|  |  |
| --- | --- |
| **­Image processing** | **Task No. 1** |
| **Task variant: Group 1**  (all B, all G, N9, all E) | |
| **Day and time** November 4, 2022  **Academic year 2022/2023** | **Full name : Piotr Czapla 234751**  **Full name : Aleksandra Banasiak 234750** |
| **Technical description of the application**  In order to fulfill the requirements we decided to focus on programming our application in C++ programming language as we had the basic knowledge from previous projects. Not to mention the research we performed to find the suitable library with open source and free usage for students. We decided to focus on features of the CImg Library which is a small and open-source C++ library for image processing and thanks to that we were able to create function which will be described precisely in further points of the report.  Additionally to create command line application we used available Program Options Parser Library which is a C++ command line arguments parser that supports the same set of options as GNU's getopt and thus closely follows the POSIX guidelines for the command-line options of a program. We used available resources from <https://github.com/badaix/popl> to implement our solution.  To run the program, the user have to open the cmake-build-debug folder in the command prompt, enter the command "--help" to display all available features.  In order to estimate the time of the performance the algorithms we used available on Windows application “PowerShell” executing the command:  Measure-Command {path to executive --command (argument if available) input path output path| Out-Default}  Since in case of calculations two input files are being used, both of them are stored as a CImg objects.  CImg<**unsigned char**> Image; CImg<**unsigned char**> Image\_for\_testing;  In case of the output path, it is stored in a character pointer.  **char**\* result;  When it comes to assigning values from the argument vector the program check if the command requires an argument and assigns the third value to the Image variable and the forth to the result variable. In case of the commands which don’t modify any picture the second input file is assigned to the Image\_for\_testing variable.  As mentioned above for command line options we used the popl.hpp header and the structure of it looks as follows:  OptionParser op(**"Allowed options"**); **auto** help\_command = op.add<Switch>(**""**, **"help"**, **"produce help message"**); **auto** brightness\_command = op.add<Value<**int**>>(**""**, **"brightness"**, **"brightness modification"**); **auto** contrast\_command = op.add<Value<**int**>>(**""**, **"contrast"**, **"contrast modification"**); **auto** negative\_command = op.add<Switch>(**""**, **"negative"**, **"turn image into negative"**);  **auto** horizontal\_flip\_command = op.add<Switch>(**""**, **"hflip"**, **"Horizontal flip"**); **auto** vertica\_flip\_command = op.add<Switch>(**""**, **"vflip"**, **"Vertical fli"**); **auto** diagonal\_flip\_command = op.add<Switch>(**""**, **"dflip"**, **"Diagonal flip"**); **auto** image\_shrink\_command = op.add<Value<**float**>>(**""**, **"shrink"**, **"Image shrinking"**); **auto** image\_enlarge\_command = op.add<Value<**float**>>(**""**, **"enlarge"**, **"Image enlargement"**);   **auto** adaptive\_median\_filter\_command=op.add<Switch>(**""**, **"adaptive"**, **"Adaptive median filter"**); **auto** arithmetic\_mean\_filter\_command=op.add<Value<**int**>>(**""**, **"amean"**, **"Arithmetic mean filter"**);  **auto** mean\_square\_error\_option = op.add<Switch>(**""**, **"mse"**, **"Mean square error"**); **auto** peak\_mean\_square\_error\_option = op.add<Switch>(**""**, **"pmse"**, **"Peak mean square error"**); **auto** signal\_to\_noise\_ratio\_option = op.add<Switch>(**""**, **"snr"**, **"Signal to noise ratio"**); **auto** peak\_signal\_to\_noise\_ratio\_option = op.add<Switch>(**""**, **"psnr"**, **"Peak signal to noise ratio"**); **auto** maximum\_difference\_option = op.add<Switch>(**""**, **"md"**, **"Maximum difference"**);  Initially we create an option parser which also takes the strings which is later displayed at the top in the command line. Moving on we create variables of type auto which store object added to the option parser. Each object can be either a Switch, a Value or an Implicit where Value and Implicit both take in an argument of given type. In case of Implicit the argument is optional. What is more each object be default take three strings, first of them being a short command, the second one being a long command and the third one as the message displayed after using the –help command.  Having done that we set the option parser to take from the main function the argument cout and the argument vector.  op.parse(argc, argv);  For checking the correctness of commands written by the user we use a for loop which checks if a unknown\_option type of exception is thrown and if so the message is displayed.  **for** (**const auto**& unknown\_option: op.unknown\_options())cout << **"This operation is not valid: "** << unknown\_option <<**" Try --help for a list of commands"**<<endl;  In case of checking with command is being used by the user we use a is\_set()function which returns true when the command is used and false by default with the following example for the contrast command which additionally passes the argument from to option parser to the contrast\_modification and saves the output using the save function which takes the abovementioned character pointer as a path:  **if**(contrast\_command->is\_set()){  contrast\_modification(Image,contrast\_command->value());Image.save(result); } | |
| **Description of implementation of basic image operations**   * Description (B1) Image brightness modification     **Figure nr.1** Original picture, before editing    **Figure nr.2** Picture after execution of brightness method with value 100  To adjust the brightness of an image, we created the function called brightness\_modification which as a variable from the user takes a constant type integer by which the brightness of the picture should be increased (effect is shown on Figure nr.1). It changes the value of all pixels by a this handed over constant. Adding a positive constant to all of the image pixel values makes the image brighter. Similarly the user can subtract a positive constant from all of the pixel values to make the image darker. As the method is created with two nested “for” loops we conclude that the time complexity is O(n2) and total milliseconds of performance this algorithm is 789.275, moreover during the process the amount of memory used is equal to O(n2) assuming that width of the input image is equal to the height of it, plus the memory taken by the output image which is equal to the size of the input image.  **void** brightness\_modification(CImg<**unsigned char**> &image,**int** constant) {  **if**(constant >= 0){  **for** (**int** x = 0; x < image.width(); x++){  **for** (**int** y = 0; y < image.height(); y++){  **float** valR = image(x, y, 0) + constant;  **float** valG = image(x, y, 1) + constant;  **float** valB = image(x, y, 2) + constant;   **if**(valR <= 255) image(x, y,0) = valR;  **else** image(x, y,0) = 255;   **if**(valG <= 255) image(x, y,1) = valG;  **else** image(x, y,1) = 255;   **if**(valB <= 255) image(x, y,2) = valB;  **else** image(x, y,2) = 255;  }  }  }**else**{  **for** (**int** x = 0; x < image.width(); x++){  **for** (**int** y = 0; y < image.height(); y++){  **float** valR = image(x, y, 0) + constant;  **float** valG = image(x, y, 1) + constant;  **float** valB = image(x, y, 2) + constant;   **if**(valR >= 0) image(x, y,0) = valR;  **else** image(x, y,0) = 0;   **if**(valG >= 0) image(x, y,1) = valG;  **else** image(x, y,1) = 0;   **if**(valB >= 0) image(x, y,2) = valB;  **else** image(x, y,2) = 0;  }  }  } }   * Description (B2) Image contrast modification     **Figure nr.3** Picture after execution of contrast method with value of 200  Contrast is the difference between maximum and minimum pixel intensity in an image. Intending change the contrast of the picture we created the method called contrast\_modification which takes value of intensity as the variable from the user. After executing this method (effect shown on Figure nr.3) it changes the contrast of an image by changing the value of the max and min intensity pixel. In this case the method is also created with two nested “for” loops which leads to the conclusion that the time complexity is O(n2) and total milliseconds of performance this algorithm is 149.5214, moreover during the process the amount of memory used is equal to O(n2) assuming that width of the input image is equal to the height of it, plus the memory taken by the output image which is equal to the size of the input image. This idea for such a way of creating the correction factor was taken from this article https://ie.nitk.ac.in/blog/2020/01/19/algorithms-for-adjusting-brightness-and-contrast-of-an-image/  **void** contrast\_modification(CImg<**unsigned char**> &image,**int** intensity){**for** (**int** x = 0; x < image.width(); x++) {  **for** (**int** y = 0; y < image.height(); y++) {**float** correction\_factor = (259\*(255+intensity))/(255\*(259-intensity));  **float** valR = correction\_factor\*(image(x, y, 0)-128)+128;  **float** valG = correction\_factor\*(image(x, y, 1)-128)+128;  **float** valB = correction\_factor\*(image(x, y, 2)-128)+128;   **if**(valR <= 0){  image(x, y, 0) = 0;  }  **else if** (valR >= 255) image(x, y,0) = 255;   **else** image(x, y, 0) = valR;   **if**(valG <= 0){  image(x, y, 1) = 0;  }  **else if** (valG >= 255) image(x, y,1) = 255;   **else** image(x, y,1) = valG;   **if**(valB <= 0){  image(x, y, 