Image Stitching

이재형

국민대학교 전자공학부

harry8121@kookmin.ac.kr

요 약

Image Stitching과정에서 Feature Detection, Description, Matching, RANSAC, Blending등의 다양한 방법론과 함수 파라미터 조정을 통한 정합 및 Stitching결과를 개선하기 위한 다양한 관점에서의 고찰과 그에 따른 성능 개선 효과에 대하여 기술하겠습니다.

I. 서론

Image Stitching 과정에서 4가지의 관점에 따라 분류해보았습니다.

첫 번째, Feature Detection 및 Description.
SIFT알고리즘을 기반으로 특징점 추출과 기술자를 생성하였는데, openCV에서 제공하는 SIFT함수의 파라미터 조정이 이미지 스티칭 성능에 어떠한 영향을 끼칠 수 있는지 알아보았습니다.

두 번째, Matcher 알고리즘 변경.
Matching의 방법에도 다양한 방법이 존재하는데, FLANN Matching, BFMatching 등 다양한 Matcher알고리즘을 사용하여 이들 간의 성능을 비교해보았습니다.

세 번째, RANSAC 파라미터 조정.
임계값을 특정 장면에 적합한 값으로 변경해보고, 데이터에 따라 임계값의 어떤 차이를 띄는지 분석해보았습니다.

네 번째, 블렌딩 방식 변경.
두 이미지나 영상이 매우 다를 경우, 경계부분에서 가시적인 경계선이 발생할 수 있는데, 이를 더 부드러운 블렌딩 방식을 통해서 해결하고자 했습니다.

II. 과제 수행 내용

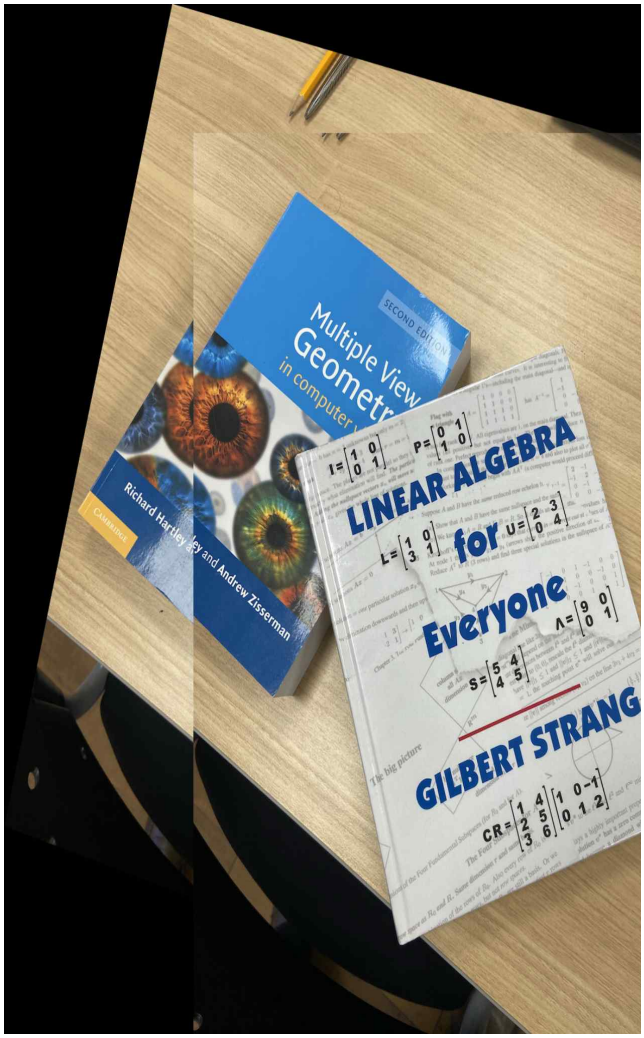
서론에서 제시된 4가지 방식을 순서로 기술하겠습니다.

