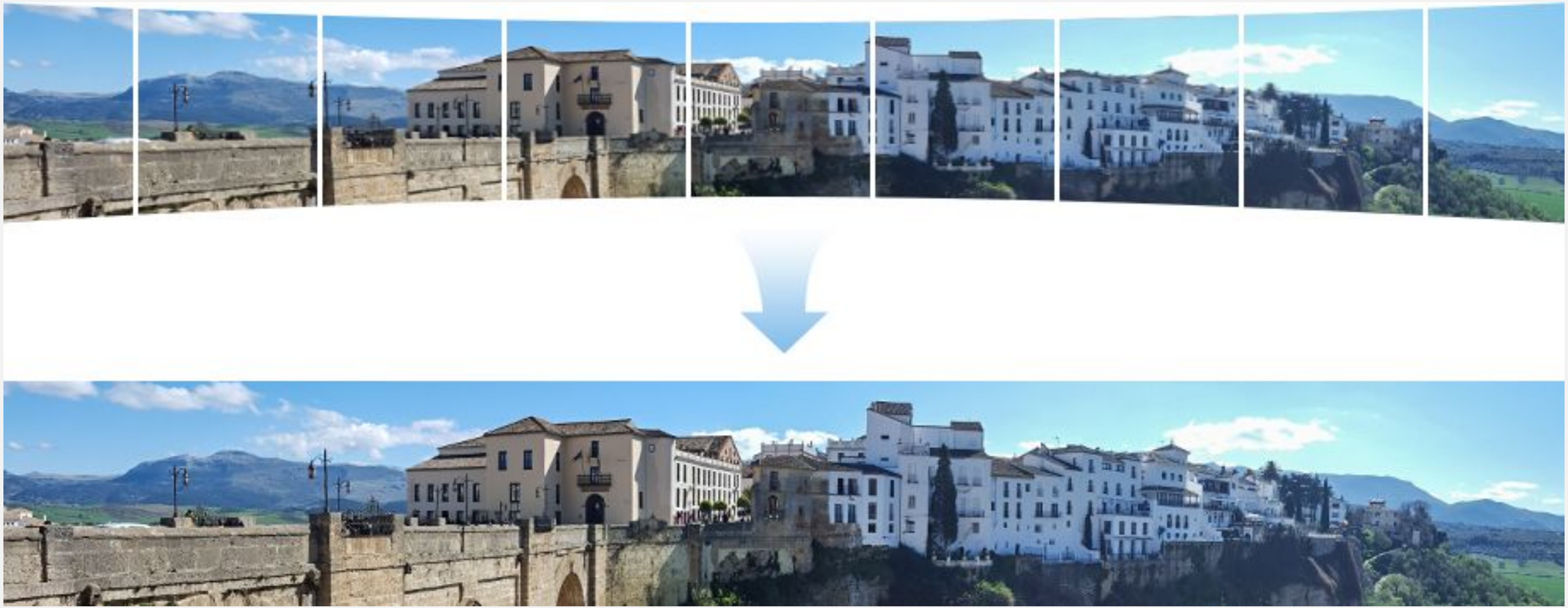
EE 698: Computer Vision

PANORAMA IMAGE STITCHING

Mukta Vedpathak
CS22BT073

Satyam Kumar
CS22BT053

Panorama image stitching is the process of combining multiple overlapping images of the same scene into a single, wide-angle or high-resolution panoramic image. This technique is widely used in photography, computer vision, and image processing to create seamless panoramic views.



Step 1: Read Input Images

```python
def createPanorama(input_img_list):
    images = []
    dims = []
    for index, path in enumerate(input_img_list):
        print(f"Loading image: {path}")
        img = cv2.imread(path)
        if img is None:
            print(f"Error: Could not load image at {path}")
        images.append(img)
        dims.append(images[index].shape)
        print(f"Image {index} dimensions: {dims[-1]}")
    return images, dims
```

- cv2.imread(path) is an OpenCV function that reads an image from a file and returns it as a NumPy array.
- The image is loaded in **BGR format** by default (instead of RGB).
- If the image cannot be read (e.g., due to an incorrect path), it returns None.
- images = [] and dims = []: These lists store the images and their dimensions, respectively.
- dims.append(images[index].shape): Extracts the shape (height, width, and channels) of the image and stores it in dims.

