A study of models for predicting computer software reliability

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:

- A Master's Thesis. Submitted in partial fulfilment of the requirements for the award of Master of Philosophy at Loughborough University.

Metadata Record: https://dspace.lboro.ac.uk/2134/27228

Publisher: © Mona Mahdi Sallih

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 2.5 Generic (CC BY-NC-ND 2.5) licence. Full details of this licence are available at: http://creativecommons.org/licenses/by-nc-nd/2.5/

Please cite the published version.
This item was submitted to Loughborough University as an MPhil thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/
A STUDY OF MODELS FOR PREDICTING

COMPUTER SOFTWARE RELIABILITY

BY

MONA MAHDI SALLIH B.Sc., DIPL.

A Master's Thesis
submitted in partial fulfilment of the requirements
for the award of Master of Philosophy
of the Loughborough University of Technology

Supervisor: DR. I.P. SCHAGEN
Department of Computer Studies

DECLARATION

I declare that this thesis is a record of research work carried out by me, and that the thesis is of my own composition. I also certify that neither this thesis nor the original work contained therein has been submitted to this or any other institution for a higher degree.

M.M. SALLIH.
To my Mother

and my Son.
ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to my supervisor, Dr. Ian Schagen, for his considerable guidance, advice and willingness to assist at any time throughout the programme of this work.

The author acknowledges also, with gratitude, the IRAQI government, especially the Foundation of Technical Institutes, for their financial support to complete this project.

Sincere gratitude and thanks to my husband, for his patience and sacrifice.
# CONTENTS

<table>
<thead>
<tr>
<th>PAGES</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DECLARATION</td>
</tr>
<tr>
<td>ii</td>
<td>ACKNOWLEDGEMENTS</td>
</tr>
<tr>
<td>1</td>
<td>ABSTRACT</td>
</tr>
<tr>
<td>3</td>
<td>Chapter 1: INTRODUCTION</td>
</tr>
<tr>
<td>10</td>
<td>Chapter 2: REVIEW FOR SOME SOFTWARE RELIABILITY RESEARCH</td>
</tr>
<tr>
<td>16</td>
<td>2.1 What is Software?</td>
</tr>
<tr>
<td>20</td>
<td>2.2 Statistical Definitions</td>
</tr>
<tr>
<td>22</td>
<td>2.3 The Development of Models for Software Reliability</td>
</tr>
<tr>
<td>26</td>
<td>2.4 Software Failure Data Collection</td>
</tr>
<tr>
<td>20</td>
<td>2.5 Model Selection Criteria</td>
</tr>
<tr>
<td>22</td>
<td>2.6 Software Reliability Models</td>
</tr>
<tr>
<td>22</td>
<td>2.6.1 Software Reliability Research by</td>
</tr>
<tr>
<td>26</td>
<td>2.6.2 The Probabilistic Model by M. Shooman</td>
</tr>
<tr>
<td>[14]</td>
<td>2.6.3 Fault Removal Model by B. Littlewood</td>
</tr>
<tr>
<td>33</td>
<td>2.6.4 Execution-Time Theory Model by D. Musa</td>
</tr>
<tr>
<td>[13]</td>
<td>2.6.5 The Problem of Modularity</td>
</tr>
<tr>
<td>37</td>
<td>2.6.6 Imperfect Debugging</td>
</tr>
<tr>
<td>40</td>
<td>2.6.7 Fourier Series Model for Describing</td>
</tr>
<tr>
<td></td>
<td>Software Failures by Larry H. Crow &amp;</td>
</tr>
<tr>
<td></td>
<td>Nozer D. Singpurwalla [29]</td>
</tr>
<tr>
<td>42</td>
<td>2.6.8 S-Shaped Reliability Growth Modelling</td>
</tr>
<tr>
<td>44</td>
<td>2.7 A Study of the Result of Fitting Some</td>
</tr>
<tr>
<td></td>
<td>Software Reliability Models</td>
</tr>
<tr>
<td>46</td>
<td>Chapter 3: DEVELOPMENT OF MORE REALISTIC</td>
</tr>
<tr>
<td></td>
<td>SOFTWARE FAILURE MODELS</td>
</tr>
<tr>
<td>46</td>
<td>3.1 Introduction</td>
</tr>
<tr>
<td>48</td>
<td>3.2 Models for Software Failure</td>
</tr>
</tbody>
</table>
3.3 Model 1 - Finite Debugging Time
3.4 Model 2 - Priority for Major Errors
3.5 Model 3 - Combination
3.6 The Simulation Process
3.7 Model (1) Simulation
3.8 Model (2) Simulation
3.9 Model (3) Simulation

Chapter 4: FITTING FUNCTIONAL MODELS
4.2 Predicting for the Future
4.3 Maximum Likelihood Estimation
4.4 Regression on the Failure Rate Curve
4.5 Least Sum of Squares Fitting
4.6 Fitting the Logistic Model to Real Software Data

Chapter 5: CONCLUSION

REFERENCES

APPENDIX A: THE SIMULATION PROGRAM

APPENDIX B: LOGISTIC CURVE FITTING AND PREDICTION PROGRAM
ABSTRACT

In the past twenty years and due to expansion in software systems the problem of software reliability has arisen. Many mathematical models have been developed to describe the behaviour of software package errors and then to get some measures from which the reliability of these software packages can be calculated.

A lot of software failure models assume that all errors are detectable at all stages of testing; eventually this leads to the assumption of monotonically increasing software reliability, characterised by a convex upwards plot of cumulative number of errors versus time. In practice, however, many real software systems show a non-monotonic reliability profile. This is shown by an "s-shaped" curve of cumulative number of errors versus time. Studying this phenomenon reveals that in early debugging the bugs may take time to manifest themselves; once these errors have been detected other bugs may then become apparent. Various reasons for this behaviour come to mind - for example, some bugs may take an appreciable time to fix and during this period no further bugs are detectable. Furthermore, a hierarchy of bugs may exist in which it is not possible to detect lower level bugs until the higher level ones have been fixed. The contents of this thesis involve a detailed study of models of software reliability, its measurements, its requirements and its effect.

Several stochastic models incorporating the above two possibilities have been constructed, and this behaviour explored using simulation techniques. In most cases, the reliability profiles so obtained
correspond quite well to those observed with real software systems. And thus a simple logistic model is appropriate to represent the behaviour of the models. Three methods have been derived to fit the parameters of the logistic curve, using both simulated and real data. The results are found to be very encouraging, especially using the least sum of squares method for the prediction of the reliability of software systems.

The thesis concludes by summarizing the main results obtained and suggestions for future work are included.
CHAPTER ONE

INTRODUCTION
If we think carefully about the process of writing a software system we find it is not as easy a job as it might look, especially when we know that programmers are limited by time and cost specified by their own managers. Under these circumstances programs tend to have errors which make them unable to do what they should do, and this has led to the need to ensure reliability in software systems.

At the beginning let us give a suitable definition of the term software reliability: it is the ability of a computer program to operate successfully without failure, provided that its environment and time parameters have been specified. Furthermore, the specification of failure should also be identified, and these measures should not be changed for a specific project.

The definition of "environment" as given by I&M&O&L [1], reflects the operational profile of the program, i.e. "the relative probabilities of occurrence of various types of run which are characterized by their input states". If a program is used in a different environment, the reliability may be different for each environment.

Although there is not a unique definition of the term failure, we shall list some of these. Littlewood [2], defines it as "a failure is any event which the user thinks of as a malfunction". I&M&O&L [1], define a failure as any departure of program output (control signals, printout, displays, etc) from requirements as the program is executed. Inspecting the definitions above we can deduce that the whole matter is relative and there is no absolute measure.

Software could be accepted but on certain conditions. Putting in measures of successful programs we can define failure as a deviation from these measures.
The basic problem then is to give an appropriate reliability measure and to exhibit a procedure to compute this reliability after inspecting the results of previous tests of the program. Of course, failures are caused by bugs (Basili & Perricone [3] define a bug as something detected within the executable code that caused the module in which it occurred to perform in-correctly). However it is not the number of faults that concerns us but the performance of the program we are after. The reason for this is that sometimes a program containing many bugs performs closer to our requirements than a program containing fewer bugs, and in this matter Littlewood, [2], says "Software reliability means operational reliability who cares how many bugs are in a program? We should be concerned with their occurrence on its operation". Thus the dynamic behaviour is important whatever we call it (failure rate, mean time to failure). However and inspite of what we have said there are some cases where we might wish to know that the software is completely bug free; such a case could be a nuclear power station safety system.

One of the measures of a good computer system is that it has got reliable software, and this can be guaranteed by ensuring that the computer programs it contains are reliable. This implies that any software development should be accompanied by software reliability measures. The reliability of a program is of a static nature - once the program has been completely debugged, it remains reliable. Thus software reliability can provide a good tool for software development evaluation.

But why do we have to show so much concern about software reliability? There are a lot of reasons that have created the need
for this concern and made software reliability as important as hardware reliability. Most important is the increasing usage of computer systems, especially in very critical fields, such as in space defence and traffic control, which makes assuring the reliability of software vital. Another main reason is that it has become clear that computer software is very costly in terms of money and labour. As error detection and error correction are now considered to be the major cost factors in software development, it is worth spending effort to make sure that the programs we are writing are going to work. Also (see [4]) software reliability can play an important role in determining when a system should be released, whether it should be accepted by the user, and the degree of user satisfaction once the system is operational.

Now how can we make sure that a program is reliable? It is not realistic to check out any computer program completely by exercising all the different options to which a program could be exposed, because one of the aims of software reliability is to reduce cost - the exhaustive testing of programs is normally prohibitively costly. Statistical methods have been used in analysing software failures so that specialists can estimate how reliable the program is. By careful choice of samples and input data for testing, the cause of failure could be identified and cured. Some researchers have managed to devise a tool by which we could know which parts of the program have been checked [5]; the statistical methods mentioned above have been used to model this data.

To obtain increased reliability one should spot the cause of failure and remove it. If it is in the program coding the code should
be corrected, or if it is a logical error the logic should be corrected. In the extreme case where the design of the software does not accept certain input the software should be modified.

An important factor in achieving increased reliability is testing. Testing should cover every bit of a program - this means that if the program consists of modules these modules should be tested and the whole program should be tested after the integration process (as the integration process could be a source of failure). When these groups of programs are incorporated into a software system the whole system should be tested also.

The process of testing is very important because it will enable us to collect failure data, which is the key factor in estimating the reliability of software.

In turn we could estimate the time needed for testing to achieve a certain level of reliability of the software system.

During software development, specifically in the testing period, well designed procedures are needed to help people involved in the testing process to take the right decisions.

When a program is written as part of a piece of software a lot of customers will be using it, this program may have a certain number of errors. Continuously testing it will help us to predict its future behaviour, especially when users create additional and new demands on the software which cause errors to appear [6].

The early stages of reliability research applied the methods followed in hardware reliability research to the software reliability problem, but although there is some similarity between software system testing and the hardware life-testing models, there are also
significant differences, because (see [7]), software faults have a design origin while most hardware faults have a physical origin. Also the objectives of the tests are different. In hardware testing the statistical emphasis is often on estimating the failure rate of an item. In software testing the main statistical emphasis is on estimating the number of errors remaining in the system.

Another reason for not using hardware methods is, as Littlewood [8], says "a hardware device is certain to fail eventually, whereas a program if perfect is certain to remain failure free".

Three factors are important in the process of estimating reliability: the model, the data, and the method used in estimating the parameters of the model. These three should all be efficient to get efficient estimation of reliability.

Many models have been developed to investigate software reliability thoroughly and deeply by examining all the factors that might affect the reliability, and by estimating all the variables that could be used to predict the overall reliability, which in turn will push the software development process forward.

I&M&O&L, [1], has defined a software reliability model as follows: "A model is a representation of a random process through which software reliability (or a directly-related quantity such as mean time-to-failure or failure rate) is characterised as a function of time and properties of the software product or the development process (notice the importance of time in designing reliability models), it specifies the general form of the dependence of software reliability on the variables mentioned".

These models usually use one of the known mathematical
distributions to illustrate the behaviour of the software failure process and adopt a stochastic process approach because of the random nature which governs the failure process. Mainly these models use statistical procedures which in turn need accurate and reliable failure data.

A model, (see [1]), generally has a statistical inference procedure associated with it. A model may be applied to a software module, subsystem or a complete system.

The expansion in the process of developing models has made the task of evaluating software reliability very complicated as these models vary in the ways they use to estimate the reliability of a software system. It is the responsibility of the people who work in this field to unify their definitions of software reliability and what causes failure in software, which is not an easy task. However few researchers have reached some agreement about these terms, which could lead in the future to adopting a limited range of models.

Although we have a wide range of models that one could choose from, these models suffer from a lack of accuracy in the predictions they offer. The main reason for this poor prediction is a lack of accurate and available failure data to be used in these models.

In this thesis three models have been developed to incorporate the behaviour of real data which for different reasons gives an "s-shaped" curve when plotting the number of failures versus time. This behaviour is explored using simulation techniques which depend on a random number generator to act as the software system and produce the detected bugs. In most cases the reliability profiles so obtained correspond quite well to those observed with real software systems.
These fairly complex stochastic models do not, however, lend themselves easily to fitting to real software failure data, since they do not yield a simple mathematical form for their failure profiles.

