

컴퓨터 비전 과제 2

정보통신공학과
12171786 박용민

과제 :

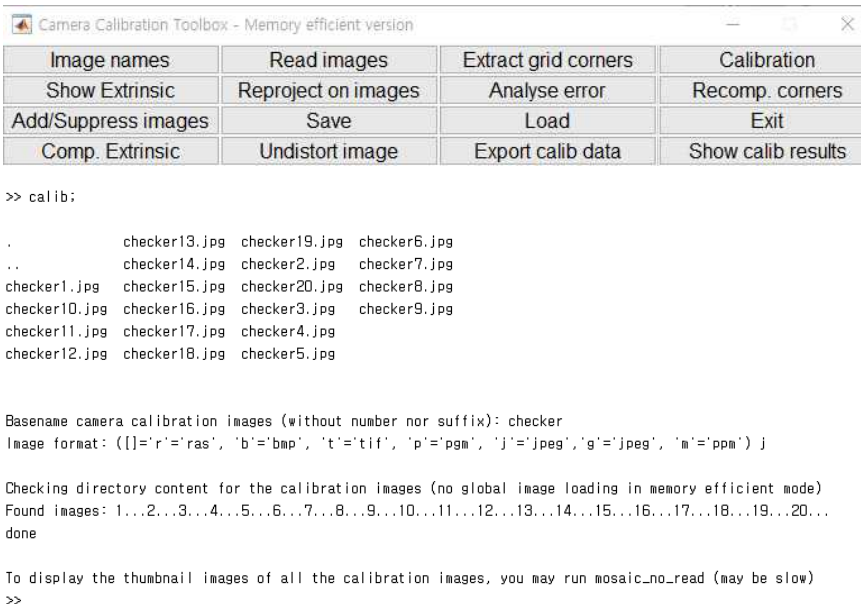
1. CalTech Matlab Camera Calibration Toolbox를 이용해서 Calibration을 진행하고 intrinsic parameter를 비교하고 분석하라.
2. Planar image stitching

언어 :

- C++ (opencv lib에서 I/O function만 가져왔다)

1번 :

- Calibration



```
>> calib;

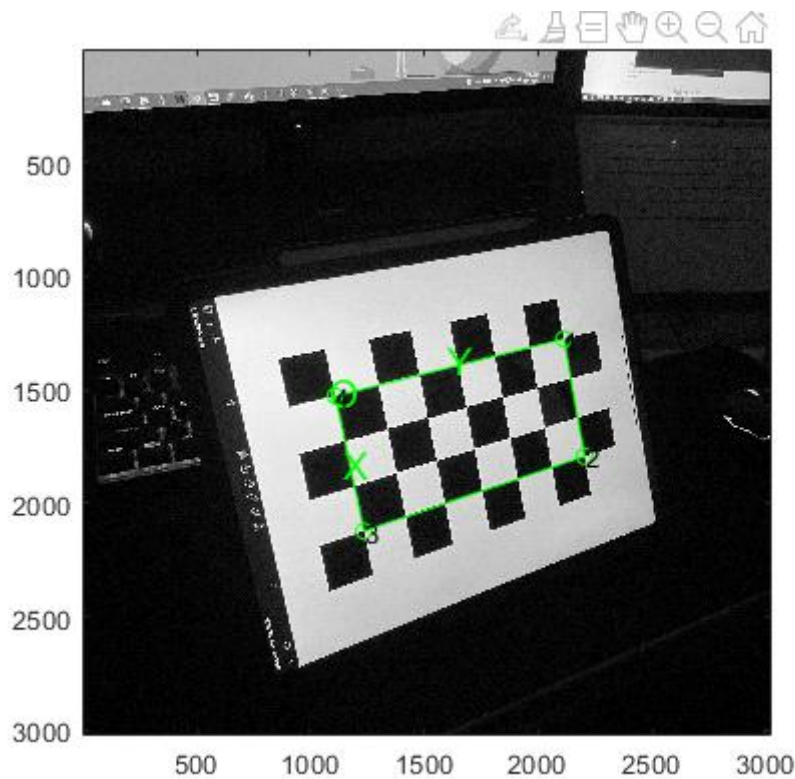
.      checker13.jpg checker19.jpg checker6.jpg
..     checker14.jpg checker2.jpg checker7.jpg
checker1.jpg checker15.jpg checker20.jpg checker8.jpg
checker10.jpg checker16.jpg checker3.jpg checker9.jpg
checker11.jpg checker17.jpg checker4.jpg
checker12.jpg checker18.jpg checker5.jpg

Basename camera calibration images (without number nor suffix): checker
Image format: ([]='r'='ras', 'b'='bmp', 't'='tif', 'p'='pgm', 'j'='jpeg', 'g'='jpeg', 'm'='ppm') j

Checking directory content for the calibration images (no global image loading in memory efficient mode)
Found images: 1...2...3...4...5...6...7...8...9...10...11...12...13...14...15...16...17...18...19...20...
done

To display the thumbnail images of all the calibration images, you may run mosaic_no_read (may be slow)
>>
```

