Circuit Simulpaper

Authors: Jaden D'Abreo, Stephen Dai, and Devan Grover

Department of Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*— This project is intended to serve as an educational tool for children in middle school to learn basic circuit functionality through drawing. The project will be implemented by using a computer vision algorithm to identify components on a hand drawn circuit, feed the identified circuit into a circuit simulator, and display the analyzed circuit, all of which will be done through a mobile application. This system hopes to serve as a fun and safe way for children to learn about circuits through the appeal of drawing and without the risks of electrical components.**

***Index Terms*— Circuits, DC Analysis, Computer Vision, Simulation, iOS Application**

I. INTRODUCTION

ELECTRICAL circuits are expensive and potentially dangerous to experiment with. Not all students have the resources available to them to learn about circuits at a young age. To purchase the bare minimum components to create a simple circuit (breadboard, wires, resistors, power supply) people must spend at least forty dollars, which is a luxury that everyone cannot afford. Furthermore, improper education can lead to dangerous situations. If a student unknowingly causes a short circuit or improperly uses a component like a multimeter, they can harm themselves and their equipment.

When learning about circuits, students typically first learn about the different symbols designating various electronic components. Learning to draw circuits and using the proper symbols are imperative steps in the process of mastering circuits. Young students are also often fond of drawing and recent studies show that drawing serves as one of the most effective ways to retain knowledge[3].

Our application hopes to solve the issues of a lack of accessibility and safety when learning about circuits and capitalize upon young students' fondness of drawing. We are creating an application in which users can take a picture of a schematic they draw, upload the picture, and then receive a simulated version of the circuit they drew with voltages and currents labeled.

This application is primarily targeted towards middle school students. Students at this age are old enough to learn the basics of circuits, have access to mobile applications, and still indulge in drawing through school. Currently, there are no technologies that allow a user to upload a drawn circuit and have it analyzed. In addition, all circuit simulators online are accessed through web applications, thus less accessible to our primary use group. Furthermore, there are no applications that accomplish the intended goals of this project.

II. USE-CASE REQUIREMENTS

The use case requirements for our application encompass many factors of our application, with a focus on accuracy and usability. The use case requirements are as follows:

1. *The computer vision algorithm must have an individual component classification accuracy of 90%*

To ensure a good user experience for our application, being able to properly identify components is required. Users should not have to constantly take pictures of their circuits to get components to be identified properly.

2. *The computer vision algorithm must have a circuit classification accuracy of 90%*

We want to not only ensure each component is detected with an accuracy of 90%; we also want to ensure that all circuits are classified with 90% accuracy. This means that for the circuit options shown to the client, 90% of the time the user's drawn circuit should be one of the shown options. If the user does not see their circuit, they need to redraw their circuit.

3. *The circuit simulator must simulate circuits with 100% accuracy.*

Our circuit simulator must have perfect accuracy, otherwise the application cannot be used as an educational tool. Given an input circuit, our simulator must output the circuit with the correct voltages/currents at each node/component.

4. *The application's user interface must receive an average rating of 80%*

We will conduct user testing of our application on the target group of middle school students and provide them with a survey to record their feedback after testing. The survey questions must all have an average score of at least 8/10.

5. *The application must be free to the user.*

To ensure that most children can use our application, we need to make it free. This app was created so that children can learn about circuits without having to spend money on components, therefore we must make it free.

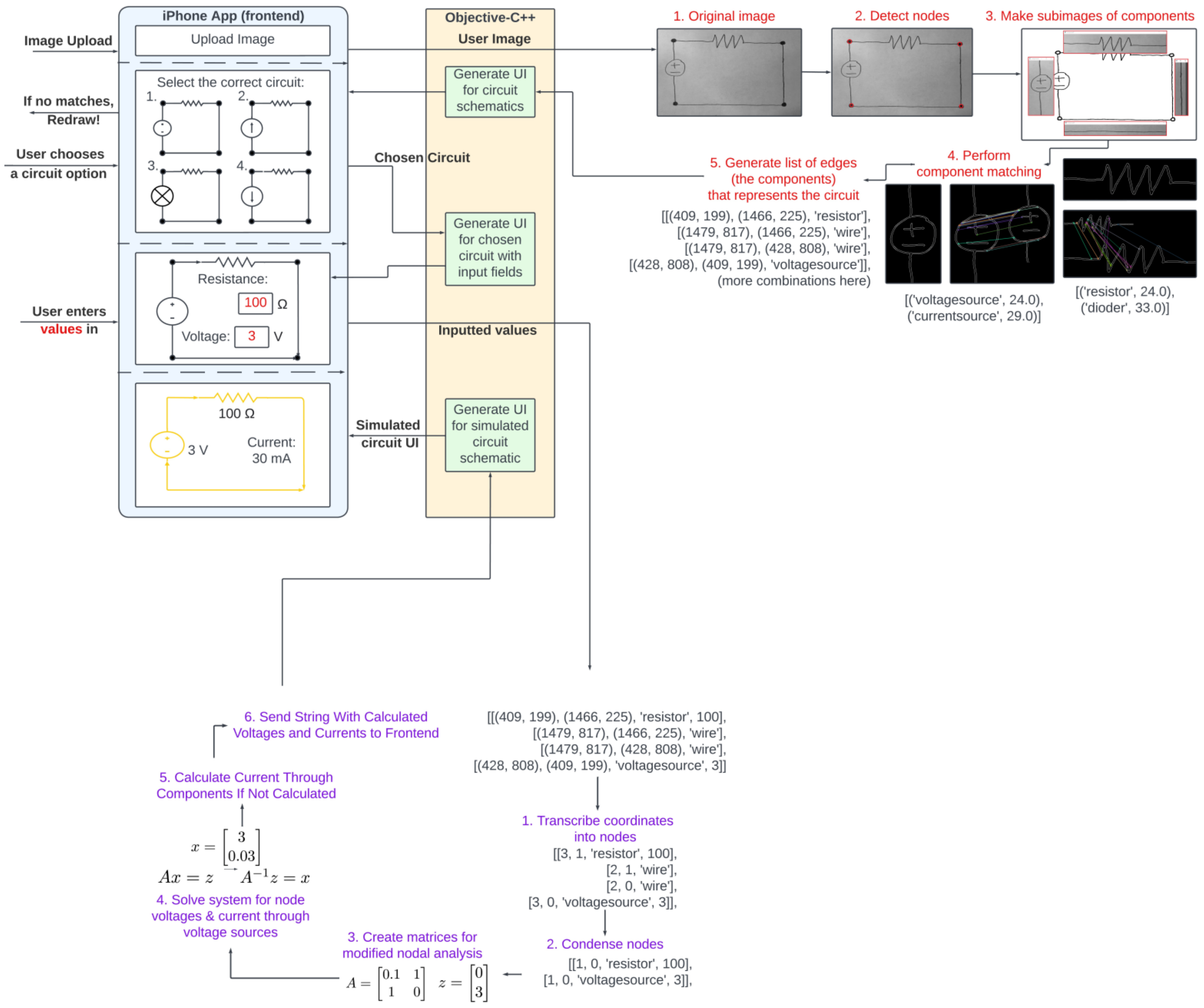
iPhone App (frontend)
Image Upload
Upload Image
Select the correct circuit:
If no matches, Redraw!
User chooses a circuit option
Resistance:
100 Ω
Voltage: 3 V
User enters values in
100 Ω
3 V
Current: 30 mA
Objective-C++
User Image
Generate UI for circuit schematics
Chosen Circuit
Generate UI for chosen circuit with input fields
Inputted values
Simulated circuit UI
Generate UI for simulated circuit schematic
1. Original image
2. Detect nodes
3. Make subimages of components
4. Perform component matching
5. Generate list of edges (the components) that represents the circuit
[[(409, 199), (1466, 225), 'resistor'],
[(1479, 817), (1466, 225), 'wire'],
[(1479, 817), (428, 808), 'wire'],
[(428, 808), (409, 199), 'voltagesource']],
(more combinations here)
[('voltagesource', 24.0),
('currentsource', 29.0)]
[('resistor', 24.0),
('dioder', 33.0)]
[[(409, 199), (1466, 225), 'resistor', 100],
[(1479, 817), (1466, 225), 'wire'],
[(1479, 817), (428, 808), 'wire'],
[(428, 808), (409, 199), 'voltagesource', 3]]
1. Transcribe coordinates into nodes
[[3, 1, 'resistor', 100],
[2, 1, 'wire'],
[2, 0, 'wire'],
[3, 0, 'voltagesource', 3]],
2. Condense nodes
[[1, 0, 'resistor', 100],
[1, 0, 'voltagesource', 3]],
3. Create matrices for modified nodal analysis
$A = \begin{bmatrix} 0.1 & 1 \\ 1 & 0 \end{bmatrix}$ $z = \begin{bmatrix} 0 \\ 3 \end{bmatrix}$
4. Solve system for node voltages & current through voltage sources
$Ax = z \rightarrow A^{-1}z = x$
$x = \begin{bmatrix} 3 \\ 0.03 \end{bmatrix}$
5. Calculate Current Through Components If Not Calculated
6. Send String With Calculated Voltages and Currents to Frontend

Fig. 1. System Architecture Block Diagram

III. Architecture and/or Principle of Operation

Our project is split up into three main parts: the computer vision system, the circuit simulator, and the front-end application. A block diagram of our application's high level functionality is shown in Fig. 1 above.

