| Image processing | Task No. 1 |
| --- | --- |
| **Task variant: variant I** | |
| **Day and time: 23.10.2023, 10:30**  Academic year: 2023/24 | **Full name: Małgorzata Komorowska**  **Full name: Filip Andrzejewski** |
| **Technical description of the application**  The application is a command line tool written in python language, designed for modification of a given image of any size and analysis of the resulting image.  All of it’s features accept input of both grayscale images and RGB images.  The application makes use of 2 external libraries and 3 built-in libraries:  External:  Pillow library - used only for loading given images into the memory and saving the images after the modifications:   * Opening and loading the image: im = Image.open("lenac.bmp") * Saving the result image to the root directory: result\_image.save("new\_image.bmp")   NumPy library - used for conversion of images to a numpy.array:   * img = np.array(image)   Built-in:  Math library - used for complex mathematical operations in 3 cases:   * Exponential function of image after contrast modification:   - new = 128 + ((i - 128) \* (math.e \*\* factor))   * Logarithmic functions in signal to noise ratio and peak signal to noise ratio calculation.   - snr\_value = 10 \* np.log10(sum\_squared\_signal / sum\_squared\_diff)  - psnr\_value = 10 \* np.log10(max\_squared\_value / sum\_squared\_diff)  Sys library - used for gathering user command input:   * if sys.argv[1] == "--brightness"   Time library - used during measurements of execution time of functions:   * time\_start = time.time()   The images are stored in the memory as instances of Image object from Pillow Library which are composed of color mode data, image dimensions and image data - array of pixel values.  The pixel values are stored either as an singular numeric value in case of grayscale (L mode) images or tuple of 3 values in case of RGB mode images.  In some cases in order to speed up the execution of the function the image data is being converted to an instance of numpy.array which allows for faster modifications. After the calculation the | |
| **Description of implementation of support functions**  In order to make work more efficient 4 support functions were created:   * Save\_image   This function saves modified image.  Parameters:  image: A PIL Image object representing the input image.  output: Path to the output file  The algorithm:   * Saves the image to desired path using PIL library function of Image class Image.save * process\_int\_or\_tuple   This function helps with processing images of unknown color modes.  Patameters:  item: a unknown type variable  The algorithm:   * Checks the type of the item parameter. * If it is a tuple (RGB color notation) it will return an array of this tuple elements * If it is not a tuple (single int in Grayscale color notation) it will return a tuple of size 3 of 3 copies of this item. * analyse\_color\_channels   This function is used to analyse the image in regards of it’s color mode.  Patameters:  image: A PIL Image object representing the input image  The algorithm:   * Saves the size of the image to variables * Checks the color mode of the image * Returns a new instance of empty PIL Image object with the size and color mode of the original image (for generating result image on a blank canvas, used mostly in the geometric functions * Returns the int value representing the number of color channels in original image * measure\_time   This function is used to measure execution time of other functions.  Patameters:  start: A flag for communicating function start (start = 1) and finish (start = 0)  The algorithm:   * If the flag is equal to 1, saves the current time using time library function to a global variable * If the flag is not equal to 0, prints the difference between start time (saved in a global variable) and current time (in seconds) | |
| **Description of implementation of basic image operations**   * Description (B1)   This function modifies the brightness of an image by adjusting the color channels based on a given factor for the image.  Parameters:  image: A PIL Image object representing the input image.  factor: An integer representing the factor by which to adjust the brightness. Should be in the range [-255, 255].  The algorithm:   * checks if the given factor is within the range [-255, 255]. If not, adjust it to the nearest boundary value within the range. * gets the width and height of the input image using the size attribute of the image. * analyzes the color channels of the input image (‘L’ or ‘RGB’ mode) * for each pixel, iterate over the color channels and calculate the new color value for each channel by adding the factor to the corresponding channel's value. * iterates over each pixel in the image using nested loops for the width and height of the image. * updates the pixel in the result image with the new color values and store it as a tuple. * Saves and shows the image   Example usage:  from PIL import Image  # Load the image  input\_image = Image.open("input\_image.bmp")  # Adjust brightness with a factor of 100  modify\_brightness(input\_image, 100)     * Description (B2)   This function modifies the contrast of an image by applying a contrast factor to each pixel in the image. The contrast factor controls the degree of contrast adjustment.  