CMPT-365 E100

Project 2- Report

Author: **Vansh Bhatt**
Student #: **301471598**

In this project, we created a Python application named Mini-Photoshop that allows users to perform core image processing operations using a Graphical User Interface (GUI) built with Tkinter. The program can open BMP files and apply various manipulations like grayscale conversion, ordered dithering, etc.

1. Imports and Global Variables

- We start the project by importing the necessary libraries for the functioning of our application. We use Tkinter to create the graphical user interface (GUI), with widgets like buttons, labels, and drop downs. We also import filedialog to help us open the bmp images and ttk to help us make the dropdown menus.

- In addition to that we use Python Imaging Library (PIL) to handle operations like open image, copy image, make image back from array etc. Lastly we also use Numpy which offers efficient handling of pixel data as arrays and makes the mathematical operations quicker.

- We also allot several global variables to keep track of original image and the new modified images which allows us to perform any function randomly without any predefined order and allow state management across functions.

2. Opening and Displaying Original BMP Image

- We use askopenfilename() from the filedialog module, which opens a file selection window restricted to .bmp files. This ensures that users can only select files of the required format. If a valid .bmp file is selected, the Image.open() method from PIL is used to load the image.We also make copies of the original image which can then further be used during different operations in our application.

- Then, since Tkinter cannot directly display images in the format returned by PIL, the image is converted into a Tkinter-compatible format using ImageTk.PhotoImage() method. Once converted, the image is further displayed within the container.

- Then, In the case of Tkinter, we keep a reference to an image object using container.image=photo, to prevent the image from being garbage-collected while it's still being displayed, which would cause it to disappear.

3. Helper and Utility Functions

A.) Display Side by Side [displayside(img1,img2)]

Many of our operations require a comparison between the original image and the newly created image. Thus we created a helper function that helps us position the two images side by side for a clear comparison.

Image.new() method creates a new blank image with a specified RGB mode, size, and color. The width of the new image is the sum of the widths of image 1 and image 2 (because they are being placed side by side).The height is the maximum height between image 1 and image 2. This ensures that even if the two images are of different heights, the new image will have enough space to fit both vertically.

We then use the paste() method places one image onto another at a specific position. Here, image 1 is pasted starting at coordinates (0, 0), which is the top-left corner of the new image canvas and the second image will start immediately to the right of the first image, since we position it horizontally based on the width of image 1. Both images will also align at the top even if their heights are different.

Then, since Tkinter cannot directly display images in the format returned by PIL, the image is converted into a Tkinter-compatible format using ImageTk.PhotoImage() method. Once converted, the image is further displayed within the container.Lastly, we keep a reference to an image object using container.image=photo, to prevent the image from being garbage-collected while it's still being displayed, which would cause it to disappear.

B.) Display [display(image)]

For the display single image function, we just take in the image passed as the function parameter and display it on our window.

Since, Tkinter cannot directly display images in the format returned by PIL, the image is converted into a Tkinter-compatible format using ImageTk.PhotoImage() method. Once converted, the image is further displayed within the container.Lastly, we keep a reference to an image object using container.image=photo, to prevent the image from being garbage-collected while it's still being displayed, which would cause it to disappear.

C.) Operation Selection Functions [core(), optional()]

We have 2 different functions here, one for our core functionalities and the other for out optional functionalities. The core() function retrieves the selected operation from the core_combo dropdown menu, which includes a set of essential operations and then match that selected operation to call our designed function. While the optional function retrieves the selected operation from the optional_combo dropdown menu and then match that selected operation to call our designed function.

4. Core Functionalities

A.) Grayscale Conversion [grayscale(toshow)]

This function converts the currently loaded image to grayscale. It starts by accessing the RGB channels of each pixel in the original image and converts the image into a numpy array.By representing the image in this format, the pixel values for each color channel (Red, Green, and Blue) become easily accessible. We use data type as float32 for accurate representation.

Then the Red channel values are accessed using R = pixels[..., 0], where 0 indexes the first element in the three-component pixel array we created, representing red intensity. Similarly we do it for green and blue.

