Data Science: Image Processing Summative, Report

CIS Username: **cmcz82**

0 Introduction

This report will attempt to convey the philosophy and methodology behind the filters I implemented, and will hopefully serve to demonstrate my understanding and application of image processing theory. Principally, I shall discuss the process of constructing and testing each filter and the time complexity that my implemented algorithms sustain (in addition to any enhancements made with time complexity in mind). I will further contribute the reasoning that leads to me choosing the filters that I ultimately implemented and their respective parameters. Note that, when considering *time complexity*, any *linear time* **NumPy** function calls will be treated as *constant time* processes due to their nature as being compiled in faster languages, generally C, and therefore executing in speeds that are orders of magnitude faster than an interpreted Python equivalent.

1 Light Leak and Rainbow Light Leak

Both my light-leak and rainbow light-leak masks were purely hard-coded. My basic *light-leak mask* is applied to the input image by iterating through the columns of said image and *darkening* the pixels that don't fall within the range of the light-leak, and when the light-leak range is reached, the pixels are brightened by an increasing amount (the amount here is based on the adjustable blending coefficient, which determines how "quickly" the maximum brightness is reached after brightening begins) until it reaches the maximum brightness. Then the opposite occurs, as brightness decreases and pixels are then darkened. The brightening and darkening are achieved via a helper-function I developed which changes the brightness of the image based on adjustable inputs (see the **adjust_brightness()** function in my code-base). The decision to hard-code the effect rather than to rely on an external mask was a conscious one: it allows for more simple inclusion of customisability in terms of the shape and intensity of the light. Figures 1 through 4 below demonstrate the effect with varying values for the blending, darkening, and brightening adjustable coefficients (Note that for the darkening coefficient, higher values denote less darkening and vice versa).

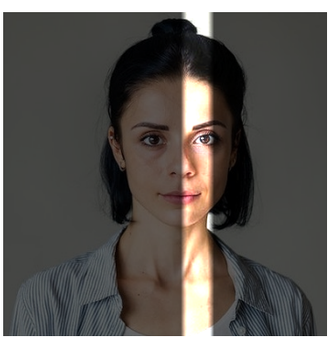

Figure 1: Darkness of 0.3, blend of 0.2, brightness of 0.8.

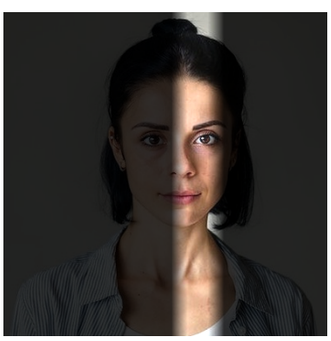

Figure 2: Darkness of 0.1, blend of 0.5, brightness of 0.2.

Figure 3: Darkness of 0.5, blend of 0.5, brightness of 0.6.

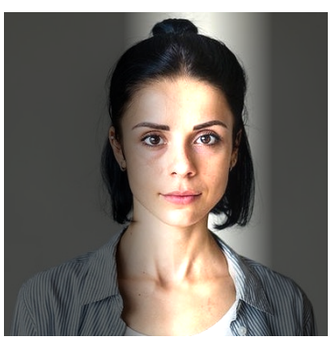

Figure 4: Darkness of 0.3, blend of 0.8, brightness of 0.6.

For the rainbow effect, my program applies the rainbow to the image after it first applies the basic light-leak effect. The rainbow is constructed within a function (the apply_rainbow_mask() helper function) that creates a rainbow (of size dependent on the blending coefficient) based on the RGB colour levels of each colour of the rainbow, with appropriate fading between them (i.e., red fades to orange, which fades to yellow, and so on). This rainbow then overlays the original image via the fade_and_blend() helper function - this suitably applies the rainbow over the light-leak area of the input image with appropriate fading at the edges of the rainbow. Figures 5 through 8 demonstrate the rainbow effect with varying blending, darkening, and brightening coefficients. (Note, the code also has an adjustable coefficient for the *strength* of the rainbow, which adjusts the relative influence the rainbow has when overlayed over the input image).

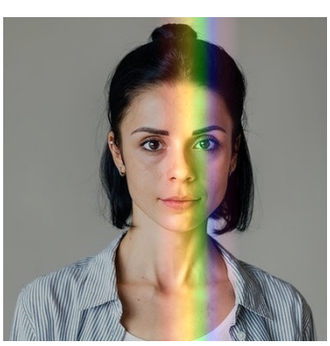

Figure 5: Brightness of 0.3, blend of 0.5, darkness of 0.7.

