and what we see here is similar to what we had in jupiter1. We can see two impulses above and below the center, they are hard to see, but they are visible here, and indicate to us that we are dealing with sinusoidal interference.

c) Start off with restoring "Jupiter1.png", and try to restore this image. Choose filters you know off, or design your own filter to restore the image. Make sure to explain the filters you use, and your reasoning for choosing them. The explanation of your filters should be grounded in theory.

Firstly I would like to get rid off the periodic noise in the red channel, to do this I have chosen to implement a notch reject filter. A notch reject filter is essentially just a band stop filter, which filters out the frequencies we don't want, which in our case are the frequencies that give the sinusoidal noise. The reason I chose the notch reject filter is that this, by far, seemed like the quickest and easiest way to remove the periodic noise. This method is very intuitive also, and pretty straight forward.
I implemented this filter in quite a simple way. First I created a filter of the same size as the image and set all of the values in the filter equal to 1, then I found the coordinates of the 'dots' in the frequency spectrum, and set those values equal to 0. Then I multiplied this filter with the red color channel. What this does is set the values that correspond to the periodic noise to 0, and does nothing to the rest of the image. To find the exact coordinates of the 'notch' in the notch reject filter I just plotted the filter over the frequency spectrum until the 'notches' and the 'peaks' matched up. All of this was done in the frequency domain, this is to make the implementation of the filter much easier (since multiplication in frequency domain is convolution in spatial domain,

from the convolution theorem). Then I returned the image back to spatial domain through an inverse fourier transform.
The result of the notch reject filtering can be seen in figure 8.
The notch reject filter, multiplied by the spectrum of the image (red channel of the image) is this:

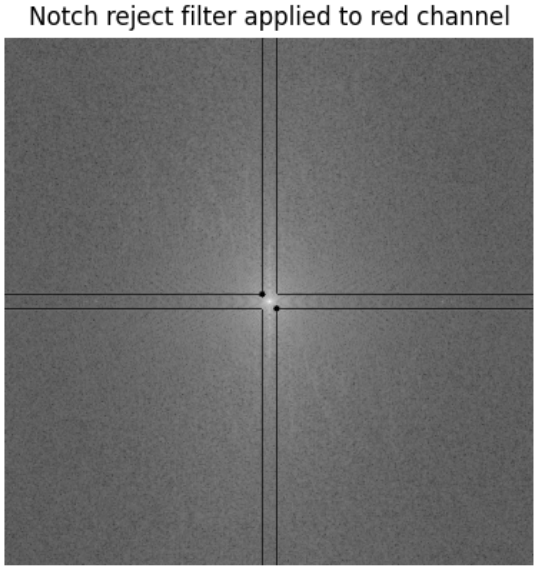


Figure 7: Notch reject filter used in the red channel of jupiter1

Note that I decided to try to filter out the 'rays' coming out of the impulses, the effect this had was minimal, but it did improve the filtering a little bit, so I decided to keep it.

Then I decided to remove the salt and pepper noise in the blue channel. Do to this I decided that a contraharmonic mean filter would be best. This is because this type of filter is particularly good at filtering out pepper noise when $Q > 0$ and good for salt when $Q < 0$. My thought was then to implement this filter twice, once with $Q < 0$ and once with $Q > 0$. However this didn't turn out too good, maybe I wasn't able to find the right parameters, or maybe it was something else, but the result was not an improvement. Therefore I decided against the contraharmonic mean filter and chose a median filter instead. A median filter is also very good at removing salt and pepper noise. This is because a median filter takes, as its name suggests, the median intensity in the kernel, logically this means that salt and pepper noise will be removed. The image will also be a little blurred, but its a trade off that is well worth it to get rid of the salt and pepper. The way I chose to implement the median filter is with a function called medianBlur() in opencv. Although it could also be implemented by creating the kernel and convolving it with the image. The kernel size I went with was 3x3 and I chose to perform the median filtering twice. The result of this can be seen in figure 8.

Lastly we need to get rid of the green pepper noise. To do this I decided to try again with the contraharmonic mean filter. I used trial and error to find the optimal value for Q. What I ended up with was Q=0 giving me the best result (with a 3x3 kernel). And one thing to note is that when Q=0 the contraharmonic mean filter reduces to an arithmetic mean filter. The way I implemented the contraharmonic mean filter was by writing a function that utilizes equation (5-26) from Gonzalez. Also note that I chose border replication for the boundary conditions in this case, as that is advantageous to zero-padding for this case. Again the result from this, is found in figure 8. This was the final step in the restoration of jupiter1, and as we can see in figure 8 the result is quite good. It's not perfect of course, and it never will be, a perfect restoration is impossible. The result we have here however is very good, most of the noise is gone, there is still some noise, especially the sinusoidal noise, however this is mainly in the edges, away from the subject of the image, so the image still looks good in that sense. We have also lost a little contrast, but that is not a problem as we can easily fix this in the enhancement process.

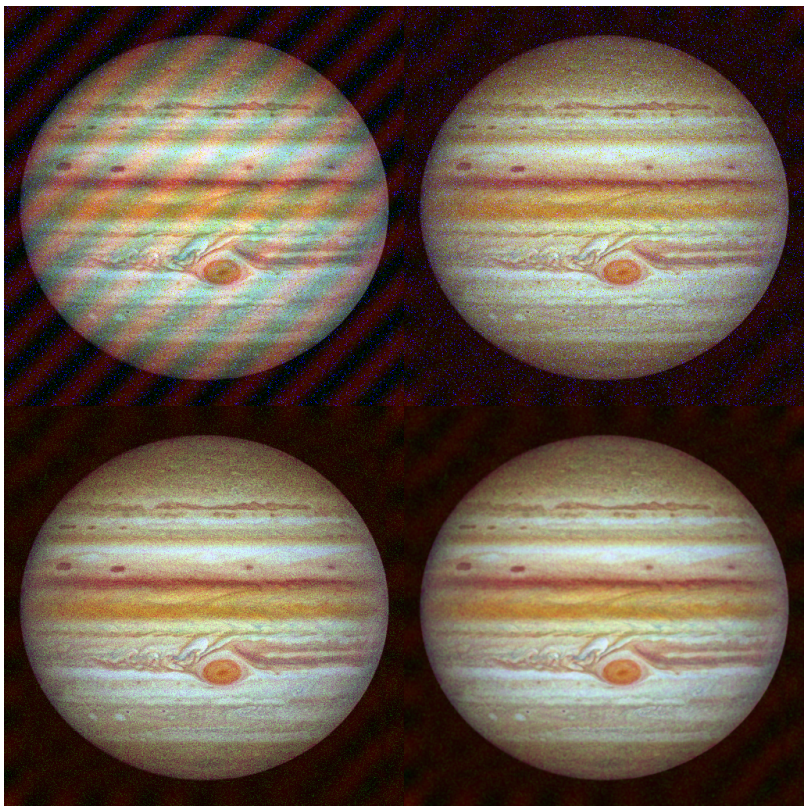


Figure 8: Progression of jupiter1 (top left is original image, top right shows image after notch reject filter in red channel, bottom left after median filter in blue channel, and bottom right shows the final image after the contraharmonic mean filtering in the green channel)

d) Now take a look at "Jupiter2.png". Will the restoration process be different in any way from the restoration process of "Jupiter1.png"? Choose filters or design filters and restore "Jupiter2.png". Explain your choices as you did in task c).

The restoration process of jupiter2 will be very similar to jupiter1. This is because we are dealing with the same type of noise, we have both the sinusoidal interference, and the salt and pepper noise. The only real distinction we have is that the degradation has affected all color channels equally. What this means for us is that there is no need to separate the image into its separate colors and perform different restoration processes to each. We will separate the channels though, since the best way to filter out the noise is to do it in each channel and then re-merge the image. This image also seems more degraded than the other, it doesn't really affect the restoration process, but rather that we cant really expect the final result to be as good as it was for jupiter1.

I began this restoration by removing the periodic noise, again I did this by creating a notch reject filter to filter out the 'impulses' in frequency domain. I did this by taking the fft of my image, then shifting the frequencies with fftshift. Then I plotted the magnitude spectrum and took note of where the impulses were, then the filter itself was constructed. The filter was constructed by initializing a matrix where each value is 1.0, then taking the frequencies that are to be filtered out and set their corresponding value to 0. I created the filter in this way, but instead of using pixel coordinates from the bottom left or anything like that, I created my function to filter out frequencies within a circle Q and positioned this circle as a function of distance from the center, this felt less arbitrary, and more generalized. The function for euclidean distance from the center was taken from Gonzalez (eq. 5.34 & 5.35). (Note that I also did this exact same method for jupiter1). The filter ended up looking like this:

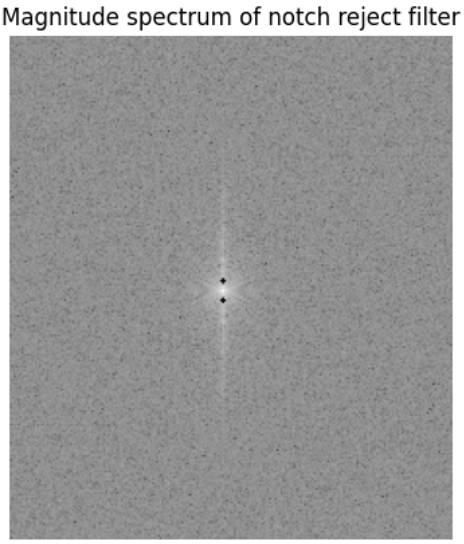


Figure 9: Notch filter over the frequency spectrum of jupiter2 (zoomed in)

And the result of the notch reject filter on the image can be seen in figure 10. Here we can see that the result is pretty good, most of the periodic noise is gone, some is still there, which is expected, but it's a massive improvement.
Now we have to remove the salt and pepper noise in the image. Here I decided to implement the same technique that I used for jupiter1. Which is a median filter, I chose this for the same reasons as for jupiter1, and I tried a few different kernel sizes, and a different number of iterations, and landed on filtering the image twice with a 3x3 kernel. The result can be seen in figure 10.

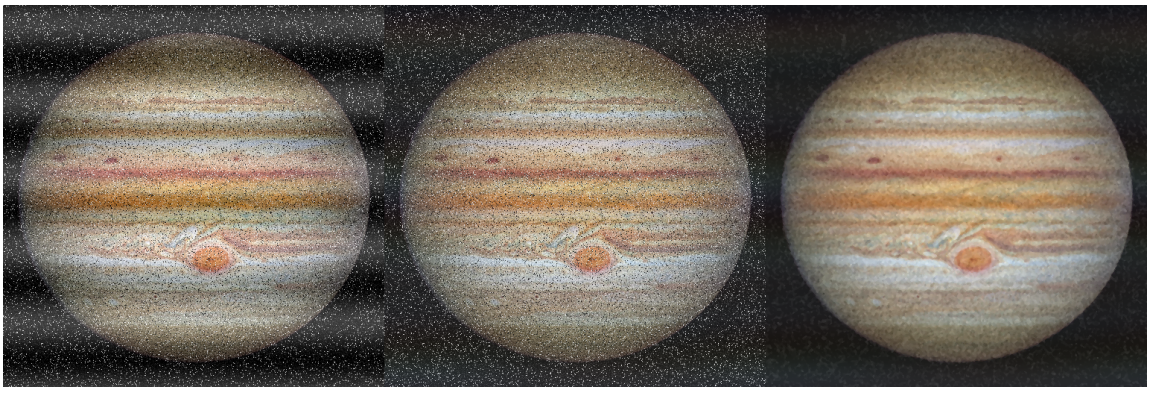


Figure 10: Progression of the restoration process of jupiter2.png (left: original image. middle: after notch reject filtering. right: after median filtering

I am quite pleased with the result from the restoration process, again this image is not perfect, and there is still some degradation, but the result is a massive improvement from the original image. The original image was very heavily degraded, so to get the image to this point is very good. The main 'problem' with the image now is that the contrast is a bit poor, the image looks kind of grey now, but we can fix this with some enhancement.

e)Finally we want to see if we can enhance the images. Use what you've learned in this course about image transformations, filtering, and edge enhancing to improve the overall quality of the images. Your solution should contain one spatial, and one Fourier enhancement method (in total, not necessarily for both images). Explain your enhancement process, discuss the chosen methods strength and potential weaknesses, and how enhancing a color image differs from enhancing a grayscale image (if at all). Finally, present your enhanced images and discuss the results.