위와 같이 MATLAB에서 calibration을 실행해서 직접 코너들을 찍어준다.



```

Processing image 18...
Loading image checker18.jpg...
Using (wintx,winty)=(32,32) - Window size = 65x65      (Note: To reset the window size, run script clearwin)
Click on the four extreme corners of the rectangular complete pattern (the first clicked corner is the origin)...
Size of each square along the X direction: dX=0.1524m
Size of each square along the Y direction: dY=0.1524m   (Note: To reset the size of the squares, clear the variables dX and dY)
If the guessed grid corners (red crosses on the image) are not close to the actual corners,
it is necessary to enter an initial guess for the radial distortion factor kc (useful for subpixel detection)
Need of an initial guess for distortion? ([]=no, other=yes)
Corner extraction...

Processing image 19...
Loading image checker19.jpg...
Using (wintx,winty)=(32,32) - Window size = 65x65      (Note: To reset the window size, run script clearwin)
Click on the four extreme corners of the rectangular complete pattern (the first clicked corner is the origin)...
Size of each square along the X direction: dX=0.1524m
Size of each square along the Y direction: dY=0.1524m   (Note: To reset the size of the squares, clear the variables dX and dY)
If the guessed grid corners (red crosses on the image) are not close to the actual corners,
it is necessary to enter an initial guess for the radial distortion factor kc (useful for subpixel detection)
Need of an initial guess for distortion? ([]=no, other=yes)
Corner extraction...

Processing image 20...
Loading image checker20.jpg...
Using (wintx,winty)=(32,32) - Window size = 65x65      (Note: To reset the window size, run script clearwin)
Click on the four extreme corners of the rectangular complete pattern (the first clicked corner is the origin)...
Size of each square along the X direction: dX=0.1524m
Size of each square along the Y direction: dY=0.1524m   (Note: To reset the size of the squares, clear the variables dX and dY)
If the guessed grid corners (red crosses on the image) are not close to the actual corners,
it is necessary to enter an initial guess for the radial distortion factor kc (useful for subpixel detection)
Need of an initial guess for distortion? ([]=no, other=yes)
Corner extraction...
done

```

20장의 사진 모두에 코너를 찍었다.

Aspect ratio optimized (est_aspect_ratio = 1) -> both components of fc are estimated (DEFAULT).
Principal point optimized (center_optim=1) - (DEFAULT). To reject principal point, set center_optim=0
Skew not optimized (est_alpha=0) - (DEFAULT)
Distortion not fully estimated (defined by the variable est_dist):
Sixth order distortion not estimated (est_dist(5)=0) - (DEFAULT) .
Initialization of the principal point at the center of the image.
Initialization of the intrinsic parameters using the vanishing points of planar patterns.

Initialization of the intrinsic parameters - Number of images: 20

Calibration parameters after initialization:

Focal Length: fc = [3085.26812 3085.26812]
Principal point: cc = [1511.50000 1511.50000]
Skew: alpha_c = [0.00000] => angle of pixel = 90.00000 degrees
Distortion: kc = [0.00000 0.00000 0.00000 0.00000 0.00000]

Main calibration optimization procedure - Number of images: 20

Gradient descent iterations: 1...2...3...4...5...6...7...8...9...10...11...12...13...14...15...16...17...18...19...20...done
Estimation of uncertainties...done

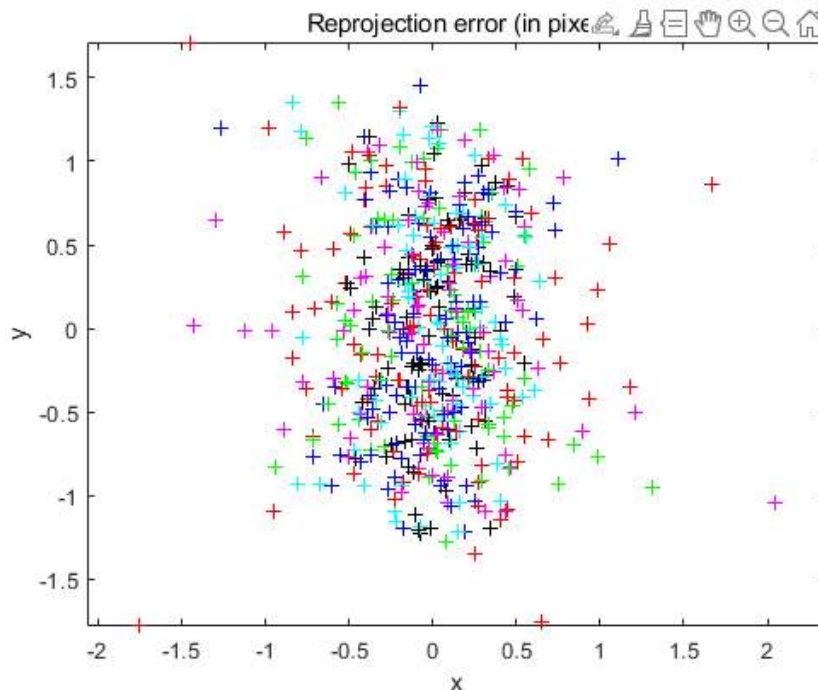
Calibration results after optimization (with uncertainties):

Focal Length: fc = [3045.65684 3046.24685] +/- [22.05522 20.96096]
Principal point: cc = [1524.26082 1521.85096] +/- [22.26216 15.48571]
Skew: alpha_c = [0.00000] +/- [0.00000] => angle of pixel axes = 90.00000 +/- 0.00000 degrees
Distortion: kc = [0.15531 -0.59817 0.00086 -0.00075 0.00000] +/- [0.02189 0.18551 0.00230 0.00344 0.00000]
Pixel error: err = [0.40485 0.63155]

