Detection, Tracing and Locating Cracks in Structures

Authors:

- Tashongedzwa Madondo - madondo2
- Teague Mitchell - thm2
- Rojas Pelliccia, Máximo - mgr9

Abstract:
Cracks occur naturally in all structures and pose difficult and dangerous situations to the integrity of the materials. Some cracks are identified in time to repair them without much effort, but the majority are discovered after they are big enough to impact negatively on safety. We expect our program to identify cracks without the need of a human supervisor making it possible to detect them as they start forming.

To develop this application, we used python3 as our main coding language and the following libraries to implement the different parts of it. We used numpy, scipy, skimage, cv2, matplotlib, pylab, and PIL. To detect the cracks, we use edge detection and filtering, and to locate and trace them, we use SIFT detectors, homography transformations, and Ransac.

1. **Introduction:** Define and motivate the problem, discuss background material or related work, and briefly summarize your approach.

Cracks pose health hazards in every structure they show, being immediate or not. How many times have we walked down the street and sprained an ankle or tripped due to cracked sidewalks, or suffered from leaks due to unnoticed cracks? With our program, we aim to quickly identify cracks formed, and forming on surfaces automatically without human intervention.

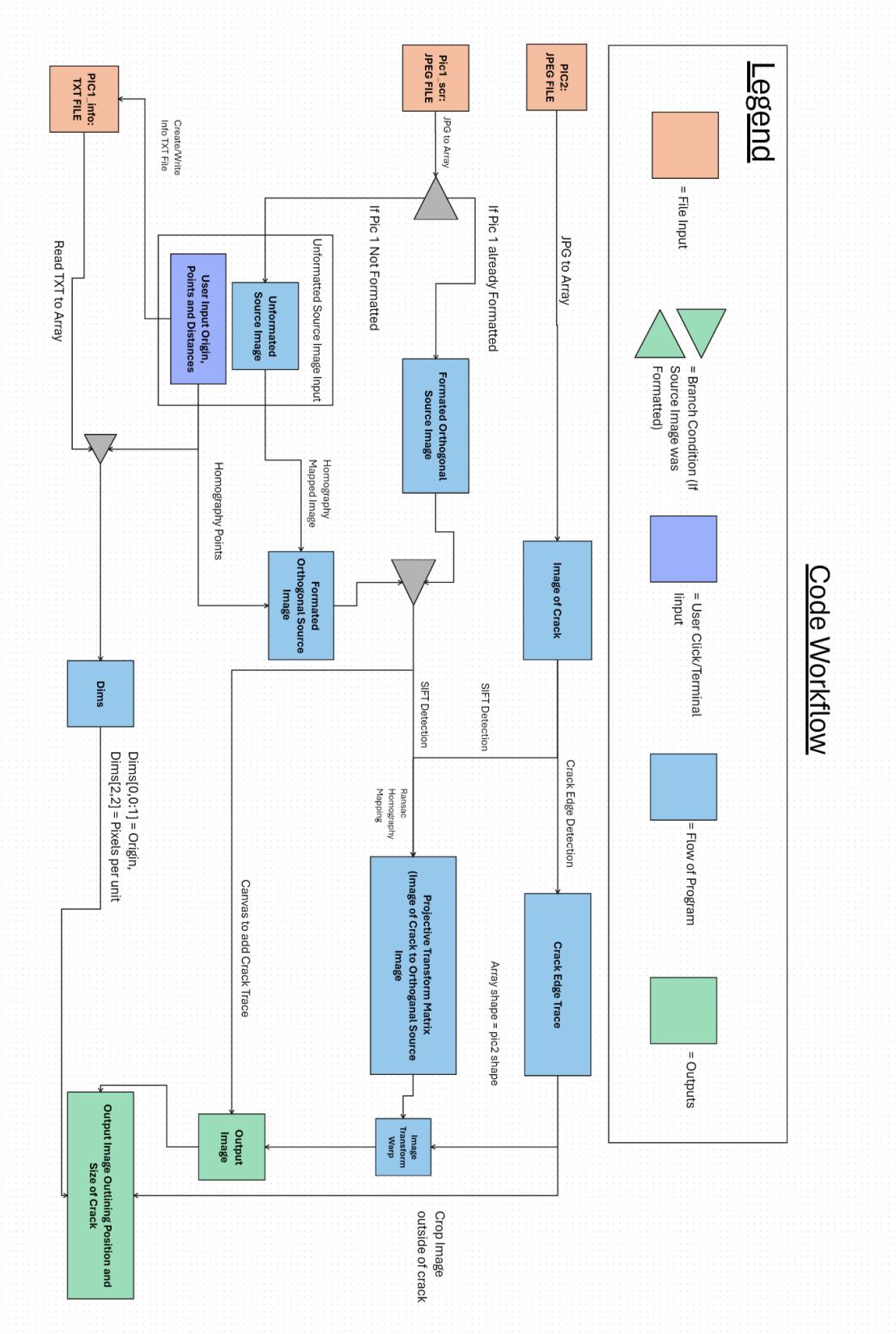
Through assignments two and three, we learned to implement SIFT detectors, homography transformations using the Ransac algorithm to choose the best one, and image stitching. This work we did proved very valuable when first starting to develop the project since it let us start from a known place and not from scratch.

To develop the project, we used Discord as our main communication channel, texting for basic interaction and hopping into voice channels to discuss points when needed. The program first matches both images using SIFT detectors and then proceeds to find the crack on the second image with edge detection. Afterward, through Ransac Homography Mapping, the crack and the complete image are stitched together to show the tracing of the crack on the output image.

2. **Details of the approach:** Include any formulas, pseudocode, diagrams -- anything that is necessary to clearly explain your system and what you have done. If possible, illustrate the intermediate stages of your approach with results images.

The program aims to return an orthogonal trace of a crack along with the dimensions of the traced crack. The crack detection program utilizes edge detection, SIFT, homography, Ransac, and orthogonal unwrapping/measuring. In execution, the source image is first orthogonalized, then the crack image homography matrix is computed with Ranac to find the projective transformation of the crack image to the source. The crack image is also put into a crack edge detection program to get the trace outline of the while retaining its original size. The projective transformation is then applied to the trace outline and laid over the source image to get the output image, further measurements are done of the trace outline array to computed properties such as position and size. Overall, the inputs (source image, crack image, source image properties text file) are processed to produce the outputs (output image, and annotated output image).

Code workflow (landscape for better visibility)



Inputs:

Pic1: JPEG File - This input is the source image for which the crack traces will be projected onto. This input must be a jpeg image of the flat plane of the concrete that completely encapsulates the image of the crack. This image can either be pre-formatted or unformatted; If image is not formatted the user will be prompted to click on 4 points in the image that form a rectangle (points may not be rectangle in source image but must be rectangle in real world), followed by 2 integers specifying the real world distance between the top-left point clicked (aka the Origin) and the top-right point clicked, and the real world distance between the origin and the bot-left point clicked. The 4 points and 2 distance given are then used to compute a homography matrix which then warps the non-orthogonal matrix into a formatted orthogonal matrix. Once a source image has been formatted, an output image jpeg (Pic1_scr) and an info txt (Pic1_info) are created to allow for pre-formatting inputs. This process is referred to as Orthogonalizing the source image throughout the paper. If Pic1 was formatted correctly prior to execution of the code than the origin and distance information is retrieved from the pic1_info TXT file. Pic1 is required to be unwrapped and orthogonal for the measurement method used function.