Once the user has uploaded their circuit image, the image will get sent to our computer vision algorithm for processing. The computer vision algorithm will first perform preprocessing on the input image to remove noise and make the nodes at the ends of each component more pronounced. Once the image has been preprocessed, a circle detection algorithm is used to detect the nodes. The algorithm will then detect all the components between nodes and create subimages of each component in the circuit.

Once the components are detected, ORB will be run on each subimage. ORB is an algorithm that generates keypoints and descriptors based on the input image. Keypoints are parts of the image that ORB determines are distinct. The descriptors are extracted from the keypoints and each one is a binary string of encoded information about each keypoint. These descriptors can then be used to match features with different images. Instead of using ORB's rBRIEF descriptors, we use BRIEF descriptors, which will be discussed later. After running ORB, we will use brute force matching to match the descriptors with precalculated descriptors from our reference dataset stored locally within the application.

At this point, we have scores associating the component subimage with a type of component, like resistor or voltage source. We will take the top three most matching component types for each sub image and generate every combination of circuits that can be made. For each of these circuits, we generate another score that is a summation of each individual component score. The algorithm will output each circuit as a netlist-like structure for our frontend and circuit simulator to use.

The five highest scoring circuits will be shown to the user on the application's user interface. The user can then select which of the shown circuits were the one that they drew. After selecting the circuit, they will enter values for each component (resistance, voltage, etc.). The circuit will then be sent to our circuit simulator, which will use modified nodal analysis to simulate the circuit. Once the circuit has been simulated, it will be sent to the front end where the circuit will be rendered. The user will then be able to see the simulated values for the circuit – this includes the voltage at every node and the current going through every component.

IV. Design Requirements

A high-level description of the design requirements can be seen below in Table 1:

TABLE I. DESIGN REQUIREMENTS

Description	**Requirement**
Computer Vision Maximum Latency	5 seconds
Maximum Application Size	100 Megabytes
Maximum Number of Components	8
Supported Components	Voltage/Current Sources, Resistors, Bulbs, Switches

We want to keep the application size low so that users do not have to uninstall or delete existing items from their devices to install our application. By looking at the size of other applications with similar functionality like "Tiny Scanner", we decided that our application would have a maximum size of 100 MB. Our app ended up being 32 MB, which is well under this requirement.

We also have restrictions on the components to maintain the effectiveness of our computer vision algorithm. We currently have a limit of eight components maximum per drawn circuit to ensure there is adequate spacing between each component and to ensure all components can fit on an image while maintaining the size requirements. The components must also be drawn horizontally and vertically at near-right angles so that we can easily detect components between nodes. We also planned to only perform steady-state DC analysis on the following components: voltage sources, current sources, resistors, bulbs, switches, and LEDs. Unfortunately, we were unable to implement diodes in the circuit simulator, which led to us dropping support for LEDs.

We also have a design requirement that the computer vision code should take a maximum of five seconds to run. This means that when the user submits their image of their drawing, they should see the circuit options within eight seconds of their submission. According to a study by Google, the probability of abandoning a mobile application's page loading increases by 90% as the page load time goes from one to five seconds[11]. The best practice according to Google is three to five seconds.

V. Design Trade Studies

A. *Offline Application*

When running our application, we decided to implement it with no need for internet connection. Due to a large portion of our target audience not having access to the devices with an internet connection[4] we decided that keeping the app offline will strengthen the argument for the accessibility use case and extend to a larger audience.

This means that we will be storing everything locally through the user's device. All that needs to be stored is reference data for dataset components. As one of the main benefits of having an online application is having access to large amounts of storage, we expect that our application will not exceed more than 100 MB. Most offline apps are anywhere between 20 to 100 MB in size, and ones that perform image parsing tend more towards the higher end of 100 MB. For example, the offline mobile application "Tiny Scanner", which generates PDFs (Portable Document Format) from pictures, has an app size of 92.5 MB, which does not include document and data storage. Because this application generates and stores PDFs, it takes up lots of documents and data storage. For our application, we will not store images of previously simulated circuits because a user can always just reupload an image of the circuit from their camera roll. Thus, we consider the 92.5 MB as an appropriate benchmark for our application storage. By constraining the amount of storage our application will use, we believe keeping the application offline will reinforce the goals of our project while also functioning the same as an online application.

B. *Mobile Application*

For how we want to display our application, we decided to implement a mobile application. We needed to decide between either a mobile application or a web application and after much thought we decided that a mobile application not only aligns with our use case of accessibility more. If the project progressed with a web application the user would have a much harder time uploading their drawing and uploading their picture rather than just doing it on their phone. In addition, this application is intended to be offline due to accessibility, thus a web application would not be possible to implement. Recent statistics show that around 71% of 12-year-old have phones and by age 14 roughly 91% have phones[5]. Furthermore, a different study stated that 95% of U.S. teens have access to a smartphone at home, while only 88% have access to a computer [1]. This comes out to around 2 million people of our target audience. As phones are much more of a necessity than laptops around the ages of 12-15, we felt like tailoring the application to be used on a phone would align more with our goal of making this project an accessible approach to help educate all those that are interested in circuits. Therefore, we decided that these reasons were enough to pursue the mobile application.

C. *Application Stack*

Swift is the most popular framework used for iOS applications today - almost all applications nowadays use it. Swift is a more modern language and easier to learn when compared to Objective-C, which is why we are using it for our frontend. Objective C has traditionally been used to develop iOS applications because it has existed for nearly forty years. Swift, which is much newer, is also 2.6 times faster than Objective-C[8] and has memory and type safety built in. We will have to use Objective-C++ because OpenCV is only available as C++ library, and Objective-C++ is compatible with both Swift and C++. To allow the Swift application to interact with the C++ code, we will need to create bridging files in Objective-C to bridge the Swift and C++ codebases.

D. Computer Vision Architecture

We opted to use a more traditional computer vision architecture that does not utilize a neural network. Given that we decided on creating a mobile application that does not require the internet, we cannot feasibly utilize a neural network as such. Loading and using a neural network locally in-app is too computationally and storage intensive for a mobile phone, especially for older iPhones. Ultimately what we considered is how much of a difference there is in terms of accuracy when using and not using a neural network. Some research that has been previously conducted has achieved a 95% accuracy in classifying electrical components using a CNN (Convolutional Neural Network)[9], and additional research has been conducted to achieve a 90% accuracy using KNN (K Nearest Neighbors) without a neural network [10]. Given we are also not using a neural network, we use this 90% as a benchmark for our design as well. We acknowledge that using a neural network would greatly increase the accuracy of classifying individual components, but this would come at the expense of integrating our mobile application with a backend server that can support the neural network. Because the image preprocessing and usage of ORB would stay the same, we have left the integration of a neural network for work that can be done post-MVP to further improve our accuracy goals if desired.

E. Circuit Segmentation

In order to be able to detect a circuit, we need to know what individual components make up the circuit. To know what the individual components are, we need to generate separate subimages of each component in the circuit and feed those subimages into a classification workflow. The most intuitive way to separate components in a circuit is by looking at pairs of adjacent nodes in a circuit, because a component is always between them. Because we also use nodes in netlists and to perform nodal analysis, first detecting the nodes in a circuit is the most intuitive first step in our computer vision subsystem.

We considered two solutions to detecting nodes: Hough line transform and Hough circle transform. Hough line transform would be used to detect the line parts of components that represent their ends, and we could denote the existence and location of a node as the point where two lines intersect. For Hough circle transform, we would require that every node be drawn as a circle, and the transform will just detect every circle in the circuit as a node. After testing both implementation ideas, we ultimately decided on using Hough circle transform and requiring the user to draw nodes as circles.

