Automated Fracture Detection

Sameed M. Siddiqui

***Abstract*— The development of machine learning and digital signal processing (DSP) techniques to analyze X-rays is vital to the democratization and improvement of the field of medical radiology. In this paper, we discuss an automated fracture detection system which can be used to automatically discuss the presence of fractures in X-ray images. We implement this system using a three step approach, first applying Canny edge detection to simplify the image, then using the Probabilistic Hough Transform to model bones as cylinders with approximately parallel lines, and finally, detecting the intersection of lines and declaring them to be indicators of potential fractures.**

I. Introduction

WE began this project with an intention to apply machine learning algorithms to identify and classify fractures in human X-rays. Due to a lack of publically-accessible X-ray image database, the scope of our system changed from machine learning-centric to DSP-centric.

In the end, the method we used was a simple three-step process: (1) Edge detection; (2) Line detection; and (3) line intersection recognition. The edge detection was implemented using a technique called the Canny edge detection, which is a more sophisticated version of the Sobel filter and incorporates a Gaussian filter, a Sobel filter, and other techniques to ensure that all edges produced have a thickness of one pixel.

Line detection was performed by the probabilistic Hough transform in the openCV library. This transform is a cousin of the standard Hough transform we implemented in class but has more sophisticated parameters. The Hough transform was used based on the observation that many human bones are cylindrical in shape, and as such, can be modeled with pairs of parallel lines. Following this logic, we then reasoned that non-broken sets of bones are generally somewhat parallel in nature, and as such, we declared intersections of the lines as potential indicators of bone fracture.

The results of our program seem to be positive. When we inputted an x-ray with a clear fracture in the ulna and radius, our program was able to detect more line intersections (e.g. indications of potential bone fracture) than when we inputted an x-ray without a fracture in the ulna and radius. However, we believe that with a little more tinkering of a few parameters, the results of our program can be significantly improved. Thus overall the project was a successful proof of concept of the ability to apply DSP techniques to identify fractures in X-rays.

II. Motivation

According to the World Health Organization, two-thirds of the world has no access to radiology services. The reason for this is a lack of imaging modalities, healthcare personnel, and interpretation services in many impoverished areas. Global organizations annually conduct many international short-term health care missions to train healthcare professionals in impoverished localities, but few of these efforts address imaging [1].

Recently, people have started to apply machine learning techniques in radiology, hoping not only to extend radiological interpretation services in areas where they are currently lacking, but also to improve radiological interpretation conducted by trained radiologists [1, 2, 3].

In this project, our goal was to create a type of proof of concept of the idea that computerized techniques can be used to help detect fractures. Namely, we wanted to see how simple a system could be while still being effective.

On a more personal note, another reason I chose this topic was because I already have a lot of signal processing experience with time-varying signals; I am currently working on research, for example, in which I analyze EEG signals and find patterns in the data. For this project, I wanted to learn something new and do something I have never done before. I have always been intimidated by image processing and so decided that this class would be a perfect opportunity for me to dive in and learn.

III. Approach

The first week in this class was spent reading on how other groups have went about detecting bone fractures. As it turns out, the best work in this field is being done by companies such as Enlitic (located in San Francisco), which are not very open about their methodologies [3]. Beyond work done in corporate world, there seem to be only a handful of peer-reviewed methods in literature, amongst them was an article by Samuel Kurniawan *et al*, in which fractures in bones were detected using an image processing suite called OpenCV [4]. While I initially attempted to create a more complicated program than the one presented by Kurniawan *et al*, the process we eventually settled on in creating our program was in many ways inspired by this paper.

The core-most requirement of the program is simple: that it should be able to distinguish between (or at least indicate some measure of likelihood) x-ray images which have broken bones and which do not. However, this goal was balanced between a

June 10, 2016

Sameed M Siddiqui is with the University of California, Los Angeles, Los Angeles, CA 90095 USA (e-mail: sameedmsiddiqu@gmail.com).

series of constraints posed to us by a variety of external factors, such as memory issues, program transfer times, and library quirks. When I first began the project, I was not aware of almost all of these constraints, but as the project progressed and as issues arose, the project naturally took on slightly different directions to meander towards the end goal of x-ray detection.

My first goal was to be able to apply some of the more sophisticated machine learning algorithms found in openCV. However, this turned out to be infeasible to do: most machine learning algorithms require a training set consisting of large databases of labeled information. Despite much searching, I was not able to find any such large database online, and so I had to take a step back in the proposed scope of the project; instead of applying machine learning algorithms, I decided that I would use more classical image processing techniques. This is where the Kurniawan *et al* paper served as a useful guide.

