

Giải thích chi tiết mã nguồn (Cell 1 – Cell 9)

Đoạn mã Python trên thực hiện quá trình ghép ảnh panorama từ ba ảnh con: ảnh trái, ảnh giữa và ảnh phải. Ta sẽ đi qua từng cell và giải thích chi tiết từng bước, đồng thời định nghĩa các khái niệm và thuật toán quan trọng.

Cell 1: Nhập thư viện và hàm hiển thị ảnh

- **Nhập thư viện:** Đoạn đầu tiên nhập các thư viện cần thiết:
- `cv2` (OpenCV): thư viện xử lý ảnh phổ biến.
- `numpy`: thư viện cho tính toán ma trận và số học.
- `matplotlib.pyplot`: thư viện vẽ đồ thị, ở đây dùng để hiển thị ảnh.
- **Cấu hình hiển thị:** `plt.rcParams['figure.figsize'] = (8, 5)` đặt kích thước mặc định cho hình ảnh hiển thị (8x5 inches).
- **Hàm** `show_img`:

```
def show_img(img_bgr, title=""):
    """Hiển thị ảnh BGR bằng matplotlib."""
    img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
    plt.figure()
    plt.imshow(img_rgb)
    plt.title(title)
    plt.axis("off")
    plt.show()
```

- Hàm này nhận một ảnh ở định dạng BGR (đúng với cách OpenCV đọc ảnh) và chuyển sang định dạng RGB để hiển thị bằng Matplotlib (vì Matplotlib mặc định hiển thị ảnh theo kênh màu RGB).
- `cv2.cvtColor` chuyển đổi không gian màu từ BGR sang RGB.
- Sau đó vẽ ảnh với `plt.imshow`, tắt trục tọa độ với `plt.axis("off")` và hiện tiêu đề nếu có.

Chú ý: OpenCV dùng định dạng **BGR** trong khi matplotlib dùng **RGB**, nên cần chuyển đổi để ảnh không bị sai màu.

Cell 2: Đọc ảnh và hiển thị

- **Đường dẫn thư mục:**

```
image_dir = r"C:\Users\Public\panorama_sift_three_pictures"
```

Chứa đường dẫn đến thư mục chứa ba ảnh cần ghép.

- **Đọc ảnh:**

```
left_img = cv2.imread(fr"{image_dir}\img_left_2.jpg")
middle_img = cv2.imread(fr"{image_dir}\img_middle_2.jpg")
right_img = cv2.imread(fr"{image_dir}\img_right_2.jpg")
```

- `cv2.imread(path)` đọc ảnh từ đường dẫn, kết quả là mảng NumPy (3 chiều) chứa pixel theo định dạng BGR.
- Ở đây dùng ký tự f-string (`fr"..."`) để nối chuỗi đường dẫn.
- **In kích thước ảnh:**

```
print("Kích thước ảnh trái :", left_img.shape)
```

`shape` trả về (chiều cao, chiều rộng, số kênh). Việc in ra giúp kiểm tra ảnh đã đọc đúng.

- **Hiển thị ảnh:** Gọi hàm `show_img` cho từng ảnh, đồng thời đặt tiêu đề:

```
show_img(left_img, "Ảnh trái")
show_img(middle_img, "Ảnh giữa")
show_img(right_img, "Ảnh phải")
```

Điều này cho phép kiểm tra trực quan ba ảnh nguồn.

Cell 3: Tính toán đặc trưng SIFT và khớp ảnh giữa hai ảnh

Hàm `sift_feature`

```
def sift_feature(image_bgr):
    """Tính keypoint & descriptor SIFT cho 1 ảnh BGR."""
    gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)
    sift = cv2.SIFT_create()
    kp, des = sift.detectAndCompute(gray, None)
    return kp, des
```

- **Chuyển ảnh sang xám:** SIFT (Scale-Invariant Feature Transform) hoạt động trên ảnh đơn sắc (grayscale), nên đầu tiên chuyển ảnh BGR sang ảnh xám. - **Tạo đối tượng SIFT:** `cv2.SIFT_create()` khởi tạo thuật toán SIFT. SIFT là phương pháp phát hiện và mô tả đặc trưng cục bộ (keypoints) ổn định theo tỉ lệ và xoay của ảnh ¹. Các đặc trưng này gồm: - **Keypoints:** các điểm quan trọng (ví dụ góc hoặc điểm có gradient mạnh) trong ảnh. - **Descriptors:** vector mô tả hình thái xung quanh mỗi keypoint, thường có chiều 128 (SIFT gốc) và chứa thông tin gradient hướng, giúp nhận dạng đặc trưng khi so sánh giữa các ảnh. - **Phát hiện và mô tả đặc trưng:** `sift.detectAndCompute(gray, None)` đồng thời phát hiện keypoints (`kp`) và tính descriptor (`des`) cho mỗi keypoint. Trả về: - `kp`: danh sách các đối tượng `KeyPoint`, mỗi đối tượng có tọa độ (x, y), kích thước, hướng, v.v. - `des`: mảng NumPy kích thước (số_keypoints, 128) chứa vector mô tả cho mỗi keypoint. - **Trả kết quả:** (`kp, des`) để sử dụng cho việc khớp ảnh sau.