Parameters:  image: A PIL Image object representing the input image.  factor: A float representing the contrast factor. Should be in the range [-10, 10]. A higher positive value increases contrast, while a lower negative value decreases contrast.  The algorithm:   * checks if the given factor is within the range [-10, 10]. If not, adjust it to the nearest boundary value within the range. * creates a lookup table (lut) of 256 elements to store the transformed values for each intensity level. * iterates over each possible intensity level (0 to 255) and calculate the transformed value. * get the width and height of the input image using the size attribute of the image. * analyzes the color channels of the input image (‘L’ or ‘RGB’ mode) * iterates over each pixel in the image using nested loops for the width and height of the image. * for each pixel, iterates over the color channels and applies the contrast adjustment. * updates the pixel in the result image with the adjusted color values and store it as a tuple * saves the result image and shows it   Example usage:  from PIL import Image  # Load the image  input\_image = Image.open("input\_image.bmp")  # Adjust contrast with a factor of 7.0  modify\_contrast(input\_image, 7.0)     * Description (B3)   This function applies a negative effect to an image, inverting the colors of each pixel to their complementary values for the image.  The algoritm uses this algotihms: new=128+(i– 128)∙efactor  where:  new is the new value calculated by the function,  i is the input value,  factor is the given factor  Parameters  image: A PIL Image object representing the input image.  Algorithm:   * gets the width and height of the input image using the size attribute of the image. * analyzes if the channel is in mode ‘RGB’ or ‘L’ * iterates over each pixel in the image using nested loops for the width and height of the image. * for each pixel, iterate over the color channels and apply the negative effect by subtracting each color channel value from 255 to get the complementary color. * updates the pixel in the result image with the new color values and store it as a tuple. * Saves and shows the image   Example:  from PIL import Image  # Load the image  input\_image = Image.open("input\_image.bmp")  # Apply the negative effect to the image  apply\_negative(input\_image)     * Description (G1)   Horizontal Flip: Flips the image horizontally.  Efficiency:  Time: O(width \* height / 2), where width and height are the dimensions of the image. It iterates through half the width for each row.  Memory: O(width \* height) for the result image.  Mathematical description:  O(x,y)=I(w−x−1,y)  Let ‘I’ be the input image, ’O’ be the output image, ‘w’ be the width of the image, ‘h’ be the height of the image.  Parameters  image: A PIL Image object representing the input image.  Algorithm   * Gets the width and height of the input image using the size attribute of the image. * Analyzes the color channels of the input image. * Iterates over each pixel in the image, flipping horizontally by swapping the left and right pixels for each row. * Updates the pixel in the result image with the horizontally flipped pixel and store it.      * Description (G2)   This function vertically flips the input image, reflecting the image along the horizontal axis.  Mathematical description:  O(x,y)=I(x,h−y−1)  Let ‘I’ be the input image, ’O’ be the output image, ‘w’ be the width of the image, ‘h’ be the height of the image.  Parameters  image: A PIL Image object representing the input image.  Algorithm   * Get the width and height of the input image using the size attribute of the image. * Analyze the color channels of the input image. * Iterate over each pixel in the image, flipping vertically by swapping the top and bottom pixels for each column. * Update the pixel in the result image with the vertically flipped pixel and store it.   Time: O(width \* height / 2), where width and height are the dimensions of the image. It iterates through half the height for each column.  Memory: O(width \* height) for the result image.     * Description (G3)   Flips the image diagonally.  Mathematical description:  O(x,y)=I(w−x−1,h−y−1)  Let ‘I’ be the input image, ’O’ be the output image, ‘w’ be the width of the image, ‘h’ be the height of the image.  Efficiency:  Time: O(width \* height), where width and height are the dimensions of the image. It iterates through each pixel once, so it's efficient.  Memory: O(width \* height) for the result image.     * Description (G4)   Shrinks the image by a given factor.  Efficiency:  Time: O(width \* height / val^2), where width and height are the dimensions of the image and val is the shrinking factor. It iterates through a reduced number of pixels.  