1994

# Optimal solution of one dimension cutting stock problem

Pei-Chun Lin
*Lehigh University*

AUTHOR:

Lin, Pei-Chun

TITLE:

# Optimal Solution of One Dimension Cutting Stock Problem

DATE: May 29, 1994

# Optimal Solution of One Dimension Cutting Stock Problem

by

Pei-Chun Lin

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Industrial Engineering

Lehigh University

4/26/1992

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

4/28/94
Date

Thesis Advisor

Chairperson of Department

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to my advisor, Associate Professor John W. Adams for his guidance, time and patience. He gave me expert advice and valuable lessons in research methodology and these guidance are sure to stay with me for my life of career.

I would also like to thank my best friend, Yi-Hui Ma for her encouragement and great help in my study.

TABLE OF CONTENTS

# ABSTRACT

The proposed research considers the problem of minimizing waste in a paper cutting operation. The operation begins with that some parent rolls of indicated widths are to be cut to fill orders for rolls of specified widths. The objective of the study is to find out the way to carry out the required cutting operation and will produce the smallest amount of waste by exploiting linear programming.

Because in the real paper cutting problem, the number of columns in the linear programming will be very large, this problem was solved by observing that the 'pricing out' operation for the paper cutting problem was equivalent to a knapsack problem. Therefore, the entering column is found by solving a knapsack problem.

A C program has been written to solve the entire problem using the ideas of linear programming combined with the knapsack solution. Some simulation result are presented at the end of this study.

# Chapter 1 INTRODUCTION

The use of operations research methods to solve problems in material cutting has been an active topic of research for many years and has been applied in a number of manufacturing areas. This paper focuses on the problem of minimizing waste in a paper cutting operation that begins with different orders of rolls of specified widths. Those rolls are to be cut from parent rolls of particularized widths and the waste occurs when the unusable widths are left over.

This study solves the problem using methods reported by Gilmore and Gomory[1] in 1961. When the variety of orders is big, there will be many different ways of cutting the parent rolls to supply the need. Gilmore and Gomory use linear programming and the number of columns in the tableau will be very large -- in some cases up to millions of columns. Gilmore and Gomory solved this problem by using the recursive method of solving a knapsack problem because they found out that the 'pricing out' operation for the paper cutting problem was equivalent to a knapsack problem. By solving a knapsack problem we can identify an entering column. This study solves the knapsack problem using a method that was described by Garfinkel and Nemhauser[2].

In this paper, a C program was written for working out the entire problem by using the ideas of Gilmore and Gomory combined with the solution of knapsack problems reported by Garfinkel and Nemhauser. This C program can be used to simulate the paper cutting problem. Orders are random, but the widths of parent rolls are specified. Some parent rolls may be more efficient than others. It is difficult to find out an analytic solution to the problem of finding the optimal widths of parent rolls. However, it may be possible to get useful information by way of simulation. In the long run, the inventory generated by consecutive simulation can be treated as time series. By using the ARIMA models describes by Box and Jenkins[3], we can forecast the amount of future inventory. Chapter 6 of this thesis contains some simulation results.

The idea of solving the cutting problem were not only used in the paper industry but also in many other industries[4]. Sweeney and Paternoster[5] have compiled a bibliography that contains more than 400 books, articles, dissertations, and working papers. Some of these references are about paper cutting, and many of them are about other material -- steel, glass, wood, plastic, and many others. In chapter 3 of this thesis we will review a few of these papers.

The following sections represent the definition of paper cutting problem,

the literature review, the solution of problem, the description of a knapsack problem and how did the knapsack problem relate to the original problem that we'd like to solve, and the results of simulation.

Notation:

(1) Let $R=(r_1, r_2, r_3, \ldots, r_k)$, $r_i$ is the width of the i-th roll in order.

(2) Let $N=(n_1, n_2, n_3, \ldots, n_k)$, $n_i$ is the number of rolls of the i-th width that have been ordered.

(3) Let $W=(w_1, w_2, w_3, \ldots, w_l)$, $w_i$ is the width of the i-th parent roll from which the orders are to be cut.

(4) Let $S=(s_1, s_2, s_3, \ldots, s_k)$, $s_i$ is either 1 or 0.

(5) Let $X=(x_1, x_2, x_3, \ldots, x_n)$, $x_i$ is the number of times the *activity i* is to be engaged in.

In the case of the basic single-machine paper cutting problem, all parent rolls have the same length, and all of the ordered rolls also have the same length as parent rolls. Since those widths demanded are smaller or equal to the widths of parent rolls ($w_i \geq r_j$ for all i, j), the cutting problem begins with finding combinations of the ordered widths, and with which to fill out the widths of parent rolls. Usually, there will be some trim loss left over from such combinations. The paper cutting problem is to find out the trimming combinations of ordered widths and to determine that the number of which parent rolls are to be cut according to each combination -- in order to satisfy the orders most efficiently[5].

The vector S is called a slitting pattern because it tells us how to cut a parent roll into rolls of narrower widths. If $s_i$ is one, a roll of width $r_i$ will be cut from a parent rolls. This imposes the following constraint, applying to a parent roll of width w.

$$s_1r_1 + s_2r_2 + s_3r_3 + \ldots + s_kr_k \leq w$$

In practical cases there are a very large number of possible slitting patterns, but only a few of them will actually be used in a given case.

For an offered set of orders, say (R,N), and an unlimited supply of parent rolls of widths W, there will be a set of feasible slitting patterns. To find out a feasible

e slitting pattern is equivalent to create an *activity, which means the cutting of a parent roll in a specified manner*. Hence, the cutting paper problem can be posed as an integer linear programming problem. Those variables $x_1$, $x_2$, ... , $x_n$ must satisfy k inequalities:

$$a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n \geq n_i, \quad (i=1.....k)$$

If an order for $n_i$ pieces of width $r_i$ is to be filled, where $a_{ij}$ is the number of pieces of length $r_i$ created by the j-th activity[1]. Moreover, vector X are supposed to minimize the cost function:

$$\sum c_i x_j \quad \text{for } i=1...l,\ j=1...n$$

where $c_i$ is the cost of the parent roll from which activity j is cut.

# Chapter 3 LITERATURE REVIEW

In 1961, Gilmore and Gomory proposed the linear programming approach, which solves the one-dimensional cutting problem that materials are manufactured to be cut to various customer widths so as to meet customer demands and also minimize waste was the first to present methods which could be practically applied to real-world problem[1]. Since then, the cutting problems have been extended from original one-dimensional problem to two-dimensional, three-dimensional problems. Because the problems were extended, those related techniques which were used to solve the cutting problem by all means increased vastly. Therefore, the research of cutting problem became diverse and complicated, and a large number of articles were published on these problems. In this paper, we narrow down the cutting problem to be only one-dimension and broadly follow the techniques which were proposed by Gilmore more Gomory. Hence, we categorize the review of literature to three parts:

(1) introduce the important bibliography of the cutting problem,

(2) recite how was the idea of linear programming adapted to the cutting problem by Gilmore and Gomory, and

(3) compare the ways of rounding due to the restriction to integers.

1. Sweeney and Paternoster[6] published a categorized, application-orientated research bibliography for cutting and packing problems. In this paper, the author listed more than 400 books, articles, dissertations and working papers which all talked about the subject -- cutting and packing problems. First of all, the author introduced fifteen references to those readers who are not familiar with the cutting and packing problems to give them a basic idea. Then, the author generated a 3×3 matrix which was formed by *solution approach* and *problem dimension*. According to the solution methodologies employed, there are three categories:

- sequential assignment heuristic
- single-pattern generating procedures, and
- multiple-pattern generating procedures.

For the other categorization, problems were divided into one-dimensional, two-dimensional, or three-dimensional problems. So as to be more convenient to those who are interested in various special topics on cutting and packing problem, the paper also listed detail citations. The main part of the bibliography is arranged chronologically by year and alphabetically by author within each year. At the end of this paper, there are two supplementary lists. One contains dissertations and theses, and the other provides recent (1986 to 1990) proceedings, presentations, and working papers. In summary, the paper

provides a good and very detail literature review on the cutting and packing problems.

2. Gilmore and Gomory published two papers of the same topic -- A linear programming approach to the cutting-stock problem. In part one, they use dynamic programming to solve the cutting problems, and in part two, they use knapsack method instead. In this study, we broadly follow their techniques to solve the cutting problem. We will recite these two method next.

Following the definition of problem in chapter 2, we know the linear programming problem which we'd like to solve is to find out $x_1, x_2, ..., x_n$ to satisfy k inequalities:

$$a_{i1}x_1 + a_{i2}x_2 + ... + a_{in}x_n \geq n_i, (i=1...k)$$

and also minimize

$$\sum c_i x_j \quad \text{for } i=1...l, j=1...n$$

where the notation is identical to chapter 2.

For integer problems, it is required that all $x_j$ be integers, but this constraint is dropped here. Next, Gilmore and Gomory introduced slack variables $x_{n+1}, ..., x_{n+m}$ satisfying:

$$\min \quad \sum c_i x_j \tag{1}$$

$$\text{s.t.} \quad a_{i1}x_1 + a_{i2}x_2 + ... + a_{in}x_n - x_{n+i} = n_i, (i=1...m) \tag{2}$$

$$x_j \geq 0, \qquad (3)$$

By using the simplex computational procedure, a given solution subject to (2) and (3) for which (1) is a minimum provides that any basic solution of equation (2) and (3) must make the value of (1) less than the given solution. In order to express the algorithm of solution, more notations are listed as follow.

Notation:

- Let $X=(x_1, x_2, ..., x_m)$ is a given basic feasible solution.
- Let $P_j=(a_{1j}, a_{2j}, ..., a_{mj})$ is the number of pieces of length $r_1, r_2, ..., r_m$ created by the j-th activity.
- Let $P=(a_1, a_2, ..., a_m)$ is an undetermined new activity that cuts from a stock length L having a cost c.
- Let A be the matrix with $P_1, ..., P_m$ as columns.
- Let $C=[c_1, c_2, ..., c_m]$ is a vector with cost coefficient.

Because $P_1, ..., P_m$ form a basis, there is the usual column vector U satisfying the equation:

$$AU = P$$

$$U = A^{-1}P$$

If and only if

$$CU > c$$

then the new activity will be an improvement over the given solution

$$CA^{-1}P > c$$

Let $CA^{-1}$ have coefficients $b_1, b_2, \ldots, b_m$.

If and only if there exist nonnegative integer $a_1, \ldots, a_m$ satisfying

$$L \geq r_1a_1 + \ldots + r_ma_m \qquad (4)$$

and

$$b_1a_1 + \ldots + b_ma_m > c \qquad (5)$$

then the activity cutting from L can be profitable.

One method of determining whether there exists positive integers $a_1, a_2, \ldots, a_m$ to satisfy equations (4) and (5) would be to determine integers which satisfy equation (4) and make $b_1a_1 + \ldots + b_ma_m$ be a maximum. If the maximum won't satisfy (5) then none would. Therefore, the problem becomes:

$$\max \quad b_1a_1 + b_2a_2 + \ldots + b_ma_m$$

$$\text{s.t.} \quad L \geq r_1a_1 + r_2a_2 + \ldots + r_ma_m$$

Using dynamic programming method which is similar to that described by Danzig[8], the problem can be defined as

$$\max \quad F_m(L) = b_1a_1 + b_2a_2 + \ldots + b_ma_m$$

$$\text{s.t.} \quad L \geq r_1a_1 + r_2a_2 + \ldots + r_ma_m$$

then

$$F_{m+1}(L) = \max\{nb_{m+1} + F_m(L - nr_{m+1})\}$$

where n is chosen such that

$$0 \leq n \leq [\, L / r_{m+1} \,]$$

square bracket denotes the largest integer part. $F_{m+1}(L)$ is the value of the most valuable combination that can be fitted into a knapsack of length L if only the first m+1 lengths can be used.

In the later paper published by Gilmore and Gomory, they described a faster knapsack method and changed the algorithm. First of all, the stock length L was replaced by $L_i$. That means it is necessary to solve a series of knapsack problem like

$$\max \quad M = \sum b_i a_i \qquad \text{for } i=1...m$$

$$\text{s.t.} \quad L \geq \sum r_i a_i \qquad \text{for } i=1...m$$

In chapter 5 of this paper, the knapsack algorithm will be described in detail.

3. Up to now, the linear programming is discussed for solving the paper-cutting problem. However, in a real paper-cutting problem, it is not permitted to have noninteger solution. Subject to this constraint, we can always round up or round down to the nearest integer but the results will be totally different. First, the amount of order must be filled. If we use the way of rounding down, then the insufficient part become another problem to be solved. In the other hand, rounding up creates extra rolls than it is ordered and those will be treated as inventory. Besides, there are some heuristic procedures to solve the pure-integer linear programming problem. In Sweeney and Paternoster's paper[6], they chose rounding as a special topic and listed

those papers related to it.

In 1966, Hillier[7] wrote a paper about the algorithm for pure integer linear programming. Initially, the optimal noninteger solution by the simplex method is required. Then considering the solutions (integer or noninteger) that satisfy both the objective function constraints and the constraints that are binding on the optimal noninteger solution. Hillier's algorithm should get a set of closer integer solution than totally rounding up which is used in the solution of problem in this study. Since the following part that we'd like to study focuses on the distribution of inventory, it is not the most important thing for this study whether we can find a best way to get the integer solution or not.

# Chapter 4 SOLUTION OF PROBLEM

The solution of the problem which was described in chapter 2 is what we would like to focus on for this study. The algorithm of solution is also the main frame of the C program which is used to do the work of simulation. Next, we will go through the whole algorithm to make the C program be easily understood and finally the C program will be attached to the appendix of this study.

## Creating the output file for the data of inventory

At the very beginning of this program, a few output file were created to store the numerical data of inventory. The reason to do this is we got rid of the limitation that the solution must be integer during the procedure of solving this problem. Therefore, those noninteger solution must be round up to fill every order and meanwhile, some extra rolls were generated. They are treated as inventory and would be spent when the next order comes in. So as to analyze the dependence of the data by exploiting the software named Statgraphic, those output file will save the data we need.

## Generating the random number to represent order

It was mentioned before that the number of order for each different size is random. A subroutine was called when simulation was started to generate

the number of order from customers.

**Setting up the initial matrix**

There are two matrices which would be updated in the process of calculating to show what we would like to know in each step of calculation. Initially, we have to set up these two matrices by input the number of parent rolls, the parent widths, and the costs of parent rolls, etc.

**Finding out the pivot row and entering column**

Every time this subroutine was called, the most recently updated shadow prices were known. After implementing the knapsack algorithm which will be recited in chapter 5, the subroutine will do calculation to update those two matrices and make judgement to know which column is the entering column and find out the pivot row.

**Updating the matrix which shows the shadow prices**

When leaving the previous subroutine, immediately we would like to know whether the optimal solution is found or not. If not, the subroutine was called to update the matrix which shows the shadow prices except the pivot row, then go to the previous subroutine again. If it is really done, then go to the final subroutine and print out the result we need.

A well known example of a knapsack problem is the cargo problem that is to maximize the total value of cargos which are going to be taken subject to the limitation of weight or space. Let $a_j > 0$ be the weight per unit and $c_j > 0$ be the value per unit of the j-th type. If the total limitation of weight or space is b, then the problem is

$$\begin{aligned} \max \quad & \sum c_j x_j \quad \text{for } j=1...n \\ \text{s.t.} \quad & \sum a_j x_j \le b \quad \text{for } j=1...n \\ & x_j \ge 0 \text{ integer } j=1...n \end{aligned}$$

where $x_j$ is the number of the j-th type included[?].

The following part is the algorithm which Garfinkel and Nemhauser developed to solve the knapsack problem and is the method used in this paper whenever we want to look for a new column or activity that will improve the solution.

The first step of calculation is to reorder the variables $a_1, a_2, ..., a_m$ (represented the ratio of shadow price divided by demanded length) so that $b_1/r_1 \ge b_2/r_2 \ge ... \ge b_m/r_m$, and reorder the stock lengths so that $L_1 > L_2 > ... > L_k$,

such that the array of parent widths became a decreasing array. Introduce a variable $a_{m+1}$ with coefficient $b_{m+1} = 0$ and $r_{m+1} = 1$. Let

$$a_1 = [L_1/r_1]$$

$$a_2 = [(L_1 - r_1 a_1) / r_2]$$

.....

$$a_m = \{[L_1 - (r_1 a_1 + ... + r_{m-1} a_{m-1})] / r_m\}$$

the bracket denotes the largest integer part again. Here, $(\alpha)_s$ represents a s-vector of nonnegative integers $a_1, a_2, ..., a_s$, where $1 \le s \le m$. A vector $(\alpha)_m$ is an extension of a vector $(\alpha)_s$, $s \le m$, if the first s coefficients of $(\alpha)_m$ are just the coefficients of $(\alpha)_s$. In the algorithm, a sequence of vectors $(\alpha)_s$, for various values of s, satisfying

$$L_1 \ge \sum r_i a_i \quad \text{for } i = 1...s$$

is generated in lexicographically descending order, where $(\alpha^1)_{s1}$ is lexicographically larger than $(\alpha^2)_{s2}$ if and only if for some i, $1 \le i \le \min\{s_1, s_2\}$, and

$$a_1^{\,1} = a_1^{\,2}$$

.....

$$a_i^{\,1} = a_i^{\,2}$$

while $a_{i+1}^{\,1} > a_{i+1}^{\,2}$

Let $t = 1$ and $M_j = c_j$, $j = 1...k$. The vector $(\alpha)_m$ is then tested to determine whether for it $\sum b_i a_i$ for $i = 1...m$ exceeds the current best values $M_j$

for applicable j, and if so these current best values are redefined.

(1) For those j, $t \leq j \leq k$, for which $L_j \geq \sum r_i a_i$ for i = 1...m and $\sum b_i a_i > M_j$ for i = 1...m, redefine $M_j$ to be $\sum b_i a_i$ for i = 1...m.

(2) Let s be the largest i, $1 \leq i \leq m$, such that $a_i \neq 0$. Thus $(\alpha)_s$ has among its coefficients all the nonzero coefficients of $(\alpha)_m$ and its last coefficient is nonzero. The lexicographically largest m-vector lexicographically smaller than $(\alpha)_m$ necessarily has its s-th coefficient one less than $a_s$; that is, it is an extension of a vector $(\alpha^1)_s$, which differs from $(\alpha)_s$ only in having $a_s$-1 as its s-th coefficient.

(3) Redefine $a_s$ to be $a_s$-1 and let t be the smallest j, $1 \leq j \leq k$, such that $L_j \geq \sum r_i a_i$ for i = 1...s and $(L_j - \sum r_i a_i) b_{s+1} > (M_j - \sum b_i a_i) r_{s+1}$, and go to step (5). If there is no such j then go to step (4).

(4) Redefine s to be the largest i, $1 \leq i \leq s-1$, such that $a_i \neq 0$, and go to (3). If there is no such i, then the current values of $M_j$, j = 1...k are the maximums to be found.

(5) Let

$$a_{s+1} = [(L_t - \sum r_i a_i) / r_{s+1}]$$

.....

$$a_m = [(L_t - \sum r_i a_i - r_{s+1}a_{s+1} - \ldots - r_{m-1}a_{m-1}) / r_m]$$

and go to step (1)[9].

