Augmented Reality Equation Plotter

Salman Naqvi, Uzair Sikora
Digital Image Processing
EE 368/ CS 232

Abstract—Graphical Visualization of an equation can often help us better understand underlying trends of real variables. Moreover, visualizing it quickly as we come across it in a journal article or textbook is often desired in our scholarly readings. Typing out that complex equation linearly into a numerical computing environment, e.g. MATLAB, can be time-consuming and hassle-prone. With the high-resolution camera, fast-processing, and cheap cellphones available to us, it is highly beneficial to have an equation recognizer and graph plotter application in our phones. Such a nifty tool can help younger students working on building their intuition to be able to visualize simple equations quickly in their thoughts. Here, we try to develop such a tool for the Android platform. We leverage the OpenCV Android library to perform client side image processing and MATLAB Image Processing tool to perform sleek mathematical character template matching and equation formation. The application can read in any printed equation and plots on screen the graph of an explicit equation composed of exponentials, polynomials, trigonometric functions and simple derivatives and integrals.

Keywords—Recursive Segmentation, Binarization, Character Recognition, Tesseract, Template Matching, Keypoint and Edge Detection, Android Development.

1 INTRODUCTION

EQUATIONS can at times be tedious to understand and more so, difficult to quickly visualize the underlying trend of. The situation worsens if an equation is a complicated mixture of polynomials, exponentials, and trigonometric functions, involving algebraic fractions and exponents. Fortunately, modern numerical computation scripting languages, such as MATLAB, can help us quickly visualize such equations. However, in case of limited access to MATLAB or to avoid writing out large equations linearly such that they are understandable by the computer, we need a tool that can snap a picture of the equation printed on a paper and in near real-time process and return the associated graph on a cell phone screen. Moreover, the graph reading process can be made interactive again by using the image processing techniques, and the user can view the graph at different perspective and read values off of it.

This entire process involves a number of image processing challenges. First one is the mathematical equation recognition. Text recognition has been widely studies in the past and efficient hand writing detection is still an area of active research. On the topic of mathematical characters recognition, a number of survey papers published in the past decade or so, [5] and [6], sum up the developments quite well. More recently, there has been even greater work being done in the area of intelligent recognition and retrieval of mathematical expressions [3]. The second process involves a fast axes detection and plotting with perspective and skew correction.

We implement these two main segments of image processing to develop a complete user-oriented cell phone application which can achieve its goal efficiently and very fast. We talk about the algorithm flow chart in the next section.

2 APPLICATION FLOWCHART

Our equation plotter tool is organized into a series of "states" that are called into activity depending on which action needs to be completed in the workflow. An overview of the states is shown in Figure 1. The application starts in the "Capture Equation" mode. In this stage, the user captures the image of the equation they would like to plot. The user can retake the image and therefore, the state restarts. Otherwise, the application moves into the "Equation Analysis" stage. Here, the image is sent to the server running MATLAB. Image segmentation, and math character recognition part of the image
processing takes place at this stage. The application then receives the converted equation string and displays it for the user next in the "Query User" stage. At this stage, the user has the option to edit the string and/or approve it for the server to return the coordinates' vectors for the plot. The application then moves into the "Axis Detection" state. Here, the remaining of the image processing takes place including the canny edge detection, dilation, key-point detection, and axis rotation estimation. After the application has analyzed the camera preview frames and detected the coordinate axes on which to plot, we move on to the next state, "Plot Points." At this point, the application calculates the location of the coordinates on the phone screen, draws the points, and returns to the "Axis Detection" state. During this loop between "Axis Detection" and "Plot Points," the user has the capability to start a new equation process and thus the user can return to the "Capture Equation" state. The state machine approach used in this application greatly simplifies the integration of different functions and processes as needed.

a PHP script to upload the image data to a file at the server. This script then waits for the MATLAB to generate an output file at which stage, it returns the contents of that file as an http string. We also ensure that the image is not bigger than 640x480 pixels which limits the file size to under 175kB for fast cell to server communication. We also require user to ensure that only the equation is captured by the image and no surrounding clutter (text, other equations, etc) are included in the image. This ensures that all regions segmented by our algorithm are actually part of the equation. We can also include the functionality of allowing the user to crop the image after taking the picture so that only the desired equation area is extracted and sent to the server. Figure 2 shows an example captured image which we will use as a reference figure throughout this paper.

