Annual Progress Report

Award No. NAG-1-605

DETECTION OF FAULTS AND SOFTWARE RELIABILITY ANALYSIS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

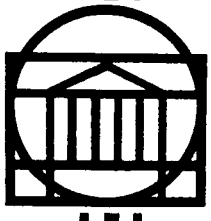
Attention: Mr. Gerald E. Migneault
FCSD M/S 130

Submitted by:

J. C. Knight
Associate Professor

Report No. UVA/528243/CS87/101

August 1986



SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA 22901

Annual Progress Report

Award No. NAG-1-605

DETECTION OF FAULTS AND SOFTWARE RELIABILITY ANALYSIS

Submitted to:

National Aeronautics and Space Aministration
Langley Research Center
Hampton, VA 23665

Attention: Mr. Gerald E. Migneault
FCSD M/S 130

Submitted by:

J. C. Knight
Associate Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCES
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/528243/CS87/101

August 1986

Copy No. _____

TABLE OF CONTENTS

PRECEDING PAGE BLANK NOT FILMED

SECTION I

INTRODUCTION

The work being carried out under this grant is an investigation of software faults. The goal is to better understand their characteristics and to apply this understanding to the software development process for crucial applications in an effort to improve software reliability. Some of the work is empirical and some analytic. The empirical work is based on the results of the Knight and Leveson experiment [1] on *N*-version programming. The analytic work is attempting to build useful models of certain aspects of the software development process.

Multi-version or *N*-version programming [2] has been proposed as a method of providing fault tolerance in software. The approach requires the separate, independent preparation of multiple (i.e. "*N*") versions of a piece of software for some application. These versions are executed in parallel in the application environment; each receives identical inputs and each produces its version of the required outputs. The outputs are collected by a voter and, in principle, they should all be the same. In practice there may be some disagreement. If this occurs, the results of the majority (assuming there is one) are taken to be the correct output, and this is the output used by the system.

The major experiment carried out by Knight and Leveson was designed to study *N*-version programming and initially investigated the assumption of independence. In the experiment, students in graduate and senior level

classes in computer science at the University of Virginia (UVA) and the University of California at Irvine (UCI), were asked to write programs from a single requirements specification. The result was a total of twenty-seven programs (nine from UVA and eighteen from UCI) all of which should produce the same output from the same input. Each of these programs was then subjected to one million randomly-generated test cases. The Knight and Leveson experiment has yielded a number of programs containing faults that are useful for general studies of software reliability as well as studies of *N*-version programming.

Our work has been in a number of areas and each area is covered separately in this report. The specific topics are:

(1) an empirical study of failure probabilities in *N*-version systems,

(2) consistent comparison in *N*-version systems,

(3) descriptions of the faults found in the Knight and Leveson experiment,

(4) analytic models of comparison testing,

(5) characteristics of the input regions that trigger faults,

(6) fault tolerance through data diversity,

(7) and the relationship between failures caused by automatically seeded faults.

In most areas, the report provided here is quite brief since the details of the research have been reported in published or submitted papers. These papers have been supplied to the sponsor separately and are merely referenced here.

SECTION II

FAILURE PROBABILITIES

Using the results of the tests performed in the Knight and Leveson experiment, we have shown that the performance of multi-version systems produced from the twenty-seven programs achieve a substantial reduction in failure probability. Thus although the faults contained in the programs were responsible for many coincident failures, there was still a substantial benefit gained from using a multi-version structure.

The study of failure probabilities considered both two and three version systems. Two version systems are important because they are being used in production for error detection. We found that the two version systems simulated from the available programs were able to correctly detect errors when they occurred with a probability of approximately 0.995.

Three version systems provide fault tolerance as well as error detection. For the programs in the sample, we observed a reduction in failure probabilities of approximately one order of magnitude.

This work was reported at the Sixteenth International Symposium on Fault-Tolerant Computing [3].

SECTION III

CONSISTENT COMPARISON

We have identified a difficulty in the implementation of *N*-version programming. The problem, which we call the *Consistent Comparison Problem*, arises for applications in which decisions are based on the results of comparisons of finite-precision numbers. We have shown that when versions make comparisons involving the results of finite-precision calculations, it is impossible to guarantee the consistency of their results. It is therefore possible that correct versions may arrive at completely different outputs for an application which does not apparently have multiple correct solutions. There is no solution to the Consistent Comparison Problem and we have been able to find only one technique for avoiding it. If this problem is not dealt with explicitly, an *N*-version system may be unable to reach a consensus even when none of its component versions fail.

A paper describing this work has been submitted to the IEEE Transactions on Software Engineering [4].

SECTION IV

FAULT DESCRIPTIONS

Partly under this grant, we have documented the details of the faults in the individual programs that were revealed by the testing [5]. A total of forty-five faults were identified in the twenty-seven programs. We have also analyzed the interaction of each fault with each other fault, and shown that there are a large number of fault pairs that exhibit statistically-correlated behavior. Since we now know the details of each fault, we have been able to examine the faults that exhibit correlated behavior. We were surprised to discover that in many cases there was no obvious similarity between faults that exhibited correlated behavior. In practice, various different and apparently unrelated faults were triggered by the same special situation in the input. It is sensitivity to special cases in the input that causes coincident failures and this sensitivity appears to be present in unrelated faults. None of the faults that we observed was attributable to any aspect of the development environment. We concluded that, for the particular application used in the experiment, there was no obvious change that could be made in the environment that would reduce the incidence of statistically-correlated faults.

The detailed descriptions of the faults and the analysis of their interactions is being prepared as a paper for submission to the IEEE Transactions on Software Engineering.

SECTION V

COMPARISON TESTING

A common argument [6] in favor of at least dual programming (i.e. *N*-version programming with $N = 2$) is that testing of safety-critical real-time software can be simplified by producing two versions of the software and executing them on large numbers of test cases without manual or independent verification of the correct output. The output is assumed correct as long as both versions of the programs agree. The argument is made that preparing test data and determining correct output is difficult and expensive for much real-time software. Since it is assumed "unlikely" that two programs will contain identical faults, a large number of test cases can be run in a relatively short time and with a large reduction in effort required for validation of test results. We refer to this approach as *comparison testing* although it is also known as back-to-back testing in the literature.

Comparison testing has been criticized on the grounds that it tends to reveal only those faults where the programs generate different outputs. Such faults are inconvenient but not dangerous to an *N*-version system since they will be detected and tolerated. Comparison testing will not reveal faults that cause identical wrong outputs and it is precisely these that will not be tolerated.