Pic2: JPEG File – This input is the image of the crack that is to be traced. This image must be contained in the source image to correctly find the mapping between the 2 images.

Pic1_info: TXT File - This input is a text that holds the array data for the dimension and origin of Pic1. If Pic1 is preformatted, the info file should have a list of the origin, the reference points, the reference distances, and the number of pixels per unit of measurement. If Pic1 was not formatted the user input is formatted into an array and written to the TXT file.

Outputs:

Output Image: JPEG File - This output is the image of the orthogonal/orthogonalized source image with the projective transformed crack trace. The crack trace should be highlighted with a white color. The name of this output will be the same as pic2 but will have a “_output” at the end of its name.

Output Image Outlining Position and Size of Crack: Matplotlib Plot and JPEG File - This output is a window that opens to the user outlining the dimensions of the detected crack and the position from the origin specified in the pic1_info array. The name of this output will be the same as pic2 but will have a “_output_dimensions” at the end of its name.

Workflow:

JPEG to Array: The program begins with the specified inputs passed in as pic1 (source image) and pic2 (crack image) (see input specifications above) after ensuring the source image is unwrapped and orthogonalized the image is converted to an array and is ready for SIFT Keypoint Detection.

Pic2 Formatting: The crack image is first converted to an array. 2 copies of the array are initialized; one of the array copies will be used for crack edge detection while the other array copy will be used for SIFT Keypoint Detection and homography mapping.

SIFT and RANSAC Homography Mapping: The homography mapping utilizes SIFT key detection, Ransac, and homography warping. First Sift Keypoint Detection is used to find all distinct areas of the concrete. Since concrete by nature has very distinct patterns, the accuracy of the keypoint detection is very close when it comes to the pair matching using the Euclidian distance. After getting a set of pairs matches that are below a threshold of 30000, a homography matrix is computed using Ransac. In our Ransac algorithm, we estimate a potential homography matrix using 4 random points in the set of pair matches. We than count the number of inliers (number of points within a square distance of 50) in that matrix estimation from the rest of the pair match set, this process is then completed 1000 times and the matrix estimation with the greatest number of inliers is returned and used to transform the crack image.

Crack Edge Detection: The crack detection code detects the "best" crack (i.e. the longest prominent contour) in the provided image. We used adaptive thresholding by trying multiple parameter sets to find the parameters that result in detecting the longest crack, as well as morphological operations to bridge small gaps between crack segments to ensure the entire crack is detected. OpenCV is used for skeletonization and contour detection. The crack detection functions are run for each set of parameters and the longest contour found is taken and those ideal parameters recorded, so that the code is robust to different image and crack sizes without manually having to modify parameters for each image. Once the best crack is found it is drawn on a blank canvas and its length in pixels and centroid are measured and outputted as well.

Crack Edge Trace Warp: Once both the crack trace image and the homography matrix are computed, the trace image is warped and then laid over the orthogonal source image to generate the output image (pic2_output). This output image will look exactly like the orthogonal source image but will have the crack trace in white.

Outputting and Annotating: After the crack trace is computed, a copy of the image is made and cropped to capture the smallest rectangle containing the crack. The size of that rectangle and the position of that rectangle relative to the origin are returned to annotate the dimensions on the output image. The program takes the output image, annotates the crack trace, creates the jpeg (_output_dimensions), and displays a matplotlib output to the user. The dimensions are

given based on the units given by the user in the info TXT file; however, the annotation will only say units so the user must remember the unit measure they input whether that be m, ft, cm, miles, etc.

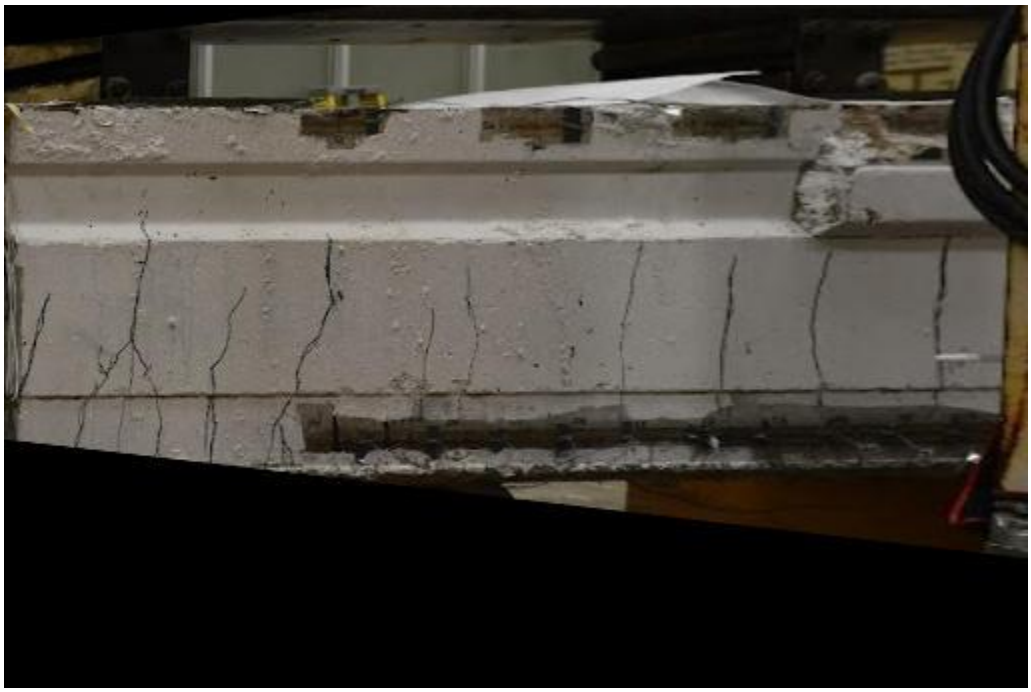
3. **Results:** Clearly describe your experimental protocols. If you are using training and test data, report the numbers of training and test images. Be sure to include example output figures. Quantitative evaluation is always a big plus (if applicable). If you have example or demo videos, put them on YouTube or some other external repository and include the links in your report.

Protocol: Our experimental protocol starts with acquiring your two input images. The first should be an image of a location, such as a sidewalk. The second should be a closeup of a crack you want to be identified and measured in the first image. Additionally, ideally know the dimensions of your first input image, such as the length and width of the depicted sidewalk. Once you have put your input images into the code, you can run the code. Once the code has begun running, you will be prompted to select four corners of your first image and the length from the top left corner to the top right corner and bottom left corner. The code will use these inputs to orthogonalize the image to help with matching the crack and measuring its length. The code will then output images showing keypoints, which you can close to progress the code, at which point it will run the crack detection code on the second image and output the crack trace and the crack measurements.