Hàm `match_sift`

```
def match_sift(kp1, des1, kp2, des2, ratio_thresh=0.75, debug_title=None,
img1=None, img2=None):
    """
    KNN matching + Lowe ratio test.
    Trả về: good_matches, H (homography 2->1 nếu tính được).
    """
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=False)
    knn_matches = bf.knnMatch(des1, des2, k=2)
    ...
```

Đây là hàm **khớp đặc trưng** giữa hai ảnh sử dụng SIFT descriptors. Giải thích từng phần: - **Khởi tạo bộ so khớp (BFMatcher):**

`cv2.BFMatcher(cv2.NORM_L2, crossCheck=False)` - **BFMatcher:** Brute-Force Matcher thực hiện so khớp brute-force, so sánh mỗi descriptor ở ảnh này với mỗi descriptor ở ảnh kia. - `cv2.NORM_L2`: sử dụng khoảng cách Euclid (L2 norm) vì descriptor SIFT có đặc tính ở không gian liên tục (float vector). - `crossCheck=False`: không yêu cầu đối xứng (thông thường để áp dụng Lowe ratio test cần `crossCheck=True`). - **Kết quả KNN:**

```
knn_matches = bf.knnMatch(des1, des2, k=2)
```

- `knnMatch` tìm hai ($k=2$) kết quả khớp tốt nhất cho mỗi descriptor trong `des1` với descriptors trong `des2`. Kết quả `knn_matches` là danh sách các cặp `(m, n)`, với `m` là match tốt nhất và `n` là match thứ hai tốt nhất. - **Lowe's Ratio Test:** Loại bỏ những cặp match không chắc chắn bằng kiểm tra khoảng cách:

```
good = []
for m, n in knn_matches:
    if m.distance < ratio_thresh * n.distance:
        good.append(m)
```

- Với mỗi cặp `(m, n)`, so sánh khoảng cách `m.distance` (match tốt nhất) với `n.distance` (nhì tốt nhất).

- Nếu `m.distance` nhỏ hơn `ratio_thresh` lần `n.distance` (ở đây `ratio_thresh=0.75`), nghĩa là tốt nhất rõ ràng hơn nhì, thì coi `m` là *good match*.

- *Lowe's Ratio Test* giúp lọc bớt các match kém tin cậy và giữ những match thật, giảm sai lệch do cấu trúc lặp lại hoặc nhiễu. - Lưu ý: Tên hàm `ratio_thresh` 0.75 là giá trị tiêu chuẩn thường dùng trong lý thuyết gốc của SIFT. - **In số lượng khớp tốt:** `print(f"[{debug_title}] Số match tốt: {len(good)}")`. Cung cấp thông tin số điểm khớp được chọn. - **Tính Homography (nếu đủ match):**

```
if len(good) >= 4:
    pts1 = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1, 1, 2)
    pts2 = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1, 1, 2)
    H, mask = cv2.findHomography(pts2, pts1, cv2.RANSAC)
    print(f"[{debug_title}] Homography:\n", H)
else:
    print(f"[{debug_title}] Không đủ match để tính homography!")
```

- **Điều kiện:** Cần ít nhất 4 match tốt (điểm) để tính homography (phép biến đổi dựa trên ma trận 3×3).

Nếu không đủ, báo lỗi và bỏ qua. - **Xác định tọa độ:**

- `pts1`: tọa độ keypoints từ ảnh thứ nhất (tương ứng với `img1` đối với `kp1`) cho mỗi match tốt.

`m.queryIdx` lấy chỉ số keypoint trong `kp1`. - `pts2`: tọa độ keypoints từ ảnh thứ hai (`kp2`) cho mỗi match tốt. `m.trainIdx` lấy chỉ số keypoint trong `kp2`. - Hai tập tọa độ này cần cùng thứ tự (keypoint i ở ảnh 1 tương ứng keypoint i ở ảnh 2).

