Homework

실습 및 과제

- img1.jpg에 band pass filter를 적용할 것

  해결방법:

  밴드 패스 필터는 로우 패스 필터와 하이 패스 필터의 중간에 위치하여 특정 주파수 범위만 통과시키는 필터이다.

  이 필터를 구현하기 위해서는 먼저 서로 다른 반지름을 가지는 두 개의 원을 만들고,

  이를 빼주는 방식으로 밴드 패스 필터를 구현한다.

  이렇게 하면 가운데 주파수 범위만 남게 되고 이를 통해 기존 로우 패스 필터나 하이 패스 필터와 동일한 방식으로 normalize를 적용하면 원하는 결과를 얻을 수 있을 것이다.

```
// Band Pass Filtering (BPF)라고함
Mat doBPF(Mat srcImg)
{

    Mat padImg = padding(srcImg);// 원본 이미지를 입력으로 받아 패딩된 이미지를 생성
    Mat complexImg = doDft(padImg);// 패딩된 이미지를 주파수 도메인으로 변환시킴
    Mat centerComplexImg = centralize(complexImg);//centralize를 통해 band pass filter의 모양을 확인할 수 있다.
    Mat magImg = getMagnitude(centerComplexImg);//앞서 centralize한 사진의 magnitude를 구함
    Mat phaImg = getPhase(centerComplexImg);//앞서 centralize한 사진의 phase를 구함

    // BPF 결과에서 최소값과 최대값을 찾는다
    double minVal, maxVal;
    Point minLoc, maxLoc;
    minMaxLoc(magImg, &minVal, &maxVal, &minLoc, &maxLoc);
    normalize(magImg, magImg, 0, 1, NORM_MINMAX);
    // magnitude의 사진에서 최대와 최소와 각각의 location을 구함
    // 이때 magnitude의 최대 최소의 범위를 0-1로 정규화함.

    Mat maskImg = Mat::zeros(magImg.size(), CV_32F);//선언한 이미지의 크기를 magImg와 동일하게 설정하고 초기값은 0으로 함.
    circle(maskImg, Point(maskImg.cols / 2, maskImg.rows / 2), 50, Scalar::all(1), -1, -1, 0);//반지름 50
    circle(maskImg, Point(maskImg.cols / 2, maskImg.rows / 2), 25, Scalar::all(0), -1, -1, 0);//반지름 25
    imshow("Band Pass Filter", maskImg);
    waitKey(0);
    destroyAllWindows();
    //25에서 50사이의 모양을 가진 band pass filter을 보여줌

    Mat magImg2;
    multiply(magImg, maskImg, magImg2);//원래의 magnitude이미지와 filter이미지를 곱함.


    normalize(magImg2, magImg2, (float)minVal, (float)maxVal, NORM_MINMAX);//정규화
    Mat complexImg2 = setComplex(magImg2, phaImg);//두 이미지를 곱한 주파수 도메인에서의 새로운 영상
    Mat dstImg = doIdft(complexImg2);//공간 도메인으로 변환

    return myNormalize(dstImg);//정규화한 이미지를 return
}
```

```cpp
Mat padding(Mat img)
{
    int dftRows = getOptimalDFTSize(img.rows);
    int dftCols = getOptimalDFTSize(img.cols);

    Mat padded;
    copyMakeBorder(img, padded, 0, dftRows - img.rows, 0, dftCols - img.cols, BORDER_CONSTANT, Scalar::all(0));
    return padded;
}


Mat doDft(Mat srcImg)
{
    Mat floatImg;
    srcImg.convertTo(floatImg, CV_32F);

    Mat complexImg;
    dft(floatImg, complexImg, DFT_COMPLEX_OUTPUT);

    return complexImg;
}
// Magnitude 영상 취득
Mat getMagnitude(Mat complexImg)
{
    Mat planes[2];
    split(complexImg, planes);
    // 실수부, 허수부 분리

    Mat magImg;
    magnitude(planes[0], planes[1], magImg);
    magImg += Scalar::all(1);
    log(magImg, magImg);
    // magnitude 취득
    // log(1 + sqrt(Re(DFT(I))^2 + Im(DFT(I))^2))

    return magImg;
}
```

