1. RGB 영상 포맷 확인(Check RGB Format.cpp)

- 컬러 이미지는 픽셀 하나에 3가지 값이 저장되는데, 순서대로 우리가 흔히 알고 있는 RGB(Red, Green, Blue)순서가 아닌, BGR(Blue, Green, Red) 순서대로 저장되어 있다.

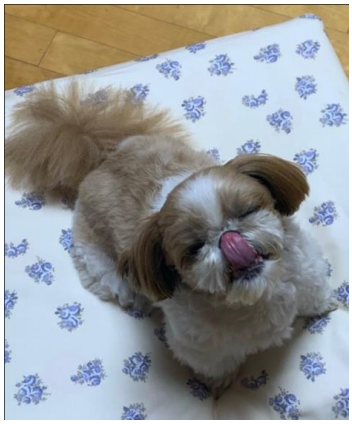
- Mat data로 접근하게 되면, BGR값이 순서대로 일렬로 저장되어 있다. 3으로 나누어 떨어지는 경우 Blue값, 나머지가 1인경우 Green값, 나머지가 2인경우 Red값이 저장되어 있는 것이다.

```
Mat img = imread("겨울이.jpg");
        printf(" ₩t(R, G, B) data : Mat::data ₩n");
        uchar* img_data = img.data;

        for (int row = 0; row < img.rows; row++)
        {
                for (int col = 0; col < img.cols; col++)
                {
                        uchar b = img_data[row * img.cols * 3 + col * 3];
                        uchar g = img_data[row * img.cols * 3 + col * 3 + 1];
                        uchar r = img_data[row * img.cols * 3 + col * 3 + 2];
                        printf("₩t (%d, %d, %d)", r, g, b);
                }
                cout << "₩n" << endl;
        }
```

- 결과값 출력은 편의상 RGB순서로 출력되도록 설정함.

- 출력결과

```
(202, 212, 224)  (201, 210, 225)  (199, 208, 225)  (198, 207, 224)  (197, 206, 223)
(196, 205, 222)  (194, 203, 220)  (194, 202, 221)  (195, 203, 222)  (198, 206, 225)
(200, 208, 229)  (201, 209, 230)  (205, 213, 234)  (203, 214, 234)  (201, 212, 232)
(199, 210, 228)  (200, 211, 229)  (198, 211, 228)  (198, 211, 228)  (199, 212, 229)
(202, 213, 231)  (202, 213, 231)  (203, 214, 232)  (204, 215, 233)  (204, 217, 236)
(205, 218, 237)  (206, 219, 238)  (208, 221, 240)  (206, 221, 240)  (206, 221, 240)
(207, 222, 241)  (207, 222, 241)  (206, 223, 241)  (206, 223, 241)  (207, 224, 242)
(207, 224, 242)  (207, 224, 242)  (206, 223, 241)  (206, 223, 241)  (206, 223, 241)
(206, 223, 241)  (206, 223, 241)  (206, 223, 241)  (205, 222, 240)  (207, 222, 241)
(208, 223, 242)  (208, 223, 242)  (208, 223, 242)  (208, 223, 244)  (208, 223, 244)
```



1. YUV (YCbCr) color space(RGB to YUV.cpp)

2.1 YUV Color Space

- 디지털 비디오 스트림 전송 포맷. 영상에서 어둡고 밝은 지 정도를 나타내는 성분인 Y(휘도:Luminance), 색상 정보를 가지는 U,V 성분(색차:chrominance)으로 구성되어 있다.

- Y는 R,G,B에 각각 가중치를 부여하여 더한 값으로 결정된다.(Y = 0.299*R + 0.587*G + 0.114*B)

- 각각의 R,G,B성분에서 Y를 빼서 색상 정보를 얻는다. U(파란 색차 성분, Cb)=(B-Y)*0.564 + delta, V(빨간 색차 성분, Cr)=(R-Y)*0.713 + delta

- delta는 8비트 이미지에서 128, 16비트에서 32768의 값을 갖는다.

