Final Project
Principal Component Analysis for Crack Patterns

Nicholas S. Fong & Minh Nguyen
AMATH 582

March 22, 2019

Abstract

Failure is an inevitable aspect of all materials as they age and wear. It is well known that failure patterns, or cracks, appear as a result, but this phenomenon is not well understood. In this report, we analyze crack patterns of composite materials using edge detection and principal components analysis. Our results strongly support the idea that good edge detection software is crucial for accurate analysis of failure patterns due to their unpredictable nature.

I. Introduction and Overview

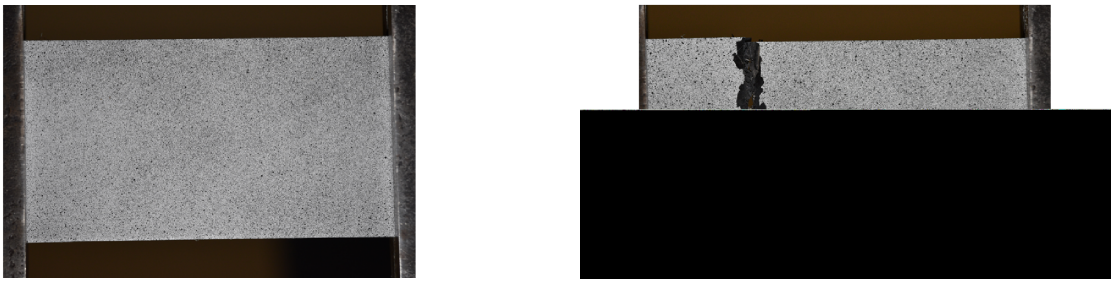
Digital Image Correlation (DIC) is a common technique used to measure the strain, or the displacement, of a specimen during a load test. At first, random black and white speckle painting is applied onto the surface of the specimen. By constantly taking images of the specimen during the test and calculating the movement of the speckles, displacement data would be obtained. However, this is not the main focus of this project; our goal is to study the crack patterns themselves, which appear in the final frames of a DIC video.

For carbon fiber composites, after failing during testing, black surfaces of the cracks would appear. Currently there are not many in-depth studies about crack sizes and morphology. Therefore, there is a need for algorithms to perform crack identification and characterization. In this report, we develop an algorithm to characterize a sample set of cracks using edge detection in MATLAB, preparing our data for Principal Components Analysis (PCA). Given the composite nature of the material and randomness of the crack patterns, we do not expect PCA to yield significant features. Regardless, we will perform this initial analysis to see if any characterizations do appear and if there are potential improvements that can be made to our algorithm in the future.

II. Theoretical Background

DIC composite specimens started out with random distributions of black and white speckles, and ended with black cracks on the surface (Fig. 1). The foundation of crack identification would certainly be edge detection. Human eyes are able to perceive the crack quite easily based on features like color and line consistency. In terms of computational algorithms, it would be gradient shifts detection. By utilizing the existing edge detection techniques, the cracks can be easily identified. However, besides the crack, the surface edges can also be identified. The same applies for the apparatus' edges, even the speckles, as well as the crack texture. To solely outline the crack, other filtering and denoising methods would be necessary. Therefore, the approach would be to test different algorithms, define advantages and disadvantages of each one, and then combining them.

Once the cracks were identified, principal component analysis can be performed to characterize the patterns.



(a) Before failing

(b) After failing

Figure 1: DIC specimen under load test

III. Algorithm Implementation and Development

Stage 1: Varying data sets

The main principle of this stage was that all methods have pros and cons, and they can detect and produce different results. However, special features would certainly stand out and remain consistent between different techniques. By running the data through the algorithm with small adjustments each time, the averaged result would present the most prominent features. On DIC specimens, those would certainly not be speckles but edges. MATLAB has many built-in toolboxes, and **edge** was primarily used for edge detecting in this method. The adjustments involved were: color scales, image sizes and applied filters.

Edge detection ability in human eyes is mainly due to color perception. A color image would always be easier to see the details compare to a grayscale one. Using this notion, the image was split into red, blue and green color scales, as well as grayscale. These four sets of data were then passed through edge detection. Since each color scale had different edge clarity on different regions, combining those results would bring out the common traits.