At the end of week 1, I had formulated a plan on how my program would accomplish fracture detection:

1. Canny edge detection
2. Bounding rectangle detection
3. Rectangle overlap detection

(It is worth noting that the final structure of the program did not use bounding rectangles but instead modeled bones as lines. This will be discussed later in the section)

At the start of the second week, I started working on Canny edge detection; my week four milestone was to successfully complete it. Canny edge detection is a popular edge detection algorithm, which is often used instead of the simple Sobel filter because it results in sharper edge detection with less noise. In Canny edge detection, edges are suppressed to be one pixel wide, which considerably simplifies the source image for future analysis. The results of the Sobel and Canny filters are compared in figure 1. The actual Canny edge detection is a multistep process:

1. A Gaussian filter is applied to the input image to filter out image noise; this is essential because otherwise the noise itself can contribute to the formation of an apparent pseudo-edge after differentiation has been done (see step 2).
2. A Sobel filter is applied. This is just a convolution of X- and Y- derivative matrices to our raw image. It is important to note that at each pixel in the image, we also save the angle of the edge by calling the function for every pixel with index [i][j]:
$$\theta[i][j] = arctan2(\frac{gradient\_x[i][j]}{gradient\_y[i][j]})$$
3. Non-maximum suppression is then applied. What this does is that at every pixel, the program looks at the direction of the edge detected. If the neighboring pixels *perpendicular* to the direction of that edge have a greater slope, then the pixel is deleted and the neighboring edge is preserved. This ensures that in any edge "neighborhood," the line thickness of an edge is only one pixel. This is one of the key differences between this and the Sobel filter.
4. Finally, we use a hysteresis-based thresholding method to threshold the edges. We define two parameters, a lower threshold and an upper threshold. All pixels with a derivative value greater than the upper threshold are immediately considered to be a part of an edge (and therefore assigned a value of 254). Pixels with a derivative value greater than the lower threshold are considered to be a part of an edge if they are neighbors with a pixel that has already been classified as an edge pixel. This way, if certain parts of the input image have a smaller gradient even though they really are on the edge, those pixels are incorporated into the edge as well.

Implementing the Canny filter took us longer than originally planned – while we thought we had completed it in the four weeks as scheduled, a few bugs popped up as time went on –

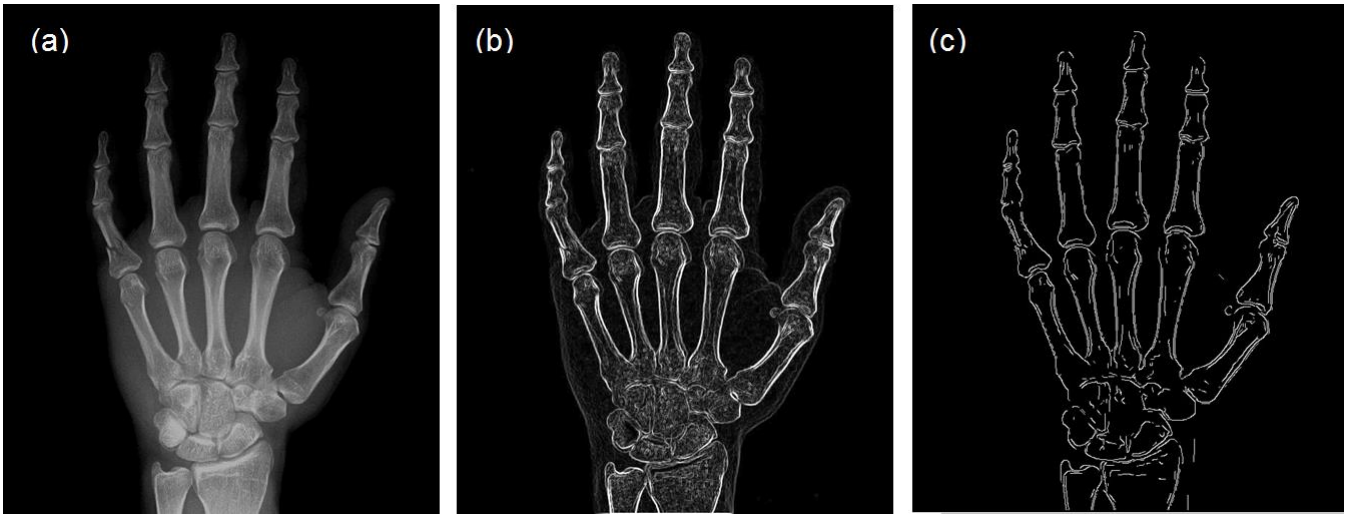

Figure 1: (a) input x-ray image **(b)** Sobel filter results **(c)** Canny edge detection results

chief-most of which was an error in my understanding non-maximum suppression.