Examples:

This example is of a bridge beam being tested for crack growth under stress. The first image is this:

I placed the corners at the corners of the beam in the image, and gave the measured distance as 3 meters long and 1 meter tall. This gave me this orthogonalized image:



The crack image I wanted to detect was this:

My outputs looked like this:

img1 shape: (341, 512, 3)

img2 shape: (202, 79, 3)

Best Inliers Count: 12 Total Distance: 47.30163743954743

Homography:

[[-1.02346586e-02 8.09932716e-04 1.31152699e-01]

[2.56105907e-03 -1.31003377e-02 9.91122631e-01]

[2.78016813e-05 2.05593161e-05 -1.38317595e-02]]

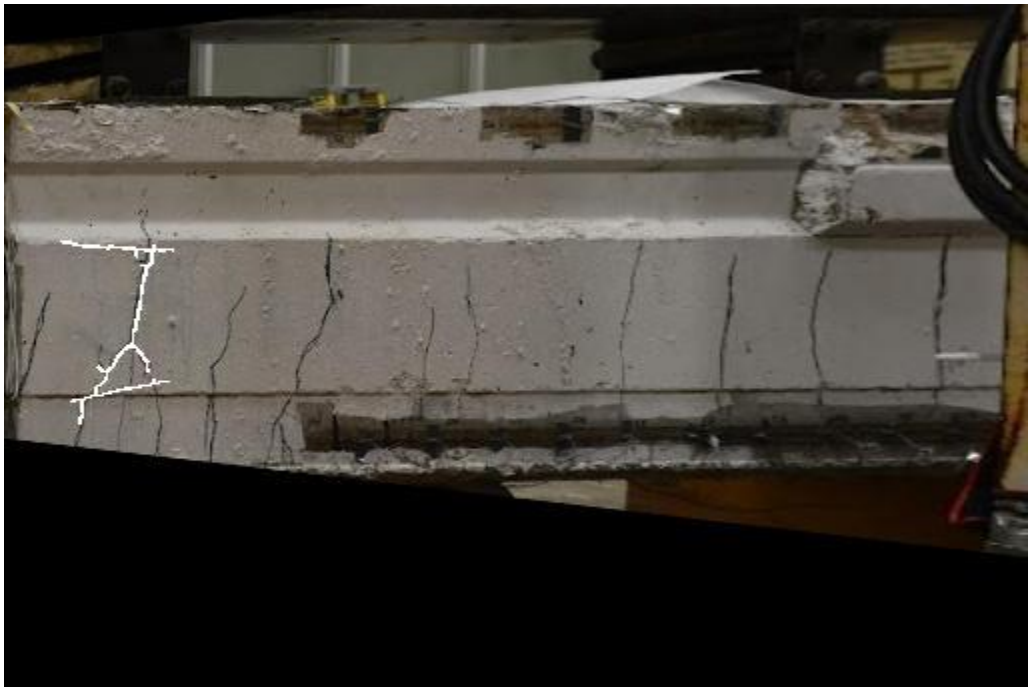
Found a suitable crack.

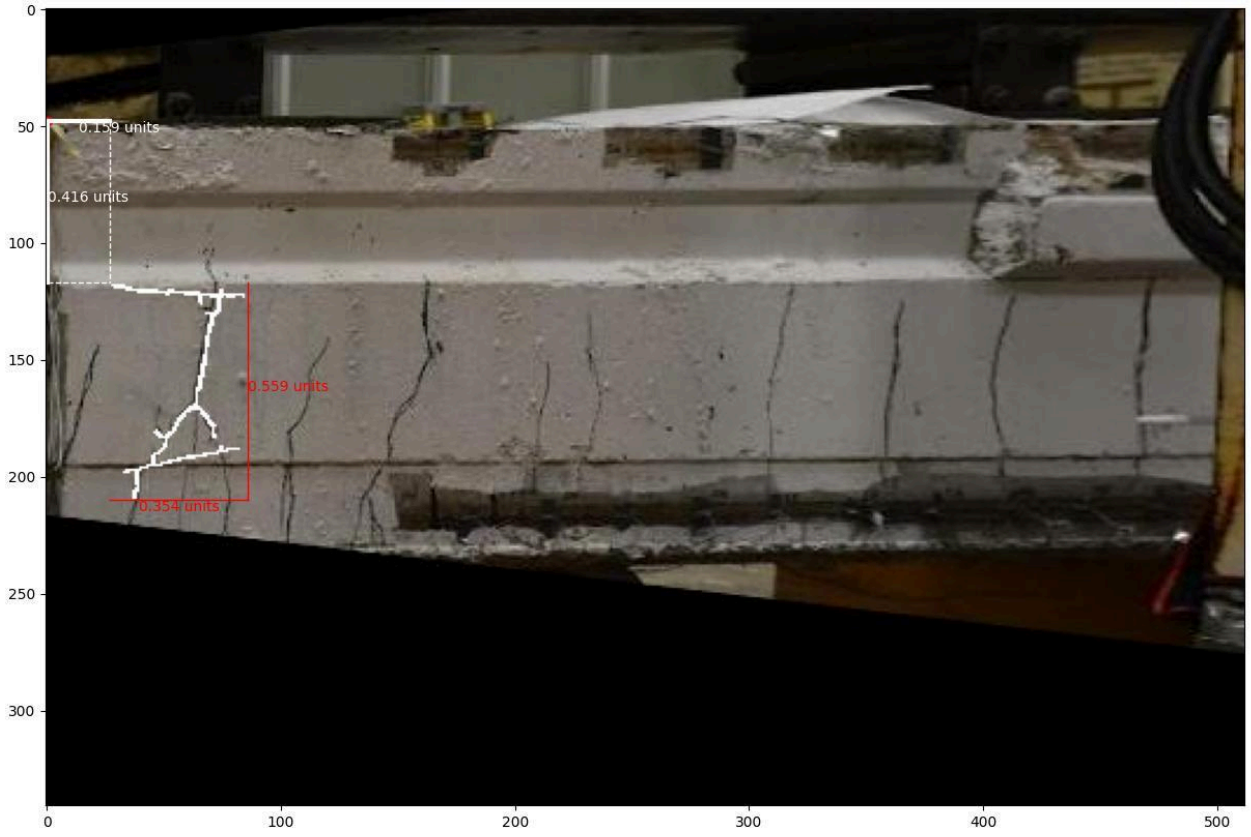
Parameters used:

block_size=25, C=5, min_size=50, morph_kernel=(3, 3), area_thresh=0, length_thresh=50