The second adjustment was the dimension of the image. Edge detection results actually varied according to the size of the image. In smaller images, the speckles were less likely to appear. However, the crack also often did not appear as fully and sharply as the bigger version. In other words, small images could capture the overall shape of the crack, while the larger provided more details. **imresize** was used to convert images to desired dimensions.

The last adjustment was the filter. Before running through **edge**, a filter was applied on the image. The purpose of filtering was to reduce the amount of noise within. As speckles were randomly distributed, filters would have an effect on it and therefore reduce their appearance in edge detection. Five filters were used in the process: Laplacian, Gaussian, disk, motion and average as they were the best fit. Figure 2 show the effect of the first three filters, as well as the result of combining all five (Fig. 3).

(a) Laplacian filter (b) Gaussian filter (c) Disk filter

Figure 2: Filtered edge results

Overall, adjustments in color scales, image sizes and filters were made before performing edge detection. There were four color scales, two image sizes, and five filters used. Hence, there were a total of forty permutations possible for combining those options. Figure 4 showed the result of combining all forty possible options altogether.

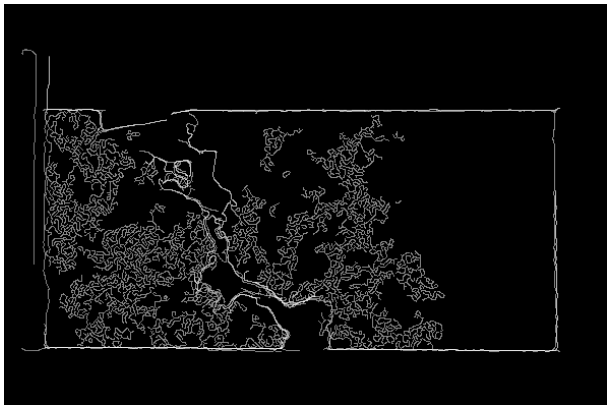


Figure 3: Combined filters

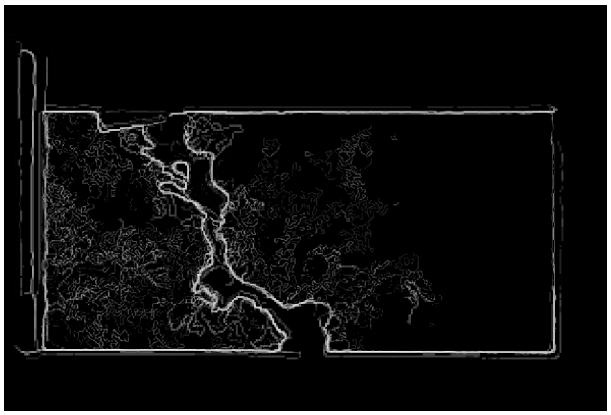


Figure 4: Mean result of 40 permutations

Stage 2: Edge Erosion

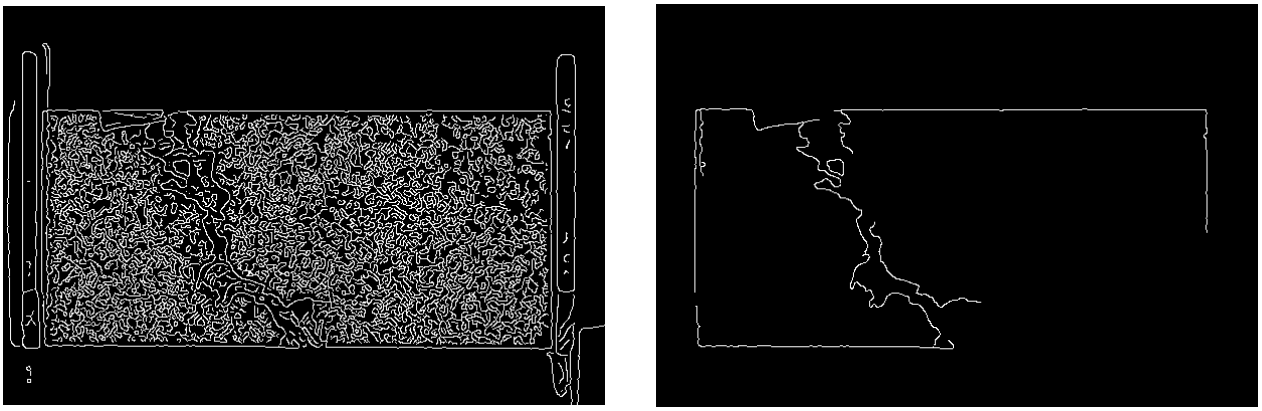
In the previous stage, the resulted edge image still contained a lot noise and undesired edges from the specimen. Therefore, in stage 2, the goal was to separate the composite specimen's outer edge from the outline of the crack by eroding the specimen's boundary while retaining the failure pattern. The next step involved centering all of the crack patterns in order to perform PCA on the data.