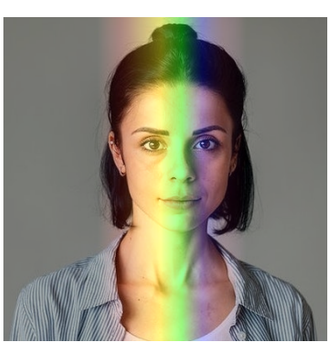

Figure 6: Brightness of 0.3, blend of 0.7, darkness of 0.6.

Figure 7: Brightness of 0.5, blend of 0.9, darkness of 0.3.

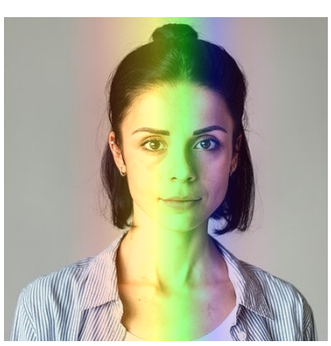

Figure 8: Brightness of 0.5, blend of 0.7, darkness of 1.

Time Complexity

Originally, to apply the light-leak mask, the function was required to iterate through every pixel in the image and apply the brightness function to each. I enhanced this by essentially darkening all of the areas of the image distinct from the light-leak *in bulk* (which can be considered to be a linear-time operation when taking into account the speed of dealing with NumPy arrays like this). Furthermore, instead of applying the brightness function pixel-by-pixel, it is applied column by column, so the function only needs to iterate through the *columns* of the image where the light-leak is applied. Similarly, to construct the rainbow mask, the function must iterate through the columns of a new NumPy array that is shape-wise identical to the input image. To apply and blend the rainbow, the function must iterate through the columns of the image where the rainbow

is blended, and the rest of the overlaying is done in bulk. Ultimately, the time complexity evaluates to **O(C)**, where C is the number of *columns* in the input image.

2 Pencil Sketch

Figure 9: Pencil effect applied with a canvas blending of 0.

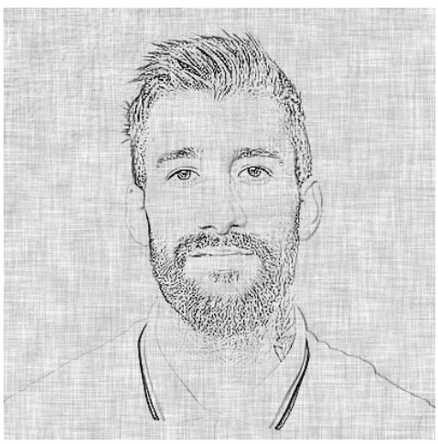

Figure 10: Canvas blending of 0.3, stroke rate of 0.4 and stroke width of 0.23

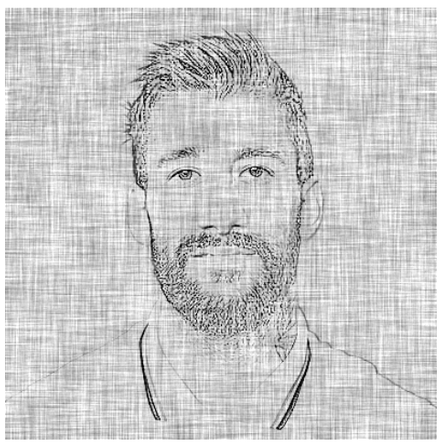

Figure 11: Canvas blending of 0.7, stroke rate of 0.4 and stroke width of 0.3

To generate the *canvas-mask*, I experimented with both salt and pepper and Gaussian noise. Initially, the Gaussian noise mask looked more promising, but after applying motion blur I found the differences between the two to be considerably minor, although nuanced. I ultimately decided to utilise both, applying them separately to different channels for the coloured effects. By default, however, my program only uses the Gaussian noise mask for the monochrome effect (although salt and pepper noise can be triggered via an adjustable parameter). With regards to motion blur, I experimented with both horizontal and vertical motion blurs. I ultimately settled on a mixture of the two, as this best matched the aesthetic of a *canvas*. By default, the kernel size for my motion blur effect is 23×23 pixels, but I allowed for this to be adjusted as an optional parameter (i.e., for the "canvas stroke-width").
Furthermore, I felt that applying the canvas to a monochrome image alone didn't really provide much of a pencil-sketch effect, which was what I wanted. Rather, it simply looked like an old black and white image. Therefore, I was encouraged to experiment with additional techniques. I settled on a method that meant creating a *blurred* version of the original greyscale image (the blurring was done via a mean blur), and then by applying image division by dividing the non-blurred greyscale image by the blurred variant. This had a strong *detail-removing* effect, which leads to a significantly more pencil-like effect, in combination with the canvas. Figures 9 through 11 above demonstrate the pencil effect applied with varying parameters for canvas blending, canvas stroke rate (i.e., the probability of noise occurring during the noise-construction phase), and canvas stroke width.