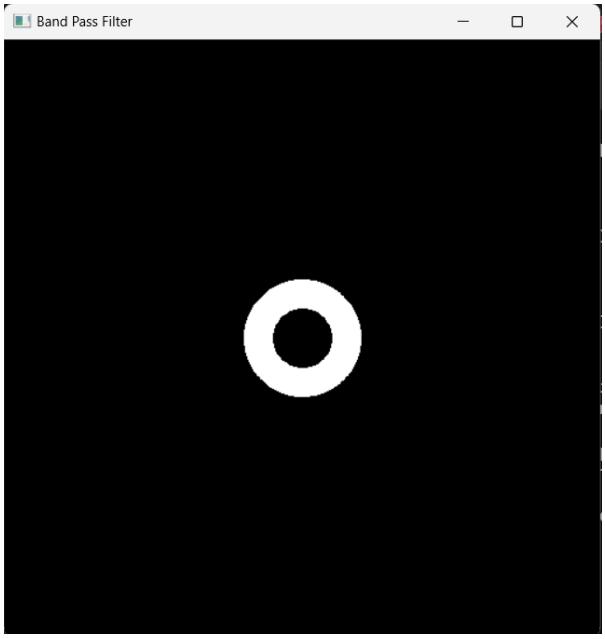
```cpp
Mat myNormalize(Mat src)
{
    Mat dst;
    src.copyTo(dst);
    normalize(dst, dst, 0, 255, NORM_MINMAX);
    dst.convertTo(dst, CV_8UC1);
    return dst;
}

// Phase 영상 취득
Mat getPhase(Mat complexImg)
{
    Mat planes[2];
    split(complexImg, planes);
    // 실수부, 허수부 분리

    Mat phaImg;
    phase(planes[0], planes[1], phaImg);
    // phase 취득

    return phaImg;
}
```

```cpp
Mat centralize(Mat complex)
{
    Mat planes[2];
    split(complex, planes);
    int cx = planes[0].cols / 2;
    int cy = planes[1].rows / 2;

    Mat q0Re(planes[0], Rect(0, 0, cx, cy));
    Mat q1Re(planes[0], Rect(cx, 0, cx, cy));
    Mat q2Re(planes[0], Rect(0, cy, cx, cy));
    Mat q3Re(planes[0], Rect(cx, cy, cx, cy));

    Mat tmp;
    q0Re.copyTo(tmp);
    q3Re.copyTo(q0Re);
    tmp.copyTo(q3Re);
    q1Re.copyTo(tmp);
    q2Re.copyTo(q1Re);
    tmp.copyTo(q2Re);

    Mat q0Im(planes[1], Rect(0, 0, cx, cy));
    Mat q1Im(planes[1], Rect(cx, 0, cx, cy));
    Mat q2Im(planes[1], Rect(0, cy, cx, cy));
    Mat q3Im(planes[1], Rect(cx, cy, cx, cy));

    q0Im.copyTo(tmp);
    q3Im.copyTo(q0Im);
    tmp.copyTo(q3Im);
    q1Im.copyTo(tmp);
    q2Im.copyTo(q1Im);
    tmp.copyTo(q2Im);

    Mat centerComplex;
    merge(planes, 2, centerComplex);

    return centerComplex;
}
```

