Data Science – Image Processing Coursework Report – jsbl33

Problem 1

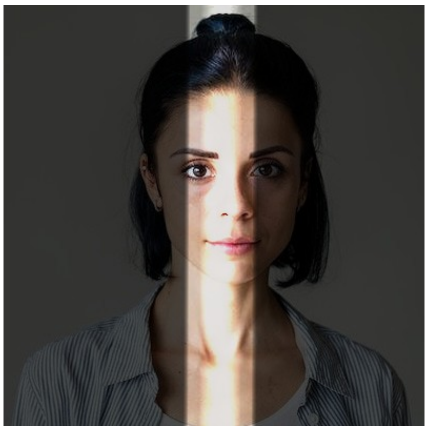


Figure 1: Problem 1 (light leak)

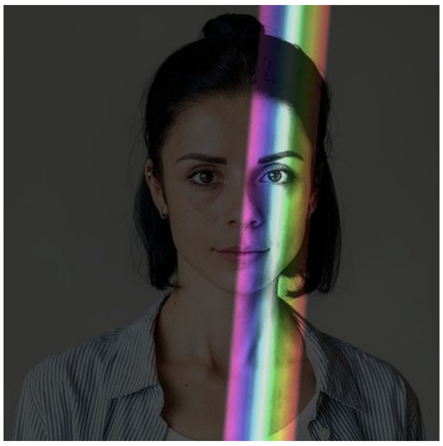


Figure 2: Problem 1 (rainbow)

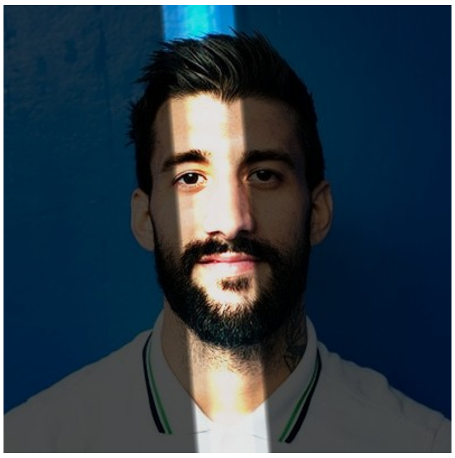


Figure 3: Problem 1 (light leak)

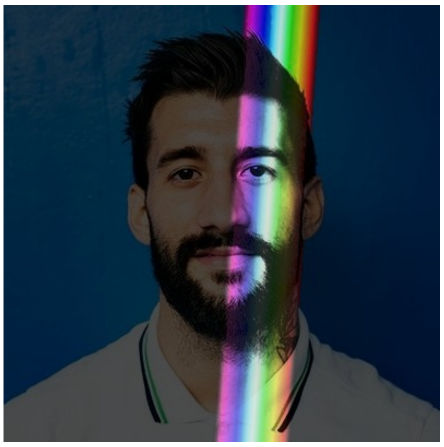


Figure 4: Problem 1 (rainbow)

My function for this problem is split up into two different sections – one for the light leak effect and one for the rainbow leak effect. In common to both is how the brightness of the image is changed – the whole image gets darkened by the coefficient specified and the area within the leak gets brightened by that same amount. The centre of the light leak has its brightness further increased, as I found this to more closely recreate the effect in the exemplar images. Conversely, the edge of the light leak has its brightness attenuate towards darkness to provide a smooth transition. The area of the leak itself is randomly generated each time to create some variety – the code simply picks two random points at the top and bottom of the image and calculates the line between them.

The rainbow light leak gets more complicated as a colour spectrum effect must be applied. I decided the easiest way to do this would be to convert each pixel to HSV, as this allows me to run through the colour spectrum by altering one channel only. The hue of each pixel is determined based on its distance to the centre of the light leak, with one side being 0 for red, and the other being 180 for purple. I also change the saturation of each pixel, making it at least 50 – this clipping prevents the rainbow from looking “washed out” in places. Value is changed in the same way as the normal light leak before the pixel is converted back into RGB and copied into the mask.

