AUTOMATED TRANSMISSION LINE FAULT ANALYSIS, THE SMT METHOD

By

Mitchell Alexander Lautigar

Abdelrahman Karrar
Associate Professor of Engineering
(Chair)

Raga Ahmed
Assistant Professor of Engineering
(Committee Member)

Abdul R. Ofoli
UC Foundation Associate Professor of Engineering
(Committee Member)

Mr. Robert Hay
(Industry Advisor)

AUTOMATED TRANSMISSION LINE FAULT ANALYSIS, THE SMT METHOD

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 Science: Engineering

The University of Tennessee at Chattanooga
Chattanooga, Tennessee

May 2019

ABSTRACT

Analyzing faults from transmission lines automatically using a computer is a complex, multi-step process that takes a literal understanding of how the computer reads the data. While there are many ways to approach this problem(e.g.. Inductance Calculation), the Square Mean Test method allows for quick, and efficient calculations of any files read in from Intelliruptors. These calculations are then grouped and classified together outside of this project to prioritize faults that need to be looked at.

DEDICATION

Special thanks to my family, my teachers 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 available to me. To all who are listed, and those who go unlisted in this paper, thank you.

"Friendship is the hardest thing in the world to explain. It's not something you learn in school. But if you haven't learned the meaning of friendship, you really haven't learned anything." -- Muhammad Ali

ACKNOWLEDGEMENTS

Special thanks to Jonathan King, Dr. Karrar and Mr. Robert Hay. Jonathan helped me to mathematically find a way to verify the number ranges I saw for Half-Cycle, Quarter-Cycle, and Eighth Cycle analysis. This in itself had stumped me for months before the project was presented, and ultimately just needed someone with a different background to solve.

I went to Dr. Karrar a year ago just wanting a second opinion on an idea, and since then, he has 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. Every second of this project has been a challenge that was sorely needed to help me grow.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

ω_e, Wavering Error

LIST OF FORMULAS

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 main problem was to achieve automated classification with sufficient accuracy (ability to distinguish faulted and non-faulted instances) and high time resolution (ability to accurately detect beginning and end of a fault frame). Higher time resolution is important for proper classification, and can further serve as input for extended analysis, such as fuse forensic analysis. Thus, the study attempts to arrive at the optimal analytical method to address these two issues. The output for this study for each individual Comtrade[1] file can be found in Table 1.1 below with the appropriate summaries of each value that can be found inside the array.

Table 1.1 Output Array

Device Name	Time Stamp	Fault Class	Lines Faulted Array	Fault Magnitude	Cycle Duration	Starting Sample	End Sample

Device Name is a value obtained from the file in which it just mentions the device name that can later be looked up to see where the device is located. There are no calculations needed for this data as it is literally carried through the script.

Time Stamp shows the time the file begins in the format of "*day/month/year_hour:minute:seconds*". The seconds value shows in terms of six decimal places so that someone can see time conception within microseconds. The time is shown in military time in respect to the Eastern Time Zone.

Fault Class is a calculated value from the SMT method that outputs either LG(Line-Ground), LL(Line-Line), LLL(Line-Line-Line)[2].

Lines Faulted Array outputs a 1x3 array in terms of a string that shows which phases are faulted. This will look like "*A B C*" where it starts at what the file calls Phase A and then goes to Phase C showing a 1 for faulted line and a 0 for a non-faulted line.

Magnitude is a value obtained from the SMT method that calculates the magnitude of the fault by taking the peak value it finds and using RMS fundamentals[3] to find a close value to the actual magnitude of the fault within the file.

Cycle Duration is how many full cycles the fault lasts. This comes from just finding the start and end point of the fault and doing a conversion to simplify the sample duration into cycle duration.

Second Duration is a calculated value obtained from the script by taking the cycle duration and multiplying it by 64 and then dividing it by the number of samples in the file(typically 1981).

Start Sample is the approximate starting sample at which the fault is calculated to begin at.

End Sample is the approximate ending sample at which the fault is calculated to end at.

After these nine values are organized into an output array, they are then placed in a list that holds all the previous output values obtained from other files. Once classification for all files in the program are done computing, the program saves this list to a text file which is then read by another program to grab the unknown or unclassified faults. The remainder of the thesis is organized as follows; Chapter 2 presents and discusses the methods attempted for fault automation. Chapters 3, 4, and 5 detail the method of choice for the automation which is the RMS calculation approach. This starts with Half-Cycle Analysis, moves to Quarter-Cycle Analysis, and finishes with the theoretical Eighth-Cycle Analysis. Final results will be discussed in Chapter 6.

CHAPTER 2

METHODOLOGY, THE GOOD, THE BAD, & THE NO

There were a total of three methods that were looked into in regards to this project. Each method will be explained in detail of how they work, and why they were passed over in favor of the RMS expansion(The SMT Method).

To start with, Matlab[4] has a prebuilt function called *Findsignal()* which has two possible inputs and three possible outputs. This looks like the following:

```
[istart,istop,dist] = findsignal(data,signal);
```

Where *Data* would be the data from the files used in the program that are then checked by the *signal* array. With respect to the output, one can see the start and stop times as *istart* and *istop* as well as a value called *dist* which is the squared Euclidean distance of the segment and the search array. This command has the strength of being able to quickly and efficiently step through files to classify data and to theoretically see all start and end times of any faults happening. However, since the input *signal* would have to account for every possible fault, a huge database of all possible faults would be needed in order to make this command useable. Since this is not a practical application of time, this command was ultimately passed over.

The next method is also another Matlab prebuilt function called *Findchangepts()* which has one base input and output. This looks like the following.

```
Ipt = findchangepts(x);
```

Where the input *x* is the data obtained from the files, and the output *ipt* is the location in which the mean of *x* changes significantly. This has some powerful uses in regards to seeing the dramatic changes in a waveform when they occur, but would be hard to debug because the function call *findchangepts()* would have to be in a while loop that will run indefinitely. While loops themselves are a powerful tool to use when programming but are notoriously difficult to debug since when they are used, it's not easily seen what is keeping them in a while loop. In terms of practicality, this command was ultimately passed over because of this.

The next method attempted came from the fundamentals of Root Mean Square calculations with the idea being of having the computer look at a fraction of the numbers represented by a single point instead of hoping to look through the whole file. However, upon looking at the practicality of the regular form of RMS calculations(also called Half-Cycle Analysis), there was a lot of error in regards to start and end time. To fix this error, an expansion of RMS was tested that proved to be much more accurate in regards to the start time and end time while still allowing for a quick step back to half-cycle analysis. This is where Quarter cycle and Eighth cycle evaluation come into play. Both of these evaluation methods come from the RMS formula and since they are a different way of looking at the RMS have been called the Square Mean Test(SMT) method.

