Homework 4- Homographies

Algorithms and Applications in Computer Vision – 046746 – Spring 2021

Students: Joshua Melamed 332708007 | Zahi Cohen 204525612

Part 1

1.1

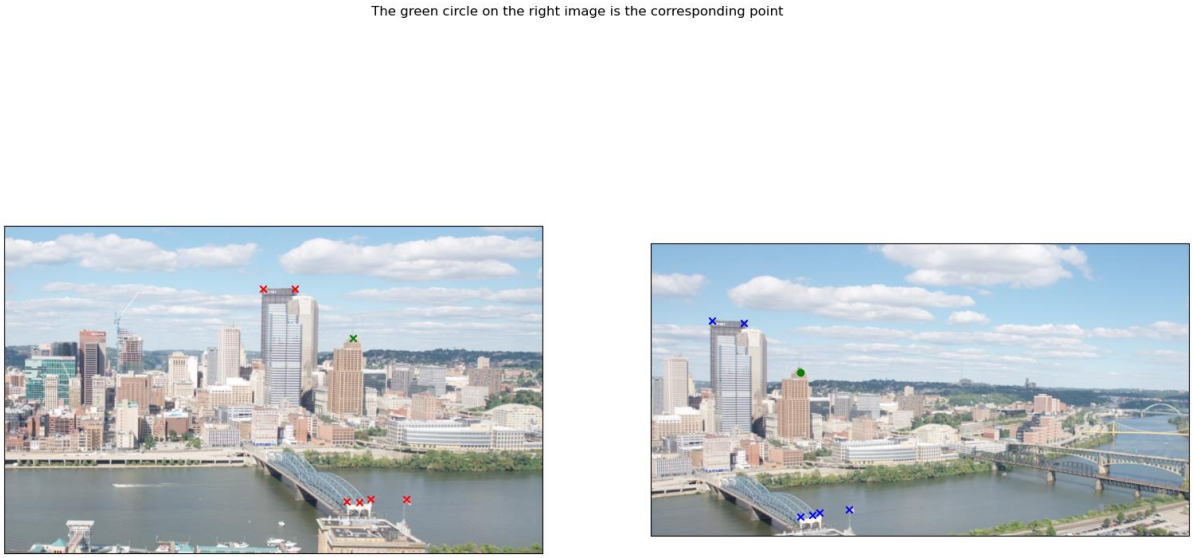
We implemented the $getPoints()$ function which enables a user to manually pick N corresponding points between two images. The function utilizes the $ginput()$ function, and draws a colored 'x' for each point chosen in each image, and returns two sets of user-chosen point pairs (in heterogeneous coordinates).

1.2

To implement the $computeH()$ function, we first constructed a matrix A like that described in the lectures - column by column - from the sets of points $p1$, $p2$. More specifically, we constructed A such that the first column of all even rows of A and the 4th column of all odd rows of A hold the corresponding x_i coordinate from $p2$. The second column of all even rows of A and the 5th column of all odd rows of A hold the corresponding y_i coordinate from $p2$. The 3rd column of all even rows of A and the 6th column of all odd rows of A have the value 1. The 7th column of all even rows of A holds $-x_i \cdot u_i$, and that of all odd rows of A holds $-x_i \cdot v_i$. The 8th column of all even rows of A holds $-y_i \cdot u_i$, and that of all odd rows of A holds $-y_i \cdot v_i$. The 9th column of all even rows of A holds $-u_i$, and that of all odd rows of A holds $-v_i$, both from $p1$.

Afterwards, we solved the equation $Ah = 0$ by finding the eigenvector V corresponding to the smallest eigenvalue of $A^T A$. The function returns $H2to1$ which is a $3x3$ matrix comprised of the elements of V that encode the homography from the second image to the first image (up to scale). An example run of the function for which $N = 6$ corresponding point pairs were used to compute H from the `/code/data/incline` images can be seen in the image below. We can see that

the arbitrary point selected in the left image (green ‘x’) is correctly projected to the right image (green circle).



