To the Graduate Council:

I am submitting herewith a thesis written by Bonnie Collett Sexton entitled "Statistical Testing of Software." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Dr. J. H. Poore, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Provost
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Bonnie Sexton

Date Nov 29, 1988

STATISTICAL TESTING OF SOFTWARE

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Bonnie Collett Sexton

December 1988

For Nick

Thank you for being so patient and waiting for me.

ACKNOWLEDGMENTS

Dr. Jesse Poore, my thesis adviser, remains for me, as for many others, a respected and loved mentor and friend. I owe special thanks to him who not only shared in the process of the work, but also read the thesis in various growth stages and helped me with his constructive criticism.

I am especially grateful to those who read the entire thesis and helped me with corrections, criticism and encouragement: Dr. Ronald Leinius, Dr. Bruce Char and Dr. David Mutchler. Many thanks to all readers for their service on the thesis committee and for their helpful comments.

I also wish to thank Dr. Bill McClain and Dr. David Straight for the excellent classroom instruction I have received during the past four years.

My most personal thanks I owe to my husband, Nick Sexton, not only for reading the thesis and performing miraculous editing, but for his constant presence and encouragement. I am grateful to all those who encouraged me to continue: parents, sister, grandmother, and friends.

ABSTRACT

Random testing or statistical testing of software has long been regarded as inferior to testing techniques that take advantage of knowledge of the software or the process of its production. However, research continues to show that statistical testing often yields better results than other forms of testing. Furthermore, as software takes on ever larger roles in medical instruments, avionics, control systems and life critical systems, the social need for certification of performance becomes important. Statistical testing allows one to make scientific statements about software. This research presents a step-by-step explanation of all the things necessary to conduct statistical tests of software.

TABLE OF CONTENTS

CHAPTER		PAGE
1	**INTRODUCTION**	**1**
1.1	Why Test Software?	1
1.2	Overview of Software Testing	3
1.2.1	White-box Testing	4
1.2.2	Black-box Testing	5
1.2.3	Incremental Testing	6
1.3	Overview of Test Data Generators	7
1.3.1	Pathwise Generators	7
1.3.2	Data Specification Test Generators	10
1.3.3	Random Test Generators	11
1.4	What is Statistical Testing?	11
1.5	Advantages of Statistical Testing	12
1.6	Purpose: How to Statistically Test Software	15
2	**GENERAL MODEL FOR STATISTICAL TESTING**	**16**
2.1	Diagram and Description	16
2.2	An Algebraic Description of Programs, Faults, and Failures	20
2.2.1	Paths	23
2.2.2	Failures	24

2.2.3 Errors and Faults 25

2.3 A Few Remarks About the Nelson Model 28

3 GENERATING TEST CASES 32

3.1 Description of the Input Domain 32

3.2 SIAD Tree Representation 37

3.3 Backus Normal Form and Dynamic Grammar 39

3.4 Summary 40

4 CHOOSING AN APPROPRIATE USER DISTRIBUTION 42

4.1 Distribution Descriptions 42

4.1.1 Continuous Distributions 43

4.1.2 Discrete Distributions 44

4.2 User-supplied Distributions 44

5 SAMPLING 49

5.1 Simple Random Sampling 49

5.2 Stratified Sampling 50

5.3 Sampling in the Testing of Software 52

5.4 Choosing the Sample Size 53

5.4.1 The Binomial Distribution 53

5.4.2 Pascal Sampling 56

5.5 Rare Events and Cleanroom Methodology 57

6 THE ORACLE: OBSERVING AND CALCULATING TEST RESULTS 63

6.1 Failures and Errors 65

6.2 Algorithm for Testing 66

6.3 Hypothesis Testing 68

6.4 Summary 70

7 WARRANTY STATEMENTS 71

7.1 What a Software Warranty Should Say 72

8 EXAMPLES 74

8.1 STATGRAPHICS SIN 74

8.1.1 Sample Size 75

8.1.2 Oracle 75

8.1.3 Warranties 75

8.2 Job Submission Program 76

8.2.1 Operational Reliability 77

8.2.2 Testing of Two FORTRAN Subroutines 77

8.3 Parallel Axis Graph 78

8.3.1 Generation 79

8.3.2 Distributions 79

8.3.3 Sample Size and Lower Bound 81

8.3.4 Algorithm for Testing 81

8.3.5 Oracle 81

8.3.6 Test Results 82

BIBLIOGRAPHY **83**

APPENDIX **87**

CONTINUOUS DISTRIBUTION PLOTS **88**

VITA **95**

LIST OF TABLES

TABLE		PAGE
1-1	Types of Errors.	3
1-2	Fitted Percentage Defects.	13
2-1	Notation for Statistical Model.	17
5-1	The Lower Bound.	55
5-2	Sample Size N	56
5-3	Notation for Rare Events.	59
5-4	Some Calculations Using $\frac{1}{\epsilon}\ln(\frac{1}{1-\gamma})$.	62

LIST OF FIGURES

FIGURE		PAGE
2-1	A Model for Statistical Testing of Software.	18
2-2	Program F.	21
2-3	U_F: The Usage Distribution of F.	22
2-4	The Testing Process.	23
2-5	Partitions on the Input Space.	25
3-1	Point Expansion.	34
3-2	Input Rule Definition.	35
3-3	Graph of Simple Input Domain.	36
3-4	Definition of Simple Input Domain.	36
4-1	Continuous Probability Density Functions.	45
4-2	Discrete Probability Density Functions.	46
4-3	User Supplied Discrete Distributions Using NAG.	48
7-1	A Software Warranty.	73
8-1	Input Definition for Increment One.	80
A-1	Normal Distributions with the Same σ.	89
A-2	Normal Distributions with the Same μ.	89
A-3	Log Normal Distribution: All Curves Have the Same μ.	90
A-4	Log Normal Distribution: All Curves Have the Same σ^2 Values.	90

A-5 Beta Distribution: When $\gamma > 1$ and $\eta > 1$ the Distribution Has a Single Peak. 91

A-6 Beta Distribution: When $\gamma < 1$ and $\eta < 1$ the Distribution is U Shaped. 91

A-7 Beta Distribution: When $\gamma < 1$ and $\eta \geq 1$ the Distribution is Reverse J Shaped; When $\eta < 1$ and $\gamma \geq 1$, it is J Shaped. 92

A-8 Beta Distribution: When $\gamma = \eta$ the Distribution is Symmetrical. 92

A-9 Exponential Distribution: A Special Case of the Gamma Distribution. 93

A-10 Uniform Distribution: A Special Case of the Beta Distribution. . 93

A-11 Gamma Distribution: Same η Values. 94

A-12 Gamma Distribution: Same λ Values. 94

CHAPTER 1

INTRODUCTION

An intuitive way to test any program is to consider the program a black box (i.e., to consider the specifications for what goes into the program with no regard for the structure inside the program), generate as many input combinations as possible, feed those inputs into the program, then check the accuracy of all outputs. Unfortunately, this method might prove expensive, if not impossible, because even the simplest programs may have astronomical input combinations that cannot all be tested. Intuitive ideas have been supplanted by a number of research and development efforts to create effective methods for testing software. This chapter is a review of key ideas from the literature and closes with a statement of the objective of this paper.

1.1 Why Test Software?

Software testing can be defined as the execution of a software system with data and a comparison of the results with the expected output for that data. There are many strategies for testing software and much disagreement as to which method is the best (i.e., which finds the most errors, is the least expensive, and gives a quantitative measure of reliability). Software testing is often

criticized as only revealing the presence of software errors but never guaranteeing that the software is error-free. Its unpopularity with practitioners stems less from the above reason than from two other reasons: (1) it draws attention to the fallibility of the programmer, and (2) it is an extremely time-consuming and tedious activity.[1]

The primary concern of all software developers should be to meet the specification with high quality software. Improved software quality leads to improved productivity. Instead of spending time on reworking old code, programming teams can develop new software and improve their skills. Some authors estimate that as much as 50% of the average commercial programmer's time is spent on debugging, a situation that also puts a burden on computer time.[2]

Table 1-1 shows estimates of the relative incidence of different types of errors. In the table, the software category is considering programs that are purchased, not created on site. Programming is done by the staff at the site; this is the category that produces the most system failures.[3]

In the first step to reliable software testing, a strong development methodology should be used for all software projects, then a testing plan should be administered to ensure that the software performs correctly.

---

[1] D. C. Ince, "The Automatic Generation of Test Data," *Computer Journal*, **30**, February 1987, p. 63.

[2] A. R. Brown and W. A. Sampson, *Program Debugging* (London: Balding & Mansell Ltd, 1973), pp. 2–3.

[3] Ibid.

Table 1-1: Types of Errors.

Percentage of System Failures Due to:

Type	Percentage
Hardware	1%
Software	2%
Operating	5%
Programming	90%
Systems	2%

Source: A. R. Brown and W. A. Sampson, *Program Debugging* (London: Balding & Mansell Ltd, 1973), p. 2.

1.2 Overview of Software Testing

Test cases may be constructed from specifications (black-box testing) or by analyzing the program structure (white-box testing). Tests may also be designed to examine the entire program or, as in module testing, the individual, logical units of a program (e.g., procedures or subprograms).

There are two approaches to combining module analysis: nonincremental and incremental. Three strategies for incremental testing are top-down, bottom-up, and thread testing.[4]

[4] Richard A. DeMillo, W. Michael McCracken, R. J. Martin, and John F. Passafiume, *Software Testing and Evaluation* (Menlo Park, California: The Benjamin/Cummings Publishing Company, Inc., 1987), pp. 19–25.

1.2.1 White-box Testing

White-box testing is based on the structure of the program, and the test cases are derived from the program's logic or picked from partitions of the input space of a program based on path domains. There are many test-case selection strategies that target known or suspected weaknesses in the construction of a program. Some that occur frequently in the literature are path testing and branch testing.

Path testing involves the testing of every path through a program at least once. Since most programs have large numbers of paths, this method is of theoretical rather than practical importance. Branch (or decision) testing requires that each "branch" in a program be tested at least once; it is the simplest approach to systematic testing.[5] "The problems associated with this criterion are that a program may contain no decision statements, a program may contain multiple entry points, some statements may only be executed if the program is entered at a particular entry point, and if a program contains exception handling routines, these routines may not be executed at all."[6]

Symbolic testing gives symbolic values to input data and program variable values. Symbolic execution of a program can be characterized by an execution tree. The tree consists of the nodes associated with the executed statements and

---

[5] William E. Howden, "An Evaluation of the Effectiveness of Symbolic Testing," *Software—Practice and Experience*, **8** (1978), p. 383.

[6] DeMillo et al., p. 21.

the directed arcs indicating program flow. This method becomes very complex with large programs, and for programs that contain loops or array variables the termination criteria are especially difficult.

1.2.2 Black-box Testing

The objective of black-box testing (or functional testing) is to examine the input-output behavior of the program (or module) and determine if this agrees with the specifications. Black-box testing may also measure the program's ability to adapt to the user environment.

Equivalence partitioning minimizes the number of test cases. The input space of a program is partitioned into equivalence classes with respect to the program's input specifications. Identification of these classes usually requires a heuristic approach. A set of heuristics is given by Myers. [7]

Boundary value testing explores those conditions where the input ranges are exercised. Rather than selecting any element in an equivalence class, one selects elements where each edge of the class is the subject of a test. One should also focus attention on the output space as the basis for test cases.[8]

Random testing selects a sample of inputs out of the total possible inputs. Selecting a sample based on an operating environment will give an operational reliability value. The advantages of random testing are (1) it only requires a ran-

---

[7] Glenford J. Myers, *The Art of Software Testing*, Wiley Series in Business Data Processing (New York: John Wiley and Sons, 1979), pp. 46–48.

[8] Ibid., pp. 50–51.

dom number generator and a small amount of support software and is therefore inexpensive, (2) it is straightforward to derive estimates of the operational reliability, and (3) the use of random numbers stresses software more than manual test cases. Some disadvantages are: (1) there is no assurance that full coverage can be attained, (2) it can be expensive in terms of human resources if thousands of test cases are required,[9] and (3) believable probability distributions may be expensive to determine. Much controversy over random testing still exists, and some recent results from this type of research in testing will be discussed later.

1.2.3 Incremental Testing

Top-down testing starts with the top module in a program and then tests the modules at lower levels progressively. Bottom-up testing starts with the modules at the bottom level (where the modules do not call other modules) and works upward. Thread testing is based on the system verification diagram (SVD) and is well suited to real-time system testing. According to Deutsch, "the SVD segments the system into demonstrable functions called threads. In this approach, software test and construction are intertwined; they do not occur separately and sequentially. The order in which the software is coded, tested, and synthesized is essentially determined by the SVD that defines the test procedure."[10]

---

[9] Ince, p. 63.

[10] Michael S. Deutsch, *Software Verification and Validation: Realistic Project Approaches*, (Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1982), pp. 19–20.

1.3 Overview of Test Data Generators

The literature contains some interesting models for synthetically generating the input domain. A test data generator is a tool that assists a user in generating test data. Three types of test data generators are (1) pathwise test data generators, (2) data specification systems, and (3) random test data generators. Pathwise test data generators have difficulty with array references and waste considerable time computing infeasible paths. Data specification systems provide a special language to specify input data files. Random test generators pick values from the input domain.[11]

1.3.1 Pathwise Generators

Huang asserted that "an obvious way to increase the probability of discovering errors through program testing is to have each and every statement in the program executed at least once during the test."[12] This is a pathwise generator as described above. Huang went on to develop methods for constructing minimal covering sets that traverse every branch in the program flow chart at least once. He then constructed a corresponding set of tests to accomplish this. After a lengthy example of how one input domain combination can be obtained from a specific path, Huang concluded that test-case generation based on the paths

---

[11] DeMillo et al., p. 107.

[12] J. C. Huang, "An Approach to Program Testing," *Computing Surveys*, September 1975, pp. 114–115.

in a program is generally too difficult in practice, and that no single algorithm exists that finds assignments to input variables that will satisfy any given path.

Ramamoorthy and his colleagues described an automated test data generator that also used the structure of a program to define the input domain.[13] (The program structure is used for partitioning the input space.) An actual input data generator for FORTRAN programs was implemented by Ramamoorthy's research team using symbolic execution to generate input constraints that fit certain paths. The problem explored in this procedure was the method of symbolic execution, not the problem of path selection. Again, the major obstacle was solving for the many constraints associated with a path. Ramamoorthy and his colleagues described a systematic trial-and-error method for generating test data.

Ramamoorthy concluded that although generating inputs from path constraints is not easy and includes problems such as determining the number of loop iterations still requiring user assistance, his work demonstrated the feasibility of a generator that partitions the input space. This method of selecting inputs, combined with other kinds of selective testing, is a valuable program testing tool for small programs.

Huang and Ramamoorthy showed that selecting inputs based on the structure of the programs (i.e., the paths) is feasible but tedious. However, even if

---

[13] C. V. Ramamoorthy, SIU-BUN F. Ho, W. T. Chen, "On the Automated Generation of Program Test Data," *IEEE Transactions on Software Engineering*, December 1976, pp. 293–300.

all paths are identified and all paths are tested, not all errors will necessarily be discovered, so there are flaws in testing the paths if the intent is to discover all errors. In general, path testing may not uncover all errors.

Myers focused on two such path-testing flaws.[14] The first flaw is what Huang and Ramamoorthy demonstrated above; i.e., that the number of paths through even the simplest programs can be extremely large. This is why solving for all path constraints is so difficult. Secondly, Myers pointed out that testing all paths may still leave a program containing errors: for example, the program may still not meet its specifications, missing paths may not be detected, and data-sensitivity errors that occur for only certain combinations of input may not be detected by testing all paths. Myers concluded that exhaustive black-box input testing is still superior to exhaustive path testing (white-box testing), but that neither is feasible.

The third type of test data generation is pathwise generators. These generators select data that will cause the execution of selected paths. Most systems use linear programming algorithms to find numerical solutions to the inequalities of path constraints. Among systems that use these techniques are SELECT, GOMORY, and BENDERS. The CASEGEN generator uses a trial-and-error method in conjunction with a random number generator. SMOTL generator uses a fast segment algorithm.[15]

---

[14] Myers, pp. 10–11.

[15] DeMillo et al., p. 109.