2.1 Feature Detection & Description

SIFT 알고리즘 기반으로 특징점을 추출하고, 기술자를 생성하고, FLANN Matcher를 통해 Matching을 진행했습니다.

```python
def get_transform_from_keypoints(img_src, img_dst):
  img_src_gray = cv2.cvtColor(img_src, cv2.COLOR_BGR2GRAY)
  img_dst_gray = cv2.cvtColor(img_dst, cv2.COLOR_BGR2GRAY)

  # Create a SIFT object and detect keypoints and descriptors for each image
  ### CODE YOURSELF ###
  sift = cv2.SIFT_create()
  kpts_src, dscrpt_src = sift.detectAndCompute(img_src_gray, None)
  kpts_dst, dscrpt_dst = sift.detectAndCompute(img_dst_gray, None)
  print(dscrpt_src, dscrpt_dst)

  # Match the descriptors
  ### CODE YOURSELF ###

  # FLANN - 인덱스 파라미터 & 검색 파라미터
  index_params = dict(algorithm=0, trees=5)
  search_params = dict(checks=50)

  # Matching
  matcher = cv2.FlannBasedMatcher(index_params, search_params)
  matches = matcher.match(dscrpt_src, dscrpt_dst)
  #print(matches)

  # Matching 그리기
  '''
  res = cv2.drawMatches(img_src, kpts_src, img_dst, kpts_dst, matches, None, flgas=cv2.DRAW_MATCHE

  cv2.imshow('Flann + SIFT', res)
  cv2.waitKey(0)-
  cv2.destroyAllWindows()

  '''

  # Find the homography matrix using the matched keypoints
  ### CODE YOURSELF ###
  src_pts = np.float32([kpts_src[m.queryIdx].pt for m in matches]).reshape(-1, 1, 2)
  dst_pts = np.float32([kpts_dst[m.trainIdx].pt for m in matches]).reshape(-1, 1, 2)
  H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
  matches = [matches[i] for i in range(len(matches)) if mask[i]]

  H, _ = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)

  # return outputs
  return H, kpts_src, dscrpt_src, kpts_dst, dscrpt_dst, matches
```

```python
def get_stitched_image(img_src, img_dst, H):
    # Compute the size of the output stitched image
    ### CODE YOURSELF ###

    # 두 이미지 행, 열 크기 가져오기
    rows_src, cols_src = img_src.shape[:2]
    rows_dst, cols_dst = img_dst.shape[:2]

    # 첫 번째 이미지 꼭지점 좌표 생성
    corners_src = np.float32([[0, 0], [0, rows_src], [cols_src, rows_src], [cols_src, 0]]).reshape(-1, 1,

    # H변환행렬을 통한 1st 이미지의 꼭지점좌표를 2nd 이미지의 좌표공간으로 변환
    corners_dst = cv2.perspectiveTransform(corners_src, H)

    # 2nd 이미지의 꼭지점 좌표와 1st 이미지의 꼭지점 좌표 합치기
    corners = np.concatenate((corners_dst, corners_src), axis=0)

    # 변환된 자표들 중 최대/최소 x, y좌표 구하기
    [x_min, y_min] = np.int32(corners.min(axis=0).ravel() - 0.5)
    [x_max, y_max] = np.int32(corners.max(axis=0).ravel() + 0.5)

    # 음수값을 보정하기 위한 offset 설정
    offset_x = -x_min if x_min < 0 else 0
    offset_y = -y_min if y_min < 0 else 0

    # 출력 이미지 너비와 높이 계산
    dst_width = max(x_max, cols_dst) - min(x_min, 0)
    dst_height = max(y_max, rows_dst) - min(y_min, 0)

    # Modify H to account for the image offset - 변환행렬 오프셋 고려
    ### CODE YOURSELF ###
    H_modified = np.array([[1, 0, offset_x], [0, 1, offset_y], [0, 0, 1]]) @ H

    # Warp the first image to the perspective of the second image - img_src를 2nd 이미지의 원근 변환에 맞게
    ### CODE YOURSELF ###
    warped_img = cv2.warpPerspective(img_src, H_modified, (dst_width, dst_height))

    # Combine the two images to create a single stitched image
    ### CODE YOURSELF ###
    stitched_image = np.copy(warped_img)
    stitched_image[offset_y:rows_dst + offset_y, offset_x:cols_dst + offset_x] = img_dst

    # return output
    return stitched_image
```



그 후에 이미지 정합 함수를 통해 스티칭한 결과는 오른쪽 위 사진과 같았습니다. 그렇다면 SIFT 함수의 파라미터에는 어떠한 것들이 있고, 각 파라미터 들이 어떠한 역할을 하고 값을 변경함으로써 어떠한 부분에서 차이점이 생기는지 분석해보았습니다.

https://docs.opencv.org/3.4/d7/d60/classcv_1_1SIFT.html

위 링크의 SIFT menber function documentation을 참조하여 'SIFT_create()'함수의 파라미터를 살펴보았습니다.

1. nfeatures : 검출 최대 특징 수

2. nOctaveLayers : 이미지 피라미드에 사용할 계층 수 -> 계층 수는 이미지의 화질에 따라 자동적으로 설정됨.

3. contrastThreshold : 필터링할 빈약한 특징 문턱 값 -> 이 문턱 값은 필터링이 적용될 때 nOctaveLayers로 나뉨.

◆ create() [2/2]

```
static Ptr<SIFT> cv::SIFT::create ( int    nfeatures,
                                    int    nOctaveLayers
                                    double contrastThresh
                                    double edgeThreshold
                                    double sigma,
                                    int    descriptorType
                                  )
```

4. edgeThreshold : 필터링할 엣지 문턱 값 -> edge-like 특징들을 필터링할 때의 문턱 값이다 문턱 값이 클수록 필터링되는 특징들이 작아져서 많은 특징들을 얻을 수 있게 된다.