A considerable amount of time was spent in week four trying to experiment with upper and lower threshold values to try to find a set which would work "universally" with all input x-rays. Sadly, we could not find such a universal threshold value pair. Thus, one of the weaknesses in this program is that the values for upper and lower threshold have to be individually chosen for each x-ray image.

It should also be noted that all of the work up until the end of week four was done on my personal laptop using the Visual Studio environment and not on the Code Composer Studio (CCS). This was done to avoid all of the quirks of the CCS when building my initial program and determining my initial logic; with the CCS, things are often more frustrating because you are not sure if your logic is incorrect or if the CCS has some unexpected quirk preventing your program from operating.

I started week 5 by porting over my code from Visual Studio to the CCS. To my surprise, all of my code worked like a charm, and I only had to make very minor changes. As a precaution, however, I changed all local variables to global ones because of the CCS's demonstrated tendency to work more smoothly with global variables. Another major advantage of using Visual Studio is that I was able to quickly change small parameters in my code (e.g. the upper and lower thresholds in the Canny edge detection). It was not possible to do the same with the CCS because building and transferring each updated program to the CCS took a longer time than compiling and running the program on my personal computer. Additionally, even a bigger hiccup was the fact that it took the CCS around 2 minutes to read the source images of my program, which had an extremely negative effect on my ability to iteratively change parameters in my code. Because of these issues and because of how smooth this process of porting code my computer to the CCS, I would highly recommend groups in the future to investigate if they could possibly do the same.

After having ported my program, the next step was for me to get openCV running so that I could use its capabilities as planned. One interesting thing to note is that the C version of openCV is actually deprecated – as of last year, the C program was completely discontinued only the C++ and Python versions remain under development. Thus, there was little support on how to properly install openCV, so after more than a few hours attempting to install openCV on my personal computer (so that I could code in Visual Studio as before, and then port things over to the CCS), I decided to change my strategy and switch over to exclusively working on CCS.

Luckily, I soon learned (by posting on support forums) that Texas Instrument actually includes a *full* version of certain openCV 1.1 libraries with our model of the LCDK, and not just the parts needed for the facedetect demo. However, this was not without problem: TI did not include one of the three basic libraries of openCV, "highgui," which is usually used to make the process of converting a normal image into an openCV image straightforward. Thus, the next step of my journey was learning how to work around this limitation in the library.

OpenCV stores its images in a data type called IplImage, which is actually a format from the Intel Image Processing Library from the late 1990's. To create a new 8-but depth image with 1 channel, one can do the following:

```
IplImage *frame =
  cvCreateImage(cvSize(width, height),
  IPL_DEPTH_8U, 1);
```

Next, it is worth noting that the image pointed to be frame can be directly modified as well through the imageData variable. For example, to set the first pixel in "frame" to 200:

```
frame->imageData[i]
```

The very interesting thing is that the imageData array is *not* of size height*width; it is of size height*widthStep, where widthStep is a data member of the IplImage structure which is automatically created to be the nearest integer multiple of four that is greater or equal to width. The makers of openCV implemented the storage in this manner because the openCV library was in the past optimized for working with images with width divisible by four.

When writing data directly to the IplImage structure, one should know that the 2D data is interpreted as concatenated row-vectors; for an image with width 12 and height 2, the first 12 members of the imageData array would correspond to the first row (e.g. height = 1) of the image. It must also be noted that if your images are not of width divisible by four, you can initialize to 0 members of imageData that are actually not in your image. For example, if an image had width 10 and height 2, one should set imageData[0...9] with the values of the first 10 pixels, and imageData[10, 11, 22, 23] to 0, because they correspond to the non-existent image pixel locations created by openCV.

This quirk with the widthStep took me some time to figure out and dominated my efforts and frustration during week 10: openCV functions do not work without the IplImage file formatted correctly, so it is essential that any student using openCV in the future know this.

Originally, I had planned to use 600x700 monochromatic images to feed into my program. However, this would have exceeded the internal memory of the LCDK. In general, this problem can be worked around if one uses the external memory available in the LCDK. However, in week 7 I learned that the external memory was inaccessible to us when used in conjunction with openCV. CCS consistently gave us a linker error informing us that there was a memory address overlap somewhere in our program. Yunxuan was extraordinarily helpful to me during this time – we looked through the linker files associated with the project, tinkered with the various memory addresses associated with different components of the LCDK, and read through the LCDK's manual provided by TI. Sadly, despite all of this effort, we were not able to determine what was causing our problem. That being said, it was a good learning experience – I have never before had to analyze such low level settings in a computer program.

