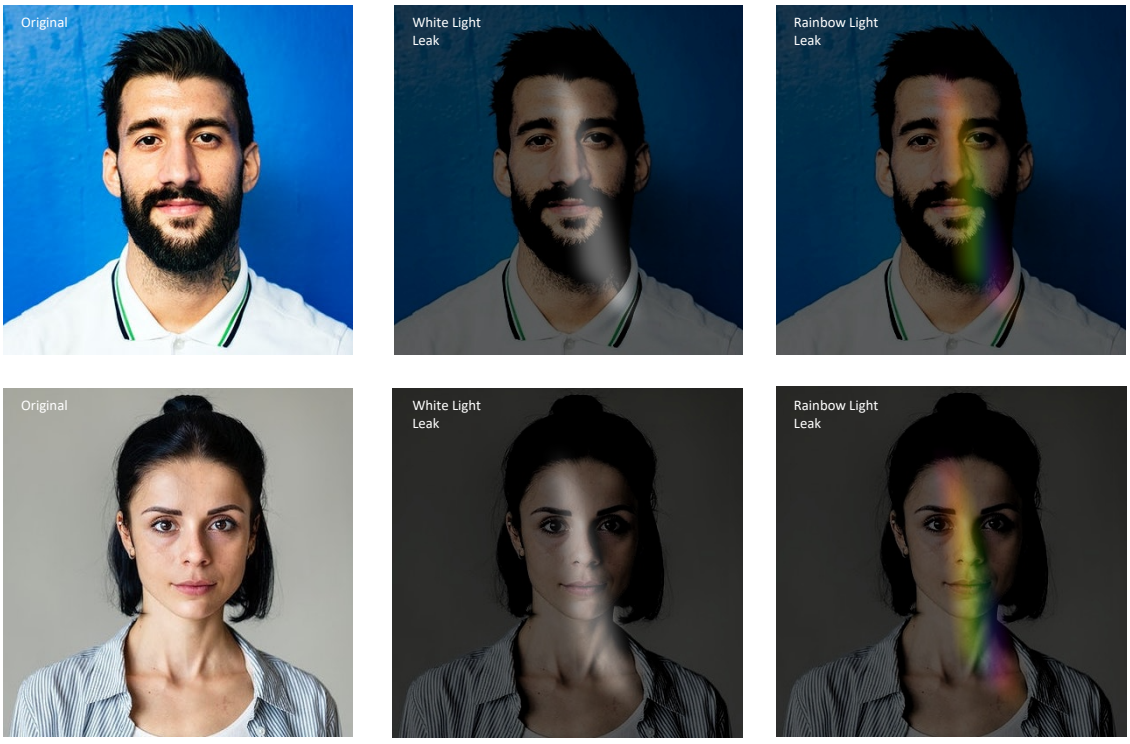
Problem 1

Description

The function first darkens the image by multiplying all pixel values by *1 – darkening coefficient.* To create the mask, the function creates a blank array of zeroes the same shape as the input image and then draws a polygon and fills it with the value [255,255,255]. Gaussian blur is then applied to smooth out the mask to create the effect of light intensity decreasing as it dissipates from its source. For the rainbow effect, I took this mask and the applied a rainbow colour map to it.
The mask and darkened image are then combined using a custom made alpha-blending function which uses the value of the blending coefficient to determine the weights of the of the darkened image and the mask in the filtered image.

Images



Time Complexity

- **Darkening Image** -> *O(3 x M x N)* as 3 pixel values need to be multiplied for each pixel in the image. The number of pixels in the image is determined by the width(M) and height(N) of the image.
- **Mask creation** -> It is hard to determine the time complexity of this section as it required the use of a few OpenCV functions, of which I do not know the time complexity. However, I would assume linear time (*O(M x N)*).
- **Blending** -> *O(3 x (3 x M x N))* as the blending process is split into three operations each taking time *O(3 x M x N)*; multiplying the input image by the *blending coefficient,* multiplying the mask by *1 – blending coefficient* and the adding the two resultant matrices together.
- **Total** -> *O(M x N)*

Problem 2

Description

The charcoal effect mask is created by creating a blank image the same shape as the input image. Random these pixels are then randomly allocated black or white values to create a salt and pepper effect across the whole mask. Motion blur is then applied to give the charcoal effect noise texture. This texture is then blended with the grey scale version of the original image in the same way as the alpha blending in problem 1. I also decided to apply OpenCV's built-in Canny edge detection on the original image and copy the edges onto the filtered image to make it appear more like a sketch.
For the coloured pencil effect, the approach is the same but two separate charcoal effect masks are made and applied on the blue and red colour channels to create a purple pencil effect.