1.3.2 Data Specification Test Generators

A data specification system assists a user in the generation of a test case by providing a data specification language to describe the input domain. The description is used to generate the input data. Some examples are ADG (a compiler that translates English-like language), GENTEXTS (a system that processes language in the form of command grammars), and DATAMACS (a flexible file generator for COBOL programs). [16]

Syntax-based data generation is the earliest method of automatically generating test data.[17] This method processes a description of the input space expressed in a notation such as Backus Normal Form (BNF). Its major use has been compiler testing. The problem with syntax-based data generation is that, while it is easy to generate programs that are syntactically correct, in the BNF there is no useful user interpretation. Some proposed solutions to this problem are dynamic syntax (the addition of rules to test the syntax) and attribute grammars (these grammars have contextual information associated with terminals and non-terminals in their definitions). The latter solutions may also solve the problem of checking test output automatically during the random testing process.[18] Verifying that the output is correct is the part of the testing pro-

---

[16] Ibid., pp. 109–110.

[17] W. H. Burkhardt, "Generating Programs from Syntax," *Computing*, January 1967, p. 53.

[18] Ince, p. 64.

cess assigned to what we call the "oracle". Automating the oracle could greatly speed up testing.

1.3.3 Random Test Generators

There are two types of random test generators. The first type uses knowledge of the program to randomly select a path domain, then another random selection is made from within that path domain. The second type chooses a random point from the input domain based on a user distribution. The main advantage of this type of random test generator is its simplicity; it is also the easiest way to introduce some independence from knowledge of the program into the testing process. This type of random test generator is also less expensive in terms of support software, and it is straightforward to derive estimates of the operational reliability.[19] Hereafter, when the phrase "test generator" is used, it will be the second type to which we will be referring.

1.4 What is Statistical Testing?

Statistical testing does not use the structure of the program to design test cases. Inputs are randomly drawn not from path domains but from the entire input space based on the operational environment. The input domain can be stratified into two partitions: specification and nonspecification. If the usage

[19] Ibid., p. 63.

environment contains points from the nonspecification partition, the oracle automatically judges these as failures. This process will be fully described in the next chapter. The size of the sample should be based on the density of failures in the input domain and using the binomial distribution.

When a failure is observed, the program undergoes an engineering change to correct the underlying error. A new version of the program is created and testing begins anew. Testing until first failure reflects a technique called Pascal Sampling. The failures are considered rare events; their number is small relative to the input domain size. A lower confidence bound can be established and a warranty statement issued for the software.

1.5 Advantages of Statistical Testing

In statistical testing the main idea is not to find all the errors (it is impossible to know that all the errors have been found no matter what method is used). The main objective is to certify that the program is correct within a specified confidence level and to find the errors most likely to occur in the real usage environment.

A good example of how statistical testing works is reported in an article by Currit, Dyer, and Mills.[20] The authors examine the software failure history of

[20] P. Allen Currit, Michael Dyer and Harlan D. Mills, "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering*, January 1986, pp. 4–5.

nine large IBM software products from a study done by Adams in 1980. Table 1-2 shows the data from that work. The Adams study identified eight failure rate

Table 1-2: Fitted Percentage Defects.

Mean Time to Problem Occurrence in Kmonths by Rate Class

Product	60	19	6	1.9	.6	.19	.06	.019
1	34.2	28.8	17.8	10.3	5.0	2.1	1.2	0.7
2	34.2	28.0	18.2	9.7	4.5	3.2	1.5	0.7
3	33.7	28.5	18.0	8.7	6.5	2.8	1.4	0.4
4	34.2	28.5	18.7	11.9	4.4	2.0	0.3	0.1
5	34.2	28.5	18.4	9.4	4.4	2.9	1.4	0.7
6	32.0	28.2	20.1	11.5	5.0	2.1	0.8	0.3
7	34.0	28.5	18.5	9.9	4.5	2.7	1.4	0.6
8	31.9	27.1	18.4	11.1	6.5	2.7	1.4	1.1
9	31.2	27.6	20.4	12.8	5.6	1.9	0.5	0.0

Source: P. Allen Currit, Michael Dyer, and Harlan D. Mills, "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering*, January 1986, p. 5.

classes that are grouped by their reported frequency of occurrence (MTTF) and are the column headings in Table 1-2. The rows in Table 1-2 represent the nine IBM software products. The table entries are the percent of errors found in each program that falls into a particular rate class. For example, product 2 might be a time-sharing subsystem for which 34.2% of all failures were in the rate class of 60 thousand months or 5000 years MTTF. Currit calls attention to the "remarkable consistency in the failure rates although the products are

quite different from each other."[21] In this context serious means the number of failures per error. As the rate class becomes less serious, the effect on the MTTF from the correction of an error lessens, the size of the MTTF increases, and the percentage of low rate errors increases.

Currit, Dyer, and Mills point out that finding errors haphazardly (as occurs in anecdote testing) is very different from finding execution failures at random (as occurs in statistical testing). So, statistical testing tends to find errors in the same order as their seriousness, gives a statistical basis for product usage, and provides a scientific basis for making statistical inferences (e.g., MTTF and variance around the MTTF) from the testing to operational environments.[22]

Duran and Wiorkowski argue that this method of sampling "can tractably quantify the degree of correctness of a program."[23] Further importance is given to this idea in two examples given by the authors.

An upper confidence bound is computed for a hypothetical program that yields no errors when tested. In the first example the program structure is not considered. In the second example paths are considered with an approximation error introduced by the authors. The examples show no great advantage in using paths to compute the confidence bound.

The authors conclude that the cost in determining path information, when it is possible to do so, may not yield significantly better results than the simpler first example. They also conclude that a random sampling procedure could be

---

[21] Ibid., p. 5.

[22] Ibid., pp. 5–7.

[23] Joe W. Duran and John J. Wiorkowski, "Quantifying Software Validity by Sampling," *IEEE Transactions of Reliability*, June 1980, p. 141.

replaced with one that "reflects the actual probability distribution on the input sequences as they will occur in production. This allows estimating the actual, operational reliability."[24]

1.6 Purpose: How to Statistically Test Software

This research provides a step-by-step method for statistical testing of software: (1) generating inputs that fall within program specifications (and outside if needed), (2) selecting inputs randomly according to the operational distribution, (3) establishing an appropriate sample size according to input domain failure density, and (4) establishing a scientific statement about the goodness of the software.

To achieve (2), inputs into a program must be generated from a description. The description will contain the rules necessary to create an input domain specific to the program. A usage environment will be simulated by the tester's best guess of what the usage environment is really like or actually observed to produce a usage distribution. An oracle will be established to judge each input as correct or incorrect. If unanticipated inputs from the nonspecification partition are found to be in the usage environment, then they are considered by the oracle as incorrect. We have accomplished an organization and explication of ideas involved in statistical testing of software. This has included careful attention to the definition of error, fault, failure, clarification of the statistical issues, and a description for conducting such tests and reporting the results.

---

[24] Ibid., p. 142.

CHAPTER 2

GENERAL MODEL FOR STATISTICAL TESTING

2.1 Diagram and Description

Table 2-1 lists new symbols and definitions introduced in this chapter. Figure 2-1 presents the general model for statistical testing of software in a schematic form. This format shows each part of the testing apparatus and its relationship to other parts.

At the top of Figure 2-1 are the two possible input sources to a program under test. The input domain can be sampled directly, or inputs may be artificially generated with a test data generator. The generator should sample according to a distribution that represents the usage environment, which is a subset of the input space. This is because many programs will accept syntactically valid data that is unlikely to actually occur in the real world operating environment. Statistical testing for the ultimate purpose of warranting the performance of software in operation by customers requires this distinction.

Another source of input to a program under test is the state data (not to be confused with extraneous states). This must be controlled by the test data generator and treated as a random variable. This might be an interesting way to stratify the input domain. State data will be a big consideration in real-time and on-line transaction processing applications.

Table 2-1: Notation for Statistical Model.

Symbol	Definition
F	Program; a computable function.
D	Domain; all possible inputs to a program.
R	Range; all possible outputs from a program.
S_F	Specification of program F.
S_D	Inputs anticipated by S_F.
$D - S_D$	Inputs not anticipated by S_F.
U_F	Probability density function of the usage environment.
O_F	Oracle for program F.
d	A point in the domain D.
P_F	Path equivalence relation that induces a partition on D.
E_i	Set of all inputs that are observed as producing failures.
CN	Cardinal number.
E_F	Union $\overset{\cup}{i}$ of all E_i.
$f.d.$	$\frac{CN(E_F)}{CN(D)}$; failure density.
$f.r.$	$U_F(E_F)$; failure rate.
T	Execution path traced through a program.

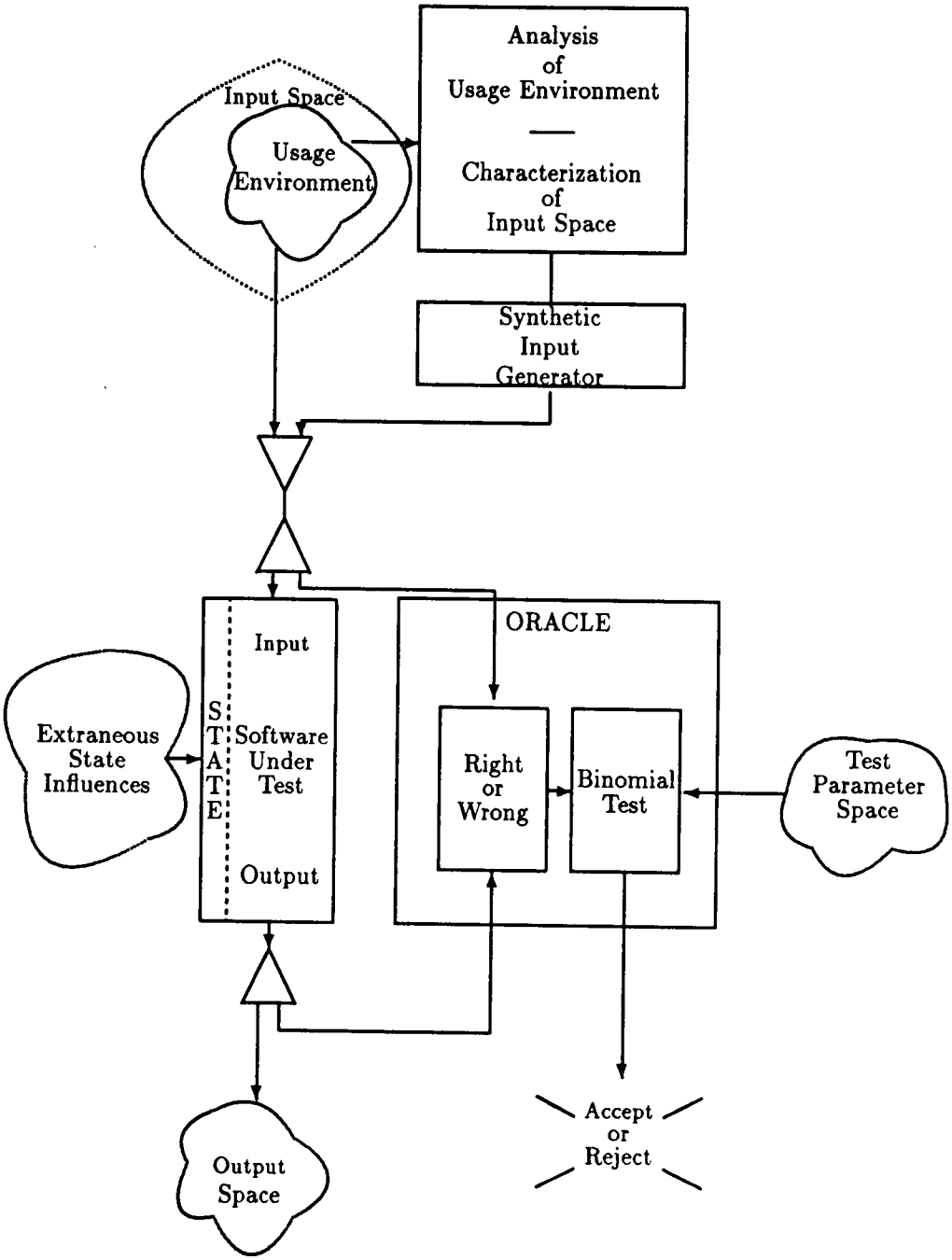


Figure 2-1: A Model for Statistical Testing of Software.

The program under test accepts inputs and produces outputs. This is done in the operational environment of a computer that is experiencing many other change-of-state influences in addition to the inputs and behavior of the software under test. Extraneous state changes might include diverse phenomena ranging from faulty hardware or an operating system gone amiss to other programs operating in a multiprogramming environment. As shown earlier in Table 1-1, these members of what we are now calling extraneous state changes account for only 10% of observable failures. Some extraneous states will not influence the behavior of the program under test, other extraneous states will cause the output to be wrong. However, Figure 2-1 shows the inescapable fact that extraneous state changes could impact the software under test and vice versa.

An oracle must be available that has the knowledge and ability to decide for each input-output pair whether or not the program under test performed correctly. Furthermore, the oracle must know the parameters of interest necessary to conduct a binomial test that will reject or accept an appropriate hypothesis. Since the efficacy of statistical testing will rest upon the ability to automate the process, the identification of a suitable oracle is important.

Each output of the software under test will be right or wrong as judged by the oracle. The binomial test is the correct statistical model for testing in this case. Although the distributions associated with the usage environment are important to the testing process and to reliability modeling, these distributions have nothing to do with judging the correctness of the outputs.

2.2 An Algebraic Description of Programs, Faults, and Failures

Consider a program to be a computable function F mapping from a finite input domain D of all possible inputs to a range set R of results:

$$F : D \rightarrow R.$$

Program F was designed and written to satisfy a specification, also a function, which we denote S_F. It is sometimes the case that a specification takes into account only plausible inputs and certain anticipated invalid inputs, and we denote the subset of D that S_F addresses S_D, $S_D \subseteq D$.

When F is in actual use, all the users of F collectively define the usage environment. Regardless of what the designers intended and anticipated, regardless of what the programmers encoded, the distribution of inputs that is processed by the program is known as the usage environment, and it is treated here as a probability density function U_F. Thus, if X is a random variable, the probability of an input $d \in D$ being presented to F is:

$$\Pr(X = d) = U_F(d).$$

Further,

$$U_F : D \rightarrow [0, 1]$$

and

$$1 = \Pr(D) = \sum U_F(d) \text{ for all } d \in D.$$

Because we are dealing with finite machines, even though the number of inputs could be astronomical, the domain of any implementation of F is strictly finite. Figure 2-2 shows the mapping of inputs in domain D through program F to range R.

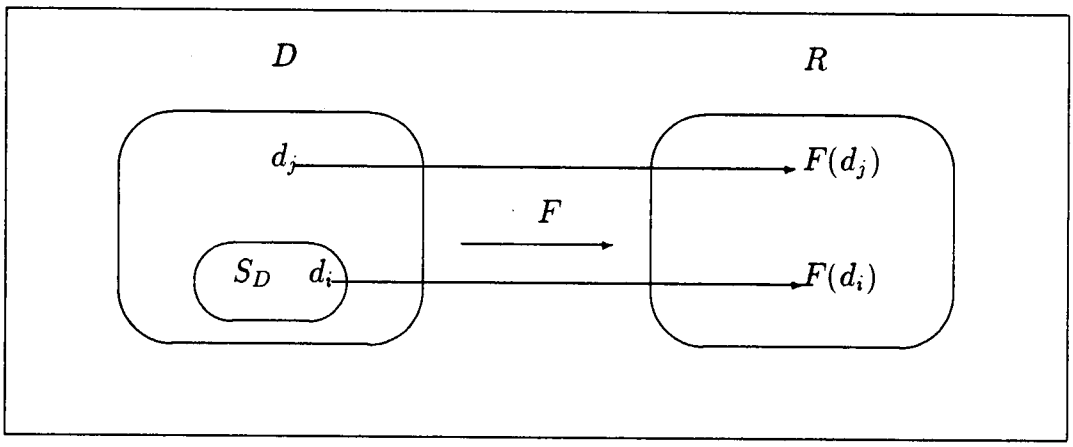


Figure 2-2: Program F.

As depicted in Figure 2-2, the inputs anticipated by S_F are the set S_D and the inputs not addressed by S_F are in $D-S_D$, the complement of S_D with respect to D. If S_F dictates that every input physically possible must be handled by F, even if it rejects some as ill-formed as a compiler might, then $S_D = D$ and $\Pr(D - S_D) = 0$. Figure 2-3 shows the usage distributions divided into S_D and $D - S_D$.

When $D - S_D \neq \emptyset$, U_F must reflect the probability of those elements being presented to F even though they are outside the specification. U_F reflects what really happens in the world of users, which may be quite different from what the framers of S_F anticipated or intended. Thus $D-S_D \neq \emptyset$ implies $\Pr(D-S_D) > 0$.