Memory: O(new\_width \* new\_height) for the result image, where new\_width and new\_height are calculated based on the shrinking factor.  Example: shrinking the image of the size 512x512px two times, we obtained 256x256px     * Description (G5)   Enlarges image by a given value.  Parameters  Image: The input image to be enlarged.  val: float; The scaling factor by which to enlarge the image. Should be greater than 0.  Time Complexity  The time complexity of this function is O(new\_width \* new\_height), where new\_width and new\_height are the new dimensions of the enlarged image.  Memory Efficiency  The memory complexity is O(1).  Example: By enlarging image 512x512px, we obtained an image 1024x1024px  Here, a short description of implementation of the aforementioned basic operations should be placed. Attention should be paid (if possible) to the efficiency of this implementation w. r. t. both the time and the memory usage (these aspects should be described). | |
| **Description of implementation of noise reduction methods**   * The first method: median filter   The algorithm calculates the median pixel values for each color channel by sorting and finding the median.  Mathematical algorithm:    Description: The median filter replaces each pixel's value with the median value of the pixels in its neighborhood, which is determined by the specified kernel size. The kernel is a square-shaped window that moves over the image, and for each pixel, it collects the pixel values within its neighborhood defined by the kernel\_size.  Parameters:  - image  - kernel\_size (int): Size of the square-shaped neighborhood for the median filter.  Efficiency considerations:  Time efficiency: The time complexity of this implementation is O(w \* h \* kernel\_size^2), where w is the image width, h is the image height.  Memory usage: The memory usage is O(kernel\_size^2), as we collect a sample array of size kernel\_size^2 for each pixel  Example:  Images without noise Image with impulse noise    Kernel size is equal to 3 Kernel size is equal to 5     * The second method   Geometric mean filter  The algorithm calculates the geometric mean of pixel values for each color channel within a specified neighborhood around each pixel in an input image.  Mathematical algorithm:    Parameters:   * Image * kernel\_size (int): The size of the square-shaped neighborhood used to calculate the geometric mean for each pixel.   Efficiency considerations:  Time efficiency: The time complexity is O(width \* height \* kernel\_size^2), This is because the filter processes each pixel in the image and, for each pixel, computes the geometric mean based on the values within the kernel.  Memory usage: The memory usage for the geometric mean filter is O(kernel\_size^2), as it collects a sample array of size kernel\_size^2 for each pixel.  Example:  Images without noise Image with uniform noise    Kernel size is equal to 3 Kernel size is equal to 5    A short description of implementation of the assigned (in the variant N) noise reduction methods should be placed in this section. Attention should be paid (if possible) to the efficiency of this implementation w. r. t. both the time and the memory usage (these aspects should be described). | |
| **Analysis of parameters of the noise reduction methods**  Results and conclusions  Original images    **Median filter**  For the low density noise   |  |  |  |  | | --- | --- | --- | --- | | Impulse 8-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  |  |  | | Normal 8-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  |  |  | | Uniform 8-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  |  | new_image | | Impulse 24-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  |  | new_image | | Normal 24-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  |  | new_image | | Uniform 24-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  |  | new_image |     For the high density noise:   |  |  |  |  | | --- | --- | --- | --- | | Impulse 8-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  |  | new_image | | Uniform 8-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  |  | new_image | | Impulse 24-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  |  | new_image | | Uniform 24-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  |  | new_image |   The conclusion is that the bigger kernel size is the lower accuracy of the removing noise is.  **Geometric mean filter**  For the low density noise:   |  |  |  |  | | --- | --- | --- | --- | | Impulse 8-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  |  |  | | Uniform 8-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  |  | new_image | | Impulse 24-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  | new_image | new_image | | Uniform 24-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  | new_image | new_image |   For the high density noise:   |  |  |  |  | | --- | --- | --- | --- | | Impulse 8-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  |  | new_image | | Uniform 8-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  |  |  | new_image | | Impulse 24-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  | new_image | new_image | new_image | | Uniform 24-bit | Denoised (kernel size = 3) | Denoised (kernel size = 5) | Denoised (kernel size = 7) | |  | new_image | new_image | new_image |   Short conclusion  For both filters, a larger kernel size results in more smoothing but may lead to some loss of image details.  The results of the experiments related to changes in values of the parameters (if there are any) of the assigned noise reduction methods should be placed here. In this section the conclusions drawn from the experiments should be also placed. | |