Irrespective of which mode was used, the created “light leak” mask is then blended with the input darkened image by addition (weighted by the blending coefficient).

In terms of computational complexity, this filter has a running time of Θ(mn) for an m x n image, as you would expect from something based on a point transform. Going through each step, darkening the image takes Θ(mn), as does creating the mask (since both require a constant amount of work per pixel). Image blending also takes Θ(mn), since each pixel requires a single addition to be made. Overall, this leads to a Θ(mn) running time.

Problem 2

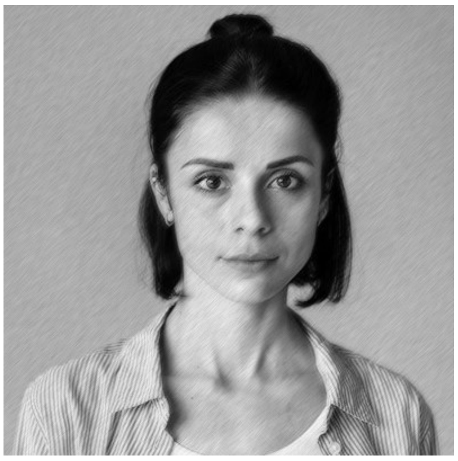


Figure 5: Problem 2 (b/w)

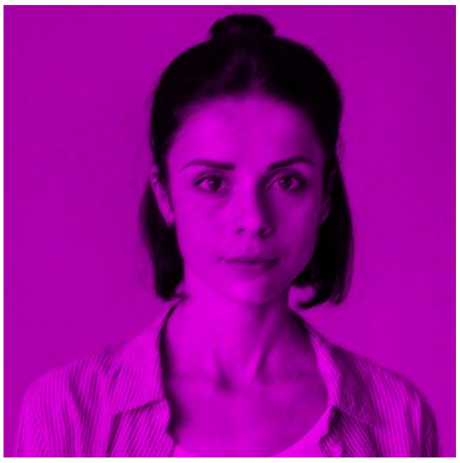


Figure 6: Problem 2 (colour)

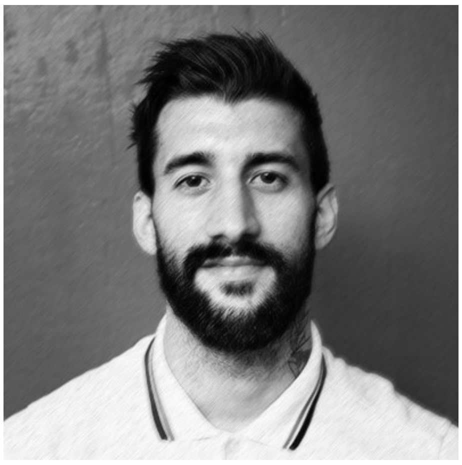


Figure 7: Problem 2 (b/w)

The core of this filter is my pencilFilter function, which first takes a greyscale image and applies Gaussian noise to it. I chose this for my custom noise texture as I felt those small random variations are more akin to pencil drawing than the stark maxima and minima of salt and pepper noise. To create a motion blur effect, I simply convolve the image with a mask that has a single line of non-zero values. After some experimentation I settled on a 7x7 matrix that applied a higher weight to pixels closer to the centre of the neighbourhood. The mask was originally convolved using my applyMask function, but after clarification in the FAQ I switched to cv2.filter2D for performance reasons. Once all of this is done, the original input and the generated image can be blended using a weighted sum to create the output.