We have found that comparison testing is a very useful and cost-effective method of fault elimination in multi-version systems. The reason is that although two faults in different programs may cause coincident failures,

our experience has been that such faults do not *always* cause coincident failures. Thus there are occasions when only one of the two programs will fail allowing comparison testing to detect the situation.

We have begun to analyze the performance of comparison testing. Our approach uses Markov models of the fault location process. The models associate states with the number of located faults and the order in which they are found. Transition probabilities between states are just the probabilities of finding particular faults on each test case. The expected number of tests to locate each fault even for faults that cause coincident failures can be determined from such models. The preliminary model shows that comparison testing is remarkably effective.

This work is incomplete but the initial simple models have been documented in a PhD dissertation proposal [7]. This document has been supplied separately to the sponsor.

SECTION VI

INPUT REGION CHARACTERISTICS

Important information for modeling program errors is the shape and size of the region of the input space that the faulty program maps incorrectly to the output space. We term such a region of the input space an *error region* or *failure crystal.* In general, it is difficult to represent and display failure crystals since the dimensionality of the input space may be quite large. However, two-dimensional cross sections of a crystal developed using a uniform grid are easy to obtain. For several of the faults identified in the Knight and Leveson experiment we have obtained such two-dimensional cross sections.

The following graphs represent two-dimensional slices of a region of the input space for the Launch Interceptor problem [1]. The regions were chosen to include some points from the failure crystal of an error from a faulty Launch Interceptor program; the graphs show the "shape" in two dimensions of a set of points that were mapped incorrectly due to the presence of the error under examination. The dimensions selected for examination correspond to the (x, y) points provided as simulated radar images to the programs. Since each point has two independent dimensions (one each for x and y), the number of available dimensions is twice the number of data points. The graphs shown were chosen because they exhibit interesting shapes of the failure regions.

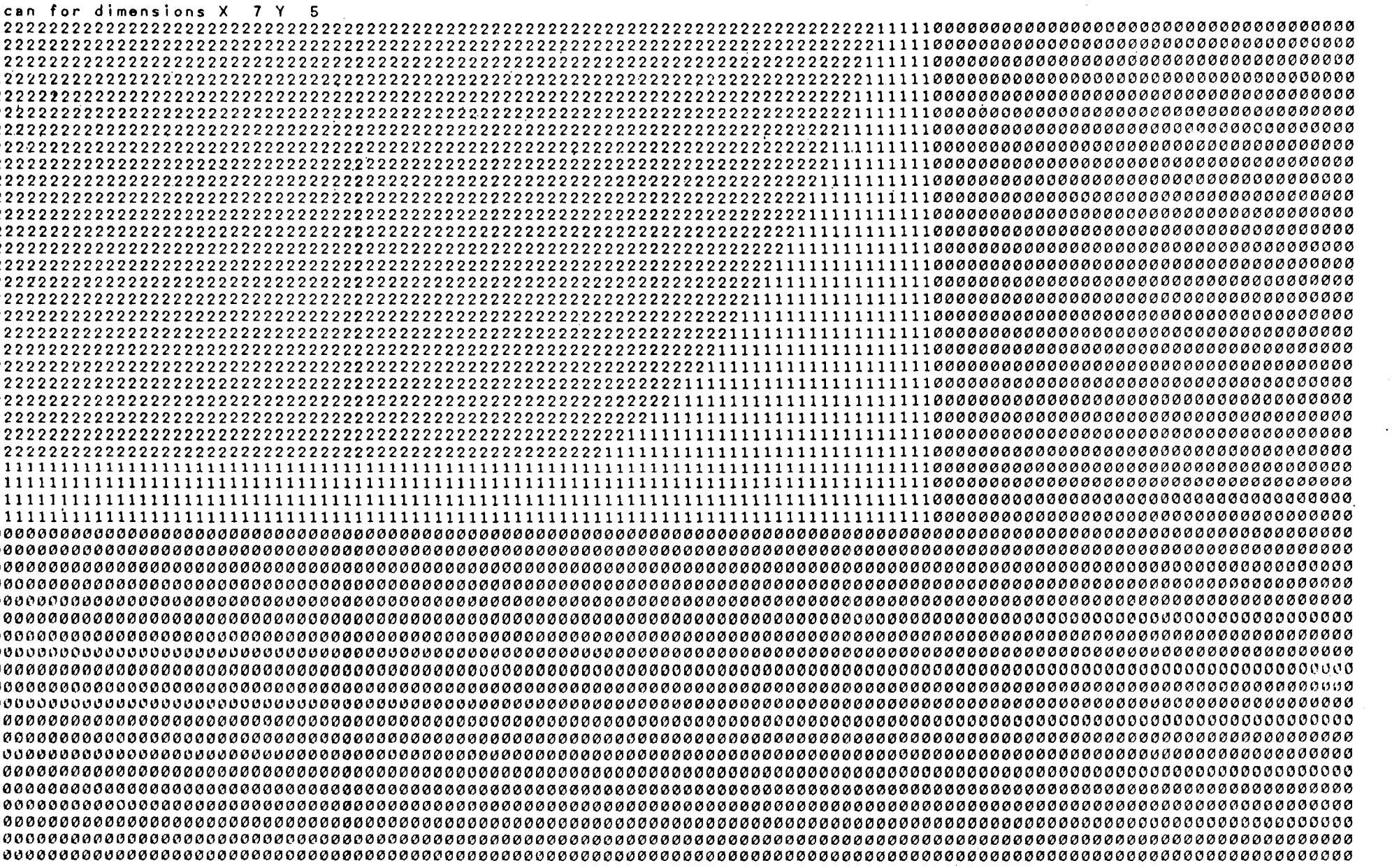
Each graph was built by forming a uniform grid in two dimensions centered around an initial starting point in the input space. The initial starting point was a failure point picked using data from the Launch Interceptor Experiment. Because both failure information for a program and the mapping of the input space to the output space are of interest, the graphs show two bits of information per grid point. The first bit describes whether the gold program and the faulty version agree at a grid point. The second bit describes whether the gold program gets the same output as it did on the initial data point; it illustrates the mapping of the input space to different regions of the output space. Specifically, a "0" in the graph means that the faulty version succeeded and the gold version obtained the same output as on the initial point. A "1" indicates that the faulty version failed and the gold version obtained the same output as on the initial point. A "2" indicates that the faulty version succeeded but that the gold version was in a new region of the output space. Finally a "3" indicates that the faulty version failed and that the gold version was in a new region of the output space.

