Iowa Research Online
The University of Iowa's Institutional Repository

University of Iowa
Iowa Research Online

Theses and Dissertations

Spring 2013

A virtual predictive environment for monitoring reliability, life time, and maintainability of printed circuit boards

Amer Basheer Dababneh
University of Iowa

Copyright 2013 Amer Basheer Dababneh

This thesis is available at Iowa Research Online: http://ir.uiowa.edu/etd/2470

Recommended Citation
Dababneh, Amer Basheer. "A virtual predictive environment for monitoring reliability, life time, and maintainability of printed circuit boards." MS (Master of Science) thesis, University of Iowa, 2013.
http://ir.uiowa.edu/etd/2470.

Follow this and additional works at: http://ir.uiowa.edu/etd

Part of the Industrial Engineering Commons

A VIRTUAL PREDICTIVE ENVIRONMENT FOR MONITORING RELIABILITY, LIFE TIME, AND MAINTAINABILITY OF PRINTED CIRCUIT BOARDS

by

Amer Basheer Dababneh

A thesis submitted in partial fulfillment
of the requirements for the
Master of Science degree in Industrial Engineering
in the Graduate College of
The University of Iowa

May 2013

Thesis Supervisor: Professor Ibrahim T. Ozbolat

Copyright by

AMER BASHEER DABABNEH

2013

All Rights Reserved

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

MASTER'S THESIS

This is to certify that the Master's thesis of

Amer Basheer Dababneh

has been approved by the Examining Committee
for the thesis requirement for the Master of Science
degree in Industrial Engineering at the May 2013 graduation.

Thesis Committee: ______________________________
Ibrahim T. Ozbolat, Thesis Supervisor

____
Yong Chen

____
Hongtao Ding

____
Timothy Marler

To my dad soul

The future belongs to those who believe in the beauty of their dreams

Eleanor Roosevelt

ACKNOWLEDGMENTS

To God, for opening for me a new era in life and for giving me guidance to walk through the unknown. I would like to express my deepest gratitude to my advisor, Professor Ibrahim Ozbolat, for his generosity, faith, and superb guidance. Thank you for your continuous support and your persistent encouragement for me to pursue the maximum prospect in this work. I could not have reached to this point without your relenting supervision.

For his generous assistance, Dr. Timothy Marler. His assistance was way beyond being thanked by words. I would like to thank Professor Yong Chen for his kind assistance. Professor Hongtao, your participation in my defense committee is highly appreciated. Also I would like to acknowledge the Electric Power Research Institute (EPRI) for their financial support in this research. I am most grateful to Mr. Ben Goerdt, and Mr. Matthew Denney for their contributions to PREVIEW software and architecture, their efforts are highly appreciated. Last but not least, I would also like to thank my parents, and friends. Their encouragement and support gave me strength through my journey in perusing my studies.

ABSTRACT

This thesis highlights and aims to develop an immerse and high fidelity virtual environment for lifetime and reliability analysis of circuit cards for nuclear power electronics. Where sue to the lack of accuracy in electronic reliability standards and the absence of a system level reliability and lifetime of PCB, the developed virtual environment allows prediction of total life time, overall reliability and maintainability for circuit cards (system level) and their components (component level) through a simulation methodology. Component repair or replace approaches are used within this simulation which gives the user the ability to choose between them based on experience and component history. As excessive temperature is the primary cause of poor reliability in electronics, quantitative accelerated life tests are designed to quantify the life of circuit cards under different thermal stresses to produce the data required for accelerated life data analysis. This research provides a better understanding and helps in predicting overall system failure characteristics for any given configuration. It allows the user to identify components which contribute the most to downtime and to determine the effect of design alternatives on system performance in a cost-effective manner.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Product reliability is the engine of quality and effectiveness, so many manufacturers and companies spend millions of dollars on product reliability. The biggest amount of management and engineering efforts goes into evaluating reliability, identifying the root causes of failure, making manufacturing changes and, comparing designs, manufacturing methods, materials and the like.

According to Ebeling, reliability was defined as "the probability that a device or system will perform a required function for a given period of time, when operated under specific conditions" [1]. In other words, reliability can be defined as a quantitative measure of non-failure operation over a given operating time interval [1]. It is important to note that specific principles and criteria have been established earlier to specify what the intended function of the item is. Furthermore, reliability has to be quantified for a specific period of time where time is considered as the major variable which has an important effect on reliability; a product becomes less reliable as its number of operating hours, without being switched off for maintenance, increases.

The most common and feasible way of expressing reliability is the mean time between failures (MTBF). Which is the mean operating time (up time) between failures of a specified item of equipment or a system [1, 4, 5]. On the other hand, failures in time unit, (failure rate(λ)) should be examined. Failure rate changes through the life of the product and gives the familiar bathtub curve, as shown in Figure 1. This curve shows the failure rate - operating time for a population of any product. It is the manufacturer's responsibility to ensure that product in the "infant mortality period" does not get to the customer. This allow only the product with a useful life period during which failures occur randomly (i.e. λ is constant) to be received by the customer. Finally, a wear-out period is reached after the product's useful life, where the failure rate increases [2, 3, 4, 5].

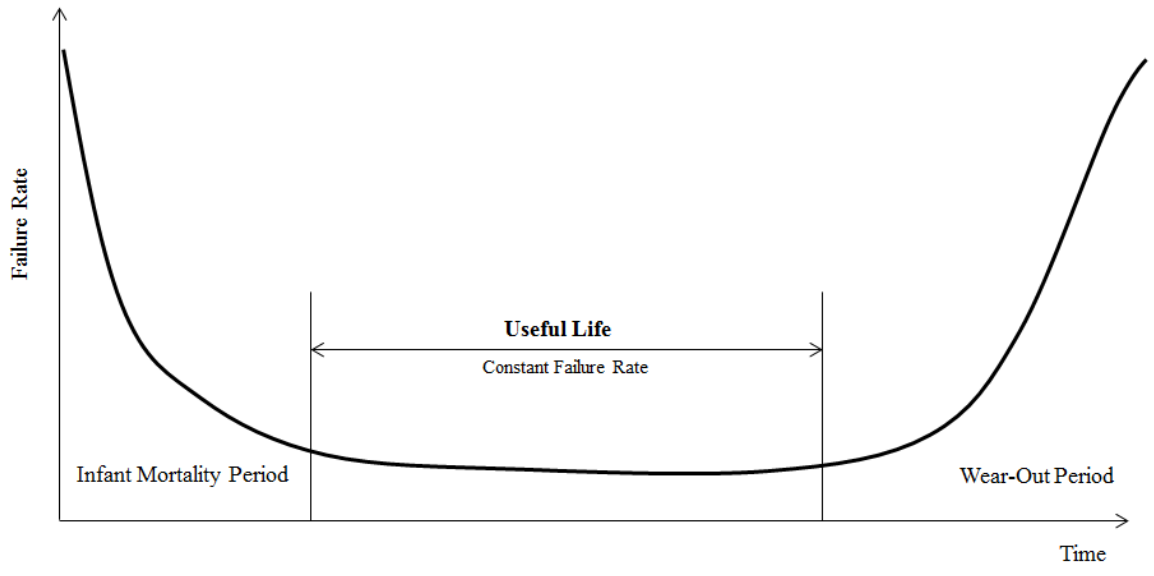


Figure 1. Bath Tube Curve

During design, development, and operational phases, this information can be very valuable to the product designers and users. Designers can use it to guide their design and find the weak points of the design, and users can use it to setup maintenance plans and schedule.

1.1. Background

Reliability of electronic devices has become a major issue of concern to manufacturers due to factors such as miniaturization, changes in manufacturing materials, and usage. Printed circuit boards (PCBs) have been widely used as the major building block of electronic equipment in modern complex systems such as aircraft and automobiles. PCB reliability plays a vital role in entire system reliability. Therefore, effective and accurate PCB reliability becomes an essential issue for many companies and users.

A lot of nuclear power plants which are in operation nowadays were built with technologies from the 1960s and 1970s. Obsolescence, component failures due to aging, lack of spare parts (original equipment manufacturer support), and high maintenance costs, have resulted in the need to a new technology in current nuclear power plants to support components reliability and maintainability, therefore; allowing these plants to operate productively and efficiently. Equipment related problems such as component failure have become very critical issues in nuclear power plants, as the reduction in plant performance is due to part and equipment failure and system performance degradation. On the other hand, nuclear power plants are being forced to achieve lower maintenance costs and delay the replacement of obsolete equipment. In nuclear power plants, circuit card malfunctions and failures can result in power reductions and other plant challenges, causing losses of up to a million dollars per day. Therefore, having a predictive life time and reliability model reduces the time and cost associated with maintaining and planning for both in-service and new circuit cards, which comprise one of key subsystems in nuclear reactors [41,42]. That, PCB reliability and maintainability modeling in nuclear power plants becomes an important issue to analyze and visualize the performance and remaining life time of in-service and new PCB of those plants.

1.2. Literature Review

One of the most effective way to examine the reliability of the product is through reliability testing, however many factors restrict the usage of this method; First, this kind of testing is expensive and time-consuming for the majority of products, especially electronic equipment [15]. Secondly, the accuracy of the results relies on the sample size [16]. Therefore, using the analytical method to quantify product reliability becomes a desired tool, particularly for product designers.

Although a plethora of research has been conducted to improve PCB reliability in the context of solder joint reliability and its fatigue life [17, 18, 43], limited research has been performed at the board level, as up to my knowledge, no studies conducted on bored level reliability and lifetime of PCB. The design for reliability in PCB was accomplished with the introduction of reliability prediction tools in the 1960s. The most widely used tools were: Military Reliability Prediction of Electronic Equipment (Mil-HDBK- 217) [6], Telcordia (Bellcore) TR-332 [7], and PRISM [9, 14].

The MIL-HDBK-217 was developed by the US Department of Defense (DoD) with the assistance of military departments, federal agencies, and industry for use by the electronic manufacturers supplying to the military [6]. It provides failure rate models for nearly every conceivable type of electronic component, such as integrated circuits (ICs), transistors, diodes, resistors, capacitors, relays, switches, and connectors. These failure rates have derived mainly from test bed and accelerated life; this data is then analyzed and used in creating usable models for each part type. The first edition of MIL-HDBK was issued in 1965 as MIL-HDBK-217A, and there was only a single point failure rate of for all monolithic integrated circuits, regardless of the stress, the material, or the architecture. In 1973 MIL-HDBK-217 B was published because it was clear to researchers that any reliability model should reflect device fabrication technology, geometries, and materials. Notice changed was observed where editions C, D, E and F of this handbook were published in 1979, 1982, 1986 and 1995 respectively [10].

Based on this handbook standard, various commercial software applications were implemented to facilitate the estimation of product reliability [8]. This handbook presents a straightforward equation for calculating the failure rate, includes models for a broad range of part types, provides many choices for environment types, is accepted and known worldwide, and is used by commercial companies and the defense industry.

However, the lack of accuracy and slow pace of updating the databases have limited the usage of these methods. Currently MIL-HDBK-217F Notice 2, dated 28 Feb

1995 is an active Military Handbook; this handbook has not been modified since 1995. Nevertheless, MIL-HDBK-217 is still the most widely known and used reliability prediction tool; most practitioners are still using it and want to continue to use it as a relatively simple prediction method. A crane survey shows that almost 80 percent of the respondents use the Military handbook, while PRISM and Telcordia are second and third [9].

Pecht [10] suggested abandoning the usage of the MIL-HDBK-217 after reviewing the development history of the reliability quantification methods, the advances in technology, and the drawbacks of these methods. His conclusion may be overstated, but he point out some problems embedded in these methods. For example, he discovered that the base failure rates of the parts were not updated quickly to reflect technological advances. Because filed data takes too long to collect and the models cannot be extrapolated, up-to-date collection of the pertinent reliability data is in itself a major undertaking, especially with rapid improvement in products. Leonard [11] reported that the MTBF predicted using MIL-HDBK-217 for a full authority electronic control (10,889 hr) was about three times different than field-observed MTBF (30,000 hr). The main reason for the disagreement is that the base failure rates of the parts are not updated as technology advances.

As stated previously, Telcordia (Bellcore) is another widely used tool in electronics reliability prediction. The Telcordia (Bellcore) standard was originally developed by AT&T Bell Labs and it focuses on equipment for the telecommunications industry. It reflects the failure rates that AT&T Bell Lab equipment was experiencing in the field [12]. This standard was primarily developed based on telecommunications data and supports only a limited number of ground environments (five environments that are only applicable to telecommunications applications) [13], whereas the MIL-HDBK-217 covers fourteen.

Telcordia Issue 2, which was released in September 2006, replaced Telcordia Issue 1, 2001. Its concept is similar to MIL-HDBK-217 [14]. However, fewer part models are covered in Telcordia than in MIL-HDBK-217, for example, but not limited to, some semiconductors, resistors, and many relays, switches are not supported by this standard [14].

PRISM is the third tool reliability prediction; its approach was released based on DoD Reliability Analysis Center's (RAC) databases. Currently, it is still gaining approval, and no reference standard is available. PRISM available as an automated tool; however, it is limited by the fact that few sets of electronic parts models are available, and no models are available for relays, inductors, switches, connections, and many semiconductors [14]. The latest version of the method, which is available in a software version, was released in 2001 [9]. These practices have provided a gap in producing field reliability prediction, so a reliability prediction methodology is presented in this thesis.

On the other hand, failure normally implies a corrective maintenance action corresponding to the repair time needed to bring an element back to regular operation and to improve its reliability. Therefore, while reliability is a measure of non-failure operation of an item, maintainability is related to its capability of being repaired or to its need of receiving maintenance. It's a characteristic of design and installation that determines the probability that a failed component or system can be restored to its normal operable state within a given timeframe [19].

PCB can be roughly divided into three sections: board, interconnections between the board and the parts on the board. Therefore, the reliability of the PCB can be found by studying the failure modes of each section. However, in this research we assumed that the PCB failures are only due to the failures of its parts; assuming that the mechanical and electrical connections between parts or between parts and the board are perfect.

During the useful life of an electronic part, its reliability is a function of several stresses, including but not limited to the electrical, mechanical, and thermal stresses to which the part is subjected [44, 45]. In particular, an increase in thermal stresses directly increases the failure rate and ultimately decreases the reliability dramatically [20]. High temperatures impose a severe stress on most electronic items since they can cause not only catastrophic failure (such as melting of solder joints), but also slow, progressive deterioration of performance levels due to chemical degradation effects. Kallis and Norris stated that "excessive temperature is the primary cause of poor reliability in electronic equipment" [21]. For example, for every 10° Celsius rise in temperature, the failure rate of most electronic components doubles [22]. Electrical and electronic parts such as circuit boards must be selected with proper temperature, performance, reliability, testability, and environmental characteristics to operate correctly and reliably when used in a specific application. To achieve the desired performance and reliability of a PCB, we have to test it under the expected extremes of operating stresses, including vibration, temperature, mechanical and electrical stresses, and sand and dust [46].

DeGroot and Goel have introduced the concept of accelerated tests [23], where the product is first run at use condition and, if it does not fail for a specified time, is then run under a higher accelerated condition, and so on until it fails. In this research, quantitative accelerated life tests are designed to quantify the life of the PCB under different thermal stresses and produce the data required for accelerated life data analysis. Considering that most of the PCBs will keep working for a long period of time after they are turned on, the transient temperature effect is ignored and the steady-state case is only considered, which means we assume that the temperature is constant.

1.3. Thesis Overview and Objectives

Due to the obsolescent and lack of accuracy of electronic reliability standards, especially the MIL-HDBK-217, and because of the reason that there is no system level model for PCB reliability and life time, this research aims to develop a predictive remaining lifetime, reliability and maintainability analysis model of circuit cards for nuclear power electronics, for both component and system level, where a circuit board methodology is presented to find out the reliability, maintainability and life time of PCB at component and board level. As the reliability for each component in a PCB is calculated for a specific time based on component age and the best fit distribution function of the “time to fail (TTF)” data for each component.

A board-level methodology is integrated within an immersive visual environment called Predictive Environment for Visualization of Electromechanical Virtual Validation (PREVIEW). PREVIEW is an interactive 3D environment that includes predictive physics based on capabilities to support virtual testing of PCBs [24]. It enables product designers to assess potential design shortcomings based on virtual physics-based test capabilities, thus reducing the time and cost associated with developing and testing several iterations of prototypes prior to production. This gives the benefit of flexibility and capability to perform a large number of “what-if" computations for early evaluation of the occurrences and analysis of the causes, minimizing the risk of the flight test activities, simulating hazardous conditions, evaluating the manufacturing process, and performing capacity analysis. In this research, PREVIEW is used as a software package that displays the developed model and offers a versatile environment that accepts modifications. This will enable new applications and interfaces with tiered solutions that can be easily implemented and eventually provide significant improvement in the reliability, maintainability, and lifetime of the PCB and its components. It helps in providing reliability data, predicting remaining lifetime, generating maintainability policy, and integrating with virtual reality systems.

This thesis is outlined as follow: In chapter two, component and system level lifetime and simulation methodology is presented, chapter three introduces component and system level reliability, where the component and system level maintainability presented in chapter four , in chapter five, thermal stress and its effect on PCB lifetime and reliability is presented, and summary are drawn in chapter six.

CHAPTER II

COMPONENT AND SYSTEM LEVEL LIFETIME ANALYSIS USING SIMULATION METHODOLOGY

It is well-known that any device that requires all of its parts to function will be less stable than any of its parts. Quantifying and examining PCB part life time correctly plays a significant role in quantifying accurate system steadfastness. However, and due to the limitation of data availability, time constraints, lack of experience, and many other factors, part and system life time has become an issue for many companies. In this research, the most available data to solve this issue is used, where a set of TTF sample data is used for each PCB component.

2.1. TTF Behavior of PCB Components

Engineering issues, such as the lifetime and reliability of a component in-service, often involve intrinsic failure mechanisms that are not exactly known. In nature, the occurrence of some engineered event is often imperfect; indeed, it may seem to occur in random, but when it is observed over a large sample or over a long period of time, there may appear a definitive "mechanism" which causes the event to occur. Sampling of a set of relevant data (observation) can provide a statistical base from which at least the nature of the mechanism may become more evident. The alternative is to estimate the behavior using some techniques including data sampling. One simple strategy to determine the data behavior is to know its distribution. Therefore, the collected TTF data is fitted to the cumulative distribution function F (t) to find its best fit distribution. There are two main approaches to check statistical distribution assumptions [25]. The first approach is via empirical procedures, which are based on the intuitive and graphical properties of the distribution. The second is the goodness of fit tests (GoF), which are based on statistical theory. GoF approach is considered more formal and reliable for assessing the underlying distribution of a data set [26]. GoF tests are essentially based on either of two distribution elements: the CDF or the probability density function (PDF). The Anderson-Darling (AD) and Kolmogorov-Smirnov (KS) methods are the most common GoF tests; they use the CDF approach and therefore belong to the class of “distance tests” [27, 28].

2.1.1 KS Method in TTF of PCB Components

In this research, the KS test method has been selected among other distance tests for the following reasons. First, it is among the best distance tests for a small number of data points. It can be easily computerized, and it is very versatile; any continuous distribution can be fit with it, where various statistical packages use it [27, 28].

To find the best fit distribution function for each PCB component, TTF data points are sorted in ascending order, and the empirical CDF (F_n) is found for each component data using the mean rank method (i/n + 1), where n is the number of data points; this method is used instead of equal rank method, as the equal rank is somewhat biased, especially when n is small. Using the mean rank assumption allows the possibility that one or more virtual data point may be present in between the i^{th} and $(i+1)^{th}$ data point. In particular, there is at least one data point below the lowest ranked data point and there is at least one data point above the highest ranked data point. Then, the most electronics appropriate and usable distributions (Weibull, exponential, normal, and lognormal) parameters were estimated in order to find the theoretical CDF (F_0) for each PCB component data set as stated in Table 1 [29].

Table 1. Theoretical CDF (Unreliability)

Distribution	Theoretical CDF (F_0(x)) (unreliability)	Parameters
Normal	$\frac{1}{2}(1+\mathrm{erf}(\frac{(x-\mu)}{\sigma\sqrt{2}}))$	$\mu = \frac{\sum_{i=1}^{n} X_i}{n}$, $\sigma = \sqrt{\frac{\sum_{i=1}^{n}(X_i-\mu)^2}{n}}$
Lognormal	$\frac{1}{2}(1+\mathrm{erf}(\frac{(lnx-\mu)}{\sigma\sqrt{2}}))$	$\mu = \frac{\sum_{i=1}^{n}\ln(X_i)}{n}$, $\sigma = \sqrt{\frac{\sum_{i=1}^{n}(\ln(X_i)-\mu)^2}{n}}$ “using the maximum likelihood method”
Exponential	$1-e^{(-\lambda x)}$	$\lambda = \frac{1}{\mu}$
Weibull	$1-e^{-\left(\frac{x}{\eta}\right)^{\beta}}$ Inverse: $x=\eta * (\ln 1/(1-F(x))^{1/\beta}$	β, η where those parameters were found using the maximum likelihood and least squares methods as the following: [30] $\beta = \frac{\left\{n.\sum_{i=1}^{n}(\ln x_i).\left(\ln\left\{\ln\left[1/\left(1-(i/n+1)\right)\right]\right\}\right)\right\}-\left\{\sum_{i=1}^{n}\ln\left\{\ln\left[1/\left(1-(i/n+1)\right)\right]\right\}.\sum_{i=1}^{n}(\ln x_i)\right\}}{\left\{n.\sum_{i=1}^{n}(\ln x_i)^2\right\}-\left\{\sum_{i=1}^{n}(\ln x_i)\right\}^2}$ $\eta = (\frac{1}{n}\sum_{i=1}^{n} x_i^{\beta})^{1/\beta}$

The four previously mentioned distribution functions were applied on the TTF data set for each PCB component, and then the maximum absolute difference (i.e. distance) between the theoretical and empirical cumulative distributions (| F_0 - F_n |) is found. Depending on the KS logic, the component TTF data set was likely to follow the assumed distribution if the maximum absolute distance between the theoretical and empirical distributions is less than other distributions’ maximum absolute distance. As a result, it represents the best fit distribution of that component TTF data. In other words, in the distance test, when the assumed distribution is correct, the theoretical (assumed) CDF (denoted by F_0) closely follows the empirical, step function CDF (denoted by F_n).

The following steps summarize the above KS test:

Let the TTF data set (days) of component x in the PCB is:

TTF (days)

338.7	308.5	317.7	313.1	322.7	294.2

I. Specify one of the four previously mentioned distributions.
II. Estimate its parameters using Table 1. For example, the mean and standard deviation for normal distribution:

Mean	**S.D**
315.82	14.85

III. Sort the TTF data in an ascending order. (See column 2 in Table 2 and Table 3).
IV. Obtain the theoretical (assumed) CDF (F_0). (See column 3 in Table 2 and Table 3).
V. Find the empirical, step function CDF (F_n). (See column 4 in Table 2 and Table 3).
VI. Calculate the absolute distance $|F_0 - F_n|$ between the theoretical and empirical distributions. (See column 5 in Table 2 and Table 3).
VII. Find the maximum absolute distance between the theoretical and empirical distributions for the specified distribution.
VIII. Repeat the previous steps for all other previously mentioned distributions
IX. The distribution with the minimum value of the maximum absolute distance $|F_0 - F_n|$ of all distribution represent the best fit distribution of the assigned data.

Table 2. Values for the KS GoF test for Normality

Row	Dataset	F_0	F_n	D= $\mathbf{|F_0 - F_n|}$
1	294.2	0.072711	0.142857	0.070146
2	308.5	0.311031	0.285715	0.025316
3	313.1	0.427334	0.428571	0.001237
4	317.7	0.550371	0.571429	0.021058
5	322.7	0.678425	0.714286	0.035861
6	338.7	0.938310	0.857143	0.081167
			Max:	0.081167

Table 3. Values for the KS GoF test for Weibull

Row	Dataset	F_0	F_n	D= $\|F_0 - F_n\|$
1	294.2	0.220598	0.142857	0.077741
2	308.5	0.305339	0.285715	0.019624
3	313.1	0.336435	0.428571	0.092136
4	317.7	0.369275	0.571429	0.202154
5	322.7	0.406793	0.714286	0.307493
6	338.7	0.536565	0.857143	0.320578
			Max:	0.320578

The KS GoF test statistic value (0.320578) is now four times higher than before. It is larger than the KS test value found for normality assumption. Based on these adaptive KS results, we reject the hypothesis that the population from which these data were obtained, is distributed Weibull.

Same procedure under lognormal and exponential distribution functions was conducted and KS test result found for each one of them. Then a comparison between the four distributions KS tests results was conducted where the distribution with the smallest KS test result representing the best fit distribution for that component TTF data. This methodology is repeated for all PCB components. As a result, a best fit distribution function can be found for each component TTF data set.

2.2. New TTF of PCB Components

In this way, a random number (between 0 and 1) was generated using a random number generator code. This number was used as a cumulative probability under a component best fit assumed distribution. That after obtaining the random number, we inserted it into the performance function and computed a new TTF (using inverse CDF), see Figure 2.

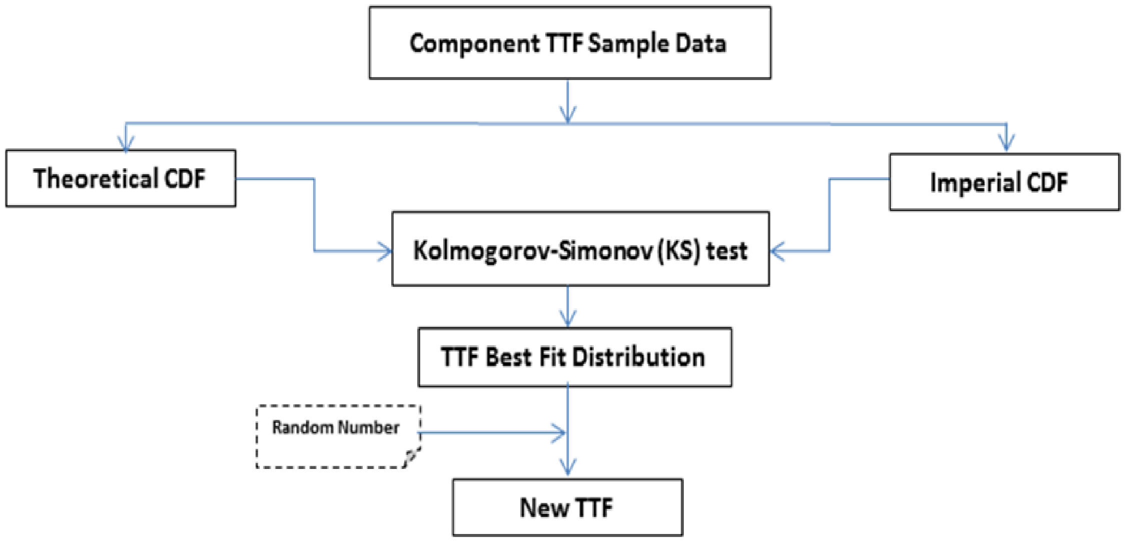

Component TTF Sample Data
Theoretical CDF
Imperial CDF
Kolmogorov-Simonov (KS) test
TTF Best Fit Distribution
Random Number
New TTF

Figure 2. New TTF: A new time to fail value is assigned to each component of a PCB based on its data best fit distribution as the predicted life time of that component.

2.3. Lifetime Range of PCB Components

Life time range for each PCB component can be calculated using the equation below:

$$\bar{X} \pm 1.5(s) \tag{1}$$

Which represent the upper and lower bounds of possible TTF. In this equation, $\bar{X}$ is the mean time to fail (MTTF), and s is a standard deviation of those TTF data for each component. In our methodology, we set the min TTF as $\bar{X} - 1.5(s)$ and the max TTF as $\bar{X} + 1.5(s)$. $\bar{X}$ and s were calculated based on each component TTF data distribution function, see Table 4.

Table 4. MTTF and standard deviation equation for each distribution [29]

	Mean ($\bar{X}$)	**Variance (S^2)**
Normal	$\frac{1}{n}\sum_{i=1}^{n} x_i$	$\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$
Lognormal	$e^{\mu+(\sigma^2/2)}$	$(e^{\sigma^2} - 1)e^{2\mu+\sigma^2}$
Weibull	$\eta * \Gamma(1+1/\beta)$ $\Gamma(n)$ *is the gamma function*	$\eta^2 * \Gamma(1+2/\beta) - \mu^2$ $\Gamma(n)$ *is the gamma function*
Exponential	Where the exponential function is a special case of a Weibull function, we used same Weibull equations with β=1	

<u>2.4. Simulation Methodology</u>

In this research, a simulation methodology was established to find out the system level lifetime and reliability. The simulation starts with the age of each component,where the time for the next fail for each component is any time between that age and the maximum TTF of that component ($\bar{X} + 1.5(s)$), and if maximum9 TTF is less than or equal to the age, or if the age is equal to zero (new component), a new TTF is generated by the developed random number generator code.

At the beginning of the simulation, the run length and the number of replications are assigned (i.e., 50 years and 100 replications, respectively). In other words, each replication consumes 50 years. As the simulation timer starts, the component with the smallest TTF fails first, resulting in a reduction in the time needed for other components to fail sequentially. Once the component with the smallest TTF fails, its position in the PCB is checked. If it is a part of a single-component parallel cluster (see Figure 3(a)), then its failure does not stop the operation of the entire PCB as parallel cluster keeps running until all of its components fail. On the other hand, if the component is in a parallel cluster that contains more than one component in a series configuration in any of its networks (see Figure 3(b)), then the network that contains more than one component in a series configuration stops operating at the minimum TTF of those components, and the entire cluster stops operating at the maximum TTF. If there is more than one cluster of parallel components connected, then the above criteria are implemented for each cluster individually. After finding one TTF for each cluster, all clusters are handled as one cluster. Maximum TTF, which represents the TTF of all such clusters, is found and eventually stops the operation of the entire circuit at that TTF.

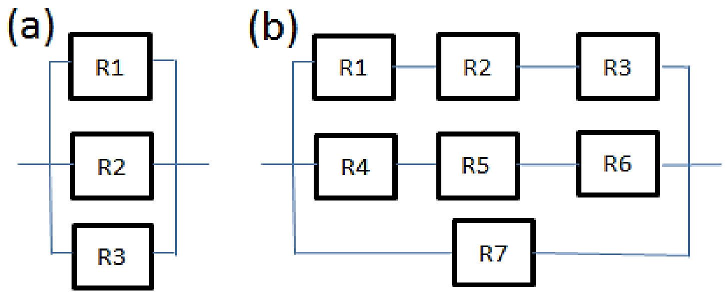


Figure 3. Parallel clusters: (a) single-component parallel cluster, (b) parallel cluster that contains more than one component in a series configuration.

Finally, if the failed component is in a series configuration, and it is not in any parallel cluster, then its failure eventually stops the entire PCB at that TTF. Component repair or replace approaches provided by this simulation give the user the ability to choose between them based on experience and component history. Once the position of the component is determined, a “time to repair” or a “time to replace” value is assigned to the component if its failure causes the failure of the entire PCB as follows:

If the failed component is in a series configuration, and not in any parallel cluster, then a “time to repair” or a “time to replace” value is assigned to that component, and the assigned “time to repair” or “time to replace” starts decreasing until it reaches zero. A TTF value and a “time to repair” or a “time to replace” value are assigned for that component, and the entire circuit resumes operation. While the circuit is in a failure mode, the TTF value of other components stop decreasing (freezing) at the time when the failed component stops operating. Once the circuit resumes operation, those components continue operating from the same time point where the failed component stops. On the other side, if the failed component is a part of a parallel cluster, then it should have the maximum TTF between all networks of that cluster, and the maximum "time to repair" or "time to replace" value between all components of this cluster is assigned, see Figure 4.

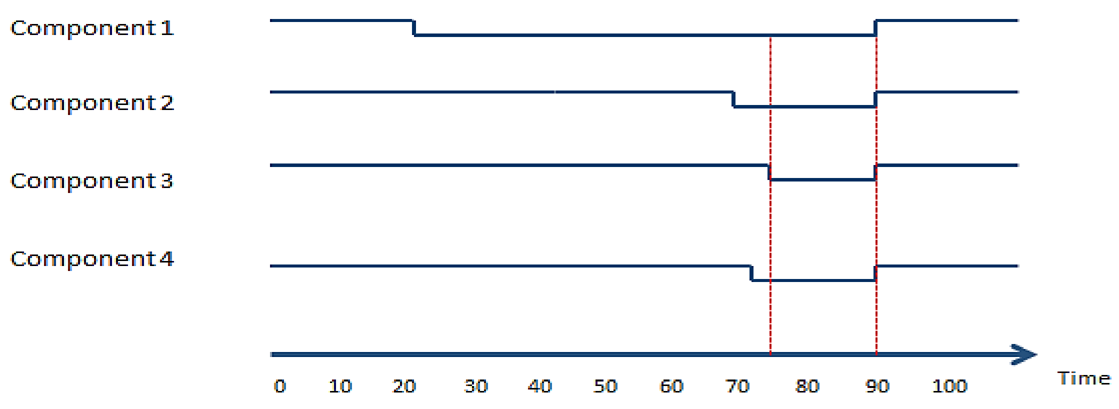
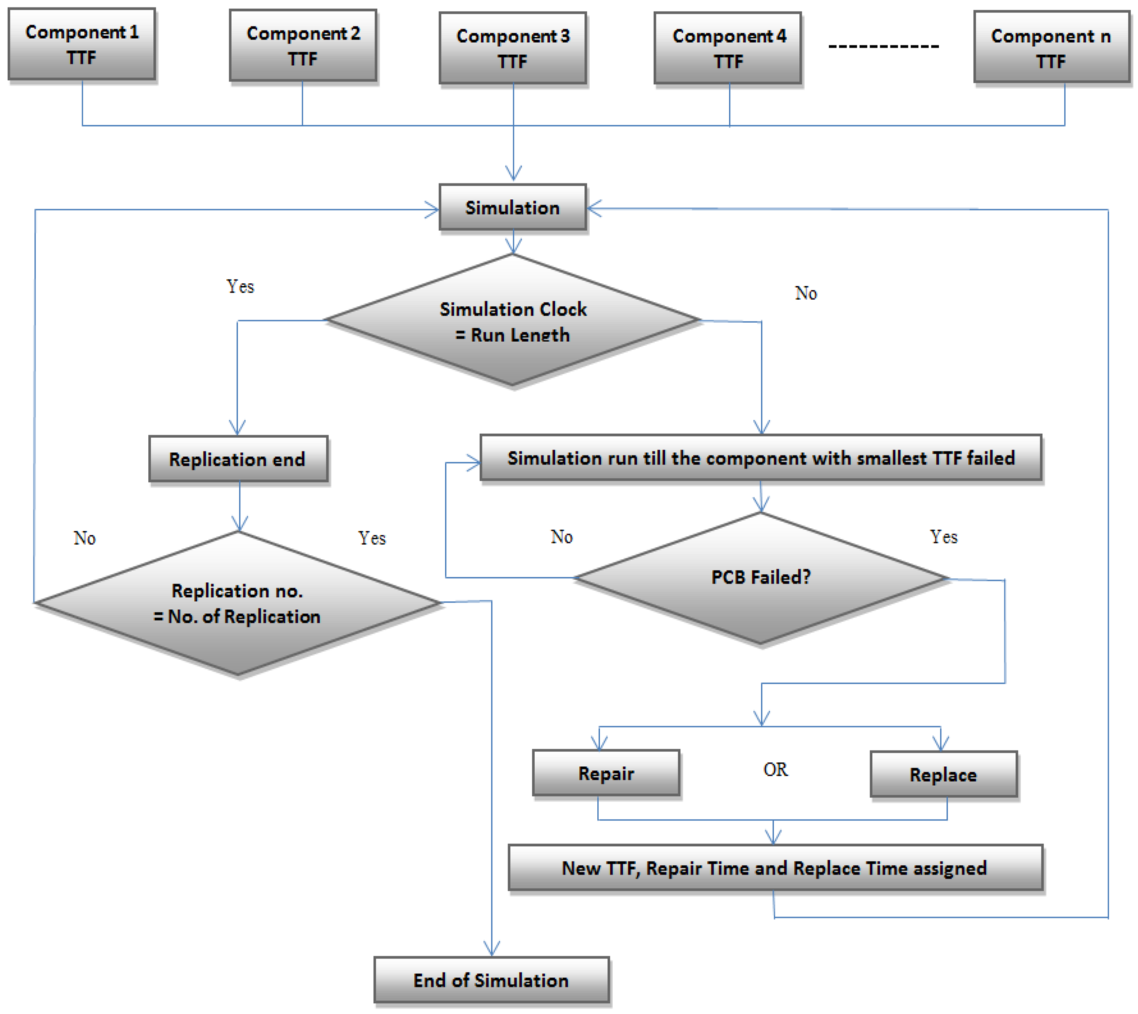


Figure 4. TTF and repair or replace time for a parallel cluster

New TTF value and "time to repair" or "time to replace" values are assigned for those cluster components, and the entire circuit resumes operation. Otherwise, if the failed component does not have the maximum TTF value, then the simulation searches for the next smallest TTF value in the PCB.

The above simulation methodology keeps running until the end of the simulation run length, which is assigned at the beginning of the run. The above methodology is illustrated in the following flow chart to clarify the procedure, see Figure 5.


Component 1 TTF
Component 2 TTF
Component 3 TTF
Component 4 TTF

Component n TTF
Simulation
Yes
Simulation Clock = Run Length
No
Replication end
Simulation run till the component with smallest TTF failed
No
Yes
Replication no. = No. of Replication
No
PCB Failed?
Yes
Repair
OR
Replace
New TTF, Repair Time and Replace Time assigned
End of Simulation

Figure 5. Flow chart that represents the new methodology for estimating overall system reliability and lifetime.

<u>2.5. PCB Lifetime and Failure List</u>

The above simulation allows finding PCB lifetime when the following criteria are satisfied:

“Time to repair” or “time to replace” intervals, where the component stops operating, are created for each component. If the components are connected in a series configuration, the combination of all their “time to repair” or “time to replace” intervals is found. The lower bound of this combination interval represents the time at which this network, and eventually the entire circuit, stops. Otherwise, if the components are connected in a parallel configuration:

I. If the parallel cluster contains only a single component network as shown in Figure 3(a), then the overlap of all their "time to repair" or "time to replace" intervals is found, and the lower bound of such represents the lifetime of this cluster, see Figure 4), where the lifetime is 75 (time unit).

II. If the parallel cluster contains more than one component in any of its networks as shown in Figure 3(b), then the combination of all "time to repair" or "time to replace" intervals of those components in that network is found, as the series configuration presented above, and the lower bound of this combination intervals represents the stoppage of this network. After that, Criterion I should be implemented.

III. If there are more than one clusters of parallel components connected, then the Criteria I-II are applied to each cluster individually, and after finding one interval for each cluster, all clusters are considered as one cluster, and Criterion I is implemented.

In other words, the entire PCB does not fail unless one of the single series components fails or the entire parallel cluster fails.

2.6. Simulation Analysis

By using the above methodology, a PCB predictive failures (i.e., first fail, second fail…etc.) is created and displayed in PREVIEW, where Equation 1 is used to find the mean, maximum, and minimum time to fail value for each failure based on 100 replications. Table 5 shows an example of PCB failure time based on 100 replications, where upper and lower TTF bound is calculated using Equation 1. The data presented in Table 5 are hypothetical data which is not derived from nuclear power plan industry.

Table 5. The List of PCB Failure Times (years)

	Replication 1	Replication 2	Replication 3	….	Replication 100	$\bar{X} \pm 1.5(s)$
First Fail(next fail)	2.89247	2.81825	2.88137		2.90746	2.8907 ± 0.06952
Second fail	3.14587	3.097856	3.15893		3.13646	3.11573 ± 0.05027
Third fail	3.30762	3.22351	3.3013		3.28573	3.2676 ± 0.06423
…						
… nth fail (till 50 years)	49.90476	49.9512	49.9554		49.9074	49.914 ± 0.3656

10 simulation runs were conducted of 100 replications to test the next fail time, the output of those runs (i.e. next fail time) and the error percentages are presented in Table 6 and Figure 6. Where a percentage of 0.8 % was found as an average error between those runs, and the maximum error was found to be 1.46%. This error percentage can be reduced by increasing the number of replications (e.g. 1000 instead of 100 replications); however this will increase the code running time.

Table 6. The List of PCB Next Fail Time (years) from 10 Runs

	Min TTF	Mean TTF	Max TTF
1st Run	2.82118	2.8907	2.96022
2nd Run	2.81825	2.90315	2.98805
3rd Run	2.80001	2.88137	2.96272
4th Run	2.78703	2.86428	2.94154
5th Run	2.82373	2.89755	2.97137
6th Run	2.75667	2.84216	2.92766
7th Run	2.81595	2.89247	2.96898
8th Run	2.80332	2.88638	2.96943
9th Run	2.8325	2.90746	2.98243
10th Run	2.7543	2.84008	2.92586
Avg. Error (RMSPE)		**0.790110964**	
Max Error (RMSPE)		**1.461042**	

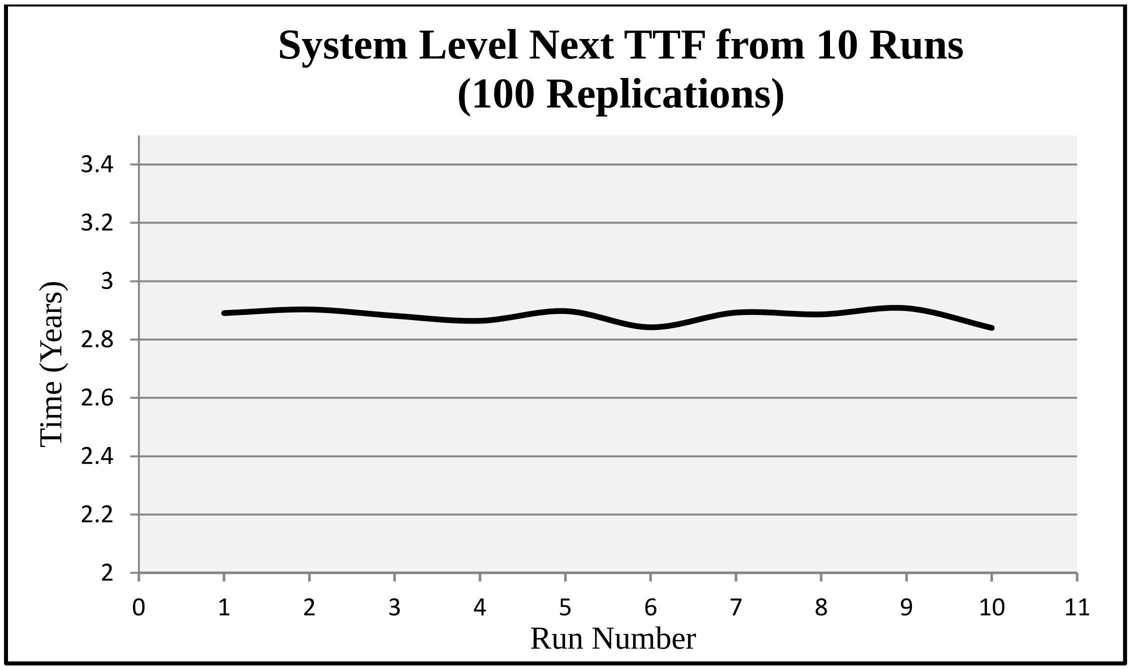


Figure 6. Entire PCB next fails' time for 10 different runs.

We performed a sensitivity analysis to understand the effect of the change in input TTF data on the next fail time of the card. Using the aforementioned simulation technology, 5% and 10% increase and decrease in the value of each datum in TFF data is analyzed. 5% increase in the input resulted in an increase that is greater than 5%. Similarly, 10% increase in input data resulted in a positive shift greater than 10%. This can be explained by the effect of network configuration. While there exists several components in series, parallel or bridge (i.e. integrated circuit, where there are multiple terminals) connection in a network, next fail range is affected by the aggregate change in the components. This mainly depends on the network configuration of the circuit card as well as the number of components. As the number of components increases, the effect of change in the input data will be greater. In addition, high number of series and bridge connections is critical and leads to more change in the output result. A similar trend is also observed when the input data is changed by +/-10% as depicted in Figure 7.

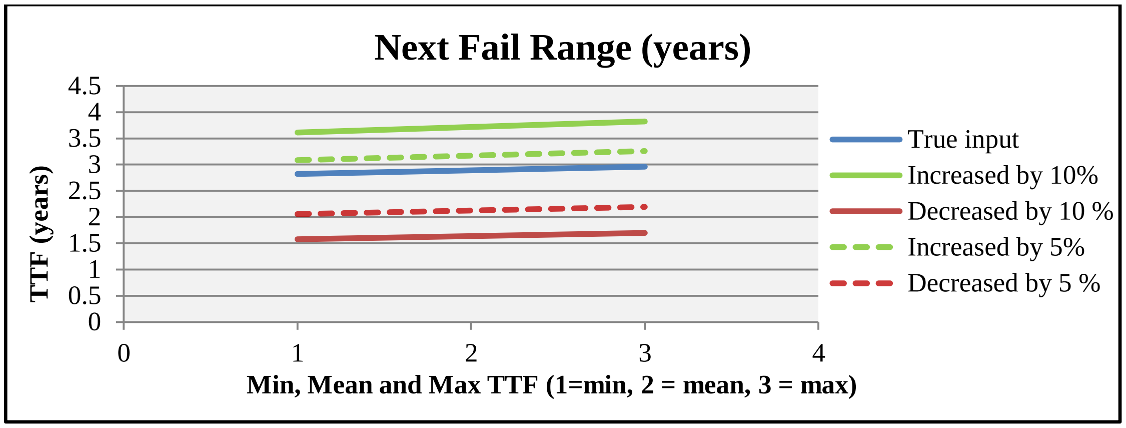


Figure 7. Sensitivity analysis of the circuit card model: effect of 5-10% change in the input data on minimum, mean and maximum TTF (The data are presented in years).

In this chapter, the remaining lifetime of PCB and its components was analyzed and calculated where the components with the shortest remaining life were highlighted. In the next chapter, reliability information is generated for component and system level, while this enables one to replace components with the highest probability of failure.

CHAPTER III

COMPONENT AND SYSTEM LEVEL RELIABILITY

Reliability is defined as the probability that a device or system will perform a required function for a given period of time without failure. It is important to note that reliability should be specified for a given period of time, as this variable has an important effect. The observed TTF is a value of the random variable T, which represents the lifetime of the component, where T takes values in the interval [0,∞), and its probability distribution function is F(*t*), or CDF. So, as the reliability is a quantitative measure of non-failure, it is useful to be expressed by R (*t*) = 1- F (*t*) where F(*t*) is the best CDF for each component based on the TTF data set, where the complement of this CDF is reliability. Using this CDF at a specific time for each component, reliability can be calculated and assigned for all PCB components. The use of appropriate data can help in ensuring adequate part life in a specific application as well as in projecting anticipated part reliability.

<u>3.1. Conditional Reliability of a PCB Components</u>

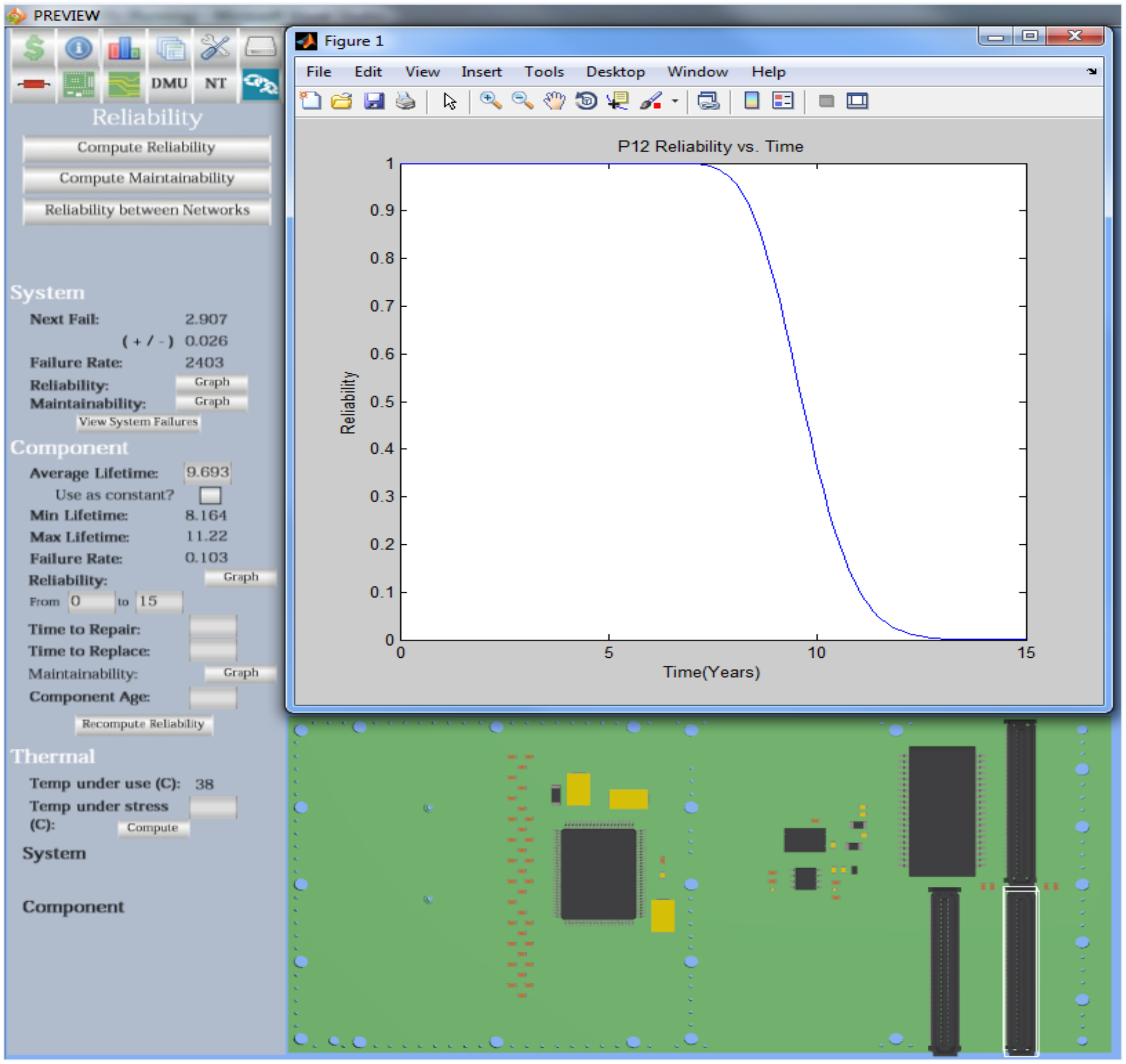
The reliability is the probability of no failures (survival) in the interval [0, *t*]. In this research, PCB components are assumed to be used for a period of time, so each component had an age and the reliability is calculated beyond the age of each component. The reliability of a component after that age (*x*) is a conditional reliability, using Bayes' rule: P [no failure (*x*, *x*+*t*)| no failure (0, *x*)]. The reliability after time (*t*), thus, is equal to:

$$R(x+t)/R(x) \tag{2}$$

And the failure rate for component level is calculated as an inverse of its MTTF, as a constant failure rate considered in this research:

$$\lambda = 1/MTTF \tag{3}$$

PREVIEW is used to display the component reliability in its visual environment. The user has the ability to choose any component on the PCB by clicking on that component, and its reliability over a specific period of time in addition to its failure rate appear, see Figure 8. Where in this figure, component P12 reliability over the time from 0 to 15 years is presented starting from age zero. And as the current age of this component is 7.35 years, the reliability from 0 to 7.35 years is 1, then it start decreasing by time.



System

Next Fail:	2.907
(+ / -)	0.026
Failure Rate:	2403
Reliability:	Graph
Maintainability:	Graph

View System Failures

Component

Average Lifetime:	9.693
Use as constant?	☐
Min Lifetime:	8.164
Max Lifetime:	11.22
Failure Rate:	0.103
Reliability:	Graph
From 0	to 15
Time to Repair:	
Time to Replace:	
Maintainability:	Graph
Component Age:	

Recompute Reliability

Thermal

Temp under use (C):	38
Temp under stress (C):	

Compute

System

Component

Figure 8. A snapshot of component reliability interface over a time period in PREVIEW (from 10 replications)

<u>3.2. Reliability Analysis at Board Level</u>

For the board level reliability, we make use of the reliability probabilistic feature for the entire PCB; this allows one to calculate reliability quantitatively. Since one of the simulation outputs is the next fail time at board level, we count failures at that time (or higher) among all replications and divide the outcome by the number of replications (100 replications). The result represents the reliability at that specific time. Figure 9 shows a board level reliability for different 10 runs, where high correlation and small deviation between those runs is shown. Boared level reliability data is found in appendix C of this thesis.

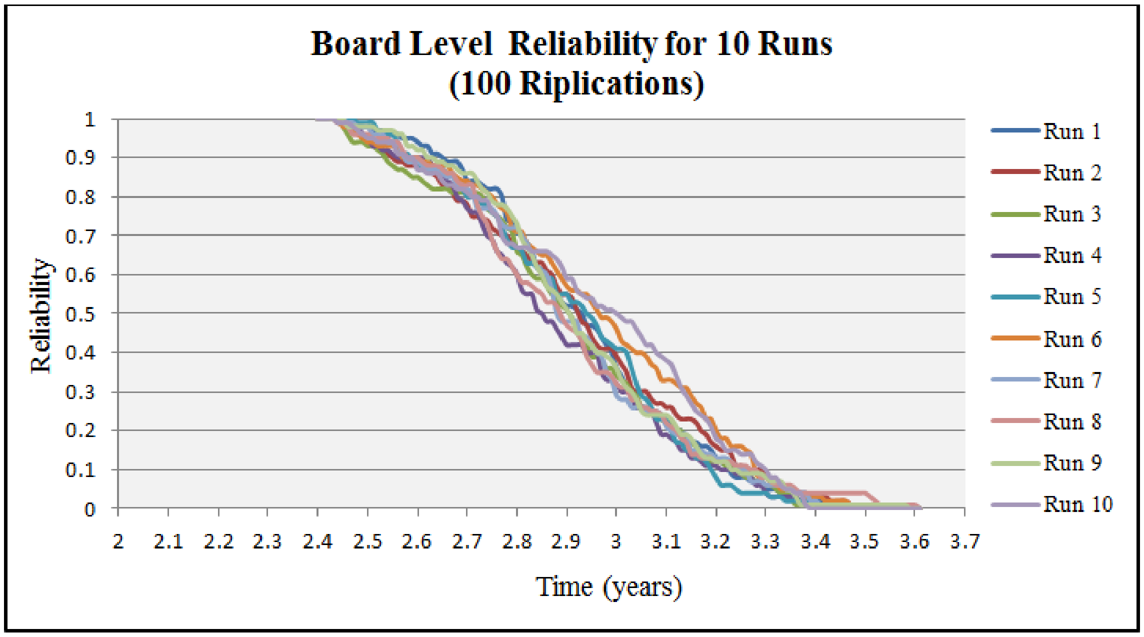


Figure 9. Board level reliability over a set of time for 10 different runs.

PREVIEW is used to display the entire PCB reliability. When the user clicks on the "Compute Reliability" button and then clicks on "Graph" under the system tab on the PREVIEW screen, the entire PCB reliability over a period of time appears in Figure 10.

