Final Project for CS 372

*Riley Kopp*

*Jonathan McKee*

# Algorithm, Application, Language Choice

* *Direct Linear Transform (Homography Warping)*
* *Image Stitching*
* *Python 3.6.8*

# Where It Is Used

*Homography warping is accomplished by taking matching points from two images and creating a homography matrix. Image warping works by creating a 3x3 matrix that allows any point from the first image to be translated to the correct (x, y) coordinates in the second image. The DLT algorithm is one of main methods of generating this matrix.*

## Other applications

* *Calibrating camera*
* *Placing videos or still images on a green screen.*
* *Implanting an image on a surface inside of another image*
* *Creating a 3d image representation from 2d images.*

## Alternative algorithms

* *Conic Correlation*
* *Line Correlation*

*All 3 algorithms are used to create the 3x3 homography matrix to warp the first image to be drawn into the second image. In addition, all 3 algorithms benefit greatly from normalizing the data before calculating the matrix. Normalizing the data before calculation allows the homographic algorithms to be more tolerant of noise and shifts in the planes. It is possible to use more than the minimum number of pairs, but the improvement to accuracy is gained by eliminating outliers instead of making the conversion process more accurate.*

*The DLT method works on corresponding points and relies on the matching of the points to be accurate. Because each correspondence generates two points of data in the homography matrix, we need at minimum four pairs of corresponding points.*

*The Line method on the other hand uses the cross product of two lines to determine the transformation matrix. This means we need a minimum of two pairs of corresponding points. It is also possible if we do not have the points we need in the image to generate some of the missing data using the line equations for the two matching lines.*

*The conic method uses conic shapes to match features between the objects. The conic descriptors allow us five degrees of a freedom, so we need at least two pairs of conics to achieve our eight degrees to fill in the homography matrix.*

## Reason for choice

Image stitching is a popular technology that is becoming more common place with everyday technology. Most newer cellphones now have the option to create panoramic photos in real-time which requires expanding the image stitching process to run multiple times in quick succession. There are multiple ways to accomplish image stitching, but to create a panoramic photo in real-time at high resolution requires improving the algorithm’s speed and accuracy to allow quicker matching.

One of the first steps of picking the algorithm we wanted was looking at the simplicity of the algorithm. Between the DLT style algorithms and other algorithms using different coordinate systems, the DLT style fit better with our understanding and allowed us to avoid additional post processing needed when switching coordinate systems.

After that we had to decide on what pre-processing style we wanted to use. OpenCV provides several different matchers and feature detectors each with their up and downsides. The Harris corner detector is simple, but lacks robustness when dealing with rotation, scale, and other types of transformations. SIFT is robust when it comes to dealing with invariants but SIFT normally takes a longer time to run and is proprietary. This left our main decision between the ORB detector and the AKAZE detector. The ORB detector was OpenCV’s answer to the propriety SIFT detector and the AKAZE detector was the FAST improvement to the KAZE detector. Because of that we opted to try out the newer AKAZE detector.

The next decision was to decide the max number of matches before moving into the RANSAC loop. The smaller the cap we picked, the faster the algorithm would run, but with the potential for a poor homography matrix. Therefore, we decided a 90% cutoff range would be good for both the nearest neighbor cutoff and the RANSAC population fit cutoff.

# How Your Project Works

* Find unique features in image 1

We use the AKAZE feature detector built into Open CV for python. The AKAZE feature detector is a multiscale feature detector that is rotation and scale invariant. To use the feature detector, we first load the image file into an array. Because of the way opencv loads the file we then need to convert the image from Blue, Green, Red into Red, Green, Blue. We also then create a grayscale copy of the image to increase the speed of processing.

* Find unique features in image 2

We then repeat the entire process with image 2.

* Find matches between features lists

Next, we use the brute force matcher to check each of the features found in steps 1 and 2 for matches. This is the brute force matcher because each feature in list 1 is checked against each feature in step 2 and the features with the highest similarity percent are selected as the match.