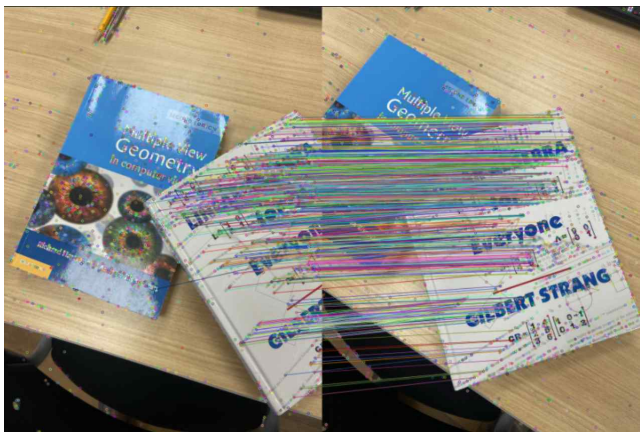
5. sigma : 이미지 피라미드 0계층에서 사용할 가우시안 필터의 시그마 값 -> 소프트 렌즈가 장착된 성능이 좋지 않은 카메라로 얻은 이미지인 경우 숫자를 줄이는 것이 좋다고 한다.

위와 같이 각 파라미터의 특징들을 기술해보았고, 이미지 스티칭에 영향을 줄만한 파라미터를 생각해보았는데, 'edgeThreshold값을 키워서 특징점이 조금 더 많이 추출된다면 영향이 있지 않을까?'라고 생각해보았습니다.

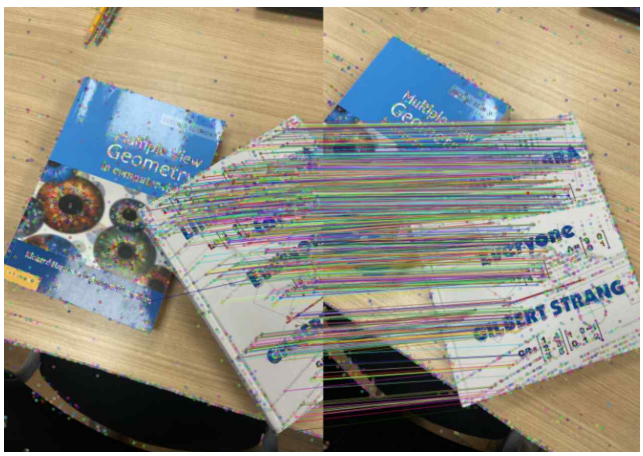
또한, 저화질 사진 2장을 가져와서 sigma값을 조정을 통해서 sigma값이 높을 때와 낮을 때 각각의 성능을 비교해보고자 했습니다.(저화질 사진을 찾기가 쉽지않아서 생략하겠습니다..)

* edgeThreshold 조절

- edgeThreshold 값을 10 -> 50으로 조정하여 matching과 stitching을 각각 확인해보았습니다.