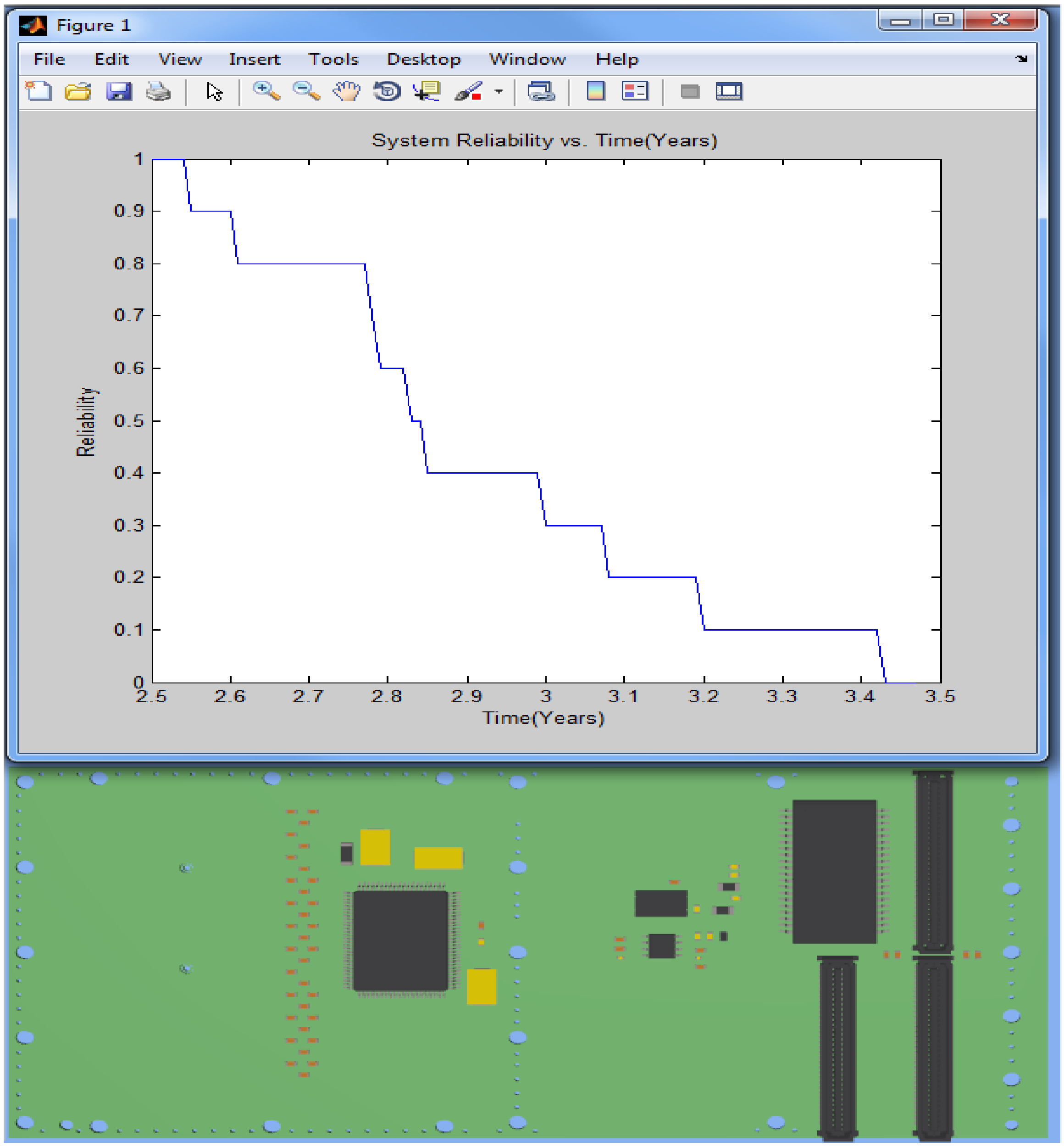


Figure 10. A PREVIEW snapshot showing board level reliability from 10 replications

Finally, a basic limitation of this reliability prediction methodology is its dependence on correct application by the user. Those who correctly apply the models and use the information in a conscientious reliability program can find the prediction a useful tool.

Sensitivity analysis based on 5% and 10% changes in input files (TTF and TTR) were conducted to examine the changes in the bored level reliability, where the result illustrated in Figure 11. We performed the sensitivity analysis to understand the effect of the change in input TTF data on the board or card level reliability. Using the simulation methodology presented in Chapter 2, 5% and 10% increase and decrease in the value of each datum in TFF data is analyzed. 5% increase in the input for each component resulted in approximately %15 shift in the reliability curve. Similarly, 10% increase in input data resulted in a positive shift greater than 25%.

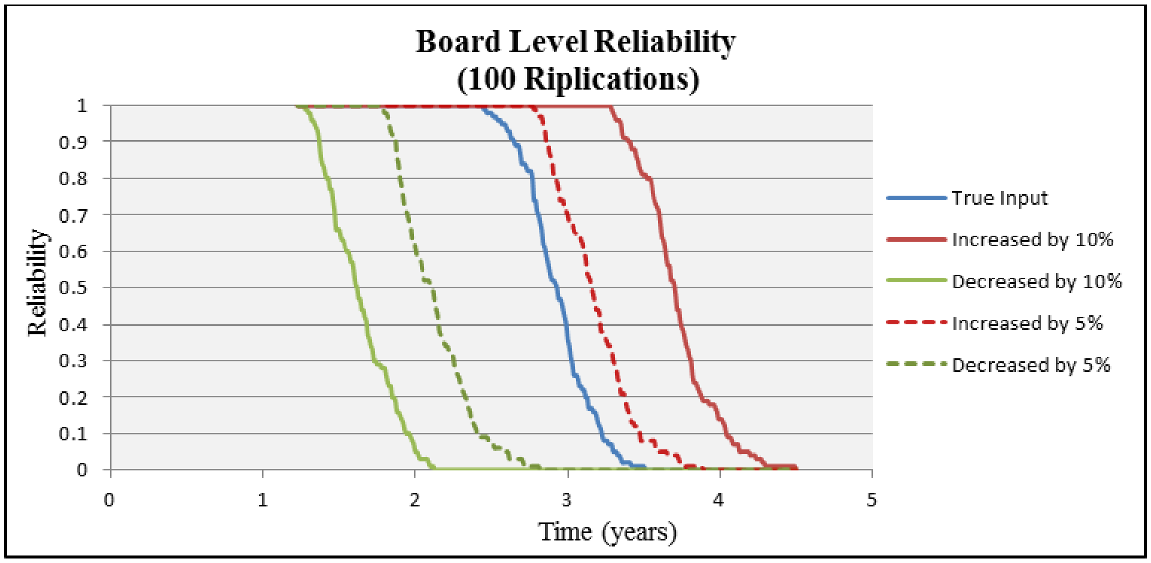


Figure 11. Board level reliability changes based on 10% and 5% changes in input file.

3.3. Reliability Between Two Nodes in PCB

A system is a combination of subsystems which assemblies in a specific arrangements in order to succeed desired functions with its intended performance and reliability. Component types, the way they are arranged in the system, the number and quality of components have a major effect on the reliability of a system. A good understanding of the relationship between a system and its constituent components should be achieved in order to make the right decision at the right time and right place.

Electronic devices such as resistors, diodes, switches, and capacitors, are circuit components which placed and positioned in a circuit structure. The placement of such components is crucial to the operation of the circuit, as different kinds of setups create a different kind of outputs, results, or purposes, when those components are in series and /or parallel configuration. It is vital that the relationship between the system and its network model be thoroughly understood before considering the analytical techniques that can be used to evaluate the reliability of these networks.

This section addresses a new criteria that creats a new feture in the reliability prediction between any two nodes on a PCB. This can give the user a new feature in calculating the reliability in any intreseted partition of the PCB, for example partions under stress. In a reliability network, often referred as a reliability block diagram (see Figure 12), components are in series from a reliability point of view if they must all work for system success or only one needs to fail for system failure. Let R*s* represent the system (or subsystem) reliability and Q*s* represent the probability of failure. Then, those reliability and probability of failure can be found by using the following equation [1, 29]:

$$\begin{aligned} R_s &= \prod_{i=1}^{n} R_i \\ Q_s &= 1 - \prod_{i=1}^{n} R_i \end{aligned} \tag{4}$$

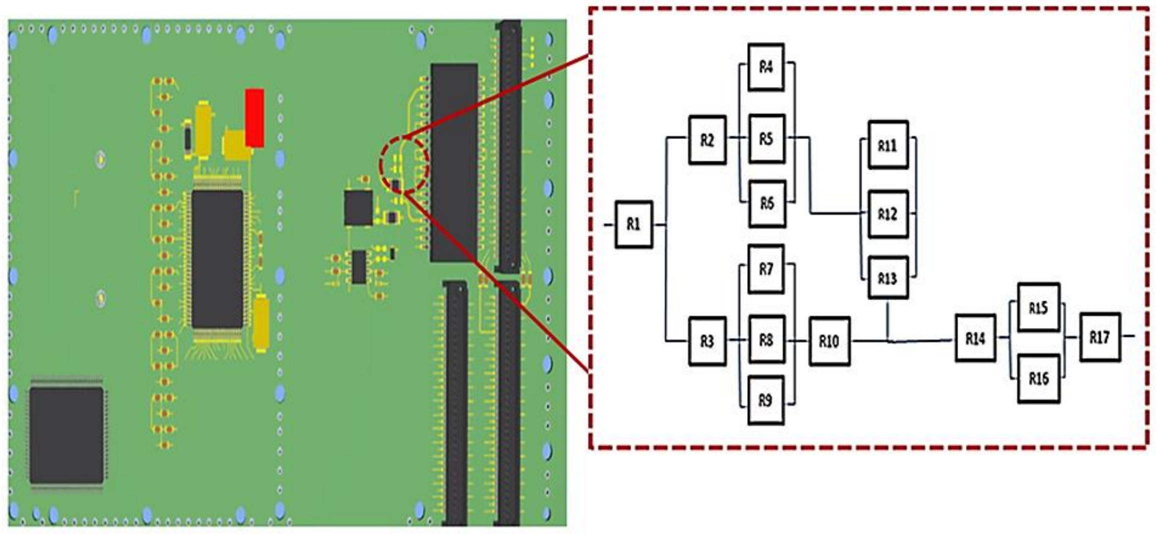


Figure 12. Reliability block diagram: describes the interrelation between the components and the defined system

In this research, reliability between any two nodes on a PCB is calculated. At the beginning, all series components in each network in the PCB are found; if Terminal 2 of a PCB component and Terminal 1 of another PCB component are on the same network ID, then those two components are considered to be in a series configuration and Equation 4 is used to find their total reliability, see Figure 13.

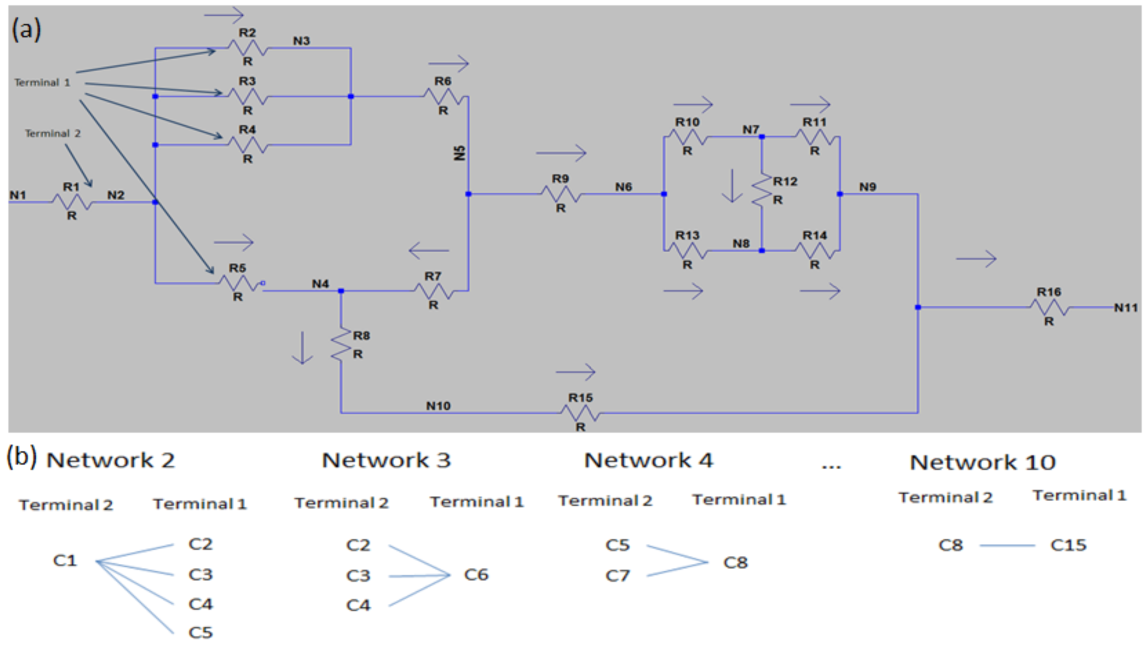


Figure 13. Connected components in each network : (a) Connections of components based on terminal ID, (b) components on the same network ID

Based on those connections on each network, a path tree is initiated; where all possible paths are found and registered, see Figure 14.

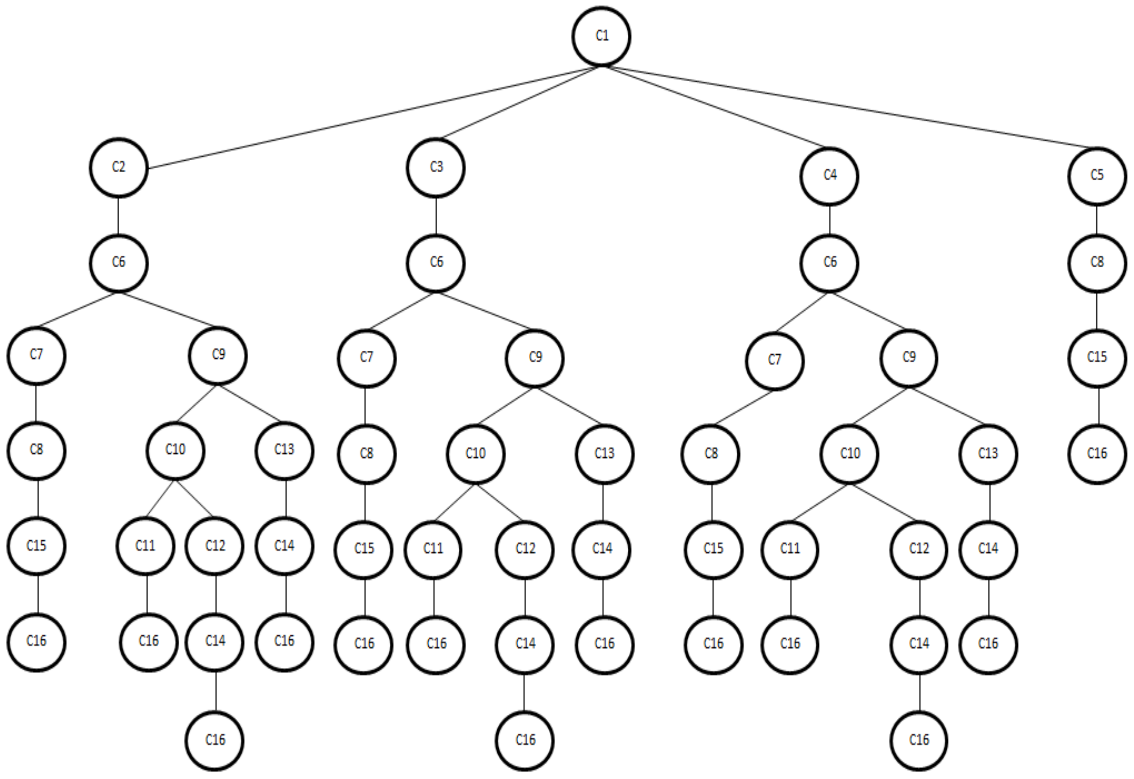


Figure 14. All possible paths between components in a series configuration based on network ID

A C++ code was established to find these paths based on the network ID of each component. Eventually, the reliability of those paths at a specific time is calculated by using Equation 4, where the highest path reliability represents the minimum reliability between the two nodes connected by those paths.

PREVIEW virtual meter was created using the previous criteria and methodologies in order to display the reliability between any two nodes on a PCB. The established virtual meter can be used to study the reliability in case of abnormal conditions on PCB (e.g. thermal stress), where the user can specify the area under interest and calculate the reliability of that area instantaneously using the virtual meter. The user can place the two probes of the virtual meter on any two nodes on a PCB, and then the minimum reliability in between is calculated and displayed on the virtual meter, see Figure 15.

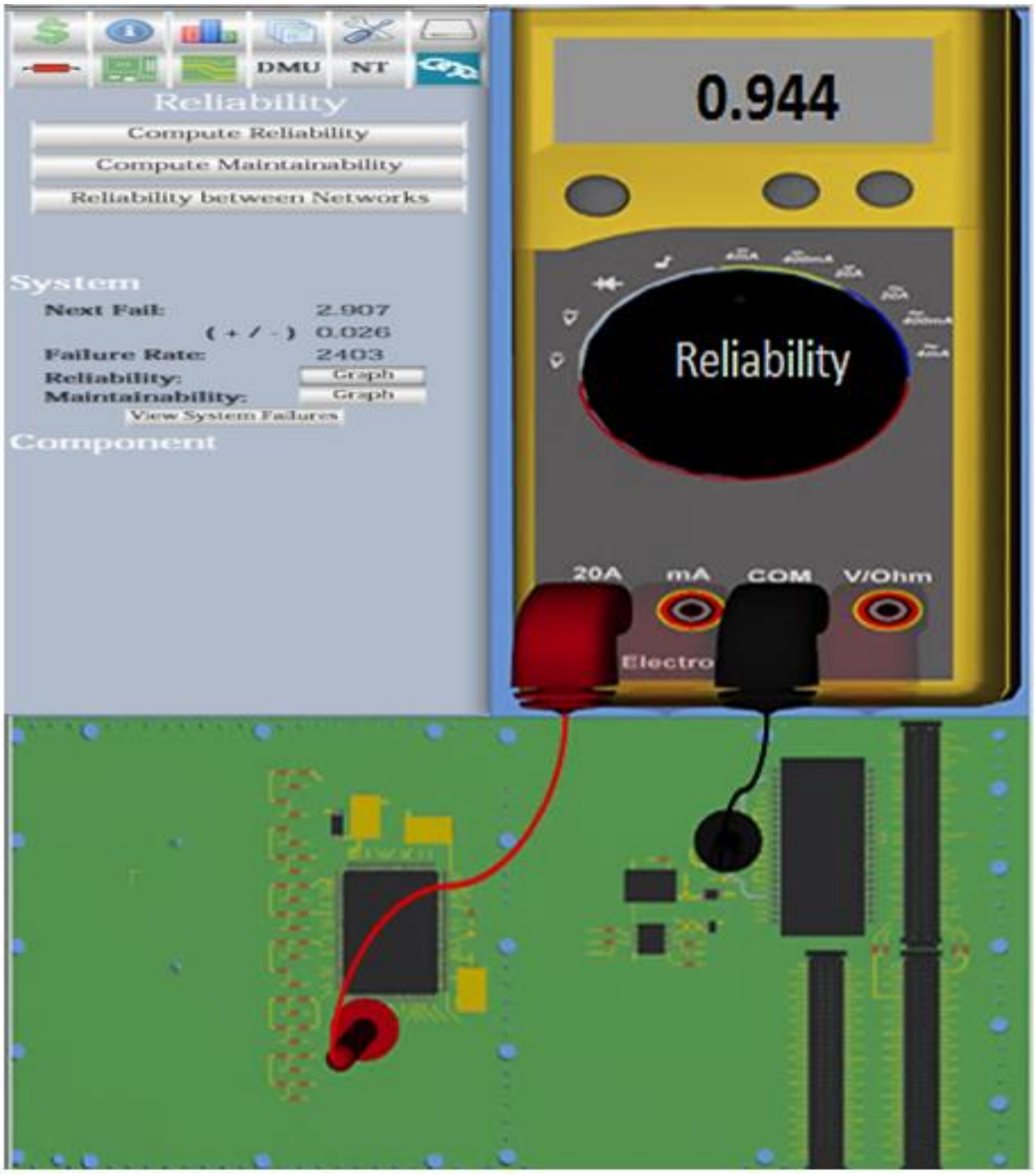


Figure 15. PREVIEW virtual meter displays the reliability between any two nodes on a PCB.

While this will enable one to replace components by retrieving or calculating reliability information, a maintainability model was developed to assist users in making ideal replacements, maintenance policy and planning as illustrated in next chapter.

CHAPTER IV

COMPONENT AND SYSTEM LEVEL MAINTAINABILITY

4.1. Introduction

On repairable system, maintenance actions can be carried out to restore system components to operate again when they fail. These actions should be taken into consideration when evaluating the behavior of the system, where monitoring the effectiveness of electronics maintenance at nuclear power plants is essential for implementation of the maintenance rules and policies.

Additional information is now needed for each system component in order to understand the overall system behavior; how long it takes for the component to be restored. To properly deal with systems, we need to first understand how components in these systems are restored, as maintenance plays a vital role in the life of a system. There is a time associated with each maintenance action (i.e., the amount of time it takes to complete the action); this time is referred to as “time to repair” or “time to replace” (TTR).

According to Ebeling, maintainability can be defined as “the probability of performing a successful repair action within a given time” [1]. In other words, maintainability determines the probability that a failed part can be restored to its normal operable state within a given timeframe, using the recommended practices and procedures [1]. For example, if it is said that a particular PCB component has 95% maintainability in three hours, this means that there is a 0.95 probability that this particular PCB component will be repaired within three hours or less. The random variable used in this research is “time to repair” and/or “time to replace”; in the same manner as TTF is the random variable in reliability [31].

4.2. Component Level Maintainability

Maintainability can be calculated by using CDF for "time to repair" and/or "time to replace" (TTR), see Table 7.

Table 7: Maintainability CDF

Distribution	Maintainability (F(t))	Parameters
Normal	$\frac{1}{2}(1+\operatorname{erf}(\frac{(t-\mu)}{\sigma\sqrt{2}})$	μ,σ
Lognormal	$\frac{1}{2}(1+\operatorname{erf}(\frac{(\ln t-\mu)}{\sigma\sqrt{2}})$	μ,σ
Exponential	$1-e^{(-\lambda t)}$	$\lambda =$ repair rate
Weibull	$1-e^{-\left(\frac{t}{\eta}\right)\beta}$	β,η

In this research, PREVIEW is used to display the maintainability, where the user has the ability to choose any component on a PCB by clicking on that component, and its maintainability graph over a specific time period is generated and displayed on a PREVIEW display as shown in Figure 16.

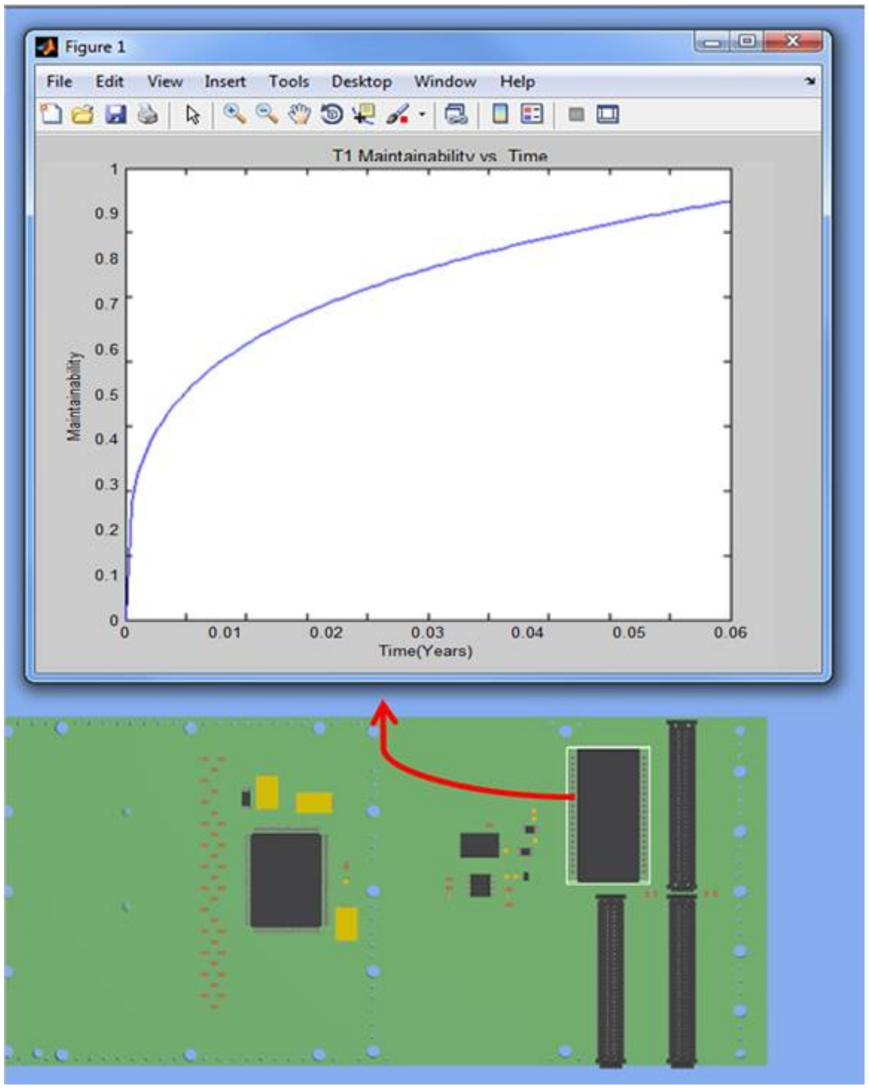


Figure 16. A screenshot of the component maintainability interface of the PREVIEW software.

4.3. Board Level Maintainability Analysis

For the board level, we make use of the maintainability probabilistic feature. Since one of the simulation outputs is "time to repair" and/or "time to replace" (for those components in which their failure leads to the entire PCB failure). Those times represent TTR for the entire circuit (system level). For calculating the system level maintainability, system TTR values, which are equal to or lower than a desired time from all replications have been counted, and then divide the outcome over the number of replications (100 replication in our simulation). The result represents the maintainability at that desired time. Figure 17 below shows PCB maintainability over a set of time for 10 different runs,each for 100 riplications where a low level of deviation is noted between all runs. Board level maintainability data is found in appendix D of this thesis.

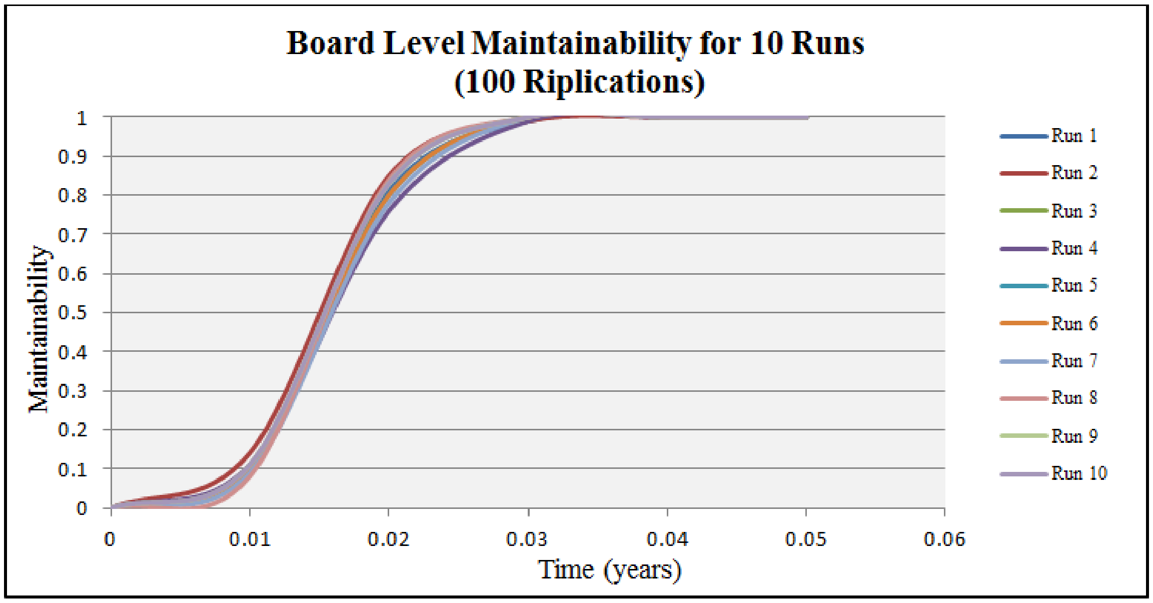


Figure 17. Board level maintainability over a set of time for 10 different runs.

PCBs life is extremely sensitive to temperature, as the temperature considered the major cause that degrades the PCB effectiveness. In next chapter, the effect of thermal load stress on the life time and reliability of PCB components and eventually of the board level is presented.

CHAPTER V

THERMAL STRESS AND ITS EFFECT ON PCB LIFETIME AND RELIABILITY

5.1. Introduction and Background

During the useful life of an electronic part, its reliability is a function of several stresses including but not limited to the electrical and the thermal stresses to which the part is subjected [33]. An increase in thermal stresses directly increases the failure rate and eventually decreases the reliability [32]. Stress reduction is another design method for reliability improvement; in most electronic components, reduction in temperature by improvement in thermal design results in reduced number of failures [2]. The failure rate of component increases many times when the working environment or stress becomes more and more severe (e.g. higher temperature) [33]. This is basically because the material properties change with the operating environment and as a result, the strength reduces. Lall *et al*. stated that for every 10 Celsius degrees rise in temperature, the failure rate of most electronic components doubles [22].

Determining reliability of PCB components is a complex task that is affected by many factors, such as the heat produced by the operation of those components, the tools and procedures used to manage and reduce that heat, the environment in which the PCB is required to operate and materials of components. Due to this complexity and the thermal effect on electronics reliability, thermal management tools have been improved to help reliability issues. Gap fillers, active cooling systems, heat pipes, and heat sinks are some of those tools. The selection of which tool(s) to use depends on some constraints in terms of cost, power required, weight, size, and reliability. Therefore, electrical and electronic parts such as those in PCBs must be selected with the proper reliability and thermal characteristics to operate correctly and reliably in specific conditions. High temperatures create a severe stress on most electronics such as PCB since they can cause not only disastrous failure, but also slow and deterioration of devise performance levels due to chemical degradation effects. Kallis and Norris stated that “excessive temperature is the primary cause of poor reliability in electronic equipment” [21].

5.2. PCB Lifetime and Arrhenius Life-Stress Model

This research established a model that uses the latest data and theoretical models to describe the effect of the thermal load stress on the life time and reliability of PCB components and eventually the entire PCB.

The Arrhenius life-stress model (or relationship) is probably the most common life-stress relationship utilized in thermal accelerated life testing [40]. It is derived from the Arrhenius reaction rate equation [40]. The Arrhenius reaction rate equation is given by [22, 34, 35]:

$$R(T) = Ae^{-\frac{Ea}{K.T}} \tag{5}$$

where R is the reaction rate (moles/meter^2 second), “A” is constant depending on material characteristic, Ea is the activation energy (electron Volts; eV), (The activation energy is the energy that a molecule must have to participate in the reaction), “K” is the Boltzman’s constant ($8.617 \times 10-5$ eV K-1), and “T” is the absolute temperature (Kelvin).

The Arrhenius life-stress model is formulated by assuming that life is proportional to the inverse reaction rate of the process, thus the Arrhenius life-stress relationship is given by [22, 35]:

$$\mathrm{L}(T) = C\exp\left(\frac{B}{T}\right) \tag{6}$$

where *“L”* represents a quantifiable life measure, such as mean life, “*T”* represents the stress level (formulated for temperature and temperature values in absolute units i.e.

degrees Kelvin), "*C*" is one of the model parameters to be determined, "*B*" is another model parameter to be determined, where $B = \frac{E_a}{K}$.

In this formula, the activation energy must be known. If the activation energy is known, then there is only one unknown parameter remaining, "*C*". In this equation it is evident that the parameter "*B*" has the same properties as the activation energy. In other words, *B* is a measure of the effect that the stress (i.e. temperature) has on the component life.

5.3. Estimation of Activation Energy

Activation energy is defined as the minimum amount of energy required to initiate a particular process. In the context of electronic device reliability; however, activation energy refers to the minimum amount of energy required to trigger a temperature-accelerated failure mechanism [36].

A failure mechanism is defined as a physical phenomenon that can lead to device failure if triggered and given enough time to progress [37]. The value of activation energy indicates the relative tendency of a failure mechanism to be accelerated by temperature; i.e., the lower the E_a, the easier it is to trigger a failure mechanism with temperature.

Different failure mechanisms have different activation energies. Table (8) shows some activation energy for various failure mechanisms commonly encountered in the semiconductor industry. The reader is referred to [38], for full list of activation energy for most failure mechanisms.

Table 8. Different failure mechanisms' activation energies

Failure Mechanisms	Activation Energy (eV)
Electromigration (grain boundary diffusion)	0.68
Electromigration (metal-to-barrier and/or dielectric x-face diffusion)	0.95
Corroded aluminum trace or bus (chloride)	0.7
Corroded aluminum trace or bus (phosphorus)	0.53
Corroded pad	0.7
High-percentage phosphorus-induced corrosion	0.53
Micro-cracking, step coverage (metal-to-barrier and/or dielectric x-face diffusion)	0.95

Activation energies of the various failure mechanisms that arise in the device are lumped together to generate weighted average activation energy for the device [22]:

$$E_{\mathrm{a-dev}} = \sum_{i=1}^{n} m_i \; E_{\mathrm{a}-i} \tag{7}$$

where m_i is the weight assigned to the activation energy of each failure mechanism (dimensionless), and $E_{\mathrm{a}-i}$ is the activation energy of the i^{th} failure mechanism.

Table 9 demonstrates an example of the variability of the dominant device failure mechanisms weights (percentages) of thousands of transistors [22, 37].

Table 9. Failure mechanisms weights for thousands of transistors

Failure Mechanisms	Survey				
	1	2	3	…	n
Electromigration		13%			
Dielectric breakdown	50%	2%	98%		
Parametric drift		38%			
Electrical overstress	20%		2%		
Latch-up	10%				
Package-related	20%	28%			
Other		19%			

Some reliability standards, solve this complexity such as global methodology for reliability engineering in electronics (FIDES) in calculating the general activation energy based on all failure mechanisms, failure percentages, component process technology (e.g., Bipolar logic, CMOS logic). Based on such standards, they found that the typical activation energy is generally near 0.7eV [35, 39].

For an effective quantitative accelerated life test, thermal stress that causes a PCB to fail under normal use conditions was chosen, and then the stress at various increased levels was specified and the TTF for each PCB component under accelerated test conditions was measured. For example, if one of the PCB components (i.e. capacitor) normally operates at 300K and high temperatures cause it to fail more quickly, then the accelerated life test for such a component may involve testing the product at 310 K and 320 K in order to stimulate the units under test to fail more quickly. In this example, the use stress level is 300K and the accelerated stress levels are 310K and 320K.

As stated earlier, all the various attempts to predict the failure rate of electronics components in nuclear power plant are still based on MIL-HDBK-217, even though they have been proven inaccurate, misleading, and damaging to cost effective and reliable design, manufacturing, testing, and support.

5.4. Quantitative Lifetime Model Under Thermal Stress

This research uses accelerated life testing to study how failure is accelerated by thermal stress and how that affects lifetime and reliability for both components and the entire PCB. Quantitative accelerated life tests (QALT) are used to quantify the life of the product and produce the data required for accelerated life data analysis. Methods for predicting reliability with the shortest time and small sample size are called accelerated life tests of the device. Accelerated tests can be defined as “tests carried out under environmental conditions more severe than normal conditions”. This means that operating a device at high stress produces the same failures that would occur at normal-use stresses, except that they happen much quicker [29,22].

Most researchers use the term “acceleration factor (AF)” to refer to the ratio between device lifetime under use level (normal use conditions) and lifetime a higher stress level [9].

$$\text{AF(T)} = \frac{Lifetime(use)}{Lifetime(accelerated)} \tag{8}$$

By using AF(T), lifetime for this component at each temperature point can be calculated. However, due to time constrains, unavailability of required test tools and lack of experience and knowledge, it becomes very difficult for many companies to get TTF data under stress conditions, where the TTF data under normal use conditions is the only available data. In this case, a justified theoretical methodology should be used in order to find the PCB life time under severe (higher-stress) conditions to help the company plan ahead.

By using the Arrhenius life stress model, and the most appropriate typical activation energy for electronics found by FIDES, a thermal acceleration factor could be found by [35]:

$$\text{AF}(T) = \frac{\text{MTTF}(T0)}{\text{MTTF}(T1)} = \exp\left(\frac{B}{T0} - \frac{B}{T1}\right) \tag{9}$$

Let:

$$\varepsilon = \exp\left(\frac{B}{T0} - \frac{B}{T1}\right) \tag{10}$$

Therefore,

$$MTTF(T1) = (MTTF(T0))\text{x}(\varepsilon^{-1}) \tag{11}$$

Acceleration factors show how TTF at a particular operating stress level (for one failure mode or mechanism) can be used to predict the equivalent TTF at a different operating stress level. The QALT is designed to quantify the life of the PCB and produce the data required for life data analysis. The QALT analysis method can employ life time under normal stress ($T0$) to quantify the life-time of the PCB under higher thermal stresses ($T1$).

5.5. Sensitivity Analysis of PCB and its Components' Lifetime Under Thermal Stress

By using the component life time under use (i.e. under normal temperature stress) as per the component life time methodology presented in chapter two of this thesis, the life time of each PCB component can be calculated under any thermal stress using Equation 11 presented before. Eventually the entire PCB life time under its components' different stress thermal values ($T1$) can be calculated. Figure 18 below shows how the temperature affects the PCB components life time. Where the lifetime of four components (resistors and capacitors) were observed over different thermal stresses. As shown in this figure, the life time is decreased to almost the half for every 10 Celsius degrees rise in temperature. For example the lifetime of component R36 is 2 years under 41^0 C, where its lifetime is 1 year under 50^0 C.

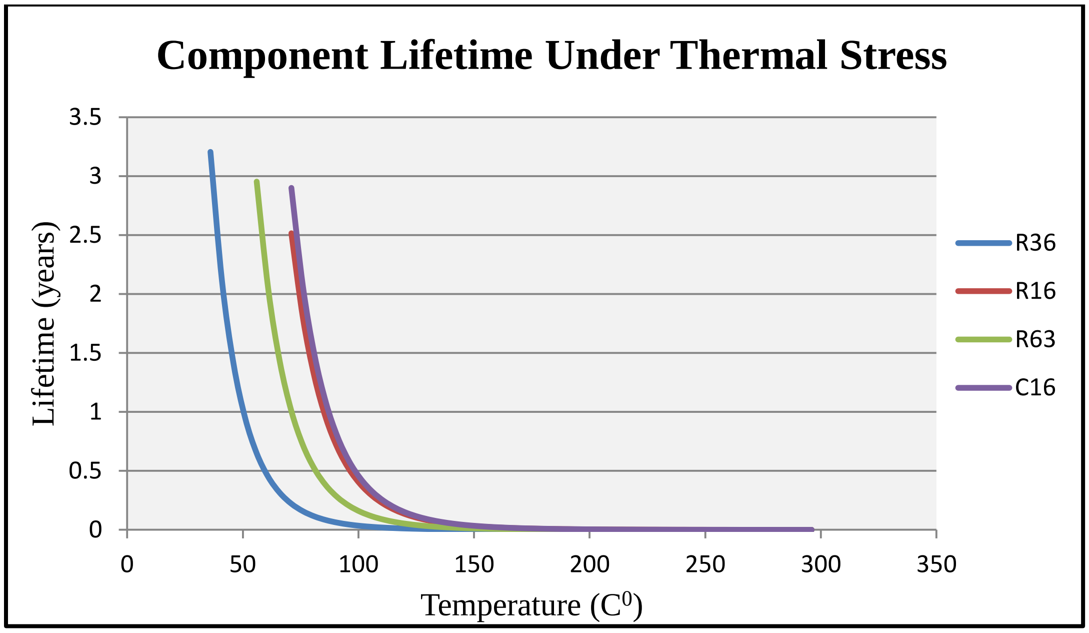


Figure 18. Four examples of PCB components life time over a thermal stress

5.5.1 Implementation and Case Study

As a case study, aluminum electrolytic capacitors' lifetime was examined and validated using the developed methodology. The NIC Technical Product Marketing Group [40] found that the life expectancy of aluminum electrolytic capacitors is 4000 hours @ +85°C (358.15 K); let *Ea* = 0.7, and the value of *B* is equal to 7543.23. By using Equation 11, and considering the thermal stress to be increased by 10° C (working at 95° *C* (368.15K)), then the acceleration factor for this component is 1.85 for every 10° C increase in thermal stress. From the above result, we can calculate L(368.18K), which is equal to $L(358.15)\ \mathrm{x}\ \varepsilon^{-1} = 1978.2$ hours.

Table 10 and Figure 19 illustrate how the aluminum electrolytic capacitors' life time decreases with increased temperature, which clarify the effect of thermal stress on the component life time. Where for every 10° C rise in temperature, aluminum electrolytic capacitors' lifetime is decreased almost by half; this result supports the electronics lifetime empirical results [22].

Table 10. Lifetime vs. Thermal Stress for Aluminum Electrolytic Capacitor

Aluminum Electrolytic Capacitor	
Life Time (Hours)	**Temperature (°C)**
1000	+105
2000	+95
4000	+85
8000	+75
16,000	+65
32,000	+55
64,000	+45
128,000	+35

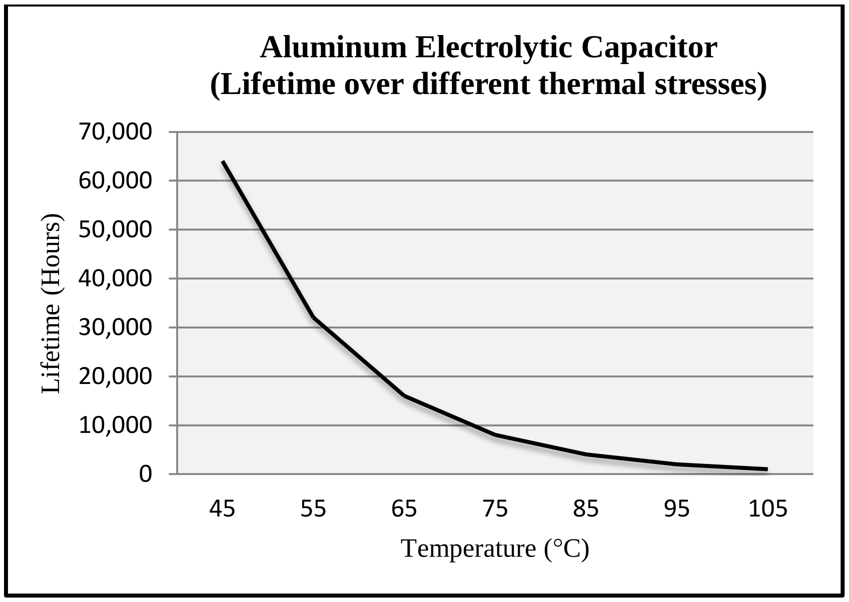


Figure 19. The lifetime of an Aluminum electrolytic capacitor decreases by increasing the temperature.

5.6. Sensitivity Analysis of PCB and its Components' Reliability Under Thermal Stress

Many reliability engineers and system designers consider temperature to be a major factor affecting the reliability of electronic equipment. They often use temperature reduction as the primary means to improve reliability. Unfortunately, in many cases, without understanding the actual reliability or the hidden costs associated with the temperature reduction.

In this research, a methodology for temperature effect on PCB failures was established. As the lifetime of each PCB component under thermal stress (*T*1) can be found by using the component lifetime under normal conditions (described in the component-level lifetime methodology presented in Chapter 2 of this thesis), then by using Equation 11, lifetime for that component under (*T*1) can be calculated. After finding the lifetime for each PCB component under its thermal stress, the system level-methodology and simulation presented in Chapter 2 of this thesis can be used with the new lifetime for components under stress. The component under stress has a lower value of TTF in comparison with its TTF values under normal conditions; the TTF values for those components are divided by the acceleration factor of each component. Eventually, the thermal effect is reflected in the PCB failure list, and the next lifetime of the entire PCB is predicted.

Component reliability and failure rates at higher stress (*T*1) can be calculated using the following equations [29]:

$$R_s(\mathrm{t}) = R_u(t.\varepsilon) \tag{12}$$

$$\lambda_s = \lambda_u.\varepsilon \tag{13}$$

$$F_s(\mathrm{t}) = F_u(t.\varepsilon) \tag{14}$$

Figure 20 shows how a reliability of a PCB component and how this reliability is affected by thermal stress. Component C16 was assessed under different thermal stresses values (72^0, 77^0 and 97^0 C respectively) and its reliability decreases over time was noted.

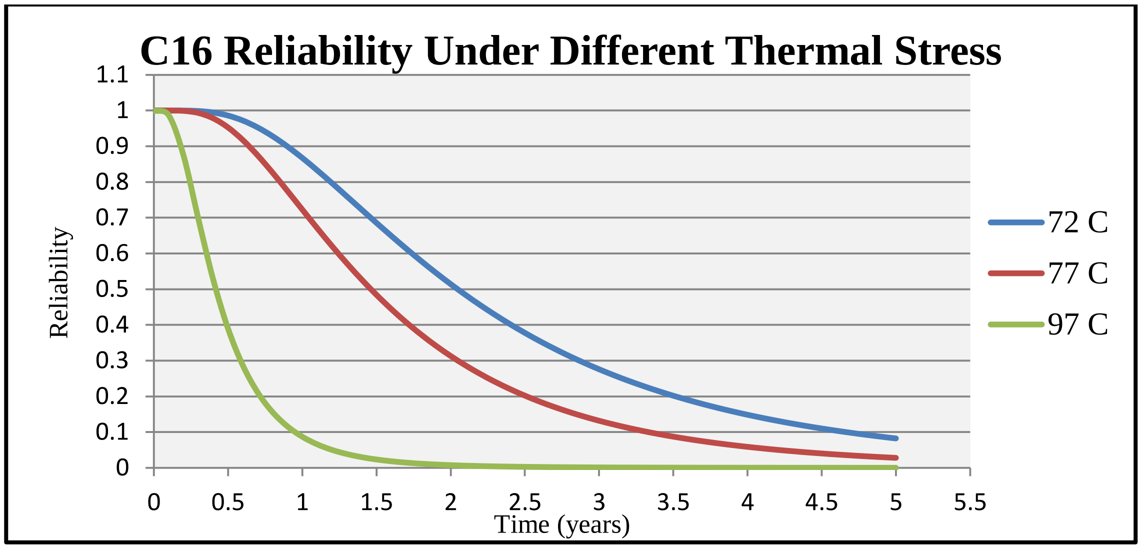


Figure 20. Example of a PCB component and how its reliability is affected by different thermal stresses

PREVIEW was used to display the thermal effect on reliability and lifetime at the component and system levels. The user can enter any thermal stress for a specific component, where the lifetime and reliability of that component and of the entire PCB appears on the PREVIEW display, see Figure 21. In Figure 21(a), a 3D graph represents how temperature (thermal stress) affects component level reliability, as the reliability slop gets steeper with the raise in temperature. In other words, component level reliability decreases by increasing the temperature of the component. As can be depicted from the graph, higher thermal stress triggers decrease in the reliability for a component. Figure 21(b) illustrates how the temperature affected the lifetime of circuit card components. The lifetime is decreased by approximately 50% with an increase in 10 $^{\circ}$C in the temperature, where the x-axis represents the increase in temperature over the normal (used) temperature. Figure 21(c) represents the effect of 10 $^{\circ}$C increase in temperature of components under thermal stress on the system level reliability significantly.

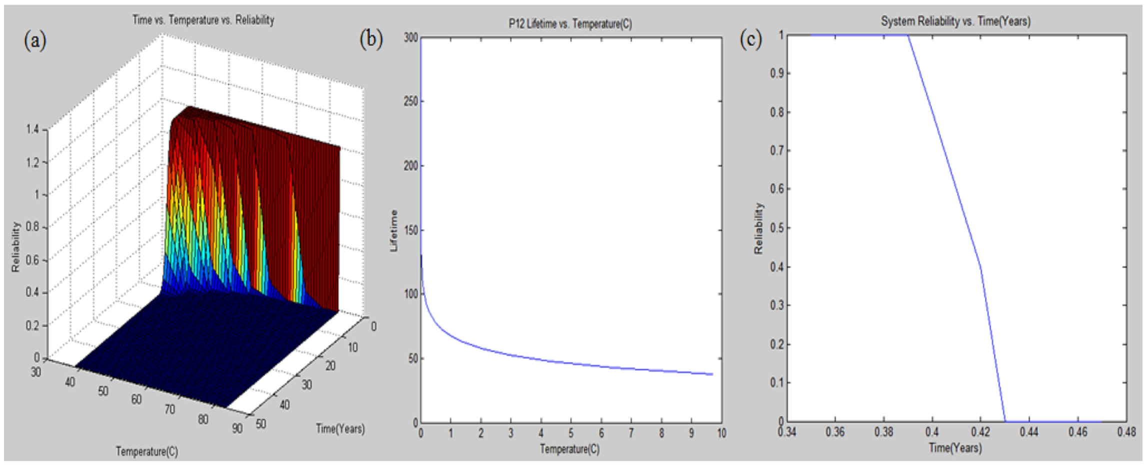


Figure 21. Effect of thermal stresses: (a) component-level reliability, (b) component lifetime, and (c) system-level reliability.

Figure 22 shows how system level reliability is affected by different thermal stresses on specific components, where three scenarios (each with different thermal stress) are shown. In the first scenario, the temperature value of components R15 and R41 was increased to 54°C and 56°C, respectively. In the second scenario, the temperature value of components C11 and R41 was increased to 74°C and 46°C, respectively. A similar procedure was applied for the last scenario where the temperature values of components C27, C11, R15, R41 and R42 was increased to 65°C, 84°C, 54°C, 46°C, and 40°C, respectively. As shown in Figure 22 (a) the reliability starts decreasing around 0.36 years, where it starts decreasing around 2.4 years as illustrated in Figure 22 (b).

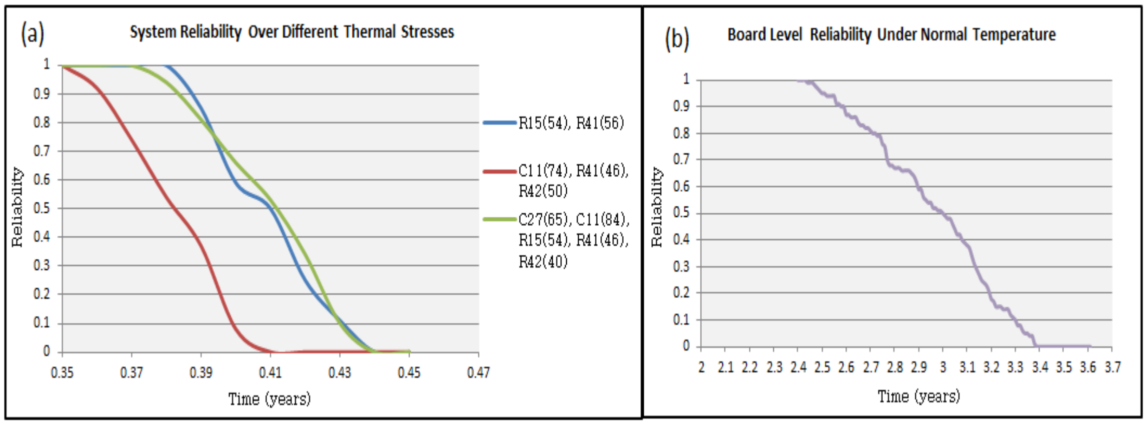


Figure 22. System level reliability under: (a) different thermal stresses (b) normal use temperature

CHAPTER VI

SUMMARY AND FUTURE WORK

6.1. Summary and Conclusion

In this thesis, we have researched, developed, and demonstrated the technology to leverage the limitations of virtual test and predictive remaining lifetime analysis of circuit cards to provide support to maintenance engineers in nuclear power plants worldwide. It highlights the development of immerse and high fidelity virtual environment for lifetime and reliability prognostics for nuclear power electronics. The developed virtual environment allows prediction of total life time, overall reliability and maintainability for circuit cards (system level) and their components (component level) through a new simulation methodology. Component repair or replace approaches are developed within this simulation tool, which gives the user the ability to choose between them based on experience and component history. Physics-based multi-scale modeling, analysis and visualization capabilities were developed to predict thermal stressor for passive and active components. In addition, this research provides a better understanding in prediction of overall system failure characteristics for any given configuration. It allows the user to identify components which contribute the most to downtime and to determine the effect of design alternatives on system performance in a cost-effective manner.

This research provides effective methodologies for determining where corrective action may be particularly helpful, and it helps predict the overall system failure characteristics for any given configuration. It can be considered a powerful process that utilizes failure information from a system's component in order to develop probability distributions for whether the system will be able to perform its intended function. It helps in identifying components that contribute the most to downtime and in determining the effect of design alternatives on system performance in a cost-effective manner (i.e., using virtual modeling rather than prototype testing).

Because excessive temperature is the primary cause of poor reliability in electronic equipment, quantitative accelerated life tests are designed to quantify the life of the PCB under different thermal stresses and produce the data required for accelerated life data analysis. This thermal stress methodology can help in making design decisions that meet the system reliability requirements, as well as determining the maximum allowable component temperature.

6.2. Future Work

During the next phase, we will maneuver the effort towards integrating other stressors into the reliability model such as mechanical loading, humidity effect, and vibrational effect. On the other hand, and as the next effort; we will pursue our efforts in developing, and demonstrating the technology to reverse engineer nuclear electronics to leverage the limitations of virtual testing and predictive remaining lifetime analysis of circuit cards in nuclear power plants. As In-service circuit cards in nuclear power plants necessitate a new module, with which older technology can be reengineered within the PREVIEW environment, which eventually will reduce decision-making time and lower the cost associated with testing and maintaining circuit cards.