Further, we use the formula taught in class to get the luminance(Y) value which will be used to make the greyscale image. $Y' = (0.299 \times R) + (0.587 \times G) + (0.114 \times B)$.
This results in a single luminance per pixel, which is then cast to an unsigned 8-bit integer for display as an image.

Lastly the line image.fromarray(y) is used to create a grayscale image from the computed luminance values stored in the array y and then we display this new image along with our original image using our displayside function.
(*toshow is used as boolean which is passed as True when we want to see the greyscale image and false if we are just converting the image to greyscale for some further operation as dithering.)

B.) Ordered Dithering [orddither()]

This function starts by converting the image to grayscale image. Then it takes that image and converts the image into a numpy array. We also take in the 4x4 Bayer matrix as taught in class and transform it into an numpy array for efficient calculations.

We then go ahead and normalize the Bayer matrix by dividing it by 16 and then multiplying it by 255 transforming the normalized values into the 8-bit grayscale range. We then retrieve the dimensions using .shape with the first dimension indicating height (number of rows) and the second dimension indicating width (number of columns) and then initialize an array of zeros for those dimensions.

We then run the algorithm taught in class to further compare our image intensity value to the bayer matrix value. And in order to figure out which value we compare it to, we perform the modulus operation on the row and column numbers of our image matrix to get something within the bayer matrix dimension range.

Lastly if the pixel intensity is greater than the threshold, the pixel is set to **255** (white) in the output array else set to 0 (black). Finally we use image.fromarray to convert the newly calculated array of 0's and 255 to an image and display that image side by side with the grayscale image.

C.) Auto-Leveling [autolev()]

This technique involves adjusting the distribution of grayscale levels in the image so that they are more uniformly spread across the intensities. We follow the idea presented to us in the slides.Here we start by converting the original colored image into a numpy array and get the total number of pixels in the image using .size .

We then initialize an array of zeros for the 256 different values. The loop iterates through each pixel intensity using pixels.flat, a flattened 1D view of the pixel array. For each pixel's intensity, the corresponding index in Count is incremented, providing a complete count of number of pixels at each level from 0 to 255. Then we determine the ideal number of pixels it should be for each intensity i.e total//256.

Then we move ahead to initialize an array of zeros called stretch, where each index will be mapped to a new level based on the pixel distribution. Current level track the current intensity and if the count for that intensity is lower than the ideal count then we increase the intensity by 1 to push it to the next level.

Lastly each pixel's intensity in pixels is replaced by its corresponding mapped value from stretch, resulting in an adjusted array, newimg, with redistributed intensity values. We then make the image from that array using .fromarray and display the new auto levelled image along with the original.

5. Optional Functionalities

A.) Inverting Colors [invert()]

Inverting colors transforms an image into its negative by subtracting each pixel value from 255.

We start by taking the original colored image and transferring it into an array of pixel values. We then loop through this array to create the new image array which is the 255-current pixel intensity. Lastly we create an image out of our new inverted image array and display the new inverted image on screen. It makes the dark areas light and the light areas dark.

B.) Increase Brightness [brighten()]

The brightness adjustment function allows users to increase the brightness incrementally by scaling the pixel values with a factor. The scaling factor starts at 1.0 and increases by 10% with each adjustment.

We start by converting the image into an array of pixels using numpy. The brightness factor is multiplied by 1.1, effectively increasing it by 10% with each function call. This approach allows for incremental brightness adjustments, making the image progressively brighter each time the function is used.

Each pixel's intensity is multiplied by the updated brightness_factor. This operation increases the intensity of each pixel, making the image brighter.

Lastly, numpy.clip function limits values in bright_pixels to the range of 0 (black) to 255 (white). Any values below 0 are set to 0, and values above 255 are set to 255. We then convert the array into an image using Image.fromarray, before displaying the new brightened image.

C.) Rotating the Image [rotate()]

The rotation function rotates the image by 90 degrees in a clockwise direction.

We reference rotatedimg globally, so that the function remembers the image's most recent rotation. This allows for cumulative 90-degree rotations each time the function is executed.