Jupiter1

I started off with jupiter1, after looking at the image a bit I decided that I would like to increase the contrast in the image, I would like to see more detail on the surface, as well as a sharper contrast between the planet and the background. So to do this I first decided to do some contrast stretching on the image. Contrast stretching is essentially the process of redefining the range of the histogram. So lets say we have an image whose intensity values only range from 100-200, this image is going to be quite low contrast, so what we can do is stretch this range to 0-255, and then we project the pixel intensities to the new range, so for example a pixel with intensity 100 in the original image, will become 0 in the processed image. Now I implemented this technique on jupiter1, even though it actually is quite well defined over the entire range 0-255. But this is still a good technique to get a little more contrast in the image. The equation I used for this was the following.

$$Image = \frac{(image_{old} - min_{old})}{(max_{old} - min_{old})} * (max_{new} - min_{new}) + min_{new} \tag{1}$$

The notation here is a bit messy, but hopefully understandable, what the expression does is just redefine the image across the new range. There is a downside to this however, to implement this function the values outside the old range $[min_{old} - max_{old}]$ will get clipped. Meaning that we lose the information in the very high and very low intensities. This is of course very unfortunate, and could be avoided, but its more complicated. And here the information that gets lost is not to important, so it is a trade-off that is worth it in this case, if we chose the right values to clip of course. I chose $min_{old} = 20$ and $max_{old} = 240$ and stretched these values to 0 and 255 respectively. The result of this process, as well as the other ones in the enhancement process can be found in figure 11

I also want to increase the amount of detail in the surface of the planet, to do this I chose to do gamma correction. This process is supposed to be good at bringing out fine details in the image. To do this I used equation 3.5 in Gonzalez ($s = cr^{\gamma}$), with c=1, and where r is the intensity values of the old image. I used trial and error to find the best value for γ and I ended up using $\gamma = 1.2$

Lastly I wanted to sharpen the image a little more, and to do this I decided to use laplacian sharpening. Laplacian sharpening is a technique that uses the laplacian to sharpen the image, and bring out finer details that are not very visible. The laplacian can be implemented both in spatial and frequency domain. However as far as I could tell they are more or less equivalent, and if there is a difference frequency domain should be a bit better. Therefore I went with the frequency domain for this filtering, also, the two previous methods I used were in spatial domain, so it would be good to do this filtering in frequency domain. To implement the laplacian filtering I more or less followed the process in page 290 in the textbook (Gonzalez & Woods). There is not much more to explain about the implementation, other than the fact that I split the image into its color channels and performed the filtering in each channel separately. The resultant image was actually not very good, there was a lot of noise and a lot of colored noise that shouldn't be there. I think the reason for this is that the noise in the image is not completely gone, and its different in each color channel, so the laplacian brings out different noise in the different channels. Which makes sense since the laplacian more or less acts as a high pass filter so it brings out high frequency noise. The solution for this was luckily very simple, in the implementation I multiply the laplacian by a constant c, which by default is -1 (as explained in Gonzales), but changing this constant will increase or decrease the effect of the laplacian filtering. So what I did was create a separate constant for each color channel and varied them until I ended up with an image that looked good. This is not the most elegant solution, but it works, it doesn't really matter that we change the variable in each color channel like this, as long as it looks good. This reasoning is especially valid in image enhancement cause of the subjectivity of the process (as explained in task 1a).

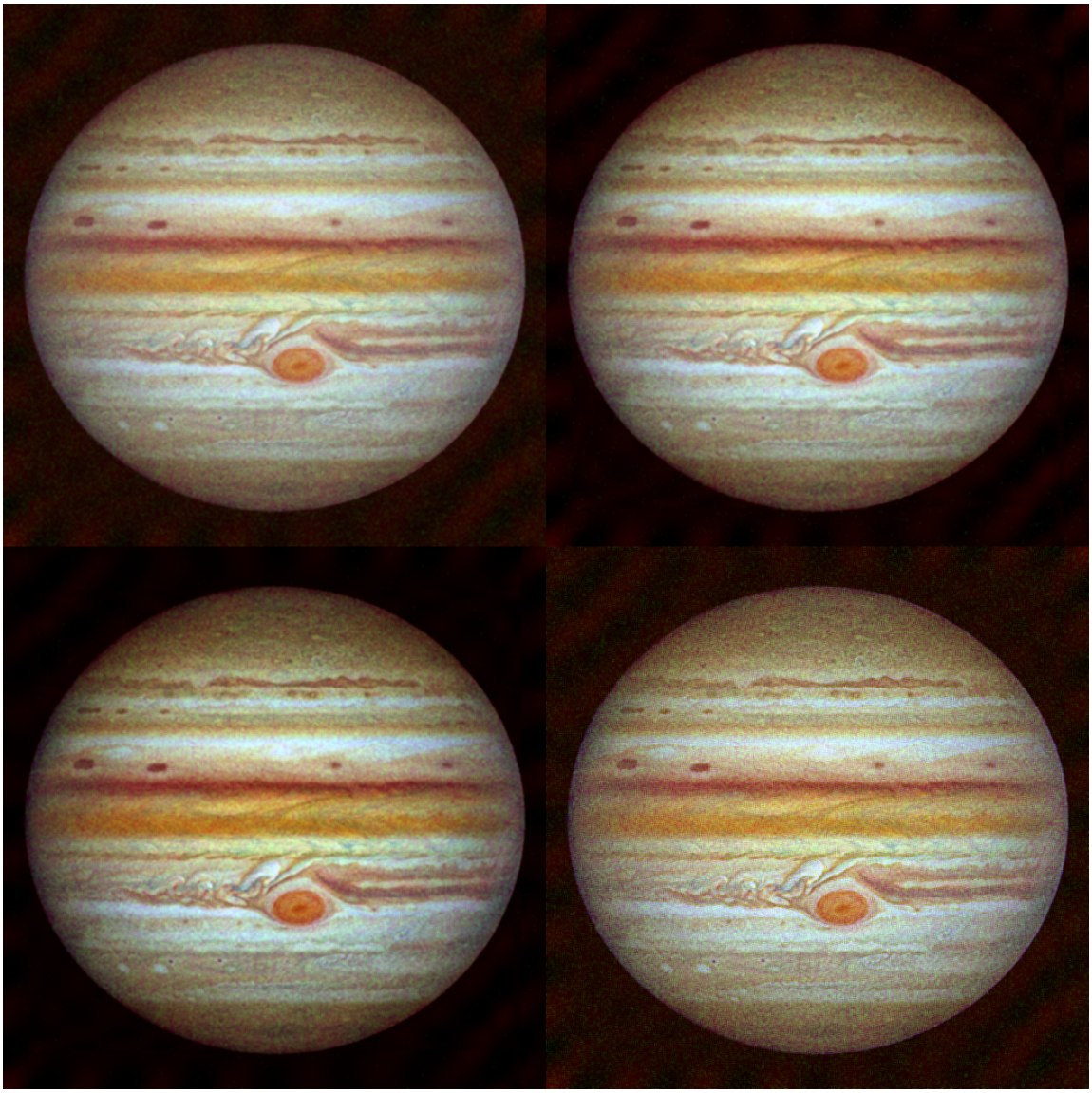


Figure 11: Progression of the enhancement of jupiter1 (top left: restored image (no enhancement). top right: contrast stretched. bottom left: gamma corrected. bottom right: laplacian filtering)

As we can see here all the images in figure 11 look a little different and which one looks best is very subjective. Personally I think the final image that has gone through all processes mentioned looks best. The laplacian filtering brought back some of the periodic noise that is not present in the other two, but the planet surface looks a bit better. As mentioned however this is completely subjective, if this is just to make the image look nice then its good, but in a different scenario like if we were conducting research on jupiter and were interested in fine details on the surface, then we

could use the same processes, but emphasized those that would increase the contrast and bring out fine details. So to sum up I think the enhancement went well, and I got a good result with more detail on the surface and a sharper transition between the planet and the black background. And here the exact outcome is not to important because of the subjectivity of image enhancement, but the important part is that the enhancement processes worked as intended.

Jupiter2

Now lets get onto enhancing jupiter2, the first thing I want to improve is the contrast in the image. The image has become a little blurry and grey after the restoration process, so I want to increase the contrast and bring out some detail. If we look at the histogram of jupiter2 we can see that it definitely is needed.

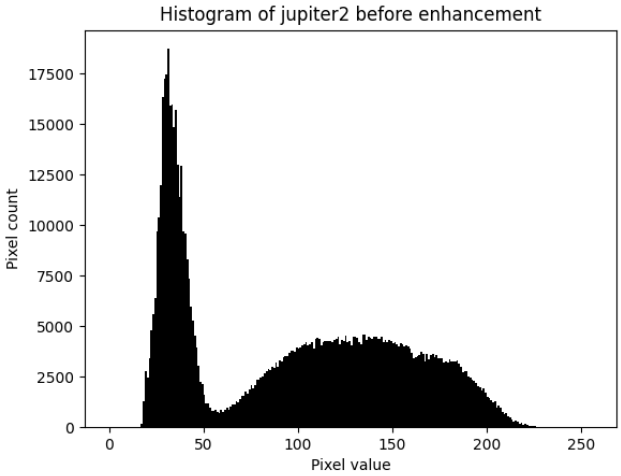


Figure 12: Histogram of jupiter2.png after restoration

In figure 12 we can see that the photo doesn't really have any pixel values lower than ≈ 25 and no values higher than ≈ 225. Which means that there is not much contrast in the image. To fix this I used contrast stretching, and I used the exact same implementation her as I did for jupiter1, only I used [25,225] as the old range and stretched it to [0,255]. The resultant image can be seen in figure 13.

I also want to enhance some detail in the subject, and to do this I went with gamma correction, as I did with jupiter1. Through trial and error I decided that $\gamma = 1.05$ was the best value, and the result of this can be seen in figure 13.