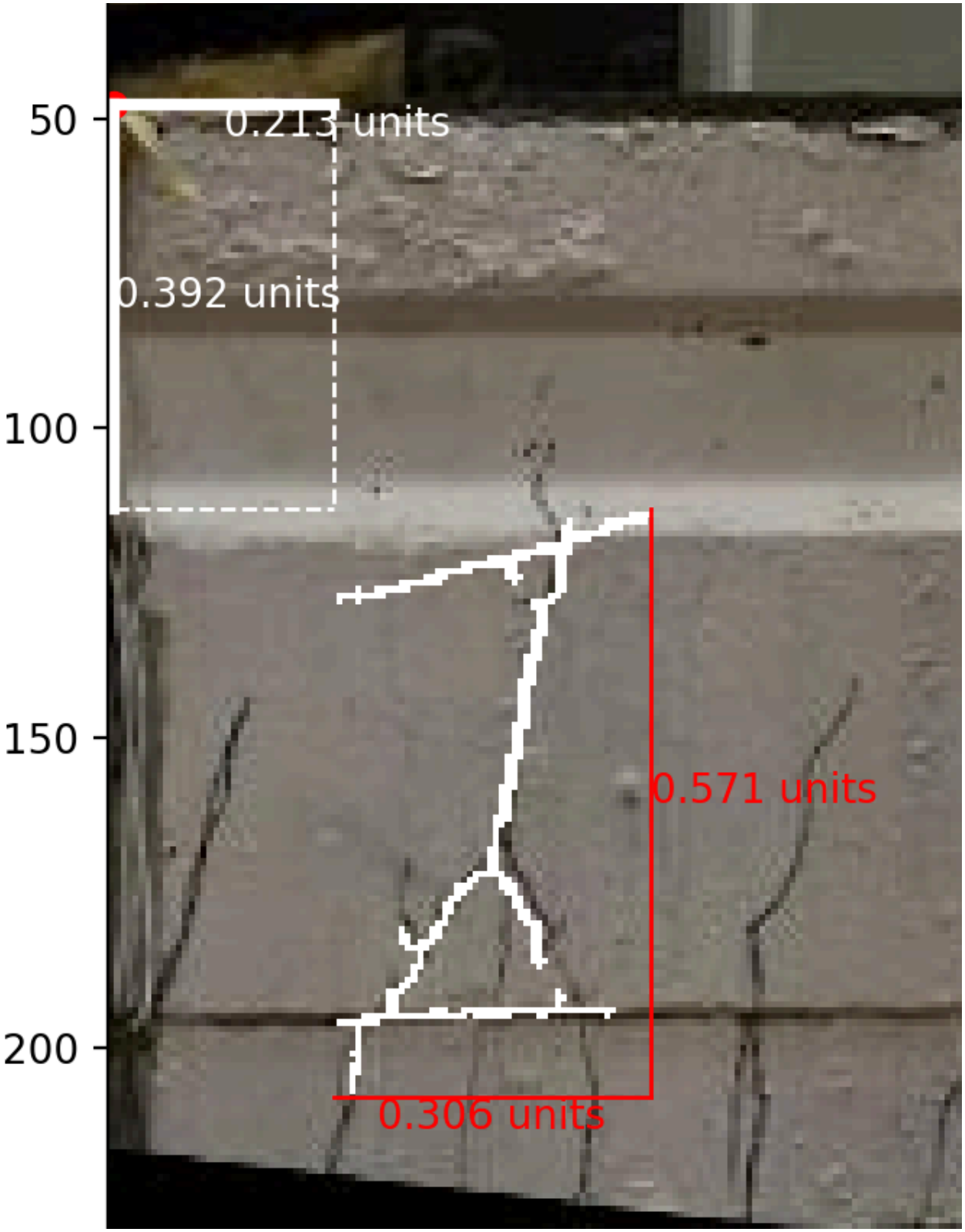
Crack position (centroid in small image): (39, 118), Length: 710.37 pixels

dims: 0.5322580645161281 47.78064516129035 166.5081285512122




0.159 units
0.416 units
0.559 units
0.354 units
0
50
100
150
200
250
300
0
100
200
300
400
500

By zooming in we can get a better look at the measurements:



Here we can see that based on the measurements I gave (in which units will be meters because the numbers I gave were in meters) The crack image's top left corner is located .2 meters along

the length and .4 meters along the height of the beam. Additionally the width of the crack is .3 meters and the height of the crack is .571 meters. As you can see above the code also outputs the length and centroid of the crack in pixels in the original crack image and the dimensions of its location in the output.

One thing worth noting is that the crack tracing in this output is not perfect. This can be a big obstacle. Edges that are not cracks will be mistaken as cracks, such as the grooves in the beam as above.

Let's take a look at some of the more intermediate steps for our next example. Using this image of a crack on concrete:

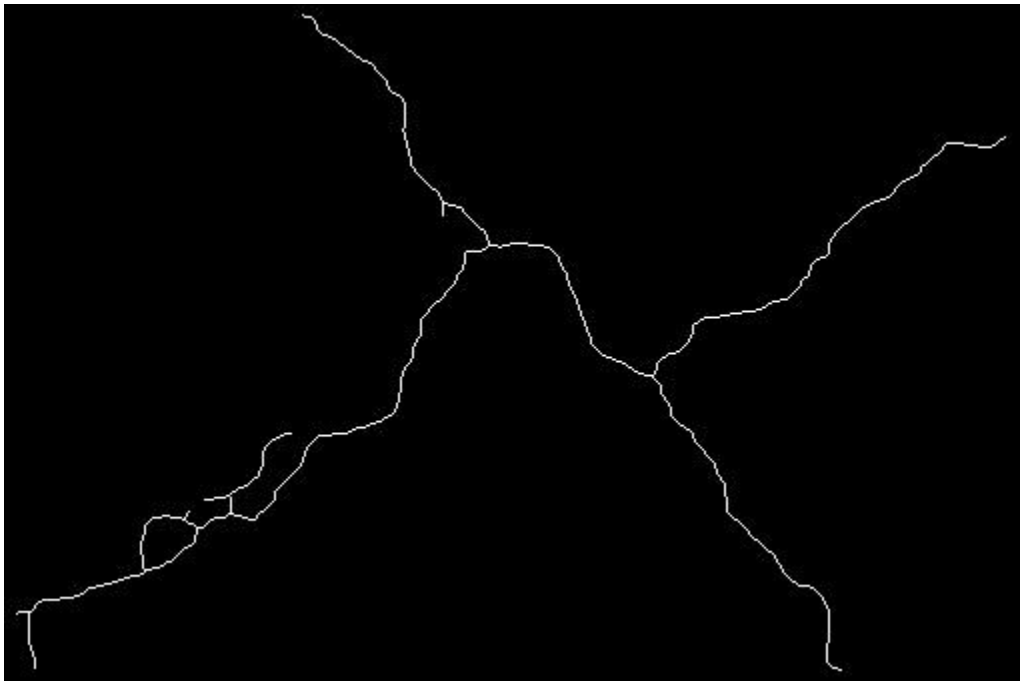
(image source: Cracks In My Newly Poured Concrete Are OK!? - Alta Home Garages & More)

The crack tracing code would take this and produce something like this:

Parameters used:

block_size=25, C=9, min_size=50, morph_kernel=(3, 3), area_thresh=0, length_thresh=50

Crack position (centroid): (80, 266), Length: 2403.96 pixels



Now let’s take off a slice we may want to be analyzing closer or more individually:

The crack detection code produces an output like this:

Parameters used:

block_size=15, C=7, min_size=50, morph_kernel=(3, 3), area_thresh=0, length_thresh=50

Crack position (centroid): (81, 87), Length: 332.02 pixels

Now we can use the full image as the first image and the small portion as the second image to place and measure the crack with respect to the larger image.