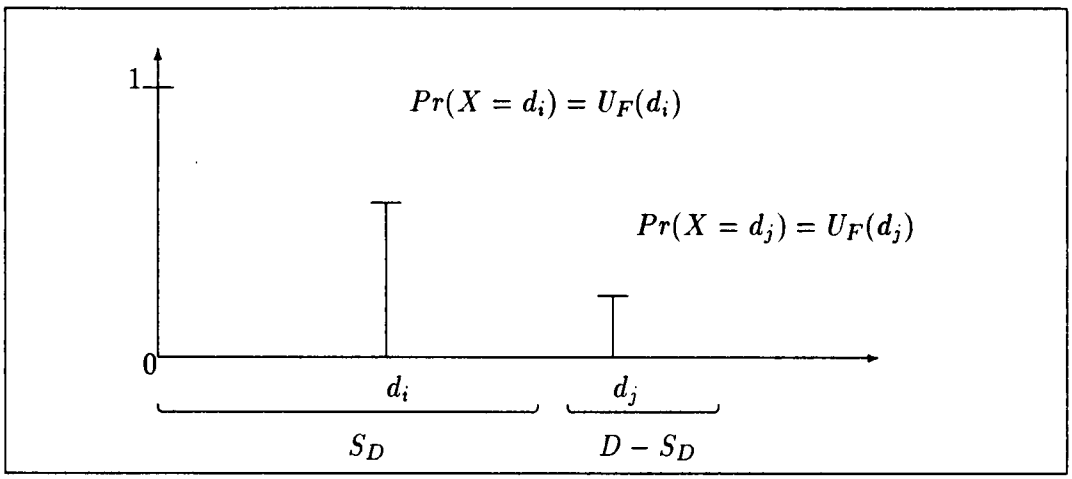


Figure 2-3: U_F: The Usage Distribution of F.

One turns next to the question of whether or not F performs according to its specification S_F. This question will be resolved by an oracle for F denoted O_F. O_F is a valuing function that judges the performance of F right or wrong and is defined as:

$$O_F : D \times F(D) \rightarrow \{0, 1\}$$

which we abbreviate to:

$$O_F : D \rightarrow \{0, 1\}$$

with the subscript F on O_F indicating O_F's knowledge of program F. The specification maps S_D into R and is presumed always correct. Thus:

$$O_F(d) = 1 \text{ iff } F(d) = S_F(d).$$

By this definition, $O_F(D - S_D) = 0$, since S_F is not defined on $D - S_D$ and presumably whatever F does with those inputs is wrong. To the extent F is correct, $F(S_D)$ will equal $S_F(S_D)$; but if F is flawed they will not be equal.

Statistical testing of a program F is based upon drawing a random sample of inputs according to U_F, observing the number processed correctly by F as judged by O_F, then performing certain statistical analyses. This process is described in detail in the following chapters. These relationships are depicted below in Figure 2-4.

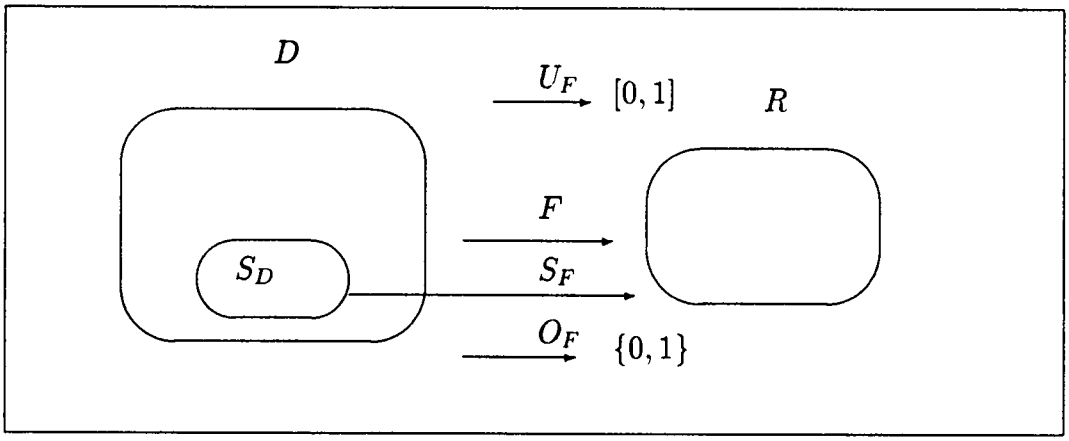


Figure 2-4: The Testing Process.

2.2.1 Paths

Each execution of F traces an execution path T through F. In general each $d \in D$ will not trace a unique path through F. We will say that two elements $d_i, d_j \in D$ are path equivalent iff, $T(d_i) \equiv T(d_j)$, iff they trace identical execution paths through F and write $< d_i, d_j > \in P_F$. P_F is an equivalence relation and induces a partition on D, D/P_F where:

$$D = D_1 \cup ... \cup D_n \text{ and}$$

$$\emptyset = D_i \cap D_j \text{ for all } (i \neq j).$$

Since all elements of a partition trace the same path and no other elements trace that path, each partition is tantamount to a path.

It is not the case that all elements of a partition will be computed correctly or all be computed incorrectly! Consider a program where "$\geq$" mistakenly occurs for "$>$". For some paths through the statement containing this error, it is conceivable that certain computations will be correct while others will be incorrect:

$$< d_i, d_j > \in P_F \not\Longrightarrow O_F(d_i) = O_F(d_j).$$

2.2.2 Failures

Whenever $O_F(d) = 0$ we say a "failure" has been observed. In fact, $O_F^{-1}(0)$ partitions D into two subsets, the subset of inputs that are judged correctly processed and the set that produces failures. As explained above this grand division does not respect the boundaries of D/P_F. Figure 2-5 shows the partitions of D.

O_F induces an equivalence relation on D, partitioning D into $O_F^{-1}(0)$ and $O_F^{-1}(1)$, which we write as D/O_F.

$$O_F^{-1}(1) \subseteq S_D$$

restates that serendipity outside S_F does not count, and

$$O_F^{-1}(0) \cap S_D \neq \emptyset$$

would imply that failures might be observed, i.e., that it would imply that "faults" (a term not yet defined) exist in F. Dotted lines in Figure 2-5 indicate

that the "0" partition of D may encroach on S_D while the "1" partition of D must be entirely in S_D. The slash marks across D represent D/P_F to illustrate that anything is possible concerning paths and failures.

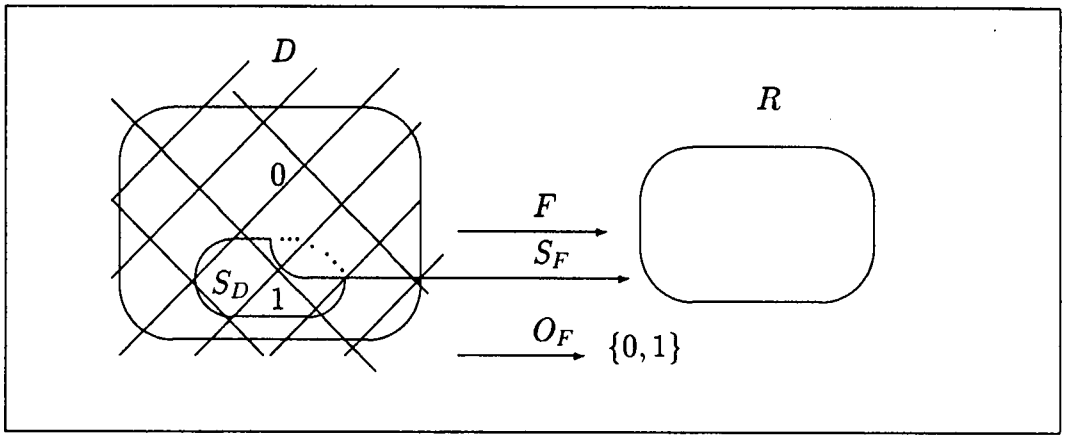


Figure 2-5: Partitions on the Input Space.

2.2.3 Errors and Faults

Let us say that if $O_F(F(d)) = 0$ is an observed failure, it was caused by some fault in F. We will identify an "error" E_i to be the set of all inputs that lead to observable failures because of a fault in the code, and $E = \{E_1, E_2, ..., E_n\}$ will be the set of all errors made in writing the program F.

Thus, an error in human thinking leads to a fault in the code, and we identify or define the error by the set of all inputs which reveal the fault when the program processing results in a failure. What can we say about E? Suppose that F is perfect (i.e., no errors when written and code contains no faults) and $O_F^{-1}(0) = \emptyset$.

By our definitions,

$$\begin{aligned} O_F^{-1}(0) = \emptyset &\Rightarrow U_F(O_F^{-1}(0)) = 0 \\ &\Rightarrow D = S_D = O_F^{-1}(1) \\ &\Rightarrow U_F(O_F^{-1}(1)) = 1. \end{aligned}$$

If F is perfect, there is no input that will exhibit a failure. Suppose F contains exactly one somewhat simple error (a single character mutation from the perfect program). Then $E = \{E_1\}$ and $E_1 \neq \emptyset$,

$$E_1 \neq \emptyset \Rightarrow (\exists d)(d \in D \wedge O_F(d) = 0)$$

which says that at least one input exists that will expose E_1. Since we are assuming only one simple error exists, we know

$$\begin{aligned} E_1 &= \{d | d \in D \wedge O_F(d) = 0\} \\ E_1 &= O_F^{-1}(0) \\ U_F(E_1) &> 0 \end{aligned}$$

and also that

$$\left.\begin{array}{ll} E_1 \subseteq D - S_D & \\ E_1 \subseteq S_D & \\ E_1 \subseteq D \text{ (i.e., some in } S_D \text{ and } D - S_D) & \end{array}\right\} \text{each are possible.}$$

Errors do not respect path partitions because, for example, if global state data is used and the error is an erroneous constant, then several paths may be traversed by the inputs that exposed the error.

Suppose two errors exist, $E = \{E_1, E_2\}$. What can we say about E? Unfortunately, there are many possibilities: $E_1 \subset E_2$, $E_1 \cap E_2 = \emptyset$, etc. The most useful information involves U_F:

$$\begin{aligned} U_F(E_1 \cup E_2) &> 0. \\ U_F(E_1 \cup E_2) &\leq U_F(E_1) + U_F(E_2) \end{aligned}$$

We will say that errors E_i and E_j are independent iff $E_i \cap E_j = \emptyset$, but note that this is not guaranteed even under the assumption of only two errors. The independence of errors can be alternatively stated as:

$$E_i \cap E_j = \emptyset \iff U_F(E_i \cup E_j) = U(E_i) + U(E_j).$$

This algebraic definition of independence of errors has little to do with human perception of the independence of errors. Since generally the intersection of all sets in E is not empty, errors do not induce an equivalence relation on D.

One need not go further with the introduction of errors except to note that as more errors and complex errors are introduced, U_F on the union of all the sets in E, E_F, increases monotonically. The failure density of a program F can now be defined as

$$f.d. = \frac{CN(E_F)}{CN(D)}$$

and the failure rate as

$$f.r. = U_F(E_F).$$

Failure rates are the important measure in the usage environment, and $f.r. = f.d.$ only in the case of all inputs being equally likely. Statistical testing causes failures to be observed that reveal faults in proportion to their seriousness (in Adams' sense).

If one knows nothing about the methodology by which a program is developed and has no useful knowledge of the structure, then we have described all the facts pertinent to this treatment of statistical testing.

It would seem possible to invent methodologies that would establish further useful facts. For example, reliability models often assert the independence of errors for analytical purposes. As we have seen, the condition is unlikely to exist in practice. However, one can imagine a methodology that would guarantee this result, or at least make it plausible. Certain methodologies result in properly structured programs, others lead to careful data structuring. Factual information assured by methodologies, language processors, and utilities could induce useful structure on the algebra or errors which, in turn, might make testing more effective.

2.3 A Few Remarks About the Nelson Model

Nelson introduced a method for designing test cases so that "their associated subsets cover the entire input domain, allowing reliability estimates to be made for expected operational use profiles."[25] Each input is selected randomly according to the operational distribution. The program is not changed when errors are

[25] Eldred Nelson, "Estimating Software Reliability from Test Data," *Microelectronics and Reliability*, **17** (1978), pp. 67–74.

detected, so the size of the remaining errors in the program remains constant. By doing this the complications associated with estimating the remaining errors due to changes made by debugging actions are avoided.

A reliability calculation is given for this model as

$$\hat{R} = 1 - \frac{x}{N}$$

where N is the total number of trials, and x is the number of failures out of N trials. This is a binomial test from which the program's probability of success is estimated.

Nelson's model is based on the random selection of test cases (input points) from a usage environment. This model establishes the fundamental principle of the statistical testing of software. This differs greatly from the conventional testing methods based on anecdotes (also called selective testing). The tendency in these conventional methods (e.g., boundary and path testing) is to hunt for errors based on the structure of the program.

Ramamoorthy and Bastani critiqued the Nelson model and described it as "the only model whose theoretical foundations are sound."[26] They note four practical drawbacks: (1) high confidence in the reliability means many test cases, (2) continuity is not considered (i.e., if the program is correct for a test case, then it is correct for all test cases tracing the same path), (3) random sampling of the input domain is assumed, and (4) program complexity measures are not considered.

Ramamoorthy and Bastani note that the input domain model was extended

---

[26] C. V. Ramamoorthy and F. B. Bastani, "Software Reliability—Status and Perspective," *IEEE Transactions on Software and Perspectives*, July 1982, p. 365.

by Nelson to overcome these drawbacks which proceeded as follows. Nelson's input domain is divided into partitions called equivalence classes. The sample space may be divided into partitions. If the classes A_1, A_2, ..., A_m are mutually exclusive and exhaustive (i.e., if their union contains all the sample points of S), then we say that the m classes form a partition of the sample space S into m events. For all partitions in a sample space, the sum of their probabilities will equal one:

$$Pr(A_1) + R(A_2) + ... + R(A_m) = 1.$$

The classes may be divided by such strategies as paths or subroutines. Ramamoorthy and Bastani "assume that there is some continuity over an equivalence class, i.e., if the program executes correctly for an input from the j-th equivalence class, then it will execute correctly for any randomly selected input from the same equivalence class with probability $1 - \epsilon_j$, where $\epsilon_j \ll 1$."[27] The new reliability equation is given as:

$$\hat{R} = \sum_{j=1}^{m} (1 - \epsilon_j) * Pr(A_j)$$

where m is the number of equivalence classes, and $Pr(A_j)$ is the probability of selecting an input from the j-th equivalence class during actual operation.

As demonstrated in the previous section, the path partitions (or equivalence classes) do not ensure that all points in the class will yield the same value from the oracle. Errors cross path boundaries. Thus, the revised Nelson model may rely on a concept of continuity that is unwarranted in the absence of empirical evidence or a methodology to support the concept. It will also be shown in later

[27] Ibid.

chapters that program complexity and large test cases are not so serious a problem when using statistical testing as Ramamoorthy and Bastani hypothesized.

CHAPTER 3

GENERATING TEST CASES

If real input streams cannot be used to test a program, then a synthetic input domain must be generated. Even if a real input domain is accessible, it may be easier for the tester to get a statistical profile of this real environment and then use the profile to simulate the real one. This synthetic domain should look as much like the real one as possible, but this is the subject of the next chapter. The purpose of this chapter is to suggest ways to manage the generation of test cases and not to defend the choice of test cases.

3.1 Description of the Input Domain

As explained in Chapter 2, the inputs take the form of (1) data read in by the program or parameters passed to it (as in a subroutine), (2) state variables such as the global variables shared throughout a program (such as the common in a FORTRAN program) or a database, and (3) the extraneous states present at the time the program is running (memory placement of the program and operating system conditions). All these inputs must somehow be reproduced in a set of test cases that represent the usage environment. The third type of input (extraneous states) may be impossible to reproduce and is often ignored.

First, consider each input point as an independent random variable. Statisti-

cal independence means that if X and Y are random variables, then probability of their intersection will equal the product of their probabilities[28] :

$$Pr(X = x, Y = y) = Pr(X = x) \cdot Pr(Y = y).$$

A synthetic input generator will produce these points and each point will be considered a test case. The definition of synthetic input generator is any program that will generate test cases using some method that simulates the usage environment.

Recall from Chapter 2 that each point of the input space in Figure 3-1 will yield one output point. These points can be represented by a vector of one or more elements. Let F be a program with input space S_F. If $d_i \in S_F$ is randomly selected, then $F(d_i)$ will result in one trial that is either correct or incorrect. The point d_i may be composed of many elements that will cause the program to produce one output.