Fig. 2: Image captured by the Android phone

3 SERVER SIDE

3.1 Image Capture

The process begins with the image capture by the cellular phone at the client’s end. Once a decently focused, uniformly lit and closed up view picture of an math equation has been taken, user posts the images to the MATLAB server. This post initiates

3.2 Binarization

We need to binarize the image well so that the only regions in the image are the math characters and then we can segment each individually. Otsu’s method can give a reasonable threshold level but with fairly disturbed background, we will see alot of background noise getting binarized as well. One technique is to use adaptive Otsu’s method which finds the local thresholds by sliding a window across the entire image. However, for our real-time application this method is very slow. Thus we instead implement the background subtraction from the image in order to remove any non-uniform lighting effects before we apply Otsu’s method to find a global threshold.

Background is determined by dilating the gray scale image with a 24x24 disc shaped structuring element. However, a simple dilation gives blocky result which hampers a good binarization. Thus, we
instead use a rank filter to blur out any characters from the image in a way that the resulting background retains its smooth blending across pixels. After trial on a number of lighting and image capture disturbance cases, we found that a rank of 160th maximum pixel works well. This background subtracted image looks as good as the binarized image 4 and it is then binarized very efficiently using Otsu’s method gray level threshold.

After binarization, we smooth the corners of the characters, by an erosion operation followed by a dilation. This is done to improve the text recognition results by template matching later on. We also perform an opening operation to remove any small salt and pepper noise that might have been picked naturally during picture taking process.

After the binarization, we can clearly visually distinguish the individual math characters. And if the pixels are high enough and the image was focused well without any motion blurring, each character is clearly separated from the other. We then begin separating the characters horizontally, going from left to right in the binarized image. We use the vertical projection to find out where the white spaces are that separate the characters. Gradient zero crossings of the image projection are used to detect the region edges, and these regions are extracted as subimages. However, our job is not done yet, because our equation might have fractions, in which case, we will have to separate the numerator and denominator first and then to parse out individual character of the numerator and denominator respectively. This will therefore require region separation in vertical dimension on each of the horizontally separated region. We also have to do so methodically so that we keep the right order of numerator followed by the denominator. After the vertical separation of fraction, we will attempt a horizontal separation again to separate the numerator regions. Moreover, there can be nested fractions inside the numerator that we should separate. In summary, we write a recursive algorithm that keeps on separating in y first to separate any lines, then in x to separate regions horizontally, then in y to separate any fractions within each region, and then it repeats until no more separation in x or y is possible. Figure 5 depicts the process visually.

**3.3 Recursive Image Segmentation**

After the binarization, we can clearly visually distinguish the individual math characters. And if the pixels are high enough and the image was focused well without any motion blurring, each character is clearly separated from the other. We then begin separating the characters horizontally, going from left to right in the binarized image. We use the vertical projection to find out where the white spaces are that separate the characters. Gradient zero crossings of the image projection are used to detect the region edges, and these regions are extracted as subimages. However, our job is not done yet, because our equation might have fractions, in which case, we will have to separate the numerator and denominator first and then to parse out individual character of the numerator and denominator respectively. This will therefore require region separation in vertical dimension on each of the horizontally separated region. We also have to do so methodically so that we keep the right order of numerator followed by the denominator. After the vertical separation of fraction, we will attempt a horizontal separation again to separate the numerator regions. Moreover, there can be nested fractions inside the numerator that we should separate. In summary, we write a recursive algorithm that keeps on separating in y first to separate any lines, then in x to separate regions horizontally, then in y to separate any fractions within each region, and then it repeats until no more separation in x or y is possible. Figure 5 depicts the process visually.