The .rotate() function accepts the angle in degrees, where a -90 degree value performs a clockwise rotation and the expand enabled will make sure no parts of the image are cut off during rotation. Lastly we display the new rotated image replacing the previous image.

D.) Resizing the Image [resize()]

This function resizes the loaded image to a smaller fixed size (200x200 pixels).

We use the .Resize() function with the size passed as parameters to resize the image to the new size. We then finally display this new resized image on our interface window.

6. GUI Design

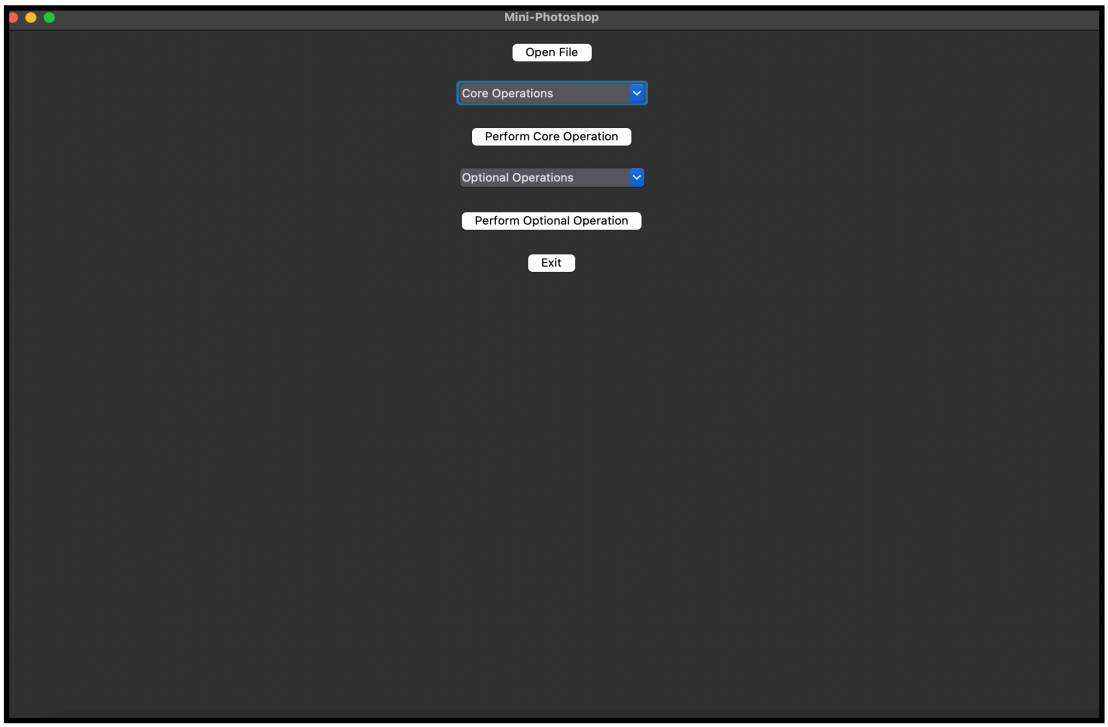
- The main application window (root) is created with Tkinter, titled "Mini-Photoshop" and sized at 1200x800 pixels.
- Openbutton is a button labeled "Open File," used for selecting a BMP file, linked to our open function.
- Coredrop is a dropdown menu using ttk from tkinter, for selecting core image operations: "Grayscale," "Ordered Dithering," and "Auto Level"; it defaults to "Core Operations" and is set to read-only.
- Corebutton is a button labeled "Perform Core Operation," which triggers the appropriate core function to apply the selected core operation.
- Optionaldrop is a second dropdown menu, providing optional operations: "Invert Colors," "Rotate 90 Degrees," "Resize Image," and "Adjust Brightness"; it defaults to "Optional Operations" and is also read-only.
- Optionalbutton is a button labeled "Perform Optional Operation," which triggers the appropriate optional function to apply the selected optional operation.
- Exitbutton allows users to close the application by calling root.quit.
- Container is a tkinter.Label widget designed to display the image, set to expand automatically to fit available space.
- Root.mainloop() starts the Tkinter event loop, keeping the application responsive and waiting for user interactions.