It was mentioned in chapter 4 that a series of data represented inventory would be generated after running the C program. Initially, we tried to figure out whether there is any relationship between the distribution of inventory and the probability distribution which was used to generate the random number for different orders. Since the extra rolls cut by each period are treated as inventory and always spent in the coming period, it seems that the data represented inventory are not independent. Hence, the data could not be fitted into any identical probability distribution. However, when data are dependent, they can be treated as time series if the periods are long enough. Therefore, we simulated at least five hundred times , sometimes up to three thousand times, and analyzed the data by the method proposed by Box and Jenkins[3]. By implementing this method, some models are derived and they can depict the nature of the system generating the time series. Besides, these models can be used for obtaining forecasts of future values of the series. In this study, we would like to know what is the attribute of the system which illustrated inventory. For instance, if the system can be represented by the model so called *autoregressive* model, that means the current value of the process is expressed as a finite, linear aggregate of previous values of the process. Intuitively, for our inventory system, it has the similar characteristic. The more we want to know is the next stock may be impacted

by the previous one, two, or three periods. The information is pretty practical for the realistic inventory control in factory. Moreover, the model can help us to predict the future inventory which also provide useful data.

To illustrate the procedures described in chapter 4 and 5, consider the following case. There are three different parent widths, 72, 78, and 84 and each individual cost is 72.0, 78.0, and 84.0. Six kinds of widths are ordered -- 5, 7, 11, 16, 23, and 25. The mapping number of rolls are generated by the binomial probability distribution with parameters 100, 0.16. Initially, all stocks are zero. In appendix B, the results of simulation for running 500 periods are presented in the sequence of widths -- 5, 7, 11, 16, 23, and 25. And in appendix C, it runs 1000 periods. For each individual width, first figure tells the amount of inventory versus time; next figure displays the forecast of the future inventory of 20 periods with 95 percent limits; and the final figure arranges the horizontal axis to amplify the shift of future inventory and makes the data clearer to read.

# Chapter 7 FURTHER STUDY

For a lot of industries, how to reduce the cost of raw materials is an important issue. In many assembly and manufacturing factories, to buy raw materials then do some processes on the raw materials are major operations. If raw materials could be cut in the method which minimizes the loss, then at the same time the cost would be saved. For instance, in a window assembly factory, the raw materials need to be cut in one dimension would be like frame and sash; in two dimensions might be glass and screen, etc. It is clear to see for those assembly industries that the cost of raw materials occupies a primary promotion of total manufacturing cost. Therefore, to make the cutting of raw materials most economic is an efficient way to cut down the cost. Since the algorithm of optimizing the cutting operations for one dimension may not be appropriate for two dimensions, the two dimensions' or even three dimensions' cutting procedure is not only a useful but also a practical field which is worth studying much more.

Besides the cost of raw materials, inventory control is also a momentous topic in industries. After doing the work of simulation which is proposed in this study, it is accessible to predict a few periods of future inventory. As we know, there exists a lot of obedient algorithm for inventory control. It must be interesting if these information can be combined together to pursue better

inventory control.

# REFERENCES


1 **P. C. Gilmore and R. E. Gomory** 1961. A Linear Programming Approach to the Cutting Stock Problem. *Opns Res.*(9):849-859.

2 **Robert S. Garfinkel and George L. Nemhauser** 1972. *Integer Programming*, p. 214. John Wiley & Sons, New York.

3 **George E. P. Box and Gwilym M. Jenkins** 1982. *Time Series Analysis: forecast-ing and control*, Holden-Day.

4 **Hartmut Stadtler** 1990. A one-dimensional cutting stock problem in the aluminum industry and its solution. *European Journal of Operational Research*(44):209-223.

5 **J. F. Pierce** 1964. *Some Large Scale Production Scheduling Problems in the Paper Industry*, p. 13. Prentice Hall, Englewood Cliffs, N.J.

6 **Paul E. Sweeney and Elizabeth Ridenour Paternoster** 1992. Cutting and Pack-ing Problems: A Categorized Application-Orientated Research Bibliography. *Opl Res. Soc.*(43):691-706.

7 **Frederick S. Hillier** 1966. Efficient Heuristic Procedures for Integer Linear Programming with an Interior. *Opns. Res.*(14):600-637.

8 **George B. Dantzig** 1957. Discrete-Variable Extremum Problems. *Opns. Res.*(5):266-277.

9 **P. C. Gilmore and R. E. Gomory** 1963. A Linear Programming Approach to the cutting Stock Problem-part 2. *Opns Res.*(9):849-859.

# APPENDIX A

# PROGRAM OF SIMULATION

```
/*****************************************************************/
/* TRIMLOSS                                                      */
/* npw is the number of parent widths                            */
/* pw[npw] is an integer array containing a list of parent widths */
/* pc[npw] is a real array containing the costs of the parent rolls */
/* mpw is the maximum parent width                               */
/* cpw is the number of different widths in order                */
/* nwo is the number of different widths in order                */
/* lw[nwo] is an integer array containing a list of widths ordered */
/* nr[nwo] is an integer array containing the number of rolls ordered */
/*****************************************************************/

#include <time.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define nwo 6
#define npw 3
#define mpw 84

int d[mpw+1], x[nwo], ip[nwo], u[nwo], z[nwo], cpw, pr=0;
int i, j, index, count;
int b[nwo+1][nwo], newh[nwo];
float sp[nwo], g[mpw+1], y[nwo], v[nwo], q[nwo], t[nwo+1], p[nwo+1];
float h[nwo+1][nwo+3];
int pw[npw] = {72, 78, 84};
int lw[nwo] = {5,7,11,16,23,25}, nr[nwo], exact[nwo], inior[nwo];
int MM[nwo] = {0,0,0,0,0,0}, stock[nwo] = {0,0,0,0,0,0};
float pc[npw] = {72.0, 78.0, 84.0};
int done=1;
int in=0;
float w, cost=0.0;
FILE *outptr0, *outptr1, *outptr2, *outptr3, *outptr4, *outptr5;

void print();
void ran();
void start();
void findp();
void rowreduce();
void knapsack();
void sort();
void sorti();
```

```
/**************************/
/* This is the main program */
/**************************/

Void main()
{
if(( outptr0 = fopen("sto5.wk1", "w+")) == NULL )
{
printf("Unable to open output file");
exit(1);
}
if(( outptr1 = fopen("sto7.wk1", "w+")) == NULL )
{
printf("Unable to open output file");
exit(1);
}
if(( outptr2 = fopen("sto11.wk1", "w+")) == NULL )
{
printf("Unable to open output file");
exit(1);
}
if(( outptr3 = fopen("sto16.wk1", "w+")) == NULL )
{
printf("Unable to open output file");
exit(1);
}
if(( outptr4 = fopen("sto23.wk1", "w+")) == NULL )
{
printf("Unable to open output file");
exit(1);
}
if(( outptr5 = fopen("sto25.wk1", "w+")) == NULL )
{
printf("Unable to open output file");
exit(1);
}

randomize();
for(count=0; count<500; count++)
{
ran();
done=0;
start();
```

```
FINDP:
        findp();
        in=in+1;
        if(!done)
                {
                rowreduce();
                goto FINDP;
                }
        print();
        }
} /* end of main */


/****************************************/
/* This subroutine is to print out the result */
/****************************************/

void print()
{
        for(i=0; i<nwo+1; i++)
                {{ for(j=0; j<nwo+3; j++)
                        printf(" %6.2f", h[i][j]);}
                printf("\n");}

        for(i=0; i<nwo+1; i++)
                {{ for(j=0; j<nwo; j++)
                        printf(" %5d", b[i][j]);}
                printf("\n");}

        printf("\nThe round-up result is: ");
        for(i=1; i<nwo+1; i++)
                {
                newh[i-1] = ceil(h[i][nwo+1]);
                printf(" %5d", newh[i-1]);
                }

        printf("\nThe cost of integer solution is: ");
        cost=0.0;
        for(index=0; index<nwo; index++)
                cost = cost + pc[(b[0][index])] * newh[index];
        printf(" %0.1f\n",cost);

        for(i=0; i<nwo; i++)
                {
```

```
                exact[i] = 0;
                for(j=0; j<nwo; j++)
                        exact[i] = exact[i] + newh[j] * b[i+1][j];
                }

        for(i=0; i<nwo; i++)
                {
                if( MM[i] != 0 )
                        stock[i] = exact[i] - MM[i];
                else
                        stock[i] = stock[i] - inior[i] + exact[i];
                printf("\nExact[%2d] = %2d      NewStock[%2d]=%d"
                        , lw[i], exact[i], lw[i], stock[i]);

                if(i==0)
                        fprintf(output0, "%2d\n", stock[i]);
                if(i==1)
                        fprintf(output1, "%2d\n", stock[i]);
                if(i==2)
                        fprintf(output2, "%2d\n", stock[i]);
                f(i==3)
                        fprintf(output3, "%2d\n", stock[i]);
                f(i==4)
                        fprintf(output4, "%2d\n", stock[i]);
                f(i==5)
                        fprintf(output5, "%2d\n", stock[i]);
                }

        if(count==499)
                {
                fclose(output0);
                fclose(output1);
                fclose(output2);
                fclose(output3);
                fclose(output4);
                fclose(output5);
                }
        printf("\n");
} /* end of subroutine print */

/*************************************************/
/* This subroutine is to get random number for order */
/*************************************************/
```

```
void ran()
{
        int i, j, k, x;
        double R[nwo];
        signed long double nom, den1, den2;
        long double Prob, t1, t3, sum;

        /* the factorial of 100 */
        nom=1;
        for(i=100; i>0; i--)
                nom=nom*i;
        prinf("\n-------------------------------------------------------\n");

        for(i=0; i<6; i++)
                {
                sum=0.0;
                R[i] = random(1000)/1000.0;
                t1 = pow(0.16, 0) * pow(0.84, 100) - R[i];

                if(fabs(t1) <= 0.001)
                        {
                        nr[i] = 0;
                        inior[i] = 0;
                        MM[i] = 0;
                        }

                else
                        {
                        sum = pow(0.16, 0) * pow(0.84, 100);
                        for(x=1; x<100; x++)
                                {
                                den1=1;
                                for(j=x; j>0; j--)
                                        den1 = den1 * j;

                                den2=1;
                                for(k=100-x; k>0; k--)
                                        den2 = den2 * k;

                        Prob = nom/(den1*den2) * pow(0.16,x) * pow(0.84,100-x);
                        sum = sum + Prob;
                        t3 = sum - R[i];
```

```
                if( fabs(t3)<0.08 )
                        {
                        nr[i] = x;
                        inior[i] = x;
                        if( inior[i] > stock[i] )
                                MM[i] = inior[i] - stock[i];
                        else
                                MM[i] = 0;
                        nr[i] = MM[i];
                        goto PRINT;
                        }
                }}

PRINT:          printf("Order[%2d]=%2d)          MustMake[%2d]=%d "
                        ,lw[i], inior[i], lw[i], MM[i]);
                printf("\n");
        }
} /* end of subroutine random */


/************************************************/
/* This subroutine is to generate the initial matrix */
/************************************************/

void start()
{
        int i, j;
        float s;
        for(i=0; i<nwo+1; i++)
                p[i] = 0.0;

        for(i=0; i<nwo+1; i++)
                for(j=0; j<nwo; j++)
                        { if(i-j==1)
                                b[i][j] = 1;
                          else
                                b[i][j] = 0;
                        }

        for(j=0; j<nwo; j++)
                b[0][j] = npw;

        for(i=0; i<nwo+1; i++)
                for(j=0; j<nwo+3; j++)
```

```
                    { if(i-j==0)
                          h[i][j] = 1.0;
                      else
                          h[i][j] = 0.0;
                    }

        for(j=1; j<nwo+1; j++)
              h[0][j] = pc[npw-1];

        t[0] = 0.0;
        for(i=1; i<nwo+1; i++)
              t[i] = nr[i-1];

        for(i=0; i<nwo+1; j++)
              {
              s=0.0;
              for(j=0; j<nwo+1; j++)
                    s=s+t[j]*h[i][j];
              h[i][nwo+1]=s;
              }
} /* end of subroutine start */


/************************************************/
/* This subroutine is to find out the pivot column */
/************************************************/

void findp()
/* The next two instructions get the most recently updated shadow prices */
{
        int i, j, rn, ip[nwo];
        float s;
        for(j=1; j<nwo+1; j++)
              sp[j-1] = h[0][j];
        for(i=npw-1; i>=0; i--)
              {
              knapsack(pw[i]);
              if( w > pc[i] )
                    {
                    s=abs(p[0]+pc[i]);
                    for(j=0; j<nwo; j++)
                          s=s+abs(p[j+1]-x[j]);
                    if(s>0.1)
                          goto RN;
```

```
                }
            }
CONT:       done=1;
            return;
RN:         rn=i;
```

/* The next three instructions find the entering column which is p */

```
    for(i=1; i<nwo+1; i++)
            p[i] = x[i-1];
    p[0] = -pc[rn];
```

/* The next five instructions compute the product of BINV and p and put it in the last column of the matrix h */

```
    for(i=0; i<nwo+1; i++)
            {
            s=0.0;
            for(j=0; j<nwo+1; j++)
                    s=s+p[j]*h[i][j];
            h[i][nwo+2]=s;
            }
```

/* The next eight instructions find the pivot row which is pr */

```
    for(i=0; i<nwo; i++)
            {
            ip[i]=i;
            if( h[i+1][nwo+1] >= 0.000001 )
                    q[i] = h[i+1][nwo+1] / h[i+1][nwo+2];
            else
                    q[i] = 10.0e10;
            }

    sort(nwo,q,ip);
    pr = ip[0]+1;
```

/* The next three instructions uodate matrix B */

```
    for(i=1; i<nwo+1; i++)
            {
            b[i][pr-1] = x[i-1];
            b[0][pr-1] = rn;
            }
```

```
} /* end of subroutine findp */


/*********************************/
/* This subroutine is for rowreduce */
/*********************************/

void rowreduce()
{
        int i, j;
        for(j=0; j<nwo+3; j++)
                h[pr][j] = h[pr][j] / h[pr][nwo+2];

        for(i=0; i<nwo+1; i++)
                {
                if( i!=pr )
                        {
                        for(j=0; j<nwo+3; j++)
                                h[i][j] = h[i][j] - h[pr][j] * h[i][nwo+2];
                        }
                }
} /* end of subroutine rowreduce */


/*****************************************/
/* This subroutine is using knapsack method */
/*****************************************/

void knapsack(cpw)
{
        int i, j, k;
        for(i=0; i<=cpw; i++)
                {
                g[i]=0.0;
                d[i]=nwo;
                }

        for(i=0; i<nwo; i++)
                {
                z[i]=0;
                ip[i]=i;
                if( sp[i]==0.0 )
                        y[i]=10.0e10;
                else
```

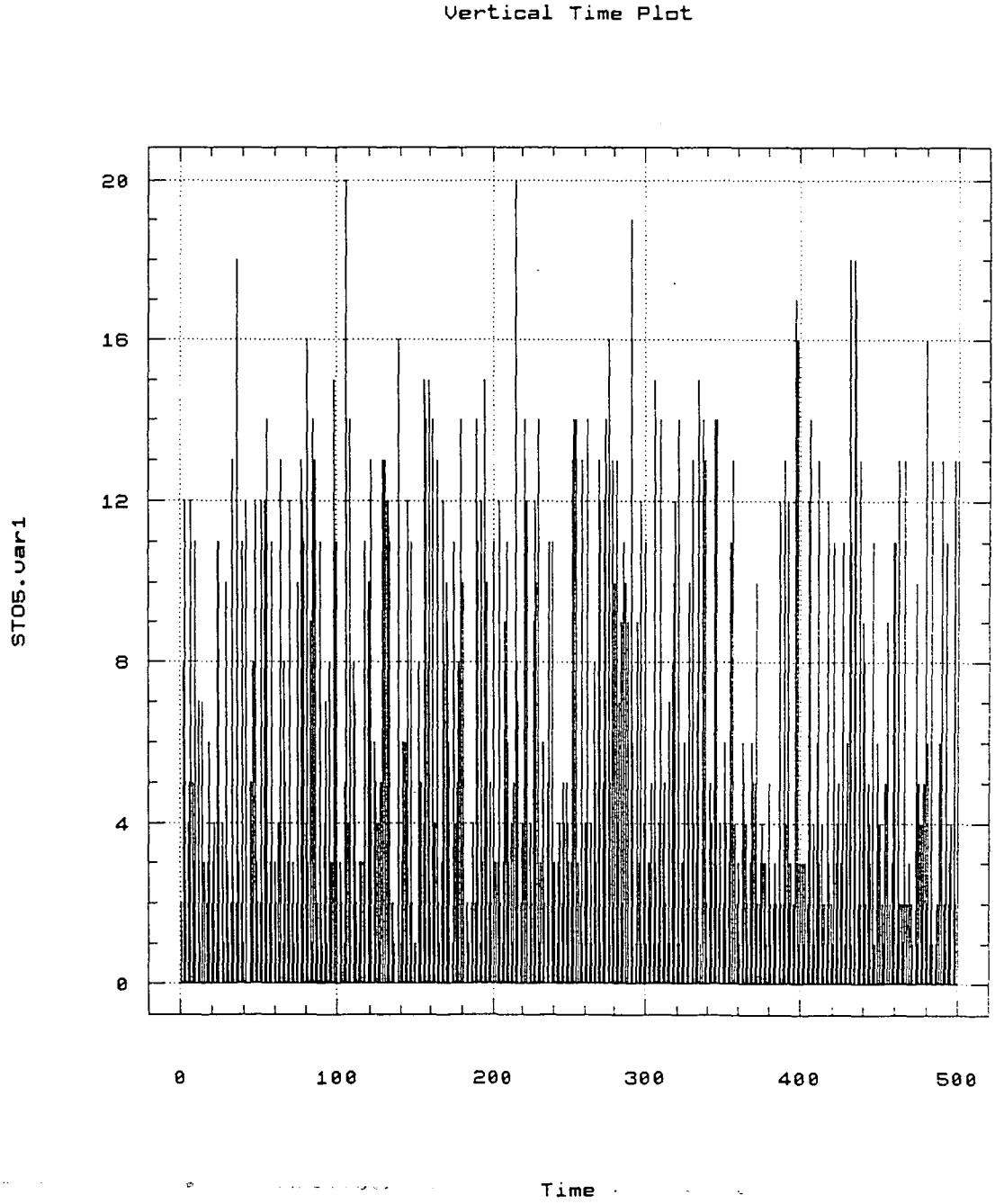
```
                y[i]=lw[i] / sp[i];
        }

    sort(nwo,y,ip);
    for(i=0; i<nwo; i++)
        {
        u[i] = lw[ip[i]];
        v[i] = sp[ip[i]];
        }

    j=0;
    k=1;

KU:
    if( k-u[j] >= 0 && j<= d[k-u[j]] )
        goto GIVE_W;
    else
        goto CPW;

GIVE_W:
    w=g[k-u[j]] + v[j];
    if(w<=g[k])
        goto CPW;
    else
        {
        g[k]=w;
        d[k]=j;
        }

CPW:
    if(j<nwo-1)
        {
        j=j+1;
        goto KU;
        }

K_CPW:
    if(k<cpw)
        {
        j=0;
        k=k+1;
        goto KU;
        }
    w=g[cpw];
```