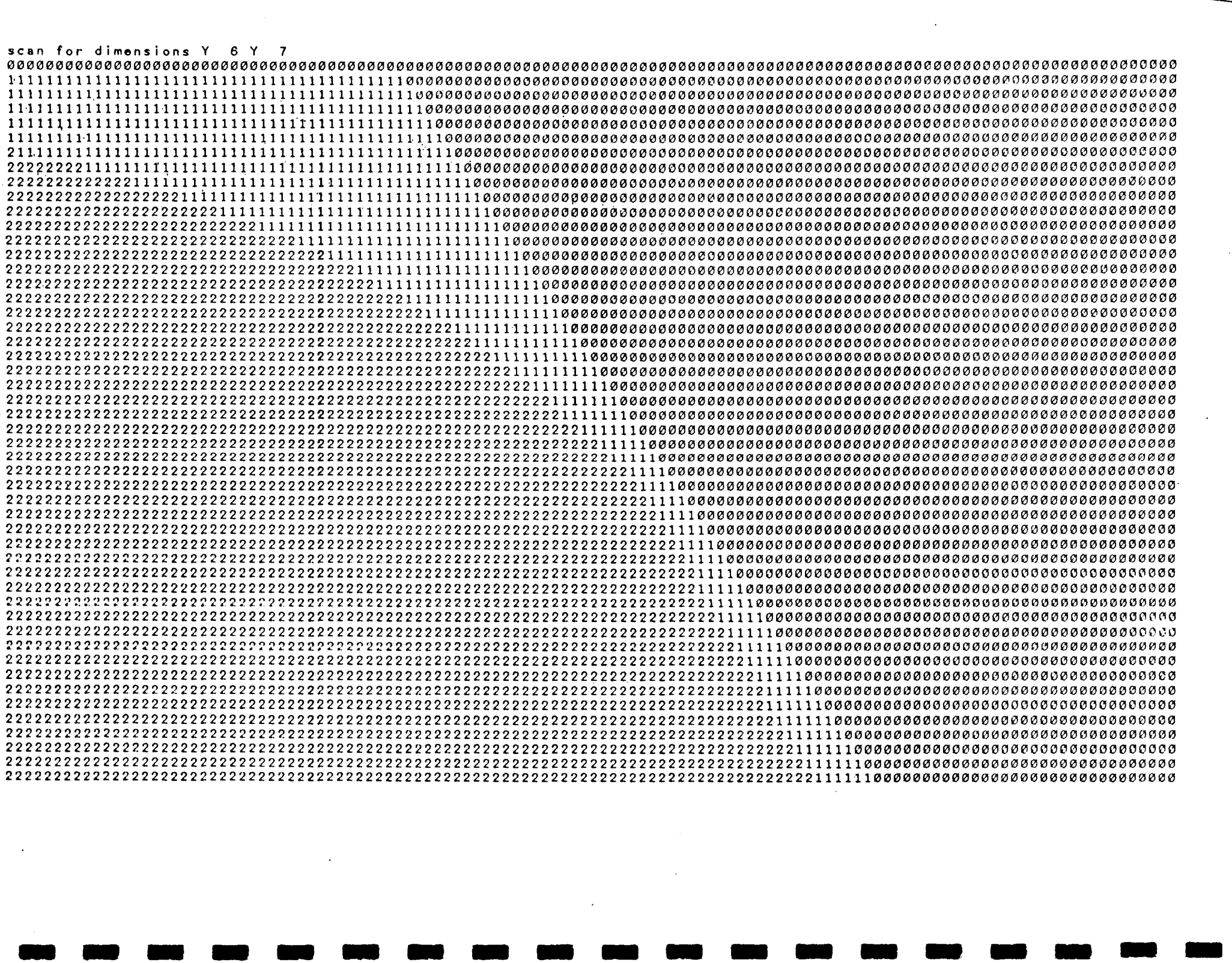
The program errors corresponding to the shown graphs are errors 6.2 and 6.3 from the fault descriptions given in the Launch Interceptor experiment [1]. Error 6.2 is an error in the determination of the size of the smallest circle containing three points. Error 6.3 is an error in which the wrong subscript is given as an array index. The first 4 graphs correspond to different cross sections of a single crystal for error 6.2; the remaining 7 graphs correspond to different cross sections of a single crystal for error 6.3. Note that the first 4 graphs share a common point (the center point), as do

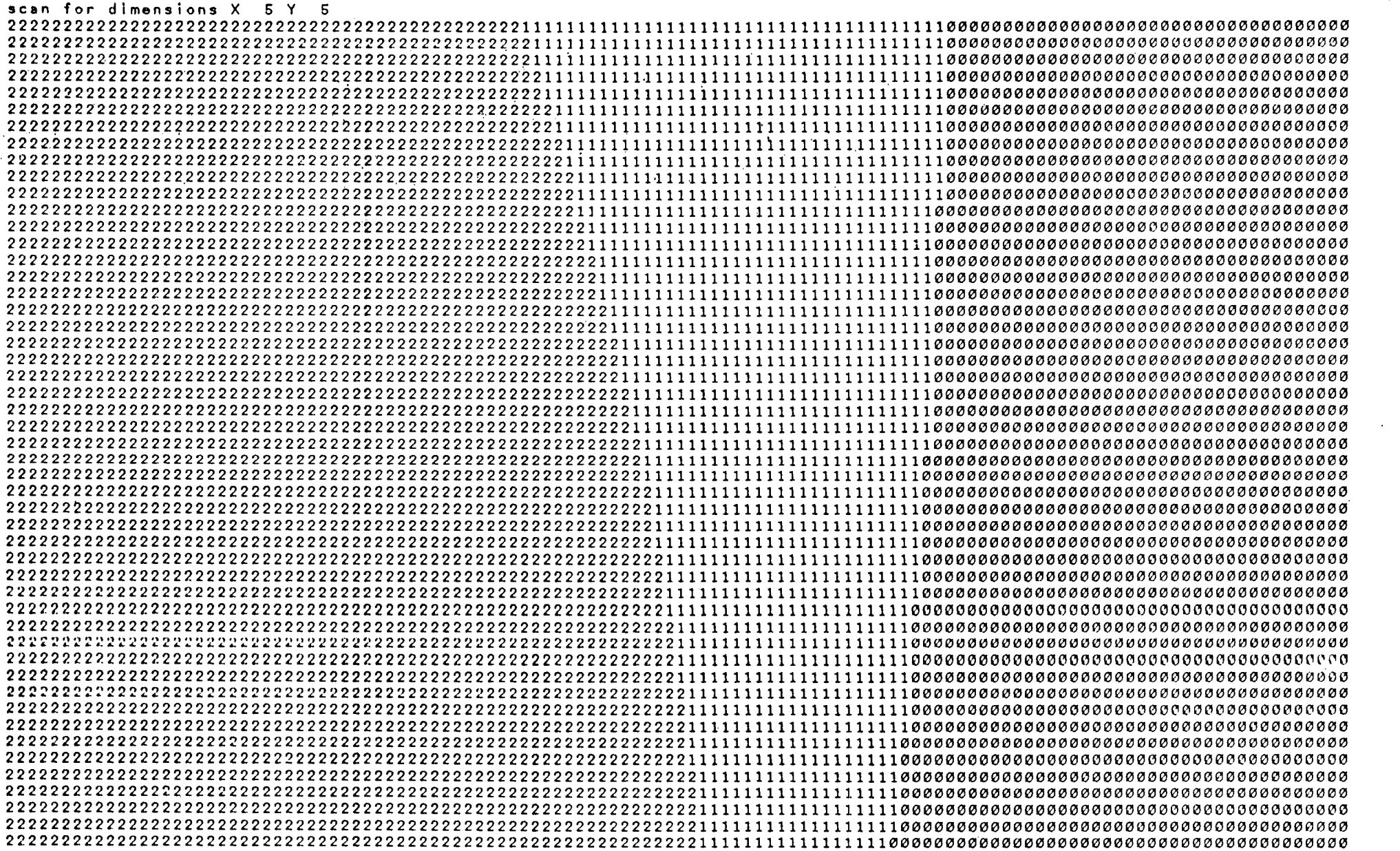
the last 7. The spacing of the grid points is identical in all of the graphs except for the first and the last. Grid points are spaced at 0.2 for the first graph, 0.025 for the middle 9 graphs, and 0.000001 for the final graph.