INTERESTING FINDINGS:

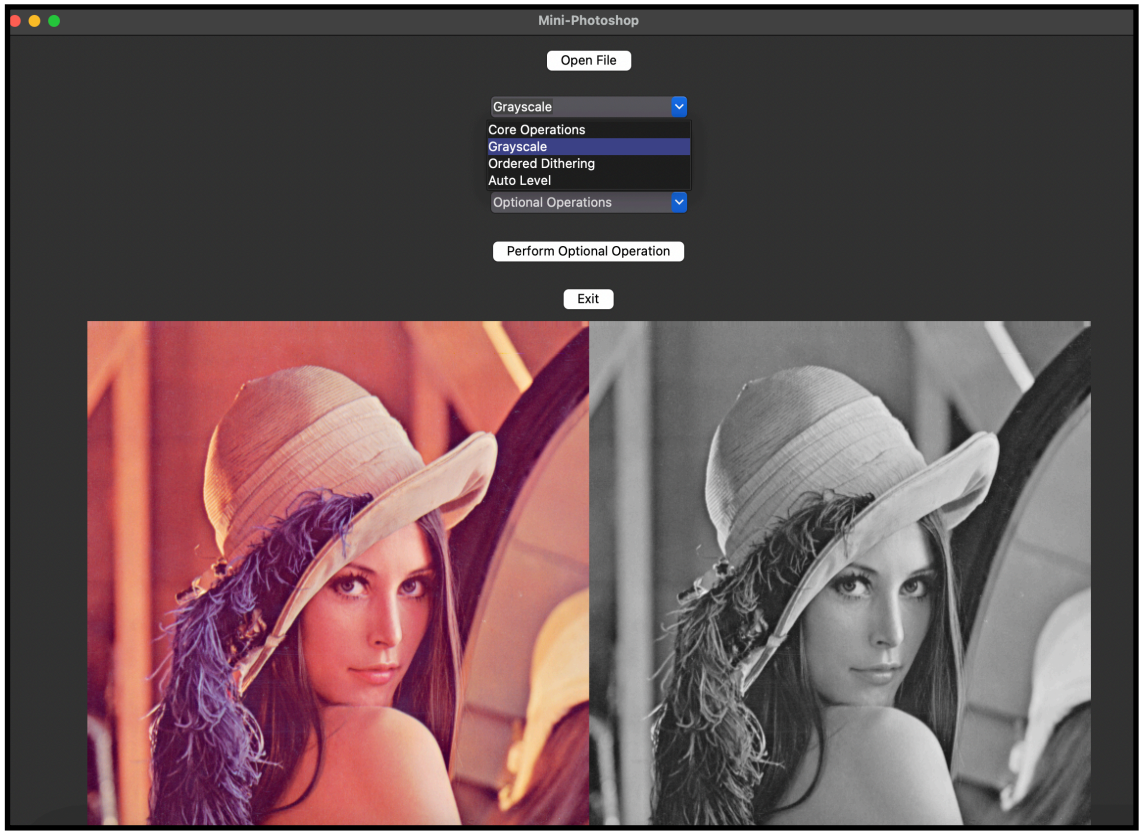
Auto-level adjustments technique tries to ensure that all intensity levels, from dark to bright, are represented, which can make details more visible in underexposed or low-contrast images. However, it doesn't always give an enhanced visual result as in our case.This is because the technique currently applies the same transformation to all images, regardless of their unique pixel distributions. Enhancing auto-level could make the results more appealing and effective for a broader range of photos.

Adjusting brightness by multiplying pixel values with a factor introduces subtle, incremental changes. This implementation's current approach—raising brightness by 10% per click—enables users to control the adjustment gradually.However, over-amplification can lead to "clipping," where pixel values max out at 255, causing a loss of detail. This incremental method could be improved by providing both positive and negative options, allowing users to darken images as well.