2.2 RGB to YUV 직접 연산

```
for (int row = 0; row < img.rows; row++)
	{
		for (int col = 0; col < img.cols; col++)
		{
			uchar b = img_data[row * img.cols * 3 + col * 3];
			uchar g = img_data[row * img.cols * 3 + col * 3 + 1];
			uchar r = img_data[row * img.cols * 3 + col * 3 + 2];
			uchar y = 0.299*r + 0.587*g + 0.114*b;
			uchar u = (b - y)*0.564 + 128; //cb
			uchar v = (r - y)*0.713 + 128; //cr
			printf("₩t (%d, %d, %d)", y, u, v);
			img_data[row * img.cols * 3 + col * 3] = y;
			img_data[row * img.cols * 3 + col * 3 + 1] = v;
			img_data[row * img.cols * 3 + col * 3 + 2] = u;
		}
		cout << "₩n" << endl;
	}
```

- y, u(cb), v(cr) 값을 2.1에서 언급한 공식대로 rgb값을 이용하여 직접 계산하여 결과출력.

- 마지막 2개의 픽셀 값 결과

2.3 cvtcolor 이용

```
//cv2.COLOR_BGR2YUV: BGR 색상 이미지를 YUV 색상 이미지로 변환
	Mat changeimg = img;
	cvtColor(img, changeimg, COLOR_BGR2YCrCb);
	uchar* changeimg_data = changeimg.data;
	for (int row = 0; row < changeimg.rows; row++)
	{
		for (int col = 0; col < changeimg.cols; col++)
		{
			//opencv에서는 YCbCr순서가 아닌 YCrCb순서로 인식한다.
			uchar y = changeimg_data[row * changeimg.cols * 3 + col * 3];
			uchar v = changeimg_data[row * changeimg.cols * 3 + col * 3 + 1];
//cr
			uchar u = changeimg_data[row * changeimg.cols * 3 + col * 3 + 2];
//cb

			printf("₩t (%d, %d, %d)", y, u, v);
		}
		cout << "₩n" << endl;
	}
```

- cvtColor(img, changeimg, COLOR_BGR2YCrCB)를 이용해 이미지 변환.

- opencv에서는 YCbCr순서가 아닌 YCrCb순서로 인식되므로, 결과 비교용 결과출력시 순서를 맞추어 출력.

- 마지막 2개의 픽셀 값이 직접 2.2의 결과와 1정도 차이가 난다.

2. Filtering 함수 작성(Varuous Filter.cpp)

```
void filter(Mat &img, Mat &output_img, Mat &mask, int mask_rows, int mask_cols) {
	Mat copy_plate(img.rows + mask_rows / 2 * 2, img.cols + mask_cols / 2 * 2, CV_32F,
Scalar(0));//복사, 계산용

	uchar* copy_data = copy_plate.data;
	uchar* img_data = img.data;
	uchar* output_data = output_img.data;
```

- void filter(원본이미지, 결과이미지, 적용필터(mask), 필터세로길이, 필터가로길이)

- 복사 및 계산용으로 filter(mask)의 절반만큼 크기가 큰 copy_plate 생성, 전체값을 0으로 초기화.

```
	//zero padding제외 가운데 원본사진복사
	for (int row = 0; row < img.rows; row++)
	{
		for (int col = 0; col < img.cols; col++)
		{
			if (img.channels() == 3) {
				copy_data[(row + mask_rows / 2)*(copy_plate.cols) * 3 + (col
+ mask_cols / 2) * 3] = img_data[row * img.cols * 3 + col * 3];
				copy_data[(row + mask_rows / 2)*(copy_plate.cols) * 3 + (col
+ mask_cols / 2) * 3 + 1] = img_data[row * img.cols * 3 + col * 3 + 1];
				copy_data[(row + mask_rows / 2)*(copy_plate.cols) * 3 + (col
+ mask_cols / 2) * 3 + 2] = img_data[row * img.cols * 3 + col * 3 + 2];
			}
			else if (img.channels() == 1) {
				copy_data[(row + mask_rows / 2)*(copy_plate.cols) + (col +
mask_cols / 2)] = img_data[row * img.cols + col];
			}
		}
	}
```