This time I just put the corners at the corners of the image and mark the length as 500 cm and the height as 330 cm.

The code then does homography, producing the following keypoint images:

img1 shape: (338, 508, 3)

Click on the top-left coordinate

Click on the bottom-left coordinate

Click on the top-right coordinate

Click on the bottom-right coordinate

Enter dist of top-left to top-right: 500

Enter dist of top-left to bottom-left: 330

img2 shape: (116, 150, 3)

Best Inliers Count: 48 Total Distance: 83.51872348352737

Homography:

[[3.03403191e-03 -3.66294113e-05 -9.85197234e-01]

[8.84725806e-05 3.09764696e-03 -1.71351433e-01]

[1.02414576e-06 7.23733955e-07 2.50715801e-03]]

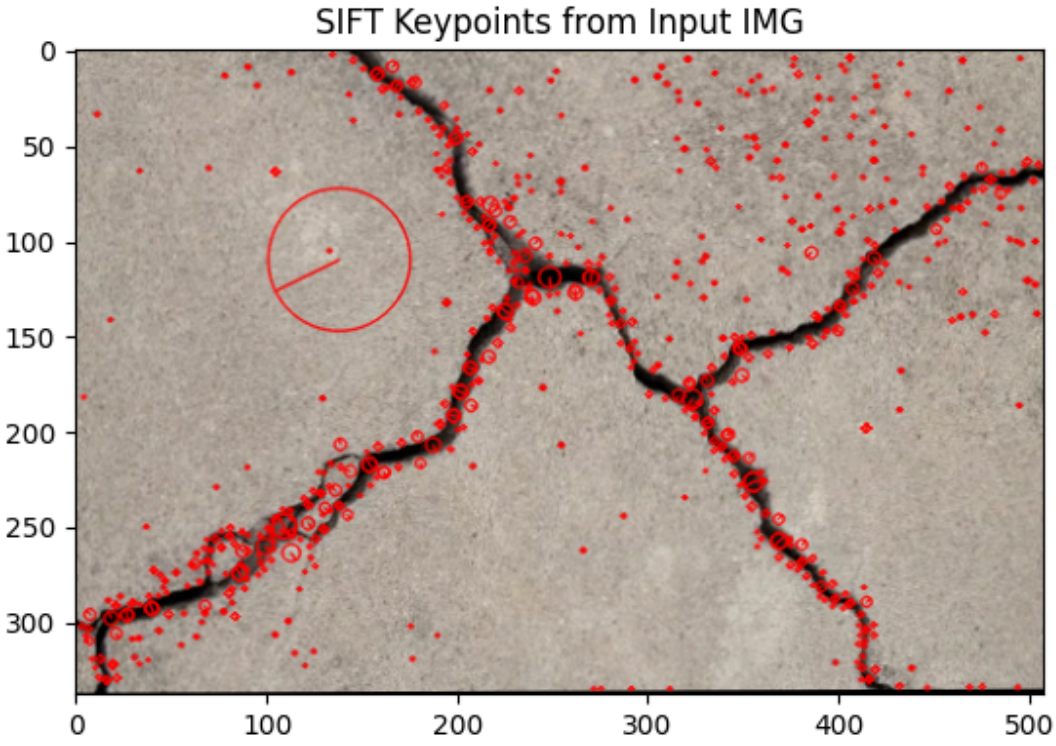
Found a suitable crack.

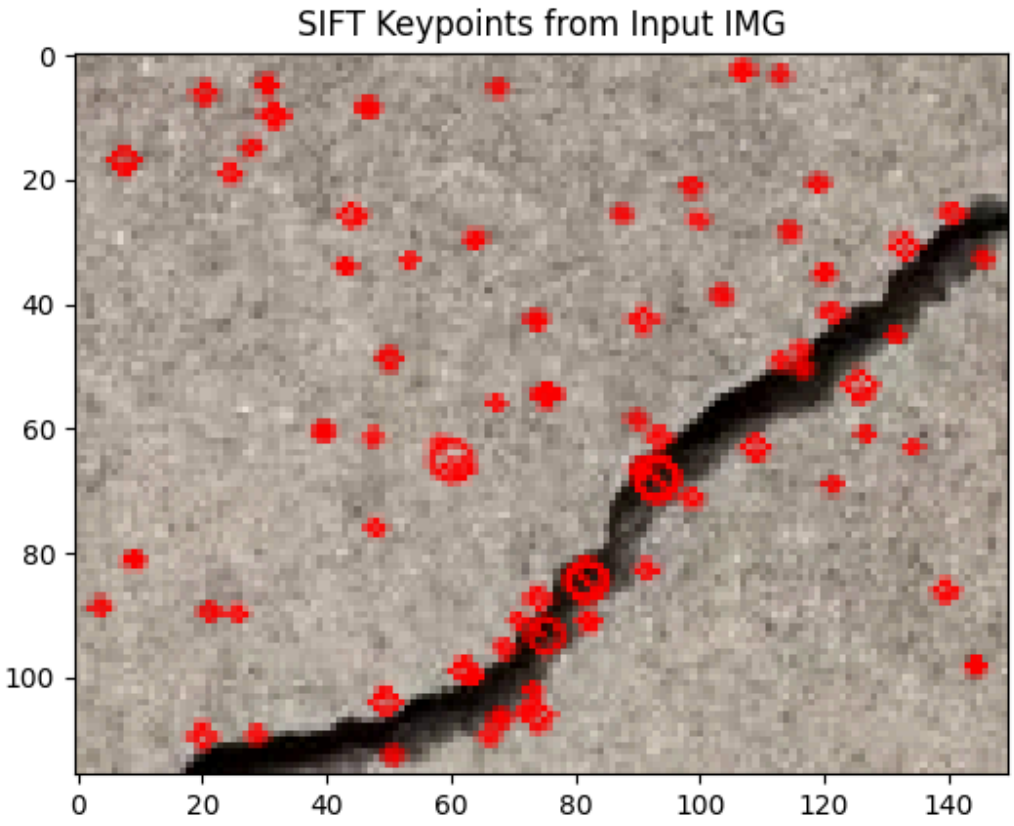
Parameters used:

block_size=15, C=7, min_size=50, morph_kernel=(3, 3), area_thresh=0, length_thresh=50