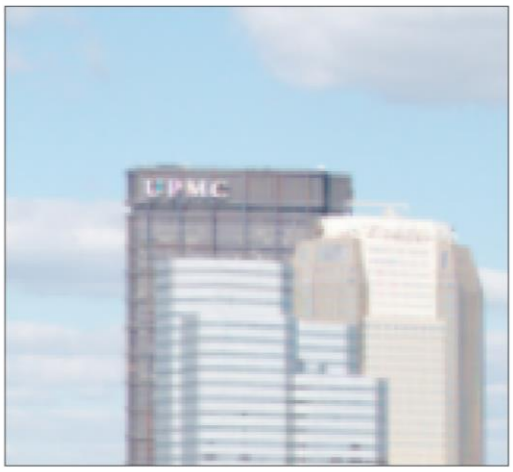
1.3

To implement the $warpH()$ function, we first implemented a function $computeOutSize()$ which calculates the necessary size of the canvas which will contain two aligned images – one warped and one regular. To calculate the output size, the matrix $H1to2$ is first applied to the four corners of the image to be warped. We then find the maximum distances along the x and y axes between the new corner coordinates. These distances are the width and height of the warped image. Afterwards, the function enlarges the canvas, so it is large enough to contain the image that does not undergo warping, while taking into consideration the locations of the corners of the warped image. The $computeOutSize()$ function also calculates the amount that the static image must be shifted down by in the y direction (*y_down_amount)* and right by in the x direction (*x_right_amount)* to be stitched correctly to the warped image. The parameter -*y_down_amount* is the y coordinate of the static (non-warped) image in the final stitched image, and the parameter -*x_right_amount* is the x coordinate of the static image in the final stitched image.

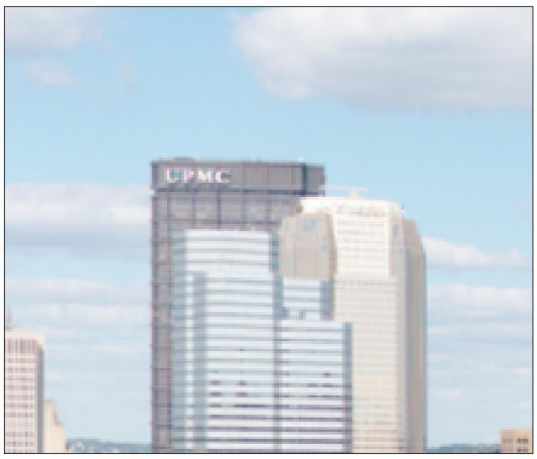
Our $warpH()$ function then creates a blank canvas which will be used to hold the warped image, with the dimensions calculated as described above (*out_size)*. It then creates the grid which is used for inverse warping. Inverse warping is utilized to guarantee that there are no holes in the warped image. The grid is defined to be of the same dimensions as the image to be warped (before warping). To fill in the pixel values of the warped image and to ensure that there are no holes, we used the $interp2d()$ function to interpolate per channel over the grid. The interpolation type can be chosen as either 'linear' or 'cubic'.

We used the $warpH()$ function to warp the *incline_L* image to the perspective of the *incline_R* image twice, once using linear interpolation, and once using cubic interpolation. The zoomed in results of both types of interpolation can be seen in the figures below. The raw images are included in the *output* folder but are not displayed here because they appear identical.

Section 1.3 - Incline_L image warped using linear interpolation



Section 1.3 - Incline_L image warped using cubic interpolation



After looking at the two images, we can see that the result of using cubic interpolation appears sharper than the result of using linear interpolation. This is logical, because cubic interpolation calculates the pixel's value based upon the 9 nearest neighbors, instead of only 4 neighbors that linear interpolation uses. However, there is a tradeoff involved: processing time. We noticed that the warping process took approximately twice as long to complete when using cubic interpolation versus linear interpolation. Therefore, we conclude that cubic interpolation should

only be used when output quality is more important than speed. For the remainder of this report, we used linear interpolation, due to time constraints.

1.4

The $imageStitching()$ function receives two images: one that has been warped and one that has not and merges them into a single image. It does so by first creating a blank image matching the dimensions of the warped image. It then 'pours' the pixels of the second image onto the blank image, according to the coordinates defined by the offset amounts that the $imageStitching()$ function was passed (*y_down_amount* and *x_right_amount)*. Then, it creates a mask of the location of all the pixels of the warped image that are zero and 'pours' the corresponding pixels from the second image into those locations. The function returns the resulting stitched image.

We called the $imageStitching()$ function on the `/code/data/incline` images, and the resulting stitched image can be seen below. We can see that the resulting stitch worked quite well: the contours of the clouds in the upper right portion of the panorama are perfectly matched up, and so are the shrubbery between the two bridges, on the upper bank of the river. The stitch is not perfect however; for example, the shrubbery on the lower bank of the river is not well aligned and we believe that this is due to imperfect manual feature selection. More accurate manual feature selection, or other more robust methods such as RANSAC would probably improve the quality of the stitch.