- copy_plate의 boundary를 제외한 중앙부분에 원본 이미지 데이터 복사.

```
	//filter 적용
	for (int row = 0; row < img.rows; row++)
	{
		for (int col = 0; col < img.cols; col++)
		{
			if (img.channels() == 3) {		//컬러이미지 일때
				float sum_b = 0;
				float sum_g = 0;
				float sum_r = 0;
				for (int mask_r = 0; mask_r < mask.rows; mask_r++) {
					for (int mask_c = 0; mask_c < mask.cols; mask_c++) {
						sum_b += mask.at<float>(mask_r, mask_c) *
```

```
copy_data[(row + mask_r)*(copy_plate.cols) * 3 + (col + mask_c) * 3];
                                                sum_g += mask.at<float>(mask_r, mask_c) *
copy_data[(row + mask_r)*(copy_plate.cols) * 3 + (col + mask_c) * 3 + 1];
                                                sum_r += mask.at<float>(mask_r, mask_c) *
copy_data[(row + mask_r)*(copy_plate.cols) * 3 + (col + mask_c) * 3 + 2];
                                        }
                                }

                                if (sum_b > 255) sum_b = 255;
                                else if (sum_b < 0) sum_b = 0;
                                if (sum_g > 255) sum_g = 255;
                                else if (sum_g < 0) sum_g = 0;
                                if (sum_r > 255) sum_r = 255;
                                else if (sum_r < 0) sum_r = 0;

                                output_data[row * img.cols * 3 + col * 3] = sum_b;
                                output_data[row * img.cols * 3 + col * 3 + 1] = sum_g;
                                output_data[row * img.cols * 3 + col * 3 + 2] = sum_r;
                        }
                        else if (img.channels() == 1) {   //흑백이미지 일때
                                float sum = 0;
                                for (int mask_r = 0; mask_r < mask.rows; mask_r++) {
                                        for (int mask_c = 0; mask_c < mask.cols; mask_c++) {
                                                sum += mask.at<float>(mask_r, mask_c) *
copy_data[(row + mask_r)*(copy_plate.cols) + (col + mask_c)];
                                        }
                                }
                                if (sum > 255) {
                                        sum = 255;
                                }
                                else if (sum < 0) {
                                        sum = 0;
                                }
                                output_data[row * img.cols + col] = sum;
                        }
                }
        }
}
```

- 입력된 filter(mask)를 copy_plate에 적용하여 convolution 연산 후, 결과를 output_img에 저장.

3. 다양한 filter 적용(Varuous Filter.cpp)

4.0 영상 준비(in main)