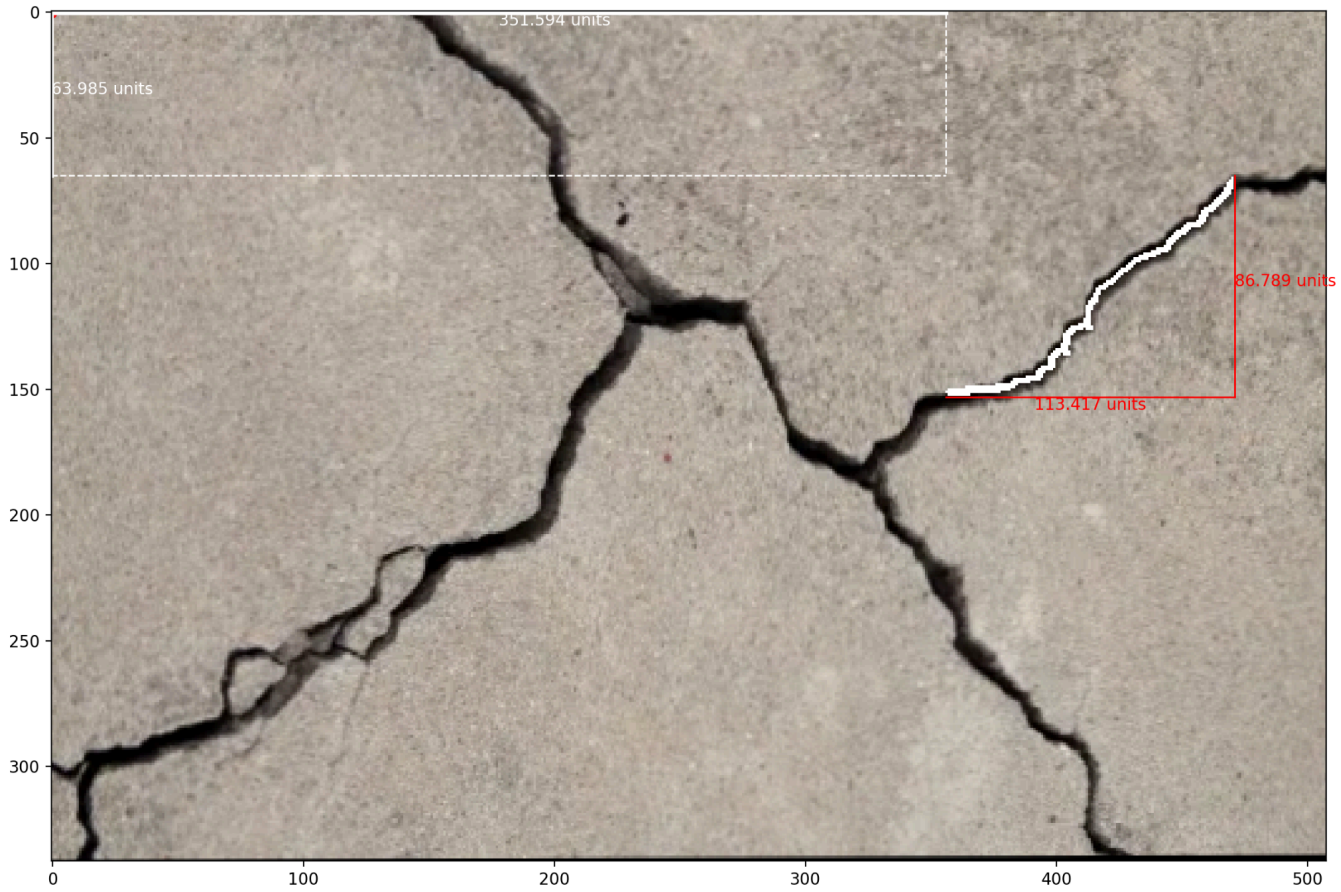
Crack position (centroid in small image): (81, 87), Length: 332.02 pixels

dims: -0.5 0.12258064516134937 1.0139536819789903


SIFT Keypoints from Input IMG
0
50
100
150
200
250
300
0
100
200
300
400
500



Finally, the outputs are produced:



Here's a demo of the code with these images:

ECE549_finalproject_demo.mkv

Here we can see in clear images without other distinct edges the code can effectively and accurately identify, trace, and measure a given crack.

4. **Discussion and conclusions:** Summarize the main insights drawn from your analysis and experiments. You can get a good project grade with mostly negative results, as long as you show evidence of extensive exploration, thoughtfully analyze the causes of your negative results, and discuss potential solutions.

Throughout testing we have determined that in the current condition, the inputs need to be carefully constructed for the program to execute perfectly, even though the use cases for this kind of program is still quite versatile, the application as of right now is tailored toward images and testing with conditions such as beam test or other places where perfects are expected to be produced. Even though this was the initial goal, upon development the potential for this program to be used in less perfect environments became more apparent.

Throughout testing a couple of flaws in the execution of the program came to light. The more variety of inputs showed us the limitations in homography mapping, crack edge detection and input design. These limitations ultimately added restrictions to the scope of acceptable inputs for our program.

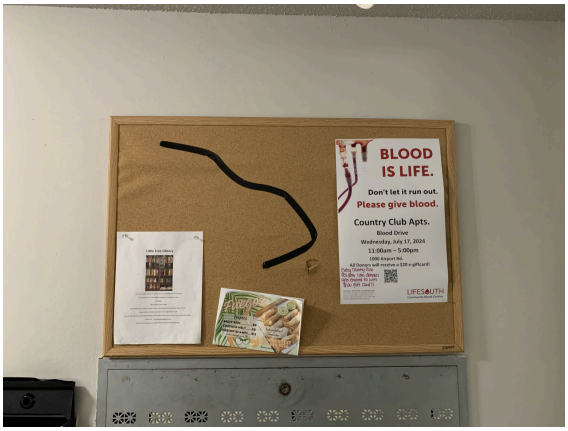
One of the issues our program faced was in the crack edge detection. Even though the program was reliable in the perfect condition, problems arose when the size of the crack did not meet the exact range expected, or when the other non-crack edges were a prominent component of the input. When the size of the cracks was too small, the program failed to read them as a prominent crack. The size cutoff upon visual inspection would not be reasonable to ignore for any user. Not registering small cracks eliminates many applications for pictures taken under reasonable conditions. When the size of the crack was too big, the program would not reliably classify correctly. The program could likely determine chunks of the crack to be or would register the large crack as just the background. This limitation affects the reliability of the program as taking a close-up can result in the crack being too big and misread, or cracks that are too large and are in positions that cannot be accepted as inputs. 3 possible solutions to this pain point in our program could be the uses of AI, the expansion of tested parameters, and or allowing user selection. Our crack edge detection code uses numerous sets of parameters to find the crack in the image; the parameter set that determines the best is then selected in the code and its image is returned dynamically. If we were to expand the range of parameters, we would use and optimize how the sweep is performed so we can broaden the range of working inputs. Throughout the development of the program this method has been used to improve the detection to bring the program this far; Expanding on the number of parameters can further improve this function. AI can also be used to determine the crack. Even though this program did not use AI as a solution, the capabilities of image classification in AI can be used to determine the crack very well. By either using models such as Vocabulary trees or using learning algorithms to determine which parameter set performed best, machine learning could be incorporated into a solution. Finally, adding a feature where the user selects the crack out of the image through means of controlling parameters sets or feeding the pixel locations would likely increase the reliability of the program at the cost of the convenience of the user.