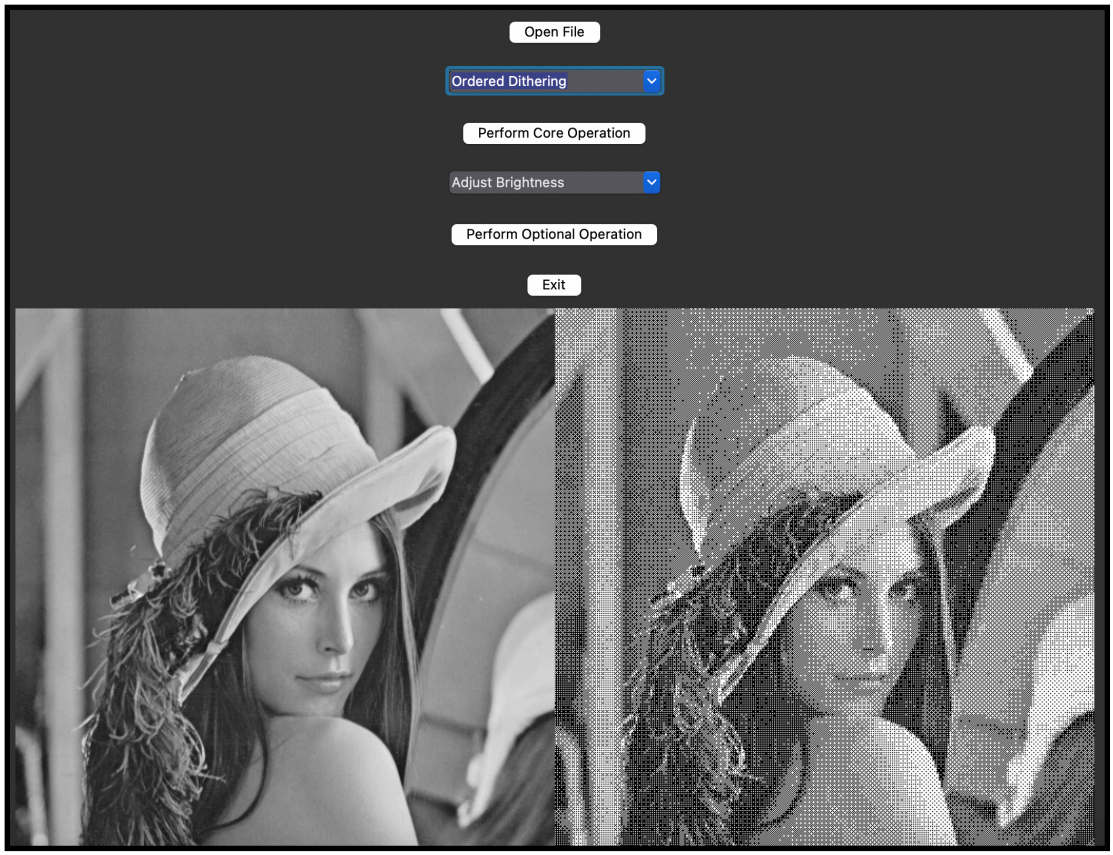
By comparing pixel intensities against a threshold grid, ordered dithering creates a visually interesting effect, making it particularly useful for low-color screens or retro aesthetics. Its simplicity yet effectiveness displays how early graphic processing techniques could produce visually appealing results even with limited resources.