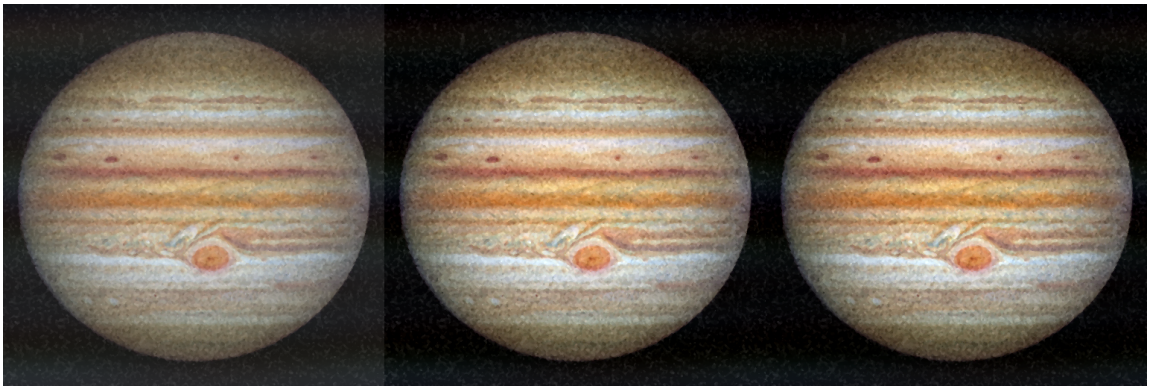


Figure 13: Progression of the enhancement process of jupiter2. (left: restored photo. Center: contrast stretched image. Right: gamma corrected image.)

Here we can see that the image has been improved a lot, the contrast especially has improved a lot. Which we can confirm by looking at the histogram after the enhancement, and compare it with figure 12.

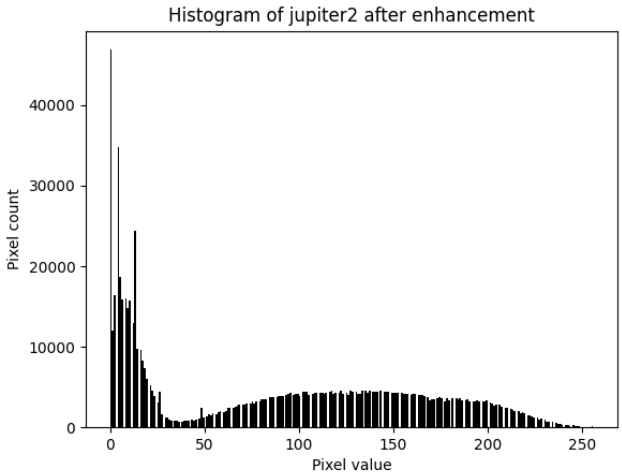


Figure 14: Histogram of jupiter2 after enhancement

Here we can see that the histogram is more 'spread out' in a way, there are pixels of all intensities, meaning we have more contrast. We can also see more detail in the planet and we have a darker background.

Now onto answering the rest of the task: enhancing a color image is quite similar to enhancing a greyscale image. We can just separate the image into its color channels and perform the enhancement on each color, then re-merge them. However it's not actually that simple, the filters and

processes we use might affect the colors differently if there are large differences between the color channels. And if we want to perform different enhancement techniques on different colors which one often might, for example if we want the image to be warmer, or if we want to see more red details. Then we have to be more aware and use intuition to check that it actually looks good, but the processes themselves are the same for color images as for greyscale images.

Task 2

a) Write an equation that can be used to perform simple image compositing and justify how the equation works. Then implement it in Python by editing the function basicComposite() in the marked area. Once you have implemented this compositing equation, include the resulting composites in your report and describe them.

For this task I just went with the most simple composition technique I could think of. We are given a source image from which we want to take the subject from, a target image which we want to paste the subject onto, and lastly a mask which defines the area of the source image that we want to paste onto the target. The mask is just a boolean array of the same size as the other images, where its value is 1 if it is a part of the source image we want to paste, and 0 everywhere else. So if we multiply the source image with the mask we have an image that is 0 (black) everywhere, except for the area we want to paste onto the target. If we add this to the target image now we will get an ugly image, that is because we get invalid values in the photo (for example if we have a white pixel in the source and add it onto a white pixel in the target we get a pixel value of 510). What we have to do first is multiply the target with the inverse of the mask, in that way we get the target image with the masked area removed. Now we can add them together, this explanation is a bit messy, but the code (with the equation) is added here:

```
comp = np.copy(target)
mask = mask[:,:,np.newaxis] # Add a third dimension to the mask so that it can be
    broadcasted with the source and target images
comp = source*mask + target*(1-mask) # creating the basic composite image
```

(a) Hard composition image

(b) Hard composition image

Figures 15a and 15b show the results of this composition. As expected the results are quite poor,

the edge of the composition is very hard because of the difference in color between the source image and the target image. Especially in 15b since the color difference is more drastic, and we clip part of the mountain. Although it looks poor because of the sharp transition and the color difference, we know that we have correctly implemented image composition, which is good news.

b) The composite produced in the part a) can be improved by smoothing the transition between the source and target images. Explain how this can be done using the same equation you wrote for part a). Implement your idea in Python by editing the function smoothComposite() in the marked area. Once you have implemented this, include the smooth edge composites in your report and describe how they differ from the ones you produced in part a).

For this task I once again implemented the most straight forward solution, which is to blur the mask before the composition process. To do this I used the GaussianBlur() function in opencv, which is pretty simple, but opencv is strict about the datatype, so I had to convert the mask to a uint8 array, and also scale it up to 255, apply the gaussian blur, then get it back to the range [0,1]. Then the process is just the same as in task a). Note here that the parameters used for the gaussian blur are completely arbitrary, and were picked because it looked good, they were not carefully designed in any way.
The code for this task is the following:

```
comp = np.copy(target)

#blurring the mask
mask = np.array(mask, dtype = np.uint8)
mask = cv2.GaussianBlur(mask*255, (101,101), 20)/255 # blurring the mask with a
    gaussian blur

mask = mask[:,:,np.newaxis] # Add a third dimension to the mask so that it can be
    broadcasted with the source and target images
comp = source*mask + target*(1-mask) # creating the smooth composite image
```

(a) Smooth composition image

(b) Smooth composition image

Figures 16a and 16b show the results of the smooth composition with gaussian blur. As we can see the result here is also pretty much as expected. The images look the same as in the previous task,

except for that the edge is smoother. The fact that the edges now are smoother makes the images look a lot better, but they still look pretty bad. The sourced image looks very out of place, the color is completely different, and also there is a bit of a gradient on the target image that is not on the source image. This also makes it obvious that this is a composite image, but it is definitely an improvement over the hard edged composition.

Task c)

Expanding the given equation with the discrete 3*3 laplacian for a pixel whose neighbours all are inside the mask can be done by taking the convolution of the image and the mask over an arbitrary pixel. We can also follow the logical derivation presented on page 179 in Gonzalez, both methods are more or less equivalent. If we take an arbitrary pixel and call it I(x,y) we can imagine overlaying the laplacian matrix over it and the laplacian of this pixel will be the sum of the products of the laplacian coefficients and the pixel values. This gives:

$$\nabla^2 I(x,y) = \nabla^2 S(x,y)$$
$$-4I(x,y) + I(x+1,y) + I(x-1,y) + I(x,y+1) + I(x,y-1) = \nabla^2 S(x,y)$$

If we now consider the case where a neighbouring pixel is outside the mask, for this case lets say that I(x,y-1) is outside the mask, we get:

$$\nabla^2 I(x,y) = \nabla^2 S(x,y)$$
$$-4I(x,y) + I(x+1,y) + I(x-1,y) + I(x,y+1) + T(x,y-1) = \nabla^2 S(x,y)$$
$$-4I(x,y) + I(x+1,y) + I(x-1,y) + I(x,y+1) = \nabla^2 S(x,y) - T(x,y-1)$$

In this case we use the fact that I(x,y) = T(x,y) if x,y correspond to a pixel outside the mask.

d) Based on your expanded equation in c), how would you solve it as a system of linear equations in the Ax = b form? Describe the contents of A, x, and b. What is the largest number of non-zero elements in any row of A? What are the unique, non-zero values in A?

If we look at the equations from c) it looks impossible to solve at first, since we have 4 unknowns and only 1 equation (3 unknowns in the case where a neighbour is outside the mask). What we need to consider however is that we can write up such an equation for every single pixel in the mask, meaning that, if we have an n pixels in the mask, we get n unknowns and n equations. Since we have so many equations we know that the best way to solve it is using linear algebra. If we look at the equations from task c we can rewrite this using matrices so we get an equation for each pixel. We need an equation for each pixel in the mask, meaning that our matrix A needs to be n*n, the right hand side of the equation for each pixel is known, and therefore only needs one column, meaning that if we set up the system of linear equations in ax=b form b will be a n*1 matrix filled with the right hand side expression in the equations above. We can also separate the left hand side expression into the coefficients, and the pixel values, i.e. we separate (-4, 1, 1, 1, 1) and (I(x,y), I(x+1,y), I(x-1,y), I(x,y+1), I(x,y-1)) into separate matrices, where the matrix of pixel values will be x and the coefficient matrix will be A. X contains all the pixel values of the new composite image, so logically it will be a column matrix with as many rows as there are pixels in the image.

But we know the pixel value for all pixels outside the mask, so we can ignore these and think of x as a n*1 matrix. And A will therefore need to be a n*n matrix

- b is a n*1 matrix whose values are $\nabla^2 S(x,y)$. If the pixel corresponding to a given row has a neighbour, for example: I(x,y-1) outside the mask -T(x,y-1) is added to that row of b.
- x is a n*1 matrix whose values correspond to the pixel values I(x,y), which is what we are solving for.
- A is a n*n matrix filled with the coefficients from the laplacian matrix. Each row correspond to the laplacian of each pixel. The number of non-zero values in a row is equal to the number of neighbours each pixel has that are inside the mask plus itself, for example if a pixel has all its neighbours inside the mask, the corresponding row in A will have 5 non zero values. If it has one neighbour outside the mask it will have 4 non-zero values etc. The only non-zero values in this matrix are -4 and 1.

To visualize what we are doing we can write the first part of the first equation from 2c like this:

$$\begin{bmatrix}-4 & 1 & 1 & 1 & 1\end{bmatrix} \times \begin{bmatrix} I(x,y) \\ I(x+1,y) \\ I(x-1,y) \\ I(x,y+1) \\ I(x,y-1) \end{bmatrix} = [-4I(x,y) + I(x+1,y) + I(x-1,y) + I(x,y+1) + I(x,y-1)]$$

Meaning that the Ax=b form of the equation from 2c becomes:

$$\begin{bmatrix}-4 & 1 & 1 & 1 & 1\end{bmatrix} \times \begin{bmatrix} I(x,y) \\ I(x+1,y) \\ I(x-1,y) \\ I(x,y+1) \\ I(x,y-1) \end{bmatrix} = [\nabla^2 S(x,y)]$$

Then we just need to expand this so we have all n equations (one for each of the n pixels in the mask) in the matrices, and we have our final equation on the form Ax=b, which we can now solve.

e) Edit the function populateRow() in the marked area to complete the code for creating a Poisson editing composite. You will need to define the values that will fill the coefficient matrix A. This function fills the matrix A by looking at one pixel at a time, and its four neighbors

The code for this is:

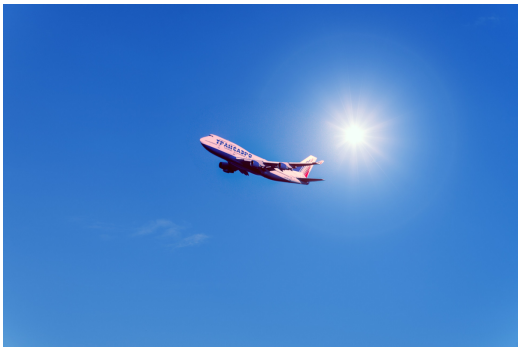
```
# if the pixel is in the mask, set the coefficient to -4
if mask[r,c] == True:
    A[ix,ix] = -4

if mask[r-1,c] == True: #checking to see if the neighbouring pixel is in the mask
    A[ix, tup2idx[(r-1,c)]] = 1 #if it is, set the corresponding coefficient in A to
     1
elif mask[r-1,c] == False: #if it is not in the mask, we need to modify matrix b
```

```
    b[ix] = b[ix] - target[r-1,c] #subtract the value of the pixel in the target
    image from the corresponding element in b

if mask[r+1,c] == True: #checking to see if the neighbouring pixel is in the mask
    A[ix, tup2idx[(r+1,c)]] = 1 #if it is, set the corresponding coefficient in A to
     1
elif mask[r+1,c] == False: #if it is not in the mask, we need to modify matrix b
    b[ix] = b[ix] - target[r+1,c] #subtract the value of the pixel in the target
    image from the corresponding element in b

if mask[r,c-1] == True: #checking to see if the neighbouring pixel is in the mask
    A[ix, tup2idx[(r,c-1)]] = 1 #if it is, set the corresponding coefficient in A to
     1
elif mask[r,c-1] == False: #if it is not in the mask, we need to modify matrix b
    b[ix] = b[ix] - target[r,c-1] #subtract the value of the pixel in the target
    image from the corresponding element in b

if mask[r,c+1] == True: #checking to see if the neighbouring pixel is in the mask
    A[ix, tup2idx[(r,c+1)]] = 1 #if it is, set the corresponding coefficient in A to
     1
elif mask[r,c+1] == False: #if it is not in the mask, we need to modify matrix b
    b[ix] = b[ix] - target[r,c+1] #subtract the value of the pixel in the target
    image from the corresponding element in b
```

The code will fill in one row of A at a time, so we are only focused on a single row.

