Transmission Line Fault Analysis

The SMT Method

By

Mitchell Alexander Lautigar

Dr. Abdelrahman Karrar

Associate Professor

(Chair)

Dr. Abdul R. Ofoli

UC Foundation Associate Professor

(Committee Member)Dr. Raga Ahmed

Assistant Professor

(Committee Member)

THE SQUARE MEAN TEST METHOD; AN RMS ALTERNATIVE

By

Mitchell Alexander Lautigar

A thesis submitted to the Faculty of the University of

Tennessee at Chattanooga in Partial

Fulfillment of the Requirements of the Degree

Of Master of Engineering

The University of Tennessee at Chattanooga

Chattanooga, Tennessee

May 2019

ABSTRACT

Analyzing commonplace faults on power lines in a smooth and effective manner via a program has one big challenge, knowing how the computer sees the data to understand how it then outputs the information. The SMT method expands on the RMS calculation to end up with a result with a much higher accuracy.

DEDICATION

Special thanks to my family and my friends who have supported me through my many late nights spent completing this project. This project has been a labor intensive effort that was only possible because of the support structure that I have.

ACKNOWLEDGEMENTS

Special thanks to Dr. Karrar and Mr. Robert Hay. I went to Dr. Karrar a year ago just wanting a second opinion on an idea, and since then, they have been an open book any time I had hit a wall on this project. I thank you for your support as well as am very appreciative of you challenging me to keep pushing the envelope. Mr. Hay, I want to thank you for giving me this opportunity to do research that is practical and applied to what I want to do in life. I never would have expected this project to become my thesis, but I'm glad it worked out the way it did.

TABLE OF CONTENTS

ABSTRACT iii

DEDICATION v

ACKNOWLEDGEMENTS vi

LIST OF TABLES ix

LIST OF FIGURES ix

LIST OF SYMBOLS ix

LIST OF ABBREVIATIONS x

CHAPTER

1. INTRODUCTION 1

Objective 1

Tested Methods 1

Significance 3

1. HALF CYLE VS. QUARTER CYCLE 5

Half cycle Basics 5

Wavering Error 8

Quarter Cycle Basics 10

Half Cycle vs. Quarter Cycle 11

1. EIGHTH CYCLE ANALYSIS AND POLAR PLOTS 13

Why use eighth cycle 13

Eighth cycle compared to Quarter cycle 14

Significance of eighth cycle 14

Polar plots 15

1. RESULTS 16

Conclusion 16

Future Research 16

APPENDIX

1. FORMULAS 17
2. TABLES 22
3. MATLAB CODE 25

VITA 48

LIST OF TABLES, FIGURES, AND SYMBOLS

TABLES

1.1 -- Output Array 1

2.1 -- Statistical Values for Quarter Cycle Evaluation 23

2.2A -- Expected Percent of files seen for normal distribution 23

2.2B -- Actual Percent of Files seen using Quarter Cycle Analysis 24

FIGURES

2.1 -- Graphical example of a fault 6

2.2 -- Phases A,B, and C Faulted for Figure 2.1 7

2.3 -- RMS Calculation for half-cycle of Figure 2.1 8

2.4 -- Graphical Representation of wavering error, half cycle 9

2.5 -- RMS calculation for quarter-cycle of Figure 2.1 10

2.6 -- Graphical Representation of wavering error, quarter cycle 11

3.1 -- "The Ant Fault" 13

3.2 -- RMS Calculcation for Eighth cycle of Figure 2.1 14

SYMBOLS

Wavering Error 8

LIST OF ABBREVIATIONS & TERMINOLOGY

* RMS: Root Mean Square
* While Loop: A programming loop that does what it's told until a condition is met
* For loop: A programming loop that does what it's told for a specific amount of times.
* File: A file of information given by EPB in the form of a comtrade file.
* LL: Line to Line Fault
* LLL: Line to Line to Line Fault
* LG: Line to Ground Fault
* SMT: Square Mean Test
* Evolving Fault: Fault that changes one class of fault to another.
* Simple Fault: A fault that starts and ends in a single file or in less than half a second.
* Complicated Fault: Fault that spans multiple files but is still the same fault. i.e. A LG fault that spans 2 files is a complicated fault.
* Intelliruptor: The recording device used by EPB to record data.
* Source Side Voltage: The side of the Intelliruptor that points upstream towards the generator.
* Load side voltage: The side of the Intelliruptor that points downstream to the consumer.
* Pulse Close: An automated attempt to restore power automatically by the system.
* Matlab: A software that was used to automate the SMT classification method.
* Array: A matrix of values.

CHAPTER 1

INTRODUCTION

The objective of this project was to automate classification for a database of faults. This is done in such a way that the program would report all faults in each file and output each fault into a simplified array. The array design is shown below for reference as Table 1.1.

Table 1.1 -- Output Array

This array would then be placed into a text file with other fault arrays and then be checked by Matlab so as to group faults by timestamps or by device proximities. This allows a simplified report to be sent to an engineer that shows which devices faulted at a similar time and which devices are experiencing the same fault.

There were multiple methods attempted for this project. Each of them has their advantages and disadvantages, but in the effort to simply, they are listed below with their pro's and con's and whether they are used.