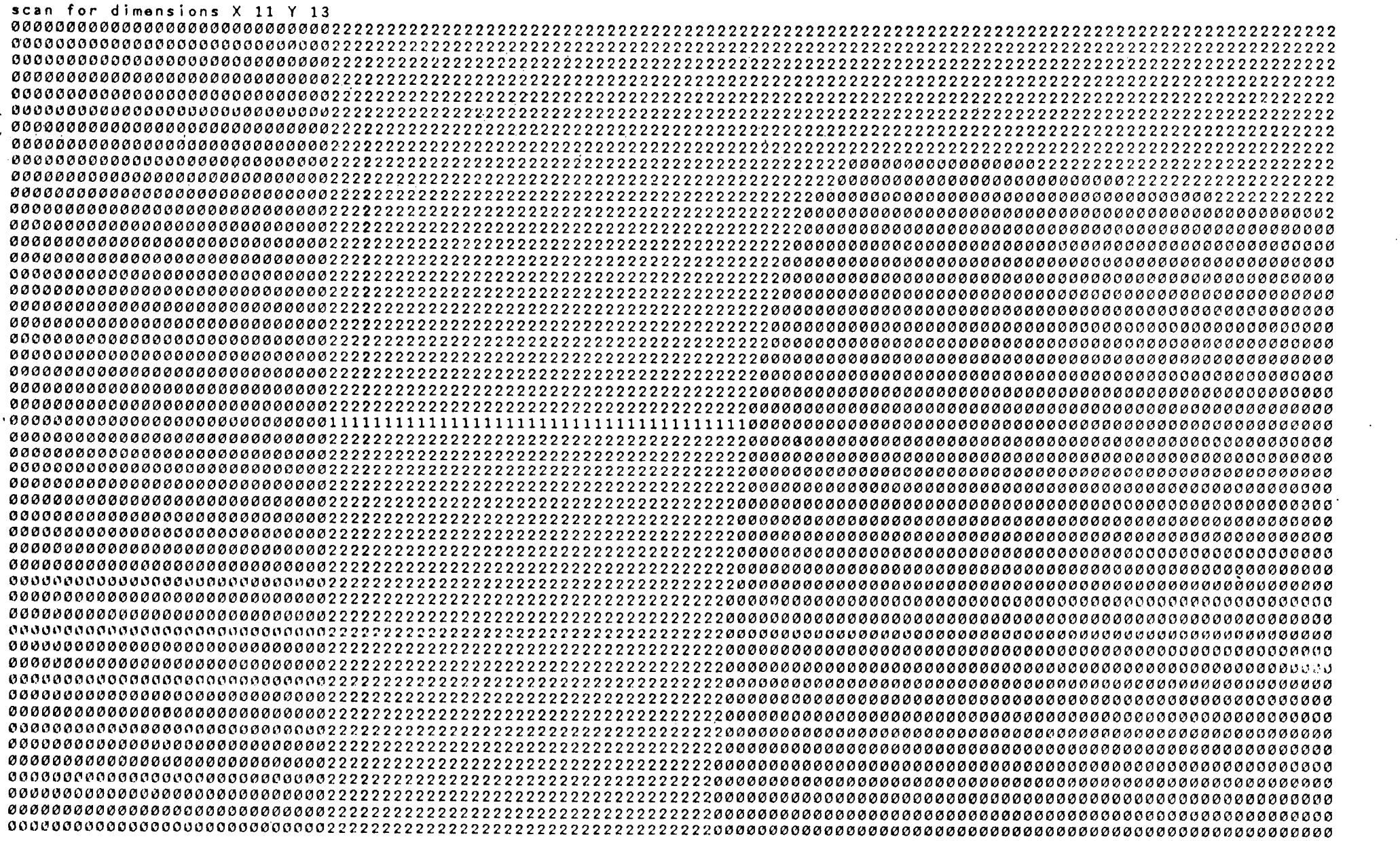

scan for dimensions X 1 X 9

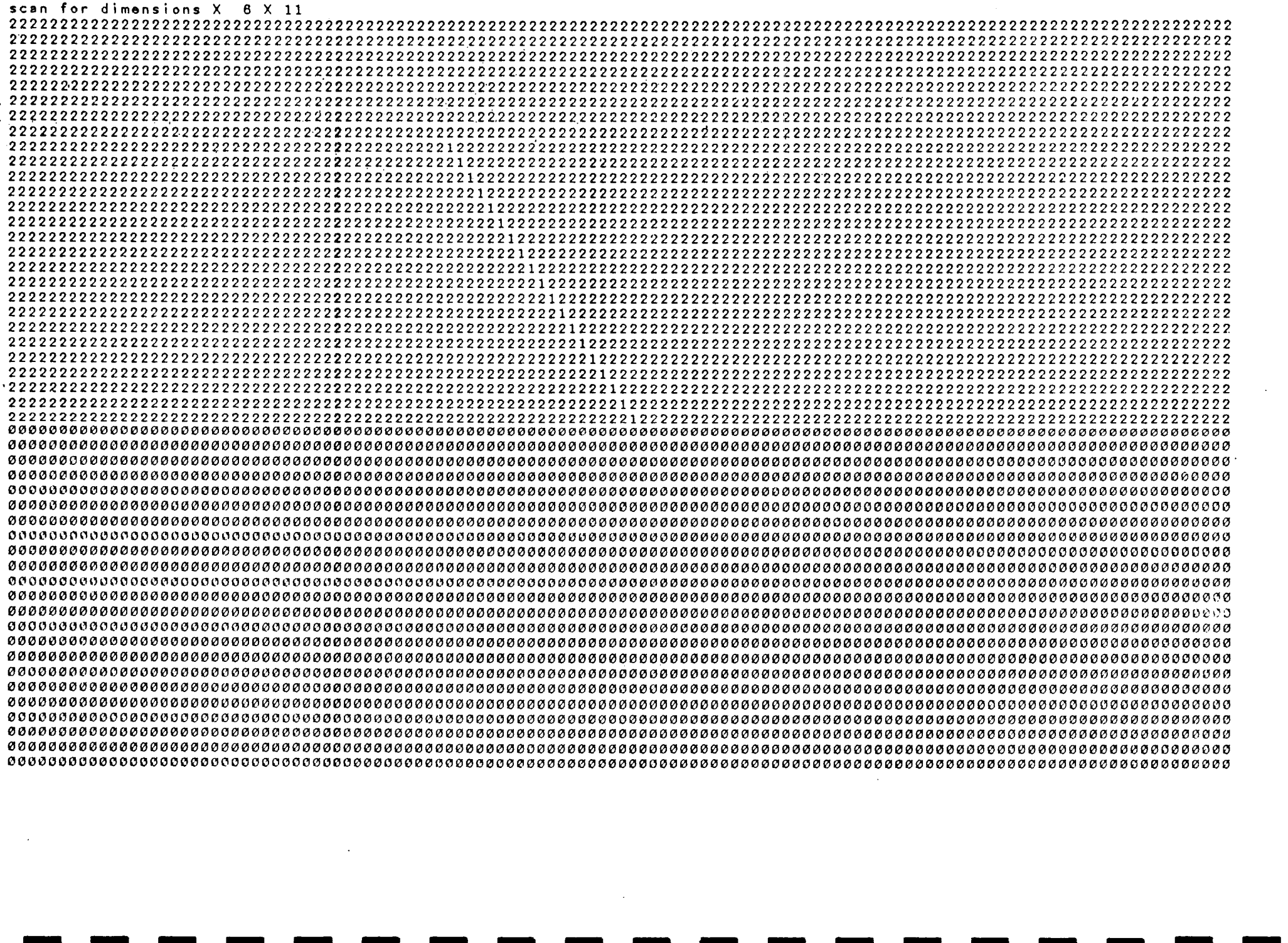
scan for dimensions X 7 Y 5




scan for dimensions Y 6 Y 7

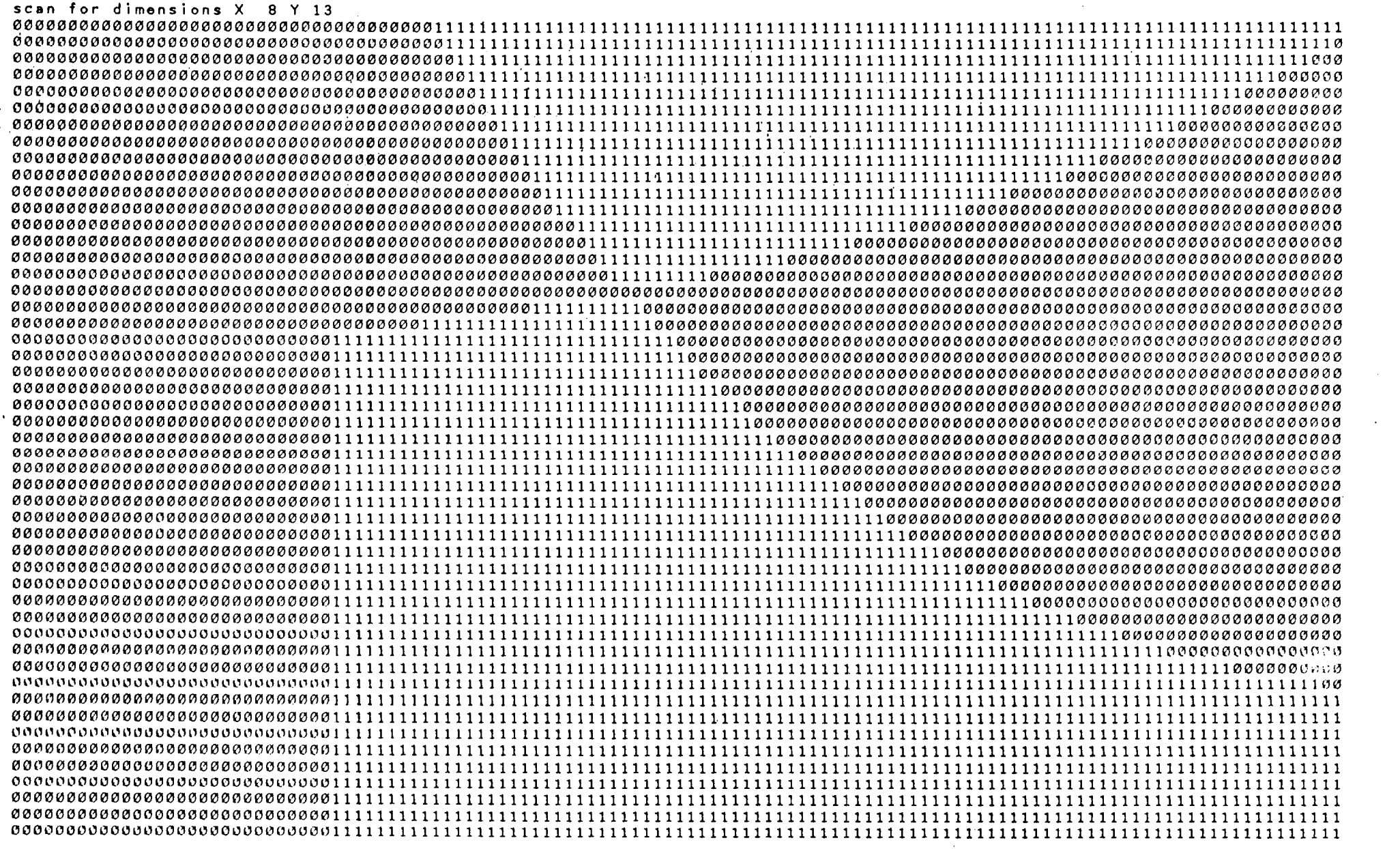

scan for dimensions X 5 Y 5


scan for dimensions X 11 Y 13

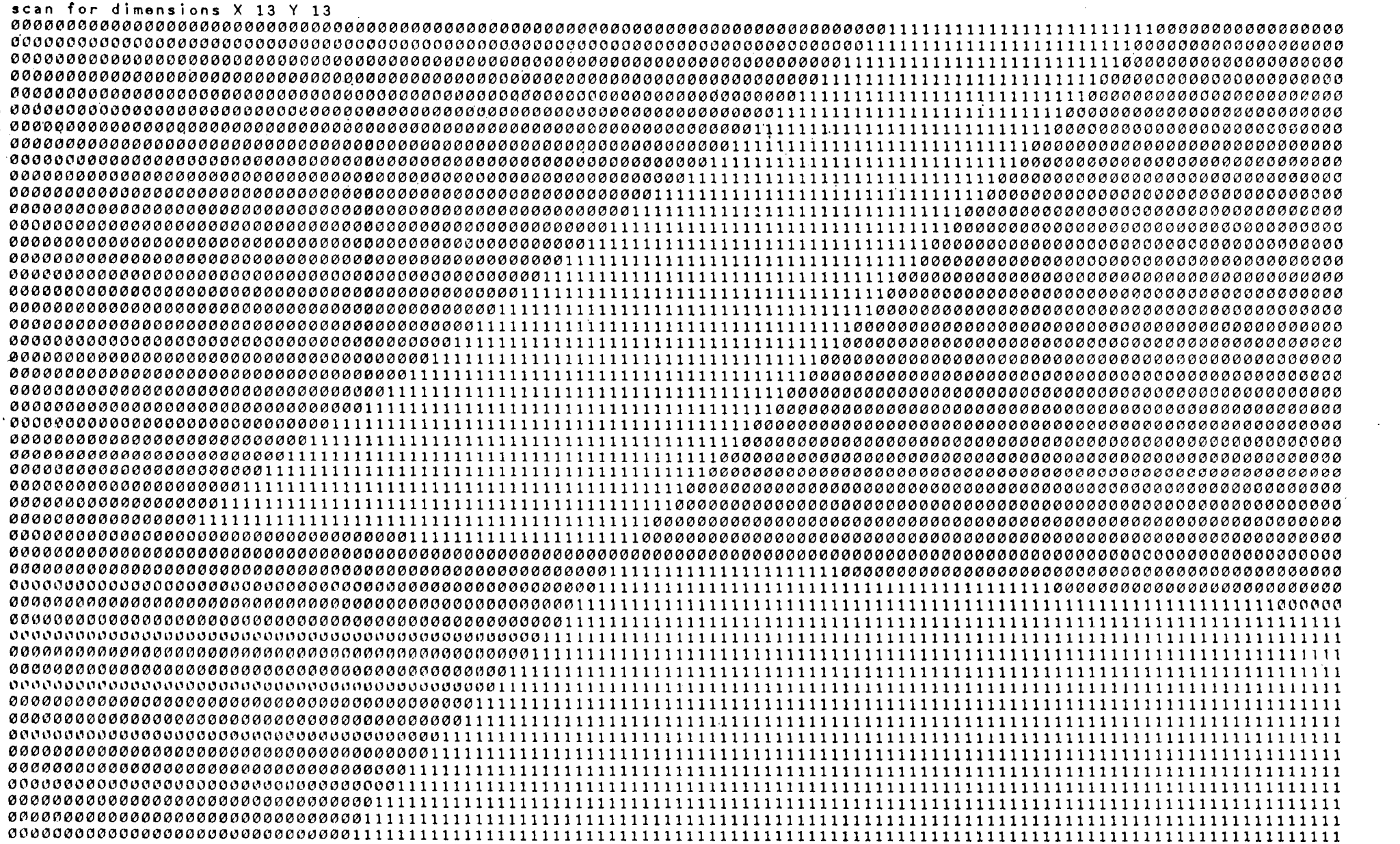

scan for dimensions X 8 X 11

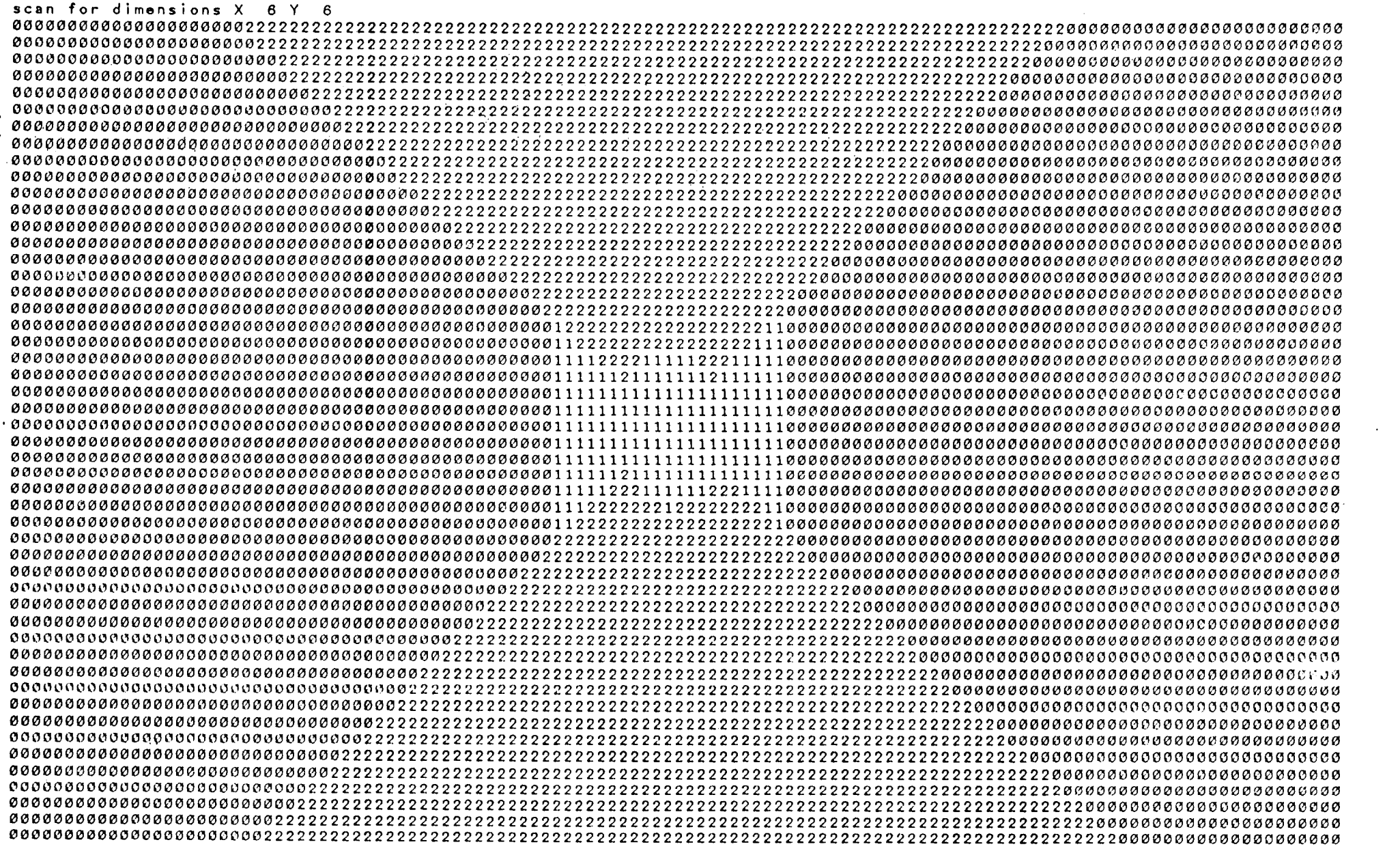

scan for dimensions X 8 Y 11


scan for dimensions X 8 Y 8


scan for dimensions X 8 Y 13

scan for dimensions X 13 Y 13




scan for dimensions X 6 Y 6

SECTION VII

FAULT TOLERANCE THROUGH DATA DIVERSITY

We have proposed a new approach to software fault tolerance that we term *data diversity*. Fault tolerance has been attacked in the past through design diversity. We suggest that it might be achieved through diversity in the data.

The basis of the approach is to execute several copies of a single program but supply each with slightly different data. The idea of executing multiple copies of a single version of software has been rejected by others as pointless. The argument for rejection is that if one copy fails they will all fail. We wonder however whether with slightly different inputs this might not be a useful approach.

In fact, a variant of this approach has been suggested and tried by industrial software developers. Their approach is to use conventional *N*-version programming but to stagger the times at which the versions read sensors so that they will each receive slightly different data values. In practice, it is not necessary or even beneficial to use different versions and it is not necessary to await changes in the data over time. The changes can be computed.

