Image Processing – Alignment Exercise

itaiwar

Introduction

The goal of this exercise was to align and blend two images together, the first being in low resolution, and the second being a high-resolution part of the first image which has undergone some projective transformation. The main techniques I used to solve this included feature detection with the Harris corner detector, feature extraction using a variant of MOPS descriptors, matching of said features, calculation of a transformation using 4 pairs of points, the RANSAC algorithm for detection of a suitable transformation for the alignment and finally backward warping and blending. I implemented all these algorithms on my own without the use of external libraries (apart from NumPy and the occasional convolution with a Sobel kernel using SciPy).

Algorithm

The algorithm to solve this problem is as follows. For each image separately:

1. **Convert the image to grayscale.**
2. **Compute Harris corner responses.**
   Input: A grayscale image, a threshold, and an optional alpha channel.
   a. Compute the corner response for every pixel in the input grayscale image, apart from those with a value of 0 in the mask in the same location (if it was provided).
   b. Identify local maximum points.
   c. Get coordinates of maximum points with value greater than the threshold.
   Output: A list if the coordinates of the points from the last step.
3. **Generate descriptors for the points retrieved from the previous step.**
   Input: A grayscale image and a list of coordinates whose descriptors we need to compute.
   a. We blur the image and subsample it (by taking every other pixel).
   b. For every coordinate provided:
      i. We observe a patch around it (after it is adjusted due to the subsampling).
      ii. Calculate the average angle θ in the patch, rotate the patch by $-\theta$ degrees so that the average angle of the patch will now be 0.
      iii. Save the rotated patch in a dictionary with the key being the point it is associated with.
   Output: A dictionary of {coordinate: descriptor}.

Now combine the results:

4. **Find matching pairs of points using a similarity measure of ratio between first and second nearest neighbors.**
   Input: Two dictionaries of points as keys and descriptors as values, the first dictionary belonging to the corners of the first image and the second belonging to the second image.
   Algorithm: If we denote the set of points we got as input from image 1 as X, and those from image 2 as Y, for each $x \in X$ we calculate the following ratio: $\frac{\text{dist}(x,\text{NN}-1)}{\text{dist}(x,\text{NN}-2)}$ where $\text{NN}-1$ stands for the 1st nearest neighbor of x in Y and $\text{NN}-2$ stands for its 2nd nearest neighbor, whereby saying "nearest" we refer to the point whose descriptor has the smallest Euclidean distance to the descriptor of

point x. We say that x and his NN y are a match if there is no other x' whose NN is also y with a ratio smaller than that of point x. We find all matching pairs between the two sets of points.
Output: This function returns an array with the pairings having score less than 1, sorted by their ratio, ascending.

5. **Estimate parameters of the transformation by using the RANSAC algorithm.**
   Input: A list of pairs, a threshold ε, and an argument p with a default value of 0.99, representing the probability that at the end this step there is a chance p that of a selection of 4 randomly sampled pairs of points will be inliers of a valid transformation from the first image to the second.
   a. Initialize the array best_inliers and best_model. Set number of iterations for the following loop to be ∞.
   b. In a loop:
      i. Randomly sample 4 pairs of points from the input list of pairs.
      ii. Create a transformation matrix $\mathbf{A}$ by solving a set of 8 linear equations supplied by the 4 pairs of points.
      iii. For every pair of points (v, u) in the list of pairs:
         a) Evaluate $d(\mathbf{A}v, u) < \varepsilon$ (after conversion of v to homogenous coordinates, calculation of $\mathbf{A}v$ and then converting back to Euclidean coordinates) to determine whether the pair satisfies the transformation $\mathbf{A}$.
         b) If it does, count it as a temporary inlier of $\mathbf{A}$.
      iv. If the number of temporary inliers is bigger than the size of best_inliers, set best_inliers to be the temporary_inliers, set best_model to be A, recalculate the number of iterations left for the outer loop.
   c. Return best_model, best_inliers
      Output: The best model along with its inliers.
6. **Using this transformation, perform backward warping from the high-resolution image onto the low-resolution image, blend the images and return the result.**
   Input: The low- and high-resolution images, and a transformation matrix from the coordinate system of the low-resolution image to that of the high-resolution one.
   a. Create a transformed version of the high-resolution image by going over every row and column of a new array with the same size as that of the low-resolution image and performing backward warping. Simultaneously, create a mask for marking empty areas of the high-resolution image using its alpha channel.
   b. Blend the transformed high-resolution image with the low-resolution image using the mask.
   Output: The blended image.

Implementation Details

Apart from using NumPy for array operations, PIL to load the images, and scipy for differentiation I implemented everything by myself. A note regarding Gaussian blurring throughout this exercise: there is no particular reason the value of σ is what it is; these are values I tested which yielded good results, so I stuck with them.

1. **Conversion to grayscale.** This is done using the PIL library with function *convert("L")*.
2. **Compute Harris corner responses:**
   The calculation is performed in the same manner we learned in class, by calculating a matrix

$$M = \sum_{(x,y)\in W} \begin{bmatrix} I_x^2 & I_x I_y \\ I_y I_x & I_y^2 \end{bmatrix}$$

for each point of interest, where W is a 3×3 window with the current point centered, and giving it a score of $\frac{\text{Det}M}{\text{Trace}M}$.