- First we need to check if the pixel corresponding to the current row of A is in the mask, if its not we don't do anything, if it is we set the coefficient corresponding to I(x,y) to -4
- See if the neighbouring pixel is inside the mask, if it is we set the value of the coefficient in this row that corresponds to the neighbouring pixel to 1. If it is not in the mask we keep the said coefficient as 0 (all values in A are 0 by default) and we modify the corresponding value of b by subtracting the value of the target image at this neighbouring pixel.
- Repeat the process for the other 3 neighbouring pixels.

And then the code we are given will loop through this algorithm for each row and populate the entirety of A.
The result of implementing this algorithm is this:



(a) Poisson composition image

(b) Poisson composition image

As we can clearly see here in figures 17a and 17b the composite image is much better than the two previous iterations. In figure 17a it's more or less impossible to see the transition between the target and source image. In figure 17b we can see that the image is edited, but its still much better than the two previous ones. The reason that we can see the transition is because of the mountain in the target image. The mountain is such a different color and has a sharp transition between it and the background sky that when we use poisson image composition, we should expect to see some brown 'smudging' under the plane. It makes logical sense to see this if we think about how the poisson editing works, we are trying to unify the colors and the gradient of the target and source image, and since the target image has a 'disturbance' like the mountain, we should get something like this in our composite image. One thing to note as well is that the color of the plane has changed in both photos, this is also expected, and a known fault of this type of image composition. But its a sacrifice that's worth doing in many cases and is why we use it, in many cases its worth it to sacrifice the color of the source image a little bit to get a better border between the images.

Task 3

Exercise 1

1.1 Recover the signal in time domain.

We have the signal:

$$X(\omega) = 2 + (1+j) \cdot \pi \cdot [\delta(\omega + \omega_0)] + (1-j) \cdot \pi \cdot [\delta(\omega - \omega_0)]$$

In angular frequency domain. By taking the inverse fourier transform of $X(\omega)$ we can recover our signal in time domain.

$$x(t) = \mathcal{F}^{-1}\{X(\omega)\}$$

By using basic sum rules we can separate the terms of $X(\omega)$ and to the inverse fourier transform of each term.

$$x(t) = \mathcal{F}^{-1}\{2\} + \mathcal{F}^{-1}\{(1+j) \cdot \pi \cdot [\delta(\omega + \omega_0)]\} + \mathcal{F}^{-1}\{(1-j) \cdot \pi \cdot [\delta(\omega - \omega_0)]\}$$

The first term is rather simple, we can calculate it, or just look it up in a fourier table (like the one provided to us) which is what I have done.

$$\mathcal{F}^{-1}\{2\} = 2\sqrt{2\pi}\delta(t)$$

For the other two terms we can just use the following property of the Dirac delta:

$$\int_{-\infty}^{\infty} f(x)\delta(x-a) = f(a)$$

Since we have a dirac delta in each of the two other terms, we can apply this rule to get:

$$x(t) = \sqrt{2\pi}2\delta(t) + \frac{\pi}{\sqrt{2\pi}}((1+j)e^{-j\omega_0 t} + (1-j)e^{j\omega_0 t})$$

By using Eulers formula and expanding the term in the parentheses we get:

$$x(t) = \sqrt{2\pi}2\delta(t) + \frac{\pi}{\sqrt{2\pi}}(cos(\omega_0 t) - jsin(\omega_0 t) + jcos(\omega_0 t) + sin(\omega_0 t) + cos(\omega_0 t) + jsin(\omega_0 t) - jcos(\omega_0 t) + sin(\omega_0 t))$$

Cancelling out terms gives:

$$x(t) = \sqrt{2\pi}2\delta(t) + \frac{\pi}{\sqrt{2\pi}}(2cos(\omega_0 t) + 2sin(\omega_0 t))$$
$$x(t) = \underline{\underline{\sqrt{2\pi}\,(2\delta(t) + cos(\omega_0 t) + sin(\omega_0 t))}}$$

1.2 Recover the signal in time domain

$$X\left(e^{j\omega}\right) = \frac{1}{(1-\alpha e^{-j\omega})}\left(1 + \frac{1}{(1-\alpha e^{-j\omega})}\right), \omega \in \mathbb{R}$$

Since we are given the fourier transform tables I have assumed we can use them for these tasks, and therefore we can solve this task rather easily. We have:

$$\begin{aligned} X\left(e^{j\omega}\right) &= \frac{1}{(1-\alpha e^{-j\omega})}\left(1 + \frac{1}{(1-\alpha e^{-j\omega})}\right) \\ &= \frac{1}{(1-\alpha e^{-j\omega})} + \frac{1}{(1-\alpha e^{-j\omega})^2} \end{aligned}$$

Just plugging in values straight from this table [1] we get:

$$x[n] = \alpha^n u[n] + (n+1)\alpha^n u[n]$$
$$x[n] = \underline{\underline{(n+2)\alpha^n u[n]}}$$

Where u[n] is the unit step function, which is defined as 1 if $n \geq 0$ and 0 if $n < 0$.

1.3 Recover the signal in time domain

$$X(k) = 3\frac{\left(1-a^N\right)}{\left(1-a\cdot e^{-i2\pi k/N}\right)}, k \in [0; N]$$

This is also just a matter of looking at the fourier transform table. For this task I have used this table [2]. 3 is just a constant, so we can just leave it, giving us:

$$x[n] = \underline{\underline{3a^n}}$$

Exercise 2

1. Compute the cyclic convolution of the following 2D spatial filter:

$$w[x, y] = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 2 \\ 1 & 2 & 3 \end{bmatrix}$$

with the following 2D discrete unit impulse:

$$f[x,y] = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

A cyclic convolution is just a regular convolution, except for how we treat the boundaries. In a cyclic convolution we pad the edges with the other side of the matrix, we can imagine that we are wrapping around to the other side of the matrix, this will look like this:

$$\left|\begin{array}{ccccc} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right|$$

Here we can see the original matrix f[x,y] marked in grey, and the added padding in white, where the padding is what we get by 'wrapping' around the matrix, or equivalently translating the matrix. The result of the cyclical convolution w*f gives can be given by flipping w by 180° and overlaying it over the padded f, and then multiplying the values, like in a regular convolution. This is a bit difficult to illustrate, but the result is:

$$\begin{bmatrix} 2 \cdot 1 & 3 \cdot 1 + 1 \cdot 1 & 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 2 \\ 1 \cdot 1 & 2 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 & 3 \cdot 1 + 2 \cdot 1 \\ 1 \cdot 1 + 1 \cdot 1 & 2 \cdot 1 + 2 \cdot 1 & 3 \cdot 1 + 1 \cdot 1 \end{bmatrix}$$

Which gives:

$$\begin{bmatrix} 2 & 4 & 4 \\ 1 & 4 & 5 \\ 2 & 4 & 4 \end{bmatrix}$$

Note that all terms that equal zero have been excluded in the middle step here, as its unnecessary and messy.

2. Compute the convolution w*f using zero-padding.

This is the exact same operation as in the previous task, only that the padding is zeros, which gives:

$$\left|\begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right|$$

$$\begin{bmatrix} 0 & 1 \cdot 1 & 1 \cdot 1 + 1 \cdot 2 \\ 1 \cdot 1 & 2 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 & 3 \cdot 1 + 2 \cdot 1 \\ 1 \cdot 1 + 1 \cdot 1 & 2 \cdot 1 + 2 \cdot 1 & 3 \cdot 1 \end{bmatrix}$$

Which gives:

$$\begin{bmatrix} 0 & 1 & 3 \\ 1 & 4 & 5 \\ 2 & 4 & 3 \end{bmatrix}$$

3. Compare the results of question 1 and 2.

If we look at the two results we got we can see that they are similar, but still pretty different. As expected we get some different values on the edges, except for the cases where there is no difference in the padding. For example if we look at the bottom left value in f, we see that the cyclic convolution produces a padding of zeros for the padding that goes into calculation for this square. The value in the center is the same for both types of convolution, which makes sense, since the padding doesn't come into effect here, meaning that the padding only affects the edges, which in this case is most of the matrix, but if we had a much bigger matrix we would see that only the edges are affected by the difference in padding, and the majority of the resultant matrices would be equal.

Task 4

a) Load the data and use the coordinates in array X and greyscale values Z to plot the point cloud image z. This point cloud was made by subsampling the pixels of the image coffee.png. Compare your plot and the original image (in a few words).

Figure 18: Coffee.png

Figure 18 shows the original image (coffee.png) that the point cloud is subsampled from.

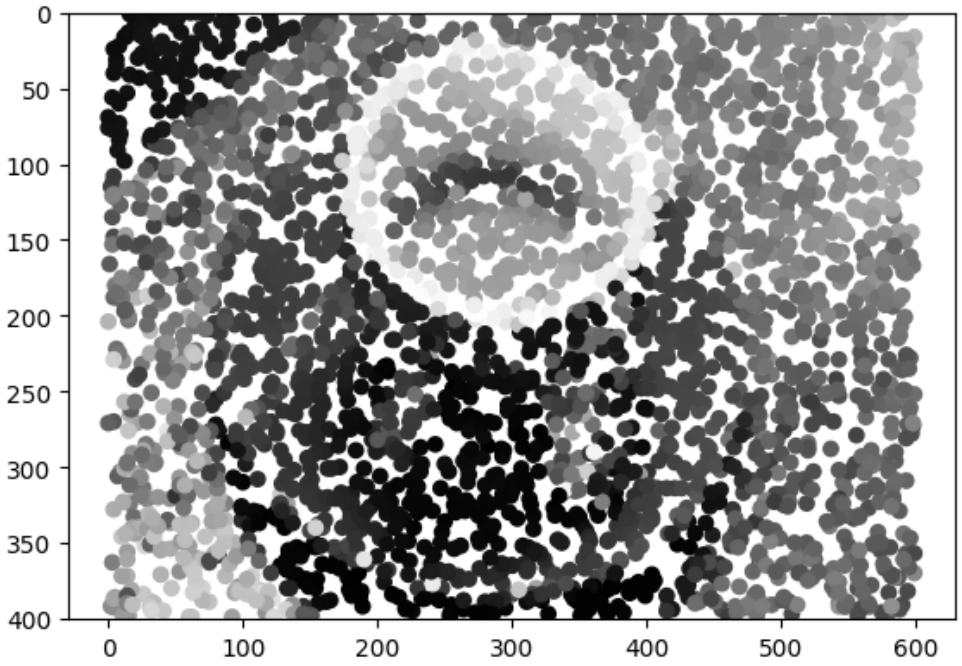


Figure 19: Scatter plot of point cloud image made by subsampling coffee.png

Figure 19 shows the point cloud image z. If we look at the image z we see that it resembles Fig 18, only it looks much worse. The reason that the point cloud image looks worse is quite simple, its a random subset of the pixels in coffee.png and therefore contains less information than the original image. Also the pixels are plotted as a scatterplot, which doesn't help, still it's clear that the image is a point cloud image of coffee.png.

b) Display the histogram of the point cloud intensity values. Give a short explanation about the shape of this histogram.