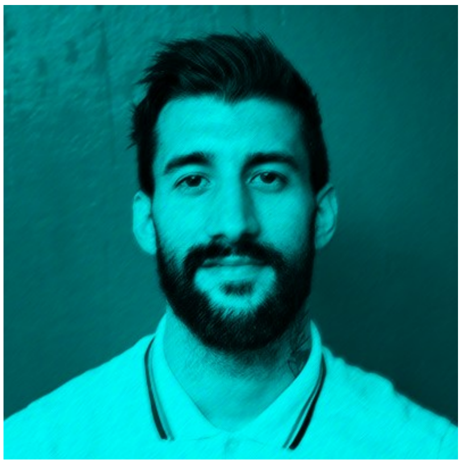
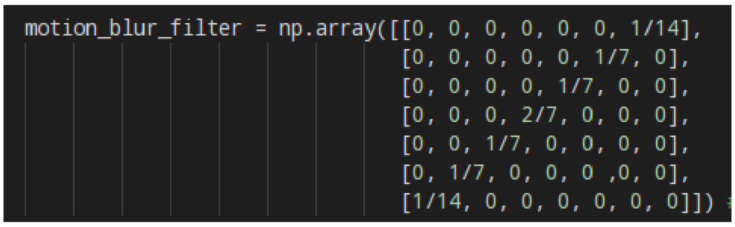


Figure 8: Problem 2 (colour)

To create the colour effect, I simply call the pencilFilter function twice to create the effect twice as instructed by the coursework. Since the Gaussian noise is random each time, this results in two different noise textures as required. Two random colour channels are then chosen, and each of the different greyscale images created by pencilFilter gets copied into one channel, producing the effect in either yellow, blue or pink depending on which channel is omitted.

To work out the complexity of the filter, we must go through each step again. Generating the noise requires Θ(mn) time for an m x n image. Assuming cv2.filter2D makes no optimisations, convolving an M x N mask with an m x n image takes Θ(mnMN), though in this case MN is a constant as the mask size is always 7x7. As such, this can be reduced to Θ(mn). Blending the images also requires Θ(mn) time. The time complexity is not affected by whether colour is used, since this only changes the time taken by a constant factor of 2. As a result, the overall complexity of this filter is Θ(mn), though it is still slower than my problem 1 as a result of the constant factors this hides.


```
motion_blur_filter = np.array([[0, 0, 0, 0, 0, 0, 1/14],
                               [0, 0, 0, 0, 0, 1/7, 0],
                               [0, 0, 0, 0, 1/7, 0, 0],
                               [0, 0, 0, 2/7, 0, 0, 0],
                               [0, 0, 1/7, 0, 0, 0, 0],
                               [0, 1/7, 0, 0, 0 ,0, 0],
                               [1/14, 0, 0, 0, 0, 0, 0]])
```
Figure 9: Motion blur mask

Problem 3

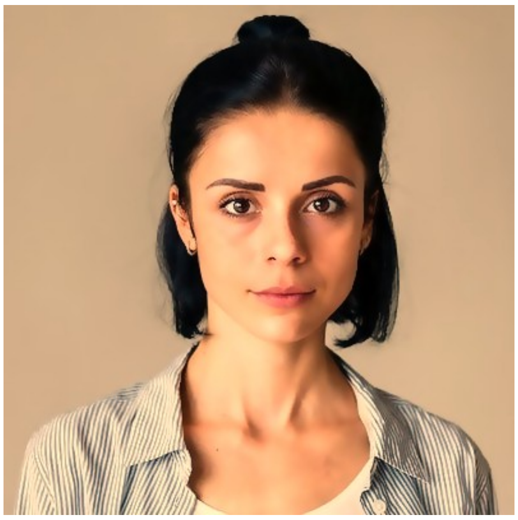


Figure 11: Problem 3 (warm colour balance)

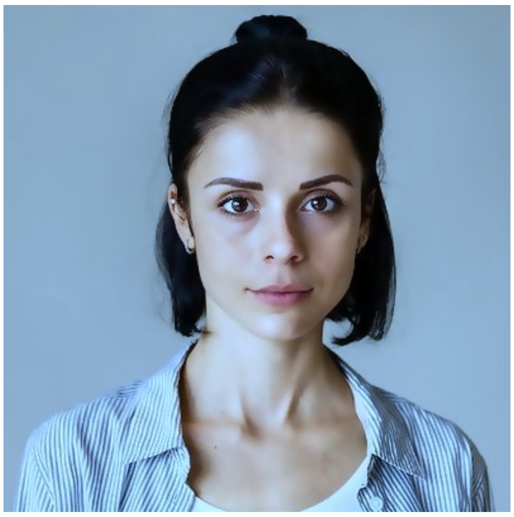


Figure 12: Problem 3 (cold colour balance)