There have been no analyses or experiments performed to evaluate the performance of either the industrial approach or our proposed data diversity. We have begun analytic and simulations studies of both and have very

encouraging but preliminary results.

SECTION VIII

SEEDED FAULTS

In the *N*-version programming method, separate development is intended to eliminate the sharing of (mis)understanding of the application; it associates independence of program failures with mutual isolation of the program designs. However, separate development has no effect on errors unbiased by knowledge of the application. For example, a programmer may inadvertently misorder certain steps in a computation or reverse the use of "and" and "or" in a conditional expression. The important characteristic of these errors is that they are not specific to the application. The separate development process does not affect their introduction. We have examined whether unbiased errors play any role in the expected independence of the resulting programs.

We have adopted an operational definition of independence: failures of two programs are dependent if a statistical measure shows a correlation of incorrect outputs for a given input. That is, programs that fail together significantly more often than expected are considered to contain dependent errors. How dependent errors are introduced does not affect the operational viewpoint of independence[1]. The statistical measure used here, a χ^2 test of an independence hypothesis, is the same as has been applied in [5]. A hypothesis that two programs fail independently is formed and the χ^2

1 We note that other authors use different definitions of independence, for example [8].

statistic is generated. When the hypothesis is rejected with a high confidence level, dependence is assumed.

As part of a separate project we have performed an experiment in error seeding. Seventeen of the twenty-seven programs produced in the Knight and Leveson experiment were selected at random, errors were seeded into all seventeen, and the resulting programs were tested. The algorithms used for seeding errors were very simple: 2 algorithms modified the bounds on for statements, 3 algorithms modified the Boolean expression in if statments, and 1 algorithm deleted assignment statements. Each of these algorithms was applied 4 times to each of the 17 programs for a total of 408 modified programs, each of which contained one seeded error. It should be stressed that the seeded errors were introduced at random without using any semantic knowledge of the program structure. To introduce one seeded error, a syntactic structure was selected at random and the seeding algorithm was applied. The seeded errors are unbiased errors.

To select seeded errors to be investigated for dependent failures a form of acceptance testing was used: seeded errors with a mean time to failure smaller than a certain threshold were disqualified from the experiment. In addition, seeded errors which caused *no* failures during the original error seeding experiment were also disqualified. 45 of the 408 seeded errors passed this acceptance test. Such an acceptance test is equivalent to the original acceptance testing done to admit the launch interceptor programs to the original *N*-version experiment. In this experiment all indigenous errors were fixed before the seeded errors were installed in the programs. Each failure

of a given program is caused only by the seeded error.

The 45 seeded programs were run over a 45,000 test case subset of the 1,000,000 randomly generated test cases used in the N-version experiment [1]. Figure 1 summarizes the results of the test cases. The graph is a 45 by 45 symmetric matrix, the upper half of which is shown. The (i, j) entry in the matrix describes failure on common test cases between program i and program j.

The programs are split into three categories according to the type of seeded error used to generate the program. The divisions are shown by lines on the graph. The first 13 programs contain for statement seeded errors, the next 29 contain if statement seeded errors, and the last 3 programs contain assignment statement seeded errors.

The programs are ordered within the divisions by type so as to make the nonzero elements cluster near the diagonal. The reordering results in "blocks" of entries on the diagonal. Each block indicates that the corresponding programs all fail together on a certain subset of test cases. Note that the reordering has no effect on the entries in the matrix; only the visual appearance has been altered. The reordering is done to show that common failures tend to group together.