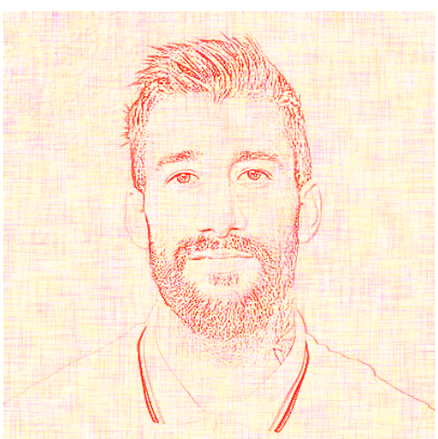

Figure 12: Canvas blend of 0.4, stroke rate of 0.3, with the "red pencil" effect.

Figure 13: Canvas blend of 0, stroke rate of 0.5, with the "green pencil" effect.

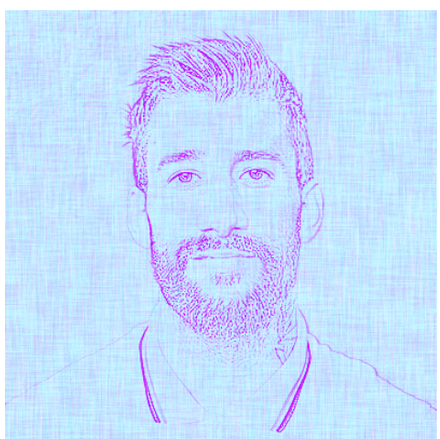

Figure 14: Canvas blend of 0.3, stroke rate of 0.5, with the "blue/purple" effect.

Now, to obtain the colour-pencil effect, I followed the approach outlined in the coursework document. This entailed splitting the image into its three BGR channels and applying a *distinct* noise texture to two of the three channels. Figures 12 through 14 present examples of the coloured-pencil effects, and the following is an outline of how each of the colour-pencil effects I implemented was achieved.

- **Blue/purple**: Gaussian noise with horizontal and verticle motion blur is applied to the green channel, and salt and pepper noise, with only vertical motion blur, is applied to the red channel.
- **Purple**: Same as above, except the effects applied to the green and red channel are switched. This lessened the amount of blue visible, leading to a more distinct purple effect.
- **Red**: Salt and pepper noise is applied to the green channel, and gaussian noise is applied to the red channel. A vertical and horizontal motion blur is applied to both noise masks.
- **Green**: Gaussian noise, with horizontal motion blur, is applied to the blue channel, and salt and pepper noise, with horizontal and vertical motion blur, is applied to the red channel.

Time Complexity

The function for this problem executes and halts with haste. The vast majority of operations within the collection of helper functions utilise NumPy array operations, which can be considered to take constant time. The only iteration required is when producing the salt and pepper mask, which will iterate through each pixel in the image and perform a constant time operation

on them. Therefore, the time complexity of the function will evaluate to **O(N)**, where N is the total number of pixels in the input image.

3 Smoothing and Beautifying

For this problem, I decided to implement a **Bilateral Smoothing** filter as I felt this best corresponded with the concept of *beautifying*. The bilateral filter elegantly smooths the image whilst, unlike many alternative *blurring* filters, preserving the shapes and edges of the original image. My program, by default, uses a relatively small window size of 5x5 for the filter. The reasoning behind this decision was that I felt a more subtle filter, that smooths and removes certain "blemishes", was more suitable than a more drastic bilateral filter, that results in a more *cartoon-ish* effect. Furthermore, my values for the spatial and range gaussian sigma values, 9 and 25 respectively, were chosen with this important fact in mind. I allowed for the window size to be an adjustable parameter, as increasing this generally increases the *quantity* of smoothing.

Figure 15: Original image, with no bilateral smoothing applied.

Figure 16: Bilateral smoothing with a 3x3 window.