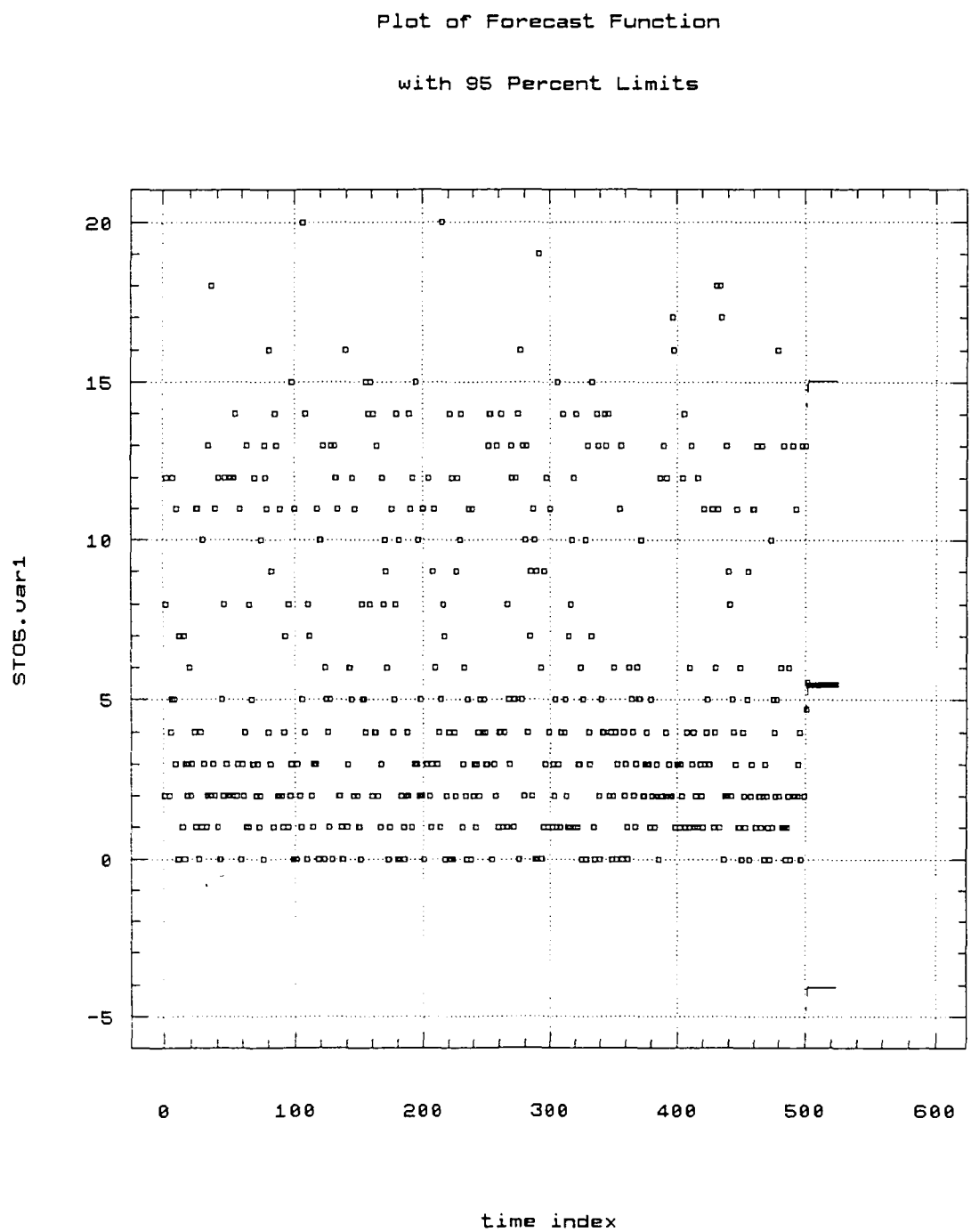
```
G_K:
        if(g[k] == 0.0)
                {
                for(i=0; i<nwo; i++)
                        ip[i] = i;
                sorti(nwo,u,ip);
                for(i=0; i<nwo; i++)
                        x[i] = z[ip[i]];
                }

        else
                {
                z[d[k]] = z[d[k]] + 1;
                k = k - u[d[k]];
                goto G_K;
                }
} /* end of subroutine knapsack */


/*********************************/
/* This is the subroutine for float */
/*********************************/

void sort(nwot,yt,ipt)
        int ipt[nwo];
        float yt[nwo];
        {
                int i, tempi, flag;
                float temp;

                do
                {
                flag=0;
                for(i=0; i<nwo-1; i++)
                        if(yt[i] > yt[i+1])
                        {
                        temp=yt[i];
                        tempi=ipt[i];
                        yt[i]=yt[i+1];
                        ipt[i]=ipt[i+1];
                        yt[i+1]=temp;
                        ipt[i+1]=tempi;
                        flag=1;
                        }
```

```
            }
            while(flag);
        }


/**********************************/
/* This is the subroutine for integer */
/**********************************/

void sorti(nwot,yt,ipt)
        int ipt[nwo];
        int yt[nwo];
        {
                int i, temp, tempi, flag;

                do
                {
                flag=0;
                for(i=0; i<nwot-1; i++)
                        if(yt[i] > yt[i+1])
                        {
                        temp=yt[i];
                        tempi=ipt[i];
                        yt[i]=yt[i+1];
                        ipt[i]=ipt[i+1];
                        yt[i+1]=temp;
                        ipt[i+1]=tempi;
                        flag=1;
                        }
                }
                while(flag);
        }
```

# APPENDIX B

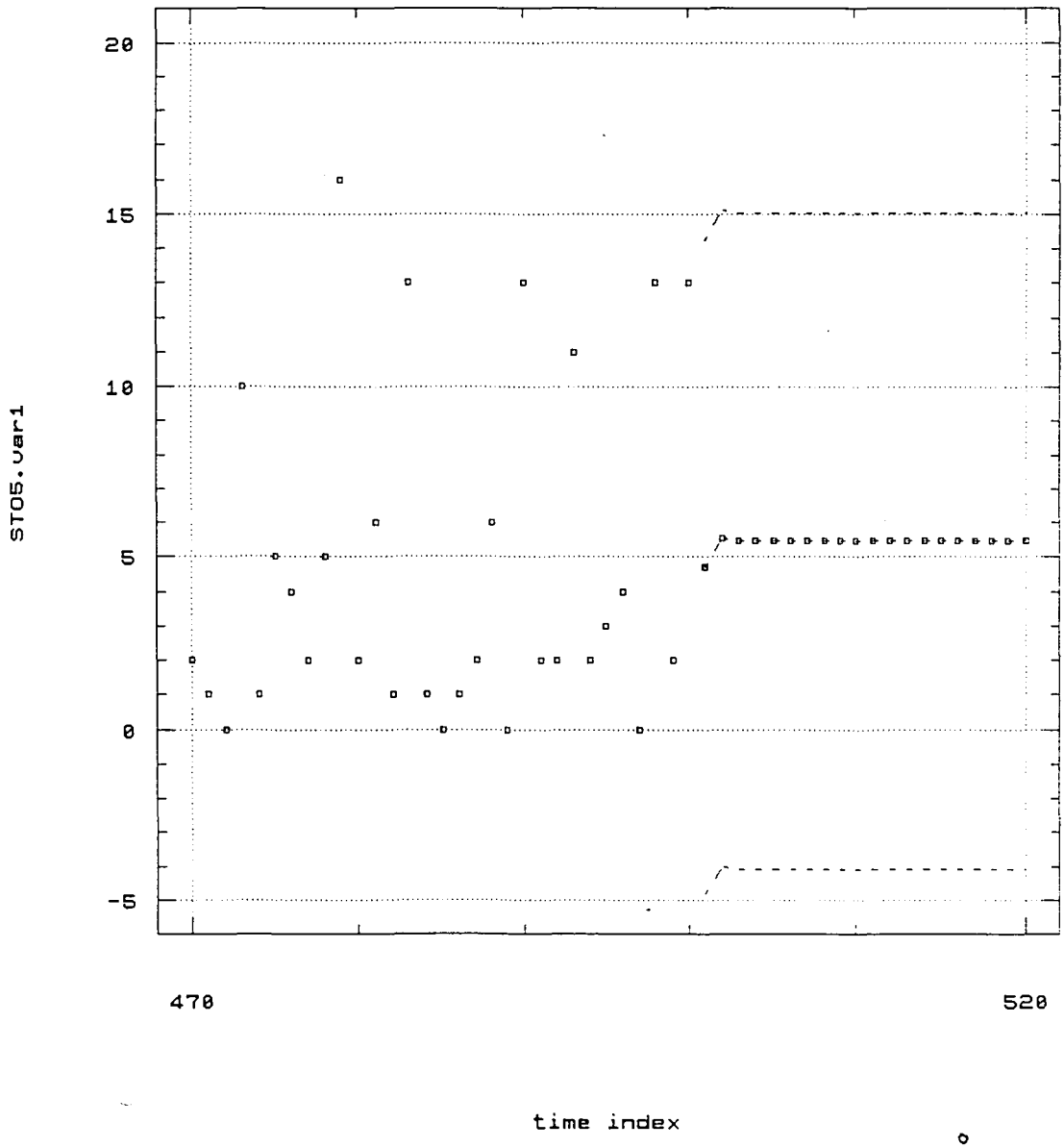
## RESULT OF SIMULATION FOR 500 PERIODS

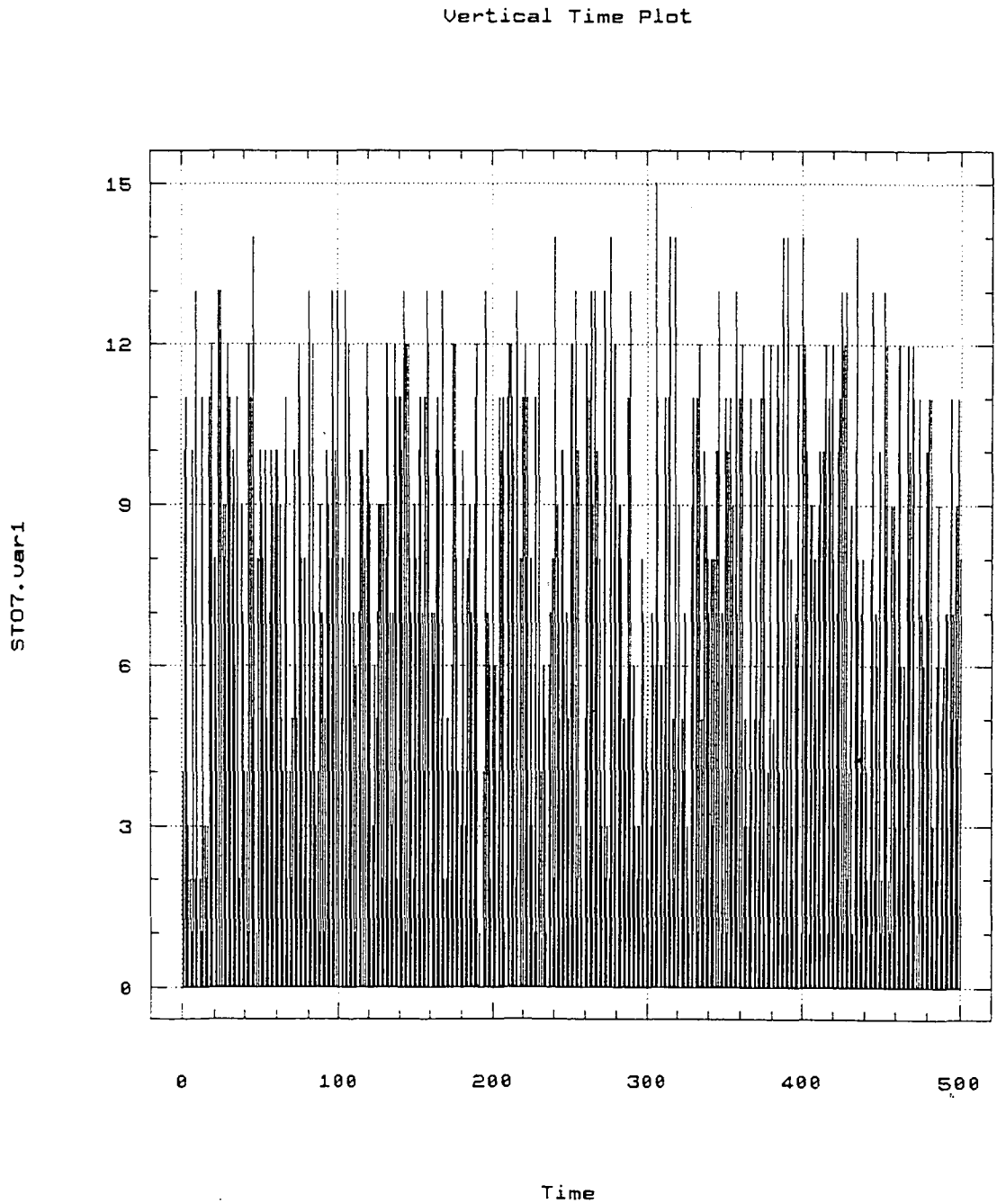

Vertical Time Plot
20
16
12
8
4
0
STD5.var1
0
100
200
300
400
500
Time


Plot of Forecast Function
with 95 Percent Limits
STO5.var1
20
15
10
5
0
-5
0
100
200
300
400
500
600
time index