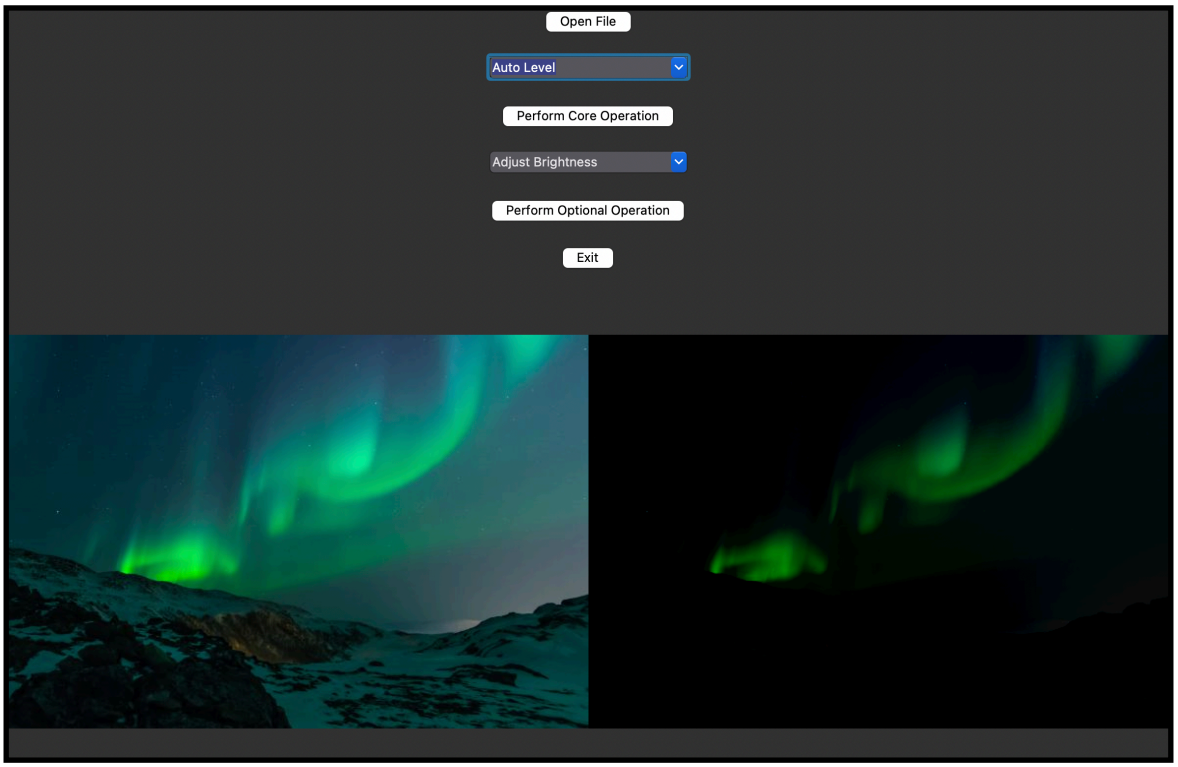
GUI



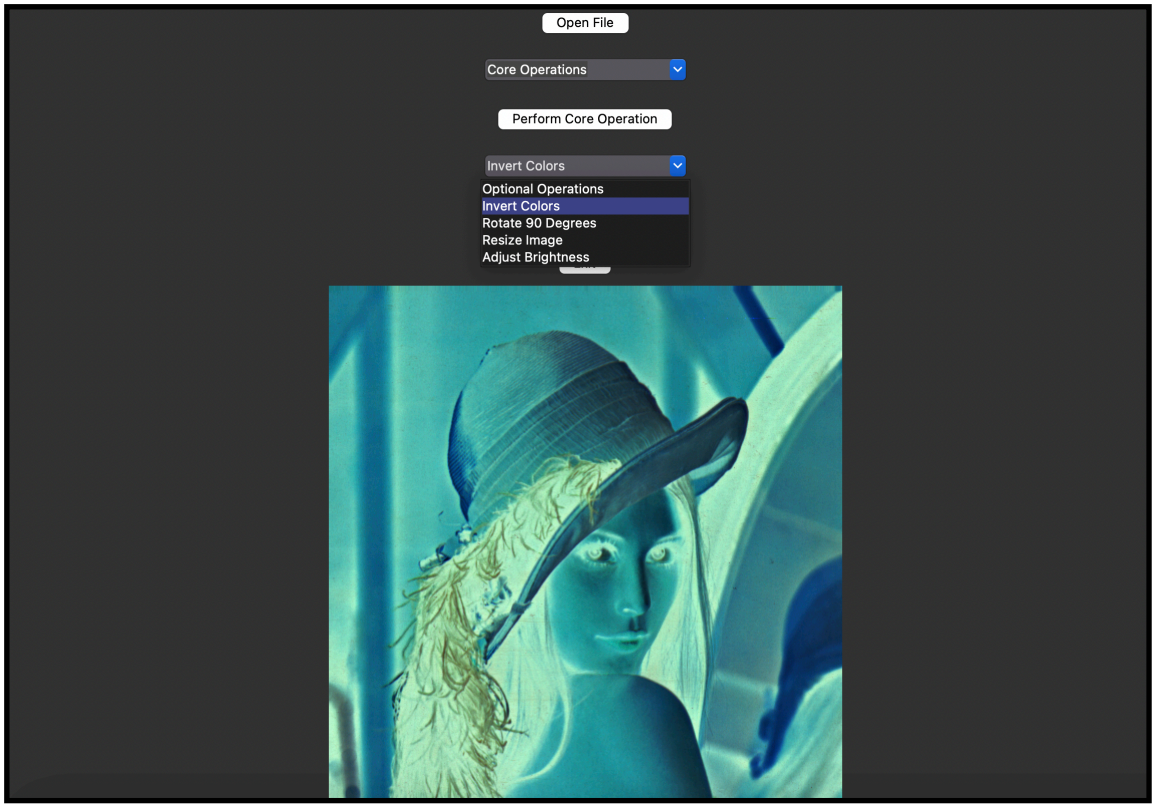
Grayscale