From testing with Hough line transform there were two fundamental problems. The first problem was that the line transform does not perform well with hand drawn lines. Hand drawn lines are usually never straight, and the transform consistently breaks down one line representing one end of a component into multiple connected lines. The other critical problem is that there are many lines that intersect in circuits that do not represent nodes. For example, the “+” symbol in a voltage source is made of two lines that intersect at 90-degree angles, which is exactly what we expect for our nodes. Thus, using Hough line transform is highly unreliable for node detection.

We believe that requiring users to draw nodes as circles is extra work for the user but has benefits that outweigh the extra effort needed. Detecting the nodes is arguably the most important step in the circuit detection: if one node is missed, that means that an entire component will be missed, rather than just one component being classified incorrectly. This means that properly detecting the nodes is critical to achieving our circuit detection accuracy marks. With the proper image preprocessing, node detection is highly reliable with Hough circle transform, which will be discussed later. Also, we believe that being able to identify where nodes are in a circuit is beneficial to the learning of our users. Being able to recognize points of shared voltages and know where current diverges is incredibly useful in doing elementary circuit analysis. Thus, we chose to use Hough circle transform and require users to draw filled-in circles for their nodes for increased circuit detection accuracy and for their own learning’s sake.

F. Feature Detection Algorithm

We considered four different combinations of feature detection algorithms, where each combination consists of keypoint and descriptor generation. Keypoints refer to the points/regions in the image that represent distinct features, and descriptors are the numerical vector representation of the image information surrounding each keypoint. Descriptors are what are used when performing matching between images to compare the similarities between features of the images.

The combinations we tested were as follows:

1. SIFT keypoint and descriptor generation,
2. ORB keypoint and descriptor generation,
3. ORB keypoint and BRIEF descriptor generation
4. FAST keypoint and BRIEF descriptor generation.

One thing to note about the second combination is that the ORB descriptors are called rBRIEF descriptors, where the “r” stands for rotation invariance. This means that if you had two images, but one was rotated, the extracted rBRIEF descriptors would match well even though one image had rotated features.

Originally, we believed that we wanted rotation invariance because we wanted horizontal and vertically aligned versions of the same type of component to match well. But we failed to consider that for certain components, namely voltage and current sources, we don’t want different orientations of the component to match well. As an example, say that we have a voltage source where the negative terminal is on the bottom, and another voltage source where the negative terminal is on the top. Because of the rotation invariance, although one component is the 180-degree rotation of the other, rBRIEF descriptors generated for each voltage source would match well.

This is problematic because a voltage source matching with one in the opposite direction would result in a different circuit being produced. In Fig. 2. we see the results of testing the combinations of feature detection. Each combination was tested with the same testing set of component images, and against the same dataset of component images. As expected, using ORB had the highest number of correctly identified components, but only when using BRIEF and not rBRIEF descriptors did we have the highest number of correctly identified orientations.

In this test, the correct component classification means that just the type of the component was identified properly (if a voltage source was upwards facing, if it was classified as a voltage source did it count as a correct component classification). From this point, when we discuss the component classification accuracy, we consider differently oriented (left, right, down, and upwards facing) current and voltage sources to be classified as different types of components, thus the component classification will consider both component type and orientation.

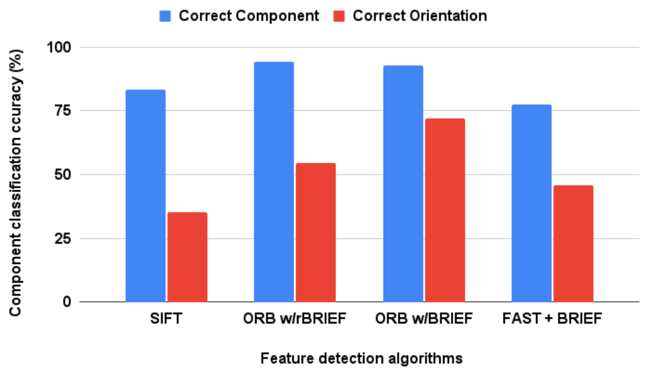

Fig. 2. Difference of individual component classification accuracies when using different feature detection algorithm pairs (keypoint and descriptor generation)

G. *Casing on Wires*

One design decision we made to classify components is to have special handling for wire components such that we did not need to perform feature matching and store them in the dataset. By doing this, we could save computation latency from needing to extract features from the wires and perform matching, as well as decrease our dataset size. Casing on wires and not other components is possible because they are unique in the way that they are simple and don't have many features to begin with, as they are just a line. How we do this is discussed in our implementation.

H. *Number of Considered Matches*

Another design choice for classifying components was how many of the feature matches should be considered when calculating the similarity score for a component and a dataset component. Typically, in object classification systems, only a subset of the total matches found between features are considered, such as the 20 best/most confident matches. Shown in Fig. 3 is the result of testing component classification accuracy when only considering the top N% of matches generated. For example, an N of 20 means that the component classification was performed where only the top 20% of matches were considered when generating the similarity score between the input component and a dataset component. The testing was performed using the same feature detection and matching algorithms, as well as having the same testing set and dataset. The results suggest that it is best to consider all (100%) of the matches generated.

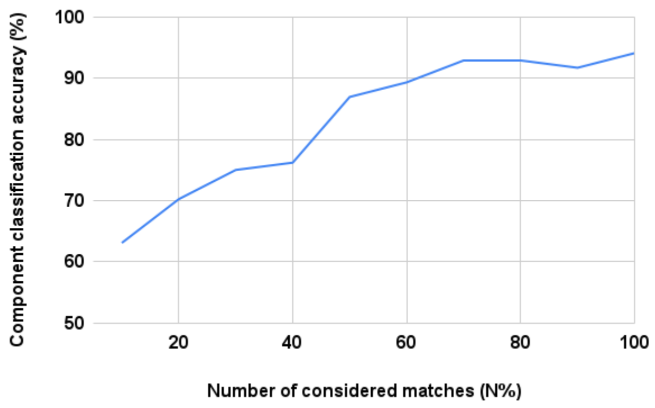

Fig. 3. Change in component classification accuracy with different percentages of considered matches

I. *Dataset Size*

The final design choice regarding component classification was the sizing of the dataset. Because for each component subimage we perform matching against every component in the dataset, we expect two outcomes: 1. The larger the dataset, the more likely it is to find a better match for a component subimage, thus the higher classification accuracy; 2. The larger the dataset, the longer it takes to perform matching against every component in the dataset. Both these expectations are supported as seen by testing results in Fig. 4 and Fig. 5. Because of the latency and application size requirement, the size of the dataset has an upper bound. Unfortunately, as we did not have enough time to explore what this maximal size is, we only used an 80 component-large dataset. We predict that before we hit the latency requirement bound, we would reach maximum application size as each additional dataset component equates to only around 2ms in increased computation latency.

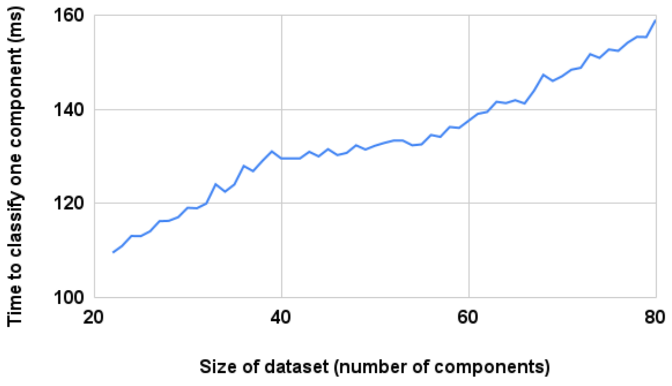

Fig. 4. Change in latency to classify one component as the size of the dataset is increased

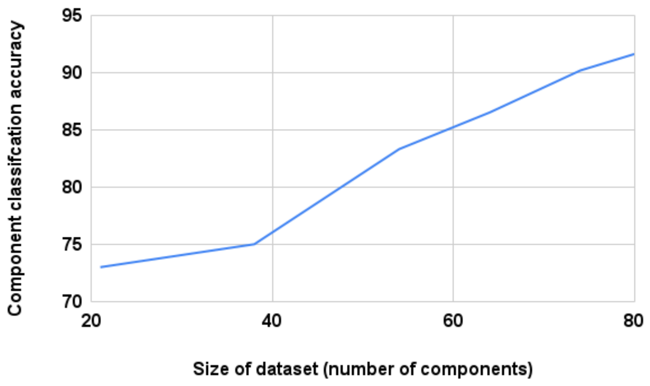

Fig. 5. Change in individual component classification accuracy as the size of the dataset is increased

J. Image Preprocessing

Before feeding images into our computer vision algorithms, we want to perform preprocessing to better isolate the user's drawing and get rid of unintended features like shadows and light marks on the paper. Upon receiving the user's image, we decided on first performing thresholding and median blurring.