Figure 17: Bilateral smoothing with a 7x7 window.

Moreover, with regards to the *Instagram-style* colour grading filter that I implemented, to demonstrate a deeper understanding of the concept of colour grading, I decided to hard-code my own *look-up table generator*. For the specific workings of it, inspect the **create_look_up_table()** helper-function in my code. Generally, it takes as an input a *list of mappings* in the form of tuples containing the input pixel values and their corresponding output values: the function uses this information to create a dictionary that maps every value between 0 and 255 to an output based on the shape of the curve implied by the input. The LUT-dictionaries created can simply be applied by scanning some image and mapping the pixel value of each channel to their corresponding output. The colour-grading filter I implemented has three variations. The standard variant, entitled "Mid-Autumn", a *high-contrast* variant, entitled "Late-Summer", and a *softer* and *warmer* variant, entitled "Late-Spring". Note the amount of contrast applied to the "Late-Summer" variant is an additional adjustable parameter.

- The standard **"Mid-Autumn"** filter provides a slightly low-contrast and high-exposure effect, with specific shifts to each colour channel that provides a more mellow palette than the original image. The *curve* constructed for each channel entails a general lightening of the darker pixels (which occurs most strongly for the blue channel, and least strongly for the green channel), as well as a general darkening of the lightest pixels.
- Conversely, the **"Late-Summer"** variant applies the exact same colour grading filter as above, except it then also applies a variable contrast-increasing effect. To produce this effect, an *S-shaped* curve is created for each channel, with the darker values being darkened and the lighter values being lightened. This darkening/lightening is variable: by adjusting the contrast_intensity parameter for the function, you can input a value between 0 and 64 which will correspond to how much the 64th-pixel value is decreased by and how much the 191st-pixel value is increased by.
- Finally, the **"Late-Spring"** variant applies a gentle *warming* and *saturation-reducing* effect to the image by applying a set of curves to each channel *prior* to the standard curves. The result is that the brighter pixels approach a more *yellow* colour and the darkest values are lighter. It works by moderately lightening all of the red pixels, and gently lightening the pixels of the other channels, in addition to significantly darkening the lightest blue pixels, and slightly darkening the lightest green pixels.

To contrast the effect of the filter variants, images with each variant applied are shown in Figures 18 through 20, along with their corresponding colour channel histograms (Figures 21 through 24). Note that each image had a bilateral smoothing with coefficient 1 applied prior to the colour grading.

Figure 18: Late-Spring filter.

Figure 19: Late-Summer filter.

Figure 20: Mid-Autumn filter.

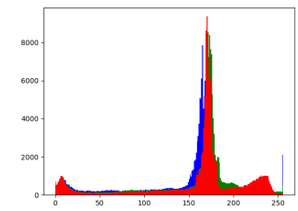

Figure 21: Input image histogram.

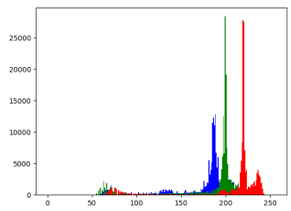

Figure 22: Histogram following Late-Spring filter.

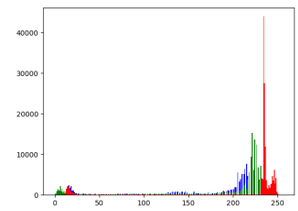

Figure 23: Histogram following Late-Summer filter.

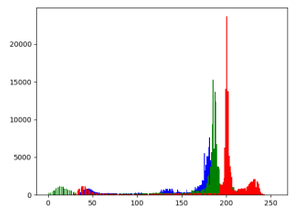

Figure 24: Histogram following Mid-Autumn filter.

Time Complexity

Out of all my functions, this one takes longest to execute. Primarily, this is due to the complexity of the bilateral smoothing. The process contained in the **apply_padding()** helper function can be assumed to take **O(1)** as it primarily consists of linear time NumPy functions. The bulk of the bilateral filter takes **O(W· N)** time, where W is the number of pixels in the w× w window of the filter (i.e., w·w=W) and N is the number of pixels in the input image. Further, my **create_look_up_table()** helper-function takes **Ω(256)** time, which evaluates to constant time. And the function for applying a LUT takes **O(N)** time as it applies a constant time operation to each co-ordinate in the image. Overall, the time complexity evaluates to **O(W·N)**.

4 Face swirl

Figure 25: Radius = 50, strength = -2.

Figure 26: Radius = 80, strength = -2.5.

