CSCI 5563: Assignment #1
Panoramic Image Generation

1 Submission

• Assignment due: Feb 5 (11:55pm)

• Skeletal code and data can be downloaded from: https://www-users.cs.umn.edu/~hspark/csci5563_S2021/hw1.zip

• Individual assignment

• Up to 2 page summary write-up with resulting visualization (more than 2 page assignment will be automatically returned).

• Submission through Canvas.

• Use Python 3

• You will complete HW1.py including the following functions:
  - MatchSIFT
  - EstimateH
  - EstimateR
  - ConstructCylindricalCoord
  - Projection
  - WarpImage2Canvas
  - UpdateCanvas

• DO NOT SUBMIT THE PROVIDED IMAGE DATA

• The function that does not comply with its specification will NOT BE GRADED.

• You are not allowed to use computer vision related package functions unless explicitly mentioned here. Please consult with TAs if you are not sure about the list of allowed functions.
2 Overview

In this assignment, you will implement a method to create a panoramic image from a set of images using image transformation as shown in Figure 1. The pseudo code can be found in Algorithm 1.

Figure 1: (a) Given a set of images of pure rotation, you will implement an algorithm (b) to automatically create a panoramic image.

Algorithm 1 Create Panoramic Image
1: Read all images.
2: $R_1 = I_3$ \textcolor{gray}{\Comment Rotation for the first image.}
3: for $i^{th}$ image in all images except the last do
4: Load consecutive images $I_i$ and $I_{i+1}$.
5: Extract SIFT feature using OpenCV for each image
6: Find the matches between two images $(x_1 \leftrightarrow x_2)$ \Comment MatchSIFT
7: Estimate the homography between images using RANSAC \Comment EstimateH
8: Compute the relative rotation matrix, $i+1R_i$ \Comment EstimateR
9: Compute the $(i + 1)^{th}$ rotation matrix, $R_{i+1}$.
10: end for
11: Generate 3D points on the cylindrical surface \Comment ConstructCylindricalCoord
12: for $i^{th}$ image in all images do
13: Project the 3D points to the $i^{th}$ camera plane \Comment Projection
14: Warp the image to the cylindrical canvas. \Comment WarpImage2Canvas
15: Update the canvas with the new warped image. \Comment UpdateCanvas
16: end for
3 Sift Feature Extraction

![Image](image.png)

(a) Image  
(b) SIFT descriptor

Figure 2: Given an image (a), you will extract SIFT features using OpenCV.

To stitch images, you need to match keypoints in images using image local descriptor. You will use SIFT descriptor \[1\] to enable matching (Figure 2).

We will use OpenCV library for SIFT extraction given your images.

(Note) You will use this library only for SIFT feature extraction and its visualization. All following visualizations and algorithms must be done by your code.

(Note) The function for SIFT feature extraction lie in the contrib module of OpenCV library, so you need to install opencv-contrib-python package additionally. Also, in newer versions of OpenCV, SIFT module is not available due to patent issue. But you can reinstall opencv-python and opencv-contrib-python package to an earlier version 3.4.2.16 easily through following two command lines.

- pip3 install opencv-python==3.4.2.16
- pip3 install opencv-contrib-python==3.4.2.16

(SIFT visualization) Use OpenCV to visualize SIFT features with scale and orientation as shown in Figure 2 (OpenCV may different colors to visualize). You may want to follow the following tutorial:
[https://docs.opencv.org/3.4.2/da/df5/tutorial_py_sift_intro.html](https://docs.opencv.org/3.4.2/da/df5/tutorial_py_sift_intro.html)
4 SIFT Feature Matching

![Raw matches](image1)

(a) Raw matches

![Filtered matches](image2)

(b) Filtered matches after ratio test

Figure 3: You will match points between the template and target image using SIFT features.

