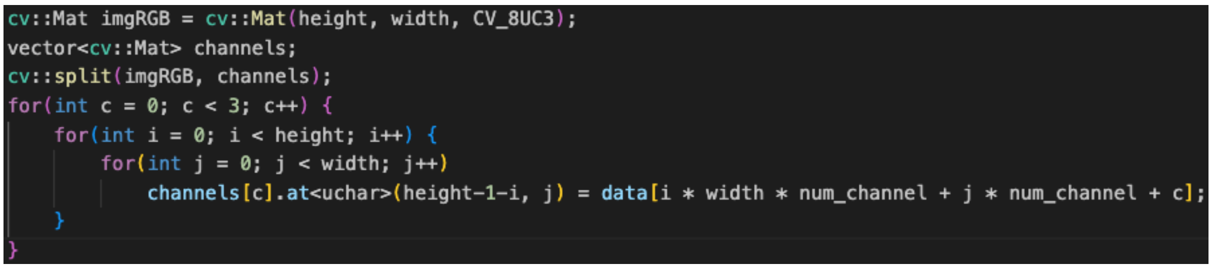
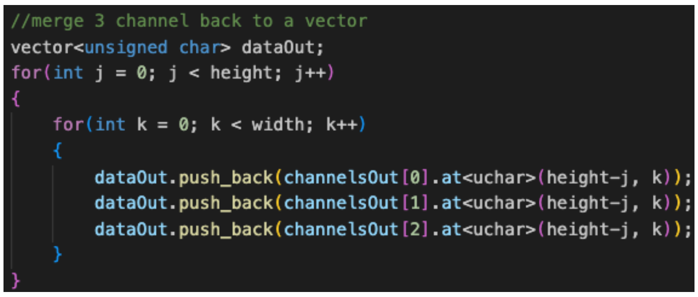
Digital Image Processing: Homework 4 Report

Implementation

First, I read and write the .bmp file by myself (the methods in hw1,2,3), not cv::imread and cv::imwrite. And for further calculation, I convert it to cv::Mat after reading the .bmp file.

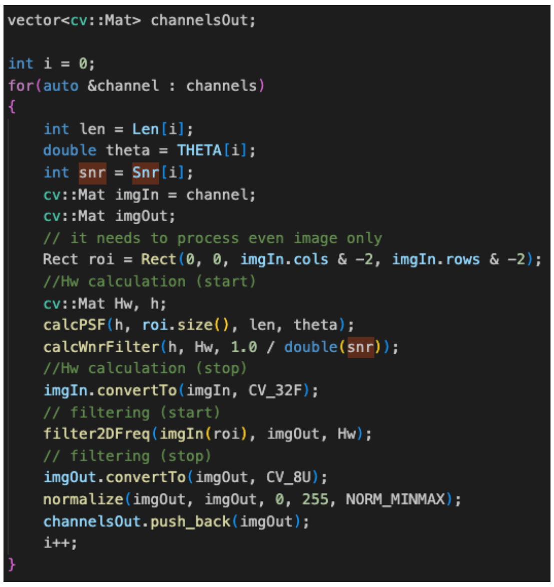


```cpp
cv::Mat imgRGB = cv::Mat(height, width, CV_8UC3);
vector<cv::Mat> channels;
cv::split(imgRGB, channels);
for(int c = 0; c < 3; c++) {
    for(int i = 0; i < height; i++) {
        for(int j = 0; j < width; j++)
            channels[c].at<uchar>(height-1-i, j) = data[i * width * num_channel + j * num_channel + c];
    }
}
```

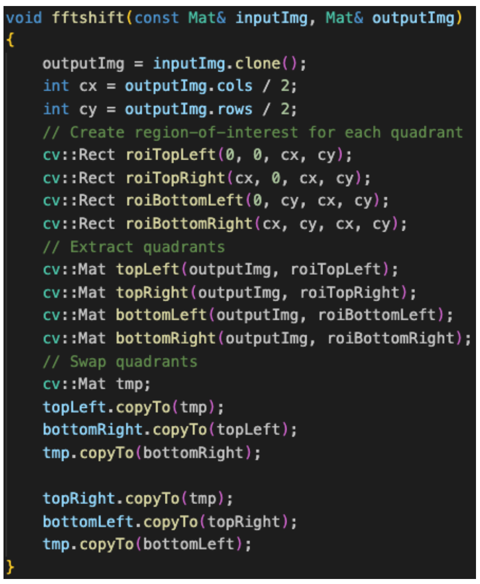


```cpp
//merge 3 channel back to a vector
vector<unsigned char> dataOut;
for(int j = 0; j < height; j++)
{
    for(int k = 0; k < width; k++)
    {
        dataOut.push_back(channelsOut[0].at<uchar>(height-j, k));
        dataOut.push_back(channelsOut[1].at<uchar>(height-j, k));
        dataOut.push_back(channelsOut[2].at<uchar>(height-j, k));
    }
}
```