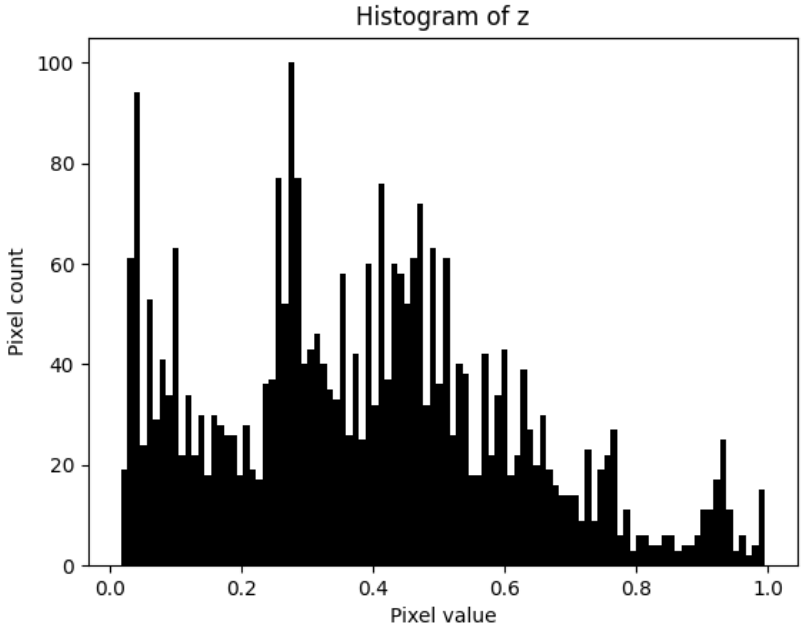


Figure 20: Histogram of point cloud image z

Figure 20 shows the histogram of the point cloud image z. What this histogram tells us is that the image is very gray, because of the 'lump' around the middle of the histogram. There is also not much white in the image, meaning its pretty dark. The values are also clumped around the middle, indicating a very low contrast image, the fact that there are no completely black pixels also contribute to a low contrast image.

c) Using the graph Laplacian L, compute the Graph Fourier modes of this irregular image domain.

The graph fourier modes of this irregular image domain, are the same as the same as the eigenvectors of the image z. We also have that the fourier 'frequencies' are the same as the eigenvalues. Since we already have the laplacian L this process is very simple. By using the numpy library we can use the following line of code to compute both the fourier modes and the frequencies.

```
1 eigenvalues, eigenvectors = np.linalg.eigh(L)
```

Here the function "np.linalg.eigh()" is a built in function that calculates the eigenvalues and eigenvectors of the matrix.. This function also gives the eigenvalues sorted by increasing value, so there is no need to sort the matrix. Note that in the code for this task I have added a check for this,

and for checking if all values are positive, these code snippets will give a warning if they are not all sorted and positive so I can go in and fix it if necessary.

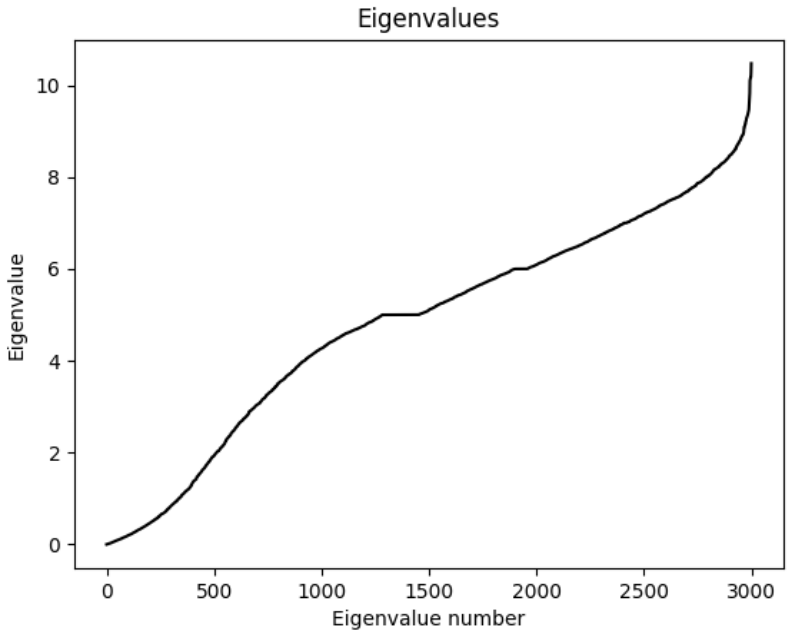
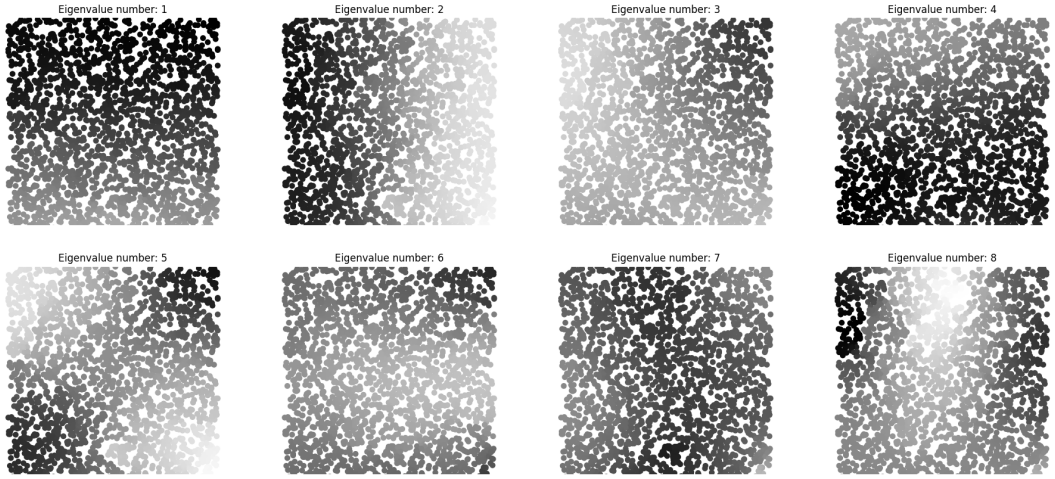


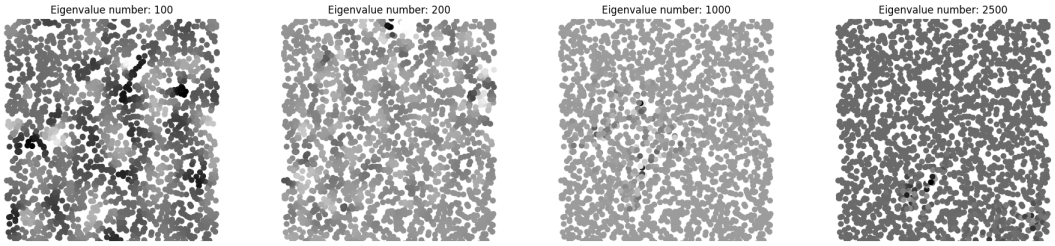
Figure 21: Plot of eigenvalues of z

Figure 21 shows the plot of all eigenvalues of the image z, as we can see they are all positive and in increasing order. (again this is double checked in the python script).

d) Plot the Fourier mode associated to the first non-zero eigenvalue on the graph. In other words, do a scatterplot of the pixels, as in question 1), where the original image intensities have been replaced by the eigenvector values. What do you observe? Plot a few different other Fourier modes and explain what you see.



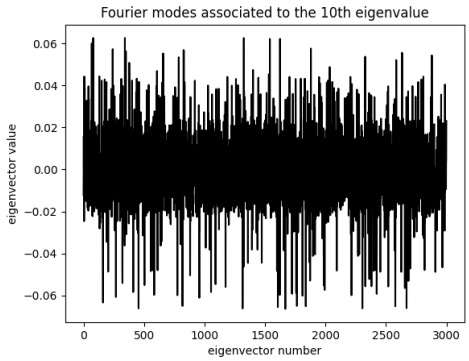
(a) Fourier mode associated with eigenvalues 1-10



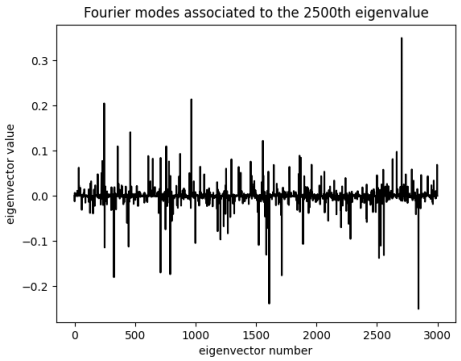
(b) Fourier modes associated with eigenvalues 100, 200, 1000, and 2500.

In figures 22a and 22b I have plotted the fourier modes associated with some of the eigenvalues. In the images we can see that the fourier modes of each eigenvalue corresponds to different information about the original image. We can see that the different eigenvalues show different areas of the image being lighter and darker, and this corresponds to different sets of information about the image z. If we think about it in terms of frequency domain we can say that they contain different sinusoidal components of the image, where low eigenvalues correspond to higher frequency components and higher eigenvalues correspond to lower frequency components. We can also show this by plotting the fourier modes (Figs 23a, 23b). So in conclusion we can say that the fourier modes associated

with each eigenvalue contain different sets of information about the image, which also makes sense if we think about what eigenvectors and eigenvalues really are.



(a) Line plot of fourier modes associated with eigenvalue 10



(b) Line plot of fourier modes associated with eigenvalue 2500

e) Perform the Graph Fourier Transform F(z) of the image z. It should be a vector with the same size as Z. Plot F(z) on a figure where the x-axis represents the Laplacian eigenvalue "graph frequency". Which frequency band the image have the highest values in?

To solve this task I used this equation:

$$\mathcal{GF}[f](\lambda_l) = \sum_{i=1}^{N} f(i)\mu_l^T(i), [3]$$

Where λ_l and μ_l are just corresponding arbitrary eigenvalue and eigenvectors respectively. And N is the number of eigenvalues we have. The above equation needs to be implemented once for each eigenvalue, computing this is actually very straightforward. We have a matrix of all the eigenvectors corresponding to each eigenvalue, and we have a matrix for z which is just Z, so implementing the above equation for each eigenvalue is just describing a matrix multiplication. Which gives

$$\mathcal{GF}[z] = \mu^T @ Z$$

Here μ is the matrix called 'eigenvectors' in my code, which is the N*N matrix of eigenvectors, and Z is the matrix we are given that represents the greyscale values of the image. And @ denotes a matrix multiplication.