Eventually, I decided that the best thing to do was to reduce the dimensions of the images used in the program. Instead of inputting X-rays of size 600x700, I changed my program to accept 150x250 sized pixels. This was approximately the maximum that the internal memory restrictions of the LCDK

could handle given the necessary computations in my program.

Further LCDK performance was the fact that we were not able to use Jerry's code to read and write images quickly: his program uses the "m_mem.h" file, which caused a memory overlap problem as previously described. This meant that even with inputs of size 150x250, image transfer to the LCDK was frustratingly slow.

Because of these limitations, we decided to optimize memory usage by converting our input x-ray images first to a monochrome bitmap format, and then in MATLAB to a text file with the bitmap value of each pixel. We then read this text file to the program as a means of transferring the input to the LCDK. This allowed us to save 66% of the space generally used when uploading a 3-channel bitmap to the LCDK.

Around week 9, we gave up on trying to optimize the LCDK's memory and decided to make our peace with the aforementioned issues. The next step was to use openCV's functions to our advantage. Our original plan was to use bounding rectangles to identify bones, but the version of openCV on the LCDK (1.1) had some limitations in bounding rectangle functionality that we did not foresee.

Thus, we changed our plan from using bounding rectangles to identify bones to using sets of lines to identify bones. To do this, we used openCV's probabilistic Hough transform (PHT) function. This is in general superior to the standard Hough transform for two major reasons: speed and versatility. Not only is the probabilistic Hough transform faster than the standard Hough transform, but it can automatically distinguish disconnected segments of lines with the same ρ and θ values.

Thus, instead of returning lines corresponding to ρ and θ, the PHT returns the two endpoints of every line segment detected. Additionally, the PHT allowed us to choose the ρ and θ resolution of the Hough transform. Furthermore, it also allowed us to choose the minimum length of continuous pixels for us to declare a line, and also the maximum length between two parallel lines such that the two lines would be joined into one bigger line. It is worth re-emphasizing that the PHT is computed on the results of the Canny edge detection, as the purpose of the edge detection was to simplify the image to allow downstream algorithms to deal with less information.

The probabilistic Hough transform worked as expected, but only with some growing pains: it was only through the failure of the PHT that we discovered the quirk of the openCV associated with the widthStep. It took us much of week 10 to understand what was going on, but after enough research, we were able to get things working as expected.

After this quirk was handled, however, things were more simple. Using the beginning and end points of each detected line segment, we were then able to use the three formulas below, to calculate slope m, y-intercept b, and the intersection of any two lines.

$$m = \frac{y_1 - y_o}{x_1 - x_0}$$

$$b = y_1 - m * x$$

$$x_{intersection} = \frac{(b_2 - b_1)}{m_1 - m_2}$$

$$y_{intersection} = m_1 * x_{intersection} + b_1$$

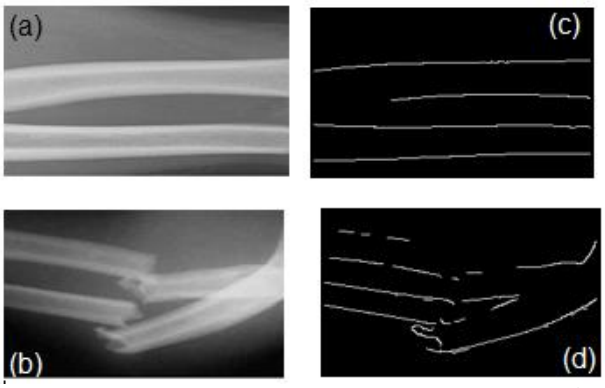

Figure 2: (2a) and (2c): the input and Canny output, respectively, of the unbroken ulna and radius x-ray. (2b) and (2d): the input and Canny output, respectively, of the broken ulna and radius x-ray.

The intersection coordinates of every pair of lines detected by the Hough transform was calculated. If the intersection point occurred within the 150x250 pixel grid of the image, then we declared that to be evidence of a potential fracture. However, if the two intersecting lines had extremely similar slopes, we did not count their intersection as evidence of a potential fracture because in some cases the PHT seemed to be too rigid in its definition of a line, "splitting" what should have been characterized as one line into two shorter lines.

It should also be noted that we do not consider the intersection point of two lines to be the location of a potential fracture, but merely evidence of a fracture. This can be seen in figure 2d, where the bottom right and bottom left lines clearly

To test our program, we used two sets of ulnar/radius x-rays of size 150x250; one set had broken bones while the other did not (figure 2).