Images

Time Complexity

- **Generating noise texture** -> *O((M x N) + (M x N x 21 x 21))* as the function first must go through and randomly allocate a value of 0 and 255 to each of the M x N pixels in the image. Then the motion blur kernel of size 21 x 21 will need to be applied to each pixel.
- **Drawing edges** -> *O(M x N x log(M x N) + (M x N))*. From research, I have found that the Canny edge detection used to get the outline of the image has time complexity of *O(M x N x log(M x N)).* To then draw these lines onto the image, requires to iterate through each pixel and set the pixel's value to 0 (or [0,0,0] in coloured image) if it is on an edge, which takes *O(M x N)*.
- **Blending** -> blending the image with noise texture takes *O(3 x (M x N))* in a grey image and O(2 x (3 x(M x N))) on a colour image (as the textures are applied to two channels).
- **Total** -> *O(M x N x log(M x N))*

Problem 3

Description

This filter works by first performing median blur to smooth out the image, using a nxn kernel where n is specified as a parameter called *blur_region.* I chose to implement this blurring technique as it retains edges and is minimally impacted by pixels which vary drastically from others within the neighbourhood (as opposed to mean filters) meaning that it is effective at 'smoothing' the image. Furthermore, as it only needs a small kernel (3x3 or 5x5) to work effectively, it could run relatively quickly.

To apply colour grading, I converted the image into the LAB colour space and applied a decreasing lookup table on the L channel and increasing lookup tables on the A and B channels. I thought this gave a much more interesting colour effect than simply increasing the values in the red channel of a BGR image to merely make the image appear 'warmer' or decreasing the values on the blue channel to make it 'cooler'.

Images




Time Complexity

- **Median blur** -> *O(3 x (3 x 3) x M x N)* as in my implementation of this blur function, for each of the *M x N* pixels image, the median value from the 3x3 neighbourhood needs to be calculated for each of the 3 BGR channels.
- **Creating LUT** -> *O(3).* Creating the LUT happens in constant time regardless of the size of the input image as all channels range from 0-255 (which is the size of the LUT created). A table needs to be made for each of the 3 channels.
- **Applying LUT** -> *O(3 x M x N)* as the function needs to iterate through each pixel on the image and map the pixel values to the relevant value in the LUT. This process needs to be carried out on all 3 BGR channels.
- **Total** -> *O(M x N)*

Problem 4

Part 1

Description

For part one, I implemented my code such that it first normalises the coordinates so that they range between -1 and 1 instead of 0 - width (or height) of image. This means that the process can be scaled up and that any transformations that are made occur around the centre of the image. The coordinates are then converted into their polar form and the value of theta is adjusted in accordance with its distance from the edge of the swirl radius (if the coordinates are outside this radius it will remain unchanged). The new coordinate values are converted back to being between 0 - width (or height) of image and are interpolated to fit an exact pixel value. These values are then reverse mapped onto the new image.

Images

Comments

Although the two transformations look remarkably similar, I personally think the implementation using bilinear interpolation gives the best result as the image appears to be a little bit smoother due to the fact that pixel values are determined by taking a distance weighted average of values of surrounding pixels as opposed to merely rounding to the nearest pixel value such as with nearest neighbour.

Time complexity

- **Swirling** -> *O(M x N)* as for each pixel a simple series of constant time operations is required to calculate its polar coordinates, transform the pixel and transfer it back to cartesian.
- **Interpolation** -> *O(1)* as in both methods of interpolation only require straight forward constant time operations. In nearest neighbour this simply consists of rounding to the nearest pixel. In bilinear interpolation, by calculating the weights of surrounding pixels and combining their values according to their weights.
- **Total** -> *O(M x N)*