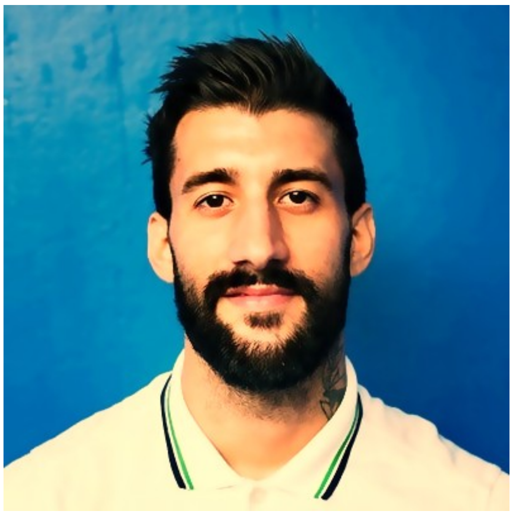


Figure 10: Problem 3 (warm colour balance)

To perform the initial smoothing out of the image, I perform bilateral filtering. Though computationally expensive (problem 3 is the slowest of my four functions), bilateral filtering allows me to apply a significant smoothing effect without losing too much definition in the features of the person or between them and the background (as happens slightly in the exemplar). Though any σ for space and colour can be used, the neighbourhood will always be a 7x7 region, as expanding the neighbourhood with σ results in an unbearably slow filter.

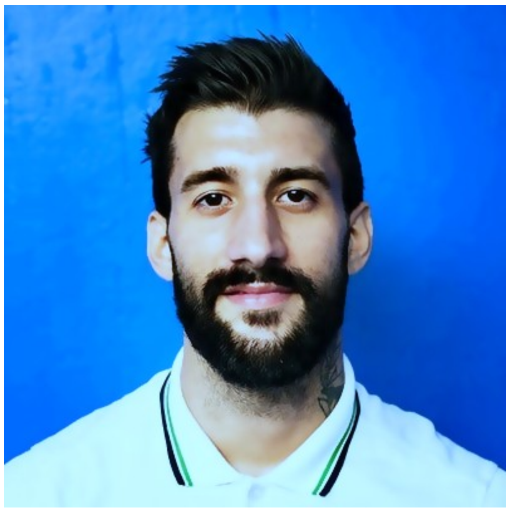


Figure 13: Problem 3 (cold colour balance)

Following this, a contrast stretch is performed on the image. This uses the naive method of taking the maximum and minimum values for c and d, since the bilateral filtering done will already have alleviated problems due to salt and pepper noise. This contrast stretch is done to improve the dynamic range of the image in an attempt to make it look better.

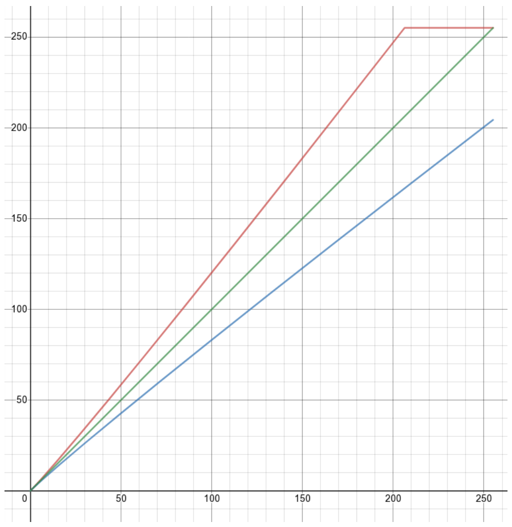


Figure 14: Colour curves for warming

The final step in the filter is to adjust the colour balance of the image. To do this, I create dictionaries in Python to map different red and blue colour values to new values. For warming images, red values are mapped to $I_{new} = max(I^{1.04}, 255)$ and blue values are mapped to $I_{new} = I^{0.96}$. I found using a slightly non-linear colour curve produced better results. To cool an image, the red and blue curves are swapped. This step produces the final image which can then be displayed.