An element takes on values that may be controlled for example by four constraints or rules: (1) type, (2) range, (3) relational, and (4) positional. The variable type can be boolean, real, integer, hexadecimal, character, etc. Each element must have rules controlling the type and range. Rules about the precision will be considered range rules. A range of characters could be described by their numeric ASCII values. If a range is not appropriate for representing character data, a list of discrete values that the element can take on would be supplied instead. Other rules for character strings, such as the string length, could be considered range rules.

---

[28] Frederick Mosteller, Robert E. K. Rourke and George B. Thomas, Jr., *Probability with Statistical Applications*, (Reading, Massachusetts: Addison-Wesley, 1970), p. 465.

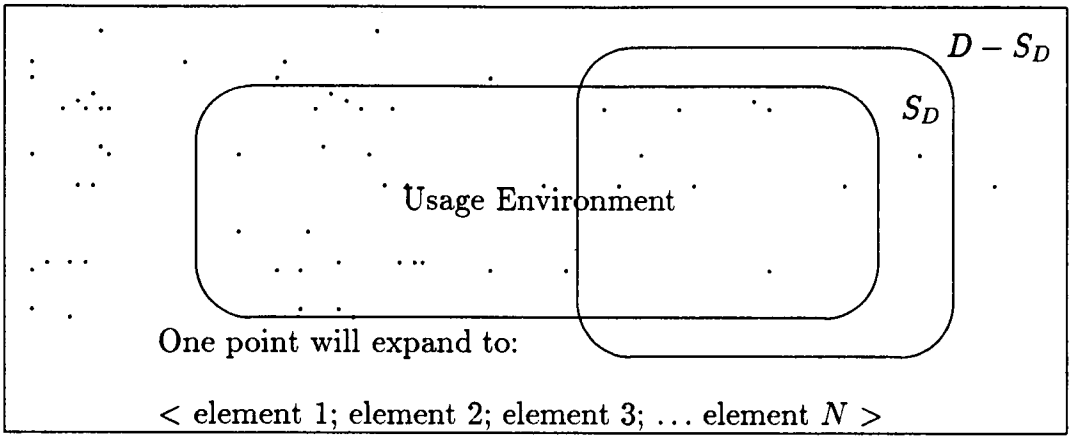


Figure 3-1: Point Expansion. Each point expands to a vector.

Relational rules describe how each variable relates to the others in the vector. Some variables may have no relational rules at all. Again, type and range rules must be specified, but relational and positional rules may not be needed. Figure 3-2 presents an example showing how relational rules can control the element DAY using a shorthand rule definition. This definition has the vector defined at the top and then each element is listed with the rules that apply. The rules are then listed below the elements. A synthetic input generating program could then be written from this prescription to produce test cases.

The range rule of element DAY is $1 \leq DAY \leq 31$, but the relational rules also restrict the value of DAY according to the value of MONTH. Another input domain is depicted in graph form in Figure 3-3. The definition of this domain uses the type, range, and relational rules. Figure 3-4 gives the vector with two elements and their rules. Notice that it is not enough to give a type and range here. The relational rule D must be used to define the complement space $D - S_D$ in the middle of the region S_D.

Vector:	< MONTH; DAY; YEAR >	
Type Definitions		
and Relational Rules:		
	1. MONTH: rules C,D	
	2. DAY: rules A,B,C and E	
	3. YEAR: rules C and F	
	Rule A:	When $MONTH = 2$
		and YEAR is a leap year;
		$DAY \leq 29$
		Else
		when $MONTH = 2$
		and YEAR is not a leap year;
		$DAY \leq 28$.
	Rule B:	When $MONTH = 4, 6, 9$, or 11;
		$DAY \leq 30$.
	Rule C:	Integer
	Rule D:	$1 \leq X \leq 12$
		(Substitute element name for X)
	Rule E:	$1 \leq X \leq 31$
	Rule F:	$X = 1988$

Figure 3-2: Input Definition Rules. Type, range, and relational rules.

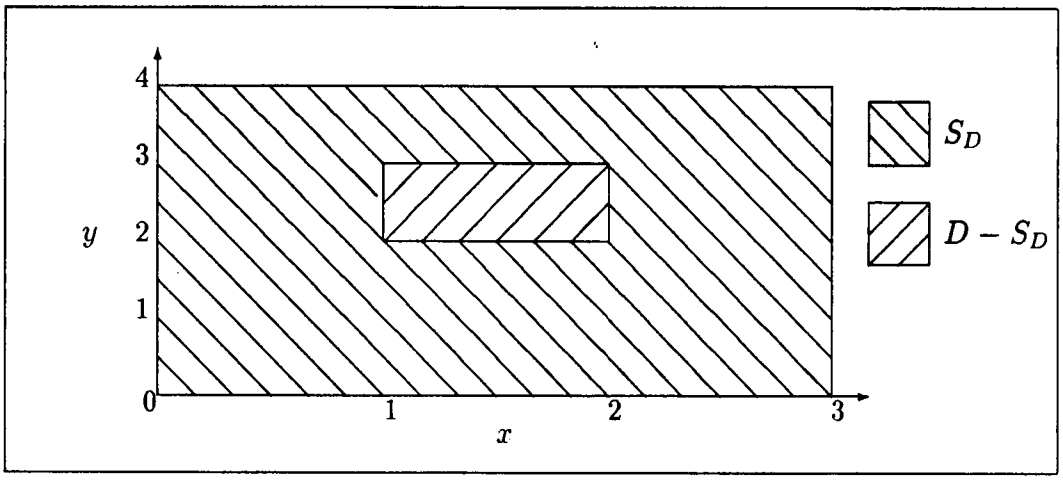


Figure 3-3: Graph of Simple Input Domain.

< XAXIS; YAXIS >

1. XAXIS:	rules A, B and D	
2. YAXIS:	rules A, C and D	
A.	Real	
B.	$0 \leq X \leq 3$	
	(Substitute element name for X)	
C.	$0 \leq X \leq 4$	
D.	XAXIS will not occur in the	
	range $1 \leq XAXIS \leq 2$	
	when YAXIS is in the	
	range $2 \leq YAXIS \leq 3$.	

Figure 3-4: Definition of Simple Input Domain.

Like relational rules, the positional rules may not be needed in the definition. The elements are already in a vector, which implies they occur in order. So what do the positional rules control? They control the rules of future elements that may occur further down in the vector. An example of this is found in an article by Hanford. [29] A FORTRAN program that does not have the IMPLICIT INTEGER (A-Z) statement might have declarations such as

INTEGER AY

INTEGER BEE

INTEGER C

and these three integer rules must be followed when elements using these integer variables are generated. In order for this FORTRAN program to be semantically and syntactically correct, the positional rules must be used.

3.2 SIAD Tree Representation

A method of keeping input definitions in structures called SIAD trees was developed by Cho.[30] SIAD is an acronym meaning symbolic input attribute decomposition tree. The SIAD tree can be used as a tool for describing the input domain of a program and as a basis for constructing test input units using random sampling. Cho describes four types of SIAD trees: regular, weighted, ruled, and network.

---

[29] K. V. Hanford, "Automatic Generation of Test Cases," *IBM Systems Journal*, 9 (4) , 1979, pp. 245–246.

[30] Chin-Kuei Cho, *Quality Programming: Developing and Testing Software with Statistical Quality Control* (New York: John Wiley & Sons, Inc., 1987), p. 162.

Cho simply calls the basic tree structure a regular tree. The regular tree can keep all information necessary to describe the range and type of an element. The weighted tree has this same information but also keeps the distribution of the elements based on the usage environment. How the distribution is arrived at is not discussed, but one would assume it would be based on an observation of the usage environment or the testers best guess of what the distribution would be. The ruled tree can handle all the above information plus rules that constrain elements by their relation to other elements. The last tree, the network tree, shows how the order of occurrence of values can be controlled.

The two examples given by Cho illustrate the generation input programs for compilers. The two compiler examples, one for FORTRAN and one for COBOL, show how a SIAD tree can be used to generate a syntactically correct program. In order to form acceptable tests for the compiler, Cho states that some "auxiliary statements such as DIMENSION, FORMAT, WRITE, STOP and END are required."[31] Positional rules are needed that control the above elements, and the SIAD tree does not handle this problem. So the SIAD tree does a good job of representing elements with type, range, and relational rules stored in the tree. A distribution of each element can also be stored in the SIAD tree. However, the problem of elements requiring positional rules is not handled by a SIAD tree.

---

[31] Ibid., p. 169.

3.3 Backus Normal Form and Dynamic Grammar

A metalanguage is a special formal notation for describing complex and flexible text structures. The input domain can be described using such a syntax notation introduced by Backus known as Backus Normal Form or Backus Naur Form (BNF).[32] There are classes of strings of a language (this could be applied to any input domain, not just languages) given a name enclosed in brackets, <>. The class is defined by one or more "writing rules." An example of such a rule is [33]:

$$< x > \rightarrow < y > +(< z >)$$

This rule says that class $< x >$ is defined by class $< y >$ followed by the character +, followed by the character (, followed by another class $< z >$, followed by the character). The classes $< y >$ and $< z >$ can be further defined by more writing rules. Notations may vary but the goal is always to fully describe a program language.

If one uses only BNF for generating test cases then they will all be grammatically correct but not necessarily semantically correct. The BNF would be accompanied by extra rules stated in English that could not be handled by BNF such as the rule:

> The constituents of an arithmetic expression must be declared real or integer.

---

[32] Peter Naur, *Concise Survey of Computer Methods* (Sweden: Studentlitteratur Lund Akademisk Forlag Kobenhavn, 1974), p. 180.

[33] Hanford, p. 244

These would be similar to the rules in the shorthand rule definition used earlier in Figure 3-2. The informal rules given above say that elements may occur in the vector if other elements have occurred earlier. The arithmetic expression may occur if the variables in the expression have been declared to be real or integer earlier. No other elements may occur until a PROGRAM statement has occurred. So what can be generated in one part of the vector generally depends in what has already occurred. Hanford describes this kind of rule, "which requires relations to hold between parts of a program" [34] (or parts of an input vector), a context-sensitive rule. His solution is to use a "dynamic grammar". This grammar will modify itself as random elements are generated. For example, each time a declaration is generated, a new rule is also added to the grammar. Then, when an arithmetic expression is generated, the new rule is available.

3.4 Summary

If the test cases needed do not require elaborate relational and positional rules, writing a shorthand rule definition or using a SIAD tree will be adequate. If more complicated test cases (such as test cases for a compiler) are required and correct syntax is all that is needed, then BNF will do the job. If the test cases must be syntactically and semantically correct, then an input generator that dynamically changes its rules must be used.

Creating a random set of test cases is crucial to statistical testing, but the means whereby this is done is really a separate problem. Methodology that

[34] Ibid., p. 245

requires statistical testing, viz. cleanroom, keeps this issue in the forefront during design and analysis, which, in turn, leads to great simplification of the allowable inputs to modules and programs.

CHAPTER 4

CHOOSING AN APPROPRIATE USER DISTRIBUTION

Once the input domain elements have been identified according to type, range, relational, and positional rules, the behavior of the usage environment must be simulated. There are two possibilities for each element of input, either its true usage distribution is known empirically or it must be stipulated on theoretical grounds. If the empirical distribution is available it would probably be used. If the distribution must be stipulated, then one can argue for the use of mathematically well studied and understood distributions. Further, the tester might make a case for using either continuous or discrete distributions, although we agree that computers are ultimately finite. This chapter reviews some readily available options, but the tester must in the final analysis defend his choices as both technically correct and believable in the context of the software being tested.

4.1 Distribution Descriptions

Each point in the input domain is an independent random variable and must be selected according to some usage environment distribution. There are many

distributions for continuous random variables that can be used to represent the usage phenomenon.

Hahn and Shapiro present graphs of distributions according to their behavior for different parameters in *Statistical Models in Engineering*. Appendix A has graphs of some well known distributions presented in this same manner.

4.1.1 Continuous Distributions

In the Hahn and Shapiro treatment, continuous distributions have three types of parameters that control the location, scale, and shape of each distribution. Not all parameters may be present in each distribution. The letter μ is the location parameter, σ and λ represent scale parameters, and η and γ represent shape parameters.[35]

Many kinds of measurements may be represented by the normal distribution, which is a symmetric distribution. The normal can represent elements that have no lower or upper bounds and bounded elements as well. An element with a lower bound of zero may be represented by a gamma, exponential, or log normal distribution. The beta distribution may be used to representing an element over a finite interval. The uniform distribution when bounded by zero and one is a special case of the beta distribution that occurs when both parameters η and γ are equal to one. This distribution may be chosen when nothing is known about the element.

---

[35] Gerald J. Hahn and Samuel S. Shapiro, *Statistical Models in Engineering*, Wiley Series on Systems Engineering and Analysis (New York: John Wiley and Sons, 1967), p. 68.

A list of probability density functions for these continuous distributions is in Figure 4–1. The parameters for each distribution are identified as being location, scale, or shape. How each of these parameters effects the curve may be seen in the graphs of Appendix A.

4.1.2 Discrete Distributions

Possible choices for a discrete distribution include the binomial, geometric, hypergeometric, negative binomial, Poisson, and the multinomial distribution. The probability density functions for these distributions appear in Figure 4-2. Many distributions, whether discrete or continuous, may be generated using a statistical package such as STATGRAPHICS or NAG.

4.2 User-supplied Distributions

Many times the input domain elements will not be represented by the usual discrete or continuous distributions mentioned above. The tester may define a special discrete distribution that best fits the element based on the tester's knowledge and analysis of the user environment.

For each discrete element value, the tester must assign a probability so that all characteristics of the probability density function are met; namely, that the function is positive on each value and they sum to one. This is how a unique probability density function can be created by the tester. The NAG library provides for the definition of a probability density function (pdf) and a cumulative probability distribution (cdf). Two subroutines in the NAG FORTRAN library,

Normal $\mathcal{N}(x;\mu,\sigma) = \begin{cases} \frac{1}{\sigma\sqrt{2\pi}} e^{\left[\frac{-(x-\mu)^2}{2\sigma^2}\right]} & -\infty < x < \infty, \quad -\infty < \mu < \infty, \quad \sigma < 0 \end{cases}$

location: μ scale: σ

Lognormal $\mathcal{L}(x;\mu,\sigma) = \begin{cases} \frac{1}{\sigma x\sqrt{2\pi}} e^{\left[\frac{-1}{2\sigma^2}(\ln x - \mu)^2\right]}, & x > 0, \quad -\infty < \mu < \infty, \quad \sigma < 0, \\ 0 & \textit{elsewhere} \end{cases}$

shape: μ , scale: σ

Beta $\mathcal{B}(x;\gamma,\eta) = \begin{cases} \frac{\Gamma(\gamma+\eta)}{\Gamma(\gamma)\Gamma(\eta)} x^{\gamma-1}(1-x)^{\eta-1}, & 0 < x < 1, \quad \gamma > 0, \quad \eta > 0 \\ 0 & \textit{elsewhere} \end{cases}$

shape: γ, η

Gamma $\mathcal{G}(x;\eta,\lambda) = \begin{cases} \frac{\lambda^\eta}{\Gamma(\eta)} x^{\eta-1} e^{-\lambda x}, & x > 0, \quad \lambda > 0, \quad \eta > 0 \\ 0 & \textit{elsewhere} \end{cases}$

shape: η, scale:λ

Exponential $\mathcal{E}(x;\lambda) = \begin{cases} \lambda e^{-\lambda x}, & x > 0 \\ 0 & \textit{elsewhere} \end{cases}$

scale : λ

Uniform $\mathcal{U}(x;\mu_0,\mu_1) = \begin{cases} \frac{1}{\mu_1-\mu_0}, & \mu_0 \le x \le \mu_1, \\ 0 & \textit{elsewhere} \end{cases}$

location: μ_0,μ_1

Figure 4-1: Continuous Probability Density Functions.

Source: Gerald J. Hahn and Samuel S. Shapiro, *Statistical Models in Engineering*, Wiley Series on Systems Engineering and Analysis (New York: John Wiley and Sons, 1967), pp. 68–80.

Binomial	$\mathcal{B}(x;p,N) = \binom{N}{x} p^x(1-p)^{N-x}$	for $x = 0, 1, 2, ..., N$
Pascal	$\mathcal{P}^*(x;p,k) = \binom{x-1}{k-1} p^k(1-p)^{x-k}$	for $x = k, k+1, k+2, ...$
Geometric	$\mathcal{G}(x:\theta) = \theta(1-\theta)^{x-1}$	for $x = 1, 2, 3, ...$
Hypergeometric	$\mathcal{H}(x;n,N,k) = \dfrac{\binom{k}{x}\binom{N-k}{n-x}}{\binom{N}{n}}$	for $x = 0, 1, 2, ..., n$, $x \leq k$ and $n - x \leq N - k$
Poisson	$\mathcal{P}(x;\lambda) = \frac{\lambda^x e^{-\lambda}}{x!}$	for $x = 0, 1, 2...$

Figure 4-2: Discrete Probability Density Functions.