Section 1.4 - Stitched Incline_L and Incline_R images

1.5

In this section, we implemented the $getPoints\_SIFT()$ function, which gets two images and outputs two pairs of corresponding keypoints between two images. The function first detects the keypoints from each image and computes their descriptors. Afterwards, it uses the brute force matcher to match the keypoints. The brute force matcher works by taking a single feature descriptor from the first image and computing the distance ($L_2\ norm$) from it to all the descriptors from the second image. The corresponding features with the lowest distance are chosen as matches. This process is repeated for all the descriptors. The function returns the points of the matches, sorted in descending order by distance. We elected to use *crossCheck=true* which ensures that the matcher returns only those matches with value (i, j) such that $i - th$ descriptor in set A has $j - th$ descriptor in set B as the best match and vice-versa. We used the SIFT and matching algorithm to create a stitched image of the `/code/data/incline` images. The resulting image can be seen below.

Section 1.5 - Stitched Incline Images using SIFT

We can see that the resulting stitched image using SIFT looks slightly better than the stitched image that we got from manual feature selection. For example, the shrubbery on the lower bank of the river is better aligned between the two images. Additionally, an advantage of using SIFT is ease of use: the user is not required to manually select point correspondences. We found (in question 1.6) that the main disadvantage of using SIFT to select keypoints is that when the resolution of the images is low, SIFT has difficulty finding good correspondences, which causes the resulting panoramas to not be properly stitched.

1.6

To build the panorama images from the images of the beach and of Pena National Palace, we first performed pre-processing on the palace images. This is because our panorama algorithm expects to receive the images ordered from left to right. Therefore, we reversed the order of the palace images, such that image five was the leftmost image in the list, image four was to the right of image five, and so on. In addition, after we noticed that our algorithm could not construct a single panorama after an entire night of running, we were forced to downscale all the input images.

Our panorama algorithm $createBigPanorama()$ works as follows: the input list of images is first split up into two lists. The first list $left\_side$ contains all the images that will be on the left side of the panorama, including the center image. The second list $right\_side$ contains all the images that will be on the right side of the panorama, also including the center image. This is because after experimenting with different panorama construction strategies, we found that this approach resulted in more natural-looking panoramas. Next, our algorithm constructs the left side of the panorama. It does so by extracting the first image in the list of images: $current\_result\_left$, and iterating over the following loop:

- Extract the next image in the list
- Select points to warp a pair of images (using either SIFT or manual point selection)
- Compute H (using either $computeH()$ or $ransacH()$
- Create the intermediate stitched image using our image stitching routine, as described in section 1.4 and update $current\_result\_left$

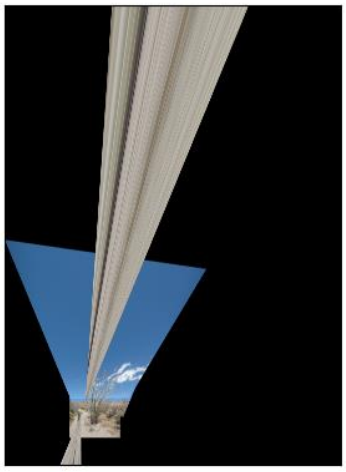
The same steps are performed for the list containing the images that make up the right side of the panorama. Finally, the two halves are stitched together to create the final panorama.

In the image below, we can see the panoramas that were created by manually selecting six point correspondences between each pair of beach images, and by using SIFT/brute force matcher to get point correspondences. To create each panorama, we downscaled the images to 60% of their original size.

The resulting images are both less than satisfactory. In particular, the lower sections of the left image are misaligned. We believe that this is due to the fact this it was difficult for us to accurately choose multiple pairs of 6 matching points, and as a result, the warps were not accurate. However, the panorama created by using SIFT looks much worse. We believe that this is since our SIFT brute force matcher returned several false correspondences, and as a result, the image is unrecognizable as a panorama. Therefore, we conclude that the advantage of manual point selection over SIFT in this case is a higher quality panorama, at the cost of convenience; it is not easy to manually select multiple point matches to create a single panorama image.