IV. Results

The first major result of our program was the success of our Canny edge detector. A 600x700 input can be seen in figure 1a, while a 150x25 input can be seen in figures 2a and 2b. Respective outputs can be seen in figures 1c, 2c, and 2d. We claimed earlier that the Canny edge results were superior to the Sobel filter results. Comparing figures 1b and 1c, it is indeed evident that the Canny results are cleaner and seem to do a better job of containing the "true" edges of our bones and have very few false positive declaration of edges. In figure 2c, the canny edge results seem to be almost exactly where we would intuitively declare edges. One piece of the bone does seem to be "missing", showing that we perhaps should have decreased either the lower or the upper threshold in that experiment.

The next step of our process was the probabilistic Hough transform. As described above, the output of the Hough transform was a set of line segments. For the input x-ray in figure 2b, 6 line segments were discovered by the PHT, as visualized in figure 3a. It is clearly evident that some line segments were missed by the PHT; this problem could be fixed with experimentation of the PHT's parameters; sadly, we did not have time to do this, partially because every cycle of changing a parameter, building the program, and sending and

running the program on the LCDK took about five minutes and was not an efficient use of our time.

It can then be noted that the intersection detection portion of the algorithm discovered six intersections in total, as displayed in figure 3b.

Next, we re-ran the experiment with the unbroken bones. As before, the PHT returned a set of line segments and the line intersection algorithm returned sets of intersections. In this case, the program returned 5 intersections. While this is less than the number of intersections from the broken bone case, the difference is only 16%, which limits our confidence in our technique.

However, if we inspect and compare figure 2c and 2d, it is apparent that there *should* be fewer line intersections in the unbroken case than in the broken case. This discrepancy may be attributed to the fact that the unbroken bones have some curvature associated with them, so the PHT splits such bones into two segments that naturally intersect. This problem can be easily rectified by either making the PHT ρ resolution coarser or by detecting that certain pairs of segments have similar slopes and are connected by some intermediate section with low derivative in the general direction of each of the two segments.

Furthermore, another improvement that can be made to make our algorithm better is to perhaps use the version of the Hough transform for curved objects – bones being, after all, curved. Furthermore, we can also introduce yet more features to analyze. For example, we could add a module that analyzes the spatial FFT of the x-rays, with points with high x and y frequencies perhaps being indicators of fracture.

V. Conclusion

All in all, this was a good project in which I learned a lot. Naturally, my knowledge for Canny edge detection and the Hough transform was improved, but even more than these successes, the consistent sources of frustration in the project were also good learning experiences – for example with regards to libraries and linkers, both of which I had zero experience with before this project.

Looking back, I think that one should only try to experiment with libraries if one intends to use them extensively. This is because it takes time to set up and understand the library – a lot longer time than I thought it would. However, I think the work I have done this quarter should be helpful for future students in understanding how to use openCV in their projects; from what I have seen, properly setting up images seems to have been the biggest battle with regards to using openCV. Of course, the memory issue remains, but a solution can likely be found given a longer investigation.

I am cautiously optimistic about the program we have created: while its results are currently not ideal, I believe given the changes I have discussed above, we can indeed build an at least respectably-reliable simple program to detect fractures in bones. Thus we have done a fair shot of reaching our goal of providing a simple proof-of-concept of using computerized techniques to identify fractures.

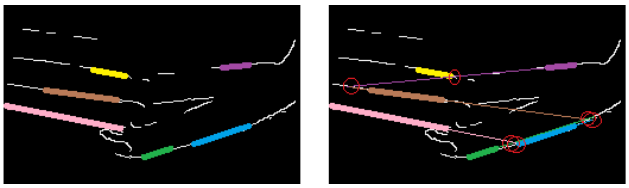

Figure 3: (3a) the line segments detected by the PHT. (3b) the intersections of lines (red circles) as detected by our program.

VI. References

[1] W. H. Organization. [Online]. Available: http://www.acr.org/~/media/ACR/Documents/PDF/Membership/RFS/global%20health%20imaging%20curriculum.pdf.

[2] "Medical Image Analysis," [Online]. Available: http://www.medicalimageanalysisjournal.com/article/S1361-8415(12)00033-3/abstract.

[3] K. Finley, "Robot radiologists are going to start analyzing x-rays," Wired Magazine, October 2015. [Online]. Available: http://www.wired.com/2015/10/robot-radiologists-are-going-to-start-analyzing-x-rays/.

[4] S. Kurniawan, K. Putra and K. Sudana, "Bone fracture detection using openCV," *Journal of Theoretical and Applied Information Technology*, vol. 64, no. 1, 2014.