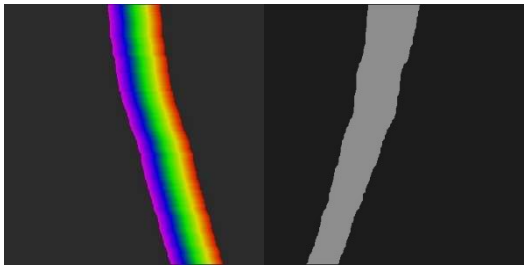
CIS: qcfb85

I have used a command line parser to implement the project, I have written example commands and documented each argument alongside the relevant command in the commands.txt file, please refer to commands.txt before attempting to run the program.

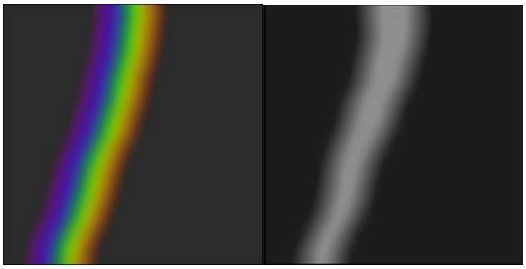
PROBLEM 1:

For Problem 1, I wanted to generate my light leak mask through Python/OpenCV itself because I thought it would be most appropriate method. To do this I created a mono-colour image of (50,50,50) and then tried to apply a leak to it by traversing through this image vertically and point transforming the elements of each row to an element of an array leak. Leak is completely (256,256,256) for 'simple' mode but is a rainbow for 'rainbow' mode.

Initial results



Final Results



To get the final results I've applied mean filtering to blur the mask to make it look more realistic, also for the 'simple' mode I applied a little randomness to shrink the leak length gradually for more realism. The direction of the leak is also decided randomly .The mask is also slightly darkened using a function I wrote called img_divide() for better contrast.

To darken the input image I blended the input image with a mono-(0,0,0) image using the blend() function I wrote for all image additions. I've also blended the mask with the darkened input image using the same function. The blend() function was made to work similarly to cv2.addWeighted() but it somehow works better(visually) so I use it for every future image addition as well. Here are the results:

The computational complexity is **O(n^3)**,blur() is the most complex part as it has a triple nested for loop.

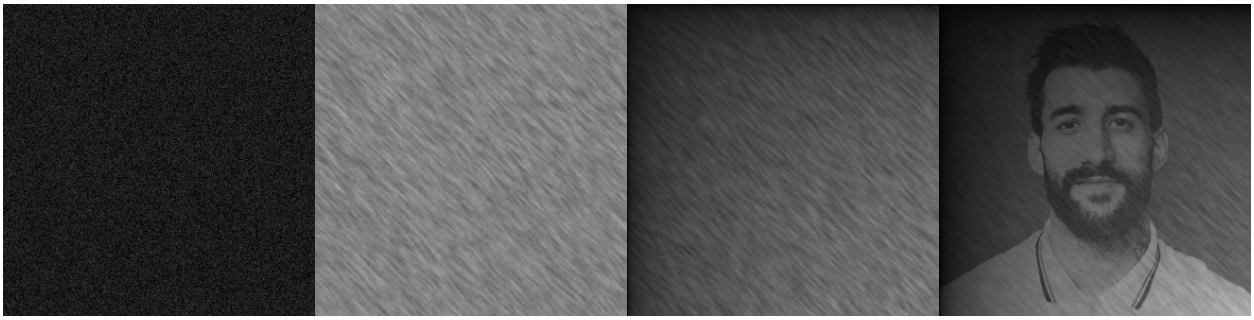
Additional arguments: **'—evaporation_coefficient'** determines how fast the length of the light leak decreases vertically ;**'—pos'** determines position of light leak ;**'—leak_thickness'** determines the thickness of the light leak.

PROBLEM 2:

For Problem 2, I first I convert the input image to Grayscale using the cvtColor() function provided by OpenCV. Then I calculate the average pixel value using my avg_col() function and store 40%(best contrast) of the number returned by avg_col() in a constant k which is used as a reference point generating noise textures.