Plotting failure rate against number of failures for these models has shown that a quadratic function is a reasonable fit to this data. A cubic polynomial has also been fitted, but there was little difference between the quadratic and the cubic functions, and thus a simple logistic model is appropriate to represent the behaviour of the models. Various methods for fitting the parameters of the logistic model have been explored, and good agreement achieved between the logistic model and the simulation results. Three methods have been derived to fit the parameters of the logistic curve: a maximum likelihood method, a quadratic regression method, and a least sum of squares method. Each method was tested by fitting part of the data and leaving the rest to be predicted. It was found that increasing the amount of data used in the fitting resulted in more accurate predictions, and that the regression method and the least squares method gave better predictions than the maximum likelihood method. Fitting the logistic model to real software failure data has also been carried out successfully and the results, especially using the least sum of squares method, are encouraging for the prediction of the reliability of software systems.
CHAPTER TWO

REVIEW FOR SOME SOFTWARE RELIABILITY RESEARCH
2.1 WHAT IS SOFTWARE?

Since this work is going to involve a very vital aspect of software, software reliability, it is convenient to give a brief idea about the nature of software.

Software includes the whole series of non-electronic support to computers, in other words, software is the non-physical part of the system [9].

Software consists of an application program (a program is a unique way of communication [10]) whose role is to solve user's problems, and systems programs which handle the problems of computer service.

Some references have referred to a third part of software, and this is the documentation appended with the first two parts to guide us in using and modifying the software.
2.2 **STATISTICAL DEFINITIONS**

Since the reliability models discussed here use statistical procedures in estimating reliability, it is necessary to give a brief idea about major statistical terms used in this thesis.

**The Principle of Maximum Likelihood [11]**

Assume that $x$ is a random variable with a distribution that depends only on some unknown population parameter ($\theta$). Let $(x_1, x_2, \ldots, x_N)$ represent a sample of $N$ independent observations drawn from $x$. Let $L(x_1, \ldots, x_N; \theta)$ represent the likelihood, or probability density, of this particular sample result given $\theta$. For each possible value of $\theta$, the likelihood of the sample result will be different, and the principle of maximum likelihood requires us to choose as our estimate that possible value of $\theta$ making $L(x_1, \ldots, x_N; \theta)$ take on its largest value: $L(x_1, \ldots, x_N; \theta) = \prod_{i=1}^{n} f(x_i; \theta)$.

**Least Sum of Squares [11]**

This is a way of fitting a curve to a set of points, or a method used to estimate the parameters of the model from a given set of data. It is reasonable to require that the curve be such that it makes the errors of estimation small, i.e. makes the sum of squares of the difference between an observed value of $y$ and the corresponding fitted curve value of $y$ a minimum.

**Regression [12]**

The relationship between two variables $x, y$ can be described as the regression of $y$ on $x$. Using regression we can determine if there is a relationship between $y$ and $x$, study the shape of the curve of the relationship and think about the reasons for the relationship.
2.3 THE DEVELOPMENT OF MODELS FOR SOFTWARE RELIABILITY

Models have been developed to predict how reliable our programs are. The process of development has been essentially continuous as some of the recent models are just extensions or modifications to earlier work, by putting in different assumptions and working according to these assumptions.

Early work concerning software reliability took a simple form, such as reporting the problems of software in the form of statistical graphs (Bell Laboratories, 1964) [6], or trying to achieve reliability through the design of the software itself. These attempts show no tendency to use error detection or reliability estimation techniques.

In the mean time the way of thinking about software reliability had been developed and started to take the form of models. These models differ in their way of handling the software failure problem either as a continuous or discrete process [6], and try to choose the optimal statistical procedure with an appropriate distribution.

Other researchers have looked at the matter in a different way - their concern was the time. They have described the failure process using calendar time (hours, minutes,...) or execution time (CPU time) as an independent variable, [13], (models expressed in execution time incorporate the effect of change in work load).

Another approach is that software reliability can be estimated through exposing the software to all possible sets of input data (which is an extremely difficult task); this is called the data domain approach, and about this Littlewood, [2], says "if we know how the program behaves for every conceivable input and could predict future input then I suppose it would be possible to predict the next failure
epoch. Unfortunately we never have such total knowledge". Then some progress was gained by Shooman, [14], who used failure data to estimate the mean time between failures, and tried to correlate between the size of the program and the number of bugs found in the program.

Jelinski & Moranda, [5], employed an exponential distribution in describing their model, concluding that the failure rate is decreased in discrete steps as a function of time and each fault contributes the same amount (\( \phi \)) to the overall failure rate.

B. Littlewood [8] in his model believes that different bugs contribute differently to the failure rate, and he wrote "a bug in frequently exercised code will cause failures more frequently than a bug in infrequently exercised code".

The design of any model should allow early prediction (preferably during module testing), using data from similar previous projects. Shooman [15] says that if early prediction is possible, the information obtained from the prediction process could be used in later stages of development of the software.

Not all of the models developed have been applied to real software projects. The reason is that it is difficult to get accurate and complete failure data which represents the failure history of the project under development or some other project similar to it. Not only that but the models themselves cannot cope with the dynamic behaviour of the programs, which results in inaccurate predictions inherited by these models.

Models vary in the methods they use to estimate reliability. A number of suitable variables are used to measure the reliability of
software packages, such as the number of remaining errors, the mean
time to next error (moment of distribution), the number of failures
in some future time interval [16], and other useful measurements.

Objectives and assumptions are vital factors and should be
considered before any model could be developed, some of the objectives
could be to achieve a certain level of reliability, or that the system
should be able to operate satisfactorily at a given time (t) in a
limited time interval, especially in more complex software systems,
and/or to achieve some economical objectives such as a cut in man-
power and time used.

Putting in the assumptions makes models clear for developer and
user; in this sense Downs, [17], has criticised early work where the
assumptions are not obvious. Models should not be fitted blindly,
for example in cases where data was not available in a form suitable
for our requirements a modification to the model should take place.
Angus, Bowen & Van DenBerg, pp.3-31, [18], show when fitting models
to the JSS project how the Geometric Poisson model is modified to
allow unequal time interval lengths in the input data. Models can
predict future performance of a program only on the assumption that
there is continuity in the behaviour of the programming efforts [8].
So there are some constraints on the periods we are allowed to use
models in, and as Littlewood, [8], mentions, models are used only
after module integration or for the periods of homogeneous behaviour
between module integration.

Models, in general, during the process of prediction either take
the number of failures or the interval between successive failures
into account. Having a numerical procedure and suitable failure data
and knowing the cumulative distribution function of the time between successive failures the reliability measures could be obtained [1]. The procedure used to fit most of the models was to estimate the model's unknown parameters using in most cases maximum likelihood or least squares methods.

If the bug counting approach is adopted, then if \( N(t) \) is the number of errors detected during period \( (0,t) \), we can estimate the number of errors for the future. Models should have the capability to predict future behaviour during the operational phase and also the capability to predict the differences in operational reliability resulting from the implementation of different testing schemas [17].

Since the software environment, for any system, is dynamic and sometimes is highly changeable, it is suggested that the developers of models should incorporate this aspect in their models specifically, by testing the model under different environments.
2.4 SOFTWARE FAILURE DATA COLLECTION

Jelinski and Moranda and some other model developers highlighted the problem of obtaining data that could be used in testing models and getting good and accurate results from these models. They found that there was no software failure data collection for analysis purposes, and as they say "nobody is interested in software failures though everybody is concerned with software reliability" [5]. It has lately been realized that to get good predictions which could improve the quality of the software depends very much on getting access to data, and collecting them needs special statistical procedures.

Careful attention must be paid to the way of collecting data, and records should be kept for every single software system especially for important systems such as nuclear reactor, etc. because the only way to get good estimates of the reliability of software is that the data should have been properly and completely recorded. This should happen as early as possible during the software development phases, as the process of collecting data could take a long time (2-3 years), and all the phases that a software may go through should be included (e.g. coding, testing, integration,...).

From the above discussion one finds it very useful to be able to guess where data reported is incomplete, and where the data is not present it should be inferred. However, a complete data set can be obtained by exercising and executing all the paths and statements in the program. It is essential to try to eliminate the bias in estimation, because some companies do not have complete software failure data or do not have data ready for analysis.

Going into the business of collecting data, it is preferable that
the people who collect data should be experienced, and if not, training should take place. They should be supplied with a unique definition and classification of errors prior to data collection (18). Usually the software is partially debugged so that the information obtained from early debugging can be used in the process of estimating reliability. For accuracy reasons the data collection and validation should proceed parallel with the process of development. The specification of the data must be determined so that the people who collect the data collect the required data because collecting unrelated data would yield wrong or misleading results. Another important factor in the process of collecting data is the examination of data to make sure it is valid, simple to collect, easily analysed, (analysing the data enables us to measure the reliability of our data) and does not contain any contradictions.

To detect errors we need first to classify the errors a piece of software may contain. These errors happen for a variety of reasons (3):

1. Misuse of the programming language
2. Error in the logic of the program
3. Error in the computational theory
4. Error in the use of the data structures
5. Error when correcting other errors

and many other reasons which may cause other kinds of errors.

Error differs not only according to their type, but also in the way they present themselves. For example Basili & Perricone, [3], mentioned that "errors occurring in modified modules are detected earlier and at a slightly higher rate than those in new modules,..., the reason for this is that the causes of error in modified modules
are due to the misinterpretation of the functional specifications". The amount of data puts limitations on our ability to predict for instance, it is difficult to predict the degree of reliability for years ahead when we only have a limited amount of data. The forms for collecting data should be designed carefully. All the possible equations should be included (with a reasonable amount of detail), because a few details will not be informative and too many details will maximise the effort needed in filling in the forms uselessly. These forms are filled in by the people who do the changes to the software, but before setting up questions the aims of the development should be specified. Each aim should have its corresponding question or questions. Aims which we cannot set up questions for should be eliminated [19], because we will not have the necessary data to achieve it.

This way will make it easy to decide the type of questions we want our form to have; without knowing the aims one is likely to obtain data in which either incomplete patterns or no patterns are visible. Further information which could help in setting up the question is the size of the software, the hardware used, etc. It is recommended that reports for each change that happens in the software should be kept separate, to make the process of classifying, checking and validating the data easy.

Data collected about particular software could be used to help in prediction for similar projects. This is very useful, especially when data is difficult or very costly to obtain.

It is important when demanding data from programmers to eliminate their anxiety by assuring them that it is a matter of collecting data not a matter of surveillance, [18].
The type of data needed most are: changes that occur in the software, errors, their type and number and how often they occurred. Further helpful information could be a description of input, the cumulative testing time, and the time to correct each fault.
2.5 MODEL SELECTION CRITERIA

"Incorrect estimation can result from the use of the wrong statistical model. Even though the error in the model is not discernible within the range of the data it may have profound defects beyond the data range" [20].

The above few lines give an indication of how important the choice of a suitable model is. This in turn makes us aware of the need to find the optimal model from among the existing models. Criteria have been established to enable the user to select the best applicable model to estimate the reliability of the software efficiently and accurately and to guarantee its flexibility to work in a wide range of software projects. To achieve this, some researchers have emphasized the need for the people working in this field to agree on a simple software reliability model and then test it to see the degree of its applicability on different software projects. But this is found to be a difficult task. It seems to be a commercial impossibility to derive and prove the validity of a perfectly general software reliability model [6]. The reason for this is that not all of the models have been used, and even the ones used have been tested in different environments. However, there are some criteria which, according to researchers can measure the value of a model and to some extent compare it with the others. But before the evaluation process starts, especially the comparison of two or more models, the following should be noticed: the data collected to test the models must have the same degree of quality, because lower quality data could greatly affect the model performance. Furthermore, since software systems differ in their nature it is useful when we compare models to observe
their behaviour in more than one system, this accounts for the possibility that a model may act differently in different systems.

More specifically a model should have the following characteristics [1]:

1. It must be simple and easy to be understood by the people concerned, and this is useful to determine its ability in application (when, and how).
2. The data the model needs must be collectable.
3. Although it is not preferable to submit incomplete data and, as we said earlier, it should be inferred or a guess should be made where data is missing, however, it is a good idea to plan our model to cope with such situations, i.e. incomplete failure data.
4. Since most programs consist of several modules and these modules will be integrated at a later stage, a model should be able to test each of these modules separately first and before they are integrated to form the complete program.
5. A model should handle the design changes to the program, and its design should account for the degree of importance of the failure of the software.
6. It is preferable that a model can combine the two interpretations: relative freedom from bugs relative freedom from failures in operation (i.e. achieve dynamic reliability of a program) [8].
7. Good models should be capable of predicting consequences of failures, whenever related data is available.
2.6 SOFTWARE RELIABILITY MODELS

The purpose of this section is to supply the reader with some information about some well known models.

2.6.1 Software Reliability Research By Z. Jelinski and P. Moranda [5]

One of the earliest and most important models is the J-M (Jelinski & Moranda) model which forms the basis for most later models. It describes the process of random failures and the reliability growth of computer software as a Poisson process. The time between successive failures of the program \( x_1, x_2, \ldots, x_n \) are exponentially distributed random variables.

What J-M assumed was that there are a large number of independent elements that produce failures, and the rate of these failure occurrences is constant in time. Once an error has occurred, it will be removed immediately without consequence. They have abandoned the idea of uniformity in the failure rate as they say "since once a failure has been detected and corrected the number of failures in the software package is reduced". But, nevertheless, they agree on the idea that there is a uniformity in the periods between the occurrence of successive errors.

They assumed that at any time the failure rate is proportional to the current content of error - it follows that they give all the errors that a program contains the same degree of importance. The initial number of errors (N) is unknown, the contribution of each bug to the total failure rate is \( \phi \), so when the first error is removed the failure rate drops by a fixed amount to \( (N-1)\phi \).