Plot of Forecast Function

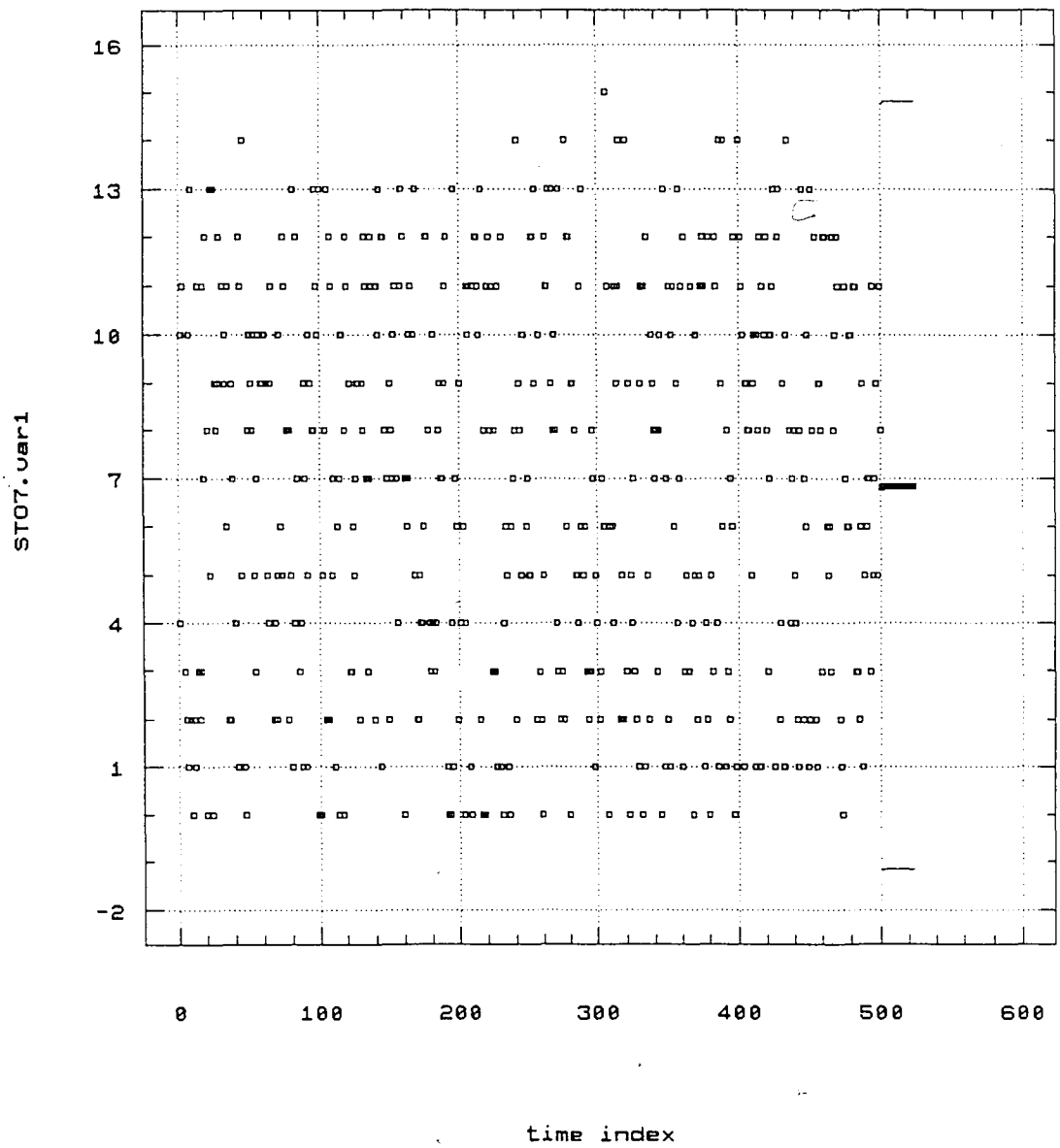
with 95 Percent Limits




Vertical Time Plot
15
12
9
6
3
0
ST07.Var1
0
100
200
300
400
500
Time

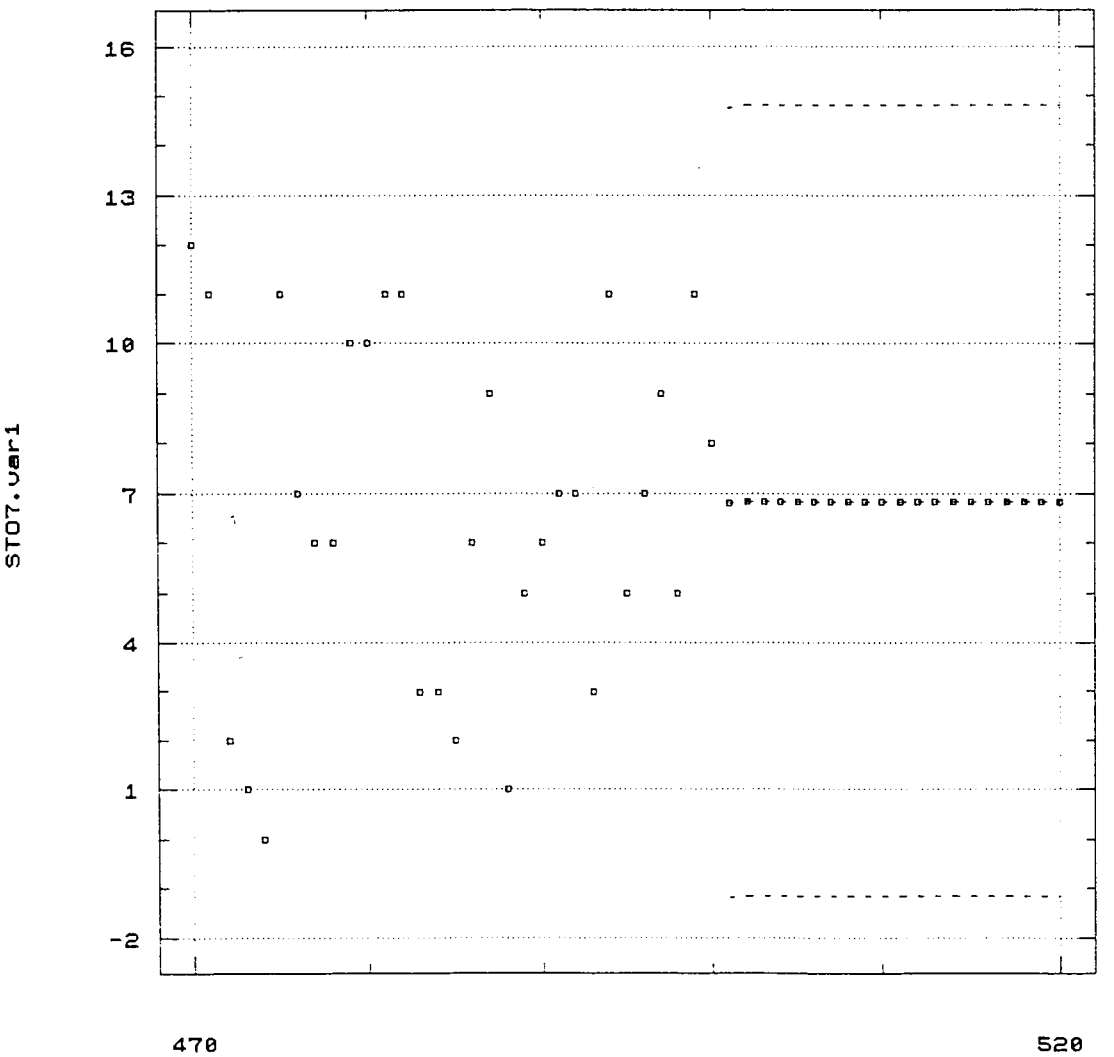
Plot of Forecast Function

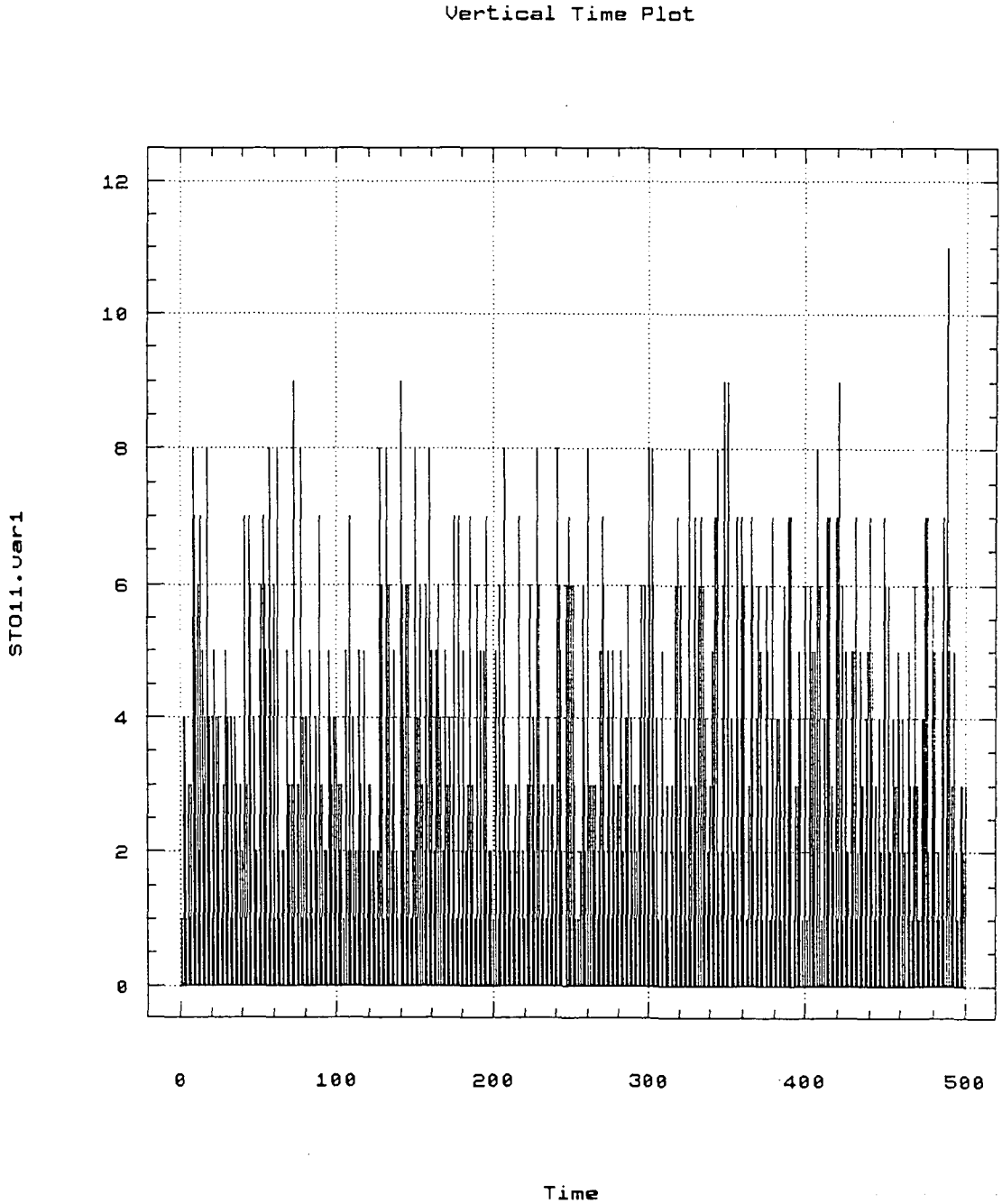
with 95 Percent Limits

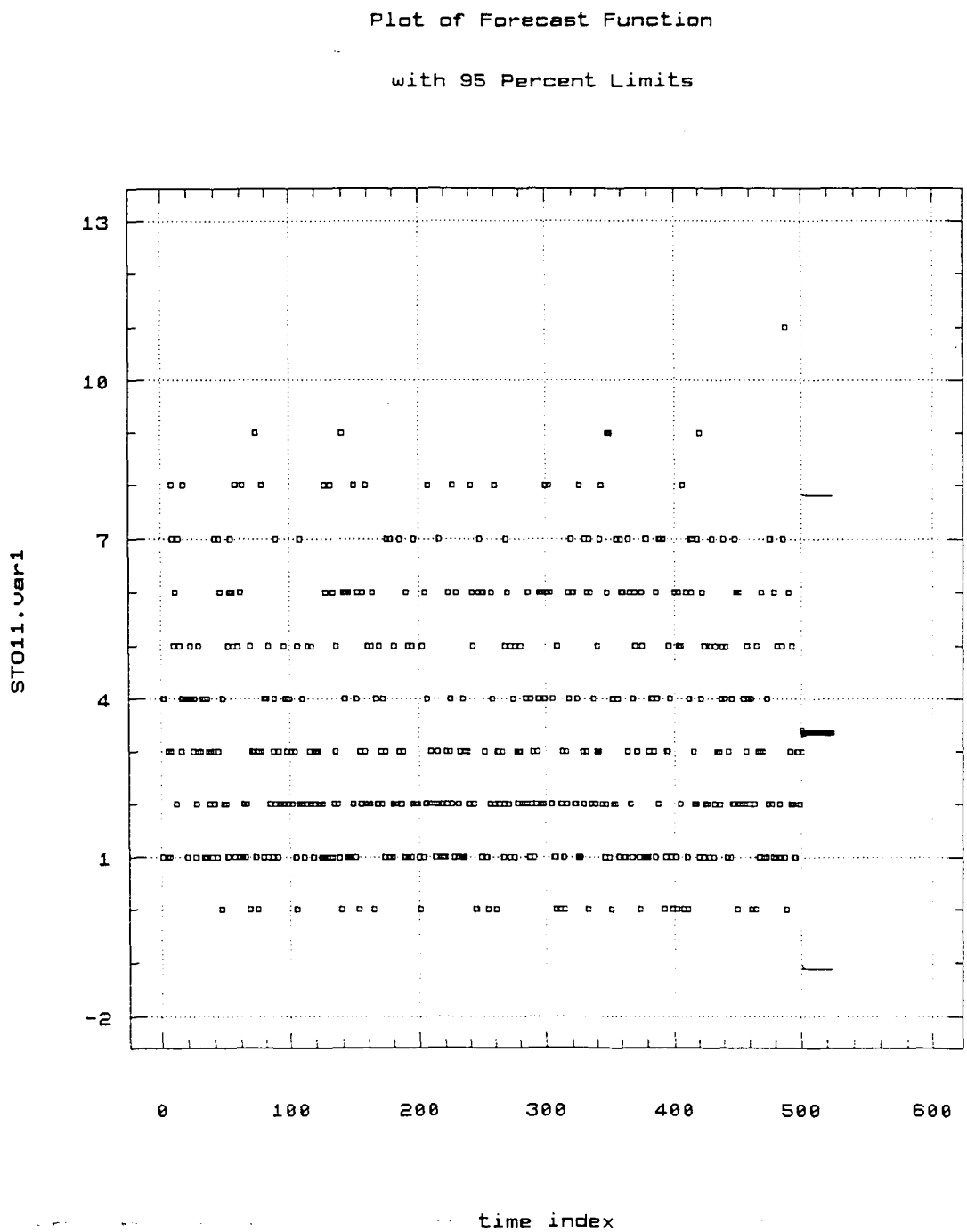


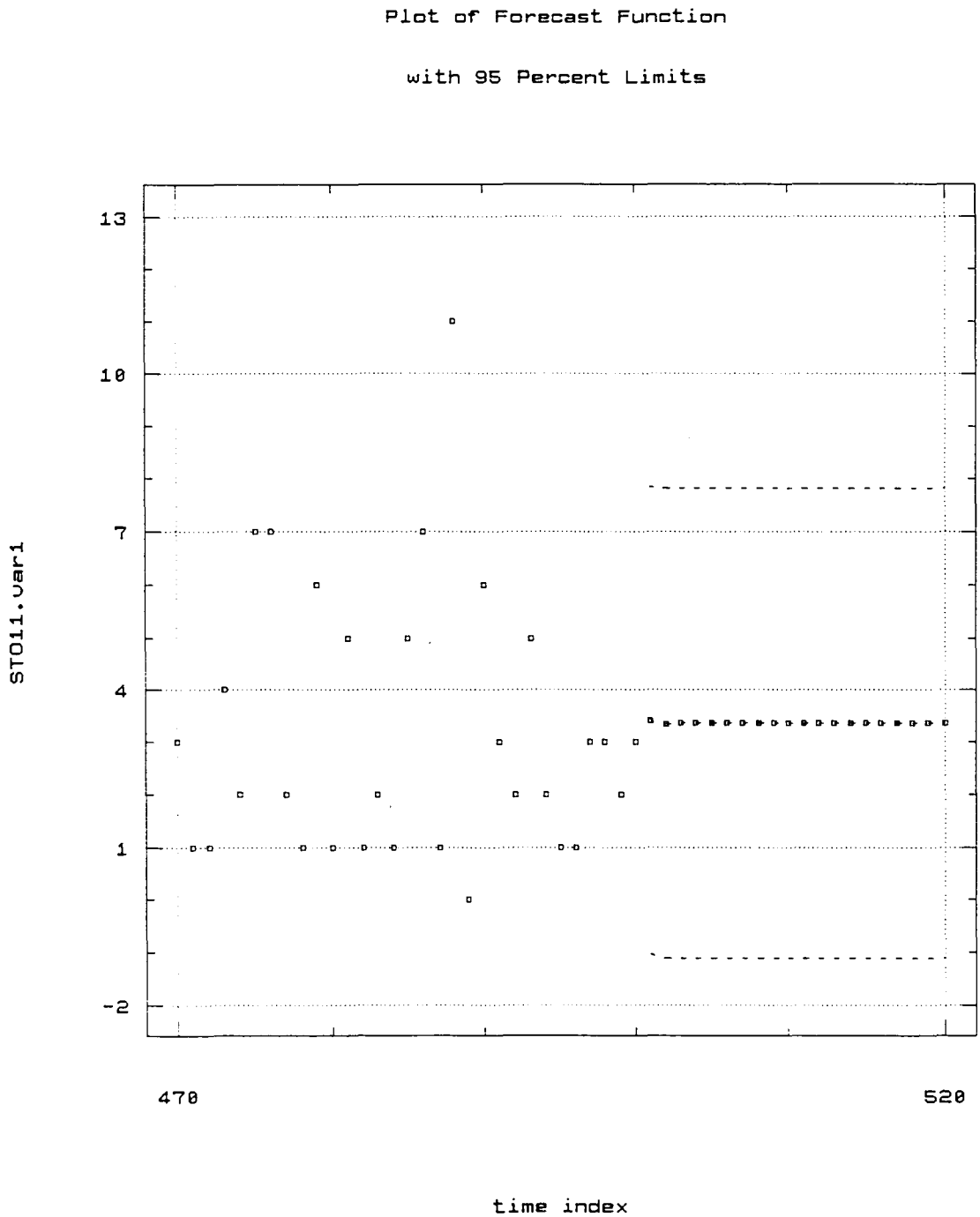
Plot of Forecast Function

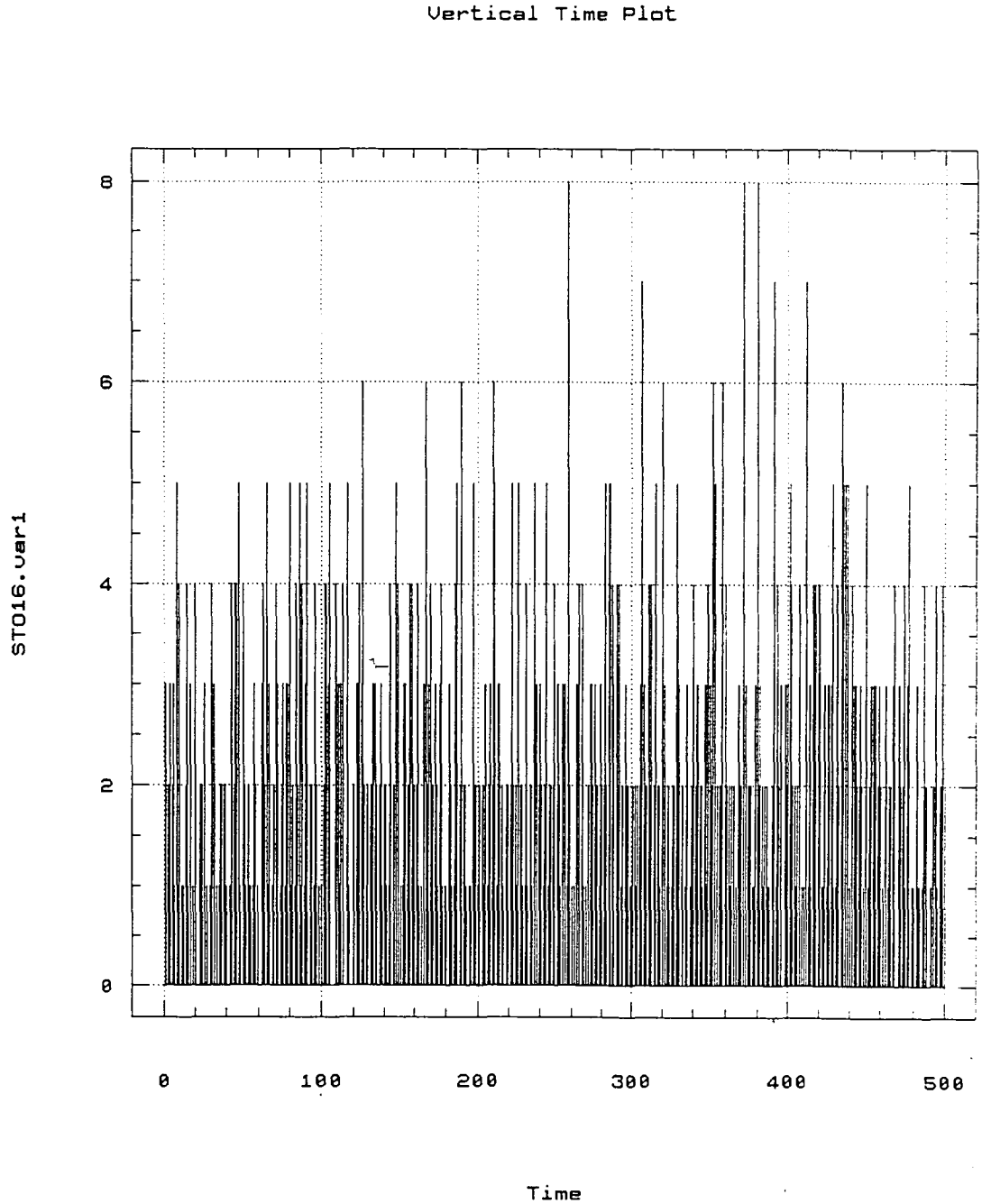
with 95 Percent Limits

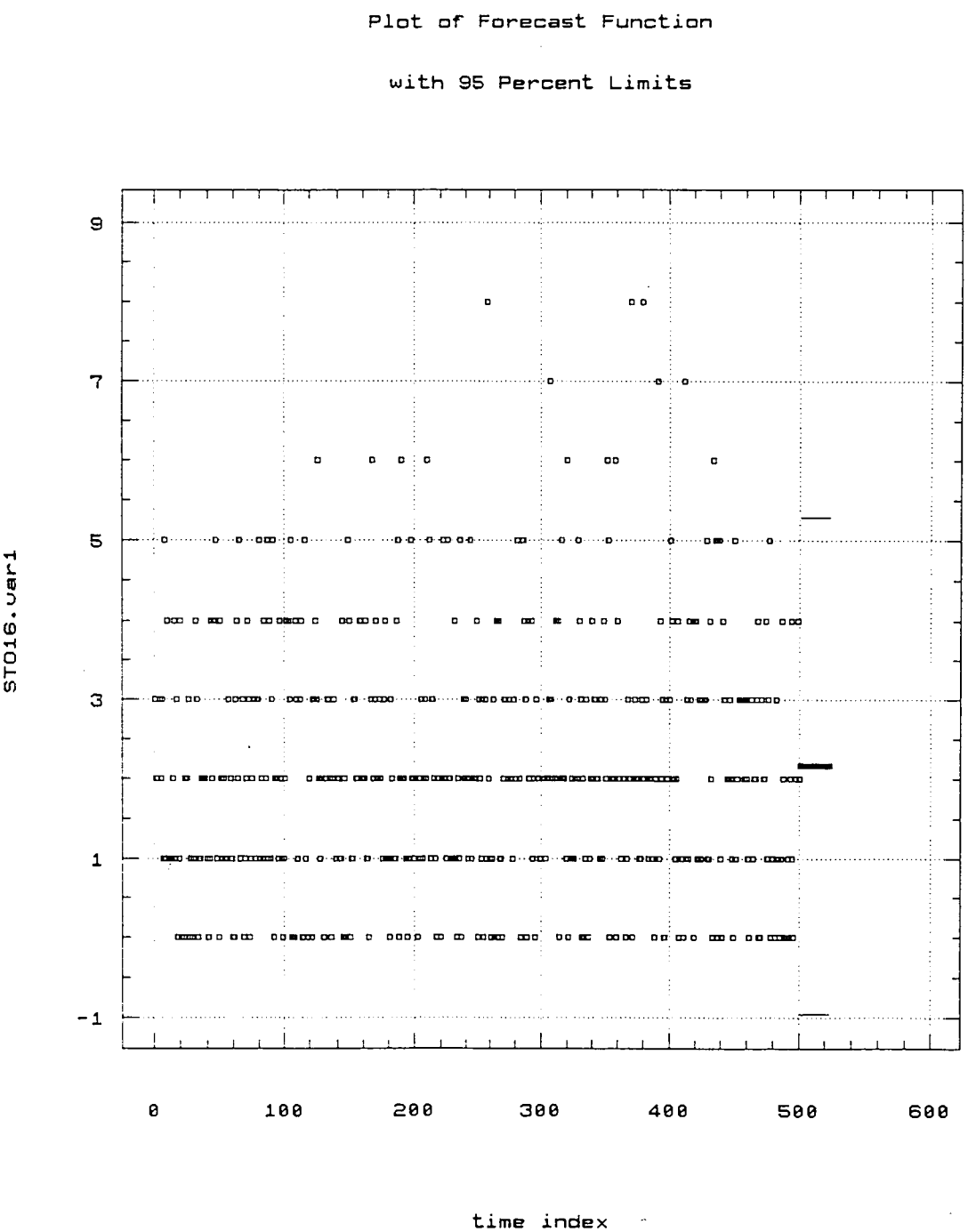



Vertical Time Plot
12
10
8
6
4
2
0
ST011.var1
0
100
200
300
400
500
Time


Plot of Forecast Function
with 95 Percent Limits
13
10
7
4
1
-2
STO11.var1
0
100
200
300
400
500
600
time index