- **Homography:** Dùng `cv2.findHomography(pts2, pts1, cv2.RANSAC)` để tính ma trận Homography H , biến đổi từ ảnh 2 sang ảnh 1.

- *Homography* là một ma trận 3×3 thể hiện phép biến đổi chiếu ảnh (projective transformation) giữa hai ảnh (thường áp dụng cho ảnh chụp trên cùng một mặt phẳng hoặc khi xoay camera).

- `cv2.RANSAC` chỉ định dùng thuật toán RANSAC (Random Sample Consensus) để tăng tính kháng ngoại lệ: nó lặp chọn ngẫu nhiên một tập con các điểm và tính H , rồi chọn H sao cho có nhiều điểm phù hợp nhất, tức bỏ trội những điểm sai khớp.

- Kết quả: `H` (3×3) và `mask` (mảng đánh dấu điểm inliers) nhưng ở đây chỉ cần H . - In ra ma trận Homography nếu tính được. - **Vẽ một số kết quả khớp (tuỳ chọn debug):**

```
if debug_title is not None and img1 is not None and img2 is not None and len(good) > 0:
    ...
    match_img = cv2.drawMatches(...)
    show_img(match_img, f"Match SIFT - {debug_title} (top {draw_n})")
```

Nếu cung cấp `debug_title`, `img1`, `img2` và có ít nhất một khớp tốt, chương trình sẽ vẽ ra hình thể hiện một số cặp điểm khớp giữa hai ảnh.

- `drawMatches` dùng để vẽ các đường nối giữa keypoints trùng nhau trên hai ảnh. - Điều này chỉ phục vụ mục đích trực quan (debug) để thấy chất lượng khớp.

- **Kết quả trả về:** Hàm trả về danh sách `good` (các match tốt) và ma trận `H` (hoặc `None` nếu không thể tính).

Cell 4: Tính SIFT và khớp cho ba ảnh

- **Tính SIFT cho từng ảnh:**

```
kp_left, des_left = sift_feature(left_img)
kp_mid, des_mid = sift_feature(middle_img)
kp_right, des_right = sift_feature(right_img)
```

Ở đây, ta gọi hàm `sift_feature` vừa tạo để tính keypoints (`kp_...`) và descriptors (`des_...`) cho ảnh trái, ảnh giữa, ảnh phải.

- **In số lượng keypoints:**

```
print("SIFT keypoints:")
print("  Trái :", len(kp_left))
print("  Giữa:", len(kp_mid))
print("  Phải:", len(kp_right))
```

Để kiểm tra có bao nhiêu điểm đặc trưng được phát hiện ở mỗi ảnh. Số lượng này phản ánh mức độ chi tiết của ảnh: ảnh phức tạp hơn thường có nhiều keypoint hơn.

• Khớp Ảnh Trái và Ảnh Giữa:

```
good_lm, H_left_to_mid = match_sift(
    kp_mid, des_mid, kp_left, des_left,
    ratio_thresh=0.75,
    debug_title="LEFT <-> MIDDLE",
    img1=middle_img, img2=left_img
)
```

- Gọi `match_sift` giữa **ảnh giữa (img1)** và **ảnh trái (img2)**. Kết quả là danh sách `good_lm` các match tốt và `H_left_to_mid` là ma trận biến đổi từ ảnh trái sang ảnh giữa.
- `debug_title` là chuỗi tiêu đề để hiển thị số liệu debug khi vẽ ảnh.
- Chú ý: Lệnh `match_sift(kp_mid, des_mid, kp_left, des_left, img1=middle_img, img2=left_img)` được sắp xếp để ma trận H tính ra biến đổi từ ảnh trái sang ảnh giữa (tức tọa độ của ảnh trái đưa về ảnh giữa). Điều này tương ứng đoạn comment "# homography: left -> middle".

• Khớp Ảnh Phải và Ảnh Giữa:

```
good_rm, H_right_to_mid = match_sift(
    kp_mid, des_mid, kp_right, des_right,
    ratio_thresh=0.75,
    debug_title="RIGHT <-> MIDDLE",
    img1=middle_img, img2=right_img
)
```

- Tương tự, ta khớp **ảnh giữa (img1)** với **ảnh phải (img2)**. Thu được `H_right_to_mid`, biến đổi từ ảnh phải sang ảnh giữa.