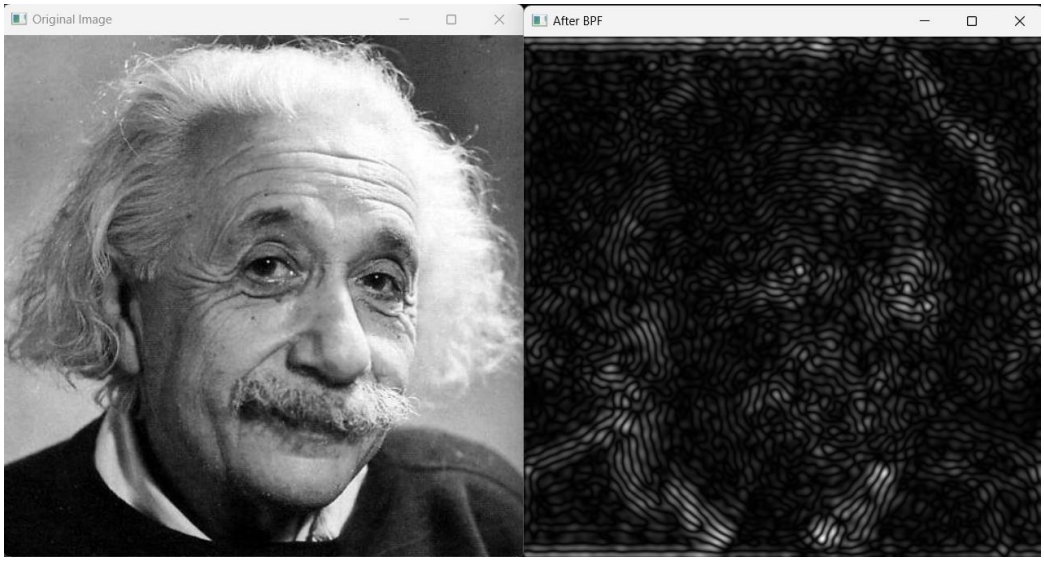
```cpp
Mat setComplex(Mat magImg, Mat phaImg)
{
    exp(magImg, magImg);
    magImg -= Scalar::all(1);
    // magnitude 계산을 반대로 수행

    Mat planes[2];
    polarToCart(magImg, phaImg, planes[0], planes[1]);
    // 극 좌표계 -> 직교 좌표계 (각도와 크기로부터 2차원 좌표)

    Mat complexImg;
    merge(planes, 2, complexImg);
    // 실수부, 허수부 합체

    return complexImg;
}
Mat doIdft(Mat complexImg)
{
    Mat idftcvt;
    idft(complexImg, idftcvt);
    // IDFT를 이용한 원본 영상 취득

    Mat planes[2];
    split(idftcvt, planes);

    Mat dstImg;
    magnitude(planes[0], planes[1], dstImg);
    normalize(dstImg, dstImg, 255, 0, NORM_MINMAX);
    dstImg.convertTo(dstImg, CV_8UC1);
    // 일반 영상의 type과 표현범위로 변환

    return dstImg;
}
```

```
int main()

{
    Mat srcImg = imread("img1.jpg", 0);//원본사진
    Mat dstImg = doBPF(srcImg);//BPF 거친 사진

    imshow("Original Image", srcImg);
    imshow("After BPF", dstImg);

    waitKey(0);
    destroyAllWindows();

    return 0;
}
```

이렇게 main함수로 실행해보면 아래와 같은 결과를 얻을 수 있다.



이 원은 밴드패스필터로 반지름이 25에서 50 사이인 부분인 필터로 이 중간부분의 값만을 통과시키는 필터이다.



이러한 필터를 통과하였을 때 위와 같은 이미지를 얻을 수 있는데 band pass filter는 들어오는 주파수에서 낮은 대역과 높은 대역이 아닌 중간 대역만을 통과하게 한다.

원래의 사진과 비교하면 결과 이미지는 얼굴의 주름과 머리의 경계를 표현한 것을 볼 수 있고 LPF를 통과할 때처럼의 블러링은 없다. 왜냐하면 저주파의 대역을 통과시키지 않기 때문이다.

또한 edge가 많이 표시되는 것을 볼 수 있는데 고주파의 대역을 통과시키지 않기 때문에 자잘자잘한 edge가 없어지지 않고 연결을 시켜주기 때문이다.