Thresholding is useful because it generates a binary image, and we can set the threshold to isolate the darker markings that would come from concentrated amounts of pencil lead or ink and ignore the effects of light shadows and accidental markings. Because the first step in our circuit detection is detecting where each node is and nodes are filled-in circles, thresholding will best isolate these nodes. Specifically, we decided on using an adaptive thresholding algorithm. Adaptive thresholding adjusts thresholds accordingly depending on local contrast and differences in illumination instead of applying one threshold for the entire image. This is beneficial for us because we expect user images to not always have the best lighting and have cast shadows on some parts of the image.

After thresholding, we decided on using median blurring over other blurring algorithms such as Gaussian blurring and bilateral blurring. Median blurring is most effective after thresholding for our use because it is the best in removing salt-and-pepper noise and details. After we perform thresholding, it is likely not only the filled-in nodes are left, but also some darker pen/pencil spots for the components as well. Because we want to isolate the nodes, median blurring sees the thin parts of the drawing that correspond to the components as noise (the pepper) and removes them. Additionally, sometimes thresholding will create white spots in the filled-in circles because the darkness of the circle is not uniform. Median blurring will fill in these white spots (the salt) because it sees it as noise, creating completely filled in circles. This way there is no possibility of a circular white spot in the node to also get detected as a circle/node.

Gaussian blurring and bilateral blurring are similar, and both use Gaussian distributions to remove noise, and bilateral blurring uses two separate distributions instead of one. From our testing we found that these two blurring techniques did not perform as expected for our desired image preprocessing. For component subimages, we use Canny edge detection instead of thresholding before feature detection because while both generate binary images, thresholding will more likely eliminate entire parts of component drawings if the pen/pencil lead is too thin and light. Additionally, we care more about the outline of the component as features, so edge detection is the most appropriate for detecting individual components. We found that gaussian and bilateral blurring did not perform well in keeping the edges of the components intact, resulting in the canny edge detection not functioning at all.

K. Circuit Options

We heavily considered whether we wanted the options of circuits we display to the user to only be valid circuits even if they drew an invalid circuit, or if we should display options of circuits that best match the user's drawing, regardless of whether the options are invalid or valid. For reference, an invalid circuit can be one that is not a closed loop, doesn't have any sources of power (no voltage or current sources), or doesn't have any load. We ultimately decided that we should display circuit options that best match the user's drawing, regardless of if the options are invalid or valid.

If we were to only display valid circuits, the user would not be able to know if they don't see their drawn circuit in the options because they drew an invalid circuit, or because the computer vision system didn't work properly. Consider the scenario where the user draws a valid circuit, but the computer vision system does not function properly and gives them options that do not match the user's circuit. The user is unable to tell if their circuit isn't an option because the computer vision didn't work, or if because the circuit they drew is invalid. As our application is designed to be a learning tool, we need to make it clear to the user when their circuit is invalid or valid. By first detecting the user's circuit, regardless of its validity, when the user chooses their circuit, we can ensure that the application and the user are on the same page, and then we can inform the user that their drawn circuit is invalid or not.

L. Value Detection

Rather than having users write down the values for each component next to the component, we elected to have users manually input the values onto the application itself. Although there are many libraries available for text detection, it will be hard for the computer vision algorithm to figure out which value corresponds to which component. This can lead to dissatisfied users because they may have to constantly redraw and take pictures of their circuit until it gets properly identified. As a further extension to our project, we could implement the ability for users to label components with their respective values in the drawing.

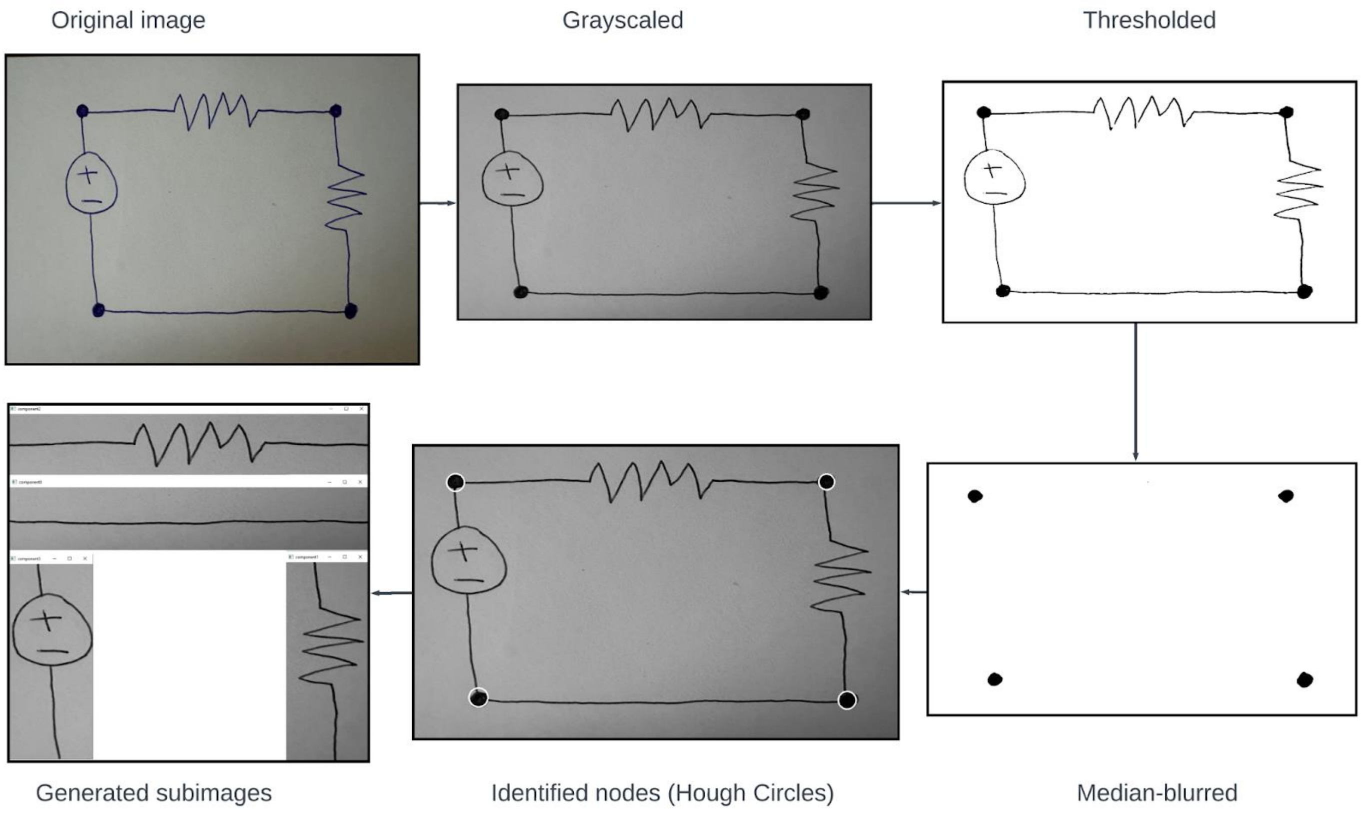

Fig. 6. Node Detection Pipeline

M. Circuit Simulator Algorithm

We initially planned to create a graph data structure that would consist of different components. Each component would have attributes that would be used when running the simulation. After doing research, we realized that this was not the best way to go about simulating circuits. Most circuit simulators like SPICE use modified nodal analysis because it is straightforward and easy to implement on computers. It also gives a system of equations which could potentially be displayed to the user to show them how the circuit was solved.

N. Circuit Simulator Diode Implementation

Modified Nodal Analysis works very well for linear components like current sources, voltage sources and resistors, but it does not inherently work well for nonlinear components like diodes. To implement nonlinear components like diodes, you must make an initial guess for the current through the diode. After making this guess, you use the derivative of the I-V curve for the diode to determine the diode's resistance at this voltage and current. This calculated resistance and guessed current will be used in the modified nodal analysis process. Once the resulting current and node voltages are determined from modified nodal analysis, the resulting current will be incorrect. You must then adjust the initial guess and repeat the process until the resulting current makes sense given the initial guess for the diode's current. Unfortunately, we were unable to implement this feature.