J-M have developed a formula from which they have estimated the
initial number of errors (N) and then from this the proportionality constant $\phi$ is estimated using the software failure data (a realization of $X_1, (x_1, x_2, \ldots, x_n)$).

Since the time between successive error is exponentially distributed, the probability density is given by

$$P(X_i) = \phi(N-(i-1))\exp(-\phi(N-(i-1))X_i), \quad (2.1)$$

where $\phi(N-(i-1))$ is the amount that the failure rate drops to after correcting an error.

Taking the likelihood function, we get

$$L(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} \phi(N-(i-1))\exp(-\phi(N-(i-1))X_i) \quad (2.2)$$

Maximising the log of the likelihood we would have

$$\log L = \sum_{i=1}^{n} \log[(N-(i-1))\phi] - \sum_{i=1}^{n} (N-(i-1))\phi X_i, \quad (2.3)$$

Then differentiating with respect to $\phi$ and then with respect to $N$ and setting the two differentiated equations to zero and $\sum X_i = T$ we get

$$\phi = \frac{n}{NT-\sum (i-1)X_i} \quad (2.4)$$

and

$$\frac{1}{N} + \frac{1}{N-1} + \ldots + \frac{1}{N-(n-1)} = \frac{n}{N-\frac{1}{T} \sum (i-1)X_i} \quad (2.5)$$

Obviously equation (2.5) should be first solved to estimate the value of $N$ so that we can estimate $\phi$. Rewriting (2.4) in terms of the estimated values,

$$\hat{\phi} = \frac{n}{\hat{N}T-\sum (i-1)\hat{X}_i} \quad (2.6)$$

J-M made some refinements in which they account for a variable exposure rate (this reflects the intensity of testing both in hours...
per day and in the number of different sites which are independently testing or using the same software).

They introduced a variable \( T \) where

\[
T = F(t) = \int_0^t E(u)du
\]

where \( E(u) \) is the exposure rate

\( F(t) \) is the failure rate.

They stated that, when \( E(t) > 0 \), \( F(t) \) is increasing. If \( E(t) = 0 \) then the function \( F(t) \) will stay constant but there will be no production of failures since there is no exposure during this period. According to this they modified the model by replacing \( X_i \) by \( \hat{X}_i \) where \( \hat{X}_i = \tau_i - \tau_{i-1} \) and \( t_i - t_{i-1} \) by \( \tau_i - \tau_{i-1} \).

The modified model is given by

\[
P(X_i) = [N-(i-1)] \phi e^{-[N-(i-1)]\phi \hat{X}_i}
\] (2.7)

The J-M model has been very widely used, but it has been found that it is not efficient in terms of its assumptions and the process of parameter estimation. This has been reflected in the inaccurate predictions obtained from applying this model. The model has faced a lot of criticism [8], [21], [22] and Littlewood [8] has tried to modify it. The main criticism was actually of the assumption that software errors have the same degree of severity [8], [23]. Littlewood's [8] answer to this assumption is "A bug in frequently exercised code will cause failures more frequently than a bug in infrequently exercised code". N. Langberg & N.D. Singpurwalla [24] suggested a remedy to this by finding an alternative survival function for \( X_i \) (the time between the \((i-1)\)th and the \(i\)th failure of software).

They used a notation from the theory of shock model and describe
the software input as a Poisson process with \((N^*)\) as the total number of "distinct input types" and \((W)\) as the intensity function by which the inputs arrive at the system. \(N \leq N^*\) where \(N\) is the number of inputs which cause failures to the system and this is unknown. They give the following formula as the new survival function,

\[
\bar{F}_i(x|N,N^*) = P(X_i \geq x|N,N^*) = \exp\left(-\frac{W(N-i+1)t}{N^*}\right), \quad t > 0
\]

(2.8)

to replace the J-M survival function

\[
\bar{F}_i(x|N,\phi) = \exp(-\phi(N-i+1)x), \quad x > 0
\]

(2.9)

where the latter is a special case of the previous one. If we note that \(\phi\) in (2.9) is \(W/N^*\) in (2.8). In [8] Littlewood stated that the J-M model is a special case of his model.

The other criticism was of the assumption concerning error removal. They assumed that errors when removed will be removed with certainty. This is not realistic because correcting an error could be a source of generating more errors (i.e. either we fail to correct the error, or correcting a bug produces another bug or bugs). This assumption has been adopted by most researchers because not using this assumption would bring us in to the field of imperfect debugging modelling.

The other criticism was about the method followed in parameter estimation. [16], K&L&M&S said in their paper that predictions obtained by applying ML analysis of the J-M model were extremely poor.

Littlewood & Verrall [21], criticised the method itself, saying that a lot of researchers have used this method for prediction, but it can be shown that for some sets of data it has produced infinite values of \(N\).
2.6.2 The Probabilistic Model by M. Shooman [14]

Shooman has developed two models: an error and a reliability model.

In the error model he assumed that the number of errors in a program is fixed and the number of remaining errors is the result of subtracting the number of corrected errors from this fixed initial value, so that again, like J-M, he assumes perfect debugging. He introduced the total number of instructions and the time spent on debugging along with the number of errors as important factors for the probability of encountering an error.

He calculated what he called a normalized error rate \( P(\tau) \) where,

\[
P(\tau) = \text{errors/total number of instructions/ month of debugging time.}
\]

According to the above formulae the normalized error rate \( P(\tau) \) has been calculated for some data and curves have been drawn of these \( P(\tau) \) results as a function of \( \tau \), where \( \tau \) is the number of months of debugging after the beginning of system testing. These curves were found to have one property and that was that these normalized error rates decreased over the entire curve, or at least over the latter half of the curve. Then he expressed the area under the curve \( P(\tau) \) by

\[
\varepsilon(\tau) = \int_{0}^{\tau} P(X) \, dX = \text{cumulative errors/total number of instructions}
\]

and the error rate \( P(\tau) = \varepsilon(\tau) / \tau \) is the slope of the \( \varepsilon(\tau) \) curve.

The normalised number of residual errors is given by

\[
\varepsilon^{(r)}(\tau) = (E_{T}/I_{T}) - \varepsilon(\tau)
\]

where \( E_{T}/I_{T} \) is number of errors per instruction (assuming that the total number of errors in the program \( E_{T} \) is constant and that the
program contains \( I_r \) instructions), and \( s_c(t) \) is the number of errors corrected (assuming all the errors detected are corrected).

The second model he developed (using the error model) is the reliability model, where he first defined errors in the software in terms of its operational phase, by the probability that a certain procedure contains an error and this procedure has been exercised by the program.

Two terms have been introduced to construct the failure rate \( z(t) \) and these are:

- \( r \), which is an instruction processing rate (taking into account the nature of the instruction and the input it receives),
- a constant \( k \) to give the appropriate weight to the effect of an error on the performance of a program. He suggests that to estimate this constant we need to know how many of the detected errors are catastrophic errors:

\[
k = \frac{\text{number of catastrophic errors detected}}{\text{total number of errors detected}}
\]

So the probability of failure in interval \( \Delta t \) is

\[
p(t < t_f < t + \Delta t | t_f > t) = z(t) \Delta t = k r (r) r_p \Delta t
\]

where \( t_f \) is time of failure,

and the reliability function in the time interval \( 0 \) to \( t \) is

\[
R(t) = \exp\left(- \int_0^t z(x) dx\right)
\]

\[
R(t) = e^{-rt}
\]

where \( r \) is the failure rate,

which means that the reliability function is an exponential function of operating time. This function decreases as operational time passes and increases as debugging time increases.

Shooman summarised his model by deriving three equations for the
MTTF (Mean Time to Failure). Firstly, he modelled the normalized error rate \( \rho(T) \) by a constant rate of error correction \( \rho_0 \) so

\[
MTTF = \frac{E_T}{k(T) [\frac{I_T}{p(T)} - \rho_0(T)]}
\]  

(2.12)

Secondly, a triangular rate of error correction was used. The normalised error rate \( \rho \) goes up linearly from the origin to the point \( \rho_p \), consuming time \( \tau_p \), then declines linearly to zero taking time \( 2\tau_p \), forming a triangle. Reaching this point the software is completely debugged. So two MTTF's are derived:

\[
MTTF = \frac{1}{\beta (1 - \delta T^2)} \quad 0 \leq T \leq \tau_p
\]

(2.13)

\[
MTTF = \frac{1}{\beta [1 + \delta (2T^2 - 4T\tau + \tau^2)]} \quad \tau_p \leq T \leq 2\tau_p
\]

(2.14)

where \( \beta = \frac{E_T}{I_T} \frac{r_p k}{p} \)

and \( \delta = \frac{\rho_p / 2\tau_p}{P_r} \frac{1}{I_T} \).

The third MTTF equation is for an exponentially decreasing model for \( \rho(T) \)

\[
MTTF = \frac{1}{\beta \exp(-k_1 T)}
\]

(2.15)

where \( \beta \) has the same value as above.

Shooman made a comparison between the three models by plotting the resulting MTTF's against the normalized time. It was found that all the models showed an increase in MTTF, but the exponential model shows this increase earlier than the other two.

In developing the exponential error rate model he assumed that the number of errors detected per month is proportional to the number present (criticized by Littlewood [2]). If \( \rho(T) \) denotes the exponential error detection rate then,

\[
\rho(T) = \frac{K_1 E_T}{I_T} \exp(-k_1 T)
\]

(2.16)

where \( K_1 \) is the proportionality constant.
Shooman tried to include the effect of varying the number and efficiency of the people working in the debugging process. He suggested that the number of man months should be recorded. Thus, if

\[ p = \frac{\text{errors/instruction/month}}{p' = \frac{\text{errors/instruction/man month}}{p = M p'} \]

where \( M \) is the average manpower working on debugging a program per month.

The manpower effect is included in the model by introducing a new proportionality constant \( K_2 \) such that

\[ K_1 = K_2 M \]

\[ \varepsilon(\tau) = \frac{E_T}{I_T} (1-\exp(-K_2 M t)) . \quad (2.17) \]

2.6.3 Fault Removal Model by B. Littlewood

A step forward, which contains a lot of criticism of early models, is Littlewood's attempt, [25], in which he criticised the idea that random discovery of faults is the only source of uncertainty. He said that the successive improvements in the overall failure rate of the program caused by removal of successive faults will be different from each other and these improvements will be uncertain, and that it is impossible to measure the improvement even after the removal of a fault. Furthermore not only removing an error increases reliability, but also the operating of a program without failure for a certain period of time.

The model is based on the following assumptions:

Firstly, there are \( N \) faults in the program and the time between
failures is exponentially distributed and each fault $i$ has an independent failure rate $\phi_i$. The sum of these failure rates will form the total failure rate of the program ($A$). Secondly, he assumed that the faults are immediately removed (perfect debugging) and the repair time is not significant.

Assuming that $N-i$ faults remain in the system, then the current program's failure rate would be

$$\lambda = \phi_1 + \phi_2 + \ldots + \phi_{N-i}$$

which show unknown different contributions to failure rates by different faults. So the failure rate of the program is

$$\text{Pdf}(t|A=A) = \lambda \exp(-\lambda t) \quad (t>0), \quad (2.18)$$

He described how the failure rate of a program changes with time and derived two equations. The first to calculate the current reliability of a piece of software is probability of no failure in time $t$

$$S_f(t) = \left[\frac{\beta + \tau}{\beta + \tau + t}\right]^{(N-i)\alpha} \quad (2.19)$$

Secondly, to calculate the current failure rate

$$\lambda(t) = (N-i)\alpha/(\beta + \tau + t), \quad (2.20)$$

where $i$ is the number of faults that have been removed, and $\tau$ is the total execution time which varies according to the time spent in debugging.

$N, \beta, \alpha$ are parameters which have to be estimated during the debugging process (one parameter more than the J-M model). The most important thing to notice about the model's assumptions is that the times between failures of a piece of software $T_1, T_2, \ldots$ have a decreasing failure rate (DFR), and these $T_1, T_2, \ldots$ are stochastically increasing.
The model describes how the faults which make the greatest contribution to the overall failure rate tend to show themselves earlier. This has been expressed as follows:
the amount by which the failure rate is reduced is $\alpha/(\beta + \tau)$; when a fault is being fixed at early debugging, $(\tau)$ is small so the expression $\alpha/(\beta + \tau)$ is big but in a later stage of debugging $\alpha/(\beta + \tau)$ is small due to larger values of $\tau$.

Littlewood introduced a schema to predict future reliability. He approaches the problem from two points of view: firstly an equation is derived to estimate reliability after a certain amount of execution time has elapsed. Thus the reliability function is:

$$R(t) = [1 - \left(\frac{\beta + \tau}{\beta + t + \tau'}^\alpha + \frac{\beta + \tau}{\beta + t + \tau'}^\alpha\right)^{N-1}],$$  \hspace{1cm} (2.21)

where $1 - \left(\frac{\beta + \tau}{\beta + t + \tau'}^\alpha\right)$ represents the probability that a failure occurs in time $t'$. Using the last expression with some information about the down time of failures will help to estimate the number of failures in a given time.

Secondly, estimating the reliability after eliminating a certain number of errors when $i$ failures have happened and we are expecting more failures ($K$) to take place, the measure MTTF is found to be appropriate for such a purpose:

$$E(\lambda) = \frac{(N-i-K)\alpha}{\beta + \tau} \cdot \frac{(N-i-K+1)\alpha}{\beta + (i+1)\alpha} \cdot \frac{(N-i-K+2)\alpha}{(N-i-K+2)\alpha + 1} \cdots \frac{(N-i)\alpha}{(N-i)\alpha + 1}$$  \hspace{1cm} (2.22)

A maximum likelihood method has been used to estimate the parameters $\alpha, \beta, N$.