1. Findsignal() command
   1. Pro: Incredibly accurate if you had a database for what every single fault would look like
   2. Con: The faults can take on an infinite amount of shapes therefore making this command inaccurate.
   3. Conclusion: This command was ultimately passed over since the faults vary too much from file to file.
2. Findchangepts() command
   1. Pro: Has the ability to see when there's a change in the signal.
   2. Con: To effectively use this command, one would have to stick this in a while loop and continue running this command until no change was detected.
   3. Conclusion: By personal preference, I'm against putting something in a while loop. This is because it may never exit the while loop therefore stopping the program from doing what it needs to do. Ultimately, this was why this command was passed over.
3. Half-Cycle Analysis -- RMS for each grouping of 32 samples
   1. Pro: The magnitude error was at an unprecedented low of <1% error.
   2. Con: The start time and end time was inaccurate and had error that could vary up to 16% error.
   3. Conclusion: Using this to calculate the magnitude of the faults is by far the most efficient use of this method. However, the start and end time assumed left much to be desired. This method was evaluated, but instead used as a checking point for the Quarter-Cycle method.
4. Quarter-Cycle Analysis -- RMS expansion from the half cycle analysis and evaluates by groupings of 16 samples
   1. Pro: Ability to see the start time and end time on a higher tier of accuracy was established as it greatly diminished the Wavering Error.
   2. Con: The computing power to do this on a mass level is roughly 4x the amount needed to do half-cycle analysis.
   3. Conclusion: While the computing power for this is exponentially larger than the half cycle analysis, the accuracy and ability to easily step it back to the half-cycle analysis matches the desired accuracy that fits the desired goals of this project. This is the default method for this project.
5. Eighth-Cycle analysis -- RMS expansion from the Quarter cycle analysis and evaluates by groupings of 8 samples.
   1. Pro: The accuracy for start time and end time is incredibly powerful, and leaves room to be improved upon to eliminate the faults that cannot be easily classified. This method also lets you verify the quarter cycle analysis calculations on a deeper level.
   2. Cons: The computing power for this is 16x larger than the half cycle analysis, and has a good percentage of error for faults that happen in less than 12 samples.
   3. Conclusion: If this method could be perfected, then this would be the desired method for this project. However, there is still a lot of research to be done to perfect this method.

To appreciate the significance of this study, imagine the following scenario. EPB, the local utility company, has many faults that happen throughout every day. Most of these faults are simple faults, or in other terms resolved within a single file, therefore ranking much lower in importance. The faults that do need to be seen investigated are the evolving or complicated faults as these are often much more technical in nature. From the sample size of reviewed files that I have, roughly 75% of all files are simple faults and in a database of over three hundred thousand faults, being able to instantly resolve 75% of those faults saves a lot of time, money, and energy that can be used elsewhere.

Chapter 2

Half/Quarter Cycle Analysis

Though there are a multitude of methods that have been mentioned, the two main methods that hold any merit are the half cycle and quarter cycle analysis methods. Both methods stem from the RMS formula and both have varying degrees of success and controllability.

Starting with half-cycle analysis, the following formula is just the RMS formula in its simplest form:

Formula 2.1

where 'n' is 32 samples since half-cycle is what's being discussed. The simpler version of this calculation is to take the expected line value and plug it into the alternative version of the equation found below:

Formula 2.1

By doing this, we are quickly able to acquire what the RMS value should be. To put this to a realistic picture, a sample file was chosen at random to be viewed for all of the following evaluation methods. The graph for the file is defined as follows:

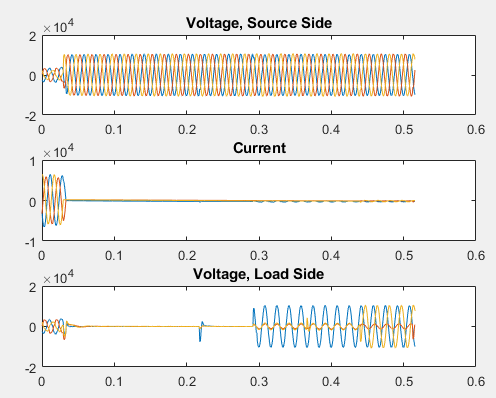


Figure 2.1 Graph showing Source/Load Voltage and Current of a fault.

As it can be seen by looking at the current at the start of the file, we see that all three phases are faulting. The source side voltage is faulting only at the beginning of the file and then resumes its normal state. The current goes to 0 after the fault ends and therefore causes the voltage on the load side of this device to fault out and attempt a pulse close. This pulse close is denoted by the blue pulse we see on the "Voltage, Load Side" subplot which then becomes the phases working towards restoration. Let's now look at a closer perspective of this graph, with a focus on the current (Figure 2.2) and compare it to the half-cycle analysis graph (Figure 2.3).

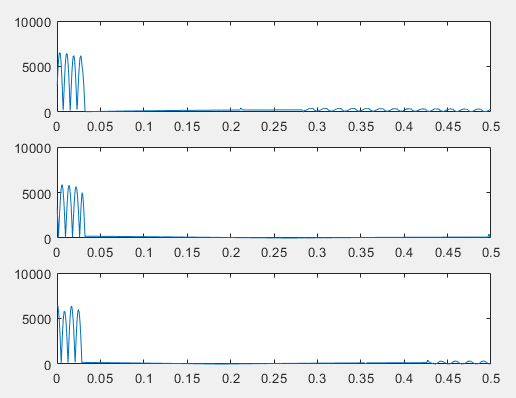


Figure 2.2 A graph that shows the absolute values of the Current for Phases A, B, and C.

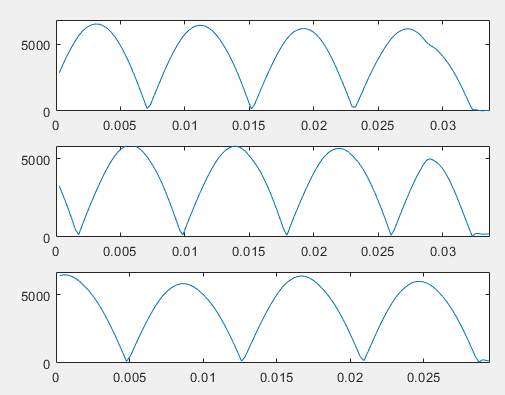


Figure 2.2(zoomed in): shows the zoomed in view of figure 2.2

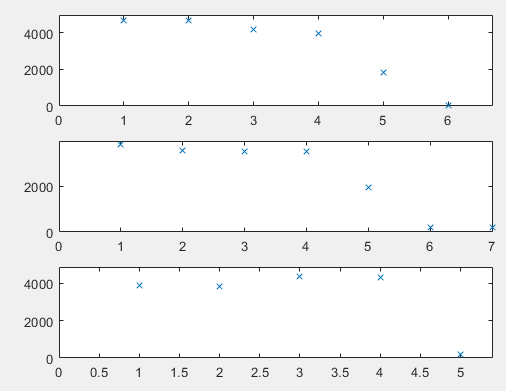


Figure 2.3 Show the RMS calculations for half-cycle.