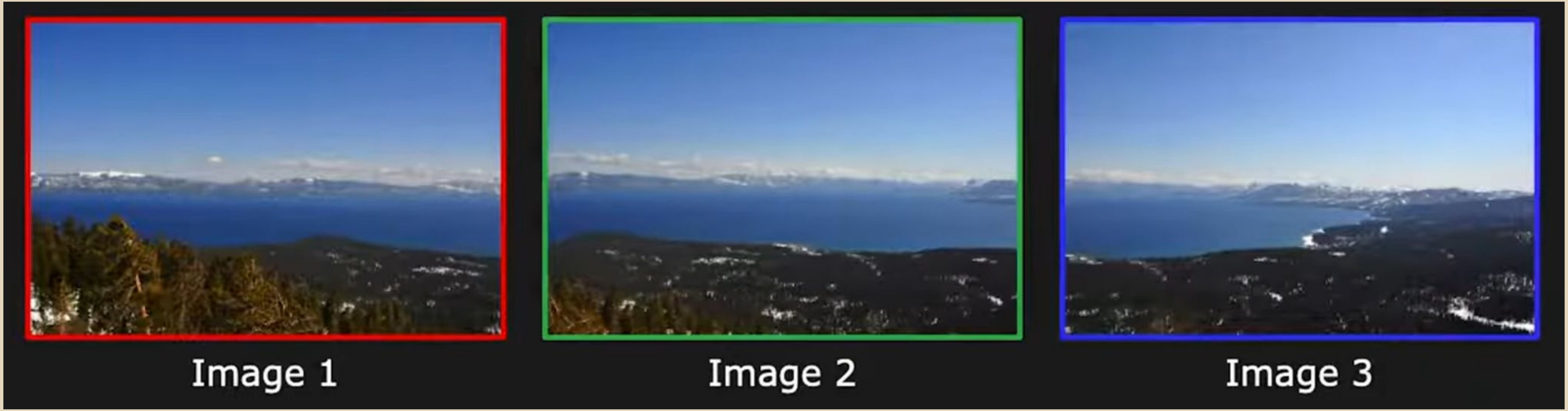


Image 1
Image 2
Image 3

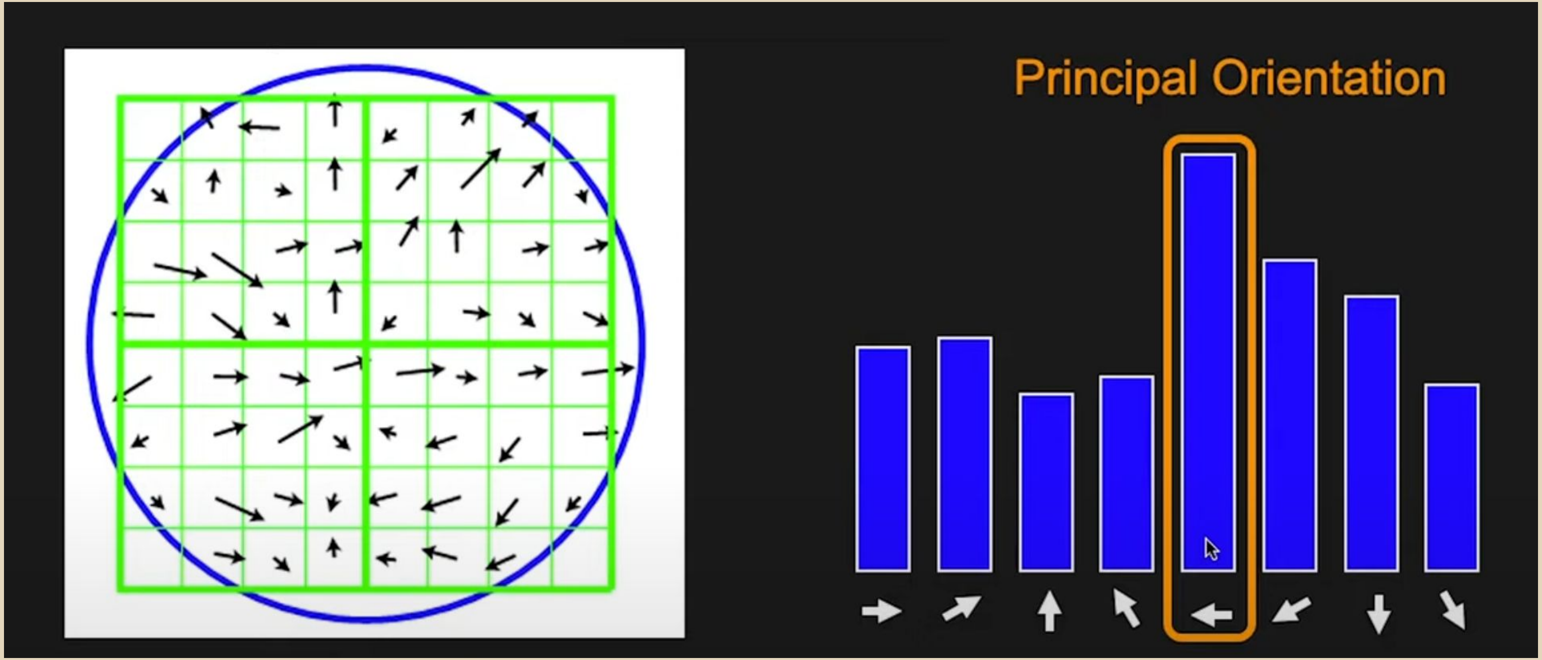
Step 2: Compute SIFT Features

- SIFT is a popular local features detection and description algorithm which uses a pyramidal approach using DOG (difference of gaussian).
- Features thus obtained will be invariant to scale. It is good for panorama kind of applications wherein images might have features variations in rotations, scale, lighting, etc.
- Here points1 is a list of key points whereas des1 is a list of descriptors

Descriptor Generation – Creating Feature Vectors:

- Around each keypoint, a 16x16 region is divided into 4x4 subregions.
- Each subregion has a gradient histogram with 8 bins (based on orientations).
- This results in a 128-dimensional feature vector per keypoint.

```python
def getMatches(image1, image2, features, ransac_iterations=150, dist_threshold=0.7):
    feature_type = cv2.SIFT_create()
    print("Detecting and computing features...")
    points1, des1 = features.detectAndCompute(image1, None)
    points2, des2 = features.detectAndCompute(image2, None)
    print(f"Detected {len(points1)} keypoints in Image 1 and {len(points2)} in Image 2")
```