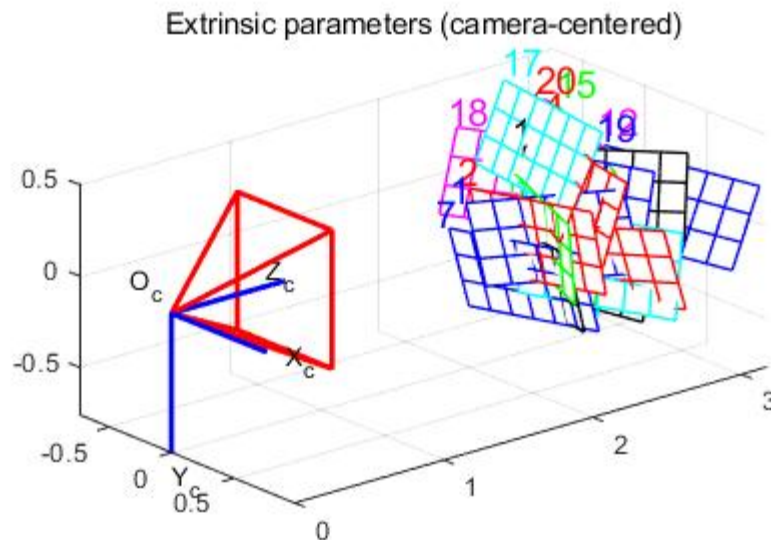
Note: The numerical errors are approximately three times the standard deviations (for reference).

Recommendation: Some distortion coefficients are found equal to zero (within their uncertainties).
To reject them from the optimization set est_dist=[1;1;0;0;0] and run Calibration

위와 같은 calibration 결과가 나왔다. focal length : [3045.65684, 3046.24685], principal point : [1524.26082, 1521.85096]



에러를 시각적으로 plot한 결과이다.



카메라를 기준으로 grid들을 3D로 plot한 모습이다.

- 결과 분석

Apple iPhone 12 Pro



Released 2020, October 23
 189g, 7.4mm thickness
 iOS 14.1, up to iOS 15
 128GB/256GB/512GB storage, no card slot

~14%
3,901,328 HITS

260
BECOME A FAN

6.1"
1170x2532 pixels

12MP
2160p

6GB RAM
Apple A14 Bionic

2815mAh
Li-Ion

MAIN CAMERA		Quad
		12 MP, f/1.6, 26mm (wide), 1.4µm, dual pixel PDAF, OIS
		12 MP, f/2.0, 52mm (telephoto), 1/3.4", 1.0µm, PDAF, OIS, 2x optical zoom
		12 MP, f/2.4, 13mm, 120° (ultrawide), 1/3.6"
		TOF 3D LiDAR scanner (depth)
Features		Dual-LED dual-tone flash, HDR (photo/panorama)
Video		4K@24/30/60fps, 1080p@30/60/120/240fps, 10-bit HDR, Dolby Vision HDR (up to 60fps), stereo sound rec.
SELFIE CAMERA		Dual
		12 MP, f/2.2, 23mm (wide), 1/3.6"
		SL 3D, (depth/biometrics sensor)
Features		HDR
Video		4K@24/30/60fps, 1080p@30/60/120fps, gyro-EIS

먼저 참조된 사이트에 들어가서 확인한 카메라의 정보이다. 후면 카메라로

촬영하였고 12Mpixel, f/1.6의 조리개 값 등등 정보가 쓰여있다. 내가 toolbox를 이용해 측정한 값인 focal length : [3045.65684, 3046.24685], principal point : [1524.26082, 1521.85096]을 분석해보겠다.

카메라		이미지	
카메라 제조업체	Apple	이미지 ID	
카메라 모델	iPhone 12 Pro	사진 크기	3024 x 3024
F-스톱	F/1.6	너비	3024픽셀
노출 시간	1/60초	높이	3024픽셀
ISO 감도	ISO-160	수평 해상도	72 DPI
노출 바이어스	0 단계	수직 해상도	72 DPI
초점 거리	4mm	비트 수준	24
조리개 최대 개방		압축	
측광 모드	패턴	해상도 단위	2
피사체 거리		색 대표	위치 조정 안 됨
플래시 모드	자동 플래시 끄	픽셀당 압축 비트	
플래시 에너지			
35mm 초점 거리	26		

먼저 위는 EXIF를 확인했을 때 나온 정보이다. principal point * 2를 했을 때 픽셀의 너비와 높이가 나와야한다. 값은 [3048.52164, 3043.70192] 이다. EXIF에서 픽셀 값은 3024 x 3024이다. 각각 오차는 24.52164, 19.70192로 오차범위 내에 속한다.

이제 focal length : [3045.65684, 3046.24685]를 분석해보기에 앞서 애플 공식 홈페이지에서 카메라에 대한 정보를 다시 확인했다.

카메라

- 프로급 12MP 카메라 시스템: 울트라 와이드, 와이드, 망원 카메라
- 울트라 와이드: f/2.4 조리개 및 120° 시야각
- 와이드: f/1.6 조리개
- 망원: f/2.0 조리개
- 2배 광학 줌인, 2배 광학 줌아웃, 4배 광학 줌 범위
- 최대 10배 디지털 줌
- LiDAR 스캐너를 활용한 야간 모드 인물 사진
- 향상된 보케 효과 및 심도 제어 기능을 지원하는 인물 사진 모드
- 6가지 효과의 인물 사진 조명(자연 조명, 스튜디오 조명, 윤곽 조명, 무대 조명, 무대 조명 모노, 하이키 모노)
- 듀얼 광학 이미지 흔들림 보정(OIS)(와이드, 망원)
- 5매(Five-element) 렌즈(울트라 와이드), 6매(Six-element) 렌즈(망원), 7매(Seven-element) 렌즈(와이드)
- 슬로 싱크 기능을 갖춘, 더욱 밝아진 True Tone 플래시
- 파노라마(최대 63MP)
- 사파이어 크리스탈 렌즈 커버
- 100% Focus Pixels(와이드)
- 야간 모드(울트라 와이드, 와이드)
- Deep Fusion(울트라 와이드, 와이드, 망원)
- 스마트 HDR 3
- Apple ProRAW
- 사진 및 Live Photo 촬영 시 넓은 색상 영역 포착
- 렌즈 보정(울트라 와이드)
- 첨단 적목 보정 기능
- 사진 위치 표시 기능
- 자동 흔들림 보정
- 고속 연사 모드
- 촬영 이미지 포맷: HEIF 및 JPEG

줌 인 / 줌 아웃을 사용하지 않은 일반 카메라로 찍었으므로 f/1.6의 조리개 값을 갖고 있다.