**3.4 Exponentiation and Fraction Detection**

Once all the regions are separated and saved as single character images, we iterate over all regions and compare the y-centroids of each region with the one preceding it and the one following it. If the y-centroid of the current region is lower than that of the preceding one, then it marks the beginning of exponentiation, and we check the beginning of exponent label in that image’s data structure. Moreover, if the y-centroid of the current region is higher than that of the preceding region then we
check the end of exponent label in that region’s data structure. After all the regions have been labeled, we individually place the \( \hat{\text{h}} \) (hat) operator followed by an open parenthesis at the start of exponent image region, and a close parenthesis at the end of exponent image region.

### 3.5 Image Properties and Resizing

It will also be useful in the character recognition phase to know a number of image properties of all the segmented character images. In particular as we shall describe later, we are most interested in knowing the following properties of each region:

1. **Aspect Ratio**, which is the ratio of the region’s major axis length to the minor axis length,
2. **Solidity**, which is the ratio of the number of region pixels to the number of pixels in the region’s convex hull,
3. **Number of Holes** in the image, which are obtained by subtracting from the number of connected components inside an image the image’s Euler number,
4. **Centroid’s y-coordinate**, which gives the y location of the image’s center of mass and is helpful in determining the separation of characters into different lines,
5. **First Hole’s Centroid’s y-coordinate**, which will be helpful in distinguishing between various similar characters like \( e \) and \( 6 \).

We store these properties along with the image in a data structure which is stored in the character image database for later use during the recognition step. But before storing the image, we extract its tight bounding box and then resize it to 64x64 pixel, for small disk space usage and easy template matching later on. The result segmented set of subimages is shown in Figure 6.

![Fig. 6: Segmented regions resized with exponents and fractions detected](image)

### 4 OCR, Optical Character Recognition

We try two main methods for recognizing the segmented math characters’ images and we compare their shortcomings. As we discussed earlier, recognizing hand-written equations is more challenging and requires that a Support Vector Machine (SVM) prediction model be trained to process these. Hand written characters also need to be clearly separated and decently consistent in shape and size with the repetitions. SVM is a supervised machine learning algorithm developed at the National Taiwan University which analyzes data input and recognizes pattern.

For us however, the task is simpler because our application is targeted towards recognizing printed equations appearing in a textbook, course notes, or journal articles. Thus our first plan of attack to accomplish reasonable OCR is to use an open source, free OCR engine, called Tesseract.

#### 4.1 Tesseract Output

Tesseract is an freely available OCR engine that was originally developed at the HP Labs and is currently being supported by Google [1]. It is known to perform well in generalized cases of text recognition where characters are limited to alphabets and digits and there is a pattern of alphabet occurrence modeled by Markov chains. It employs many other tools to recognize text than the simple template matching and it needs to be trained for any other character set other than standard english letters and numbers.

In our case, since our goal was to be able to detect, exponentials, trigonometric functions, polynomials, simple derivatives, and general math symbols, our character set was large and so alien for Tesseract that the output was really embarrassing. Moreover, we also allowed for different fonts, which all cumulatively caused Tesseract to give very poor results to the expression shown earlier. The result in which \( y, =, \), \( \hat{\text{h}} \), and \( e \) were not recognized is shown below.

![Fig. 7: Tesseract Output to our example image](image)

Hence, we use an in-house template matching algorithm combined with post-processing of results which generates remarkably better results.

#### 4.2 Template Matching Database

First, we set up a template matching database by writing down all possible characters that we expect to see in the types of equations we would like to solve. These amount to 35 characters. We also allow for equations to be written in 5 commonly
used fonts, including Arial, Times New Roman, Courier, Cambria Standard, and Cambria Math. The database is fed into the algorithm in the form of an image containing all characters written in all fonts separated in different lines. The algorithm runs a recursive image segmentation to quickly parse out bounding boxes of all characters in the proper order, computes all the helpful image properties as discussed earlier for all characters, and then resizes the images to 64x64 templates. The process is extremely fast and robust, and makes the template matching database easily expandable. Figure 8 shows the image used to populate the database.