In terms of complexity, the bilateral filter takes Θ(mnMN) time for an m x n image and an M x N neighbourhood. As with problem 2, my neighbourhood is of constant size so the complexity is Θ(mn). The contrast stretch needs O(n) time to find the maximum and minimum, followed by Θ(mn) operations to change the brightness of each image. Similarly, the colour balance adjustments are just a point transform, requiring Θ(mn) operations. This means that overall the filter is Θ(mn), though yet again this hides some large constants that make it significant slower than any of the other problems.

Problem 4

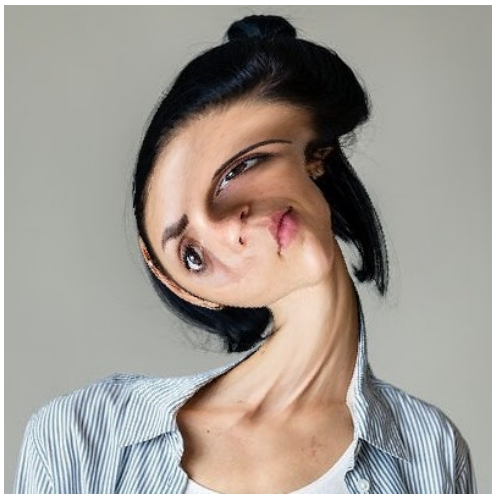


Figure 15: Problem 4 (nearest neighbour, no prefilter)

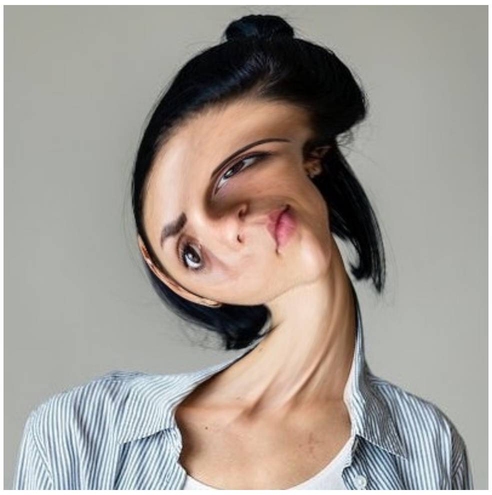


Figure 17: Problem 4 (bilinear interpolation, no prefilter)

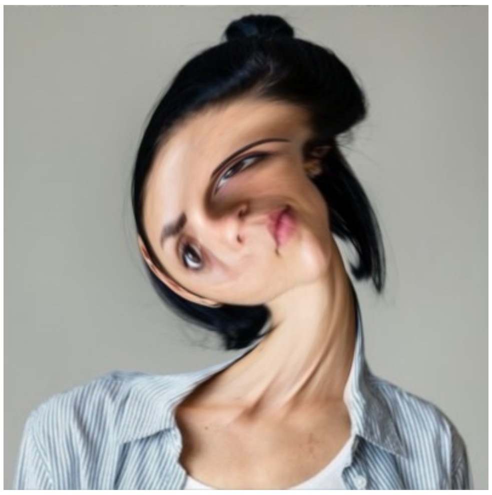


Figure 16: Problem 4 (bilinear interpolation, prefiltering)

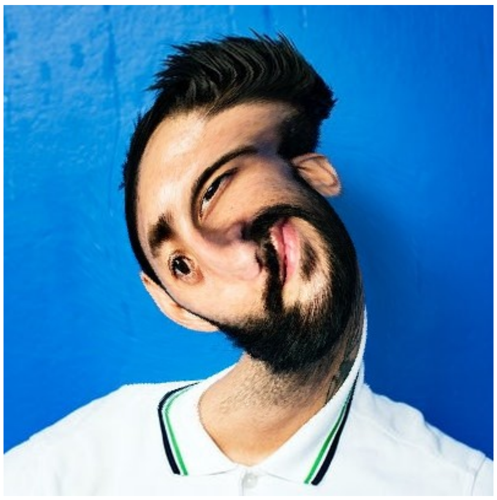


Figure 20: Problem 4 (nearest neighbour, no prefilter)