APPENDIX A

SAMPLE OF HISTORICAL TIME-TO-FAILURE (YEARS) FOR A PCB COMPONENTS' SAMPLE

C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	$\ldots C_n$
11.50	11.10	9.81	11.60	11.00	11.60	8.11	10.20	9.77	8.85
9.13	10.60	9.71	10.40	9.66	9.01	9.68	10.50	8.92	9.60
9.45	11.30	11.60	9.02	11.60	9.07	8.32	8.82	11.60	10.10
10.00	11.70	9.73	11.80	11.10	10.00	10.00	11.80	8.43	10.10
11.30	8.09	8.26	10.70	9.11	10.80	11.10	9.20	10.40	10.30
10.10	9.53	11.00	8.97	9.99	10.30	10.30	8.26	11.80	9.12
9.68	10.00	10.70	9.19	11.20	10.10	8.80	8.66	8.34	8.60
9.70	9.25	9.41	9.57	9.03	10.10	8.93	9.99	10.80	9.75
10.40	10.30	9.67	8.99	11.00	10.50	11.50	10.40	10.90	11.10
8.02	10.10	8.05	11.90	9.11	11.10	10.90	9.72	11.50	8.24
11.10	9.11	11.50	8.87	8.03	9.50	9.58	10.40	8.11	9.15
11.20	8.41	11.10	10.90	9.57	9.67	11.00	9.27	9.52	11.80
11.20	9.99	11.00	10.10	10.40	8.74	8.42	9.46	9.09	9.62
10.20	8.29	10.60	11.30	8.07	11.00	9.73	9.95	11.30	11.20
8.82	9.93	9.05	8.92	10.40	9.54	11.40	10.10	11.10	9.98
8.17	8.34	11.90	8.04	8.38	9.51	9.03	8.61	11.40	10.50
9.24	10.10	9.83	10.30	10.30	10.10	8.74	11.00	8.97	10.50
11.20	10.80	10.80	8.37	10.50	10.10	8.08	12.00	9.70	8.87
8.55	10.20	10.60	10.10	11.60	9.56	10.70	9.35	9.77	11.80
9.15	12.00	8.11	11.90	9.56	8.85	11.00	8.19	9.11	10.60
8.54	11.80	8.47	8.23	10.60	11.30	9.47	8.02	11.30	8.78
10.60	10.40	8.77	8.43	10.10	8.83	9.89	9.44	9.59	9.27
10.40	9.00	8.47	10.40	9.29	10.90	8.67	10.40	9.61	11.80
11.50	11.30	11.40	8.03	8.42	10.70	8.56	10.10	10.40	9.40
9.74	9.16	11.30	9.77	8.40	11.90	9.09	10.60	11.20	10.80

APPENDIX B

SAMPLE OF HISTORICAL TIME-TO-REPAIR/REPLACE (YEARS) FOR A PCB COMPONENTS SAMPLE

C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	$\ldots C_n$
0.00228	0.00181	0.00048	0.01280	0.00761	0.00926	0.01080	0.00634	0.00798
0.00102	0.00623	0.00479	0.00678	0.00093	0.00540	0.00430	0.00317	0.00830
0.00581	0.00328	0.00876	0.00916	0.01360	0.00410	0.00294	0.00257	0.00320
0.00655	0.00401	0.00900	0.00833	0.00046	0.00982	0.00004	0.01200	0.00126
0.00725	0.00580	0.00930	0.01230	0.00538	0.00864	0.01360	0.00862	0.00396
0.00914	0.00179	0.00928	0.00993	0.00810	0.00270	0.01350	0.01330	0.01240
0.00121	0.00400	0.00180	0.00760	0.00629	0.00889	0.00735	0.00635	0.00524
0.00961	0.00970	0.00003	0.00588	0.00125	0.00156	0.00787	0.01250	0.00884
0.00808	0.00013	0.00617	0.00524	0.00165	0.01280	0.00131	0.01110	0.00337
0.00220	0.00278	0.00192	0.00387	0.01180	0.00363	0.01320	0.01070	0.00440
0.01330	0.00814	0.01120	0.00450	0.01180	0.00024	0.00796	0.00356	0.01150
0.01070	0.00300	0.00696	0.00522	0.00848	0.00918	0.00102	0.00203	0.00595
0.00973	0.00434	0.01290	0.01200	0.00352	0.00210	0.00027	0.00616	0.00756
0.00219	0.00686	0.00866	0.00352	0.00649	0.00491	0.01260	0.00339	0.01280
0.00957	0.01140	0.00724	0.00722	0.00539	0.00460	0.00638	0.00608	0.01020
0.00046	0.01010	0.00827	0.01280	0.00725	0.00609	0.00276	0.00675	0.00546
0.01140	0.00276	0.00717	0.00289	0.00162	0.00050	0.00043	0.00956	0.01090
0.00774	0.00268	0.00728	0.01270	0.01130	0.01210	0.00685	0.00982	0.00316
0.01210	0.01060	0.00140	0.00746	0.00571	0.01130	0.00704	0.00271	0.00537
0.00226	0.00852	0.00563	0.01030	0.00927	0.01310	0.01130	0.00442	0.01320
0.01010	0.00084	0.00690	0.00515	0.01060	0.00008	0.00714	0.00274	0.00386
0.01260	0.00128	0.00576	0.01220	0.00431	0.00221	0.00073	0.00627	0.00115
0.00978	0.00685	0.01120	0.00430	0.00462	0.00460	0.00784	0.00782	0.00472
0.00850	0.00841	0.00659	0.00357	0.00564	0.01040	0.00196	0.00554	0.00442
0.00705	0.00333	0.00729	0.01250	0.01350	0.00755	0.00154	0.00537	0.00432

APPENDIX C

BOARD LEVEL RELIABILITY DATA FOR 10 RUNS

Time Years)	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
2.43	1	1	1	1	1	1	1	1	1	1
2.44	1	0.99	1	1	1	1	1	1	1	0.99
2.45	0.99	0.99	0.98	0.99	1	0.98	1	0.99	1	0.99
2.46	0.99	0.98	0.96	0.98	1	0.98	0.99	0.98	0.98	0.99
2.47	0.98	0.98	0.94	0.98	1	0.97	0.99	0.96	0.98	0.98
2.48	0.98	0.98	0.94	0.98	0.99	0.96	0.98	0.96	0.98	0.97
2.49	0.98	0.95	0.94	0.96	0.99	0.95	0.98	0.96	0.98	0.96
2.5	0.98	0.94	0.93	0.95	0.99	0.94	0.98	0.96	0.98	0.95
2.51	0.97	0.94	0.93	0.93	0.99	0.94	0.96	0.95	0.98	0.95
2.52	0.97	0.93	0.92	0.93	0.97	0.93	0.96	0.95	0.97	0.94
2.53	0.97	0.92	0.91	0.92	0.96	0.93	0.96	0.95	0.97	0.94
2.54	0.96	0.91	0.89	0.92	0.96	0.93	0.94	0.95	0.97	0.94
2.55	0.96	0.9	0.88	0.91	0.96	0.93	0.94	0.94	0.97	0.94
2.56	0.96	0.89	0.87	0.9	0.94	0.92	0.94	0.94	0.96	0.91
2.57	0.95	0.89	0.87	0.9	0.91	0.9	0.91	0.92	0.96	0.91
2.58	0.95	0.88	0.86	0.9	0.91	0.9	0.89	0.9	0.93	0.9
2.59	0.95	0.88	0.85	0.9	0.89	0.9	0.89	0.9	0.93	0.9
2.6	0.94	0.88	0.85	0.9	0.89	0.9	0.89	0.9	0.92	0.87
2.61	0.93	0.88	0.84	0.9	0.87	0.89	0.88	0.89	0.92	0.87
2.62	0.93	0.87	0.83	0.9	0.87	0.89	0.87	0.88	0.9	0.86
2.63	0.91	0.86	0.82	0.88	0.87	0.88	0.87	0.88	0.9	0.86
2.64	0.91	0.85	0.82	0.88	0.87	0.88	0.87	0.88	0.89	0.86
2.65	0.9	0.83	0.82	0.86	0.87	0.88	0.86	0.87	0.89	0.84
2.66	0.89	0.83	0.82	0.84	0.86	0.88	0.85	0.86	0.88	0.83
2.67	0.89	0.81	0.81	0.83	0.86	0.86	0.85	0.86	0.88	0.83
2.68	0.89	0.79	0.81	0.8	0.84	0.85	0.83	0.84	0.87	0.82
2.69	0.87	0.79	0.81	0.79	0.82	0.84	0.83	0.83	0.86	0.82
2.7	0.84	0.78	0.81	0.77	0.8	0.84	0.82	0.83	0.86	0.81
2.71	0.84	0.75	0.81	0.76	0.8	0.83	0.81	0.83	0.86	0.8
2.72	0.84	0.75	0.81	0.76	0.8	0.83	0.77	0.78	0.84	0.8
2.73	0.83	0.74	0.8	0.73	0.78	0.82	0.77	0.75	0.82	0.79
2.74	0.82	0.74	0.77	0.7	0.78	0.81	0.77	0.73	0.81	0.79
2.75	0.82	0.72	0.76	0.69	0.76	0.8	0.77	0.69	0.79	0.76
2.76	0.82	0.71	0.75	0.66	0.75	0.78	0.75	0.66	0.78	0.75
2.77	0.8	0.7	0.74	0.65	0.72	0.77	0.73	0.64	0.78	0.7
2.78	0.74	0.7	0.72	0.63	0.69	0.76	0.72	0.64	0.77	0.68
2.79	0.74	0.68	0.71	0.62	0.67	0.72	0.72	0.62	0.75	0.68

2.8	0.71	0.67	0.66	0.6	0.67	0.71	0.72	0.6	0.73	0.67
2.81	0.7	0.66	0.65	0.56	0.66	0.71	0.7	0.58	0.69	0.67
2.82	0.68	0.65	0.62	0.55	0.63	0.68	0.68	0.58	0.67	0.67
2.83	0.66	0.63	0.6	0.55	0.63	0.67	0.65	0.57	0.65	0.66
2.84	0.62	0.63	0.59	0.51	0.62	0.66	0.62	0.56	0.62	0.66
2.85	0.61	0.63	0.59	0.5	0.61	0.65	0.61	0.55	0.6	0.66
2.86	0.59	0.61	0.57	0.48	0.59	0.65	0.6	0.53	0.57	0.66
2.87	0.57	0.6	0.55	0.48	0.59	0.63	0.56	0.53	0.56	0.65
2.88	0.55	0.57	0.53	0.46	0.55	0.61	0.5	0.52	0.55	0.64
2.89	0.53	0.55	0.52	0.44	0.55	0.59	0.48	0.49	0.53	0.62
2.9	0.52	0.55	0.51	0.42	0.55	0.57	0.48	0.47	0.51	0.59
2.91	0.52	0.54	0.49	0.42	0.53	0.56	0.48	0.46	0.49	0.59
2.92	0.51	0.51	0.47	0.42	0.53	0.56	0.48	0.45	0.45	0.56
2.93	0.5	0.48	0.43	0.42	0.52	0.55	0.44	0.43	0.45	0.55
2.94	0.47	0.47	0.42	0.4	0.5	0.55	0.44	0.39	0.43	0.54
2.95	0.47	0.44	0.39	0.4	0.49	0.53	0.42	0.37	0.42	0.54
2.96	0.46	0.42	0.39	0.4	0.47	0.51	0.4	0.35	0.4	0.52
2.97	0.44	0.41	0.38	0.35	0.44	0.49	0.39	0.35	0.4	0.52
2.98	0.42	0.41	0.36	0.33	0.43	0.49	0.35	0.35	0.39	0.51
2.99	0.4	0.41	0.36	0.33	0.42	0.48	0.33	0.34	0.38	0.51
3	0.36	0.39	0.34	0.32	0.41	0.46	0.29	0.32	0.36	0.5
3.01	0.34	0.37	0.33	0.3	0.41	0.43	0.28	0.31	0.33	0.49
3.02	0.31	0.35	0.32	0.3	0.4	0.42	0.28	0.31	0.31	0.48
3.03	0.29	0.31	0.3	0.29	0.36	0.41	0.26	0.28	0.3	0.48
3.04	0.26	0.3	0.29	0.26	0.32	0.4	0.26	0.28	0.28	0.46
3.05	0.26	0.3	0.26	0.26	0.29	0.4	0.26	0.26	0.25	0.44
3.06	0.26	0.3	0.25	0.25	0.28	0.39	0.25	0.26	0.24	0.42
3.07	0.25	0.28	0.25	0.25	0.26	0.37	0.25	0.25	0.24	0.42
3.08	0.23	0.27	0.24	0.22	0.23	0.36	0.25	0.25	0.24	0.4
3.09	0.23	0.27	0.23	0.19	0.23	0.33	0.24	0.24	0.24	0.39
3.1	0.22	0.26	0.22	0.19	0.23	0.33	0.21	0.22	0.24	0.38
3.11	0.22	0.26	0.22	0.18	0.2	0.33	0.2	0.21	0.23	0.37
3.12	0.2	0.24	0.2	0.17	0.17	0.32	0.19	0.2	0.21	0.34
3.13	0.2	0.23	0.2	0.15	0.17	0.31	0.19	0.18	0.19	0.31
3.14	0.17	0.23	0.18	0.15	0.15	0.31	0.19	0.17	0.19	0.29
3.15	0.17	0.23	0.17	0.13	0.15	0.29	0.15	0.14	0.18	0.27
3.16	0.17	0.22	0.16	0.13	0.13	0.27	0.15	0.14	0.16	0.25
3.17	0.16	0.2	0.14	0.12	0.13	0.26	0.15	0.13	0.14	0.24
3.18	0.16	0.19	0.12	0.11	0.12	0.23	0.14	0.13	0.13	0.23
3.19	0.15	0.17	0.12	0.11	0.1	0.23	0.14	0.13	0.13	0.21
3.2	0.13	0.16	0.12	0.11	0.08	0.2	0.13	0.12	0.12	0.18

3.21	0.12	0.15	0.12	0.1	0.06	0.18	0.13	0.12	0.12	0.17
3.22	0.11	0.15	0.11	0.1	0.06	0.18	0.13	0.12	0.12	0.15
3.23	0.09	0.14	0.11	0.1	0.06	0.16	0.12	0.12	0.1	0.15
3.24	0.08	0.11	0.1	0.09	0.05	0.16	0.11	0.11	0.1	0.15
3.25	0.08	0.11	0.09	0.09	0.04	0.16	0.1	0.11	0.09	0.14
3.26	0.08	0.1	0.08	0.09	0.04	0.15	0.1	0.11	0.09	0.14
3.27	0.07	0.1	0.08	0.08	0.04	0.14	0.07	0.1	0.09	0.14
3.28	0.07	0.1	0.07	0.06	0.04	0.09	0.07	0.1	0.09	0.12
3.29	0.07	0.1	0.07	0.06	0.04	0.08	0.07	0.08	0.09	0.11
3.3	0.05	0.08	0.06	0.04	0.04	0.08	0.06	0.08	0.08	0.1
3.31	0.05	0.07	0.06	0.03	0.03	0.08	0.06	0.07	0.07	0.08
3.32	0.05	0.06	0.05	0.03	0.03	0.07	0.06	0.06	0.07	0.08
3.33	0.04	0.05	0.04	0.03	0.03	0.05	0.06	0.06	0.07	0.06
3.34	0.04	0.05	0.04	0.03	0.02	0.05	0.05	0.06	0.05	0.05
3.35	0.03	0.05	0.03	0.03	0.02	0.05	0.05	0.06	0.05	0.05
3.36	0.02	0.04	0.01	0.03	0.02	0.04	0.04	0.05	0.03	0.04
3.37	0.02	0.04	0.01	0.02	0.02	0.04	0.04	0.04	0.01	0.04
3.38	0.02	0.04	0.01	0.01	0.02	0.03	0.04	0.04	0.01	0.01
3.39	0.02	0.03	0.01	0.01	0.01	0.03	0.02	0.04	0.01	0
3.4	0.02	0.03	0.01	0.01	0.01	0.03	0.02	0.04	0.01	0
3.41	0.02	0.03	0.01	0.01	0.01	0.03	0.01	0.04	0.01	0
3.42	0.01	0.03	0.01	0.01	0.01	0.02	0.01	0.04	0.01	0
3.43	0.01	0.02	0.01	0.01	0.01	0.02	0	0.04	0.01	0
3.44	0.01	0.02	0.01	0	0.01	0.02	0	0.04	0.01	0
3.45	0.01	0.02	0.01	0	0.01	0.02	0	0.04	0.01	0
3.46	0.01	0.02	0.01	0	0.01	0.02	0	0.04	0.01	0
3.47	0.01	0.01	0.01	0	0.01	0.01	0	0.04	0.01	0
3.48	0.01	0.01	0.01	0	0.01	0.01	0	0.04	0.01	0
3.49	0.01	0.01	0.01	0	0.01	0.01	0	0.04	0.01	0
3.5	0.01	0.01	0.01	0	0.01	0	0	0.04	0.01	0
3.51	0	0.01	0.01	0	0.01	0	0	0.03	0.01	0
3.52	0	0.01	0.01	0	0.01	0	0	0.02	0.01	0
3.53	0	0.01	0.01	0	0.01	0	0	0.01	0.01	0
3.54	0	0.01	0.01	0	0.01	0	0	0.01	0.01	0
3.55	0	0	0.01	0	0.01	0	0	0.01	0.01	0
3.56	0	0	0	0	0.01	0	0	0.01	0.01	0
3.57	0	0	0	0	0	0	0	0.01	0.01	0
3.58	0	0	0	0	0	0	0	0.01	0.01	0
3.59	0	0	0	0	0	0	0	0.01	0	0
3.6	0	0	0	0	0	0	0	0.01	0	0
3.61	0	0	0	0	0	0	0	0	0	0

APPENDIX D

BOARD LEVEL MAINTAINABILITY DATA FOR 10 RUNS

Time (Years)	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
0	0	0	0	0	0	0	0	0	0	0
0.01	0.1	0.14	0.1	0.11	0.1	0.1	0.09	0.08	0.11	0.11
0.02	0.81	0.85	0.83	0.76	0.8	0.8	0.78	0.84	0.83	0.83
0.03	0.99	0.99	1	0.99	1	1	1	1	1	1
0.04	1	1	1	1	1	1	1	1	1	1
0.05	1	1	1	1	1	1	1	1	1	1