Source: John E. Freund and Ronald E. Walpole, *Mathematical Statistics* (New Jersey: Prentice-Hall, Inc., 1980), pp. 164–191.

G05EXF and G05EYF, allow the user to create a specific discrete distribution. Figure 4-3 shows how these subroutines can be used.

```
      PROGRAM USERPROG
      INTEGER NP,IP,NR,IFAIL,IX,G05EYF
      LOGICAL LP
      REAL P(NP), R(NR)
C     SUBROUTINE G05EXF (P,NP,IP,LP,R,NR,IFAIL)
C     P - BEFORE ENTRY, P MUST CONTAIN THE PDF OR CDF.
C     NP - THE DIMENSION OF P.
C     IP - VALUE OF THE VARIATE, ASSUMED TO BE A WHOLE NUMBER.
C     LP - INDICATE IF THE DISTRIBUTION IS A PDF OR CDF.
C     R - THE REFERENCE VECTOR.
C     NR - THE DIMENSION OF R.
C     IFAIL - ERROR DETECTION FLAG.
C     IX - RECEIVES PSEUDO-RANDOM NUMBERS FROM G05EYF.
      DATA P(1), P(2), P(3), P(4), P(5), P(6), P(7), P(8), P(9),
     * P(10) /0.0,0.1,0.2,0.4,0.5,0.6,0.8,0.9,1.0,1.0/
      CALL GO5CBF(0) ! INITIALIZE THE RANDOM NUMBER GENERATOR.
      IFAIL = 0
      CALL GO5EXF(P, 10, 0, .TRUE., R, 19, IFAIL)
      DO 20 I=1,5
      IX = G05EYF(R,19)
      TYPE *, IX
20    CONTINUE
      STOP
      END
```

Figure 4-3: User Supplied Discrete Distributions Using NAG.

Source: NAG FORTRAN Library Routine Document, Mark 10, University of Liverpool, 1982.

CHAPTER 5

SAMPLING

Sampling is used primarily for estimating population parameters such as mean and variance. In testing software, the interest is in the proportion of input points causing program failure to the total number of trials or points. Exactly what constitutes a failure in this dichotomous population was defined in Chapter 2 and will be discussed further in Chapter 7. Sampling will be with respect to the distribution justified by the tester. The general guidance on sampling follows.

5.1 Simple Random Sampling

Random sampling is the selection of N points (called a sample) from a finite or infinite collection of objects (called the population, which in the case of software testing is the entire input domain D, and the total number of points in D is $CN(D)$) such that each point has an equal chance of being selected.[36] When testing a program for adherence to specifications, a simple random sample is often the best choice for testing. Simple random sampling is the recommended procedure for testing that portion of the input domain D that covers the inputs anticipated by S_F.

---

[36] William G. Cochran, *Sampling Techniques*, Wiley Series (New York: John Wiley & Sons, 1977), p. 18.

5.2 Stratified Sampling

Whereas simple random sampling gives good estimates of the population parameters with minimal cost and difficulty, stratified sampling may be used to improve the precision of the estimates of population parameters when the population can be readily partitioned into different groups or strata. Thus it is appropriate to use stratified random sampling when the population elements may be separated into nonoverlapping groups, with a simple random sample selected from each stratum.[37]

The first step is to determine exactly what constitutes each stratum and how many strata there are. Recall that the input domain is divided into two distinct nonoverlapping subsets S_D and the complement $D - S_D$. The latter subset is not considered for stratification because the results of any points coming from $D - S_D$ are immediately considered incorrect. So when might S_D be stratified? In the case where $D = S_D$ (a compiler that must respond to any input), the domain could be stratified into two groups: those points containing well-formed programs, and those that do not.

When testing points from S_D, what exactly is being measured? When a point from this group causes the program to execute correctly, then the functional capability of the program meets the functional requirements for well-formed programs. Sampling from S_D, then, will give a measure of goodness about how well the program designers have met the specification requirements for well-formed programs.

---

[37] Ibid., p. 89.

Sampling in the $D - S_D$ will reveal a measure of goodness about the program's ability to handle the unexpected. Sampling of this group would measure how extensive the designers have validated input. It may make more sense in some cases to use the test results from this subset as a way of fine-tuning the program to handle unexpected inputs gracefully. In some systems this may not be important and this subset will not even be sampled. In other systems, such as an online database where incorrect input could corrupt the database and ruin the system, it is very relevant.

Stratification by paths is not such a good idea. As stated earlier, partitioning by program paths is very difficult and the results may not be much better than one simple random sample.[38] Duran and Ntafos did simulation comparisons of random versus partition testing. They found that using larger numbers of random test cases was less expensive and generally as good or superior to partition testing.[39]

If the specifications state that certain types of invalid data are to be trapped, then one way to handle this is to extend the ranges and rules to accommodate it (generally the boundaries are checked). An alternative is to divide by the rules of the input domain where appropriate and make each a separate stratum. This decision is ultimately the tester's and depends on what the input domain looks like and what the tester wants to know.

Other reasons for stratifying S_D might be to profile different types of users into groups, different time periods that the program runs, or different hardware

---

[38] Duran and Wiorkowski, pp. 141–144.

[39] J. W. Duran and Simeon C. Ntafos, "An Evaluation of Random Testing," *IEEE Transactions on Software Engineering*, July 1984, pp. 439–440.

configurations that the program runs on–whatever the tester feels might be important to include in the reliability statement about the program.

5.3 Sampling in the Testing of Software

The testing of software differs from the classical examples of hardware testing. A machine or product (such as a light bulb or a circuit) can be tested under the same conditions many times with different results. One reason for this is that a physical machine or product wears with time. Software does not wear out, so the same input point always yields the same output result (if the initial and extraneous states are equivalent). For this reason a program can be described as a function because inputs going through the software are always mapped to the same output result. The definition of a computer program was given in Chapter 2 as a computable function F on the input domain D.

The input domain is finite, because all possible inputs must be represented by a machine (the computer) that has physical limitations. For the purpose of sampling and determining lower confidence bounds, the input domain may be treated as infinite because it is very large or sampling is with replacement. According to Kendall and Stuart:

> Sampling with replacement from a finite population may, in fact, be regarded as sampling from an infinite population, for the process will never exhaust the supply. We may, however, have to deal with a population which is infinite in rather a different sense, namely, that of a limiting form. We may, for example, wish to consider the probability of a sample from the positive integers or the real numbers from 0 to 1. The latter case presents itself in sampling

> from a continuous frequency–distribution which we must necessarily regard as infinite.[40]

If the total number of points in D (N_{pop}) is known and is small, then an exhaustive test might be performed.

5.4 Choosing the Sample Size

The sample size is determined before the testing takes place and is based on the amount of error the tester is willing to tolerate. We are looking for a sample size that will give us a "good" estimate of the reliability of the program with reasonable confidence. We use the binomial distribution to estimate sample size.

5.4.1 The Binomial Distribution

When there are only two outcomes to a trial such as success and failure, and the trials are independent and identically distributed, then the number of failures (or successes) in N trials is distributed according to the binomial distribution. Recall that independent trials, each with two possible outcomes and the probabilities of these two outcomes remaining unchanged from one trial to another, are called Bernoulli trials. A binomial distribution consists of a fixed number, N, of Bernoulli trials.[41] The density function of this binomial distribution is represented by[42]:

---

[40] Maurice Kendall and Alan Stuart, *The Advanced Theory of Statistics* (New York: Macmillan Publishing Co., Inc., 1977), p. 221.

[41] Mosteller, p. 130.

[42] Ibid., p. 139.

$$\mathcal{B}(N-x;p,N) = \begin{pmatrix} N \\ N-x \end{pmatrix} p^{N-x}(1-p)^x$$

where N = total number of trials, N=1,2,...,

x = number of failures, x=0,1,...,N,

and p = probability of a success, $0 \leq p \leq 1$.

Its usefulness here is to estimate sample sizes by considering the lower confidence bounds about the estimated probability of success.

This binomial distribution may be approximated by other distributions (such as the standard normal or Poisson). Even though the calculation of the sums in the above equation is straightforward, it can get tedious when N and x are large. When N is large (at least 50) the standard normal may be used.[43] But this is unacceptable if p is not close to 0.5. Because our interest is in values of p that are expected to be close to 1 (.999, .99, .95), the binomial function will be used for all calculations.

The lower confidence bound will be used to provide a means of estimating sample size. The fraction $1-\alpha$ is called the confidence coefficient or degree of confidence. The interval computed for a particular sample is called a $(1-\alpha)\%$ lower confidence bound. For instance, when $\alpha = 0.05$ the level of confidence is 0.95 and we obtain a 95% lower confidence bound. The approximate $100(1-\alpha)\%$ lower confidence bound is obtained by solving the following binomial equation for P_L (the lower confidence bound for p) with a fixed N and α.[44]

---

[43] Norman L. Johnson and Samuel Kotz, *Discrete Distributions* (New York: Houghton Mifflin Company, 1969), p. 61.

[44] Ibid., p. 58.

$$\sum_{j=x}^{N} \binom{N}{j} P_L^j (1 - P_L)^{N-j} = \alpha \tag{5.1}$$

Table 5-1 shows various lower confidence limits computed with the above formula for a fixed N, α, and $N - x$. For a constant confidence level and $N - x$, the lower confidence level increases with increasing N. Also, for a constant N and $N - x$, the lower bound decreases with increasing confidence level.

Table 5-1: The Lower Bound.

P_L about $\hat{p} = \frac{x}{N}$ for fixed N, $N - x$, and α.

		$(1-\alpha)$% Confidence Levels			
N	N-x	90	95	99	99.9
10	0	.794	.741	.630	.501
	1	.663	.605	.495	.376
50	0	.955	.942	.912	.870
	1	.924	.908	.874	.829
100	0	.977	.970	.955	.933
	1	.961	.953	.935	.911
500	0	.995	.994	.990	.986
	1	.992	.990	.987	.981
1000	0	.998	.997	.995	.993
	1	.995	.994	.992	.989

For a fixed p, α, and $N - x$, a value for N may be obtained by solving the above equation for N. Since this is a discrete distribution there may be no exact value of N for the parameters we choose, so the next closest one that yields an answer greater than α is selected. This insures that the sample is large enough to get the desired lower confidence bound. Table 5-2 shows values for N when

solving with a fixed α, $N - x$, and p. Notice that for a lower confidence bound of 99.9%, the values for N become very large in comparison to the rest of the table.

To select a sample size, first choose a confidence level. Next choose the lower bound P_L that you are willing to accept as a criterion for a reliable program if zero failures occur. For example from Table 5-2, 0.99 is an appropriate confidence coefficient and you are willing to accept a lower bound to the reliability of 0.99, select 460 or more as the sample size.

Table 5-2: Sample Size N

The Sample Size N That Would Generate Lower Confidence Limits P_L For Given $N - x$ and α.

		$(1-\alpha)$% Confidence Levels			
P_L	$N - x$	90	95	99	99.9
.9	0	21	28	43	65
	1	37	45	63	88
.95	0	44	58	89	134
	1	76	92	129	180
.99	0	229	298	458	687
	1	387	472	661	919
.999	0	2301	2994	4602	6904
	1	3888	4741	6635	9229

5.4.2 Pascal Sampling

When using the binomial distribution, the number of observed failures x is the random variable. In the testing of software, the number of acceptable failures for any experiment must be one in view of the requirements for independence

and the algebraic revelations of Chapter 2. So here we are fixing the number of failures in advance and allowing N, the number of trials, to vary. The number of test units required to test a program until one failure is observed is at least one, and this number is a random variable at the beginning of the test. This sampling method is known as Pascal sampling.[45]

The conditional probability, given p, that N trials will be required to produce x failures is given by the Pascal probability distribution[46] :

$$\begin{aligned} Pr(N|p) &= \{N \text{ trials are required to produce } x \text{ failures } |p\} \\ &= \frac{(N-1)!}{(x-1)!(N-x)!} p^{N-x}(1-p)^x, 0 < p < 1, \\ \text{where} \quad & N = x, x+1, ..., \text{ and } x = 1, 2, ... \end{aligned}$$

The distribution above is also commonly called the negative binomial distribution.

5.5 Rare Events and Cleanroom Methodology

The concepts of rare events may begin with the hypergeometric distribution which deals with finite populations. The hypergeometric distribution may be approximated by the binomial distribution leading to results that are similar to those found in Table 5-2. When dealing with high-quality software that has been designed under a methodology that eliminates more than "90 percent of

---

[45] Harry F. Martz and Ray A. Waller, *Bayesian Reliability Analysis*, Wiley Series in Probability and Mathematical Statistics (New York: John Wiley & Sons, 1982), pp. 253–254.

[46] Ibid., p. 254.

the total product defects before the first execution," a more rigorous sampling technique may be needed to uncover the small amount of errors present.[47] The cleanroom process developed at IBM is such a methodology.

In their article "Cleanroom Software Engineering," Mills, Dyer, and Linger describe a process where the first priority is defect prevention rather than defect removal. They focus on defect prevention by using human mathematical verification in place of program debugging. Even though human verification is fallible, they have found that it still produces software that is sufficiently robust to go to system test without debugging.[48]

If this process is combined with a formal design methodology, 90 percent of the total product defects in a program can be found before it is executed. Mills and Poore have set the cleanroom product threshold at 3 errors per 1000 lines of code, whereas the industry experience is 75 errors per thousand for the first time the software is put on the machine.[49] If more than three are found, they return to the design phase. The methodology in their process is described by Mills, Linger, and Hevner and consists of three basic system structures that provide different views of the system: black box, state machine, and clear box.[50]

Whatever sampling technique is used, the problem that must always be addressed is how large the sample size should be to satisfy the $(1-\alpha)$ and p levels. "The determination of the necessary sample size is more difficult when the num-

---

[47] Harlan D. Mills, Michael Dyer, and Richard C. Linger, "Cleanroom Software Engineering," *IEEE Transactions on Software Engineering*, September 1987, p. 20.

[48] Ibid., pp. 19–20.

[49] Harlan D. Mills and Jesse H. Poore, "Bringing Software Under Statistical Quality Control," *Quality Progress*, in publication.

[50] Harlan D. Mills, Richard C. Linger, and Alan R. Hevner, *Principles of Information Systems Analysis and Design* (New York: Harcourt Brace Jovanovich, 1986), p. 7.

ber of defectives (input space points causing the program to fail) is small relative to the population size."[51] Such is the case when testing a program designed with the cleanroom methodology. Table 5-3 contains additional notation needed in the following discussion of rare events.

Table 5-3: Notation for Rare Events.

Symbol	Definition
N_{pop}	D; the size of the dichotomous population
N	the total sample size, $N \leq N_{pop}$
N_f	E_F ; the number of defectives in D
X	the number of defectives in the sample
x	a value which the r.v. X has taken on
$\mathcal{H}(x; N_{pop}, N, N_f)$	the hypergeometric distribution
$\mathcal{B}(x; p, N)$	the binomial distribution

There are five assumptions made about the sampling model.[52]

1. Sampling is done randomly without replacement.

2. Sampling is from a finite population.

3. The population is dichotomous, i.e. each member can be classified as defective or nondefective (for our purposes the program fails or passes). Inspection perfectly classifies each member (the oracle does this).

---

[51] Richard W. Madsen and James E. Holstein, "Determining Sample Size When Searching for Rare Items," *IEEE Transactions on Reliability*, December 1982, p. 451.

[52] Ibid., p. 431.

4. The random number of defectives in the sample will follow a hypergeometric distribution with pdf:

$$\mathcal{H}(x; N_{pop}, N, N_f) = \frac{\begin{pmatrix} N_f \\ X \end{pmatrix} \begin{pmatrix} N_{pop} - N_f \\ N - x \end{pmatrix}}{\begin{pmatrix} N_{pop} \\ N \end{pmatrix}}$$

where x is a value which the r.v. X has taken on and x must also be an integer satisfying:

$$\max\{0, N - (N_{pop} - N_f)\} \leq x \leq \min\{N, N_f\}.$$

5. An inspection plan for a lot of items or set of components is as follows: inspect a random sample of N items. Accept the lot iff no defectives are found in the sample.