- Spatial domain, frequency domain 각각에서 sobel filter를 구현하고img2.jpg에 대해 비교할 것

```
Mat mySobelFilterInFrequencyDomain(Mat srcImg) {

    // X 성분 커널 정의
    int kernelX[3][3] = { -1,0,1,
                          -2,0,2,
                          -1,0,1 };

    // Y 성분 커널 정의
    int kernelY[3][3] = { -1, -2, -1,
                           0, 0, 0,
                           1, 2, 1 };

    // 원본 이미지를 주파수 도메인으로 변환
    Mat complexImg = doDft(srcImg);
    // 주파수 도메인 이미지를 중앙 이동
    Mat centerComplexImg = centralize(complexImg);
    // 이미지의 magnitude 구하기
    Mat srcmag = getMagnitude(centerComplexImg);
    // 이미지의 phase 구하기
    Mat phaImg = getPhase(centerComplexImg);



    // X, Y 성분 커널을 주파수 도메인 형식으로 변환
    Mat sobelX = Mat(Size(3, 3), CV_8UC1, kernelX);
    Mat sobelY = Mat(Size(3, 3), CV_8UC1, kernelY);

    // X, Y 성분 커널을 주파수 도메인으로 변환
    Mat complexX = doDft(sobelX);
    Mat complexY = doDft(sobelY);

    // X , Y 성분 커널의 size를 이미지와 일치하도록 조절
    resize(complexX, complexX, Size(complexImg.cols, complexImg.rows));
    resize(complexY, complexY, Size(complexImg.cols, complexImg.rows));

    // X 성분 주파수 도메인 이미지를 중앙으로 이동
    Mat centerComplexImgX = centralize(complexX);
    // Y 성분 주파수 도메인 이미지를 중앙으로 이동
    Mat centerComplexImgY = centralize(complexY);
```

```
// X, Y 성분 중앙화한 이미지의 magnitude 구하기
Mat sobelMagX = getMagnitude(centerComplexImgX);
Mat sobelMagY = getMagnitude(centerComplexImgY);

// X, Y 성분의 최소값과 최대값을 변수에 저장
double minValX, maxValX;
double minValY, maxValY;

// X, Y 성분의 최소값과 최대값을 찾고 0부터 1까지의 범위로 정규화
minMaxLoc(sobelMagX, &minValX, &maxValX);
normalize(sobelMagX, sobelMagX, 0, 1, NORM_MINMAX);
minMaxLoc(sobelMagY, &minValY, &maxValY);
normalize(sobelMagY, sobelMagY, 0, 1, NORM_MINMAX);

// 결과 이미지 저장 변수
Mat X;
Mat Y;
// 행렬 곱셈을 위해 이미지를 CV_32F 형식으로 변환
Mat srcmat;
Mat Xmat;
Mat Ymat;

// 이미지를 CV_32F 형식으로 변환
srcmag.convertTo(srcmat, CV_32F);
sobelMagX.convertTo(Xmat, CV_32F);
sobelMagY.convertTo(Ymat, CV_32F);

// X,Y 성분을 이미지의 magnitude와 곱
multiply(Xmat, srcmat, X);
multiply(Ymat, srcmat, Y);

// X와 Y 성분의 결과 이미지를 X와 Y 성분의 최소 및 최대 값으로 정규화
normalize(X, X, (float)minValX, (float)maxValX, NORM_MINMAX);
normalize(Y, Y, (float)minValY, (float)maxValY, NORM_MINMAX);

// X, Y 성분 이미지를 원본의 phase 이미지와 병합
Mat getX = setComplex(X, phaImg);
Mat getY = setComplex(Y, phaImg);

// 병합된 이미지를 공간 도메인으로 변환하고 0에서 255까지 정규화
Mat resultX = myNormalize(doIdft(getX));
Mat resultY = myNormalize(doIdft(getY));

// X와 Y 성분 이미지의 합 반환
return (resultX + resultY);
```

```cpp
Mat mySobelFilter(Mat srcImg)
{
    int kernelX[3][3] = { -1, 0, 1,
                          -2, 0, 2,
                          -1, 0, 1 };
    int kernelY[3][3] = { -1, -2, -1,
                          0, 0, 0,
                          1, 2, 1 };
    //x성분과 y성분의 커널을 선언한다

    Mat dstImg(srcImg.size(), CV_8UC1);
    uchar* srcData = srcImg.data;
    uchar* dstData = dstImg.data;
    int width = srcImg.cols;
    int height = srcImg.rows;

    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            dstData[y * width + x] = (abs(myKernelConv3x3(srcData, kernelX, x, y, width, height)) +
                abs(myKernelConv3x3(srcData, kernelY, x, y, width, height))) /
                2;
        }
    }

    return dstImg;
}
int myKernelConv3x3(uchar* arr, int kernel[3][3], int x, int y, int width, int height)
{
    int sum = 0;
    int sumKernel = 0;

    for (int j = -1; j <= 1; j++)
    {
        for (int i = -1; i <= 1; i++)
        {
            if ((y + j) >= 0 && (y + j) < height && (x + i) >= 0 && (x + i) < width)
            {
                sum += arr[(y + j) * width + (x + i)] * kernel[i + 1][j + 1];
                sumKernel += kernel[i + 1][j + 1];
            }
        }
        //시그마 계산을 할 때 컨볼루션 식을 사용하여 계산
        //각 원소들의 총합을 구해서 최종적으로 행렬의 총 합을 구해 나눈다.
    }
    if (sumKernel != 0)
    {
        return sum / sumKernel;
    }
    else
    {
        return sum;
    }
}
```