APPENDIX E

LIFE TIME / RELIABILITY CODE

```
#include "stdafx.h"

#define PI 3.14159265358979323846264338;

int componentNumberNewValue;
int manualComponentSave;
string manualComponent;

struct thermal{
	string partID;
	double MTTF;
	double epsillon;
	int componentNumber;
};

struct manualAges{
	double age;
	string component;
};

struct lifeRanges{
	double maxFail;
	double minRemain;

	vector <double> fail;
	vector <double> live;
};

struct parallelFail{
	string initialComponent;
	vector <string> componentsWithin;
};

struct failing{
	double mean;
	double standardDeviation;

	double meanFix;
	double standDevFix;

	int partsFailed;

	vector <string> partID;
	vector <bool> chip;
	vector <bool> chipConnection;
	vector <int> parallelComponent;
	vector <bool> parallel;
	vector <double> fix;
	vector <double> SD;
	vector <double> mu;
	vector <double> failed;
	double maxFail;
	double minFail;
	string initialParallel;
	vector <string> parallelWithin;
};

struct parallelComponent{
	string partID;

	int current_position;
	int netIN;
	int netOUT;

	double MTTF;
	double maxFail;
	double minRemain;

	bool crash;
	bool connectionMade;
	bool componentCheck;
	bool extraNetworks;
	bool neverFail;

	vector <int> componentsWithin;
	vector <int> allNets;
	vector <double> fails;
	vector <double> unFails;
};

struct seriesComponent{
	string partID;

	int networkIN;
	int networkOUT;

	double lowMTTF;
	double MTTF;
	double TTRepair;
	double TTReplace;
	double Fix;

	bool connectionMade;
	bool extraNetworks;
	bool Fail;
	bool seriesFail;
	bool unFail;
	bool seriesUnfail;
	bool repairMIN;
	bool replaceMIN;
	bool immediateFix;
	bool immediateFail;
	bool chipConnection;

	vector <int> allNets;
	vector <int> componentsWithin;
	vector <double> fails;
	vector <double> unFails;
};
```

```
struct componentNetworks{
	string partID;

	double MTTF;

	int networkIN;
	int networkOUT;
	vector <int> extraNets;
	vector <int> allNets;
};

// structures used to hold the final values of the replication or replications
struct statisticsVectors{
	string partID;

	int componentNumber;
	int timesFailed;
	int distribution;

	double mttrSaveSave;
	double replaceSaveSave;
	double mttfSaveSave;
	double MTTR;
	double epsillon;
	double reliabilityAge;
	double age;
	double mean;
	double sigma;
	double logMean;
	double logSigma;
	double eta;
	double beta;
	double expoMean;
	double expoSigma;
	double weibullMean;
	double weibullSigma;
	double logNormMean;
	double logNormSigma;
	double stdDEVTBF;
	double halfWidthTBF;
	double componentFailProb;
	double MTTF;
	double MTTFsave;
	double TTR_total;
	double replaceTotal;
	double MTTF_total;
	double TTR_save;
	double TTReplace_save;
	double lowerRange;
	double upperRange;
	double invLowerRange;
	double invUpperRange;
	double distance;
	double failureRate;
	double compFailProbTotal;
	double TTFmin;
	double TTFmax;
	vector <double> reliability;
	vector <double> reliabilityTime;
	vector <double> failureRateAverage;
	vector <double> timeBetweenFail;
	vector <double> timeBetweenFailCDF;
	vector <double> repair;
	vector <double> repairCDF;
	vector <double> replace;
	vector <double> replaceCDF;
	vector <double> replicationMTTF;
	vector <int> TBFrank;
	vector <int> repairRank;
	vector <int> replaceRank;

	bool lowestMTBF;
	bool FAIL;
	bool replacement;
	bool repairing;
	bool parallelComponent;
	bool distributionFound;
	bool initialThermalFail;
};

// structure containing the elements of each component in a PCB
struct Component{
	int componentNumber;
	int distributionTypeHoldTBF;
	int distributionTypeHoldRepair;
	int distributionTypeHoldReplace;
	int rank;

	double epsillon;
	double MTTF;
	double sigma;
	double inRepair;
	double inReplace;
	double reliability;
	double TBF;					// Components Mean Time 
Between Failure
	double TTR;					// Components Mean Time To 
Repair
	double replace;					// Components replacement 
assignment
	double normalDistance;
	double exponentialDistance;
	double logNormalDistance;
	double WeibullDistance;
	double normalProbability;
	double exponentialProbability;
	double logNormalProbability;
	double weibullProbability;
	double CDF;
	double value;
	double maxTTF;
	double minTTF;
	double logMTTF;
	double logSIGMA;
	double beta;
	double eta;
	double repairEta;
	double repairBeta;
	double replaceEta;
	double replaceBeta;
	double repairSigma;
	double replaceSigma;

	vector <double> inputMTTF;
	vector <double> inputMTTR;
	vector <double> inputMTTReplace;

	//bool printFAIL;				// component print fail statement
	bool printReplace;			// componnent print replace statement
	bool printRepair;			// component print repair statement
	bool subtractTBF;			// component subtract statement
	bool subtractTTR;			// component subtract mttr statement
	bool subtractReplace;			// component subtract replacement
statement
	bool replacement;			// choice to replace not repair
	bool repairing;				// choice to repair not replace
	bool FAIL;			// components fail declaration if a component
fails due to network connections
	bool polyNum;
	bool cdfed;
	bool parallelComponent;
};

struct systemReliability{
	int networkIn;
	int networkOut;

	bool extraNetworks;
	bool connectionMade;
	double timeBegin;
	double timeEnd;

	vector <int> allNets;
	vector <double> reliability;
};

// function prototypes (in order of use)
int numberColumns(string columnCount);
void importFREPLREPA_files(statisticsVectors *, int numRows, int numCols, string
*fileName, int whatToFill);
void importNetworksFile(componentNetworks *, string *fileName, int numCols);
void importAgeFile(statisticsVectors*, int numCols, string *fileName);
void importThermalFile(vector <thermal> & thermalComponent, string *fileName);

void CDF_calc(statisticsVectors *, statisticsVectors *, int numRows, int numCols,
int CDF_TYPE);
double MEAN(Component *, int numRows);
double STDDEV(Component *, int numRows, double mean);
double log_mean(Component *, int n);
double log_sigma(Component *, int n, double logMean);
double betaCalc(Component *, int numRows, double mean, double sigma);
double etaCalc(Component * , int numRows, double beta, double mean);
double lognormalFinalSigma(double mean, double sigma);
double lognormalFinalMean(double mean, double sigma);
double exponentialSigma(double eta);
double exponentialMean(double eta);
double weibullSigma(double beta, double eta);
double weibullMean(double beta, double eta);
void initialMeans(statisticsVectors *, Component *, int numCols);

failing totalFails(failing failTable, double manualTTF, double manualTTR, double
manualReplace, double timeInc, double runLength, statisticsVectors *, int numCols,
int boolManualRepair, int boolManualReplace, int boolManualTTF, int replication,
vector <thermal> & thermalComponent, Component * pointerToResCapLife, Component
*);

void normal_probability(Component *, int numRows, double mean, double sigma);
void exponential_probability(Component *, int numRows, double mean);
void lognormal_probability(Component *, int numRows, int n, double sigma, double
mean);
void weibull_probability(Component *, int numRows, double betas, double etas);

double find_distances(Component *,statisticsVectors *, statisticsVectors *, int
numRows, double mean, double sigma, double expoMean, double beta, double eta,
double logMean, double logSigma, double logNormSigma, double logNormMean, int
componentPosition, int randomNumberType, bool generateNumber);
double find_lowest_distance(Component *,statisticsVectors *, statisticsVectors *,
int numRows, double sigma, double mean, double expoMean, double beta, double eta,
double logMean, double logSigma, double logNormSigma, double logNormMean, int
componentPosition, int randomNumberType, bool generateNumber);
double randomNumberGenerator(Component *,statisticsVectors *, statisticsVectors *,
int numRows, double min, double max, int distributionType, double mean, double
sigma, double expoMean, double beta, double eta, double logMean, double logSigma,
double logNormSigma, double logNormMean, int componentPosition, bool
generateNumber);
double NRoot(double num, double root);
double find_erf(double mean, double sigma, double randomProb, int normalType);
void findNewValue(Component *, statisticsVectors *, int numCols, statisticsVectors
*, int componentNumberNewValue, int randomNumberType, int currentRowLength, bool
generateNumber, vector <thermal> & thermalComponent, Component *);
void chooseDistribution(Component *, int randomNumberType, int distributionType);

void halfWidth(statisticsVectors *, int numCols);
void failureRate(statisticsVectors *, statisticsVectors *, Component *, int
numCols, double runLength);
void componentFailProb(statisticsVectors *, int numCols);

lifeRanges checkParallel(componentNetworks *, statisticsVectors *, Component *,
vector <lifeRanges> & finalLife, vector <systemReliability> & reliable, int
numCols, int saveSpotLowestTBF, bool allRep, double runLength);
lifeRanges chipFail(vector <seriesComponent> & chipLife, double runLength);
lifeRanges componentChipFail(vector <seriesComponent> & compConnectChip, double
runLength);
void seriesConnection(Component *, componentNetworks *, vector <seriesComponent> &
series, vector <systemReliability> & reliable, statisticsVectors *, int numCols,
bool allRep, double runLength);
lifeRanges findComponentLowestRange(vector <parallelComponent> &
parallelComponents, vector <seriesComponent> & series, vector <lifeRanges> &
finalLife, lifeRanges chips, lifeRanges compChipConnections, double runLength) ;
```

```
void reliability_calc(statisticsVectors *, statisticsVectors *, Component *,
vector <systemReliability> & componentReliability, int numCols, double runLength,
int boolManualTTF, bool MTTFyes, bool componentInput, int replication);
double finalNormal(statisticsVectors *, int componentPosition, double time, double
mean, double sigma);
double finalExponential(statisticsVectors *, int componentPosition, double time,
double expoMean);
double finalLognormal(statisticsVectors *, int componentPosition, double logMean,
double logSigma, double time);
double finalWeibull(statisticsVectors *, int componentPosition, double time,
double etas, double betas);
int reliabilitySeries(vector <systemReliability> & reliable);
int reliabilityParallel(vector <systemReliability> & reliable);
int advancedReliabilityParallel(vector <systemReliability> & reliable);
void reliabilityNetworks(vector <systemReliability> & reliable);
void minMaxTTF(Component *, statisticsVectors *, int numCols);
void computeSystemReliability(systemReliability & reliable, vector <failing> &
failTableIntegrated, vector <failing> & outputFailTable, int runLength, int
numberOfReplications);

void meanGenerateTTF(vector <statisticsVectors> & exportGeneratedMTTR);
void outputFinalTable(statisticsVectors *, int numCols, double lifetime, int
boolManualTTF, string fileName, string maintainFile, systemReliability & reliable,
vector <systemReliability> & componentReliability, int boolManualRepair, int
boolManualReplace, failing failTable, vector <parallelFail> & parallelFailing,
Component *, vector <failing> & outputFailTable, vector<failing> &
failTableIntegrated, vector <thermal> & thermalComponent, vector <double> &
systemFailureRate, vector <statisticsVectors> & exportGeneratedMTTR, vector
<failing> & outputMTTR);
void manual_TTF_TTR_REPLACE(Component *, int numCols, int type);
void MTTF_total(statisticsVectors *, statisticsVectors *, Component *, int
numCols, int replications, int numberOfReplications, int boolManualReplace, int
boolManualRepair, int boolManualTTF, double manualTTR, double manualReplace);
double advancedParallel(vector <seriesComponent> & series, statisticsVectors *
pointerToResCapStats, vector <parallelComponent> & advancedParallel);
double errorFunction(double x);
double advancedSeries(vector <seriesComponent> & series, statisticsVectors *
pointerToResCapStats, vector <parallelComponent> & parallelComponents, double
runLength);

void main(int argc, char* argv[])
{
       fstream setupFile;
       string setupFilePath = argv[1];
       setupFile.open(setupFilePath.c_str());
       string currentLine;
       vector <manualAges> manual_age;
       manualAges currentManual;
       string componentID;
       double age;
       vector <thermal> thermalComponent;

       getline(setupFile,currentLine,'\n');
       string TBF_filename = currentLine;
       getline(setupFile,currentLine,'\n');
       manualComponent = currentLine;
       getline(setupFile,currentLine,'\n');
       int boolManualTTF = atoi(currentLine.c_str());
```

```
	getline(setupFile,currentLine,'\n');
	double manualTTF = atof(currentLine.c_str());
	getline(setupFile,currentLine,'\n');
	int boolManualRepair = atoi(currentLine.c_str());
	getline(setupFile,currentLine,'\n');
	double manualTTR = atof(currentLine.c_str());
	getline(setupFile,currentLine,'\n');
	int boolManualReplace = atoi(currentLine.c_str());
	getline(setupFile,currentLine,'\n');
	double manualReplace = atof(currentLine.c_str());
	getline(setupFile,currentLine,'\n');
	string Repair_filename = currentLine;
	getline(setupFile,currentLine,'\n');
	string replaceFile = currentLine;
	getline(setupFile,currentLine,'\n');
	string network_filename = currentLine;
	getline(setupFile,currentLine,'\n');
	string ageFile = currentLine;
	getline(setupFile,currentLine,'\n');
	string thermalFile = currentLine;
	getline(setupFile,currentLine,'\n');
	string outputDirectory = currentLine;
	getline(setupFile,currentLine,'\n');
	string maintainFile = currentLine;

	while(getline(setupFile, currentLine, '\n'))
	{
		setupFile >> componentID >> age;

		currentManual.age = age;
		currentManual.component = componentID;

		manual_age.push_back(currentManual);
	}

	importThermalFile(thermalComponent, &thermalFile);

	setupFile.close();
	setupFile.clear();

	int partsFailed = 0;
	int fileChoice = 1;				// User to choice to end or
continue the program
	int repairReplace = 0;				// Choice of repair or
replace
	int replication = 0;			// current replication number
	int numRows = -1;				// number of rows
	int numCols=0;					// number of columns
(components)
	int whatToFill;					// determines which
table to fill: TBF, TTR, or replace
	int numberOfReplications = 100;			// total number of
replications (user specified)
	int componentSave;
	int manualChoice = 0;
	int compon = 0;
	int R;
	int failedPosition = 0;
	int componentsFailed = 0;

	double time = 0;				// holds the current time
	double timeInc = 0.1;					// increments the 
times
	double mean;
	double sigma;
	double logMean;
	double logSigma;
	double beta;					// shape parameter
	double eta=0;					// scale parameter
	double runLength = 50;			// total runtime of the sequence
	double minMTTFvalue;
	double expoMean;
	double expoSigma;
	double logNormMean;
	double logNormSigma;
	double wiebullMean;
	double wiebullSigma;
	double lowestFailedTBF = DBL_MAX;
	double numberFailed = 0;

	bool generateNumber = true;

	string mttr;
	string replace;
	string Line;

	fstream TBF_FILE;
	fstream TBF_fileRead;

	Component * pointerToResCapLife;				// 
containes values for subtracting and failing/fixing components
	Component * pointerToResCapCalculation;			// 
contains calculations to find a new value for a component
	Component * pointerToResCapInput;

	statisticsVectors * pointerToResCapStats;			// contains 
all data (sample and random) obtained within a single replication
	statisticsVectors * pointerToResCapStatsIntegrated;		// contains 
all data (sample and random) obtained throughout ALL replications

	vector <lifeRanges> finalLife;
	lifeRanges finalFails;

	vector <failing> failTableIntegrated;
	systemReliability reliable;
	vector <systemReliability> componentReliability;
	vector <systemReliability> systemReliability;
	vector <double> lowestTTF;
	vector <double> lowestPosition;
	vector <double> systemFailureRate;
	vector <parallelFail> parallelFailing;
	vector <statisticsVectors> exportGeneratedMTTR;

	lifeRanges systemMinTTF;
	lifeRanges systemMaxTTF;
	failing failTable;
	vector <failing> outputFailTable;

	componentNetworks * networks;

	// random seed generator
	srand(GetTickCount());

	// jump to new replication
	newReplication:

	fileChoice = 1;
	numRows = -1;

		// open TBF file
		TBF_FILE.open(TBF_filename.c_str());

			// if file is available to open
			if(TBF_FILE.is_open())
			{
				// get the first line in the file
				while (getline(TBF_FILE, Line, '\n'))
				{
					if(Line.length() > 0)
					{
						TBF_FILE.close();
						TBF_FILE.clear();
						break;
					}
				}
			}

			// call numberColumns function
			numCols = numberColumns(Line);

			// array of structures containing the values of statistical
calculations
			pointerToResCapStats = new statisticsVectors[numCols];

			// if on the initial replication
			if(replication == 0)
			{
				pointerToResCapStatsIntegrated = new
statisticsVectors[numCols];
				pointerToResCapInput = new Component[numCols];
			}

			// reopen TBF file
			TBF_fileRead.open(TBF_filename.c_str());

			// if file is available to open
			if(TBF_fileRead.is_open())
			{
				// find the number of rows in the input files
				while (getline(TBF_fileRead, Line,'\n'))
				{
					numRows++;
				}
```

```
                            // resize the vectors in statisticalVectors to the
number of rows in the file
                            for(int col = 0; col < numCols; col++)
                            {

       pointerToResCapStats[col].timeBetweenFail.resize(numRows);

       pointerToResCapStats[col].timeBetweenFailCDF.resize(numRows);

       pointerToResCapStats[col].repair.resize(numRows);

       pointerToResCapStats[col].repairCDF.resize(numRows);

       pointerToResCapStats[col].replace.resize(numRows);

       pointerToResCapStats[col].replaceCDF.resize(numRows);

       pointerToResCapStats[col].TBFrank.resize(numRows);

       pointerToResCapStats[col].repairRank.resize(numRows);

       pointerToResCapStats[col].replaceRank.resize(numRows);

                                   if(replication == 0)
                                   {

       pointerToResCapStatsIntegrated[col].TBFrank.resize(numRows);

       pointerToResCapStatsIntegrated[col].repairRank.resize(numRows);

       pointerToResCapStatsIntegrated[col].replaceRank.resize(numRows);

       pointerToResCapStatsIntegrated[col].timeBetweenFail.resize(numRows);

       pointerToResCapStatsIntegrated[col].repair.resize(numRows);

       pointerToResCapStatsIntegrated[col].replace.resize(numRows);

       pointerToResCapInput[col].inputMTTF.resize(numRows);

       pointerToResCapInput[col].inputMTTR.resize(numRows);

       pointerToResCapInput[col].inputMTTReplace.resize(numRows);
                                   }
                            }

                            //close file
                            TBF_fileRead.close();

                            // set fileChoice to 2 as to prevent re-looping
                            fileChoice = 2;
                     }

                     // clear contents if any are in
                     TBF_fileRead.clear();

                     // fill the TBF vector
                     whatToFill = 0;
```

```
                    importFREPLREPA_files(pointerToResCapStats, numRows, numCols,
&TBF_filename, whatToFill);

       // enter TTR file and fill repair vector
       whatToFill = 1;

       importFREPLREPA_files(pointerToResCapStats, numRows, numCols,
&Repair_filename, whatToFill);

       // enter Replace file and fill replace vector
       whatToFill = 2;

       importFREPLREPA_files(pointerToResCapStats, numRows, numCols, &replaceFile,
whatToFill);

       if(replication == 0)
       {
              networks = new componentNetworks[numCols];
              failTable.chip.resize(numCols);
              failTable.chipConnection.resize(numCols);
              failTable.parallel.resize(numCols);
              exportGeneratedMTTR.resize(numCols);
       }

       if(replication == 0)
       {
              importNetworksFile(networks, &network_filename, numCols);
       }

       if(replication == 0)
       {
              for(int col = 0; col < numCols; col++)
              {
                     for(int col1 = 0; col1 < numCols; col1++)
                     {
                            if(networks[col].extraNets.size() > 0 &&
                                   networks[col1].extraNets.size() == 0)
                            {
                                   for(unsigned int net = 0; net <
networks[col].allNets.size(); net++)
                                   {
                                          for(unsigned int net1 = 0; net1 <
networks[col1].allNets.size(); net1++)
                                          {
                                                 if(networks[col].allNets[net] ==
networks[col1].allNets[net1])
                                                 {

       failTable.chipConnection[col1] = true;
                                                 }
                                          }
                                   }
                            }
                     }
              }
       }

       for(int comp = 0; comp < numCols; comp++)
```

```
	{
		for(int comp1 = comp; comp1 < numCols; comp1++)
		{
			if((networks[comp].extraNets.size() == 0 &&
				networks[comp1].extraNets.size() == 0) &&
				failTable.parallel[comp] != true &&
				comp != comp1 &&
				(networks[comp].networkIN == networks[comp1].networkIN
&&
				networks[comp].networkOUT ==
networks[comp1].networkOUT))
			{
				failTable.parallelComponent.push_back(comp1);

				pointerToResCapStatsIntegrated[comp1].parallelComponent
= true;
				pointerToResCapStats[comp1].parallelComponent = true;

				failTable.parallel[comp] = true;
			}
		}
	}

	for(int comp = 0; comp < numCols; comp++)
	{
		if(pointerToResCapStats[comp].parallelComponent != true)
		{
			pointerToResCapStats[comp].parallelComponent = false;
		}
	}

	if(replication == 0)
	{
		for(int comp = 0; comp < numCols; comp++)
		{
			if(failTable.parallel[comp] != true)
			{
				failTable.parallel[comp] = false;
			}

			if(pointerToResCapStatsIntegrated[comp].parallelComponent !=
true)
			{
				pointerToResCapStatsIntegrated[comp].parallelComponent
= false;
			}
		}
	}

	// set component numbers for statistical vectors (must be corresponding
column
	for(int col = 0; col < numCols; col++)
	{
		pointerToResCapStats[col].componentNumber = col+1;
		pointerToResCapStats[col].timesFailed = 0;
		pointerToResCapStats[col].partID = networks[col].partID;
		pointerToResCapStats[col].compFailProbTotal = 0;
```

```
            if(replication == 0)
            {
                  pointerToResCapStatsIntegrated[col].MTTF_total = 0;
                  pointerToResCapStatsIntegrated[col].TTR_total = 0;
                  pointerToResCapStatsIntegrated[col].replaceTotal = 0;
                  pointerToResCapStatsIntegrated[col].partID = 
networks[col].partID;
                  exportGeneratedMTTR[col].partID = networks[col].partID;

                  if(col == manualComponentSave)
                  {
                        if(boolManualRepair == 1)
                        {
                              pointerToResCapStatsIntegrated[col].TTR_total = 
manualTTR;
                        }

                        if(boolManualReplace == 1)
                        {
                              pointerToResCapStatsIntegrated[col].replaceTotal 
= manualReplace;
                        }
                  }

                  if(networks[col].extraNets.size() > 0)
                  {
                        failTable.chip[col] = true;
                  }

                  else
                  {
                        failTable.chip[col] = false;
                  }
            }
      }

      for(unsigned int comp = 0; comp < thermalComponent.size(); comp++)
      {
            for(int col = 0; col < numCols; col++)
            {
                  if(pointerToResCapStats[col].partID == 
thermalComponent[comp].partID)
                  {
                        for(int row = 0; row < numRows; row++)
                        {
                              pointerToResCapStats[col].timeBetweenFail[row] = 
pointerToResCapStats[col].timeBetweenFail[row] / thermalComponent[comp].epsillon;
                        }
                  }
            }
      }

      importAgeFile(pointerToResCapStats, numCols, &ageFile);

      if(boolManualTTF == 1 ||
            boolManualRepair == 1 ||
            boolManualReplace == 1)
      {
```

```
                for(int comp = 0; comp < numCols; comp++)
                {
                        if(pointerToResCapStatsIntegrated[comp].partID ==
manualComponent)
                        {
                                manualComponentSave = comp;
                        }
                }

                if(boolManualTTF == 1)
                {
                        pointerToResCapStats[manualComponentSave].MTTF = manualTTF;
                }
        }

        /*for(unsigned int comp = 0; comp < thermalComponent.size(); comp++)
        {
                for(int col = 0; col < numCols; col++)
                {
                        if(thermalComponent[comp].partID ==
pointerToResCapStats[col].partID)
                        {
                                pointerToResCapStats[comp].MTTF =
thermalComponent[comp].MTTF;
                                thermalComponent[col].componentNumber = comp;
                        }
                }
        }*/

        // sort the Component structure in ascending order of fixed input file
values
        for(int i=0; i<numCols; i++)
        {
                std::sort(pointerToResCapStats[i].timeBetweenFail.rbegin(),
pointerToResCapStats[i].timeBetweenFail.rend(), std::greater<double>());
                std::sort(pointerToResCapStats[i].repair.rbegin(),
pointerToResCapStats[i].repair.rend(), std::greater<double>());
                std::sort(pointerToResCapStats[i].replace.rbegin(),
pointerToResCapStats[i].replace.rend(), std::greater<double>());
        }

        // if the program is on the initial replication; save the values to
pointerToResCapStatsIntegrated which holds all values from all replicationss
        if(replication == 0)
        {
                for(int q = 0; q < numRows; q++)
                {
                        for(int j = 0; j < numCols; j++)
                        {
                                pointerToResCapStatsIntegrated[j].timeBetweenFail[q] =
pointerToResCapStats[j].timeBetweenFail[q];
                                pointerToResCapStatsIntegrated[j].replace[q] =
pointerToResCapStats[j].replace[q];
                                pointerToResCapStatsIntegrated[j].repair[q] =
pointerToResCapStats[j].repair[q];
                        }
                }
        }
```

```
	// if on the initial replication give the integrated vector a component 
number and set the number of times value to 0
	if(replication == 0)
	{
		for(int col = 0; col < numCols; col++)
		{
			pointerToResCapStatsIntegrated[col].age = 
pointerToResCapStats[col].age;
			pointerToResCapStatsIntegrated[col].componentNumber = 
pointerToResCapStats[col].componentNumber;
			pointerToResCapStatsIntegrated[col].timesFailed = 0;
			pointerToResCapStatsIntegrated[col].compFailProbTotal = 0;
			pointerToResCapStatsIntegrated[col].distributionFound = false;

			for(int comp = 0; comp < numRows; comp++)
			{
				pointerToResCapInput[col].inputMTTF[comp] = 
pointerToResCapStats[col].timeBetweenFail[comp];
				pointerToResCapInput[col].inputMTTR[comp] = 
pointerToResCapStats[col].repair[comp];
				pointerToResCapInput[col].inputMTTReplace[comp] = 
pointerToResCapStats[col].replace[comp];
			}
		}
	}

	for(int comp = 0; comp < numCols; comp++)
	{
		for(unsigned int comp1 = 0; comp1 < manual_age.size(); comp1++)
		{
			if(pointerToResCapStats[comp].partID == 
manual_age[comp1].component)
			{
				pointerToResCapStats[comp].age = manual_age[comp1].age;

				if(replication == 0)
				{
					pointerToResCapStatsIntegrated[comp].age = 
manual_age[comp1].age;
				}
			}
		}
	}

	// call the CDF_calc function and calculate the CDF of each component 
vector type
	for(int CDF_TYPE = 0; CDF_TYPE < 3; CDF_TYPE++)
	{
		CDF_calc(pointerToResCapStats, pointerToResCapStatsIntegrated, 
numRows, numCols, CDF_TYPE);
	}

	if(replication == 0)
	{
		for(int q = 0; q < numRows; q++)
		{
			for(int j = 0; j < numCols; j++)
```

```
                        {
                                pointerToResCapStatsIntegrated[j].TBFrank[q] =
pointerToResCapStats[j].TBFrank[q];
                                pointerToResCapStatsIntegrated[j].repairRank[q] =
pointerToResCapStats[j].repairRank[q];
                                pointerToResCapStatsIntegrated[j].replaceRank[q] =
pointerToResCapStats[j].replaceRank[q];
                        }
                }
        }

        // create array for lifetime sequence
        pointerToResCapLife = new Component[numCols];

        // initialize the structure elements in the pointerToResCapLife array
        for(int col = 0; col < numCols; col++)
        {
                // values to be used and implented in the program
                pointerToResCapLife[col].componentNumber = col+1;
                pointerToResCapLife[col].FAIL = false;
                pointerToResCapLife[col].subtractTBF = false;
                pointerToResCapLife[col].subtractTTR = false;
                pointerToResCapLife[col].subtractReplace = false;
                pointerToResCapLife[col].printRepair = false;
                pointerToResCapLife[col].printReplace = false;
                pointerToResCapLife[col].repairing = false;
                pointerToResCapLife[col].replacement = false;
                pointerToResCapLife[col].polyNum = false;
                pointerToResCapLife[col].cdfed = false;
        }

        // component position in statisticalVectors;
        for(int q = 0; q < numCols; q++)
        {
                // randomNumberType 0 = TTF, randomNumberType 1 = TTR,
randomNumberType 2 = replace
                for(int randomNumberType = 0; randomNumberType < 3;
randomNumberType++)
                {
                        // array used for the calculations to generate a new TBF, TTR,
or Replace
                        pointerToResCapCalculation = new Component[numRows];

                        // assign component numbers for use in the new variable
calculations
                        for(int s = 0; s < numRows; s++)
                        {
                                pointerToResCapCalculation[s].componentNumber =
pointerToResCapStats[q].componentNumber;
                        }

                        // if calculation is for tbf set the CDF and TBF calculator to
the tBf values
                        if(randomNumberType == 0)
                        {
                                for(int s = 0; s < numRows; s++)
                                {
```

```
                                pointerToResCapCalculation[s].value = 
pointerToResCapStats[q].timeBetweenFail[s];
                                pointerToResCapCalculation[s].CDF = 
pointerToResCapStats[q].timeBetweenFailCDF[s];
                                pointerToResCapCalculation[s].rank = 
pointerToResCapStats[q].TBFrank[s];
                        }
                }

                // if calculation is for ttr set the CDF and TTR calculator to 
the ttr values
                else if(randomNumberType == 1)
                {
                        for(int s = 0; s < numRows; s++)
                        {
                                pointerToResCapCalculation[s].value = 
pointerToResCapStats[q].repair[s];
                                pointerToResCapCalculation[s].CDF = 
pointerToResCapStats[q].repairCDF[s];
                                pointerToResCapCalculation[s].rank = 
pointerToResCapStats[q].repairRank[s];
                        }
                }

                // if calculation is for replace set the CDF and Replace 
calculator to the replace values
                else if(randomNumberType == 2)
                {
                        for(int s = 0; s < numRows; s++)
                        {
                                pointerToResCapCalculation[s].value = 
pointerToResCapStats[q].replace[s];
                                pointerToResCapCalculation[s].CDF = 
pointerToResCapStats[q].replaceCDF[s];
                                pointerToResCapCalculation[s].rank = 
pointerToResCapStats[q].replaceRank[s];
                        }
                }

                // find mean of all MTBF
                mean = MEAN(pointerToResCapCalculation, numRows);

                // find standard deviation of all values
                sigma = STDDEV(pointerToResCapCalculation, numRows, mean);

                // find the log mean
                logMean = log_mean(pointerToResCapCalculation, numRows);

                // find the standard deviation of the logs
                logSigma = log_sigma(pointerToResCapCalculation, numRows, 
logMean);

                // solve for beta
                beta = betaCalc(pointerToResCapCalculation, numRows, mean, 
sigma);

                // solve for eta
```

```
                    eta = etaCalc(pointerToResCapCalculation, numRows, beta, 
mean);

                    // find the area below the normal curve (normalProbability) 
through interpolation of an input z table
                    normal_probability(pointerToResCapCalculation, numRows, mean, 
sigma);

                    expoMean = exponentialMean(eta);
                    expoSigma = exponentialSigma(eta);

                    // solve probabilities for exponential
                    exponential_probability(pointerToResCapCalculation, numRows, 
expoMean);

                    logNormMean = lognormalFinalMean(logMean, logSigma);
                    logNormSigma = lognormalFinalSigma(logMean, logSigma);

                    // solve for the lognormal probability (Use log mean and log 
standard deviation)
                    lognormal_probability(pointerToResCapCalculation, numRows, 
numCols, logNormMean, logNormSigma);

                    wiebullMean = weibullMean(beta, eta);
                    wiebullSigma = weibullSigma(beta, eta);

                    // solve the weibull probability
                    weibull_probability(pointerToResCapCalculation, numRows, beta, 
eta);

                    if(randomNumberType == 0)
                    {
                           //std::cout << logMean << "\t" << logSigma << "\t" << 
logNormMean << "\t" << logNormSigma << std::endl;
                           pointerToResCapStatsIntegrated[q].logMean = 
logNormMean;
                           pointerToResCapStatsIntegrated[q].logSigma = 
logNormSigma;
                           pointerToResCapStatsIntegrated[q].MTTF = mean;
                           pointerToResCapStatsIntegrated[q].sigma = sigma;
                           pointerToResCapStatsIntegrated[q].expoMean = expoMean;
                           pointerToResCapStatsIntegrated[q].expoSigma = 
expoSigma;
                           pointerToResCapStatsIntegrated[q].weibullMean = 
wiebullMean;
                           pointerToResCapStatsIntegrated[q].weibullSigma = 
wiebullSigma;
                    }

                    // if it is supposed to be generating a TTF
                    if(randomNumberType == 0)
                    {
                           //for(unsigned int comp = 0; comp < 
thermalComponent.size(); comp++)
                           //{
                           //     if(pointerToResCapLife[q].componentNumber == 
thermalComponent[comp].componentNumber)
                           //     {
                            //          // create a new time to fail
                            //          pointerToResCapLife[q].TBF = 
find_distances(pointerToResCapCalculation, pointerToResCapStats, numRows, mean, 
sigma, expoMean, beta, eta, logMean, logSigma, logNormSigma, logNormMean, q, 
randomNumberType, generateNumber)/thermalComponent[comp].epsillon;
                            //
                            //          complete = true;
                            //      }
                            //}

                            // create a new time to fail
                            pointerToResCapLife[q].TBF = 
find_distances(pointerToResCapCalculation, pointerToResCapStats, 
pointerToResCapStatsIntegrated, numRows, mean, sigma, expoMean, beta, eta, 
logMean, logSigma, logNormSigma, logNormMean, q, randomNumberType, 
generateNumber);

                            // include this time to fail in both statistical 
vectors

       pointerToResCapStats[q].timeBetweenFail.push_back(pointerToResCapLife[q].TB
F);

                            // adds the new element into the end of the integrated 
array (same concept as push_back)

       //pointerToResCapStatsIntegrated[q].timeBetweenFail.push_back(pointerToResC
apLife[q].TBF);


       std::sort(pointerToResCapStats[q].timeBetweenFail.rbegin(), 
pointerToResCapStats[q].timeBetweenFail.rend(), std::greater<double>());

                            for(unsigned int rawr = 0; rawr < 
pointerToResCapStats[q].timeBetweenFail.size(); rawr++)
                            {
                                   if(pointerToResCapLife[q].TBF == 
pointerToResCapStats[q].timeBetweenFail[rawr])
                                   {

       pointerToResCapStats[q].TBFrank.push_back(rawr + 1);
                                   }
                            }

                            std::sort(pointerToResCapStats[q].TBFrank.rbegin(), 
pointerToResCapStats[q].TBFrank.rend(), std::greater<double>());

                            for(unsigned int rawr = 0; rawr < 
pointerToResCapStats[q].timeBetweenFail.size(); rawr++)
                            {
                                   if(pointerToResCapLife[q].TBF < 
pointerToResCapStats[q].timeBetweenFail[rawr])
                                   {
                                          pointerToResCapStats[q].TBFrank[rawr] = 
pointerToResCapStats[q].TBFrank[rawr]+1;
                                   }
                            }
                            // save the best fit distribution for that component
                            pointerToResCapLife[q].distributionTypeHoldTBF =
pointerToResCapCalculation[0].distributionTypeHoldTBF;

                            if(replication == 0)
                            {
                                   pointerToResCapInput[q].distributionTypeHoldTBF
= pointerToResCapCalculation[0].distributionTypeHoldTBF;


       if(pointerToResCapInput[q].distributionTypeHoldTBF == 1)
                                   {
                                          // find mean
                                          pointerToResCapInput[q].MTTF =
MEAN(pointerToResCapCalculation, numRows);
                                          // find standard deviation
                                          pointerToResCapInput[q].sigma =
STDDEV(pointerToResCapCalculation, numRows, pointerToResCapInput[q].MTTF);
                                   }

                                   else
if(pointerToResCapInput[q].distributionTypeHoldTBF == 2)
                                   {
                                          // find mean
                                          pointerToResCapInput[q].MTTF = expoMean;
                                          // find standard deviation
                                          pointerToResCapInput[q].sigma =
expoSigma;
                                   }

                                   else
if(pointerToResCapInput[q].distributionTypeHoldTBF == 3)
                                   {
                                          // find mean
                                          pointerToResCapInput[q].MTTF =
logNormMean;
                                          pointerToResCapInput[q].logMTTF =
logMean;
                                          pointerToResCapInput[q].logSIGMA =
logSigma;
                                          // find standard deviation
                                          pointerToResCapInput[q].sigma =
logNormSigma;
                                   }

                                   else
                                   {
                                          pointerToResCapInput[q].beta = beta;
                                          pointerToResCapInput[q].eta = eta;
                                          // find mean
                                          pointerToResCapInput[q].MTTF =
wiebullMean;
                                          // find standard deviation
                                          pointerToResCapInput[q].sigma =
wiebullSigma;
                                   }
                            }
                            if(replication == 0)
                            {
                                   pointerToResCapStatsIntegrated[q].distribution = 
pointerToResCapCalculation[0].distributionTypeHoldTBF;


       pointerToResCapStatsIntegrated[q].distributionFound = true;
                            }
                     }

                     // if the variable thats supposed to be generated is repair
                     else if(randomNumberType == 1)
                     {
                            // create a new time to repair
                            pointerToResCapLife[q].TTR = 
find_distances(pointerToResCapCalculation, pointerToResCapStats, 
pointerToResCapStatsIntegrated, numRows, mean, sigma, expoMean, beta, eta, 
logMean, logSigma, logNormSigma, logNormMean, q, randomNumberType, 
generateNumber);

                            // include this time to repair in both statistical 
vectors

       pointerToResCapStats[q].repair.push_back(pointerToResCapLife[q].TTR);


       exportGeneratedMTTR[q].repair.push_back(pointerToResCapLife[q].TTR);
                            // resize the integrated repair vector and add the new 
element in this extra spot (similar to push back

       //pointerToResCapStatsIntegrated[q].repair.push_back(pointerToResCapLife[q]
.TTR);

                            std::sort(pointerToResCapStats[q].repair.rbegin(), 
pointerToResCapStats[q].repair.rend(), std::greater<double>());

                            for(unsigned int rawr = 0; rawr < 
pointerToResCapStats[q].repair.size(); rawr++)
                            {
                                   if(pointerToResCapLife[q].TTR == 
pointerToResCapStats[q].repair[rawr])
                                   {

       pointerToResCapStats[q].repairRank.push_back(rawr + 1);
                                   }
                            }

                            std::sort(pointerToResCapStats[q].repairRank.rbegin(), 
pointerToResCapStats[q].repairRank.rend(), std::greater<double>());

                            for(unsigned int rawr = 0; rawr < 
pointerToResCapStats[q].repair.size(); rawr++)
                            {
                                   if(pointerToResCapLife[q].TTR < 
pointerToResCapStats[q].repair[rawr])
                                   {
                                          pointerToResCapStats[q].repairRank[rawr] 
= pointerToResCapStats[q].repairRank[rawr]+1;
```

```
                                        }
                                }

                                // resize the CDF vector to the same size as the repair
vector

        pointerToResCapStats[q].repairCDF.resize(pointerToResCapStats[q].repair.siz
e());

                                // save the best fit distribution of that component for
repair
                                pointerToResCapLife[q].distributionTypeHoldRepair =
pointerToResCapCalculation[0].distributionTypeHoldRepair;

                                if(replication == 0)
                                {

        pointerToResCapInput[q].distributionTypeHoldRepair =
pointerToResCapCalculation[0].distributionTypeHoldRepair;


        if(pointerToResCapInput[q].distributionTypeHoldRepair == 1)
                                        {
                                                // find mean
                                                pointerToResCapInput[q].inRepair =
MEAN(pointerToResCapCalculation, numRows);
                                                pointerToResCapInput[q].repairSigma =
STDDEV(pointerToResCapCalculation, numRows, pointerToResCapInput[q].inRepair);
                                        }

                                        else
if(pointerToResCapInput[q].distributionTypeHoldRepair == 2)
                                        {
                                                // find mean
                                                pointerToResCapInput[q].inRepair =
expoMean;
                                                pointerToResCapInput[q].repairSigma =
expoSigma;
                                        }

                                        else
if(pointerToResCapInput[q].distributionTypeHoldRepair == 3)
                                        {
                                                // find mean
                                                pointerToResCapInput[q].inRepair =
logNormMean;
                                                pointerToResCapInput[q].repairSigma =
logNormSigma;
                                        }

                                        else
                                        {
                                                // find mean
                                                pointerToResCapInput[q].inRepair =
wiebullMean;
                                                pointerToResCapInput[q].repairSigma =
wiebullSigma;
```

```
                                    pointerToResCapInput[q].repairBeta = 
beta;
                                    pointerToResCapInput[q].repairEta = eta;
                              }
                        }
                  }

                  // if the variable thats supposed to be generated is the 
replace
                  else if(randomNumberType == 2)
                  {
                        // create a new time to replace
                        pointerToResCapLife[q].replace = 
find_distances(pointerToResCapCalculation, pointerToResCapStats, 
pointerToResCapStatsIntegrated, numRows, mean, sigma, expoMean, beta, eta, 
logMean, logSigma, logNormSigma, logNormMean, q, randomNumberType, 
generateNumber);

                        // include this time to replace in both statistical 
vectors

      pointerToResCapStats[q].replace.push_back(pointerToResCapLife[q].replace);

                        // resize the integrated array vector and add the new 
value to that spot

      //pointerToResCapStatsIntegrated[q].replace.push_back(pointerToResCapLife[q
].replace);

                        std::sort(pointerToResCapStats[q].replace.rbegin(), 
pointerToResCapStats[q].replace.rend(), std::greater<double>());

                        for(unsigned int rawr = 0; rawr < 
pointerToResCapStats[q].replace.size(); rawr++)
                        {
                              if(pointerToResCapLife[q].replace == 
pointerToResCapStats[q].replace[rawr])
                              {

      pointerToResCapStats[q].replaceRank.push_back(rawr + 1);
                              }
                        }

                        std::sort(pointerToResCapStats[q].replace.rbegin(), 
pointerToResCapStats[q].replace.rend(), std::greater<double>());

                        for(unsigned int rawr = 0; rawr < 
pointerToResCapStats[q].replace.size(); rawr++)
                        {
                              if(pointerToResCapLife[q].replace < 
pointerToResCapStats[q].replace[rawr])
                              {
                                    pointerToResCapStats[q].replaceRank[rawr] 
= pointerToResCapStats[q].replaceRank[rawr]+1;
                              }
                        }
```

```
                            // resize the CDF vector to the same size as the value
vector

       pointerToResCapStats[q].replaceCDF.resize(pointerToResCapStats[q].replace.s
ize());

                            // print the new replace value
                            //std::cout << "Component " <<
pointerToResCapLife[q].componentNumber << "\tNew REPLACE\t" <<
pointerToResCapLife[q].replace << std::endl;

                            // save hte best fit distribution for the replace of
that component
                            pointerToResCapLife[q].distributionTypeHoldReplace =
pointerToResCapCalculation[0].distributionTypeHoldReplace;

                            if(replication == 0)
                            {

       pointerToResCapInput[q].distributionTypeHoldReplace =
pointerToResCapCalculation[0].distributionTypeHoldReplace;


       if(pointerToResCapInput[q].distributionTypeHoldReplace == 1)
                                   {
                                          // find mean
                                          pointerToResCapInput[q].inReplace =
MEAN(pointerToResCapCalculation, numRows);
                                          pointerToResCapInput[q].replaceSigma =
STDDEV(pointerToResCapCalculation, numRows, pointerToResCapInput[q].inReplace);
                                   }

                                   else
if(pointerToResCapInput[q].distributionTypeHoldReplace == 2)
                                   {
                                          // find mean
                                          pointerToResCapInput[q].inReplace =
expoMean;
                                          pointerToResCapInput[q].replaceSigma =
expoSigma;
                                   }

                                   else
if(pointerToResCapInput[q].distributionTypeHoldReplace == 3)
                                   {
                                          // find mean
                                          pointerToResCapInput[q].inReplace =
logNormMean;
                                          pointerToResCapInput[q].replaceSigma =
logNormSigma;
                                   }

                                   else
                                   {
                                          // find mean
                                          pointerToResCapInput[q].inReplace =
wiebullMean;
```

```
                                          pointerToResCapInput[q].replaceSigma =
wiebullSigma;
                                          pointerToResCapInput[q].replaceBeta =
beta;
                                          pointerToResCapInput[q].replaceEta = eta;
                                   }
                            }
                     }

                     // reset the values used
                     mean = 0;
                     sigma = 0;
                     logMean = 0;
                     logSigma = 0;
                     eta = 0;
                     beta = 0;

                     // delete the pointerToResCapCalculation structure array due
to its continuous reuse
                     delete []pointerToResCapCalculation;
              }
       }

       bool resizeOnce = false;

       if(replication == 0)
       {
              //initialMeans(pointerToResCapStatsIntegrated, pointerToResCapInput,
numCols);

              for(unsigned int comp = 0; comp < failTable.parallel.size(); comp++)
              {
                     for(unsigned int comp1 = comp; comp1 <
failTable.parallel.size(); comp1++)
                     {
                            if((failTable.parallel[comp] &&
                                   failTable.parallel[comp1]) &&
                                   comp != comp1 &&
                                   (networks[comp].networkIN ==
networks[comp1].networkIN &&
                                   networks[comp].networkOUT ==
networks[comp1].networkOUT))
                            {
                                   if(!resizeOnce)
                                   {

       parallelFailing.resize(parallelFailing.size() + 1);
                                          parallelFailing[parallelFailing.size() -
1].initialComponent = pointerToResCapStatsIntegrated[comp].partID;

                                          resizeOnce = true;
                                   }

                                   parallelFailing[parallelFailing.size() -
1].componentsWithin.push_back(pointerToResCapStatsIntegrated[comp1].partID);

                                   if(pointerToResCapLife[comp].TBF <
pointerToResCapLife[comp1].TBF)
```

```
                    {
                        pointerToResCapLife[comp].TBF =
pointerToResCapLife[comp1].TBF;
                    }

                    if(pointerToResCapLife[comp1].replace>
pointerToResCapLife[comp1].TTR &&
                        pointerToResCapLife[comp1].replace >
pointerToResCapLife[comp].TTR &&
                        pointerToResCapLife[comp1].replace >
pointerToResCapLife[comp].replace)
                    {
                        pointerToResCapLife[comp].replace =
pointerToResCapLife[comp1].replace;
                    }

                    else if(pointerToResCapLife[comp1].TTR >
pointerToResCapLife[comp1].replace &&
                        pointerToResCapLife[comp1].TTR >
pointerToResCapLife[comp].TTR &&
                        pointerToResCapLife[comp1].TTR >
pointerToResCapLife[comp].replace)
                    {
                        pointerToResCapLife[comp].TTR =
pointerToResCapLife[comp1].TTR;
                    }


    failTable.parallelComponent[failTable.parallelComponent.size()-1]++;

    failTable.parallelWithin.push_back(pointerToResCapStatsIntegrated[comp1].pa
rtID);
                }
            }
        }

        resizeOnce = false;
    }

    if(replication == 0)
    {
        for(unsigned int comp = 0; comp < parallelFailing.size(); comp++)
        {
            for(unsigned int within = 0; within <
parallelFailing[comp].componentsWithin.size(); within++)
            {
                for(unsigned int within1 = 0; within1 <
parallelFailing[comp].componentsWithin.size(); within1++)
                {
                    if(within != within1 &&

    parallelFailing[comp].componentsWithin[within] ==
parallelFailing[comp].componentsWithin[within1])
                    {

    parallelFailing[comp].componentsWithin.erase(parallelFailing[comp].componen
tsWithin.begin()+within1);
                    }
```

```
                                comp = 0;
                        }
                }
        }
    }

    // used to determine whether the program should be dealing with TBF, TTR,
or replace currrently
    int randomNumberType;

    /*for(unsigned int comp = 0; comp < thermalComponent.size(); comp++)
    {
        for(int col = 0; col < numCols; col++)
        {
            if(pointerToResCapLife[col].componentNumber ==
thermalComponent[comp].componentNumber)
            {
                pointerToResCapLife[col].TBF =
thermalComponent[col].MTTF;
            }
        }
    }*/

    if(boolManualTTF == 1)
    {
        pointerToResCapLife[manualComponentSave].TBF = manualTTF;
    }

    if(boolManualRepair == 1)
    {
        pointerToResCapLife[manualComponentSave].TTR = manualTTR;

        if(boolManualReplace == 0)
        {
            pointerToResCapLife[manualComponentSave].replace = DBL_MAX;
        }
    }

    if(boolManualReplace == 1)
    {
        pointerToResCapLife[manualComponentSave].replace = manualReplace;

        if(boolManualRepair == 0)
        {
            pointerToResCapLife[manualComponentSave].TTR = DBL_MAX;
        }
    }

    minMaxTTF(pointerToResCapInput, pointerToResCapStatsIntegrated, numCols);

    for(int comp = 0; comp < numCols; comp++)
    {
        if(boolManualTTF == 0 ||
            (boolManualTTF == 1 &&
            comp != manualComponentSave))
        {
            if(pointerToResCapStatsIntegrated[comp].age != 0)
```

```
                    {
                            pointerToResCapLife[comp].TBF = 
pointerToResCapInput[comp].maxTTF;
                    }

                    else
                    {
                            findNewValue(pointerToResCapLife, 
pointerToResCapStatsIntegrated, numCols, pointerToResCapStats, 
componentNumberNewValue, 0, pointerToResCapStats[comp].timeBetweenFail.size(), 
false, thermalComponent, pointerToResCapInput);
                    }
            }

            pointerToResCapLife[comp].TBF -= 
pointerToResCapStatsIntegrated[comp].age;

            if(pointerToResCapLife[comp].TBF <=0)
            {
                    pointerToResCapStats[comp].timesFailed++;
                    pointerToResCapStatsIntegrated[comp].timesFailed++;

                    if(boolManualTTF == 0 ||
                            (boolManualTTF == 1 &&
                            comp != manualComponentSave))
                    {
                            randomNumberType = 0;

                            componentNumberNewValue = comp;

                            findNewValue(pointerToResCapLife, 
pointerToResCapStatsIntegrated, numCols, pointerToResCapStats, 
componentNumberNewValue, randomNumberType, 
pointerToResCapStats[componentNumberNewValue].repair.size(), generateNumber, 
thermalComponent, pointerToResCapInput);
                    }

                    else if(boolManualTTF == 1 &&
                            comp == manualComponentSave)
                    {
                            pointerToResCapLife[comp].TBF = manualTTF;
                    }
            }

            else
            {
                    double max = pointerToResCapLife[comp].TBF;
                    pointerToResCapLife[comp].TBF = ((double)rand() / 
RAND_MAX)*max;
            }
    }

    for(int comp = 0; comp < numCols; comp++)
    {
            pointerToResCapLife[comp].parallelComponent = 
pointerToResCapStatsIntegrated[comp].parallelComponent;
    }
    // set default time incrementor to 0.1 (can be changed)
```

```
        if(replication < numberOfReplications)
        {
                failFix:

                if(time > 200)
                {
                        failTable.fix.push_back(50);

                        goto exitLoop;
                }

                // if the part has failed
                while(pointerToResCapLife[failedPosition].FAIL)
                {
                        lowestFailedTBF = DBL_MAX;

                        time+=timeInc;

                        // the component has not decrmented in time and user chose to
repair the piece
                        if(pointerToResCapLife[failedPosition].repairing)
                        {
                                // decement time from mttr
                                pointerToResCapLife[failedPosition].TTR -= timeInc;
                        }

                        // the component has not yet had time decremented and the user
chose to replace the component
                        if(pointerToResCapLife[failedPosition].replacement)
                        {
                                // decrement time from replace time holder
                                pointerToResCapLife[failedPosition].replace -= timeInc;
                        }

                        // if the component has been fixed and it just failed
                        if(pointerToResCapLife[failedPosition].TTR <= 0 ||
                                pointerToResCapLife[failedPosition].replace <= 0)
                        {
                                // the component no longer fails
                                pointerToResCapLife[failedPosition].FAIL = false;

                                componentsFailed = 0;

                                // repair is decremented to zero and has not specified
so
                                if(pointerToResCapLife[failedPosition].TTR <= 0)
                                {
                                        if(pointerToResCapLife[failedPosition].TTR < 0)
                                        {
                                                time +=
pointerToResCapLife[failedPosition].TTR;
                                        }

                                        // choice is no longer to repair
                                        pointerToResCapLife[failedPosition].repairing =
false;

                                        if(boolManualRepair == 0)
```

```
                                    {
                                           // set randomNumberType to TTR
                                           randomNumberType = 1;

                                           // save array position
                                           componentNumberNewValue = failedPosition;

                                           // find a new TTR
                                           findNewValue(pointerToResCapLife,
pointerToResCapStatsIntegrated, numCols, pointerToResCapStats,
componentNumberNewValue, randomNumberType,
pointerToResCapStats[componentNumberNewValue].repair.size(), generateNumber,
thermalComponent, pointerToResCapInput);
                                    }

                                    else if(boolManualRepair == 1 &&
                                           failedPosition != manualComponentSave)
                                    {
                                           // set randomNumberType to TTR
                                           randomNumberType = 1;

                                           // save array position
                                           componentNumberNewValue = failedPosition;

                                           // find a new TTR
                                           findNewValue(pointerToResCapLife,
pointerToResCapStatsIntegrated, numCols, pointerToResCapStats,
componentNumberNewValue, randomNumberType,
pointerToResCapStats[componentNumberNewValue].repair.size(), generateNumber,
thermalComponent, pointerToResCapInput);
                                    }

                                    else if(boolManualRepair == 1 &&
                                           failedPosition == manualComponentSave)
                                    {
                                           pointerToResCapLife[failedPosition].TTR =
manualTTR;
                                    }
                                    //manual_TTF_TTR_REPLACE(pointerToResCapLife,
numCols, 2);
                             }

                             // if the replace has been decremented to 0
                             else if(pointerToResCapLife[failedPosition].replace <=
0)
                             {
                                    if(pointerToResCapLife[failedPosition].replace <
0)
                                    {
                                           time +=
pointerToResCapLife[failedPosition].replace;
                                    }

                                    // part is no longer being replaced
                                    pointerToResCapLife[failedPosition].replacement
= false;

                                    if(boolManualReplace == 0)
                                    {
                                            // set randomNumberType to Replace
                                            randomNumberType = 2;

                                            // save current array position
                                            componentNumberNewValue = failedPosition;

                                            // call functions to obtain a new Replace
                                            findNewValue(pointerToResCapLife,
pointerToResCapStatsIntegrated, numCols, pointerToResCapStats,
componentNumberNewValue, randomNumberType,
pointerToResCapStats[componentNumberNewValue].replace.size(), generateNumber,
thermalComponent, pointerToResCapInput);
                                    }

                                    else if(boolManualReplace == 1 &&
                                            failedPosition != manualComponentSave)
                                    {
                                            // set randomNumberType to Replace
                                            randomNumberType = 2;

                                            // save current array position
                                            componentNumberNewValue = failedPosition;

                                            // call functions to obtain a new Replace
                                            findNewValue(pointerToResCapLife,
pointerToResCapStatsIntegrated, numCols, pointerToResCapStats,
componentNumberNewValue, randomNumberType,
pointerToResCapStats[componentNumberNewValue].replace.size(), generateNumber,
thermalComponent, pointerToResCapInput);
                                    }

                                    else if(boolManualReplace == 1 &&
                                            failedPosition == manualComponentSave)
                                    {

        pointerToResCapLife[failedPosition].replace = manualReplace;
                                    }
                            }

                            componentsFailed = 0;

                            goto reEnterTime;
                    }
            }

            // time loop
            for(time = 0; time < runLength; time += timeInc)
            {
                    reEnterTime:

                    if(time > 50)
                    {
                            goto exitLoop;
                    }

                    // if a partID has not had the tbf decremented and it is still
working
```

```
                    if(componentsFailed == 0)
                    {
                           // search through all resistors
                           for(R = 0; R < numCols; R++)
                           {
                                  if(!pointerToResCapLife[R].parallelComponent)
                                  {
                                         // decrement each components time between
failures  by the time incrementer
                                         pointerToResCapLife[R].TBF -= timeInc;

                                         // when a part fails ( time between
failure is equal to 0 and it has not already failed
                                         if(pointerToResCapLife[R].TBF <= 0)
                                         {
                                                componentsFailed++;


       lowestTTF.push_back(pointerToResCapLife[R].TBF);

                                                // set the component to fail
                                                pointerToResCapLife[R].FAIL =
true;

                                                // if repairReplace is 1, user
chose to replace component, mark replacement as true
                                                if(pointerToResCapLife[R].replace
< pointerToResCapLife[R].TTR)
                                                {

       pointerToResCapLife[R].replacement = true;
                                                }

                                                // if repairReplace is 2, user
chose to repair component, mark repairing as true
                                                else
                                                {

       pointerToResCapLife[R].repairing = true;
                                                }


       //manual_TTF_TTR_REPLACE(pointerToResCapLife, numCols, 1);

                                                // increment total number of parts
that have failed, and print the total number
                                                partsFailed++;

                                                //std::cout << std::endl << "The
number of parts that have failed at time " << time << " are " << partsFailed <<
std::endl;
                                         }
                                  }
                           }
                    }

                    if(componentsFailed > 0)
                    {
```

```
                              lowestFailedTBF = pointerToResCapLife[0].TBF;

                              int save = 0;

                              for(int comp = 1; comp < numCols; comp++)
                              {
                                     if(pointerToResCapLife[comp].TBF <
lowestFailedTBF &&
                                            pointerToResCapLife[comp].FAIL)
                                     {
                                            lowestFailedTBF =
pointerToResCapLife[comp].TBF;
                                            save = comp;
                                     }
                              }

                              lowestFailedTBF += timeInc;

                              for(int comp = 0; comp < numCols; comp++)
                              {
                                     if(pointerToResCapLife[comp].FAIL &&
                                            pointerToResCapLife[comp].TBF < 0 &&
                                            save != comp)
                                     {
                                            pointerToResCapLife[comp].TBF -=
(lowestFailedTBF-timeInc);
                                            pointerToResCapLife[comp].FAIL = false;
                                     }

                                     else if(!pointerToResCapLife[comp].FAIL &&

       !pointerToResCapLife[comp].parallelComponent)
                                     {
                                            pointerToResCapLife[comp].TBF -=
(lowestFailedTBF-timeInc);
                                     }

                                     else if(pointerToResCapLife[comp].FAIL &&
                                            save == comp)
                                     {
                                            failedPosition = comp;

                                            if(boolManualTTF == 0)
                                            {
                                                   // save current array position
                                                   componentNumberNewValue = comp;

                                                   // set randomNumberType to TBF
                                                   randomNumberType = 0;
                                                   lowestPosition.push_back(comp);

                                                   // call functions to obtain a new
TBF

                                                   findNewValue(pointerToResCapLife,
pointerToResCapStatsIntegrated, numCols, pointerToResCapStats,
componentNumberNewValue, randomNumberType,
pointerToResCapStats[componentNumberNewValue].timeBetweenFail.size(),
generateNumber, thermalComponent, pointerToResCapInput);
                                            }

                                            else if(boolManualTTF == 1 &&
                                                   comp != manualComponentSave)
                                            {
                                                   // save current array position
                                                   componentNumberNewValue = comp;
                                                   lowestPosition.push_back(comp);

                                                   // set randomNumberType to TBF
                                                   randomNumberType = 0;

                                                   // call functions to obtain a new
TBF
                                                   findNewValue(pointerToResCapLife,
pointerToResCapStatsIntegrated, numCols, pointerToResCapStats,
componentNumberNewValue, randomNumberType,
pointerToResCapStats[componentNumberNewValue].timeBetweenFail.size(),
generateNumber, thermalComponent, pointerToResCapInput);
                                            }

                                            else if(boolManualTTF == 1 &&
                                                   comp == manualComponentSave)
                                            {
                                                   lowestPosition.push_back(comp);


       pointerToResCapStats[manualComponentSave].timesFailed++;

       pointerToResCapStatsIntegrated[manualComponentSave].timesFailed++;


       pointerToResCapLife[manualComponentSave].TBF = manualTTF;
                                            }
                                     }
                              }

                              time += lowestFailedTBF;
                       }

                       if(componentsFailed > 0)
                       {
                              componentsFailed = 0;
                              numberFailed++;
                              lowestTTF.clear();
                              lowestPosition.clear();

                              goto failFix;
                       }
                }
                // END TIME SEQUENCE

                exitLoop:

                // if the current replication is still less than the total number of
replications
```

```
			if(replication < numberOfReplications)
			{
				// increment replications
				replication++;

				failureRate(pointerToResCapStats,
pointerToResCapStatsIntegrated, pointerToResCapInput, numCols, runLength);

				componentFailProb(pointerToResCapStats, numCols);

				systemFailureRate.push_back(numberFailed);

				numberFailed = 0;

				reliability_calc(pointerToResCapStats,
pointerToResCapStatsIntegrated, pointerToResCapInput, componentReliability,
numCols, runLength, boolManualTTF, true, true, replication);

				for(int comp = 0; comp < numCols; comp++)
				{
					pointerToResCapStats[comp].TTR_total = 0;
					pointerToResCapStats[comp].replaceTotal = 0;

					for(unsigned int col = 0; col <
pointerToResCapStats[comp].repair.size(); col++)
					{
						pointerToResCapStats[comp].TTR_total +=
pointerToResCapStats[comp].repair[col];
					}

					for(unsigned int col = 0; col <
pointerToResCapStats[comp].replace.size(); col++)
					{
						pointerToResCapStats[comp].replaceTotal +=
pointerToResCapStats[comp].replace[col];
					}

					pointerToResCapStats[comp].TTR_total =
pointerToResCapStats[comp].TTR_total / pointerToResCapStats[comp].repair.size();
					pointerToResCapStats[comp].replaceTotal =
pointerToResCapStats[comp].replaceTotal /
pointerToResCapStats[comp].replace.size();
				}

				failTable = totalFails(failTable, manualTTF, manualTTR,
manualReplace, timeInc, runLength, pointerToResCapStatsIntegrated, numCols,
boolManualRepair, boolManualReplace, boolManualTTF, replication, thermalComponent,
pointerToResCapLife, pointerToResCapInput);

				if(failTable.failed.size() > 0 &&
					failTable.fix.size() >0)
				{
					failTableIntegrated.push_back(failTable);
				}

				// delete all arrays except for the total integrated one
				delete []pointerToResCapStats;
				delete []pointerToResCapLife;
```

```
                    failTable.failed.clear();
                    failTable.fix.clear();
                    failTable.partID.clear();

                    // if the replication number is less than the total number of
replications required, goto begininning of program
                    if(replication < numberOfReplications)
                    {
                           goto newReplication;
                    }

                    // if it is the last replication create a new table of all
statisticalVectors and print out to a file
                    else
                    {
                           meanGenerateTTF(exportGeneratedMTTR);

                           double AVG = 0;

                           int leastFailed = INT_MAX;
                           double averageFailure = 0;

                           for(unsigned int comp = 0; comp <
failTableIntegrated.size(); comp++)
                           {
                                  if(failTableIntegrated[comp].failed.size() <
leastFailed)
                                  {
                                         leastFailed =
failTableIntegrated[comp].failed.size();
                                  }

                                  averageFailure +=
failTableIntegrated[comp].partsFailed;
                           }

                           averageFailure =
averageFailure/failTableIntegrated.size();
                           averageFailure = ceil(averageFailure);

                           systemFailureRate[0] = averageFailure;

                           vector <failing>
outputMTTR(failTableIntegrated.size());

                           for(unsigned int comp = 0; comp < outputMTTR.size();
comp++)
                           {
                                  outputMTTR[comp].meanFix =
failTableIntegrated[comp].fix[0] - failTableIntegrated[comp].failed[0];
                           }

                           outputFailTable.resize(leastFailed);

                           componentReliability.resize(numCols);
```

```
                            for(unsigned int comp = 0; comp <
outputFailTable.size(); comp++)
                            {
                                   outputFailTable[comp].mean = 0;
                                   outputFailTable[comp].standardDeviation = 0;
                                   outputFailTable[comp].meanFix = 0;
                                   outputFailTable[comp].standDevFix = 0;
                            }

                            int yous = 0;
                            int numberFailed = 0;

                            for(unsigned int comp = 0; comp <
failTableIntegrated.size(); comp++)
                            {
                                   if(yous < leastFailed)
                                   {
                                          outputFailTable[yous].meanFix +=
failTableIntegrated[comp].fix[yous];
                                          outputFailTable[yous].mean +=
failTableIntegrated[comp].failed[yous];
                                          numberFailed++;
                                   }

                                   if(comp == failTableIntegrated.size() -1 &&
                                          yous != leastFailed)
                                   {
                                          outputFailTable[yous].mean =
outputFailTable[yous].mean / numberFailed;
                                          outputFailTable[yous].meanFix =
outputFailTable[yous].meanFix / numberFailed;

       outputFailTable[yous].mu.push_back(outputFailTable[yous].mean);

                                          numberFailed = 0;

                                          comp = 0;
                                          yous++;
                                   }

                                   if(comp == failTableIntegrated.size()-1 &&
                                          yous == leastFailed-1)
                                   {
                                          goto exit1;
                                   }
                            }

                            exit1:

                            double SD = 0;
                            double sdFix = 0;
                            yous = 0;
                            numberFailed = 0;
                            double powerFail;
                            double powerFix;
                            minMTTFvalue = 0;
                            for(unsigned int comp = 0; comp <