Observing Figure 2.2, which is the absolute value of the current for each phase, we see in phase A (The top subplot) that there are four peaks. It also can be seen that Phase B and Phase C have 4 peaks. Yet, when we look at Figure 2.3, which is the RMS values of each half cycle, we see that there are 5 RMS points that are well above the normal values for Phase A and B while Phase C has 4 RMS points above the average. At first, this was assumed to be an error in calculation, but since the program is set to read from where the graph first intersects the x-axis at 0 or maxes out, and then count in amounts of 32 samples, we see what is called Wavering Error, or as denoted previously, . Now, wavering error is the error that happens when the fault continues or originates from half of the samples after or prior to where the fault conception was noted to be at. For the half-cycle wave analysis, this wavering error looks like the following on a sine wave:

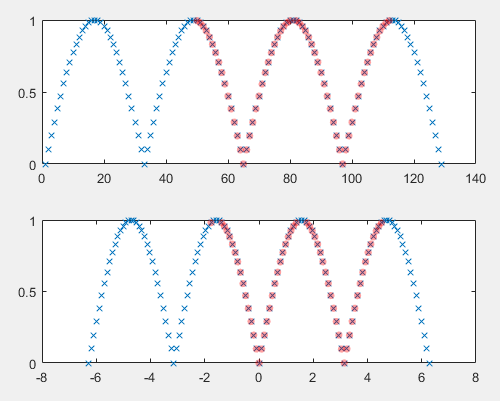


Figure 2.4 Show the Wavering Error for half cycle analysis.

The figure above has two subplots of the same graph where we are simply plotting x vs. the absolute value of y. The following commands in Matlab were used as the input and then simply plotted against each other.

X = -2\*pi: 2\*pi/64:2\*pi;

Y = sin(x);

This example of wavering showing the RMS for half wave evaluation will reflect the fault happening at the third RMS value, although it can actually begin in the latter half of the 2nd RMS value and end in the early half of the 4th RMS value. The formula for wavering error can now be defined as:

; where 'n' is the number of samples per RMS value.

In this case, since we are evaluating in half-cycles, 'n' is 32; this results in the wavering error of samples, or a max error 64 samples of error. With this in mind, looking back at figure 2.2, one can now see why the RMS values show 5 values instead of the previously seen 4 values. How can we reduce this error? In all actuality, this would have the cycle duration equal to 2.5 cycles long when it clearly is not that. This is where Quarter-cycle evaluation comes into play.

Quarter cycle evaluation is an expanded version of half-cycle analysis that theoretically cuts the wavering error in half. What will be seen later on is that the quarter cycle analysis does much more than this, but for now, let it be assumed that the quarter cycle evaluation will cut the wavering error in half. The proof for half-cycle analysis into quarter cycle analysis can be found in Appendix A for reference. Looking at figure 2.5 below and comparing it to figure 2.2, we can see a more accurate picture of what is happening.

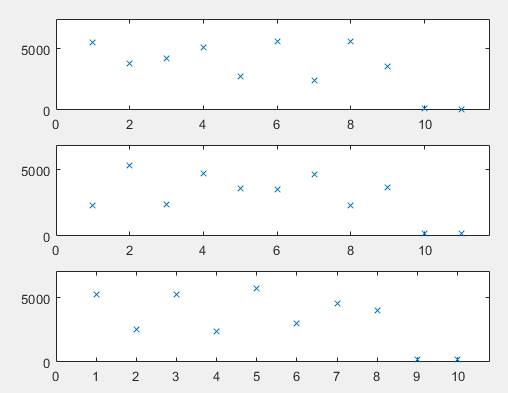


Figure 2.5 Quarter cycle evaluation of the fault shown in Figure 2.2

Since the evaluation is now happening in quarter cycles or 16 samples, we can count on phase A and B a total of 9 RMS values well above the average and phase C has 8 RMS values above the average. Note, every 4 RMS values is a full cycle. Therefore, we see that the cycle duration is 2.25 cycles. Comparing this to figure 2.2, this is much more accurate when you convert this cycle duration into time(formula for this located in Appendix A).

With a direct comparison between quarter cycle and half cycle evaluation now visible, it is time to discuss the differences in wavering error. For quarter cycle analysis, the wavering error on a graph looks like figure 2.6 below.

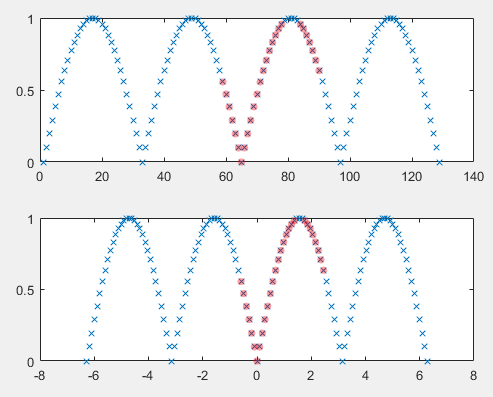


Figure 2.6 Shows wavering error for Quarter Cycle analysis.

Now, what makes the wavering error much more favorable here for quarter cycle analysis, is that in terms of looking at the big picture from the regular RMS calculation, the fault happens within the first or last quarter of the half cycle on either side of the actual fault. This translates to 8 samples of error. When you consider 8 samples, it's arguably negligible. Each file holds 1981 samples in total, and the resulting 8 samples of error translates to roughly 4 milliseconds of error. Obviously, while we'd expect the wavering error to be samples, it would take a very quirky fault to reach the expected wavering error.

I've included a table of statistics (Table 2.1) to evaluate this method on a normal bell curve in appendix B and then went ahead and figured out how many of the files fit within each standard deviation (Table 2.2). These tables will elaborate in more detail why this method is the optimum method to use. However, while this could account for 98% of all files within 3 standard deviations, there was still 2% of files that were non-conforming. This also would aid towards the endeavor of implementing phaser plots to assist engineers in viewing the faults.

CHAPTER 3

EIGHT CYCLE EVALUATION AND POLAR PLOTS

To describe why we need eighth cycle analysis, let us have an example of a fault that happens too quick to be understood by quarter cycle analysis (Figure 3.1).