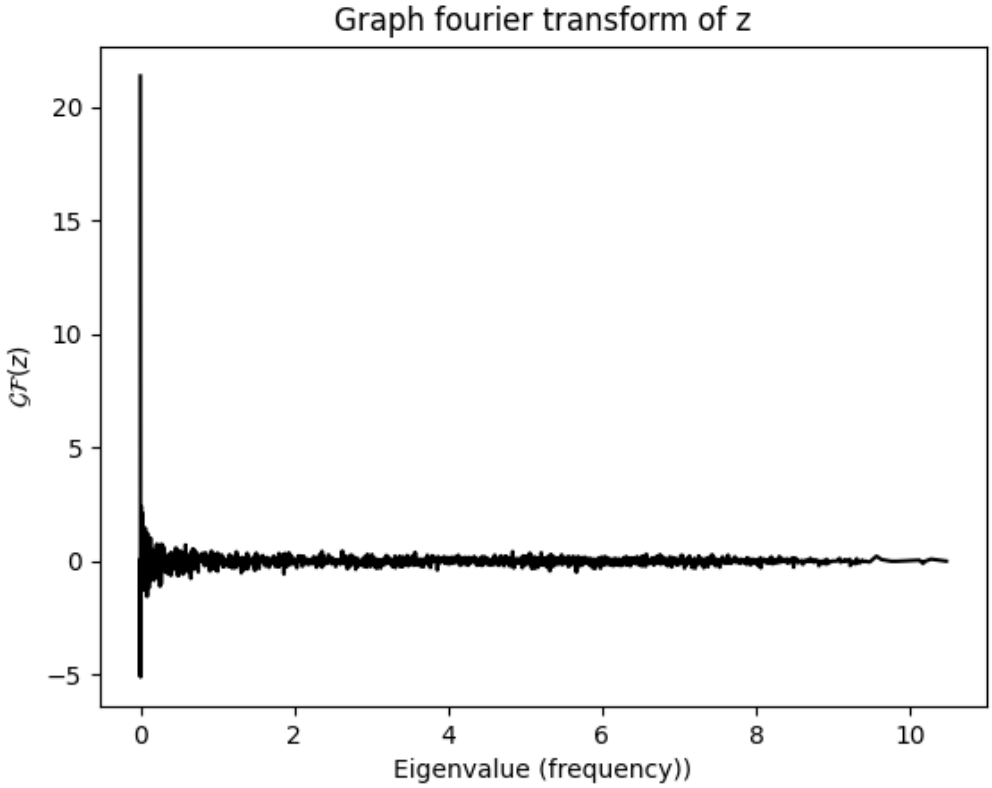
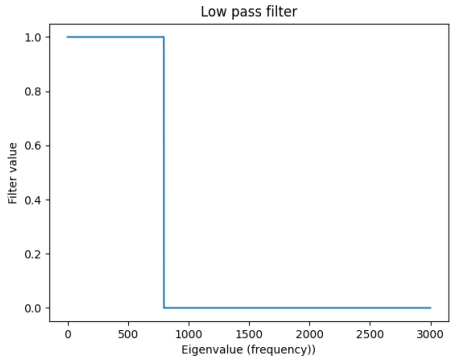


Figure 24: Graph fourier transform of z

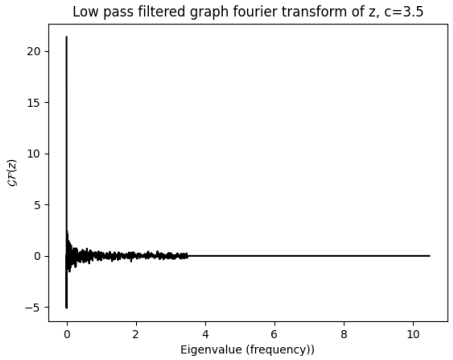
Figure 24 shows the result of the graph fourier transform, what we can immediately see is that the lower frequencies definitely have the highest values. We can see a very big spike around zero, and then it decreases with increasing frequency.

f) Apply an ideal lowpass filter to F(z) with a cutoff frequency c of your choice and perform the inverse graph Fourier transform to get a lowpass filtered version of z. Plot the resulting point cloud image. What do you observe? Adjust the cutoff frequency c such that the effect is visible on the image and give the value of c.

To create the ideal low pass filter I simply just created a copy of the 'eigenvalues' array and iterated through it and defined that all values below the cutoff is equal to 1, and all other values are 0. This way I have an array of the same shape as 'eigenvalues' which is 1 for the values we want to keep, and 0 else, multiplying this by the gft of z then filters out all frequencies above the cutoff. Note that the filter I have created here is the same as a unit step function shifted to the cutoff frequency c.



(a) Ideal lowpass filter (c=3.5)



(b) Ideal lowpass filter applied to the gft (c=3.5)

Figure 25a shows a plot of the ideal lowpass filter, and figure 25b shows the gft after applying the ideal lowpassfilter.
To see what the resultant image looks like we need to take the inverse graph fourier transform which is given by:

$$\mathcal{IGF}[\hat{f}](i) = f(i) = \sum_{l=0}^{N-1} \hat{f}(\lambda_l)\, \mu_l(i), [3]$$

The same logic used for the forward graph fourier transform can be applied here, giving us:

```
lowpassfiltered_Z = eigenvectors @ lowpassfiltered_GFT
```

Meaning we have that the IGFT of the lowpass filtered image is the eigenvectors matrix multiplied with the filtered GFT of z. The resultant image is:

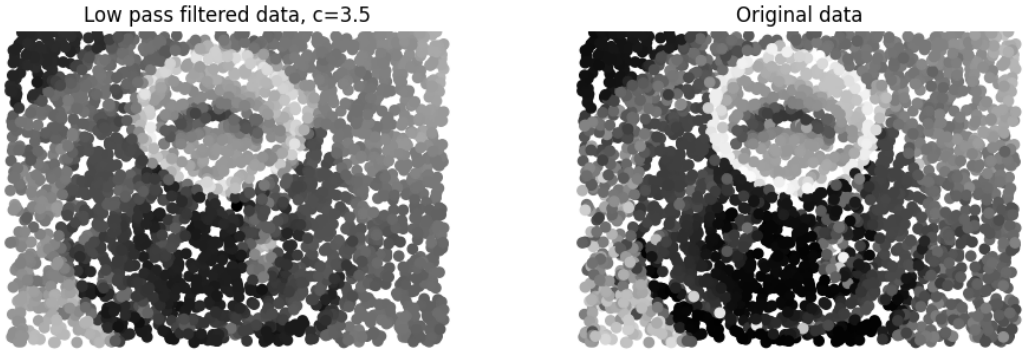
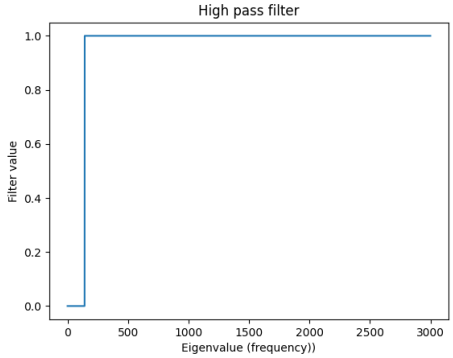


Figure 26: Lowpassfiltered z (c=3.5)

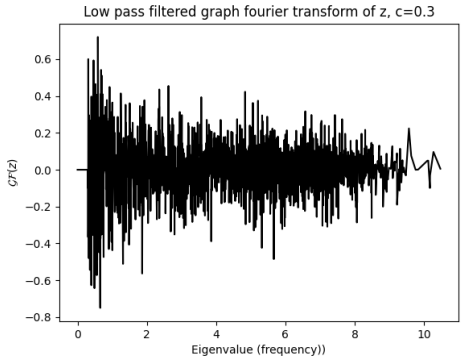
Figure 26 shows the lowpassfiltered image on the left side as well as the original point cloud image on the left. The cutoff frequency for the lowpass filter was chosen to be c=3.5, the reason for this exact value is just that it shows the effect of the filter on the image in a very clear way. What we can observe from looking at the lowpassfiltered image is that the higher frequency components of the image have been filtered out, this is especially visible around the opening of the coffee cup. Here we can see quite a sharp transition from black/grey to white in the unfiltered image, in comparison the LP-filtered image has a smoother transition and its more grey. Which corresponds to a lower frequency. We can also see that in the top left corner of the image, the transition from grey to black is sharper in the unfiltered image. There is not much more high frequency components of the image, so the filtering is not very visible elsewhere in the image, but we have without a doubt successfully filtered the image.

g) Same question as in 6), but replace the lowpass filter by an ideal high-pass filter.

For this task I did the exact same as in the previous task only I converted the lowpass filter to a highpass filter by taking one minus the lowpass filter, which logically, corresponds to a highpassfilter.



(a) Ideal highpass filter (c=0.3)



(b) Ideal highpass filter applied to the gft (c=0.3)

Figures 27a and 27b show the highpassfilter and the highpassfiltered graph fourier transform of z. Note that the scales on figures 27b and 25b are not equal, which is why they look so different.

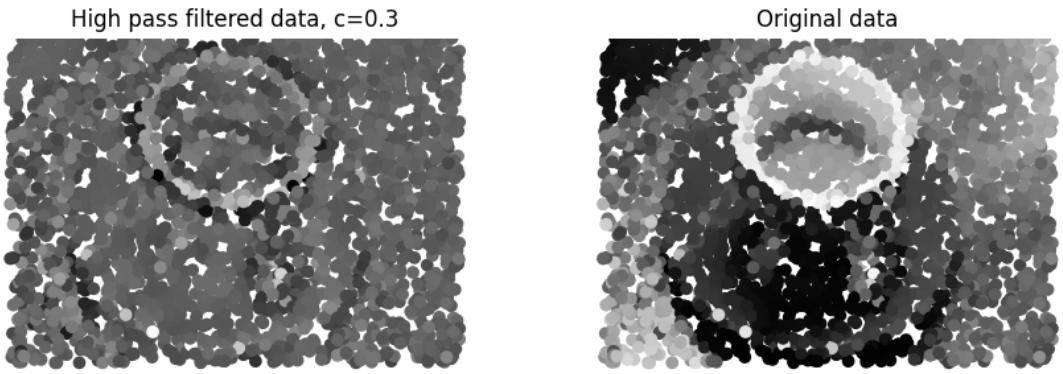
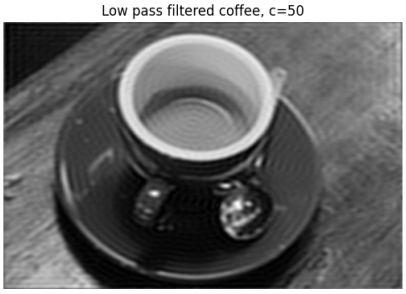


Figure 28: Highpassfiltered z (c=0.3)

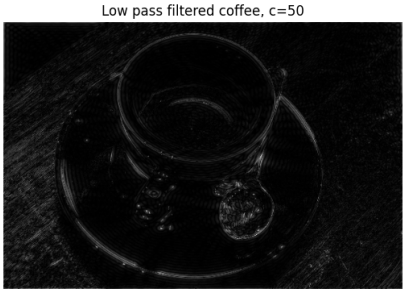
Figure 28 shows the highpassfiltered image on the left and the original on the right. The value for c, was again chosen by looking at the filtered image and seeing what shows that the image has been filtered properly. Here we can very clearly see that we have successfully applied the highpassfilter, we can clearly only see the high frequency components on the image. We can see the border of the top of the coffee cup as well as the border of the saucer and the edge of the table. All of which are the sharp transitions in the image, which corresponds to the high frequencies. So I would say that this is a very good result of highpass filtering.

h) Conclusion: does the filtering on this graph/point cloud domain works as in the standard image domain? Justify your answer in a few sentences.