Fig. 8: Binarized image of the 35 characters with 5 font styles fed into the algorithm for forming the template database

After having the database setup, we use the segmented subimages from the equation image to find the best matched template in the database, which has an associated ASCII character also stored in the database. Since both the template images and the equation segmented images are of size 64x64 pixels, it is easier to compute their matrix dot product. The template giving the maximum dot product is taken to be the best match. We show the template matching text recognition result for our example case below.

\[ y = (x'(2) + 2x + 2(x))xe(x'(s) + 5x) \]

Fig. 9: Template Matching Output to our example

**Smart Character Fixing based on Image Properties**

We notice that certain characters are matched incorrectly to a template in the database. In particular, \( x \) was mistaken with a / (forward slash), while 3 was mistaken with an \( s \). This confusion between various characters can result if the characters appearing in the image are skewed (skew can result during image capture), pixelated (pixelation can result if frame was not zoomed in enough), blurred (due to instability during camera operation), or poorly captured due to improper lighting conditions. However, we can tackle with a decent amount of confusion by further filtering false positives using template and segmented images region properties. First, we ran the algorithm on more than a dozen images of different equations and font styles, and under different lighting, zoom, and skew conditions. We figure out what character pairs are most likely to be confused and then apply smart character fixing. This is shown in the "Confusion Matrix" plotted in Figure 10:

Confusion matrix would keep on becoming more dense and mature as we apply the recognition on more and more diverse cases. And as it populates further, we will get more information about remedies to adapt in correcting false positives. Based on the current state of the confusion matrix, we adapted more than 2 dozen post-recognition checks to correct the false results. For instance, '8' is the biggest culprit which not so surprisingly gets matched with many character images. Thus, we deals with its set of confusion pair as follows:

- If the recognized character is 8, but the original image has 1 hole, then if the y-centroid of the hole is above the y-centroid of the region then recognize it as 'e', otherwise recognize it
as '6', and if there were no holes in the original image and the AR is little bit higher than than of '8', then recognize it as '5', otherwise in case of small AR, recognize it as 's'.

With smart fixing in place of major characters such as 8, (, ), 6, l, etc, our recognition results improve as shown below. Now, 'x' is correctly replaced with '/', but 's' was still wrongly replaced with a '5'. We would need to acquire more image features to distinguish such similar characters in the future. For now, we get the remaining problems with the recognition fixed manually by sending the recognized equation shown below back to the user.

\[ y = \frac{(x'(2) + 2x + 2x')}{e(x'(5) + 5x)} \]

Fig. 11: Recognized Equation after Smart Fixing

4.3 Equation parsing in MATLAB Recognizable Format

After user corrects the equation and sends it back, our string parsing algorithm kicks in to put the standard linearized equation into a form that is recognized by a MATLAB or Mathematica. Major changes to the string in this stage includes:

- adding '.' before the multiplication, division, or exponentiation with an independent variable,
- adding multiplication sign before the parentheses,
- replacing \(e^x\) with an \(\exp(x)\),
- \(d/dx\) with \(\text{diff}(x)\) for differentiation, etc.

Ready to solve equation looks like this:

\[ y = \frac{((x'(2) + 2.\times x + 2.x'))}{(\exp(x'(5) + 5.\times x))}; \]

Fig. 12: MATLAB Recognizable equation as Returned by the Parsing algorithm

MATLAB solves the equation for a given range of x values, which user can also specify, and intelligently chooses the y-limits to show the important plot features within the range of x, and sends back to the client, the string of x, y coordinates and their respective ranges.

5 CLIENT SIDE

5.1 Coordinate Axis Detection

Once the server has processed the user corrected equation character string, the user is presented with the camera preview frame. At this stage in the Android application, the plot is created and overlaid onto the image of the user provided coordinate axes. The application must continually locate the axes and plot the points as the image of the axes shifts and rotates in the preview frame. We implemented the image processing on the phone using the OpenCV 2.4.5 [2] Java API. The image processing is done between preview frames, and therefore, for the application to remain stable (and not frustrating to the user), the image processing must conclude quickly to minimize lag. Therefore, this adds a limitation to the complexity of the image processing operations.

The goal of the coordinate axes detection system is to locate the endpoint coordinates of the two axes. From these endpoints, the plot area, origin, rotation, and scale of the plot can be determined. We start the detection process by first extracting the grayscale intensity image from the camera preview image. The Android camera provides images which are in the YUV colorspace. It follows that we can simply use the Y data from the frame and set it as the grayscale image.