Also he derived some other useful formulae to estimate the time
needed to make the software failure free. Littlewood, [21], had discussed the problem of using maximum likelihood as a way of parameter estimation, especially the problem of getting infinite estimates of N for some sets of data. He suggested a test to be carried out and if the test is passed by a given set of data then a maximum likelihood method can be used. This condition is:

If (i-1) is the number of errors removed

t_i is the observed time between failures

n is the number of failures already observed

then the following must hold:

$$\frac{\sum_{i=1}^{n} (i-1)t_i}{\sum_{i=1}^{n} (i-1)} > \frac{\sum t_i}{n}$$

If this condition is not satisfied this will indicate that the estimation of N would be infinite, because this condition guarantees that the regression line of T_i (the time between successive failures) on i has a positive slope.

Littlewood, [21], compares the assumptions of his model with the J-M model. He says that "there are two sources of uncertainty surrounding the reliability growth taking place during debugging "one of these is the flow of input; as he said, knowing the current failure rate of a program (\(\lambda_1\)) does not give an idea about the time between successive failures T_i. The other source of uncertainty is the "bug-fixing" process. The J-M model assumes that the amount the failure rate (\(\lambda_1\)) drops by is known, so we are left only with the successive times between failures as a source of uncertainty. Littlewood, [8],
has argued that it is not violating the reliability measure for a debugger to end the debugging process before eliminating all the bugs from a program. We may find at later stages of debugging that the estimated number of errors $N$ gets big, but this is as a consequence of getting smaller improvement in the reliability and not because of increasing the estimation of failure rate.

2.6.4 Execution-Time Theory Model by D. Musa, [13]

This model is based on execution (cpu) time. The assumptions are mainly similar to that of J-M model, i.e. the failure intervals are independent and the execution time is exponentially distributed with constant failure rate changes. Whenever a fault is corrected, the failure rate is proportional to the number of faults remaining. This is represented by the proportionality constant which caters for faults caused by known and unknown reasons. Musa differentiates between the operational phase where faults are detected and the test phase where faults are corrected.

A formula is derived to estimate the number of faults detected and corrected in an execution time $T$:

$$n = N_0 \cdot \left[1 - \exp \left( -\frac{cT}{M_0T_0} \right) \right]$$

(2.23)

where $N_0$ is the remaining bugs in the system, $M_0$ the total number of faults that cause failure, $T_0$ is the MTTF at the start of the test, and $c$ is the ratio of equivalent operation time to test time.

There is a relation between the number of errors in the system and the errors that cause failures, through the proportionality factor which will be denoted by $B$, as follows:
\[ N_0 = BM_0, \quad n = Bm \]

Then the expected number of faults that cause failure is given by

\[ m = M_0 \left( 1 - \exp \left( - \frac{CT}{M_0 T_0} \right) \right) \quad (2.24) \]

The present mean time to failure (MTTF) measured by execution time is

\[ T = T_0 \exp \left( \frac{CT}{M_0 T_0} \right) \quad (2.25) \]

and the reliability for the coming period of time \( \tau' \) is

\[ R = \exp \left( - \frac{\tau'}{T} \right) \quad (2.26) \]

Another useful equation is derived to measure the execution time needed to increase MTTF from \( T_1 \) to \( T_2 \)

\[ \Delta T = \left( \frac{M_0 T_0}{C} \right) \ln \left( \frac{T_2}{T_1} \right) \quad (2.27) \]

In the calendar-time component of the model, he assumed the following:
the resources available, consisting of personnel to identify (I) and correct an error (F) and the computer time (C) are fixed through the test period. The failure correction is a Poisson process and the failure-correction effort is exponentially distributed. He also assumed that the expenditures needed to increase MTTF depend on computer time which in turn depends on execution time and the number of failures, and is given by the equation

\[ \Delta x_k = \theta_k \Delta T + \mu_k \Delta m \quad (2.28) \]

where \( \Delta T \) is the increment in the execution time, \( \Delta m \) is the increment in the faults that cause failure, \( \theta_k \) is an execution time coefficient of resource expenditure and \( \mu_k \) is a failure coefficient of resource expenditure.

The process of correcting errors is illustrated as taking items
from a queue filled by a debugger with a limited length \( m_Q \). The value of \( P_F \) (failure-correction personnel utilization factor) is calculated:

\[
P_F = (1 - \frac{1}{m_Q})^{1/m_Q}
\]

(2.29)

where \( P_F \) is the probability level for a certain queue length \( m_Q \).

Then he derived a formula to relate the calendar-time intervals to the execution time intervals. He introduced two MTTF, \( T_{k_1}, T_{k_2} \), where each of these represent MTTF at the beginning and at the end of the time interval.

We have \( k, k' \), where these indices take a combination of any two of \( C, I, F \), then MTTF for this combination would be given by

\[
T_{kk'} = \frac{C(P_{k}, P_{k'}, \rho_{k}, \rho_{k'})}{P_{k}, P_{k'}, \rho_{k}, \rho_{k'}}
\]

(2.30)

where \( P \) is the resource quantity, \( \rho \) is the resource utilization, \( k, k' \) have the values \( (C, F), (F, I) \) and \( (I, C) \).

The transition points are those values of \( T \) at which the derivative of calendar time with respect to execution time resulting from one resource become greater than that from another.

### 2.6.5 The Problem of Modularity

A module is a named subfunction, subroutine or the main program of the software system [3]. Software modules have to be integrated in the later stages of software development. This means that these modules should be put in order and connected with each other to perform the required task within the software subsystem or the software system for which they are written.
Some researchers have looked at the problem of modularity and its effect on software failure, as failures might increase when these modules are integrated with each other. It is found that the failure rates in each module are less than the transmitting rate between the modules. Solutions have been suggested, such as that programmers should produce some kind of detailed specification explaining how these modules are going to communicate with each other.

A semi-Markov law has been adopted, [23], to describe the exchange of control between program modules, because it believed that the probability $P_{ij}$ (where $P_{ij} \subseteq \rho_i$ (program transmits from module $i$ to module $j$)) of calling a given module from another module is a function only of the calling and called module. It is also assumed that the time spent in each module is a random variable with a distribution affected by these calling and called modules.

Going back to the problem of software failure, the process is described as a stochastic process, with the randomness coming from two sources. The first is which module would be called next and the second is the time spent in each module. A formula has been derived to calculate the overall failure rate of a program. The total number of failures occurring in each module is summed, and this total is added to the total number of failures occurring in transitions between the modules. Thus the total failure rate would be:

$$\sum_i a_i V_i + \sum_{i,j} b_{ij} \lambda_{ij} , \quad i \neq j ,$$

where $V_i$: the failure rate for each module

$\lambda_{ij}$: the failure rate between the modules

$a_i$: the proportion of time spent in module $i$

$b_{ij}$: is the frequency of $i$-$j$ transfer of control.
Littlewood suggests that the a's and b's could be obtained directly by monitoring software testing. The above discussion would direct us to think deeply about the problem of how much test effort should be spent before and after the integration phase. Kubat & Koch [26], designed a decision procedure with inputs \((N_i, Q_i)\), \(i = 1, \ldots, M\), where \(N_i\) is the mean number of errors remaining in module \(i\), \(Q_i\) is the mean detection time of a failure in module \(i\) at the current time point. Knowing the amount of time allocated to do the testing will help to identify and eliminate as many errors as possible.

Mathematically we want to minimize the following:

\[
\sum_{i=1}^{m} W_i N_i \exp(-t_i/Q_i)
\]

where \(N_i \exp(-t_i/Q_i)\) is the mean number of errors still remaining in module \(i\) after testing for time \(t_i\). \(W_i\) is a weight measuring any cost or damages resulting from a removal of a fault from module \(i\), \(t_i\) is the allocated testing time to module \(i\), \(t_i > 0, i = 1, \ldots, M\).

They conclude by applying the above formula to several examples. They managed (with optimal allocation of test time) to reduce the remaining bugs by 50%.

2.6.6 Imperfect Debugging

It has long been known that the debugging process is one of the sources of uncertainty, in software development, since correcting an error does not mean that our program has been freed from that error. There is always a probability that this error will remain in the program, or even worse cause other errors to rise to the surface.

People working in software reliability have tried to model this
phenomenon. Their work is based on some assumptions, such as a bug will be removed only when a failure happens, and that only one fault is fixed at a certain time (this could create a problem unless we regard the faults of one type appearing at the same time as one fault).

There is another problem, and this is what Downs, [17] called the "obscure failures" problem. This means that a failure happens and because of a lack of information no effort is spent to identify the error.

Kubat and Koch, [27] denote the probability that a failure will be removed by \( P \). Then the probability that the bug will remain in the system is \( 1 - P \), and the decrease in the error rate will be given by the average number of errors fixed at the time of error discovery. So:

\[
\text{the average proportion of errors fixed} = 1 - \frac{\text{no. of errors reappearing during testing}}{\text{total number of errors detected}}
\]

(2.31)

Downs, [17], has argued the same way, saying that an error either creates another error (only one error is created because he assumed that the numbers of errors in the software would be the same) or an error cannot be found.

He denotes the probability that the number of faults will stay the same after an attempt to remove a fault by \( q \) and the probability that this number would be increased by \( q^{r+1} \). \( r \) is the increment in the no. of faults in the software after an attempt of fault removal.

A formula is derived to account for the expected number of bugs removed in any instant by the removing of a bug \( R \) as follows:

\[
R = \sum_{k=0}^{\infty} \frac{1}{k!} k \Pr(k \text{ faults are removed})
\]
Notice the range of $k$, which means either adding further bugs or at most one will be removed.

Goel & Okumoto, [28], also were interested in imperfect debugging. They assumed that errors were corrected with probability $p$ and with probability $q$ the error is not. The process of debugging goes on as long as $0 < p < 1$.

If the total number of errors is $N$, then

$$X(t) = N, \, t = 0.$$  

Then at some time after debugging starts the state of $X(t) = i$ either change to $X(t) = i - 1$ with probability $p$ or remains unchanged with probability $q$ (i is the number of errors at some time). The essential part of the model is the following formula derived to calculate the probability that the program during debugging will change from state $i$ to state $j$ at time $t$ ($Q_{ij}(t)$), (where $i$ represents the state of the software before removing the error and $j$ is the state after removing that error). First the probabilities $Q_{ij}(t)$ are obtained as follows: $Q_{N,N-1}(t) = P_{N,N-1} \cdot F_N(t)$, i.e. multiplying the probabilities $P_{ij}$ (which are the elements of a sparse matrix with two diagonals of $p$s and $q$s) by $F_i(t)$ (which is the cumulative distribution function for $i$ ($i$ is exponentially distributed). Thus the probability $Q_{ij}(t)$ would be obtained from the following sparse matrix:

$$\begin{bmatrix}
0 & 1 & 2 & \cdots & \cdots & N-2 & N-1 & N \\
1 & \cdot & \cdot & \cdots & \cdots & \cdot & \cdot & \cdot \\
F_1(t) & \cdot & \cdot & \cdots & \cdots & qF_1(t) & \cdot & \cdot \\
N-1 & \cdot & \cdot & \cdots & pF_{N-1}(t) & \cdot & \cdot & \cdot \\
N & \cdot & \cdot & \cdots & \cdot & qF_N(t) & \cdot & \cdot \\
\end{bmatrix}$$
2.6.7 Fourier Series Model for Describing Software Failures by Larry H. Crow & Nozer D. Singpurwalla [29]

There is very useful relationship between the Poisson and exponential distributions: "if the number of occurrences of an event in a given unit of time has a Poisson distribution with parameter \( \mu \), then the time between successive occurrences of the event will have an exponential distribution with parameter \( \beta = 1/\mu \)" [11]. Most of the known models adopt the above fact, assuming that the times between program failures are independent and identically exponentially distributed random variables. Crow & Singpurwalla claim that this might be true when bugs are fixed each time a failure happens, but what happens really is a clustering of software failures, (they define clustering as the grouping of similar objects). A Fourier series model is developed to analyse clustered software data.

They used a device called spectrogram to determine if there is clustering in the data and whether the clustering is systematic, since systematic clustering can be reasonably well predicted. The failure behaviour for complex stochastic-state computer systems is investigated. Software failure occurs in clusters if the time between successive failures are short for some failures and long for the others. To reveal clustering a time series (a sequence of observations indexed by time) analysis technique is used. A test was conducted on a sequence of cumulation times \( T_1 < T_2 < \ldots < T_{n+1} \), and the result was that the times between failures were found not to be independent and identically exponentially distributed. Also a test was made for clustering and for cyclical trend.

Now if the cyclical pattern is denoted by \( f(i) \) then \( f(i) \) can be
expressed as a linear equation, consisting of sine and cosine terms
(Fourier series representation of \( f(i) \)). Assuming \( n \) is odd and let
\( q=(n-1)/2 \). Then, for \( k_j=j, j=1,2,\ldots,q \) we may write:

\[
f(i) = a_0 + \sum_{j=1}^{q} a(k_j) \cos \frac{2\pi}{n} k_j i + b(k_j) \sin \frac{2\pi}{n} k_j i \quad (2.32)
\]

The least squares method is used to solve for \( a_0, a(k_j), b(k_j) \).

