Advanced Programming Group Assignment

Image Filters, Projections and Slices

Group Insertion Sort		
Student Name	**GitHub usernames**	**Tasks**
Michael Tsesmelis	acse-mct22	Brightness, Gaussian Blur 2D&3D, Slices
Shichao Hu	acse-sh322	Box Blur, MIP, MinIP, AIP
Jie Wang	edsml-jw3222	Grayscale, Automatic Colour Balance, Sobel, Prewitt, Scharr, Roberts'Cross
Suhan Zhao	edsml-sz1222	Median Blur 2D&3D
Jiajia Zheng	acse-jz622	Histogram equalization, Slices

1 Algorithms Explanation

1.1 2D Image Filter

1.1.1 Color Correction

a. Grayscale

The grayscale equation combines the pixels of the three R, G and B channels into a single greyscale channel by means of the following equation:

$$Gray = 0.21 \cdot R + 0.72 \cdot G + 0.07 \cdot B$$

The grayscale equation is based on the brightness of colours. The weight is determined by the contribution of different colours to the luminance. As the human eye is most sensitive to green and least sensitive to blue, green is given the greatest weight and blue the least. This equation removes the colour information from the image and the result represents the brightness of each pixel, with large values indicating lighter shades of grey.

b. Automatic color balance

In order to achieve an equal average of the pixel values of the three R, G and B channels, the pixels of each channel need to be scaled. The scale factors are the following:

$$\frac{average\ of\ all\ RGB\ channels}{average\ of\ target\ channel}$$

Because the scale factor may larger than 1, the scaled pixel value may exceed 255, if so I just replace the value with 255.

c. Brightness

If a positive or negative input is given to the brightness function, we simply subtract the value from each colour channel, which either increases or decreases the brightness but generally keeps the colour palette of the image. We call add_brightness, which applies this change and caps the acceptable change to values to a range of 0 to 255. If we do not input a value to the brightness function, we simply try to bring the average brightness (here the average of all pixel values in all colour channels) to a value of 128. We do this by calculating the original average value, taking the difference, and adding or subtracting the amount to all original pixels. However, because of the cap on the values, some pixels will not be able to be increased or decreased by the required amount and this can skew the average.

d. Histogram equalization

The histogram equalization aims to increase the contrast of an image by re-distributing the pixel values. We call histogram_equalize, which maps original pixel values to new values that resulting a more uniform distribution of gray levels through following steps.

Step 1: Computing the histogram of an image. For our image, the histogram has 256 bins ranging from 0 (the lowest gray level) to 255 (the highest gray level).

Step 2: Normalize the histogram values by dividing the total number of pixels (256) and calculate the cumulative distribution function (CDF) of normalized histogram. This involves summing up the normalized histogram values from the lowest gray level to the highest.

Step 3: Scaling CDF values to the range of all pixel values (0 to 255) by multiplying the maximum pixel value.

Step 4: Applying the histogram equalization to the original grayscale image by replacing each pixel value with its corresponding value in the equalized CDF.

1.1.2 Image blur

a. Median blur

First, we add reflected padding around the edges of the 2D matrix (represented in a 1D form). We then apply the convolution, image pixel-by-pixel in the external loops, and kernel pixel-by-pixel in the inner loops. At each kernel pixel, we add values to a list. We tried using vectors, but this entailed significant slowdowns. Once we have the values of the entire kernel, we use our own util::get_median_odd function to sort the values and then retrieve the middle one. We sort the values thanks to a simple list-based histogram, which is cheap to access and store. This is important since we run the get_median_odd function a very high number of times: the amount of pixels in the image.

b. Box blur

Box blur uses a similar method to median blur, except that the kernel computes a uniform weighted average of neighboring values. We simply build a kernel where all values are equal and together sum up to 1. At every kernel pixel, we multiply the weight of the respective kernel cell with the pixel value. The result is added to a running sum for the entire kernel. We have developed a separate convolution function for this part, since the gaussian blur uses a very similar structure and we can combine the two methods.

c. Gaussian blur

Gaussian blur uses a similar method to median blur, except that the kernel computes a weighted average of neighboring values according to the Gaussian function. We have built a function that builds a kernel for us, based on the formula:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

At every kernel pixel, we multiply the weight of the kernel location with the pixel value. The result is added to a running sum for the entire kernel. We also add the kernel weight to a factor variable. We normalise the gaussian kernel so that all weights sum up to 1.