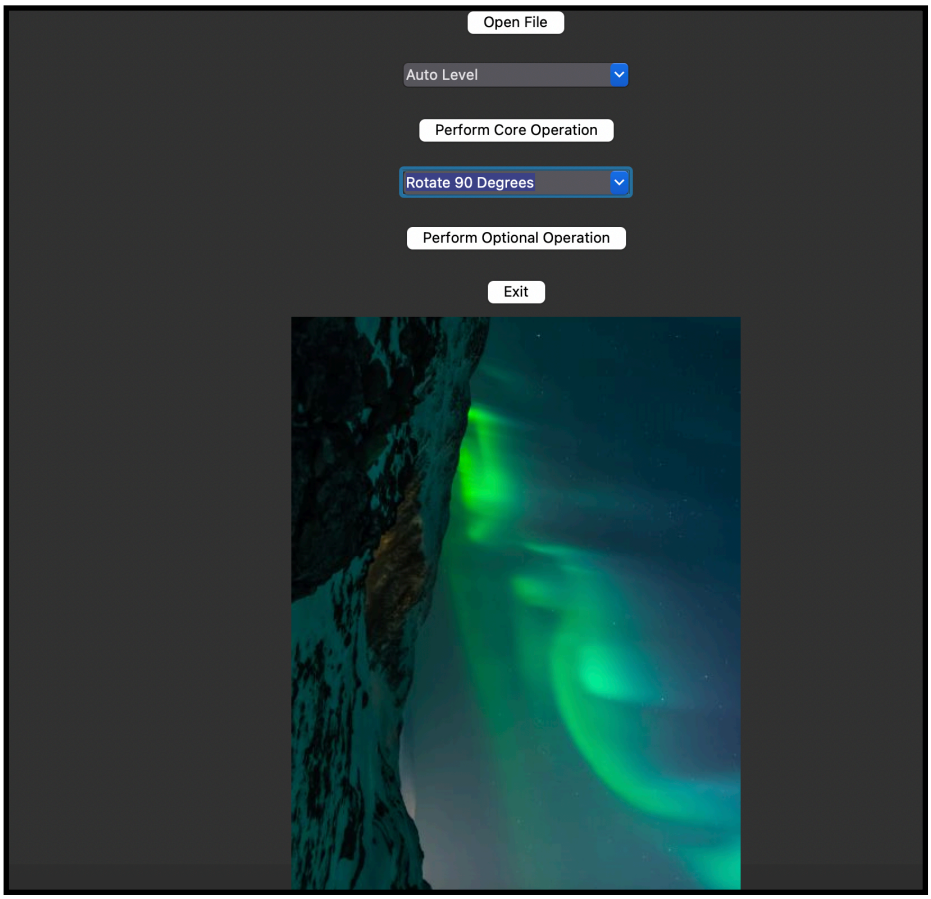
Ordered dithering



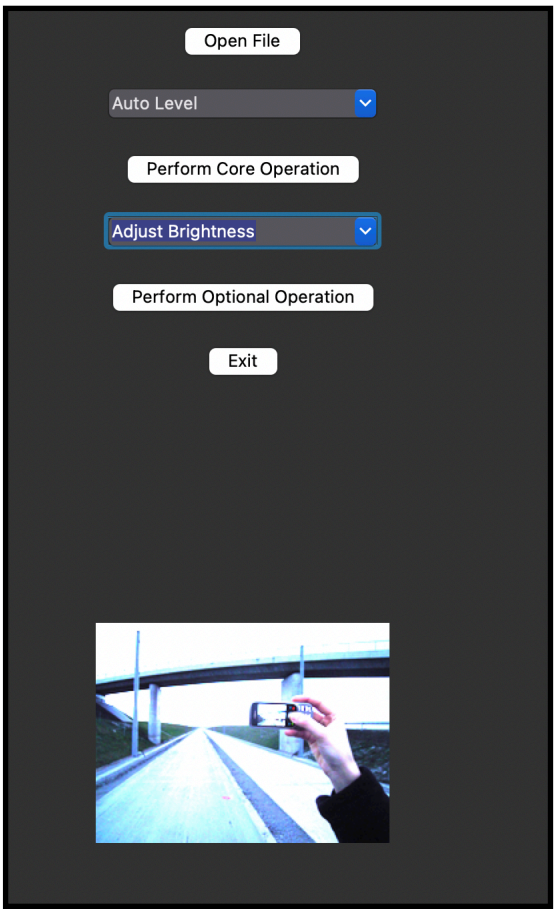