To do this, we first blur the image with a Gaussian filter with $\sigma = 2$. Then, we compute I_x and I_y using a convolution with Sobel operators on the blurred image and finally calculate $I_x^2, I_y^2, I_x \cdot I_y$ for each cell. Now, we can easily compute for any point (x, y) the matrix $\begin{bmatrix} I_x^2 & I_x I_y \\ I_y I_x & I_y^2 \end{bmatrix}$ used for calculating M – we just need to sum the matrices we calculated belonging to cells in the window of interest. We store the score in an array the size of the input image. After testing, it became apparent that simply taking all points whose score is above the threshold at this stage produces limited results since there will be clusters of many points for a single feature of the original image (visualization provided later). A better solution would be to only consider points with local maximal scores to minimize the number of corner points per feature. The way I did this was by observing how local maximum points of the responses correspond to a zero crossing of their derivative from positive to negative. To find these crossings, I generated the signs of the derivatives of the responses (*np.sign*), and then differentiated again in the same direction using the *np.diff* function which calculates $out[i] = a[i+1] - a[i]$. Two adjacent cells with the first cell being positive and the second cell being negative give a value of -2. So, we take note of the cells with a negative value, these are the maximum points in the direction we differentiated in twice. We do this both for the x and y directions, keeping only points which were maximal in both directions. We finally return the coordinates of these points whose value is greater than the threshold. This collection of points is now sparser.

3. **Generate descriptors for the points retrieved from the previous step.**
   Initially I tried to create MOPS descriptors exactly as we've been taught. However, I soon realized that these descriptors were unsuitable for this use case as too many different points produced descriptors which were too similar, resulting in too many bad matches. I wanted to make the descriptors more distinct. I did this by reducing the image only once instead of twice, using a bigger descriptor and by not normalizing it. This worked.
   We initialize a dictionary to hold our descriptors with their keys. We blur the image with a Gaussian filter with $\sigma = 3$ and subsample it by taking every other pixel. We calculate the gradient angle array by differentiating the reduced image in both directions (giving I_x and I_y), and calculating *np.degrees(np.arctan2*(I_y,I_x)*)*. We then smooth the gradient angle array with a Gaussian filter with $\sigma = 1$. The size of the descriptor we will return is 21×21 pixels, having a radius of 10 around the center. For each coordinate we now calculate its reduced coordinate by integer dividing it by 2, and calculate its angle θ by averaging a window of 21×21 pixels around it. Rotation of a square while maintaining its size results in black pixels where there is now no data, since the rotation doesn't look at a wider patch than the one provided. My solution was to rotate a bigger patch and then crop it. Hence a descriptor is formed by taking a patch of 41×41, rotating it by angle $-\theta$ degrees, and then cropping it to size 21×21.
4. **Pair matching.** For the implementation I used nested for loops, a temporary array to store the ratios being calculated for the current point. A dictionary was used to prevent cases where the same point can point to two different points. At the end of the function the dictionary is converted to an array of pairs, sorted by their score, ascending. This is to permit the end user to consider only a predefined number of pairs while still maintaining a threshold on the score.
5. **Transformation calculation and RANSAC.** The transformation is calculated using np.linalg.solve(), having prepared beforehand an array of shape 8×8 containing the system of equations. These are two rows a pair gives: $\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1'x_1 & -y_1x_1' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1'y_1 & -y_1y_1' \end{bmatrix} = \begin{bmatrix} x_1' \\ y_1' \end{bmatrix}$

Using the result of this system we find matrix $\mathbf{A}$ of our transformation. Now, for all pairs of points p_1, p_2 , we convert the first point $p_1 = [x \quad y]^{\mathrm{T}}$ to the homogeneous coordinate system: $[x \quad y \quad 1]^{\mathrm{T}}$, calculate $\mathbf{A} \cdot [x \quad y \quad 1]^{\mathrm{T}} = [\mathrm{x}' \quad \mathrm{y}' \quad \mathrm{w}']^{\mathrm{T}}$, and then convert back to Euclidean coordinates by finding $p_1' = [x'/w' \quad y'/w']^{\mathrm{T}}$. We say that a pair is an inlier for transformation $\mathbf{A}$ if $d\left(p_1', p_2\right) < \varepsilon$ is true. After evaluating all points, we determine whether this transformation had more inliers compared to all the transformations before it. If so, we keep it as our current best and remeber the number of inliers. We don't run this algorithm enough times for it to go over all combinations of 4 pairs of points since this number is enormous (for 100 pairs, there are $\binom{100}{4} = 3{,}921{,}225$ different possible selections). However, if we opt to ensure in probability p that in some iteration we choose 4 inliers, then the number of total iterations comes down to $N = \frac{log(1-p)}{log\left(1-(inline/pairs)^4\right)}$, which given 40% inliers with a probability of 0.99, reduces the number of iterations to 176 (as an example). However, we may not know in advance what the ratio of inliers is, so I initially set N to be np.inf, and updated N during the run of the algorithm after finding a transformation providing more inliers than the best one before it. (As more inliers are found, the ratio $inline\ /\ pairs$ grows making N shrink). I kept $p = 0.99$ as it returned satisfactory results despite the overhead.