VI. System Implementation

We can separate our implementation into three separate subsystems: the computer vision system, the circuit simulator system, and the mobile application. We can further separate the computer vision system into three separate workflows: parsing the user image, performing individual component detection, and generating circuit data structures.

A. User Image Parsing

Before running feature detection algorithms on individual images of components, we must generate the subimages of the individual components from the user's original image. This process is shown in Fig. 6. Once the user uploads the image of their drawing, we load it as grayscale to perform adaptive thresholding. Adaptive thresholding will create a binary image from the grayscale image. After thresholding, our binary image is entirely white except for the drawing itself, which is black.

Next, we apply a median blur to the binary image. Median blurring is most effective in removing noise from images. The first reason why we do this is because the blurring will remove lighter pen/pencil markers from the image that corresponds to the components themselves. This will allow us to isolate only the filled-in nodes that the user has drawn. Also, the blurring makes the nodes themselves more filled-in and distinct. Likely from the thresholding the less-filled in parts of the node will be removed, leaving a black circle with spots of white. The blurring will fill in this circle, which will remove all the possibly smaller, white circles that could be accidentally identified with Hough Circle Transform.

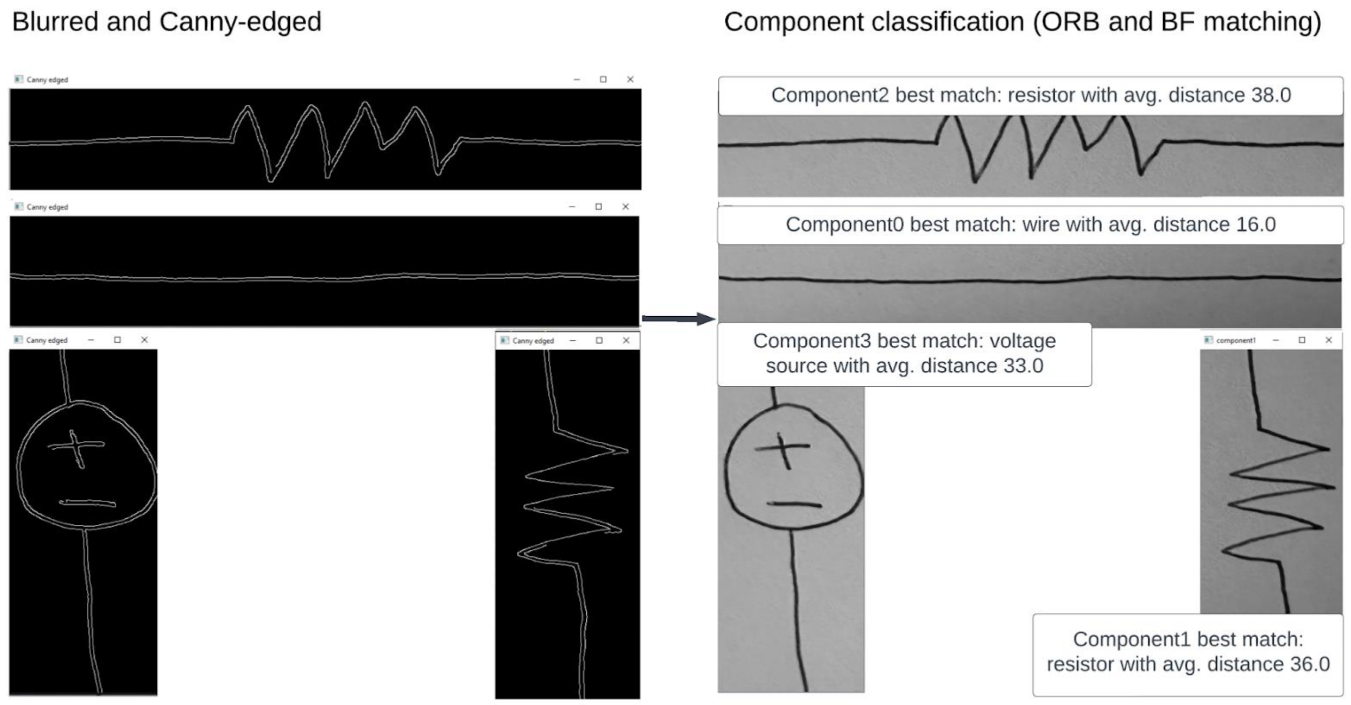
Blurred and Canny-edged
Component classification (ORB and BF matching)
Canny edged
Component2 best match: resistor with avg. distance 38.0
Component0 best match: wire with avg. distance 16.0
Component3 best match: voltage source with avg. distance 33.0
component1
Component1 best match: resistor with avg. distance 36.0

Fig. 7. Canny Edge Detection & Component Classification

Now that we have an image with just black circles representing nodes, we use Hough Circle Transform to identify the location of each circle with x and y pixel coordinates. We know that a component must be between each pair of neighboring nodes, thus we have the coordinates that represent the far ends of the component, and we can use them to extract subimages of each component. The subimages are taken from the original grayscale image.

Originally in our interim demonstration, the bounds of the subimages were hard coded to be certain dimensions. Our final idea for generating the subimages is to first consider a straight line between the given pair of nodes, and then iterate left/downwards and right/upwards until we no longer detect any black pixels. This allowed for adaptively and consistently creating the subimages such that they always included the entirety of the component.

B. Individual Component Classification

Before doing feature detection, we first perform median blurring, denoising, adaptive thresholding, and median blurring again. This is a combination of preprocessing that we have tested that best works for these individual component subimages. We then perform Canny edge detection, which will isolate the edges of the drawing and black out everything else, as seen in Fig. 7. For feature detection we use the ORB to generate keypoints, and then generate BRIEF descriptors from those keypoints. The descriptors are binary feature vectors that represent all the important features in the subimage.

Using these BRIEF descriptors, we can perform brute force matching with BRIEF descriptors we have stored in the application's data storage. These stored descriptors have associated component names with them, such as "resistor", or "voltage source". We perform the brute-force matching with Hamming distance to quantify the accuracy of each match of features; the lower the distance, the better the match. Taking an average of the distances across all matches, we now have a score we can use to quantify the similarity between a subimage and a dataset image. Doing this with each subimage and each set of BRIEF descriptors in storage, we can rank what the best component match is for each subimage. See Fig. 7 as an example of the Canny edge detection and classifying the best match for subimages.

As previously mentioned, we do not perform feature detection and matching on components that are wires. In order to determine whether a component is a wire, we run Hough Lines transform on the edge-detected image, and then find the bottom/leftmost and top/rightmost coordinates of all the line segments. If the difference between these coordinates is below a certain threshold (the component is thin enough), then we classify the component as a wire and skip the feature detection and matching. The reason we use Hough Lines transform here and not just attempting to find the lowest and highest white pixels is because noise that may not have been already removed would produce incorrect results.

C. Circuit Data Structure Generation

From parsing the user image and detecting individual components, we now have identified nodes and the component types that represent edges between each node. From this, we want to generate a graph-like data structure so that the frontend of the mobile application and the circuit simulator can easily reconstruct the circuit. This structure will be a modified netlist, where each component is represented by three values: an (x, y) node coordinate corresponding to the negative terminal node, an (x, y) node coordinate corresponding to the positive terminal node, and a string corresponding to the component type (ex: "resistor").

Originally a regular netlist was going to be generated at this point, where nodes were simply identified using an index, such as "N1" for node one, and "N2" for node two. Because a netlist only accounts for how components are connected, a circuit with the exact same components but rotated clockwise is represented by the same netlist of the not rotated version. For users, it may be confusing why the voltage source they drew is on the top of the circuit instead of on the left. Thus, it would be better if we could preserve the way that the user drew the circuit. By sending the node coordinates, which provides absolute positioning values, we can preserve the orientation that the circuit was drawn.