| **Analysis of the noise reduction methods w. r. t. the possible applications for various types of noise**  Results and conclusions  Comparison between the median filter and geometric mean filter for low intensity noise   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | original | Impulse 8-bit | Denoised with median filter (kernel size = 3) | Denoised with geometric mean filter (kernel size = 3) | |  |  |  |  |  | | mse | 0 | 156.66 | 16.10 | 751.44 | | pmse | 0 | 0.00243 | 0.00025 | 0.01155 | | snr | 0 | 20.523 | 30.38 | 13.69 | | psnr | 0 | 26.18 | 36.06 | 19.37 | | md | 0 | 231 | 67 | 231 | |  | original | Uniform 8-bit | Denoised with median filter (kernel size = 3) | Denoised with geometric mean filter (kernel size = 3) | |  |  |  |  |  | | mse | 0 | 167.46 | 22.9 | 78.55 | | pmse | 0 | 0.00261 | 0.00035 | 0.00120 | | snr | 0 | 20.21 | 28.85 | 23.50 | | psnr | 0 | 25.89 | 22.9 | 29.18 | | md | 0 | 51 | 85 | 158 | |  | original | Impulse 24-bit | Denoised with median filter (kernel size = 3) | Denoised with geometric mean filter (kernel size = 3) | |  |  |  |  |  | | mse | 0 | [143.30, 155.05, 168.42] | [18.28, 18.18, 18.66] | [ 48.76, 137.68, 131.76] | | pmse | 0 | [0.0022, 0.0024, 0.0026] | [0.0003, 0.0003, 0.0003] | [0.0007 0.0021 0.0020] | | snr | 0 | [23.86, 19.14, 18.68] | [32.81, 28.39, 28.19] | [28.55 19.60 19.70] | | psnr | 0 | [26.57, 26.23, 25.87] | [35.51, 35.53, 35.42] | [31.25, 26.74, 26.93] | | md | 0 | [64, 64, 64] | [68, 69, 74] | [115, 196, 198] | |  | original | Uniform 24-bit | Denoised with median filter (kernel size = 3) | Denoised with geometric mean filter (kernel size = 3) | |  |  |  |  |  | | mse | 0 | [227.82, 234.44, 245.67] | [28.43, 28.52, 28.12] | [56.19, 143.28, 67.91] | | pmse | 0 | [0.0035, 0.0036, 0.0038 ] | [0.0004, 0.0004, 0.0004] | [0.0009, 0.0022, 0.0010] | | snr | 0 | [21.85, 17.29, 17.00] | [30.89 26.44 26.40] | [27.93, 19.43, 22.58] | | psnr | 0 | [24.55, 24.43, 24.23] | [33.59, 33.58, 33.64] | [30.63, 26.57, 29.81] | | md | 0 | [50, 50, 50] | [78, 75, 79] | [72, 191, 177] |   For high density noise   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | original | Impulse 8-bit | Denoised with median filter (kernel size = 3) | Denoised with geometric mean filter (kernel size = 3) | |  |  |  |  | new_image | | mse | 0 | 1606.73 | 23.54 | 5813.30 | | pmse | 0 | 0.025 | 0.00036 | 0.08940 | | snr | 0 | 10.39 | 28.72 | 4.81 | | psnr | 0 | 16.07 | 34.41 | 10.49 | | md | 0 | 234 | 171 | 234 | |  | original | Uniform 8-bit | Denoised with median filter (kernel size = 3) | Denoised with geometric mean filter (kernel size = 3) | |  |  |  |  |  | | mse | 0 | 1243.68 | 62.67 | 800.80 | | pmse | 0 | 0.019 | 0.00096 | 0.01231 | | snr | 0 | 11.50 | 24.48 | 13.42 | | psnr | 0 | 17.18 | 30.16 | 19.09 | | md | 0 | 102 | 94 | 206 | |  | original | Impulse 24-bit | Denoised with median filter (kernel size = 3) | Denoised with geometric mean filter (kernel size = 3) | |  |  |  |  | new_image | | mse | 0 | [810.04, 865.98, 924.49] | [20.34 20.6 21.28] | [966.14, 1755.64, 2278.10] | | pmse | 0 | [0.0125, 0.0133, 0.0142] | [0.0003, 0.0003, 0.0003] | [0.0149, 0.0270, 0.0350] | | snr | 0 | [16.34 11.61 11.24] | [32.34 27.85 27.62] | [15.58, 8.54, 7.32] | | psnr | 0 | [19.06, 18.76, 18.47] | [35.06, 35.00, 34.85] | [18.28, 15.69, 14.56] | | md | 0 | [144, 144, 144] | [141, 147, 143] | [255, 218, 220] | |  | original | Uniform 24-bit | Denoised with median filter (kernel size = 3) | Denoised with geometric mean filter (kernel size = 3) | |  |  |  |  | new_image | | mse | 0 | [1605.79, 1732.62, 1877.19] | [50.54, 53.89, 50.73] | [1314.62, 2320.43, 3091.97] | | pmse | 0 | [0.0247, 0.0266, 0.0289] | [0.0008, 0.0008, 0.0008] | [0.0202, 0.0357, 0.0476] | | snr | 0 | [13.37, 8.60, 8.16] | [28.39 23.68 23.84] | [14.24, 7.33, 6.00] | | psnr | 0 | [16.07, 15.74, 15.40] | [31.09, 30.82, 31.08] | [16.94, 14.46, 13.23] | | md | 0 | [152, 152, 152] | [133, 142, 130] | [250, 213, 209] |   Short conclusion  The median filter is more effective at reducing salt-and-pepper noise than geometric mean filter.  Objective (coefficients E) and subjective conclusions related to the quality of the results obtained by the assigned noise reduction methods for different types of noise should be placed here. The images provided on the web page should be applied for this purpose. | |
| **Teacher's remarks**  This is a section for teacher's remarks for the laboratory group (please leave some free place). | |