Ở bước này, ta đã tính được **Homography** cho hai cặp ảnh (trái ↔ giữa và phải ↔ giữa), để chuẩn bị ghép ảnh sau này. Homography thể hiện cách ánh xạ hình chiếu giữa các ảnh, cho phép ta biến đổi ảnh trái hoặc ảnh phải vào hệ tọa độ của ảnh giữa.

Cell 5: Định nghĩa các hàm dựng canvas, phối ảnh và cắt xén

Cell này định nghĩa ba hàm chính:

1. `compute_global_canvas`

```
def compute_global_canvas(left_img, middle_img, right_img,
                          H_left_to_mid, H_right_to_mid):
    """Tính canvas chung & offset_matrix cho cả 3 ảnh."""
    ...
    corners_left = np.float32([[0,0], [wL,0], [0,hL], [wL,hL]]).reshape(-1,
1,2)
    corners_mid = np.float32([[0,0], [wM,0], [0,hM], [wM,hM]]).reshape(-1,
1,2)
    corners_right = np.float32([[0,0], [wR,0], [0,hR], [wR,hR]]).reshape(-1,
1,2)
```

```

    corners_left_warp = cv2.perspectiveTransform(corners_left,
    H_left_to_mid)
    corners_right_warp = cv2.perspectiveTransform(corners_right,
    H_right_to_mid)
    corners_mid_warp = corners_mid # ảnh giữa làm gốc

    all_corners = np.concatenate(
        (corners_left_warp, corners_mid_warp, corners_right_warp),
        axis=0
    )
    [x_min, y_min] = all_corners.min(axis=0).ravel()
    [x_max, y_max] = all_corners.max(axis=0).ravel()
    ...
    offset_matrix = np.array([
        [1, 0, -x_min],
        [0, 1, -y_min],
        [0, 0, 1]
    ], dtype=float)
    print("Canvas size:", canvas_w, "x", canvas_h)
    return canvas_w, canvas_h, offset_matrix

```

Mục tiêu hàm này là xác định **kích thước của ảnh canvas chung** (giáo diện chứa panorama) và ma trận offset cần thêm vào khi warp ảnh. Chi tiết: - **Tọa độ các góc**: Đối với mỗi ảnh (trái, giữa, phải), xác định tọa độ bốn góc:

- `[0, 0]` (góc trên-trái), `[w, 0]` (góc trên-phải), `[0, h]` (góc dưới-trái), `[w, h]` (góc dưới-phải).

Đây là vị trí góc tính theo kích thước gốc của ảnh.

- **Biến đổi góc**:

- Dùng `cv2.perspectiveTransform(corners_left, H_left_to_mid)` để biến đổi tọa độ góc của ảnh trái sang hệ tọa độ của ảnh giữa (qua ma trận `H_left_to_mid`). Kết quả là

`corners_left_warp`. - Tương tự, biến đổi góc của ảnh phải bằng `H_right_to_mid`. -

`corners_mid_warp = corners_mid` vì ảnh giữa xem như tọa độ gốc không bị biến đổi. - **Tập hợp tất cả góc**: Kết hợp các góc đã biến đổi thành một mảng `all_corners`. - **Tính miền cực tiểu và cực đại**:

- `x_min, y_min` là tọa độ nhỏ nhất (đáy khung) trong tất cả điểm góc. - `x_max, y_max` là tọa độ lớn nhất. - Đây sẽ là vùng cần thiết để chứa tất cả các ảnh sau khi warp vào chung một coordinate frame. - **Xử lý dịch chuyển**:

- Dịch chuyển `x_min, y_min` xuống mức (0,0) của canvas (phần ảnh bên trái/ trên tiêu chuẩn được dịch về đầu canvas). Điều này tránh tọa độ âm khi warp.

- Dịch chuyển `x_min, y_min` xuống mức (0,0) của canvas (phần ảnh bên trái/ trên tiêu chuẩn được dịch về đầu canvas). Điều này tránh tọa độ âm khi warp.

- `offset_matrix` là ma trận dịch chuyển (3x3) để thêm vào ma trận homography khi warp ảnh:

$$\text{offset_matrix} = \begin{bmatrix} 1 & 0 & -x_{\min} \\ 0 & 1 & -y_{\min} \\ 0 & 0 & 1 \end{bmatrix}.$$

- Nhân `offset_matrix` với `H` sẽ đảm bảo ảnh được dịch về giữa canvas dương. - **Xóa biên đen dưới cùng (nâng ảnh lên)**:

- `delta_y = -100` được cộng vào chiều cao canvas, vì dường như tác giả muốn loại bỏ phần viền đen ở dưới và "nâng" panorama lên (thông thường panorama có phần dưới đen do chênh lệch hình chiếu). -

Lưu ý: Việc này là tùy vào dữ liệu cụ thể, không phải thuật toán chung. - **Trả về**: `(canvas_w, canvas_h, offset_matrix)`. Kích thước canvas chung và ma trận offset.

