Image Processing Coursework

Problem 1

nvzf61

Outline:

A mask is generated, with respect to the input image dimensions, via code. Gaussian blur is applied to the mask to reduce the harsh borders of the mask, allowing it to blend more smoothly with the image. It is then element-wise multiplied with the image to extract the masked section of the image. For the light filter, the masked section of the image and the image itself are added together with respective weightings. The result is clipped to ensure values lie between 0 and 255. For the rainbow filter, a rainbow matrix is created via code. This is set to the hue of the masked section of the image, before being added to the image.

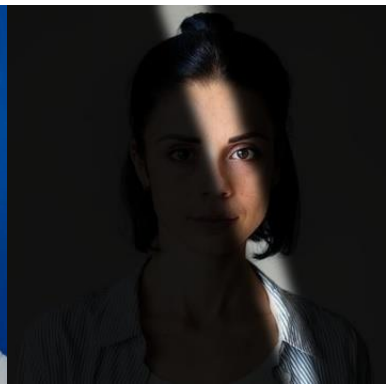
Computational complexity:

Assume the image has dimensions n*n. Then, generating the mask is $O(n^2)$. The gaussian blur function creates a kernel with dimensions k*k, this takes $O(k^2)$ which is also $O(n^2)$ since $k \leq n$. Appling the kernel takes $O(n^2)$ since it is applied to every pixel in the image. This means the gaussian blur takes $O(n^2)$. Element-wise multiplication and addition of two matrices are both $O(n^2)$. For the rainbow filter, cv2.cvtcolor takes $O(n^2)$ and all other operations perform the same or better so are not discussed. In conclusion, both light and rainbow leak take $O(n^2)$.

Images:

(1) Light leak

(2) Rainbow light leak



(3) Varying image and mask weight

(4) Varying mask blend weight

(5) & (6) Varying rainbow colour shift and step size

Problem 2

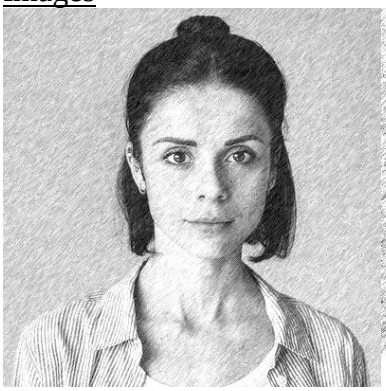
Outline

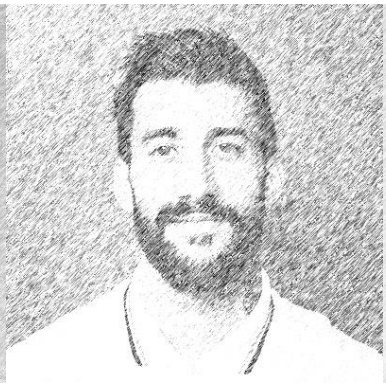
The input image is converted to greyscale and two gaussian noise images are created. Salt and pepper noise was also tried but commented out as it didn't produce the desired effect. An empty kernel is created and filled with the number 1 on the diagonal. It is then normalised so that the filter doesn't affect the average intensity of the image. The kernel is applied to the 2 noises to create a motion blur effect. For the greyscale pencil effect, the motion blur noise image is added with weighting to the greyscale image. For the colour pencil effect, the two motion blur noises are added with weightings to two respective colour channels of the BGR image.

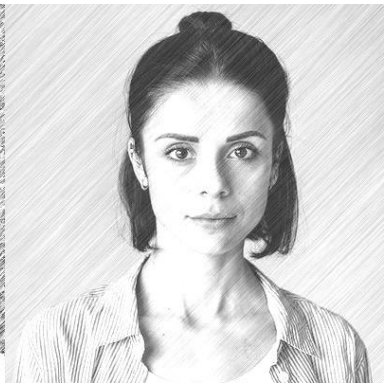
Computational complexity:

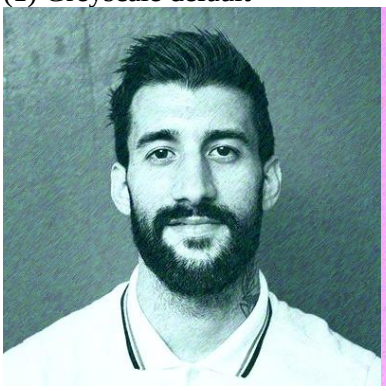
Assume the image has dimensions n*n. Converting the image to greyscale is $O(n^2)$. Creating both noise textures with dimensions n*n is $O(n^2)$ each. Creating and applying the motion blur filter using cv2.filter2D is $O(n^2)$ as previously discussed. Element-wise multiplication and addition of matrices are also $O(n^2)$. For the colour pencil effect, all operations are $O(n^2)$ or better. In conclusion, both pencil effects take $O(n^2)$.