Figure 27: Radius = 200, strength = -3.

Figure 28: Radius = 120, strength = 3.

Figure 29: Radius = 240, strength = 4.

My face-swirl filter works by iterating through the pixel coordinates of the input image and applying a *reverse-mapping*. That is to say, it calculates the pixel value that will *land* on the current co-ordinate as it works its way through the image. The mapping itself is dependent on the idea of polar coordinates. The process, at each co-ordinate, calculates a *theta* value that is dependent on the *strength* and *radius* of the swirl, and most crucially the *absolute distance* of the pixel coordinates from the centre of the image (or, from the co-ordinates that the user has defined as the centre: see the execution commands .txt for more on this adjustable parameter). In particular, no pixels that lie outside the *padded radius* of the swirl (more on the idea of padding in a moment) are transformed at all, and within the "padded" segment of the radius, the strength of the swirl gradually increases from 0 to the value specified by the user (or the default value). Without this functionality in place, *every* pixel would get transformed, even if very slightly. Figures 24 through 28 above are examples of the swirl applied to the same image with varying values set for the radius and strength. (Note, all images use bilinear interpolation.)

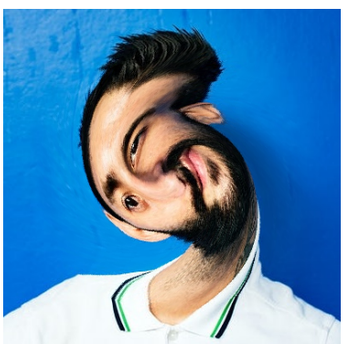

Figure 30: Nearest neighbour interpolation.

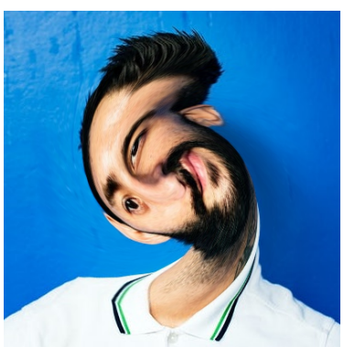

Figure 31: Bilinear interpolation.

Futhermore, when the mapping has taken place, interpolation is required to handle the (almost certain) occurances of mappings from co-ordinates that do not have integer values. The two interpolation strategies I implemented were **nearest-neighbour** and then **bilinear** interpolation. The nearest neighbour interpolation simply chooses the nearest viable pixel. It is the simplest to implement, however it results in very noticeable aliasing artifacts. Conversely, bilinear interpolation is more complicated (it takes a weighted average of the closest four co-ordinates), but it produces significantly better results with almost no noticeable aliasing artifacts in the transformed image. Figures 30 and 31 demonstrate a comparison between nearest neighbour and bilinear interpolation for a swirl radius of 160 and swirl strength of -3. (Note that the negative value here simply indicates an *anti-clockwise* swirl, with strength of 3.)

When it came to implementing the low-pass pre-filter, I quickly realised that a simple *ideal* low-pass filter was insufficient for the desired purpose, so I implemented a slightly more complicated ***Butterworth Low-pass filter*** instead. This entailed having the filter *blend* from black to white rather than having a solid white circle. My algorithm for this was derived from page 273 of Gonzalez&Woods. I also decided I didn't want the blurring effect to be too drastic, which appears to diminish the quality of the image, rather I simply wanted a reduction in aliasing artifacts in the transformed image. Therefore, I settled on a cut-off frequency of 60 (or, more generally, a three-twentieths of the number of rows in the image), and by default the filter has an order of 3. I found that the low-pass filter wasn't really necessary when bilinear interpolation was used, as this already minimised aliasing artifacts quite significantly and by applying the low-pass filter, the only real difference is blurring. However, the low-pass filter certainly had a positive impact in reducing aliasing artifacts when nearest-neighbour interpolation was used instead. A demonstration of the filter, presented in figure 32, applied to face swirls with both nearest neighbour and bilinear interpolation is shown in Figures 33 and 34, with a swirl strength of -2.5 and the a radius of 160.

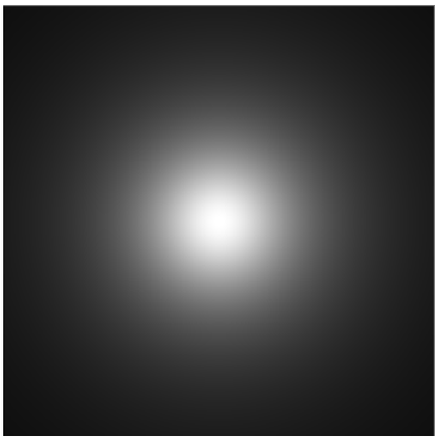

Figure 32: The Butterworth low-pass filter with cut-off frequency of 70 and order of 1.