초점 거리를 계산해보면 focal length : [3045.65684, 3046.24685]에 픽셀 값인 1.4 μ m를 곱해보면 4.26392mm, 4.26475mm가 나온다. 이는 EXIF에 있는 초점거리 4mm와 오차범위 내에 있다.

결과를 분석해봤을 때 optimization이 잘 되었다고 볼 수 있다.

2번 :

1) 전체적인 코드 설명 (주석이 있지만 부가 설명)

- Gaussian Elimination

```
18 void Gaussian_Elimination_func(float* input, int n){ // 행렬로 이루어진 선형 방정식을 풀어주는 Gaussian Elimination 함수
19
20     float* A = input;
21     int i = 0;
22     int j = 0;
23     // m = rows, n = cols
24     int m = n - 1; // 8과 9 이므로
25     while (i < m && j < n)
26     {
27         // column j를 기준으로 잡아서 row i부터 시작
28         int maxi = i;
29         for (int k = i + 1; k < m; k++)
30         {
31             if (fabs(A[k * n + j]) > fabs(A[maxi * n + j]))
32             {
33                 maxi = k;
34             }
35         }
36
37         if (A[maxi * n + j] != 0)
38         {
39             // i와 maxi의 순서를 바꾼다. (위부터 큰 순서대로 sort 돼야 하기 때문)
40             if (i != maxi)
41             {
42                 for (int k = 0; k < n; k++)
43                 {
44                     float aux = A[i * n + k];
45                     A[i * n + k] = A[maxi * n + k];
46                     A[maxi * n + k] = aux;
47                 }
48             }
49             // A[i,j] 바꾸기 전 A[maxi,j]의 값을 가지고 있는 상태
50             // 행 i의 원소를 A[i,j]로 나눈다
51             float A_ij = A[i * n + j];
52             for (int k = 0; k < n; k++)
53             {
54                 A[i * n + k] /= A_ij;
55             }
56             // A[i,j] = 1이 됐을 것이다.
57             for (int u = i + 1; u < m; u++)
58             {
59                 // A[u,j]와 행 i를 곱한 값에서 행 u를 빼준다.
60                 float A_uj = A[u * n + j];
61                 for (int k = 0; k < n; k++)
62                 {
63                     A[u * n + k] -= A_uj * A[i * n + k];
64                 }
65                 // A[u,j] = 0 이다
66             }
67             i++;
68         }
69         j++;
70     }
71
72     // 다시 대입해준다
73     for (int i = m - 2; i >= 0; i--)
74     {
75         for (int j = i + 1; j < n - 1; j++)
76         {
77             A[i * n + m] -= A[i * n + j] * A[j * n + m];
78         }
79     }
```

행렬로 이루어진 선형 방정식을 풀기 위해 가우스 소거법을 위한 함수를 만들었다. Homography를 계산할 때 쓰이며 가장 자주 쓰이면서도 간단한 알고리즘을 사용했다.

- Find Homography

```

79 void findHomography_func(vector<Point2f> src, vector<Point2f> dst, Mat & result){ // Homography를 찾아주는 함수
80
81     float P[8][9] = // 선형 방정식의 앞쪽 행렬이다.
82     {
83         {-src[0].x, -src[0].y, -1, 0, 0, 0, src[0].x * dst[0].x, src[0].y * dst[0].x, -dst[0].x },
84         { 0, 0, 0, -src[0].x, -src[0].y, -1, src[0].x * dst[0].y, src[0].y * dst[0].y, -dst[0].y },
85
86         {-src[1].x, -src[1].y, -1, 0, 0, 0, src[1].x * dst[1].x, src[1].y * dst[1].x, -dst[1].x },
87         { 0, 0, 0, -src[1].x, -src[1].y, -1, src[1].x * dst[1].y, src[1].y * dst[1].y, -dst[1].y },
88
89         {-src[2].x, -src[2].y, -1, 0, 0, 0, src[2].x * dst[2].x, src[2].y * dst[2].x, -dst[2].x },
90         { 0, 0, 0, -src[2].x, -src[2].y, -1, src[2].x * dst[2].y, src[2].y * dst[2].y, -dst[2].y },
91
92         {-src[3].x, -src[3].y, -1, 0, 0, 0, src[3].x * dst[3].x, src[3].y * dst[3].x, -dst[3].x },
93         { 0, 0, 0, -src[3].x, -src[3].y, -1, src[3].x * dst[3].y, src[3].y * dst[3].y, -dst[3].y },
94     };
95
96     Gaussian_Elimination_func(P[0][0], 9); // 해당 선형 방정식을 가우스 소거법을 사용해서 풀어준다.
97
98     float aux_H[3][3] = { // 소거법을 활용해 풀어준 결과 행렬
99         { P[0][8], P[1][8], P[2][8] },
100        { P[3][8], P[4][8], P[5][8] },
101        { P[6][8], P[7][8], 1 }
102    };
103
104    for (int i = 0; i < 3; i++){
105        for (int j = 0; j < 3; j++){
106            result.at<float>(i, j) = aux_H[i][j]; // 결과 Mat에 대입해줘서 출력
107        }
108    }
109 }

```

이 전에 설명한 가우스 소거법을 이용해 Homography를 계산해주는 함수이다. 8개의 점이 필요한 Homography 계산이므로 8 x 9의 행렬을 이용해 3 x 3의 Homography를 계산한다. 행렬의 H11, H12 ... 등 원소들의 올바른 위치에 계산한 값을 넣어준다.