The spectrogram (i.e. a plot of \( \rho^2(k_j) = a^2(k_j) + b^2(k_j) \) versus
the frequency \( k_j/n \) for \( j=1,2,\ldots,q \) can be used to discover periods
in a time series by identifying the frequencies \( k_j/n \) associated with
value of \( \rho(k_j) \) which are larger than the others.

If \( \rho^2(k_j) \) values are equally large at all the \( q \) frequencies \( k_j/n \)
we conclude that there is no sign of clustering in the data.

This spectrogram is used in developing a parsimonious model.

The Fourier series representation of the trend \( f(i) \) consists of \( n \)
terms with coefficients \( a(k_j), b(k_j), a_0, k_j=1,\ldots,q, \quad q=(n-1)/2 \) if
\( n \) is odd, and \( q=n/2 \) if \( n \) is even. Using the spectrogram, the \( a(k_j) \)
and \( b(k_j) \) which correspond to small values of \( \rho^2(k_j) \) are abandoned,
while the values of \( a(k_j) \) and \( b(k_j) \) for which \( \rho^2(k_j) \) is large are
kept to analyse the behaviour of the data and then to be used in the
Fourier series model.

Now if this is defined as follows:

\( k_j=j, j \in I \), where we have large value of \( (k_j) \), the parsimonous
model for the times between software failures is:

\[
f(i) = a_0 + \sum_{k_j=i, i \in I} a(k_j) \cos \frac{2\pi}{n} k_j i + b(k_j) \sin \frac{2\pi}{n} k_j i. \quad (2.33)
\]

The model has been applied to 3 sets of real data, each set representing
sequences of times between failures of a software system, as the time
time between detectable errors is observed. The values of \( f(i) \) (estimated
Fourier series) were obtained and plotted against the observed data
\( t_i \) to determine how closely the model fitted the data. Also from
the plotting a periodical clustering is obtained and the values
associated with large \( p^2(k_j) \) were used in the Fourier series model.

2.6.8 S-Shaped Reliability Growth Modelling

S. Yamada, M. Ohba and S. Osaki, [30], have modified the Goel &
Okumoto model [31,32,33] so that it can cope with real data behaviour.
which takes an "S-shaped" form when plotting cumulative errors detected
versus time.

A mean value function which shows an S-shaped growth curve has
been incorporated in the new model. The main assumptions of their
model are:

1. System failures occur randomly.
2. The initial number of errors is a random variable.
3. "The time between failures \((k-1)\) and \((k)\) depends on the time to
   failure \((k-1)\)".
4. Errors are removed instantly with certainty.

If \( N(t), t \geq 0 \), denotes the cumulative number of software errors detected
up to time \( t \), \( a \) is the expected number of errors to be detected,
\( b \) is the error detection rate, \( M(t) \) is the mean-value function in the
NHPP (Non-Homogeneous Poisson Process) model having an S-shaped growth
curve.

Thus the reliability (the probability of no failure in time
\((s,s+t))\),
\[
R(t|s) = \exp[-\{M(t+s)-M(s)\}]
= \exp[-a((1+bs) e^{-bs}-(1+b(t+s)) e^{-b(t+s)})]
\]

and the expected number of remaining errors is given by,

\[
n(t) = E[N(\infty)-N(t)] = a(1+bt)e^{-bt}.
\]

They used a maximum likelihood function to estimate the model parameters, obtaining the following two equations to be solved numerically when data is available, the data set used in the form of pairs \( (t_k, y_k) \) \( k=1,2,\ldots,n \), where \( y_k \) is the number of software errors detected up to time \( t_k \)

\[
y_n = a(1-(1+bt_n) e^{-bt_n})
\]

\[
2 e^{-bt_n} = \sum_k \frac{(y_n-y_{k-1})(t_k^e t_{k-1}^e-b t_k-b t_{k-1})}{(1+bt_{k-1}) e^{-b t_{k-1}}-(1+bt_k) e^{-b t_k}}
\]

The model has been applied to real data and by applying the Kolmogrov-Smirnov goodness-of-fit test they have shown that at a 5% level of significance the S-shaped NHPP model with the values obtained from the real data for \( a,b \), were \( a=37.9 \), \( b=0.312 \), adequately fits the data. (The number of errors detected and used in the model was 31, and the actual number was 42, the estimated number was 37.9).

They compared the model with three other models: the NHPP for (Goel & Okumoto) and two models of fitting curves (logistic and Gompertz curves). The comparison was based on regression analysis, using least squares method and obtaining two measures, the sum of squares of the differences which found to be less than the other models for the NHPP model with \( M(t) \). The model also gives a reasonable estimation of the number of errors to be eventually detected, also better than the others.
2.7 A STUDY OF THE RESULT OF FITTING SOME SOFTWARE RELIABILITY MODELS

A very intensive test of some of the above models was carried out by applying six software reliability models to a major command control communication and intelligence system [18].

The models were: the Geometric Poisson (Morandas [34], Non-homogeneous Poisson, (Goel & Okumoto, [32]), imperfect debugging (Goel, [28]). Generalized Poisson (Schafer, et al [35]), IBM Poisson (Brooks & Motley, [36]), and Binomial (Schafer et al [35]).

These models are fitted to data collected from current (unfinished) projects relating to U.S. Defence software.

A pseudo-maximum likelihood or pseudo-least squares principle was used to estimate the models' parameters.

Inspite of certain mathematical modifications and careful use of the project data, none of these models performed efficiently and their representation of the data was very poor. This is shown clearly in the divergence of the procedure which was used to estimate the model parameters.

These results come from the fact that all the models suggest a curve convex upwards when plotting the errors detected versus time. Plotting the data, this curve should be considered to be "S-shaped". This study [18] reveals that these models' similar poor performance is not a coincidence. This is mainly because their underlying assumptions are the same, and similar to the original J-M model. Because of this similarity these models face the same criticisms, especially of their two main assumptions:—1. all errors have the same constant rate of occurrence, 2. the expected number of errors
to be detected in a time interval is proportional to the number of errors remaining. Not only the above models have these assumptions, but also others (e.g. Shooman [37], Musa [38]), [16], show some similarity to the original J-M model. And even some of these researchers' assumptions violate the actual behaviour of their data, (e.g. Shooman has assumed a monotonically decreasing error rate, while when he plotted the error rate against time (pp. 499-500, [14]) a glance at these shows it is well fitted by a quadratic function). Thus we need models of the debugging process which adopt an alternative approach (different to the monotonically decreasing error rate). To this we shall turn our attention in the next chapter.
CHAPTER THREE

DEVELOPMENT OF MORE REALISTIC SOFTWARE FAILURE MODELS
3.1 INTRODUCTION

To summarise briefly some of the major ideas discussed in the previous chapter, we said that the basic idea of previous models (see J-M [5]) is that we have N bugs and each of these bugs has an occurrence rate \( \lambda_i \) which has a given probability distribution. For example a simple version of this general class of models is:

\[
f(\lambda) = a \exp(-a \lambda) \quad \text{for} \quad \lambda > 0,
\]

where \( a > 0 \) is a fixed constant and this leads to the formula

\[
\Lambda(t) = \frac{Na}{(\alpha + t)^2}, \quad (3.1)
\]

to calculate the total failure rate at a given time \( t \). This leads to the formula

\[
n(t) = \frac{Nt}{\alpha + t}, \quad (3.2)
\]

to calculate the expected number of bugs when a time \( t \) has elapsed.

From the above example, it can be seen that models based on the assumption that all bugs are detectable at all times leads inevitably to monotonically decreasing failure rates and convex upwards plots of numbers of bugs detected against time (see Figures 3.1, 3.2).
Many variants of this original simple J-M model have been proposed but they all make the assumption of monotonically decreasing failure rate as a function of number of errors detected, in other words, what is called "reliability growth". Angus, Bowen and Van denberg [18], attempted to fit several different software failure models to a range of data that they had available relating to U.S. Defence software. They found disappointing results - no one model would adequately represent even most of the data. Five of their data sets have been plotted in Figure 3.3, in terms of number of errors detected versus elapsed time from the start of testing. Only two of these (data sets 3 and 4) could be realistically fitted by a model displaying continuous reliability growth - the other three show quite different behaviour. In practice it has been found that plotting the number of errors detected against time gives an s-shaped curve, ([18], pp.3-21, 3-32), rather than a convex upwards curve. Studying this phenomenon reveals that in early debugging the bugs take time to manifest themselves. Once these errors have been detected others may become apparent.

Therefore the idea that the rate of finding errors is a monotonically decreasing function of the number of errors found should be abandoned. We need to model the debugging process which gives an insight into exactly how the debugging rate does vary. This will be our subject in the next section.
3.2 MODELS FOR SOFTWARE FAILURE

In an attempt to explain the behaviour of real data many explanations could be given for the s-shaped curve produced by plotting errors detected against time. Some of these are:

1. If the debugging process is not constant, i.e. there is a change in the environment such as less effort being spent at the beginning of the debugging process, this could lead to fewer errors being detected in the early stage of the process. That is why most researchers in this field assume that the quantities of the resources, e.g. failure-correction personnel, etc., are constant.

2. If the correction of errors carried out during the debugging process generates further errors, then imperfect debugging could lead to a cluster of errors.

Some researchers have tried to handle these problems; for example the IBM Poisson model ([18], pp.3-8) attempted to account for the insertion of errors during the correction process and this leads to the mathematical model:

\[ N_1 = b_1 N_0 - a Q_0 \]

where \( N_1 \) is the number of errors at risk on test occasion (i) and (a) is the probability of correcting errors in the system without reinserting additional errors and exposing others to discovery, \( b_1 \) is the fraction of the system which is under test, \( Q_0 \) is the number of errors detected in that portion of the system under test prior to the ith test occasion.

3. The structure of a piece of software may consist of several modules, and some of these modules which contain many bugs will only be exercised in a later stage of the debugging process.
4. Software may contain two kinds of bugs; the first being major bugs which take some time to be fixed, during which time the system is down and no other errors can be detected. The second are minor bugs which take an insignificant time to be fixed.

5. We may postulate a hierarchy of errors, such that "secondary" bugs cannot be detected until all the "primary" bugs have been detected and removed. In other words, the debugging process does not "see" all possible errors from the beginning, and thus the error detection rate is not monotonically decreasing.

Three software reliability models based on (4.) and (5.) mentioned earlier have been proposed for estimating and predicting the reliability yielded by the error behaviour in a software system.

Using these three models consists of two stages: first developing the statistical inference procedure, secondly supplying it with the required data to test its ability for prediction and estimation.
3.3 **MODEL 1 - FINITE DEBUGGING TIME**

Let us assume that a piece of software contains $M$ major bugs and $N$ minor bugs. The major bugs each have a probability $\lambda_M$ of being detected per unit time, and the minor bugs have a detection rate of $\lambda$ per unit time. A minor bug is corrected instantly - a major bug takes a finite time to be corrected, which is a negatively exponentially distributed random variable with mean $\beta$.

Let $m(t) = E[\text{number of major bugs found at time } t]$

$$m(t) = \frac{M(t-m\beta)}{\lambda_M+t-m\beta} \quad (3.3)$$

where $t^*=t-m\beta$ (see equation (3.2))

$$m = \frac{M(t-m\beta)}{\lambda_M+t-m\beta} \quad (3.4)$$

$$\Rightarrow \quad \lambda_Mm + mt - m^2\beta = Mt - Mm\beta$$

$$\Rightarrow \quad \beta^2 - (\lambda_M-t-M\beta)m + Mt = 0$$

Solving this quadratic equation we get

$$m(t) = \frac{\lambda_M + t + M\beta - \sqrt{(\lambda_M + t + M\beta)^2 - 4M\beta t}}{2\beta} \quad (3.5)$$

Expected total number of bugs detected will be:

$$n(t) = \frac{N(t-m\beta)}{\lambda+t-m\beta} + m \quad (3.6)$$
3.4 MODEL 2 - PRIORITY FOR MAJOR ERRORS

Again we assume that the system contains $M$ major errors and $N$ minor errors. This time, however, the situation is different. The major errors are fixed immediately on detection, but it is not possible to detect any of the minor errors until all the major errors have been detected and removed. This "two-stage" model is intended to be a simplification of the general observation that deeper errors in a system cannot be observed until all the major errors have been removed.

Let $\lambda_M$ be major bug occurrence rate = $\lambda_M$/unit time

$\lambda$ is minor bug occurrence rate = $\lambda$/unit time

$n_M$ is the total number of major bugs detected at time $t$

$n$ is the total number of minor bugs detected at time $t$

The rate of detecting major bugs is

$$\frac{dn_M}{dt} = \lambda_M (M-n_M) \quad (3.7)$$

$$\Rightarrow \int \frac{dn_M}{M-n_M} = \int \lambda_M dt$$

$$\Rightarrow -\log(M-n_M) = \lambda_M t + c$$

$$\Rightarrow M-n_M = k e^{-\lambda_M t}$$

Now when $t=0$, $n_M=0$, i.e. $k=M$

$$\Rightarrow n_M = M(1-e^{-\lambda_M t})$$

Also $P[\text{all major bugs found}]$

$$= (1-e^{-\lambda_M t})^N$$
3.5 MODEL 3 - COMBINATION

Where Model (1) assumes that there is a delay when a major bug is encountered and the system goes down until the major bug is fixed, and Model (2) gives priority to major bugs being found and fixed, Model (3) combines these two ideas by allowing major bugs to emerge before minor bugs and assuming that the system goes down for a certain time until the major bug is fixed.