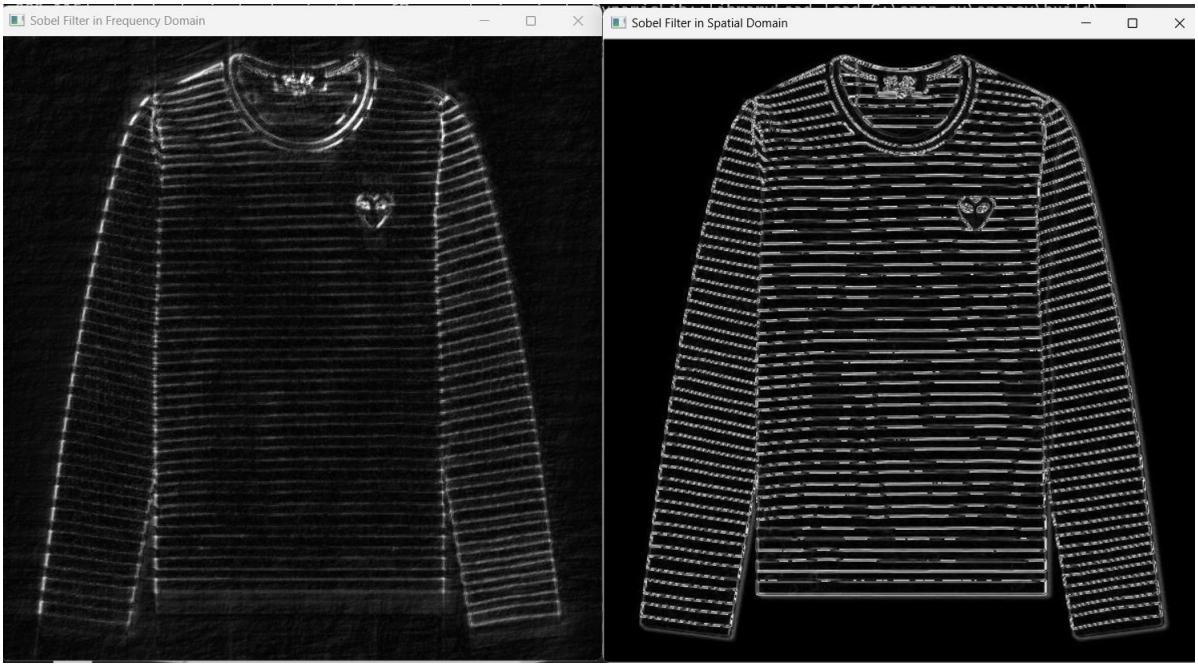
```cpp
int main()
{
    Mat srcImg = imread("img2.jpg", 0);
    Mat frequencyDstImg = mySobelFilterInFrequencyDomain(srcImg);
    Mat spatialDstImg = mySobelFilter(srcImg);

    imshow("Original Image", srcImg);
    imshow("Sobel Filter in Frequency Domain", frequencyDstImg);
    imshow("Sobel Filter in Spatial Domain", spatialDstImg);

    waitKey(0);
    destroyAllWindows();

    return 0;
}
```

먼저 Sobel 필터는 이미지에서 edge를 감지하기 위한 컨볼루션 필터이다
이 필터는 이미지의 밝기 값이 급격하게 변화하는 지점을 찾아내어 edge를 감지하는데 자신이 설정한 커널과의 유사성이 큰 곳에 반응하여 그곳에 있는 edge를 감지한다.
에지의 감지를 위해 Sobel 필터는 주로 X 및 Y 방향으로 edge를 감지하는 두 가지 커널을 사용한다.

공간 영역에서의 Sobel 필터는 수평 방향의 edge를 감지하는 X 성분 커널과 수직 방향

의 edge를 감지 Y 성분 커널로 구성한다
그 다음 원본 이미지에 각각의 커널을 컨볼루션하여 X 및 Y 성분에 대한 화소 값을 계산하여
edge를 찾고 공간영역에서의 sobel 필터를 거친 이미지를 출력한다.