- RANSAC

```

112 Mat RANSAC_func(vector<Point2f> src, vector<Point2f> dst){ // RANSAC을 수행해주는 함수
113
114     int N = 2000; // 반복 횟수
115     float T = 3; // threshold
116     int size = src.size(); // 입력 keypoint들의 개수
117
118     int max_cnt = 0; // 가장 많은 inlier의 개수를 담아줄 변수
119
120     Mat Best_Homo = Mat::zeros(3, 3, CV_32FC1); // 가장 많은 inlier를 뽑아낸 호모그래피를 저장해줄 Mat 초기화
121
122     for (int i = 0; i < N; i++) // N번 반복해주는 for문
123     {
124         int k[4] = { -1, };
125         k[0] = floor((rand() % size)); // 4개씩 총 keypoint의 개수 범위 내에서 만큼 난수를 생성해주는 난수 생성기
126         do
127         {
128             k[1] = floor((rand() % size));
129             while (k[1] == k[0] || k[1] < 0);
130
131             do
132             {
133                 k[2] = floor((rand() % size));
134                 while (k[2] == k[0] || k[2] == k[1] || k[2] < 0);
135
136                 do
137                 {
138                     k[3] = floor((rand() % size));
139                     while (k[3] == k[0] || k[3] == k[1] || k[3] == k[2] || k[3] < 0);

```



```

141 printf("random sample : %d %d %d %d\n", k[0], k[1], k[2], k[3]); // 몇번째 특징점들이 뽑혔는지 출력
142
143 vector<Point2f> src_sample; // 뽑힌 번호들의 특징점(x)들을 담아서 컨테이너
144 vector<Point2f> dst_sample; // 뽑힌 번호들의 특징점(x')들을 담아서 컨테이너
145
146 src_sample.push_back(src[k[0]]);
147 src_sample.push_back(src[k[1]]);
148 src_sample.push_back(src[k[2]]);
149 src_sample.push_back(src[k[3]]);
150
151 dst_sample.push_back(dst[k[0]]);
152 dst_sample.push_back(dst[k[1]]);
153 dst_sample.push_back(dst[k[2]]);
154 dst_sample.push_back(dst[k[3]]);
155
156 Mat Homo_sample = Mat::zeros(3, 3, CV_32FC1); // 뽑힌 특징점들로 만든 호모그래피를 담은 3 x 3 매트릭스 초기화 및 선언
157
158 FindHomography_func(src_sample, dst_sample, Homo_sample); // find homography 함수 선언
159
160 Mat Distance = Mat::zeros(size, 1, CV_32FC1); // 호모그래피의 거리를 계산한 값들을 담은 (size) x 1 매트릭스
161
162 for (int i = 0; i < size; i++) { // 특징점 개수만큼 반복
163     Mat A_calc = Mat::zeros(3, 1, CV_32FC1); // 특징점을 Mat으로 옮기는 것과 동시에 homogeneous coordinate로 변환해서 담은 3 x 1 매트릭스
164     Point2f A_calc_Homo; // H x X 를 계산하여 다시 cartesian coordinate의 변환해서 담아둘 변수
165
166     A_calc.at<float>(0, 0) = src[i].x;
167     A_calc.at<float>(1, 0) = src[i].y;
168     A_calc.at<float>(2, 0) = 1.0f; // homogeneous coordinate로 변환
169
170     Mat H_mul_X = (3, 1, CV_32FC1, Homo_sample * A_calc); // H x X 계산
171
172     A_calc_Homo.x = H_mul_X.at<float>(0, 0) / H_mul_X.at<float>(2, 0); // (wx, wy, w)에서 wx / w를 계산해서 x 값 삽입
173     A_calc_Homo.y = H_mul_X.at<float>(1, 0) / H_mul_X.at<float>(2, 0); // (wx, wy, w)에서 wy / w를 계산해서 y 값 삽입
174
175     Distance.at<float>(i, 0) = norm(dst[i] - A_calc_Homo); // norm 함수를 이용해서 거리 계산
176 }
177
178 int cnt = 0; // inlier의 개수를 저장해줄 변수
179
180 for (int j = 0; j < size; j++)
181 {
182     float data = Distance.at<float>(j, 0); // data에 거리 옮기기
183     if (data < T) // threshold 와 비교
184     {
185         cnt++;
186     }
187 }
188
189 if (cnt > max_cnt) // 가장 많은 inlier의 개수를 가진 호모그래피 저장
190 {
191     Best_Homo = Homo_sample;
192     max_cnt = cnt;
193 }
194
195
196 cout << "Max inliers : " << max_cnt << endl;
197 cout << "Best Homography : " << Best_Homo << endl << endl;
198
199 return Best_Homo;
200 }

```

Homography의 Robust Estimation을 위해 RANSAC 함수를 만들었다. 정의해야 할 parameter로는 반복 횟수와 error threshold가 있다. 이들은 최적의 효과를 내는 파라미터로 설정하였다. 먼저 반복 횟수만큼 for문을 설정한다. for문 안에서 난수 생성기를 이용해 4개의 난수를 만들어서 특징점 4쌍을 선정한다. 이렇게 8개의 점으로 Homography를 구하고 $X' = HX$ 계산을 이용해 모든 특징점 쌍 끼리의 오차율을 구한다. 그 오차율을 이용해 threshold보다 작으면 inlier로 세어준다. 이렇게 섀 inlier가 가장 많은 Homography를 Best_Homo라는 Mat로 반환해준다.