It would be desirable to have a high probability of obtaining at least one defective in the sample when the number of defectives in the population is equal to some specific value[53]:

$$Pr(X \geq 1 | N_{pop}, N, N_f) = \gamma.$$

A direct solution to the above equation would require using the hypergeometric distribution. This would be tedious; therefore, the authors use the binomial distribution to approximate the hypergeometric. If one is searching for rare items, then the binomial in which N_f is the sample size should be used. This was

[53] Ibid., p. 451.

labeled Binomial II by Duncan.[54] The solution to the problem of determining the sample size N to satisfy the above equation is:

$$\begin{aligned} Pr(X \geq 1 | N_{pop}, N, N_f) &= 1 - \mathcal{H}(0; N_{pop}, N, N_f) \\ &\approx 1 - \mathcal{B}_{II}(0; N/N_{pop}, N_f) \\ &= 1 - [(1 - N/N_{pop})^{N_f}]. \end{aligned}$$

Setting this last expression equal to γ and solving for N yields the conclusion of the Madsen and Holstein analysis. This estimate of N is:

$$N \approx N_{pop}[1 - (1 - \gamma)^{1/N_f}].$$

Now to adapt this result to high quality software, introduce a failure rate $\epsilon = \frac{N_f}{N_{pop}}$, where $N_f = \epsilon N_{pop}$ and the equation becomes:

$$N \approx N_{pop}[1 - (1 - \gamma)^{\frac{1}{\epsilon N_{pop}}}].$$

This relates N to the density of failure-bound inputs. So by L'Hospital's Rule:

$$\lim_{N_{pop} \to \infty} N_{pop}[1 - (1 - \gamma)^{\frac{1}{\epsilon N_{pop}}}] = \frac{1}{\epsilon} \ln(\frac{1}{1 - \gamma}).$$

For a large input space the sample size depends on the density of failure-bound inputs and the probability one sets of finding at least one such input. $N_{pop}[1-(1-\gamma)^{\frac{1}{\epsilon N_{pop}}}]$ is an increasing function of N_{pop} and the limiting value is an upper bound on the sample size necessary to establish a worthwhile lower confidence bound. Table 5-4 shows some sample sizes using this calculation. Let $P_L = 1 - \epsilon, x = 0$ and compare with Table 5-2.

---

[54] A.J. Duncan, *Quality Control and Industrial Statistics* (Chicago: Richard D. Irwin, Inc., 1965), pp. 87–88.

Table 5-4: Some Calculations Using $\frac{1}{\epsilon}\ln(\frac{1}{1-\gamma})$.

	ϵ	
γ	10^{-3}	10^{-5}
.9	$2.3 * 10^3$	$2.3 * 10^5$
.99	$2 * 2.3 * 10^3$	$2 * 2.3 * 10^5$

CHAPTER 6

THE ORACLE: OBSERVING AND CALCULATING TEST RESULTS

Choosing an oracle that is both reliable and fast is the most difficult step in the testing process. In the previous section it was demonstrated how many samples would be needed to provide acceptable confidence statements. In the case of software that is already of high quality, it is essential that large samples be drawn to find the small percentage of errors. Verifying that outputs from a program are correct can be very tedious, if not impossible, without a good oracle. The ideal is an automatic oracle.

The oracle can be another software product, a previous version of the software under test, a calculator, or any means to establish that the output is indeed correct. There may be programs for which there are only human oracles.

Human oracles are used most often. Usually the human is also the programmer. It is not a good idea to allow the programmer to use anecdotal testing on his or her own program. This is comparable to studying for a test when one already knows which questions will be asked. In the right situations, humans, individually or in teams, can be very effective oracles. If the software produces a graphic result, plays music, turns a valve, or mills a metal part, human oracles

can be very fast. A team of experts could be called upon to develop a set of rules or template with which to compare program output.

Other computer programs can serve as oracles. A symbolic algebra system like Maple might be suitable for testing numerical software. Since numerical software will use floating point hardware or software and perhaps undergo repeated transformations from floating point format to file storage format and to display format, a symbolic algebra system would be an effective oracle. Many well-established software systems such as NAG could be used to verify mathematical programs. A common practice is to move software from one computer to another. The software residing on the old machine can be used as an oracle to test the software on the new machine. New releases can be tested against previous releases with humans verifying the new capabilities of the new release. In general, any existing software that has earned the respect of the user community might serve as an oracle for new software.

Each time an input point is selected and given to the program under test, the result of this one trial is fail or pass. The oracle decides if the trial has failed, and this is recorded as the test continues. If the tester does not have access to the source code of a program, then all that can be done is to record the observed failures. Nelson[55], Duran and Wiorkowski[56], Schick and Wolverton[57], and Ramamoorthy and Bastani[58] all describe this as the operational reliability where we can estimate the reliability of a program by observing the outcomes of

---

[55] Nelson, p. 71.

[56] Duran and Wiorkowski, p. 142.

[57] George J. Schick and Ray W. Wolverton, "An Analysis of Competing Software Reliability Models," *IEEE Transactions on Software Engineering*, March 1978, p. 117.

[58] Ramamoorthy and Bastani, p. 355.

a sample of runs taken from the operating (or user) environment. If x failures out of N runs are observed, then the reliability estimate denoted by $\hat{R}$, is:

$$\hat{R} = 1 - \frac{x}{N}$$

The calculation $\hat{R}$ says nothing about the number of faults in the program or the confidence we have in $\hat{R}$. The failures are reported but the faults are not fixed. For software developers who want to also correct their programs as well as provide a warranty, an algorithm for determining when to stop and estimate the operational reliability is needed.

6.1 Failures and Errors

Recall from Chapter 2 that failures are revealed by inputs that lead to valuation by the oracle as wrong, i.e., $O_F(d) = 0$. An error is a set of inputs that lead to observable failure for the same program version, $E_i = \{d | d \in D \wedge O_F(d) = 0\}$. The execution failures described above are run-time failures as opposed to compile and link failures. Compile and link failures are not considered, and testing does not begin until after a clean compile and link have been achieved.

Faults causing execution failures are introduced into the software by human mistakes. "Human mistakes which create indigenous errors include: (1) deficiencies, misinterpretation, or disputed interpretation of design criteria (requirements), (2) logical inconsistencies or syntactical mistakes, (3) mistakes made in transcribing design into program statements or into the input data."[59] In the

[59] Schick and Wolverton, p. 108.

design and test phases of a software project, we are interested in the faults found and not just counting failures.

Ramamoorthy and Bastani give a formal definition of the size of an error as "the probability that an element selected from I (input domain) according to the test case selection criterion, results in failure due to that error. An error is easy (to detect) if it has a large size (i.e., if it affects many input elements)."[60] This concept of error size is identical to that given in Chapter 2 in terms of inputs and their probabilities of occurring. The final statement of goodness about the program will be valid only for that operating environment as reflected by the probability density function.

6.2 Algorithm for Testing

Consult Table 5-2 for the selection of parameters. What reliability must you have? What is the desired confidence level you must have? How many samples must be run? Can you afford the time and cost to run such a test? We can only tell the tester how these interrelate and show the trade-offs. The tester must select a set of parameters.

If fixes (or engineering changes[61]) are to be made each time a failure is observed, then the effect on the output must be considered. After the first observed failure, we make an engineering modification and at this point then we no longer have Bernoulli trials because the probability of an outcome has changed. This can be compared to flipping a coin and then altering the coin

[60] Ramamoorthy and Bastani, p. 357.

[61] Currit et al., p. 5.

each time before flipping again. The probability of the outcome will change; therefore a new series of trials is conducted each time.

The algorithm for testing software with the purpose of establishing the lower confidence bound selected above is as follows:

1. Start with the first test case and test the current version until:

- **(a)** The first failure is observed.
- **(b)** N test cases have been performed.
- **(c)** The number of observed faults > some stopping rule.

2. If condition (a) above then:

- **(a)** Allow the designers to fix the program. This new program becomes the current version.
- **(b)** Go back to step 1 and start again for the current version.

3. Else if condition (b) from step 1 then:

- **(a)** Confidence bound established. Issue a warranty statement (see next chapter).
- **(b)** Stop.

4. Else if condition (c) from step 1 then:

- **(a)** Send the software back to the design phase because it is not of sufficient quality to continue testing.
- **(b)** Stop.

The number of iterations is determined by the stopping criterion set by the tester. This stopping criterion determines how many engineering changes the tester is willing to allow before returning the software.

When a failure is observed, all necessary changes to the program that will correct the fault are considered one engineering change. If a new fault is introduced as the result of an engineering change, this will be reflected in the reliability of the next version. This new fault will have the same chance of being discovered as in the previous version because the sample is always taken from the same operating environment. The final product will be a current version of the software that runs for N trials with no observed failures.

6.3 Hypothesis Testing

It may be of interest to test whether the reliability claimed by a software manufacturer falls inside a specified region. We may accomplish this by using the idea of hypothesis testing.

For example, a company claims that their software has a reliability of at least 0.90. Thus the manufacturer's claim is that p, the probability of a correct result, is greater than or equal to 0.90. This establishes a region of acceptable values of p and we may test the hypothesis H_0 that p is ≥ 0.90. This is known as the null hypothesis and may be written as :

$$H_0 : p \geq 0.90.$$

An alternate hypothesis H_1 may be defined as the complement of this region and may be written as:

$$H_1 : p < 0.90.$$

To determine whether we accept the null hypothesis as being true, take a sample of 150 inputs and observe the output. Suppose the results of the test show 128 correct outputs.

For purposes of this test, it is convenient to use the normal approximation to the binomial distribution. We define a variable Z to be:

$$Z = (X - Np_0)/(Np_0(1 - p_0))^{.5}$$

where p_0 is the Pr(success$|H_0$ is true) and

X is the number of correct outputs.

For large N, Z is approximately distributed as standard normal. For the given outcome of the experiment, the calculation of Z is:

$$\begin{aligned} Z &= \frac{(128 - ((150)(.90))}{((150)(.90)(.10))^{.5}} \\ Z &= \frac{(128 - 135)}{3.674} \\ Z &= -1.908 \end{aligned}$$

We choose a critical region such that the error in rejecting the null hypothesis when in fact it is true is less than some small fraction $\alpha, 0 < \alpha < 1$. This error is referred to as a type I error. That is, if the test were repeated many times, the null hypothesis would be rejected about α percent of the time when in fact the null hypothesis is true. Here, we do not consider the alternate distribution and the associated error of accepting the null hypothesis H_0 when in fact the alternate hypothesis H_1 is true. This error is referred to as β or type II error, and unfortunately it is frequently overlooked when testing hypotheses. In any

application of the techniques outlined in this research, the β error should be considered.

From tables of the standard normal distribution[62] we find that the critical region for Z is $Z < -1.645$. We will reject H_0 if the calculated Z statistic is less than -1.645. Since $Z = -1.908$ is less than the critical region, we reject the null hypothesis and thus reject the claim by the manufacturer that the reliability of their software is 90% or better.

6.4 Summary

So if we do not have access to the source code of a program, then the best that can be done is to count the number of observed failures. This gives the operational reliability discussed earlier. A lower confidence bound can also be found by using equation 5.1. If changes can be made, then testing until the first observed failure and making engineering changes will result in a current version that we can make a scientific statement about. A lower confidence bound can be calculated and a stopping criterion for engineering changes can also be imposed if the software team has a goal in mind (cleanroom methodology).

Both of these methods are based on random testing from the operating environment. Both will provide early detection of failures that are linked to large or serious errors. Both will yield an estimate of the operational reliability and a lower confidence bound.

---

[62] Mosteller et al., p. 473.

CHAPTER 7

WARRANTY STATEMENTS

Any product that is purchased by a consumer should have a warranty statement, and software is no exception. The purpose of a warranty is to protect the consumer. The company with a fair and responsible warranty will have a competitive edge. Methods to guarantee or warranty that software will work correctly are presently not found in commercial software. Most software warranties are filled with disclaimers. Here are some examples of popular software product warranties:

> The software disks and the User Manual are free from defects in materials and workmanship; the software is properly recorded on the disks; the User Manual is substantially complete and contains all the information that is necessary for operation of the software; the software functions substantially as described in the User Manual. Except as provided above, this software is provided "as is" without warranty of any kind, whether expressed or implied, including, but not limited to the implied warranties of merchantability and fitness for a particular purpose. Our entire liability and your exclusive remedy shall be the replacement of any software disk or user manual returned to us which does not meet our limited warranty.
>
> —Persoft

> NO WARRANTY OF PERFORMANCE. We make no warranties, either express or implied, regarding the performance of the Program or the results that may be obtained by using the Program. ACCORDINGLY, THE PROGRAM AND ITS DOCUMENTATION ARE SOLD "AS IS" WITHOUT WARRANTY AS TO THEIR PERFORMANCE, MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. The entire risk as to the results and performance of the Program is assumed by Purchaser. Should the program prove defective, Purchaser (and not us or any of our dealers) assumes the entire cost of all necessary servicing, repair or correction.
>
> —Microsoft

7.1 What a Software Warranty Should Say

The warranty should include:

- The methodology used in designing the program.
- The method of testing and sampling.
- The test results.
- If appropriate to the methodology (for example, cleanroom), a statement of MTTF based upon a certain model would be in order.
- If statistical testing is used, give the version number of the software as a reference to the audit trail of engineering changes.
- A description of the input space, ie. usage distribution for which the software has been tested.

Figure 7-1 shows a more detailed software warranty than those shown previously.

The XYZ software package is sold with the following warranty:

Purpose:	This program scans an APT program and produces numerically controlled machine readable output or a diagnostic message describing a flaw in the APT program.
Input Domain:	A single well-formed APT program containing statements described in the APT programmers guide.
Output Domain:	A correct machine readable output or message.
Methodology:	The XYZ corporation uses software development methodology (SDM) for all its development.
Sample:	The test sample was based on a usage environment profile created from real APT programmer inputs at company ABC.
Testing:	Statistical testing.
Test Results:	The current version 2.5 of XYZ has been tested with 500 randomly generated test cases yielding no observed failures. Only ten engineering changes were made and five were attributed to usage environment adaptation.
Final Statement:	We are 95% confident that the XYZ software package has a reliability of 99% or greater. We do not warrant that the software will work outside the conditions defined above.

Figure 7-1: A Software Warranty.

CHAPTER 8

EXAMPLES

This chapter is based on a collection of experiments conducted by the author and her colleagues in several seminars on software reliability held at the University of Tennessee. The experiments reflect the different stages of understanding that we had about software testing, and the last experiment, the parallel axis graph, uses all the methodology of software testing described in this paper.

8.1 STATGRAPHICS SIN

The software chosen for this experiment is STATGRAPHICS. The SIN function was tested in the range of potential input values defined to be all eight decimal point values from zero radians to $\pi/2$ radians. This represents 157,079,633 values. When one considers that the number of possible beginning states for the SIN routine may be hundreds or thousands, it is obvious that exhaustive testing is impossible. To statistically test all such combinations, it would be necessary to identify all initial states and all possible inputs. This is still too many to test, therefore a single initial state of the SIN is assumed. There will be one output for each input. The input has a uniform distribution over the defined range. The output will have a binomial distribution since it will be graded as right or wrong, right meaning correct to eight decimal places, wrong otherwise.

8.1.1 Sample Size

A set of 150 random numbers between zero and $\pi/2$ was selected by the STATGRAPHICS random number generator for a continuous uniform distribution. These were used as the input values. The results of the test showed 128 correct sin values.

8.1.2 Oracle

The STATGRAPHICS user manual makes no statement about the number of decimal places of accuracy for the SIN routine. The output of the SIN function was considered correct to the eighth decimal place if it agreed with the output of the oracle to the eighth decimal place. The oracle was a VAX FORTRAN program using the same input. The FORTRAN program used REAL*8 variables which supposedly produce output correct to at least 15 decimal places.

8.1.3 Warranties

Out of 150 units tested, 22 were observed to cause incorrect output. Therefore the operational reliability of this program is

$$\hat{R} = 1 - \frac{22}{150} = 85.3\%.$$

The lower confidence bound $P_L = 77.3$ is calculated from equation 5.1. In our warranty statement we can say that STATGRAPHICS has a reliability of 77.3% or greater. If the manufacturer of STATGRAPHICS had made a statement

about the reliability of their product we could use the hypothesis test from Chapter 6 to see if it was true.

8.2 Job Submission Program

This experiment was performed on a FORTRAN program that submits other batch jobs to be run according to a table containing submission dates. Input to the program consists of the batch job name and the date it is to be submitted. The program compares each date with the current date and when there is a match, the program submits the job. The input space was partitioned as follows: 90% of all inputs meet the specifications of the program and 10% of all inputs are outliers (do not meet specifications). The outlier partition was further subdivided into two kinds of outliers: 3% random ASCII characters and 7% random integers.

The program was tested with 500 inputs generated as described above. No method was used to choose this sample size because at the time of this experiment, no method was yet agreed upon and an operational reliability was only being considered for the warranty. All inputs in the first partition (meet the specifications) caused correct execution of the program. The subpartition of integer outliers caused the program to execute as though valid inputs were used. When an outlier input produces an output that is indistinguishable from output produced by valid inputs, this is considered a failure. The random ASCII characters caused run time errors in every instance. This was considered the correct response for the program and not counted as an error.

