1 Problem 1

To darken the image I have chosen to convert the image to HSV colour space and multiply the value component of each pixel by a scale factor (the darkening coefficient). I chose to do it in the HSV space instead of BGR because it is more apparent what is happening to the image as the value component is analogous to the brightness of the pixel. Then for the simple light leak mask I have used the original image and specified a region defined by 4 bounding lines defined as lambda functions. Within this region I have used a polynomial $(-\frac{1}{h} * (y-h)^2 + h)$ where h is half the image height) to vary the brightness to give the effect of light leaking through a window. The brightness peaks at half of the image's height and then fades off as the light approaches the edges of the image. This deals with the light leaking vertically. To make the light leak effect horizontally I apply a Gaussian blur to the light leak mask which decreases the sharpness of the edges and blends the colours at the edges so they are smoother. After this, I blend the images according to the blending coefficient to overlay the light leak mask on the darkened image.

For the rainbow mask the bounding region, Gaussian blur and image blend are all the same. The difference is in how the mask is generated within the region. This was a bit more difficult because the region is slanted. I create a copy of the image and convert it to the HSV colour space. I then calculate the total width of the region. Next, for each pixel inside the region I vary the hue with respect to the total width of the region to create the rainbow effect and then apply the same polynomial to the value component in the light leak mask to create the light leak effect for the rainbow.

For the Gaussian blur I chose to use parameters $n = 4$ giving a neighbourhood of width and height $2 * n + 1 = 9$ with $\sigma = 2.2$. This is quite a substantial blur but it was very good at giving the light leak effect in my experimentation and produced an effect that I felt was suitable for the task.

In terms of the computational complexity it is $O(height * width * channels * n^2)$ (where n is the neighbourhood size) since the Gaussian blur requires looping over each pixel on each channel which is a triple-nested loop. The generation of the masks, darkening the image and blending the mask with the darkened image are each $O(height * width)$ since they work on each pixel but they do not require looping over each channel since they apply directly to each pixel. Thus, the Gaussian blur dominates the complexity and we have $O(height * width * channels * n^2)$. This aligns with my observations on different sized images.

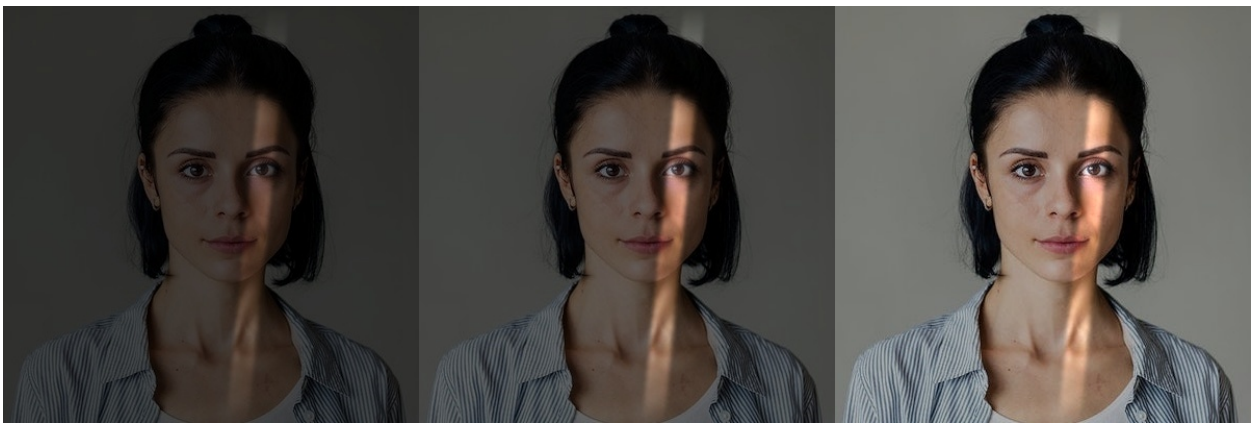

Figure 1: Simple Tests: (blending, darkening). 1: (0.7, 0.4), 2: (1.0, 0.4), 3: (1, 0.7)