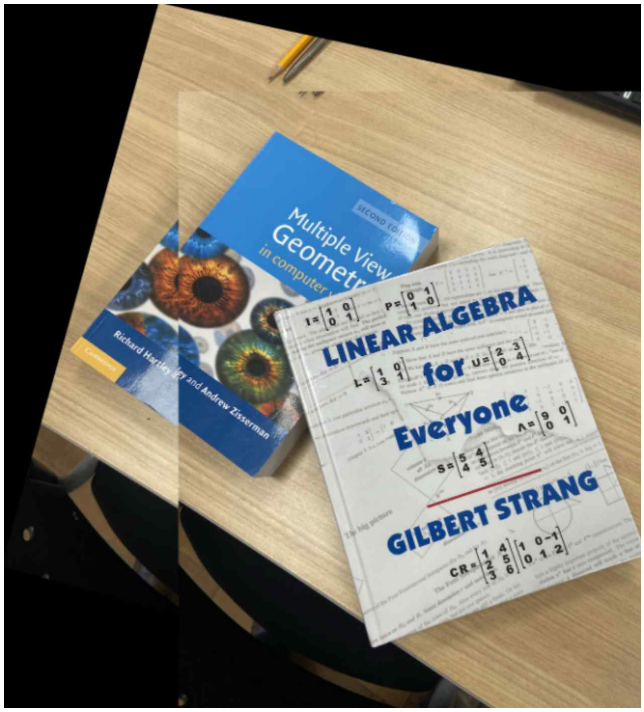


edgeThreshold = 10

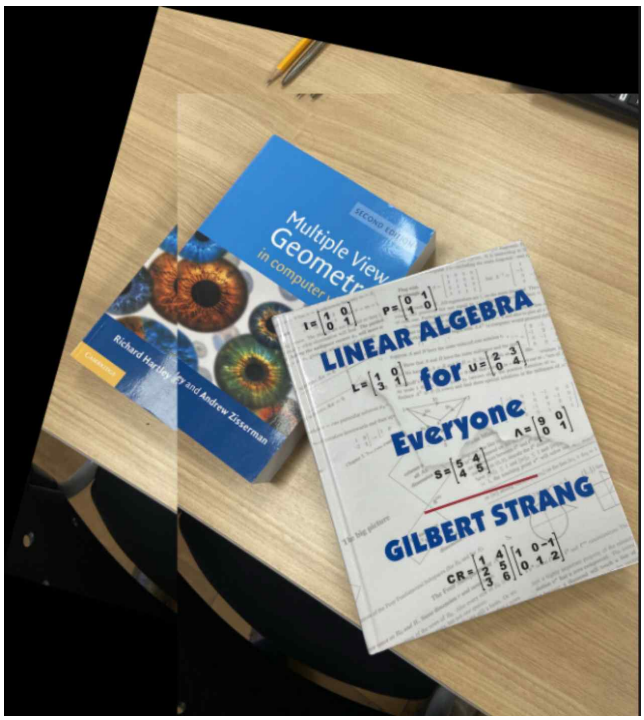


edgeThreshold = 50

먼저, matching된 결과 사진을 통해 임계값을 50으로 키웠을 때 조금 더 많은 특징점들이 추출되어 matching되는 것을 확인할 수 있었습니다. 아주 미세한 차이지만 위의 두사진에서 흰색 책의 아래 부분을 보면 아래에 있는 사진에서 matching이 더 많이 되었음을 육안으로 확인할 수 있습니다.



edgeThreshold = 10



edgeThreshold = 50

하지만, stitcing결과에서는 육안으로 확인할 수 있는 큰 차이점이 존재하지는 않았습니다. 이론적으로 추출된 특징점의 수가 많을 수록 이미지의 노이즈나 이상치에 더 강건하게 대응할 수 있고, 더 많은 영역을 커버할 수 있게 되어 이미지 스티칭의 성능이 향상된다는 것을 알고 있지만 육안으로 확인되지 않은 점이 아쉬웠습니다.

2.2 Matcher알고리즘

openCV에서 제공하는 Matching함수에 대해서 조사해보았습니다.

https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html

위의 링크를 통해 matching방법에 크게 2가지가 있다는 것을 알게되었습니다.

1. FLANN based Matcher

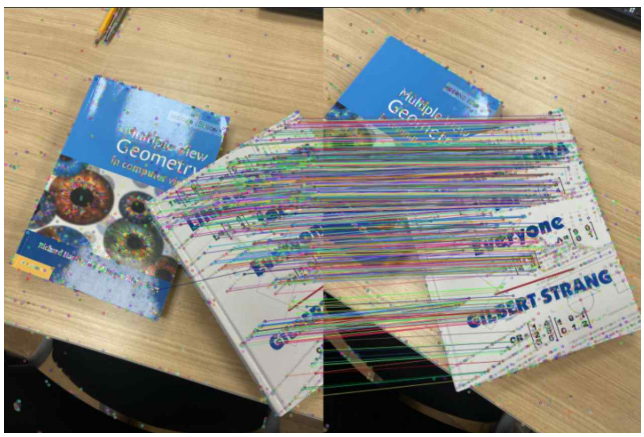
- FLANN은 Fast Library for Approximate Nearest Neighbors로, 대규모 데이터셋에서 빠른 최근접 이웃 탐색 및 고차원 기능에 최적화된 알고리즘이 포함되어 있습니다. 대규모 데이터셋의 경우 두 번째 matching방법인 BFMatcher보다 빠르게 작동합니다.

- 알고리즘에 사용에 관련된 파리미터들을 2개의 dict형태로 보내주어야 합니다.
하나는 indexParams = dict(algorithm=0, trees=5) 이고,
다른 하나는 searchParams = dict(checks=50)입니다.