failTableIntegrated.size(); comp++)
                            {
                                   if(yous < leastFailed)
                                   {
                                          powerFix =
pow((failTableIntegrated[comp].fix[yous] - outputFailTable[yous].meanFix),2);
                                          powerFail =
pow((failTableIntegrated[comp].failed[yous] - outputFailTable[yous].mean),2);
                                          sdFix = sdFix + powerFix;
                                          SD = SD + powerFail;
                                          numberFailed++;
                                   }

                                   if(comp == failTableIntegrated.size()-1 &&
                                          yous != leastFailed)
                                   {
                                          sdFix = sdFix / leastFailed;
                                          SD = SD / leastFailed;

                                          outputFailTable[yous].standDevFix =
sqrt(sdFix);
                                          outputFailTable[yous].standardDeviation =
sqrt(SD);

       outputFailTable[yous].SD.push_back(failTableIntegrated[comp].standardDeviat
ion);

                                          SD = 0;
                                          sdFix = 0;

                                          yous++;

                                          comp = 0;
                                   }

                                   if(comp == failTableIntegrated.size()-1 &&
                                          yous == leastFailed-1)
                                   {
                                          goto exit;
                                   }
                            }

                            exit:

                            for(unsigned int comp = 0; comp <
outputFailTable.size(); comp++)
                            {
                                   outputFailTable[comp].maxFail =
outputFailTable[comp].mean + (1.5 * outputFailTable[comp].standardDeviation);
                                   outputFailTable[comp].minFail =
outputFailTable[comp].mean - (1.5 * outputFailTable[comp].standardDeviation);

                                   if(outputFailTable[comp].minFail < 0)
                                   {
                                          outputFailTable[comp].minFail = 0;
                                   }
                            }
```

```
				for(unsigned int comp = 0; comp < 
outputFailTable.size(); comp++)
				{
					for(unsigned int row = 0; row < 
failTableIntegrated.size(); row++)
					{
						
	outputFailTable[comp].partID.push_back(failTableIntegrated[row].partID[comp
]);
					}
				}

				for(unsigned int comp = 0; comp < 
outputFailTable.size(); comp++)
				{
					for(unsigned int row = 0; row < 
outputFailTable[comp].partID.size(); row++)
					{
						for(unsigned int row2 = 0; row2 < 
outputFailTable[comp].partID.size(); row2++)
						{
							if(row != row2 &&
							
	outputFailTable[comp].partID[row] == outputFailTable[comp].partID[row2])
							{
								
	outputFailTable[comp].partID.erase(outputFailTable[comp].partID.begin()+row
2);

								row2 = 0;
							}
						}
					}
				}

				computeSystemReliability(reliable, failTableIntegrated, 
outputFailTable, runLength, numberOfReplications);

				if(boolManualTTF == 0 &&
					boolManualRepair == 0 &&
					boolManualReplace == 0)
				{
					componentReliability.resize(numCols);
					componentSave = 0;

					MTTF_total(pointerToResCapStats, 
pointerToResCapStatsIntegrated, pointerToResCapLife, numCols, replication, 
numberOfReplications, boolManualReplace, boolManualRepair, boolManualTTF, 
manualTTR, manualReplace);

					reliability_calc(pointerToResCapStatsIntegrated, 
pointerToResCapStatsIntegrated, pointerToResCapInput, componentReliability, 
numCols, runLength, boolManualTTF, false, false, replication);

					//finalFails = systemFailProb(networks, 
pointerToResCapLife, pointerToResCapStatsIntegrated, finalLife, reliable, numCols, 
replication, runLength);
```

```
                                  //systemMTTF = finalFails.fail[0];
                                  //systemFailRate = 1 / finalFails.fail[0];
                                  // allow the user to check the reliability of a
component of their choosing as well as the time

                                  //minMaxTTF(pointerToResCapStatsIntegrated,
numCols);

                                  for(int comp = 0; comp < numCols; comp++)
                                  {

       pointerToResCapStatsIntegrated[comp].MTTFsave =
pointerToResCapStatsIntegrated[comp].MTTF;
                                  }

                                  int minComponentPosition = 0;

                                  for(int comp = 0; comp < numCols; comp++)
                                  {

       if(pointerToResCapStatsIntegrated[comp].TTFmin < 0)
                                         {

       pointerToResCapStatsIntegrated[comp].TTFmin = 0;
                                         }

                                         pointerToResCapStatsIntegrated[comp].MTTF
= pointerToResCapStatsIntegrated[comp].TTFmin;
                                  }

                                  //systemMinTTF = checkParallel(networks,
pointerToResCapStatsIntegrated, pointerToResCapLife, finalLife, reliable, numCols,
minComponentPosition, true, runLength);

                                  /*if(systemMinTTF.fail[0] < 0)
                                  {
                                         systemMinTTF.fail[0] = 0;
                                  }*/

                                  for(int comp = 0; comp < numCols; comp++)
                                  {
                                         pointerToResCapStatsIntegrated[comp].MTTF
= pointerToResCapStatsIntegrated[comp].TTFmax;
                                  }

                                  //systemMaxTTF = checkParallel(networks,
pointerToResCapStatsIntegrated, pointerToResCapLife, finalLife, reliable, numCols,
minComponentPosition, true, runLength);

                                  for(int comp = 0; comp < numCols; comp++)
                                  {
                                         pointerToResCapStatsIntegrated[comp].MTTF
= pointerToResCapStatsIntegrated[comp].MTTFsave;
                                  }
```

```
					outputFinalTable(pointerToResCapStatsIntegrated, 
numCols, minMTTFvalue, boolManualTTF, outputDirectory, maintainFile, reliable, 
componentReliability, boolManualRepair, boolManualReplace, failTable, 
parallelFailing, pointerToResCapInput, outputFailTable, failTableIntegrated, 
thermalComponent, systemFailureRate,exportGeneratedMTTR, outputMTTR);
				}

				else if(boolManualTTF == 1)
				{

	pointerToResCapStatsIntegrated[manualComponentSave].MTTF = manualTTF;

	pointerToResCapStatsIntegrated[manualComponentSave].distribution = 2;

	pointerToResCapStatsIntegrated[manualComponentSave].expoMean = manualTTF;

					componentReliability.resize(numCols);
					componentSave = 0;

					MTTF_total(pointerToResCapStats, 
pointerToResCapStatsIntegrated, pointerToResCapLife, numCols, replication, 
numberOfReplications, boolManualReplace, boolManualRepair, boolManualTTF, 
manualTTR, manualReplace);

					if(boolManualRepair == 1)
					{

	pointerToResCapStatsIntegrated[manualComponentSave].TTR_total = manualTTR;
					}

					if(boolManualReplace == 1)
					{

	pointerToResCapStatsIntegrated[manualComponentSave].replaceTotal = 
manualReplace;
					}

					reliability_calc(pointerToResCapStatsIntegrated, 
pointerToResCapStatsIntegrated, pointerToResCapInput, componentReliability, 
numCols, runLength, boolManualTTF, false,false, replication);

					//finalFails = systemFailProb(networks, 
pointerToResCapLife, pointerToResCapStatsIntegrated, finalLife, reliable, numCols, 
replication, runLength);

					//computeSystemReliability(reliable);

					//systemMTTF = finalFails.fail[0];
					//systemFailRate = 1 / finalFails.fail[0];

					//minMaxTTF(pointerToResCapStatsIntegrated, 
numCols);


	pointerToResCapStatsIntegrated[manualComponentSave].TTFmax = manualTTF;

	pointerToResCapStatsIntegrated[manualComponentSave].TTFmin = manualTTF;
```

```
                                for(int comp = 0; comp < numCols; comp++)
                                {

       pointerToResCapStatsIntegrated[comp].MTTFsave =
pointerToResCapStatsIntegrated[comp].MTTF;
                                }

                                int minComponentPosition = 0;

                                for(int comp = 0; comp < numCols; comp++)
                                {

       if(pointerToResCapStatsIntegrated[comp].TTFmin < 0)
                                       {

       pointerToResCapStatsIntegrated[comp].TTFmin = 0;
                                       }

                                       pointerToResCapStatsIntegrated[comp].MTTF
= pointerToResCapStatsIntegrated[comp].TTFmin;
                                }

                         //     systemMinTTF = checkParallel(networks,
pointerToResCapStatsIntegrated, pointerToResCapLife, finalLife, reliable, numCols,
minComponentPosition, true, runLength);

                                /*if(systemMinTTF.fail[0] < 0)
                                {
                                       systemMinTTF.fail[0] = 0;
                                }*/

                                for(int comp = 0; comp < numCols; comp++)
                                {
                                       pointerToResCapStatsIntegrated[comp].MTTF
= pointerToResCapStatsIntegrated[comp].TTFmax;
                                }

                                //systemMaxTTF = checkParallel(networks,
pointerToResCapStatsIntegrated, pointerToResCapLife, finalLife, reliable, numCols,
minComponentPosition, true, runLength);

                                for(int comp = 0; comp < numCols; comp++)
                                {
                                       pointerToResCapStatsIntegrated[comp].MTTF
= pointerToResCapStatsIntegrated[comp].MTTFsave;
                                }

                                outputFinalTable(pointerToResCapStatsIntegrated,
numCols, minMTTFvalue, boolManualTTF, outputDirectory, maintainFile, reliable,
componentReliability, boolManualRepair, boolManualReplace, failTable,
parallelFailing, pointerToResCapInput, outputFailTable,failTableIntegrated,
thermalComponent, systemFailureRate, exportGeneratedMTTR, outputMTTR);
                         }

                         else if(boolManualRepair == 1 ||
                                boolManualReplace == 1)
                         {
                                componentReliability.resize(numCols);
```

```
                                    componentSave = 0;

                                    MTTF_total(pointerToResCapStats,
pointerToResCapStatsIntegrated, pointerToResCapLife, numCols, replication,
numberOfReplications, boolManualReplace, boolManualRepair, boolManualTTF,
manualTTR, manualReplace);

                                    if(boolManualRepair == 1)
                                    {

       pointerToResCapStatsIntegrated[manualComponentSave].TTR_total = manualTTR;
                                    }

                                    if(boolManualReplace == 1)
                                    {

       pointerToResCapStatsIntegrated[manualComponentSave].replaceTotal =
manualReplace;
                                    }

                                    reliability_calc(pointerToResCapStatsIntegrated,
pointerToResCapStatsIntegrated, pointerToResCapInput, componentReliability,
numCols, runLength, boolManualTTF, false,false, replication);

                                    //finalFails = systemFailProb(networks,
pointerToResCapLife, pointerToResCapStatsIntegrated, finalLife, reliable, numCols,
replication, runLength);

                                    //systemMTTF = finalFails.fail[0];
                                    //systemFailRate = 1 / finalFails.fail[0];
                                    // allow the user to check the reliability of a
component of their choosing as well as the time

                                    //minMaxTTF(pointerToResCapStatsIntegrated,
numCols);

                                    for(int comp = 0; comp < numCols; comp++)
                                    {

       pointerToResCapStatsIntegrated[comp].MTTFsave =
pointerToResCapStatsIntegrated[comp].MTTF;
                                    }

                                    int minComponentPosition = 0;

                                    for(int comp = 0; comp < numCols; comp++)
                                    {

       if(pointerToResCapStatsIntegrated[comp].TTFmin < 0)
                                           {

       pointerToResCapStatsIntegrated[comp].TTFmin = 0;
                                           }

                                           pointerToResCapStatsIntegrated[comp].MTTF
= pointerToResCapStatsIntegrated[comp].TTFmin;
                                    }
```

```
					//systemMinTTF = checkParallel(networks, 
pointerToResCapStatsIntegrated, pointerToResCapLife, finalLife, reliable, numCols, 
minComponentPosition, true, runLength);

					/*if(systemMinTTF.fail[0] < 0)
					{
						systemMinTTF.fail[0] = 0;
					}*/

					for(int comp = 0; comp < numCols; comp++)
					{
						pointerToResCapStatsIntegrated[comp].MTTF 
= pointerToResCapStatsIntegrated[comp].TTFmax;
					}

					//systemMaxTTF = checkParallel(networks, 
pointerToResCapStatsIntegrated, pointerToResCapLife, finalLife, reliable, numCols, 
minComponentPosition, true, runLength);

					for(int comp = 0; comp < numCols; comp++)
					{
						pointerToResCapStatsIntegrated[comp].MTTF 
= pointerToResCapStatsIntegrated[comp].MTTFsave;
					}

					outputFinalTable(pointerToResCapStatsIntegrated, 
numCols, minMTTFvalue, boolManualTTF, outputDirectory, maintainFile, reliable, 
componentReliability, boolManualRepair, boolManualReplace, failTable, 
parallelFailing, pointerToResCapInput, outputFailTable,failTableIntegrated, 
thermalComponent, systemFailureRate, exportGeneratedMTTR, outputMTTR);
				}
			}
		}
	}

	// END PROGRAM
	return;
}

// name: numberColumns
// inputs: columnCount = string = entire first line of the input file
// outputs: numCols = int = number of words in that column
// description: finds the number of columns in an input file by taking in the 
entire first row
//			and incrementing a counter every time a new word is 
found after a whitespace
int numberColumns(string columnCount)
{
	// number of columns
	int numCols=0;

	// converts the string into a stringstream for string manipulation
	// converts a string into a stream interface
	stringstream ss(columnCount);
	string word;

	// before every white space read in word, and increment number of columns
	while( ss >> word )
	{
		numCols++;
	}

	// return number of columns (# of columns = # of components)
	return numCols;
}

// name: importFREPLREPA_files
// inputs: pointerToResCapStats = statisticsVectors = contains the 6 tables for
each replication
//			numRows = int = number of rows contained in each components
TBF, TBR, or replace
//			numCols = int = number of resistors
//			fileName = string = name of the file the user specified
//			whatToFill = int = determines which table to fill (TBF, TBF,
Replace)
// Description: opens a user specified file and copies all the data from the table
into the tables
//				created in the statisticsVectors struct
void importFREPLREPA_files(statisticsVectors * pointerToResCapStats, int numRows,
int numCols, string *fileName, int whatToFill)
{
	fstream FILE_IN;
	string Line;

		string FILE = *fileName;

		// open file
		FILE_IN.open(FILE.c_str());

			// if file is available to open
			if(FILE_IN.is_open())
			{
				// run through the program column by column, then row
by row and fill the specified vector
				while(getline(FILE_IN,Line,'\n'))
				{
					for(int row = 0; row < numRows; row++)
					{
						for(int col = 0; col < numCols; col++)
						{
							// TBF vector
							if(whatToFill == 0)
							{
								FILE_IN >>
pointerToResCapStats[col].timeBetweenFail[row];
							}

							// repair vector
							else if(whatToFill == 1)
							{
								FILE_IN >>
pointerToResCapStats[col].repair[row];
							}

							// replace vector
							else
```

```
                                                {
                                                        FILE_IN >>
pointerToResCapStats[col].replace[row];
                                                }
                                        }
                                }
                        }

                        //close file
                        FILE_IN.close();
                }

        FILE_IN.clear();

        return;
}

// Name: importNetworksFile
// Inputs: networks = componentNetworks = pointer to component networks structure
where networks will be stored
//                      fileName = pointer to string = pointer to the file name
specified by user containing the network ID's
// Description: Goes into a file which contains the network information, and
places all terminal 1 networks for a
//                              component in a vector, and the terminal 2 (network out)
into a seperate vector.
void importNetworksFile(componentNetworks * networks, string *fileName, int
numCols)
{
        int fileChoice = 1;
        int componentPosition = 0;
        int terminal;
        int network;

        bool firstNumber = true;

        fstream FILE_IN;

        string Line;
        string part;

        // while file not found, keep reentering files
        while(fileChoice != 2)
        {
                string FILE = *fileName;

                // open file
                FILE_IN.open(FILE.c_str());

                // if the user chooses to open a file
                if(fileChoice == 1)
                {
                        // if file is available to open
                        if(FILE_IN.is_open())
                        {
                                // run through the program column by column, then row
by row and fill the specified vector
                                while(getline(FILE_IN,Line,'\n'))
```

```
                            {
                                    FILE_IN >> part >> terminal >> network;

                                    if(part == manualComponent)
                                    {
                                            manualComponentSave = componentPosition;
                                    }

                                    if(firstNumber)
                                    {
                                            networks[componentPosition].partID =
part;

        networks[componentPosition].allNets.push_back(network);

                                            if(terminal == 1)
                                            {

        networks[componentPosition].networkIN = network;
                                            }

                                            else if(terminal == 2)
                                            {

        networks[componentPosition].networkOUT = network;
                                            }

                                            else
                                            {

        networks[componentPosition].extraNets.push_back(network);
                                            }

                                            firstNumber = false;
                                    }

                                    else if(part ==
networks[componentPosition].partID)
                                    {

        networks[componentPosition].allNets.push_back(network);

                                            if(terminal == 1)
                                            {

        networks[componentPosition].networkIN = network;
                                            }

                                            else if(terminal == 2)
                                            {

        networks[componentPosition].networkOUT = network;
                                            }

                                            else
                                            {

        networks[componentPosition].extraNets.push_back(network);
```

```
                                                }
                                        }

                                        else if(part !=
networks[componentPosition].partID)
                                        {
                                                componentPosition++;
                                                networks[componentPosition].partID =
part;

        networks[componentPosition].allNets.push_back(network);

                                                if(terminal == 1)
                                                {

        networks[componentPosition].networkIN = network;
                                                }

                                                else if(terminal == 2)
                                                {

        networks[componentPosition].networkOUT = network;
                                                }

                                                else
                                                {

        networks[componentPosition].extraNets.push_back(network);
                                                }
                                        }
                                }

                                //close file
                                FILE_IN.close();

                                // set fileChoice to 2 as to prevent re-looping
                                fileChoice = 2;
                        }
                }
        }

        FILE_IN.clear();

        return;
}

// Name: importAgeFile
// inputs: pointerToResCapStats- statisticsVector - contains all statistical
information
//                  numCols - int - number of components in the PCB
//                  fileName - string - file directory
// Description: finds the ComponentAge input file and inports the ages of all
components
void importAgeFile(statisticsVectors * pointerToResCapStats, int numCols, string
*fileName)
{
        fstream FILE_IN;
        string line;
        string part;
        string FILE;
        double age;

        FILE = *fileName;

        FILE_IN.open(FILE.c_str());

        if(FILE_IN.is_open())
        {
                // run through the program column by column, then row by row and
fill the specified vector
                while(getline(FILE_IN,line,'\n'))
                {
                        FILE_IN >> part >> age;

                        for(int comp = 0; comp < numCols; comp++)
                        {
                                if(pointerToResCapStats[comp].partID == part)
                                {
                                        pointerToResCapStats[comp].age = age;
                                }
                        }
                }
        }

        FILE_IN.close();
        FILE_IN.clear();

        return;
}

void importThermalFile(vector <thermal> & thermalComponent, string *fileName)
{
        fstream FILE_IN;
        string line;
        string part;
        string FILE;
        double MTTF;
        double Eo;
        int numbers = 0;

        thermal thermalInc;

        FILE = *fileName;

        FILE_IN.open(FILE.c_str());

        if(FILE_IN.is_open())
        {
                // run through the program column by column, then row by row and
fill the specified vector
                while(getline(FILE_IN,line,'\n'))
                {
                        FILE_IN >> part >> MTTF >> Eo;

                        thermalInc.partID = part;
                        thermalInc.MTTF = MTTF;
                        thermalInc.epsillon = Eo;

                        for(unsigned int comp = 0; comp < thermalComponent.size(); 
comp++)
                        {
                                if(thermalInc.partID == thermalComponent[comp].partID)
                                {
                                        numbers++;
                                }
                        }

                        if(numbers == 0)
                        {
                                thermalComponent.push_back(thermalInc);
                        }

                        numbers = 0;
                }
        }

        FILE_IN.close();
        FILE_IN.clear();

        return;
}

// name: CDF_calc
// inputs: pointerToResCapStats = statisticsVectors = contains the 3 filled and 3 
empty tables (to be filled)
//                      pointerToResCapStatsIntegrated = statisticsVectors = contains 
the 3 final output tables from all replications
//                      numRows = int = number of rows contained in each component 
column
//                      numCols = int = number of resistors (number of columns)
//                      CDF_TYPE = int = determines what the program should be taking 
the CDF of
// description: takes in the data from one component (either TTF, TTR, or replace) 
and finds the CDF of that
//                              component
void CDF_calc(statisticsVectors * pointerToResCapStats, statisticsVectors * 
pointerToResCapStatsIntegrated, int numRows, int numCols, int CDF_TYPE)
{
        int i;
        int sameValue = 0;
        Component * CDF;

        // solve the CDF
        for(int cols = 0; cols < numCols; cols++)
        {
                // create a new CDF array of Component structure
                CDF = new Component[numRows];

                // fill rows
                for(int s = 0; s < numRows; s++)
                {
                        // file CDF with TBF
                        if(CDF_TYPE == 0)
                        {
```

```
                        CDF[s].value = 
pointerToResCapStats[cols].timeBetweenFail[s];
                    }

                    // fille CDF with TTR
                    else if( CDF_TYPE == 1)
                    {
                        CDF[s].value = pointerToResCapStats[cols].repair[s];
                    }

                    // fill CDF with repair
                    else if( CDF_TYPE == 2)
                    {
                        CDF[s].value = pointerToResCapStats[cols].replace[s];
                    }

                    // otherwise fille it with TBF?
                    else
                    {
                        CDF[s].value = 
pointerToResCapStatsIntegrated[cols].timeBetweenFail[s];
                    }

                    // initialize other structure elements required in CDF 
calculation
                    CDF[s].polyNum = false;
                    CDF[s].cdfed = false;
                }

                // solves the CDF for each of the components based on ascending 
order
                for(int q = 0; q < numRows; q++)
                {
                    // current rank
                    i = q + 1;

                    // check to see if and which components have multiple values 
that are the same
                    for(int L = q+1; L < numRows; L++)
                    {
                        if(CDF[q].value == CDF[L].value &&
                            !CDF[L].polyNum)
                        {
                            sameValue++;

                            CDF[L].polyNum = true;
                        }
                    }

                    // if 2 or more part have the same MTBF then the CDF will be 
the sum of both resistors
                    if(sameValue > 0 &&
                        !CDF[q].cdfed)
                    {
                        CDF[q].CDF = (i+sameValue)/(numRows+1);
                        CDF[q].cdfed = true;
                        CDF[q].rank = i+sameValue;
```

```
                              for(int j = 0; j < numRows; j++)
                              {
                                     if(CDF[q].value == CDF[j].value &&
                                            q!=j)
                                     {
                                            CDF[j].CDF = CDF[q].CDF;
                                            CDF[j].rank = CDF[q].rank;
                                            CDF[j].cdfed = true;
                                     }
                              }
                       }

                       // otherwise the CDF should increment accordingly with the
rank
                       else
                       {
                              if(!CDF[q].cdfed)
                              {
                                     CDF[q].rank = i;
                                     CDF[q].CDF = i/(numRows+1);
                                     CDF[q].cdfed = true;
                              }
                       }

                       sameValue = 0;
              }

              for(int s = 0; s < numRows; s++)
              {
                       // fill TBF CDF vector with data obtained
                       if(CDF_TYPE == 0)
                       {
                              pointerToResCapStats[cols].timeBetweenFailCDF[s] =
CDF[s].CDF;
                              pointerToResCapStats[cols].TBFrank[s] = CDF[s].rank;
                       }

                       // fill the TTR CDF vector with data obtained
                       else if(CDF_TYPE == 1)
                       {
                              pointerToResCapStats[cols].repairCDF[s] = CDF[s].CDF;
                              pointerToResCapStats[cols].repairRank[s] = CDF[s].rank;
                       }

                       // fill the replace CDF with teh data obtained
                       else if(CDF_TYPE == 2)
                       {
                              pointerToResCapStats[cols].replaceCDF[s] = CDF[s].CDF;
                              pointerToResCapStats[cols].replaceRank[s] =
CDF[s].rank;
                       }

                       // otherwise just fill the TBF
                       else
                       {

       pointerToResCapStatsIntegrated[cols].timeBetweenFailCDF.push_back(CDF[s].CD
F);
```

```
				}
			}

			// delete dynamic array for reuse
			delete []CDF;
		}

		// return NULL
		return;
	}

	// function name: MEAN
	// inputs: pointerToResCapCalculation - pointer to Component - pointer to
	resistor/capacitor Component structure
	//			numRows = int = number of values in components
	randomNumberType
	// output: mean - double - mean value of the MTBF of all copmonents
	// Description: takes in all the MTBF's of the components and finds the mean
	double MEAN(Component * pointerToResCapCalculation, int numRows)
	{
		int n;
		double sum = 0;
		double mean;

		n = numRows;;

		// finds the total sum of the component
		for(int q = 0; q < numRows; q++)
		{
			sum += pointerToResCapCalculation[q].value;
		}

		// divide the total sum of the values by the number of values
		mean = sum/numRows;

		// return the mean back to the main function
		return mean;
	}

	// function name: STDDEV
	// inputs: pointerToResCapCalculation - pointer to Component - pointer to
	resistor/capacitor Component structure
	//			numRows = int = number of values contained within a specific
	components randomNumberType
	//			mean = double - mean value of all MTBF
	// output: mtbf_sigman - double - standard deviation of all values
	// Description: calculates the standard deviation of the components MTBF by taking
	the square root of the summation of squared
	//				deviations divided by the total number of components
	double STDDEV(Component * pointerToResCapCalculation, int numRows, double mean)
	{
		double squareDevSum = 0;
		double sigmaSquared;
		double sigma;

		// calculate the total summation of hte squared deviations
		for(int q = 0; q < numRows; q++)
		{
```

```
		squareDevSum += ((pointerToResCapCalculation[q].value - mean) * 
(pointerToResCapCalculation[q].value - mean));
	}

	// divide the total squared deviations by the number of values
	sigmaSquared = squareDevSum/numRows;

	// standard deviation is equal to the square root of variance (sigmaSquared 
in this case)
	sigma = sqrt(sigmaSquared);

	// return standard deviation to main
	return sigma;
}

// function name: log_MEAN
// inputs: pointerToResCapCalculation - pointer to Component - pointer to 
resistor/capacitor Component structure
//			n = int = number of values contained in the components 
randomNumberType
// output: mean - double - mean value of all copmonents
// Description: takes in all the time values of the components and finds the mean
double log_mean(Component * pointerToResCapCalculation, int n)
{
	double logMean = 0;

	// find the sum of all the values natural logs
	for(int q = 0; q < n; q++)
	{
		logMean += log(pointerToResCapCalculation[q].value);
	}

	// divide the sum by the number of values
	logMean = logMean / n;

	// return the logMean to main
	return logMean;
}

// name: log_sigma
// inputs: pointerToResCapCalculation = Component = structure used in calculating 
parameters
//			n = number of rows in file
//outputs: logSigma = double = standard deviation for logs
// description: calculates the log standard deviation based upon the the log 
values of each component
double log_sigma(Component * pointerToResCapCalculation, int n, double logMean)
{
	double logSigma=0;
	double logValMeanSummation=0;
	double tAfterSquared = 0;

	// calculate the summation of the log of the squared deivations
	for(int q = 0; q < n; q++)
	{
		logValMeanSummation += 
pow((log(pointerToResCapCalculation[q].value)-logMean),2);
	}
```

```
        // divide this sum by the number of values
        logValMeanSummation = logValMeanSummation/n;

        // take the square root of the variance (logValMeanSummation)
        logSigma = sqrt(logValMeanSummation);

        // return logSigma to main
        return logSigma;
}

// function name: betaCalc
// inputs: pointerToResCapCalculation - pointer to Component - pointer to
resistor/capacitor Component structure
//                  numRows - int - number of values contained in a components
vectors
//                  mean = double - mean value
//                  sigma - double - standard deviation between the components
// Description: finds the value of beta for the weibull distribution
double betaCalc(Component * pointerToResCapCalculation, int numRows, double mean,
double sigma)
{
        double beta;
        double inDoubleLog;
        int n = numRows;
        double value;

        // create vectors to hold all the summation calculation values to calculate
the summation later
        std::vector<double> firstSummation(n);
        std::vector<double> secondSummation(n);
        std::vector<double> thirdSummation(n);
        std::vector<double> fourthSummation(n);

        // calculate values to be summed
        for(int q = 0; q < n; q++)
        {
                inDoubleLog = (1.0000 / ( 1.0000 -
(pointerToResCapCalculation[q].rank / (n + 1.0000) ) ) );
                inDoubleLog = log(inDoubleLog);
                inDoubleLog = log(inDoubleLog);

                value = pointerToResCapCalculation[q].value;
                value = log(value);

                firstSummation[q] = value * inDoubleLog;
                secondSummation[q] = inDoubleLog;
                thirdSummation[q] = value;
                fourthSummation[q] = value*value;
        }

        double sumOfFirst=0;
        double sumOfSecond=0;
        double sumOfThird=0;
        double sumOfFourth=0;

        // calculate the sums
        for(unsigned int q = 0; q < firstSummation.size(); q++)
```

```
	{
		sumOfFirst += firstSummation[q];
		sumOfSecond += secondSummation[q];
		sumOfThird += thirdSummation[q];
		sumOfFourth += fourthSummation[q];
	}

	// use thes summations in the beta calculation equation
	beta = (((n * sumOfFirst) - (sumOfSecond * sumOfThird)) / ((n * 
sumOfFourth) - (sumOfThird * sumOfThird)));

	// clear the vector contents of each for reuse
	firstSummation.clear();
	secondSummation.clear();
	thirdSummation.clear();
	fourthSummation.clear();

	// return beta to the main function
	return beta;
}

// name: etaCalc
// inputs: pointerToResCapCalculation = Component = structure used in calculating 
parameters
//		   numRows = int = number of rows in TTF file
//		   beta = double = shape parameter
//		   mean = double = mean of all values in a components input file
// outputs: eta = double = scale parameter
// description: obtains the components eta based on the values
double etaCalc(Component * pointerToResCapCalculation, int numRows, double beta, 
double mean)
{
	double eta;
	double n = numRows;
	double value=0;
	double i = 1.00000;
	double root = 1/beta;

	// sum all values in column
	for(int q = 1 ; q <= numRows;q++)
	{
		value += pow(pointerToResCapCalculation[q-1].value,beta);
	}

	// divide the sum by the number of values
	eta = value / n;

	// eta^(1/beta)
	eta = pow(eta,root);

	// return eta to the main function
	return eta;
}

// name: weibullMean
// inputs: beta - double - beta of the values of the given component
//		   eta - double - eta of hte values of the given component
// outputs: mean - double - mean of the calculated weibull distribution
// description: calculates the mean of a component that follows a weibull
distribution
double weibullMean(double beta, double eta)
{
	float roundedGammaAlpha;
	double gammaAlpha;
	double removePara = 1;
	double table_alpha;
	double table_gamma;
	double mean;
	double gammaTotal;

	fstream gammaOpen;

	string line;

	gammaAlpha = 1 + 1/beta;

	vector <double> alpha;
	vector <double> gamma;

	while(gammaAlpha >= 1.995)
	{
		gammaAlpha--;

		removePara *= gammaAlpha;
	}

	roundedGammaAlpha = floorf(gammaAlpha * 100 + 0.5) / 100;

	gammaOpen.open("gammaTable.txt");

	if(gammaOpen.is_open())
	{
		while(getline(gammaOpen,line,'\n'))
		{
			gammaOpen >> table_alpha >> table_gamma;

			alpha.push_back(table_alpha);
			gamma.push_back(table_gamma);
		}

		gammaOpen.close();
		gammaOpen.clear();
	}

	for(unsigned int table = 1; table < alpha.size(); table++)
	{
		if(roundedGammaAlpha == alpha[table])
		{
			table_gamma = gamma[table];
		}
	}

	gammaTotal = table_gamma*removePara;

	mean = gammaTotal * eta;
	return mean;
}

// name: weibullSigma
// inputs: beta - double - beta for corresponding component
//		       eta - double - eta for corresponding component
// outputs: sigma - double - standard deviation of component
// description: calculates the standard devation of a component given the shape
//			     and scale parameters
double weibullSigma(double beta, double eta)
{
	double roundedGammaAlpha1;
	double roundedGammaAlpha2;
	double variance;
	double sigma;
	double table_gamma;
	double table_alpha;
	double gamma1;
	double gamma2;
	double gammaTotal;
	double remove1 =1;
	double remove2 =1;

	fstream gammaOpen;

	string line;

	vector <double> alpha;
	vector <double> gamma;

	eta = eta * eta;

	gamma1 = 1 + 2/beta;
	gamma2 = 1 + 1/beta;

	while(gamma1 >= 1.995)
	{
		gamma1--;

		remove1 *= gamma1;
	}

	while(gamma2 >= 1.995)
	{
		gamma2--;

		remove2 *= gamma2;
	}

	roundedGammaAlpha1 = floorf(gamma1 * 100 + 0.5) / 100;
	roundedGammaAlpha2 = floorf(gamma2 * 100 + 0.5) / 100;

	gammaOpen.open("gammaTable.txt");

	if(gammaOpen.is_open())
	{
		while(getline(gammaOpen,line,'\n'))
		{
			gammaOpen >> table_alpha >> table_gamma;

			alpha.push_back(table_alpha);
			gamma.push_back(table_gamma);
		}

		gammaOpen.close();
		gammaOpen.clear();
	}

	for(unsigned int table = 1; table < alpha.size(); table++)
	{
		if(roundedGammaAlpha1 == alpha[table])
		{
			gamma1 = gamma[table];
		}
	}

	for(unsigned int table = 1; table < alpha.size(); table++)
	{
		if(roundedGammaAlpha2 == alpha[table])
		{
			gamma2 = gamma[table];
		}
	}

	gamma1 = gamma1*remove1;
	gamma2 = gamma2*remove2;

	gamma2 = gamma2 * gamma2;

	gammaTotal = gamma1 - gamma2;

	variance = eta * eta * gammaTotal;

	sigma = sqrt(variance);

	return sigma;
}

// name: exponentialMean
// inputs: eta - double - eta value corresponding to component
// outputs: mean - double - mean of the exponential component
// description: calculates the mean of a component which follows an exponential
//                         distribution, mean will be equal to eta
double exponentialMean(double eta)
{
	double mean;

	mean = eta;

	return mean;
}

// name: exponentialSigma
// inputs: eta - double - eta value corresponding ot component
// outputs: sigma - double - standard deviation of component
// description: calculates the standard deviation of a component following
//                          an exponential distribution, mean is equal to eta
double exponentialSigma(double eta)
{
       double sigma;

       sigma = eta * eta * 3;

       sigma = sqrt(sigma);

       return sigma;
}

// name: lognormalFinalMean
// inputs: mean - double - mean of a lognormal distribution
//                   sigma - double - standard deviation of a component
// outputs: mean - double - final lognormal mean of component
// description: calculates the mean of a component that is following the
//                          lognormal distribution
double lognormalFinalMean(double mean, double sigma)
{
       double meanNew;

       meanNew = exp((mean + ((sigma * sigma)/2)));

       return meanNew;
}

// name: lognormalFinalSigma
// inputs: mean - double - log mean of the current component
//                   sigma - double - log stand dev of the current component
// outputs: std - double - standard deviation of a component
// description: calculates the standard deviation of a component which follows
//                          the lognormal distribution
double lognormalFinalSigma(double mean, double sigma)
{
       double stdev;

       stdev = exp(((2*mean)+(sigma*sigma))) * (exp((sigma * sigma)) - 1);

       stdev = sqrt(stdev);

       return stdev;
}

// function name: normal_probability
// inputs: pointerToResCapCalculation - pointer to Component - pointer to
resistor/capacitor Component structure
//                  numRows - int - number of values contained in a components
randomNumberType
//                  mean = double - mean value of all MTBF
//                  sigma - double - standard deviation between the components
MTBF
// Description: goes into the formatted zTable and interpolates a value of z if it
is not found as "nice"
//                          value on the z table
void normal_probability(Component * pointerToResCapCalculation, int numRows,
double mean, double sigma)
{
        double errorFunc;

        // CDF(x) = 0.5 * (1 + erf((x-mu)/(sigma(root(2))))
        for(int q = 0; q < numRows; q++)
        {
                errorFunc = ((pointerToResCapCalculation[q].value - mean) / (sigma * 
sqrt(2.0)));

                pointerToResCapCalculation[q].normalProbability = 0.5 * ( 1 + 
errorFunction(errorFunc));
        }

        return;
}

// function name: exponential_probability
// inputs: pointerToResCapCalculation - pointer to Component - pointer to 
resistor/capacitor Component structure
//                      NumberResCapLines - int - number of lines containing 
components that are not chips
//                      totalResistors - int - number of components in the PCB
//                      mean = double - mean value of all MTBF
// Description: finds the exponential probabilities with the exponential 
cumulative distribution function
void exponential_probability(Component * pointerToResCapCalculation, int numRows, 
double mean)
{
        double exponent;

        // finds the exponential probability F(x)
        for(int q = 0; q < numRows; q++)
        {
                exponent = -(pointerToResCapCalculation[q].value/mean);

                pointerToResCapCalculation[q].exponentialProbability = 1 - 
exp(exponent);
        }

        return;
}

// function name: lognormal_probability
// inputs: pointerToResCapCalculation - pointer to Component - pointer to 
resistor/capacitor Component structure
//                      numRows = int = number of values contained in a components 
vector type
//                      logMean = double - mean log value of all values
//                      logSigma - double - standard deviation of the log between the 
components values
// Description: finds the lognormal CDF probabilities through calculations
void lognormal_probability(Component * pointerToResCapCalculation, int numRows, 
int n, double logMean, double logSigma)
{
        double errorFunc;

        // calculate the lognormal probability of each function using boost 
libraries for error function calculation
        for(int q = 0; q < numRows; q++)
```

```
	{
		errorFunc = ((log(pointerToResCapCalculation[q].value) - logMean) / 
(logSigma * sqrt(2.0)));

		pointerToResCapCalculation[q].logNormalProbability = 0.5 * ( 1 + 
errorFunction(errorFunc));
	}

	return;
}

// function name: weibull_probability
// inputs: pointerToResCapCalculation - pointer to Component - pointer to 
resistor/capacitor Component structure
//			numRows - int - number of values contained within a specific 
components vector
//			betas = double = shape parameter
//			etas = double = scale parameter
// Description: Finds the CDF of each component using the weibull distribution
void weibull_probability(Component * pointerToResCapCalculation, int numRows, 
double betas, double etas)
{
	double power;

	//CDF(x) = 1 - exp(-(x/eta)^beta)
	for( int q = 0; q < numRows; q++)
	{
		power = pointerToResCapCalculation[q].value/etas;
		power = pow(power,betas);
		power = -power;
		pointerToResCapCalculation[q].weibullProbability = 1.0000 - 
(exp(power));
	}

	return;
}

// function name: errorFunction
// inputs: x - double - value to be calculated within the error function 
parameters
// outputs: function - double - value of x calculated as an error funciton value
// description: calculates a value in terms of the error function
double errorFunction(double x)
{
	/* erf(z) = 2/sqrt(pi) * Integral(0..x) exp( -t^2) dt
	erf(0.01) = 0.0112834772 erf(3.7) = 0.9999998325
	Abramowitz/Stegun: p299, |erf(z)-erf| <= 1.5*10^(-7)
	*/
	double a1 = 0.254829592;
	double a2 = -0.284496736;
	double a3 = 1.421413741;
	double a4 = -1.453152027;
	double a5 = 1.061405429;
	double p = 0.3275911;

	int sign = 1;

	if(x < 0)
```

```
        {
                sign = -1;
        }

        x = fabs(x);

        double t = 1.0/(1.0 + p * x);
        double y = 1.0 - (((((a5 * t +a4) * t) + a3)*t + a2)*t + a1)*t*exp(-x*x);

        y = sign * y;

        return y;
}

void initialMeans(statisticsVectors * pointerToResCapStatsIntegrated, Component *
pointerToResCapInput, int numCols)
{
        double mean;
        double sigma;
        double logMean;
        double logSigma;
        double eta;
        double beta;
        int numRows;

        for(int type = 0; type < 3; type++)
        {
                for(int col = 0; col < numCols; col++)
                {
                        Component * pointerToResCapCalculation = new
Component[pointerToResCapInput[col].inputMTTF.size()];;

                        if(type == 0)
                        {
                                numRows = pointerToResCapInput[col].inputMTTF.size();

                                for(int comp = 0; comp < numRows; comp++)
                                {
                                        pointerToResCapCalculation[comp].value =
pointerToResCapInput[col].inputMTTF[comp];
                                }
                        }

                        else if(type == 1)
                        {
                                numRows = pointerToResCapInput[col].inputMTTR.size();

                                for(int comp = 0; comp < numRows; comp++)
                                {
                                        pointerToResCapCalculation[comp].value =
pointerToResCapInput[col].inputMTTR[comp];
                                }
                        }

                        else if(type == 2)
                        {
                                numRows =
pointerToResCapInput[col].inputMTTReplace.size();
```

```
				for(int comp = 0; comp < numRows; comp++)
				{
					pointerToResCapCalculation[comp].value = 
pointerToResCapInput[col].inputMTTReplace[comp];
				}
			}

			if(pointerToResCapInput[col].distributionTypeHoldTBF == 1 ||
				pointerToResCapInput[col].distributionTypeHoldRepair == 
1 ||
				pointerToResCapInput[col].distributionTypeHoldReplace 
== 1)
			{
				if(pointerToResCapInput[col].distributionTypeHoldTBF == 
1)
				{
					// find mean
					pointerToResCapInput[col].MTTF = 
MEAN(pointerToResCapCalculation, numRows);
					// find standard deviation
					pointerToResCapInput[col].sigma = 
STDDEV(pointerToResCapCalculation, numRows, pointerToResCapInput[col].MTTF);
				}

				else 
if(pointerToResCapInput[col].distributionTypeHoldRepair == 1)
				{
					// find mean
					pointerToResCapInput[col].inRepair = 
MEAN(pointerToResCapCalculation, numRows);
				}

				else 
if(pointerToResCapInput[col].distributionTypeHoldReplace == 1)
				{
					// find mean
					pointerToResCapInput[col].inReplace = 
MEAN(pointerToResCapCalculation, numRows);
				}
			}

			else if(pointerToResCapInput[col].distributionTypeHoldTBF == 2 
||
				pointerToResCapInput[col].distributionTypeHoldRepair == 
2 ||
				pointerToResCapInput[col].distributionTypeHoldReplace 
== 2)
			{
				mean = MEAN(pointerToResCapCalculation, numRows);

				// find standard deviation
				sigma = STDDEV(pointerToResCapCalculation, numRows, 
mean);

				// solve for beta
				beta = betaCalc(pointerToResCapCalculation, numRows, 
mean, sigma);
```

```
                            // solve for eta
                            eta = etaCalc(pointerToResCapCalculation, numRows,
beta, mean);

                            if(pointerToResCapInput[col].distributionTypeHoldTBF ==
2)
                            {
                                   pointerToResCapInput[col].MTTF =
exponentialMean(eta);

                                   pointerToResCapInput[col].sigma =
exponentialSigma(eta);
                            }

                            else
if(pointerToResCapInput[col].distributionTypeHoldRepair == 2)
                            {
                                   // find mean
                                   pointerToResCapInput[col].inRepair =
exponentialMean(eta);
                            }

                            else
if(pointerToResCapInput[col].distributionTypeHoldReplace == 2)
                            {
                                   // find mean
                                   pointerToResCapInput[col].inReplace =
exponentialMean(eta);
                            }
                     }

                     else if(pointerToResCapInput[col].distributionTypeHoldTBF == 3
||
                            pointerToResCapInput[col].distributionTypeHoldRepair ==
3 ||
                            pointerToResCapInput[col].distributionTypeHoldReplace
== 3)
                     {
                            // find the log mean
                            logMean = log_mean(pointerToResCapCalculation,
numRows);

                            // find the standard deviation of the logs
                            logSigma = log_sigma(pointerToResCapCalculation,
numRows, logMean);

                            if(pointerToResCapInput[col].distributionTypeHoldTBF ==
3)
                            {
                                   pointerToResCapInput[col].MTTF =
lognormalFinalMean(logMean, logSigma);
                                   // find standard deviation
                                   pointerToResCapInput[col].sigma =
lognormalFinalSigma(logMean, logSigma);
                            }
                            else
if(pointerToResCapInput[col].distributionTypeHoldRepair == 3)
                            {
                                   // find mean
                                   pointerToResCapInput[col].inRepair =
lognormalFinalMean(logMean, logSigma);
                            }

                            else
if(pointerToResCapInput[col].distributionTypeHoldReplace == 3)
                            {
                                   // find mean
                                   pointerToResCapInput[col].inReplace =
lognormalFinalMean(logMean, logSigma);
                            }
                     }

                     else if(pointerToResCapInput[col].distributionTypeHoldTBF == 4
||
                            pointerToResCapInput[col].distributionTypeHoldRepair ==
4 ||
                            pointerToResCapInput[col].distributionTypeHoldReplace
== 4)
                     {
                            // find mean
                            mean = MEAN(pointerToResCapCalculation, numRows);

                            // find standard deviation
                            sigma = STDDEV(pointerToResCapCalculation, numRows,
mean);

                            // solve for beta
                            beta = betaCalc(pointerToResCapCalculation, numRows,
mean, sigma);

                            // solve for eta
                            eta = etaCalc(pointerToResCapCalculation, numRows,
beta, mean);

                            if(pointerToResCapInput[col].distributionTypeHoldTBF ==
4)
                            {
                                   pointerToResCapInput[col].MTTF =
weibullMean(beta, eta);

                                   pointerToResCapInput[col].sigma =
weibullSigma(beta, eta);
                            }

                            else
if(pointerToResCapInput[col].distributionTypeHoldRepair == 4)
                            {
                                   // find mean
                                   pointerToResCapInput[col].inRepair =
weibullMean(beta, eta);
                            }
                            else
if(pointerToResCapInput[col].distributionTypeHoldReplace == 4)
                            {
                                   // find mean
                                   pointerToResCapInput[col].inReplace =
weibullMean(beta, eta);
                            }
                     }

                     delete [] pointerToResCapCalculation;
              }
       }

       return;
}

// function name: find_distances
// inputs: pointerToResCapCalculation - pointer to Component - pointer to
resistor/capacitor Component structure
//                   numRows - int - number of lines containing components that are
not chips
//                   mean = double = mean of the values in a components
randomNumberType
//                   sigma = double = standard deviation of the values in a
components randomNumberType
//                   beta = double = beta value for weibull calculation
//                   eta = double = eta value for weibull calculation
//                   logMean = double = mean of the logs
//                   logSigma = double = standard deviation of the logs
//                   componentPosition = int = position of the component in the
array
//                   randomNumberType = int = type of random number (TBF, TTR,
replace) the function should be retrieving
//                   generateNumber = bool = determines when it runs through this
function if it should also generate a probability as well as the CDF
// outputs: value - double - random value that was generated
// Description: Finds the distance for all components of types of distributions,
then goes into the lowest
//                          distance function, and returns the lowest distance
double find_distances(Component * pointerToResCapCalculation, statisticsVectors *
pointerToResCapStats, statisticsVectors * pointerToResCapStatsIntegrated, int
numRows, double mean, double sigma, double expoMean,
       double beta, double eta, double logMean, double logSigma, double
logNormSigma, double logNormMean, int componentPosition, int randomNumberType,
bool generateNumber)
{
       double normalDistance;
       double exponentialDistance;
       double logNormalDistance;
       double weibullDistance;
       double value;

       // find distances D = |Fo - Fn|
       for(int q = 0; q < numRows; q++)
       {
              // Distance = | Theoretical Value - Empirical Value |
              normalDistance = pointerToResCapCalculation[q].CDF -
pointerToResCapCalculation[q].normalProbability;
```

```
              exponentialDistance = pointerToResCapCalculation[q].CDF - 
pointerToResCapCalculation[q].exponentialProbability;
              logNormalDistance = pointerToResCapCalculation[q].CDF - 
pointerToResCapCalculation[q].logNormalProbability;
              weibullDistance = pointerToResCapCalculation[q].CDF - 
pointerToResCapCalculation[q].weibullProbability;

              // take the absolute value of the calculated distance
              pointerToResCapCalculation[q].normalDistance = fabs(normalDistance);
              pointerToResCapCalculation[q].exponentialDistance = 
fabs(exponentialDistance);
              pointerToResCapCalculation[q].logNormalDistance = 
fabs(logNormalDistance);
              pointerToResCapCalculation[q].WeibullDistance = 
fabs(weibullDistance);
       }

       // go to lowest distance function to find a random number based on the best 
fit distribution
       value = find_lowest_distance(pointerToResCapCalculation, 
pointerToResCapStats, pointerToResCapStatsIntegrated, numRows, sigma, mean, 
expoMean, beta, eta, logMean, logSigma, logNormSigma, logNormMean, 
componentPosition, randomNumberType, generateNumber);

       // if a new number should be generated, return that value
       if(generateNumber)
       {
              return value;
       }

       // otherwise return 0
       else
       {
              return 0;
       }
}

// function name: find_lowest_distance
// inputs: pointerToResCapCalculation - pointer to Component - pointer to 
resistor/capacitor Component structure
//                   pointerToResCapStats = pointer to statisticsVectors = saves 
all values into the table
//                   numRows - int - number of lines containing components that are 
not chips
//                   mean = double = mean of the values in a components 
randomNumberType
//                   sigma = double = standard deviation of the values in a 
components randomNumberType
//                   beta = double = beta value for weibull calculation
//                   eta = double = eta value for weibull calculation
//                   logMean = double = mean of the logs
//                   logSigma = double = standard deviation of the logs
//                   componentPosition = int = position of the component in the 
array
//                   randomNumberType = int = type of random number (TBF, TTR, 
replace) the function should be retrieving
//                   generateNumber = bool = determines when it runs through this 
function if it should also generate a probability as well as the CDF
```

```
// outputs: value - double - random value that was generated
// Description: finds the highest distance of each distribution type, then finds
the lowest of those four distances, then
//                          the lowest distance will be the distribution that, taht
component type will follow throughout the lifetime of the
//                          replication, then will generate a random value based on
the distribution type
double find_lowest_distance(Component * pointerToResCapCalculation,
statisticsVectors * pointerToResCapStats, statisticsVectors *
pointerToResCapStatsIntegrated, int numRows, double sigma,
       double mean, double expoMean, double beta, double eta, double logMean,
double logSigma, double logNormSigma, double logNormMean, int componentPosition,
int randomNumberType, bool generateNumber)
{
       double highestNormalDistance =
pointerToResCapCalculation[0].normalDistance;
       double highestExponentialDistance =
pointerToResCapCalculation[0].exponentialDistance;
       double highestLogNormalDistance =
pointerToResCapCalculation[0].logNormalDistance;
       double highestWeibullDistance =
pointerToResCapCalculation[0].WeibullDistance;
       double lowest_distance;
       int distributionType = 0;
       double normalCV;
       double expCV;
       double logCV;
       double weibullCV;
       double numRow = numRows;
       double deleteMe = 0;

       // finds the highest normal distance
       for(int q = 1; q < numRows; q++)
       {
              if(pointerToResCapCalculation[q].normalDistance >
highestNormalDistance)
              {
                     highestNormalDistance =
pointerToResCapCalculation[q].normalDistance;
              }
       }

       // finds the highest exponential distance
       for(int q = 1; q < numRows; q++)
       {
              if(pointerToResCapCalculation[q].exponentialDistance >
highestExponentialDistance)
              {
                     highestExponentialDistance =
pointerToResCapCalculation[q].exponentialDistance;
              }
       }

       // finds the highest log normal distance
       for(int q = 1; q < numRows; q++)
       {
              if(pointerToResCapCalculation[q].logNormalDistance >
highestLogNormalDistance)
```

```
			{
				highestLogNormalDistance = 
pointerToResCapCalculation[q].logNormalDistance;
			}
		}

		// finds the highest weibull distance
		for(int q = 1; q < numRows; q++)
		{
			if(pointerToResCapCalculation[q].WeibullDistance > 
highestWeibullDistance)
			{
				highestWeibullDistance = 
pointerToResCapCalculation[q].WeibullDistance;
			}
		}

		// calculate the critical value of each distribution type (they will all 
equal; done simply for readibility)
		normalCV = 1.358 / (sqrt(numRow));
		expCV = 1.358 / (sqrt(numRow));
		logCV = 1.358 / (sqrt(numRow));
		weibullCV = 1.358 / (sqrt(numRow));

		// if the lowest of these four values in normal
		if(highestNormalDistance < highestExponentialDistance &&
			highestNormalDistance < highestLogNormalDistance &&
			highestNormalDistance < highestWeibullDistance)
		{
			// set the lowest normal distance to the lowest distance
			lowest_distance = highestNormalDistance;

			// set distributionType to normal
			distributionType = 1;

			// save the distribution type for that components TBF, TTR, or 
replace
			chooseDistribution(pointerToResCapCalculation, randomNumberType, 
distributionType);

			// generate a new number of the program finds it should
			if(generateNumber)
			{
				return randomNumberGenerator(pointerToResCapCalculation, 
pointerToResCapStats, pointerToResCapStatsIntegrated, numRows, 0, 1, 
distributionType, mean, sigma, expoMean, beta, eta, logMean, logSigma, 
logNormSigma, logNormMean, componentPosition, generateNumber);
			}
		}

		// if the exponential distance is lower thna the other distances
		else if( highestExponentialDistance < highestNormalDistance &&
			highestExponentialDistance < highestLogNormalDistance &&
			highestExponentialDistance < highestWeibullDistance)
		{
			// set the lowest distance
			lowest_distance = highestExponentialDistance;
```

```
              // save exponential as the best fit distribution
              distributionType = 2;

              // save that distribution type to taht components TBF, TTR< or 
replace
              chooseDistribution(pointerToResCapCalculation, randomNumberType, 
distributionType);

              // if a new number should be generated, do it based off the 
exponential calculations
              if(generateNumber)
              {
                     return randomNumberGenerator(pointerToResCapCalculation, 
pointerToResCapStats, pointerToResCapStatsIntegrated, numRows, 0, 1, 
distributionType, mean, sigma, expoMean, beta, eta, logMean, logSigma, 
logNormSigma, logNormMean, componentPosition, generateNumber);
              }
       }

       // if the lognormal distance is lower than the other 3 distances
       else if(highestLogNormalDistance < highestNormalDistance &&
              highestLogNormalDistance < highestExponentialDistance &&
              highestLogNormalDistance < highestWeibullDistance)
       {
              // sets the lowest log normal distance to the lowest distance
              lowest_distance = highestLogNormalDistance;

              // set the distributionType to lognormal
              distributionType = 3;

              // save the distribution type to that components TBF, TTR, or 
replace
              chooseDistribution(pointerToResCapCalculation, randomNumberType, 
distributionType);

              // generate a new value based on the lognormal distribution
              if(generateNumber)
              {
                     return randomNumberGenerator(pointerToResCapCalculation, 
pointerToResCapStats, pointerToResCapStatsIntegrated, numRows, 0, 1, 
distributionType, mean, sigma, expoMean, beta, eta, logMean, logSigma, 
logNormSigma, logNormMean, componentPosition, generateNumber);
              }
       }

       // if weibull distance is less than the other 3 distances
       else if(highestWeibullDistance < highestNormalDistance &&
              highestWeibullDistance < highestExponentialDistance &&
              highestWeibullDistance < highestLogNormalDistance)
       {
              // set the lowest weibull distance to the lowest distance
              lowest_distance = highestWeibullDistance;

              // set distributionType to weibull
              distributionType = 4;

              // save the distribution type for that components TBF, TTR, or 
replace
```

```
		chooseDistribution(pointerToResCapCalculation, randomNumberType, 
distributionType);

		// generate a new number based on the weibull distribution
		if(generateNumber)
		{
			return randomNumberGenerator(pointerToResCapCalculation, 
pointerToResCapStats, pointerToResCapStatsIntegrated, numRows, 0, 1, 
distributionType, mean, sigma, expoMean, beta, eta, logMean, logSigma, 
logNormSigma, logNormMean, componentPosition, generateNumber);
		}
	}

	// if a new number was supposed to be generated, return that new value to 
the main

	return 0;
}

// name: chooseDistribution
// inputs: pointerToResCapCalculation = Component = decides which distribution 
type a component should always take
//			randomNumberType = int = descriptor determinant of whether it 
should be saving the TBF distribution, TTR distribution or replace distribution
//			distributionType = int = descriptor for which type of 
distribution the component should always follow
// Description: finds the initial distribution type that the component will follow 
from just the sample data, and finds
//				the best distribution type, then saves that 
distribution type to the component, so the component will
//				always follow the same distribution type throughout the 
replication.
void chooseDistribution(Component * pointerToResCapCalculation, int 
randomNumberType, int distributionType)
{
	// if TBF
	if(randomNumberType == 0)
	{
		// save distribution type for that component
		pointerToResCapCalculation[0].distributionTypeHoldTBF = 
distributionType;
	}

	// if TTR
	else if(randomNumberType == 1)
	{
		// save distribution type for that component
		pointerToResCapCalculation[0].distributionTypeHoldRepair = 
distributionType;
	}

	// if replace
	else
	{
		// save distribution type for that component
		pointerToResCapCalculation[0].distributionTypeHoldReplace = 
distributionType;
	}
```

```
}

// name: randomNumberGenerator
// inputs: pointerToResCapCalculation = Component = does the calculations
//                    pointerToResCapStats = statisticsVector = contains all values
for tbf, repair, and replace
//                    numRows = int = number of rows in ttf file
//                    min = double = minimum possible random number
//                    max = double = maximum possible random number
//                    distributionType = int = type of distribution to solve for
//                    mean = double = mean of components
//                    sigma = double = standard deviation between values of a
component
//                    beta = double = beta value for weibull distribution
//                    eta = double = eta value for weibull distribution
//                    logMean = double = mean of the log values for each component
//                    logSigma = double = standard deviation of the log values for
each component
//                    componentPosition = int = location of component within the
array
// outputs: the final randomly generated variable
// description: determines (based on lowest distance) the type of distribution to
use in order to calculate a time
//                           out of a given randomly generated probability within a
program specfied range (0 to 1)
double randomNumberGenerator(Component * pointerToResCapCalculation,
statisticsVectors * pointerToResCapStats, statisticsVectors *
pointerToResCapStatsIntegrated,
       int numRows, double min, double max, int distributionType, double mean,
double sigma, double expoMean, double beta, double eta, double logMean,
       double logSigma, double logNormSigma, double logNormMean, int
componentPosition, bool generateNumber)
{
       double randomNumber = 2;
       int normalType = 0;
       double erf_Y;
       double FRR;

       if(pointerToResCapStatsIntegrated[componentPosition].distributionFound)
       {
              distributionType =
pointerToResCapStatsIntegrated[componentPosition].distribution;
       }

       while(randomNumber >= 1)
       {
              randomNumber = (double)rand() / RAND_MAX;
       }

       switch(distributionType)
       {
              //distributionType = normal
       case 1:
              normalType = 1;

              // if probability is less than 0.5
              if(randomNumber < 0.5)
              {
                     erf_Y = (1 - randomNumber * 2);
              }

              // otherwise
              else
              {
                     erf_Y = (randomNumber * 2 - 1);
              }

              // if value is in the range call erf function
              if(erf_Y >= 0 &&
                     erf_Y <= 1)
              {
                     FRR = find_erf(mean, sigma, erf_Y, normalType);
              }

              // return value to main
              return FRR;

              break;

              //distributionType = exponential
       case 2:
              // calculate new value based on exponential
              FRR = -(expoMean*log((1-randomNumber)));

              // return value to main
              return FRR;

              break;

              //distributionType = lognormal
       case 3:
              // if random number is less than 0.5
              if(randomNumber < 0.5)
              {
                     erf_Y = (1 - randomNumber * 2);
              }

              // otherwise
              else
              {
                     erf_Y = (randomNumber * 2 - 1);
              }

              //lognormal
              normalType = 2;

              // if random value is in the range
              if(erf_Y >= 0 &&
                     erf_Y <= 1)
              {
                     // call find_erf and find value
                     FRR = find_erf(logNormMean, logNormSigma, erf_Y, normalType);
              }
              // return value to main
              return FRR;
```

```
              break;

              //distributionType = weibull
       case 4:
              // calculate value based on weibull
              FRR = NRoot((log((1/(1-randomNumber)))),beta);
              FRR = eta * FRR;

              // return to main
              return FRR;

              break;

              // should never happen
       default:
              FRR = 0;

              return FRR;
              break;
       }
}

//name: NRoot
// inputs: num = double = value being rooted
//                   root = double = value doing the rooting
// outputs: the root of num
// description: solves the value being rooted, when its not a square root
double NRoot(double num, double root)
{
       root=1/root;//divide by 1 to get 1/root e.g 1/2=0.5
       return(pow(num, root));//raise num to root to get answer
}

// name: find_erf
// inputs: mean = double = mean of all components
//                   sigma = double = standard deviation of all compeonts
//                   randomProb = double = value contained within the error
funciton
//                   normalType = int = normal or lognormal
// outputs: randomValue = double = calculated value of x
// decription: runs through the formatted ERF table and finds the closest value to
it
double find_erf(double mean, double sigma, double random_F_Y, int normalType)
{
       fstream ERFTable;
       string currentLine;
       double randomValue;

       random_F_Y = errorFunction(random_F_Y);

       // if randomProb has found the area in which it is supposed to be placed
give it a value and solve;
       if(normalType == 1)
       {
              randomValue = random_F_Y*sigma*sqrt(2.0) + mean;
       }
```

```
	// if randomProb has found the area in which it is supposed to be placed 
give it a value and solve;
	else if(normalType == 2)
	{
		randomValue = random_F_Y*sigma*sqrt(2.0) + mean;
	}

	// return calculated value
	return randomValue;
}

// mame: findNewValue
// inputs: pointerToResCapLife = component = contains parameters for failing and 
replacing/repairing components
//			pointerToResCapStatsIntegrated = statisticaVector = contains 
all new values for every replicaiton
//			pointerToResCapStats = statisticsVector = contains all new 
values for that specific replication
//			numCols - int = number of columns in TTF file
//			componentNumberNewValue = int = component recieving a new 
value
//			randomNumberType = int = type of random number to generate 
(TTF, repair, replace)
//			currentR`


//			generateNumber = bool = determines if a number should be 
generated
//description: generates a new time to fail, replace, or repair, depending upon 
the fail, or replace/repair type
void findNewValue(Component * pointerToResCapLife, statisticsVectors * 
pointerToResCapStatsIntegrated, int numCols,
	statisticsVectors * pointerToResCapStats, int componentNumberNewValue, int 
randomNumberType, int currentRowLength, bool generateNumber, vector<thermal> & 
thermalComponent,
	Component * pointerToResCapInput)
{
	Component * pointerToResCapCalculation = new Component[currentRowLength];

	double mean;
	double sigma;
	double logMean;
	double logSigma;
	double eta;
	double beta;
	double expoMean;
	double expoSigma;
	double logNormMean;
	double logNormSigma;
	double wiebullMean;
	double wiebullSigma;

	// if a new number should be generated
	for(int s = 0; s < currentRowLength; s++)
	{
		if(generateNumber)
		{
			// set component number
```

```
                    pointerToResCapCalculation[s].componentNumber = 
pointerToResCapStats[componentNumberNewValue].componentNumber;
              }
       }

       // if a new number should be generated
       if(generateNumber)
       {
              // if TBF
              if(randomNumberType == 0)
              {
                     // set to initially be less than 0 to enter loop
                     pointerToResCapLife[componentNumberNewValue].TBF = -1;

                     // TBF must be greater than 0
                     while(pointerToResCapLife[componentNumberNewValue].TBF <= 0)
                     {
                            // generate new random number
                            pointerToResCapLife[componentNumberNewValue].TBF = 
randomNumberGenerator(pointerToResCapCalculation, pointerToResCapStats, 
pointerToResCapStatsIntegrated, currentRowLength, 0, 1, 
pointerToResCapInput[componentNumberNewValue].distributionTypeHoldTBF, 
pointerToResCapInput[componentNumberNewValue].MTTF, 
pointerToResCapInput[componentNumberNewValue].sigma, 
pointerToResCapInput[componentNumberNewValue].MTTF, 
pointerToResCapInput[componentNumberNewValue].beta, 
pointerToResCapInput[componentNumberNewValue].eta, 
pointerToResCapInput[componentNumberNewValue].logMTTF, 
pointerToResCapInput[componentNumberNewValue].logSIGMA, 
pointerToResCapInput[componentNumberNewValue].sigma, 
pointerToResCapInput[componentNumberNewValue].MTTF, componentNumberNewValue, 
generateNumber);
                     }

                     // add the new value to the end to the pointerToResCapStats, 
timebetweenfail vector

       pointerToResCapStats[componentNumberNewValue].timeBetweenFail.push_back(poi
nterToResCapLife[componentNumberNewValue].TBF);

                     // add this new value to the end of hte integrated array also

       //pointerToResCapStatsIntegrated[componentNumberNewValue].timeBetweenFail.p
ush_back(pointerToResCapLife[componentNumberNewValue].TBF);


       std::sort(pointerToResCapStats[componentNumberNewValue].timeBetweenFail.rbe
gin(), pointerToResCapStats[componentNumberNewValue].timeBetweenFail.rend(), 
std::greater<double>());

       //std::sort(pointerToResCapStatsIntegrated[componentNumberNewValue].timeBet
weenFail.rbegin(), 
pointerToResCapStatsIntegrated[componentNumberNewValue].timeBetweenFail.rend(), 
std::greater<double>());

                     for(unsigned int rawr = 0; rawr < 
pointerToResCapStats[componentNumberNewValue].timeBetweenFail.size(); rawr++)
                     {
                            if(pointerToResCapLife[componentNumberNewValue].TBF ==
pointerToResCapStats[componentNumberNewValue].timeBetweenFail[rawr])
                            {

       pointerToResCapStats[componentNumberNewValue].TBFrank.push_back(rawr + 1);
                            }
                     }


       std::sort(pointerToResCapStats[componentNumberNewValue].TBFrank.rbegin(),
pointerToResCapStats[componentNumberNewValue].TBFrank.rend(),
std::greater<double>());

                     for(unsigned int rawr = 0; rawr <
pointerToResCapStats[componentNumberNewValue].timeBetweenFail.size(); rawr++)
                     {
                            if(pointerToResCapLife[componentNumberNewValue].TBF <
pointerToResCapStats[componentNumberNewValue].timeBetweenFail[rawr])
                            {

       pointerToResCapStats[componentNumberNewValue].TBFrank[rawr] =
pointerToResCapStats[componentNumberNewValue].TBFrank[rawr]+1;
                            }
                     }

              //     // resize the CDF vector to the same size as the TBF bector

       pointerToResCapStats[componentNumberNewValue].timeBetweenFailCDF.resize(poi
nterToResCapStats[componentNumberNewValue].timeBetweenFail.size());
              }

              // if TTR
              else if(randomNumberType == 1)
              {
                     // set to initially be less than 0 to enter loop
                     pointerToResCapLife[componentNumberNewValue].TTR = -1;

                     // TTR must be greater than 0
                     while(pointerToResCapLife[componentNumberNewValue].TTR < 0)
                     {
                            // generate new random number
                            pointerToResCapLife[componentNumberNewValue].TTR =
randomNumberGenerator(pointerToResCapCalculation, pointerToResCapStats,
pointerToResCapStatsIntegrated, currentRowLength, 0, 1,
pointerToResCapInput[componentNumberNewValue].distributionTypeHoldRepair,