This modification has been carried out on Model (2). So the parameters used are the same as those of Model (2); with the additional parameter $\beta$, the expected down time.
3.6 THE SIMULATION PROCESS

Discrete time simulations were carried out to study the performance characteristics of these three models and to derive the profile of errors detected against time. The software system (represented by these three models) is treated as a discrete event system. This choice was because the behaviour of the simulated system itself is both random and dynamic. The system is simulated in discrete time intervals, and the software data (bugs) are represented by pseudo-random numbers generated by a pseudo-random number generator. These random numbers are compared with figures representing the probabilities of finding major or minor bugs or of fixing a major bug. These probabilities have been calculated by reference to the parameters of each model mentioned earlier.

In the simulation process the occurrence rate of bugs is treated as a constant, unlike the assumption mentioned earlier where it is assumed to be as a random variable.
3.7 **MODEL (1) SIMULATION**

In order to derive the profile of errors detected against time, we assume that within a small time interval $\Delta t$, the following events may occur:

i. A major bug may be detected and the system then goes down. The probability of this is $m \lambda_M \Delta t$, where $m$ is the current number of undetected major bugs.

ii. A minor bug may be detected, and is immediately corrected. The probability of this is $n \lambda \Delta t$, where $n$ is the current number of undetected minor bugs.

iii. If the system is down, it may become active again, through the latest major bug being corrected. The probability of this is $\Delta t/\beta$.

Two hundred iterations of this simulation model were carried out to produce an average profile of number of errors detected against time. The values of the parameters used were:

- No. of major bugs $M = 4$
- No. of minor bugs $N = 40$
- Major bug occurrence rate $\lambda_M = 20$/unit time
- Minor bug occurrence rate $\lambda = 1$/unit time
- Average time to fix major bug $\beta = 1$ time unit.

Figure 3.4 illustrates the average number of errors detected as a function of time, and Figure 3.5 is a similar plot for a single realisation of the simulation model. In Figure 3.6 a plot of average error detection rate (averaged over 200 iterations, and smoothed by averaging over 17 adjacent time-steps) as a function of number of errors is given - the shape of this confirms that reliability is not constantly increasing for this model. Appendix A contains the complete program for Model (1) simulation.
3.8 **MODEL (2) SIMULATION**

Again we assume that the system contains M major errors and N minor errors. The major errors are fixed immediately on detection, but it is not possible to detect any of the minor errors until all the major errors have been detected and removed.

A two hundred iteration simulation of this model was carried out, with the following parameters:

- No. of major bugs $M = 4$
- No. of minor bugs $N = 40$
- Major bug occurrence rate $\lambda_M = 0.5/\text{unit time}$
- Minor bug occurrence rate $\lambda = 1.0/\text{unit time}$

Figure 3.7 illustrates the average number of errors against time, and Figure 3.8 is the profile for a single realisation. Figure 3.9 shows the average error detection rate as a function of the number of bugs, (averaging over 200 iterations).

3.9 **MODEL (3) SIMULATION**

It is obviously straightforward to combine the two previous models, so that major errors are detected before minor errors and also take a non-zero time to fix. Two hundred iterations of this simulation model were run, with the same parameters as Model (2), and the results are illustrated in Figures 3.10-3.12, as we did with the previous models.

Examination of the results of these simulation studies shows that it is possible to generate reasonable approximations to the behaviour of real software systems with fairly simple assumptions.
Examples of data sets from Angus et al report (1983)

1 - APS/APC/IT
2 - APS/ASZ/IT
3 - APS/DAZ/IT
4 - APS/ZBZ/SD
5 - SUS/CON/IT

FIGURE 3.3
Model One, 200 Iterations

Figure 3.4
Model One, Single Iteration

Figure 3.5
Model One, Failure Rates Vs. No. Of Bugs

Figure 3.6
Model Two, 200 Iterations

Figure 3.7
Model Two, Single Iteration

Figure 3.8
Model Two, Failure Rates Vs. No. Of Bugs

Figure 3.9
Model Three, 200 Iterations

Figure 3.10
Model Three, Single Iteration

Figure 3.11
Model Three, Failure Rates Vs. No. Of Bugs

Figure 3.12
CHAPTER FOUR

FITTING FUNCTIONAL MODELS
The models we have developed, although simple in their structure, are mathematically rather too complex to fit directly to software reliability data in any easy fashion. Ideally, we need a functional form of model which can be fitted to data and used to predict its future form, and yet reflects the behaviour of the models we have studied.

The above results (see Figures 3.6, 3.9, 3.12) show that the curves we obtained from plotting the failure rate against the number of errors detected are approximately parabolic, so these models may be fitted by a logistic curve (quadratic function).

The statistical package GENSTAT (A General-Purpose Statistical Package suitable for statistical manipulation, [39]) was used to fit polynomial models to these curves.

The values (X) are the average number of errors detected during the simulation runs and (Y) was the calculated failure rate (dn/dt).

Two kinds of polynomial were tried: the quadratic function,

\[ Y = ax^2 + bx + c, \]

where \(a < 0\) and \(c\) is a constant, and a cubic function,

\[ Y = ax^3 + bx^2 + cx + d. \]

The process of fitting these models involves finding the REGRESSION COEFFICIENTS for \(x, x^2, x^3\) and for the constant and also involves analysis of variance (to give an idea of how well the model fits the data). The amount of change in model sum of squares gained by moving from the quadratic to the cubic function is given by the program. This was found to be insignificant.

The observed and fitted values are listed together with the residuals (the difference between the observed and the fitted values of Y).
The observed and fitted values (for the quadratic and cubic function) are plotted on the same figure for comparison purposes in Figures (4.1-4.3).
4.2 PREDICTING FOR THE FUTURE

As mentioned earlier, inspection of Figures (3.6, 3.9, 3.12) shows that the rate of failure might be fitted quite reasonably by a quadratic function of the number of bugs, i.e.,

\[
\frac{dn}{dt} = a(n-b_2)(b_1-n) \quad \text{(see Figure 4.4)},
\]

(4.1)

where \(a>0, b_2<0\) are constants, \(b_1>0\) is the total number of bugs, \(n\) is the expected number of errors at time \(t\). (Compare with the J-M model, for which

\[
\frac{dn}{dt} = (N-n)\phi \quad \text{(see Figure 4.5)},
\]

from equation (4.1) we can write,

\[
\frac{dn}{(n-b_2)(b_1-n)} = adt
\]

\[
\Rightarrow \frac{dn}{n-b_2} + \frac{dn}{b_1-n} = a(b_1-b_2)dt
\]

\[
\Rightarrow \frac{n-b_2}{b_1-n} = k \exp[a(b_1-b_2)t]
\]

when \(t=0, n=0\), so

\[
k = \frac{-b_2}{b_1}
\]

\[
\therefore \quad n = \frac{-b_1b_2(1-\exp[-a(b_1-b_2)t])}{b_1\exp[-a(b_1-b_2)t]-b_2}
\]

(4.2)

This "logistic" curve has a suitable s-shaped form which will match our simulation model results and the profiles of real software failure data. As \(t\) tends to \(\infty\), the number of errors detected tends to \(b_1\), which is thus the total number of errors in the system (\(N\) in the J-M model) (see Figure 4.6). In fact, this model may be considered to be a natural extension of the J-M model, and contains the latter as a
special case. If $a$ approaches 0 and $-ab_2$ equals $\phi$, then our logistic model reduces to the J-M model. Thus all the examples which can be fitted by the J-M model can also be fitted by this one, as well as many other cases which are not covered by the J-M model.

The problem now arises of fitting this logistic model to software failure data so as to predict the remainder of the software failure profile. Experiments were carried out using the results of the simulation models (single iteration) as raw data for the model fitting. Three methods were used to estimate the models' parameters, Maximum likelihood, Regression, Least squares. These methods will be investigated in the following section.
4.3 MAXIMUM LIKELIHOOD ESTIMATION

If we are given the following information:

N total bugs found so far, and
t_n = time between finding the nth and (n+1)th bug, for n=0,...,N-1,
then after the nth error, the rate of occurrence of errors is

\[ \lambda_n = \frac{dn}{dt} = a(n-b_2)(b_1-n) \]

so the probability of time \( t_n \) passing before the next error is given by a negative exponential distribution.

\[ f(\lambda_n, t_n) = \lambda_n \exp(-\lambda_n t_n), \lambda_n > 0, \ t_n > 0 \]

(4.3)

\[ \text{Total likelihood} = \prod_{n=0}^{N-1} \lambda_n \exp(-\lambda_n t_n) \]

since the value of \( \lambda_n \) that maximises L will be the same as the value that maximizes log L, log L is calculated

\[ \log L = \sum_{n=0}^{N-1} (\ln \lambda_n - \lambda_n t_n) \]

\[ = \sum_{n=0}^{N-1} \ln \lambda_n - \sum_{n=0}^{N-1} \lambda_n t_n \]

\[ = \sum_{n=0}^{N-1} \ln(a(n-b_2)(b_1-n)) - a \sum_{n=0}^{N-1} (n-b_2)(b_1-n)t_n \]

(4.4)

4.3.1 Calculation of Maximum Likelihood Estimate

To find the maximum likelihood estimates for \( a, b_1, b_2 \) we need to maximise the log likelihood L with respect to the three variables.

Thus obtaining three formulae:

\[ \frac{dL}{da} = \sum_{n=0}^{N-1} \frac{1}{a(n-b_2)(n-b_2)} (b_1-n)(n-b_2) \]
\[
\frac{\partial L}{\partial b_1} = n\sum \left( \frac{1}{a(b_1-n)(n-b_2)} \right) (a_{n+1} - ab_2)
\]

\[
\frac{\partial L}{\partial b_2} = \sum (ab_{n+1} - a_n) n \frac{1}{a(b_1-n)(n-b_2)} (ab_{n+1} - a_n)
\]

The process of estimation was carried out, setting the above three equations to zero, using the NAG library routine E04DFF, (see [40]), and feeding this procedure with the data obtained \( t_i \) (the difference in the time of bug occurrences between the \( n \) and \( n+1 \)th bug) in the previous simulation process for each of the 3 sets of model data.

Once the values of \( a, b_1 \) and \( b_2 \) which maximise \( L \) had been determined, the process of predicting was carried out by evaluating the equation (4.2) for successive values of time \( (t) \), substituting \( b_1, b_2, a \) by their estimated values. The results (cumulative number of bugs) are plotted as a function of time, Figures (4.7, 4.8, 4.9).

The results, however, were disappointing, in as far as the match between the logistic curve produced and the error versus time profile was poor in each case. This is mainly because maximum likelihood methods are inherently poor at fitting models with exponential distribution of times.
4.4 REGRESSION ON THE FAILURE RATE CURVE

A second method of estimating the parameters is to use the regression coefficients obtained from fitting a quadratic curve to the failure rate from the simulation process. The parameters \( a, b_1, b_2 \) are calculated from these regression coefficients and these parameters are substituted in the formula 4.2. The plots produced show (Figures 4.10, 4.11, 4.12) that the regression method gives better results than the maximum likelihood method, as the curves approach more closely the simulation ones.

4.5 LEAST SUM OF SQUARES FITTING

It was decided to fit the logistic curve directly to the simulation data, using a least sum of squares criterion to give a "best fit".

If \( N \) bugs have been found, and,

\[ T_n = \text{Time at which the } n\text{th bug is found, } n=1, \ldots, N, \]

then at time \( T_n \) compute, for a given set of values of \( a, b_1 \) and \( b_2 \),

\[ \hat{n} = \text{expected number of errors at time } T_n \text{ from equation } (4.2). \]

From

\[ E = \sum_{n=1}^{N} (n-\hat{n})^2, \quad (4.5) \]

and minimise \( E \) with respect to \( a, b_1, b_2 \). Again, this was carried out using a NAG routine [40], in this case E04CGF. Better results were obtained this time, as would be expected. These are shown clearly in Figures 4.13, 4.14, 4.15.
4.6 FITTING THE LOGISTIC MODEL TO REAL SOFTWARE DATA

So far we have seen the results of experimenting with the data obtained from the simulation process. The true test of the simple logistic model is when it is fitted to real software data, and used to predict the future performance of software systems. The real data used was the five sets mentioned earlier, related to U.S. Defence software (see Figure 3.3). The sets 1, 2, 3, 5 are from the integration test (pp. 4-2, 4-4, 4-14, 4-21, [18]), and the fourth set (pp. 4-17, [18]) is from the independent test. It is clear from the above sets of data that we may not possess the times at which every error was detected, but only values of cumulative numbers detected at certain times.

The same procedures are followed as with the simulation data, i.e. obtaining failure rates for these sets of data and then fitting a logistic model to them (see Figures 4.16, 4.17, 4.18, 4.19, 4.20).

In the prediction process the maximum likelihood method has been omitted since it did not give good results for the simulation process. Thus it is likely to give the same poor results for the real data. To test a model of this type properly it is essential that it be fitted to a subset of the data, up to a certain time, and then the fitted model be used to predict the "future" profile. This can then be compared with the true profile, and the effectiveness of the model as a predictor be evaluated.

The process of prediction using the least sum of squares goes as follows:

If $n_i$ errors have been detected at time $T_i$, for $i=1, \ldots, m$, then we may form the error sum squares as before:
$$E = \sum_{i=1}^{m} (\hat{n}_i - n_i)^2$$

where \( \hat{n}_i \) is the expected number of errors obtained by substituting \( T_i \) into equation (4.2).