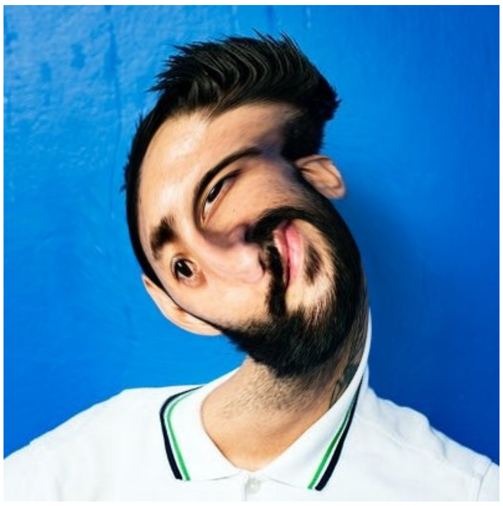


Figure 19: Problem 4 (bilinear interpolation, no prefilter)

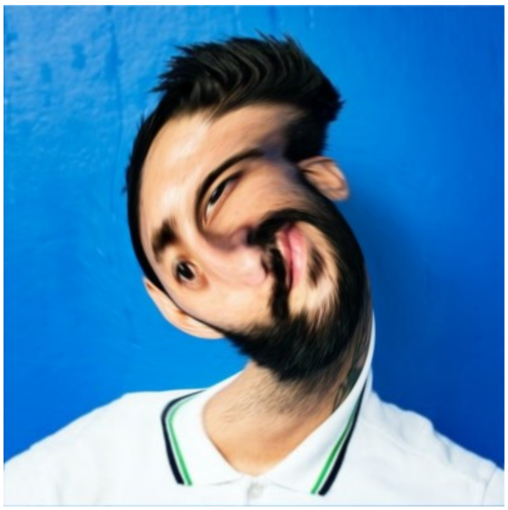
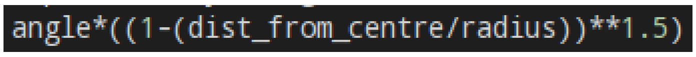


Figure 18: Problem 4 (bilinear interpolation, prefiltering)

The basic version of this filter works by creating a blank output image, and then iterating over every pixel in that image. For each pixel, it takes its coordinates, transforms them into polar form, increments the angle component by a set amount (based on distance from the centre) and transforms the result into Cartesian form. The actual increment is calculated from the angle parameter using this formula:


```
angle*((1-(dist_from_centre/radius))**1.5)
```

The power is used to keep a small angle at the edges that rapidly increases as you get towards the centre. This prevents a particularly harsh divide at the edge of the swirl (which was a problem for me early on in development). Once these new coordinates are calculated, they are rounded to the nearest integer to get the nearest neighbour and the value at that location copied over.

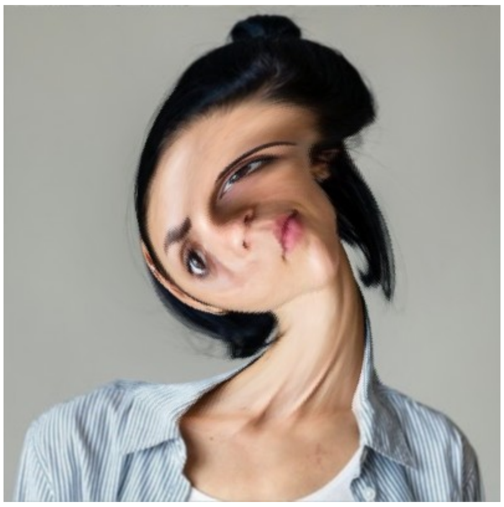


Figure 21: Problem 4 (nearest neighbour, prefiltering)

If bilinear interpolation is used, the principle is the same, but instead the

4 nearest pixels are used to calculate the new value. Though it's hard to see on this document as the images have been shrunk, this helps smooth out the edges that have been swirled in the output.