The desired end goal from the chosen method is to have an understandable and replicable method to handle a large number of files in an efficient and smooth manner. This lets the faults that are not easily solvable be pushed to the front of an engineer's attention so that they can be solved.

CHAPTER 3

HALF-CYCLE ANALYSIS, THE RMS FORMULATION

When referring to Half-Cycle Analysis, it is important to understand why this method was the starting point for all the methods that followed. To start with, Half-Cycle Analysis is really just the regular RMS formula. This is defined as follows:

$$RMS = \sqrt{\frac{\sum_{i=1}^{n} y(1,i)^2}{n}} \quad \text{Formula 3.1}$$

For a standard file size of 1981 samples (this will be the assumed length for all files mentioned from this point forward) one can group number ranges by every half cycle, or in simpler words, every RMS value calculated will represent a specified amount of samples in relation to half-cycle. It takes 64 samples to make a full cycle; which makes half-cycle have a sample length of 32 samples. From here on out, the sample length for any cycle evaluation will be denoted as "n" which has *n=32* for half-cycle analysis. Therefore, it can be seen that the amount of numbers to evaluate for half-cycle analysis is defined as:

$$\frac{1981}{n} = \frac{1981}{32}(For\ Half - Cycle) = 61.91 \cong 62 \quad \text{Formula 3.2}$$

Evaluating 62 numbers takes much less processing power while still offering a range of accuracy that will be assessed and evaluated later in this paper. This is the biggest benefit to using half-cycle analysis but there is one more benefit found with this method; it's easy to debug. Having to look at 62 numbers, as a programmer, is exponentially quicker than looking at 1981 numbers.

Let's take an example of a fault that happened on the power grid some time ago. The following fault happened on one of EPB's Intelliruptors[5], and this fault will be referred to as 'Fault E6.'

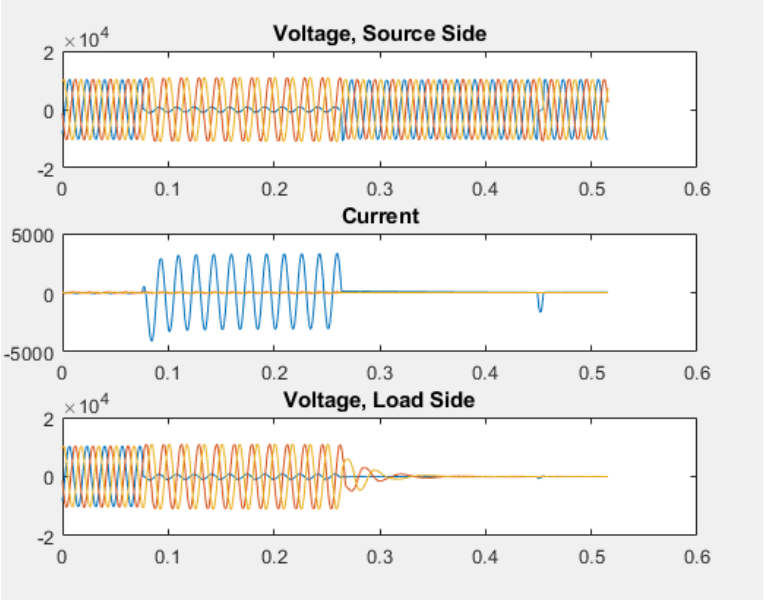


Figure 3.1 Graphical Representation of Fault 'E6'

What can be seen in this fault is a simple Line to Ground Fault(LG for short) that starts and resolves within the file with an attempted pulse-close attempt located close to 0.45 seconds. The source side voltage is able to resume regular operations while the load side remains faulted out through the duration of the file. Zooming into the file and then putting some traces on the current graph yields Figure 3.2 below.

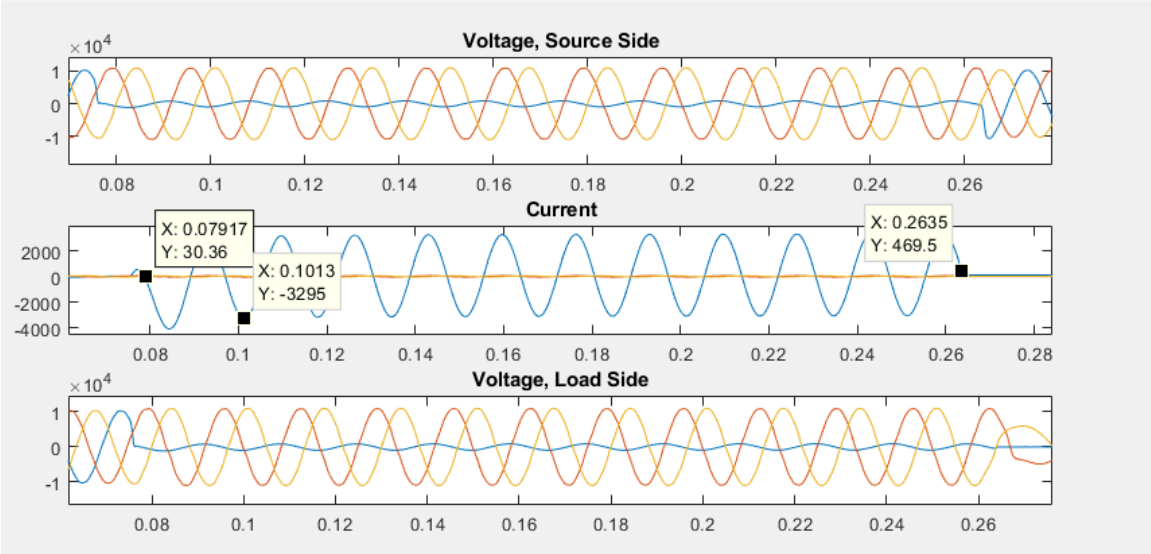


Figure 3.2 Enlarged portion of Fault E6 Showing Fault Magnitude and Duration

Figure 3.2 above just shows the duration of the fault with the start time, end time, and the magnitude of the fault marked. This figure has been provided as a check and will be referred to as the E6 Fault Solution. With this said, it is time to change the focus from the solution to the method for Half-Cycle Analysis. When the Current graph is separated from the voltages and focused only on the absolute value of the current of the faulted phase, the following figure is obtained.