In the main script CRACK_PATTERNS.M, we begin by setting a threshold under which a given pixel value will be sent to 0. This helps prevent the background from corrupting the data analysis, since we only wish to focus on the failure pattern. We then loop through all of the original images, first producing the outline of the crack, and then re-centering the crack in a 400x600 image for proper PCA. To prepare for image erosion, we convert each image to double-precision grayscale, destroy pixels below the given threshold and apply one of five filters (we will take the average of them). This converted image is then sent to the function ISOLATECRACK.M, where it is filtered.

Our next step consists of edge detection on the converted image, using two different threshold values and expanding the lines in the image using **imdilate**. In the code, "grain" refers to a low threshold for edge detection, so it retains a lot of the internal patterns on the slab (Fig. 5a). That way, when we perform an expansion, we fill in the remaining holes and reconstruct the outer edge of the slab. "Macro" refers to a high threshold for edge detection, so it is meant to show only the crack and the perimeter of the slab (Fig. 5b). We expanded the outline until the crack pattern was thick and the slab edge as well as any texture noise consisted of thin lines, which is why we have separate morphological structuring elements (**strel**) for macro and grain. In our code, **strel** simply expands pixel lines vertically and horizontally by a specified number. Then, we gradually destroy the thin lines using **imerode** and **bwareaopen**, which removes connected pixel clusters at or below a specified population.

On a final note, we mention that the Canny algorithm for edge detection was used because it was the best one for simultaneously circumventing noise while also picking up on weaker gradients in the image. This is important, because although there is plenty of texture noise

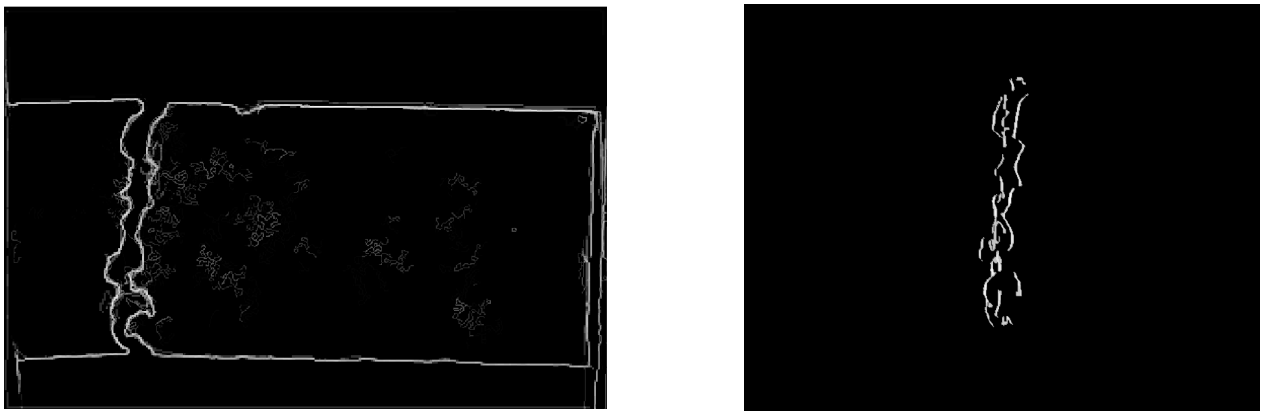
in the data, there are also varying gradient strengths that we wish to pick up for analyzing crack patterns because the cracks themselves vary in intensity compared to the background. See the MATLAB documentation of **edge** for further details.