Minimisation of the sum of squares using E04CGF led to good results for real data (see Appendix B). In Figures 4.21-4.25, results are shown for the 5 sets of software failure data displayed in Figure 3.3. In each case, only part of the data has been fitted and the logistic model used to predict the "future" performance. Comparison of this with the actual "future" profile shows a reasonably good predictive ability. The regression method was also tried; three of the five sets (1,3,4) have given reasonable results (see Figures 4.26-4.28). For the other two, set 2 and 5, the results were not encouraging. Finally it has been found that there is a strong relationship between the amount of data used in the process of prediction and the amount of error generated in the estimated number of bugs. This relation (see Figure 4.29) shows that any increase in the amount of data used will increase the accuracy of that estimation. But demanding more data for the sake of accuracy should not affect the aim of minimizing the cost we are after, and a balance should be maintained between the two.
Model One, Failure Rates Fitting

Legend
△ simulated curve
X quadratic curve
□ cubic curve

Figure 4.1
Figure 4.2
Model Three, Failure Rates Fitting

Legend
- simulated curve
- quadratic curve
- cubic curve

Figure 4.3
Model One, Maximum Likelihood Method, Using All The Data

Figure 4.7

*All the data is used to estimate the values of $a, b_1, b_2$. 

Legend
- simulation
- estimation
Model Two: Maximum Likelihood Method, Using All The Data

Figure 4.8
Model Three, Maximum Likelihood Method, Using All The Data

Figure 4.9
Model One, Regression Method, Using All The Data

Figure 4.10
Model Two, Regression Method, Using All The Data

Figure 4.11
Model Three, Regression Method, Using All The Data

Figure 4.12
Model 1, Least Sum Of Squares Method, Using 90\% Of The Data

Figure 4.13
Model 2, Least Sum Of Squares Method, Using 90% Of The Data

Figure 4.14
Model 3, Least Sum Of Squares Method, Using 90% Of The Data

Figure 4.15

Legend

Δ simulation
× estimation
Set One, Failure Rates Fitting

Figure 4.16
Set Two, Failure Rates Fitting

Figure 4.17
Set Three, Failure Rates Fitting

Figure 4.18
Set Four, Failure Rates Fitting

Legend
- real data curve
- quadratic curve
- cubic curve

Figure 4.19
Set Five, Failure Rates Fitting

Legend
- real data curve
- quadratic curve
- cubic curve

Figure 4.20
Set One, Least Sum Of Squares Method, Using 76% Of The Data

Figure 4.21

Legend

Δ real data
Χ estimation
Set Two, Least Sum Of Squares Method, Using 42% Of The Data

Figure 4.22
Set Three, Least Sum Of Squares Method, Using 74% Of The Data.

Figure 4.23
Set Four, Least Sum Of Squares Method, Using 86% Of The Data

Figure 4.24
Set Five, Least Sum Of Squares Method, Using 55% Of The Data

Figure 4.25
Set One, Regression Method, Using All The Data

Figure 4.26
Set Three, Regression Method, Using All The Data

Figure 4.27
Set Four, Regression Method, Using All The Data

Figure 4.28
Percentage Of Accuracy Estimation Vs. Percentage Of Data Used

Figure 4.29
CHAPTER FIVE

CONCLUSION
The problem of software reliability is getting a great deal of attention. This is shown in the increased number of studies presented. Some of these have been discussed in some detail (Chapter Two). Different approaches to the problem were presented, starting from developing models to measure and predict current and future reliability and how to select the appropriate model, feeding it with accurate software failure data, and ending with a presentation of the main features of some well known models. In the following chapters we have attempted to modify the ideas on which some of the early models are based (e.g. J-M model), such as giving the same degree of importance to all the errors a program contains. This is not necessarily the case as we found in practice - plotting real data (representing number of errors detected) against time often gives an s-shaped curve due to differences in the importance of the errors the system contains.

This work has been motivated by a desire to develop a simple but informative model for the reliability of a piece of software. Three simulation models have been used to reflect the behaviour of real software, and to attempt to discover some of the underlying mechanisms which give rise to software failure profiles.

Plotting the simulation results (the number of bugs as a function of time) we get the 's-shaped' curves we were expecting to have. Also from plotting the averaged failure rate against the number of errors it has been found that these curves approximate a parabola. As a result of this these models may be fitted by a logistic curve. This simple logistic model has been shown to represent these results fairly well and to give reasonable predictions of software behaviour. To predict the future failure rates three methods are used:
1. Maximum likelihood estimate

2. Regression on failure rate data

3. Least squares

The least-squares methods are shown to be the best of the three.

The logistic model can be considered to be a "second order" model, by comparison with the J-M model, which can be thought of as a "first order" model and is a special case of the logistic model.

Comparison with the model proposed by Yamada, Ohba and Osaki [30] shows a number of similarities, in particular, the s-shaped nature of the error profile. They test their model with one set of data only, and also compare it with a logistic curve. However, their logistic curve has been poorly fitted to the data and it is not clear that their model is a significantly better fit in general.

What is indicated is that models of this type are needed to predict accurately the behaviour of real software systems. This work is concluded by showing that in the process of predictions, the more data we used the more accuracy we get.

5.1 Suggestions for Future Work

Most of the models developed (including our models) assume that errors are removed with certainty. This affects the ability of these models to measure and predict reliability. It would be very beneficial if these three models were able to handle the problem of imperfect debugging, by allowing for the probability that the number of bugs remain the same after fixing a bug.

A second area in which this work could be extended is handling the integration problem. Software consists of modules, and these
modules have to be integrated at some stage. This integration process would lead to more bugs appearing. We need a model which incorporates this fact into its structure. This probably could be done by designing a matrix to keep the errors which result from the transmitting process between the modules, and then treating these errors as a different level of error (major and minor) as we did before. These are just suggestions and these two problems could be approached from different angles.
REFERENCES


removal in computer-programs and hardware designs", IEEE
Transactions on Reliability, Vol. R-30, No. 4, October 1981,
pp. 313-320.

[26] P. Kubat and H.S. Koch, "Managing test-procedures to achieve

[27] P. Kubat and H.S. Koch, "Pragmatic testing protocols to measure
software reliability", IEEE Transactions on Reliability,

[28] A.K. Goel, K. Okumoto, "Bayesian software prediction models,
An imperfect debugging model for reliability and other
quantitative measures of software systems", RADC-TR-78-155,

[29] L.H. Crow and N.D. Singpurwalla, "An empirically developed
Fourier series model for describing software failures",

modelling for software error detection", IEEE Transactions

for software reliability and other performance measures",
IEEE Transactions on Reliability, Vol. R-28, 1979,
August, pp. 206-211.


APPENDIX A

THE SIMULATION PROGRAM
program simul(input, output);

(* This program illustrates the complete simulation *)
(* process for the first model, it needs to be supplied *)
(* with the required number of iterations and the value *)
(* of am where am is the major bug occurrence rate *)

const dt = 0.01; b = 1; an = 1;
type
  status = (up, down);
  entry = record
    countm, countn, count, diff : integer;
  end;
var
  mn, am : real;
  a : array [1..3000] of entry;
  c, m, n, i, k, kk, mx : integer;
  sys: status;

function random(var c: integer): real;
begin
  c := (25173 * c + 13849) mod 65536;
  random := c / 65536;
end;

procedure initialise;
var i : integer;
begin
  for i := 1 to 3000 do
    begin
      a[i].countm := 0;
      a[i].countn := 0;
      a[i].count := 0;
      a[i].diff := 0;
    end;
end;

procedure initial;
var z : integer;
begin
  for z := i + 1 to 3000 do
    begin
      a[z].countm := a[z].countm + 4;
      a[z].countn := a[z].countn + 40;
    end;
end;
procedure addtions;
var 1 : integer;
begin
  for 1 := 1 to mx do
  begin
    a[l].count := a[l].countm + a[l].countn;
    if 1 >= 2 then a[l].diff:=a[l].count-a[l-1].count;
  end;
end;

procedure writeres;
var 1, f, j, nav, av:integer;
begin
  1 := 1; f:=0; nav:=8;
  write('time','maj. bugs','min. bugs');
  write('bugs/kk','failure rate');
  writeln;
  write('____________________________');
  writeln;
  while 1 <= mx do
  begin
    av := 0;
    write(1:5,a[l].countm, a[l].countn);
    write(' ',a[l].count/kk:7:4);
    if 1 >=2 then
    begin
      write(' ',a[l].diff/(kk*dt):7:4);
    end;
    writeln;
    l := 1 + 49 + f;
    f := f;
  end;
end;

begin
 initialise;
k := 0; c := 500; mx := 0;
 writeln;
 write('enter the no. of iterations');
 writeln;
 readin(kk);writeln;
 write('enter the value of am, 0.025-0.5');
 writeln;
 readin(am);
 write('First model with number of iterations',kk:4,')
and am=',am:6:4);
writeln;writeln;
repeat
  m :=4; n :=40; i :=0;
sys := up;

repeat
  i := i+1;
  mn := random(c);
  if sys = up then
    (*test for major bug*)
    begin
      if mn <= m*dt/am then
        begin
          if m >= 1 then
            begin
              m := m-1;
              sys := down;
            end;
        end;
    end
  else
    (*test for minor bug*)
    begin
      if mn <= n*dt/an then
        begin
          n:=n-1;
        end;
    end;
  end
else
  begin
    if mn <= dt/b then
      (*enable the system*)
      sys := up;
    end;
end;

a[i].countm := a[i].countm + 4 - m;
a[i].countn := a[i].countn + 40 - n;
  until n = 0;
inial;
if i >= mx then mx := i;
k := k+1;
until k = kk ;
addtions;
writeres;
end.
simull

enter the no. of iterations \((KK)\)
7200

enter the value of \(am\), 0.025-0.5
?0.05

First model with number of iterations 200 and \(am=0.0500\)

<table>
<thead>
<tr>
<th>time</th>
<th>maj. bugs</th>
<th>min. bugs</th>
<th>bugs/kk</th>
<th>failure rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>156</td>
<td>0</td>
<td>0.7800</td>
<td>1.0000</td>
</tr>
<tr>
<td>50</td>
<td>286</td>
<td>4</td>
<td>1.4500</td>
<td>1.0000</td>
</tr>
<tr>
<td>100</td>
<td>386</td>
<td>6</td>
<td>1.9600</td>
<td>1.0000</td>
</tr>
<tr>
<td>150</td>
<td>477</td>
<td>77</td>
<td>2.7700</td>
<td>2.5000</td>
</tr>
<tr>
<td>200</td>
<td>559</td>
<td>288</td>
<td>4.2350</td>
<td>2.0000</td>
</tr>
<tr>
<td>250</td>
<td>601</td>
<td>670</td>
<td>6.3550</td>
<td>9.0000</td>
</tr>
<tr>
<td>300</td>
<td>660</td>
<td>1418</td>
<td>10.3900</td>
<td>7.5000</td>
</tr>
<tr>
<td>350</td>
<td>704</td>
<td>2230</td>
<td>14.6700</td>
<td>3.5000</td>
</tr>
<tr>
<td>400</td>
<td>743</td>
<td>3147</td>
<td>19.4500</td>
<td>11.5000</td>
</tr>
<tr>
<td>450</td>
<td>757</td>
<td>4003</td>
<td>23.8000</td>
<td>15.0000</td>
</tr>
<tr>
<td>500</td>
<td>767</td>
<td>4740</td>
<td>27.5350</td>
<td>7.0000</td>
</tr>
<tr>
<td>550</td>
<td>777</td>
<td>5336</td>
<td>30.5650</td>
<td>8.0000</td>
</tr>
<tr>
<td>600</td>
<td>783</td>
<td>5926</td>
<td>33.5450</td>
<td>4.5000</td>
</tr>
<tr>
<td>650</td>
<td>796</td>
<td>6404</td>
<td>36.0000</td>
<td>6.5000</td>
</tr>
<tr>
<td>700</td>
<td>798</td>
<td>6799</td>
<td>37.9850</td>
<td>5.0000</td>
</tr>
<tr>
<td>750</td>
<td>798</td>
<td>7122</td>
<td>39.6000</td>
<td>1.0000</td>
</tr>
<tr>
<td>800</td>
<td>798</td>
<td>7329</td>
<td>40.6350</td>
<td>1.5000</td>
</tr>
<tr>
<td>850</td>
<td>800</td>
<td>7569</td>
<td>41.8450</td>
<td>3.0000</td>
</tr>
<tr>
<td>900</td>
<td>800</td>
<td>7704</td>
<td>42.5200</td>
<td>0.0000</td>
</tr>
<tr>
<td>950</td>
<td>800</td>
<td>7789</td>
<td>42.9450</td>
<td>0.0000</td>
</tr>
<tr>
<td>1000</td>
<td>800</td>
<td>7844</td>
<td>43.2200</td>
<td>0.0000</td>
</tr>
<tr>
<td>1050</td>
<td>800</td>
<td>7872</td>
<td>43.3600</td>
<td>0.0000</td>
</tr>
<tr>
<td>1100</td>
<td>800</td>
<td>7893</td>
<td>43.4650</td>
<td>0.0000</td>
</tr>
<tr>
<td>1150</td>
<td>800</td>
<td>7908</td>
<td>43.5400</td>
<td>0.0000</td>
</tr>
<tr>
<td>1200</td>
<td>800</td>
<td>7915</td>
<td>43.5750</td>
<td>0.0000</td>
</tr>
<tr>
<td>1250</td>
<td>800</td>
<td>7920</td>
<td>43.6000</td>
<td>0.0000</td>
</tr>
<tr>
<td>1300</td>
<td>800</td>
<td>7942</td>
<td>43.7100</td>
<td>0.0000</td>
</tr>
<tr>
<td>1350</td>
<td>800</td>
<td>7962</td>
<td>43.8100</td>
<td>0.0000</td>
</tr>
<tr>
<td>1400</td>
<td>800</td>
<td>7984</td>
<td>43.9200</td>
<td>1.0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>1450</td>
<td>800</td>
<td>7994</td>
<td>43.9700</td>
<td>0.0000</td>
</tr>
<tr>
<td>1500</td>
<td>800</td>
<td>7996</td>
<td>43.9800</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
In this appendix we would like to give a brief explanation how this program works.