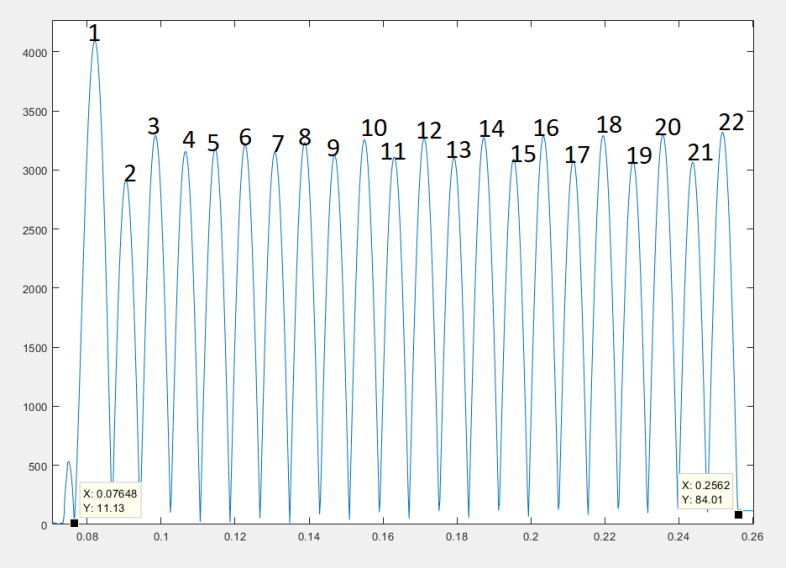


Figure3.3 The Absolute Value of the Faulted Current for Fault E6

The graph in Figure 3.3 above shows 22 marked half-cycles which is what would be expected when the RMS calculation is completed. Yet, when the RMS for half-cycle Evaluation happens as depicted below in Figure 3.4, it is found that this is not the case and what is actually seen is 24 half-cycle marks(this is approximately 0.1858 seconds for fault duration) for the RMS calculations.

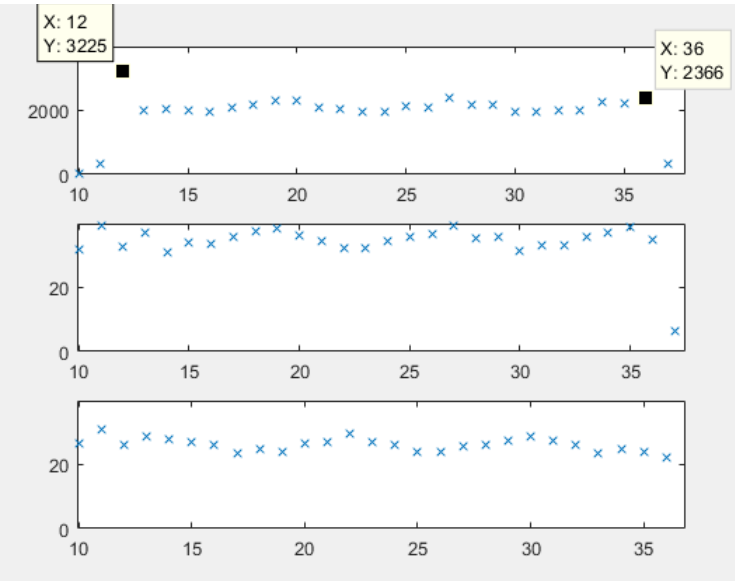


Figure 3.4 Half-Cycle RMS of Fault E6.

Wavering Error

The reason for these two extra RMS marks is not an easy one to see. The answer comes from the difference of half-cycle length within the actual fault. Examining Figure 3.5 below, and comparing the time gaps just between the first seen half-cycle and the second seen half-cycle, a clear difference can be seen.

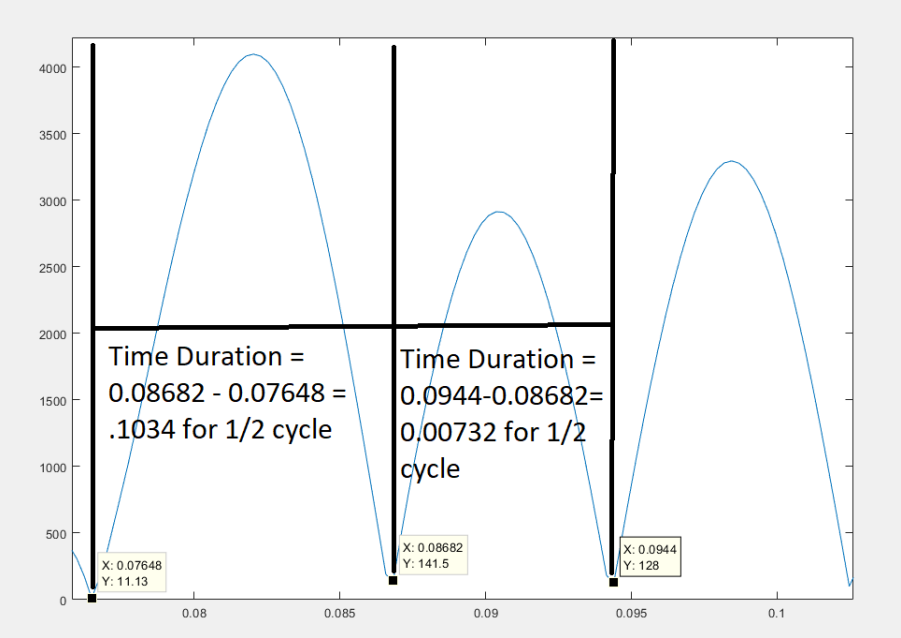


Figure 3.5 Graphical Representation of Wavering Error

This difference, or what is referred to as Wavering Error, is only 3 milliseconds in time. However, if multiple half-cycles have this error, it can add up. Reexamining figure 3.3 where the number of half-cycles are counted, and converting them into time, there is a discrepancy that can quickly be seen and observed as shown below in Figure 3.6

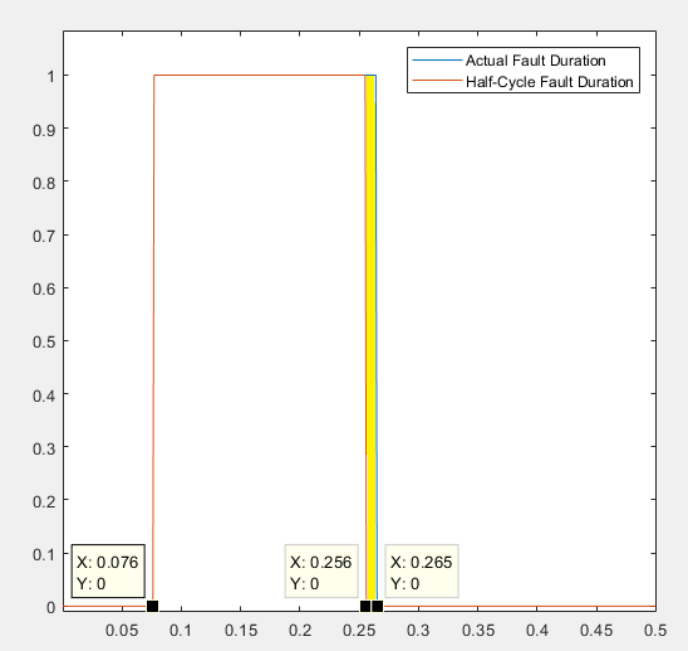


Figure 3.6 Total Wavering Error shown in terms of a unit step between expected cycles faulted and actual.