Principal Orientation

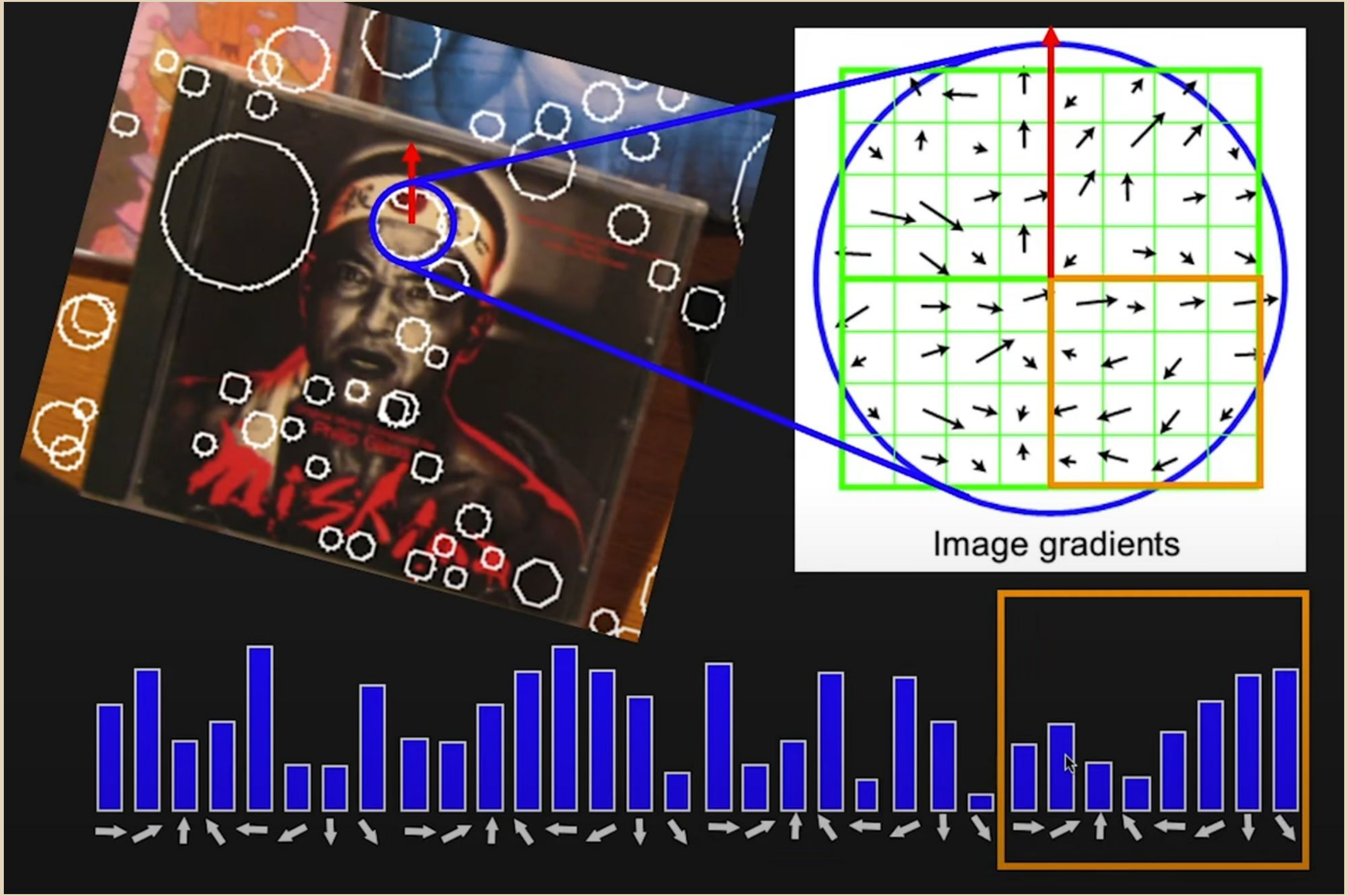


Image gradients

Step 3: Match strong interest points

```python
matcher = cv2.FlannBasedMatcher()
matches = matcher.knnMatch(des1, des2, k=2)
print(f"Total matches found: {len(matches)}")

imp = []
for i, (one, two) in enumerate(matches):
    if one.distance < dist_threshold * two.distance:
        imp.append((one.trainIdx, one.queryIdx))
print(f"Filtered important matches: {len(imp)}")
```

- **FLANN (Fast Library for Approximate Nearest Neighbors)**

- knnMatch(des1, des2, k=2) finds the two best matches for each descriptor in des1 from des2 using the k-nearest neighbors (k-NN) algorithm.

- This loop filters out weak matches using the **Lowe's ratio test**:
  one.distance is the distance of the best match.
  two.distance is the distance of the second-best match.

  If the ratio one.distance < dist_threshold * two.distance, the match is considered good.