2. feather_blend_two

```
def feather_blend_two(imgA, imgB):
    """
    Feather blending 2 ảnh cùng kích thước, dùng 0 làm nền không dữ liệu.
    Trả về ảnh đã blend.
    """
    assert imgA.shape == imgB.shape
    h, w = imgA.shape[:2]
    maskA = np.any(imgA != 0, axis=2)
    maskB = np.any(imgB != 0, axis=2)
    onlyA = maskA & ~maskB
    onlyB = maskB & ~maskA
    overlap = maskA & maskB
    result = np.zeros_like(imgA)
    result[onlyA] = imgA[onlyA]
    result[onlyB] = imgB[onlyB]
    if np.any(overlap):
        X = np.tile(np.arange(w), (h, 1))
        x0 = X[overlap].min()
        x1 = X[overlap].max()
        if x1 == x0:
            wA = np.full((h, w), 0.5, dtype=np.float32)
        else:
            wA = (x1 - X).astype(np.float32) / (x1 - x0)
        wB = 1.0 - wA
        for c in range(3):
            chanA = imgA[:, :, c].astype(np.float32)
            chanB = imgB[:, :, c].astype(np.float32)
            blend = chanA * wA + chanB * wB
            tmp = result[:, :, c].astype(np.float32)
            tmp[overlap] = blend[overlap]
            result[:, :, c] = np.clip(tmp, 0, 255).astype(np.uint8)
    return result
```

Hàm này **pha trộn (blending)** hai ảnh `imgA` và `imgB` có cùng kích thước bằng phương pháp *feather blending* (pha trộn lông vũ). Chi tiết: - **Kiểm tra kích thước:** `assert imgA.shape == imgB.shape`

đảm bảo hai ảnh có cùng chiều cao, chiều rộng, và kênh màu. - **Tạo mặt nạ vùng có dữ liệu:**

- `maskA = np.any(imgA != 0, axis=2)`: giá trị `True` ở những pixel của ảnh A mà không hoàn toàn là `[0,0,0]` (tức vùng có dữ liệu, trong khi nền không dữ liệu quy ước là đen `[0,0,0]`). - Tương tự `maskB` cho ảnh B. - **Xác định vùng riêng và chung:**

- `onlyA = maskA & ~maskB`: vùng chỉ có dữ liệu ở ảnh A (B không có). - `onlyB = maskB & ~maskA`: vùng chỉ có dữ liệu ở ảnh B. - `overlap = maskA & maskB`: vùng có dữ liệu ở cả hai ảnh (giao nhau). - **Tạo ảnh kết quả ban đầu:** `result = np.zeros_like(imgA)` (ban đầu đen). - **Ghi giá trị không chồng lên nhau:**

- Ở những pixel chỉ có ở ảnh A (`onlyA`), sao chép từ `imgA`. - Ở những pixel chỉ có ở ảnh B (`onlyB`), sao chép từ `imgB`. - **Xử lý vùng overlap:** Nếu tồn tại vùng giao nhau: - Tạo ma trận `X` kích thước `(h×w)`

chứa giá trị tọa độ `x` của mỗi pixel. Dùng `np.tile(np.arange(w), (h,1))` tạo lưới `x` theo hàng. -

Tìm `x0`, `x1` là cột trái nhất và phải nhất của vùng overlap (tối thiểu và tối đa của `X` trong vùng

overlap). - Tính **trọng số pha trộn** w_A cho ảnh A:

- Nếu $x_1 == x_0$ (vùng overlap chỉ một cột), đặt $w_A = 0.5$ (bằng nhau).

- Ngược lại, giả sử ảnh A nằm bên trái, ảnh B bên phải: tính

$$w_A = \frac{x_1 - X}{x_1 - x_0}, \quad w_B = 1 - w_A.$$

Như vậy, ở bên trái overlap (x gần x_0), w_A lớn (gần 1) và w_B nhỏ, ưu tiên màu của ảnh A; tiến dần sang phải, w_A giảm, w_B tăng, ưu tiên ảnh B.