The SIFT is composed of scale, orientation, and 128 dimensional local feature descriptor (integer), \( f \in \mathbb{Z}^{128} \). You will use the SIFT features to match between two images, \( I_1 \) and \( I_2 \). Use two sets of descriptors from the template and target, find the matches using nearest neighbor with the ratio test. You may use \texttt{NearestNeighbors} function imported from \texttt{sklearn.neighbors} (You can install \texttt{sklearn} package easily by “pip3 install -U scikit-learn”), or \texttt{scipy.spatial}.

```python
def MatchSIFT(loc1, desc1, loc2, desc2):
    ...
    return x1, x2
```

**Input:** \( \text{loc1} \in \mathbb{R}^{n_1 \times 2} \) and \( \text{desc1} \in \mathbb{R}^{n_1 \times 128} \) are keypoint locations and their SIFT descriptors where \( n_1 \) is the number of keypoints detected in image 1.

**Output:** \( x1, x2 \in \mathbb{R}^{n \times 2} \) are matches (correspondences) where \( n \) is the number of matches.

**Description:** Each row of \( x1 \) and \( x2 \) contains the \((x, y)\) coordinate of the point correspondence in \( I_1 \) ad \( I_2 \), respectively. The matching function is supposed to filter the correspondences based on bi-directional consistency and the ratio test.

(Note) You can only use SIFT module of \texttt{OpenCV} for the SIFT descriptor extraction. Matching with the ratio test needs to be implemented by yourself.
5 Robust Homography Estimation

Figure 4: You will compute a homography using SIFT matches filtered by RANSAC.

Given the matches based on SIFT descriptors, you will compute a homography that
fits to the image transformation (i.e., pure rotation).

```python
def EstimateH(x1, x2, ransac_n_iter, ransac_thr):
    ...
    return H, inlier
```

**Input:** \( x_1 \in \mathbb{R}^{n \times 2} \) and \( x_2 \in \mathbb{R}^{n \times 2} \) are the set of correspondences. \( \text{ransac}_n \text{_iter} \) and \( \text{ransac}_n \text{_thr} \) are the number of iterations and the error threshold for RANSAC.

**Output:** \( H \in \mathbb{R}^{3 \times 3} \) homography and the set of inlier indices \( \text{inlier} \in \mathbb{Z}^k \) where \( k \) is the number of inliers.

**Description:** The estimated homography will map \( x_1 \) to \( x_2 \), i.e., \( x_2 = H x_1 \). You can visualize the filtered inliers based on RANSAC as shown in Figure 4.

```python
def EstimateR(H, K):
    ...
    return R
```

**Input:** \( H \in \mathbb{R}^{3 \times 3} \) and \( K \in \mathbb{R}^{3 \times 3} \) are the estimated homography and camera intrinsic parameter.

**Output:** \( R \in \mathbb{R}^{3 \times 3} \) is the relative rotation matrix (i.e., orthogonal matrix) from image 1 to image 2.

**Description:** The computed rotation matrix is the relative rotation between the image 1 and image 2. This rotation can be used to compute the absolute rotation of the \((i+1)^{th}\) image given that of the \(i^{th}\) image.
6 Coordinate Transformation

You will warp an image to the panoramic canvas that is the unfolded surface of a cylinder where the width and height of the canvas are $W_c$ and $H_c$, respectively. A point in the cylindrical surface can be parametrized by angle $\phi \in [0, 2\pi]$ and height $h$ as shown in Figure 6. Note that the angle ranging $[0, 2\pi]$ can be converted to the pixel coordinate ranging $[0, W_c]$ using $\phi = \frac{w \cdot 2\pi}{W_c}$ (we assume the width of the canvas is equivalent to a full rotation, i.e., $W_c \equiv 2\pi$). For the height of the canvas, it is identical to the height of the source image, i.e., $H_c = H$.

To warp the image, you will apply the backward (inverse) warping scheme, i.e., you transport the pixel at the transformed coordinate in the image to the canvas coordinate, $(w, h) \rightarrow (u, v)$ as shown in Figure 5. This mapping from the canvas to the source image can be derived as a composition of mappings through a 3D point on the cylindrical surface (Figure 6): a point in the canvas $(w, h)$ is mapped to a 3D point $p \in \mathbb{R}^3$ on the surface of the cylinder, and in turn, the 3D point is mapped to the point in the source image $(u, v)$ through projection, i.e., $(\phi, h) \rightarrow p \rightarrow (u, v)$.