Section 1.6 - Stitched Beach Images Using Manually Selected Points

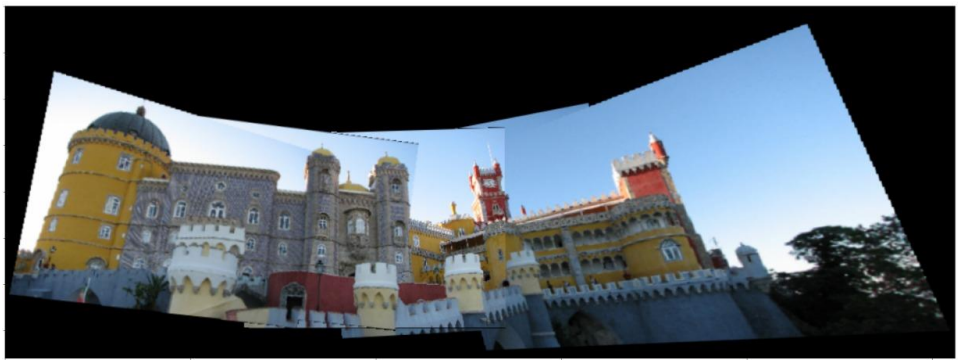
Section 1.6 - Stitched Beach Images Using SIFT



In the first image below, one can see the resulting panorama of the palace, when 6 point pairs were manually selected, and downscaling to 45% of original size was used (in the interest of time). In the second image below, one can see the panorama of the palace when the ten best SIFT features were used to warp images, and the images were downscaled to 10% of their original size. In our opinion, the panorama created using SIFT looks better than the panorama created with manual point selection. We believe that this is due to the fact that in this case (in contrast to the beach images), the images were rich enough with details and there were many 'good' features for the brute force SIFT algorithm to match. As a result, accurate homographies were computed, and high quality warps were created.

Section 1.6 - Stitched Palace Images Using Manually Selected Points

Section 1.6 – Stitched Palace Images using SIFT



1.7

We implemented the $ransacH()$ function which automatically computes Homographies between two images. The function performs the following algorithm:

- Choose 4 random indices from the point pairs $p1, p2$.
- Compute the homography H from the four point correspondences.
- Calculate the total distance according to $L_2\ norm$ (squared) between the points in the first image, and the projected points based on the calculated H.
- Count the number of inliers.
- Update maximum number of inliers.
- If new maximum number of inliers, update list of inlier indices.
- Repeat $nIter$ times.
- Calculate H based on the largest number of inliers and return it.

We chose default hyperparameter values of $nIter = 250, tol = 3$ because we found that in most cases, they provided a reasonable running time along with accurate output (H). We used the $ransacH()$ function to compute the homography between the beach images and the Pena National Palace images, both with manually selected points, and points chosen using SIFT. We then used the homographies to stitch the images together, and the results can be seen in the figures below.

Section 1.7 – Stitched Beach Images using Manually Selected Points and RANSAC

As we can see from the images above, both the panoramas are of higher quality than the beach panoramas that were created without RANSAC. We believe that this improvement is due to the ability of RANSAC to calculate the homography H based on inliers only. As a result, the calculation of H is more robust when it comes to dealing with inaccurately chosen point pairs, whether they were chosen manually, or using SIFT. We can see an especially noticeable improvement between the beach panorama that used SIFT from the previous question (that did *not* use RANSAC). Therefore, we conclude that RANSAC greatly improves the robustness of the computed homographies to mismatched features and is especially suited to be used with SIFT.

In the images below, one can see the panoramas created using RANSAC along with manual point selection and SIFT. Both the results look decent in our opinion. We can see that the stitching itself is good in both panoramas. However, the images that comprise the panorama created using SIFT are slightly better aligned. Neither of the palace panoramas are noticeabley different from those created without the use of RANSAC. As such, we conclude that RANSAC indeed improves the quality of panorama constructions when used to stitch images that do not contain many details (such as the beach images). This is because with RANSAC, we do not consider outliers when computing the homography matrix H. However, if the images contain sufficient features, use of RANSAC is unnecessary, and even incurs a slowdown.

Section 1.7 - Stitched Palace Images Using Manually Selected Points and RANSAC

Section 1.7 - Stitched Palace Images Using SIFT and RANSAC

1.8

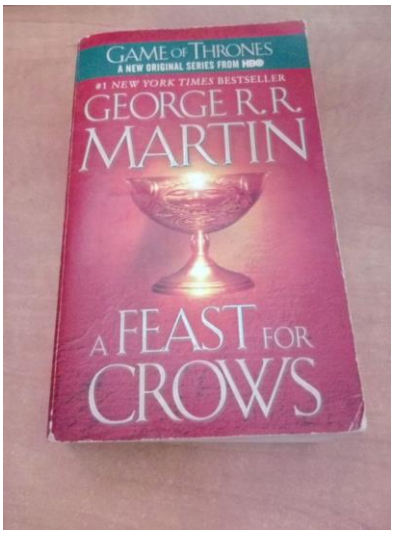
We went outside and took 3 pictures of the Taub Computer Science Building. The 3 images can be seen below.