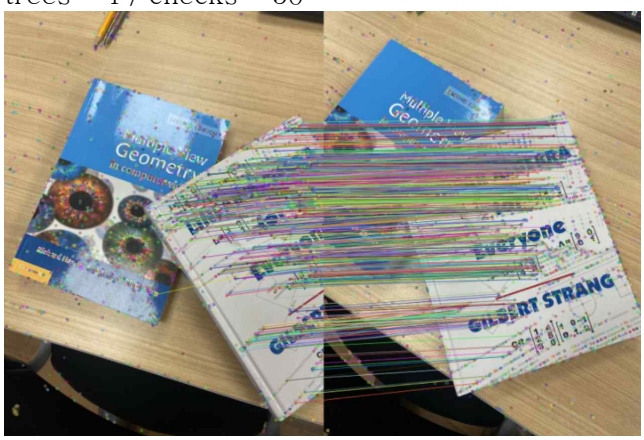
- 첫 번째 딕셔너리의 파라미터인 trees는 트리 기반 인덱싱 방식을 사용하여 근사적인 최근접 이웃을 검색할 때 트리의 개수를 결정합니다. ->
더 많은 트리를 사용하여 근사적인 최근접 이웃 검색이 더 정교해질 수 있지만, 과도한 트리 개수는 계산 비용을 증가시킬 수 있으므로 상황에 맞게 조절해야 합니다.

- 두 번째 딕셔너리 searchParams는 인덱스 내에서 트리가 얼마만큼 반복하며 왔다갔다해야하는 지를 정해줍니다. 높은 값일수록 정확도가 높지만, 연산 시간이 더 오래걸린다는 점이 있습니다.

- >> 결론적으로 indexParams와 searchParams의 파라미터들을 조정하여 matching의 정확도를 더 높이거나 낮추는 방향으로 조절하여 이미지 스티칭의 결과를 비교해볼 수 있을 것이라고 생각했습니다.



trees = 4 / checks = 50



trees=10 / checks = 100

2. Brute-Force Matcher

- Brute-Force는 추출된 특징점을 비교하여 해당 특징점들 간의 거리를 계산하고, 거리가 가장 가까운 특징점들을 매칭하는 방식입니다.

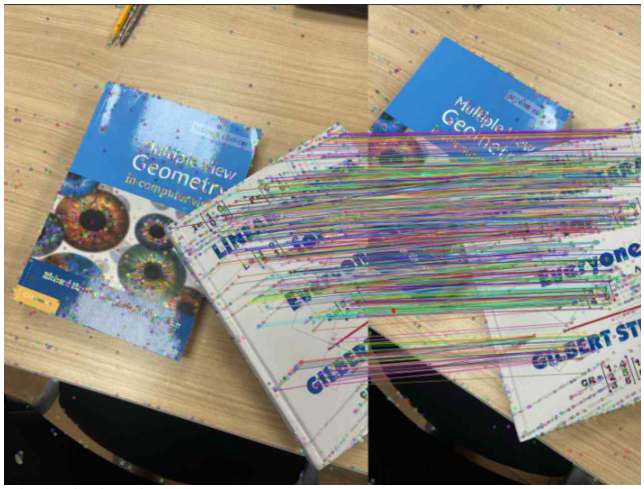
- 간단하고 직관적이지만, 큰 규모의 데이터셋에서는 계산 비용이 크고 느려질 수 있다는 단점이 존재합니다.

```python
def get_transform_from_keypoints(img_src, img_dst):
    img_src_gray = cv2.cvtColor(img_src, cv2.COLOR_BGR2GRAY)
    img_dst_gray = cv2.cvtColor(img_dst, cv2.COLOR_BGR2GRAY)

    # Create a SIFT object and detect keypoints and descriptors for each image
    sift = cv2.SIFT_create()
    kpts_src, dscrpt_src = sift.detectAndCompute(img_src_gray, None)
    kpts_dst, dscrpt_dst = sift.detectAndCompute(img_dst_gray, None)
    #print(dscrpt_src, dscrpt_dst)

    # Match the descriptors using Brute-Force matcher
    bf = cv2.BFMatcher()
    matches = bf.match(dscrpt_src, dscrpt_dst)

    # Sort the matches in the order of their distances
    matches = sorted(matches, key=lambda x: x.distance)

    # Find the homography matrix using the matched keypoints
    src_pts = np.float32([kpts_src[m.queryIdx].pt for m in matches]).reshape(-1, 1,
    dst_pts = np.float32([kpts_dst[m.trainIdx].pt for m in matches]).reshape(-1, 1,
    H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
    matches = [matches[i] for i in range(len(matches)) if mask[i]]

    # Return outputs
    return H, kpts_src, dscrpt_src, kpts_dst, dscrpt_dst, matches
```

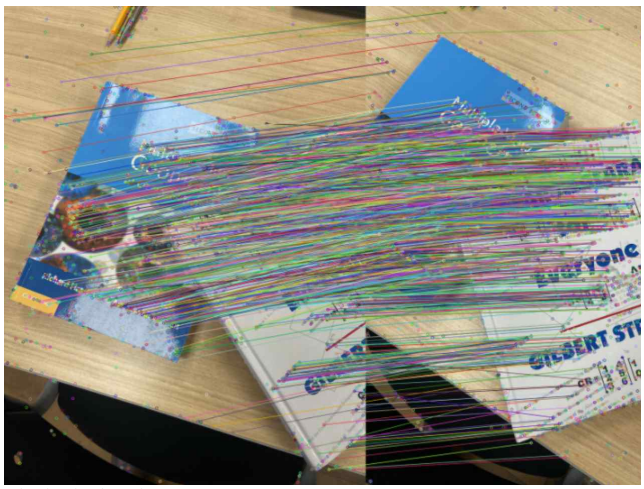
- 위의 코드는 cv2.BFMatcher()함수를 사용하여 Brute-Force matcher 객체를 생성하고, bf.match()를 호출하여 디스크립터를 매칭한 것입니다.
매칭된 결과는 거리를 기준으로 정렬되어 있는 것을 알 수 있습니다.