From looking at figures 26 and 28 we can conclude that the filtering in this graph domain works in the same way as for a standard image. We would get more or less the same result if we applied a filter to a 'regular' image in frequency domain. We see clearly that the respective frequencies have been filtered out of the image. To prove this a bit more conclusively we can apply a highpass- and lowpassfilter to the coffee.png image in frequency domain and see that we get a similar result.



(a) coffee.png lowpassfiltered in frequency domain



(b) coffee.png highpassfiltered in frequency domain

Here we can see in figures 29a and 29b the result of filtering the image coffee.png in the frequency domain. I just wrote some quick code to implement the filters, both are ideal and the cutoff frequency is in the image title, the exact way they work and are implemented are not important, so I have excluded them here. We can see here that we get the same results as we got from filtering them in the graph/point cloud domain. So we can conclusively say that the filtering works the same way in this graph/point cloud domain as in the standard image domain.

References

[1] Engineering Tables/DTFT Transform Table - Wikibooks, open books for an open world, February 2023. [Online; accessed 6. Mar. 2023].

[2] Engineering Tables/DFT Transform Table - Wikibooks, open books for an open world, February 2023. [Online; accessed 6. Mar. 2023].

[3] Contributors to Wikimedia projects. Graph Fourier transform - Wikipedia, October 2022. [Online; accessed 7. Mar. 2023].

Note that the textbook (Digital Image Processing 4th edition by Rafael C. Gonzales & Richard E: Woods) and the lecture notes have been used for help, inspiration and information throughout, but as they are the reading material for the course they have not been cited throughout.

Code

Task 1(a-d)

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# importing the images
jupiter1 = cv2.imread(r'Homeexam supplementary data\Jupiter1.png')
```

```
jupiter2 = cv2.imread(r'Homeexam supplementary data\Jupiter2.png')

"""---Task b---"""
# creating the histograms
jupiter1_hist, bin_edges_jup1 = np.histogram(np.array(jupiter1), bins=256)
jupiter2_hist, bin_edges_jup2 = np.histogram(np.array(jupiter2), bins=256)

# plotting the greyscale histograms
plt.bar(bin_edges_jup1[0:-1], jupiter1_hist, color = 'k')
plt.xlabel('Pixel value')
plt.ylabel('Number of pixels')
plt.title('Greyscale histogram of Jupiter1.png')
plt.show()

plt.bar(bin_edges_jup2[0:-1], jupiter2_hist, color = 'k')
plt.xlabel('Pixel value')
plt.ylabel('Number of pixels')
plt.title('Greyscale histogram of Jupiter2.png')
plt.show()

# creating the color histogram of jupiter1
for i in range(0, 3):
    histogram, bin_edges = np.histogram(np.array(jupiter1)[:, :, i], bins=256, range
    =(0, 256))
    plt.plot(bin_edges[0:-1], histogram, color='bgr'[i])
plt.title("Color Histogram of Jupiter1.png")
plt.xlabel("Color value")
plt.ylabel("Pixel count")
plt.show()


"""---Task c---"""

def notch_reject_filter(shape, Q, u_k, v_k):
    """Creates a notch reject filter with a circle of zeros around (u_k, v_k) with
    radius Q,
    where u_k and v_k is the distance from the center of the filter to the center of
     the circle of zeros."""
    M, N = shape
    # Initialize filter with zeros
    H = np.zeros((M, N))

    # Traverse through filter
    for u in range(0, M):
        for v in range(0, N):
            #find the distance from the center (using pythagoras)
            D_k = np.sqrt((u - M/2 - u_k)**2 + (v - N/2 - v_k)**2)
            D_min_k = np.sqrt((u - M/2 + u_k)**2 + (v - N/2 + v_k)**2)

            if D_min_k <= Q or D_k <= Q:
                H[u, v] = 0.0 # creating a circle of zeros around with radius Q at
    pos (u_k, v_k)
            else:
                H[u, v] = 1.0 # set to one everywhere else
    return H

# applying the filters to jupiter1

```

```
# Firstly I split the image into its separate color channels
# since the different color channels are not affected by the same noise
# so they need different filters
blue, green, red = cv2.split(jupiter1) # separating color channels

# creating the magnitude spectrum of the red channel
# so we can see which frequencies to filter out
magnitude_img = np.fft.fft2(red)
magnitude_img = np.fft.fftshift(magnitude_img)
magnitude_img = np.log(np.abs(magnitude_img))

# creating the notch reject filter
H1 = notch_reject_filter(red.shape, Q= 3.0, u_k=7, v_k=7) #blocking out the bright
    spots
H2 = np.full((red.shape), 1.0)
for i in range (red.shape[0]): #blocking out the bright lines
    for j in range (red.shape[0]):
        if i == 249 and j <249:
            H2[i, j] = 0.0
        elif j == 249 and i < 249:
            H2[i, j] = 0.0
        elif i == 263 and j > 263:
            H2[i,j] = 0.0
        elif j == 263 and i > 263:
            H2[i,j] =0.0
        elif i == 249 and j > 263:
            H2[i, j] = 0.0
        elif j == 249 and i > 263:
            H2[i, j] = 0.0
        elif i == 263 and j < 249:
            H2[i, j] = 0.0
        elif j == 263 and i < 249:
            H2[i, j] = 0.0
H = H1*H2

# applying the nr filter to the red channel
red = np.fft.fft2(red) # fourier transform
red = np.fft.fftshift(red) # shift zero frequency to center
red = red*H # apply filter
red = np.fft.ifftshift(red) # shift zero frequency back
red = np.fft.ifft2(red) # inverse fourier transform
red = np.abs(red) # get magnitude

# plotting the magnitude spectrum of the red channel and the filter

plt.imshow(H*magnitude_img, cmap='gray')
plt.title('Notch reject filter applied to red channel')
plt.axis('off')
plt.show()

plt.imshow(magnitude_img, cmap='gray')
plt.title('Magnitude spectrum of red channel')
plt.axis('off')
plt.show()
red = np.array(red, dtype=np.uint8)# converting back to uint8 (the 'correct' data
    type)

# then we will apply a median filter (twice) to the blue channel
```

```
blue = cv2.medianBlur(cv2.medianBlur(blue, 3), 3)
blue = np.array(blue, dtype=np.uint8)

# then we will remove the pepper in the green channel
# by applying a contraharmonic mean filter (Q=0 so its actually just an arithmetic mean filter)
def contraharmonic_mean(img, size, Q):
    kernel = np.full(size, 1.0) # creating the kernel
    a = cv2.filter2D((np.power(img, (Q + 1))), -1, kernel, borderType=cv2.BORDER_REPLICATE)
    b = cv2.filter2D((np.power(img, Q)), -1, kernel,  borderType=cv2.BORDER_REPLICATE)
    filtered = a/b
    return filtered

green = contraharmonic_mean(green,(3,3), 0.0)
green = np.array(green, dtype=np.uint8)

# then we will merge the color channels back together
restored_jupiter1 = cv2.merge((blue, green, red))
restored_jupiter1 = np.array(restored_jupiter1, dtype=np.uint8)

#plotting the final result
cv2.imshow('Restored Jupiter1', restored_jupiter1)
cv2.waitKey(0)
cv2.destroyAllWindows()

"""---Task d---"""

def filter_img(img, H):
    """Applies a filter H on an image img (filter is applied on all color channels)
    """
    #plot magnitude spectrum of image
    magnitude_img = np.fft.fft2(cv2.cvtColor(img, cv2.COLOR_BGR2GRAY))
    magnitude_img = np.fft.fftshift(magnitude_img)
    magnitude_img = np.log(np.abs(magnitude_img))

    blue,green,red = cv2.split(img) # separating color channels
    color_channels = [blue, green, red]
    for i in range(3):
        color_channels[i] = np.fft.fft2(color_channels[i]) # fourier transform
        color_channels[i] = np.fft.fftshift(color_channels[i]) # shift zero frequency to center
        color_channels[i] = color_channels[i]*H # apply filter
        color_channels[i] = np.fft.ifftshift(color_channels[i]) # shift zero frequency back
        color_channels[i] = np.fft.ifft2(color_channels[i]) # inverse fourier transform
        color_channels[i] = np.abs(color_channels[i]) # get magnitude

    img = cv2.merge((color_channels[0], color_channels[1], color_channels[2]))
    img = np.array(img, dtype=np.uint8)

    #plotting the magnitude spectrum of the image and the filter
    plt.imshow(H*magnitude_img, cmap='gray')
    plt.title('Magnitude spectrum of notch reject filter')
    plt.axis('off')
    plt.show()
```

```
    plt.imshow(magnitude_img, cmap='gray')
    plt.title('Magnitude spectrum of image')
    plt.axis('off')
    plt.show()
    return img

# applying the filters to jupiter2
H = notch_reject_filter(jupiter2[:,:,0].shape, Q=1.0, u_k=5, v_k=0)
notchfiltered_jupiter2 = filter_img(jupiter2, H)
restored_jupiter2 = cv2.medianBlur(cv2.medianBlur(notchfiltered_jupiter2, 3), 3)

#plotting the final result
cv2.imshow('Restored Jupiter2', restored_jupiter2)
cv2.waitKey(0)
cv2.destroyAllWindows()

# plotting the color histograms of the original, and restored jupiter1
for i in range(0, 3):
    histogram, bin_edges = np.histogram(np.array(jupiter1)[:, :, i], bins=256, range
    =(0, 256))
    plt.plot(bin_edges[0:-1], histogram, color='bgr'[i])
plt.title("Color Histogram of Jupiter1.png")
plt.xlabel("Color value")
plt.ylabel("Pixel count")
plt.show()