Feather blending kiểu này tạo ra chuyển tiếp mượt ở vùng giao nhau. - Trộn kênh màu:

- Với mỗi kênh màu (0,1,2), lấy mảng hai kênh tương ứng (chanA, chanB). Dùng công thức

$\text{chanA} * w_A + \text{chanB} * w_B$ để pha trộn giá trị màu. Kết quả là mảng `blend`. - Chỉ áp dụng ở những pixel overlap: `tmp[overlap] = blend[overlap]` rồi gán vào `result`. - Kết quả trả về là ảnh trộn hoàn chỉnh.

Đặc điểm nổi bật: Feather blending là phép pha trộn tuyến tính dựa trên khoảng cách, cho hình ảnh mượt không thấy đường nối đột ngột. Đây là cách đơn giản và hiệu quả trong trường hợp ảnh có chồng lấn khu vực liền kề.

3. `auto_crop_non_black`

```
def auto_crop_non_black(pano_bgr):  
    """Tự động crop bỏ viền đen quanh ảnh panorama."""  
    gray = cv2.cvtColor(pano_bgr, cv2.COLOR_BGR2GRAY)  
    _, thresh = cv2.threshold(gray, 1, 255, cv2.THRESH_BINARY)  
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL,  
                                   cv2.CHAIN_APPROX_SIMPLE)  
  
    if not contours:  
        return pano_bgr  
    largest = max(contours, key=cv2.contourArea)  
    x, y, w, h = cv2.boundingRect(largest)  
    cropped = pano_bgr[y:y+h, x:x+w]  
    return cropped
```

Hàm này tự động **cắt bỏ phần nền đen** xung quanh ảnh panorama. - **Chuyển sang ảnh xám:**

`gray = cv2.cvtColor(pano_bgr, cv2.COLOR_BGR2GRAY)`. - **Nhị phân hóa:**

`cv2.threshold(gray, 1, 255, cv2.THRESH_BINARY)` tạo ảnh nhị phân `thresh` sao cho pixel >1 thành 255 (trắng), pixel ≤ 1 (đen) là 0. Điều này đánh dấu vùng có dữ liệu (non-black) thành màu trắng. - **Tìm đường bao:** `cv2.findContours(thresh, cv2.RETR_EXTERNAL, ...)` tìm tất cả đường viền của các vùng trắng (RETR_EXTERNAL chỉ lấy đường viền ngoài). - **Tìm khung bao lớn nhất:**

Do ảnh panorama nhiều khả năng chỉ có một vùng chính có dữ liệu, ta lấy

`largest = max(contours, key=cv2.contourArea)` để có contour diện tích lớn nhất. - **Tạo**

bounding box: `cv2.boundingRect(largest)` trả về `(x, y, w, h)`, tọa độ góc trên-trái và chiều rộng, cao của hình chữ nhật bao quanh contour lớn nhất. - **Cắt ảnh:** Lấy vùng `(y:y+h, x:x+w)` từ ảnh gốc, loại bỏ mọi thứ ngoài khung. Trả về `cropped`. - Nếu không tìm thấy contour nào (không có dữ liệu), trả lại ảnh gốc.

Đặc điểm: Hàm này giúp loại bỏ các viền đen thừa do warp và blend ảnh, kết quả là ảnh panorama chỉ chứa phần có dữ liệu.

Cell 6: Warp ba ảnh vào canvas chung

```
if H_left_to_mid is None or H_right_to_mid is None:
    raise RuntimeError("Không tính được homography ...")
canvas_w, canvas_h, offset_matrix = compute_global_canvas(...)
left_warp = cv2.warpPerspective(left_img, offset_matrix.dot(H_left_to_mid),
                                (canvas_w, canvas_h))
mid_warp = cv2.warpPerspective(middle_img, offset_matrix, (canvas_w,
                                                           canvas_h))
right_warp = cv2.warpPerspective(right_img,
                                  offset_matrix.dot(H_right_to_mid), (canvas_w, canvas_h))
show_img(left_warp, "Left warp")
...
```

- **Kiểm tra Homography:** Nếu không tính được `H_left_to_mid` hoặc `H_right_to_mid`, dừng chương trình báo lỗi (không thể tiếp tục ghép ảnh). - **Tính canvas chung:**

```
canvas_w, canvas_h, offset_matrix = compute_global_canvas(left_img,
                                                           middle_img, right_img, H_left_to_mid, H_right_to_mid)
```

Như giải thích ở trên, hàm này xác định kích thước canvas (chiều rộng, chiều cao) đủ để chứa cả ba ảnh sau khi biến đổi, và trả về `offset_matrix` để đảm bảo ảnh được đặt đúng vị trí. - **Warp (biến đổi phối cảnh) ảnh:** Dùng `cv2.warpPerspective` cho từng ảnh: - Ảnh **trái**:

`cv2.warpPerspective(left_img, offset_matrix.dot(H_left_to_mid), (canvas_w, canvas_h))`. - `offset_matrix.dot(H_left_to_mid)` kết hợp ma trận dịch chuyển với homography, tạo thành ma trận cuối để warp từ ảnh trái sang tọa độ canvas. - Kết quả `left_warp` là ảnh trái đã được chiếu vào canvas chung. - Ảnh **giữa**: `cv2.warpPerspective(middle_img, offset_matrix, (canvas_w, canvas_h))`. - Ở đây chỉ dùng `offset_matrix` (không homography) vì ảnh giữa là cơ sở, chỉ cần dịch chuyển vào canvas. - Ảnh **phải**:

`cv2.warpPerspective(right_img, offset_matrix.dot(H_right_to_mid), (canvas_w, canvas_h))`. - **Hiển thị kết quả warp:** Gọi `show_img` cho từng ảnh đã warp để kiểm tra. Mỗi ảnh giờ nằm trong một khung lớn `canvas_w x canvas_h`, với phần còn lại (không có ảnh) là đen.

Chú ý: `warpPerspective` áp dụng phép biến đổi dựa trên ma trận 3x3 cho toàn bộ ảnh; điểm ảnh mới của ảnh gốc được tính theo công thức chiếu. Kết quả là ảnh trong hệ tọa độ mới.

Cell 7: Ghép panorama tạm thời (trái + giữa)

```
pano_left_mid = feather_blend_two(mid_warp, left_warp)
show_img(pano_left_mid, "Panorama tạm thời (trái + giữa)")
```

- **Feather blending 2 ảnh:** Kết quả của cell trước là hai ảnh `mid_warp` (giữa) và `left_warp` (trái) đã ở chung canvas. Để tạo panorama tạm thời gồm ảnh trái + ảnh giữa, gọi:

```
pano_left_mid = feather_blend_two(mid_warp, left_warp)
```

Hàm `feather_blend_two` sẽ pha trộn hai ảnh này: - Vùng chỉ có ở ảnh trái được giữ nguyên từ `left_warp`. - Vùng chỉ có ở ảnh giữa giữ nguyên từ `mid_warp`. - Vùng overlap (nếu có) được trộn mượt theo trọng số tăng dần qua giao diện. - **Hiển thị:** `show_img` hiển thị panorama tạm thời kết quả. Nếu mọi thứ tốt, ảnh trái và giữa được ghép liền mạch mà không thấy rõ đường nối.

Lưu ý: Ở bước này, chúng ta có cả hai bản warping riêng và bản pha trộn (nối 2 ảnh). Panorama tạm thời này hữu ích để xem có bị lệch hay không giữa hai ảnh.

Cell 8: Ghép panorama tạm thời (giữa + phải)

```
pano_mid_right = feather_blend_two(mid_warp, right_warp)
show_img(pano_mid_right, "Panorama tạm thời (giữa + phải)")
```

Tương tự như cell 7, nhưng giờ ghép ảnh giữa và ảnh phải: - `pano_mid_right = feather_blend_two(mid_warp, right_warp)` pha trộn `mid_warp` và `right_warp`. - Hiển thị kết quả panorama tạm thời gồm ảnh giữa + ảnh phải.

Kết quả này cho ta kiểm tra việc khớp giữa ảnh giữa và ảnh phải. Sau khi có cả hai panorama tạm (trái+giữa và giữa+phải), ta chuẩn bị bước kết hợp cả ba ảnh.

Cell 9: Tạo panorama cuối cùng gồm 3 ảnh, cắt xén và lưu kết quả

Định nghĩa hàm crop dải dọc

```
def crop_vertical_band(img, keep_ratio=0.85):
    """
    Giữ lại một dải theo chiều dọc (giữa ảnh),
    cắt bớt phía trên & phía dưới cho gọn khung.
    keep_ratio: tỉ lệ chiều cao muốn giữ lại (0<keep_ratio<=1).
    """
    h, w = img.shape[:2]
    band_h = int(h * keep_ratio)
    top = (h - band_h) // 2          # cắt đối xứng trên - dưới
    bottom = top + band_h
    return img[top:bottom, :]
```

- Hàm này **cắt xén thêm theo chiều dọc** để làm gọn phần trên và dưới của ảnh panorama. - Giải thích: - `keep_ratio=0.85` nghĩa là giữ lại 85% chiều cao giữa của ảnh, cắt 15% (7.5% ở trên và 7.5% ở dưới, xấp xỉ). - Tính `band_h = int(h * keep_ratio)` là chiều cao dải giữ lại. - `top = (h - band_h) // 2`, `bottom = top + band_h` chia đều vùng cắt trên và dưới. - Trả về `img[top:bottom, :]`, ảnh bị cắt phần trên/bottom.