8.2.1 Operational Reliability

Out of the five hundred units tested, 35 were observed to cause incorrect output. Therefore the operational reliability of this program is

$$\hat{R} = 1 - \frac{35}{500} = 93\%.$$

The lower confidence bound $P_L = 89.8$ is calculated from equation 5.1. We can state in our warranty that the program has a reliability of 89.8% or greater. The case can be made from the definitions in Chapter 2 that the 10% random ASCII characters and integers were not part of the specifications and should also be counted as producing incorrect output, in which case the reliability would be

$$\hat{R} = 1 - \frac{85}{500} = 83\%.$$

The lower confidence bound is $P_L = 78.7$. We can state that the program has a reliability of 78.7% or greater.

8.2.2 Testing of Two FORTRAN Subroutines

Two short FORTRAN subroutines were tested using statistical testing methods. The subroutines are part of a larger postprocessor system used in producing mylar tapes for numerically controlled machines. The input spaces were examined and randomly generated. We choose .99 as our confidence coefficient from Table 5-2. Five hundred inputs were used for both subroutines. Subroutine one has two input variables, CURENT and REQEST. It returns one value and does no error checking. The input range for both variables is $0 \leq X \leq 360$.

Out of 500 units none were found to cause the subroutine to fail. Therefore the operational reliability is 100%. The lower bound for the module from Table 5-2 is 0.99. We can say that this module will perform with a reliability of 99% or greater.

The second subroutine test also revealed no failures in 500 units. The operational reliability and confidence limit are the same as above.

8.3 Parallel Axis Graph

A cleanroom project under the direction of Dr. Jesse H. Poore was completed during a spring quarter seminar at the University of Tennessee by a a nine-member cleanroom team. The purpose of the project was to design, code and test the first increment of a graphics software package using cleanroom methodology. Increment one of the Parallel Axis Graph (PAG) software package was designed in nine weeks using the cleanroom methodology and demonstrated to a small group of faculty and businessmen. This was the first software tested using the methods outlined in this paper.

The project was tested with randomly generated files reflecting a usage environment conceived by the testers. It was assumed that the usage environment would contain only those inputs that were expected by the specification. We do acknowledge that in a real user environment this would be a naive assumption. This test yielded two measurements: (1) the lower bound for the reliability of the program, and (2) the number of faults found in the program and number of runs between these faults.

The first measurement reflects how well the initial program version was designed and yields a measure of goodness for the program's ability to meet specifications. The second measurement supplies information for MTTF predictions in the certification model and determines if the software is of cleanroom quality. The purpose is to find the most serious errors in the current increment and to find a final reliability measure from which a warranty statement can be issued.

8.3.1 Generation

All vector elements were generated as they naturally occurred in the input domain. The VECTOR and MATRIX elements depend on N and are generated after the N value has occurred. A random variable M is introduced to control the row size of MATRIX. Figure 8-1 shows the input domain definition.

8.3.2 Distributions

Element N was generated with a Gamma distribution. The Gamma distribution is bounded on the left side and is infinite on the right. It is thought that usage of this element will be heaviest at $N = 20$. Therefore, the η and λ parameters are 20 and 1, respectively. The distribution is artificially bounded on the right at 100. The M element is also represented by a Gamma distribution for similar reasons, and the parameters for η and λ are 10 and 1 respectively with an artificial boundary of 50 on the right.

All real values in element MATRIX will be generated with a uniform distribution and will have single precision. A discrete uniform distribution is also

Vector: < N; VECTOR; MATRIX >

1. N: rules A,C,G
2. VECTOR: $< V_1, V_2, V_3, V_4 >$
3. MATRIX: $M_{11}M_{12}M_{13}...M_{1N}$

.

.

.

.

$M_{M1}M_{M2}.....M_{MN}$

4. $V_1...V_4$: rules A,E,G
5. $M_{M1}...M_{MN}$: rules B,D,F,G

Rule A: Integer

Rule B: Real

Rule C: 0 <X≤ 100

Rule D: 0 <M≤ 50

Rule E: $0 \leq X \leq N$

Rule F: $-\infty \leq X \leq +\infty$

Rule G: Followed by white space

Figure 8-1: Input Definition for Increment One.

used for each VECTOR element with the upper bound being the previously generated N.

8.3.3 Sample Size and Lower Bound

Sample sizes are based on a chosen significance level before the test begins. The outcome of each trial is either pass or fail, therefore the binomial will be used to estimate sample size and establish a lower bound.

8.3.4 Algorithm for Testing

Two measurements are sought from the statistical test: (1) a lower bound for the reliability of the final version, and (2) a MTTF prediction from the certification model. The algorithm is the same as that presented in Chapter 6. The stopping criterion will be based on the 3 faults per 1000 lines of code given for cleanroom software by Mills and Poore. The PAG program has about 700 lines of code, so 3 is the stopping criterion. From Table 5-2 we choose a confidence coefficient of 0.99 and a sample size of 100.

8.3.5 Oracle

The oracle for the PAG software is of two forms: (1) an existing, well-established software package (STATGRAPHICS) verifies the maximum, minimum, and midpoint of each test case and generates a graph; and (2) seminar participants visually verify the output.

8.3.6 Test Results

The software was tested on a PC and linked with a TURBO C graphics library. On the sixth trial a failure was observed. The minimum and maximum values were being written over the axis labels. The error was corrected by a design team member and the testing continued for an additional 102 trials with no observed failures. It is interesting to note that the one error discovered was introduced during the code walkthrough.

Version one of the software fails to reach 100 units with no failures; a failure is recorded on trial 6. Version two, however, passes the 100 units with no observed failures, and the testers continue for another two trials to finish the set of test cases. The software is cleanroom quality because only one engineering change was needed. The following warranty statement can be issued:

> Increment One of PAG was developed using cleanroom methodology techniques and meets the quality standards of cleanroom software. We are 99% confident that the software will perform reliably at or above the 95% level. The certification model[63] was used to produce a certified MTTF estimate based upon the observed failure data. In order to include in the MTTF calculation the 102 cases that were successfully run after the failure was observed during the sixth run, the times (runs) between failure input to the certification model were 5 and 102. Based upon the above failure data, the certified MTTF estimate calculated using the certification model is 2081 runs.

In effect, the choice of 102 for the time to the second failure assumes that a failure would occur during the next run. Therefore, the MTTF estimate is actually a worst-case estimate. Obviously, if more cases were successfully run before a second failure was detected, then the MTTF estimate would be higher.

---

[63] Currit et al., pp. 3–11.

BIBLIOGRAPHY

BIBLIOGRAPHY

Brown, A. R., and W. A. Sampson. *Program Debugging*. London: Balding & Mansell Ltd, 1973.

Burden, Richard L., Douglas J. Faires, and Albert C. Reynolds. *Numerical Analysis*. Boston, Massachusetts: Prindle, Weber & Schmidt, 1981.

Burkhardt, W. H. "Generating Programs from Syntax." *Computing*, January 1967, pp. 53-73.

Char, Bruce W., K. O. Geddes, G. H. Gonnet, and S. M. Watt. *Maple User's Manual*. 5^{th} ed. Waterloo, Ontario, Canada: WATCOM Publications Ltd., 1988.

Cho, Chin-Kuei. *Quality Programming: Developing and Testing Software with Statistical Quality Control*. New York: John Wiley & Sons, 1987.

Cobb, Kim C. "Selection of a Software Reliability Model Based Upon Failure Data Characteristics." Masters Thesis. Univ. of Tenn., 1988.

Cochran, William G. *Sampling Techniques*. New York: John Wiley & Sons, 1977.

Currit, P. Allen, Michael Dyer, and Harlan D. Mills. "Certifying the Reliability of Software." *IEEE Transactions on Software Engineering*, January 1986, pp. 3–11.

DeMillo, Richard A., W. Michael McCracken, R. J. Martin, and John F. Passafiume. Menlo Park, California: The Benjamin/Cummings Publishing Company, Inc., 1987.

Deutsch, Michael S. *Software Verification and Validation: Realistic Project Approaches*. Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1982.

Duncan, A. J. *Quality Control and Industrial Statistics*. Chicago: Richard D. Irwin, Inc., 1965.

Duran, Joe W., and Simeon C. Ntafos. "An Evaluation of Random Testing." *IEEE Transactions on Software Engineering*, July 1984, pp. 438–444.

Duran, Joe W., and John J. Wiorkowski. "Quantifying Software Validity by Sampling." *IEEE Transactions on Reliability*, June 1980, pp. 141–144.

Freund, John E., and Ronald E. Walpole. *Mathematical Statistics*. New Jersey: Prentice-Hall, Inc., 1980.

Hanford, K. V. "Automatic Generation of Test Cases." *IBM Systems Journal*, 9 (April 1979), pp. 242–257.

Hahn, Gerald J., and Samuel S. Shapiro. *Statistical Models in Engineering*. New York: John Wiley and Sons, 1967.

Howden, William E. "An Evaluation of the Effectiveness of Symbolic Testing." *Software—Practice and Experience*, 8 (1978), pp. 381–397.

Huang, J. C. "An Approach to Program Testing." *Computing Surveys*, September 1975, pp. 113–128.

Ince, D. C. "The Automatic Generation of Test Data." *Computer Journal*, 30 (February 1987), pp. 63–69.

Johnson, Norman L., and Samuel Kotz. *Discrete Distributions*. New York: Houghton Mifflin Co., 1969.

Kendall, Maurice, and Alan Stuart. *The Advanced Theory of Statistics*. New York: Macmillan Publishing Co., Inc., 1977.

Madsen, Richard W., and James E. Holstein. "Determining Sample Size When Searching for Rare Items." *IEEE Transactions on Reliability*, December 1982, pp. 451–454.

Mann, Nancy R., Ray. E. Schafer, and Nozer D. Sinpurwalla. *Methods for Statistical Analysis of Reliability and Life Data*. New York: John Wiley and Sons, 1974.

Martz, Harry F., and Ray A. Waller. *Bayesian Reliability Analysis*. New York: John Wiley & Sons, 1982.

Mills, Harlan D., Michael Dyer, and Richard C. Linger. "Cleanroom Software Engineering." *IEEE Transactions on Software Engineering*, September 1987, pp. 19–25.

Mills, Harlan D., Richard C. Linger, and Alan R. Hevner. *Principles of Information Systems Analysis and Design.* New York: Harcourt Brace Jovanovich, 1986.

Mills, Harlan D., and Jesse H. Poore. "Bringing Software Under Statistical Quality Control." *Quality Progress*, 1988. In publication.

Mosteller, Frederick, Robert E. K. Rourke, and George B. Thomas, Jr. *Probability with Statistical Applications.* Reading, Massachusetts: Addison–Wesley, 1970.

Myers, Glenford J. *The Art of Software Testing.* New York: John Wiley & Sons, 1979.

NAG, Mark 10. University of Liverpool.

Naur, Peter. *Concise Survey of Computer Methods.* Sweden: Studentlitteratur Lund Akademisk Forlag Kobenhavn, 1974.

Nelson, Eldred. "Estimating Software Reliability from Test Data." *Microelectronics and Reliability*, 17 (1978), pp. 67–73.

Poore, Jesse H. "Derivation of Local Software Quality Metrics." *Software—Practice and Experience*, 1988. In publication.

Ramamoorthy, C. V., SIU–BUN F. Ho, and W. T. Chen. "On the Automated Generation of Program Test Data." *IEEE Transactions on Software Engineering*, December 1976, pp. 293–300.

Ramamoorthy, C. V., and F. B. Bastani. "Software Reliability—Status and Perspective." *IEEE Transactions on Software Engineering*, July 1982, pp. 354–370.

Scheaffer, Richard L., William Mendenhall, and Lyman Ott. *Elementary Survey Sampling.* Belmont, California: Wadsworth Publishing Company Inc., 1979.

Schick, George J., and Ray W. Wolverton. "An Analysis of Competing Software Reliability Models." *IEEE Transactions on Software Engineering*, March 1978, pp. 104–220.

STATGRAPHICS, rel. 2.6. STSC.

APPENDIX

APPENDIX

CONTINUOUS DISTRIBUTION PLOTS

This appendix presents graphs of six continuous probability density functions listed in Figure 4-1. The graphs show how the parameters for location, scale, and shape effect each curve. All graphs were generated on a VAX using FORTRAN programs and a Hewlett-Packard 7475a plotter.

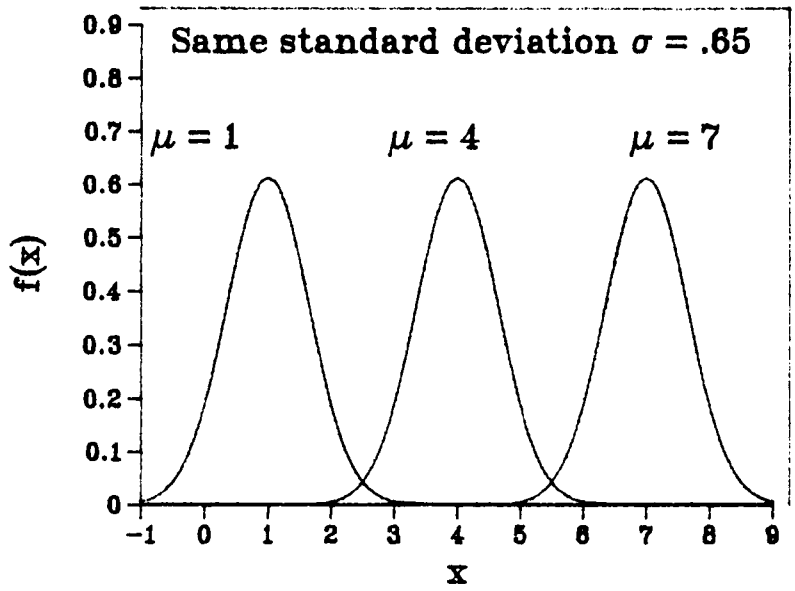


Figure A-1: Normal Distributions with the Same σ.

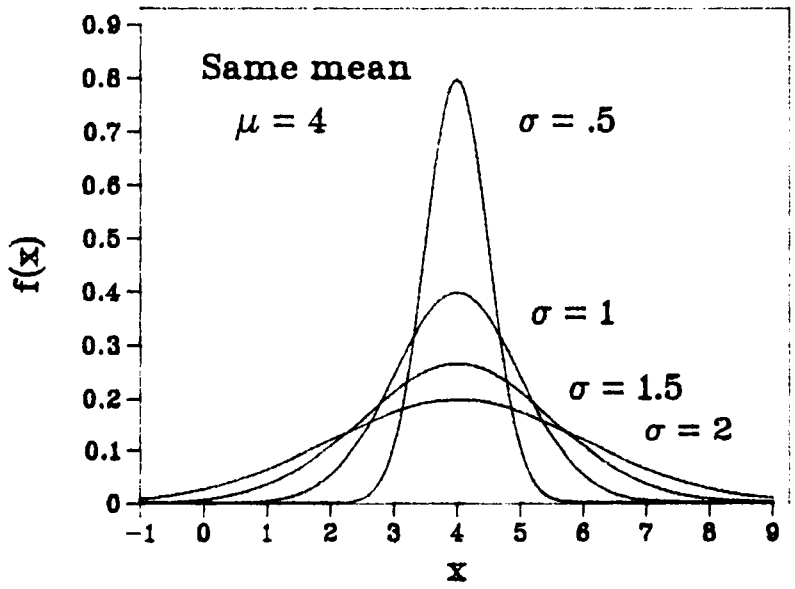


Figure A-2: Normal Distributions with the Same μ.

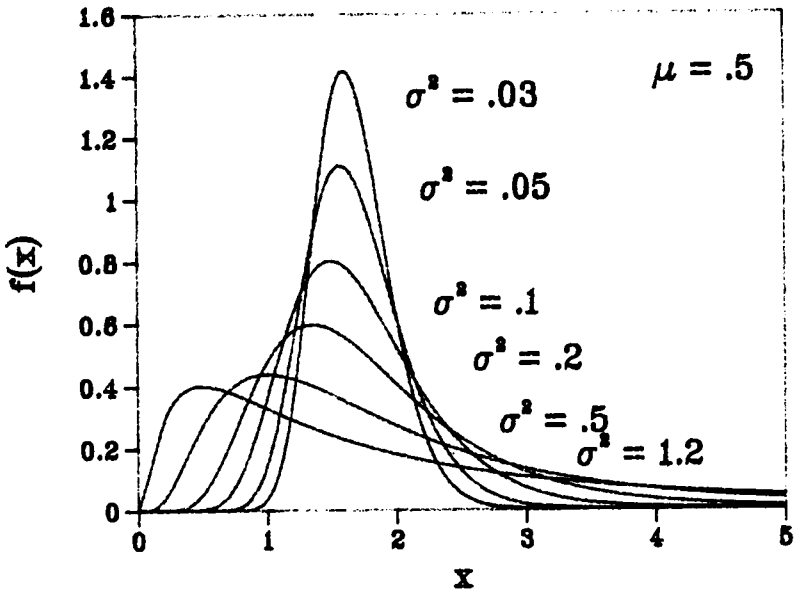


Figure A-3: Log Normal Distribution: All Curves Have the Same μ.