- FLANN matcher방식과 비교해보면 이미지 스티칭 결과가 크게 다르지 않았습니다. 지금껏 나온 결과를 확인해본 결과 두 이미지가 블렌딩 되는 방식에 문제가 있다는 것을 파악했습니다. 다음으로는 RANSAC 파리미터를 조절하며 결과값을 확인해보고 마지막으로는 블렌딩 방식을 바꾸어서 이미지 스티칭 결과를 확인해보았습니다.

2.3 RANSAC 파라미터 조정



파라미터 값 : 2.0



파라미터 값 : 100.0

- RANSAC 알고리즘은 '이상치(outliers)'에 강한 특징을 가지고 있습니다. 호모그래피 행렬을 추정하기 위해 매칭된 keypoint중에서 가장 일치하는 point집합을 찾는 방법입니다.

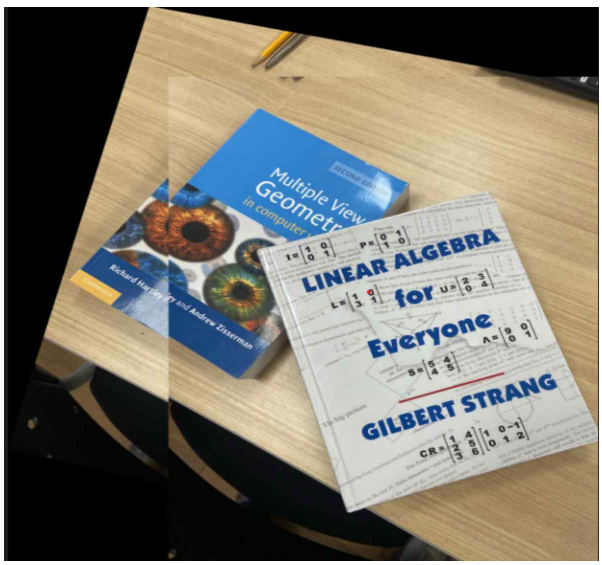
- RANSAC 알고리즘에 사용되는 임계값(T)는 거리의 임계값을 의미하며, 호모그래피 행렬을 추정하기 위해 사용되는 포인트 쌍의 임계값을 결정합니다. 이러한 임계값을 키우거나 줄이는 것은 동작에 영향을 줄 수 있습니다.

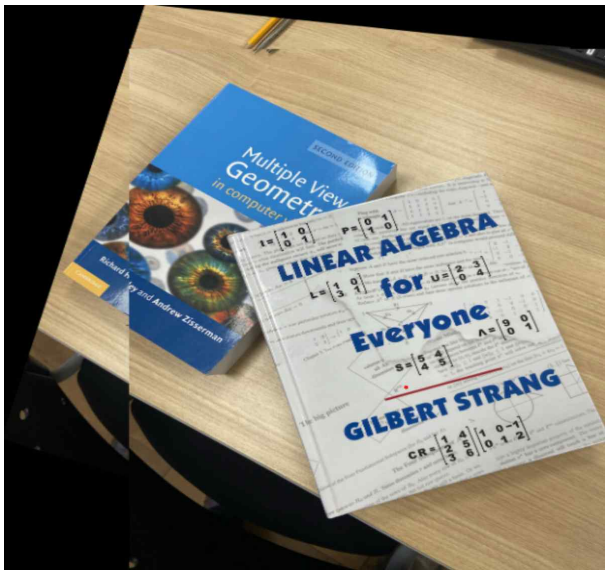
  -> 임계값을 키웠을 때
  : 더 많은 포인트 쌍이 호모그래피 행렬 추정에 사용됩니다. 이는 잘못된 매칭을 찾는 정확도를 향상시킬 수 있지만, 이상치의 영향을 받을 가능성이 높아집니다. 따라서 이상치가 적을 경우 정확도가 향상될 것입니다.

  -> 임계값을 줄였을 때
  : 적은 포인트 쌍이 호모그래피 행렬 추정에 사용됩니다. 이는 정확한 매칭을 찾는 정확도를 향상시킬 수 있지만, 마찬가지 이상치의 영향을 받을 가능성도 높아집니다. 따라서 이상치가 많을 경우 정확도가 향상될 것입니다,

- 위와 같은 특성을 고려하여 RANSAC 파라미터 값을 낮추거나 높여서 이미지 스티칭 결과를 살펴보았습니다.

