Project 8 Image Processing

The goal of the program is to read a image and store it in the self-made data structure(class), and then

1. Count the intensity histogram.
2. Compute Fast Fourier Transform(FFT) of the image.
3. Extract the contour of the image.

Project Introduction

The structure of the project is like this:

src: Folder of the src program, including:

1. "pcsc.hh": Declaration of classes and functions.
2. "pcsc.cc": Implementation of classes and functions.
3. "main.cc": Main program.

image: Folder to store the input and output images.

test: Folder of the test program, including:

1. test.cc: Test correctness of the program with 6 cases.

CMakeLists.txt: Text to compile the whole program.

Flow of the Project

To implement our goals, I declared two class: "complex" and "Image2D". Class "complex" is a class to store complex numbers, it is utilized when calculating FFT. To simplify the program of complex number calculation, I overloaded operator of "=", "==", "+", "-", "*", "/" respectively. Class "Image2D" is built to store the loaded images and the FFT of the image. Details of the members and methods introduction are in the program doxygen.

When Constructing an "Image2D" object, we need to write the path of the image. I used OpenCV to load the image with version 3.2.0, and then transform it to a 2-dim array and store into the "ppimage", and record the number of rows and columns into member rows, cols, respectively. The FFT of the image is square, and the constructor calculate its side length which is equals to the largest number between rows and cols and then allocate memory of the "FFTImage" dynamically. I chose the picture of Lebron James, my favourite basketball player, to be the input image showed in Figure 1.

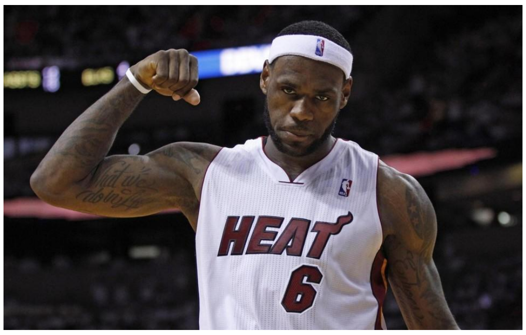


Figure 1: Input image

The method "ComputeHistogram" is to count the histogram of the "ppimage". In this program we traverse every image pixel and count the total number of every graylevel, and then draw the histogram through black rectangles. The histogram of the input image is showed in Figure 2.

To compute the FFT of the image, we can call the "FFT2D" method. It will calculate the 2-dimensional FFT and store the result into member "FFTImg". 2-dim FFT can be implemented through twice 1-dim FFT to rows and columns respectively. I will not write the FFT algorithm in detail in the report since this algorithm is really complicated and it is not the focus of the project.

To Output the image of the FFT, we can call the "OutputFFTImage" method, which first call the "FFT2D" to get the complex FFT, and transform the "FFTImg" to cv::Mat type, after Calculating the log of the magnitude of each complex "FFTimg" pixel, we need to split the image to 4 parts through the center of the FFT image. Then we exchange the subimage between 1st and 3rd quadrant, and also exchange the subimage between 2rd and 4th quardrant.

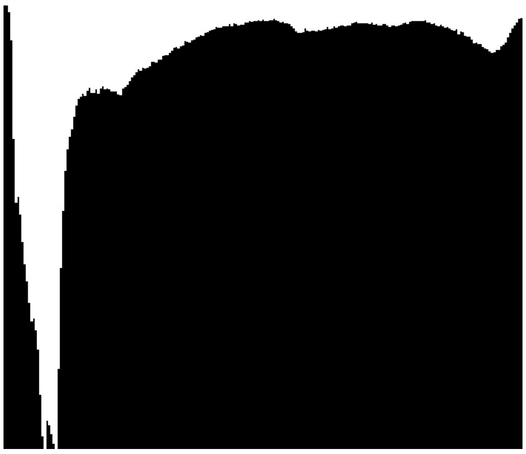


Figure 2: Input image

the result of 2-dim FFT is showed in Figure 3. The distance to the center means the frequency of the image. The brighter the pixel, the higher the amplitude of the particular frequency. Through Figure 3 we can see that the majority of the image is with low frequency.