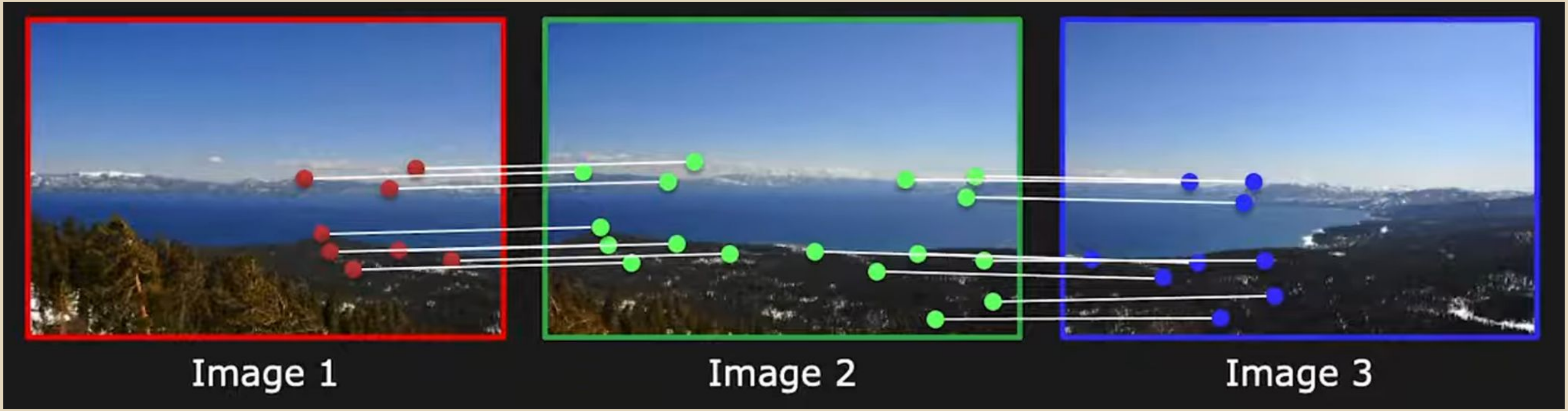
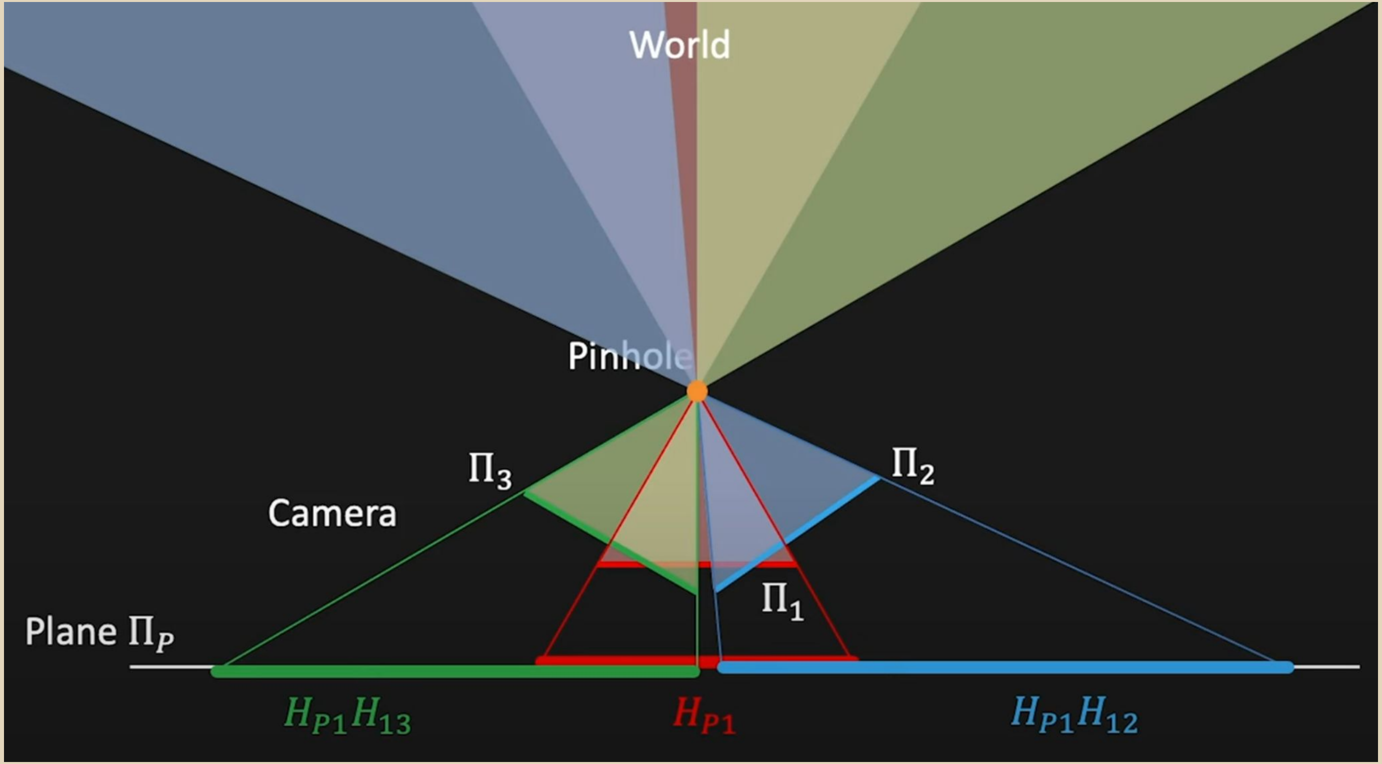