We then feed this grayscale image into OpenCV’s Canny edge detector to extract the edges of the coordinate axes. The Canny edge detector provides a good binarization of the image if there are no extraneous edges in the camera’s field of view such as printed text or stray marks. The primary issue with allowing extraneous blobs in the image is that they might be candidates for the possible endpoints of the two lines. These marks can be removed after Canny edge detection, using a region labelling and region properties algorithm. However, as we will see below, OpenCV’s implementation of this algorithm slows down the coordinate detection too greatly to be utilized.

The next stage of the detection process is to feed the Canny edge detected image into a closing operation to fill in the edges. The Canny edge detector [8] returns an image with the outline of the axis. We use a dilation and erosion operation implemented by OpenCV using a structuring element that is a 20 by 20 pixel square. The result is an image with a region that connects the two sides of each of the edges. Finally, we use a Harris keypoint detector to detect the corners in the image. Typically, this will result in two keypoints being detected at each endpoint of the line. In addition to these points, keypoints are detected along the edges of the lines due to irregularities in the drawn line or due to
stray marks that were left over after the binarization process. We detect the endpoints of the lines by just picking the outermost keypoints (keypoints with the maximum x, minimum x, maximum y, and minimum y). Theoretically, on a clean paper with no major stray marks, this process will work unless the user has rotated the axis too far. In this case, some of the outlying keypoints will be the same, and the application will not be able to plot the points. Picking these four keypoints allows the application to calculate the position of the origin. The origin is determined by finding the intersection of the two lines formed by the four keypoints. A simple ratio of determinants easily solves the system of two linear equations. The next step is to start plotting points.

\[ r' = \sqrt{\left(\frac{aL_x'}{L_x}\right)^2 + \left(\frac{bL_y'}{L_y}\right)^2} \]  
\[ \theta' = \arctan \frac{bL_y'/L_y}{aL_x'/L_x} \]

Where \( r' \) is the distance from the origin of the new point, \( \rho \) is the angle the point subtends from the horizontal direction on the phone, and \( a' \) and \( b' \) are the coordinates of the point in the phone’s coordinate system. \( L_x' \) and \( L_y' \) can be calculated by finding the distance between the two end keypoints and the origin. Each point the MATLAB server responds with can be transformed using this method, with another check to determine if the point is in the positive y quadrants or the negative, as the calculation of \( \rho \) would be a subtraction instead of an addition.

6 Experimental Results

6.1 Equation Recognition

Algorithm was tested on several types of equations and under various illumination conditions for taking equation’s image. We see convincing results in all these attempts. Unfortunately a perfect output is far from real at this stage; mostly at least one character is recognized wrong or parentheses are not closed. Fortunately, the user intervention at this stage solves the remaining issues.

For our detected results, we explore further the reason of getting poor template matching for certain types of equations. For our reference example, we plot a histogram of the distribution of matched characters to the different font styles in Figure 14. One not so obvious, but easily justifiable observation, is that characters get wrongly matched to Arial font style a lot and hardly to the Times New Roman. This is to be expected because Arial font characters have no complicated edges and have most thick regions.

Our smart character fixing jumps in to the rescue and we get really got post-processing results. Figure 15 shows the results of the algorithm applied to a number of different equation images and the corresponding results. We notice that most bad results come in when we are trying to match character pairs: s, 8, e, c; i, l, t, —; and 6, (, 9, ) etc. Again, as seen in the figure; most matching is to the Arial font style characters, although all the images used had
6.2 Coordinate Axis Detection

The approach that we decided to use for the final application using a Canny edge detector, close oper-
ation, and Harris keypoint detector, was chosen after implementing several different approaches and examining the tradeoff between quality of detection and processing time. Since the real time plotting aspect of the application needed to be processed with minimal lag time from the actual preview frames, the approach we would choose would have to be both quick and effective enough.

Figure 16 shows the timing results for the three approaches we tested. Each test was performed using the same hand drawn coordinate axes to be detected. The first approach used background subtraction by subtracting the original grayscale image with a dilated version. The flat dilation was performed using a structuring element shaped like a circle with a diameter of 25 pixels. Then adaptive thresholding was used with a block size of 10, and the largest region was kept. The endpoints of the lines could be determined by finding the outlying bright pixels. This processing was completed in 1320ms on the Motorola DROID phone.