2) = 0;  }  **else if** (valB >= 255) image(x, y,2) = 255;   **else** image(x, y,2) = valB;  }  }}   * Description (B3) Negative     **Figure nr.4** Picture after execution of negative method  Negative transformation refers to subtracting pixel values from (L-1), where L is the maximum possible value of the pixel, and replacing it with the result. To negatively transform an image, we loop through the pixels using two for loops – again the time complexity is O(n2) and total milliseconds of performance this algorithm is 63.351, moreover during the process the amount of memory used is equal to O(n2) assuming that width of the input image is equal to the height of it, plus the memory taken by the output image which is equal to the size of the input image. If the image is RGB, the red, green, and blue values are subtracted from (L-1) and the result is stored in place of the values. In the case of greyscale images, the intensity of the pixels is subtracted instead. Negative transformation is done to bring attention to detail in the darker regions of the image. The effect of the executed method is visible on the Figure nr.4  **void** negative(CImg<**unsigned char**> &image){  **for** (**int** x = 0; x < image.width(); x++) {  **for** (**int** y = 0; y < image.height(); y++){   **float** valR = image(x, y, 0);  **float** valG = image(x, y, 1);   **float** valB = image(x, y, 2);   **float** negative1 = 255-valR;  **float** negative2 = 255-valG;  **float** negative3 = 255-valB;  image(x, y, 0) = negative1;  image(x, y, 1) = negative2;  image(x, y, 2) = negative3;  }  } }   * Description (G1) Horizontal flip     **Figure nr.5** Picture after execution of horizontal flip method  Horizontal Flip is a data augmentation technique that takes both rows and columns of such a matrix and flips them horizontally. As a result, you will get an image flipped horizontally along the y-axis (shown on Figure nr.5). We implemented this algorithm in a function called horizontal\_flip which is created with two nested loops iterating over given image width and then over the height, which again leads to time complexity O(n2) and total milliseconds of performance this algorithm is 64.5069, moreover during the process the amount of memory used is equal to O(n2) assuming that width of the input image is equal to the height of it, plus the memory taken by the output image which is equal to the size of the input image.  **void** horizontal\_flip(CImg<**unsigned char**> &image){  CImg<**unsigned char**> buffer = image;  **for** (**int** x = 0; x < image.width(); x++) {  **for** (**int** y = 0; y < image.height(); y++) {  buffer(x, y, 0) = image(image.width()-x, y, 0);  buffer(x, y, 1) = image(image.width()-x, y, 1);  buffer(x, y, 2) = image(image.width()-x, y, 2);  }  }  image = buffer; }   * Description (G2) Vertical flip     **Figure nr.6** Picture after execution of vertical flip method  Vertical Flip flips algorithm works the same as described in horizontal flip with one distinction that image flips along the x-axis (result shown on Figure nr.6). Analogously the algorithm has the time complexity O(n2) and total milliseconds of performance this algorithm is 68.9925, moreover during the process the amount of memory used is equal to O(n2) assuming that width of the input image is equal to the height of it, plus the memory taken by the output image which is equal to the size of the input image.  **void** vertical\_flip(CImg<**unsigned char**> &image){  CImg<**unsigned char**> buffer = image;  **for** (**int** x = 0; x < image.width(); x++) {  **for** (**int** y = 0; y < image.height(); y++) {  buffer(x, y, 0) = image(x, image.height()-y, 0);  buffer(x, y, 1) = image(x, image.height()-y, 1);  buffer(x, y, 2) = image(x, image.height()-y, 2);  }  }  image = buffer; }   * Description (G3) Diagonal flip     **Figure nr.7** Picture after execution of diagonal flip method  The colored image can be represented as a 3 order matrix. The first order is for the rows, the second order is for the columns and the third order is for the layers, the pixel value will determine the color of the pixel based on the color format. Approach we took to implement diagonal flip is separating each layer, then flipping every loops iterate the width and height leading to time complexity of O(n2) and total milliseconds of performance this algorithm is 64.6001, moreover during the process the amount of memory used is equal to O(n2) assuming that width of the input image is equal to the height of it, plus the memory taken by the output image which is equal to the size of the input image.  **void** diagonal\_flip(CImg<**unsigned char**> &image){  CImg<**unsigned char**> buffer = image;  **for** (**int** x = 0; x < image.width(); x++) {  **for** (**int** y = 0; y < image.height(); y++) {  buffer(x, y, 0) = image(image.width()-x, image.height()-y, 0);  buffer(x, y, 1) = image(image.width()-x, image.height()-y, 1);  buffer(x, y, 2) = image(image.width()-x, image.height()-y, 2);  }  }  image = buffer; }   * Description (G4) Image shrinking     **Figure nr.8** Picture after execution of shrink method  We decided to use the Nearest neighbor Image Scaling algorithm as it is the simplest and fastest implementation of image scaling technique. The existing pixel values are the only information we have access to in order to generate a larger or smaller version of that image. (The shrink effect is shown of Figure nr.8). When it comes to time complexity we deal with two nested loops as it results with O(n2) and total milliseconds of performance this algorithm is 45.1901. In case of memory allocation, since the program accepts values only from the 0-1 range, the output image takes space equal to the size of the input image times parameter2 and the space complexity is equal to O(n2) assuming that width of the input image is equal to the height of it.  **void** shrink(CImg<**unsigned char**> &image,**float** multiplier){  CImg<**unsigned char**> buffer (image.width()\*multiplier,image.width()\*multiplier,1,3,0);  **for** (**int** x = 0; x < buffer.width(); x++) {  **for** (**int** y = 0; y < buffer.height(); y++) {  buffer(x, y, 0) = image(x/multiplier, y/multiplier, 0);  buffer(x, y, 1) = image(x/multiplier, y/multiplier, 1);  buffer(x, y, 2) = image(x/multiplier, y/multiplier, 2);  }  }  image = buffer; }   * Description (G5) Image enlargement     **Figure nr.9** Picture after execution of enlargement method  As described upon we used the Nearest neighbor Image Scaling algorithm and for larger versions of the original image, we took the original pixel values and place them analogically across the new specified dimension, so we fill up our new canvas size and the using resampling algorithm refilling those vacant positions (The final result of invoking this method is shown on Figure nr.9). Similarly to previously mentioned algorithm the time complexity is O(n2) and total milliseconds of performance this algorithm is 170.558. In case of memory allocation, since the program accepts values only from the 1-inf range,the output image takes space equal to the size of the input image times parameter2 and the space complexity is equal to O(n2) assuming that width of the input image is equal to the height of it.  **void** enlarge(CImg<**unsigned char**> &image,**float** multiplier){  CImg<**unsigned char**> buffer (image.width()\*multiplier,image.width()\*multiplier,1,3,0);  **for** (**int** x = 0; x < buffer.width(); x++) {  **for** (**int** y = 0; y < buffer.height(); y++) {  buffer(x, y, 0) = image(x/multiplier, y/multiplier, 0);  buffer(x, y, 1) = image(x/multiplier, y/multiplier, 1);  buffer(x, y, 2) = image(x/multiplier, y/multiplier, 2);  }  }  image = buffer; } | |
| **Description of implementation of noise reduction methods**   * The first method : Adaptive median filter     **Figure nr. 10** Before and after- result of the method adaptive\_median\_filter  An adaptive median filter performs spatial processing to reduce noise in an image. The filter compares each pixel in the image to the surrounding pixels, when one of the pixel values differ significantly from the majority of the surrounding pixels, the pixel is treated as noise. The filtering algorithm then replaces the noise pixel by the median values of the surrounding pixels(result shown on Figure nr.10). This process repeats until all noise pixels in the image are removed. As the implementation of the algorithm contains three nested “for” loops the time complexity is O(n2) and total seconds of performance this algorithm is shown in section “Analysis of the noise reduction methods w. r. t. the possible applications for various types of noise”. the space complexity is equal to O(n2) assuming that width of the input image is equal to the height of it plus the size , plus the memory taken by the output image which is equal to the size of the input image . The filter itself is divided into 4 functions. stage\_B being responsible for checking if the difference between the given intensity and minimum and maximum intensity and bigger than zero and smaller than zero respectively.stage\_A checks if the difference between the median intensity and min and maximum intensity is bigger than zero and smaller than zero respectively, if so the program moves to stage\_B, and of not program checks if the maximum allowed window is not exceeded. If the condition is true it return a number which is later on used in the median function and if not the intensity is returned. Having all this data the median function using the functions from the #include **<vector>** store all the values from the given image into a vector structure and after sorting them obtains both minimum and maximum intensity values. Having done that program checks if the size of the vector is an even or an odd number and takes the median of intensity values accordingly.  **int** stage\_B(**int** zxy, **int** zmed, **int** zmin, **int** zmax){  **int** B1=zxy-zmin;  **int** B2=zxy-zmax;  **if**(B1>0 && B2<0){  **return** zxy;  }  **else**{  **return** zmed;  } } **int** stage\_A(**int** zxy, **int** zmed, **int** zmin, **int** zmax, **int** Sxy, **int** Smax){  **int** A1 = zmed - zmin;  **int** A2 = zmed - zmax;  **if**(A1 > 0 && A2 < 0){  **return** stage\_B(zxy, zmed, zmin, zmax);  }  **else**{  **if**(Sxy < Smax){  **return** 123456;  }  **else** {  **return** zxy;  }  } } **int** median(CImg<**unsigned char**> &image1,**int** x,**int** y,**int** z){  **int** Sxy = 1;  **int** Smax = image1.width() - x;  **if**(image1.height()-y < Smax){  Smax = image1.height() - y;  }  **if**(x < Smax){  Smax = x;  }  **if**(y < Smax){  Smax = y;  }  **int** median = 123456;  **while**(median == 123456){  std::vector<**int**> pixels;  **for** (**int** i=x-Sxy;i<x+Sxy;i++){  **for** (**int** j=y-Sxy;j<y+Sxy;j++){  pixels.push\_back(image1(i, j, z));  }  }  std::sort(pixels.begin(),pixels.end());  **int** zmed;  **int** zxy = image1(x,y,z);  **int** zmin = pixels.front();  **int** zmax = pixels.back();      **if**(pixels.size()%2 == 0){  zmed = (pixels[pixels.size()/2-1]+pixels[pixels.size()/2])/2;  }  **else**{  zmed=pixels[pixels.size()/2];  }  median = stage\_A(zxy,zmed,zmin,zmax,Sxy,Smax);  Sxy++;  }  **return** median; } **void** adaptive\_median\_filter(CImg<**unsigned char**> &image){CImg<**unsigned char**> buffer = image;  **for** (**int** x = 1; x < image.width(); x++) {  **for** (**int** y = 1; y < image.height(); y++) {**for**(**int** z=0;z<3;z++){  buffer(x,y,z)= median(image,x,y,z);  }  }  }  image=buffer;}   * The second method : Arithmetic mean filter     **Figure nr. 11** Before and after- result of the method arithmetic\_mean\_filter  Arithmetic mean filter calculates the average value in a set of pixel values. In other words, we sum up all pixel values within the set and divide them by the size of that set. Convolution works by multiplying each coefficient in the kernel that coincides with the pixel value and summing it all up to form output value in the centre of it. The formula of this filter is shown on Figure nr. 12. As the implementation of the algorithm contains two nested loops the time complexity is O(n2) and total seconds of performance this algorithm is shown in section “Analysis of the noise reduction methods w. r. t. the possible applications for various types of noise”. and the space complexity is equal to O(n2) assuming that width of the input image is equal to the height of it , plus the memory taken by the output image which is equal to the size of the input image.    **Figure nr.12** Formula of the arithmetic mean filter  **void** arithmetic\_mean\_filter(CImg<**unsigned char**> &image,**int** constant) {CImg<**unsigned char**> buffer = image;  **int** modifier=(2\*constant+1)\*(2\*constant+1);  **for** (**int** x = constant; x < image.width()-constant; x++) {  **for** (**int** y = constant; y < image.height()-constant; y++) {  **for**(**int** z=0;z<3;z++){  **double** result=0;  **for**(**int** i = x-constant;i<=x+constant;i++){  **for**(**int** j = y-constant;j<=y+constant;j++){  result+=((image(i,j,z))/(pow(2\*constant+1,2)));}  }  buffer(x,y,z)=**int**(result);  }  }  }  image=buffer; } | |
| **Analysis of parameters of the noise reduction methods**   * Adaptive-median filter   When we consider value 0 of the passed parameter the effect of the filter is not visible. Although with given value greater than 1 there are noticeable changes. We consider this filter as more effective one as it does not blur the image as the one described below, the efficiency of the noise reduction maintains the quality.   * Arithmetic mean filter   As in previous example when the passed parameter is 0 the effect of the filter is not visible. However when given value greater than 1 there are visible differences. The output of processing image tend to be blurred. Continued increasing parameter slightly upgrades the noise reduction, image is not distorted but even more blurry, but the time devoted for processing it is significantly elongated. | |
| **Analysis of the noise reduction methods w. r. t. the possible applications for various types of noise**  Results and conclusions    **Table nr.1** Comparision of the errors after performing noise reduction functions  ­­­According to our observations adaptive-median filter in comparison to the arithmetic mean filter provides better results. The reason we state that thesis is that arithmetic mean filter leaves the output image very blurry, while adaptive median filter doesn’t, as the result does not live up to expectations due to a loss in image quality. | |
| **Teacher's remarks** | |