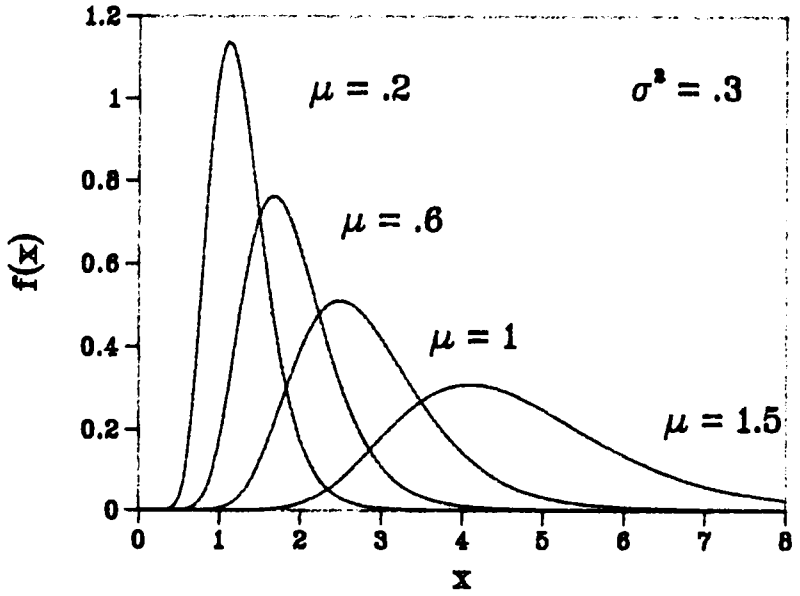


Figure A-4: Log Normal Distribution: All Curves Have the Same σ^2 Values.

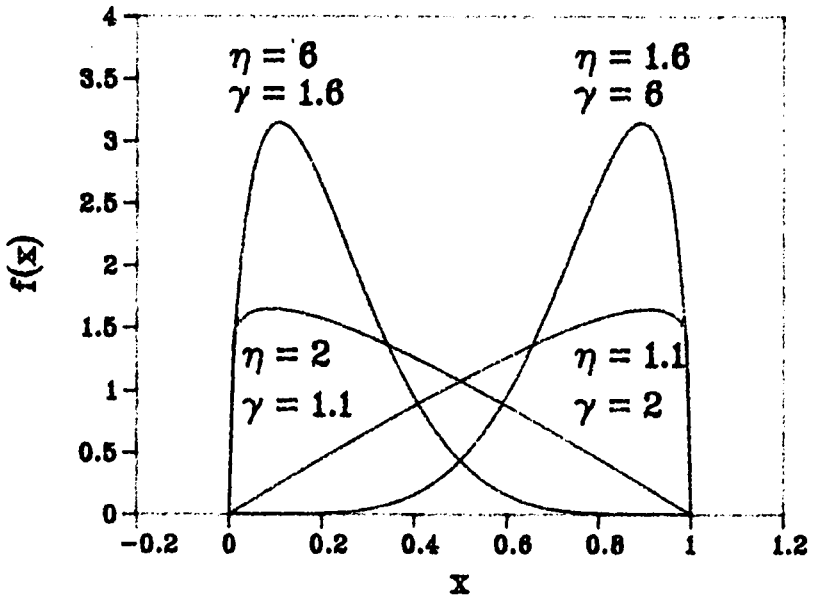


Figure A-5: Beta Distribution: When $\gamma > 1$ and $\eta > 1$ the Distribution Has a Single Peak.

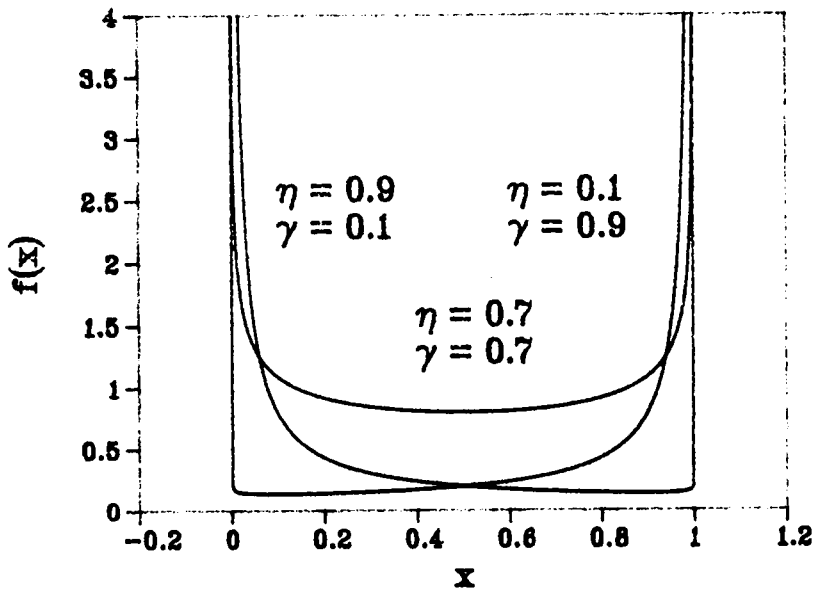


Figure A-6: Beta Distribution: When $\gamma < 1$ and $\eta < 1$ the Distribution is U Shaped.

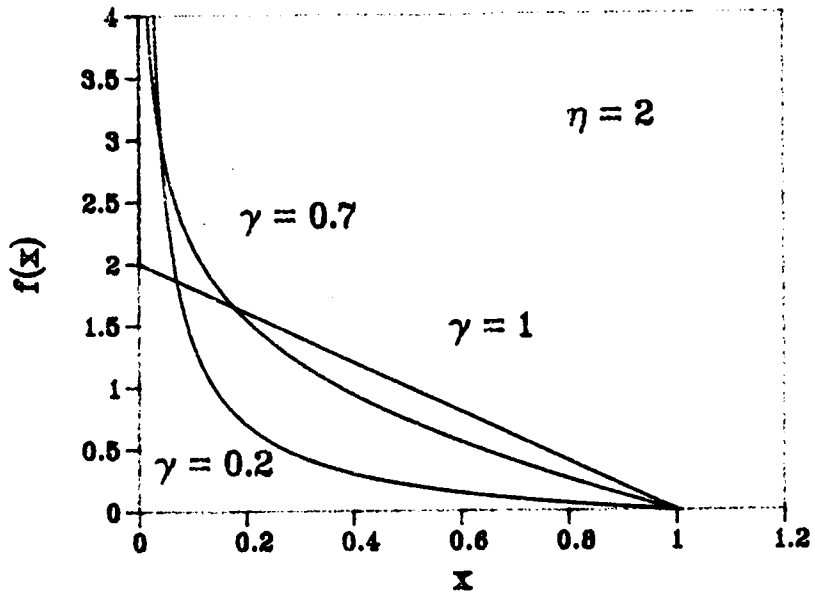


Figure A-7: Beta Distribution: When $\gamma < 1$ and $\eta \geq 1$ the Distribution is Reverse J Shaped; When $\eta < 1$ and $\gamma \geq 1$, it is J Shaped.

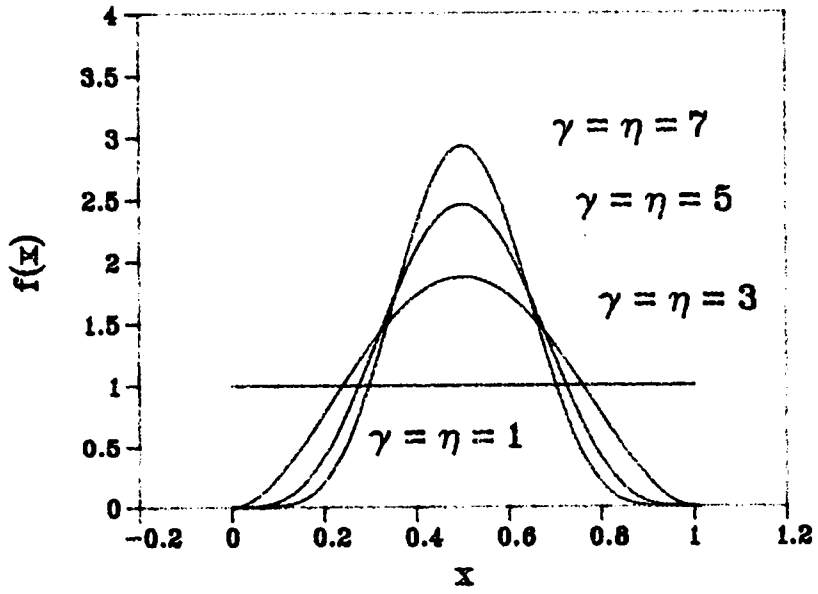


Figure A-8: Beta Distribution: When $\gamma = \eta$ the Distribution is Symmetrical.

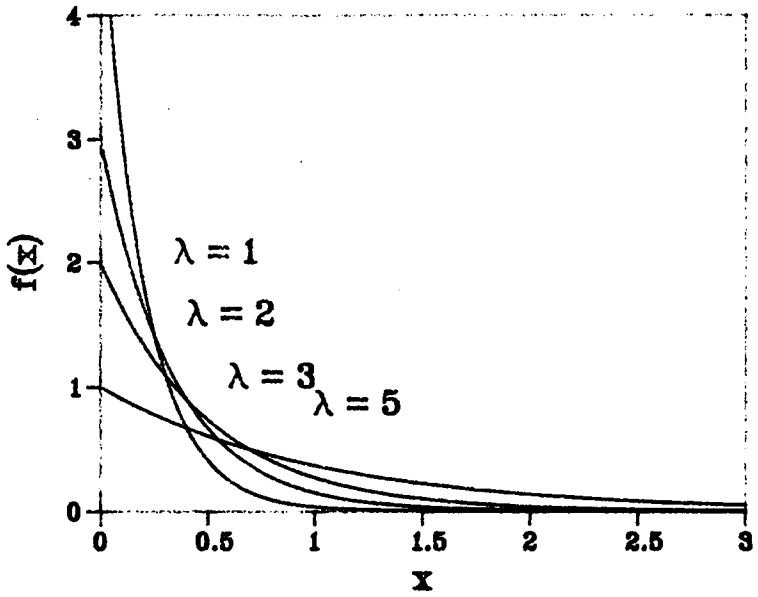


Figure A-9: Exponential Distribution: A Special Case of the Gamma Distribution.

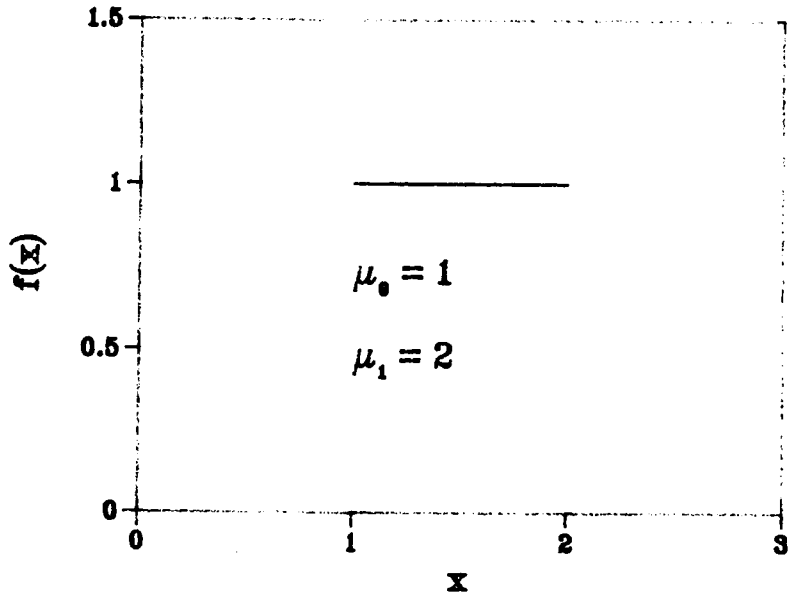


Figure A-10: Uniform Distribution: A Special Case of the Beta Distribution.

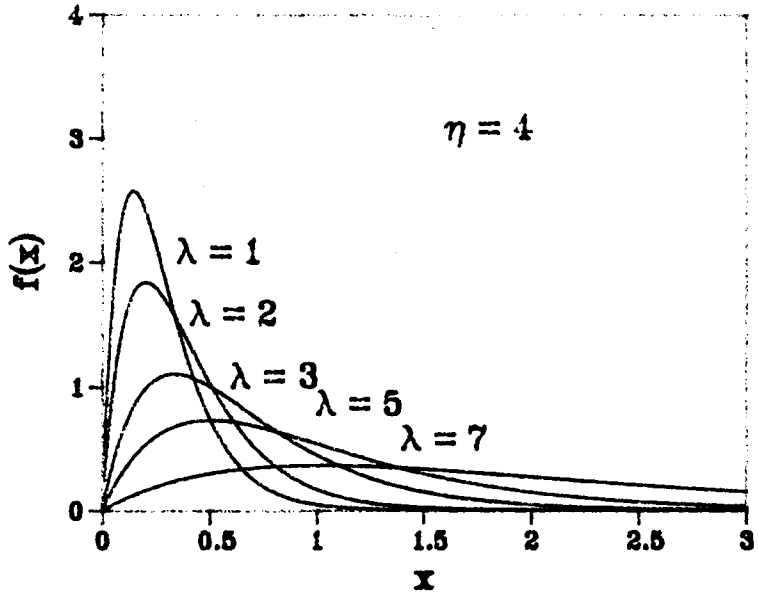


Figure A-11: Gamma Distribution: Same η Values.

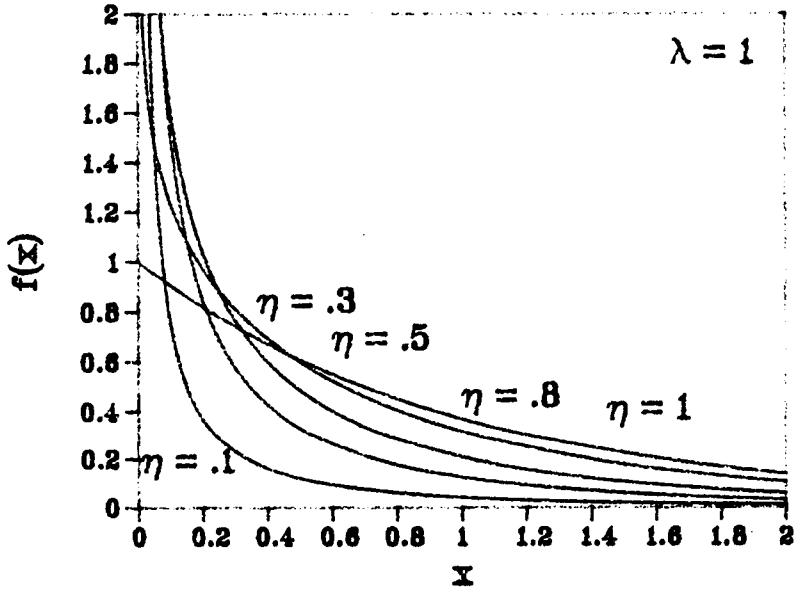


Figure A-12: Gamma Distribution: Same λ Values.

VITA

Bonnie Collett Sexton was born in Louisville, Kentucky on April 7, 1957. She graduated from Valley High School in Valley Station, Kentucky in June 1975. The following September she entered Western Kentucky University in Bowling Green and in June 1979 graduated with a Bachelor of Arts degree in Music. In June 1984 she entered East Tennessee State University; two years later she graduated with a Bachelor of Science degree in Computer Science. After graduation from ETSU, she accepted a position as a computing analyst with Martin Marietta Energy Systems, Inc. in Oak Ridge, Tennessee.

Ms. Sexton entered the Graduate School of the University of Tennessee, Knoxville in January 1985. She received the Master of Science degree in Computer Science in December 1988. Ms. Sexton is a member of the Association for Computing Machinery and presently makes her home in Knoxville, Tennessee.