- Make Panorama

```

201
202 Mat makePanorama_Func(Mat Left_Img, Mat Right_Img, int Distance_Thresh, int Match_Min) { // 입력된 두 이미지를 병합해주는 함수
203
204     Mat Left_Img_Gray, Right_Img_Gray; // Grayscale로 변환한 이미지를 담을 매트릭스
205
206     cvtColor(Left_Img, Left_Img_Gray, CV_BGR2GRAY); // Grayscale로 변환
207     cvtColor(Right_Img, Right_Img_Gray, CV_BGR2GRAY); // Grayscale로 변환
208
209     // -----SIFT 특징 검출기-----
210
211     Ptr<SiftFeatureDetector> Detector = SIFT::create(300);
212     vector<KeyPoint> kpts_left, kpts_right;
213     Detector->detect(Left_Img_Gray, kpts_left);
214     Detector->detect(Right_Img_Gray, kpts_right);
215
216     Ptr<SiftDescriptorExtractor> Extractor = SIFT::create(100, 4, 3, false, true);
217     Mat img_des_left, img_des_right;
218     Extractor->compute(Left_Img_Gray, kpts_left, img_des_left);
219     Extractor->compute(Right_Img_Gray, kpts_right, img_des_right);
220
221     BFMatcher matcher(NORML2); // Brute Force Matcher — L2 norm 사용
222     vector<DMatch> matches;
223     matcher.match(img_des_left, img_des_right, matches);
224
225     double Distance_Max = matches[0].distance; // Match들의 거리를 통해 정제
226     double Distance_Min = matches[0].distance;
227     double dist;
228     for (int i = 0; i < img_des_left.rows; i++) {
229         dist = matches[i].distance;
230         if (dist < Distance_Min) Distance_Min = dist;
231         if (dist > Distance_Max) Distance_Max = dist;
232     }
233     printf("max_dist : %f\n", Distance_Max);
234     printf("min_dist : %f\n", Distance_Min);
235
236     vector<DMatch> Matched_Fin;
237     do {
238         vector<DMatch> Matched_Good;
239         for (int i = 0; i < img_des_left.rows; i++) {
240             if (matches[i].distance < Distance_Thresh + Distance_Min)
241                 Matched_Good.push_back(matches[i]);
242         }
243         Matched_Fin = Matched_Good;
244         Distance_Thresh -= 1;
245     } while (Distance_Thresh != 2 && Matched_Fin.size() > Match_Min);
246
247     vector<Point2f> left, right; // Match가 끝난 페어들의 좌표를 각각 left와 right 벡터에 저장
248     for (int i = 0; i < Matched_Fin.size(); i++) {
249         left.push_back(kpts_left[Matched_Fin[i].queryIdx].pt);
250         right.push_back(kpts_right[Matched_Fin[i].trainIdx].pt);
251     }
252     cout << "Number of Keypoints : " << right.size() << ", " << left.size() << endl; // 특징점 총 개수 출력
253
254     //Mat mat_homo = findHomography(left, right, RANSAC); // built-in과 나의 함수 비교
255
256     Mat mat_homo = RANSAC_Func(left, right); // Homography를 만들어줌과 동시에 RANSAC algorithm을 이용해 가장 좋은 Homography를 찾아주는 함수
257
258     // cout << "Built-in Homography : " << mat_homo_1 << endl << endl; // built-in과 나의 함수 비교
259
260     Mat Result_Img;
261     warpPerspective(Left_Img, Result_Img, mat_homo, // Inverse Warping를 통해 projection 실행
262         Size(Left_Img.cols + 3, Left_Img.rows + 1.2), INTER_CUBIC);
263
264     Mat Pano_Img;
265     Pano_Img = Result_Img.clone();
266     Mat roi(Pano_Img, Rect(0, 0, Right_Img.cols, Right_Img.rows)); // 이미지 합성
267     Right_Img.copyTo(roi);
268
269     int cut_x = 0, cut_y = 0;
270     for (int y = 0; y < Pano_Img.rows; y++) { // Black Cut - 이미지에서 검은색 부분을 잘라주는 코드
271         for (int x = 0; x < Pano_Img.cols; x++) {
272             if (Pano_Img.at<Vec3b>(y, x)[0] == 0 &&
273                 Pano_Img.at<Vec3b>(y, x)[1] == 0 &&
274                 Pano_Img.at<Vec3b>(y, x)[2] == 0) {
275                 continue;
276             }
277             if (cut_x < x) cut_x = x;
278             if (cut_y < y) cut_y = y;
279         }
280     }
281     Mat Pano_Img_Fin;
282     Pano_Img_Fin = Pano_Img(Range(0, cut_y), Range(0, cut_x));
283
284     return Pano_Img_Fin;
285

```

두 영상을 받아서 파노라마를 만들어주는 함수이다. 영상들을 Grayscale로

변환하고 그 영상들을 SIFT feature extractor를 이용해 특징을 검출한다. 검출한 특징들을 Brute Force(L2 norm)를 이용해서 매칭을 해주어 쌍을 만들어준다. 그 쌍들을 거리를 측정해서 threshold를 이용해서 정제해준다. 이후 그 특징점들을 RANSAC 함수에 넣어서 Homography를 추출한다. Homography를 이용해서 InverseWarping과 Stitching, 그리고 Black Cut을 진행한다.