The second approach we tested was using the Canny edge detector with an OpenCV function that provided the coordinates of the endpoints of the lines it detected. This method completed in 2632ms.

Finally, we tested the current approach (explained in the Methods above). This performed in 978ms.

All three approaches were able to detect the axes. However, the large delays caused by the two former approaches pointed to using the last approach as the final approach.

Figure 17 shows the original camera frame containing the coordinate axis to be detected. After performing the Canny edge detection algorithm, the output image is shown in Figure 18. This image shows the detected edges as a line of pixels. As can be seen, the binarization process also removes
the background illumination variation. Figure 19 shows the result of the closing operation. There is now only one region in the image. Next, the Harris keypoint detector is used to find the endpoints of the axis. Figure 20 shows the keypoints that were detected. Since the lines are not perfectly straight, the Canny edge, and closing operations leave small corners that the keypoint detector picks up. This is not an issue because of our outlier keypoint selector.

Figure 21 is a photograph of the phone screen displaying the plot that was generated using the points returned by the server. As shown, the plot displays correctly overlaid on the drawn coordinate axes.

7 DISCUSSION

7.1 Coordinate Axis Detection

The results for the axis detection approach are promising. For relatively clean images, the process works as expected. However, the approach does not take into account outlying regions and therefore the axes would not be accurately detected. The approach of region filtering with the Canny edge detector and Harris keypoint detector could remedy this issue. However, the delay additional processing introduces is an issue that must be dealt with. For relatively newer phones, the processing power upgrade possibly could limit the time lag caused by the processing and more sophisticated detection could be performed.
If future interactivity features of the application are taken into consideration, the approach used here for axis detection might be beneficial. It is both fast and accurate enough to detect the axis, but in addition, keypoints detected within the bounding box around the plot area defined by the limits of the axis are not possible axis endpoints. Therefore, marks made in the interior of the plot do not affect the detection. An additional feature of the application could take advantage of this by allowing the user to draw points in the interior of the plot area. These could be detected as regions or possible keypoints that do not fall near the axis. The coordinates of these user defined points could be provided for further plot information.

8 Future Work
This tool has many useful extensions and many beneficial applications. As a first next step, we can work on making the real-time plotting of the graph insensitive to perspective distortion along with skewness. We would also like to make the OCR insensitive to image capture, in particular, making it insensitive to skew and perspective distortion. We can use the RANSAC MATLAB Toolbox from the Graphics and Media Lab of Lomonosov Moscow State University [7] to compute RANdom SAmple Consensus (RANSAC) that fits to the top and the baseline of the equation in the image. In this way we can find the corrected bounding box of the image and apply the projective affine transform to deskew and perspective compensate the image. Also, to improve the character recognition in the current setup, we would like to make a more detailed confusion matrix and encode more classifiers for minimizing false positives. We would also like to allow for including the handwritten equation recognition capability by training an SVM as discussed earlier. Another goal of the project that we stopped short of achieving was to add the interactive plot reading capability, where user can put a dot within the axes premises on the paper, and the tool can overlay on the screen the reading of the graph at that location. Finally, being able to visualize 3D graphs would be a gem of an accomplishment for this type of a project.

9 Conclusion
We have successfully showed that near real-time printed explicit mathematical equation plotting can be achieved using Android OpenCV and MATLAB server. We get really nice results with relatively simple template matching scheme with relatively low number of post-matching image classifiers.

Acknowledgments
The authors would like to thank Prof. Bernd Girod for teaching them nifty image processing tools to solve many useful real-world problems. They are also indebted to the helpful Teaching Assistants, David Chen and Matt Yu, who was ever-ready to discuss project ideas and provide critical feedback. Moreover, various intellectual discussions held with Sam Tsai, graduate student at the SCIEN Lab, were really mind-opening and helpful for crafting pathway for character recognition. They also extend their thanks to friend, Patrick Blaes, for helping out in setting up the server.

References