for i in range(0, 3):
    histogram, bin_edges = np.histogram(np.array(restored_jupiter1)[:, :, i], bins
    =256, range=(0, 256))
    plt.plot(bin_edges[0:-1], histogram, color='bgr'[i])
plt.title("Color Histogram of restored Jupiter1.png")
plt.xlabel("Color value")
plt.ylabel("Pixel count")
plt.show()
```

Task 1e

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

#importing the restored images
jupiter1 = cv2.imread(r'Plots\restored_jupiter1.png')
jupiter2 = cv2.imread(r'Plots\restored_jupiter2.png')

"""Creating filters"""
def contrast_stretching(img, min, max):
    """Performs contrast stretching on the input image, min and max
    refer to the values that are stretched, i.e. min becomes 0 and max becomes 255
    """
    img = np.clip(img, min, max)
    newmin = 0
    newmax = 255
    newimg = (img - min) / (max - min) * (newmax - newmin) + newmin
    return newimg

def apply_contrast_stretching(img, oldmin=20, oldmax=240):
    """Applies contrast stretching to to the input image, converting oldmin to 0 and
```

```
     oldmax to 255
    (only works for color images)"""
    b, g, r = cv2.split(img)
    r = contrast_stretching(r, oldmin, oldmax)
    g = contrast_stretching(g, oldmin, oldmax)
    b = contrast_stretching(b, oldmin, oldmax)
    new_img = cv2.merge((b, g, r))
    new_img = np.array(new_img, dtype = 'uint8')
    return new_img

def apply_gamma_correction(img, gamma=1.1):
    """Performs gamma correction on the input image"""
    img = np.array(255*(img / 255) ** gamma, dtype = 'uint8')
    return img

def laplacian(f, c):
    """Calculates the laplacian of the input image,
    and adds it to the image with a constant c (output is clipped to (0,1))"""
    f = f/255 #normalize the input function
    F = np.fft.fft2(f) #2D discrete Fourier transform
    F = np.fft.fftshift(F) # shifting the ffrequency components

    P, Q = F.shape #dimensions of the image
    H = np.zeros((P, Q), dtype=np.float32) #initializing the transfer function
    for u in range(P):
        for v in range(Q):
            # iterating through the transfer function, and calculating appropriate
    values
            D2 = ((u-P/2)**2 + (v-Q/2)**2) #euclidean distance from center squared
            H[u][v] = -4*(np.pi**2)*D2 # transfer function

    Lap = np.fft.ifftshift(H*F) # calculating the laplacian and shifting the
    frequency components back
    Lap = np.real(np.fft.ifft2(Lap)) # discarding the imaginary part of the
    laplacian (we only care about the real part)

    # we need to scale the laplacian to be between -1 and 1
    # (since the values are very big, so we need the laplacian and the image to be
    on the same scale)
    oldrange = (np.max(Lap) - np.min(Lap))
    newrange = 1 - -1
    Lap_scaled = (((Lap - np.min(Lap)) * newrange) / oldrange) - 1 #scaling the
    laplacian to (-1, 1)


    g = f + c*Lap_scaled # adding the laplacian to the image (scaled by c)
    g = np.clip(g, 0, 1) #clipping the image to (0, 1) since we need the image to be
     no
    return g

def apply_laplacian(img, red_c = -1, green_c = -1, blue_c = -1):
    """Applies the laplacian filter to input image (only works for color images)
    The values of c are chosen to give the best results"""
    b, g, r = cv2.split(img)
    b = laplacian(b, blue_c)*255
    g = laplacian(g, green_c)*255
    r = laplacian(r, red_c)*255
    b = np.array(b, dtype = 'uint8')
```

```
    g = np.array(g, dtype = 'uint8')
    r = np.array(r, dtype = 'uint8')
    x = cv2.merge((b, g, r))
    return x

"""Applying filters"""
# For jupiter1
# contrast stretching
jupiter1_contrast = apply_contrast_stretching(jupiter1, 15, 245)

# gamma correction
jupiter1_gamma = apply_gamma_correction(jupiter1_contrast, 1.2)

# laplacian
jupiter1_laplacian = apply_laplacian(jupiter1, -0, -0.2, -1)

#compare with original
compare1 = np.concatenate((jupiter1, jupiter1_contrast), axis=1)
compare2 = np.concatenate((jupiter1_gamma, jupiter1_laplacian), axis=1)
compare = np.concatenate((compare1, compare2), axis=0)
cv2.imshow('Before and after enhancement jupiter1', compare)
cv2.waitKey(0)
cv2.destroyAllWindows()


# For jupiter2
# contrast stretching
jupiter2_contrast = apply_contrast_stretching(jupiter2, 25, 225)

# gamma correction
jupiter2_gamma = apply_gamma_correction(jupiter2_contrast, 1.05)

# compare with original
compare2 = np.concatenate((jupiter2, jupiter2_contrast, jupiter2_gamma), axis=1)
cv2.imshow('Before and after enhancement jupiter2', compare2)

cv2.waitKey(0)
cv2.destroyAllWindows()

#show before and after histogram of jupiter2
plt.hist(jupiter2.ravel(),256,[0,256], color = 'k')
plt.title('Histogram of jupiter2 before enhancement')
plt.xlabel('Pixel value')
plt.ylabel('Pixel count')
plt.show()

plt.hist(jupiter2_gamma.ravel(),256,[0,256], color = 'k')
plt.title('Histogram of jupiter2 after enhancement')
plt.xlabel('Pixel value')
plt.ylabel('Pixel count')
plt.show()
```

Task 4

```
import numpy as np
import matplotlib.pyplot as plt

#importing the data files
```

```
point_cloud = np.load(r'Homeexam supplementary data\point_cloud.npz')
point_cloud_laplacian = np.load(r'Homeexam supplementary data\point_cloud_Laplacian.npz')

#extracting the data into arrays
X = point_cloud['X'] #coordinates
Z = point_cloud['Z'] #greyscale values
L = point_cloud_laplacian['L'] #Laplacian

"""---Task 4.1---"""
# creating a scatter plot of the data with (0,0) at top left corner
plt.scatter(X[:,1], X[:,0], c=Z, cmap='gray')
plt.ylim(400, 0)
plt.show()

"""---Task 4.2---"""
#plot histogram of Z
plt.hist(Z, bins=100, color='k')
plt.title('Histogram of z')
plt.xlabel('Pixel value')
plt.ylabel('Pixel count')
plt.show()

"""---Task 4.3---"""
# computing the eigenvalues and eigenvectors of the laplacian (frequencies and fourier modes)
eigenvalues, eigenvectors = np.linalg.eigh(L)

# plotting the eigenvalues
plt.plot(eigenvalues, color='k')
plt.title('Eigenvalues')
plt.xlabel('Eigenvalue number')
plt.ylabel('Eigenvalue')
plt.show()

#implementing a few checks to make sure the eigenvalues are sorted and positive and that the eigenvectors are the right dimensions
# checking the dimensions of the eigenvectors
print(eigenvectors.shape)

# check if we have all positive values
if np.min(eigenvalues) < 0:
    print('The eigenvalues are not all positive...')

#check if the eigenvalues are sorted
for i in range(len(eigenvalues)-1):
    if eigenvalues[i] > eigenvalues[i+1]:
        print('The eigenvalues are not sorted...')
        break

"""---Task 4.4---"""
# plotting the fourier modes associated to the n-th eigenvalue
# note that the eigenvectors are the columns in the eigenvectors matrix, not the rows
n = 1
plt.scatter(X[:,0], X[:,1], c=eigenvectors[:,n], cmap='gray')
plt.axis('off')
plt.ylim(400, 0)
```

```
plt.title('Eigenvalue number: ' + str(n))
plt.show()

# also plotting the  line plot of the fourier mode
plt.plot(eigenvectors[:,n], color='k')
plt.title('Eigenvalue number: 10')
plt.xlabel('eigenvector number')
plt.ylabel('eigenvector value')
plt.title('Fourier modes associated to the ' + str(n) + 'th eigenvalue')
plt.show()

"""---Task 4.5---"""
# computing the graph fourier transform of z
GFT = eigenvectors.T @ Z # computing the graph fourier transform

# plotting the graph fourier transform of z
plt.plot(eigenvalues, GFT, color='k')
plt.title('Graph fourier transform of z')
plt.xlabel('Eigenvalue (frequency))')
plt.ylabel('$\mathcal{GF}(z)$')
plt.show()

# double checking the dimensions of GFT
print(GFT.shape)


"""---Task 4.6---"""
# creating the low-pass filter with cutoff frequency c
c=3.5 # cutoff frequency
filter = np.copy(eigenvalues)
for i in range(len(filter)):
    if filter[i] < c:
        filter[i] = 1
    else:
        filter[i] = 0

lowpassfiltered_GFT = GFT*filter # filtering the gft

# plotting the filter
plt.plot(filter)
plt.title('Low pass filter')
plt.xlabel('Eigenvalue (frequency))')
plt.ylabel('Filter value')
plt.show()

# plotting the filtered graph fourier transform
plt.plot(eigenvalues,lowpassfiltered_GFT, color='k')
plt.title('Low pass filtered graph fourier transform of z, c='+ str(c))
plt.xlabel('Eigenvalue (frequency))')
plt.ylabel('$\mathcal{GF}(z)$')
plt.show()

# computing the inverse graph fourier transform of the filtered graph fourier
    transform of z
lowpassfiltered_Z = eigenvectors @ lowpassfiltered_GFT

# plotting the filtered data and the original data
plt.subplot(1,2,1)
```

```
plt.scatter(X[:,1], X[:,0], c=lowpassfiltered_Z, cmap='gray')
plt.axis('off')
plt.ylim(600, 0)
plt.title('Low pass filtered data, c='+ str(c))
plt.subplot(1,2,2)
plt.scatter(X[:,1], X[:,0], c=Z, cmap='gray')
plt.axis('off')
plt.ylim(600, 0)
plt.title('Original data')
plt.show()


"""---Task 4.7---"""
# creating the high-pass filter with cutoff frequency c
c=0.3 # cutoff frequency
filter = np.copy(eigenvalues)
for i in range(len(filter)):
    if filter[i] < c:
        filter[i] = 1
    else:
        filter[i] = 0

highpassfiltered_GFT = GFT*(1-filter) # filtering the gft

# computing the inverse graph fourier transform of the filtered graph fourier
    transform of z
highpassfiltered_Z = eigenvectors @ highpassfiltered_GFT

# plotting the filter
plt.plot(1-filter)
plt.title('High pass filter')
plt.xlabel('Eigenvalue (frequency))')
plt.ylabel('Filter value')
plt.show()

# plotting the filtered graph fourier transform
plt.plot(eigenvalues,highpassfiltered_GFT, color='k')
plt.title('Low pass filtered graph fourier transform of z, c='+ str(c))
plt.xlabel('Eigenvalue (frequency))')
plt.ylabel('$\mathcal{GF}(z)$')
plt.show()

# plotting the filtered image and the original image
plt.subplot(1,2,1)
plt.scatter(X[:,1], X[:,0], c=highpassfiltered_Z, cmap='gray')
plt.axis('off')
plt.ylim(600, 0)
plt.title('High pass filtered data, c='+ str(c))
plt.subplot(1,2,2)
plt.scatter(X[:,1], X[:,0], c=Z, cmap='gray')
plt.axis('off')
plt.ylim(600, 0)
plt.title('Original data')
plt.show()


"""---Task 4.8---"""
# filtering the original image (coffee.png) with a low-pass filter and a high-pass
```

```
    filter to compare the results with the results from task 4.6 and 4.7
coffee = plt.imread('Homeexam supplementary data\coffee.png') # loading the image
coffee_ft = np.fft.fftshift(np.fft.fft2(coffee)) # computing the fourier transform
    of the image

# creating the filter
filter = np.copy(coffee_ft)
c=50
for i in range(len(filter)):
    for j in range(len(filter[0])):
        if np.sqrt((i-len(filter)/2)**2+(j-len(filter[0])/2)**2) < c:
            filter[i][j] = 1
        else:
            filter[i][j] = 0

#plotting the filter
plt.imshow(np.abs(filter), cmap='gray')
plt.axis('off')
plt.title('Low pass filter, c='+ str(c))
plt.show()

highpass_coffee_ft = coffee_ft*(1-filter) # highpassfiltering in the frequency
    domain
lowpass_coffee_ft = coffee_ft*filter # lowpassfiltering in the frequency domain
highpass_coffee = np.fft.ifft2(np.fft.ifftshift(highpass_coffee_ft)) # computing the
     inverse fourier transform of the highpassfiltered image
lowpass_coffee = np.fft.ifft2(np.fft.ifftshift(lowpass_coffee_ft)) # computing the
    inverse fourier transform of the lowpassfiltered image

# plotting the results
plt.imshow(np.abs(lowpass_coffee), cmap='gray')
plt.axis('off')
plt.title('Low pass filtered coffee, c='+ str(c))
plt.show()
plt.imshow(np.abs(highpass_coffee), cmap='gray')
plt.axis('off')
plt.title('Low pass filtered coffee, c='+ str(c))
plt.show()
```