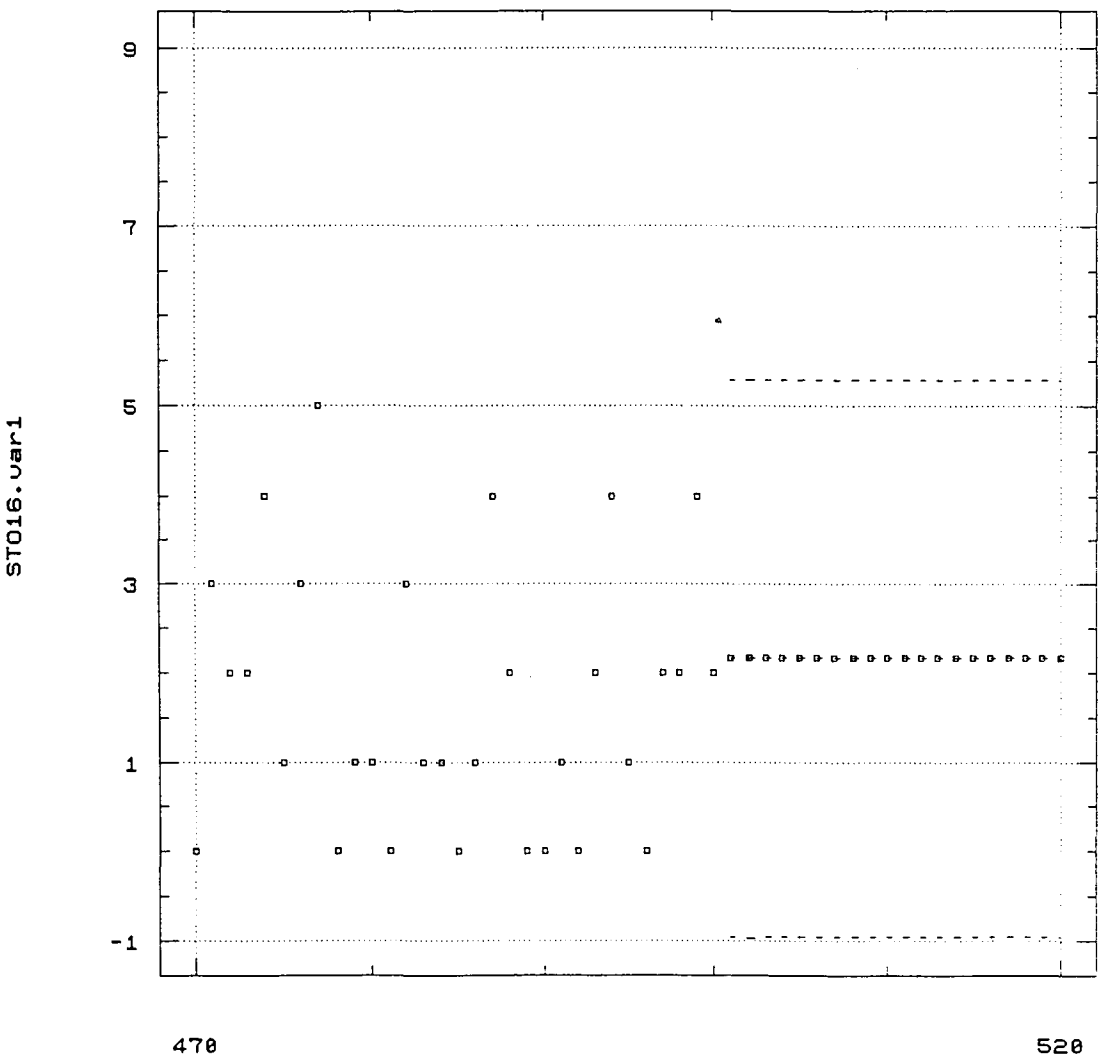

Plot of Forecast Function
with 95 Percent Limits
13
10
7
4
1
-2
ST011.var1
470
520
time index

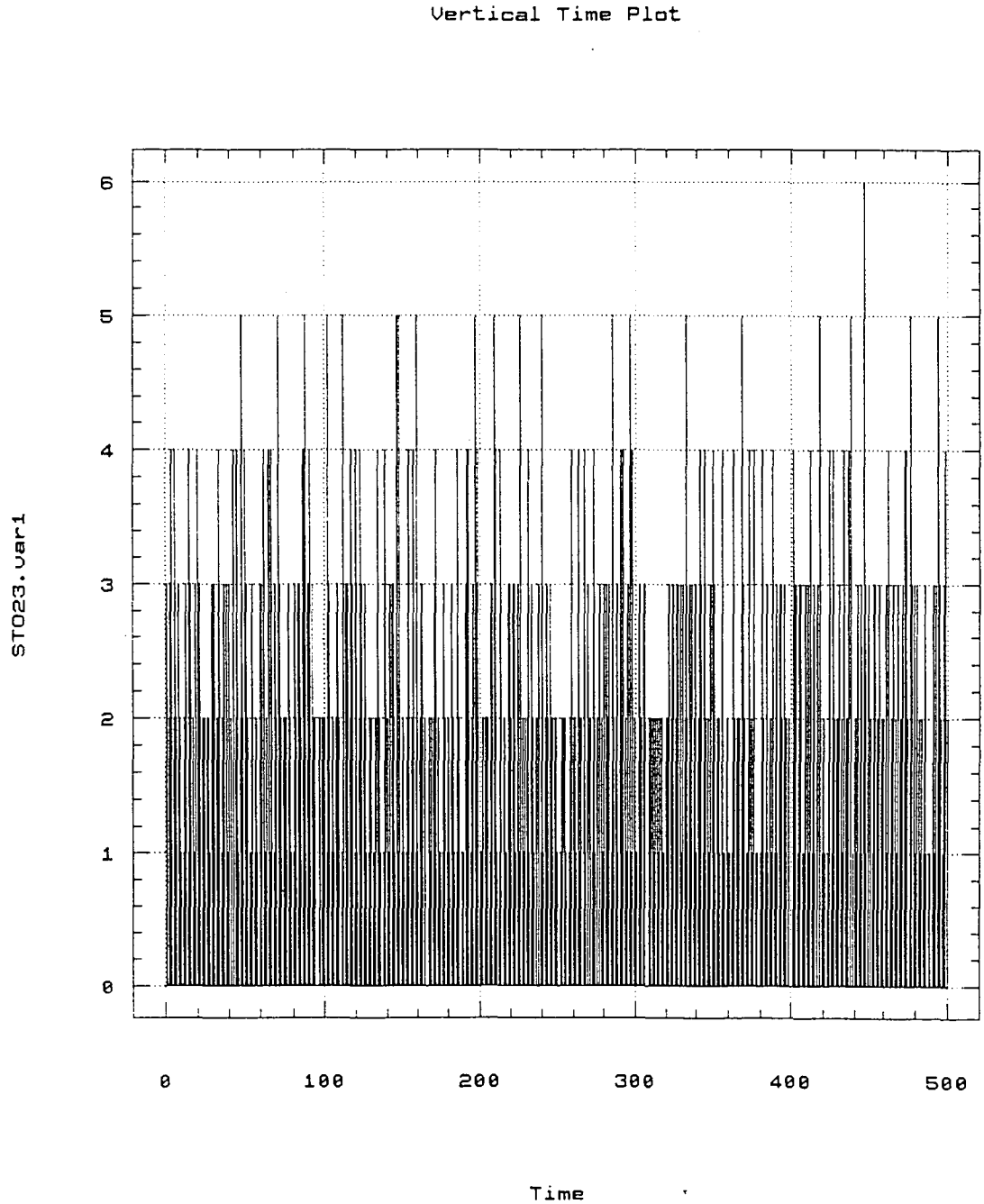

Vertical Time Plot
ST016.var1
8
6
4
2
0
0
100
200
300
400
500
Time


Plot of Forecast Function
with 95 Percent Limits
ST016.var1
9
7
5
3
1
-1
0
100
200
300
400
500
600
time index