I do the wiener deconvolution for each channel separately, while I also test transform the image to grayscale and calculate the wiener filter once. The results of these 2 circumstances may not differ much. I compute PSF first, and then compute the Wiener filter based on the PSF and a specific signal to noise ratio. Then, converts the image to 32-bit floating-point format, applies frequency domain filtering using the Wiener filter, and finally converts the result back to 8-bit unsigned integer format.

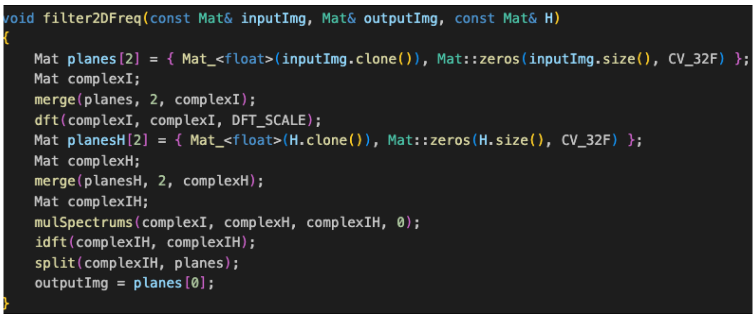


```cpp
vector<cv::Mat> channelsOut;

int i = 0;
for(auto &channel : channels)
{
    int len = Len[i];
    double theta = THETA[i];
    int snr = Snr[i];
    cv::Mat imgIn = channel;
    cv::Mat imgOut;
    // it needs to process even image only
    Rect roi = Rect(0, 0, imgIn.cols & -2, imgIn.rows & -2);
    //Hw calculation (start)
    cv::Mat Hw, h;
    calcPSF(h, roi.size(), len, theta);
    calcWnrFilter(h, Hw, 1.0 / double(snr));
    //Hw calculation (stop)
    imgIn.convertTo(imgIn, CV_32F);
    // filtering (start)
    filter2DFreq(imgIn(roi), imgOut, Hw);
    // filtering (stop)
    imgOut.convertTo(imgOut, CV_8U);
    normalize(imgOut, imgOut, 0, 255, NORM_MINMAX);
    channelsOut.push_back(imgOut);
    i++;
}
```

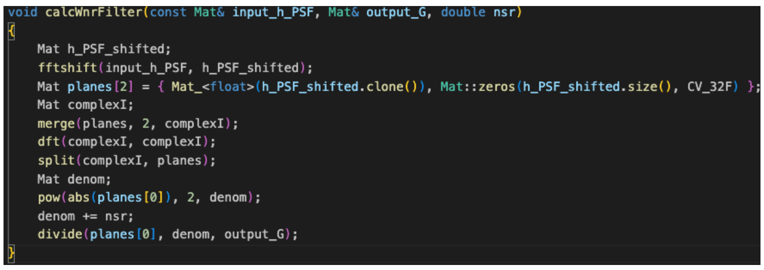


```cpp
void fftshift(const Mat& inputImg, Mat& outputImg)
{
    outputImg = inputImg.clone();
    int cx = outputImg.cols / 2;
    int cy = outputImg.rows / 2;
    // Create region-of-interest for each quadrant
    cv::Rect roiTopLeft(0, 0, cx, cy);
    cv::Rect roiTopRight(cx, 0, cx, cy);
    cv::Rect roiBottomLeft(0, cy, cx, cy);
    cv::Rect roiBottomRight(cx, cy, cx, cy);
    // Extract quadrants
    cv::Mat topLeft(outputImg, roiTopLeft);
    cv::Mat topRight(outputImg, roiTopRight);
    cv::Mat bottomLeft(outputImg, roiBottomLeft);
    cv::Mat bottomRight(outputImg, roiBottomRight);
    // Swap quadrants
    cv::Mat tmp;
    topLeft.copyTo(tmp);
    bottomRight.copyTo(topLeft);
    tmp.copyTo(bottomRight);

    topRight.copyTo(tmp);
    bottomLeft.copyTo(topRight);
    tmp.copyTo(bottomLeft);
}
```