- 오른쪽 위 2개의 사진을 보면, 파라미터 값에 따라 매칭된 특징점들의 분포가 많이 다른 것을 육안으로도 확인할 수 있습니다. 확실히 파라미터값이 아주 큰 100일때 많은 포인트 쌍이 행렬 추정에 사용된다는 것을 알 수 있었습니다.





- 이전 페이지의 위에 있는 왼쪽과 오른쪽 두 사진은 각각 RANSAC 파리미터의 값이 2.0, 100.0일때의 이미지 스티칭 결과입니다. 확실히 다른 결과를 육안으로 확인할 수 있었습니다. 하지만 아직도 블렌딩 방식으로 인하여 두 사진이 그냥 위로 겹쳐지기만 하는 느낌이 많이 들었습니다.

- 마지막으로는 블렌딩 방식의 변화로 조금 더 나은 이미치 스티칭 결과를 도출해보려고 시도해보았습니다.

2.4 Image Blending 방식

- 마지막으로는 블렌딩 방식의 변화로 조금 더 나은 이미치 스티칭 결과를 도출해보려고 시도해보았습니다.

- 이미지 블렌딩이란 두 개 이상의 이미지를 조합하여 새로운 이미지를 생성하는 것입니다. 여러가지 방법이 있지만 가장 일반적이고 기본적인 'Alpha Bleding' 방법을 선택하여 코드를 작성했습니다.

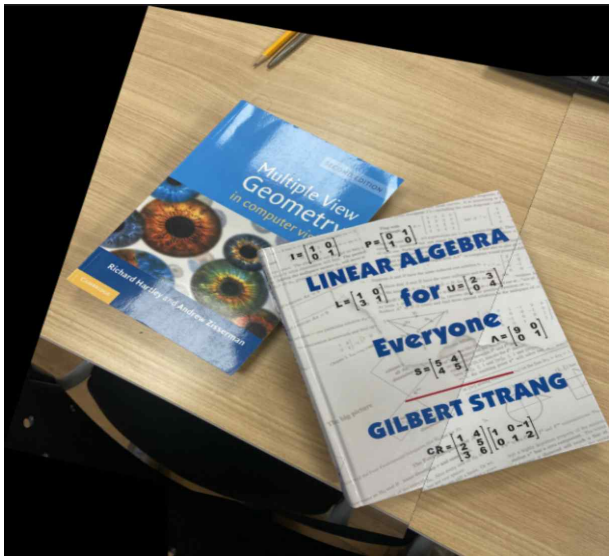
- 알파 블렌딩은 두 이미지의 가중 평균을 계산하여 새로운 이미지를 생성하는 방법으로, 각 픽셀의 가중치를 조절하여 원하는 블렌딩 효과를 얻을 수 있습니다.

- 코드 상에 변수 'alpha'는 알파 블렌딩의 가중치를 조절하는 값으로 0과 1사이의 값을 사용합니다. 0에 가까울수록 첫 번째 이미지가 강조되고, 1에 가까울수록 두 번째 이미지가 강조됩니다.

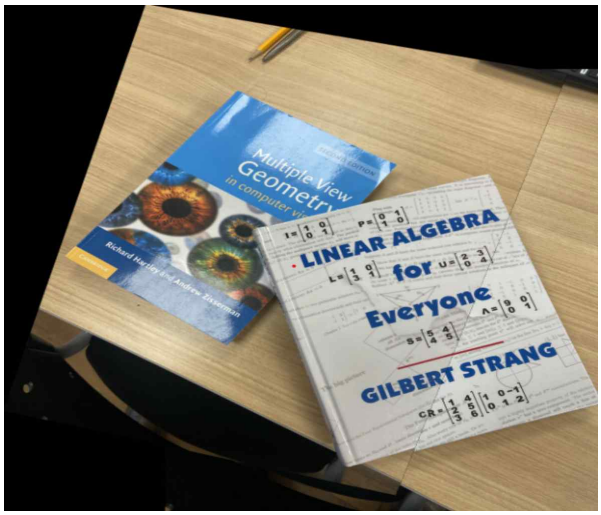
- 즉, alpha 값을 조절하여 스무스한 블렌딩 효과를 얻을 수 있습니다.

- 오른쪽의 3개의 사진은 각각 다른 alpha값에 따른 이미지 스티칭 결과입니다.