Image 1
Image 2
Image 3

Step4: Calculate the homography

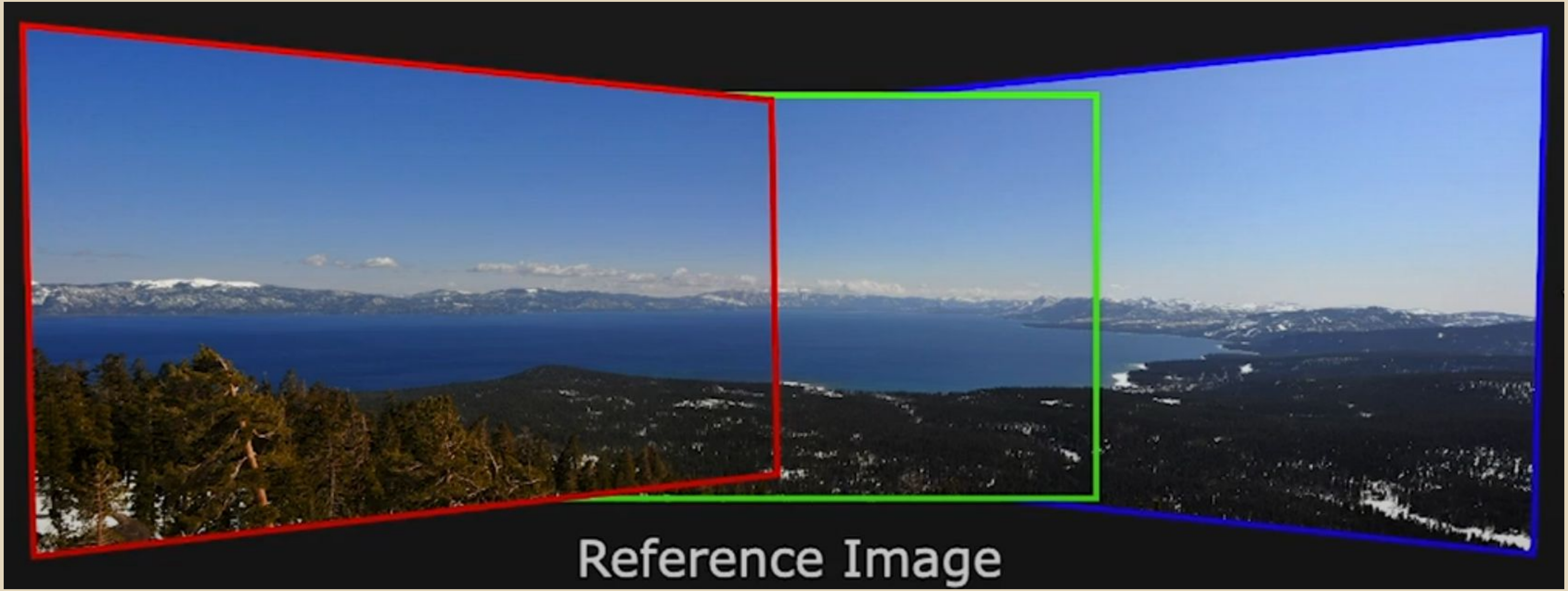
- **RANSAC** (Random Sample Consensus) is a robust method to estimate a model (homography matrix) from noisy data.
- It works by selecting a minimal subset of points (4 points are enough for homography).
- The function dlt_calibrate() calculates a homography matrix using the **Direct Linear Transform (DLT)** algorithm.
- The difference between actual and estimated points is squared and accumulated as error. The homography matrix with the smallest error (p_final) is returned.

```python
def ransac_calibrate(real_points, image_points, total_points, image_path, iterations):
    index_list = list(range(total_points))
    iterations = min(total_points - 1, iterations)
    errors = []
    combinations = []
    p_estimations = []

    for i in range(iterations):
        selected = random.sample(index_list, 4)
        combinations.append(selected)

        real_selected = [real_points[x] for x in selected]
        image_selected = [image_points[x] for x in selected]

        # Debug selected points
        print(f"Selected Real Points: {real_selected}")
        print(f"Selected Image Points: {image_selected}")

        p_estimated = dlt_calibrate(real_selected, image_selected, 4)

        # Debug projection matrix
        print(f"Projection Matrix (Iteration {i}):\n{p_estimated}")

        not_selected = list(set(index_list) - set(selected))
        error = 0
        for num in tqdm(not_selected, desc=f"Iteration {i+1}"):
            test_point = list(real_points[num]) + [1]
            try:
                xest, yest = calculate_image_point(p_estimated, np.array(test_point), image_path)
                point_error = np.square(abs(np.array(image_points[num]) - np.array([xest, yest])))
                error += point_error
                #print(f"Point {num} Error: {point_error}")
            except ValueError:
                continue

        mean_error = np.mean(error)
        errors.append(mean_error)
        p_estimations.append(p_estimated)

        print(f"Iteration {i} Mean Error: {mean_error}")

    p_final = p_estimations[errors.index(min(errors))]
    return p_final, errors, p_estimations
```


World
Pinhole
Π_3
Π_2
Camera
Π_1
Plane Π_P
$H_{P1}H_{13}$
H_{P1}
$H_{P1}H_{12}$

RANSAC (Random Sample Consensus) algorithm:

- Randomly choose s samples. (s=4)
- Typically, s is the minimum number of samples required to fit a model.
- Fit the model to the randomly chosen samples. (Homography matrix)
- Count the number M of data points (inliers) that fit the model within a specified error threshold ε.
- Repeat Steps 1–3 N times.
- Choose the model with the largest number of inliers.