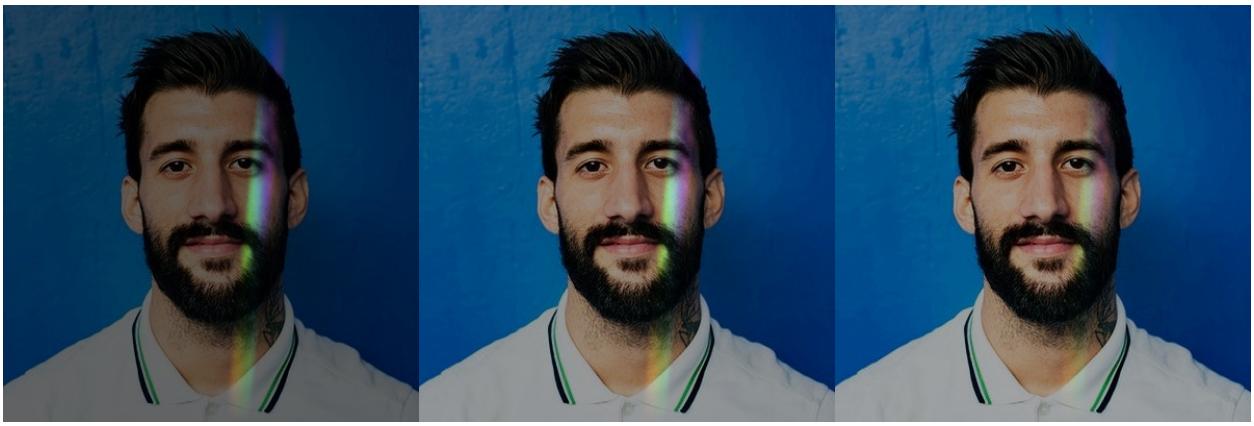

Figure 2: Rainbow Tests: (blending, darkening). 1: (1.0, 0.4), 2: (1.0, 0.6), 3: (0.8, 0.8)

2 Problem 2

For this problem I started by applying a Gaussian blur to the image. I did this so that I could extract the edges with a Laplacian filter which I then used to sharpen the edges of the image to create a pencil like effect on the edges of the image as they are typically more pronounced in drawings. Smoothing was essential because otherwise there was too much noise in the Laplacian output. Next, I applied motion blur to the sharpened version of the image to amplify the pencil edges. To create the charcoal shading effect I implemented histogram equalisation and finally I applied a bilateral filter to the result to smooth over the extremes caused by the histogram equalisation. This also amplified the shading effect by blending the edges of different shades. Finally, I blended this noise texture with the original sharpened image.

For the coloured pencil effect I used a similar idea but the motion blur applied was slightly different for each channel affected in order to create a small amount of distortion in the image so that the colours can be distinguished easier. Also, the Gaussian blur applied had different parameters for each channel so that the noise generated throughout the process would differ slightly to create a more natural drawing effect.

The grey-scale version has computational complexity of $O(height * width * n^2)$ (where n is the neighbourhood size again) since it is only being applied to a single channel. The histogram equalisation on a single channel is $O(height * width)$ whereas in the Gaussian blur and bilateral filter I loop over each pixel individually and within a neighbourhood which is $O(height * width * n^2)$. For the coloured pencil version I also have to loop over the channels in these operations so they become $O(height * width * channels * n^2)$ where channels in this case is usually 2 but using the CLI it is possible to specify more or less. So in general, this filter is $O(height * width * channels * n^2)$ which is what I observed with the times in my testing.

Figure 3: Simple: 0.0, 0.4 and 0.6 blending coefficients

Figure 4: Coloured: 0.6 blending coefficient with effect on channels [0, 1], [0, 2] and [1, 2]

3 Problem 3

To smooth the image I applied a bilateral filter to the image. I chose this because it is a high quality filter that respects the edges of the image whilst also providing a great blurring effect on the image. This was very useful when applied to a face because it prevented the edges of the face bleeding into the background (and vice versa) whilst also providing a face smoothing effect to remove imperfections like many of Instagram's (and similar companies) own filters do. By smoothing the face over it also reduced any gradients on the face which will be important for applying the colour curve and doing this helped to reduce the patchiness of the image once the colour curve was applied. I chose a neighbourhood size of 5x5 and $\sigma = 6$ for both spatial and intensity because in my testing this was enough to smooth out the imperfections of the skin without overdoing the blurring effect whilst also keeping the neighbourhood size small enough to allow for a reasonable run time.

For the colour curves I implemented a function that took in a set of points that defined the colour curve I wanted to make. It then constructed a polynomial passing through these points and evaluated it at any point that I specified. It did this using Lagrange Interpolating Polynomials. This was perfect for my colour curves because it allowed me to restrict the values to between 0 and 255 whilst also allowing me to 'drag' the curve around to wherever I wanted it.