The data supplied should be in the form of a file with two columns. The first represents the cumulative time and the second represents the corresponding cumulative number of errors detected.

The initial values of the parameters \( \alpha, b_2, b_1 \), are estimated from the data itself according to the following equations:

\[
\begin{align*}
-ab_1b_2 &= S_0 \\
\frac{1}{4}(b_1+b_2) &= n_{\text{max}} \\
\frac{\alpha}{4}(b_1-b_2)^2 &= S_{\text{max}}
\end{align*}
\]

where \( S_0 \) is the rate of error occurrence at time 0, \( S_{\text{max}} \) is the maximum rate of error occurrence and \( n_{\text{max}} \) is the number of errors at \( S_{\text{max}} \).

These initial values will form the starting points for the minimisation of the sum of squares of errors, which leads to the best values of \( \alpha, b_2, b_1 \) being estimated. Substituting these estimated values in equation (4.2) we get the predicted number of errors at any given time.
program prog3

This program implements the least squares method to predict the no. of bugs. The example below shows the prediction process for real data, clearly it needs to be supplied with some information (e.g., the number of accumulative time between failures, the file contains the data... etc.

implicit double precision(a-h,o-z)
common fn
common nmax
common nz1

double precision f,x(3), w(45)
double precision fn(0:100,0:1)
integer iw(5)
character *6 nv2
write (6,99993)
read (*,* )nz1
nmax =nz1
n = 3
dif = 0
write (6,99992)
read (*,"(a)" )nv2
open(5,file = nv2,form = "formatted",status = "old")
do 7 j = 0,nmax

read(5,* ) (fn(j,i), i = 0,1)
close(5)
write(6,99999)

c estimate the initial values of a,b1,b2
c from the data available
so = fn(0,1)/fn(0,0)
do 11 j = 0,nmax-1
dtm = fn(j+1,0) - fn(j,0)
dtm = max(dtm,1.0e-6)

dd = fn(j+1,1)-fn(j,1)/dtm

dr = max(dif,dd)
if (dd .lt. dif) jj = j
endif

tnmx = (fn(jj,1) + fn(jj+1,1))/2
smax = dif
if (so .ge. smax) so = 0.5*smax

bb2 = tnmx*(sqrt(smax- sc)-sqrt(smax))/sqrt(smax-so)

bb1 = 2*tnmx - bb2

bb1 = max(1.1*fn(nmax,1),bb1)
al = -so / (bb1 * bb2)
x(1) =al
x(2) =bb2
x(3) =bb1
iw = 5
lw = 45
ifail = 1
call e04cgf(n, x, f, lw, liw, w, lw, ifail)
if (ifail .ne.0) write(6,99998)ifail
if (ifail .eq. 1) go to 20
write(6,99991)
write(6,2) a1,bb2,bb1
write(6,99997)f
write(6,99996)
write(6,1)(x(i),i=1,n)

write(6,99995)
read(*,*)nv3
write(6,3)
read(*,*)nv4
write(6,40)
estimating the no. of errors using the estimated values of a,b1,b2
do 8 k = 1,nv3,nv4
    a = exp((-x(1))*(x(3)-x(2)))*k
    exn = -x(3)*x(2)*(-a)/(x(3)*a-x(2))
8 write(6,99994)k,exn
stop
99999 format(///23h eo4cgf program results/)
99998 format(///16h error exit type,i3)
99997 format(///23h function value on exit, f12.4)
99996 format(///" the estimated values of a,b2,b1 are:")
   1 format(3e12.4)
99994 format(i4,8x,f12.4)
99993 format(///" enter the no. of data available")
99992 format(///" enter the name of data file")
99995 format(///" enter the total time you want to predict for")
   3 format(///" enter the time interval you want to predict for")
40 format(///" acc. time",4x,"predicted no. of errors ")
99991 format(/// the initial values of a,b2,b1 are:")
   2 format(f12.4,2x,f12.4,2x,f12.4)
end

subroutine funct1(n,xc,fc)

Called from e04cgf (Nag routine)
Function evaluation routine

implicit double precision(a-h,o-z)
common fn
common nmax
common nz1
double precision x(n),fc
double precision alpha, b2, b1
double precision fn(0:100,0:1)
nmax = nz
alpha = xc(1)
b2 = xc(2)
b1 = xc(3)
f = 0.0
do 5 j = 0,nmax
t = fn(j,0)
a = ((-alpha*(b1-b2))*t)
a = min(80,a)
c a = max(a,1.0d-12)
a = exp(a)
 censor estimate the no. of errors using the
c initial values of a,b1,b2
rn = -b1*b2*(1-a)/(b1*a-b2)
c using least squares principle
fc = fc + (fn(j,1) - rn)**2
5 continue
c write(6,99999) b1,b2,alpha,fc
99999 format(1h ,2e12.4,2x,e12.4,2x,e12.4)
return
c end of function evaluation routine
end

prog3

enter the no. of data available
10

enter the name of data file
rlsl

eo4cgf program results

the initial values of a,b2,b1 are:
0.0730  -0.1181  33.1181
function value on exit 38.9057

the estimated values of $a,b_2,b_1$ are:
$0.3667E-02 -0.5001E-01 0.3688E+02$

enter the total time you want to predict for 107

enter the time interval you want to predict for 1

<table>
<thead>
<tr>
<th>acc. time</th>
<th>predicted no. of errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0072</td>
</tr>
<tr>
<td>2</td>
<td>0.0155</td>
</tr>
<tr>
<td>3</td>
<td>0.0250</td>
</tr>
<tr>
<td>4</td>
<td>0.0359</td>
</tr>
<tr>
<td>5</td>
<td>0.0483</td>
</tr>
<tr>
<td>6</td>
<td>0.0625</td>
</tr>
<tr>
<td>7</td>
<td>0.0788</td>
</tr>
<tr>
<td>8</td>
<td>0.0974</td>
</tr>
<tr>
<td>9</td>
<td>0.1187</td>
</tr>
<tr>
<td>10</td>
<td>0.1430</td>
</tr>
<tr>
<td>11</td>
<td>0.1708</td>
</tr>
<tr>
<td>12</td>
<td>0.2026</td>
</tr>
<tr>
<td>13</td>
<td>0.2390</td>
</tr>
<tr>
<td>14</td>
<td>0.2805</td>
</tr>
<tr>
<td>15</td>
<td>0.3280</td>
</tr>
<tr>
<td>16</td>
<td>0.3822</td>
</tr>
<tr>
<td>17</td>
<td>0.4440</td>
</tr>
<tr>
<td>18</td>
<td>0.5146</td>
</tr>
<tr>
<td>19</td>
<td>0.5950</td>
</tr>
<tr>
<td>20</td>
<td>0.6867</td>
</tr>
<tr>
<td>21</td>
<td>0.7911</td>
</tr>
<tr>
<td>22</td>
<td>0.9100</td>
</tr>
<tr>
<td>23</td>
<td>1.0451</td>
</tr>
<tr>
<td>24</td>
<td>1.1985</td>
</tr>
<tr>
<td>25</td>
<td>1.3728</td>
</tr>
<tr>
<td>26</td>
<td>1.5699</td>
</tr>
<tr>
<td>27</td>
<td>1.7932</td>
</tr>
<tr>
<td>28</td>
<td>2.0453</td>
</tr>
<tr>
<td>29</td>
<td>2.3296</td>
</tr>
<tr>
<td>30</td>
<td>2.6495</td>
</tr>
<tr>
<td>31</td>
<td>3.0086</td>
</tr>
<tr>
<td>32</td>
<td>3.4107</td>
</tr>
<tr>
<td>33</td>
<td>3.8595</td>
</tr>
<tr>
<td>34</td>
<td>4.3588</td>
</tr>
<tr>
<td>35</td>
<td>4.9124</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>36</td>
<td>5.5235</td>
</tr>
<tr>
<td>37</td>
<td>6.1951</td>
</tr>
<tr>
<td>38</td>
<td>6.9297</td>
</tr>
<tr>
<td>39</td>
<td>7.7288</td>
</tr>
<tr>
<td>40</td>
<td>8.5930</td>
</tr>
<tr>
<td>41</td>
<td>9.5217</td>
</tr>
<tr>
<td>42</td>
<td>10.5129</td>
</tr>
<tr>
<td>43</td>
<td>11.5632</td>
</tr>
<tr>
<td>44</td>
<td>12.6676</td>
</tr>
<tr>
<td>45</td>
<td>13.8194</td>
</tr>
<tr>
<td>46</td>
<td>15.0107</td>
</tr>
<tr>
<td>47</td>
<td>16.2321</td>
</tr>
<tr>
<td>48</td>
<td>17.4731</td>
</tr>
<tr>
<td>49</td>
<td>18.7228</td>
</tr>
<tr>
<td>50</td>
<td>19.9697</td>
</tr>
<tr>
<td>51</td>
<td>21.2024</td>
</tr>
<tr>
<td>52</td>
<td>22.4103</td>
</tr>
<tr>
<td>53</td>
<td>23.5833</td>
</tr>
<tr>
<td>54</td>
<td>24.7128</td>
</tr>
<tr>
<td>55</td>
<td>25.7914</td>
</tr>
<tr>
<td>56</td>
<td>26.8133</td>
</tr>
<tr>
<td>57</td>
<td>27.7742</td>
</tr>
<tr>
<td>58</td>
<td>28.6714</td>
</tr>
<tr>
<td>59</td>
<td>29.5037</td>
</tr>
<tr>
<td>60</td>
<td>30.2711</td>
</tr>
<tr>
<td>61</td>
<td>30.9746</td>
</tr>
<tr>
<td>62</td>
<td>31.6162</td>
</tr>
<tr>
<td>63</td>
<td>32.1987</td>
</tr>
<tr>
<td>64</td>
<td>32.7252</td>
</tr>
<tr>
<td>65</td>
<td>33.1993</td>
</tr>
<tr>
<td>66</td>
<td>33.6248</td>
</tr>
<tr>
<td>67</td>
<td>34.0053</td>
</tr>
<tr>
<td>68</td>
<td>34.3447</td>
</tr>
<tr>
<td>69</td>
<td>34.6467</td>
</tr>
<tr>
<td>70</td>
<td>34.9149</td>
</tr>
<tr>
<td>71</td>
<td>35.1525</td>
</tr>
<tr>
<td>72</td>
<td>35.3626</td>
</tr>
<tr>
<td>73</td>
<td>35.5482</td>
</tr>
<tr>
<td>74</td>
<td>35.7119</td>
</tr>
<tr>
<td>75</td>
<td>35.8561</td>
</tr>
<tr>
<td>76</td>
<td>35.9830</td>
</tr>
<tr>
<td>77</td>
<td>36.0945</td>
</tr>
<tr>
<td>78</td>
<td>36.1925</td>
</tr>
<tr>
<td>79</td>
<td>36.2785</td>
</tr>
<tr>
<td>80</td>
<td>36.3540</td>
</tr>
<tr>
<td>81</td>
<td>36.4201</td>
</tr>
<tr>
<td>82</td>
<td>36.4781</td>
</tr>
<tr>
<td>83</td>
<td>36.5289</td>
</tr>
<tr>
<td>84</td>
<td>36.5733</td>
</tr>
<tr>
<td>85</td>
<td>36.6122</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
</tr>
<tr>
<td>86</td>
<td>36.6463</td>
</tr>
<tr>
<td>87</td>
<td>36.6761</td>
</tr>
<tr>
<td>88</td>
<td>36.7021</td>
</tr>
<tr>
<td>89</td>
<td>36.7249</td>
</tr>
<tr>
<td>90</td>
<td>36.7449</td>
</tr>
<tr>
<td>91</td>
<td>36.7623</td>
</tr>
<tr>
<td>92</td>
<td>36.7775</td>
</tr>
<tr>
<td>93</td>
<td>36.7908</td>
</tr>
<tr>
<td>94</td>
<td>36.8024</td>
</tr>
<tr>
<td>95</td>
<td>36.8126</td>
</tr>
<tr>
<td>96</td>
<td>36.8215</td>
</tr>
<tr>
<td>97</td>
<td>36.8292</td>
</tr>
<tr>
<td>98</td>
<td>36.8360</td>
</tr>
<tr>
<td>99</td>
<td>36.8419</td>
</tr>
<tr>
<td>100</td>
<td>36.8471</td>
</tr>
<tr>
<td>101</td>
<td>36.8516</td>
</tr>
<tr>
<td>102</td>
<td>36.8556</td>
</tr>
<tr>
<td>103</td>
<td>36.8590</td>
</tr>
<tr>
<td>104</td>
<td>36.8620</td>
</tr>
<tr>
<td>105</td>
<td>36.8647</td>
</tr>
<tr>
<td>106</td>
<td>36.8670</td>
</tr>
<tr>
<td>107</td>
<td>36.8690</td>
</tr>
</tbody>
</table>