(a) "Grain" with low threshold detection (b) "Macro" with high threshold detection

Figure 5: Edge detection thresholds

In the final step, we send a normalized image averaged over all filter types to SHIFTCRACK.M, a function that locates the centroid of the failure pattern and uses it for centering. In this function, we convert the image to binary format in order to calculate the centroid coordinates. Once we have these coordinates as integers, we use them to shift the original grayscale crack pattern so that the centroid lies in the middle of a 400x600 pixel image. The result is in Fig. 6.

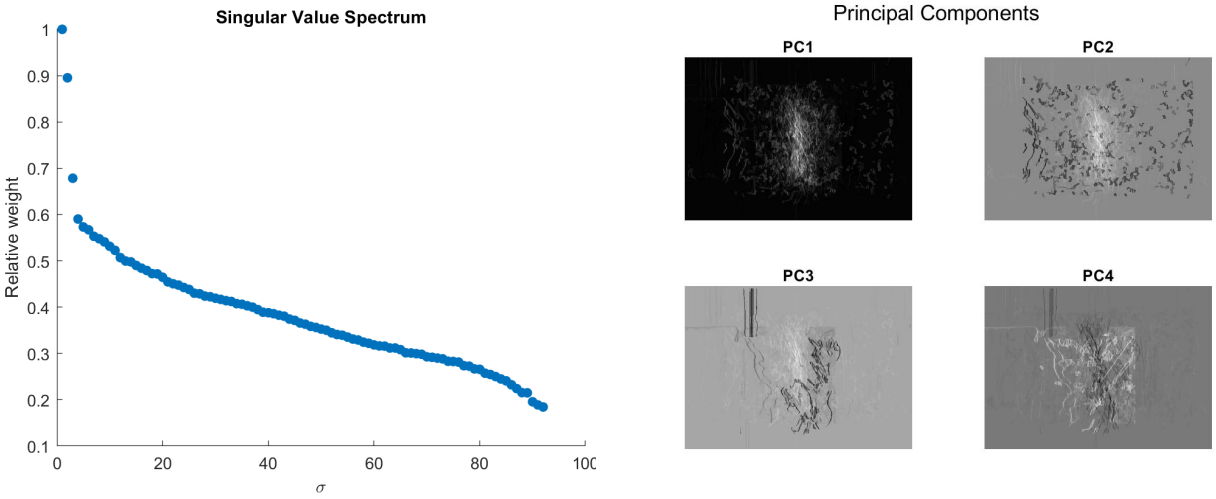


(a) Before eroding and centering (b) After eroding and centering

Figure 6: Edge erosion technique to remove unwanted lines

Once the centering is complete for all of our images, we mean-center the vector version of each image and perform PCA using the singular value decomposition. We then plot the singular value spectrum as well as the principal components of the data.

IV. Computational Results



(a) Singular value spectrum (b) First 4 principal components of the data

Figure 7: PCA results

We expected the crack patterns in the composite material to be fairly random, and our results of PCA support this hypothesis. In figure 7a, our singular value spectrum of the data suggests that there are three dominant modes that characterize our set of failure patterns, although the spectrum's fall-off is very gradual and therefore heavily influenced by noise. Figure 7b shows the four strongest principal components of the data. PC1 shows a general crack outline against a black field, confirming the obvious result that most variation in the data lies on the slab's failure region. PC2 shows speckle patterns, which are also a common aspect of each composite material. PC3 and PC4, however, do not show any obvious or useful characterization of the data. Because the crack patterns varied so widely, it would make sense that PCA may not be the optimal method of finding common features among crack patterns.