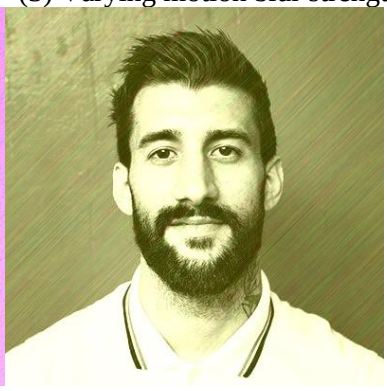
Images


(1) Greyscale default


(2) Varying mask weight


(3) Varying motion blur strength




(4), (5) & (6) Same as above but with the 3 colour pencil effects

Problem 3

Outline

A gaussian kernel is created and applied to the input image. Gaussian blurring is chosen as it is a quick and efficient way to smooth an image without discrimination. Since the sample images are already of high quality, very little blurring is required as it detracts from the definition of the image, hence gaussian is perfect. There are 3 modes. For mode 1, histogram equalisation is performed on the V channel of the HSV image as an attempt to balance contrast. For mode 2, a warmth colour curve is applied via hard-coded look up tables. This increases the red colour channel values and decreases the blue colour channels and vice versa. For mode 3, a custom contrast enhancement is applied using univariate spline with 5 sample points. This increases and decreases the light and dark sections of each colour channel respectively and vice versa.

Computational complexity:

Assume the image has dimensions n*n. Creating and applying the gaussian kernel is $O(n^2)$ as previously discussed. For mode 1, generating the histogram is $O(n^2)$ since all pixels in the image are added to their respective bins. All subsequent operations are $O(n^2)$ or better. For mode 2, all preliminary calculations are O(1). Applying the LUT on each channel is $O(n^2)$ respectively since all pixels are mapped to a new value. In conclusion, all modes are $O(n^2)$

Images:


(1) Histogram equalisation


(2) Colour cooling


(1) Contrast reduction

(1) Histogram equalisation w/ Gaussian smoothing

(2) Colour warming w/ Gaussian smoothing

(3) Contrast enhancement w/ Gaussian smoothing

Problem 4

Overview:

• By iterating through the pixels in the range of the swirl radius from the centre of the image, the polar coordinates for each pixel are calculated. The strength of the swirl is calculated for the pixel with respect to its distance from the centre of the image. The angle of the pixel is changed with respect to its distance from the centre. The input cartesian coordinates are calculated based on the new polar coordinates after the swirl.

• If interpolation == 0, nearest neighbour interpolation is used to calculate the pixel coordinates to use from the input image. If interpolation == 1, bilinear interpolation is used to calculate the value of the pixel as a weighted sum of the four neighbouring pixels in the input image, with respect to its distance from them.

• If low pass == 0, there is no Fourier low pass filter. If low pass == 1, a circular Fourier low pass filter is used to smooth the image before the geometric transform. If low pass == 2, a gaussian Fourier low pass filter is used instead.

• If inverse == 1, the new angle of the pixel is taken in the opposite direction to reverse the swirl. The cartesian coordinates are calculated and either nearest neighbour or bilinear are used to calculate the value of the pixel from the already swirled image rather than the input image. The image produced by the inverse is subtracted from the input image to show the error made by the interpolation strategies.

Computational complexity:

• Assume the image has dimensions n*n. Iterating through the pixels within the swirl radius of the centre is $O(n^2)$ since the size of the swirl must be less than the size of the image. The polar coordinate manipulation all takes O(1). Nearest neighbour interpolation is also O(1) as it only rounds each x and y value. Bilinear interpolation is essentially a weighted sum of 4 numbers so has complexity O(1). Hence the geometric transform is $O(n^2)$.

• The 2D fast Fourier transform is a 1D fast Fourier transform, with complexity O(nlog(n)), in each axis. Hence the fast Fourier and inverse fast Fourier transform is $O(n^2\log(n))$. All other operations are $O(n^2)$ or better. Since the low pass filter occurs before the geometric transform, the algorithm is $O(n^2\log(n))$ with the low pass filter on.