주파수 도메인에서의 곱은 공간 영역에서의 컨볼루션과 동일하다.
X 및 Y 성분의 커널을 Fourier 변환하여 주파수 도메인으로 이동하게하여 주파수 도메인에서의 필터링 작업을 할 수 있게 한다.
원본 이미지도 Fourier 변환을 수행하고 변환된 이미지의 이미지 강도인 magnitude 부분을 얻게 한 뒤
추출된 magnitude와 X 성분 커널과 Y 성분 커널을 요소별로 행렬 곱을 하여
아까 말했다시피 공간 영역에서의 컨볼루션과 동일한 효과를 곱으로 주파수 도메인에서 구현한다.
이러한 곱셈 결과를 다시 공간 도메인으로 변환하기 위해 역 Fourier 변환을 하고 두 결과를 합산한다.
그 이후 정규화 과정을 거치게 하여 주파수 도메인에서의 sobel filter과정을 마친다.



첫번째 사진은 주파수 영역에서의 Sobel 필터 구현한 결과의 사진인데
주파수 영역에서 Sobel 필터는 주파수 도메인으로 변환된 이미지에 주파수 도메인에서의 필터를 적용하고 아까와 같은 과정을 거쳐
다시 역 퓨리에로 공간영역에서의 필터로 나온 것인다.
주파수 영역에서의 Sobel 필터는 두번째 그림인 공간 영역에서의 필터에 비해 주파수 도메인에서의 필터링 과정에서 발생한 정보 손실 때문에 edge가 흐릿하다.

주파수 영역에서의 필터링 과정에서 정규화 또는 주파수 도메인으로 변환 후 재조정하는 과정에서 정보가 손실될 수 있는데
변환 후 재조정하는 과정에서 행렬 값이 부족하거나 비어있을 경우 결과 이미지가 어둡게 나타날 수 있다.
이러한 오차는 주로 주파수 도메인에서의 연산 및 역변환 과정에서 발생할 수 있고 잘못된 정규화나 부족한 행렬 값 등이 그림이 이렇게 나오는 원인이라고 생각한다.

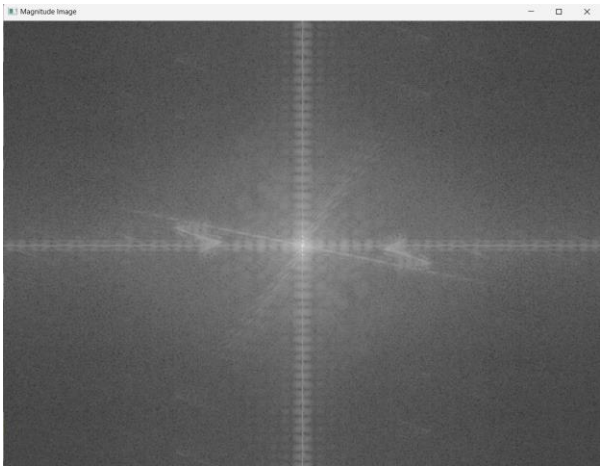
■ img3.jpg에서 나타나는 flickering 현상을 frequency domain filtering을 통해 제거할 것

해결방법:
이렇게 flickering 현상을 없애려면 magnitude에서의 edge와 수직인 성분을 가려주면 그 edge또한 사라진다.
이 그림에서는 가로의 에지를 사라지게 해야한다. 따라서 magnitude에서 세로 부분을 검은색의 mask로 가려주면 자연스레 flikering현상이 사라질 것이다.