The most important parts of this report to discuss, however, are the strengths and drawbacks of the algorithm that we have implemented. Although edge detection is a built-in MATLAB function, it returns more edges in the image than we are actually interested in. The most difficult part was separating the edges of the slab from the edges of the failure pattern. In order to do this, we had to create a separate outline of the slab as a whole and essentially remove this from an edge-image with both the crack and the slab perimeter. However, it is impossible to accurately reconstruct the slab perimeter, so subtracting it directly in order to isolate the crack will not do (instead, it merely introduces new patterns into the image). The approach we used was to retain the crack pattern as a thick line in the edge image, while retaining the perimeter of the slab as a thin line. Eroding the edge patterns effectively removes the perimeter and simply sharpens the failure pattern.

In addition, there is also a trade-off between choosing a high and low threshold gradient

over which to perform edge detection. If the gradient threshold is set too low, texture details that are not relevant to the analysis will remain. This is useful for filling in the slab and extracting the perimeter, but it also introduces unwanted details in the image that may corrupt the failure pattern. If the threshold is too high, there may not be a sufficient amount of edge details to retain the crack pattern at all, although texture details will definitely be eliminated.

V. Summary and Conclusions

Analyzing failure patterns of composite materials is a complex process, and there are many ways to do so. In this report, we have used edge detection to produce good-quality crack outlines for use in PCA. As one would expect, the patterns are fairly random, and PCA does not give much information regarding common characteristics that these patterns share. However, different types of materials have different failure traits, so choosing a single material, collecting more data, and improving the edge detection process may lead to more informative results.

Appendix A: MATLAB Functions

Below is a listing of some key MATLAB functions used in the project. See the MATLAB documentation for further details.

- `adapthisteq(image,Name,Value)`: Adjust contrast of the image.
- `bwareaopen(image,threshold)`: Removes clusters of pixels at or below a specified threshold population.
- `bwperim(image)`: Returns the perimeter of a binary image.
- `edge(image,algorithm,[threshold1 threshold2])`: Applies a specified edge-detection algorithm to an image, with a given pixel gradient range over which to apply it. Returns a binary image locating the edges of the image.
- `fspecial(name)`: Call a predefined special built-in function.
- `imbinarize(image)`: Returns a binary version of a grayscale image. In other words, all pixel intensities are reduced to 1's and 0's.
- `imdilate(image,structure_object)`: Expands the true values in a binary image according to a specified morphological structure.
- `imerode(image,structure_object)`: Erodes the true values in a binary image according to a specified morphological structure.
- `imfill(image,'holes')`: Fills in clusters of zero pixel intensity in an image.
- `imfilter(image,function)`: Filter image using a function.
- `imresize(image,[size])`: Resize image.
- `mean(data,dim)`: Calculate the average of the data in certain dimension.
- `rgb2gray(image)`: Convert RGB image to grayscale.
- `svd(A)`: Returns the singular value decomposition $U\Sigma V^*$ of a matrix A.

Appendix B: MATLAB Codes

crack_patterns.m

```
clear all; close all; clc
load raw400x600.mat

%% Isolate crack patterns
numImages=size(dataraw,4);
pix_thresh=80;      % threshold below which pixel values -> 0
isolatedCracks=zeros(numImages,size(dataraw,1)*size(dataraw,2));

list = {'average' 'disk' 'gaussian' 'laplacian' 'motion'};
dim = [400 600];
for j=1:numImages
    for k=1:length(list)
        currentImage=uint8(dataraw(:,:,:,j));
        currentImage=double(rgb2gray(currentImage));
        currentImage=currentImage.*(currentImage>pix_thresh);
        crack(:,:,k)=isolateCrack(currentImage,list(k));
    end
    avg(:,:,j) = mean(crack,3);
    avg(:,:,j) = avg(:,:,j)/max(max(avg(:,:,j)));
    avgShifted=shiftCrack(avg(:,:,j));
    isolatedCracks(j,:)=reshape(avgShifted,1,[]);
end


%% Examine all images
for j=1:numImages
    crack=isolatedCracks(j,:);
    crack=reshape(crack,[size(dataraw,1),size(dataraw,2)]);
    original=uint8(dataraw(:,:,:,j));
    original=double(rgb2gray(original));
     imshow(imfuse(original,crack))  % display original image with crack outline
%     pcolor(crack), shading interp, colormap(gray)
%     axis off, title("Sample Failure Pattern to be used in PCA")
     pause(1)
end

%% PCA
for j=1:numImages
    currentMean=mean(isolatedCracks(j,:));
```

```
    isolatedCracks(j,:)=isolatedCracks(j,:)-currentMean;
end
[U,S,V]=svd(isolatedCracks,'econ');

%% Singular value spectrum
figure
singvals=diag(S)/max(diag(S));
scatter(1:length(singvals),singvals,'Filled')
title("Singular Value Spectrum"), xlabel("\sigma"), ylabel("Relative weight")

%% principal components
figure(), sgtitle("Principal Components")
subplot(2,2,1)
pcolor(reshape(V(:,1),size(dataraw,1),size(dataraw,2)))
shading interp, colormap(gray), axis off, title("PC1")
subplot(2,2,2)
pcolor(reshape(V(:,2),size(dataraw,1),size(dataraw,2)))
shading interp, colormap(gray), axis off, title("PC2")
subplot(2,2,3)
pcolor(reshape(V(:,3),size(dataraw,1),size(dataraw,2)))
shading interp, colormap(gray), axis off, title("PC3")
subplot(2,2,4)
pcolor(reshape(V(:,4),size(dataraw,1),size(dataraw,2)))
shading interp, colormap(gray), axis off, title("PC4")
```