• The inverse swirl is O(1) as it performs the polar coordinate manipulation and interpolation within the same pixel iteration loop, so has no effect on the algorithm complexity. In conclusion, the complexity of problem four is $O(n^2\log(n))$.

Notes on speed and efficiency:

• Since all calculations are performed in a while loop to allow for real time adjustment of parameters, the function can become slow when the swirl radius is increased too high or more complex procedures, such as bilinear interpolation, are in action. To reduce this, the loop is restricted to only the pixels affected by the swirl (within the swirl radius from the centre). The mapping of these pixels is to/from the pixels within the swirl radius from the centre of the input image as only the angle of these pixels are altered, not their distance from the centre. Hence, there are no missing pixel values upon reverse mapping.

• To increase the speed of the algorithm, the inverse is calculated during the same loop as the result image is created. Since reverse mapping cannot occur on an incomplete image (it would try and retrieve values that haven't yet been generated), the inverse is calculated on the result image from the previous loop. This creates a slight delay when calculating the inverse image and the image difference.

Images:

Swirl radius and angle:

(1) Default swirl (2) Increasing swirl angle (3) Increasing swirl radius

Interpolation:

(1) & (2) Nearest neighbour (NN) (3) & (4) Bilinear (BI)

As you can see by comparing (3) and (4), there is a visible improvement on the image quality using bilinear interpolation as the result looks smoother and less pixelated.

Low pass:

(1) No low pass filter (2) Circular low pass (3) Gaussian low pass

(4) Zoomed version of (1) (5) Zoomed version of (2) (6) Zoomed version of (3)

(1), (2), (3), (4), (5) & (6) all with nearest neighbour interpolation

As you can see by comparing (5) and (6) against (4), there is a visible improvement on the image quality using a low pass filter. The circular low pass filter produces ringing artifacts, as shown in (2), which can be reduced by increasing the radius of the low pass filter. However, increasing the radius reduces the amount of artifact reduction as less blurring of the input image occurs. On the other hand, a gaussian low pass filter can be used to reduce the ringing artifacts as shown in (3).

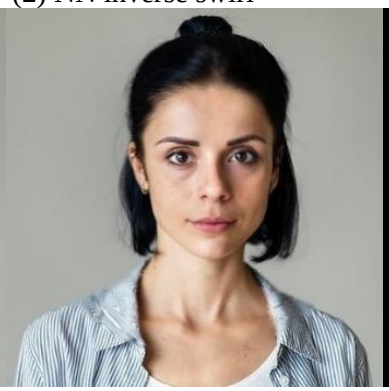
Inverse:

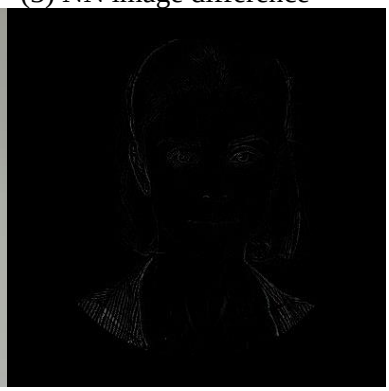

(1) NN swirl

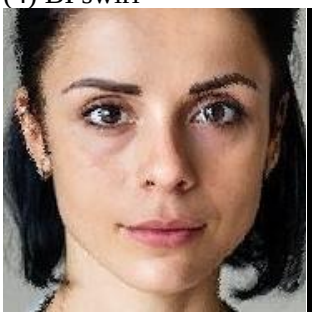

(2) NN inverse swirl


(3) NN image difference


(4) BI swirl


(5) BI inverse swirl


(6) BI image difference


(7) Zoomed of (2)


(8) Zoomed of (3)


(9) Zoomed of (5)


(10) Zoomed of (6)

As you can see by comparing (9) against (7), bilinear interpolation produces a reverse image swirl much closer to the original input image as there are less visible aliasing artifacts. This can be checked using the image differences (8) and (10). The nearest neighbour interpolation shows more brighter regions indicating a larger deviation from the original image. Also, when switching to bilinear interpolation, the error shown on the image difference is more restricted to the area of higher detail such as her head and eye outlines. This is because a weighted average of the four neighbouring pixels is more likely to give an inaccurate value at areas of sudden value change compared to areas of more consistent pixel values.