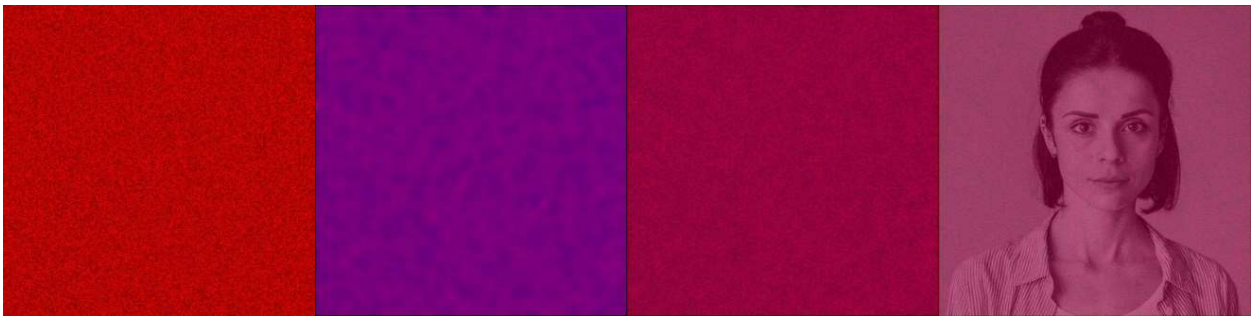
For the 'mono' option, I generate noise by applying randint(k-75,k+25) on each pixel on an image with all pixels assigned to zero and then darken the noise texture a bit more using the function img_divide(). To this texture I then add diagonal motion blur using cv2.filter2D(). The kernel matrix I use for this task has been customized to create thick diagonal strokes with a user customizable kernel size. To this texture I then add a brightness gradient from the top left of the image to the bottom right(The image gradually becomes brighter from these 2 corners. After this I blend the noise texture with the grayscale input using my blend() function and then apply the brightness gradient to this as well for a cooler effect.

Random Noise **Motion Blur** **Gradient** **Final Image**



For the 'colour' option, I first convert the grayscale image to an RGB image. Then I create 2 noise textures for 2 monochrome images, (0,G,205) and (125,G,0). One is point transformed and the other is point transformed and blurred. These 2 are then combined with the original upscaled grayscale image using 2 operations of blend().G is user defined (It's 0 for the sample below).

Red Noise **Blue Noise** **Blended Noise** **Final Image**



The computational complexity for 'mono' is **O(n^2)** due to various nested for loops but the complexity of 'coloured' is actually **O(n^3)** because it calls blur() which has this complexity.

Additional arguments: **'—kernel_size'** determines length and texture of the pencil strokes;**'—g_factor'** determines the strength of the brightness gradient ;**'—green_value'** ,determines the RGB Green value for the coloured pencil effect; **'—'rb_noise_ratio'**, determines the blending coefficient between the red and blue noise textures.

PROBLEM 3:

For Problem 3, I first smooth the image through mean filtering using the blur() function I used in the previous tasks. Here are the results:

Blur(-br)=0.4 **blur(-br)=0.8** **blur(-br)=1.6**



To beautify the image I have implemented 3 different techniques which can either be used separately or together to give the user as much choice as possible. The first filter uses look up table transformations for which I have hardcoded 2 Look up tables using for loops(one for a 'cool' image and the other for a 'warm' image). The second technique I have tried is histogram equalization on the Y channel of a YCbCr image(convert BGR to YCbCr, equalize histogram on Y, convert YCbCr back to BGR) which creates a really bright creating a cool effect. Here are the results:

LUT **LUT** **Sharp LUT** **Sharp LUT**

YCbCr HE **YCbCr HE** **HE + LUT** **All 3 Filters**

The computational complexity is **O(n^3)** due to the use of the blur() function.

Additional arguments: **'—darkening_coefficient'** determines how much the image is darkened before it is subjected to filters;**'—is_Sharpen'** ,sharpen or not; **'—is_LUT'** ,apply LUT or not; **'—is_Hist'** ,apply YCbCr Histogram equalization or not.