pointerToResCapInput[componentNumberNewValue].inRepair,
pointerToResCapInput[componentNumberNewValue].repairSigma,
pointerToResCapInput[componentNumberNewValue].inRepair,
pointerToResCapInput[componentNumberNewValue].repairBeta,
pointerToResCapInput[componentNumberNewValue].repairEta,
pointerToResCapInput[componentNumberNewValue].inRepair,
pointerToResCapInput[componentNumberNewValue].repairSigma,
pointerToResCapInput[componentNumberNewValue].repairSigma,
pointerToResCapInput[componentNumberNewValue].inRepair, componentNumberNewValue,
generateNumber);

                     }
```

```
                    // add the new value to the end to the pointerToResCapStats,
repair vector
	
	pointerToResCapStats[componentNumberNewValue].repair.push_back(pointerToRes
CapLife[componentNumberNewValue].TTR);

                    // add this new value to the end of hte integrated array also

	//pointerToResCapStatsIntegrated[componentNumberNewValue].repair.push_back(
pointerToResCapLife[componentNumberNewValue].TTR);


	std::sort(pointerToResCapStats[componentNumberNewValue].repair.rbegin(),
pointerToResCapStats[componentNumberNewValue].repair.rend(),
std::greater<double>());

	//std::sort(pointerToResCapStatsIntegrated[componentNumberNewValue].repair.
rbegin(), pointerToResCapStatsIntegrated[componentNumberNewValue].repair.rend(),
std::greater<double>());

                    for(unsigned int rawr = 0; rawr <
pointerToResCapStats[componentNumberNewValue].repair.size(); rawr++)
                    {
                           if(pointerToResCapLife[componentNumberNewValue].TTR ==
pointerToResCapStats[componentNumberNewValue].repair[rawr])
                           {

	pointerToResCapStats[componentNumberNewValue].repairRank.push_back(rawr +
1);
                           }
                    }


	std::sort(pointerToResCapStats[componentNumberNewValue].repairRank.rbegin()
, pointerToResCapStats[componentNumberNewValue].repairRank.rend(),
std::greater<double>());

                    for(unsigned int rawr = 0; rawr <
pointerToResCapStats[componentNumberNewValue].repair.size(); rawr++)
                    {
                           if(pointerToResCapLife[componentNumberNewValue].TTR <
pointerToResCapStats[componentNumberNewValue].repair[rawr])
                           {

	pointerToResCapStats[componentNumberNewValue].repairRank[rawr] =
pointerToResCapStats[componentNumberNewValue].repairRank[rawr]+1;
                           }
                    }

                    // resize the CDF vector to the same size as the TTR bector

	pointerToResCapStats[componentNumberNewValue].repairCDF.resize(pointerToRes
CapStats[componentNumberNewValue].repair.size());
              }

              // if replace
              else if(randomNumberType == 2)
              {
                    // set to initially be less than 0 to enter loop
                    pointerToResCapLife[componentNumberNewValue].replace = -1;

                    // Replace must be greater than 0
                    while(pointerToResCapLife[componentNumberNewValue].replace <
0)
                    {
                          // generate new random number
                          pointerToResCapLife[componentNumberNewValue].replace =
randomNumberGenerator(pointerToResCapCalculation, pointerToResCapStats,
pointerToResCapStatsIntegrated, currentRowLength, 0, 1,
pointerToResCapInput[componentNumberNewValue].distributionTypeHoldReplace,
pointerToResCapInput[componentNumberNewValue].inReplace,
pointerToResCapInput[componentNumberNewValue].replaceSigma,
pointerToResCapInput[componentNumberNewValue].inReplace,
pointerToResCapInput[componentNumberNewValue].replaceBeta,
pointerToResCapInput[componentNumberNewValue].replaceEta,
pointerToResCapInput[componentNumberNewValue].inReplace,
pointerToResCapInput[componentNumberNewValue].replaceSigma,
pointerToResCapInput[componentNumberNewValue].replaceSigma,
pointerToResCapInput[componentNumberNewValue].inReplace, componentNumberNewValue,
generateNumber);
                    }

                    // add the new value to the end to the pointerToResCapStats,
replace vector

      pointerToResCapStats[componentNumberNewValue].replace.push_back(pointerToRe
sCapLife[componentNumberNewValue].replace);

                    // add this new value to the end of hte integrated array also

      //pointerToResCapStatsIntegrated[componentNumberNewValue].replace.push_back
(pointerToResCapLife[componentNumberNewValue].replace);


      std::sort(pointerToResCapStats[componentNumberNewValue].replace.rbegin(),
pointerToResCapStats[componentNumberNewValue].replace.rend(),
std::greater<double>());

      //std::sort(pointerToResCapStatsIntegrated[componentNumberNewValue].replace
.rbegin(), pointerToResCapStatsIntegrated[componentNumberNewValue].replace.rend(),
std::greater<double>());

                    for(unsigned int rawr = 0; rawr <
pointerToResCapStats[componentNumberNewValue].replace.size(); rawr++)
                    {
                          if(pointerToResCapLife[componentNumberNewValue].replace
== pointerToResCapStats[componentNumberNewValue].replace[rawr])
                          {

      pointerToResCapStats[componentNumberNewValue].replaceRank.push_back(rawr +
1);
                          }
                    }


      std::sort(pointerToResCapStats[componentNumberNewValue].replaceRank.rbegin(
), pointerToResCapStats[componentNumberNewValue].replaceRank.rend(),
std::greater<double>());

                    for(unsigned int rawr = 0; rawr <
pointerToResCapStats[componentNumberNewValue].replace.size(); rawr++)
                    {
                            if(pointerToResCapLife[componentNumberNewValue].replace
< pointerToResCapStats[componentNumberNewValue].replace[rawr])
                            {

       pointerToResCapStats[componentNumberNewValue].replaceRank[rawr] =
pointerToResCapStats[componentNumberNewValue].replaceRank[rawr]+1;
                            }
                    }

                    // resize the CDF vector to the same size as the replace
vector

       pointerToResCapStats[componentNumberNewValue].replaceCDF.resize(pointerToRe
sCapStats[componentNumberNewValue].replace.size());
              }
       }

       // delete array for future use in generating new values
       delete []pointerToResCapCalculation;
}

// name: failureRate
// inputs: pointerToResCapStatsIntegrated = statisticsVectors = contains the final
table and values of all replications
//                    numCols = int = number of columns contained within the input
files (number of resistors [components])
// description: finds the failure rate of the final output from all replications
void failureRate(statisticsVectors * pointerToResCapStats, statisticsVectors *
pointerToResCapStatsIntegrated, Component * pointerToResCapInput, int numCols,
double runLength)
{
       // find lower and upper range of mean with half width
       for(int col = 0; col < numCols; col++)
       {
              pointerToResCapStatsIntegrated[col].failureRate =
1/pointerToResCapInput[col].MTTF;
       }

       return;
}

// name: componentFailProb
// inputs: pointerToResCapStats = statisticsVectors = contains data from a single
replication or all replications;
//                    numCols = int = number of columns (components) in input files
// Description: Caculates the lifetime failure of a component by dividing the
number of times a part has failed
//                           by the total number of failures in the current
replication, or all replications
void componentFailProb(statisticsVectors * pointerToResCapStats, int numCols)
{
       double sumTotalFailures = 0;
```

```
        // summation of the times failed for all components
        for(int col = 0; col < numCols; col++)
        {
                sumTotalFailures += pointerToResCapStats[col].timesFailed;
        }

        // calculate oomponents probability of failure
        for(int col = 0; col < numCols; col++)
        {
                pointerToResCapStats[col].componentFailProb = 
pointerToResCapStats[col].timesFailed/sumTotalFailures;
        }

        return;
}

// Name: MTTF_total
// inputs: pointerToResCapStats - statisticsVectors = contains the MTTF of each 
component
//                   numCols - int - number of columns or components
//                   replicaitons - int - current replication
//                   numberOfReplications - int - total number of replications
// Description: adds up all the MTTF values from each replication and then divides 
that by the total
//                           number of replications
void MTTF_total(statisticsVectors * pointerToResCapStats, statisticsVectors * 
pointerToResCapStatsIntegrated, Component * pointerToResCapLife,
        int numCols, int replications, int numberOfReplications, int 
boolManualReplace, int boolManualRepair, int boolManualTTF, double manualTTR,
        double manualReplace)
{
        for(int col = 0; col < numCols; col++)
        {
                if(col != manualComponentSave)
                {
                        pointerToResCapStatsIntegrated[col].TTR_total = 0;
                        pointerToResCapStatsIntegrated[col].replaceTotal = 0;
                        pointerToResCapStatsIntegrated[col].MTTF = 0;

                        for(unsigned int row = 0; row < 
pointerToResCapStatsIntegrated[col].repair.size(); row++)
                        {
                                pointerToResCapStatsIntegrated[col].TTR_total += 
pointerToResCapStatsIntegrated[col].repair[row];
                        }

                        for(unsigned int row = 0; row < 
pointerToResCapStatsIntegrated[col].replace.size(); row++)
                        {
                                pointerToResCapStatsIntegrated[col].replaceTotal += 
pointerToResCapStatsIntegrated[col].replace[row];
                        }
                }

                else
                {
                        if(boolManualRepair == 1)
                        {
                                pointerToResCapStatsIntegrated[col].TTR_total = 
manualTTR;

                                if(boolManualReplace == 0)
                                {
                                        pointerToResCapStatsIntegrated[col].replaceTotal 
= DBL_MAX;
                                }
                        }

                        if(boolManualReplace == 1)
                        {
                                pointerToResCapStatsIntegrated[col].replaceTotal = 
manualReplace;

                                if(boolManualRepair == 0)
                                {
                                        pointerToResCapStatsIntegrated[col].TTR_total = 
DBL_MAX;
                                }
                        }
                }
        }

        for(int col = 0; col < numCols; col++)
        {
                if(col != manualComponentSave)
                {
                        pointerToResCapStatsIntegrated[col].TTR_total = 
pointerToResCapStatsIntegrated[col].TTR_total / 
pointerToResCapStatsIntegrated[col].repair.size();
                        pointerToResCapStatsIntegrated[col].replaceTotal = 
pointerToResCapStatsIntegrated[col].replaceTotal / 
pointerToResCapStatsIntegrated[col].replace.size();
                }
        }

        for(int comp = 0; comp < numCols; comp++)
        {
                if((comp != manualComponentSave &&
                        boolManualTTF == 1) ||
                        boolManualTTF == 0)
                {
                        for(unsigned int comp1 = 0; comp1 < 
pointerToResCapStatsIntegrated[comp].replicationMTTF.size(); comp1++)
                        {
                                pointerToResCapStatsIntegrated[comp].MTTF += 
pointerToResCapStatsIntegrated[comp].replicationMTTF[comp1];
                        }

                        pointerToResCapStatsIntegrated[comp].MTTF = 
pointerToResCapStatsIntegrated[comp].MTTF / numberOfReplications;
                }
        }

        return;
}
```

```
// name: totalFails
// inputs: failTable - failing struct - contains all fails to be outputed
//                  manualTTF - double - TTF entered by user
//                  manualTTR - double - TTR entered by the user
//                  manualReplace - double - Replace entered by the user
//                  timeInc - double - duration by which the time should be
incremented
//                  runLength = double - length of the simulation
//                  pointerToResCapStats - statisticsVectors - contains all MTTF
values for each component
//                  numCols - int - number of components within the PCB
//                  boolManualRepair - int - determines whether a repair value was
entered manually
//                  boolManualReplace, int - determines whether a replace value
was entered manually
//                  boolManualTTF - int - determines whether a TTF value was
entered manually
// Output: failTable - failing - contains all information for the system level
failures
// Description: finds all the fails that will happen throughout the 50 year
lifetime of a PCB and places them in a neaty nice table
failing totalFails(failing failTable, double manualTTF, double manualTTR, double
manualReplace, double timeInc, double runLength, statisticsVectors *
pointerToResCapStatsIntegrated,
       int numCols, int boolManualRepair, int boolManualReplace, int
boolManualTTF, int replication, vector <thermal> & thermalComponent, Component *
pointerToResCapLife,
       Component * pointerToResCapInput)
{
       double lowestFailedTBF = DBL_MAX;
       int componentsFailed = 0;
       vector <double> lowestTTF;
       int failedPosition = 0;
       double time;
       failTable.partsFailed = 0;
       int counter = 0;

       for(int comp = 0; comp < numCols; comp++)
       {
              for(unsigned int col = 0; col < thermalComponent.size(); col++)
              {
                     if(pointerToResCapStatsIntegrated[comp].partID ==
thermalComponent[col].partID)
                     {
                            pointerToResCapInput[comp].maxTTF =
pointerToResCapInput[comp].maxTTF / thermalComponent[col].epsillon;
                     }
              }

              pointerToResCapStatsIntegrated[comp].mttfSaveSave =
pointerToResCapStatsIntegrated[comp].MTTF;
              pointerToResCapStatsIntegrated[comp].mttrSaveSave =
pointerToResCapStatsIntegrated[comp].TTR_total;
              pointerToResCapStatsIntegrated[comp].replaceSaveSave =
pointerToResCapStatsIntegrated[comp].replaceTotal;

              if(pointerToResCapStatsIntegrated[comp].age != 0)
```

```
		{
			pointerToResCapStatsIntegrated[comp].MTTF = 
pointerToResCapInput[comp].maxTTF;
		}

		else if(pointerToResCapStatsIntegrated[comp].age == 0 && 
			((boolManualTTF == 1 && comp != manualComponentSave) || 
			boolManualTTF == 0))
		{
			findNewValue(pointerToResCapLife, 
pointerToResCapStatsIntegrated, numCols, pointerToResCapStatsIntegrated, comp, 0, 
pointerToResCapStatsIntegrated[comp].timeBetweenFail.size(), true, 
thermalComponent, pointerToResCapInput);
			pointerToResCapStatsIntegrated[comp].MTTF = 
pointerToResCapLife[comp].TBF;
		}

		else if(boolManualTTF == 1 && comp == manualComponentSave)
		{
			pointerToResCapStatsIntegrated[comp].MTTF = manualTTF;
		}

		if(boolManualRepair != 1 || 
			(boolManualRepair == 1 && 
			comp != manualComponentSave))
		{
			findNewValue(pointerToResCapLife, 
pointerToResCapStatsIntegrated, numCols, pointerToResCapStatsIntegrated, comp,1, 
pointerToResCapStatsIntegrated[comp].repair.size(), true, thermalComponent, 
pointerToResCapInput);
			pointerToResCapStatsIntegrated[comp].TTR_total = 
pointerToResCapLife[comp].TTR;
		}

		else if(boolManualRepair == 1 && 
			comp == manualComponentSave)
		{
			pointerToResCapStatsIntegrated[comp].TTR_total = manualTTR;
		}

		if(boolManualReplace != 1 || 
			(boolManualReplace == 1 && 
			comp != manualComponentSave))
		{
			findNewValue(pointerToResCapLife, 
pointerToResCapStatsIntegrated, numCols, pointerToResCapStatsIntegrated, comp,2, 
pointerToResCapStatsIntegrated[comp].replace.size(), true, thermalComponent, 
pointerToResCapInput);
			pointerToResCapStatsIntegrated[comp].replaceTotal = 
pointerToResCapLife[comp].replace;
		}

		else if(boolManualReplace == 1 && 
			comp == manualComponentSave)
		{
			pointerToResCapStatsIntegrated[comp].TTR_total = 
manualReplace;
		}
```

```
              pointerToResCapStatsIntegrated[comp].FAIL = false;
              pointerToResCapStatsIntegrated[comp].replacement = false;
              pointerToResCapStatsIntegrated[comp].repairing = false;

              pointerToResCapStatsIntegrated[comp].MTTFsave =
pointerToResCapStatsIntegrated[comp].MTTF;

              pointerToResCapStatsIntegrated[comp].TTR_save =
pointerToResCapStatsIntegrated[comp].TTR_total;
              pointerToResCapStatsIntegrated[comp].TTReplace_save =
pointerToResCapStatsIntegrated[comp].replaceTotal;

              pointerToResCapStatsIntegrated[comp].MTTF -=
pointerToResCapStatsIntegrated[comp].age;

              if(pointerToResCapStatsIntegrated[comp].MTTF <= 0)
              {
                     findNewValue(pointerToResCapLife,
pointerToResCapStatsIntegrated, numCols, pointerToResCapStatsIntegrated, comp, 0,
pointerToResCapStatsIntegrated[comp].timeBetweenFail.size(), true,
thermalComponent, pointerToResCapInput);
                     pointerToResCapStatsIntegrated[comp].MTTF =
pointerToResCapLife[comp].TBF;
              }

              else if(pointerToResCapStatsIntegrated[comp].MTTF >= 0 &&
                     pointerToResCapStatsIntegrated[comp].age == 0 &&
                     (boolManualTTF == 0 ||
                     (boolManualTTF == 1 &&
                     comp != manualComponentSave)))
              {
              //     double max = pointerToResCapStatsIntegrated[comp].MTTF;
                     //pointerToResCapStatsIntegrated[comp].MTTF = ((double)rand()
/ RAND_MAX)*max;
                     findNewValue(pointerToResCapLife,
pointerToResCapStatsIntegrated, numCols, pointerToResCapStatsIntegrated, comp, 0,
pointerToResCapStatsIntegrated[comp].timeBetweenFail.size(), true,
thermalComponent, pointerToResCapInput);
                     pointerToResCapStatsIntegrated[comp].MTTF =
pointerToResCapLife[comp].TBF;
              }

              else if(pointerToResCapStatsIntegrated[comp].MTTF >= 0 &&
                     pointerToResCapStatsIntegrated[comp].age > 0 &&
                     (boolManualTTF == 0 ||
                     (boolManualTTF == 1 &&
                     comp != manualComponentSave)))
              {
                     pointerToResCapStatsIntegrated[comp].MTTF = -1;

                     while(pointerToResCapStatsIntegrated[comp].MTTF <
pointerToResCapStatsIntegrated[comp].age ||
                            pointerToResCapStatsIntegrated[comp].MTTF >
pointerToResCapInput[comp].maxTTF)
                     {
                            findNewValue(pointerToResCapLife,
pointerToResCapStatsIntegrated, numCols, pointerToResCapStatsIntegrated, comp, 0,
pointerToResCapStatsIntegrated[comp].timeBetweenFail.size(), true,
thermalComponent, pointerToResCapInput);
                                        pointerToResCapStatsIntegrated[comp].MTTF =
pointerToResCapLife[comp].TBF;

                                        if(counter == 2)
                                        {
                                                pointerToResCapStatsIntegrated[comp].MTTF =
pointerToResCapInput[comp].maxTTF;
                                        }

                                        counter++;
                                }

                                pointerToResCapStatsIntegrated[comp].MTTF -=
pointerToResCapStatsIntegrated[comp].age;

                                counter = 0;
                        }
                }

                if(boolManualTTF == 1)
                {
                        pointerToResCapStatsIntegrated[manualComponentSave].MTTF =
manualTTF;
                }

                if(boolManualRepair == 1)
                {
                        pointerToResCapStatsIntegrated[manualComponentSave].TTR_total =
manualTTR;
                }

                if(boolManualReplace == 1)
                {
                        pointerToResCapStatsIntegrated[manualComponentSave].replaceTotal =
manualReplace;
                }

                for(unsigned int comp = 0; comp < thermalComponent.size(); comp++)
                {
                        for(int col = 0; col < numCols; col++)
                        {
                                if(pointerToResCapStatsIntegrated[col].partID ==
thermalComponent[comp].partID)
                                {
                                        pointerToResCapStatsIntegrated[col].MTTF =
pointerToResCapStatsIntegrated[col].MTTF / thermalComponent[comp].epsillon;
                                }
                        }
                }

                // if the part has failed
                while(pointerToResCapStatsIntegrated[failedPosition].FAIL)
                {
                        failFix:

                        time+=timeInc;
```

```
              if(time > 200)
              {
                     failTable.fix.push_back(50);

                     goto exitLoop;
              }

              // the component has not decrmented in time and user chose to repair
the piece
              if(pointerToResCapStatsIntegrated[failedPosition].repairing)
              {
                     // decement time from mttr
                     pointerToResCapStatsIntegrated[failedPosition].TTR_total -=
timeInc;
              }

              // the component has not yet had time decremented and the user chose
to replace the component
              if(pointerToResCapStatsIntegrated[failedPosition].replacement)
              {
                     // decrement time from replace time holder
                     pointerToResCapStatsIntegrated[failedPosition].replaceTotal -=
timeInc;
              }

              // if the component has been fixed and it just failed
              if(pointerToResCapStatsIntegrated[failedPosition].TTR_total <= 0 ||
                     pointerToResCapStatsIntegrated[failedPosition].replaceTotal <=
0)
              {
                     // the component no longer fails
                     pointerToResCapStatsIntegrated[failedPosition].FAIL = false;

                     componentsFailed = 0;

                     // repair is decremented to zero and has not specified so
                     if(pointerToResCapStatsIntegrated[failedPosition].TTR_total <=
0)
                     {

       if(pointerToResCapStatsIntegrated[failedPosition].TTR_total < 0)
                            {

       //pointerToResCapStatsIntegrated[failedPosition].TTR_total -= timeInc;
                                   time +=
pointerToResCapStatsIntegrated[failedPosition].TTR_total;
                            }

                            // choice is no longer to repair

       pointerToResCapStatsIntegrated[failedPosition].repairing = false;

                            if(boolManualRepair == 0 ||
                                   (boolManualRepair == 1 &&
                                   failedPosition != manualComponentSave))
                            {
                                   findNewValue(pointerToResCapLife,
pointerToResCapStatsIntegrated, numCols, pointerToResCapStatsIntegrated,
failedPosition,1, pointerToResCapStatsIntegrated[failedPosition].repair.size(),
false, thermalComponent, pointerToResCapInput);

       pointerToResCapStatsIntegrated[failedPosition].TTR_total =
pointerToResCapLife[failedPosition].TTR;
                            }

                            else if(boolManualRepair == 1 &&
                                   failedPosition == manualComponentSave)
                            {

       pointerToResCapStatsIntegrated[failedPosition].TTR_total = manualTTR;
                            }


       //manual_TTF_TTR_REPLACE(pointerToResCapStatsIntegrated, numCols, 2);
                     }

                     // if the replace has been decremented to 0
                     else
if(pointerToResCapStatsIntegrated[failedPosition].replaceTotal <= 0)
                     {

       if(pointerToResCapStatsIntegrated[failedPosition].replaceTotal < 0)
                            {
                            //
       pointerToResCapStatsIntegrated[failedPosition].replaceTotal -= timeInc;
                                   time +=
pointerToResCapStatsIntegrated[failedPosition].replaceTotal;
                            }

                            // part is no longer being replaced

       pointerToResCapStatsIntegrated[failedPosition].replacement = false;

                            if(boolManualReplace == 0 ||
                                   (boolManualReplace == 1 &&
                                   failedPosition != manualComponentSave))
                            {
                                   findNewValue(pointerToResCapLife,
pointerToResCapStatsIntegrated, numCols, pointerToResCapStatsIntegrated,
failedPosition,2, pointerToResCapStatsIntegrated[failedPosition].replace.size(),
false, thermalComponent, pointerToResCapInput);

       pointerToResCapStatsIntegrated[failedPosition].replaceTotal =
pointerToResCapLife[failedPosition].replace;
                            }

                            else if(boolManualReplace == 1 &&
                                   failedPosition == manualComponentSave)
                            {

       pointerToResCapStatsIntegrated[failedPosition].TTR_total = manualReplace;
                            }
                     }
```

```
                    if(failTable.chip[failedPosition] ||
                          failTable.chipConnection[failedPosition])
                    {
                          failTable.fix.push_back(time);
                    }

                    componentsFailed = 0;

                    goto reEnterTime;
              }
       }

       // time loop
       for(time = 0; time < runLength; time += timeInc)
       {
              reEnterTime:

              if(time > 50)
              {
                    goto exitLoop;
              }

              // if a partID has not had the tbf decremented and it is still
working
              if(componentsFailed == 0)
              {
                    // search through all resistors
                    for(int R = 0; R < numCols; R++)
                    {

       if(!pointerToResCapStatsIntegrated[R].parallelComponent)
                          {
                                 // decrement each components time between
failures  by the time incrementer
                                 pointerToResCapStatsIntegrated[R].MTTF -=
timeInc;

                                 // when a part fails ( time between failure is
equal to 0 and it has not already failed
                                 if(pointerToResCapStatsIntegrated[R].MTTF <= 0)
                                 {
                                        componentsFailed++;


       lowestTTF.push_back(pointerToResCapStatsIntegrated[R].MTTF);

                                        // set the component to fail
                                        pointerToResCapStatsIntegrated[R].FAIL =
true;

                                        // if repairReplace is 1, user chose to
replace component, mark replacement as true

       if(pointerToResCapStatsIntegrated[R].replaceTotal <
pointerToResCapStatsIntegrated[R].TTR_total)
                                        {

       pointerToResCapStatsIntegrated[R].replacement = true;
```

```
                                        }

                                        // if repairReplace is 2, user chose to
repair component, mark repairing as true
                                        else
                                        {

        pointerToResCapStatsIntegrated[R].repairing = true;
                                        }


        //manual_TTF_TTR_REPLACE(pointerToResCapStatsIntegrated, numCols, 1);

                                        //std::cout << std::endl << "The number
of parts that have failed at time " << time << " are " << partsFailed <<
std::endl;
                                }
                        }
                }
        }

        if(componentsFailed > 0)
        {
                lowestFailedTBF = pointerToResCapStatsIntegrated[0].MTTF;

                int save = 0;

                for(int comp = 1; comp < numCols; comp++)
                {
                        if(pointerToResCapStatsIntegrated[comp].MTTF <
lowestFailedTBF &&
                                pointerToResCapStatsIntegrated[comp].FAIL)
                        {
                                lowestFailedTBF =
pointerToResCapStatsIntegrated[comp].MTTF;
                                save = comp;
                        }
                }

                lowestFailedTBF += timeInc;

                for(int comp = 0; comp < numCols; comp++)
                {
                        if(pointerToResCapStatsIntegrated[comp].FAIL &&
                                pointerToResCapStatsIntegrated[comp].MTTF < 0 &&
                                save != comp)
                        {
                                pointerToResCapStatsIntegrated[comp].MTTF -=
(lowestFailedTBF-timeInc);
                                pointerToResCapStatsIntegrated[comp].FAIL =
false;
                        }

                        else if(!pointerToResCapStatsIntegrated[comp].FAIL &&

        pointerToResCapStatsIntegrated[comp].parallelComponent)
                        {
```

```
					pointerToResCapStatsIntegrated[comp].MTTF -= 
(lowestFailedTBF-timeInc);
				}

				else if(pointerToResCapStatsIntegrated[comp].FAIL && 
					save == comp)
				{
					failedPosition = comp;

					if((boolManualTTF == 1 && comp != 
manualComponentSave) ||
						boolManualTTF == 0)
					{
						findNewValue(pointerToResCapLife, 
pointerToResCapStatsIntegrated, numCols, pointerToResCapStatsIntegrated, comp, 0, 
pointerToResCapStatsIntegrated[comp].timeBetweenFail.size(), false, 
thermalComponent, pointerToResCapInput);
						pointerToResCapStatsIntegrated[comp].MTTF 
= pointerToResCapLife[comp].TBF;

						for(unsigned int col = 0; col < 
thermalComponent.size(); col++)
						{
	if(pointerToResCapStatsIntegrated[comp].partID == 
thermalComponent[col].partID)
							{
	pointerToResCapStatsIntegrated[comp].MTTF = 
pointerToResCapStatsIntegrated[comp].MTTF / thermalComponent[col].epsillon;
							}
						}
					}

					else if(boolManualTTF == 1 && comp == 
manualComponentSave)
					{
						pointerToResCapStatsIntegrated[comp].MTTF 
= manualTTF;
					}

					if(failTable.chip[comp] ||
						failTable.chipConnection[comp])
					{
	failTable.partID.push_back(pointerToResCapStatsIntegrated[comp].partID);
					}
				}
			}

			time += lowestFailedTBF;

			if(failTable.chip[failedPosition] ||
				failTable.chipConnection[failedPosition])
			{
				failTable.failed.push_back(time);
			}
```

```
                        componentsFailed++;
                }

                if(componentsFailed > 0)
                {
                        componentsFailed = 0;
                        lowestTTF.clear();
                        failTable.partsFailed++;

                        goto failFix;
                }
        }
        // END TIME SEQUENCE

        exitLoop:

        for(int comp = 0; comp < numCols; comp++)
        {
                pointerToResCapStatsIntegrated[comp].FAIL = false;
                pointerToResCapStatsIntegrated[comp].replacement = false;
                pointerToResCapStatsIntegrated[comp].repairing = false;

                pointerToResCapStatsIntegrated[comp].MTTF = 
pointerToResCapStatsIntegrated[comp].mttfSaveSave;
                pointerToResCapStatsIntegrated[comp].TTR_total = 
pointerToResCapStatsIntegrated[comp].mttrSaveSave;
                pointerToResCapStatsIntegrated[comp].replaceTotal = 
pointerToResCapStatsIntegrated[comp].replaceSaveSave;
        }

        pointerToResCapStatsIntegrated[manualComponentSave].MTTF = manualTTF;

        return failTable;
}
//
// name: reliability_calc
// inputs: pointerToResCapStats = statisticsVectors = contains data obtained from 
that replication
//                      pointerToResCapStatsIntegrated = statisticsVectors = contains 
data obtained from all replications
//                      numCols = int = number of columns (components) in the table 
(input file)
// description: obtains a new distribution type for all values obtained from the 
sample and the lifetime of the replications
//                              and uses that new distribution to determine the 
reliability of a component by the mean of the TBF values.
void reliability_calc(statisticsVectors * pointerToResCapStats, statisticsVectors 
* pointerToResCapStatsIntegrated, Component * pointerToResCapInput, vector 
<systemReliability> & componentReliability, int numCols, double runLength, int 
boolManualTTF, bool MTTFyes, bool componentInput, int replication)
{
        // give pointerToResCapCalculation the Component structure

        // initializations required to generate the components reliability

        bool generate = false;
        int numRows;
        double mean;
```

```
	double sigma;
	double logMean;
	double logSigma;
	double beta;
	double eta;
	double useless;
	int i = 0;
	int sameValue = 0;
	double timeInc = 0.1;
	int col;

	double lognormalMean;
	double lognormalSigma;
	double wiebullMean;
	double wiebullSigma;
	double expoMean;
	double expoSigma;

	for(col = 0; col < numCols; col++)
	{
			// if normal
		if(pointerToResCapInput[col].distributionTypeHoldTBF == 1)
			{
				if(!MTTFyes)
				{

	pointerToResCapStatsIntegrated[col].reliabilityAge = 
finalNormal(pointerToResCapStatsIntegrated, col, 
pointerToResCapStatsIntegrated[col].age, pointerToResCapInput[col].MTTF, 
pointerToResCapInput[col].sigma);

					for(double time = 0; time < runLength+timeInc; 
time+=timeInc)
					{
						// calculate reliability based on normal 
equation

	componentReliability[col].reliability.push_back(finalNormal(pointerToResCap
StatsIntegrated, col, time + pointerToResCapStatsIntegrated[col].age, 
pointerToResCapInput[col].MTTF, pointerToResCapInput[col].sigma));
					}
				}

				else
				{

	pointerToResCapStatsIntegrated[col].replicationMTTF.push_back(pointerToResC
apInput[col].MTTF);
				}
			}

			// if Exponential
			else if(pointerToResCapInput[col].distributionTypeHoldTBF == 
2)
			{
				if(!MTTFyes)
				{
```

```
	pointerToResCapStatsIntegrated[col].reliabilityAge = 
finalExponential(pointerToResCapStatsIntegrated, col, 
pointerToResCapStatsIntegrated[col].age, pointerToResCapInput[col].MTTF);

					for(double time = 0; time < runLength+timeInc; 
time+=timeInc)
					{
						// calculate reliability based on 
exponential equation

	componentReliability[col].reliability.push_back(finalExponential(pointerToR
esCapStatsIntegrated, col, time + pointerToResCapStatsIntegrated[col].age, 
pointerToResCapInput[col].MTTF));
					}
				}

				else
				{

	pointerToResCapStatsIntegrated[col].replicationMTTF.push_back(pointerToResC
apInput[col].MTTF);
				}
			}

			// if lognormal
			else if(pointerToResCapInput[col].distributionTypeHoldTBF == 
3)
			{
				if(!MTTFyes)
				{

	pointerToResCapStatsIntegrated[col].reliabilityAge = 
finalLognormal(pointerToResCapStatsIntegrated, col, 
pointerToResCapInput[col].logMTTF, pointerToResCapInput[col].logSIGMA, 
pointerToResCapStatsIntegrated[col].age);

					for(double time = 0; time < runLength+timeInc; 
time+=timeInc)
					{
						// calculate reliability based on 
lognormal equation

	componentReliability[col].reliability.push_back(finalLognormal(pointerToRes
CapStatsIntegrated, col, pointerToResCapInput[col].logMTTF, 
pointerToResCapInput[col].logSIGMA, time + 
pointerToResCapStatsIntegrated[col].age));
					}
				}

				else
				{

	pointerToResCapStatsIntegrated[col].replicationMTTF.push_back(pointerToResC
apInput[col].MTTF);
				}
			}
```

```
                    // if weibull
                    else if(pointerToResCapStatsIntegrated[col].distribution == 4)
                    {
                            if(!MTTFyes)
                            {

        pointerToResCapStatsIntegrated[col].reliabilityAge =
finalWeibull(pointerToResCapStatsIntegrated, col,
pointerToResCapStatsIntegrated[col].age, pointerToResCapInput[col].eta,
pointerToResCapInput[col].beta);

                                    for(double time = 0; time < runLength+timeInc;
time+=timeInc)
                                    {
                                            // calculate reliability based on weibull
equation

        componentReliability[col].reliability.push_back(finalWeibull(pointerToResCa
pStatsIntegrated, col, time + pointerToResCapStatsIntegrated[col].age,
pointerToResCapInput[col].eta, pointerToResCapInput[col].beta));
                                    }
                            }

                            else
                            {

        pointerToResCapStatsIntegrated[col].replicationMTTF.push_back(pointerToResC
apInput[col].MTTF);
                            }
                    }
        }

        if(!MTTFyes)
        {
                for(unsigned int comp = 0; comp < componentReliability.size();
comp++)
                {
                        for(unsigned int inside = 0; inside <
componentReliability[comp].reliability.size(); inside++)
                        {
                                componentReliability[comp].reliability[inside] =
componentReliability[comp].reliability[inside] /
pointerToResCapStatsIntegrated[comp].reliabilityAge;
                        }
                }
        }

        return;
}

// name: finalNormal
// inputs: pointerToResCapStatsIntegrated = statisticsVectors = contains all
cumulative data obtained throughout the lifetime of all replications
//                  componentPosition = int = element number in array getting
modified
//                  mean = double = mean of all TBF values from sample and
obtained
```

```
// outputs: reliability = double = reliability value obtained if the normal CDF
calculation is used
// Description: calculates the normal probability of a component if this is the
correct distrbution type
double finalNormal(statisticsVectors * pointerToResCapStatsIntegrated, int
componentPosition, double time, double mean, double sigma)
{
       double errorFunc;
       double reliability;
       double normal;

       // reliability = 1 - CDF(x)
       // ERF(x) = (x - mu)/(sigma*sqrt(2))
       // CDF(x) = (1/2) * ( 1 + ERF(x))
       errorFunc = ((time - mean) / (sigma * sqrt(2.0)));

       normal = 0.5 * ( 1 + errorFunction(errorFunc));

       reliability = 1 - normal;

       // return reliability to reliability_calc function
       return reliability;
}

// name: finalExponential
// inputs: pointerToResCapStatsIntegrated = statisticsVectors = contains all
cumulative data obtained throughout the lifetime of all replications
//                   componentPosition = int = element number in array getting
modified
// outputs: exponent = double = reliability value obtained if the exponential CDF
calculation is used
// Description: calculates the exponential probability of a component if this is
the correct distrbution type
double finalExponential(statisticsVectors * pointerToResCapStatsIntegrated, int
componentPosition, double time, double expoMean)
{
       double exponent;
       double reliability;
       double exponential;

       // reliability = 1 - CDF(x)
       // CDF(x) = 1 - e^(-x/mu)
       exponent = (-time/expoMean);

       exponential = 1.000 - exp(exponent);

       reliability = 1.000 - exponential;

       // return reliability to reliability_calc function
       return reliability;
}

// name: finalLognormal
// inputs: pointerToResCapStatsIntegrated = statisticsVectors = contains all
cumulative data obtained throughout the lifetime of all replications
//                   componentPosition = int = element number in array getting
modified
```

```
// outputs: errorFunc = double = reliability value obtained if the Lognormal CDF
calculation is used
// Description: calculates the Lognormal probability of a component if this is the
correct distrbution type
double finalLognormal(statisticsVectors * pointerToResCapStatsIntegrated, int
componentPosition, double logMean, double logSigma, double time)
{
       double errorFunc;
       double reliability;
       double lognormal;

       // reliability = 1 - CDF(x)
       // ERF(x) = (ln(x) - mu)/(sigma*sqrt(2)) -> note the mean and STDDEV of
this distribution type will use the natual log of each value in its calculation
       // CDF(x) = (1/2) * ( 1 + ERF(x))
       errorFunc = ((log(time) - logMean) / (logSigma * sqrt(2.0)));

       //std::cout << time << "\t" << logMean << "\t" << logSigma << "\t" <<
errorFunc << std::endl;
       //     getchar();

       lognormal = 0.5 * ( 1.000 + errorFunction(errorFunc));

       reliability = 1.000 - lognormal;

       // return reliability to reliability_calc function
       return reliability;
}

// name: finalWeibull
// inputs: pointerToResCapStatsIntegrated = statisticsVectors = contains all
cumulative data obtained throughout the lifetime of all replications
//                   componentPosition = int = element number in array getting
modified
// outputs: power = double = reliability value obtained if the weibull CDF
calculation is used
// Description: calculates the weibull probability of a component if this is the
correct distrbution type
double finalWeibull(statisticsVectors * pointerToResCapStatsIntegrated, int
componentPosition, double time, double etas, double betas)
{
       double power;
       double reliability;
       double weibull;

       // reliability = 1 - CDF(x)
       // CDF(x) =  (x/eta
       power = time/etas;
       power = pow(power,betas);
       power = -power;
       weibull = 1.0000 - (exp(power));
       reliability = 1.000 - weibull;

       // return reliability to reliability_calc function
       return reliability;
}
```

```
void computeSystemReliability(systemReliability & reliable, vector <failing> &
failTableIntegrated, vector <failing> & outputFailTable, int runLength, int
numberOfReplications)
{
       double timeInc = 0.01;
       double numberAboveEqual = 0;
       double reliabilityAverage = 0;
       bool rely_begin = true;;
       bool rely_end = false;

       for(double time = 0; time < runLength; time+=timeInc)
       {
              for(unsigned int comp = 0; comp < failTableIntegrated.size();
comp++)
              {
                     if(failTableIntegrated[comp].failed[0] >= time)
                     {
                            numberAboveEqual++;
                     }
              }

              reliabilityAverage = numberAboveEqual/numberOfReplications;

              if(reliabilityAverage > 0 &&
                     reliabilityAverage < 1 &&
                     !rely_end)
              {
                     if(reliabilityAverage == 1)
                     {
                            rely_end = true;
                     }

                     reliable.reliability.push_back(reliabilityAverage);

                     if(rely_begin)
                     {
                            reliable.timeBegin = time;

                            rely_begin = false;
                     }
              }

              else if(reliabilityAverage == 1 &&
                     rely_end)
              {
                     reliable.timeEnd = time;
                     rely_end = false;

                     goto end;
              }

              numberAboveEqual = 0;
       }

       end:
       return;
}
```

```
void minMaxTTF(Component * pointerToResCapInput, statisticsVectors *
pointerToResCapStatsIntegrated, int numCols)
{
	for(int comp = 0; comp < numCols; comp++)
	{
		if(pointerToResCapStatsIntegrated[comp].distribution == 1)
		{
			pointerToResCapInput[comp].maxTTF =
pointerToResCapInput[comp].MTTF + pointerToResCapInput[comp].sigma*1.5;
			pointerToResCapInput[comp].minTTF =
pointerToResCapInput[comp].MTTF - pointerToResCapInput[comp].sigma*1.5;
		}

		else if(pointerToResCapStatsIntegrated[comp].distribution == 2)
		{
			pointerToResCapInput[comp].maxTTF =
pointerToResCapInput[comp].MTTF + pointerToResCapInput[comp].sigma*1.5;
			pointerToResCapInput[comp].minTTF =
pointerToResCapInput[comp].MTTF - pointerToResCapInput[comp].sigma*1.5;
		}

		else if(pointerToResCapStatsIntegrated[comp].distribution == 3)
		{
			pointerToResCapInput[comp].maxTTF =
pointerToResCapInput[comp].MTTF + pointerToResCapInput[comp].sigma*1.5;
			pointerToResCapInput[comp].minTTF =
pointerToResCapInput[comp].MTTF - pointerToResCapInput[comp].sigma*1.5;
		}

		else
		{
			pointerToResCapInput[comp].maxTTF =
pointerToResCapInput[comp].MTTF + pointerToResCapInput[comp].sigma*1.5;
			pointerToResCapInput[comp].minTTF =
pointerToResCapInput[comp].MTTF - pointerToResCapInput[comp].sigma*1.5;
		}

		if(pointerToResCapInput[comp].minTTF < 0)
		{
			pointerToResCapInput[comp].minTTF = 0;
		}
	}

	return;
}

void meanGenerateTTF(vector <statisticsVectors> & exportGeneratedMTTR)
{
	double average = 0;

	for(unsigned int comp = 0; comp < exportGeneratedMTTR.size(); comp++)
	{
		for(unsigned int row = 0; row <
exportGeneratedMTTR[comp].repair.size(); row++)
		{
			average += exportGeneratedMTTR[comp].repair[row];
		}
		exportGeneratedMTTR[comp].MTTR = 
average/exportGeneratedMTTR[comp].repair.size();

		average = 0;
	}

	return;
}

// name: outputFinalTable
// inputs: pointerToResCapStatsIntegrated = statisticsVectors = all values, new 
and from file generated throughout the programs run life (all replications
//			numCols = int = number of columns conatined in TTF file
//			lifetime = double = lifetime of the PCB (equal to the lowest 
MTBF of all components)
//			halfWidth = double = halfWidth of the component containing the 
lowest MTBF
// description: prints out all the values generated throughout the program into a 
table in a .txt format
void outputFinalTable(statisticsVectors * pointerToResCapStatsIntegrated, int 
numCols, double lifetime, int boolManualTTF, string fileName,
	string maintainFile, systemReliability & reliable, vector 
<systemReliability> & componentReliability, int boolManualRepair, int 
boolManualReplace, failing failTable,
	vector <parallelFail> & parallelFailing, Component * pointerToResCapInput, 
vector <failing> & outputFailTable, vector<failing> & failTableIntegrated,
	vector <thermal> & thermalComponent, vector <double> & systemFailureRate, 
vector <statisticsVectors> & exportGeneratedMTTR, vector <failing> & outputMTTR)
{
	double TTFmin = DBL_MAX;
	double TTFmax = 0;
	double lowestTTF = DBL_MAX;
	float time = pointerToResCapStatsIntegrated[0].reliabilityAge;
	double timeInc = 0.5;
	int numberReps = failTableIntegrated.size();
	int number = 0;
	unsigned int increment = 0;

	fstream outputTable;
	fstream outputIdentifier;

	string txtFile;
	string identifier;
	string buffer = fileName;
	string bufferMaintain = maintainFile;

	maintainFile.append("Generated MTTR.txt");

	outputIdentifier.open(maintainFile.c_str(), ios::out);

	if(outputIdentifier.is_open())
	{
		for(unsigned int comp = 0; comp < exportGeneratedMTTR.size(); 
comp++)
		{
			outputIdentifier << exportGeneratedMTTR[comp].partID;

			if(comp < exportGeneratedMTTR.size()-1)
```

```
                        {
                                outputIdentifier << setw(10);
                        }
                }

                outputIdentifier << std::endl;

                unsigned int yous = 0;

                for(unsigned int comp= 0; comp <
exportGeneratedMTTR[yous].repair.size(); comp++)
                {
                        while(yous < exportGeneratedMTTR.size())
                        {
                                outputIdentifier <<
exportGeneratedMTTR[yous].repair[comp];

                                if(yous < exportGeneratedMTTR.size()-1)
                                {
                                        outputIdentifier << setw(10);
                                }

                                yous++;
                        }

                        yous = 0;

                        outputIdentifier << std::endl;
                }
        }

        maintainFile.clear();
        maintainFile = bufferMaintain;

        outputIdentifier.clear();
        outputIdentifier.close();

        ////for(int comp = 0; comp < numberReps; comp++)
        ////{
        //      stringstream column;

        //      int col = 1;

        //      column << col;

        //      fileName.append(column.str());
        //      fileName.append(" Rep Fail.txt");

        //      outputIdentifier.open(fileName.c_str(), ios::out);

        //      outputIdentifier << "FAIL" << setw(20) << "FIX" << setw(20) <<
"CAUSE" << std::endl;

        //      for(unsigned int row = 0; row <
failTableIntegrated[0].failed.size(); row++)
        //      {
        //              outputIdentifier << failTableIntegrated[0].failed[row] <<
setw(20) << failTableIntegrated[0].fix[row] << setw(20);
```

```
	//		for(unsigned int part = 0; part < 
failTableIntegrated[0].partID.size(); part++)
	//		{
	//			outputIdentifier << setw(10) << 
failTableIntegrated[0].partID[part];
	//		}

	//		outputIdentifier << std::endl;
	//	}

	//	outputIdentifier.close();
	//	outputIdentifier.clear();
	//	fileName.erase();
	//	fileName.clear();

	//	fileName = buffer;
	////}

	maintainFile.append("System MTTR.txt");

	outputIdentifier.open(maintainFile.c_str(), ios::out);

	outputIdentifier << "MTTR" << std::endl;

	for(unsigned int row = 0; row < outputMTTR.size(); row++)
	{
		outputIdentifier << outputMTTR[row].meanFix << std::endl;
	}

	outputIdentifier << numberReps << std::endl;

	outputIdentifier.close();
	outputIdentifier.clear();
	fileName.erase();
	fileName.clear();

	fileName = buffer;

	fileName.append("System Failure Rate.txt");

	outputIdentifier.open(fileName.c_str(), ios::out);

	outputIdentifier << "Failure Rate" << std::endl;

	outputIdentifier << systemFailureRate[0] << std::endl;

	outputIdentifier.close();
	outputIdentifier.clear();
	fileName.erase();
	fileName.clear();

	fileName = buffer;

	if(boolManualTTF == 0 &&
		boolManualRepair == 0 &&
		boolManualReplace == 0)
	{
```

```
              for(int col = 0; col < numCols; col++)
              {
                     identifier = pointerToResCapStatsIntegrated[col].partID;

                     //CreateDirectory(fileName.c_str(), NULL);
                     //fileName.append("Component Reliability\\");
                     fileName.append(identifier);
                     fileName.append(".txt");

                     outputIdentifier.open(fileName.c_str(), ios::out);

                     outputIdentifier << "MTTF" << setw(15) << "1.5 x sigma" <<
setw(15) << "TTFmin" << setw(15) << "TTFmax" << setw(15) << "Failure Rate" <<
setw(15) << "Distribution" << setw(15);

                     if(pointerToResCapStatsIntegrated[col].distribution == 1)
                     {
                            outputIdentifier << "MEAN" << setw(14)<< "Standard
Deviation" << std::endl;
                            outputIdentifier << pointerToResCapInput[col].MTTF <<
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) <<
pointerToResCapInput[col].minTTF << setw(14) << pointerToResCapInput[col].maxTTF
<< setw(14) << pointerToResCapStatsIntegrated[col].failureRate << setw(14) <<
"NORMAL" << setw(15) << pointerToResCapInput[col].MTTF << setw(15) <<
pointerToResCapInput[col].sigma << std::endl;
                     }

                     else if(pointerToResCapStatsIntegrated[col].distribution == 2)
                     {
                            outputIdentifier << "MEAN" << std::endl;
                            outputIdentifier << pointerToResCapInput[col].MTTF <<
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) <<
pointerToResCapInput[col].minTTF<< setw(14) << pointerToResCapInput[col].maxTTF <<
setw(14) << pointerToResCapStatsIntegrated[col].failureRate << setw(14) <<
"EXPONENTIAL" << setw(15) << pointerToResCapInput[col].MTTF << std::endl;
                     }

                     else if(pointerToResCapStatsIntegrated[col].distribution == 3)
                     {
                            outputIdentifier << setw(14) << "MU" << setw(14) <<
"sigma" <<std::endl;
                            outputIdentifier << pointerToResCapInput[col].MTTF <<
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) <<
pointerToResCapInput[col].minTTF << setw(14) << pointerToResCapInput[col].maxTTF
<< setw(14) << pointerToResCapStatsIntegrated[col].failureRate << setw(14) <<
"LOGNORMAL" << setw(15) << pointerToResCapInput[col].logMTTF << setw(15) <<
pointerToResCapInput[col].logSIGMA << std::endl;
                     }

                     else
                     {
                            outputIdentifier << "Beta" << setw(14) << "Eta" <<
std::endl;
                            outputIdentifier << pointerToResCapInput[col].MTTF <<
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) <<
pointerToResCapInput[col].minTTF << setw(14) << pointerToResCapInput[col].maxTTF
<< setw(14) << pointerToResCapStatsIntegrated[col].failureRate << setw(14) <<
"WEIBULL" << setw(15) << pointerToResCapInput[col].beta << setw(15) <<
pointerToResCapInput[col].eta << std::endl;
                    }

                    outputIdentifier.close();
                    outputIdentifier.clear();

                    fileName.erase();
                    fileName.clear();

                    fileName = buffer;

                    //fileName.append("Component Reliability\\");
                    fileName.append(identifier);
                    fileName.append(" Means.txt");

                    outputIdentifier.open(fileName.c_str(), ios::out);

                    outputIdentifier << "MTTF" << setw(15) << "1.5 x sigma" <<
setw(15) << "MTTRepair" << setw(15) << "MTTReplace" << std::endl;

                    if(pointerToResCapStatsIntegrated[col].distribution == 1)
                    {
                           outputIdentifier << pointerToResCapInput[col].MTTF <<
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) <<
pointerToResCapInput[col].inRepair << setw(15) <<
pointerToResCapInput[col].inReplace << std::endl;
                    }

                    else if(pointerToResCapStatsIntegrated[col].distribution == 2)
                    {
                           outputIdentifier << pointerToResCapInput[col].MTTF <<
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) <<
pointerToResCapInput[col].inRepair << setw(15) <<
pointerToResCapInput[col].inReplace << std::endl;
                    }

                    else if(pointerToResCapStatsIntegrated[col].distribution == 3)
                    {
                           outputIdentifier << pointerToResCapInput[col].MTTF <<
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) <<
pointerToResCapInput[col].inRepair << setw(15) <<
pointerToResCapInput[col].inReplace << std::endl;
                    }

                    else
                    {
                           outputIdentifier << pointerToResCapInput[col].MTTF <<
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) <<
pointerToResCapInput[col].inRepair << setw(15) <<
pointerToResCapInput[col].inReplace << std::endl;
                    }

                    outputIdentifier.close();
                    outputIdentifier.clear();
                    fileName.erase();
                    fileName.clear();
                    fileName = buffer;

                    for(unsigned int comp = 0; comp < thermalComponent.size();
comp++)
                    {
                           if(thermalComponent[comp].partID ==
pointerToResCapStatsIntegrated[col].partID)
                           {
                                  number++;
                           }
                    }

                    if(number == 0)
                    {
                           fileName.append(identifier);
                           fileName.append(" Reliability.txt");

                           outputIdentifier.open(fileName.c_str(), ios::out);

                           outputIdentifier << "Time" << setw(15) << "Reliability"
<< std::endl;

                           time = 0;

                           for(unsigned int rely = 0; rely <
componentReliability[col].reliability.size(); rely++)
                           {
                                  outputIdentifier << time << setw(15) <<
componentReliability[col].reliability[rely] << std::endl;
                                  time+=0.1;
                           }

                           outputIdentifier.close();
                           outputIdentifier.clear();
                           fileName.erase();
                           fileName.clear();

                           fileName = buffer;
                    }

                    number = 0;
             }

             fileName.append("System Level Lifetime.txt");

             outputIdentifier.open(fileName.c_str(), ios::out);

             outputIdentifier << "Min Fail" << setw(20) << "Fail" << setw(20) <<
"Max Fail" << setw(20) << "Fix" << setw(27) << "Component ID" << std::endl;

             for(unsigned int comp = 0; comp < outputFailTable.size(); comp++)
             {
                    outputIdentifier << outputFailTable[comp].minFail << setw(20)
<< outputFailTable[comp].mean << setw(20) << outputFailTable[comp].maxFail <<
setw(20) << outputFailTable[comp].meanFix << setw(20) <<
outputFailTable[comp].partID[0];

                    if(outputFailTable[comp].partID.size() > 1)
```

```
				{
					for(unsigned int part = 1; part < 
outputFailTable[comp].partID.size(); part++)
					{
						outputIdentifier << setw(10) << 
outputFailTable[comp].partID[part];
					}
				}

				outputIdentifier << std::endl;
			}

			outputIdentifier.close();
			outputIdentifier.clear();
			fileName.erase();
			fileName.clear();

			fileName = buffer;

			fileName.append("System Reliability.txt");

			outputIdentifier.open(fileName.c_str(), ios::out);

			outputIdentifier << "Time" << setw(15) << "Reliability" << 
std::endl;

			time = reliable.timeBegin-0.05;
			timeInc = 0.01;

			for(int comp = 0; comp < 5; comp++)
			{
				if(time >-0.01 &&
					time < 0)
				{
					outputIdentifier << "0" << setw(20) << "1" << 
std::endl;
				}

				else if(time >= 0.01)
				{
					outputIdentifier << time << setw(20) << "1" << 
std::endl;
				}

				time+=timeInc;
			}

			while(increment < reliable.reliability.size())
			{
				outputIdentifier << time << setprecision(5) << setw(20) << 
reliable.reliability[increment] << std::endl;

				time+=timeInc;
				increment++;
			}

			for(int comp = 0; comp < 5; comp++)
			{
```

```
                        outputIdentifier << time << setw(20) << "0" << std::endl;
                        time+=timeInc;
                }

                outputIdentifier.close();
                outputIdentifier.clear();
                fileName.erase();
                fileName.clear();

                fileName = buffer;
        }

        else if(boolManualTTF == 1)
        {
                for(int col = 0; col < numCols; col++)
                {
                        if(col != manualComponentSave)
                        {
                                identifier = 
pointerToResCapStatsIntegrated[col].partID;

                                //CreateDirectory(fileName.c_str(), NULL);
                                //fileName.append("Component Reliability\\");
                                fileName.append(identifier);
                                fileName.append(".txt");

                                outputIdentifier.open(fileName.c_str(), ios::out);

                                outputIdentifier << "MTTF" << setw(15) << "1.5 x sigma" 
<< setw(15) << "TTFmin" << setw(15) << "TTFmax" << setw(15) << "Failure Rate" << 
setw(15) << "Distribution" << setw(15);

                                if(pointerToResCapStatsIntegrated[col].distribution == 
1)
                                {
                                        outputIdentifier << "MEAN" << setw(14)<< 
"Standard Deviation" << std::endl;
                                        outputIdentifier << 
pointerToResCapInput[col].MTTF << setw(14) << pointerToResCapInput[col].sigma*1.5 
<< setw(14) << pointerToResCapInput[col].minTTF << setw(14) << 
pointerToResCapInput[col].maxTTF << setw(14) << 
pointerToResCapStatsIntegrated[col].failureRate << setw(14) << "NORMAL" << 
setw(15) << pointerToResCapInput[col].MTTF << setw(15) << 
pointerToResCapInput[col].sigma << std::endl;
                                }

                                else 
if(pointerToResCapStatsIntegrated[col].distribution == 2)
                                {
                                        outputIdentifier << "MEAN" << std::endl;
                                        outputIdentifier << 
pointerToResCapInput[col].MTTF << setw(14) << pointerToResCapInput[col].sigma*1.5 
<< setw(14) << pointerToResCapInput[col].minTTF << setw(14) << 
pointerToResCapInput[col].maxTTF << setw(14) << 
pointerToResCapStatsIntegrated[col].failureRate << setw(14) << "EXPONENTIAL" << 
setw(15) << pointerToResCapInput[col].MTTF << std::endl;
                                }
```

```
                            else
if(pointerToResCapStatsIntegrated[col].distribution == 3)
                            {
                                   outputIdentifier << setw(14) << "MU" << setw(14)
<< "sigma" <<std::endl;
                                   outputIdentifier <<
pointerToResCapInput[col].MTTF << setw(14) << pointerToResCapInput[col].sigma*1.5
<< setw(14) << pointerToResCapInput[col].minTTF << setw(14) <<
pointerToResCapInput[col].maxTTF << setw(14) <<
pointerToResCapStatsIntegrated[col].failureRate << setw(14) << "LOGNORMAL" <<
setw(15) << pointerToResCapInput[col].logMTTF << setw(15) <<
pointerToResCapInput[col].logSIGMA << std::endl;
                            }

                            else
                            {
                                   outputIdentifier << "Beta" << setw(14) << "Eta"
<< std::endl;
                                   outputIdentifier <<
pointerToResCapInput[col].MTTF << setw(14) << pointerToResCapInput[col].sigma*1.5
<< setw(14) << pointerToResCapInput[col].minTTF << setw(14) <<
pointerToResCapInput[col].maxTTF << setw(14) <<
pointerToResCapStatsIntegrated[col].failureRate << setw(14) << "WEIBULL" <<
setw(15) << pointerToResCapInput[col].beta << setw(15) <<
pointerToResCapInput[col].eta << std::endl;
                            }

                            outputIdentifier.close();
                            outputIdentifier.clear();

                            fileName.erase();
                            fileName.clear();

                            fileName = buffer;

                            //fileName.append("Component Reliability\\");
                            fileName.append(identifier);
                            fileName.append(" Means.txt");

                            outputIdentifier.open(fileName.c_str(), ios::out);

                            outputIdentifier << "MTTF" << setw(15) << "1.5 x sigma"
<< setw(15) << "MTTRepair" << setw(15) << "MTTReplace" << std::endl;

                            if(pointerToResCapStatsIntegrated[col].distribution ==
1)
                            {
                                   outputIdentifier <<
pointerToResCapInput[col].MTTF << setw(14) << pointerToResCapInput[col].sigma*1.5
<< setw(14) << pointerToResCapInput[col].inRepair << setw(15) <<
pointerToResCapInput[col].inReplace << std::endl;
                            }

                            else
if(pointerToResCapStatsIntegrated[col].distribution == 2)
                            {
                                   outputIdentifier <<
pointerToResCapInput[col].MTTF << setw(14) << pointerToResCapInput[col].sigma*1.5
 << setw(14) << pointerToResCapInput[col].inRepair << setw(15) <<
 pointerToResCapInput[col].inReplace << std::endl;
                            }

                            else
 if(pointerToResCapStatsIntegrated[col].distribution == 3)
                            {
                                   outputIdentifier <<
 pointerToResCapInput[col].MTTF << setw(14) << pointerToResCapInput[col].sigma*1.5
 << setw(14) << pointerToResCapInput[col].inRepair << setw(15) <<
 pointerToResCapInput[col].inReplace << std::endl;
                            }

                            else
                            {
                                   outputIdentifier <<
 pointerToResCapInput[col].MTTF << setw(14) << pointerToResCapInput[col].sigma*1.5
 << setw(14) << pointerToResCapInput[col].inRepair << setw(15) <<
 pointerToResCapInput[col].inReplace << std::endl;
                            }

                            outputIdentifier.close();
                            outputIdentifier.clear();
                            fileName.erase();
                            fileName.clear();

                            fileName = buffer;
                     }
              }

              identifier =
 pointerToResCapStatsIntegrated[manualComponentSave].partID;

              //CreateDirectory(fileName.c_str(), NULL);

              //fileName.append("Component Reliability\\");
              fileName.append(identifier);
              fileName.append(".txt");

              outputIdentifier.open(fileName.c_str(), ios::out);

              outputIdentifier << "MTTF" << setw(15) << "1.5 x sigma" << setw(15)
 << "TTFmin" << setw(15) << "TTFmax" << setw(15) << "Failure Rate" << setw(15) <<
 "Distribution" << setw(15) << "MEAN" << std::endl;
              outputIdentifier <<
 pointerToResCapStatsIntegrated[manualComponentSave].expoMean << setw(15) << 0.0 <<
 setw(15) << pointerToResCapStatsIntegrated[manualComponentSave].expoMean <<
 setw(15) << pointerToResCapStatsIntegrated[manualComponentSave].expoMean <<
 setw(15) << pointerToResCapStatsIntegrated[manualComponentSave].failureRate <<
 setw(15) << "EXPONENTIAL" << setw(15) <<
 pointerToResCapStatsIntegrated[manualComponentSave].expoMean << std::endl;

              outputIdentifier.close();
              outputIdentifier.clear();
              fileName.erase();
              fileName.clear();

              fileName = buffer;
```

```
		fileName.append(identifier);
		fileName.append(" means.txt");

		outputIdentifier.open(fileName.c_str(), ios::out);

		outputIdentifier << "MTTF" << setw(15) << "1.5 x sigma" << setw(15)
<< "MTTRepair" << setw(15) << "MTTReplace" << std::endl;
		outputIdentifier <<
pointerToResCapStatsIntegrated[manualComponentSave].expoMean << setw(15) << "0.0"
<< setw(15) << pointerToResCapStatsIntegrated[manualComponentSave].TTR_total <<
setw(15) << pointerToResCapStatsIntegrated[manualComponentSave].replaceTotal <<
std::endl;

		outputIdentifier.close();
		outputIdentifier.clear();
		fileName.erase();
		fileName.clear();

		fileName = buffer;

		for(int col = 0; col < numCols; col++)
		{
			if(col != manualComponentSave)
			{
				identifier =
pointerToResCapStatsIntegrated[col].partID;

				for(unsigned int comp = 0; comp <
thermalComponent.size(); comp++)
				{
					if(thermalComponent[comp].partID ==
pointerToResCapStatsIntegrated[col].partID)
					{
						number++;
					}
				}

				if(number == 0)
				{
					fileName.append(identifier);
					fileName.append(" Reliability.txt");

					outputIdentifier.open(fileName.c_str(),
ios::out);

					outputIdentifier << "Time" << setw(15) <<
"Reliability" << std::endl;

					time = 0;

					for(unsigned int rely = 0; rely <
componentReliability[col].reliability.size(); rely++)
					{
						outputIdentifier << time << setw(15) <<
componentReliability[col].reliability[rely] << std::endl;
						time+=0.1;
					}
```

```
                              outputIdentifier.close();
                              outputIdentifier.clear();
                              fileName.erase();
                              fileName.clear();

                              fileName = buffer;
                       }

                       number = 0;
                }
         }

         fileName.append("System Level Lifetime.txt");

         outputIdentifier.open(fileName.c_str(), ios::out);

         outputIdentifier << "Min Fail" << setw(20) << "Fail" << setw(20) <<
"Max Fail" << setw(20) << "Fix" << setw(27) << "Component ID" << std::endl;

         for(unsigned int comp = 0; comp < outputFailTable.size(); comp++)
         {
                outputIdentifier << outputFailTable[comp].minFail << setw(20)
<< outputFailTable[comp].mean << setw(20) << outputFailTable[comp].maxFail <<
setw(20) << outputFailTable[comp].meanFix << setw(20) <<
outputFailTable[comp].partID[0];

                if(outputFailTable[comp].partID.size() > 1)
                {
                       for(unsigned int part = 1; part <
outputFailTable[comp].partID.size(); part++)
                       {
                              outputIdentifier << setw(10) <<
outputFailTable[comp].partID[part];
                       }
                }

                outputIdentifier << std::endl;
         }

         outputIdentifier.close();
         outputIdentifier.clear();
         fileName.erase();
         fileName.clear();

         fileName = buffer;

         fileName.append("System Reliability.txt");

         outputIdentifier.open(fileName.c_str(), ios::out);

         outputIdentifier << "Time" << setw(15) << "Reliability" <<
std::endl;

         time = reliable.timeBegin-0.05;
         timeInc = 0.01;

         for(int comp = 0; comp < 5; comp++)
         {
```

```
                    if(time >-0.01 &&
                           time < 0)
                    {
                           outputIdentifier << "0" << setw(20) << "1" <<
std::endl;
                    }

                    else if(time >= 0)
                    {
                           outputIdentifier << time << setw(20) << "1" <<
std::endl;
                    }

                    time+=timeInc;
             }

             while(increment < reliable.reliability.size())
             {
                    outputIdentifier << time << setprecision(5) << setw(20) <<
reliable.reliability[increment] << std::endl;

                    time+=timeInc;
                    increment++;
             }

             for(int comp = 0; comp < 5; comp++)
             {
                    outputIdentifier << time << setw(20) << "0" << std::endl;
                    time+=timeInc;
             }

             outputIdentifier.close();
             outputIdentifier.clear();
             fileName.erase();
             fileName.clear();

             fileName = buffer;
      }

      else if(boolManualReplace == 1 ||
             boolManualRepair == 1)
      {
             for(int col = 0; col < numCols; col++)
             {
                    identifier = pointerToResCapStatsIntegrated[col].partID;

                    //CreateDirectory(fileName.c_str(), NULL);
                    //fileName.append("Component Reliability\\");
                    fileName.append(identifier);
                    fileName.append(".txt");

                    outputIdentifier.open(fileName.c_str(), ios::out);

                    outputIdentifier << "MTTF" << setw(15) << "1.5 x sigma" <<
setw(15) << "TTFmin" << setw(15) << "TTFmax" << setw(15) << "Failure Rate" <<
setw(15) << "Distribution" << setw(15);

                    if(pointerToResCapStatsIntegrated[col].distribution == 1)
                    {
                          outputIdentifier << "MEAN" << setw(14)<< "Standard 
Deviation" << std::endl;
                          outputIdentifier << pointerToResCapInput[col].MTTF << 
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) << 
pointerToResCapInput[col].minTTF << setw(14) << pointerToResCapInput[col].maxTTF 
<< setw(14) << pointerToResCapStatsIntegrated[col].failureRate << setw(14) << 
"NORMAL" << setw(15) << pointerToResCapInput[col].MTTF << setw(15) << 
pointerToResCapInput[col].sigma << std::endl;
                    }

                    else if(pointerToResCapStatsIntegrated[col].distribution == 2)
                    {
                          outputIdentifier << "MEAN" << std::endl;
                          outputIdentifier << pointerToResCapInput[col].MTTF << 
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) << 
pointerToResCapInput[col].minTTF<< setw(14) << pointerToResCapInput[col].maxTTF << 
setw(14) << pointerToResCapStatsIntegrated[col].failureRate << setw(14) << 
"EXPONENTIAL" << setw(15) << pointerToResCapInput[col].MTTF << std::endl;
                    }

                    else if(pointerToResCapStatsIntegrated[col].distribution == 3)
                    {
                          outputIdentifier << setw(14) << "MU" << setw(14) << 
"sigma" <<std::endl;
                          outputIdentifier << pointerToResCapInput[col].MTTF << 
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) << 
pointerToResCapInput[col].minTTF << setw(14) << pointerToResCapInput[col].maxTTF 
<< setw(14) << pointerToResCapStatsIntegrated[col].failureRate << setw(14) << 
"LOGNORMAL" << setw(15) << pointerToResCapInput[col].logMTTF << setw(15) << 
pointerToResCapInput[col].logSIGMA << std::endl;
                    }

                    else
                    {
                          outputIdentifier << "Beta" << setw(14) << "Eta" << 
std::endl;
                          outputIdentifier << pointerToResCapInput[col].MTTF << 
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) << 
pointerToResCapInput[col].minTTF << setw(14) << pointerToResCapInput[col].maxTTF 
<< setw(14) << pointerToResCapStatsIntegrated[col].failureRate << setw(14) << 
"WEIBULL" << setw(15) << pointerToResCapInput[col].beta << setw(15) << 
pointerToResCapInput[col].eta << std::endl;
                    }

                    outputIdentifier.close();
                    outputIdentifier.clear();

                    fileName.erase();
                    fileName.clear();

                    fileName = buffer;

                    //fileName.append("Component Reliability\\");
                    fileName.append(identifier);
                    fileName.append(" Means.txt");

                    outputIdentifier.open(fileName.c_str(), ios::out);
                    outputIdentifier << "MTTF" << setw(15) << "1.5 x sigma" <<
setw(15) << "MTTRepair" << setw(15) << "MTTReplace" << std::endl;

                    if(pointerToResCapStatsIntegrated[col].distribution == 1)
                    {
                           outputIdentifier << pointerToResCapInput[col].MTTF <<
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) <<
pointerToResCapInput[col].inRepair << setw(15) <<
pointerToResCapInput[col].inReplace << std::endl;
                    }

                    else if(pointerToResCapStatsIntegrated[col].distribution == 2)
                    {
                           outputIdentifier << pointerToResCapInput[col].MTTF <<
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) <<
pointerToResCapInput[col].inRepair << setw(15) <<
pointerToResCapInput[col].inReplace << std::endl;
                    }

                    else if(pointerToResCapStatsIntegrated[col].distribution == 3)
                    {
                           outputIdentifier << pointerToResCapInput[col].MTTF <<
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) <<
pointerToResCapInput[col].inRepair << setw(15) <<
pointerToResCapInput[col].inReplace << std::endl;
                    }

                    else
                    {
                           outputIdentifier << pointerToResCapInput[col].MTTF <<
setw(14) << pointerToResCapInput[col].sigma*1.5 << setw(14) <<
pointerToResCapInput[col].inRepair << setw(15) <<
pointerToResCapInput[col].inReplace << std::endl;
                    }

                    outputIdentifier.close();
                    outputIdentifier.clear();
                    fileName.erase();
                    fileName.clear();

                    fileName = buffer;

                    for(unsigned int comp = 0; comp < thermalComponent.size();
comp++)
                    {
                           if(thermalComponent[comp].partID !=
pointerToResCapStatsIntegrated[col].partID)
                           {
                                  number++;
                           }
                    }

                    if(number == 0)
                    {
                           fileName.append(identifier);
                           fileName.append(" Reliability.txt");
```

```
					outputIdentifier.open(fileName.c_str(), ios::out);

					outputIdentifier << "Time" << setw(15) << "Reliability" 
<< std::endl;

					time = 0;

					for(unsigned int rely = 0; rely < 
componentReliability[col].reliability.size(); rely++)
					{
						outputIdentifier << time << setw(15) << 
componentReliability[col].reliability[rely] << std::endl;
						time+=0.1;
					}

					outputIdentifier.close();
					outputIdentifier.clear();
					fileName.erase();
					fileName.clear();

					fileName = buffer;
				}

				number = 0;
			}

			fileName.append("System Level Lifetime.txt");

			outputIdentifier.open(fileName.c_str(), ios::out);

			outputIdentifier << "Min Fail" << setw(20) << "Fail" << setw(20) << 
"Max Fail" << setw(20) << "Fix" << setw(27) << "Component ID" << std::endl;

			for(unsigned int comp = 0; comp < outputFailTable.size(); comp++)
			{
				outputIdentifier << outputFailTable[comp].minFail << setw(20) 
<< outputFailTable[comp].mean << setw(20) << outputFailTable[comp].maxFail << 
setw(20) << outputFailTable[comp].meanFix << setw(20) << 
outputFailTable[comp].partID[0];

				if(outputFailTable[comp].partID.size() > 1)
				{
					for(unsigned int part = 1; part < 
outputFailTable[comp].partID.size(); part++)
					{
						outputIdentifier << setw(10) << 
outputFailTable[comp].partID[part];
					}
				}

				outputIdentifier << std::endl;
			}

			outputIdentifier.close();
			outputIdentifier.clear();
			fileName.erase();
			fileName.clear();
```

```
		fileName = buffer;

		fileName.append("System Reliability.txt");

		outputIdentifier.open(fileName.c_str(), ios::out);

		outputIdentifier << "Time" << setw(15) << "Reliability" <<
std::endl;

		time = reliable.timeBegin-0.05;
		timeInc = 0.01;

		for(int comp = 0; comp < 5; comp++)
		{
			if(time >-0.01 &&
				time < 0)
			{
				outputIdentifier << "0" << setw(20) << "1" <<
std::endl;
			}

			else if(time >= 0)
			{
				outputIdentifier << time << setw(20) << "1" <<
std::endl;
			}

			time+=timeInc;
		}

		while(increment < reliable.reliability.size())
		{
			outputIdentifier << time << setprecision(5) << setw(20) <<
reliable.reliability[increment] << std::endl;

			time+=timeInc;
			increment++;
		}

		for(int comp = 0; comp < 5; comp++)
		{
			outputIdentifier << time << setw(20) << "0" << std::endl;
			time+=timeInc;
		}

		outputIdentifier.close();
		outputIdentifier.clear();
		fileName.erase();
		fileName.clear();

		fileName = buffer;
	}

	return;
}
```