This difference of 10 milliseconds is what is defined as Wavering Error where the total difference between what is expected and what is actually there are not the same. This is what took the greatest amount of time to understand and comprehend. Since wavering error can now be seen as a problem, the limits of wavering error needs to be defined. After studying thousands of files, Figure 3.6 shows a graphical picture of what wavering error looks like for Half-cycle analysis. Please note that Figure 3.7 below is a mock example and not related to Fault E6.

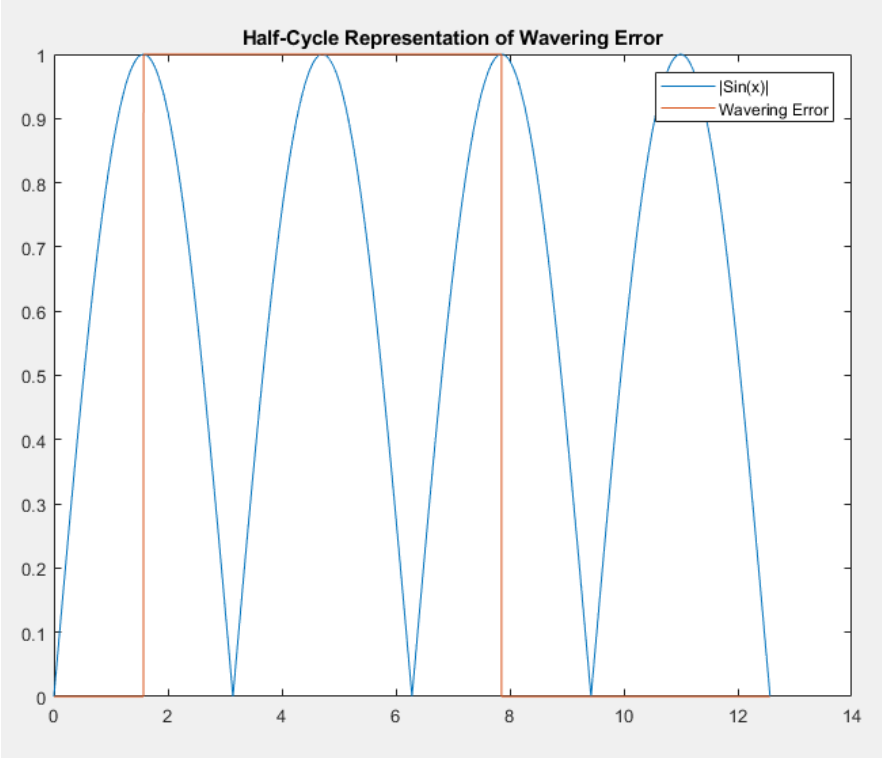


Figure 3.7 Wavering Error Half-Cycle Illustrative Waveform

In Figure 3.7, function A(|Sin(x)|, the blue graph) is marked in comparison against Function B(Wavering Error, the Green Graph). Function B shows the area where a fault conception could begin or end and still be marked within the full half-cycle located within the range of Function B. This therefore causes Wavering Error to be defined within the following range:

$$\omega_e = (Current_{Cycle} * n) \pm n \quad \text{Formula 3.3}$$

With wavering error defined, the results for half-cycle analysis are all that's left. To evaluate the validity of the following methods, I will be comparing the mean and standard deviation for all values I calculate versus a normal distribution. For a normal distribution, what would be expected is as follows.

Half-Cycle Results

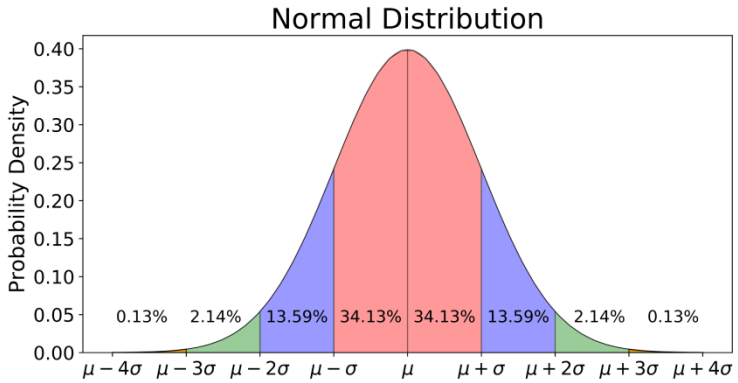


Figure 3.8 Normal Distribution Curve

Figure 3.8 shows expected percentages within different standard deviations with the key point behind using the normal distribution as a baseline, is that it provides a range of what is acceptable in terms of accuracy. Within one standard deviation, it is expected to see 68.26%(2*34.13) of all files. Within 2 standard deviations, it is expected to see 95.44%(2*34.13+2*13.59) of all files. Lastly, within three standard deviations, it is expected to see 99.72% of all files. Using this to test the Half-Cycle Analysis, The following calculations were obtained for a sample size of 300 files where the start time, end time, and magnitude were independently marked by sight and compared against the script.

Table 3.1 Half Cycle Statistics for Percent Error.

	Magnitude	Start Time	End Time
Mean:	0.2175	8.3245	9.2142
St. Deviation:	1.2354	2.7931	3.1264
Mean:	0.40513	9.7193	10.4539
St. Deviation:	1.6529	3.2102	4.5684

The format in Table 3.1 above shows two separate views of the statistics for Half-cycle Statistics. The top half of the table, the glass half full view, is showing the mean and standard deviation for the percent error values without taking the absolute value, and the bottom half of the table, the glass half empty view, shows the same for the absolute value of the percent error values. The difference between them is relatively miniscule with the glass half empty view having a wider range of what is acceptable. When these values were used to see how many files fall within varying Standard Deviations, the following was obtained.

Table 3.2 File Percentage vs. Standard Deviations

Actual Values	Magnitude	Start Time	End Time
1 St. Dev.	0.8421	0.7212	0.6876
2 St. Dev.	0.8716	0.7832	0.7489
3 St. Dev	0.9235	0.8657	0.7815

The takeaway from this comparison is that the accuracy is there for the magnitude, but the start time and end time leave much to be desired. The biggest downfall to the half-cycle analysis method is that the majority of the faults don't happen over a long period of time. Roughly 30% of the files resolve in less than 3full cycles(64 samples per cycle) which the half-cycle analysis isn't always able to pick up on. This was the start of the Quarter Cyle Evaluation, which will be referred to as the SMT method.