- Panorama + main

```
286 void Panorama_func() { // 파노라마를 전체적으로 진행해주는 함수
287
288
289     Mat Image_left = imread("home1.jpg", IMREAD_COLOR); // 세장의 사진 불러오기
290     Mat Image_center = imread("home2.jpg", IMREAD_COLOR);
291     Mat Image_right = imread("home3.jpg", IMREAD_COLOR);
292
293     resize(Image_left, Image_left, Size(512, 512), INTER_AREA); // 모두 같은 사이즈로 다운 샘플링
294     resize(Image_center, Image_center, Size(512, 512), INTER_AREA);
295     resize(Image_right, Image_right, Size(512, 512), INTER_AREA);
296
297     if (Image_left.empty() || Image_center.empty() || Image_right.empty()) exit(-1); // 위 내용이 제대로 진행 안되었으면 종료
298
299     Mat result; // 결과를 담아줄 매트릭스
300
301     flip(Image_left, Image_left, 1); // 왼쪽 사진 뒤집기
302     flip(Image_center, Image_center, 1); // 가운데 사진 뒤집기
303
304     result = makePanorama_func(Image_left, Image_center, 4, 100); // 왼쪽과 가운데 먼저 병합
305
306     flip(result, result, 1); // 왼쪽 + 가운데 다시 뒤집기
307     imshow("ex_panorama_result1", result);
308     imwrite("ex_panorama_result1.png", result);
309
310     waitKey();
311
312     result = makePanorama_func(Image_right, result, 2, 60); // (왼쪽 + 가운데) 와 오른쪽 병합
313
314     imshow("ex_panorama_result2", result);
315     imwrite("ex_panorama_result2.png", result);
316
317     waitKey();
318
319     destroyAllWindows();
320 }
321
322 int main() {
323     Panorama_func();
324 }
```

파노라마와 메인 함수이다. 메인 함수를 줄이기 위해 그 역할을 해줄 파노라마 함수를 만들었다. 영상을 불러오고 다운 샘플링, 그리고 make panorama 함수 실행까지 한다.

2) 결과

총 6장의 사진으로 진행했다. 내가 찍은 3장의 사진과 좀 더 실험적으로 parameter들을 바꿔가며 해보고 싶어서 같은 수업을 듣는 최재혁 학우에게 부탁해서 사진을 3장 더 받았다.

먼저 내가 찍은 사진부터 결과를 분석해보겠다.

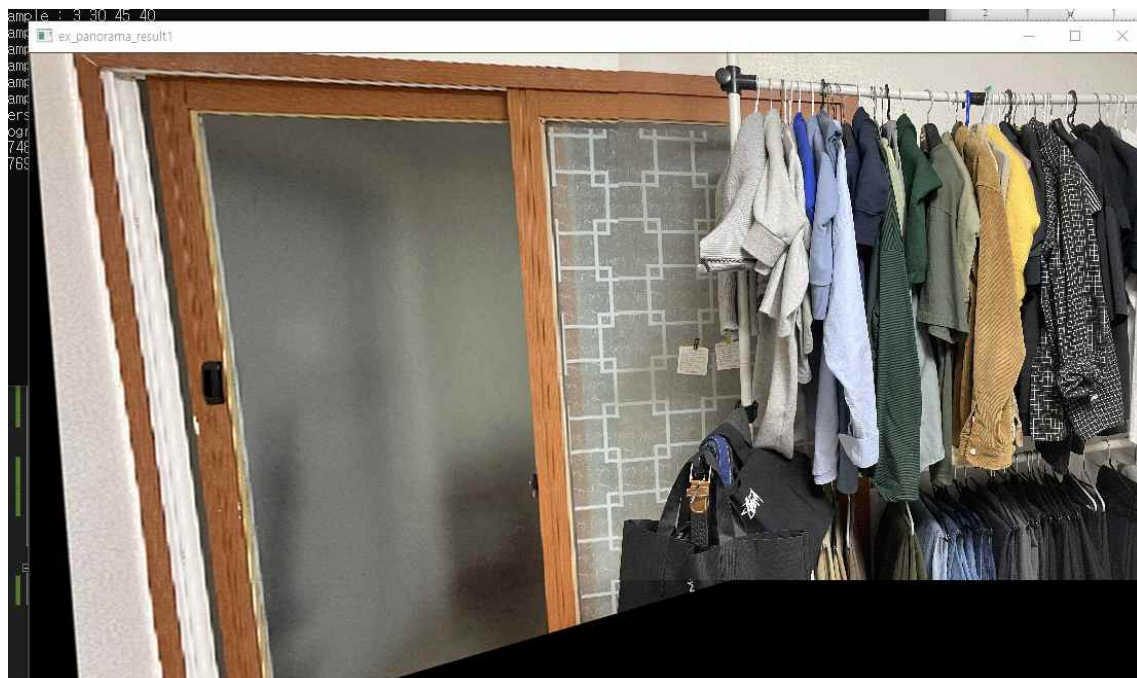
- 1 -

```
Number of Keypoints : 46, 46  
random sample : 41 21 32 4
```

먼저 왼쪽, 가운데 사진에서 찾은 특징점의 개수이다.

```
Max inliers : 39  
Best Homography : [0.56231678, -0.019860078, 316.70612;  
-0.18117487, 0.93977684, 12.76729;  
-0.00087697909, 4.9443221e-05, 1]
```

최대 Inlier의 개수는 46개 중에 39개가 나왔다. 그 39개를 갖는 최적의 Homography 또한 출력해봤다.