Plot of Forecast Function

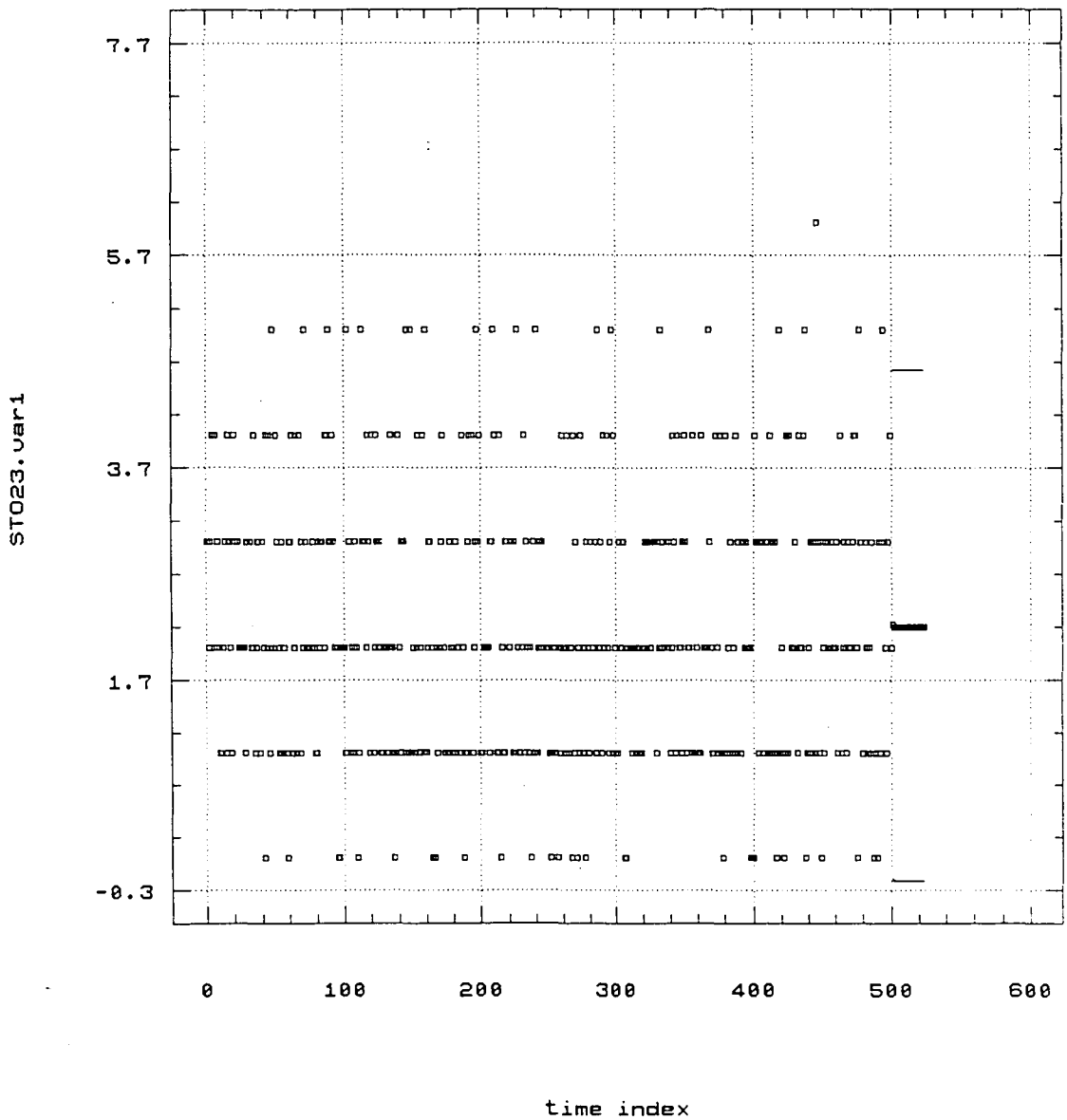
with 95 Percent Limits

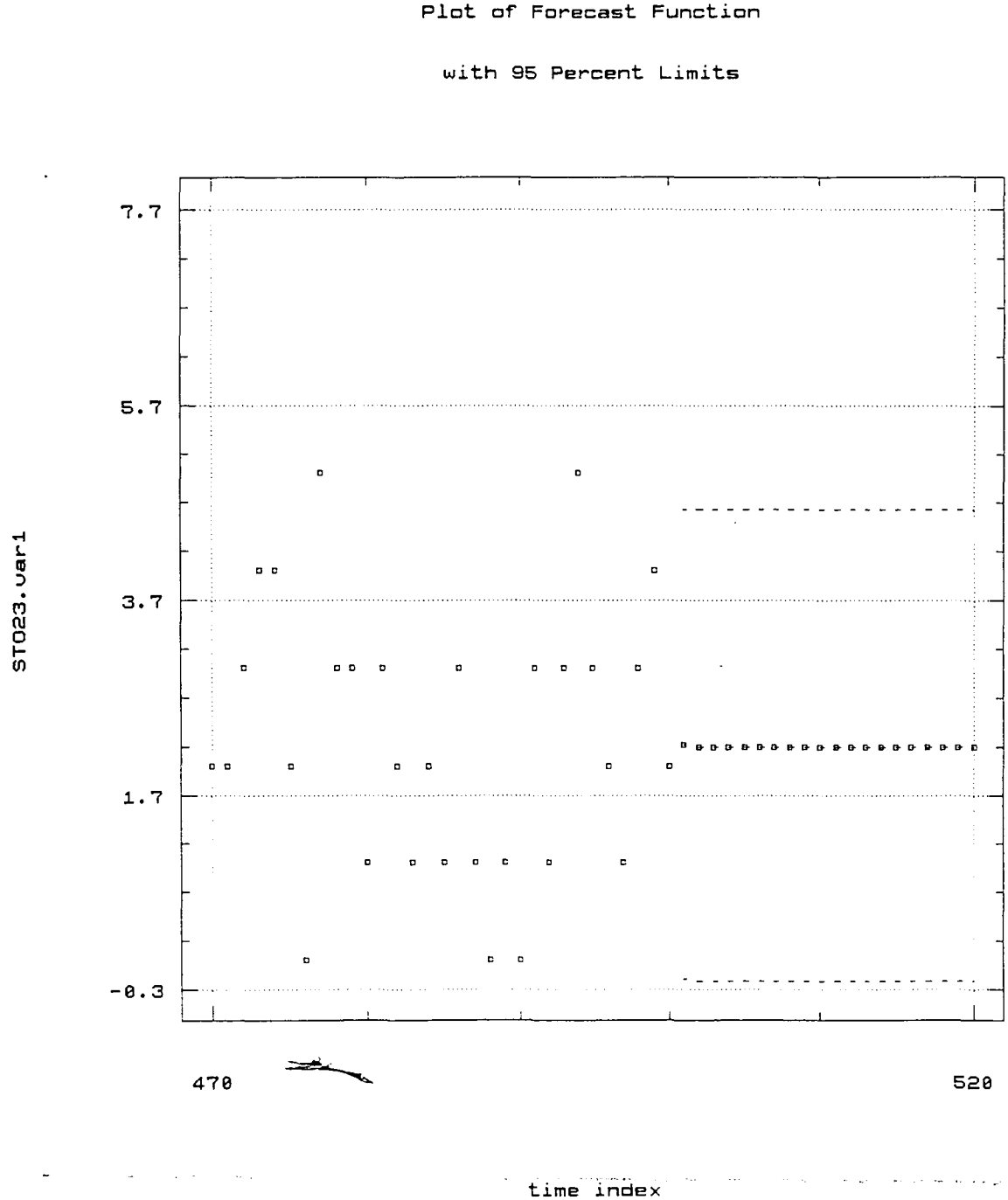


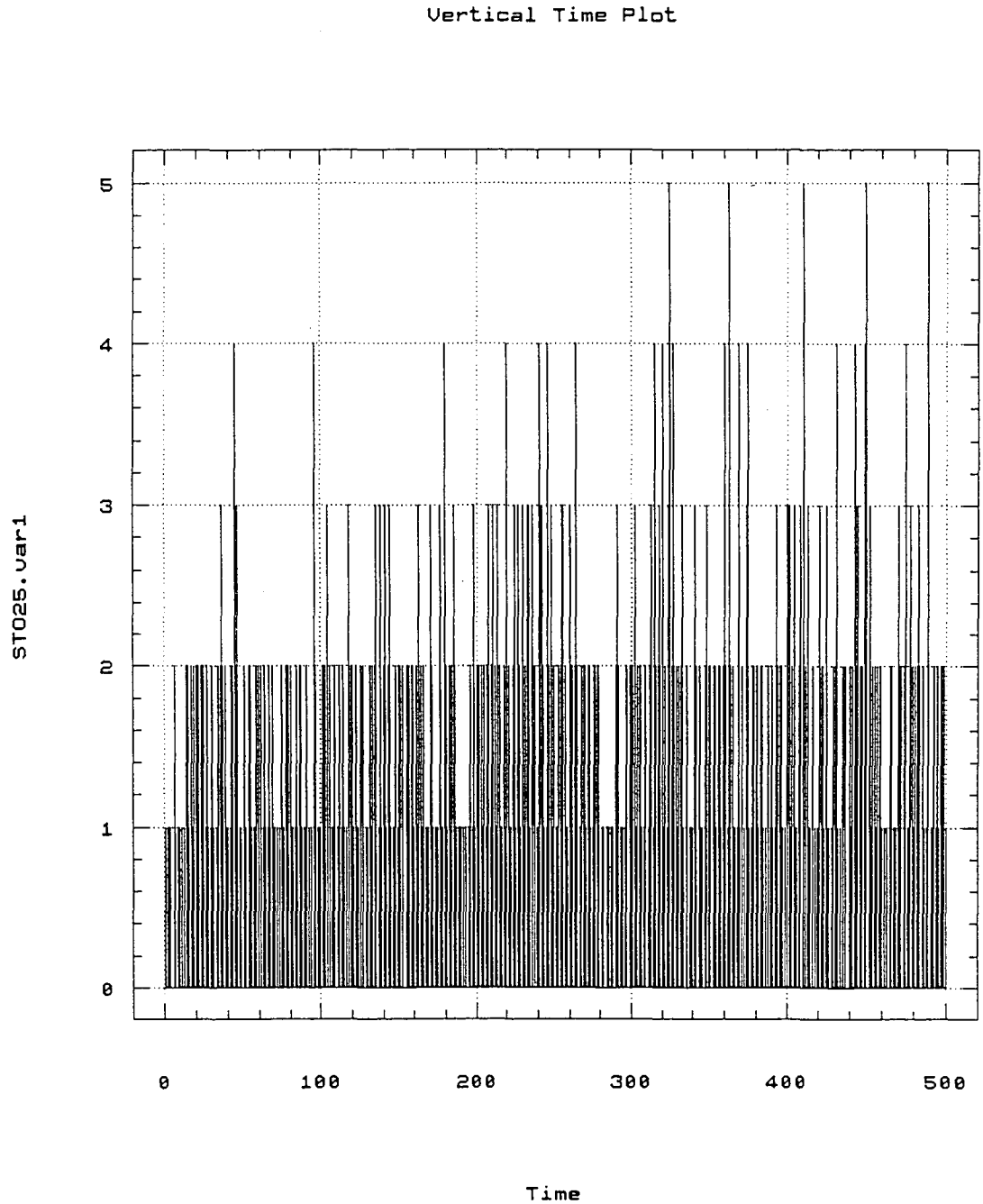

Vertical Time Plot
STO23.var1
6
5
4
3
2
1
0
0
100
200
300
400
500
Time

Plot of Forecast Function

with 95 Percent Limits

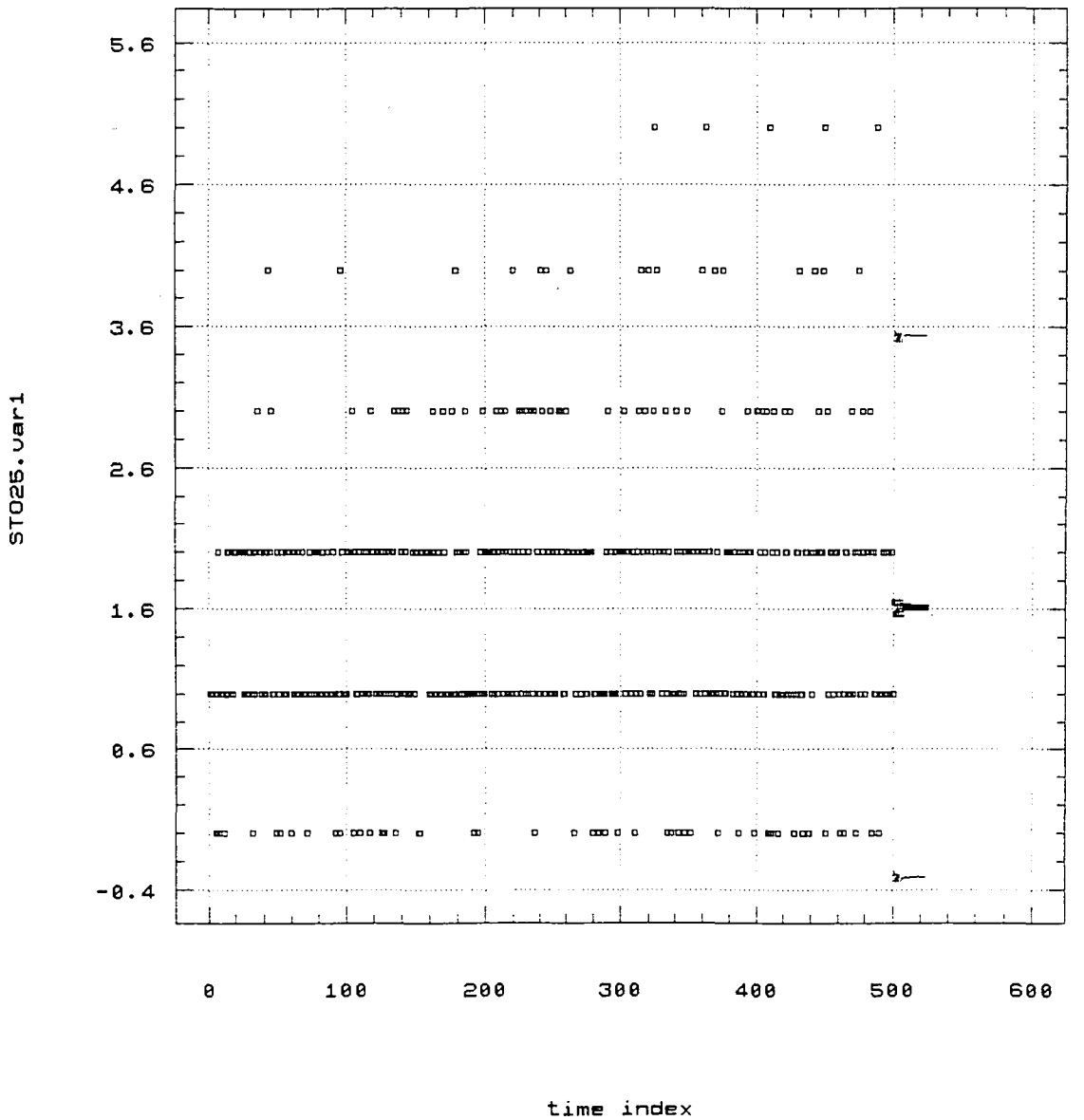


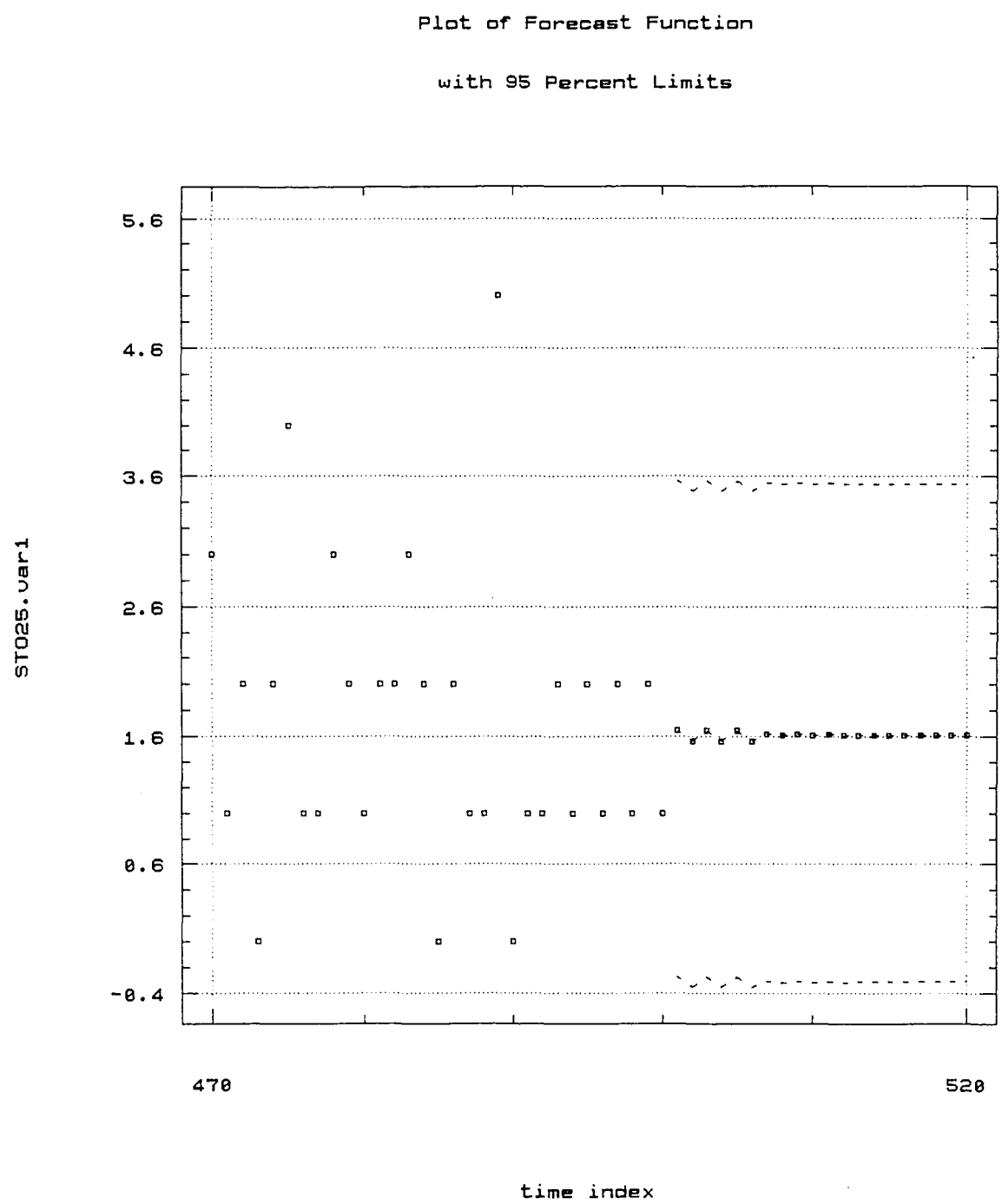

Plot of Forecast Function
with 95 Percent Limits
7.7
5.7
3.7
1.7
-0.3
ST023.var1
470
520
time index


Vertical Time Plot
ST025.var1
5
4
3
2
1
0
0
100
200
300
400
500
Time

Plot of Forecast Function

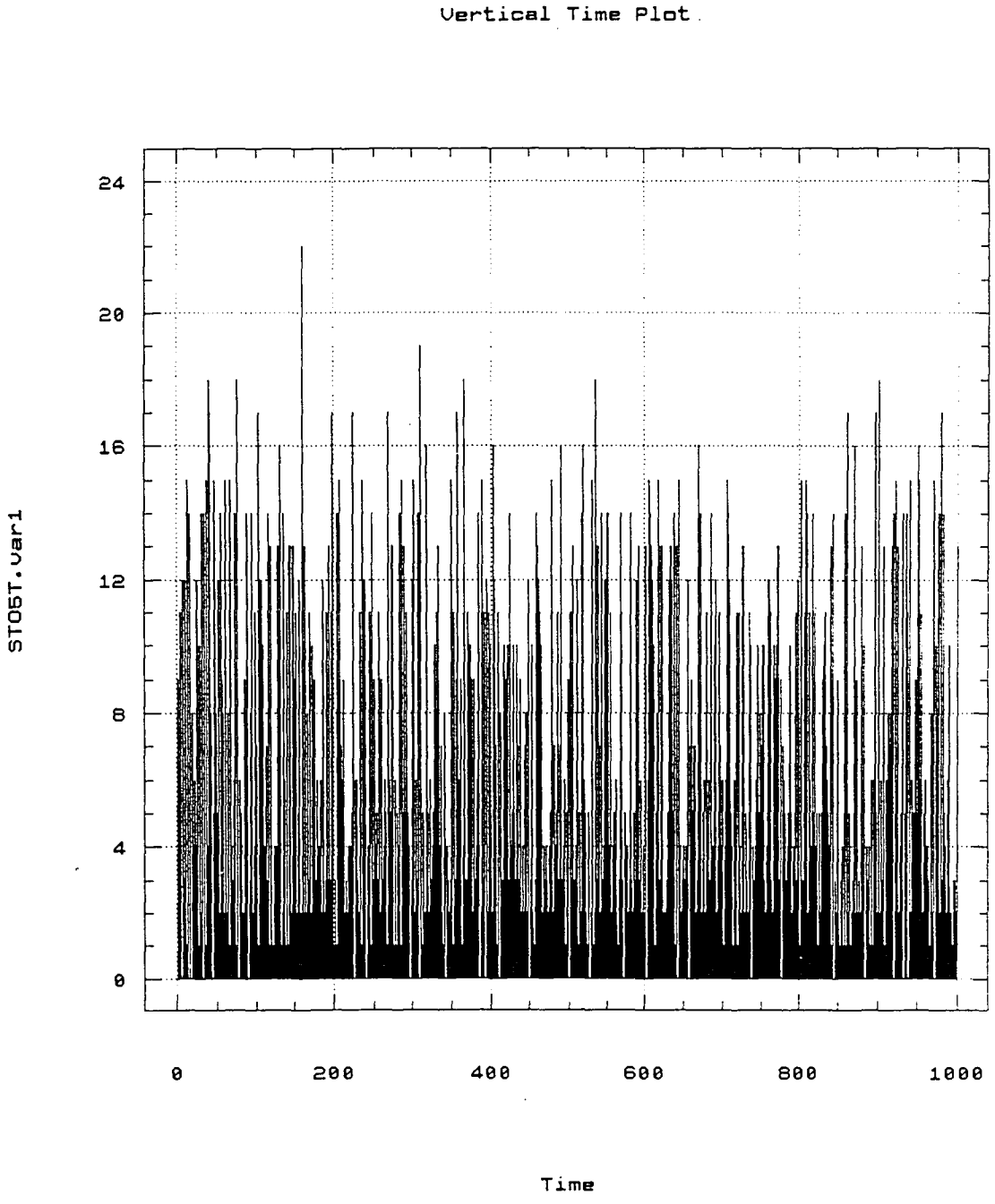
with 95 Percent Limits




Plot of Forecast Function
with 95 Percent Limits
5.6
4.6
3.6
2.6
1.6
0.6
-0.4
ST025.var1
470
520
time index

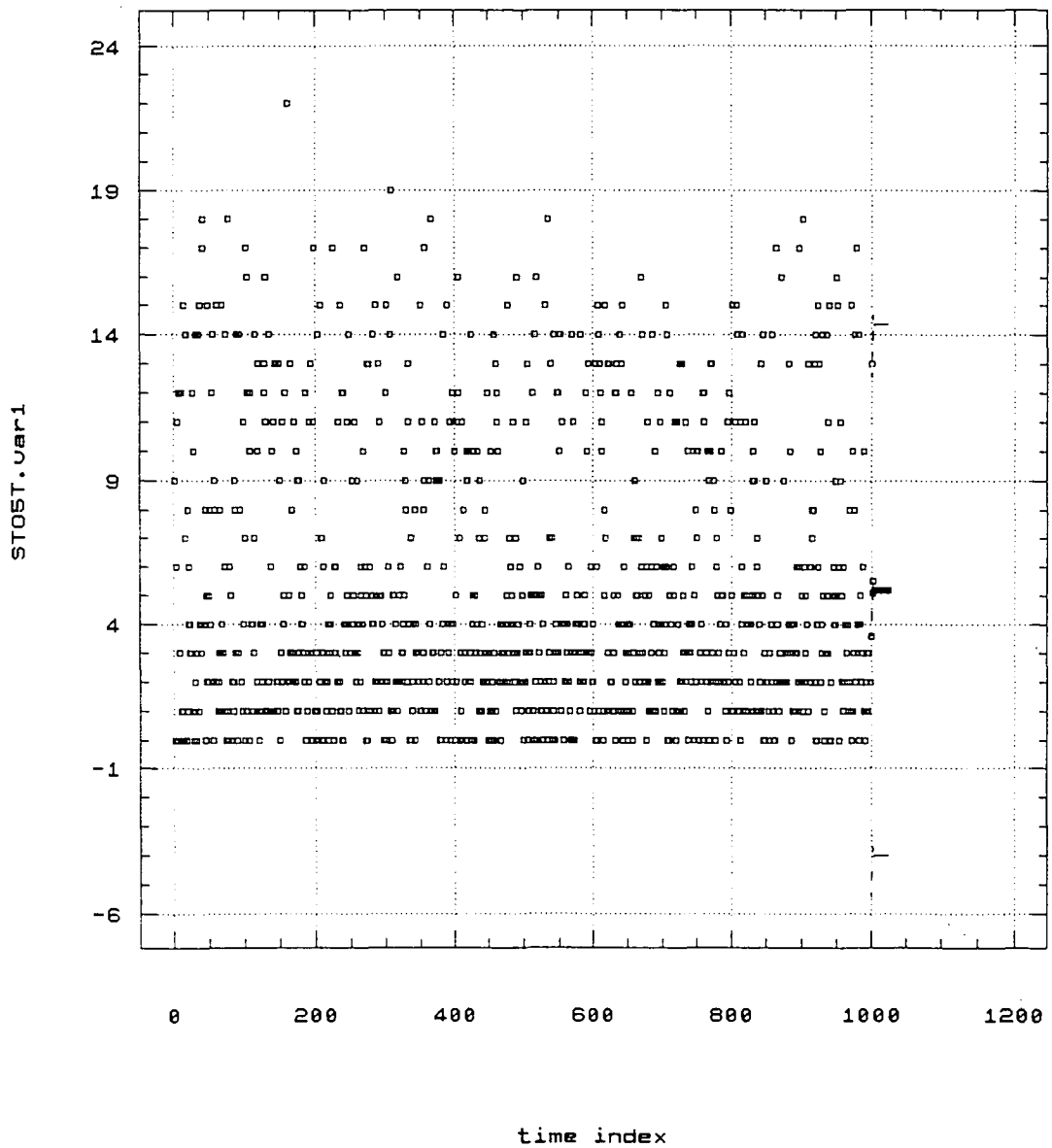
# APPENDIX C

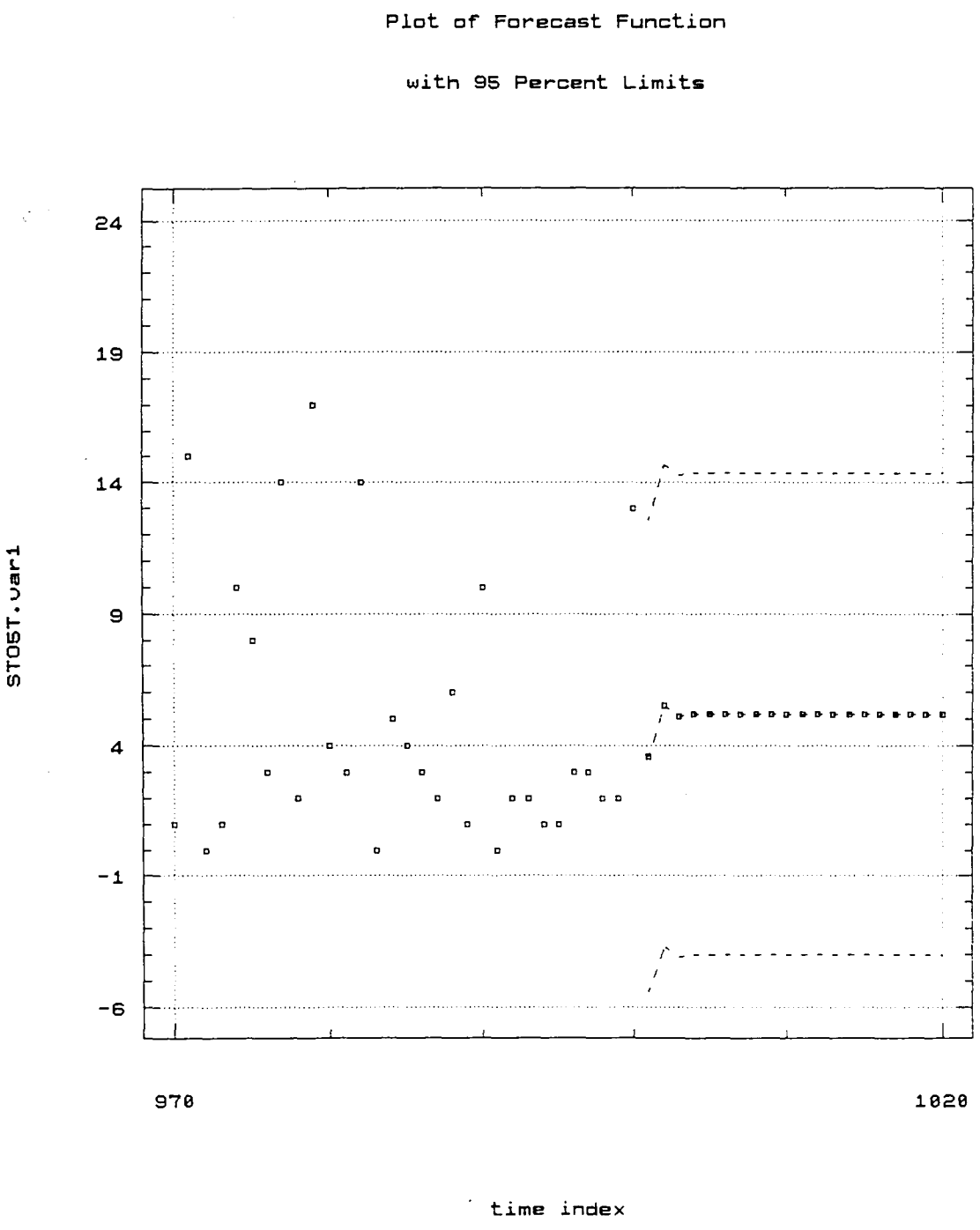
## RESULT OF SIMULATION FOR 1000 PERIODS