Because we want to generate the five most matching circuits, we must generate potential circuits and scores associated with each circuit. For each individual component, we consider the three best component types, and then generate every combination of these components. Because our maximum supported component limit is eight, the maximum number of combinations we generate would be $3^8 = 6561$ different circuits. The score is simply a summation of the Hamming distance scores that are already associated with each component classification, so the lower the total score, the more confident the component combination is:

$$score = \sum(individual\ component\ distances)$$

D. Dataset

Our dataset is not a collection of images of individual components, but a YML file which contains a dictionary format of components: [component type]: [BRIEF descriptors]. Because BRIEF descriptors are representations of all the important features in an image, we save on file storage size and computation latency by storing the descriptors and not the actual images of dataset components. The total file size for the dataset is 4.3 MB, corresponding to 80 component images.

The breakdown of the dataset is as follows:

- Eight light bulbs (four horizontal, four vertical)
- 16 switches (eight horizontal, eight vertical)
- Eight resistors (four horizontal, four vertical)
- 24 current sources (six left-facing, six right-facing, six down-facing, six up-facing)
- 24 voltage sources (six left-facing, six right-facing, six down-facing, six up-facing)

Because all light bulbs and resistors are drawn identically when they are horizontal or vertical, we only need eight for each orientation. For switches, because there are four different ways a switch can be drawn horizontally and four different ways a switch can be drawn vertically, having 16 allows for two different images for each way a switch can be drawn. For current and voltage sources, because we consider each orientation of these components as separate classifications, we have six for each of the four possible orientations. As a reminder, we no longer needed to have wires in our dataset because we determine them by evaluating the width of the component. Also, as previously mentioned in the design trade studies, we would have liked to have a bigger dataset for higher classification accuracy, but this relative number of each component type would stay the same as we would increase the dataset size.

E. Circuit Collapsing

The data structure given to the circuit simulator contains the names of the components, their node coordinates, and their associated values. This list of components consists of wires which need to be removed for simulation purposes. Once the simulator receives the data structure from the frontend, it performs a node collapsing algorithm. It creates a dictionary mapping from coordinate to node numbers. The algorithm will first go through the components and set nodes connected by wires to be considered the same node.

Once this pass has been performed, it will then go through all the node numbers and ensure they are enumerated from 0 in a counterclockwise formation. After this pass, a netlist will be generated using the new nodes instead of the coordinates that were input previously. Furthermore, this new netlist does not contain any wires because the wire connections have been accounted for when creating the node numbers.

F. Circuit Simulator

We used modified nodal analysis to solve and simulate circuits[2]. The simulator takes in the above-mentioned netlist-like structure as input and generates matrices that will be used to solve a system of equations describing the circuit. Modified nodal analysis tries to solve the equation:

$$Ax = z$$

in which A describes the connections and conductance of the elements of the circuit, x describes the unknown values that we are trying to solve, and z describes the current and voltage sources. The A matrix is of size $(v + n)(v + n)$, where v is the number of voltage sources and n is the number of nodes. It is constructed of four smaller matrices:

$$A = \begin{bmatrix} G & B \\ C & D \end{bmatrix}$$

The G matrix details the conductance of the circuit elements, the B and C matrices detail the connections of the voltage sources, and the D matrix will always be a zero matrix if the circuit only contains independent voltage sources. The simulator will generate the required matrices and solve the equation above for the x matrix, which contains the unknown voltages and currents through independent voltage sources. Once the node voltages have been determined, Ohm's Law is used to calculate the current going through each individual component. Once these values are determined, they are added to a formatted string with all the listed values and sent to the frontend for rendering. Although modified nodal analysis generates a larger system of equations than other algorithms such as traditional nodal analysis or mesh analysis, modified nodal analysis is easier to implement algorithmically on a computer system.

G. *Mobile Application*

The mobile application is written in Swift. The home screen consists of a functionality description with a list of supported components, two tips, one for drawing circuits and one for taking a picture of the drawing, and an example image to help the user upload their circuit correctly. Once the user clicks the "Upload Circuit" button they are taken to a page where they are given the option to either upload an image from their phone gallery or take a picture. Once the user uploads their image the image is displayed in the center of the screen and a next button appears on the top right of the screen.

The next button is linked to a wrapper function in Objective C++ that allows the computer vision system to read the image uploaded. When the computer vision algorithms finish, five netlists will be generated. The wrapper function converts the five net lists returned from the computer vision into a compatible type for Swift. Based on these netlists, we will craft and display circuits.

For this we will utilize reference images stored in the application's file data for each component and connect them with lines for proper wiring and circles for distinction of nodes. Each circuit will be displayed on a swipe-able page. The circuits are ordered from the first page being the highest confidence match to the last page being the fifth highest in confidence. The user selects one of the circuits by tapping on one of the pages that display a circuit. The user will then be taken to a page to input all the values of each component drawn. This includes resistors, lightbulbs, voltage sources, and current sources. Inputs are taken the SI units, amps, volts, and ohms.

Once all the components have inputted values, an "Analyze Circuit" button appears above the circuit representation. Once the user has clicked the button, the contents of the completed circuit will be sent to the circuit simulator via another Objective C++ wrapper, and on the final page the results from the simulator are displayed under a tab that says, "Show Results". The analysis includes voltages at every node and current through every component. At any point the user can refresh the app and go back to the home page to start the process over.

VII. Test, Verification and Validation

A. *Testing for Individual Component Classification Accuracy*

To test the individual component classification, we compiled a total of 181 component subimages. In order to generate these component subimages, we used our existing subimage generation code such that these subimages used in testing would match the expected generated component subimages format from full circuits. This testing set was drawn by both members of our team and members of our test group.

The breakdown of the components in the testing set is as follows:

- 62 current sources (13 downwards, 18 leftwards, 18 rightwards, 13 upwards)
- 12 light bulbs
- 13 resistors
- 40 switches
- 46 voltage sources (9 downwards, 13 leftwards, 15 rightwards, 11 upwards)
- 6 wires

There are more components in the testing set for those that had a harder time being classified properly (switches, current sources) in development. Notably light bulbs, resistors, and wires had extremely high correct classification rates. The result from testing was that 166 of the 181 test subimages (**91.7%**) were correctly classified, which meets our individual component classification accuracy use case requirement. A classification was deemed correct if the best of the three component matches, we output was the correct component. To reiterate, this accuracy value considers current sources and voltage sources of different orientations as their own, unique components. The raw score for the number of components that were properly identified (not considering orientation) was 176 of the 181 test subimages (97.2%).

B. *Testing for Full Circuit Classification Accuracy*

To test the full circuit classification, we compiled a total of 52 images of circuits drawn by both our test group and members of our team. The circuits were made up of a range of four to seven total components. Image capturing of the drawings were done in multiple different settings (bedroom, living room, dining room, classroom, open school environment) in order to capture varying appropriate and reasonable lighting settings.

Notably, all the tested settings provided the same type of effect on the captured image, which was having a slight shadow from the phone used to take the images. Of the 52 images, 43 were classified correctly (**82.7%**). A classification was deemed correct if within three different captures of the circuit image (without modifying the drawing itself), one of the five displayed circuit options was correct. We determined that this was a fair evaluation of correct classification because of how bad photos captured affected circuit classification (poor lighting, drawing out of focus, etc.). If the circuit could be correctly classified without the drawing itself being changed, it felt fair that the classification should count as correct.

One limitation of our testing was factoring bad drawing skills, which is hard to quantify. From our testing we felt that we had representation from a reasonable range of drawing skills, but our live demonstration proved otherwise. Another interesting result is that we had a higher classification accuracy at the time of our final presentation (~85%). At the time this was a reasonable result because it was just higher than the individual component classification accuracy (~83%), which made sense because all the incorrect circuits came from incorrectly identified components. Because we display five different circuit options, we expect that the circuit classification accuracy is slightly higher than the individual component accuracy.

As we did more testing from 28 to 52 images, we got some misidentified circuits because of failure to recognize the correct number and location of nodes. This is why our circuit classification accuracy went down from our final presentation, and why it is no longer closely related to the individual component classification accuracy.

C. Circuit Simulator Testing

The circuit simulator was tested using a program that randomly generated netlists with a maximum of eight components. The function would only create valid netlists with components consisting of current sources, voltage sources, switches, resistors, and lightbulbs. After creating the netlist, we would simulate the circuit on the circuit simulator using our own simulator. The results of the simulation from our simulator would then be compared to the results from running a simulation of the circuit in Altium Designer.

Fifty circuits of varying structures, sizes, and components were all tested using this pipeline, and all fifty were simulated correctly. This means that given a netlist, our simulator returns the correct voltage at every node and current through each component **100%** of the time, meeting our use case requirement of perfect circuit simulator accuracy.