CHAPTER 4

QUARTER CYCLE ANALYSIS, THE SMT FORMULATION

SMT allows for a breakdown of Half-Cycle Analysis so that a higher tier of accuracy can be obtained for start time and end time. With respect to the debate of Speed vs. Accuracy, Half-Cycle can be accurate. But it isn't precise. SMT bridges the gap of accurate vs. precise by looking at 124 calculated RMS values instead of the 62 calculated RMS values for Half. While having double the RMS values to check, this is still quicker than 1981 numbers, and is still able to be debugged with relative ease. The core idea of Quarter-Cycle Analysis is understanding that Current waveforms tend to follow a predictable pattern. Appendix A, Formula A.1 has the proof for expanding the Half-Cycle formulation into Quarter-Cycle Analysis, but the end formula is copied below for reference.

$$RMS_{quarter} = \sqrt{\frac{A+B}{2}} = \frac{Peak_{Value}}{\sqrt{2}};$$ Formula 4.1

Since it is known that most wave forms are usually sinusoidal in nature, it is logical, in terms of breaking Half-Cycle Evaluation into an expanded version, that the variables "A" and "B" should be roughly equal to each other. With this inferred, variables A and B can then be written in a new formulation.

$$Assuming\ A = B, A = \frac{{P_v}^2}{2} = B$$ Formula 4.2

This allows for a very easy check to validate start time and end time which were the biggest weak spots of Half-Cycle Analysis. Figure 4.1 shows the RMS values in regards to Quarter-Cycle Analysis.

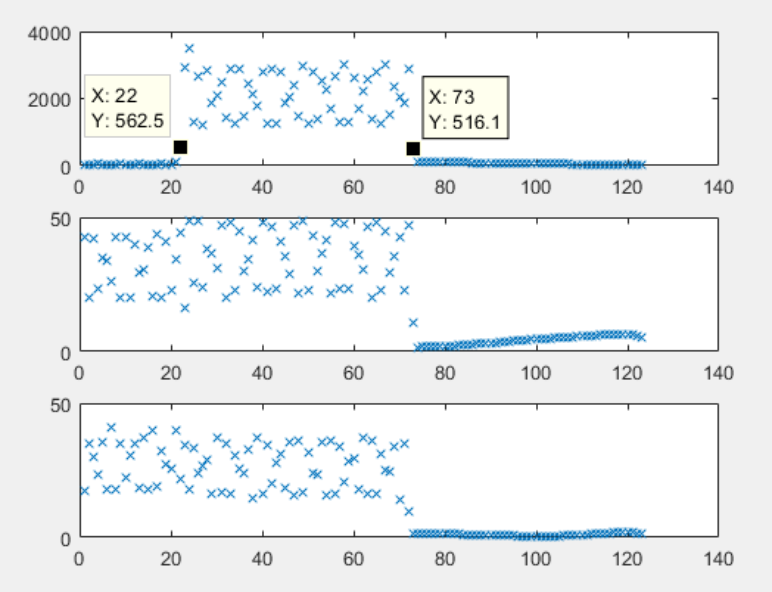


Figure 4.1 SMT Analysis of Fault E6

It is seen that the starting point happens at the end of the 22nd RMS Value and ends at the beginning of the 73rd RMS Value. This gives a cycle duration equal to 12.25 cycles(roughly 0.195 seconds) which is different than what was seen by Half-Cycle (0.1858 Seconds). The milliseconds difference between SMT and Half Cycle Analysis shows dramatically improved accuracy as will be seen in greater detail later. Since Wavering Error was a problem with half-cycle, and there was a defined limit for wavering error, the wavering error for SMT will follow the same definition as defined in Formula 3.1

Wavering Error, Quarter Cycle

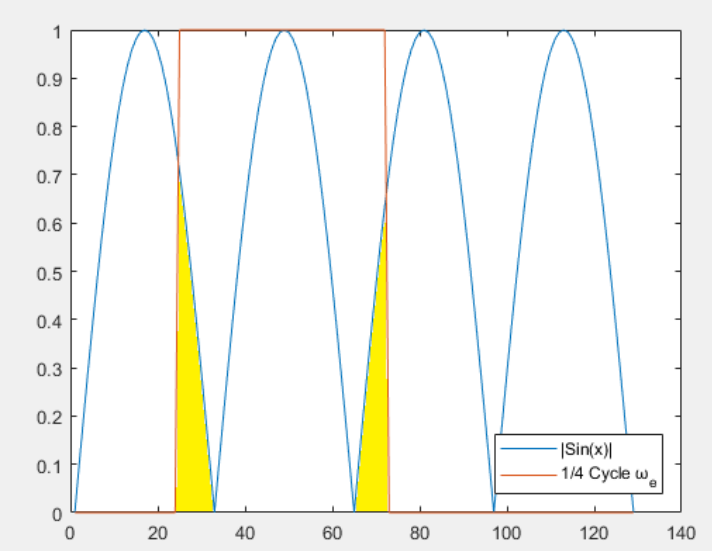


Figure 4.2 Wavering Error Representation for SMT

The shaded area noted above is the expected wavering error for SMT. However, it will be seen that this is not the actual Wavering Error. Wavering Error, mathematically is 8 samples for Quarter-Cycle which correlates to roughly 4 milliseconds of error. If the fault happens within the last, or first, 8 samples, there are not going to be many false hits that cause a misread. This causes a high level of accuracy to ultimately be seen. Taking the values for start time, end time, and magnitude in relation to Fault E6 Half Cycle Analysis, Quarter-Cycle Analysis and the E6 Solutions, the following table is found.

RMS vs. SMT

Table 4.1 RMS vs. SMT vs. Actual Values for Fault E6

	Magnitude	Start Time	End Time
Actual	3295	0.07917	0.2635
Half-Cycle	3388.5	0.0969	0.2908
Quarter Cycle	3295.2	0.0808	0.2686

For fault E6, there is no comparison. Half-Cycle is accurate, but the SMT method is precise. Comparing Quarter-Cycle against the normal Distribution with the exact same specifications defined for Table 3.2 yields Table 4.2.

Table 4.2 SMT Results

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

The results from Table 4.2 are very powerful when compared against Table 3.2. Yet there are differences that should be noted. Starting with the difference in Magnitude, it can be seen that Half-Cycle Analysis has a higher level of accuracy in regards to the SMT method. The reasoning for this can be noted from Wavering Error. During the fault, cycles aren't always the expected sample length. The fix to this would be to have a code that instead of counting in 32 samples, was able to look between each minimum point and do the RMS formulation for those ranges in terms of half-cycle and quarter-cycle analysis. There are other methods that could bypass this issue, such as handling the fault analysis through other methods(i.e. Digital evaluations[6])There was a work around that was implemented and that that brings the accuracy for Magnitude on par with Start time and End Time accuracy. Since the Start time and End time was incredibly precise, and therefore able to give a pretty accurate sample range, Matlab instead looks between the sample ranges and finds the maximum value. This still has some bugs that are relative to the samples per RMS value, but it fixes the majority of known issues.