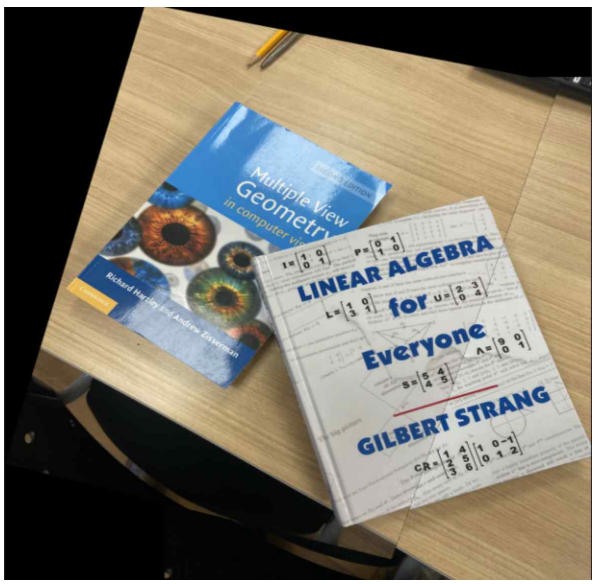
- 기존에 그저 이미지를 겹쳐서 하나의 이미지로 합성하는 방식이 아닌 알파 블렌딩을 이용하여 확실히 더 스무스한 블렌딩 통해 하나로 잘 합성된 이미지를 구할 수 있었습니다.



alpha = 0.5



alpha = 1.0



alpha = 0

3. 실험 결과 및 분석

2번 과정에서 제시된 4가지 방법론에서 각각의 결과값에 대한 이미지를 첨부하고 결과에 대한 분석을 진행하였기에 결론에서 마무리 짓도록 하겠습니다.

4. 결론

지금까지 서론에서 제시한 4가지 방법을 통한 특징점 추출부터 매칭, 이미지 스티칭 과정까지의 성능 비교를 해보았습니다.

특징점 추출 알고리즘 또한 다양하고, 그렇게 추출된 특징점을 매칭하는 알고리즘 등등 이미지 스티칭과정에 필요한 다양한 알고리즘이 존재함을 알 수 있었고, 각 알고리즘이 어떤 방식으로 구현되고 어떠한 장단점이 존재하는지 openCV에서 제공하는 함수들의 파라미터를 변화시키면서 알 수 있게 되었습니다.

위의 보고서는 단순히 제공해주신 예시를 통해서 분석한 것이지만, 입력되는 이미지의 형태와 종류에 따라서 수 많은 다양한 스티칭 결과를 얻을 수 있을 것임을 예상할 수 있었습니다.

참고문헌

[1] chatGPT

```
# 첫 번째 이미지 꼭지점 좌표 생성
corners_src = np.float32([[0, 0], [0, rows_src], [cols_src, rows_src], [cols_src, 0]]).resha
1, 2)

# H변환행렬을 통한 1st 이미지의 꼭지점좌표를 2nd 이미지의 좌표공간으로 변환
corners_dst = cv2.perspectiveTransform(corners_src, H)

# 2nd 이미지의 꼭지점 좌표와 1st 이미지의 꼭지점 좌표 합치기
corners = np.concatenate((corners_dst, corners_src), axis=0)

# 변환된 자표들 중 최대/최소 x, y좌표 구하기
[x_min, y_min] = np.int32(corners.min(axis=0).ravel() - 0.5)
[x_max, y_max] = np.int32(corners.max(axis=0).ravel() + 0.5)

# 음수값을 보정하기 위한 offset 설정
offset_x = -x_min if x_min < 0 else 0
offset_y = -y_min if y_min < 0 else 0

# 출력 이미지 너비와 높이 계산
dst_width = max(x_max, cols_dst) - min(x_min, 0)
dst_height = max(y_max, rows_dst) - min(y_min, 0)

# Modify H to account for the image offset - 변환행렬 오프셋 고려
### CODE YOURSELF ###
H_modified = np.array([[1, 0, offset_x], [0, 1, offset_y], [0, 0, 1]]) @ H

# Warp the first image to the perspective of the second image - img_src를 2nd 이미지
근 변환에 맞게 변환
### CODE YOURSELF ###
warped_img = cv2.warpPerspective(img_src, H_modified, (dst_width, dst_height))

# Combine the two images to create a single stitched image
### CODE YOURSELF ###
stitched_image = np.copy(warped_img)
stitched_image[offset_y:rows_dst + offset_y, offset_x:cols_dst + offset_x] = img_dst

# return output
return stitched_image
위 코드에서 이미지를 블렌딩할 때 조금 더 스무스하게 하는 코드를 작성해줘
```

- 기존에 그냥 겹쳐서 블렌딩하는 방식의 코드를 입력으로 넣어주고 스무스한 블렌딩하는 코드를 GPT로부터 얻었습니다.

- 또한, get_transform_from_keypoints()함수는 GPT의 코드를 거의 비슷하게 사용했습니다.