E. Mobile Application Testing

To test the mobile application, we gave a test group of 7 individuals, ages 12 to 14, a usability survey as well as a series of tasks to test the full pipeline of the system. The tasks included going back and forth between pages in the application, drawing and uploading an image of a circuit, inputting values for components, and analyzing the results from the simulator.

After completing all these tasks, the users were asked these questions:

- "How easy was it to upload your circuits?"
- "How useful were the tips on the home screen when drawing your circuit?"
- "How clear were the schematics of all the circuits displayed?"
- "Were the headers of the page useful when asked to do complete some task?"
- "Do you think adding more tips and headers would make the app clearer?"
- "Was it clear how to input values?"

After rounding the average score for each question, the average score of the survey was **78.5%**. The main issues with the UI came from the inputting values page. The question "Was it clear how to input values" got the lowest average score of a 6/10. Due to Swift's code structure, there were too many issues when trying to display values next to the components that the user input values for. Instead, we opted for adding tags to the end of component names, ordered from top to bottom and left to right if there were repeated components. For example, if there were two resistors in parallel the leftmost would be "resistor_0" and the rightmost would be "resistor_1." Similarly, if there two resistors on top of one another the topmost would be "resistor_0" and the bottommost would be "resistor_1." However, doing this caused confusion amongst our test group.

When there were many repeated components and a larger circuit, most users lost track of which component they inputted a value for. When surveying, a lot of clarification was asked for when inputting values, however once explained, the majority of the individuals understood the relationship between the tag and the components. All other survey questions had an average score of 7 or above and the users did not report any other hardships when using the application. However, we did proceed to add an example image on the homepage as more than half of the individuals wanted additional tips on drawing circuits.

F. Complete Integration Testing

When testing the full pipeline, we first tested the image uploading functionality. A picture from the phone application would be uploaded straight from the in-app camera feature or from the user's photo library, and this picture would need to be read into data by the computer vision system. After fully integrating this was functioning 100% of the time. We were able to get this success rate by reading file paths from the Swift codebase and feeding it as a parameter to the computer vision system, which then created a dedicated image data structure.

The second test was regarding netlist parsing to create the circuit UI. This was tested by manually checking the display on the phone application and cross-referencing it with the drawn picture and the netlists returned. With a circuit that met our requirements of a maximum of eight components, the UI was able to display the five circuits returned from the computer vision system 100% of the time.

Lastly, we tested the net list received by the circuit simulator net list expected from the values inputted by the user. Because the expected accuracy for the circuit simulator was 100%, this portion of testing was crucial for the entire system. After cross-referencing the output of the circuit simulator with the user's input, we were able to achieve 100% accuracy as well.

G. Application Size and Latency

The application had a final size of **32 MB**, which we were able to verify by looking at the application's storage specifications in the settings of an iPhone. The computer vision latency was **~3.2 seconds**. As a reminder, this is the time it takes from when the user submits their image of their drawn circuit to when the five circuit options are displayed on their screen. This latency was measured by logging timestamps when the circuit classification function exposed to the frontend of the application was called to when the function returned. This was done during the circuit classification accuracy tests, which spanned 52 different circuit images. As mentioned previously, we would have liked to increase our dataset to use more of the remaining 68 MB we allowed the application to have, as well as use the 1.8 seconds remaining of latency to improve the classification accuracies. Notable to mention is that the longest task to perform was the adaptive thresholding, which is a necessary algorithm to use to best detect the circuit nodes. For future improvements, we would definitely want to explore increasing the dataset size as each addition to the dataset is approximately equal to two milliseconds of added latency.

VIII. Project Management

A. Schedule

Fig. 8 is the Gantt Chart with the schedule of work as well as each team member's weekly tasks.

B. Team Member Responsibilities

Each member has taken on one of the main three subsystems of the project.

- Stephen Dai: Computer Vision
- Devan Grover: Circuit Simulator
- Jaden D'Abreo: iPhone Application

In addition to the ownership of their respective subsystems, Jaden and Devan created the bridging files and integration that allowed the computer vision code and circuit simulator to work seamlessly with the application. Each member was also in charge of the testing for each of their respective subsystems.

C. Bill of Materials and Budget

All necessary materials will be items the user must have themselves. These materials consist of an iPhone or iPad, a writing utensil, and paper.

D. Risk Management

We initially met issues when dealing with components that act differently based on their orientation. We initially wanted to use ORB's rBRIEF descriptors because they are rotationally invariant – this meant that our algorithm would not have issues detecting components if they were in different orientations. We realized however, that by doing so we were unable to detect the direction of voltage and current sources. As a result, we had to use BRIEF descriptors – even though this meant having to take images in different directions for our dataset, we were able to achieve a higher accuracy for voltage sources and current sources.

Furthermore, we initially had issues being behind schedule due to the infrastructure change of our project from a web application to an iOS application. Initially, we developed our code in Python and created an initial version of a web app. We realized that an application would be better because it would be more accessible to users and easier to use. As a result, we had to change our codebase from Python to Swift, Objective-C++, and C++. To mitigate the risk from this switch, we all did work over Fall and Thanksgiving breaks to catch up.

There were lots of integration issues that we encountered when trying to build the C++ code for iOS. Since we started this step of integration early, we were able to spend lots of time on this issue and get it fixed before we started our final integration towards the end of our project.

Due to there being too many issues when displaying the input and final values next to the correct component, we decided to modify the UI of the application. Instead of displaying the values we opted to add tags at the end of components to refer to components in the order top to bottom left to right. Similarly, we display final values in a string at the bottom in the bottom of the page with values at each node and current through each component, matching the tags on the input value page. While this negatively affected the usability of the application, it kept the application functional.

IX. Ethical Issues

When designing our product, we concluded that there were three main ethical concerns that come with our system: public health, public safety, and public welfare. As our system is a phone application, its use would lead to an increase in screen time for our primary user group, a public health concern. One common problem with extensive phone usage is digital eye strain, where long term damage to the eyes can be caused from prolonged phone usage. This is especially concerning for kids whose eyes are still developing and are more prone to long term damage. Our application is designed to not need extensive use because the users do not draw their circuits in the app themselves. Only until the user has finished drawing their circuit on paper do they then start to interact with our application. In addition to this, the runtime for the total pipeline of the application should take no longer than 2 minutes, and around half this time would the user be interacting with the phone application.

We require access to the camera and photos for the user to either take or upload a photo of their circuit drawing. This is a potential security and privacy risk for the user as they entrust us with their photos, a public safety concern. Privacy and security have become a significant aspect of social and global concern. Most relevant to our application is how we handle the user's photo. We need to ensure to the user that the image is stored securely and the information we gather about the image is solely related to generating circuit information and nothing else. Our project design targets this area well in two ways. The first is that our application operates completely offline and only uses local storage. In order for there to be a security breach, the user's phone must actually be hacked, which means they would already have access to the user's photos. The second is regarding how we would store photos. As we wanted to keep the application lightweight, we do not store any images that the user uploads. The phone application stores the file path when the user uploads an image, and once they are done with it, either analyzing it or replacing it, that image gets wiped from local storage. By keeping the application offline and lightweight, we mitigate all potential privacy concerns.

Lastly, a misapplication of our project is students using it to do their homework for them, a public welfare concern. Cheating seriously affects public welfare, especially for engineers, as cheating means that students do not properly learn material and fundamentals that they will need to apply in the real world. The classic example we hear is if you would use a bridge built by an engineer that cheated in school. An important design choice that we made to combat this is that when we display the calculated current and voltage values, we only display what they are, and not any of the work that shows the calculations done to get those values. Because students almost always need to show work for them to get credit on their assignments, they will need to calculate the values themselves and then they can check their answers, which is fine because they did the work anyways.

X. Related Work

There are numerous online circuit simulators such as LTSpice. But all these simulators require building the circuit in the application itself and are not free of cost to users. Regarding the computer vision aspect, there exists research that was conducted to contrast the use of SIFT and ORB with drawn electrical components[7], but these components were digitally and not hand drawn, and they only classified individual components and not entire circuits. Research has been conducted to reconstruct full circuits from drawn circuits[6], but this uses digitally drawn circuits by providing an online GUI, and it also uses a CNN. Neither of these tools are available to the public.

XI. Summary

The Circuit Simulpaper system is designed to allow a younger audience to learn basic circuit functionality through an iPhone application coupled with drawing. It serves as an educational tool that can be accessible for all. The system requires three items, items that most households already contain, to produce a desired circuit analysis. Foreseeable challenges include circuit detection accuracy, integration between the phone application and the computer vision aspect, and user testing. We are confident that through our thorough design and integration plans we will be able to overcome all challenges and provide a high-quality educational tool.