I stored the restored result of each channel at *chennelOut*. Note that I've shifted the FFT to ensure consistency in interpreting frequency components and performing operations in the frequency domain.

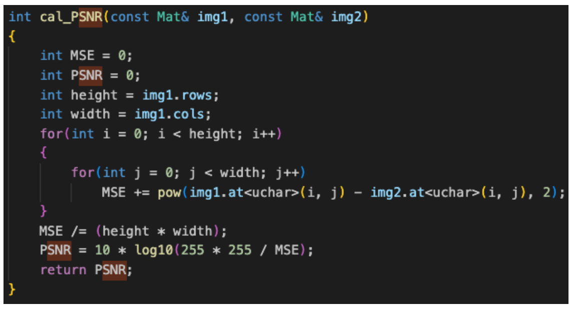


```cpp
void filter2DFreq(const Mat& inputImg, Mat& outputImg, const Mat& H)
{
    Mat planes[2] = { Mat_<float>(inputImg.clone()), Mat::zeros(inputImg.size(), CV_32F) };
    Mat complexI;
    merge(planes, 2, complexI);
    dft(complexI, complexI, DFT_SCALE);
    Mat planesH[2] = { Mat_<float>(H.clone()), Mat::zeros(H.size(), CV_32F) };
    Mat complexH;
    merge(planesH, 2, complexH);
    Mat complexIH;
    mulSpectrums(complexI, complexH, complexIH, 0);
    idft(complexIH, complexIH);
    split(complexIH, planes);
    outputImg = planes[0];
}
```



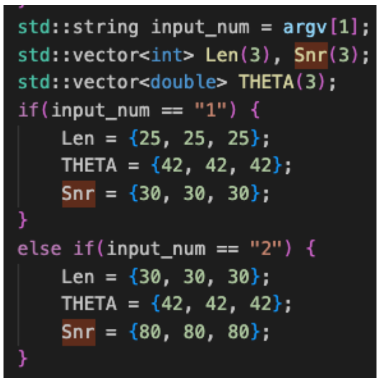
```cpp
void calcWnrFilter(const Mat& input_h_PSF, Mat& output_G, double nsr)
{
    Mat h_PSF_shifted;
    fftshift(input_h_PSF, h_PSF_shifted);
    Mat planes[2] = { Mat_<float>(h_PSF_shifted.clone()), Mat::zeros(h_PSF_shifted.size(), CV_32F) };
    Mat complexI;
    merge(planes, 2, complexI);
    dft(complexI, complexI);
    split(complexI, planes);
    Mat denom;
    pow(abs(planes[0]), 2, denom);
    denom += nsr;
    divide(planes[0], denom, output_G);
}
```

Finally I calculate the PSNR as below (from the formula).



```cpp
int cal_PSNR(const Mat& img1, const Mat& img2)
{
    int MSE = 0;
    int PSNR = 0;
    int height = img1.rows;
    int width = img1.cols;
    for(int i = 0; i < height; i++)
    {
        for(int j = 0; j < width; j++)
            MSE += pow(img1.at<uchar>(i, j) - img2.at<uchar>(i, j), 2);
    }
    MSE /= (height * width);
    PSNR = 10 * log10(255 * 255 / MSE);
    return PSNR;
}
```

I set the parameters for this degradation function as follows.



```cpp
std::string input_num = argv[1];
std::vector<int> Len(3), Snr(3);
std::vector<double> THETA(3);
if(input_num == "1") {
    Len = {25, 25, 25};
    THETA = {42, 42, 42};
    Snr = {30, 30, 30};
}
else if(input_num == "2") {
    Len = {30, 30, 30};
    THETA = {42, 42, 42};
    Snr = {80, 80, 80};
}
```