PROBLEM 4:

For Problem 4, I have used bilinear interpolation with a reverse mapping to create a face swirl. I felt it wasn't appropriate to use nearest neighbor interpolation as it creates a jagged image for a non-linear warps and its use should only be restricted to linear transformations. I have used an algorithm similar(with a few adjustments) to scikit-images'(I have coded it myself) to compute the swirl to get a very nice looking swirl, algorithm link:

https://scikit-image.org/docs/dev/auto_examples/transform/plot_swirl.html

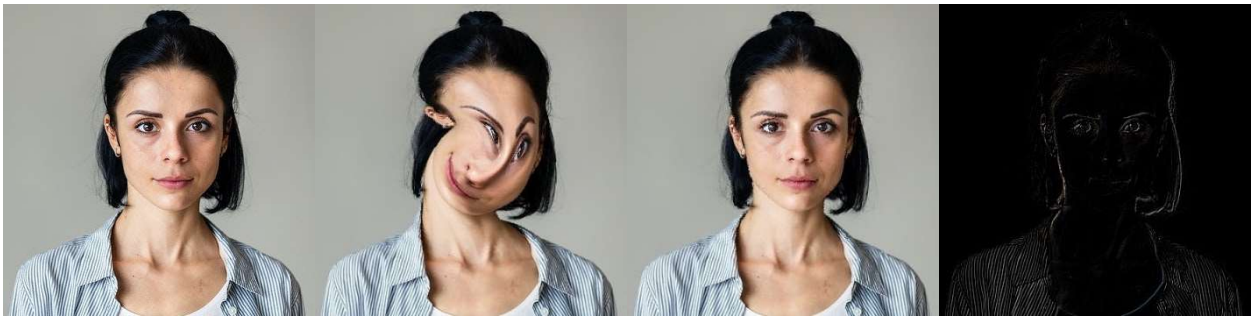
Pre-filtering(In my case mean-filtering) improves the quality of the swirl obtained, the swirl without pre-filtering tends to be more jagged especially when you increase the strength of the swirl. The quality of the inverse operation is also better with pre-filtering, here is a comparison for swirl_strength=5 and swirl_radius=120 :

Input **Swirl** **Inverse**

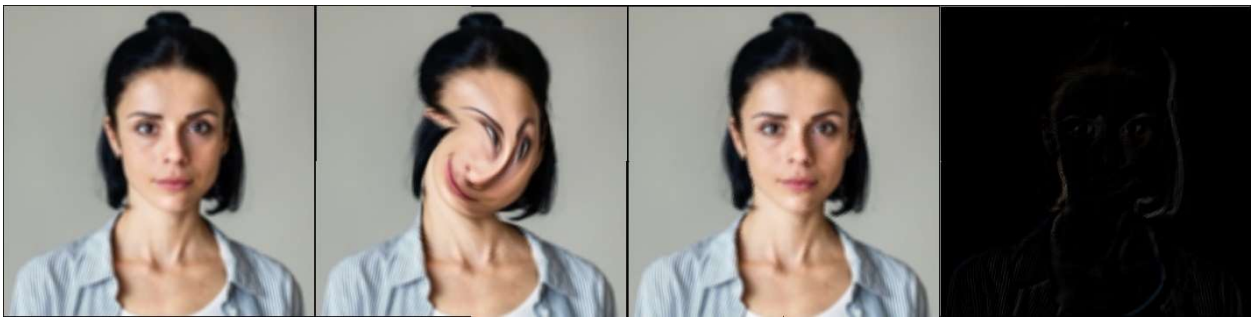
Smooth input **Swirl** **Inverse**

The only disparities between the input and the reverted swirl images are near the edge areas which are slightly defective, but overall the inverse operation is pretty accurate. The inverse operation on the filtered image is more accurate. The inverse operation only starts to deform the image when the swirl strength is increased to more than 7 but this isn't really much of an issue as the swirl strength is so strong that the swirl created makes the face completely unrecognizable anyway. I will show one more example in the next page with a lower, more feasible swirl strength of 2 and radius of 100. I've also shown the subtraction operation for this example.

Input **Swirl** **Inverse** **Subtraction**



Smooth Input **Swirl** **Inverse** **Subtraction**



The Computational complexity for this filter is **O(n^2)** for the version without low pass filtering but with mean filtering the complexity is **O(n^3)** as the blur() function is being called.

Additional arguments: **'—rotation'** determines how much the input image is rotated during the swirl(0 for no rotation)**'—blurring_amount'** ,blurring amount for low-pass filtering