isolateCrack.m

```
function crack = isolateCrack(originalImage,type)
% Erode edges of image in order to separate the crack pattern from the slab perimeter.
    % Convert a single image to grayscale doubles
    originalImage = adapthisteq(originalImage,'clipLimit',0.01,'Distribution',...
       'rayleigh');
    originalImage = imfilter(originalImage,fspecial(char(type)));

    % Perform edge detection on slab with two different gradient thresholds
    thresholds=[0.001 0.7];
    edgeGrain=edge(originalImage,'Canny',[0 thresholds(1)]);
    edgeMacro=edge(originalImage,'Canny',[0 thresholds(2)]);

    % Construct strel structres for grainy and macro images
    se90g=strel('line',5,90); se0g=strel('line',5,0);
    se90m=strel('line',2,90); se0m=strel('line',2,0);
    grainDilate=imdilate(edgeGrain,[se90g se0g]);
    macroDilate=imdilate(edgeMacro,[se90m se0m]);

    % Fill in the expanded image, and produce a perimeter
    filled=imfill(grainDilate,'holes');
    slabPerimeter=bwperim(filled);

    % Subtract most of the outer boundary, leaving a thin line
    macroDilateThin=macroDilate-imdilate(slabPerimeter,[se90m se0m]);
    macroDilateThin=macroDilateThin>0;

    % Erode crack and eliminate remaining thin boundary lines
    isolatedCrackandSpecs=imerode(macroDilateThin,[se90m se90m]);

    % Eliminate remaining specs
    pixel_limit=25; % max number of pixels in a connected region to remove
    crack=bwareaopen(isolatedCrackandSpecs,pixel_limit);

    % Set outer columns = 0
    crack(:,1:75)=0; crack(:,525:end)=0;
end
```

shiftCrack.m

```
function shifted = shiftCrack(crack)
    % shift crack pattern so that its centroid is at the center of matrix
    % also, converts to black and white
    currentCrack=imbinarize(crack);
    [y, x]=ndgrid(1:size(currentCrack, 1), 1:size(currentCrack, 2));
    centroid=mean([x(logical(currentCrack)), y(logical(currentCrack))]);
    centroid=round(centroid);     % convert to int

    % centroid=[col,row].  Shift 1's (600-centroidX) columns and
    % (400-centroidY) rows, provided it does not exceed dimensions
    shifted=zeros(400,600);
    rowShift=200-centroid(2); colShift=300-centroid(1);
    for row=1:400
        for col=1:600
            if currentCrack(row,col)==1 ...
                && row+rowShift<=400 ...
                && col+colShift<=600 ...
                && row+rowShift>0 ...
                && col+colShift>0 ...
                    % shift all grayscale (not binary) values
                    shifted(row+rowShift,col+colShift)=crack(row,col);
            end
        end
    end
end
```