Reference Image

Step 5: Let’s do the stitching…
Method 1: Max Pixel Intensity

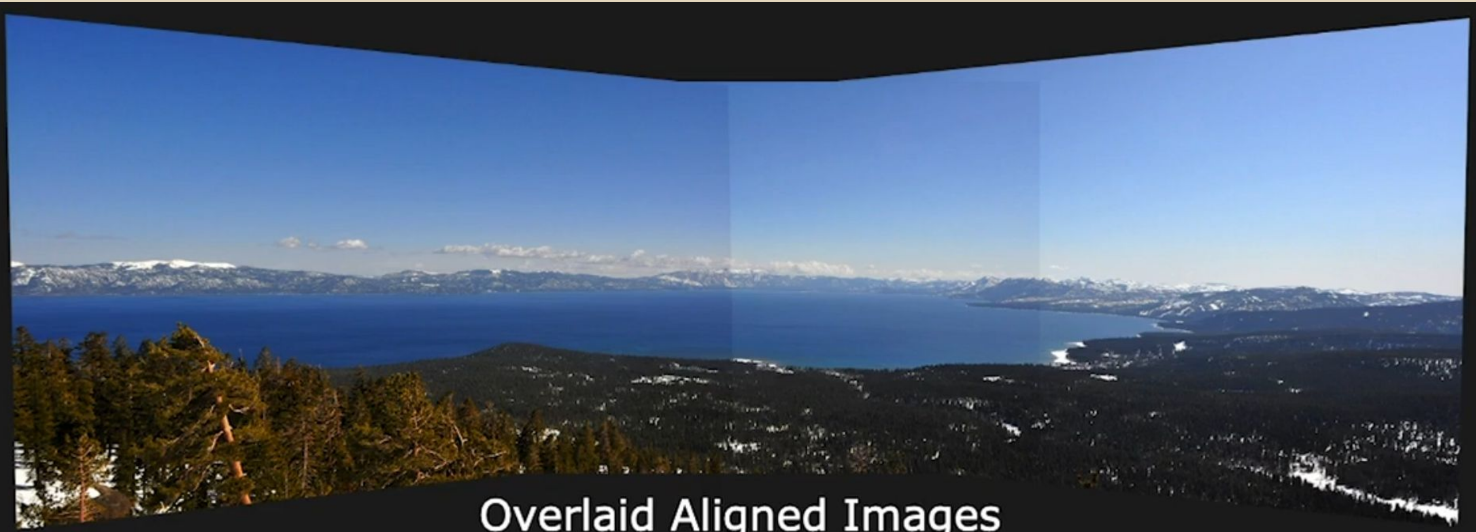
Preserves Overlapping Features

- If one image has a darker region while the other has a brighter region, this method keeps the brighter details.

Handles Empty/Black Regions

- When a pixel belongs only to one image, the other image will have a zero (black) intensity at that location.
- Taking max() ensures the non-zero value is used.

```
hmax = max(image1.shape[0], image1.shape[0])
wmax = max(image1.shape[1], image1.shape[1])
out = cv2.warpPerspective(image2, H, (4*wmax,4*hmax))
print("Perspective warp completed.")

final_out = np.zeros(out.shape, np.uint8)
h, w = final_out.shape[:2]
print(f"Final output image size: {h}x{w}")

for ch in tqdm(range(3), desc="Merging images"):
    for x in range(h):
        for y in range(w):
            final_out[x, y, ch] = max(out[x, y, ch], image1[x, y, ch]
            if x < image1.shape[0] and y < image1.shape[1] else 0)
print("Image merging completed.")

final_out = getNonZeroImage(final_out)
return final_out, out, des1[0]
```