I decided to use HSV colour space instead of BGR again because I did not want to substantially change the colour but rather enhance the shade of the colour by increasing its brightness and saturation to give a more 'tanned' effect like many social media filters do. To do this I increased the saturation and brightness more around the central values (near 128) whilst ensuring it smoothed off as it approached 0 and 255.

In terms of computational complexity, computing the lookup tables was $O(height * width)$ and lookup is $O(1)$. However, as I was using a bilateral filter again which takes $O(height * width * channels * n^2)$ (where n is the neighbourhood size) this term quickly dominated and thus the overall complexity of the filter was $O(height * width * channels * n^2)$. To improve efficiency I have made use of numpy's functions to manipulate the arrays as matrices which led to at least a 3x speed up in my initial testing.

Figure 5: Comparing no filter to different bilateral parameters: normal, (2, 6, 6), (5, 12, 12)

Figure 6: Comparing no filter to different bilateral parameters: normal, (2, 6, 6), (5, 12, 12)

4 Problem 4

4.1 Part a

To perform the swirl I convert each pixel's coordinates to polar coordinates with the centre as the central pixel of the image (img.width // 2, img.height // 2). Then I check if the radius, r, is less than (or equal) to the swirl radius, R. If it is not then the pixel is unaffected. Otherwise, I adjust the angle, θ, using the swirl angle, ϕ, according to the formula $\theta = \theta + \frac{\phi(1-r)}{R}$. This creates the swirl effect by rotating the pixels closer to the origin more. Then I convert back to Cartesian coordinates. However, the values won't necessarily be integer so I used interpolation to determine the pixel value. Nearest neighbour simply rounds the floats and takes the pixel corresponding to the rounded values. Bilinear interpolation averages a 3x3 neighbourhood around the rounded pixel's coordinates.

The complexity of the swirl algorithm is $O(height * width)$ for nearest neighbour interpolation and $O(height * width * n^2)$ for bilinear (where n is the neighbourhood size). This is because I have to check each pixel's coordinates so I have a nested loop. The bilinear then also uses a loop (abstracted away by numpy.average however) which computes the average around each pixel within the swirl.

Figure 7: $R = 170, \phi = \frac{\pi}{2}$ with bilinear and NN respectively. Final image is $R = 240$ with bilinear.

4.2 Part b

For the pre-filtering I have used a Butterworth low pass filter applied to the Fourier transform of the image. This creates a blurring effect by reducing the high frequencies in the image. I chose to use this because it created a smooth blurring effect on the image without a ringing effect occurring. This made it suitable because I wanted to eliminate artifacts instead of introduce extra ones before the swirl was applied. The complexity of this algorithm was $O(height * width * channels)$ as it needs to be applied to each channel separately. However, doing the convolution in Fourier transform was more efficient as it simply became a multiplication of the mask with the Fourier transform of the channel.

Figure 8: $R = 170, \phi = \frac{\pi}{2}$ with bilinear. Image 2 and 3 have a low pass filter ($n = 1$ in both and $K = 50$ and $K = 100$ respectively)

In the central image in the figure above, there is considerably less artifacts compared with the original. This is because by blurring the image before the swirl is applied we smooth the colours so that when interpolation is used the result is more consistent in the output. It also creates a motion blur effect when the image is swirled helping to enhance the whirlpool effect.

4.3 Part c

The inverse transformation of the swirl is simply to swirl it with the same angle and radius but in the opposite direction. This would undo the effects of the swirl. However, as a result of having to apply interpolation because of the non-integer values the image cannot be restored exactly (unless the floats were stored). Thus, this caused there to be differences between the transformed image and the original image.

Figure 9: Reverse with difference using bilinear interpolation

Figure 10: Reverse with difference using nearest neighbour interpolation

The result of the inverse with bilinear looks smoother than the inverse with nearest neighbour. This is because by looking at a neighbourhood of the pixel instead of just accepting the rounded coordinates the algorithm is essentially applying a mean filter to the pixel. This is why the difference resembles the structure of the image within the swirl radius in the bilinear image but not in the nearest neighbour image. There is significantly less artifacting in the bilinear inverse compared with the nearest neighbour inverse.

Outside of the swirl radius circle the difference image is black. This indicates that there was no change between the two images at these pixels. This is expected because we do not alter the pixels if their radius does not exceed the swirl radius.