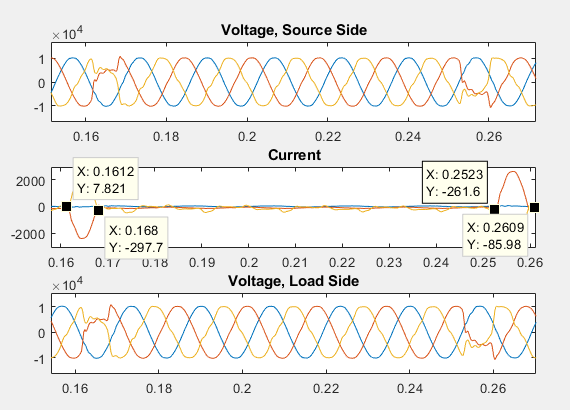


Figure 3.1 Graphical Representation of "The Ant Fault"

In this example, there is, what has been jokingly dubbed "The Ant Fault": the only way that this fault logically could occurr would be if an Ant were caught between two power lines as they bumped each other. The 8 milliseconds we see of each fault would be its body turning to ash. Looking at Figure 3.1, it can be seen that both faults start and end in a time period close to 8 milliseconds. These types of faults are few and far between, but for the quarter cycle evaluation, each fault happens in less than a quarter of a cycle and therefore are marked as ignorable faults or not seen at all. This is where the eighth cycle evaluation theoretically comes into play. In terms of the actual code, if the script sees no errors marked at the end of the code, or the cycle duration is marked as less than 1 cycle, we could step into eighth cycle analysis. If we step back to Figure 2.2 and Figure 2.3 and view it in terms of every 8 samples, we get a picture that looks like the following.

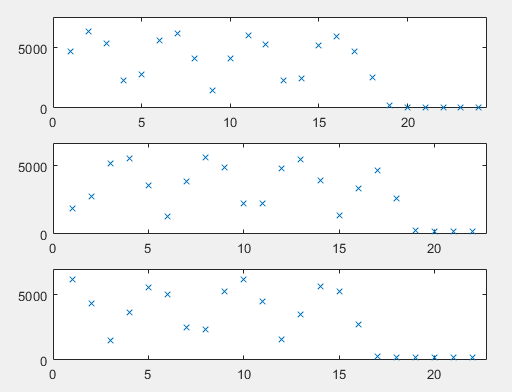


Figure 3.2 Eighth cycle analysis of the file shown in Figure 2.2

Looking at Phase A, B, and C, we see that the following RMS calculations shows that Phase A and B are faulted for 18 RMS Values, and Phase C is faulted for 16 RMS values. Knowing that for eighth cycle evaluation, every 8 RMS value equals 1 full cycle, it can be noted that the cycle duration does actually last 2.25 cycles. This validates the quarter cycle analysis for this specific file, and it cuts down on the possible wavering error. Since the processing power is much higher for this method, it is purely theoretical with its own quirks that need much more time to resolve.

Eighth cycle analysis was tested in an effort to achieve perfect classification in the endeavor of implementing polar plots. The polar plots themselves are a function that were never completed because of lack of time, but the premise of these plots would be to have a picture of the fault before, during, and after the fault happens. Because this function was never fully completed, it will be omitted from the code in Appendix C.

CHAPTER 4

RESULTS

In conclusion, the Square Mean Test (SMT) is effectively an expansion/decomposition of the RMS calculation so that a computer can quickly calculate the necessary values of Current and Voltage with high levels of accuracy. While eighth cycle analysis has some merit, anything after quarter cycle analysis has diminishing returns. These diminishing returns show in a higher level of processing power needed than would be considered practical. In theory, one could look at each individual point on the current arrays and see exactly where the fault began. Humorously though, the computing power that would be required for this is roughly 128x more than the half cycle analysis and since a supercomputer was not readily accessible, the small error from quarter cycle analysis is just fine for all practical purposes.

If this project were to be wrapped up and handed to someone else, it would be strongly encourage looking at the decomposition of the RMS equation so that the accuracy of using Matlab to classify faults can increase and theoretically reach the 100% accuracy mark. While this will not happen in the next couple of years, if computers continue to improve in the same exponential trend, we could see perfect classification within the next two decades.

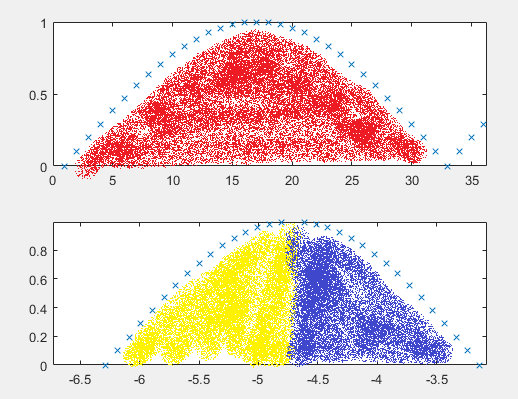
Appendix A

Formulas

Formula A.1 -- RMS expansion from half-cycle to quarter cycle

Given: ; where for all practical purposes n is 32 samples for half cycle evaluation,

We can see that quarter cycle evaluation is simply evaluating half of the half-cycle evaluation as shown below.



Therefore, the RMS values for quarter cycle have this formulation:

; again assuming that n = 32 or for quarter cycle n = 16. Rewriting this equation in a neater fashion yields the following:

Now, let's put a practical number for RMS. Assuming we are using this quarter cycle evaluation to evaluate "Y = sin(x)." The amplitude of this function is 1, and therefore we expect:

If we set the RMS for the quarter cycle evaluation equal to the expected RMS value, and substitute a variable of "A" and "B" for simplicity, we obtain the following.

Which can be rewritten as:

And knowing that we will need to find the mean of A and B combined, the following formula is obtained.