We then used our functions to create a beautiful panorama. We elected to manually detect keypoints, and to use RANSAC to compute the homographies. The resulting panorama can be seen below and we are satisfied with the result.

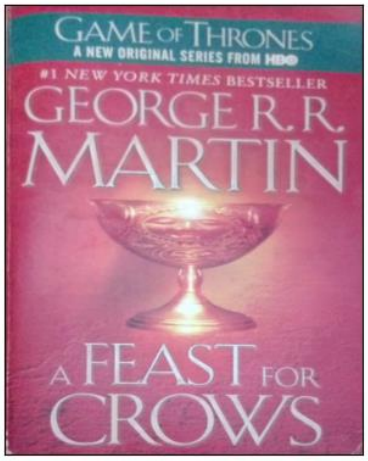
Part 2

2.1

We implemented the $create\_ref()$ function, which receives the path of an unaligned, uncropped photograph of a book and warps the image into a rectangular shape in the $x - y$ plane. The function first prompts the user to choose the corners of the book in the input image, in clockwise order, in a similar manner to the $get\_points()$ function. The function then creates an image of zeros, whose dimensions are 440×350 $[pixels]$ (approximately the aspect ratio matching the book) whose corners are the points that will be used to compute a homography. The function then uses our function $computeH()$ to compute the homography that maps the input book to the rectangle defined by the image of zeros. Afterwards, the function uses our function $warpH()$ to warp the input image as if it were photographed directly from the front. This image ref_image is returned. The original image (left), and the resulting reference image



Section 2.1 - Result of create_ref() on A Feast For Crows

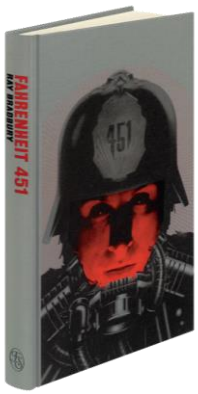


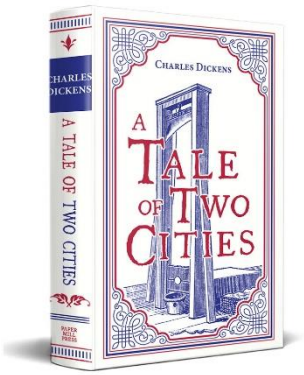
(right) can be seen below. We can see that the reference image looks quite good and is comparable to the provided example.

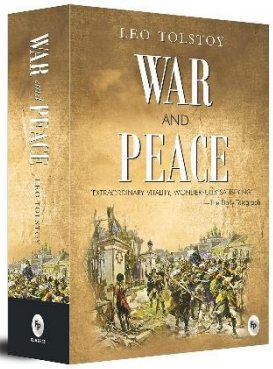
2.2

We took another picture of the same book, but this time, placed it in a scene with additional books around it. The scene can be seen in the image below.

Afterwards, we took 3 images of other books and stitched them in the scene above. The 3 images that we used (before warping) can be seen below.







We implemented the $im2im()$ function as follows: the function receives the reference image, which is already aligned by the $create\_ref()$ function, and the path to the image of the scene. The user is asked to select the corners of the book in the scene which is to be replaced. Afterwards, the function creates a rectangle whose corners represent the corners of the reference image. Then, the function uses our $computeH()$ function to compute the homography between the reference image and the book in the scene. Then, the function uses our $warpH()$ function to perform the actual warp, which warps the reference image to the perspective of the book in the scene. Finally, the function creates a mask, which corresponds to all the locations where the image of the warped reference image is zero, and 'pours' the pixels from the image of the scene into those locations, and returns this stitched image. We ran the $im2im()$ function three times using the images of books shown above. One of the examples of the image stitching results can be seen in the image below.

Section 2.2 - Final result of book stitching - Example 3

We can see that the resulting image looks very real. The placement of the corners of the warped book cover is not perfect, but we believe that this is a human error, and could be rectified by selecting the locations of the corners of the book in the scene more accurately.