Auto level



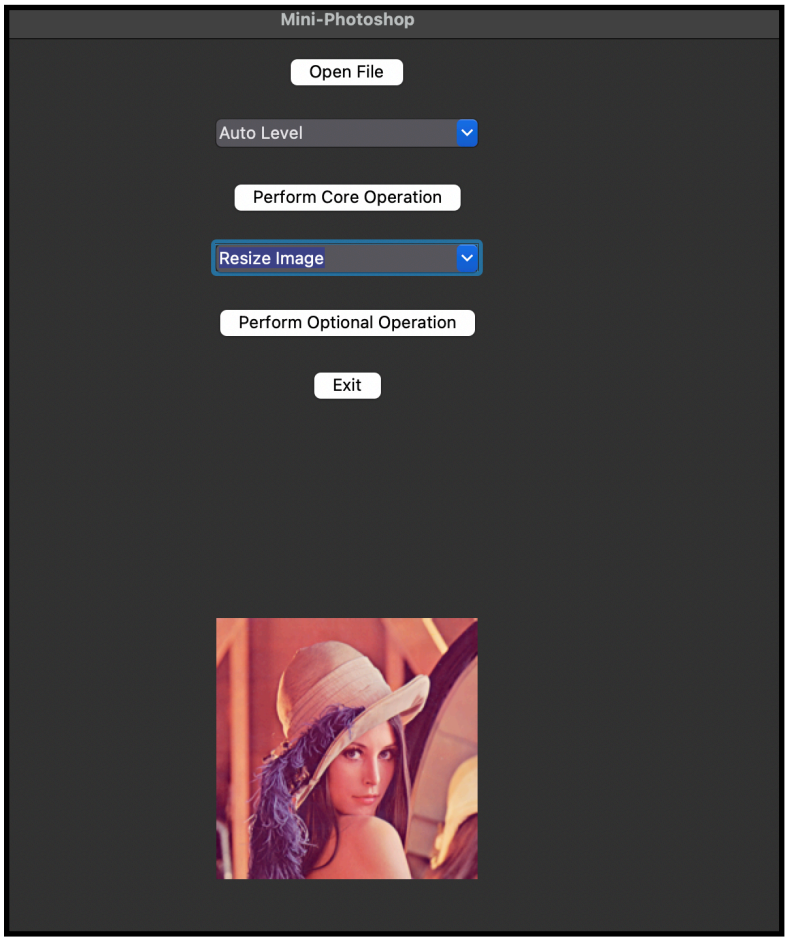
Inverse



Rotate



Adjust Brightness



Resize