Or in other words, A+B should equal to 1. Why this is important, is that the moment these values don't equal to 1, we have a fault starting, and the moment A+B returns to equal 1, the fault has ended. This formulation can also be written in the following formulation

Which shows the RMS value as 'n', which is the number of parts involved in the decomposition. For Quarter cycle analysis, n = 2 since there are two RMS value's per half-cycle. More of why the quadratic is in the pi notation can be found in Formula A.2

Formula A.2 -- RMS expansion from quarter cycle to eighth cycle

This is where things begin to get more complicated. In the previous formula, we found that A+B has to equal 1. But what happens now if we break A and B independently? Let us instead use the following formation:

By solving for a+b+c+d, we find that they have to be equivalent to 2 in order to have 0.5 under the square root. When one tries to follow the pattern of before, of raising the expected RMS value to the 4th power, we find that there is a quadratic decay in which still satisfies the previously stated equation and that in effect, the pi notation above actually has to be considered in terms of a quadratic. What this means is that for the following equation, let n = 4 since that's the number of parts we have in a half cycle.

Doing this yields the power to be 5 instead and if we assume the magnitude to equal 1, we find that 'a' and 'c' should equal to 0.1767. 'b' and 'd' therefore to satisfy the requirement of a+b+c+d having to equal to 2, we find that:

Anyone struggling to see this reasoning need only copy the following code into Matlab.

X = 0:2\*pi/64:2\*pi;

Y = sin(x);

A = mean(y(1:9).^2);

B = mean(y(9:17).^2);

C = mean(y(17:25).^2);

D = mean(y(25:33).^2);

Now, as someone who is innately curious as to why this is, it involves the golden ratio and the fact that sine and cosine both change slope drastically as one steps through the waveform. However, this is beyond the scope of this project.

Formula A.3 -- Converting to Cycles and to time length

Let's assume some variables to be defined as follows.

for half cycle, this means c = 2; for quarter cycle, this means c = 4; and for eighth cyle, this means c = 8;

Appendix B

Table

Table 2.1 Shows the Mean and standard deviation for Magnitude, Start Time, and End Time in two separate fashions. The top 2 rows show the error where I do not take the absolute value of each independent value and the bottom 2 rows show the error where I do take the absolute value of each independent value. What this accomplishes is that I can see how well my numbers fit for the files that I have, and that based on my sample pool, I can extrapolate to what my worst case error would be with many more files.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Magnitude | Start Time | End Time |
| Mean: | 0.343785729 | 0.972707317 | 1.435829268 |
| St. Deviation: | 2.257067831 | 0.805065284 | 1.571328201 |
|  |  |  |  |
| Mean: | 0.80757697 | 1.006853659 | 1.485487805 |
| St. Deviation: | 2.133956599 | 0.761380879 | 1.523880907 |

Table 2.2A -- This is what one would expect under a normal distribution bell curve

|  |  |  |  |
| --- | --- | --- | --- |
| Expected Values | Magnitude | Start Time | End Time |
| 1 St. Dev. | 0.682 | 0.682 | 0.682 |
| 2 St. Dev. | 0.89 | 0.89 | 0.89 |
| 3 St. Dev | 0.97 | 0.97 | 0.97 |

Table 2.2B -- This is how many of my files fall within a certain standard deviation. This shows that for the brunt of files that pass through this program, my precision is better than what is expected barring the few strange acorns in the mix.

|  |  |  |  |
| --- | --- | --- | --- |
| Actual Values | Magnitude | Start Time | End Time |
| 1 St. Dev. | 0.7195 | 0.939 | 0.9634 |
| 2 St. Dev. | 0.89 | 0.9512 | 0.9756 |
| 3 St. Dev | 0.939 | 0.9756 | 0.9878 |

Appendix C

Matlab Code

* Run\_me() -- The base function that just pulls all comtrade files, converts them to matlab structures, and then classifies them.

function run\_me()

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%The following Code was designed, tested, and programmed originally by

%Mitch Lautigar. Though the code is open source, please either leave this

%comment block in here, or properly cite me for my code.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%The program below takes in the output array after creating an array for

%all files in the folder and writes it into a string array that gets

%written to a .txt file.

file\_list = dir('\*.CFG');

file\_list2 = dir('\*.DAT');

x = '';

for i = 1:length(file\_list)

%load(file\_list(i).name);

dataCFG = readCFG\_COMTRADE(file\_list(i).name);

data = readDAT\_COMTRADE(file\_list2(i).name,dataCFG);

report = loadandgraph(data,dataCFG);

%pause(2);

x = [x;cellstr(report)];

end

c = clock;

c = num2str(c);

c = strsplit(c,' ');

c = strjoin([c,'.txt'],'\_');

fid = fopen(c,'w');

fprintf(fid,'%s ~~~~~ %s ~~~~~ %s ~~~~~ %s ~~~~~ %s ~~~~~ %s ~~~~~ %s ~~~~~ %s \r\n', [string(x(:,1)),string(x(:,2)),string(x(:,3)),string(x(:,4)),string(x(:,5)),string(x(:,6)),string(x(:,7)),string(x(:,8))]');

fclose(fid);

end

* Loadandgraph()-- The Home screen of this whole product that breaks down the order of what happens quickly and efficiently.

function [output\_array,max,debug\_values] = loadandgraph(data,dataCFG)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%The following Code was designed, tested, and programmed originally by

%Mitch Lautigar. Though the code is open source, please either leave this

%comment block in here, or properly cite me for my code.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%This program takes in the structures from the .mat files and communicates

%between the other functions and then relays that information to run\_me()

%while also saving the graph as a .png file.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Input Definitions:

%data -- matlab struct that holds all numerical values

%dataCFG -- matlab struct that holds some of the specific bits of

%information(i.e. the time array, device name, so on and so forth)

%Output Definition:

%output\_array -- This is the array outputted and eventually saved to a text

%file.