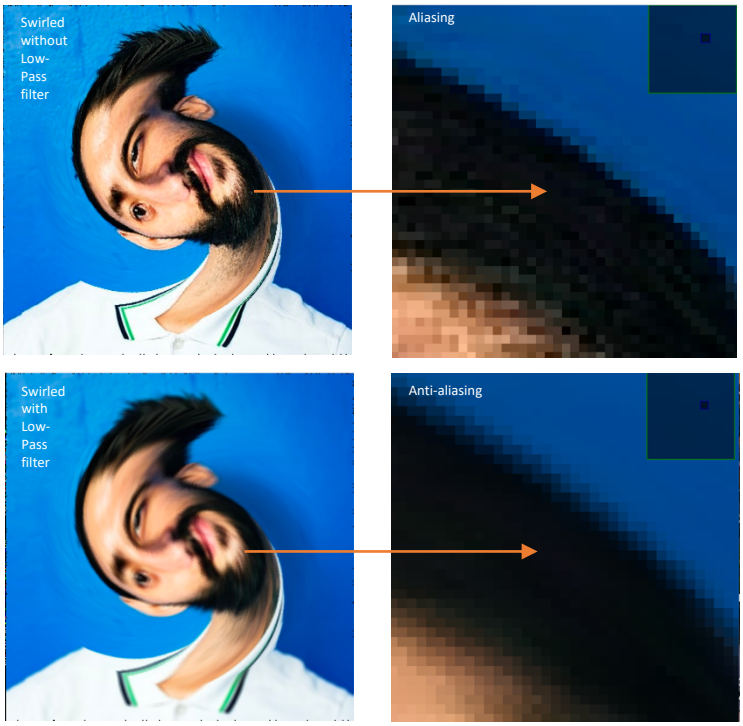
Part 2

Description

The low pass filter in this instance is a simple convolution using a 5x5 kernel where each pixel in the neighbourhood is weighted $1/5^2$. Applying this convolution has clear anti-aliasing effects as evident in the images below.

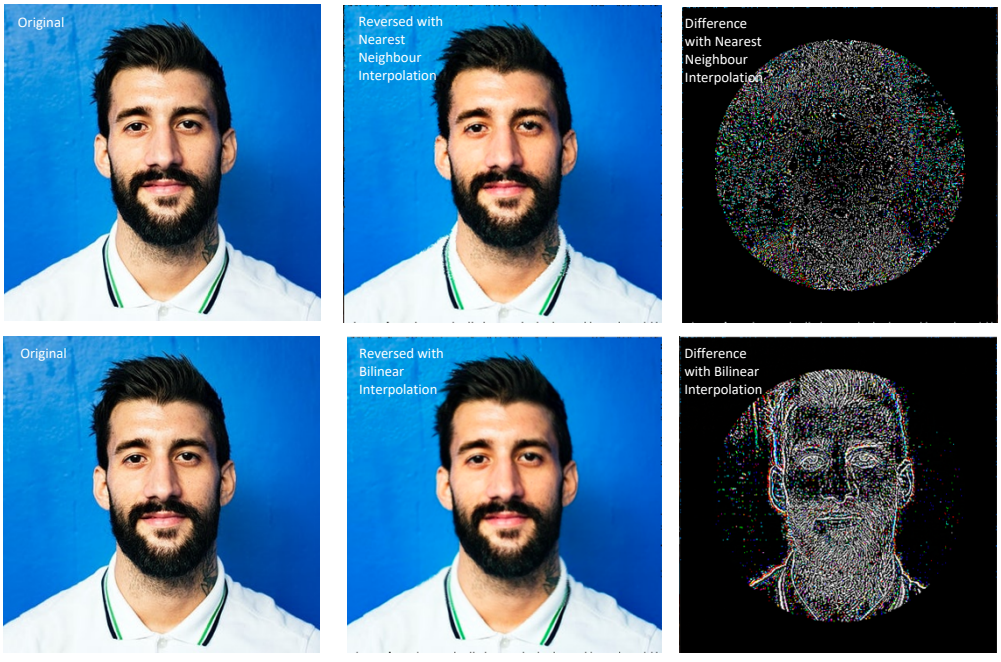
[redacted]

Images



Part 3

Images



Comments

The images above show that reversing the swirl, leaves an area of change around where the original image was swirled. In the example where nearest neighbour interpolation the differences between the images appears to be evenly distributed around the area of the swirl, whereas, in the example using bilinear interpolation, the differences appear to concentrate more around the edges of the original image. I believe these discrepancies to be caused by the rounding of pixel values and pixel coordinates involved in interpolation.