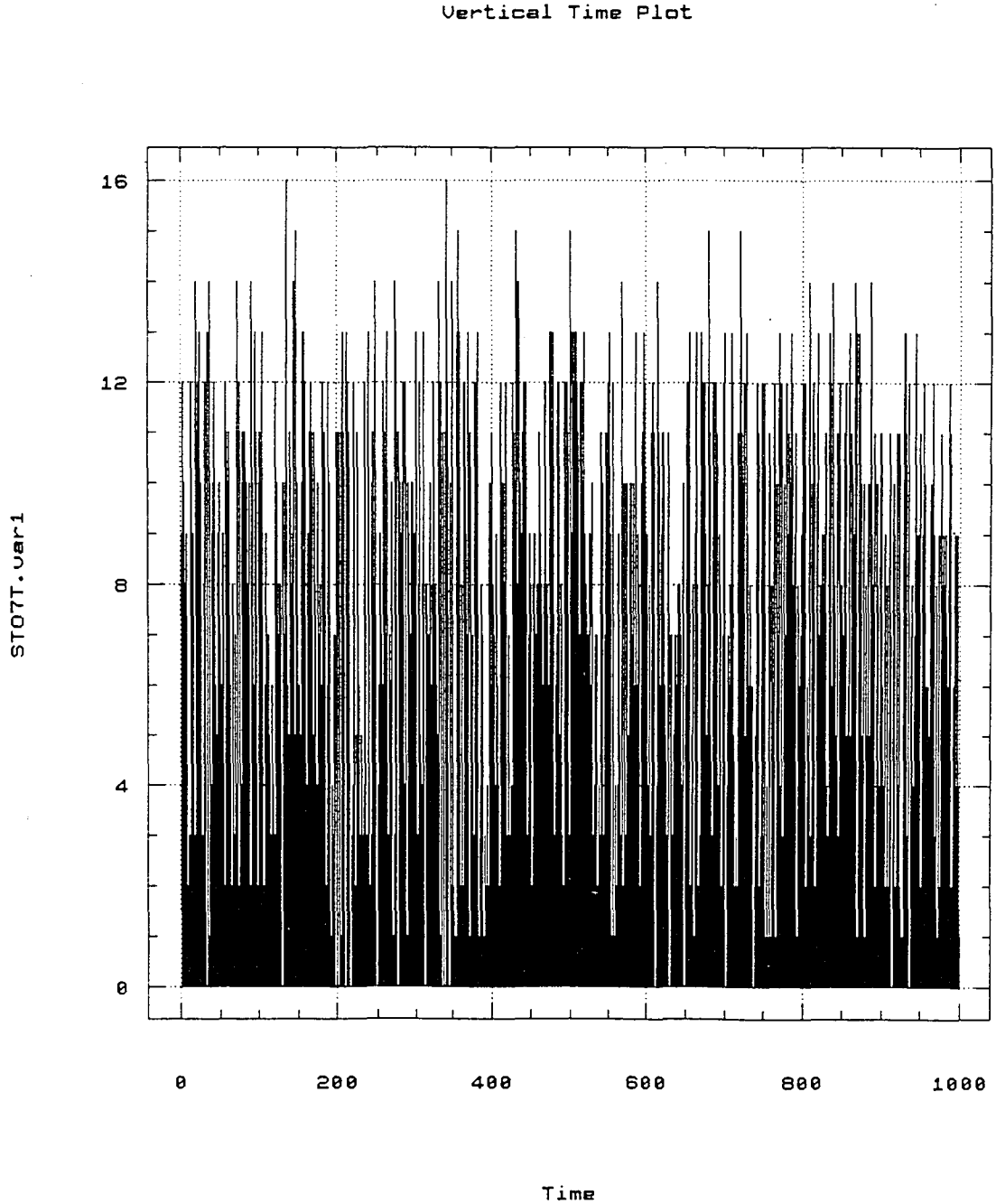

Vertical Time Plot
24
20
16
12
8
4
0
STO5T.var1
0
200
400
600
800
1000
Time

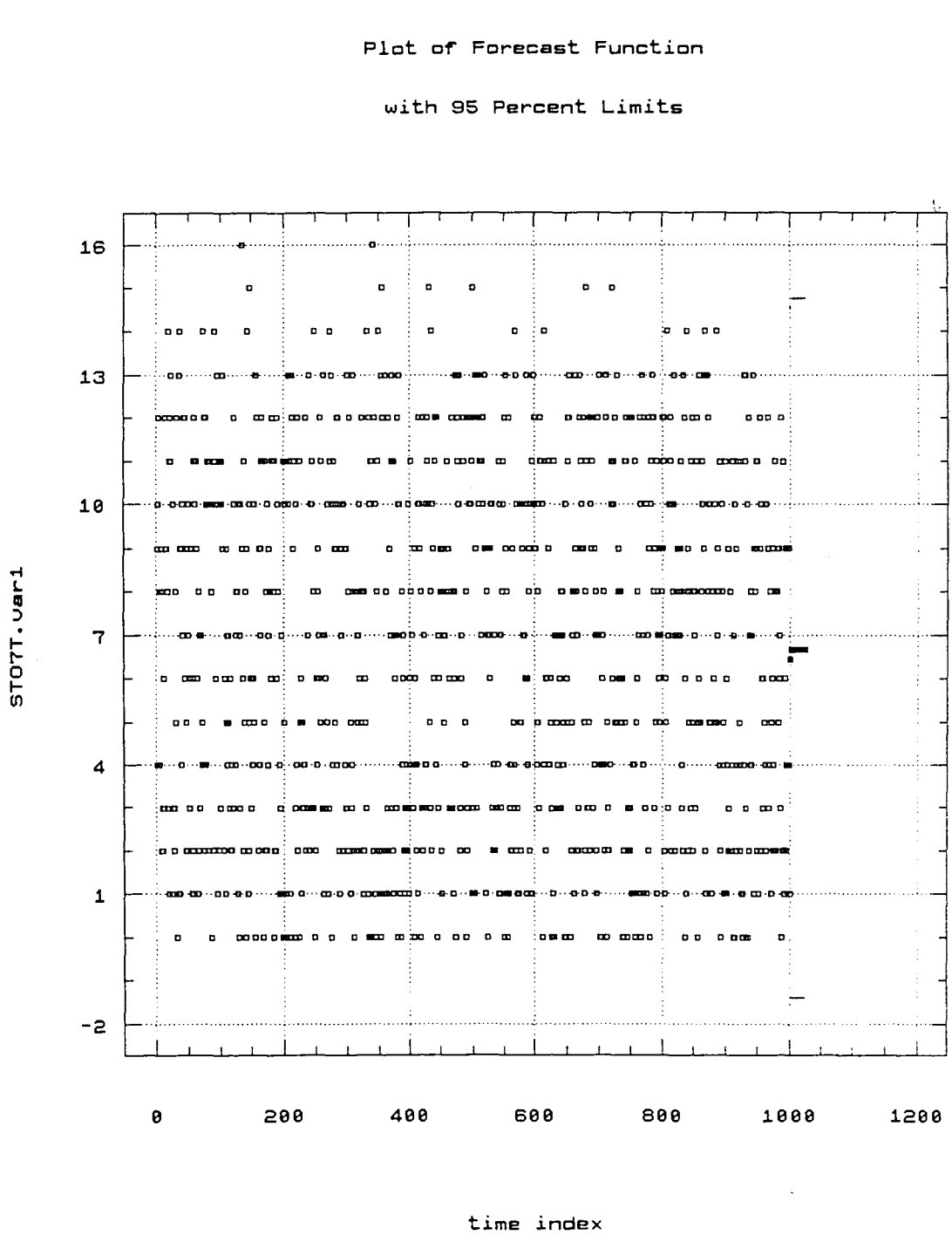
Plot of Forecast Function

with 95 Percent Limits



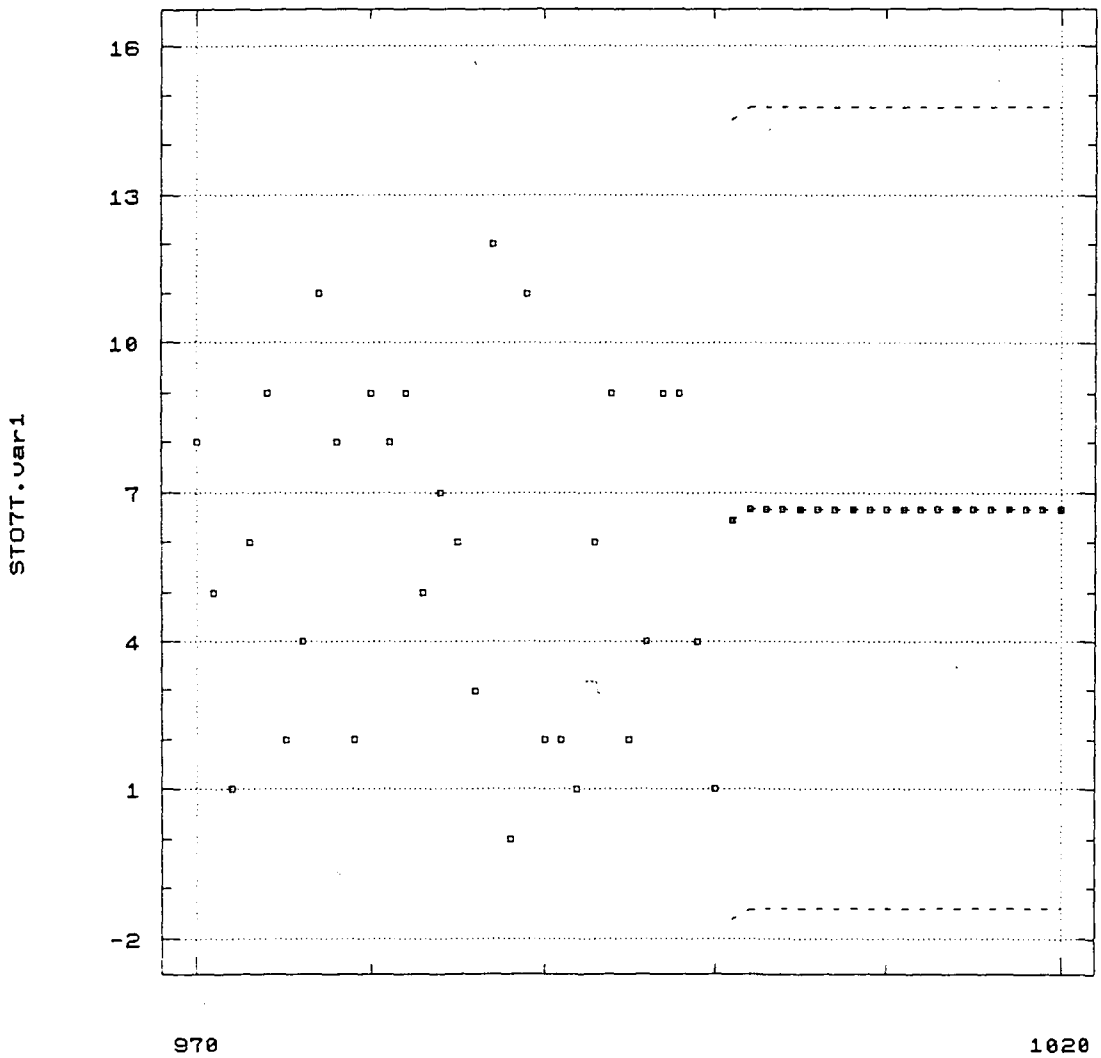

Plot of Forecast Function
with 95 Percent Limits
24
19
14
9
4
-1
-6
STOST.var1
970
1020
time index

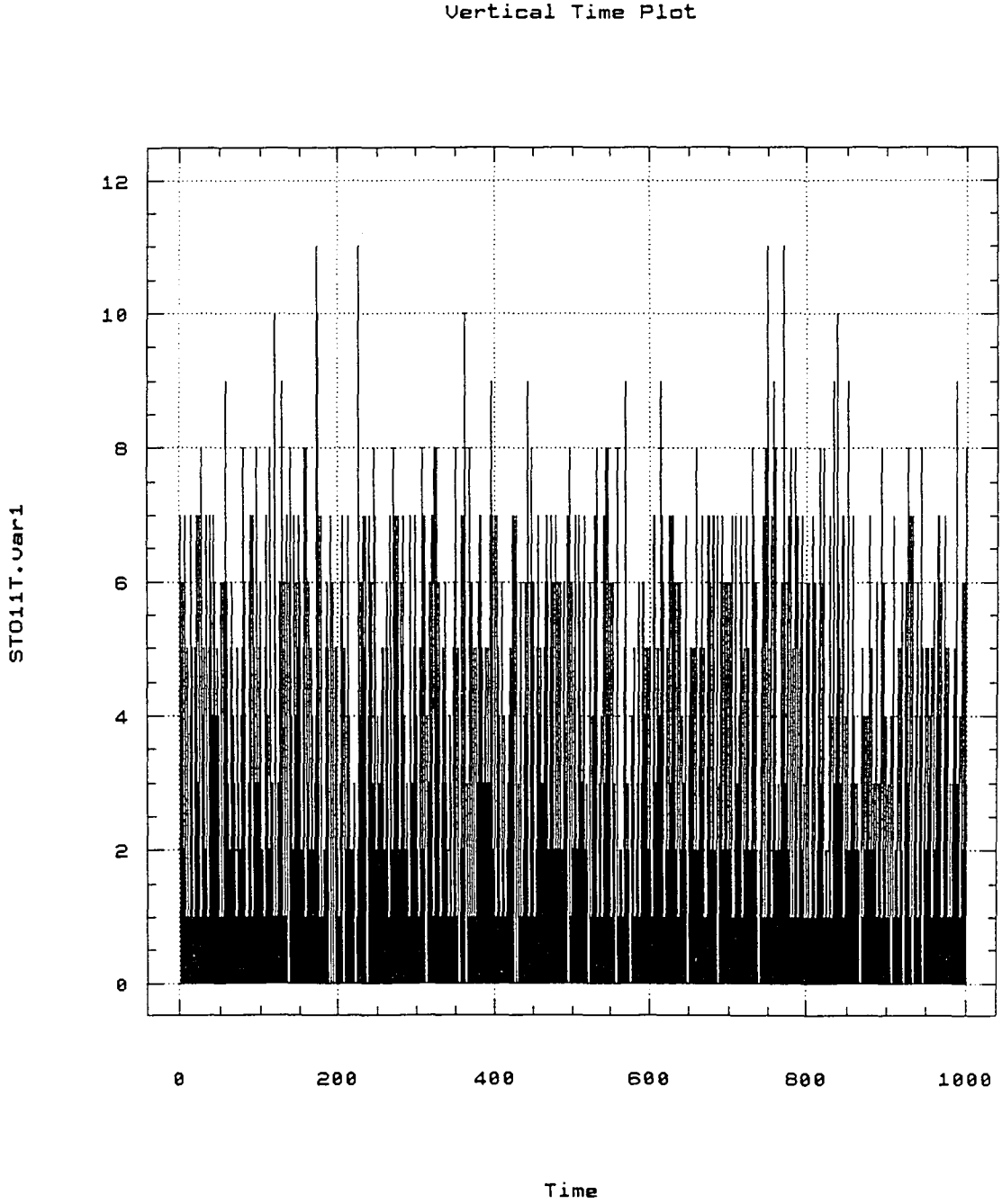

Vertical Time Plot
16
12
8
4
0
STO7T.var1
0
200
400
600
800
1000
Time

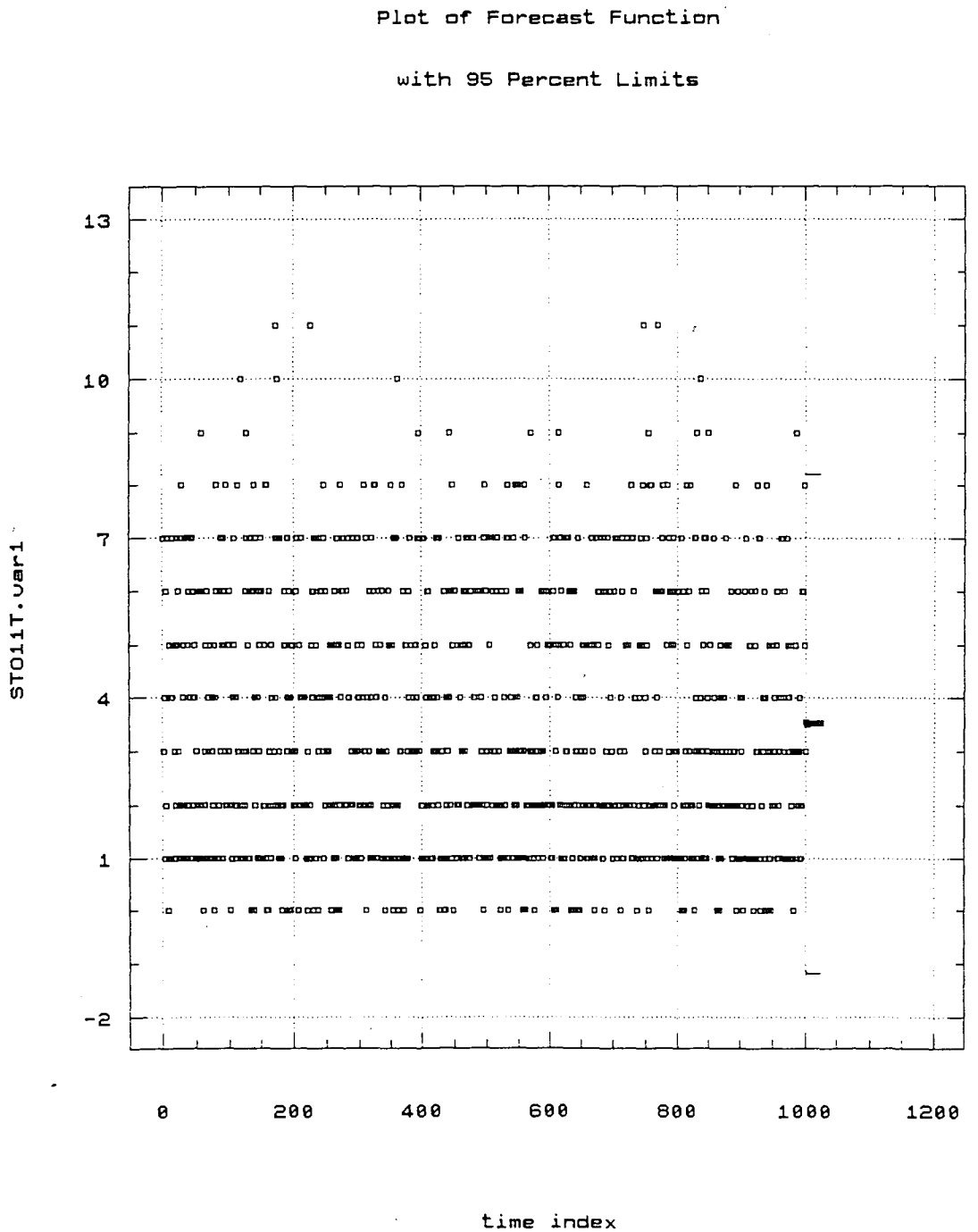

Plot of Forecast Function
with 95 Percent Limits
16
13
10
7
4
1
-2
STO7T.var1
0
200
400
600
800
1000
1200
time index