Overlaid Aligned Images

Step 5: Let’s do the stitching…
Method 2: Weighted Distance Maps

- Uses distance transforms to calculate weights for each pixel in overlapping regions.
- Assign higher weights to pixels closer to their original image centers.
- Combine images using these weights.

```python
#Create binary masks for each image
h1, w1 = image1.shape[:2]
h2, w2 = image2.shape[:2]
hmax = max(h1, h2)
wmax = w1 + w2  # Combine width of both images

# Warp image2 to the size of the combined image
out = cv2.warpPerspective(image2, H, (wmax, hmax))
print("Perspective warp completed.")

# Save the warped image for debugging
cv2.imwrite("warped_image2.jpg", out)

# Create binary masks for each image
mask1 = np.zeros((hmax, wmax), dtype=np.uint8)
mask1[:h1, :w1] = 1

mask2 = cv2.warpPerspective(np.ones_like(image2[:, :, 0], dtype=

# Compute distance transforms
dist1 = distance_transform_edt(mask1)
dist2 = distance_transform_edt(mask2)

# Normalize distance maps
weights1 = dist1 / (dist1 + dist2 + 1e-6)
weights2 = dist2 / (dist1 + dist2 + 1e-6)

# Create a combined image to hold both images
combined_image = np.zeros((hmax, wmax, 3), dtype=np.uint8)
combined_image[:h1, :w1] = image1

# Blend the images using the computed weights
blended_image = np.zeros_like(combined_image, dtype=np.float32)
for ch in tqdm(range(3), desc="Blending channels"):
    blended_image[:, :, ch] = (
        weights1 * combined_image[:, :, ch] +
        weights2 * out[:, :, ch]
    )
```

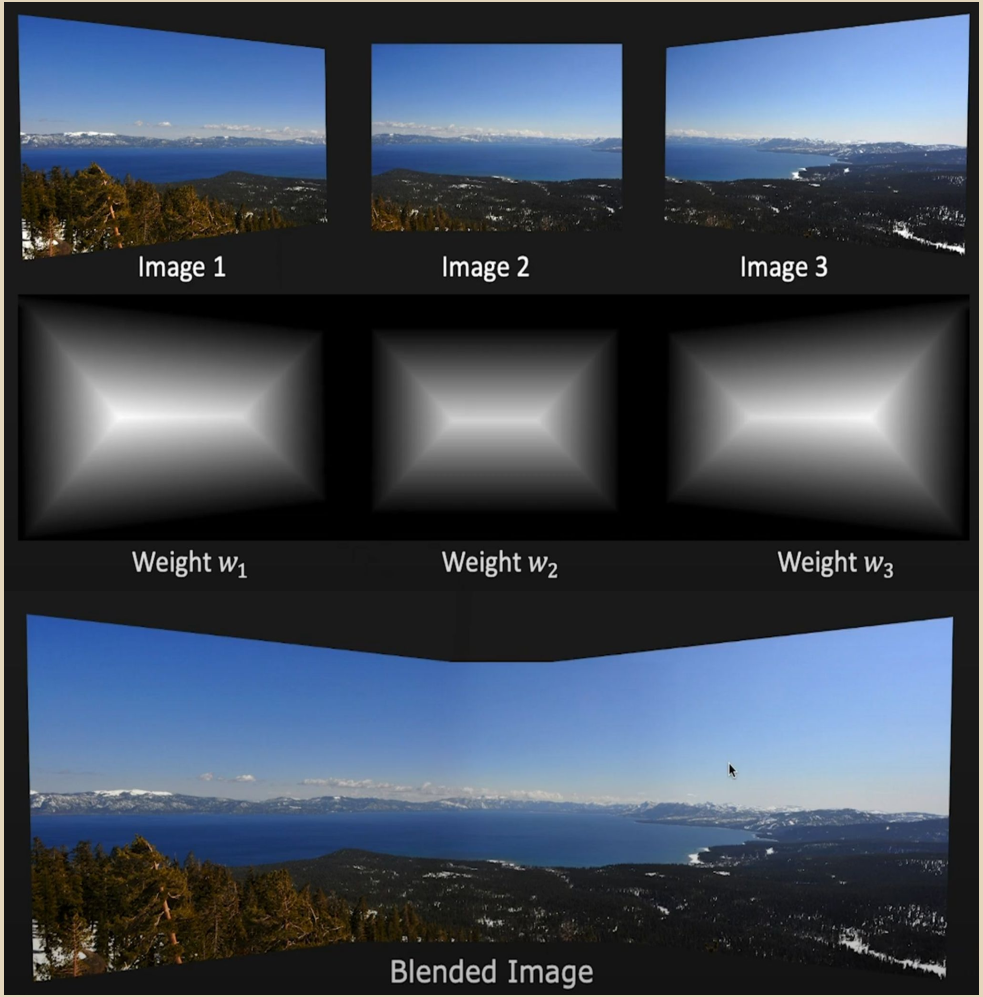


Image 1
Image 2
Image 3
Weight w_1
Weight w_2
Weight w_3
Blended Image

THANK YOU!

References: https://medium.com/tech-that-works/how-does-panorama-work-image-stitching-bf1a9f0e4fa5
https://www.youtube.com/watch?v=J1DwQzab6Jg&list=PL2zRqk16wsdp8KbDfHKvPYNGF2L-zQASc&index=1