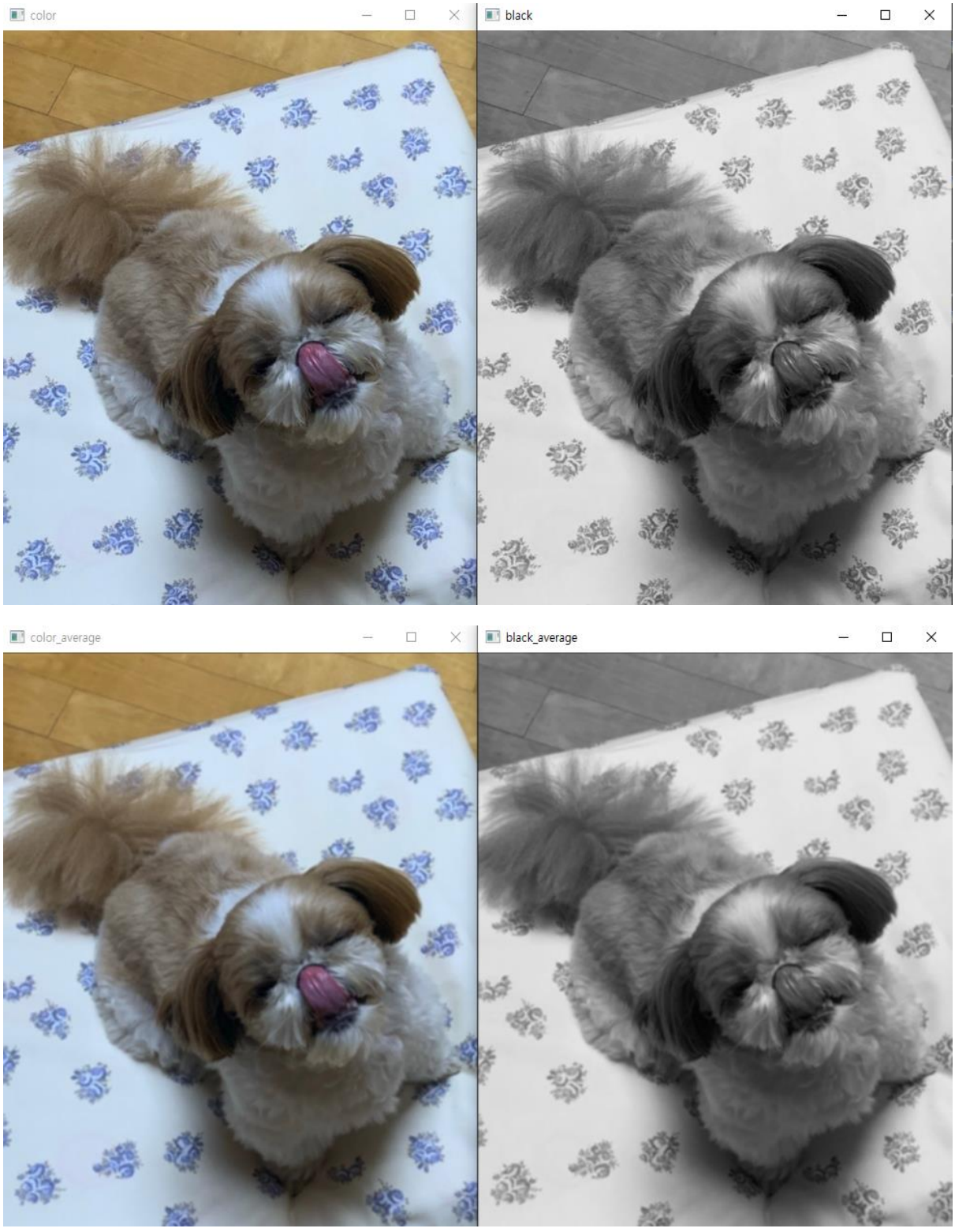
```
Mat color_img = imread("겨울이.jpg"); //컬러이미지
int h = color_img.rows;
int w = color_img.cols;
Mat black_img; //흑백이미지
cvtColor(color_img, black_img, cv::COLOR_RGB2GRAY);
```

4.1.1 3X3 moving average 수행 (in main)