Đặc điểm: Đây là thao tác tùy chỉnh để làm ảnh đẹp hơn (có thể cắt bỏ bầu trời/trần nhà hoặc viền đen còn sót).

Ghép panorama ba ảnh và xử lý cuối

```
pano_full = feather_blend_two(pano_left_mid, right_warp)
show_img(pano_full, "Panorama 3 ảnh (chưa crop)")
```

- **Ghép (trái+giữa) với phải:** Đầu tiên `pano_left_mid` đã là ảnh trái+giữa. Giờ ta pha trộn tiếp `pano_left_mid` và `right_warp` (ảnh phải đã warp) để được ảnh panorama đầy đủ gồm **3 ảnh**.
- Kết quả là `pano_full`, hiển thị và lưu. Đây là panorama chưa qua cắt xén (chứa nhiều phần nền đen dư thừa).

```
pano_cropped = auto_crop_non_black(pano_full)
show_img(pano_cropped, "Panorama 3 ảnh (đã auto-crop viền đen)")
```

- **Auto crop:** Gọi hàm `auto_crop_non_black` để tự động cắt bỏ viền đen quanh `pano_full`. Lưu kết quả `pano_cropped` và hiển thị. Ảnh này chỉ còn phần có dữ liệu thực.

```
pano_final = crop_vertical_band(pano_cropped, keep_ratio=0.85)
show_img(pano_final, "Panorama 3 ảnh (đã auto-crop + crop trên/dưới)")
```

- **Crop dải dọc:** Gọi hàm `crop_vertical_band` để giữ lại 85% chiều cao giữa `pano_cropped`. Kết quả `pano_final` có phần trên và dưới được cắt bớt. - **Lưu kết quả:** Cuối cùng lưu ảnh:

```
cv2.imwrite(save_raw, pano_full)
cv2.imwrite(save_crop, pano_cropped)
cv2.imwrite(save_final, pano_final)
```

- `save_raw`: panorama chưa crop (còn nền đen xung quanh). - `save_crop`: panorama đã cắt đen tự động. - `save_final`: panorama cuối cùng (sau cả hai bước crop). - In đường dẫn file đã lưu.

Kết luận

Toàn bộ quy trình trên thực hiện **ghép ảnh panorama** bằng cách sau: 1. Đọc ba ảnh (trái, giữa, phải). 2. Tính đặc trưng SIFT và descriptor cho từng ảnh. 3. Khớp các đặc trưng giữa ảnh trái với ảnh giữa, và giữa với phải, sử dụng brute-force KNN và kiểm tra Lowe để giữ match tốt nhất. 4. Tính **Homography** cho từng cặp, dùng RANSAC để loại bỏ nhiễu. Homography này cho phép chuyển ảnh trái và phải về hệ toạ độ của ảnh giữa. 5. Tính **canvas chung** đủ lớn để chứa cả ba ảnh sau khi biến đổi, đồng thời tính ma trận dịch chuyển offset. 6. Dùng `warpPerspective` để chiếu mỗi ảnh vào canvas chung. 7. Dùng **feather blending** (pha trộn lông vũ) để kết hợp ảnh chồng lấp mượt mà giữa ảnh trái+giữa, và giữa+phải. 8. Cuối cùng, kết hợp tất cả vào một panorama, rồi tự động cắt bỏ biên đen và cắt dọc để ảnh gọn gàng hơn. 9. Lưu kết quả ở các trạng thái: chưa crop, đã crop biên, và crop trên/dưới.

Mỗi bước trên đảm bảo cả mặt toán học (Homography, masking, blending) và thực thi (dùng OpenCV) được minh giải rõ ràng. Các khái niệm quan trọng như **SIFT**, **Homography**, **RANSAC**, **feather blending** đều đã được định nghĩa và áp dụng nhằm đảm bảo khả năng ghép ảnh chính xác và mượt mà.

1 Introduction to SIFT(Scale Invariant Feature Transform) - Medium

<https://medium.com/@deepanshut041/introduction-to-sift-scale-invariant-feature-transform-65d7f3a72d40>