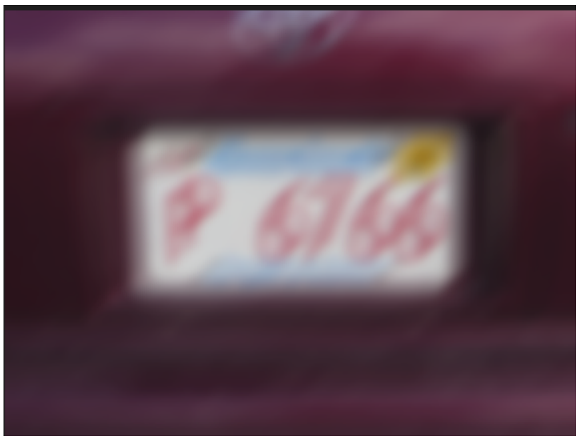
Discussion

I didn't do shiftFFT at first, this will make my results all black somehow. Convert BMP file from vector to cv::Mat need to be careful, because the coordinate of cv::Mat is not the same as our last few HW implementations.

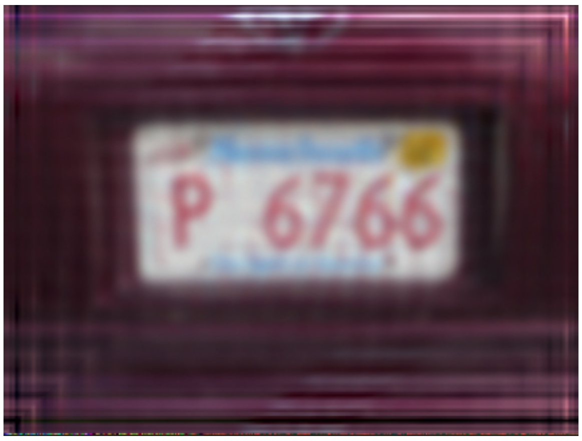
Do wiener deconvolution separately at each channel worked, as well as converting the original image to grayscale and do it once.

Results

input 1: PSNR: 22

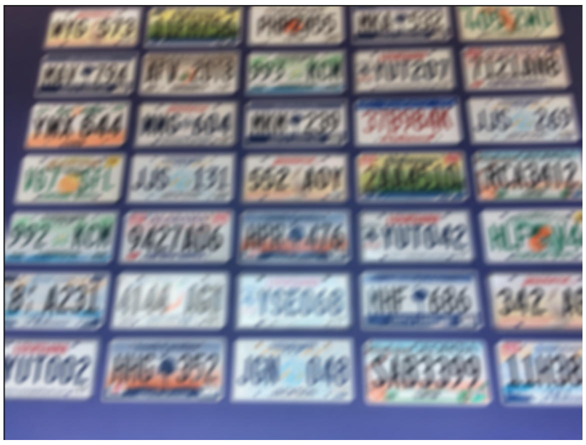


before

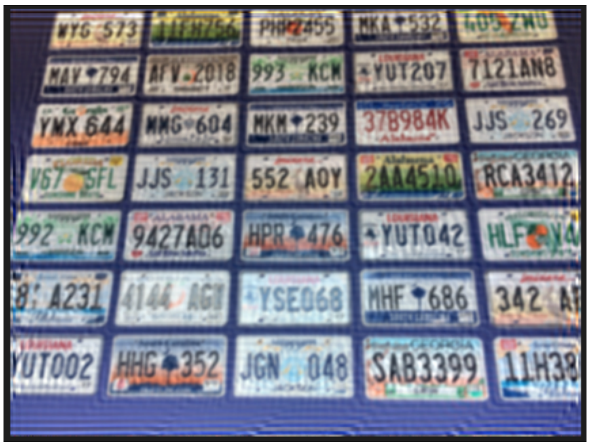


after

input 2:



before



after

WYG573	JIFH756	PHP2455	MKA532	405ZHU
MAV794	AFV2018	993KCM	YUT207	7121AN8
YMX644	MMG604	MKM239	378984K	JJS269
V67SFL	JJS131	552AOY	2AA4510	RCA3412
992KCM	9427A06	HPR476	YUT042	HLFV4
8A231	4144AGV	YSE068	MHF686	342A
YUT002	HHG352	JGN048	SAB3399	11H38