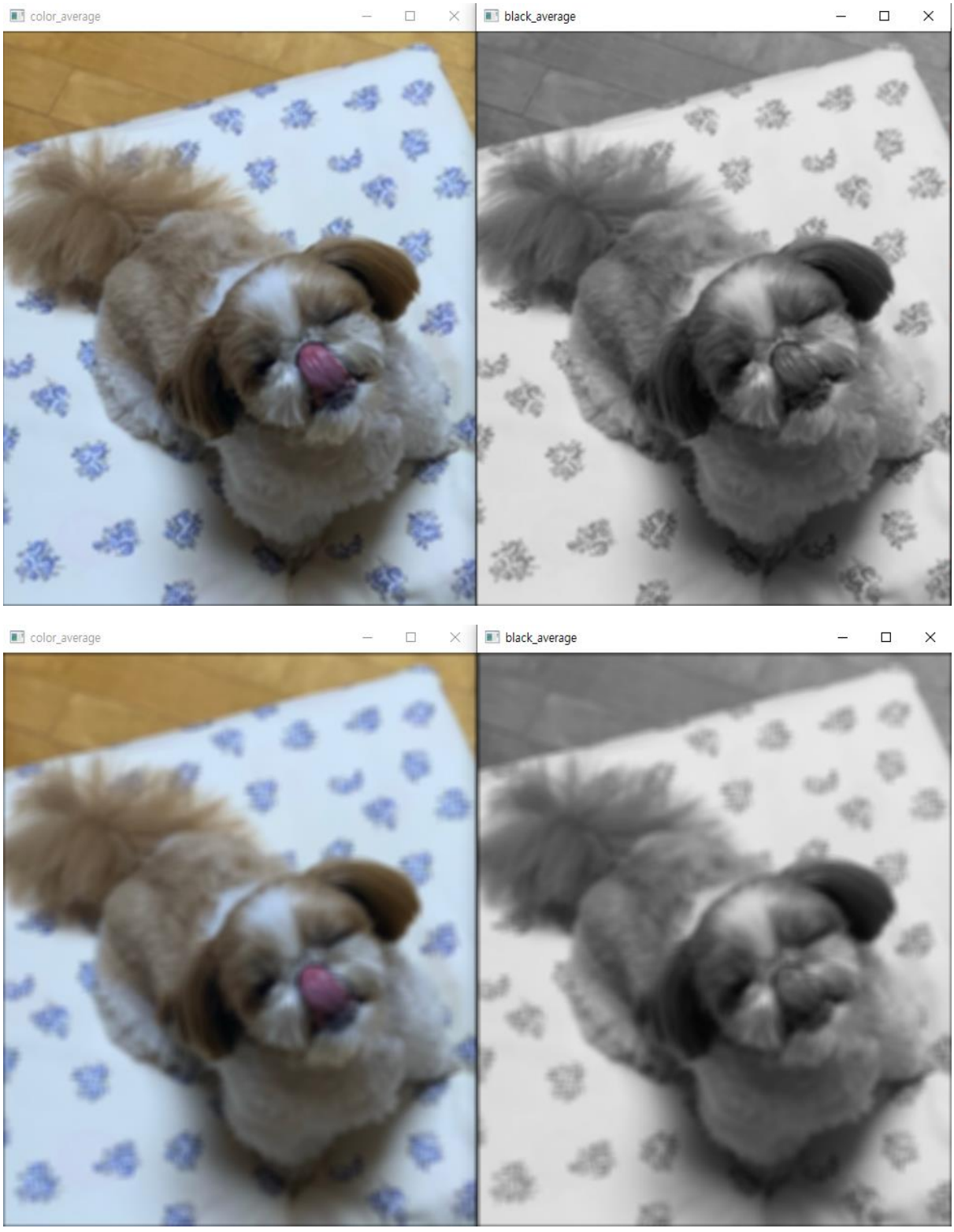
```
// 3X3 moving average
float aver_data[] = {
        1 / 9.f,1 / 9.f,1 / 9.f,
        1 / 9.f,1 / 9.f,1 / 9.f,
        1 / 9.f,1 / 9.f,1 / 9.f
};
Mat aver_mask(3, 3, CV_32F, aver_data);

Mat color_aver_outimage(h, w, CV_8UC3, Scalar(0));
filter(color_img, color_aver_outimage, aver_mask, 3, 3);
Mat black_aver_outimage(h, w, CV_8UC1, Scalar(0));
filter(black_img, black_aver_outimage, aver_mask, 3, 3);

imshow("color_average", color_aver_outimage), imshow("black_average",
black_aver_outimage);
waitKey();
```

- 3에서 구현한 filter함수에 moving average수행을 위한 3X3 크기의 aver_mask 대입.

4.1.2 moving average 구현 결과 확인 (상측 원본, 하측 filter적용 이후)



4.1.3 moving average 구현 결과 확인 (상측 5X5, 하측 9X9)