Looking at start-time and end-time, there is a drastic change within the accuracy when comparing Half-Cycle and the SMT method. Within 1 standard deviation, there are 94% of files for start time, and 96% of files for end time. The drastic jump from Half-Cycle for these variables relates to the negligible wavering error for SMT that rarely causes a false hit. Yet, the issue of RMS value lengths not being what they are expected to be prevents the desired perfection aesthetic. Matlab has been instructed with the SMT() function call to group by a set number of samples per RMS value. Fixing this would dramatically change the accuracy and have Quarter-Cycle evaluation see close to 100% accuracy.

SMT Limitations

Now, for the few faults that can't be read accurately by Matlab. The question of how one might begin a solution to solve the 3% would begin on understanding why the fault happens. Let's take the following fault as an example. Note, the Magnitude isn't considered for this fault.

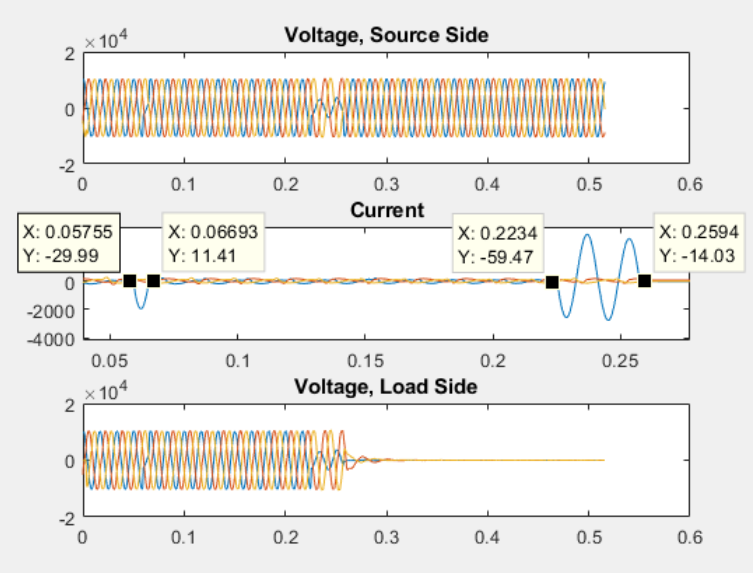


Figure 4.3 The Fault that Breaks SMT

Looking at the first pulse, it can be seen that the pulse lasts nearly 10 milliseconds. This correlates to 18.58 samples. Yet, the wave itself is a full half-cycle which should be 32 samples by our eyes. This pulse itself happens to quick where it can't be seen by Half-Cycle Analysis since the numbers don't affect the average under the square root enough to trigger a fault. Quarter-Cycle sees the fault, but because there is an extra 3 samples, the accuracy is off. Instead, the output obtained from the script, with respect to Quarter-Cycle Analysis is as follows:

Table 4.3 The 3% Errors.

	Start Time	**End Time**
1st Fault - Script	.0646	.0727
1st Fault - Actual	0.05755	0.06693
2nd Fault - Script	0.2342	0.2696
2nd Fault - Actual	0.2234	0.2594

Since the script grabs by every set amount of samples, there are some samples that are left out. This causes a misread by the script where the time duration is accurate, but not precise as should be expected from the quarter-cycle analysis method in regards to start time and end time. This misread carries through to the conception of the start time for the 2nd fault on this file which then effects the start time of the end time. The attempted solution for this error was the implementation of eighth cycle analysis. However, Eighth Cycle Analysis takes much more processing power and the accuracy obtained from Eighth Cycle Analysis isn't worth the time it takes in comparison to SMT Analysis.

CHAPTER 5

EIGHTH CYCLE ANALYSIS, THE PERFECTIONIST'S HAPPY PLACE

Having the ability to read what was roughly 98% of all files available through SMT Analysis was a remarkable achievement. However, the last 2% still remained elusive and too much of a temptation to pass up. Everything that follows is purely theoretical. There are ways to validate this method, but the change in percentage read by Eighth-Cycle Analysis proved to be a much higher cost in speed vs. accuracy. The formulation for Eighth-Cycle Analysis can be found in Appendix A, but the short answer is as follows.

$$RMS_{Eighth} = \sqrt{\frac{a+b+c+d}{4}} = \frac{P_v}{\sqrt{2}} \quad \text{Formula 5.1}$$

This has a simplified state where in terms of symmetry on a Sinusoidal wave, the following can be assumed:

$$\begin{gathered} a = d \\ b = c \\ a + b + c + d = 2{P_v}^2 \\ a + b = A \\ c + d = B \end{gathered} \quad \text{Formula 5.2}$$

These assumptions allow for a proximity of values in regards to implementing Eighth-Cycle Analysis. Other than the previous fault where the fault that happens in 18 to 19 samples, the main selling point of using this method is for files that have been nicknamed "The Ant Fault" files. An example of this is as follows in figure 5.1

"The Ant Fault"

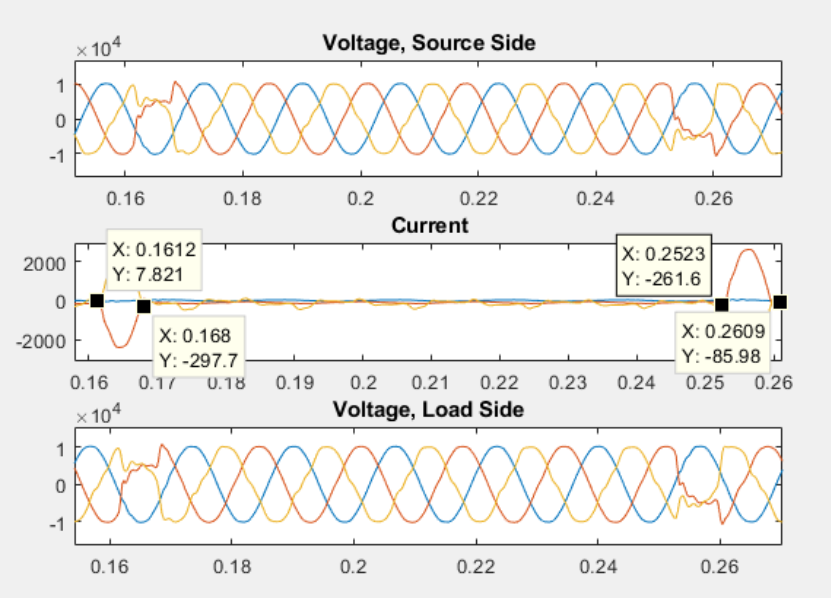


Figure 5.1 Graphical Representation of "The Ant Fault"

These types of faults earned their nickname because they last for roughly 8 milliseconds and the only logical explanation for this fault duration is from the time it would take for an ant to be turned to ash as it is caught between two power lines that lightly bump each other in the wind. In general, these faults are few and not incredibly important. Yet, these faults happen too quickly for Quarter-Cycle to catch. This is where Eighth-Cycle analysis could come into play. If the program gets an empty output array, or a fault duration of less than 1.5 cycles, it could check and validate the fault to improve accuracy. This would be the best implementation for Eighth-Cycle analysis since the return on investment for Eighth Cycle analysis isn't worth the time it takes; especially since Eighth-Cycle analysis tends to just validate what the Quarter-Cycle Analysis has defined already. Comparing Eighth-Cycle analysis against the normal distribution yields the Table 5.1 below.