APPENDIX F

MAINTAINABILITY CODE

```
#include "stdafx.h"

#define PI 3.14159265358979323846264338;

int componentNumberNewValue;

// structures used to hold the final values of the replication or replications
struct statisticsVectors{
	string partID;

	int componentNumber;
	int distribution;

	double mean;
	double sigma;
	double logMean;
	double logSigma;
	double eta;
	double beta;
	double expoMean;
	double expoSigma;
	double weibullMean;
	double weibullSigma;
	double stdDEVTBF;
	double halfWidthTBF;
	double componentFailProb;
	double TTR_total;
	double lowerRange;
	double upperRange;
	double invLowerRange;
	double invUpperRange;
	double distance;
	double failureRate;
	double compFailProbTotal;

	vector <double> reliability;
	vector <double> reliabilityTime;
	vector <double> failureRateAverage;
	vector <double> repair;
	vector <double> repairCDF;
	vector <int> repairRank;

	bool lowestMTBF;
};

// structure containing the elements of each component in a PCB
struct Component{
	int componentNumber;
	int distributionTypeHoldRepair;
	int rank;

	double sigma;
	double inRepair;
	double inReplace;
	double reliability;
	double TTR;					// Components Mean Time To 
Repair
	double normalDistance;
	double exponentialDistance;
	double logNormalDistance;
	double WeibullDistance;
	double normalProbability;
	double exponentialProbability;
	double logNormalProbability;
	double weibullProbability;
	double CDF;
	double value;

	vector <double> inputMTTR;

	bool printRepair;		// component print repair statement
	bool subtractTTR;		// component subtract mttr statement
	bool repairing;				// choice to repair not replace
	bool FAIL;		// components fail declaration if a component 
fails due to network connections
	bool polyNum;
	bool cdfed;
	bool parallelComponent;
};

struct systemMaintainability{
	bool connectionMade;

	double timeBegin;
	double timeEnd;

	vector <double> systemMTTR;
	vector <double> maintainability;
};

// function prototypes (in order of use)
int numberColumns(string columnCount, vector <statisticsVectors> & identifiers);
void importFREPLREPA_files(statisticsVectors *, int numRows, int numCols, string 
*fileName);
void importSystemMTTR(vector <double> & systemRepair, string *fileName);
void CDF_calc(statisticsVectors *, statisticsVectors *, int numRows, int numCols, 
int CDF_TYPE);

double MEAN(Component *, int numRows);
double STDDEV(Component *, int numRows, double mean);
double log_mean(Component *, int n);
double log_sigma(Component *, int n, double logMean);
double betaCalc(Component *, int numRows, double mean, double sigma);
double etaCalc(Component * , int numRows, double beta, double mean);
double lognormalFinalSigma(double mean, double sigma);
double lognormalFinalMean(double mean, double sigma);
double exponentialSigma(double eta);
double exponentialMean(double eta);
double weibullSigma(double beta, double eta);
double weibullMean(double beta, double eta);
```

```
void calculateNewMeans(statisticsVectors *, int numCols);
void normal_probability(Component *, int numRows, double mean, double sigma);
void exponential_probability(Component *, int numRows, double mean);
void lognormal_probability(Component *, int numRows, int n, double sigma, double
mean);
void weibull_probability(Component *, int numRows, double betas, double etas);

double find_distances(Component *,statisticsVectors *, int numRows, double mean,
double sigma, double expoMean, double beta, double eta, double logMean, double
logSigma, double logNormSigma, double logNormMean, int componentPosition, bool
generateNumber);
double find_lowest_distance(Component *,statisticsVectors *, int numRows, double
sigma, double mean, double expoMean, double beta, double eta, double logMean,
double logSigma, double logNormSigma, double logNormMean, int componentPosition,
bool generateNumber);
double randomNumberGenerator(Component *,statisticsVectors *, int numRows, double
min, double max, int distributionType, double mean, double sigma, double expoMean,
double beta, double eta, double logMean, double logSigma, double logNormSigma,
double logNormMean, int componentPosition, bool generateNumber);
double NRoot(double num, double root);
double find_erf(double mean, double sigma, double randomProb, int normalType);
void findNewValue(Component *, statisticsVectors *, int numCols, statisticsVectors
*, int componentNumberNewValue, int currentRowLength, bool generateNumber);
void chooseDistribution(Component *, int distributionType);

double PCBLifetime(statisticsVectors *, int numCols);
void standdevTBF(statisticsVectors *, int numCols);
void halfWidth(statisticsVectors *, int numCols);
void failureRate(statisticsVectors *, statisticsVectors *, int numCols, double
runLength);
void componentFailProb(statisticsVectors *, int numCols);

void maintainability_calc(statisticsVectors *, statisticsVectors *, vector
<systemMaintainability> & componentMaintainability, int numCols, double runLength,
bool calcMain);
double finalNormal(statisticsVectors *, int componentPosition, double time, double
mean, double sigma);
double finalExponential(statisticsVectors *, int componentPosition, double time,
double expoMean);
double finalLognormal(statisticsVectors *, int componentPosition, double logMean,
double logSigma, double time);
double finalWeibull(statisticsVectors *, int componentPosition, double time,
double etas, double betas);
void systemMaintainable(systemMaintainability &, statisticsVectors *, int numCols,
double runLength, vector <double> & systemRepair);

void outputFinalTable(statisticsVectors *, int numCols, string fileName, vector
<systemMaintainability> & componentMaintainability, systemMaintainability &,
double runLength);
void manual_TTF_TTR_REPLACE(Component *, int numCols, int type);
void MTTF_total(statisticsVectors *, statisticsVectors *, Component *, int
numCols, int replications, double numberOfReplications);
double errorFunction(double x);

int main(int argc, char* argv[])
{
       int partsFailed = 0;
```

```
	int fileChoice = 1;				// User to choice to end or 
continue the program
	int repairReplace = 0;				// Choice of repair or 
replace
	int replication = 0;			// current replication number
	int numRows = -1;				// number of rows
	int numCols=0;					// number of columns 
(components)
	int compon = 0;
	int manualChoice = 0;
	int failedPosition = 0;

	double time = 0;				// holds the current time
	double timeInc = 0.1;				// increments the times
	double mean;
	double sigma;
	double logMean;
	double logSigma;
	double beta;					// shape parameter
	double eta=0;					// scale parameter
	double runLength = 50;				// total runtime of the 
sequence
	double expoMean;
	double expoSigma;
	double logNormMean;
	double logNormSigma;
	double wiebullMean;
	double wiebullSigma;

	bool generateNumber = true;

	string mttr;
	string Line;
	string Repair_filename;
	string MTTR_fileName;
	string outputDirectory = argv[3];

	vector <double> firstFail;

	vector <double> systemRepair;

	fstream TTR_FILE;
	fstream TTR_fileRead;

	Component * pointerToResCapLife;					// 
containes values for subtracting and failing/fixing components
	Component * pointerToResCapCalculation;				// 
contains calculations to find a new value for a component

	statisticsVectors * pointerToResCapStats;			// contains 
all data (sample and random) obtained within a single replication
	statisticsVectors * pointerToResCapStatsIntegrated;		// contains 
all data (sample and random) obtained throughout ALL replications

	vector <systemMaintainability> componentMaintainability;
	systemMaintainability maintain;
	vector <statisticsVectors> identifiers;
```

```
        string componentMaintain;

        // random seed generator
        srand(GetTickCount());

        // jump to new replication

        fileChoice = 1;
        numRows = -1;

        // prompt user to specify name of file to open and open file
        // file contains TTF values
        while(fileChoice != 2)
        {
                // if on the initial replication
                if(replication == 0)
                {
                        componentMaintain = argv[1];
                }

                // open TBF file
                TTR_FILE.open(componentMaintain.c_str());

                // if the user chooses to open a file
                if(fileChoice == 1)
                {
                        // if file is available to open
                        if(TTR_FILE.is_open())
                        {
                                // get the first line in the file
                                while (getline(TTR_FILE, Line, '\n'))
                                {
                                        if(Line.length() > 0)
                                        {
                                                TTR_FILE.close();
                                                TTR_FILE.clear();
                                                break;
                                        }
                                }
                        }

                        // call numberColumns function
                        numCols = numberColumns(Line, identifiers);

                        // array of structures containing the values of statistical
calculations
                        pointerToResCapStats = new statisticsVectors[numCols];

                        // if on the initial replication
                        if(replication == 0)
                        {
                                pointerToResCapStatsIntegrated = new
statisticsVectors[numCols];
                        }

                        // reopen TBF file
                        TTR_FILE.open(componentMaintain.c_str());
```

```
                    // if file is available to open
                    if(TTR_FILE.is_open())
                    {
                           // find the number of rows in the input files
                           while (getline(TTR_FILE, Line,'\n'))
                           {
                                  numRows++;
                           }

                           // resize the vectors in statisticalVectors to the
number of rows in the file
                           for(int col = 0; col < numCols; col++)
                           {

       pointerToResCapStats[col].repair.resize(numRows);

       pointerToResCapStats[col].repairCDF.resize(numRows);

       pointerToResCapStats[col].repairRank.resize(numRows);

                                  if(replication == 0)
                                  {

       pointerToResCapStatsIntegrated[col].repairRank.resize(numRows);

       pointerToResCapStatsIntegrated[col].repair.resize(numRows);
                                  }
                           }

                           //close file
                           TTR_FILE.close();

                           // set fileChoice to 2 as to prevent re-looping
                           fileChoice = 2;
                    }

                    // clear contents if any are in
                    TTR_FILE.clear();
             }
       }

       Repair_filename = argv[1];

       importFREPLREPA_files(pointerToResCapStats, numRows, numCols,
&Repair_filename);

       MTTR_fileName = argv[2];
       importSystemMTTR(systemRepair, &MTTR_fileName);

       // set component numbers for statistical vectors (must be corresponding
column
       for(int col = 0; col < numCols; col++)
       {
             pointerToResCapStats[col].componentNumber = col+1;
             pointerToResCapStats[col].partID = identifiers[col].partID;
             pointerToResCapStatsIntegrated[col].TTR_total = 0;
             pointerToResCapStatsIntegrated[col].partID =
identifiers[col].partID;
```

```
	}

	// sort the Component structure in ascending order of fixed input file 
values
	for (int i=0; i<numCols; i++)
	{
		std::sort(pointerToResCapStats[i].repair.rbegin(), 
pointerToResCapStats[i].repair.rend(), std::greater<double>());
	}

	for(int q = 0; q < numRows; q++)
	{
		for(int j = 0; j < numCols; j++)
		{
			pointerToResCapStatsIntegrated[j].repair[q] = 
pointerToResCapStats[j].repair[q];
		}
	}

	for(int col = 0; col < numCols; col++)
	{
		pointerToResCapStatsIntegrated[col].componentNumber = 
pointerToResCapStats[col].componentNumber;
	}

	// call the CDF_calc function and calculate the CDF of each component 
vector type
	for(int CDF_TYPE = 0; CDF_TYPE < 2; CDF_TYPE++)
	{
		CDF_calc(pointerToResCapStats, pointerToResCapStatsIntegrated, 
numRows, numCols, CDF_TYPE);
	}

	for(int q = 0; q < numRows; q++)
	{
		for(int j = 0; j < numCols; j++)
		{
			pointerToResCapStatsIntegrated[j].repairRank[q] = 
pointerToResCapStats[j].repairRank[q];
		}
	}

	// create array for lifetime sequence
	pointerToResCapLife = new Component[numCols];

	// initialize the structure elements in the pointerToResCapLife array
	for(int col = 0; col < numCols; col++)
	{
		// values to be used and implented in the program
		pointerToResCapLife[col].componentNumber = col+1;
		pointerToResCapLife[col].FAIL = false;
		pointerToResCapLife[col].subtractTTR = false;
		pointerToResCapLife[col].printRepair = false;
		pointerToResCapLife[col].repairing = false;
		pointerToResCapLife[col].polyNum = false;
		pointerToResCapLife[col].cdfed = false;
	}
	// component position in statisticalVectors;
	for(int q = 0; q < numCols; q++)
	{
		// array used for the calculations to generate a new TBF, TTR, or
Replace
		pointerToResCapCalculation = new Component[numRows];

		// assign component numbers for use in the new variable calculations
		for(int s = 0; s < numRows; s++)
		{
			pointerToResCapCalculation[s].componentNumber =
pointerToResCapStats[q].componentNumber;
		}

		for(int s = 0; s < numRows; s++)
		{
			pointerToResCapCalculation[s].value =
pointerToResCapStats[q].repair[s];
			pointerToResCapCalculation[s].CDF =
pointerToResCapStats[q].repairCDF[s];
			pointerToResCapCalculation[s].rank =
pointerToResCapStats[q].repairRank[s];
		}

		// find mean of all MTBF
		mean = MEAN(pointerToResCapCalculation, numRows);

		// find standard deviation of all values
		sigma = STDDEV(pointerToResCapCalculation, numRows, mean);

		// find the log mean
		logMean = log_mean(pointerToResCapCalculation, numRows);

		// find the standard deviation of the logs
		logSigma = log_sigma(pointerToResCapCalculation, numRows, logMean);

		// solve for beta
		beta = betaCalc(pointerToResCapCalculation, numRows, mean, sigma);

		// solve for eta
		eta = etaCalc(pointerToResCapCalculation, numRows, beta, mean);

		// find the area below the normal curve (normalProbability) through
interpolation of an input z table
		normal_probability(pointerToResCapCalculation, numRows, mean,
sigma);

		expoMean = exponentialMean(eta);
		expoSigma = exponentialSigma(eta);

		// solve probabilities for exponential
		exponential_probability(pointerToResCapCalculation, numRows,
expoMean);

		logNormMean = lognormalFinalMean(logMean, logSigma);
		logNormSigma = lognormalFinalSigma(logMean, logSigma);
```

```
              // solve for the lognormal probability (Use log mean and log
standard deviation)
              lognormal_probability(pointerToResCapCalculation, numRows, numCols,
logNormMean, logNormSigma);

              wiebullMean = weibullMean(beta, eta);
              wiebullSigma = weibullSigma(beta, eta);

              // solve the weibull probability
              weibull_probability(pointerToResCapCalculation, numRows, beta, eta);

              pointerToResCapStatsIntegrated[q].beta = beta;
              pointerToResCapStatsIntegrated[q].eta = eta;
              pointerToResCapStatsIntegrated[q].logMean = logNormMean;
              pointerToResCapStatsIntegrated[q].logSigma = logNormSigma;

              // create a new time to repair
              pointerToResCapLife[q].TTR =
find_distances(pointerToResCapCalculation, pointerToResCapStats, numRows, mean,
sigma, expoMean, beta, eta, logMean, logSigma, logNormSigma, logNormMean, q,
generateNumber);

              // save the best fit distribution of that component for repair
              pointerToResCapLife[q].distributionTypeHoldRepair =
pointerToResCapCalculation[0].distributionTypeHoldRepair;

              pointerToResCapStatsIntegrated[q].distribution =
pointerToResCapLife[q].distributionTypeHoldRepair;

              // reset the values used
              mean = 0;
              sigma = 0;
              logMean = 0;
              logSigma = 0;
              eta = 0;
              beta = 0;

              // delete the pointerToResCapCalculation structure array due to its
continuous reuse
              delete []pointerToResCapCalculation;
       }

       componentMaintainability.resize(numCols);

       // allow the user to check the reliability of a component of their choosing
as well as the time
       maintainability_calc(pointerToResCapStats, pointerToResCapStatsIntegrated,
componentMaintainability, numCols, runLength, true);

       systemMaintainable(maintain, pointerToResCapStatsIntegrated, numCols,
runLength, systemRepair);

       outputFinalTable(pointerToResCapStatsIntegrated, numCols, outputDirectory,
componentMaintainability, maintain, runLength);

       // END PROGRAM
       return 0;
}
```

```
// name: numberColumns
// inputs: columnCount = string = entire first line of the input file
// outputs: numCols = int = number of words in that column
// description: finds the number of columns in an input file by taking in the
entire first row
//                          and incrementing a counter every time a new word is
found after a whitespace
int numberColumns(string columnCount, vector <statisticsVectors> & identifiers)
{
       // number of columns
       int numCols=0;

       // converts the string into a stringstream for string manipulation
       // converts a string into a stream interface
       stringstream ss(columnCount);
       string word;

       // before every white space read in word, and increment number of columns
       while( ss >> word )
       {
              numCols++;
              identifiers.resize(numCols);
              identifiers[numCols-1].partID = word;
       }

       // return number of columns (# of columns = # of components)
       return numCols;
}

// name: importFREPLREPA_files
// inputs: pointerToResCapStats = statisticsVectors = contains the 6 tables for
each replication
//                   numRows = int = number of rows contained in each components
TBF, TBR, or replace
//                   numCols = int = number of resistors
//                   fileName = string = name of the file the user specified
//                   whatToFill = int = determines which table to fill (TBF, TBF,
Replace)
// Description: opens a user specified file and copies all the data from the table
into the tables
//                          created in the statisticsVectors struct
void importFREPLREPA_files(statisticsVectors * pointerToResCapStats, int numRows,
int numCols, string *fileName)
{
       fstream FILE_IN;
       string Line;
       // try to reopen a new file

       string FILE = *fileName;

       // open file
       FILE_IN.open(FILE.c_str());

       // if file is available to open
       if(FILE_IN.is_open())
       {
```

```
		// run through the program column by column, then row by row and 
fill the specified vector
		while(getline(FILE_IN,Line,'\n'))
		{
			for(int row = 0; row < numRows; row++)
			{
				for(int col = 0; col < numCols; col++)
				{
					FILE_IN >> 
pointerToResCapStats[col].repair[row];
				}
			}
		}

		//close file
		FILE_IN.close();
	}

	FILE_IN.clear();

	return;
}

void importSystemMTTR(vector <double> & systemRepair, string *fileName)
{
	fstream FILE_IN;
	double MTTR;

	string Line;

	string FILE = *fileName;

	// open file
	FILE_IN.open(FILE.c_str());

	// if file is available to open
	if(FILE_IN.is_open())
	{
		// run through the program column by column, then row by row and 
fill the specified vector
		while(getline(FILE_IN,Line,'\n'))
		{
			FILE_IN >> MTTR;

			systemRepair.push_back(MTTR);
		}
	}

	int size = systemRepair.size()-1;

	systemRepair.erase(systemRepair.begin() + size);
	systemRepair.erase(systemRepair.begin() + size-1);

	FILE_IN.close();
	FILE_IN.clear();

	return;
}
```

```
// name: CDF_calc
// inputs: pointerToResCapStats = statisticsVectors = contains the 3 filled and 3
empty tables (to be filled)
//                  pointerToResCapStatsIntegrated = statisticsVectors = contains
the 3 final output tables from all replications
//                  numRows = int = number of rows contained in each component
column
//                  numCols = int = number of resistors (number of columns)
//                  CDF_TYPE = int = determines what the program should be taking
the CDF of
// description: takes in the data from one component (either TTF, TTR, or replace)
and finds the CDF of that
//                          component
void CDF_calc(statisticsVectors * pointerToResCapStats, statisticsVectors *
pointerToResCapStatsIntegrated, int numRows, int numCols, int CDF_TYPE)
{
	int i;
	int sameValue = 0;
	Component * CDF;

	// solve the CDF
	for(int cols = 0; cols < numCols; cols++)
	{
		// create a new CDF array of Component structure
		CDF = new Component[numRows];

		// fill rows
		for(int s = 0; s < numRows; s++)
		{
			// fille CDF with TTR
			if( CDF_TYPE == 1)
			{
				CDF[s].value = pointerToResCapStats[cols].repair[s];
			}

			// initialize other structure elements required in CDF
calculation
			CDF[s].polyNum = false;
			CDF[s].cdfed = false;
		}

		// solves the CDF for each of the components based on ascending
order
		for(int q = 0; q < numRows; q++)
		{
			// current rank
			i = q + 1;

			// check to see if and which components have multiple values
that are the same
			for(int L = q+1; L < numRows; L++)
			{
				if(CDF[q].value == CDF[L].value &&
					!CDF[L].polyNum)
				{
					sameValue++;
```

```
                                        CDF[L].polyNum = true;
                                }
                        }

                        // if 2 or more part have the same MTBF then the CDF will be
the sum of both resistors
                        if(sameValue > 0 &&
                                !CDF[q].cdfed)
                        {
                                CDF[q].CDF = (i+sameValue)/(numRows+1);
                                CDF[q].cdfed = true;
                                CDF[q].rank = i+sameValue;

                                for(int j = 0; j < numRows; j++)
                                {
                                        if(CDF[q].value == CDF[j].value &&
                                                q!=j)
                                        {
                                                CDF[j].CDF = CDF[q].CDF;
                                                CDF[j].rank = CDF[q].rank;
                                                CDF[j].cdfed = true;
                                        }
                                }
                        }

                        // otherwise the CDF should increment accordingly with the
rank
                        else
                        {
                                if(!CDF[q].cdfed)
                                {
                                        CDF[q].rank = i;
                                        CDF[q].CDF = i/(numRows+1);
                                        CDF[q].cdfed = true;
                                }
                        }

                        sameValue = 0;
                }

                for(int s = 0; s < numRows; s++)
                {
                        // fill the TTR CDF vector with data obtained
                        if(CDF_TYPE == 1)
                        {
                                pointerToResCapStats[cols].repairCDF[s] = CDF[s].CDF;
                                pointerToResCapStats[cols].repairRank[s] = CDF[s].rank;
                        }
                }

                // delete dynamic array for reuse
                delete []CDF;
        }

        // return NULL
        return;
}
```

```
// function name: MEAN
// inputs: pointerToResCapCalculation - pointer to Component - pointer to
resistor/capacitor Component structure
//                    numRows = int = number of values in components
randomNumberType
// output: mean - double - mean value of the MTBF of all copmonents
// Description: takes in all the MTBF's of the components and finds the mean
double MEAN(Component * pointerToResCapCalculation, int numRows)
{
       int n;
       double sum = 0;
       double mean;

       n = numRows;;

       // finds the total sum of the component
       for(int q = 0; q < numRows; q++)
       {
              sum += pointerToResCapCalculation[q].value;
       }

       // divide the total sum of the values by the number of values
       mean = sum/numRows;

       // return the mean back to the main function
       return mean;
}

// function name: STDDEV
// inputs: pointerToResCapCalculation - pointer to Component - pointer to
resistor/capacitor Component structure
//                    numRows = int = number of values contained within a specific
components randomNumberType
//                    mean = double - mean value of all MTBF
// output: mtbf_sigman - double - standard deviation of all values
// Description: calculates the standard deviation of the components MTBF by taking
the square root of the summation of squared
//                           deviations divided by the total number of components
double STDDEV(Component * pointerToResCapCalculation, int numRows, double mean)
{
       double squareDevSum = 0;
       double sigmaSquared;
       double sigma;

       // calculate the total summation of hte squared deviations
       for(int q = 0; q < numRows; q++)
       {
              squareDevSum += ((pointerToResCapCalculation[q].value - mean) *
(pointerToResCapCalculation[q].value - mean));
       }

       // divide the total squared deviations by the number of values
       sigmaSquared = squareDevSum/numRows;

       // standard deviation is equal to the square root of variance (sigmaSquared
in this case)
       sigma = sqrt(sigmaSquared);
```

```
	// return standard deviation to main
	return sigma;
}

// function name: log_MEAN
// inputs: pointerToResCapCalculation - pointer to Component - pointer to 
resistor/capacitor Component structure
//			n = int = number of values contained in the components 
randomNumberType
// output: mean - double - mean value of all copmonents
// Description: takes in all the time values of the components and finds the mean
double log_mean(Component * pointerToResCapCalculation, int n)
{
	double logMean = 0;

	// find the sum of all the values natural logs
	for(int q = 0; q < n; q++)
	{
		logMean += log(pointerToResCapCalculation[q].value);
	}

	// divide the sum by the number of values
	logMean = logMean / n;

	// return the logMean to main
	return logMean;
}

// name: log_sigma
// inputs: pointerToResCapCalculation = Component = structure used in calculating 
parameters
//			n = number of rows in file
//outputs: logSigma = double = standard deviation for logs
// description: calculates the log standard deviation based upon the the log 
values of each component
double log_sigma(Component * pointerToResCapCalculation, int n, double logMean)
{
	double logSigma=0;
	double logValMeanSummation=0;
	double tAfterSquared = 0;

	// calculate the summation of the log of the squared deivations
	for(int q = 0; q < n; q++)
	{
		logValMeanSummation += 
pow((log(pointerToResCapCalculation[q].value)-logMean),2);
	}

	// divide this sum by the number of values
	logValMeanSummation = logValMeanSummation/n;

	// take the square root of the variance (logValMeanSummation)
	logSigma = sqrt(logValMeanSummation);

	// return logSigma to main
	return logSigma;
}
```

```
// function name: betaCalc
// inputs: pointerToResCapCalculation - pointer to Component - pointer to
resistor/capacitor Component structure
//                  numRows - int - number of values contained in a components
vectors
//                  mean = double - mean value
//                  sigma - double - standard deviation between the components
// Description: finds the value of beta for the weibull distribution
double betaCalc(Component * pointerToResCapCalculation, int numRows, double mean,
double sigma)
{
       double beta;
       double inDoubleLog;
       int n = numRows;
       double value;

       // create vectors to hold all the summation calculation values to calculate
the summation later
       std::vector<double> firstSummation(n);
       std::vector<double> secondSummation(n);
       std::vector<double> thirdSummation(n);
       std::vector<double> fourthSummation(n);

       // calculate values to be summed
       for(int q = 0; q < n; q++)
       {
              inDoubleLog = (1.0000 / ( 1.0000 -
(pointerToResCapCalculation[q].rank / (n + 1.0000) ) ) );
              inDoubleLog = log(inDoubleLog);
              inDoubleLog = log(inDoubleLog);

              value = pointerToResCapCalculation[q].value;
              value = log(value);

              firstSummation[q] = value * inDoubleLog;
              secondSummation[q] = inDoubleLog;
              thirdSummation[q] = value;
              fourthSummation[q] = value*value;
       }

       double sumOfFirst=0;
       double sumOfSecond=0;
       double sumOfThird=0;
       double sumOfFourth=0;

       // calculate the sums
       for(unsigned int q = 0; q < firstSummation.size(); q++)
       {
              sumOfFirst += firstSummation[q];
              sumOfSecond += secondSummation[q];
              sumOfThird += thirdSummation[q];
              sumOfFourth += fourthSummation[q];
       }

       // use thes summations in the beta calculation equation
       beta = (((n * sumOfFirst) - (sumOfSecond * sumOfThird)) / ((n *
sumOfFourth) - (sumOfThird * sumOfThird)));
```

```
        // clear the vector contents of each for reuse
        firstSummation.clear();
        secondSummation.clear();
        thirdSummation.clear();
        fourthSummation.clear();

        // return beta to the main function
        return beta;
}

// name: etaCalc
// inputs: pointerToResCapCalculation = Component = structure used in calculating
parameters
//                  numRows = int = number of rows in TTF file
//                  beta = double = shape parameter
//                  mean = double = mean of all values in a components input file
// outputs: eta = double = scale parameter
// description: obtains the components eta based on the values
double etaCalc(Component * pointerToResCapCalculation, int numRows, double beta,
double mean)
{
        double eta;
        double n = numRows;
        double value=0;
        double i = 1.00000;
        double root = 1/beta;

        // sum all values in column
        for(int q = 1 ; q <= numRows;q++)
        {
               value += pow(pointerToResCapCalculation[q-1].value,beta);
        }

        // divide the sum by the number of values
        eta = value / n;

        // eta^(1/beta)
        eta = pow(eta,root);

        // return eta to the main function
        return eta;
}

// name: weibullMean
// inputs: beta - double - beta of the values of the given component
//                  eta - double - eta of hte values of the given component
// outputs: mean - double - mean of the calculated weibull distribution
// description: calculates the mean of a component that follows a weibull
distribution
double weibullMean(double beta, double eta)
{
        float roundedGammaAlpha;
        double gammaAlpha;
        double removePara = 1;
        double table_alpha;
        double table_gamma;
        double mean;
        double gammaTotal;
        fstream gammaOpen;

        string line;

        gammaAlpha = 1 + 1/beta;

        vector <double> alpha;
        vector <double> gamma;

        while(gammaAlpha >= 1.995)
        {
                gammaAlpha--;

                removePara *= gammaAlpha;
        }

        roundedGammaAlpha = floor(gammaAlpha * 100.0 + 0.5) / 100.0;

        gammaOpen.open("gammaTable.txt");

        if(gammaOpen.is_open())
        {
                while(getline(gammaOpen,line,'\n'))
                {
                        gammaOpen >> table_alpha >> table_gamma;

                        alpha.push_back(table_alpha);
                        gamma.push_back(table_gamma);
                }

                gammaOpen.close();
                gammaOpen.clear();
        }

        for(unsigned int table = 1; table < alpha.size(); table++)
        {
                if(roundedGammaAlpha == alpha[table])
                {
                        table_gamma = gamma[table];
                }
        }

        gammaTotal = table_gamma*removePara;

        mean = gammaTotal * eta;

        return mean;
}

// name: weibullSigma
// inputs: beta - double - beta for corresponding component
//                 eta - double - eta for corresponding component
// outputs: sigma - double - standard deviation of component
// description: calculates the standard devation of a component given the shape
//                         and scale parameters
double weibullSigma(double beta, double eta)
{
```

```
double roundedGammaAlpha1;
double roundedGammaAlpha2;
double variance;
double sigma;
double table_gamma;
double table_alpha;
double gamma1;
double gamma2;
double gammaTotal;
double remove1 =1;
double remove2 =1;

fstream gammaOpen;

string line;

vector <double> alpha;
vector <double> gamma;

eta = eta * eta;

gamma1 = 1 + 2/beta;
gamma2 = 1 + 1/beta;

while(gamma1 >= 1.995)
{
       gamma1--;

       remove1 *= gamma1;
}

while(gamma2 >= 1.995)
{
       gamma2--;

       remove2 *= gamma2;
}

roundedGammaAlpha1 = floor(gamma1 * 100 + 0.5) / 100;
roundedGammaAlpha2 = floor(gamma2 * 100 + 0.5) / 100;

gammaOpen.open("gammaTable.txt");

if(gammaOpen.is_open())
{
       while(getline(gammaOpen,line,'\n'))
       {
              gammaOpen >> table_alpha >> table_gamma;

              alpha.push_back(table_alpha);
              gamma.push_back(table_gamma);
       }

       gammaOpen.close();
       gammaOpen.clear();
}

for(unsigned int table = 1; table < alpha.size(); table++)
```

```
	{
		if(roundedGammaAlpha1 == alpha[table])
		{
			gamma1 = gamma[table];
		}
	}

	for(unsigned int table = 1; table < alpha.size(); table++)
	{
		if(roundedGammaAlpha2 == alpha[table])
		{
			gamma2 = gamma[table];
		}
	}

	gamma1 = gamma1*remove1;
	gamma2 = gamma2*remove2;

	gamma2 = gamma2 * gamma2;

	gammaTotal = gamma1 - gamma2;

	variance = eta * gammaTotal;

	sigma = sqrt(variance);

	return sigma;
}

// name: exponentialMean
// inputs: eta - double - eta value corresponding to component
// outputs: mean - double - mean of the exponential component
// description: calculates the mean of a component which follows an exponential
//                          distribution, mean will be equal to eta
double exponentialMean(double eta)
{
	double mean;

	mean = eta;

	return mean;
}

// name: exponentialSigma
// inputs: eta - double - eta value corresponding ot component
// outputs: sigma - double - standard deviation of component
// description: calculates the standard deviation of a component following
//                          an exponential distribution, mean is equal to eta
double exponentialSigma(double eta)
{
	double sigma;

	sigma = eta * eta * 3;

	sigma = sqrt(sigma);

	return sigma;
}
```

```
// name: lognormalFinalMean
// inputs: mean - double - mean of a lognormal distribution
//                 sigma - double - standard deviation of a component
// outputs: mean - double - final lognormal mean of component
// description: calculates the mean of a component that is following the
//                       lognormal distribution
double lognormalFinalMean(double mean, double sigma)
{
       double meanNew;

       meanNew = exp((mean + ((sigma * sigma)/2)));

       return meanNew;
}

// name: lognormalFinalSigma
// inputs: mean - double - log mean of the current component
//                 sigma - double - log stand dev of the current component
// outputs: std - double - standard deviation of a component
// description: calculates the standard deviation of a component which follows
//                       the lognormal distribution
double lognormalFinalSigma(double mean, double sigma)
{
       double stdev;

       stdev = exp(((2*mean)+(sigma*sigma))) * (exp((sigma * sigma)) - 1);

       stdev = sqrt(stdev);

       return stdev;
}

// function name: normal_probability
// inputs: pointerToResCapCalculation - pointer to Component - pointer to
resistor/capacitor Component structure
//                 numRows - int - number of values contained in a components
randomNumberType
//                 mean = double - mean value of all MTBF
//                 sigma - double - standard deviation between the components
MTBF
// Description: goes into the formatted zTable and interpolates a value of z if it
is not found as "nice"
//                       value on the z table
void normal_probability(Component * pointerToResCapCalculation, int numRows,
double mean, double sigma)
{
       double errorFunc;

       // CDF(x) = 0.5 * (1 + erf((x-mu)/(sigma(root(2))))
       for(int q = 0; q < numRows; q++)
       {
              errorFunc = ((pointerToResCapCalculation[q].value - mean) / (sigma *
sqrt(2.0)));

              pointerToResCapCalculation[q].normalProbability = 0.5 * ( 1 +
errorFunction(errorFunc));
       }
```

```
	return;
}

// function name: exponential_probability
// inputs: pointerToResCapCalculation - pointer to Component - pointer to
resistor/capacitor Component structure
//			NumberResCapLines - int - number of lines containing
components that are not chips
//			totalResistors - int - number of components in the PCB
//			mean = double - mean value of all MTBF
// Description: finds the exponential probabilities with the exponential
cumulative distribution function
void exponential_probability(Component * pointerToResCapCalculation, int numRows,
double mean)
{
	double exponent;

	// finds the exponential probability F(x)
	for(int q = 0; q < numRows; q++)
	{
		exponent = -(pointerToResCapCalculation[q].value/mean);

		pointerToResCapCalculation[q].exponentialProbability = 1 -
exp(exponent);
	}

	return;
}

// function name: lognormal_probability
// inputs: pointerToResCapCalculation - pointer to Component - pointer to
resistor/capacitor Component structure
//			numRows = int = number of values contained in a components
vector type
//			logMean = double - mean log value of all values
//			logSigma - double - standard deviation of the log between the
components values
// Description: finds the lognormal CDF probabilities through calculations
void lognormal_probability(Component * pointerToResCapCalculation, int numRows,
int n, double logMean, double logSigma)
{
	double errorFunc;

	// calculate the lognormal probability of each function using boost
libraries for error function calculation
	for(int q = 0; q < numRows; q++)
	{
		errorFunc = ((log(pointerToResCapCalculation[q].value) - logMean) /
(logSigma * sqrt(2.0)));

		pointerToResCapCalculation[q].logNormalProbability = 0.5 * ( 1 +
errorFunction(errorFunc));
	}

	return;
}
```

```
// function name: weibull_probability
// inputs: pointerToResCapCalculation - pointer to Component - pointer to
resistor/capacitor Component structure
//                  numRows - int - number of values contained within a specific
components vector
//                  betas = double = shape parameter
//                  etas = double = scale parameter
// Description: Finds the CDF of each component using the weibull distribution
void weibull_probability(Component * pointerToResCapCalculation, int numRows,
double betas, double etas)
{
       double power;

       //CDF(x) = 1 - exp(-(x/eta)^beta)
       for( int q = 0; q < numRows; q++)
       {
              power = pointerToResCapCalculation[q].value/etas;
              power = pow(power,betas);
              power = -power;
              pointerToResCapCalculation[q].weibullProbability = 1.0000 -
(exp(power));
       }

       return;
}

// function name: errorFunction
// inputs: x - double - value to be calculated within the error function
parameters
// outputs: function - double - value of x calculated as an error funciton value
// description: calculates a value in terms of the error function
double errorFunction(double x)
{
       /* erf(z) = 2/sqrt(pi) * Integral(0..x) exp( -t^2) dt
       erf(0.01) = 0.0112834772 erf(3.7) = 0.9999998325
       Abramowitz/Stegun: p299, |erf(z)-erf| <= 1.5*10^(-7)
       */
       double a1 = 0.254829592;
       double a2 = -0.284496736;
       double a3 = 1.421413741;
       double a4 = -1.453152027;
       double a5 = 1.061405429;
       double p = 0.3275911;

       int sign = 1;

       if(x < 0)
       {
              sign = -1;
       }

       x = fabs(x);

       double t = 1.0/(1.0 + p * x);
       double y = 1.0 - (((((a5 * t +a4) * t) + a3)*t + a2)*t + a1)*t*exp(-x*x);

       y = sign * y;
       return y;
}

// function name: find_distances
// inputs: pointerToResCapCalculation - pointer to Component - pointer to 
resistor/capacitor Component structure
//                   numRows - int - number of lines containing components that are 
not chips
//                   mean = double = mean of the values in a components 
randomNumberType
//                   sigma = double = standard deviation of the values in a 
components randomNumberType
//                   beta = double = beta value for weibull calculation
//                   eta = double = eta value for weibull calculation
//                   logMean = double = mean of the logs
//                   logSigma = double = standard deviation of the logs
//                   componentPosition = int = position of the component in the 
array
//                   randomNumberType = int = type of random number (TBF, TTR, 
replace) the function should be retrieving
//                   generateNumber = bool = determines when it runs through this 
function if it should also generate a probability as well as the CDF
// outputs: value - double - random value that was generated
// Description: Finds the distance for all components of types of distributions, 
then goes into the lowest
//                          distance function, and returns the lowest distance
double find_distances(Component * pointerToResCapCalculation, statisticsVectors * 
pointerToResCapStats, int numRows, double mean, double sigma, double expoMean,
       double beta, double eta, double logMean, double logSigma, double 
logNormSigma, double logNormMean, int componentPosition, bool generateNumber)
{
       double normalDistance;
       double exponentialDistance;
       double logNormalDistance;
       double weibullDistance;
       double value;

       // find distances D = |Fo - Fn|
       for(int q = 0; q < numRows; q++)
       {
              // Distance = | Theoretical Value - Empirical Value |
              normalDistance = pointerToResCapCalculation[q].CDF - 
pointerToResCapCalculation[q].normalProbability;
              exponentialDistance = pointerToResCapCalculation[q].CDF - 
pointerToResCapCalculation[q].exponentialProbability;
              logNormalDistance = pointerToResCapCalculation[q].CDF - 
pointerToResCapCalculation[q].logNormalProbability;
              weibullDistance = pointerToResCapCalculation[q].CDF - 
pointerToResCapCalculation[q].weibullProbability;

              // take the absolute value of the calculated distance
              pointerToResCapCalculation[q].normalDistance = fabs(normalDistance);
              pointerToResCapCalculation[q].exponentialDistance = 
fabs(exponentialDistance);
              pointerToResCapCalculation[q].logNormalDistance = 
fabs(logNormalDistance);
              pointerToResCapCalculation[q].WeibullDistance = 
fabs(weibullDistance);
```

```
	}

	// go to lowest distance function to find a random number based on the best 
fit distribution
	value = find_lowest_distance(pointerToResCapCalculation, 
pointerToResCapStats, numRows, sigma, mean, expoMean, beta, eta, logMean, 
logSigma, logNormSigma, logNormMean, componentPosition, generateNumber);

	// if a new number should be generated, return that value
	if(generateNumber)
	{
		return value;
	}

	// otherwise return 0
	else
	{
		return 0;
	}
}

// function name: find_lowest_distance
// inputs: pointerToResCapCalculation - pointer to Component - pointer to 
resistor/capacitor Component structure
//		pointerToResCapStats = pointer to statisticsVectors = saves 
all values into the table
//		numRows - int - number of lines containing components that are 
not chips
//		mean = double = mean of the values in a components 
randomNumberType
//		sigma = double = standard deviation of the values in a 
components randomNumberType
//		beta = double = beta value for weibull calculation
//		eta = double = eta value for weibull calculation
//		logMean = double = mean of the logs
//		logSigma = double = standard deviation of the logs
//		componentPosition = int = position of the component in the 
array
//		randomNumberType = int = type of random number (TBF, TTR, 
replace) the function should be retrieving
//		generateNumber = bool = determines when it runs through this 
function if it should also generate a probability as well as the CDF
// outputs: value - double - random value that was generated
// Description: finds the highest distance of each distribution type, then finds 
the lowest of those four distances, then
//			the lowest distance will be the distribution that, taht 
component type will follow throughout the lifetime of the
//			replication, then will generate a random value based on 
the distribution type
double find_lowest_distance(Component * pointerToResCapCalculation, 
statisticsVectors * pointerToResCapStats, int numRows, double sigma, 
	double mean, double expoMean, double beta, double eta, double logMean, 
double logSigma, double logNormSigma, double logNormMean, int componentPosition, 
bool generateNumber)
{
	double highestNormalDistance = 
pointerToResCapCalculation[0].normalDistance;
```

```
       double highestExponentialDistance = 
pointerToResCapCalculation[0].exponentialDistance;
       double highestLogNormalDistance = 
pointerToResCapCalculation[0].logNormalDistance;
       double highestWeibullDistance = 
pointerToResCapCalculation[0].WeibullDistance;
       double lowest_distance;
       int distributionType = 0;
       double normalCV;
       double expCV;
       double logCV;
       double weibullCV;
       double numRow = numRows;
       double deleteMe = 0;

       // finds the highest normal distance
       for(int q = 1; q < numRows; q++)
       {
              if(pointerToResCapCalculation[q].normalDistance > 
highestNormalDistance)
              {
                     highestNormalDistance = 
pointerToResCapCalculation[q].normalDistance;
              }
       }

       // finds the highest exponential distance
       for(int q = 1; q < numRows; q++)
       {
              if(pointerToResCapCalculation[q].exponentialDistance > 
highestExponentialDistance)
              {
                     highestExponentialDistance = 
pointerToResCapCalculation[q].exponentialDistance;
              }
       }

       // finds the highest log normal distance
       for(int q = 1; q < numRows; q++)
       {
              if(pointerToResCapCalculation[q].logNormalDistance > 
highestLogNormalDistance)
              {
                     highestLogNormalDistance = 
pointerToResCapCalculation[q].logNormalDistance;
              }
       }

       // finds the highest weibull distance
       for(int q = 1; q < numRows; q++)
       {
              if(pointerToResCapCalculation[q].WeibullDistance > 
highestWeibullDistance)
              {
                     highestWeibullDistance = 
pointerToResCapCalculation[q].WeibullDistance;
              }
       }
       // calculate the critical value of each distribution type (they will all
equal; done simply for readibility)
       normalCV = 1.358 / (sqrt(numRow));
       expCV = 1.358 / (sqrt(numRow));
       logCV = 1.358 / (sqrt(numRow));
       weibullCV = 1.358 / (sqrt(numRow));

       // if the lowest of these four values in normal
       if(highestNormalDistance < highestExponentialDistance &&
              highestNormalDistance < highestLogNormalDistance &&
              highestNormalDistance < highestWeibullDistance)
       {
              // set the lowest normal distance to the lowest distance
              lowest_distance = highestNormalDistance;

              // set distributionType to normal
              distributionType = 1;

              // save the distribution type for that components TBF, TTR, or
replace
              chooseDistribution(pointerToResCapCalculation, distributionType);

              // generate a new number of the program finds it should
              if(generateNumber)
              {
                     return randomNumberGenerator(pointerToResCapCalculation,
pointerToResCapStats, numRows, 0, 1, distributionType, mean, sigma, expoMean,
beta, eta, logMean, logSigma, logNormSigma, logNormMean, componentPosition,
generateNumber);
              }
       }

       // if the exponential distance is lower thna the other distances
       else if( highestExponentialDistance < highestNormalDistance &&
              highestExponentialDistance < highestLogNormalDistance &&
              highestExponentialDistance < highestWeibullDistance)
       {
              // set the lowest distance
              lowest_distance = highestExponentialDistance;

              // save exponential as the best fit distribution
              distributionType = 2;

              // save that distribution type to taht components TBF, TTR< or
replace
              chooseDistribution(pointerToResCapCalculation, distributionType);

              // if a new number should be generated, do it based off the
exponential calculations
              if(generateNumber)
              {
                     return randomNumberGenerator(pointerToResCapCalculation,
pointerToResCapStats, numRows, 0, 1, distributionType, mean, sigma, expoMean,
beta, eta, logMean, logSigma, logNormSigma, logNormMean, componentPosition,
generateNumber);
              }
       }
       // if the lognormal distance is lower than the other 3 distances
       else if(highestLogNormalDistance < highestNormalDistance &&
              highestLogNormalDistance < highestExponentialDistance &&
              highestLogNormalDistance < highestWeibullDistance)
       {
              // sets the lowest log normal distance to the lowest distance
              lowest_distance = highestLogNormalDistance;

              // set the distributionType to lognormal
              distributionType = 3;

              // save the distribution type to that components TBF, TTR, or
replace
              chooseDistribution(pointerToResCapCalculation, distributionType);

              // generate a new value based on the lognormal distribution
              if(generateNumber)
              {
                     return randomNumberGenerator(pointerToResCapCalculation,
pointerToResCapStats, numRows, 0, 1, distributionType, mean, sigma, expoMean,
beta, eta, logMean, logSigma, logNormSigma, logNormMean, componentPosition,
generateNumber);
              }
       }

       // if weibull distance is less than the other 3 distances
       else if(highestWeibullDistance < highestNormalDistance &&
              highestWeibullDistance < highestExponentialDistance &&
              highestWeibullDistance < highestLogNormalDistance)
       {
              // set the lowest weibull distance to the lowest distance
              lowest_distance = highestWeibullDistance;

              // set distributionType to weibull
              distributionType = 4;

              // save the distribution type for that components TBF, TTR, or
replace
              chooseDistribution(pointerToResCapCalculation, distributionType);

              // generate a new number based on the weibull distribution
              if(generateNumber)
              {
                     return randomNumberGenerator(pointerToResCapCalculation,
pointerToResCapStats, numRows, 0, 1, distributionType, mean, sigma, expoMean,
beta, eta, logMean, logSigma, logNormSigma, logNormMean, componentPosition,
generateNumber);
              }
       }

       return 0;
}

// name: chooseDistribution
// inputs: pointerToResCapCalculation = Component = decides which distribution
type a component should always take
```

```
//                  randomNumberType = int = descriptor determinant of whether it
should be saving the TBF distribution, TTR distribution or replace distribution
//                  distributionType = int = descriptor for which type of
distribution the component should always follow
// Description: finds the initial distribution type that the component will follow
from just the sample data, and finds
//                         the best distribution type, then saves that
distribution type to the component, so the component will
//                         always follow the same distribution type throughout the
replication.
void chooseDistribution(Component * pointerToResCapCalculation, int
distributionType)
{
       // save distribution type for that component
       pointerToResCapCalculation[0].distributionTypeHoldRepair =
distributionType;
}

// name: randomNumberGenerator
// inputs: pointerToResCapCalculation = Component = does the calculations
//                  pointerToResCapStats = statisticsVector = contains all values
for tbf, repair, and replace
//                  numRows = int = number of rows in ttf file
//                  min = double = minimum possible random number
//                  max = double = maximum possible random number
//                  distributionType = int = type of distribution to solve for
//                  mean = double = mean of components
//                  sigma = double = standard deviation between values of a
component
//                  beta = double = beta value for weibull distribution
//                  eta = double = eta value for weibull distribution
//                  logMean = double = mean of the log values for each component
//                  logSigma = double = standard deviation of the log values for
each component
//                  componentPosition = int = location of component within the
array
// outputs: the final randomly generated variable
// description: determines (based on lowest distance) the type of distribution to
use in order to calculate a time
//                         out of a given randomly generated probability within a
program specfied range (0 to 1)
double randomNumberGenerator(Component * pointerToResCapCalculation,
statisticsVectors * pointerToResCapStats,
       int numRows, double min, double max, int distributionType, double mean,
double sigma, double expoMean, double beta, double eta, double logMean,
       double logSigma, double logNormSigma, double logNormMean, int
componentPosition, bool generateNumber)
{
       double randomNumber = 2;
       int normalType = 0;
       double erf_Y;
       double FRR;

       while(randomNumber >= 1)
       {
              randomNumber = (double)rand() / RAND_MAX;
       }
```

```
switch(distributionType)
{
       //distributionType = normal
case 1:
       normalType = 1;

       // if probability is less than 0.5
       if(randomNumber < 0.5)
       {
              erf_Y = (1 - randomNumber * 2);
       }

       // otherwise
       else
       {
              erf_Y = (randomNumber * 2 - 1);
       }

       // if value is in the range call erf function
       if(erf_Y >= 0 &&
              erf_Y <= 1)
       {
              FRR = find_erf(mean, sigma, erf_Y, normalType);
       }

       // return value to main
       return FRR;

       break;

       //distributionType = exponential
case 2:
       // calculate new value based on exponential
       FRR = -(expoMean*log((1-randomNumber)));

       // return value to main
       return FRR;

       break;

       //distributionType = lognormal
case 3:
       // if random number is less than 0.5
       if(randomNumber < 0.5)
       {
              erf_Y = (1 - randomNumber * 2);
       }

       // otherwise
       else
       {
              erf_Y = (randomNumber * 2 - 1);
       }

       //lognormal
       normalType = 2;

       // if random value is in the range
```

```
		if(erf_Y >= 0 &&
			erf_Y <= 1)
		{
			// call find_erf and find value
			FRR = find_erf(logNormMean, logNormSigma, erf_Y, normalType);
		}

		// return value to main
		return FRR;

		break;

		//distributionType = weibull
	case 4:
		// calculate value based on weibull
		FRR = NRoot((log((1/(1-randomNumber)))),beta);
		FRR = eta * FRR;

		// return to main
		return FRR;

		break;

		// should never happen
	default:
		FRR = 0;

		return FRR;
		break;
	}
}

//name: NRoot
// inputs: num = double = value being rooted
//		   root = double = value doing the rooting
// outputs: the root of num
// description: solves the value being rooted, when its not a square root
double NRoot(double num, double root)
{
	root=1/root;//divide by 1 to get 1/root e.g 1/2=0.5
	return(pow(num, root));//raise num to root to get answer
}

// name: find_erf
// inputs: mean = double = mean of all components
//		   sigma = double = standard deviation of all compeonts
//		   randomProb = double = value contained within the error
funciton
//		   normalType = int = normal or lognormal
// outputs: randomValue = double = calculated value of x
// decription: runs through the formatted ERF table and finds the closest value to
it
double find_erf(double mean, double sigma, double random_F_Y, int normalType)
{
	fstream ERFTable;
	string currentLine;
	double randomValue;
	double erfX;
```

```
        double X;
        double x1;
        double erfX1;

        // open inputtable
        ERFTable.open("erfTable.txt", ios::in);

        if(ERFTable.is_open())
        {
                // while the error function table is not at the end
                while(getline(ERFTable, currentLine, '\n'))
                {
                        ERFTable >> X >> erfX >> x1 >> erfX1;

                        if(random_F_Y > 0.999999304)
                        {
                                random_F_Y = 0.999999304;
                        }

                        // if randomProb has found the area in which it is supposed to
be placed give it a value and solve;
                        if((random_F_Y <= erfX &&
                                random_F_Y >= erfX1) &&
                                normalType == 1)
                        {
                                randomValue = X*sigma*sqrt(2.0) + mean;
                        }

                        // if randomProb has found the area in which it is supposed to
be placed give it a value and solve;
                        else if((random_F_Y <= erfX &&
                                random_F_Y >= erfX1) &&
                                normalType == 2)
                        {
                                randomValue = X*sigma*sqrt(2.0) + mean;
                        }
                }
        }

        ERFTable.clear();
        ERFTable.close();

        // return calculated value
        return randomValue;
}

// name: reliability_calc
// inputs: pointerToResCapStats = statisticsVectors = contains data obtained from
that replication
//                      pointerToResCapStatsIntegrated = statisticsVectors = contains
data obtained from all replications
//                      numCols = int = number of columns (components) in the table
(input file)
// description: obtains a new distribution type for all values obtained from the
sample and the lifetime of the replications
//                              and uses that new distribution to determine the
reliability of a component by the mean of the TBF values.
```

```
void maintainability_calc(statisticsVectors * pointerToResCapStats,
statisticsVectors * pointerToResCapStatsIntegrated, vector <systemMaintainability>
& componentMaintainability, int numCols, double runLength, bool calcMain)
{
       // give pointerToResCapCalculation the Component structure
       Component * pointerToResCapCalculation;
       Component * CDF;

       // initializations required to generate the components reliability
       bool generate = false;
       int numRows;
       double mean;
       double sigma;
       double logMean;
       double logSigma;
       double beta;
       double eta;
       int i = 0;
       int sameValue = 0;
       double timeInc = 0.01;

       double lognormalMean;
       double lognormalSigma;
       double wiebullMean;
       double wiebullSigma;
       double expoMean;
       double expoSigma;

       for(int col = 0; col < numCols; col++)
       {
              numRows = pointerToResCapStatsIntegrated[col].repair.size();

              pointerToResCapCalculation = new Component[numRows];
              CDF = new Component[numRows];

              // assign values for use in the new variable calculations
              for(int s = 0; s < numRows; s++)
              {
                     pointerToResCapCalculation[s].componentNumber =
pointerToResCapStatsIntegrated[col].componentNumber;
                     CDF[s].value = pointerToResCapStatsIntegrated[col].repair[s];
                     CDF[s].polyNum = false;
                     CDF[s].cdfed = false;
              }

              std::sort(pointerToResCapStatsIntegrated[col].repair.rbegin(),
pointerToResCapStatsIntegrated[col].repair.rend(), std::greater<double>());

              // solves the CDF for each of the components based on ascending
order
              for(int q = 0; q < numRows; q++)
              {
                     // current rank
                     i = q + 1;

                     // check to see if and which components have multiple values
that are the same
                     for(int L = q+1; L < numRows; L++)
```

```
                    {
                           if(CDF[q].value == CDF[L].value &&
                                  !CDF[L].polyNum)
                           {
                                  sameValue++;

                                  CDF[L].polyNum = true;
                           }
                    }

                    // if 2 or more part have the same MTBF then the CDF will be
the sum of both resistors
                    if(sameValue > 0 &&
                           !CDF[q].cdfed)
                    {
                           CDF[q].CDF = (i+sameValue)/(numRows+1);
                           CDF[q].cdfed = true;

                           for(int j = 0; j < numRows; j++)
                           {
                                  if(CDF[q].value == CDF[j].value &&
                                         q!=j)
                                  {
                                         CDF[j].CDF = CDF[q].CDF;
                                         CDF[j].cdfed = true;
                                  }
                           }
                    }

                    // otherwise the CDF should increment accordingly with the
rank
                    else
                    {
                           if(!CDF[q].cdfed)
                           {
                                  CDF[q].CDF = i/(numRows+1);
                                  CDF[q].cdfed = true;
                           }
                    }

                    sameValue = 0;
             }

             for(int s = 0; s < numRows; s++)
             {
                    pointerToResCapCalculation[s].CDF = CDF[s].CDF;
                    pointerToResCapCalculation[s].value =
pointerToResCapStatsIntegrated[col].repair[s];
                    pointerToResCapCalculation[s].rank =
pointerToResCapStatsIntegrated[col].repairRank[s];
             }

             delete []CDF;

             // find mean
             mean = MEAN(pointerToResCapCalculation, numRows);

             // find standard deviation
```

```
              sigma = STDDEV(pointerToResCapCalculation, numRows, mean);

              // find the log mean
              logMean = log_mean(pointerToResCapCalculation, numRows);

              // find the standard deviation of the logs
              logSigma = log_sigma(pointerToResCapCalculation, numRows, logMean);

              // solve for beta
              beta = betaCalc(pointerToResCapCalculation, numRows, mean, sigma);

              // solve for eta
              eta = etaCalc(pointerToResCapCalculation, numRows, beta, mean);

              expoMean = exponentialMean(eta);
              expoSigma = exponentialSigma(eta);

              lognormalMean = lognormalFinalMean(logMean, logSigma);
              lognormalSigma = lognormalFinalSigma(logMean, logSigma);

              wiebullMean = weibullMean(beta, eta);
              wiebullSigma = weibullSigma(beta, eta);

              pointerToResCapStatsIntegrated[col].beta = beta;
              pointerToResCapStatsIntegrated[col].eta = eta;
              pointerToResCapStatsIntegrated[col].logMean = lognormalMean;
              pointerToResCapStatsIntegrated[col].logSigma = lognormalSigma;
              pointerToResCapStatsIntegrated[col].expoMean = expoMean;
              pointerToResCapStatsIntegrated[col].expoSigma = expoSigma;
              pointerToResCapStatsIntegrated[col].weibullMean = wiebullMean;
              pointerToResCapStatsIntegrated[col].weibullSigma = wiebullSigma;

              // find the area below the normal curve (normalProbability) through
interpolation of an input z table
              normal_probability(pointerToResCapCalculation, numRows, mean,
sigma);

              // solve probabilities for exponential
              exponential_probability(pointerToResCapCalculation, numRows,
expoMean);

              // solve for the lognormal probability (Use log mean and log
standard deviation)
              lognormal_probability(pointerToResCapCalculation, numRows, numCols,
lognormalMean, lognormalSigma);

              // solve the weibull probability
              weibull_probability(pointerToResCapCalculation, numRows, beta, eta);

              // finds a new distribution type for the component with all values
(sample and generated)
              //useless = find_distances(pointerToResCapCalculation,
pointerToResCapStatsIntegrated,
pointerToResCapStatsIntegrated[col].timeBetweenFail.size(), mean, sigma, expoMean,
beta, eta, logMean, logSigma, lognormalSigma, lognormalMean, col, 0, generate);

              pointerToResCapCalculation[0].distributionTypeHoldRepair =
pointerToResCapStatsIntegrated[col].distribution;
```

```
              // if normal
              if(pointerToResCapCalculation[0].distributionTypeHoldRepair == 1)
              {
                     pointerToResCapStatsIntegrated[col].TTR_total = mean;

                     for(double time = 0; time < runLength+timeInc; time+=timeInc)
                     {
                            // calculate reliability based on normal equation

       componentMaintainability[col].maintainability.push_back(finalNormal(pointer
ToResCapStatsIntegrated, col, time, mean, sigma));
                     }
              }

              // if Exponential
              else if(pointerToResCapCalculation[0].distributionTypeHoldRepair ==
2)
              {
                     pointerToResCapStatsIntegrated[col].TTR_total = mean;

                     for(double time = 0; time < runLength+timeInc; time+=timeInc)
                     {
                            // calculate reliability based on exponential equation

       componentMaintainability[col].maintainability.push_back(finalExponential(po
interToResCapStatsIntegrated, col, time, expoMean));
                     }
              }

              // if lognormal
              else if(pointerToResCapCalculation[0].distributionTypeHoldRepair ==
3)
              {
                     pointerToResCapStatsIntegrated[col].TTR_total = mean;

                     for(double time = 0; time < runLength+timeInc; time+=timeInc)
                     {
                            // calculate reliability based on lognormal equation

       componentMaintainability[col].maintainability.push_back(finalLognormal(poin
terToResCapStatsIntegrated, col, logMean, logSigma, time));
                     }
              }

              // if weibull
              else
              {
                     pointerToResCapStatsIntegrated[col].TTR_total = mean;

                     for(double time = 0; time < runLength+timeInc; time +=
timeInc)
                     {
                            // calculate reliability based on weibull equation

       componentMaintainability[col].maintainability.push_back(finalWeibull(pointe
rToResCapStatsIntegrated, col, time, eta, beta));
```

```
				}
			}

			delete []pointerToResCapCalculation;
		}
		//getchar();

		return;
	}

	void systemMaintainable(systemMaintainability & maintain, statisticsVectors *
	pointerToResCapStatsIntegrated, int numCols, double runLength, vector <double> &
	systemRepair)
	{
		double numberFailed = 0;
		double timeInc = 0.01;
		double average = 0;
		double maintainAverage = 0;
		bool maintain_begin = true;
		bool maintain_end = true;

		for(unsigned int comp = 0; comp < systemRepair.size(); comp++)
		{
			average+=systemRepair[comp];
		}

		average = average/systemRepair.size();

		for(double time = 0; time < runLength; time+=0.01)
		{
			for(unsigned int comp = 0; comp < systemRepair.size(); comp++)
			{
				if(systemRepair[comp] <= time)
				{
					numberFailed++;
				}
			}

			maintainAverage = numberFailed/systemRepair.size();

			if(maintainAverage == 1 &&
				maintain_end)
			{
				maintain.timeEnd = time;
				maintain_end = false;
			}

			if(maintainAverage > 0 &&
				maintainAverage < 1)
			{
				maintain.maintainability.push_back(maintainAverage);

				if(maintain_begin)
				{
					maintain.timeBegin = time;

					maintain_begin = false;
				}
			}

			numberFailed = 0;
		}

		return;
}

// name: finalNormal
// inputs: pointerToResCapStatsIntegrated = statisticsVectors = contains all
cumulative data obtained throughout the lifetime of all replications
//			componentPosition = int = element number in array getting
modified
//			mean = double = mean of all TBF values from sample and
obtained
// outputs: reliability = double = reliability value obtained if the normal CDF
calculation is used
// Description: calculates the normal probability of a component if this is the
correct distrbution type
double finalNormal(statisticsVectors * pointerToResCapStatsIntegrated, int
componentPosition, double time, double mean, double sigma)
{
		double errorFunc;
		double maintainability;

		// reliability = 1 - CDF(x)
		// ERF(x) = (x - mu)/(sigma*sqrt(2))
		// CDF(x) = (1/2) * ( 1 + ERF(x))
		errorFunc = ((time - mean) / (sigma * sqrt(2.0)));

		maintainability = 0.5 * ( 1 + errorFunction(errorFunc));

		// return reliability to reliability_calc function
		return maintainability;
}

// name: finalExponential
// inputs: pointerToResCapStatsIntegrated = statisticsVectors = contains all
cumulative data obtained throughout the lifetime of all replications
//			componentPosition = int = element number in array getting
modified
// outputs: exponent = double = reliability value obtained if the exponential CDF
calculation is used
// Description: calculates the exponential probability of a component if this is
the correct distrbution type
double finalExponential(statisticsVectors * pointerToResCapStatsIntegrated, int
componentPosition, double time, double expoMean)
{
		double exponent;
		double maintainability;

		// reliability = 1 - CDF(x)
		// CDF(x) = 1 - e^(-x/mu)
		exponent = (-time/expoMean);

		maintainability = 1.000 - exp(exponent);

		// return reliability to reliability_calc function
```

```
	return maintainability;
}

// name: finalLognormal
// inputs: pointerToResCapStatsIntegrated = statisticsVectors = contains all
cumulative data obtained throughout the lifetime of all replications
//			componentPosition = int = element number in array getting
modified
// outputs: errorFunc = double = reliability value obtained if the Lognormal CDF
calculation is used
// Description: calculates the Lognormal probability of a component if this is the
correct distrbution type
double finalLognormal(statisticsVectors * pointerToResCapStatsIntegrated, int
componentPosition, double logMean, double logSigma, double time)
{
	double errorFunc;
	double maintainability;

	// reliability = 1 - CDF(x)
	// ERF(x) = (ln(x) - mu)/(sigma*sqrt(2)) -> note the mean and STDDEV of
this distribution type will use the natual log of each value in its calculation
	// CDF(x) = (1/2) * ( 1 + ERF(x))
	errorFunc = ((log(time) - logMean) / (logSigma * sqrt(2.0)));

	double err = errorFunction(errorFunc);
	maintainability = 0.5 * ( 1.000 + err);

	// return reliability to reliability_calc function
	return maintainability;
}

// name: finalWeibull
// inputs: pointerToResCapStatsIntegrated = statisticsVectors = contains all
cumulative data obtained throughout the lifetime of all replications
//			componentPosition = int = element number in array getting
modified
// outputs: power = double = reliability value obtained if the weibull CDF
calculation is used
// Description: calculates the weibull probability of a component if this is the
correct distrbution type
double finalWeibull(statisticsVectors * pointerToResCapStatsIntegrated, int
componentPosition, double time, double etas, double betas)
{
	double power;
	double maintainability;

	// reliability = 1 - CDF(x)
	// CDF(x) =  (x/eta
	power = time/etas;
	power = pow(power,betas);
	power = -power;
	maintainability = 1.0000 - (exp(power));

	// return reliability to reliability_calc function
	return maintainability;
}

// name: outputFinalTable
// inputs: pointerToResCapStatsIntegrated = statisticsVectors = all values, new
and from file generated throughout the programs run life (all replications
//                  numCols = int = number of columns conatined in TTF file
//                  lifetime = double = lifetime of the PCB (equal to the lowest
MTBF of all components)
//                  halfWidth = double = halfWidth of the component containing the
lowest MTBF
// description: prints out all the values generated throughout the program into a
table in a .txt format
void outputFinalTable(statisticsVectors * pointerToResCapStatsIntegrated, int
numCols, string fileName, vector <systemMaintainability> &
componentMaintainability
       , systemMaintainability & maintain, double runLength)
{
       double time = 0;
       double timeInc = 0.01;
       int increment = 0;

       fstream outputTable;
       fstream outputIdentifier;

       string txtFile;
       string identifier;
       string buffer = fileName;

       for(int col = 0; col < numCols; col++)
       {
              identifier = pointerToResCapStatsIntegrated[col].partID;

              fileName.append(identifier);
              fileName.append(" Maintainability.txt");

              outputIdentifier.open(fileName.c_str(), ios::out);

              outputIdentifier << "Time" << setw(15) << "Maintainability" <<
std::endl;

              for(unsigned int main = 0; main <
componentMaintainability[col].maintainability.size(); main++)
              {
                     outputIdentifier << time << setw(15) <<
componentMaintainability[col].maintainability[main] << std::endl;
                     time+=timeInc;
              }

              time = 0;

              outputIdentifier.close();
              outputIdentifier.clear();
              fileName.erase();
              fileName.clear();

              fileName = buffer;
       }

       fileName.append("System Maintainability.txt");

       outputIdentifier.open(fileName.c_str(), ios::out);
```

```
        outputIdentifier << "time" << setw(20) << "maintainability" << std::endl;

        time = maintain.timeBegin-0.05;
        timeInc = 0.01;

        for(int comp = 0; comp < 5; comp++)
        {
                if(time >-0.01 &&
                        time < 0)
                {
                        outputIdentifier << "0" << setw(20) << "0" << std::endl;
                }

                else if(time >= 0)
                {
                        outputIdentifier << time << setw(20) << "0" << std::endl;
                }

                time+=timeInc;
        }

        while(increment < maintain.maintainability.size())
        {
                outputIdentifier << time << setw(20) <<
maintain.maintainability[increment] << std::endl;

                time+=timeInc;
                increment++;
        }

        for(int comp = 0; comp < 5; comp++)
        {
                outputIdentifier << time << setw(20) << "1" << std::endl;
                time+=timeInc;
        }

        outputIdentifier.close();
        outputIdentifier.clear();
        fileName.erase();
        fileName.clear();
}
```