Over the course of this project, we encountered many issues with each separate subsystem that we were able to help mitigate by changing aspects of our design. We initially started to create a Python web application to house our project, but after a month we realized that this was not ideal. Creating a phone app would make our app easier to use and have a wide audience of middle schoolers. We also encountered issues with our computer vision algorithm not being able to detect the direction of components – with our initial classification algorithm using rBRIEF descriptors, we had a correct direction classification accuracy of ~50%. To mitigate this issue and properly detect the direction of voltage and current sources, we switched our computer vision algorithm to use BRIEF descriptors.

We also encountered issues with the phone application due to none of our group members not having any prior experience with Swift or Objective-C++. There was a learning curve that we had to initially overcome to start development. This learning curve was hard to initially overcome because we changed our application's structure from a web application to an iOS application in October, so we had to learn iOS app development very quickly to stay on schedule.

Integration was also much harder than we initially expected, and we are very thankful that we tried to initially integrate the simulator with the application very early on. This allowed us to setup the infrastructure needed to run C++ code on our iOS application, which helped when importing the computer vision code later on in the integration process.

Trying to create a project with many new languages and technologies that we had never used before (CV, Swift, Objective-C++, C++) was a very had process because we had to learn the new languages and technologies while actively working on the project. Requiring us to do this did allow us to learn at a very high rate, which was extremely gratifying.

Glossary of Acronyms

BRIEF	Binary Robust Independent Elementary Features
CNN	Convolutional Neural Network
DC	Direct Current
FAST	Features from Accelerated Segment Test
GUI	Graphical User Interface
I-V	current-Voltage
LED	Light Emitting Diode
MB	Megabytes
ms	Milliseconds
ORB	Oriented FAST and Rotated BRIEF
rBRIEF	Rotation-invariant Binary Robust Independent Elementary Features
SI	International System of Units
SIFT	Scale-Invariant Feature Transform
SPICE	Simulation Program with Integrated Circuit Emphasis
UI	User Interface
YML	YAML (programing language)

References

[1] Anderson, M. (2018, May 31). *Teens, Social Media and Technology 2018*. Pew Research Center: Internet, Science & Tech. https://www.pewresearch.org/internet/2018/05/31/teens-social-media technology-2018/

[2] Cheever, E. (n.d.). *Analysis of Circuits*. Analysis of circuits. https://lpsa.swarthmore.edu/Systems/Electrical/mna/MNA1.html

[3] Heideman, P. D., Flores, K. A., Sevier, L. M., & Trouton, K. E. (2017). *Effectiveness and adoption of a drawing-to-learn study tool for recall and problem solving: Minute sketches with folded lists*. CBE life sciences education. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5459246/

[4] Mejía, D. (2023, April 13). Four out of five households with children owned tablets. Census.gov. https://www.census.gov/library/stories/2023/04/tablets-more-common-in-households-with-children.html#:~:text=The%20share%20jumped%20to%2075,17%20years%20old%20%E2%80%94%20owned%20tablets

[5] Miller, C. (2023, March 13). *When should you get your kid a phone?*. Child Mind Institute. https://childmind.org/article/when-should-you-get-your-kid-a-phone/

[6] Keerthi Priya, A., Gaganashree, N., Hemalatha, K. N., Chembeti, J. S., Kavitha, T. (2022). AI-based online hand drawn engineering symbol classification and recognition. Lecture Notes in Networks and Systems, 195–204. https://doi.org/10.1007/978-981-16-8512-5_22

[7] Pavithra, S., Shreyashwini, N. K., Bhavana, H. S., Nikhitha, G., &amp; Kavitha, T. (2023). Hand-drawn electronic component recognition using Orb. Procedia Computer Science, 218, 504–513. https://doi.org/10.1016/j.procs.2023.01.032

[8] *Swift*. A powerful open language that lets everyone build amazing apps. Apple. (n.d.). https://www.apple.com/in/swift/

[9] Haiyan Wang,Tianhong Pan, efei,Anhui and Jiangsu.(2020) "Hand-drawn electronic component recognition using deep learning algorithm"China, Int J:Computer Application in Technology

[10] Dewangan, A. and A. Dhole,(2018) "KNN based hand drawn electrical circuit recognition" International Journal for Research in Applied Science & Engineering Technology:p. 1111-1115.

[11] An, D. (2018, February). *Find out how you stack up to new industry benchmarks for mobile page speed*. Google. https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/

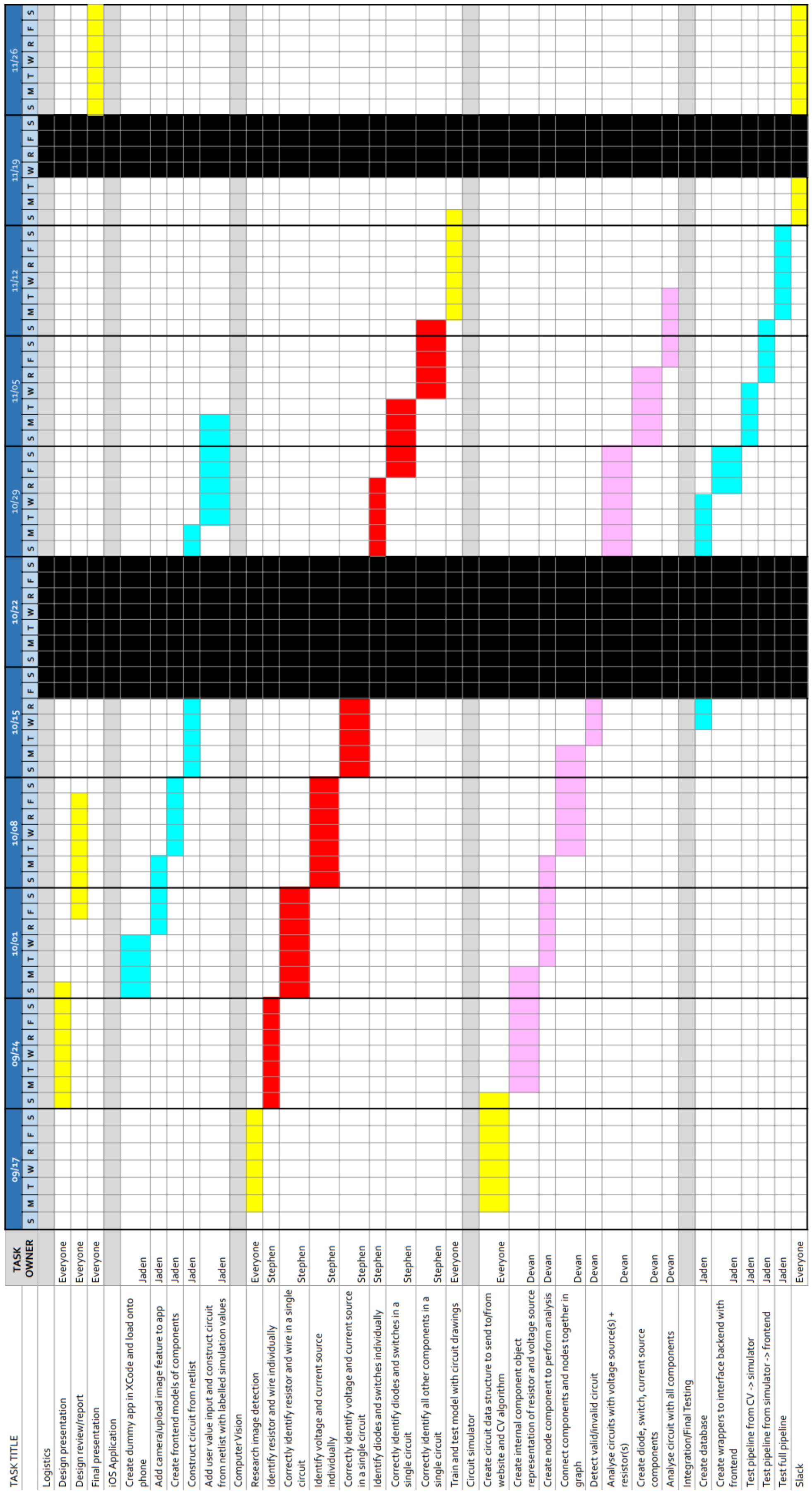

Fig. 8. Gantt Chart Schedule