1.1.3 Edge detection

a. Sobel filter

The Sobel operator includes two 3×3 convolutional kernels, as follows:

$$horizontal: \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad vertical: \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

For each group of 3×3 pixels in the target image, perform the following convolution algorithm to calculate gradient $(G_x,\ G_y)$, The magnitude of the gradient is then used to determine the edges:

$$target: \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix}$$

$$G_x = horizontal * target = -a_1 + a_3 - 2b_1 + 2b_3 - c_1 + c_3$$

$$G_y = vertical * target \quad = -a_1 - 2a_2 - a_3 + c_1 + 2c_2 + c_3$$

$$G_{xy} = \sqrt{G_x^2 + G_y^2} = b_2$$

So, we update the value of b_2, then move down the filter and update the value of c_2. For each row and column, my algorithm will update all but the first and last pixel. We can solve this problem via padding, but it is not necessary for high resolution images.

b. Prewitt & Scharr & Roberts' Cross filter

The Prewitt, Scharr and Roberts' Cross operators are similar with Sobel operators and use the same convolution algorithm, with the Roberts' Cross filter being the fastest but least accurate due to its simple kernel. Scharr filter is the slowest but the most accurate. However, for this reason, Scharr is very sensitive to noise and if the input image contains a lot of noise, the output quality will instead be inferior to other algorithms. Therefore, it requires the input image to have a very high quality.

Besides, applying Gaussian blur before applying any edge detection filter can improve the quality of the output. As the edge detection filter calculates the gradient of the image, it is very sensitive to noise and will misidentify them as edges. The Gaussian blur removes noise from the image and blurs the edges, so the edge detection filter can output smoother and more accurate edges. For the given test image, 5×5 Gaussian blur kernel is most appropriate. Smaller kernels will retain too much noise and larger kernels will result in blurred edge transitions.

1.2 3D Data Volume

1.2.1 Filters

a. 3D Gaussian

The 3D gaussian has the same structure as the two-dimensional version, except that we have added a dimension to the paddeddata and the convolution loop. The paddeddata is therefore a double pointer, which after experiments runs just as fast as working with a flattened 3D representation. We of course must also work with a three-dimensional kernel, which uses the same formula above except using the squared values of x, y and z representing the three dimensions.

b. 3D Median

For the 3D median, we must now find the median of cubic kernels. Such an exponentially growing function naturally adds a huge burden to the overall performance of the algorithm, which is mostly penalised by data accesses and the get_median_odd function. Additionally, the image is three dimensional now as well, with more pixels to calculate the kernel for, and therefore we are doubly penalised.

1.2.2 Projections

In order to obtain the projection over a specified slab, the 'Slab' function is defined, and it will be called in every projection function; it ensures that the input slab range is within the valid bounds of the volume data.

a. Maximum intensity projection (MIP)

For the mip function, it allocates memory for the output image with the same width, height, and channel as the input volume data. Then, we use a nested loop to glance over each pixel position and its value in each slice, the maximum intensity value is obtained by using if statement. At the end of the loop, the output array will be stored with maximum values.

b. Minimum intensity projection (MinIP)

Similar to MIP, but we initially set all values in the output array to 255, in order to successfully find the minimum intensity value at every pixel position by comparing the value.

c. Average intensity projection (AIP)

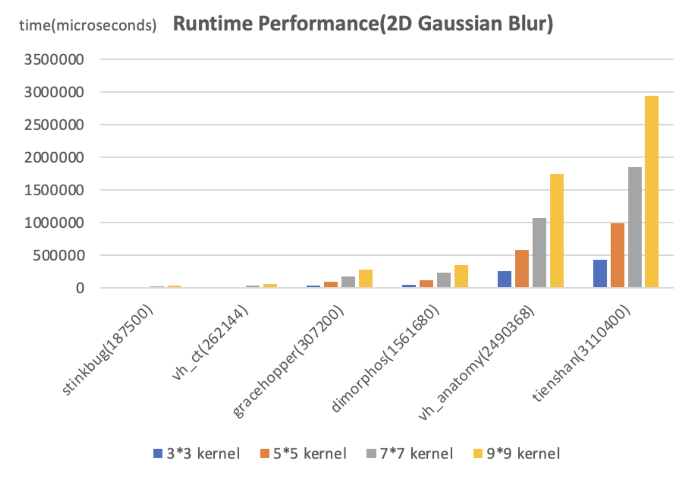
The aip function computes either the median or mean of the intensity values through applying 'use_median' boolean flag. For median, it iterates through every pixel position, and for each iteration, all slice values at that position are stored in an array, and we then locate the median value through calling util:get_median function and assigning it to the output array for that position. For the mean, a vector is created and initialized with zeros, iterations are used to sum all slice values at each pixel position, store them in the vector, and then through another iteration we divide the sum value for each pixel position by the number of slices (the depth of the volume) to find the mean, this mean is then assigned to the corresponding position in the output array.