%max & debug values -- only used for someone debugging the script. They are

%a lazy way to check the script numbers against what is seen on the graphs

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%-------------------------------------------------------------------------%

% Voltage Upstream %

%-------------------------------------------------------------------------%

Vx\_1 = data.analog.VX1;

Vx\_2 = data.analog.VX2;

Vx\_3 = data.analog.VX3;

%-------------------------------------------------------------------------%

% Voltage Downstream %

%-------------------------------------------------------------------------%

Vy\_1 = data.analog.VY1;

Vy\_2 = data.analog.VY2;

Vy\_3 = data.analog.VY3;

%-------------------------------------------------------------------------%

% Current %

%-------------------------------------------------------------------------%

i\_1 = data.analog.I1;

i\_2 = data.analog.I2;

i\_3 = data.analog.I3;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

t = data.time;

device\_name = dataCFG.recording\_device;

Date\_time = [dataCFG.startdate,'\_',dataCFG.starttime];

i\_values = [i\_1';i\_2';i\_3'];

%-------------------------------------------------------------------------%

% Plot Values %

%-------------------------------------------------------------------------%

[num\_samples,~] = size(t);

[power\_line,fault\_summary\_array\_i,~,ste] = compart\_classify(Vx\_1,Vx\_2,Vx\_3,Vy\_1,Vy\_2,Vy\_3,i\_1,i\_2,i\_3,t);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[current\_output] = set\_array(i\_values,power\_line(4:6,:),fault\_summary\_array\_i,device\_name,Date\_time,num\_samples,ste);

[a,~] = size(current\_output);

array\_delete = [];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%false hits with magnitude less than specified amps.

speced\_amps = 900;

speced\_amps\_check = 1000;

if a > 1

for i = 1:a

mag\_check = str2num(cell2mat(cellstr(current\_output(i,5))));

if mag\_check < speced\_amps

array\_delete = [array\_delete,i];

end

if (mag\_check >= speced\_amps) && (mag\_check <= speced\_amps\_check)

cycle\_check = str2num(cell2mat(cellstr(current\_output(i,5))));

if cycle\_check >= 1

array\_delete = [array\_delete,i];

end

end

end

if length(array\_delete) ~= a

current\_output(array\_delete,:) = [];

end

end

output\_array = current\_output

[e,~] = size(output\_array);

for i = 1:e

b\_samp(i,1) = str2num(cell2mat(output\_array(i,8)));

e\_samp(i,1) = str2num(cell2mat(output\_array(i,9)));

max(i,1) = str2num(cell2mat(output\_array(i,5)));

end

tims = [b\_samp,e\_samp] ./ length(t);

freq = num\_samples / length(power\_line);

debug\_values = [tims];

[q,w] = size(output\_array);

if (q > 1) || (min(str2num(cell2mat(output\_array(:,6))) < 2))

append\_array(1:q,1) = cellstr("double\_check");

output\_array = [output\_array,append\_array];

else

append\_array(1:q,1) = cellstr("all good");

output\_array = [output\_array,append\_array];

end

%%{

graph\_title = strjoin([output\_array(1,1:3),'png'],'\_\_');

c = strsplit(graph\_title,'/');

c = strjoin(c,'\_');

c = strsplit(c,':');

c = strjoin(c,'-');

figure

subplot(3,1,1)

plot(t,Vx\_1,t,Vx\_2,t,Vx\_3)

title('Voltage, Source Side')

subplot(3,1,2)

plot(t,i\_1,t,i\_2,t,i\_3)

title('Current')

subplot(3,1,3)

plot(t,Vy\_1,t,Vy\_2,t,Vy\_3)

title('Voltage, Load Side')

print('-f1',c,'-dpng')

close(figure(1))

%}

End

* Compart\_classify() -- This is where the fault is identified and set into a value called "fault\_summary\_array."

function [power\_line,fault\_summary\_array\_i,fault\_type,start\_time\_error] = compart\_classify(Vx\_1,Vx\_2,Vx\_3,Vy\_1,Vy\_2,Vy\_3,i\_1,i\_2,i\_3,t)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%The following Code was designed, tested, and programmed originally by

%Mitch Lautigar. Though the code is open source, please either leave this

%comment block in here, or properly cite me for my code.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Notes from creator:

%Out of all the files for this program, this is where 99% of all debugging

%will take place. If you have the program running, put a pause where you

%see the following line of code:

% "fault\_array(1:4,counter) =

% {char(ftca);num2str(cycle\_counter+1);num2str(starting\_point);num2str(lines\_faulted)};"

% and before you step into the fault loop, look at the fault array called

% fault which will tell you which phase is faulted at each grouping from

% the SMT values. The key point to know there is a definite fault with the

% for loop at the end of this function is when the "fault\_array" has a

% matrix of [0 0 0] as it's last value at the bottom.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%This program itself will look at the SMT values of all the inputs and find

%the "fault\_array" output. The fault\_array output is just an array with all

%data needed to compute the values that are computed in the "set\_array"

%function.

%{

---------------------------------------------------------------------------

This function works effectively to take in the values of a comtrade file

and classify the faults accordingly. The following steps will be commented

and broken down for ease of understanding.

Input Breakdown

Vx\_1: The voltage of Phase A on the source side.

Vx\_2: The voltage of Phase B on the source Side.

Vx\_3: The voltage of Phase C on the source side.

Vy\_1: The voltage of phase A on the load side.

Vy\_2: The voltage of phase B on the load side.

Vy\_3: The voltage of phase C on the load side.

i\_1: The current through phase A.

i\_2: The current through phase B.

i\_3: The current through phase C.

Output Breakdown

1. current\_value: The square mean test value computed by the code for all 3

phases of current stacked in a single array with phase A being row 1 of the

array,phase B row 2, and phase C row 3.

2. source\_voltage: The square mean test value computed by the code for all 3

phases of voltage stacked in a single array with phase A being row 1 of the

array,phase B row 2, and phase C row 3.

3. load\_voltage: The square mean test value computed by the code for all 3

phases of voltage stacked in a single array with phase A being row 1 of the

array,phase B row 2, and phase C row 3.

4. fault\_length: an array that contains the number of lines faulted at each

individual sample of the square mean test array.

5. fault\_error: an array that contains what faults happened in this file

and will be used later and is designed to be in the report of this code.

---------------------------------------------------------------------------

%}

%The subplot below is designed for any using the hard coded functions to be

%able to see the graphs of the original values to eyeball what the fault

%is. Most of the time, this will be commented out unless it's needed.

%{

figure

subplot(3,1,1)

plot(t,Vx\_1,t,Vx\_2,t,Vx\_3)

title('Voltage, Source Side')

subplot(3,1,2)

plot(t,i\_1,t,i\_2,t,i\_3)

title('Current')

subplot(3,1,3)

plot(t,Vy\_1,t,Vy\_2,t,Vy\_3)

title('Voltage, Load Side')

%}

%-------------------------------------------------------------------------%

%Send each individual input into the squaretestmean function to acquire a

%simplified array that can be used for comparison. The values are then

%grouped together into arrays specified to current, load voltage, and

%source voltage.