APPENDIX G

VIRTUAL METER CODE

```
#include "stdafx.h"

struct componentNetworks{
	string partID;

	double reliability;
	double tempUnderStress;

	int networkIn;
	int networkOut;

	bool connectionMade;
	bool extraNetworks;
	bool thermal;

	vector <int> netIN;
	vector <int> netOUT;

	vector <int> extraNets;
	vector <int> allNets;
	vector <string> componentsWithin;
};

struct compIn{
	int networkIn;
	int networkOut;

	string componentAbove;
	string currentComponent;

	vector <double> withinReliability;
	vector <string> componentsWithin;
	vector <int> allNets;
};

struct beginningNetwork{
	int networkIn;
	int networkOut;

	string partID;

	double componentReliability;

	bool thermal;

	vector <int> netIN;
	vector <int> netOUT;
	vector <double> reliability;
	vector <string> componentsWithin;
	vector <compIn> componentData;

	vector <double> withinReliability;
	vector<int> allNets;
};

struct componentCombinations{
	string parentID;
	string childID;

	int comboNumber;
	int network;

	double reliability;

	vector <string> components;

	bool chipEnter;
	bool chipExit;
	bool initial;
	bool initial2;
	bool final;
};

struct paths{
	vector <string> components;
	double reliability;
	bool lowestRely;

	bool complete;
	bool endeded;
};

class multimeter{
	public:

	void importNetworksFile(vector <componentNetworks> & multiMeter, string 
fileName, float time);
	void deleteConnectNetworks(vector <componentNetworks> & multiMeter, int 
terminal1, int terminal2);
	void makeSeriesConnection(vector <componentNetworks> & multiMeter);
	void findCombinations(vector <componentCombinations> &, vector 
<componentNetworks> &);
	int createPaths(vector <componentNetworks> &, vector <paths> &, vector 
<componentCombinations> &, int terminal1, int terminal2);
	void createPathway1(vector <componentNetworks> &, vector <paths> &, vector 
<componentCombinations> &);
	void createPathway2(vector <componentNetworks> &, vector <paths> &, vector 
<componentCombinations> &);
	void calcReliability(vector <componentNetworks> &, vector <paths> &);
	void outputPath(vector <paths> &, vector <componentNetworks> &, string 
output_path, int pathReturn);
	void createNetworkInOut(vector <componentNetworks> &, vector 
<componentCombinations> &);
	double initialConnection(vector <componentNetworks> &, int terminal1, int 
terminal2);
	void deletePathsGreater(vector <paths> &);
};

// Name: importNetworksFile
// Inputs: networks = componentNetworks = pointer to component networks structure
where networks will be stored
//                   fileName = pointer to string = pointer to the file name
specified by user containing the network ID's
// Description: Goes into a file which contains the network information, and
places all terminal 1 networks for a
//                          component in a vector, and the terminal 2 (network out)
into a seperate vector.
void multimeter::importNetworksFile(vector <componentNetworks> & multiMeter,
string fileName, float time)
{
       int fileChoice = 1;
       int componentPosition = 0;
       int terminal;
       int network;
       double tempUnderStress;
       double loopTime;

       double reliability;
       double temp;

       bool firstNumber = true;

       fstream FILE_IN;

       string Line;
       string part;

       string FILE = fileName;
       string fileBuffer = FILE;

       FILE.append("connections.txt");

       // open file
       FILE_IN.open(FILE.c_str());

       multiMeter.resize(1);

       // if file is available to open
       if(FILE_IN.is_open())
       {
              // run through the program column by column, then row by row and
fill the specified vector
              while(getline(FILE_IN,Line,'\n'))
              {
                     FILE_IN >> part >> terminal >> network;

                     if(firstNumber)
                     {
                            multiMeter[componentPosition].partID = part;

       multiMeter[componentPosition].allNets.push_back(network);

                            if(terminal == 1)
                            {
                                   multiMeter[componentPosition].networkIn =
network;
```

```
                                    multiMeter[componentPosition].extraNetworks = 
false;
                              }

                              else if(terminal == 2)
                              {
                                    multiMeter[componentPosition].networkOut = 
network;
                              }

                              else
                              {
                                    
      multiMeter[componentPosition].extraNets.push_back(network);
                                    multiMeter[componentPosition].extraNetworks = 
true;
                              }

                              firstNumber = false;
                        }

                        else if(part == multiMeter[componentPosition].partID)
                        {
                              
      multiMeter[componentPosition].allNets.push_back(network);

                              if(terminal == 1)
                              {
                                    multiMeter[componentPosition].networkIn = 
network;
                                    
                                    multiMeter[componentPosition].extraNetworks = 
false;
                              }

                              else if(terminal == 2)
                              {
                                    multiMeter[componentPosition].networkOut = 
network;
                              }

                              else
                              {
                                    
      multiMeter[componentPosition].extraNets.push_back(network);
                                    multiMeter[componentPosition].extraNetworks = 
true;
                              }
                        }

                        else if(part != multiMeter[componentPosition].partID)
                        {
                              componentPosition++;

                              multiMeter.resize(multiMeter.size() + 1);

                              multiMeter[componentPosition].partID = part;
                              
      multiMeter[componentPosition].allNets.push_back(network);
						if(terminal == 1)
						{
							multiMeter[componentPosition].networkIn = 
network;
							multiMeter[componentPosition].extraNetworks = 
false;
						}

						else if(terminal == 2)
						{
							multiMeter[componentPosition].networkOut = 
network;
						}

						else
						{
	multiMeter[componentPosition].extraNets.push_back(network);
							multiMeter[componentPosition].extraNetworks = 
true;
						}
				}
			}

			//close file
			FILE_IN.close();
	}

	FILE_IN.clear();

	FILE = fileBuffer;

	FILE.append("ComponentStressTemps.txt");

	FILE_IN.open(FILE.c_str());

	if(FILE_IN.is_open())
	{
		while(getline(FILE_IN, Line,'\n'))
		{
			FILE_IN >> part >> tempUnderStress;

			for(vector<componentNetworks>::iterator comp = 
multiMeter.begin(); comp != multiMeter.end(); comp++)
			{
				if(part == comp->partID)
				{
					comp->thermal = true;
					comp->tempUnderStress = tempUnderStress;
				}
			}
		}
	}

	for(vector<componentNetworks>::iterator comp = multiMeter.begin(); comp != 
multiMeter.end(); comp++)
	{
		if(comp->thermal != true)
		{
			comp->thermal = false;
			comp->reliability = 1;
		}
	}

	FILE_IN.close();
	FILE_IN.clear();

	FILE = fileBuffer;

	for(vector<componentNetworks>::iterator comp = multiMeter.begin(); comp !=
multiMeter.end(); comp++)
	{
		if(!comp->thermal)
		{
			FILE.append("Reliability\\");
			FILE.append(comp->partID);
			FILE.append(" Reliability.txt");

			FILE_IN.open(FILE.c_str());

			if(FILE_IN.is_open())
			{
				while(getline(FILE_IN, Line, '\n'))
				{
					FILE_IN >> loopTime >> reliability;

					if((loopTime < time+0.002 &&
						loopTime > time-0.002))
					{
						comp->reliability = reliability;
					}
				}
			}

			FILE_IN.close();
			FILE_IN.clear();

			FILE = fileBuffer;
		}

		else
		{
			FILE.append("Reliability\\");
			FILE.append(comp->partID);
			FILE.append(" Thermal Reliability.txt");

			FILE_IN.open(FILE.c_str());

			if(FILE_IN.is_open())
			{
				while(getline(FILE_IN, Line, '\n'))
				{
					FILE_IN >> loopTime >> reliability >> temp;

					if(temp == comp->tempUnderStress)
```

```
                                        {
                                                if((loopTime < time+0.002 &&
                                                        loopTime > time-0.002))
                                                {
                                                        comp->reliability = reliability;
                                                }
                                        }
                                }
                        }

                        FILE_IN.close();
                        FILE_IN.clear();

                        FILE = fileBuffer;
                }
        }

        for(unsigned int comp = 0; comp < multiMeter.size(); comp++)
        {
                std::cout << multiMeter[comp].reliability << std::endl;
        }

        getchar();

        FILE_IN.clear();
        FILE_IN.close();

        getchar();

        return;
}

double multimeter::initialConnection(vector <componentNetworks> & multiMeter, int
terminal1, int terminal2)
{
        vector<componentNetworks>::iterator component;
        vector<int>::iterator networkIN;
        vector<int>::iterator networkOUT;
        vector<double>::iterator reliabilities;

        for(vector<componentNetworks>::iterator multi = multiMeter.begin(); multi
!= multiMeter.end(); multi++)
        {
                for(unsigned int comp1 = 0; comp1 < multi->allNets.size(); comp1++)
                {
                        if(comp1 < (multi->allNets.size()/2))
                        {
                                multi->netIN.push_back(multi->allNets[comp1]);
                        }

                        else
                        {
                                multi->netOUT.push_back(multi->allNets[comp1]);
                        }
                }
        }

        double lowRely = 0;
```

```
	vector<double> lowestReliability;

	for(component = multiMeter.begin(); component!= multiMeter.end();
component++)
	{
		for(networkIN = component->netIN.begin(); networkIN != component-
>netIN.end(); networkIN++)
		{
			for(networkOUT = component->netOUT.begin(); networkOUT !=
component->netOUT.end(); networkOUT++)
			{
				if(*networkIN == terminal1 &&
					*networkOUT == terminal2)
				{
					lowestReliability.push_back(component-
>reliability);
				}
			}
		}
	}

	for(reliabilities = lowestReliability.begin(); reliabilities !=
lowestReliability.end(); reliabilities++)
	{
		if(*reliabilities > lowRely)
		{
			lowRely = *reliabilities;
		}
	}

	return lowRely;
}

// name: deleteConnectNetworks
// intput: multiMeter = componentNetworks = contains all networks within the PCB
//			terminal1 = int = input terminal selected by the user
//			terminal2 = int = output terminal selected by the user
// description: runs through all components and if a component is on the opposing
side of where the connectio shoudl start
//				it will delete that component to decrease run time
void multimeter::deleteConnectNetworks(vector <componentNetworks> & multiMeter,
int terminal1, int terminal2)
{
	for(vector<componentNetworks>::iterator multi = multiMeter.begin(); multi
!= multiMeter.end(); multi++)
	{
		if(multi->networkIn == terminal2)
		{
			multiMeter.erase(multi);

			multi = multiMeter.begin();
		}

		else if(multi->networkOut == terminal1)
		{
			multiMeter.erase(multi);
```

```
                        multi = multiMeter.begin();
                }
        }

        return;
}

// name: makeSeriesConnection
// input: multiMeter = componentNetworks = contains all information for all
components;
// description: makes all series connections at the beginning of the program (will
not make a connection if a component
//                          contains the input or output terminals) to reduce the
size of the "web".
void multimeter::makeSeriesConnection(vector <componentNetworks> & multiMeter)
{
        int otherConnections = 0;

        for(vector<componentNetworks>::iterator multi = multiMeter.begin(); multi
!= multiMeter.end(); multi++)
        {
                multi->connectionMade = false;
        }

        for(vector<componentNetworks>::iterator multi = multiMeter.begin(); multi
!= multiMeter.end(); multi++)
        {
                for(vector<componentNetworks>::iterator multi1 = multiMeter.begin();
multi1 != multiMeter.end(); multi1++)
                {
                        if(multi != multi1 &&
                                multi->networkOut == multi1->networkIn &&
                                multi->networkIn != multi1->networkOut &&
                                !multi->connectionMade &&
                                !multi1->connectionMade &&
                                !multi->extraNetworks &&
                                !multi1->extraNetworks)
                        {
                                for(vector<componentNetworks>::iterator multi2 =
multiMeter.begin(); multi2 != multiMeter.end(); multi2++)
                                {
                                        if(multi != multi2 &&
                                                multi1 != multi2 &&
                                                !multi->connectionMade)
                                        {
                                                for(unsigned int net = 0; net < multi2-
>allNets.size(); net++)
                                                {
                                                        if(multi2->allNets[net] == multi-
>networkOut)
                                                        {
                                                                otherConnections++;
                                                        }
                                                }
                                        }
                                }

                                if(otherConnections == 0)
                        {
                              multi->reliability = multi->reliability * 
multi1->reliability;

                              multi1->connectionMade = true;

                              multi->networkIn = multi1->networkIn;
                              multi->allNets[0] = multi1->networkIn;
                        }

                        else
                        {
                              otherConnections = 0;
                        }
                  }
            }
      }

      for(vector<componentNetworks>::iterator multi = multiMeter.begin(); multi 
!= multiMeter.end(); multi++)
      {
            if(multi->connectionMade)
            {
                  multiMeter.erase(multi);

                  multi = multiMeter.begin();
            }
      }

      return;
}

//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
void multimeter::createNetworkInOut(vector <componentNetworks> & multiMeter, 
vector <componentCombinations> & componentCombo)
{
      componentCombinations singleCombo;
      int counter = 0;

      vector<componentNetworks>::iterator comp1;
      vector<componentNetworks>::iterator comp2;

      vector<componentCombinations>::iterator combo;

      vector<int>::iterator netInIter;
      vector<int>::iterator netOutIter;

      counter = 0;
```

```
	for(comp1 = multiMeter.begin(); comp1 != multiMeter.end(); comp1++)
	{
		for(comp2 = multiMeter.begin(); comp2 != multiMeter.end(); comp2++)
		{
			if(comp1->partID != comp2->partID)
			{
				for(unsigned int net = 0; net < comp1->netOUT.size();
net++)
				{
					for(unsigned int net1 = 0; net1 < comp2-
>netIN.size(); net1++)
					{
						if(componentCombo.size() > 0)
						{
							for(combo =
componentCombo.begin(); combo != componentCombo.end(); combo++)
							{
								if(combo->parentID ==
comp1->partID &&
									combo->childID ==
comp2->partID)
								{
									counter++;
								}
							}
						}

						if(comp1->netOUT[net] == comp2-
>netIN[net1] &&
							counter == 0)
						{
							singleCombo.parentID = comp1-
>partID;
							singleCombo.childID = comp2-
>partID;

	singleCombo.components.push_back(comp1->partID);

	singleCombo.components.push_back(comp2->partID);
							singleCombo.network = comp2-
>netIN[net1];

	componentCombo.push_back(singleCombo);
						}

						counter = 0;
					}
				}
			}
		}
	}

	return;
}

///////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////
int multimeter::createPaths(vector <componentNetworks> & multiMeter, vector
<paths> & pathways, vector <componentCombinations> & componentCombo, int
terminal1, int terminal2)
{
	multimeter MULTIMETER;
	paths currentPath;
	int counter = 0;
	int countFinal = 0;
	int countBegin = 0;
	int numberComponents = 2;
	int countCompleted = 0;
	int count = 0;

	for(vector<componentCombinations>::iterator combo = componentCombo.begin();
combo != componentCombo.end(); combo++)
	{
		combo->initial2 = false;
		combo->initial = false;
		combo->final = false;
	}

	//find the initial components and final components combinations
	for(vector<componentNetworks>::iterator multi = multiMeter.begin(); multi
!= multiMeter.end(); multi++)
	{
		for(vector<componentCombinations>::iterator combo =
componentCombo.begin(); combo != componentCombo.end(); combo++)
		{
			for(unsigned int comp2 = 0; comp2 < multi->netIN.size();
comp2++)
			{
				if(multi->netIN[comp2] == terminal1 &&
					multi->partID == combo->parentID)
				{
					combo->initial = true;

					pathways.resize(pathways.size()+1);

					pathways[counter].components.push_back(combo-
>parentID);
					pathways[counter].components.push_back(combo-
>childID);

					pathways[counter].complete = false;
					pathways[counter].lowestRely = false;

					counter++;
				}
			}
```

```
                        for(unsigned int comp2 = 0; comp2 < multi->netOUT.size(); 
comp2++)
                        {
                                if(multi->netOUT[comp2] == terminal2 &&
                                        multi->partID == combo->childID)
                                {
                                        combo->final = true;
                                }
                        }
                }
        }

        for(vector<componentCombinations>::iterator combo = componentCombo.begin(); 
combo != componentCombo.end(); combo++)
        {
                if(combo->final)
                {
                        countFinal++;
                }

                if(combo->initial)
                {
                        countBegin++;
                }
        }

        if(countBegin == 0 ||
                countFinal == 0)
        {
                return 0;
        }

        while(counter != 0)
        {
                counter = 0;
                numberComponents++;

                MULTIMETER.createPathway1(multiMeter, pathways, componentCombo);

                for(vector<componentCombinations>::iterator combo = 
componentCombo.begin(); combo != componentCombo.end(); combo++)
                {
                        if(combo->initial2)
                        {
                                counter++;
                        }
                }

                for(vector<paths>::iterator currentPath = pathways.begin(); 
currentPath != pathways.end(); currentPath++)
                {
                        if(currentPath->components.size() < numberComponents &&
                                !currentPath->complete)
                        {
                                pathways.erase(currentPath);

                                currentPath = pathways.begin();
                        }
```

```
		}

		for(vector<paths>::iterator currentPath = pathways.begin();
currentPath != pathways.end(); currentPath++)
		{
			if(currentPath->complete)
			{
				countCompleted++;
			}
		}

		if(countCompleted > 0)
		{
			MULTIMETER.calcReliability(multiMeter, pathways);
			MULTIMETER.deletePathsGreater(pathways);
		}

		/*for(vector<paths>::iterator currentPath = pathways.begin;
currentPath != pathways.end(); currentPath++)
		{
			for(vector<string>::iterator pathComp = currentPath-
>components.begin(); pathComp != currentPath->components.end(); pathComp++)
			{
				for(vector<componentNetworks>::iterator comp =
multiMeter.begin(); comp != multiMeter.end(); comp++)
				{
					if(comp->partID == *pathComp &&
						comp->extraNetworks)
					{
						count++;
					}
				}

				if(count == 2)
				{*/



		countCompleted = 0;

		for(unsigned int comp = 0; comp < pathways.size(); comp++)
		{
			for(unsigned int comp1 = 0; comp1 <
pathways[comp].components.size(); comp1++)
			{
				std::cout << pathways[comp].components[comp1] << "\t";
			}

			std::cout << std::endl;
		}

		std::cout << "--------------------------------------------------" <<
std::endl;

		for(unsigned int comp = 0; comp < pathways.size(); comp++)
		{
			if(pathways[comp].complete)
			{
```

```
                            for(unsigned int comp1 = 0; comp1 <
pathways[comp].components.size(); comp1++)
                            {
                                   std::cout << pathways[comp].components[comp1] <<
"\t";
                            }

                            std::cout << std::endl;
                     }
              }

              std::cout << counter << std::endl;

              if(counter == 0)
              {
                     break;
              }

              numberComponents++;

              counter = 0;

              MULTIMETER.createPathway2(multiMeter, pathways, componentCombo);

              for(vector<componentCombinations>::iterator combo =
componentCombo.begin(); combo != componentCombo.end(); combo++)
              {
                     if(combo->initial)
                     {
                            counter++;
                     }
              }

              for(vector<paths>::iterator currentPath = pathways.begin();
currentPath != pathways.end(); currentPath++)
              {
                     if(currentPath->components.size() < numberComponents &&
                            !currentPath->complete)
                     {
                            pathways.erase(currentPath);

                            currentPath = pathways.begin();
                     }
              }

              for(vector<paths>::iterator currentPath = pathways.begin();
currentPath != pathways.end(); currentPath++)
              {
                     if(currentPath->complete)
                     {
                            countCompleted++;
                     }
              }

              if(countCompleted > 0)
              {
                     MULTIMETER.calcReliability(multiMeter, pathways);
                     MULTIMETER.deletePathsGreater(pathways);
```

```
		}

		countCompleted = 0;

		for(unsigned int comp = 0; comp < pathways.size(); comp++)
		{
			for(unsigned int comp1 = 0; comp1 <
pathways[comp].components.size(); comp1++)
			{
				std::cout << pathways[comp].components[comp1] << "\t";
			}

			std::cout << std::endl;
		}

		std::cout << "-------------------------------------------------" <<
std::endl;

		for(unsigned int comp = 0; comp < pathways.size(); comp++)
		{
			if(pathways[comp].complete)
			{
				for(unsigned int comp1 = 0; comp1 <
pathways[comp].components.size(); comp1++)
				{
					std::cout << pathways[comp].components[comp1] <<
"\t";
				}

				std::cout << std::endl;
			}
		}

		std::cout << counter << std::endl;

		if(counter == 0)
		{
			break;
		}
	}

	return 1;
}

//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
void multimeter::createPathway1(vector <componentNetworks> & multiMeter, vector
<paths> & pathways, vector <componentCombinations> & componentCombo)
{
	int counter = 0;
	int pathwayCounter = 0;

	vector<componentCombinations>::iterator combo;
	vector<componentCombinations>::iterator combo1;

	vector<paths>::iterator pathIter;
	vector<paths>::iterator pathREITER;

	paths saveInitialPath;

	vector<paths> pathwayInitial(pathways);

	for(pathIter = pathwayInitial.begin(); pathIter != pathwayInitial.end();
pathIter++)
	{
		if(!pathIter->complete)
		{
			saveInitialPath.components.resize(pathIter-
>components.size());

			std::copy(pathIter->components.begin(), pathIter-
>components.end(), saveInitialPath.components.begin());

			for(combo = componentCombo.begin(); combo !=
componentCombo.end(); combo++)
			{
				if(combo->initial &&
					!combo->final)
				{
					for(combo1 = componentCombo.begin(); combo1 !=
componentCombo.end(); combo1++)
					{
						for(vector<string>::iterator comp =
saveInitialPath.components.begin(); comp != saveInitialPath.components.end();
comp++)
						{
							if(*comp == combo1->childID &&
								combo->childID == combo1-
>parentID &&

	saveInitialPath.components[saveInitialPath.components.size()-1] == combo1-
>parentID)
							{
								pathwayCounter++;
							}
						}

						if(pathwayCounter == 0)
						{
							if(counter == 0)
							{
								if(combo->childID ==
combo1->parentID &&
									pathIter-
>components[pathIter->components.size()-1] == combo1->parentID)
								{
									counter++;
```

```
                                                                                combo1->initial2 =
true;

                                                                                pathIter-
>components.push_back(combo1->childID);

                                                                                if(combo1->final)
                                                                                {
                                                                                        pathIter-
>complete = true;
                                                                                        combo1-
>initial2 = false;

                                                                                        break;
                                                                                }
                                                                        }
                                                                }

                                                                else
                                                                {
                                                                        if(combo->childID ==
combo1->parentID &&
        saveInitialPath.components[saveInitialPath.components.size()-1] == combo1-
>parentID)
                                                                        {
                                                                                counter++;

                                                                                combo1->initial2 =
true;


        pathways.resize(pathways.size()+1);


        pathways[pathways.size()-
1].components.resize(saveInitialPath.components.size());


        std::copy(saveInitialPath.components.begin(),
saveInitialPath.components.end(), pathways[pathways.size()-1].components.begin());


        pathways[pathways.size()-1].components.push_back(combo1->childID);

        pathways[pathways.size()-1].lowestRely = false;

        pathways[pathways.size()-1].complete = false;

                                                                                if(combo1->final)
                                                                                {

        pathways[pathways.size()-1].complete = true;
                                                                                        combo1-
>initial2 = false;

                                                                                        break;
                                                                                }
```

```
                                                        }
                                                }
                                        }

                                        pathwayCounter = 0;
                                }
                        }

                        counter = 0;
                }

                saveInitialPath.components.clear();
        }
}

        for(vector<componentCombinations>::iterator combo = componentCombo.begin();
combo != componentCombo.end(); combo++)
        {
                combo->initial = false;
        }

        return;
}

//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
void multimeter::createPathway2(vector <componentNetworks> & multiMeter, vector
<paths> & pathways, vector <componentCombinations> & componentCombo)
{
        int counter = 0;
        int pathwayCounter = 0;

        paths saveInitialPath;

        vector<componentCombinations>::iterator combo;
        vector<componentCombinations>::iterator combo1;

        vector<paths>::iterator pathIter;

        vector<paths> pathwayInitial(pathways);

        for(pathIter = pathwayInitial.begin(); pathIter != pathwayInitial.end();
pathIter++)
        {
                if(!pathIter->complete)
                {
                        saveInitialPath.components.resize(pathIter-
>components.size());
```

```
					std::copy(pathIter->components.begin(), pathIter-
>components.end(), saveInitialPath.components.begin());

					for(combo = componentCombo.begin(); combo !=
componentCombo.end(); combo++)
					{
						if(combo->initial2 &&
							!combo->final)
						{
							for(combo1 = componentCombo.begin(); combo1 !=
componentCombo.end(); combo1++)
							{
								for(vector<string>::iterator comp =
saveInitialPath.components.begin(); comp != saveInitialPath.components.end();
comp++)
								{
									if(*comp == combo1->childID &&
										combo->childID == combo1-
>parentID &&

	saveInitialPath.components[saveInitialPath.components.size()-1] == combo1-
>parentID)
									{
										pathwayCounter++;
									}
								}

								if(pathwayCounter == 0)
								{
									if(counter == 0)
									{
										if(combo->childID ==
combo1->parentID &&
											pathIter-
>components[pathIter->components.size()-1] == combo1->parentID)
										{
											counter++;

											combo1->initial =
true;

											pathIter-
>components.push_back(combo1->childID);

											if(combo1->final)
											{
												pathIter-
>complete = true;
												combo1-
>initial = false;


												break;
											}
										}
									}

									else
									{
```

```
                                                        if(combo->childID ==
combo1->parentID &&

       saveInitialPath.components[saveInitialPath.components.size()-1] == combo1-
>parentID)
                                                        {
                                                               counter++;

                                                               combo1->initial =
true;


       pathways.resize(pathways.size()+1);


       pathways[pathways.size()-
1].components.resize(saveInitialPath.components.size());


       std::copy(saveInitialPath.components.begin(),
saveInitialPath.components.end(), pathways[pathways.size()-1].components.begin());


       pathways[pathways.size()-1].components.push_back(combo1->childID);

       pathways[pathways.size()-1].lowestRely = false;

       pathways[pathways.size()-1].complete = false;

                                                               if(combo1->final)
                                                               {

       pathways[pathways.size()-1].complete = true;
                                                                      combo1-
>initial = false;


                                                                      break;
                                                               }
                                                        }
                                                 }
                                          }

                                          pathwayCounter = 0;
                                   }
                            }
                     }

                     saveInitialPath.components.clear();

                     counter = 0;
              }
       }

       for(vector<componentCombinations>::iterator combo = componentCombo.begin();
combo != componentCombo.end(); combo++)
       {
              combo->initial2 = false;
       }
```

```
	return;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
void multimeter::deletePathsGreater(vector <paths> & pathways)
{
	double highestReliability;

	for(vector<paths>::iterator currentPath = pathways.begin(); currentPath !=
pathways.end(); currentPath++)
	{
		if(currentPath->lowestRely)
		{
			highestReliability = currentPath->reliability;
		}
	}

	for(vector<paths>::iterator currentPath = pathways.begin(); currentPath !=
pathways.end(); currentPath++)
	{
		if(currentPath->reliability < highestReliability &&
			!currentPath->lowestRely)
		{
			pathways.erase(currentPath);

			currentPath = pathways.begin();
		}
	}

	return;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
void multimeter::calcReliability(vector <componentNetworks> & multiMeter, vector
<paths> & pathways)
{
	double highest_reliability = 0;
	for(vector<paths>::iterator currentPath = pathways.begin(); currentPath != 
pathways.end(); currentPath++)
	{
		currentPath->reliability = 1;
		currentPath->lowestRely = false;
	}

	for(vector<paths>::iterator pathIter = pathways.begin(); pathIter < 
pathways.end(); pathIter++)
	{
		for(unsigned int comp2 = 0; comp2 < pathIter->components.size(); 
comp2++)
		{
			for(vector<componentNetworks>::iterator multi = 
multiMeter.begin(); multi != multiMeter.end(); multi++)
			{
				if(pathIter->components[comp2] == multi->partID)
				{
					pathIter->reliability = pathIter->reliability * 
multi->reliability;
				}
			}
		}
	}

	for(vector<paths>::iterator pathIter = pathways.begin(); pathIter < 
pathways.end(); pathIter++)
	{
		if(pathIter->reliability > highest_reliability &&
			pathIter->complete)
		{
			highest_reliability = pathIter->reliability;
		}
	}

	for(vector<paths>::iterator pathIter = pathways.begin(); pathIter < 
pathways.end(); pathIter++)
	{
		if(pathIter->reliability == highest_reliability &&
			pathIter->complete)
		{
			pathIter->lowestRely = true;
		}
	}

	return;
}

//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////
```

```
void multimeter::outputPath(vector <paths> & pathways, vector <componentNetworks>
& multiMeter, string output_path, int pathReturn)
{
	int completeCounter = 0;
	fstream FILE_OUT;

	output_path.append("data\\Reliability\\");

	string output_buffer = output_path;

	output_path.append("Lowest_Reliability_path.txt");

	FILE_OUT.open(output_path.c_str(), ios::out);

	for(vector<paths>::iterator iter = pathways.begin(); iter !=
pathways.end(); iter++)
	{
		if(iter->lowestRely)
		{
			completeCounter++;
		}
	}

	if(FILE_OUT.is_open())
	{
		if(completeCounter != 0 &&
			pathReturn == 1)
		{
			for(unsigned int comp = 0; comp < pathways.size(); comp++)
			{
				if(pathways[comp].lowestRely)
				{
					for(unsigned int comp1 = 0; comp1 <
pathways[comp].components.size(); comp1++)
					{
						FILE_OUT <<
pathways[comp].components[comp1] << setw(15);

						for(unsigned int comp2 = 0; comp2 <
multiMeter.size(); comp2++)
						{
							if(multiMeter[comp2].partID ==
pathways[comp].components[comp1])
							{
								FILE_OUT <<
multiMeter[comp2].reliability << std::endl;
							}
						}
					}
				}
			}
		}

		else
		{
			FILE_OUT << "Components Are Not Connected" << std::endl;
		}
	}
```

```
        FILE_OUT.clear();
        FILE_OUT.close();

        output_path = output_buffer;

        output_path.append("Pathway_Reliability.txt");

        FILE_OUT.open(output_path.c_str(), ios::out);

        if(FILE_OUT.is_open())
        {
                if(completeCounter != 0 &&
                        pathReturn == 1)
                {
                        for(unsigned int comp = 0; comp < pathways.size(); comp++)
                        {
                                if(pathways[comp].lowestRely)
                                {
                                        FILE_OUT << pathways[comp].reliability;
                                }
                        }
                }

                else
                {
                        FILE_OUT << "Components Are Not Connected" << std::endl;
                }
        }

        FILE_OUT.clear();
        FILE_OUT.close();

        return;
}

////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
int main(int argc, char* argv[])
{
        multimeter MULTIMETER;
        int seriesConnections = 0;
        int parallelConnections = 0;
        int initialConnection = 0;

        bool endConnections = false;
        bool endConnection = false;

        int pathReturn;
        int terminal1 = atoi(argv[2]);
```

```
	int terminal2 = atoi(argv[3]);
	float time = atof(argv[4]);

	double lowRely = 0;

	string fileName = argv[1];
	string inputComponent;
	string outputComponent;
	string outputDirectory = argv[5];

	vector <componentNetworks> multiMeter;
	vector <paths> pathways;
	vector <componentCombinations> componentCombo;

	//modify vector to contain all information that is contained in the file
	MULTIMETER.importNetworksFile(multiMeter, fileName, time);

	lowRely = MULTIMETER.initialConnection(multiMeter, terminal1, terminal2);

	if(lowRely == 0)
	{
		MULTIMETER.deleteConnectNetworks(multiMeter, terminal1, terminal2);

		MULTIMETER.makeSeriesConnection(multiMeter);

		//MULTIMETER.findCombinations(componentCombo, multiMeter);

		MULTIMETER.createNetworkInOut(multiMeter, componentCombo);

		pathReturn = MULTIMETER.createPaths(multiMeter, pathways, 
componentCombo, terminal1, terminal2);

		MULTIMETER.calcReliability(multiMeter, pathways);

		MULTIMETER.outputPath(pathways, multiMeter, outputDirectory, 
pathReturn);
	}

	else
	{
		paths path;

		pathReturn = 1;

		path.reliability = lowRely;
		vector<componentNetworks>::iterator component;

		for(component = multiMeter.begin(); component != multiMeter.end(); 
component++)
		{
			if(component->reliability == lowRely)
			{
				path.components.push_back(component->partID);
			}
		}

		path.lowestRely = true;
```

```
              pathways.push_back(path);

              MULTIMETER.outputPath(pathways, multiMeter, outputDirectory, 
pathReturn);
       }

       pathways.clear();
       multiMeter.clear();
       componentCombo.clear();

       return 0;
}
```

APPENDIX H

THERMAL CODE

```
#include "stdafx.h"

struct Component{
	string partID;
	string distributionWord;

	int distributionType;

	double MTBFold;
	double failure_rate_old;
	double failure_rate_new;
	double Reliability_old;
	double mean;
	double sigma;
	double beta;
	double eta;
	double logMean;
	double logSigma;
	double temperatureUnderUse;
	double tempUnderStress;
	double standardDeviation;
	double minTTF;
	double maxTTF;
	double MTBFunderStress;
	double epsillon;

	bool underStressCalculate;

	vector <double> MTBFnew;
	vector <double> reliability_new;
};

struct networkReliability{
	string ID;

	int networkIn;
	int networkOut;

	double tempUnderUse;
	double temperatureUnderUse;

	bool extraNetworks;
	bool connectionMade;

	vector <int> allNets;
	vector <double> reliability;
	vector <double> time;
};

class REL_THERMAL {
public:
	double errorFunction(double x);
	void import_file(string fileName, vector <Component> & component, vector <networkReliability> & relyCalc);

	double B();

	double epsillonOriginal(double temp0, double temp1);

	void mtbfNew(double time0, double Eo, vector <Component> & component, int currentPlace);

    // add a term to the polynomial
    void failureRate(double En, vector <Component> & component, int comp);

	void reliability(vector <Component> & component, double time1, double time0, double epsillonOriginal, int element);

	void exportFile(vector <Component> & component, string fileOut, vector <networkReliability> & relyCalc, bool erase);
};

void REL_THERMAL::import_file(string fileName, vector <Component> & component, vector <networkReliability> & relyCalc)
{
	fstream FILE_IN;

	string line;
	string id;
	Component componentNumber;

	networkReliability networks;

	string DT;
	double MF;
	double FR;
	double sigma;
	double ttfmin;
	double ttfmax;
	double val1;
	double val2;
	string fileName_buffer = fileName;
	double tempUnderUse;
	int terminal;
	int network;
	int componentPosition = 0;
	bool firstNumber = true;
	double time;
	double reliable;
	double tempUnderStress;
	int numberComponents = 0;
	int position = 0;

	fileName.append("ComponentStressTemps.txt");

	FILE_IN.open(fileName.c_str());

	while(getline(FILE_IN,line,'\n'))
	{
		FILE_IN >> id >> tempUnderStress;

		numberComponents++;
	}

	component.resize(numberComponents);

	FILE_IN.close();
	FILE_IN.clear();

	fileName = fileName_buffer;

	fileName.append("ComponentStressTemps.txt");

	FILE_IN.open(fileName.c_str());

	while(getline(FILE_IN,line,'\n'))
	{
		FILE_IN >> id >> tempUnderStress;

		component[position].partID = id;

		for(unsigned int comp = 0; comp < component.size(); comp++)
		{
			if(component[comp].partID == id)
			{
				component[comp].tempUnderStress = tempUnderStress;
			}
		}

		position++;
	}

	FILE_IN.close();
	FILE_IN.clear();

	fileName = fileName_buffer;

	for(unsigned int comp = 0; comp < component.size(); comp++)
	{
		fileName.append("Reliability\\");

		fileName.append(component[comp].partID);
		fileName.append(".txt");

		FILE_IN.open(fileName.c_str());

		while(getline(FILE_IN,line,'\n'))
		{
			FILE_IN >> MF >> sigma >> ttfmin >> ttfmax >> FR >> DT >> val1
>> val2;

			component[comp].MTBFold = MF;
			component[comp].failure_rate_old = FR;
			component[comp].distributionWord = DT;
			component[comp].standardDeviation = sigma;
			component[comp].minTTF = ttfmin;
			component[comp].maxTTF = ttfmax;
```

```
        if(component[comp].distributionWord == "NORMAL")
        {
            component[comp].distributionType = 0;
            component[comp].mean = val1;
            component[comp].sigma = val2;
        }

        else if(component[comp].distributionWord == "EXPONENTIAL")
        {
            component[comp].distributionType = 1;
            component[comp].mean = val1;
        }

        else if(component[comp].distributionWord == "LOGNORMAL")
        {
            component[comp].distributionType = 2;
            component[comp].logMean = val1;
            component[comp].logSigma = val2;
        }

        else if(component[comp].distributionWord == "WEIBULL")
        {
            component[comp].distributionType = 3;
            component[comp].beta = val1;
            component[comp].eta = val2;
        }
    }

    FILE_IN.clear();
    FILE_IN.close();

    fileName = fileName_buffer;
}

fileName.append("ComponentUseTemps.txt");

FILE_IN.open(fileName.c_str());

while(getline(FILE_IN,line,'\n'))
{
    FILE_IN >> id >> tempUnderUse;

    for(unsigned int comp = 0; comp < component.size(); comp++)
    {
        if(component[comp].partID == id)
        {
            component[comp].temperatureUnderUse = tempUnderUse;
        }
    }
}

FILE_IN.close();
FILE_IN.clear();

fileName = fileName_buffer;

return;
```

```
}

double REL_THERMAL::B()
{
       double actEnergy = 0.7;
       double boltzman = 0.00008617;

       return (actEnergy/boltzman);
}

double REL_THERMAL::epsillonOriginal(double temp0, double temp1)
{
       REL_THERMAL Rel_thermal;

       double B = Rel_thermal.B();
       double Eo;
       double firstB = B/temp0;
       double secondB = B/temp1;
       double subtraction = firstB - secondB;

       Eo = exp(subtraction);

       return Eo;
}

void REL_THERMAL::mtbfNew(double time0, double Eo, vector <Component> & component,
int currentPlace)
{
       REL_THERMAL Rel_thermal;

       component[currentPlace].MTBFnew.push_back(component[currentPlace].MTBFold /
Eo);

       if(component[currentPlace].underStressCalculate)
       {
              Eo =
Rel_thermal.epsillonOriginal((component[currentPlace].temperatureUnderUse+273.15),
(component[currentPlace].tempUnderStress+273.15));

              component[currentPlace].epsillon = Eo;

              component[currentPlace].MTBFunderStress =
component[currentPlace].MTBFold / Eo;

              component[currentPlace].minTTF = component[currentPlace].minTTF /
Eo;

              component[currentPlace].maxTTF = component[currentPlace].maxTTF /
Eo;

              component[currentPlace].underStressCalculate = false;
       }

       return;
}

void REL_THERMAL::failureRate(double Eo, vector <Component> & component, int comp)
{
	component[comp].failure_rate_new = component[comp].failure_rate_old * Eo;

	return;
}

void REL_THERMAL::reliability(vector <Component> & component, double time1, double
epsillonOriginal, double time0, int element)
{
	REL_THERMAL Rel_thermal;

		if(component[element].distributionType == 0)
		{
			double errorFunc;
			double reliability;
			double normal;

			// reliability = 1 - CDF(x)
			// ERF(x) = (x - mu)/(sigma*sqrt(2))
			// CDF(x) = (1/2) * ( 1 + ERF(x))
			errorFunc = ((time1 - component[element].MTBFold) /
(component[element].sigma * sqrt(2.0)));

			normal = 0.5 * ( 1 + Rel_thermal.errorFunction(errorFunc));

			reliability = 1 - normal;

			component[element].reliability_new.push_back(reliability);
		}

		else if(component[element].distributionType == 1)
		{
			double exponent;
			double reliability;
			double exponential;

			// reliability = 1 - CDF(x)
			// CDF(x) = 1 - e^(-x/mu)
			exponent = (-time1/component[element].MTBFold);

			exponential = 1 - exp(exponent);

			reliability = 1 - exponential;

			// return reliability to reliability_calc function
			component[element].reliability_new.push_back(reliability);
		}

		else if(component[element].distributionType == 2)
		{
			double errorFunc;
			double reliability;
			double lognormal;

			// reliability = 1 - CDF(x)
			// ERF(x) = (ln(x) - mu)/(sigma*sqrt(2)) -> note the mean and
STDDEV of this distribution type will use the natual log of each value in its
calculation
			// CDF(x) = (1/2) * ( 1 + ERF(x))
			errorFunc = ((log(time1) - component[element].logMean) / 
(component[element].logSigma * sqrt(2.0)));

			lognormal = 0.5 * ( 1 + Rel_thermal.errorFunction(errorFunc));

			reliability = 1 - lognormal;

			// return reliability to reliability_calc function
			component[element].reliability_new.push_back(reliability);
		}

		else if(component[element].distributionType == 3)
		{
			double power;
			double reliability;
			double weibull;

			// reliability = 1 - CDF(x)
			// CDF(x) =  (x/eta
			power = time1/component[element].eta;
			power = pow(power,component[element].beta);
			power = -power;
			weibull = 1.0000 - (exp(power));

			reliability = 1 - weibull;

			// return reliability to reliability_calc function
			component[element].reliability_new.push_back(reliability);
		}

	return;
}

double REL_THERMAL::errorFunction(double x)
{
	/* erf(z) = 2/sqrt(pi) * Integral(0..x) exp( -t^2) dt
	erf(0.01) = 0.0112834772 erf(3.7) = 0.9999998325
	Abramowitz/Stegun: p299, |erf(z)-erf| <= 1.5*10^(-7)
	*/
	double a1 = 0.254829592;
	double a2 = -0.284496736;
	double a3 = 1.421413741;
	double a4 = -1.453152027;
	double a5 = 1.061405429;
	double p = 0.3275911;

	int sign = 1;

	if(x < 0)
	{
		sign = -1;
	}

	x = fabs(x);

	double t = 1.0/(1.0 + p * x);
	double y = 1.0 - (((((a5 * t +a4) * t) + a3)*t + a2)*t + a1)*t*exp(-x*x);
```

```
       y = sign * y;

       return y;
}

void REL_THERMAL::exportFile(vector <Component> & component, string fileOut,
vector <networkReliability> & relyCalc, bool erase)
{
       string line;
       fstream FILE_OUT;
       float time = 0;
       double temperatureKelvin;

       fileOut.append("data\\Reliability\\");

       string fileOut_Buffer = fileOut;

       if(!erase)
       {
              for(unsigned int comp = 0; comp < component.size(); comp++)
              {
                     fileOut.append(component[comp].partID);
                     fileOut.append(" Thermal.txt");

                     FILE_OUT.open(fileOut.c_str(), ios::out);

                     FILE_OUT << "partID" << setw(20) << "MTBF" << setw(20) << "min
TTF" << setw(20) << "max TTF" << setw(20) << "failure Rate" << std::endl;

                     FILE_OUT << component[comp].partID << setw(20) <<
component[comp].MTBFunderStress << setw(20) << component[comp].minTTF << setw(20)
<< component[comp].maxTTF << setw(20) << component[comp].failure_rate_new <<
std::endl;

                     FILE_OUT.close();
                     FILE_OUT.clear();

                     fileOut = fileOut_Buffer;

                     fileOut.append(component[comp].partID);
                     fileOut.append(" Thermal Lifetime.txt");

                     FILE_OUT.open(fileOut.c_str(), ios::out);

                     FILE_OUT << "Lifetime" << setw(20) << "Temperature (C)" <<
std::endl;

                     temperatureKelvin = component[comp].temperatureUnderUse;

                     for(unsigned int life = 0; life <
component[comp].MTBFnew.size(); life++)
                     {
                            FILE_OUT << component[comp].MTBFnew[life] << setw(20)
<< temperatureKelvin << std::endl;

                            temperatureKelvin = temperatureKelvin+5;
                     }
```

```
			FILE_OUT.close();
			FILE_OUT.clear();

			fileOut = fileOut_Buffer;

			fileOut.append(component[comp].partID);
			fileOut.append(" Thermal Reliability.txt");

			FILE_OUT.open(fileOut.c_str(), ios::out);

			FILE_OUT << "Time" << setw(20) << "Reliability" << setw(20) <<
"Temperature (C)" << std::endl;

			temperatureKelvin = component[comp].temperatureUnderUse;

			for(unsigned int reliability = 0; reliability <
component[comp].reliability_new.size(); reliability++)
			{
				FILE_OUT << time << setw(20) <<
component[comp].reliability_new[reliability] << setw(20) << temperatureKelvin <<
std::endl;

				time+=0.1;

				if(time > 50)
				{
					time = 0;
					temperatureKelvin = temperatureKelvin+5;
				}
			}

			FILE_OUT.close();
			FILE_OUT.clear();

			fileOut = fileOut_Buffer;
		}

		fileOut = fileOut_Buffer;
	}

	fileOut.append("Thermal MTTF.txt");

	FILE_OUT.open(fileOut.c_str(), ios::out);

	FILE_OUT << "Component" << setw(20) << "MTTF" << setw(20) << "Epsillon" <<
std::endl;

	for(unsigned int comp = 0; comp < component.size(); comp++)
	{
		FILE_OUT << component[comp].partID << setw(20) <<
component[comp].MTBFunderStress << setw(20) << component[comp].epsillon <<
std::endl;
	}

	FILE_OUT.close();
	FILE_OUT.clear();

	fileOut = fileOut_Buffer;
	return;
}

int main(int argc, char* argv[])
{
	REL_THERMAL Rel_thermal;

	double epsillonNew;
	double time0 = 0;
	double tempUnderUse;
	double tempUnderStress;
	double tempOriginal = 25+273.15;
	double time1=0;
	double epsillonOriginal;

	vector <Component> component;
	vector <networkReliability> relyCalc;

	string inputDirectory = argv[1];
	string fileOut = argv[2];

	Rel_thermal.import_file(inputDirectory, component, relyCalc);

	for(unsigned int comp = 0; comp < component.size(); comp++)
	{
		component[comp].underStressCalculate = true;

		tempUnderStress = component[comp].tempUnderStress + 273.15;

		tempUnderUse = component[comp].temperatureUnderUse + 273.15;

		epsillonOriginal = Rel_thermal.epsillonOriginal(tempUnderUse, 
tempUnderStress);

		Rel_thermal.failureRate(epsillonOriginal, component, comp);

		for(double temp = tempUnderUse; temp < 573.15; temp+=5)
		{
			epsillonOriginal = Rel_thermal.epsillonOriginal(tempUnderUse, 
temp);

			Rel_thermal.mtbfNew(time0, epsillonOriginal, component, comp);

			for(time0 = 0; time0 < 50.1; time0+=0.1)
			{
				if(temp != tempOriginal)
				{
					time1 = time0 * epsillonOriginal;
				}

				Rel_thermal.reliability(component, time1, time0, 
epsillonOriginal, comp);
			}
		}
	}

	for(unsigned int comp = 0; comp < component.size(); comp++)
	{
		for(unsigned int comp1 = 0; comp1 < relyCalc.size(); comp1++)
		{
			if(relyCalc[comp1].ID == component[comp].partID)
			{
				for(unsigned int rely = 0; rely < 
relyCalc[comp1].reliability.size(); rely++)
				{
					relyCalc[comp1].reliability[rely] = 
component[comp].reliability_new[rely];
				}
			}
		}
	}

	Rel_thermal.exportFile(component, fileOut, relyCalc, false);

	return 0;
}
```

APPENDIX I

CODES (STDAFX.H)

```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//

#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>
#include <fstream>
#include <string.h>
#include <iostream>
#include <iomanip>
#include <stdlib.h>
#include <windows.h>
#include <sstream>
#include <time.h>
#include <math.h>
#include <cmath>
#include <vector>
#include <limits.h>
#include <numeric>
#include <algorithm>

using namespace std;
```

REFERENCES

1. Ebeling, C.: An Introduction to Reliability and Maintainability Engineering, McGraw-Hill, New York. 1997.

2. Klutke, G.; Kiessler, P.; Wortman M.: A Critical Look at the Bathtub Curve, IEEE Transactions on Reliability, Vol. 52, no. 1. 2003.

3. Henley, E.; Kumamoto, H.: Reliability Engineering and Risk Assessment, Prentice-Hall, 1981.

4. Lamberson, L.; Kapur, K.: Reliability in Engineering Design, John Wiley & Sons, 1977.

5. Nelson W.: Applied. Life Data Analysis, John Wiley & Sons, 662 pages, 2003.

6. Military Standardization Handbook, MIL-HDBK-217F, Reliability Prediction of Electronic Equipment, US Department of Defense, 1995

7. Reliability Prediction Procedure for Electronic Equipment, Bellcore/Telcordia TR-332, issue 2, 2006.

8. Kamrani, A.; Azimi, M: Systems Engineering Tools and Methods, CRC, New York, 2010.

9. White, M.; Bernstein, J.: Microelectronics Reliability: Physics-of-Failure Based Modeling and Lifetime Evaluation, 2008.

10. Pecht, M.: Electronic Reliability Engineering in the 21st century, Int'l Symposium on Electronic Materials and Packaging, 2001, pp. 1-7.

11. Leonard, C.: Mechanical Engineering Issues and Electronics Equipment Reliability: Incurred Costs without Compensating Benefits, IEEE Transactions on components, Hybrids, and Manufacturing Technology, Vol. 15, No. 6, 1992.

12. Tortorella, M.: Service reliability theory and engineering fundamentals, Quality Technology and Quantitative Management, Vol. 2, No. 1, pp. 17-37, 2005.

13. The Handbook of Performability Engineering, Springer, pp. 269, 2008.

14. Barbati, S.: Common reliability analysis methods and procedures, Reliawind, Edition B, 2009.

15. Condra, L., Reliability Improvement with Design of Experiment, Second Edition, Marcel Dekker, Inc., 2001.

16. Kececioglu, D.: Reliability & Life Testing Handbook, DEStech Publications, Inc., Vol.2, 2002, pp. 877.

17. Akay, H.U.; Liu, Y.; Rassain, M.: Simplification of Finite Element Models for Thermal Fatigue Life Prediction of PBGA Packages, Journal of Electronic Packaging, Vol. 125, 2003, pp. 347-353.

18. Davuluri, P.; Shetty S.; Dasgupta A.: Thermo mechanical Durability of High I/O BGA Packages, Transactions of the ASME, Vol. 124, 2002, pp. 266-270.

19. Torell, W.; Avelar, V.: Mean Time between Failure: Explanation and Standards, Schneider Electric, Rev.1, 1996.

20. Han, B.: Thermal Stresses in Microelectronics Subassemblies, Journal of Thermal Stresses, Vol. 26, 2003, pp.583–613.

21. Kallis James M.; Norris Michael D.: Effect of Steady-State Operating Temperature on Power Cycling Durability of Electronic Assemblies, Proceedings 12th Biennial Conference on Reliability, Stress Analysis and Failure Prevention, 1997, pp. 219-228.

22. Lall, P.; Pecht, M.; Hakim, E.: Influence of Temperature on Microelectronics and System Reliability, CRC, NY USA, 2006.

23. DeGroot, M.; Goel, P.: Bayesian estimation and optimal designs in partially accelerated life testing, Nav. Res. Logist. Quart. 26, pp.223–235 , 1979.

24. Ozbolat, I.; Dababneh, A.; Elgaali, O.; Zhang, Y.; Marler,T.; Turek, S.: A Model Based Enterprise Approach in Electronics Manufacturing, Computer-Aided Design & Applications, 2012, pp. 847-856.

25. Mann, N.; Schafer, R.; Singpurwalla, N.; Wiley, J.: Methods for Statistical Analysis of Reliability and Life Data, John Wiley & Sons; 1 edition, NY, 1974.

26. Romeu, J.; Grethlein, C.: A Practical Guide to Statistical Analysis of Material Property Data, AMPTIAC, 2000.

27. Romeu, J.: Kolmogorov-Smirnov GoF Test, RIAC START, Volume 10, Number 6, 2003.

28. Przemieniecki, J.: Mathematical Methods in Defense Analyses, American Institute of Aeronautics and Astronautics, Inc, Alexander Bell Drive, Reston, VA p. 255, 2000.

29. Engineering Statistics Handbook, NIST, 2003. http://www.itl.nist.gov/div898/handbook/

30. Al-Fawzan, M.: Methods for Estimating the Parameters of the Weibull Distribution, King Abdul-Aziz City for Science and Technology, Riyadh, Saudi Arabia, 2000.

31. Pizka, M.; Deissenboeck, F.: How to effectively define and measure maintainability, SMEF 2007 - 4th Software Measurement European Forum, number ISBN 9-788870-909425, Rome, Italy, May 2007.

32. ADI Reliability Handbook, Analog Devices, Inc., 2000. http://www.analog.com/static/imported-files/quality_assurance/reliability_handbook.pdf

33. Naikan, Reliability Engineering and Life Testing, PHI Learning Private Limited, New Delhi, 2009.

34. Vassiliou, P.; Mettas, A.: Understanding Accelerated Life-Testing Analysis, Annual reliability and maintainability Symposium, 2003.

35. Franck, B.; Adamantios, M.: Temperature Acceleration Models in Reliability Predictions, IEEE, Reliability and Maintainability Symposium, San Jose, CA, USA, January 25-28, 2010, pp.1-6.

36. Escobar, L.; Meeker, W.: A Review of Accelerated Test Models, Statist. Sci. Volume 21, Number 4, 2006, 552-577.

37. www.siliconfareast.com/activation-energy.htm

38. Failure mechanisms and Models for Semiconductor Devices, JEDEC Solid State Technology Association, Publication No. 122C, Revision of JEP122-B, 2003.

39. FIDES Guide 2009.

40. Life expectancy of aluminum electrolytic capacitors (rev.1), NIC Technical Product Marketing Group, New York, 1997.

41. Advanced control systems to improve nuclear power plant reliability and efficiency, IAEA-TECDOC-952, 1997.

42. McElhaney, K.; Staunton, R.: Reliability Estimation Check Valves and other Components, ASME Pressure Vessels & Piping Conference, 1996.

43. Lin, Y.; Chen, X.; Liu, X.; Lu, G.: Effect of substrate flexibility on solder joint reliability, ELSEVIER, Microelectronics Reliability 45 (143–154), 2005.

44. Edwards, D.: PCB Design and Its Impact on Device Reliability, Electronic Design, 2012.

45. Wemekamp, J.: Preparing for Lead-Free Electronics, Military Embedded Systems, October 27, 2009.

46. Bailey, C.; Stoyanov, S.; Lu, H.: Reliability Predictions for High Density Packaging, Proceeding of IEEE HDP, 121-127, 2004. DOI: 10.1109/HPD.2004.1346684.