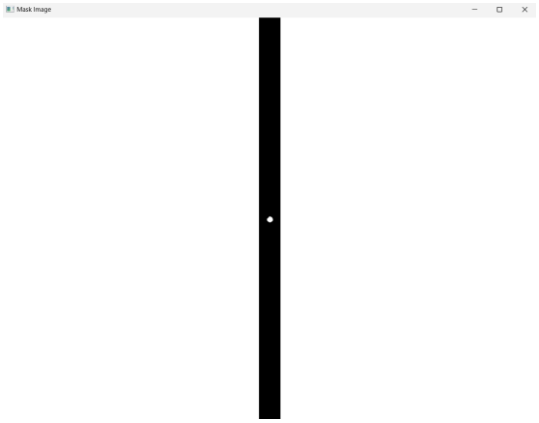
```
// Flicker를 제거하는 함수
Mat removeFlicker(Mat srcImg)
{
    Mat padImg = padding(srcImg);
    Mat complexImg = doDft(padImg);
    Mat centerComplexImg = centralize(complexImg);
    Mat magImg = getMagnitude(centerComplexImg);
    Mat phaImg = getPhase(centerComplexImg);

    double minVal, maxVal;
    Point minLoc, maxLoc;
    minMaxLoc(magImg, &minVal, &maxVal, &minLoc, &maxLoc);// 크기 정보의 최솟값과 최댓값 계산
    normalize(magImg, magImg, 0, 1, NORM_MINMAX);// 크기 정보를 0에서 1 범위로 정규화한다

    Mat maskImg = Mat::ones(magImg.size(), CV_32F);
    rectangle(maskImg, Rect(Point(maskImg.cols / 2 - 20, 0), Point(maskImg.cols / 2 + 20, maskImg.rows)), Scalar::all(0), -1);
    circle(maskImg, Point(maskImg.cols / 2, maskImg.rows / 2), 5, Scalar::all(1), -1, -1, 0);
    imshow("Mask Image", maskImg);
    // 크기 정보와 같은 크기의 모두 1로 이루어진 행렬 생성하고
    // 중심 영역을 검은색으로 채우고 중심점을 흰색으로 채운다

    Mat maskedMagImg;
    multiply(magImg, maskImg, maskedMagImg);// 크기 정보에 마스크를 적용해서 중심 영역을 제외한 영역을 0으로 만든다
    imshow("Masked Magnitude Image", maskedMagImg);



    normalize(maskedMagImg, maskedMagImg, (float)minVal, (float)maxVal, NORM_MINMAX);
    // 정규화된 크기 정보를 원래 범위로 변환한다

    Mat complexImg2 = setComplex(maskedMagImg, phaImg);
    Mat dstImg = doIdft(complexImg2);
    return myNormalize(dstImg);
    // 주파수 도메인 역변환 수행하고 생성된 이미지를 정규화해서 반환한다.
}
```

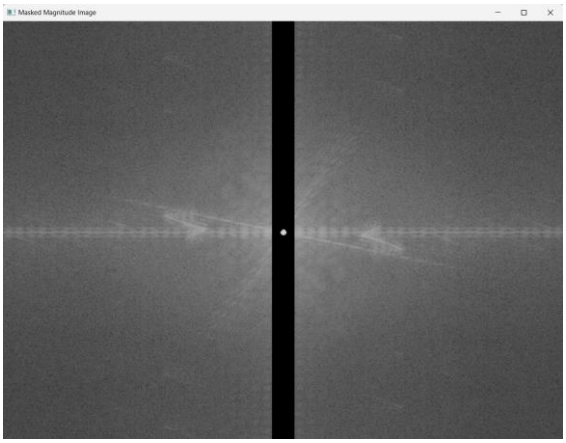
```
void showMagnitude(Mat srcImg)
{
    Mat padImg = padding(srcImg);
    Mat complexImg = doDft(padImg);
    Mat centerComplexImg = centralize(complexImg);
    Mat magImg = getMagnitude(centerComplexImg);
    Mat phaImg = getPhase(centerComplexImg);

    // 크기 정보 정규화하고 크기정보를 시각화하여 보여준다
    double minVal, maxVal;
    Point minLoc, maxLoc;
    minMaxLoc(magImg, &minVal, &maxVal, &minLoc, &maxLoc);
    normalize(magImg, magImg, 0, 1, NORM_MINMAX);
    imshow("Magnitude Image", magImg);
}
```

Magnitude를 보여주는 함수로 크기 정보를 정규화하고 시각화하여 보여준다.



이렇게 magnitude가 나타나는데 이때 가운데 저 세로로 가운데에 있는 하얀 부분이 그림에서의 지워야 할 가로부분을 만들기 때문에
크기에서 가운데 세로를 가려주면 된다. 이를 가리기 위해 위에 있는 함수로 마스크를 만들어서 아래와 같이 가려줄 수 있다.





이를 통한 결과의 사진을 보면 원본에서 마스킹 한 필터를 씌우면 오른쪽 그림과 같이 flickering이 많이 사라진 결과가 나오는 것을 볼 수 있다.