Eighth vs. Quarter Cycle

Table 5.1Eighth Cycle Results

Actual Values	Magnitude	Start Time	End Time
1 St. Dev.	0.7312	0.9421	0.9658
2 St. Dev.	0.9124	0.9657	0.9856
3 St. Dev	0.9410	0.9832	0.9910

Comparing this against Table 4.2, there are no notable changes in accuracy and the return on investment isn't worth the exponential increase in time needed to run the Eighth-Cycle Analysis. The specified time required for each method will be discussed in chapter 6 in greater detail, but for one basic college student's computer to run 300 files automatically, it took about double the amount of time as it did that same computer to run the program in Quarter-Cycle mode.

To summarize Eighth-Cycle Analysis, it is the perfectionist's happy place in regards to this project. It could be managed and perfected, but there would need to be a much higher level of computing power available as well as a necessity to solve the quirks that plague Quarter-Cycle and Half-Cycle. Furthermore, while Eighth-Cycle Analysis can handle files such as the infamous "Ant Fault" files, it's not practical when Quarter-Cycle can handle 97-98% of all the files.

CHAPTER 6

RESULTS & CONCLUSIONS

In conclusion, the Square Mean Test (SMT) is 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 (SMT) analysis has diminishing returns. These diminishing returns show in a higher level of processing power needed than would be considered practical. The following table shows the theoretical computing time required for each grouping of samples extrapolated from the time required for Half,Quarter, and Eighth Cycle Analysis.

Table 6.1 Time Required per Cycle Evaluation

	32 Samples	16 Samples	8 Samples	4 Samples	2 Samples	1 Sample
Time(Minutes)	3.4	5.8	12.2	22.6	37	55.4

With the biggest error for this program known as trying to find a way to handle the inconsistent half-cycle sample lengths, anyone who continues with this research would be encouraged to start there. It would be expected that the resolution of this issue would bring Eighth-Cycle analysis onto the map as a viable method for Automated Transmission Line Fault Analysis. However, until that day, The SMT method, or Quarter Cycle Analysis will have to do.

Out of the 300 files tested, it was found that 180 of them were found to be LG faults, 75 of them were found to be LL faults, and 45 of them were found to be LLL faults. An example of a LL, and a LLL fault can be found as Figure 6.1 below.

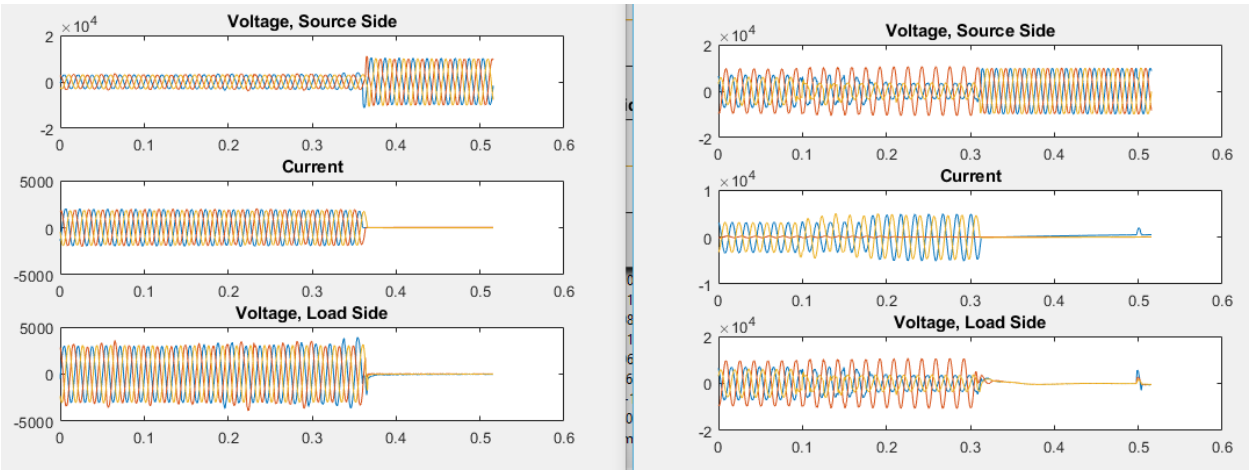


Figure 6.1 LLL(Left) and LL(Right) Representation

In conclusion, it can be stated that the most reliable and practical method for fault automation turns out to be the RMS method, specifically, its expansion into the SMT method. The SMT method adds precision and efficiency into classification within reasonable times. More research is needed to address the accuracy/time conflict. There are cases where extreme accuracy might be worth the time investment(Fuse Forensics Analysis), but it remains an engineering decision backed by the studies to determine what type of analysis is appropriate for a desired outcome.

REFERENCES

1. IEEE Standard Common Format for Transient Data Exchange (COMTRADE) for Power Systems," in IEEE Std C37.111-1999 , volume, number, pp.1-55, 15 Oct. 1999 doi: 10.1109/IEEESTD.1999.90571

2. Glover, J. Duncan, Mulukutla S. Sarma, and Thomas Overbye. Power System Analysis & Design, 6th edition. Cengage Learning, 2016.

3. Hart, Daniel W. *Power electronics*. Tata McGraw-Hill Education, 2011.

4. MathWorks, Inc. *MATLAB: The language of technical computing. Desktop tools and development environment, version 7*. Vol. 9. MathWorks, 2005.

5. Baker, John, and Mike Meisinger. "Experience with a Distributed-Intelligence, Self-Healing Solution for Medium-Voltage Feeders on the Isle of Wight." *2011 2nd IEEE PES International Conference and Exhibition on Innovative Smart Grid Technologies*. IEEE, 2011.

6. Meena, Radhey Shyam, and M. K. Lodha. "Unsymmetrical Fault Analysis & Protection of the Existing Power System." *International Journal of Multidisciplinary Research and Modern Education*, Issue 1, Vol. 1(2015).