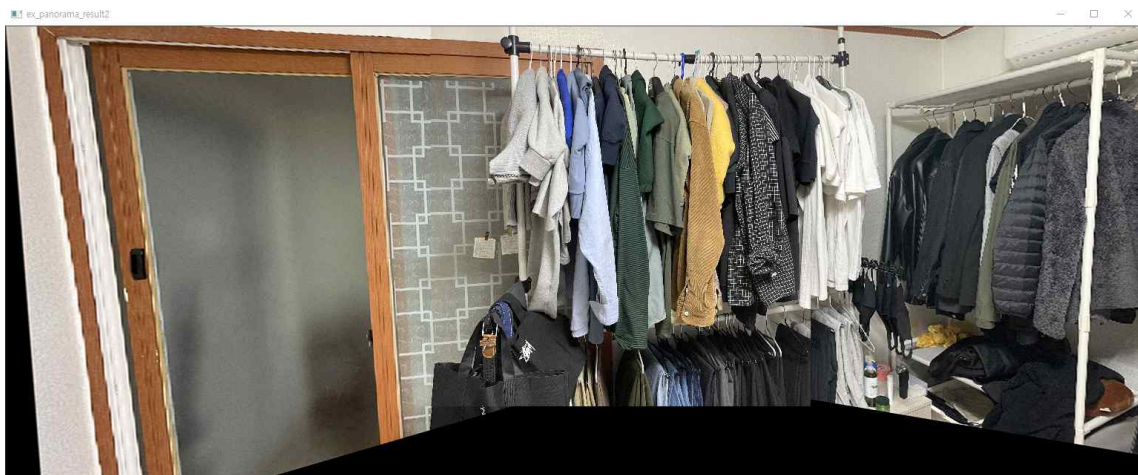
그렇게 해서 얻은 결과 값이다. 꽤나 자연스럽게 Stitching이 된 것을 확인할 수 있다.


```
Number of Keypoints : 19, 19  
random sample : 14 15 3 4
```

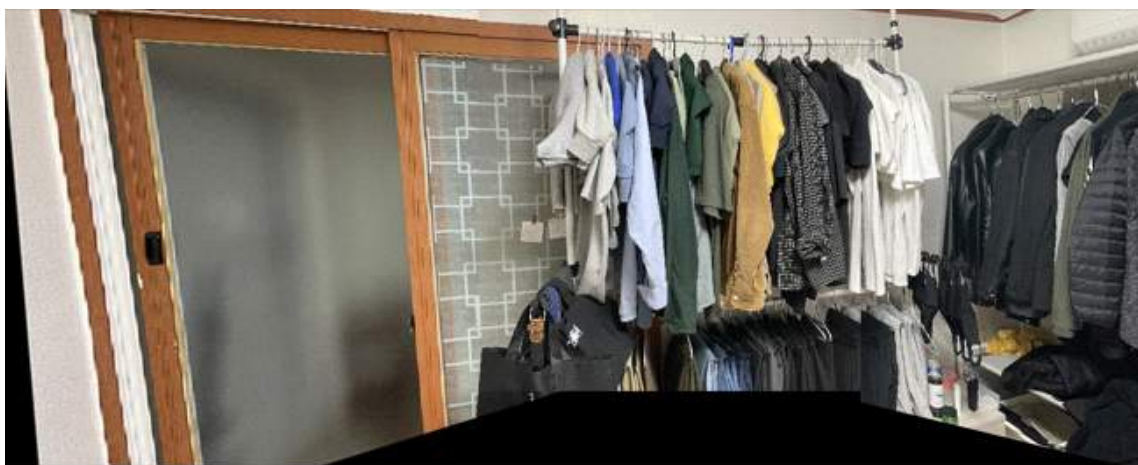
다음 특징점은 19개가 나왔다.

```
Max inliers : 11  
Best Homography : [0.11639965, 0.10480632, 938.69507;  
-0.21891052, 0.93567783, 18.165211;  
-0.00074633432, 6.2492159e-05, 1]
```

그 중에 찾은 Inlier의 개수와 최적의 Homography이다.



최종적으로 나온 결과이다. 오른쪽 사진은 약간 부자연스럽게 연결되었다.
이를 보고 내 함수가 이상한가 싶어 Built-in 함수로도 실험해보았다.



이것이 built-in find_homography로 만든 사진이다. 이것도 부자연스러운
것을 봐서 특징 검출기에서 특징을 적게 찾은 이유인 것 같다. 하지만 특징의
개수를 늘리려고 distance threshold를 널널하게 하면 너무 많아지면서

정확도가 더 떨어진다. 그래서 최재혁 학우에게 받은 사진으로 다시 결과를 체크해보겠다.



이것이 결과이다. 이것 또한 오른쪽 아래가 약간 부자연스러운 것을 확인할 수 있는데, 그 부분을 제외한 나머지는 거의 티가 안 날 정도로 자연스럽다. 오른쪽 아래에서 부자연스러운 것은 왼쪽 사진을 먼저 병합하고 오른쪽 사진을 진행하는데 이 과정에서 사진 픽셀 값이 늘어나고 검은 부분이 늘어서 특징을 검출하는 데에 어려움이 생긴 것으로 예상된다.

그래도 직접 만든 함수치고 결과가 상당히 괜찮게 나왔다고 생각한다. 실습을 해본 적이 없어 매우 어려운 과제였지만 굉장히 흥미롭고 재밌게 한 것 같다.