Similarly, prefiltering also reduces aliasing artefacts, which seem to occur when many pixels in the output map to a small number of pixels in the input. This is especially prominent at edges. In my prefiltering, I chose to implement a Gaussian low-pass filter to blur the source image. The difference is not hugely clear when using bilateral filtering, but can still be seen (for instance in the eyes on figures 16 and 17). When using nearest neighbour interpolation, the prefiltering has a much greater impact. Whilst the same aliasing problems occur, the prefiltering smooths out the edges in advance, so the aliasing artefacts are not as obvious.

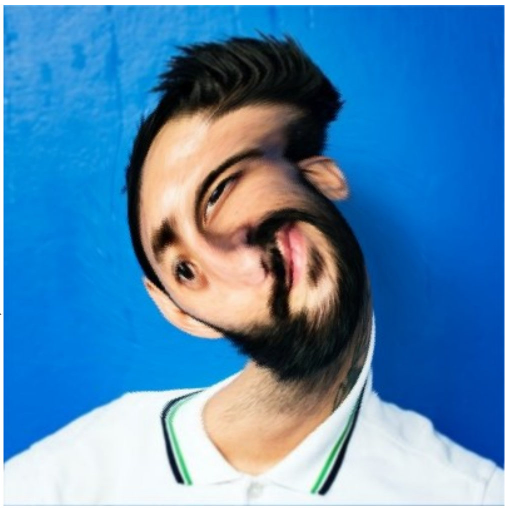


Figure 22: Problem 4 (nearest neighbour, prefiltering)

In terms of computational complexity, implementing the low-pass filtering requires use of a 2D FFT, which is O(mnlog$_2$n) for an m x n image. The filter itself requires some constant. amount of work be done on every value in the Fourier representation, giving a ϴ(mn) complexity. Transforming the image using IFFT similarily takes O(mnlog$_2$n). The swirl effect requires a fixed amount of work to be done for each pixel in the image, requiring ϴ(mn) time. This means that overall, the effect is O(mnlog$_2$n) with prefiltering and ϴ(mn) without.

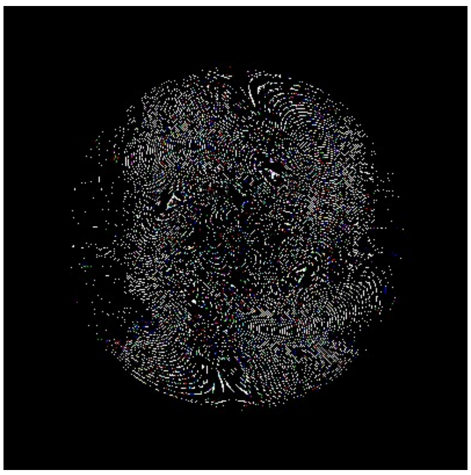


Figure 23: Task 4.3 with nearest neighbour

Applying the image warp and then immediately inverting it gives an image that looks almost identical to the original, though an image subtraction reveals significant differences between the original and the inverted transformation. Furthermore, the nature of the differences varies based on the type of interpolation used. With nearest neighbour interpolation, the differences do not appear to have any consistency and are presumably where rounding errors mean the inverse transformation does not map a pixel exactly back to where it originally was. With bilinear interpolation, the difference looks much more like edge detection, with the outline of the person clearly visible. Presumably this is because edges where neighbouring pixels have very different colours are where bilinear interpolation will result in the greatest difference from the original image.

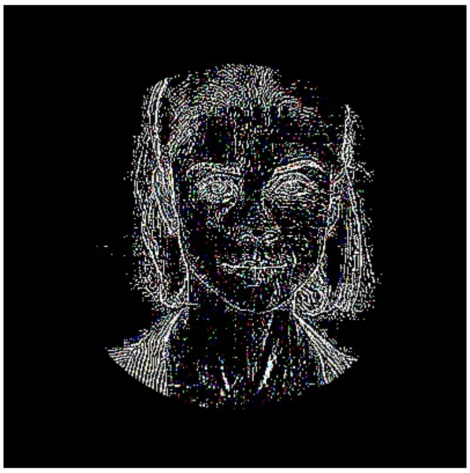


Figure 24: Task 4.3 with bilinear interpolation