Another issue our program faced was the design of our input. There are 2 forms of inputs, in the file locations, and the click/write of the points distance. Even though this issue is minute compared to the others, using the program becomes a hassle due to the amount of hurdles the user experiences. If we optimized the way we receive user information, the restrictions of the program would feel as prominent. One example is the inputting of the origin points and dimensions. If we had functionality that reduced the number of times the user needs to perform the task, the experience would improve greatly. Such functionality that allows the user to save info with the use of change the input to _scr has been implemented, however, it was not added in the final working code. Another example would be size adjustments. Our program has an input size restriction that the user must follow; if we implemented resizing the inputs in the code, the user would not need to go through the process of resizing the inputs.

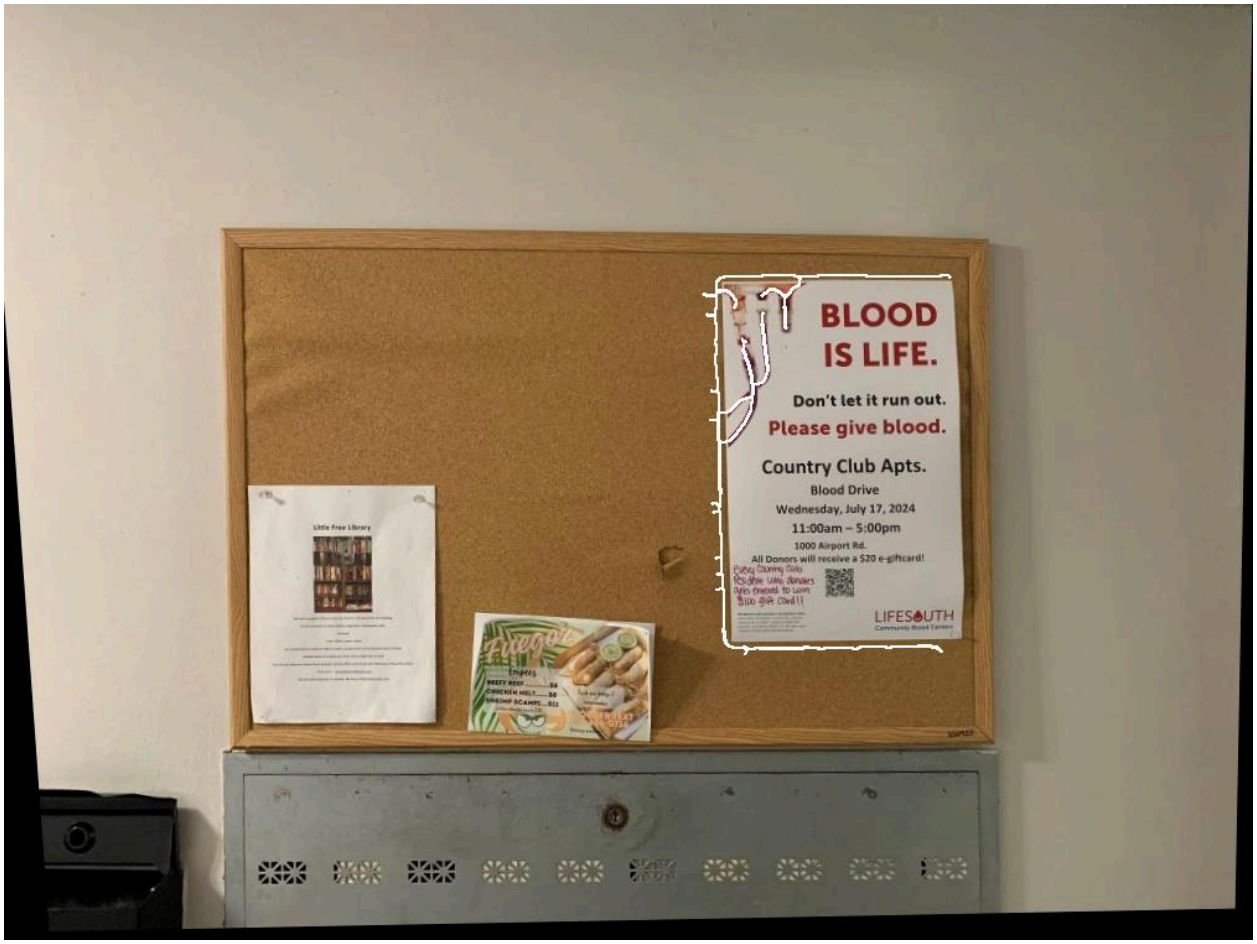
The greatest issue of our program was the homography mapping. The program has heavy constraints when it comes to input size and input properties. Firstly, the program uses SIFT

keypoint detection and Ransac. For acceptable inputs the method performs reliably; however, for size issues such as the input being too large, or the difference in range and views for the image being to great, the mapping would take too long to compute and not find the correct mapping simply because there are too many keypoints. A solution to the issue could be to test multiple parameters sets in code and output the healthiest looking set at the cost of runtime. Another solution could be to limit the number of pairs accepted. The adjustments were implemented but did not mitigate the problem enough. Lastly, the image size could be adjusted in code to ensure the size would not overwhelm the code. Secondly, the mapping did not execute as expected due to many of the nuance properties of the images. The main reason behind this was the orthogonalization of the source image; this process made SIFT and RANSAC much harder. The primary use for orthogonalization is to increase accuracy in measuring the crack. This issue was fix but the improvement was too late to added to the final code. By saving the orthogonal homography matrix in the info txt file and warping the output image with it, the mapping became more reliable. Finally, by incorporating other alignment methods such as sum of square difference or normalized cross correlation, the program can potentially become more reliable with inputs that have the source and crack images be taken from similar angles which is not a hard restriction on the user. If the issue with homography mapping can be fixed, the use case of this program would expand greatly beyond the set goals.

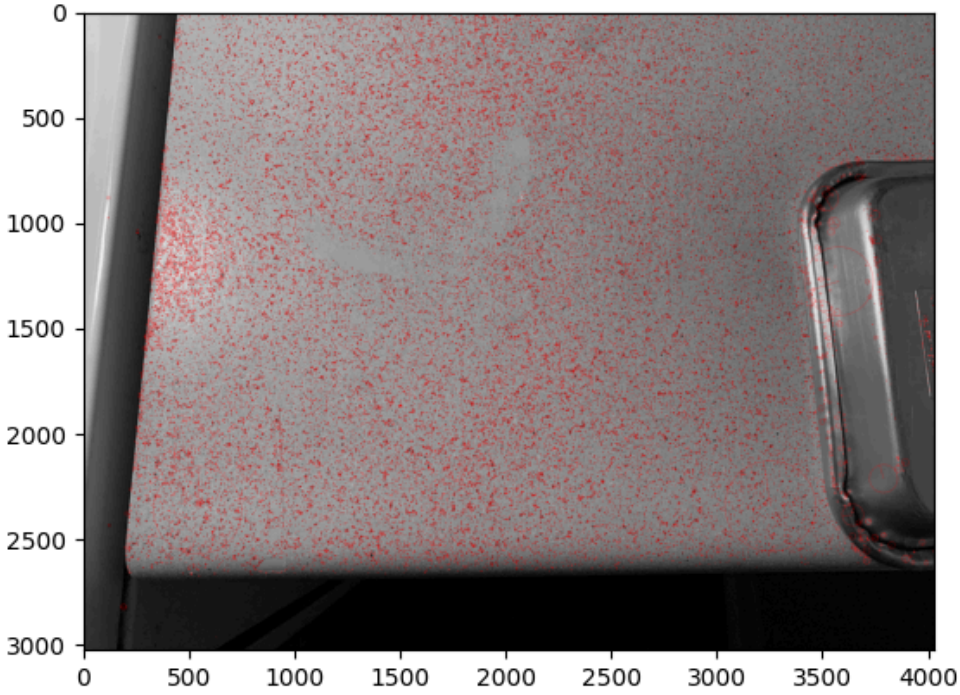
Here are some examples of difficulties faced:

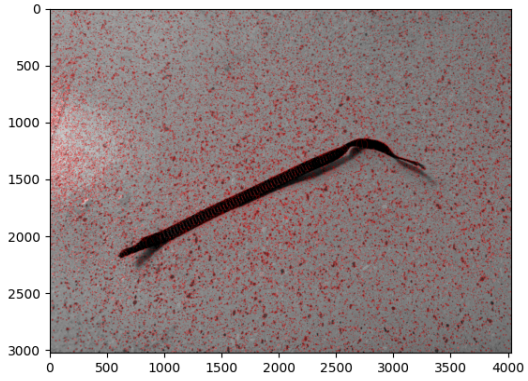


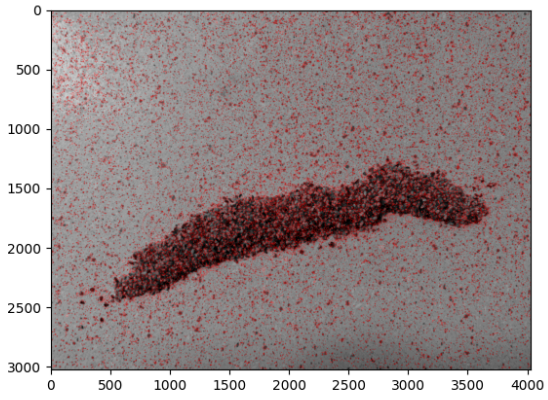
Here's an example output we tried that suffered from difficulties in edge detection. The numerous shared features mean even though the original image does not have the black tape (acting as our crack here) the homography can easily find matches and correctly place the location of things from the first to the second. However, this has drawbacks. The output we ended up getting was:



Instead of properly tracing the crack, it traced the edge of one of the posters. This can put good candidates for keypoint matches at odds with the images that we actually would like to match and trace cracks onto, for example if we wanted to show crack growth over time or something like that. The original image won't always have the crack on it as it is in our trace, but images that are likely to find the correct crack position for are also likely to mistake other distinct features as cracks. In cases unlike this poster image where there aren't distinct features to match with, instead of finding a false crack it usually simply fails to find or place any crack all.



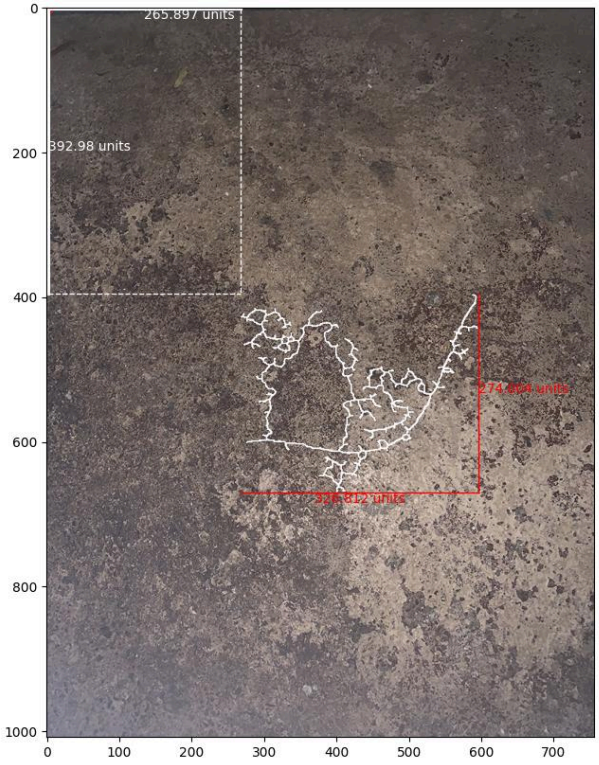




Here we test the program on a table counter with both a pepper spill as well as a ribbon and the “cracks”. The program found so many keypoints in the images that the mapping process was overwhelmed with false inliers. The pictures take were both too large and too keypoint-filled for the program to handle even though the picture were only 4032x3024 (taken with Iphone XR)

However, there are somewhat successful cases as well. For these inputs:

For these the crack again is not present in the first image. However, homography and tracing is significantly more successful. The outputs look like:



The crack detection code picks up a lot of non-crack noise in the discoloration of the ground, but the intended crack is also traced and successfully placed in the image. So, there are a lot of limitations, but it is not impossible to place cracks in images without the cracks, which opens up a lot of applications if the code could be refined to succeed on this more.

Overall, the functionality of the program is satisfactory for environments with the perfect conditions set in the baseline goal. Even though the program does struggle to achieve the higher goal of non-perfect conditions, there is evidence that the application can expand beyond just concrete images but faults in a variety of textures if it meets the conditions of being distinct enough to perform mapping well, and the user knowing the dimensions of the source view. After seeing the expansive applications of this program, we began to test it on texture and environments outside of concrete: floors, walls, tables, as well as spills, dents, and tape. A new higher goal based on this new range of application would be to detect and measure any keypoint on any flat plane with textures that require moderate to few points of distinction.

5. **Connection to course material:** How does the topic of your project relate to the topics, techniques, or themes covered in this semester's syllabus? Be specific -- i.e., list specific lectures, MPs, etc. and relate them to specific parts of your project. If the connection is obvious, this section can be quite short.

As it was previously discussed, our project is deeply connected to MPs number two and three where SIFT detection and Homographic Ransac Mapping were part of the assignment to achieve the necessary outcomes. We also used D. Lowe's paper ijcv04.pdf (ubc.ca) (Distinctive Image Features from Scale-Invariant Keypoints) for guidelines and inspiration. Lectures 05, 07, and 10 were also frequently referred to for insight on Edge Detection, SIFT, and Homography.

6. **Statement of individual contribution:** Required if there is more than one group member.

Tasho worked on the SIFT detection and homography

Teague and Máximo worked on the edge detection

Everyone collaborated on combining the two into one workflow as well as testing and troubleshooting and working on the report.

7. **References:** including URLs for any external code or data used.

https://www.sciencedirect.com/science/article/pii/S1110016817300236

D. Lowe. Distinctive image features from scale-invariant keypoints.