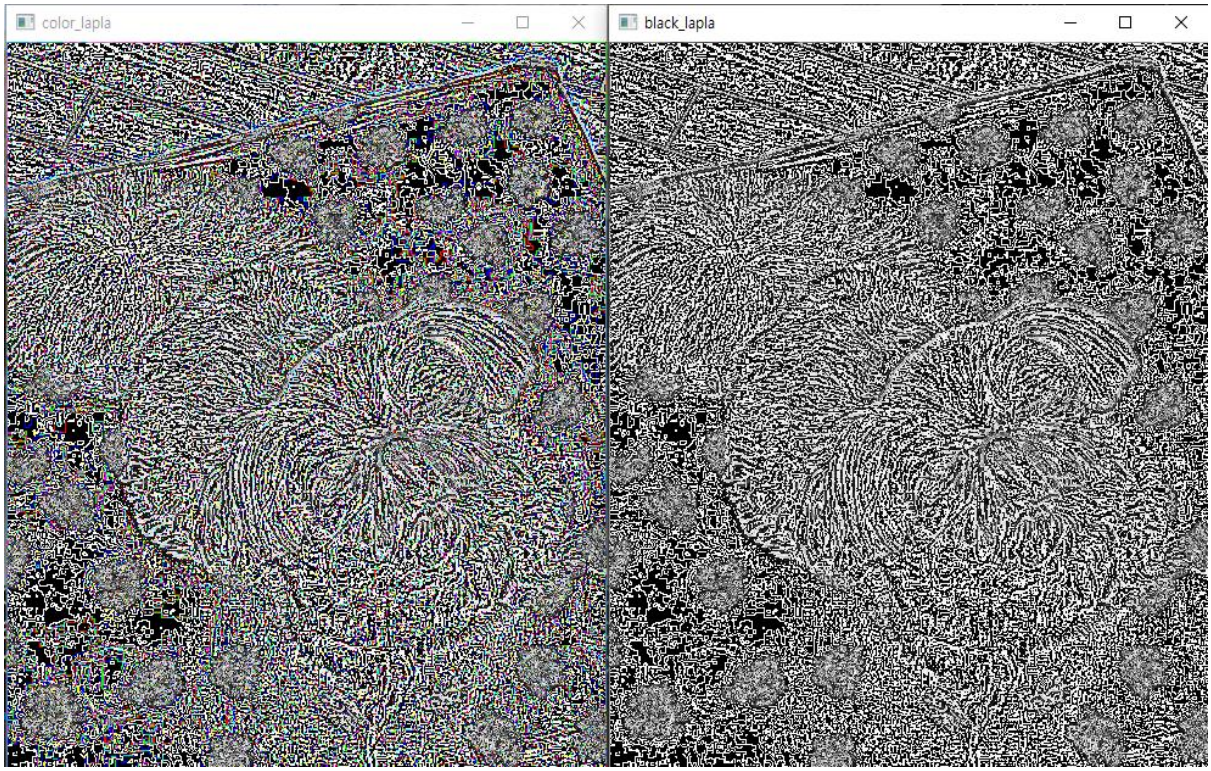


4.2.1 Laplacian 수행 (in main)

```
float laplacian_data[] = {
                -1, -1, -1,
                -1, 8, -1,
                -1, -1, -1
        };
        Mat lapla_mask(3, 3, CV_32F, laplacian_data);
        Mat color_lapla_outimage(h, w, CV_8UC3, Scalar(0));
        filter(color_img, color_lapla_outimage, lapla_mask, 3, 3);
        Mat black_lapla_outimage(h, w, CV_8UC1, Scalar(0));
        filter(black_img, black_lapla_outimage, lapla_mask, 3, 3);

        imshow("color", color_img), imshow("black", black_img);
        imshow("color_lapla", color_lapla_outimage), imshow("black_lapla",
black_lapla_outimage);
        waitKey();
```

4.2.2 Laplacian 수행 결과 이미지

- filter 함수에 Laplacian 필터 {-1,-1,-1,-1,8,-1,-1,-1,-1} 대입 결과. (좌측 컬러, 우측 흑백이미지)



4.3.1 Sharpening filter 구현

```
void sharpening(Mat &img, Mat &output_img) {


	uchar* img_data = img.data;

	uchar* lap_data;
	uchar* output_data = output_img.data;

	float laplacian_data[] = {
		-1, -1, -1,
		-1, 8, -1,
		-1, -1, -1
	};
	Mat lapla_mask(3, 3, CV_32F, laplacian_data);

	if (img.channels() == 3) {
		Mat lapla_outimage(img.rows, img.cols, CV_8UC3, Scalar(0));
		filter(img, lapla_outimage, lapla_mask, 3, 3);
		lap_data = lapla_outimage.data;
		for (int row = 0; row < img.rows; row++) {
			for (int col = 0; col < img.cols; col++) {
				float sum_b = 0;
				float sum_g = 0;
				float sum_r = 0;
				sum_b = img_data[row * img.cols * 3 + col * 3] + lap_data[row
* lapla_outimage.cols * 3 + col * 3];
				sum_g = img_data[row * img.cols * 3 + col * 3 + 1] +
lap_data[row * lapla_outimage.cols * 3 + col * 3 + 1];
				sum_r = img_data[row * img.cols * 3 + col * 3 + 2] +
lap_data[row * lapla_outimage.cols * 3 + col * 3 + 2];
				if (sum_b > 255) sum_b = 255;
				else if (sum_b < 0) sum_b = 0;
				if (sum_g > 255) sum_g = 255;
				else if (sum_g < 0) sum_g = 0;
				if (sum_r > 255) sum_r = 255;
				else if (sum_r < 0) sum_r = 0;
				output_data[row * output_img.cols * 3 + col * 3] = sum_b;
				output_data[row * output_img.cols * 3 + col * 3 + 1] = sum_g;
				output_data[row * output_img.cols * 3 + col * 3 + 2] = sum_r;
			}
		}
	}
	else if (img.channels() == 1) {
		Mat lapla_outimage(img.rows, img.cols, CV_8UC1, Scalar(0));
		filter(img, lapla_outimage, lapla_mask, 3, 3);
		lap_data = lapla_outimage.data;
		for (int row = 0; row < img.rows; row++) {
			for (int col = 0; col < img.cols; col++) {
				float sum = 0;
				sum = img_data[row * img.cols + col] + lap_data[row *
lapla_outimage.cols + col];
				if (sum > 255) sum = 255;
```

```
                    else if (sum < 0) sum = 0;
                    output_data[row * output_img.cols + col] = sum;
                }
            }
        }
}
```