6. **Backward warping and blending:** We perform backward warping from high-resolution to low-resolution using the provided transformation. We initialize a NumPy array for the transformed image, and a 2D mask for the blending created by considering the alpha channel of the image, as high alpha values indicate regions to be kept. The transformed image is populated by iterating over its pixels, and determining the corresponding coordinates in the higher resolution image in the same manner we did in RANSAC. Pixels with high alpha values are assigned the RGB values from the high-resolution image, while the mask is updated accordingly with either zeroes (if kept) or ones (if discarded). To ensure smooth blending, the binary mask can be blurred with a Gaussian filter. Image blending is achieved using splatting: $result = (1 - mask) \cdot H + mask \cdot L$, where H is the transformed high-resolution image and L is the low-resolution image.

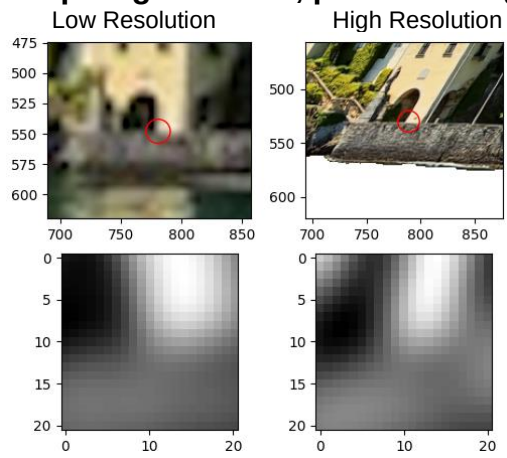
Visual Results

Harris corner responses

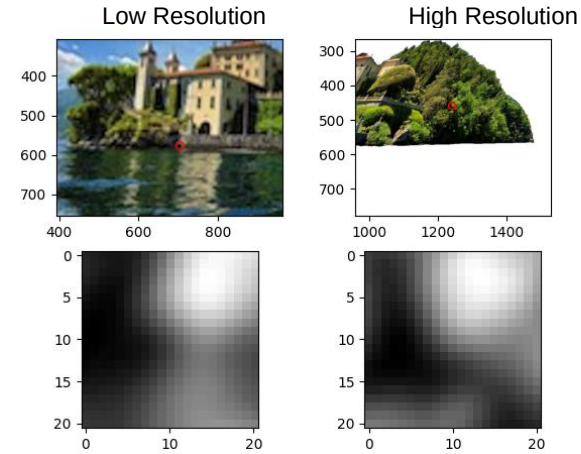
Low Resolution Threshold = 1000, used in my result | High Resolution | With maxima filtering | Without maxima filtering High Resolution, threshold = 4000

This is a visualization of the points where a response was recorded, represented as a white dot on their respective image (you may need to zoom in to see them). The right-most image is what we get when keeping all points above the threshold with no other filter: many points around the same corner give similar responses, and entire corners are left out because all their points are below the threshold. Our goal is to minimize the number of points around each corner, and maxima filtering achieves this. I chose a threshold of 1000 since it gave a decent number of corners, compared to 4000 for example.

Descriptor generation, pair matching, and RANSAC:



A: A pair of points which were matched which RANSAC returned as inliers.



B: A pair of points which were matched but were found to be outliers during RANSAC.

The images in the bottom row are the descriptors for the points circled in red in the images on top. Example A and B both illustrate a match made between two points in the two different pictures after comparing their descriptor. In both examples we can see how similar the descriptors of the matching points are (justifying the pairing), even if the points point to completely different parts of the image (in which case RANSAC will find this match as an outlier for the optimal transformation).

Backward warping and blending – results

High-res image after backward warping

Lake: Result

Desert: Result

These are my results with a Harris corner threshold of 1000 and RANSAC parameter $\varepsilon = 5$. A flaw in this solution is that the transformation we found may not always ensure that the two images are perfectly aligned, and we may not always get the same transformation. By a close examination of the left most tower in the middle figure one can see a part of the original low-resolution tower that the transformed image did not cover. We try to overcome this with blending, but that may result in a distortion of the edges of the high-resolution image which contain important details.

Conclusion

The exercise may not seem challenging at first, but diving deep into it unveils a whole different case. Never did I envision myself programmatically aligning two images without dragging and rotating the image with my mouse, not to mention implementing every aspect of it myself from scratch.
Debugging the algorithm was very challenging as it takes a long time for it to run and it had many potential pitfalls. I leveraged Matplotlib to my advantage, creating visualizations, like those provided above, to ease the understanding of the execution. Many components of this algorithm can be further enhanced, be it the point descriptors or the way local Harris corners with maxima are extracted. I am very pleased with my results. This is an exercise I will surely never forget.