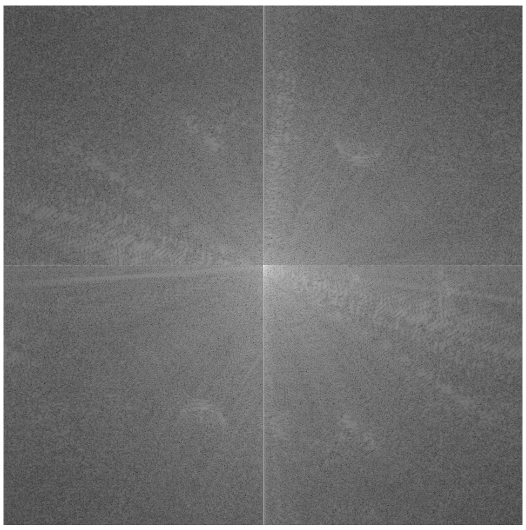


Figure 3: Input image

We can call method "ContourExtration" to extract the contour of the image. I used Laplacian to be the operator which is stored in float array "w". You can edit the array "w" to change the operator. We then use the operator to calculate the convolution of the image and store the result to the cv::Mat variable and return it. The extracted contour of the image is showed in Figure 4.

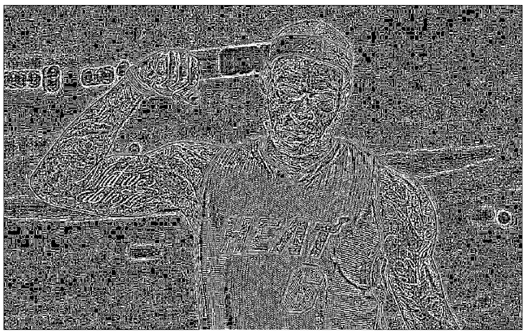


Figure 4: Input image

Project compilation

We use "CMakeLists.txt" to help compile the project. Since I used OpenCV with version 3.2.0 to read and write the image, we need to add "find_package(OpenCV 3 REQUIRED)" in the text.

At "src" part, we will generate the executable file "main", so we need to add "add_executable(main src/main.cc)" in the text and we need to link the pcsc.cc and ${OpenCV_LIBS} because both of them are included in the main.cc so we need to add "target_link_libraries(main ${OpenCV_LIBS})" and "target_link_libraries(main pcsc)" in the text.

At "test" part, we will generate the executable file "test_pcsc" so we need to add "add_executable(test_pcsc test/test.cc)" in the text. We also need to include OpenCV which will be mentioned later, so we also need to add "target_link_libraries(test_pcsc ${OpenCV_LIBS})" and we will use googletest to test the project and pcsc.cc is also needed to include so we need to add "target_link_libraries(test_pcsc gtest_main gtest pthread pcsc)" in the text.

The detail of the compilation can be seen in "CMakeLists.txt".

Test of the project

We test two functions of the project through program "test/test.cc". First, we test if the class "Image2D" can correctly store the loaded image. I made the "CheckStore" function to compare each pixel of image loaded by "cv::imread" method to the transformed array "ppimage". the bool output is true if and only if all of the difference is no larger than 1e-5. Except "james.jpg", I also used two other images named "frog.jpg" and "earth.jpg" respectively. Both of them are stored in folder "image". Both of them are showed in Figure 5. Therefore there are three test cases to test the correctness of the class storage.

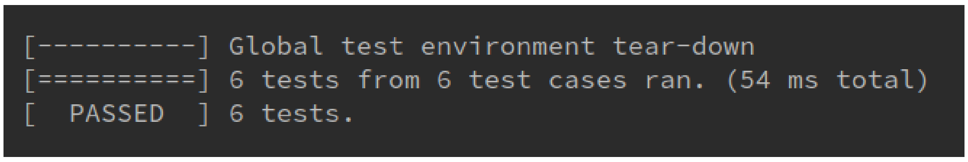
(a) frog.jpg

(b) frog.jpg

Figure 5: Two test images

Second, we test if all the overloading operators of class "complex" are executed correctly. We test four operators, "+", "-", "*", "/" respectively. I created three cases. For each case, I construct two "complex" objects "a" and "b" with different values, and then run "+", "-", "*", "/" and compare the result to the true result. For example, if a denotes "4+2i" and b denotes"3+4i", we will expect the result of "a+b" should be "7+6i".

Therefore, we have 6 test cases for the project, 3 for testing correctness of class "Image2D" storage and 3 for testing correctness of overloading operators of class "complex". All of the 6 test cases returned the expected values shown in Figure 6. Details of the test cases can be seen in program "test/test.cc".



```
[----------] Global test environment tear-down
[==========] 6 tests from 6 test cases ran. (54 ms total)
[  PASSED  ] 6 tests.
```

Figure 6: Test result

Limitations and Problems

1. I only implemented the project with 1-channel input image, I can expand the image with 3 channels, e.g. RGB image. It can be implemented by spliting the three channels, processing each channel respectively, and then merge the three channels.
2. I don't implement the optional task: filtering image by using Fourier transform.
3. I can use Template to make the whole program more generic