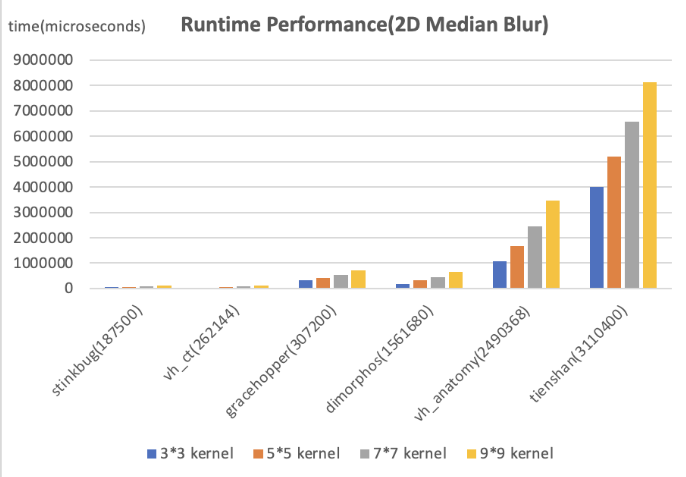
1.2.3 Slices

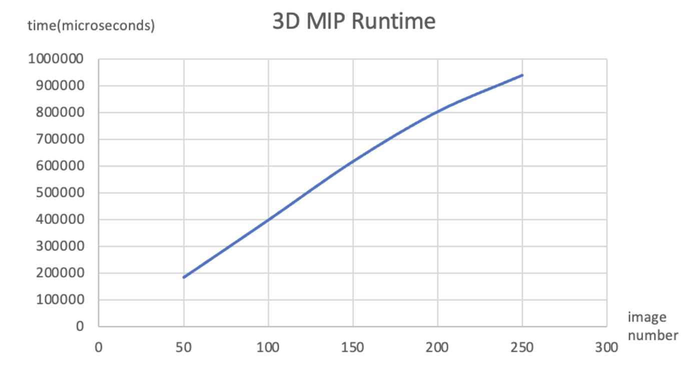
For the slices, we have decided to opt for a semi-flattened dataset of the volume, with each image staying flat as in the 2D case. Thus, we work with a double pointer. We have witnessed significant speedups thanks to this method, since the data is contiguous in memory, and the computer does not have to “chase pointers” across three dimensions. We flatten the data by copying each image into a single vector, as we did for all 2D filters. Afterwards, we simply retrieve the data at the right locations, which is especially nice in the case of a YZ-slice as there is a periodic location in each single image where we need to retrieve the pixels to build the slice.

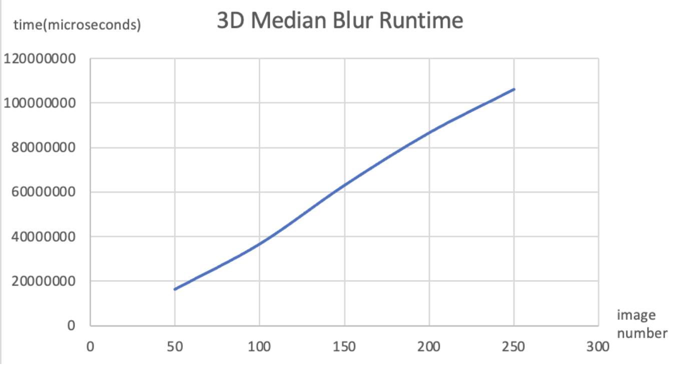
2 Performance Evaluation

Below two figures show the library performance under different conditions: the run time becomes longer for larger image size, kernel size and volume size. We see exponential increases in time with respect to kernel size.









3 Potential Improvement

In designing our library, we opted for ease-of-use. Our filters and transformations run with very little lines of code, and once the image is loaded into an Image or Filter object, it is very easy to apply many different filters without having to keep track of size changes, or having to input the data to the filters again. Originally, our Filter objects kept state of the output images, which allowed for sequential transformations. However, based on our inspiration from scipy in Python, this is not commonplace and so we removed that option. Our library is therefore memory-lightweight. To increase speeds on some functions, we even switch between two-dimensional and flat representations of 3D slices.
The big disadvantage of our library is the little flexibility of the UI, as well as poor error mitigation according to modern software development and cybersecurity standards. A lot more code could be implemented to properly clean all the inputs to the functions. We also have room for improvement when it comes to parametrising the functions. Scipy offers a flurry of parameters for each of its functions, which allow the users to precisely define how they want to transform their images. Our parameters are limited mostly to an odd kernel size or to an integer brightness value, we offer little in flexibility or modularity.