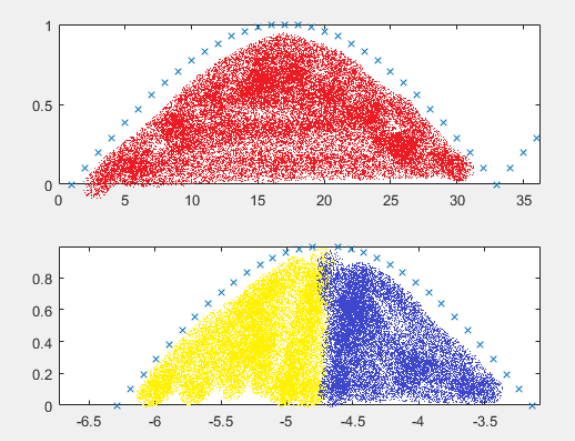
APPENDIX A

FORMULAS

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

Given: $RMS = \sqrt{\frac{\sum_{i=1}^{n} y(1,i)^2}{n}}$; where for all practical purposes n is 32 samples for half cycle evaluation,

One 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:

$RMS_{quarter} = \sqrt{\frac{\left(\frac{\sum_{i=1}^{n/2} y(1,i)^2}{n/2} + \frac{\sum_{i=\frac{n}{2}+1}^{n} y(1,i)^2}{n/2}\right)}{2}}$; again assuming that n = 32 or for quarter cycle n = 16.

Rewriting this equation in a neater fashion yields the following:

$$RMS_{quarter} = \sqrt{\frac{\frac{\sum_{i=1}^{16} y(1,i)^2}{16} + \frac{\sum_{i=17}^{32} y(1,i)^2}{16}}{2}}$$

Now, let's put a practical number for RMS. With the assumption to use this quarter cycle evaluation to evaluate "Y = sin(x)." The amplitude of this function is defined as the peak value. It is therefore expected:

$$RMS = \frac{p_v}{\sqrt{2}} = \frac{\sqrt{2}}{2} * p_v \cong 0.7071 * p_v$$

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

$$RMS_{quarter} = \sqrt{\frac{\left(\frac{\sum_{i=1}^{16} y(1,i)^2}{16} + \frac{\sum_{i=17}^{32} y(1,i)^2}{16}\right)}{2}} = \frac{1}{\sqrt{2}} * p_v;$$

$$RMS_{quarter} = \sqrt{mean(\ mean(y(1,1:16)^2) + mean(y(1,17:32)^2)\ )} = \frac{1}{\sqrt{2}} * p_v;$$

$$Letting \begin{matrix} A = mean(y(1,1:16)^2) \\ B = mean(y(1,17:32)^2) \end{matrix}$$

$$RMS_{quarter} = \sqrt{\frac{A+B}{2}} = \frac{1}{\sqrt{2}} * p_v;$$

Or in other words, A+B should equal to Peak value squared. Why this is important, is that the moment these values don't equal to the peak value, there is a fault starting, and the moment A+B returns to equal the peak value, the fault has ended.

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

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

$$RMS_{Eighth} = \sqrt{\frac{a+b+c+d}{4}} = \frac{p_v}{\sqrt{2}};$$

Which comes from letting a+b = A and c+d = B.

By doing this, it is found that

$$\frac{a+b+c+d}{4} = \frac{{p_v}^2}{2};$$

Which by implementing Symmetry of Sinusoidal waveforms to let the following statements be true.

$$a = d$$
$$b = c$$
$$a + b = A$$
$$c + d = B$$

It can be found that:

$$a + b = 2 * {P_v}^2$$

However, a second equation is needed between these two variables to find a noticeable and reliable pattern to the RMS formulation for Eighth Cycle. Instead, what was found was a range of numbers that was later verified by a fellow graduate student from the Math Department at UTC. Before stepping into their method, let's talk about the pattern that emerged while working through thousands of files.

	RMS Formulation	Range
1/2 Cycle(32 Samples)	$\frac{P_v}{(\sqrt{2})^1}$	0.7069 - 0.7073,average of 0.7071
1/4 Cycle(16 Samples)	$\frac{P_v}{(\sqrt{2})^2}$	0.45 - 0.55, average of 0.5
1/8 Cycle(8 Samples)	$\frac{P_v}{(\sqrt{2})^3}$ or $\frac{P_v}{(\sqrt{2})^4}$	0.354 - 0.25, average of 0.3

This provided a baseline that up until eighth cycle, proved infallible. But having it in a range and not being able to understand why proved to be frustrating. What follows is the proof that gives credibility to the number ranges that were found to be true for a large amount of files.

With the observed results not fitting the predicted pattern as seen when Eighth-Cycle Evaluation is implemented, there has to be another relationship. In essence, RMS, which uses a summation is an integral. Therefore, focusing on the ratio's of integrals with respect to the previous method's sample range yields the following.

Since RMS for Half-Cycle is assumed to be correct with no issue, there is no proof needed. Since this is a Sine wave, the range for half-cycle ranges over 32 samples or from 0 to pi.

Therefore, when approaching Quarter-Cycle Analysis, using the following ratio of integrals yields the following:

$$\frac{\int_0^{\pi/2} \sin(x)\,dx}{\int_0^{\pi} \sin(x)\,dx} * P_v = \frac{1}{2} * P_v \; which\ is\ within\ the\ expected\ range\ found\ previously.$$

Applying the same ratio's to Eighth Cycle Analysis yields the following:

$$\frac{\int_0^{\pi/4} \sin(x)\,dx}{\int_0^{\pi/2} \sin(x)\,dx} * P_v \cong \left(\frac{2-\sqrt{2}}{2}\right) * P_v$$

$$\cong 0.293 * P_v \; which\ is\ within\ the\ expected\ range\ found\ previously$$

Since integrals are fundamentally integrals, the ratios of integrals can be used to validate mathematically the ranges found from the SMT/RMS Method.

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

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

$c = number\ of\ cycles\ per\ half\ cycle$. for half cycle, this means c = 2; for quarter cycle, this means c = 4; and for eighth cyle, this means c = 8;

$n = number\ of\ faulted\ RMS\ values$

$$Cycle\ Duration(C_d) = \frac{n}{c}$$

$$Time\ Faulted = C_d * \frac{64}{C} = \frac{64n}{C^2}$$

APPENDIX B

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)),s
tring(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_t
ime,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_fa
ulted)};"
% 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_s
mt),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_sa
mple)];

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(s
tarting_sample)];
  for i = 1:length(ending)
    fault_summary(1:4,i) =
[cellstr(fault_classification(1,i));cellstr(num2str(lines_faulted(:,i)'));num
2cell(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 Master's 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