A χ^2 test was used to test the hypothesis that each pair of programs fails independently. The results of these tests appear in Figure 1. On the graph, an "R" indicates that the independence hypothesis is rejected with a confidence of 99.5%. An "I" indicates that insufficient data (less than 5

common failures) existed to make the χ^2 test meaningful. An "N" indicates that the independence hypothesis is not rejected. Notice that although the graph does not contain any "N" entries, blank entries, which represent zero common failures, may be interpreted as likely "N" entries. Since all 45 of the seeded programs were produced by seeding errors into 17 original programs, there are entries in the matrix corresponding to two distinct seeded errors being evaluated in the same base program. Since this situation does not reflect accurately a scenario possible in the separate development of programs, these entries are marked in lower case "r", "i", and "n", respectively.

An examination of figure 1 shows that within a category of seeded error, dependence among errors is common. No dependence has been demonstrated among different categories of seeded errors, although the graph suggests that more testing might reveal such a dependence. It is clear that, for this example, dependence among errors of the same category is more likely than dependence among errors of different categories.

Informally, unbiased errors are the "best" errors that can be hoped for in a separate development environment. Unbiased errors model program defects that are free from influence propagated among the development teams. The experiment shows that unbiased errors do not, in general, cause independent failures. Each separate development process employs similar tools (loops, decisions, and sequences) to solve the same problem. The experiment shows that similar misuse of programming tools results in programs with similar faults. The implication for *N*-version programming is

that care must be taken to ensure that separately developed programs actually are independent, or, more realistically, that expected dependence among the N versions be taken into account.

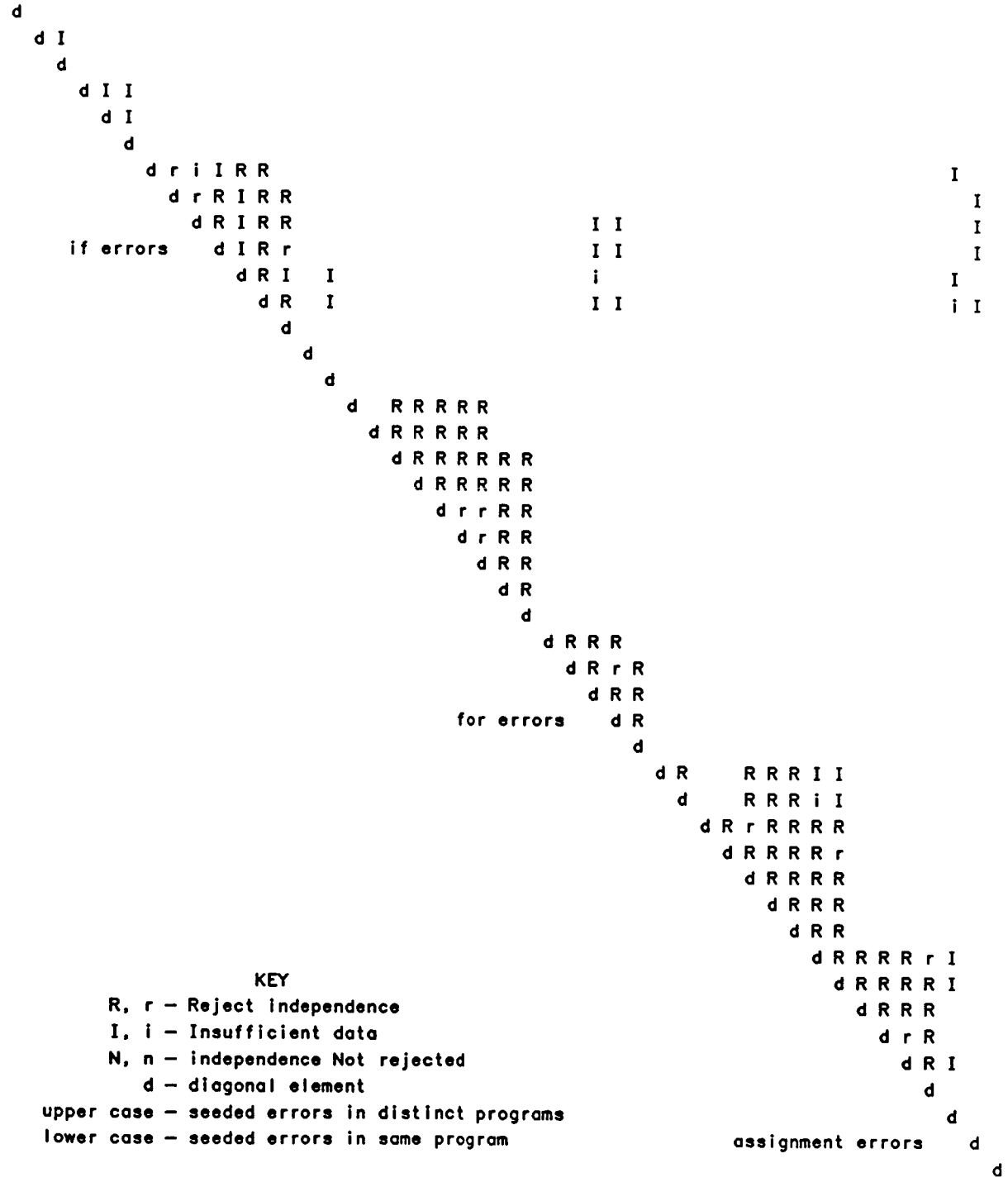


Figure 1: Results of Independence Hypothesis Test

REFERENCES

(1) J.C. Knight and N.G. Leveson, "An Experimental Evaluation Of The Assumption Of Independence In Multi-Version Programming,", *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 1, January 1986.

(2) L. Chen and A. Avizienis, "*N*-Version Programming: A Fault-Tolerance Approach To Reliability Of Software Operation", *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, Toulouse, France, pp. 3-9, June 1978.

(3) J.C. Knight and N.G. Leveson, "An Empirical Study Of Failure Probabilities In Multi-Version Software",

(4) S.S. Brilliant, J.C. Knight, and N.G. Leveson, "The Consistent Comparison Problem In *N*-Version Software", submitted to *IEEE Transactions on Software Engineering*.

(5) S.S. Brilliant, J.C. Knight, and N.G. Leveson, "Analysis Of Faults In A Multi-Version Software Experiment", in preparation.

(6) C.V. Ramamoorthy, Y.R. Mok, E.B. Bastani, G.H. Chin, and K. Suzuki. "Application Of A Methodology For The Development And Validation Of Reliable Process Control Software,", *IEEE Trans. on Software Engineering*, vol. SE-7, no. 6, pp. 537-555, Nov. 1981.

(7) S.S. Brilliant, "Testing Multi-Version Software", Ph.D. Dissertation Proposal, University of Virginia, June, 1986.

(8) A. Avizienis "The N-Version Approach To Fault Tolerant Software", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12 (December 1985).