* Limit number of matches using nearest neighbor search

We use a nearest neighbor search to reduce the total number of matches. The nearest neighbor search checks the distance between matches for the first and second highest match for a pair of features and then divides them to create a ratio. We can then use this ratio and exclude any points with a ratio over a selected threshold. The theory behind this search is that if our feature has a high match percent to two or more features, then we have no way to determine which match is the correct one, so we exclude that feature all together. If the ratio is low, it means that the chance that we have the wrong match is low and we can then keep that match.

* (optional) Display the matching features

We have the option at this point to draw the first 50 highest ratio matches with a connecting line to allow a visual confirmation of a match.

* Use RANSAC to determine first homography matrix over threshold

RANSAC uses a random sample of 4 matches to calculate a homography matrix. We then compare the correctness of this matrix by applying it to all the points in image list 1 and then checking if they are within a tolerance of their matching point in image list 2. Once all the points are checked we divide the correct points by the total number of points to get our population fit score. The method of RANSAC we implemented exits the RANSAC loop as soon as the population fit is above the threshold returning the calculated homography matrix.

* + Calculate the homography matrix

The homography matrix is created by crafting an 8x8 matrix by layering the points of each match 2 rows at a time. Once all the rows are stacked together the matrix is ran through numpy’s SVD function in order to decompose the matrix into 3 smaller matrices (U, S, V). The V matrix is a 3x3 matrix that contains our homography matrix. But to make it work we normalize it by dividing all elements by the last element in the matrix.

* Combine and draw matched images

The first part of this step is preparing a new array to place the image into. We use the sizes of the original color images to estimate the size of the new image. We then copy image 2 directly into the new array. Next we multiply the the (X, Y, 1) coordinates of every pixel in image 1 by the homography matrix and place the pixel value into the calculated location in the new image. Once the entire image 1 has been processed we crop the new image by looking for the outer borders of the image that all contain 0 (black). The last step of the programis to output the combined image so the user can now see the two images combined into 1.

## Correctness (team only)

*[Formally, justify why your algorithm is correct or why it works. You must formally use loop invariant, pre-and post conditions, etc. Any and all properties for a given algorithm, if applicable, are required. Assume you are speaking to someone who has completed this course]*

# Run time

*[Name and explain why the project has this big-O run time as the theoretical run time* **for your implementation***]*

*[A* ***fully*** *labeled graph* runtime graph for varying n with a minimum of 10 points goes here. It **must clearly show** the run time. If you have more than 1 value that affects input, you may just vary one for the graph, but be clear which one you used. Two graphs would be ideal, and you may use a minimum of 5 points for each in this case)

(team only): Formally prove the run time through instruction counting, probability, or recursion analysis depending on your problem. You **will need** to use pseudocode to prove this.

# Built-in Code Correctness Tests

(team only, you must have a minimum of 6 rather than 3 built-in tests)

|  |  |  |  |
| --- | --- | --- | --- |
| ***Test Case*** | ***Description*** | ***Input*** | ***Actual output*** |
| *[test 1]* |  |  |  |
| *[test 2]* |  |  |  |
| … |  |  |  |

Alternatively, you may have this in the following format:

## Test 1; name

Description

### Input

The input

### Output

The output

# Program usage or README

[OPTIONAL, and only used if needed]

# References

<https://docs.opencv.org/3.4/dc/dc3/tutorial_py_matcher.html>

<https://web.mit.edu/be.400/www/SVD/Singular_Value_Decomposition.htm>

<http://www.epacis.net/ccis2016/papers/paper_121.pdf>

E. Dubrofsky, R. Woodham, Combining Line and Point Correspondences for Homography Estimation. ISVC ’08 Proceedings of the 4th International Symposium on Advances in Visual Computing, Part II, 2008. 202 -213

E. Dubrofsky. Homography estimation. PhD thesis, UNIVERSITY OF BRITISH COLUMBIA (Vancouver, 2009)

J. Kannala, M. Salo, and J. Heikkilä. Algorithms for computing a planar homography from conics in correspondence. In British Machine Vision Conference, 2006. 118