Figure 33: The Butterworth low-pass filter applied prior to an image swirled with nearest neighbour interpolation.

Figure 34: The Butterworth low-pass filter applied prior to an image swirled with bilinear interpolation.

To illustrate the relative success of the filter when different interpolations or pre-filtering techniques are used, the swirl can be reversed to allow for image subtraction between the original input image and the reverse-swirl. Image subtraction is a useful technique for examining the *difference* between two images and is generally more efficient than image division. Figures 35 through 38 present the results of image subtraction between the original image and reversals of the swirl filter when nearest-neighbour interpolation, nearest-neighbour interpolation with an intial Butterworth low-pass pre-filter, bilinear-interpolation, and bilinear-interpolation with an intial Butterworth low-pass pre-filter are used respectively. By the way, notice that only the points *within* the radius are transformed: this slightly reduces the runtime when the swirl area is smaller than the image and, in my opinion, increases the elegance of the outcome.

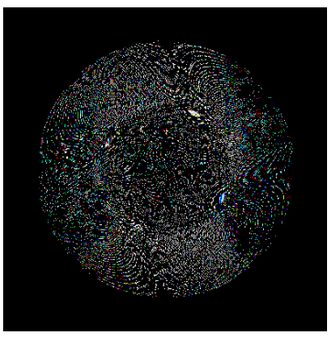

Figure 35: Image subtraction between the original image and a reversal of a swirl with nearest neighbour interpolation applied during the mapping.

Figure 36: Image subtraction between the original image and a reversal of a swirl with nearest neighbour interpolation and pre-filtering.

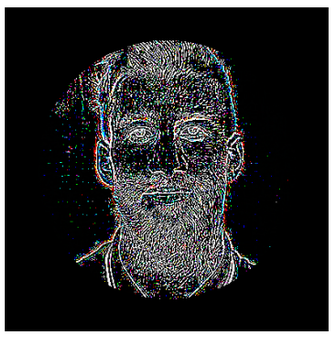

Figure 37: Image subtraction between the original image and a reversal of a swirl with bilinear interpolation applied during the mapping.

Figure 38: Image subtraction between the original image and a reversal of a swirl that used nearest bilinear and pre-filtering.

These figures strongly suggest that a more sophisticated interpolation technique is significantly more powerful when it comes to reducing aliasing than a sophisticated low-pass prefilter. In particular, when comparing figure 35 with figure 37, the distribution of noise in the latter is significantly more concentrated around edges and areas of detail (such as the hair and facial features), whereas there is a more even distribution of noise in the former. This implies that the reversal of the swirl when bilinear interpolation is produced results in an image that appears to be an only slightly blurred version of the original image, whereas with nearest neighbour interpolation the result is more different, and sharded, than the input image. Furthermore, when observing the results when pre-filtering is applied, there is a clear improvement with regards to the similarity of the reversed swirl with the original image when pre-filtering and nearest neighbour interpolation is used, however, little difference is noticeable between the pre-filtering or not when bilinear interpolation was used - in fact, the quantity of noise is slightly reduced when pre-filtering is *not* used, suggesting the pre-filtering only introduced a larger difference. Of course, when pre-filtering is utilised, there is noise outside of the radius of the swirl because the entire image is blurred first.

Time Complexity

For the standard swirl effect, the only pixels that are affected are those within the swirl circle, which encapsulates $\pi \cdot r^2$ pixels, where r is the swirl *radius*. On those pixels, a series of constant-time operations occur, and both nearest-neighbour and bilinear interpolation strategies take constant time. Therefore, the time complexity of the swirl is $\mathbf{O}(r^2)$, which is less than $\mathbf{O(N)}$ (where N is the number of pixels in the input image). When considering the filter in its entirety, the Butterworth low-pass pre-filter must also be considered. To produce this filter, every pixel of an image the size of the input image must be iterated through, with a constant time operation applied to each. Overall, the Butterworth low-pass filter takes $\mathbf{O(N)}$ time. Since $N > r^2$, when the prefiltering is applied the overall time complexity of the process is $\mathbf{O(N)}$.