[i1\_smt, i1] = squaretestmean(i\_1');

[i2\_smt, i2] = squaretestmean(i\_2');

[i3\_smt, i3] = squaretestmean(i\_3');

[Vx1\_smt, vx1] = squaretestmean(Vx\_1');

[Vx2\_smt, vx2] = squaretestmean(Vx\_2');

[Vx3\_smt, vx3] = squaretestmean(Vx\_3');

[Vy1\_smt, vy1] = squaretestmean(Vy\_1');

[Vy2\_smt, vy2] = squaretestmean(Vy\_2');

[Vy3\_smt, vy3] = squaretestmean(Vy\_3');

start\_time\_error = [vx1 i1 vy1; vx2 i2 vy2; vx3 i3 vy3];

%{

figure

subplot(3,1,1)

plot(1:length(Vy1\_smt),Vx1\_smt,'x',1:length(Vy1\_smt),Vx2\_smt,'x',1:length(Vy1\_smt),Vx3\_smt,'x')

title('Voltage SMT, Source Side')

subplot(3,1,2)

plot(1:length(Vy1\_smt),Vy1\_smt,'x',1:length(Vy1\_smt),Vy2\_smt,'x',1:length(Vy1\_smt),Vy3\_smt,'x')

title('Voltage SMT, Load Side')

subplot(3,1,3)

plot(1:length(Vy1\_smt),i1\_smt,'x',1:length(Vy1\_smt),i2\_smt,'x',1:length(Vy1\_smt),i3\_smt,'x')

title('Current SMT')

%}

current\_value = [i1\_smt; i2\_smt; i3\_smt];

source\_voltage = [Vx1\_smt;Vx2\_smt;Vx3\_smt];

load\_voltage = [Vy1\_smt;Vy2\_smt;Vy3\_smt];

power\_line = [source\_voltage;current\_value;load\_voltage];

%-------------------------------------------------------------------------%

%This is where the fault classification begins to take place. The first for

%loop effectively takes the stacked current array of all 3 phases and steps

%through each line at each individual value to see what that number is

%doing. Here's a quick explanation of how the following numbers were chosen

%but can quickly be edited as needed.

%For this method, after the values have been computed, we take the nominal

%line voltage and find the RMS value of it so as to compare it to the

%Square mean Test method and obtain a percent error between the two values.

%Once the percent error values have been found, I then have specified that

%if there is more than 10 percent difference, it's a fault.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

line\_voltage = 10000;

v\_check = line\_voltage / sqrt(2);

vol\_pe\_x = abs(v\_check - source\_voltage) / v\_check .\* 100;

vol\_pe\_y = abs(v\_check - load\_voltage) / v\_check .\* 100;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[~,b] = size(vol\_pe\_x);

fault\_array\_x(1:3,1:b) = 0;

fault\_array\_y(1:3,1:b) = 0;

fault\_array\_i(1:3,1:b) = 0;

for i = 1:b

for k = 1:3

if vol\_pe\_x(k,i) > 15

fault\_array\_x(k,i) = 1;

end

if vol\_pe\_y(k,i) > 15

fault\_array\_y(k,i) = 1;

end

if (current\_value(k,i) > 250)

fault\_array\_i(k,i) = 1;

end

end

end

if sum(sum(fault\_array\_i)) ~= 0

[fault\_summary\_array\_i,fault\_type\_i] = fault\_evaluate(fault\_array\_i);

[fault\_summary\_array\_i] = fault\_check(fault\_summary\_array\_i,1);

fault\_type = fault\_type\_i;

else

end

* Set\_array() -- This actually does the calculations and formats everything into the output array

function [output\_array] = set\_array(i\_values,current\_value,fault\_array,name,dt,num\_samples,ste)

%1 & 2

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%The following Code was designed, tested, and programmed originally by

%Mitch Lautigar. Though the code is open source, please either leave this

%comment block in here, or properly cite me for my code.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%The following code computes the output array that is as follows

%Name of device, fault\_type, magnitude of fault, cycle duration, second

%duration, start sample, end sample

freq = round(length(current\_value)/4);

fault\_type = fault\_array(1,:); %3

lines\_faulted = fault\_array(2,:); %4

cycle\_duration = cell2mat(fault\_array(4,:));

startpoint = cell2mat(fault\_array(3,:));

[~,b] = size(fault\_array);

beta = [];

if b == 0

output\_array = [name,dt,"blip","blip","blip","blip","blip","blip","blip" ];

elseif b ~= 0

for i = 1:b

fa = str2num(cell2mat(lines\_faulted(1,i)));

ste2 = mean(ste(:,2) );

cycle\_durated = cycle\_duration(1,i) / 4; %6

if startpoint(1,i) == 1

start\_samples = 0;

else

start\_samples = round((startpoint(1,i) ) \* 8 );%8

end

cycle\_samples = round(cycle\_durated \* freq);

if cycle\_durated <= 3

if start\_samples == 0

mag\_max = max(max(abs(i\_values(:,1:(start\_samples\*2+cycle\_samples+32)))));

else

mag\_max = max(max(abs(i\_values(:,(start\_samples\*2-16):(start\_samples\*2+cycle\_samples+32)))));

end

else

mag\_max = max(max(abs(i\_values(:,start\_samples\*2+16:start\_samples\*2+cycle\_samples)))); %5

end

end\_sample = start\_samples + cycle\_samples;%9

second\_duration = cycle\_samples / num\_samples; %7

output\_array(i,1:9) = [name,dt,fault\_type(1,i),string(lines\_faulted(1,i)),num2str(mag\_max),num2str(cycle\_durated),num2str(second\_duration),num2str(start\_samples),num2str(end\_sample)];

end

end

* Fault\_evaluate() -- An internal function of compart\_classify that effectively looks at the fault\_array\_i and figures out where each fault begins and end.