Figure 6 defines the coordinate system. The camera center is located at the cylinder center, $O$ where the green and blue arrows indicate the $y$ and $z$ axes, respectively. We assume the source image and canvas have the same focal length, i.e., the radius of the cylinder is the focal length, $(f = K_{11}$ where $K$ is the intrinsic parameter of the source image). Given the geometry, it is possible to derive the coordinate of the 3D point (red) in terms of $\phi$, $h$, $f$, $W_c$ and $H_c$. 

Figure 5: You will find a mapping from the panoramic canvas to the image to transport the pixel, i.e., $(w, h) \rightarrow (u, v)$. 

Figure 6: Cylindrical coordinate.
CSCI 5563: Assignment #1
Panoramic Image Generation

def ConstructCylindricalCoord(Wc, Hc, K):
    ...
    return p

Input: Wc and Hc are width and height of the canvas. You will set the Wc to be (# of images)×W/2 where W is the width of the image, and the Hc to be the height of the image H. K is the intrinsic parameter of the source images.

Output: p ∈ ℝ^{Hc×Wc×3} is the 3D points corresponding to all pixels in the canvas.

Description: Given the size of the canvas, you can generate the canvas coordinates using numpy.meshgrid, and express the corresponding 3D points using the geometry in Figure 6.

def Projection(p, K, R, W, H):
    ...
    return u, mask

Input: p ∈ ℝ^{Hc×Wc×3} is a set of 3D points that correspond to every pixel in the canvas image. W and H are the source image size.

Output: u ∈ ℝ^{Hc×Wc×2} is the 2D projection of the 3D points p, and mask ∈ \{0, 1\}^{Hc×Wc} is the corresponding binary mask indicating valid pixels.

Description: If a point is valid, the corresponding mask entry is one, and zero otherwise. A projected point is not valid:
- if the 3D point is behind the camera
- if the projected point is beyond the image boundary.

For instance, the black pixels in Figure 5 are the ones that are not valid for the first source image.

def WarpImage2Canvas(image_i, u, mask_i):
    ...
    return canvas_i

Input: image_i ∈ [0, 255]^{H×W×3} is the \(i\)th RGB image (uint8 format) with W width and H height, u ∈ ℝ^{Hc×Wc×2} is the mapped 2D pixel locations in the source image for pixel transport, and mask_i ∈ \{0, 1\}^{Hc×Wc} is the valid pixel indicator.

Output: canvas_i ∈ [0, 255]^{Hc×Wc×3} is the canvas image generated by the \(i\)th source image (uint8 format).

Description: You will use the mapping \((w, h) \rightarrow (u, v)\) to generate the \(i\)th canvas image. A new canvas image can be created, and for each pixel location in the canvas image, its pixel value can be copied from u that specifies the mapped location in the source image. You would expect to generate a canvas image similar to Figure 5. You may use scipy.interpolate for batch pixel warping with bilinear interpolation.
7 Update Canvas Image

Figure 7: Given the previous canvas image and new canvas image, you composite them using the valid pixels.

You will generate the composite of the canvas image, recursively. In Figure 7, given the canvas image generated by the first image (Figure 5) and newly generated canvas image from the new source image, the canvas image can be updated based on the valid pixels.

```python
def UpdateCanvas(canvas, canvas_i, mask_i):
    ...
    return canvas
```

**Input:** `canvas ∈ [0, 255]^{Hc×Wc×3}` is the previously generated canvas, `canvas_i ∈ [0, 255]^{Hc×Wc×3}` is the newly generated canvas image from `WarpImage2Canvas`, and `mask_i ∈ \{0, 1\}^{Hc×Wc}` is the valid pixel indicator.

**Output:** `canvas ∈ [0, 255]^{Hc×Wc×3}` is the updated canvas image (uint8 format).

**Description:** This updating function composites two canvas images based on the valid pixel indicator.

**References**