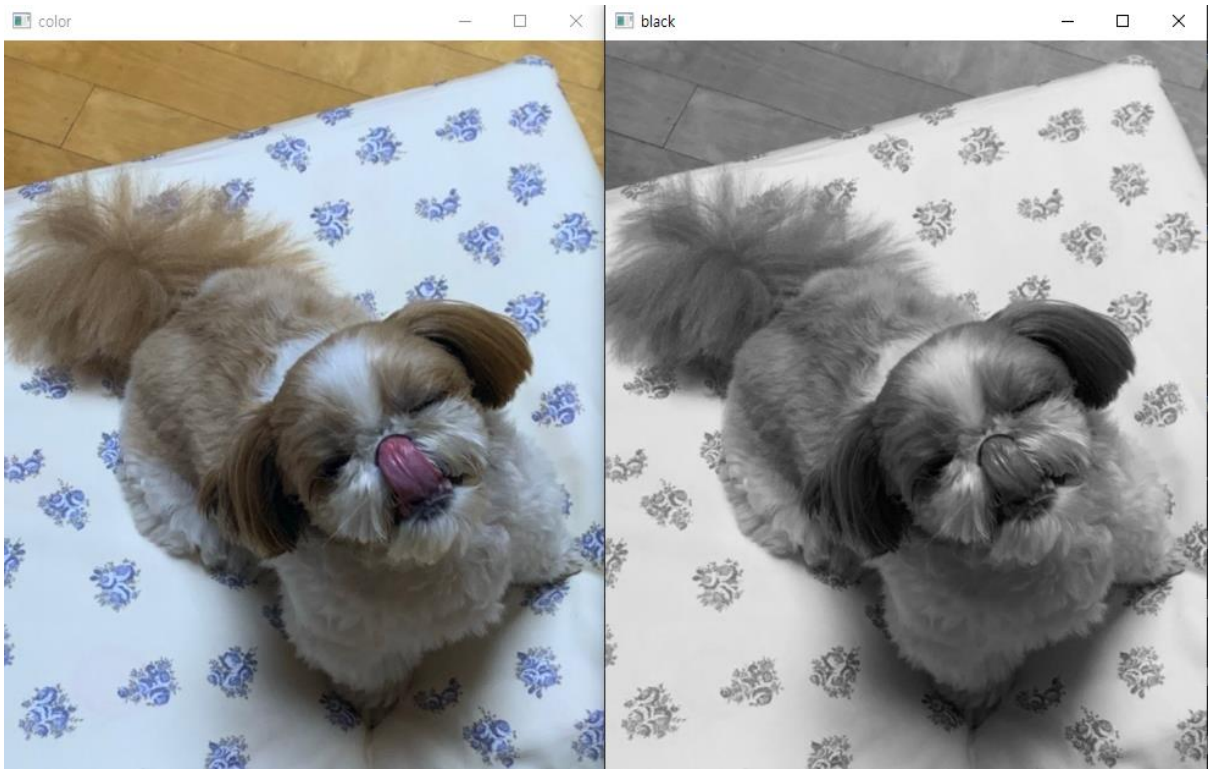
- void sharpening(원본이미지, 결과이미지)

- 함수 내부에서 3.의 filter 함수를 사용하여 4.2의 라플라시안 필터를 사용한 결과(엣지가 추출된) 이미지를 원본 이미지에 더하는 함수.