function [fault\_summary,fault\_type] = fault\_evaluate(fault\_array)

fault\_test = sum(fault\_array);

begin\_fault = 1;

output = [];

fault\_class = {'Clear','LG','LL','LLL'};

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Part 3, find the fault type

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fault\_type = [];

if fault\_test(1,1) ~= 0

fault\_type = [fault\_type,'Ongoing'];

elseif fault\_test(1,end) ~= 0

fault\_type = [fault\_type,'Continuing'];

elseif (fault\_test(1,1) == 0) && (fault\_test(1,end) == 0)

fault\_type = ['contained'];

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Part 1, find the beginning and end start time for all faults.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%This while loop finds an array that is then used to calculate the end

%time.

while length(fault\_test) >= 1

lines\_faulted = fault\_test(1,1);

y = find(fault\_test ~= lines\_faulted);

if isempty(y) == 1

end\_fault = length(fault\_test);

else

end\_fault = y(1,1)-1;

end

be\_end = [begin\_fault;end\_fault];

output = [output,be\_end];

fault\_test(:,begin\_fault:end\_fault) = [];

end

output = [[0;0],output];

rolling\_sum = 0;

%This for loop calculates the actual end time.

for i = 1:length(output)-1

rolling\_sum = rolling\_sum + output(2,i+1);

ending(1,i) = rolling\_sum;

end

begin = 1 + ending;

begin(:,end) = [];

begin = [1,begin]; %Calculate the beginning time.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Part 2, find the fault class

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for i = 1:length(ending)

spot\_check = floor( (ending(1,i) + begin(1,i) ) / 2);

lines\_faulted(1:3,i) = fault\_array(:,spot\_check);

fault\_classification(1,i) = fault\_class(1,sum(lines\_faulted(:,i))+1);

end

zeta = find(strcmpi(fault\_classification,'Clear') == 1);

fault\_classification(:,zeta) = [];

begin(:,zeta) = [];

ending(:,zeta) = [];

lines\_faulted(:,zeta) = [];

output(:,1) = [];

output(:,zeta) = [];

cycle\_duration = output(2,:);

%fault\_summary\_array = [cellstr(fault\_abbrev);num2cell(phase\_set);num2cell(counter\_array);num2cell(starting\_sample)];

for i = 1:length(ending)

fault\_summary(1:4,i) = [cellstr(fault\_classification(1,i));cellstr(num2str(lines\_faulted(:,i)'));num2cell(begin(1,i));num2cell(cycle\_duration(1,i))];

end

end

* Fault\_check -- Looks through the array from fault\_evaluate and gets rid of redundancies that can be seen. This effectively combines any misgnomers together into a simplified output array.

function [fault\_array\_corrected] = fault\_check(fault\_array,cycle)

if cycle == 1 % Quarter cycle

wiggle = 2;

else

wiggle = 6;

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[~,b] = size(fault\_array);

delta\_b = 1;

c = 0;

delta\_c = 1;

while delta\_c ~= 0

while delta\_b ~= 0

fan = cell2mat(fault\_array(3:4,:));

fas = fault\_array(1:2,:);

ad = [];

if b ~= 1

wa = [-wiggle wiggle];

for i = 1:b-1

fc = fan(:,i);

fp = fan(:,i+1);

check = fp(1,1) - fc(1,1);

if (check > wiggle) && (fc(2,1) < wiggle)

ad = [ad,i];

else

fpc = fp(1,1) + wa;

if (sum(fc) >= fpc(1,1) ) && (sum(fc) <= fpc(1,2) ) && (min([fc(2,1),fp(2,1)]) < 4 \* wiggle)

if fan(2,i+1) > fan(2,i)

fas(:,i) = fas(:,i+1);

elseif fan(2,i+1) < fan(2,i)

fas(:,i+1) = fas(:,i);

else

if sum(str2num(cell2mat(fas(2,i)))) >= sum(str2num(cell2mat(fas(2,i))))

fas(:,i+1) = fas(:,i);

else

fas(:,i) = fas(:,i+1);

end

end

fan(1,i+1) = fc(1,1);

fan(2,i+1) = fp(2,1) + fc(2,1);

ad = [ad,i];

end

end

end

fault\_array = [fas;num2cell(fan)];

fault\_array(:,ad) = [];

fault\_array\_corrected = fault\_array;

[~,b2] = size(fault\_array\_corrected);

delta\_b = b2-b;

if delta\_b ~= 0

b = b2;

fault\_array = fault\_array\_corrected;

end

end

if b == 1

delta\_b = 0;

fault\_array\_corrected = fault\_array;

end

end

[~,c\_new] = size(fault\_array\_corrected);

if c\_new - c == 0

delta\_c = 0;

else

c = c\_new;

end

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[~,b] = size(fault\_array\_corrected);

ad = [];

fan = cell2mat(fault\_array\_corrected(3:4,:));

fas = fault\_array\_corrected(1:2,:);

for i = 1:b-1

if (strcmpi(string(fas(1,i)),string(fas(1,i+1))) == 1) && ( (sum(fan(:,i)) >= fan(1,i+1) - wiggle) && (sum(fan(:,i)) <= fan(1,i+1) + wiggle) )

fan(2,i) = fan(2,i) + fan(2,i+1);

ad = [ad,i+1];

end

end

fault\_array\_corrected = [fas;num2cell(fan)];

fault\_array\_corrected(:,ad) = [];

end

VITA

Mitch Lautigar was born in Shreveport, Louisiana to the parents of Joseph Lautigar and Crystal Marcantel. He is the oldest son of two children, with a younger brother. He attended Chattanooga State Community College for his freshman and sophomore years of college where he majored in Computer Science. From there, he switched to Tennessee Technical University where he majored in Electrical Engineering. It was here that the knowledge of programming was acquired in Matlab which was paramount to this thesis project. Mitch completed his Bachelors of Science in Electrical Engineering in May 2017, and then transferred to UTC to attain a Masters of Science in Electrical Engineering at the University of Tennessee, Chattanooga. Mitch graduated with his Master's of Science degree in May 2019 and is enjoying living life one day at a time.

*There are only two ways to live your life. One is as though nothing is a miracle. The other is as though everything is a miracle."*

--Albert Einstein