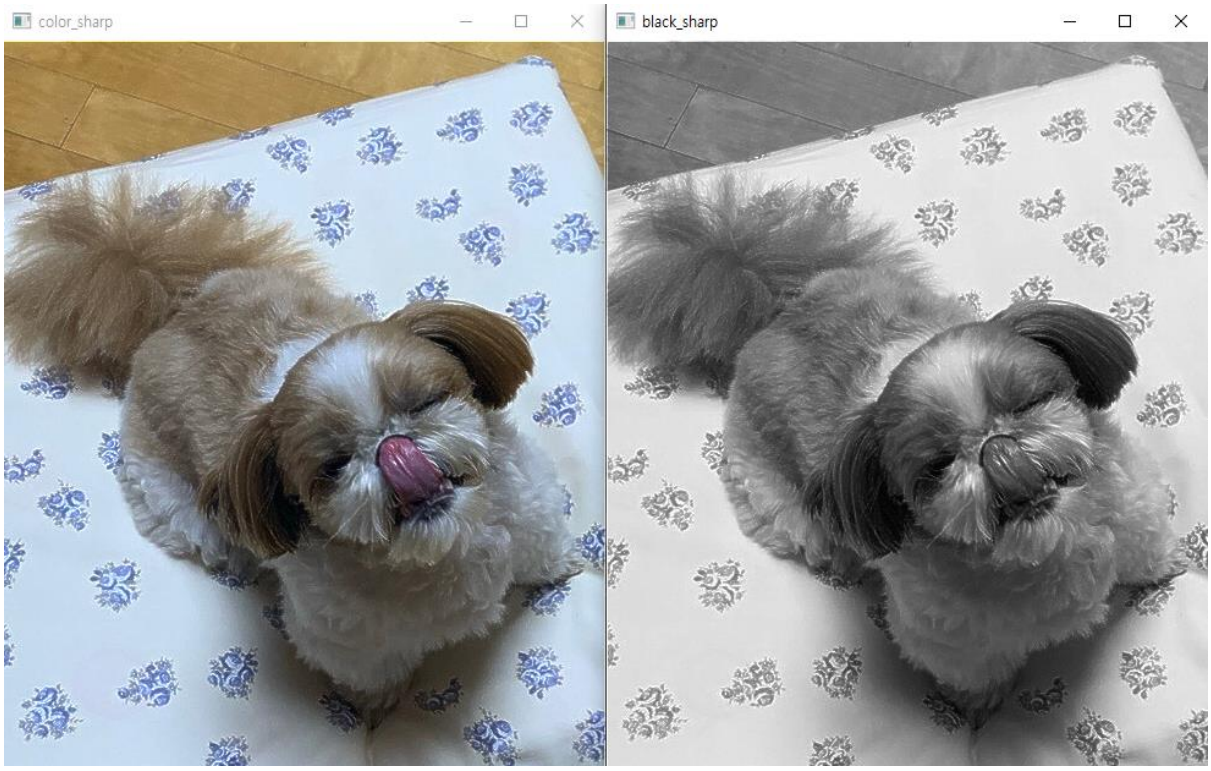
- 라플라시안에 사용한 필터가 { 0, -1, 0, -1, 4, -1, 0, -1, 0 } 또는 { -1, -1, -1, -1, 8, -1, -,1 -,1 -,1 } 인 경우에는 원본에 엣지 추출 결과를 더해야 하지만, { 0, 1, 0, 1, -4, 1, 0, 1, 0 } 또는 { 1, 1, 1, 1, 8, 1, 1, 1, 1 }인 경우에는 원본에서 추출 결과를 빼야한다. (위 코드에선 더하였음.)

4.3.2 Sharpening filter 구현 결과 확인

- 원본이미지



- { 0, -1, 0, -1, 4, -1, 0, -1, 0 } 라플라시안 필터사용



- { -1, -1, -1, -1, 8, -1, -,1 -,1 -,1 } 라플라시안 필터사용