Plot of Forecast Function

with 95 Percent Limits

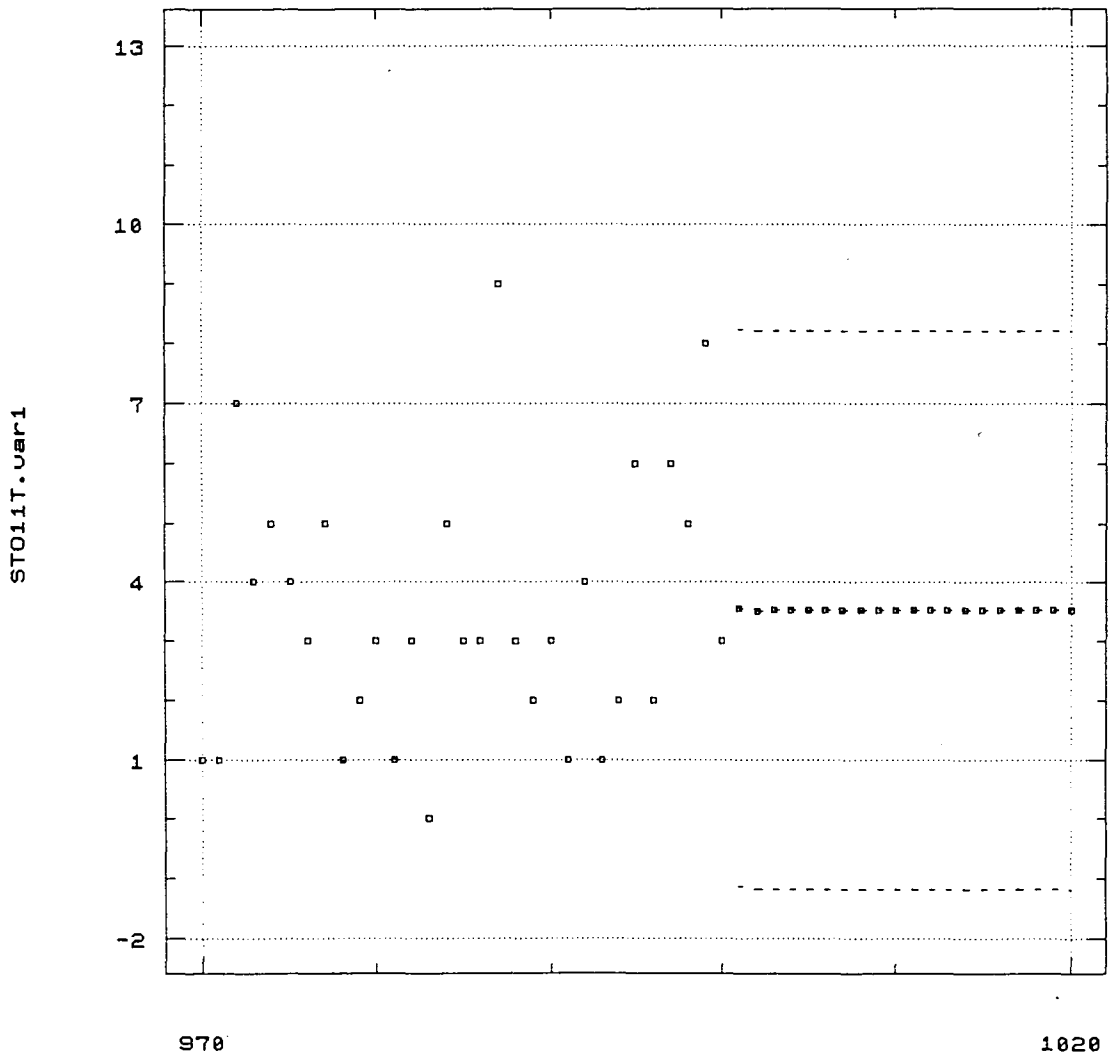


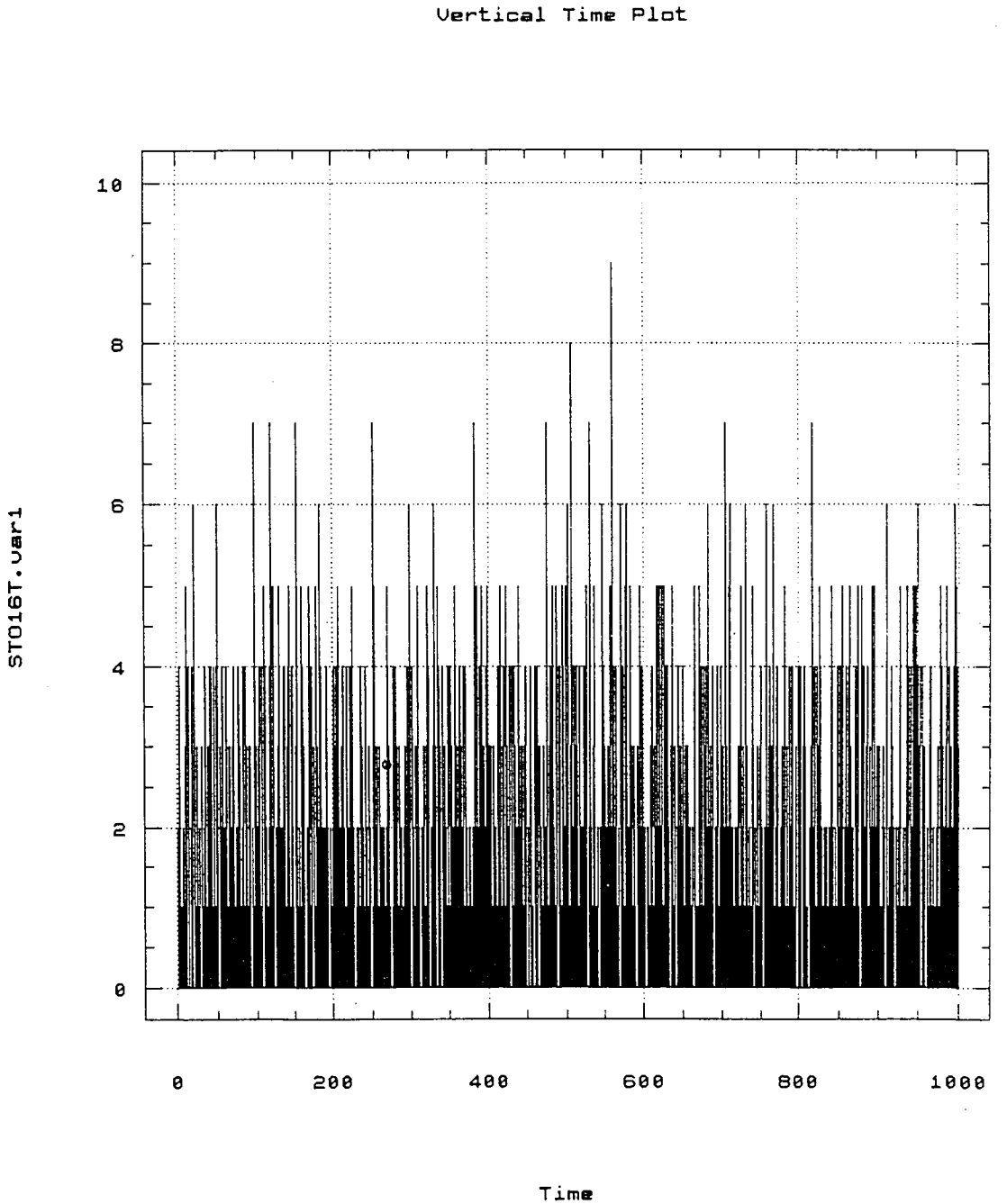

Vertical Time Plot
STO11T.var1
12
10
8
6
4
2
0
0
200
400
600
800
1000
Time

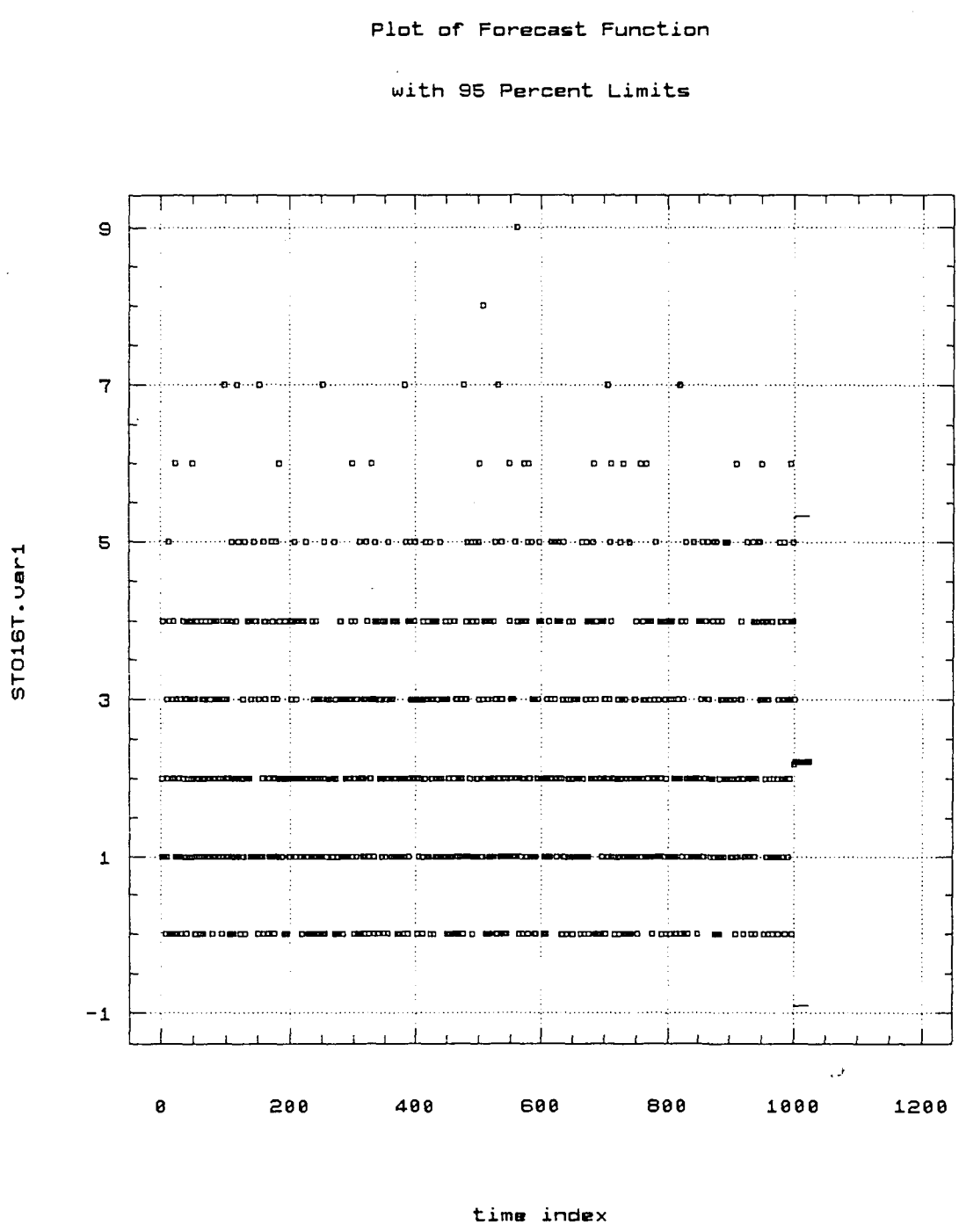

Plot of Forecast Function
with 95 Percent Limits
13
10
7
4
1
-2
ST011T.var1
0
200
400
600
800
1000
1200
time index

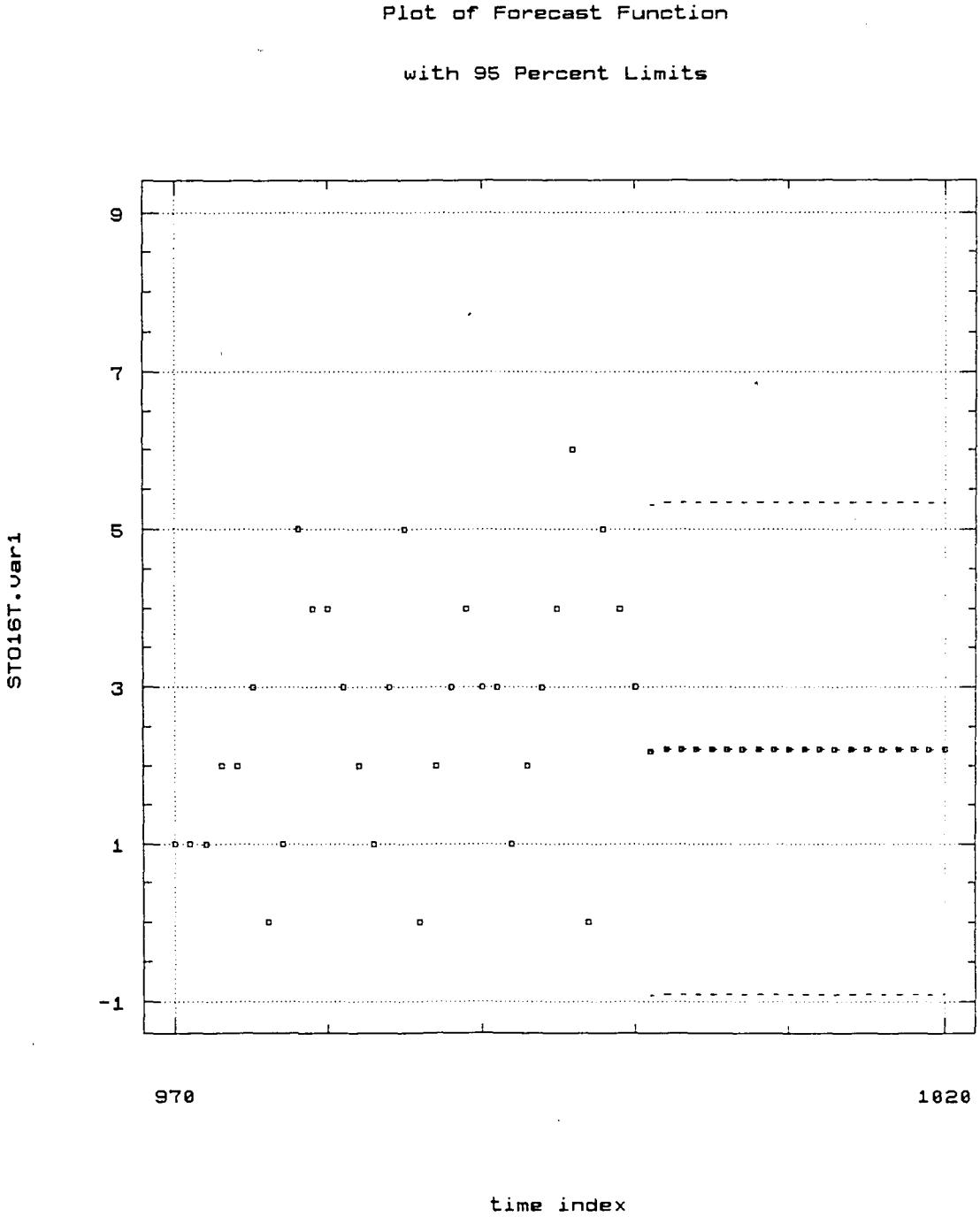
Plot of Forecast Function

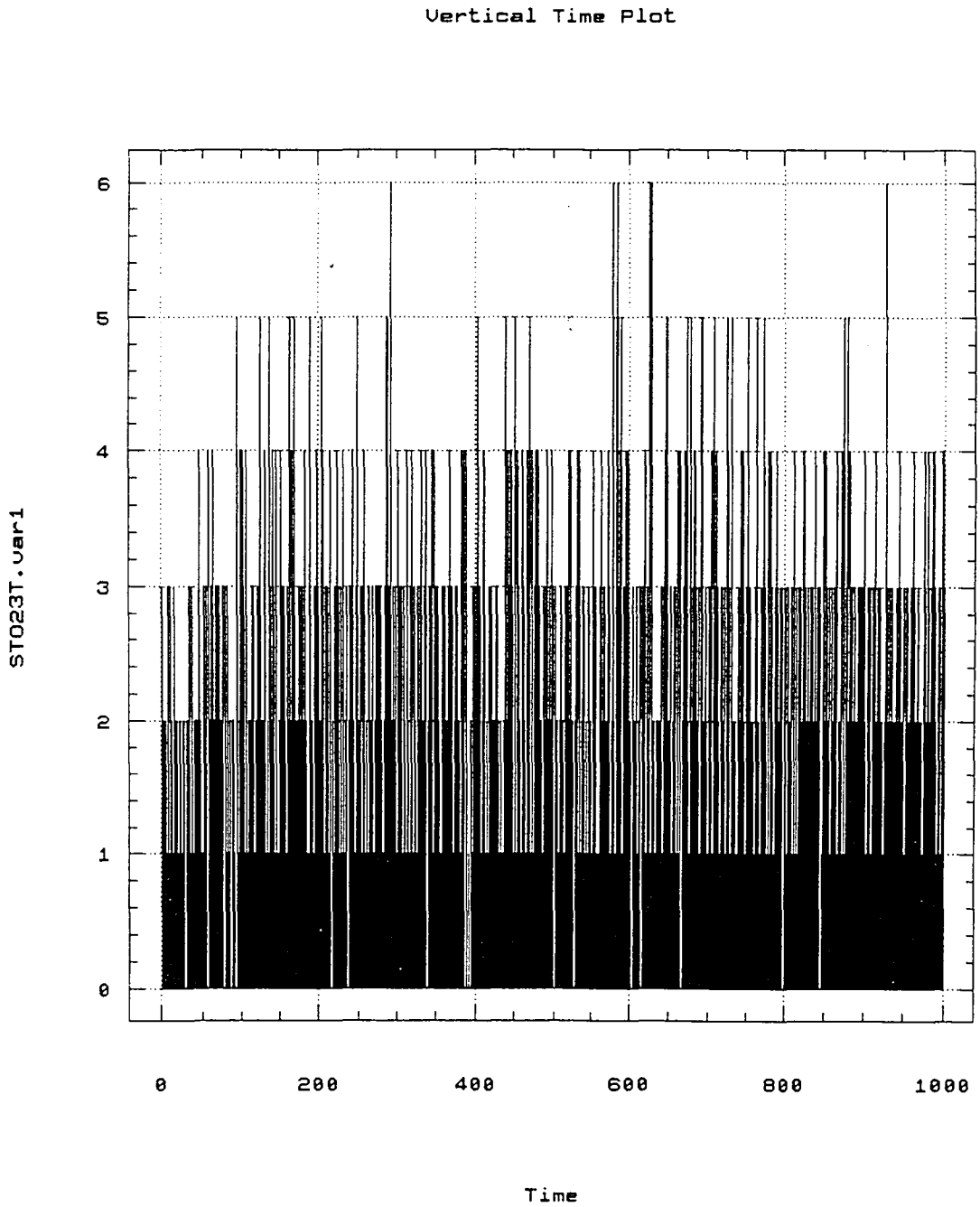
with 95 Percent Limits

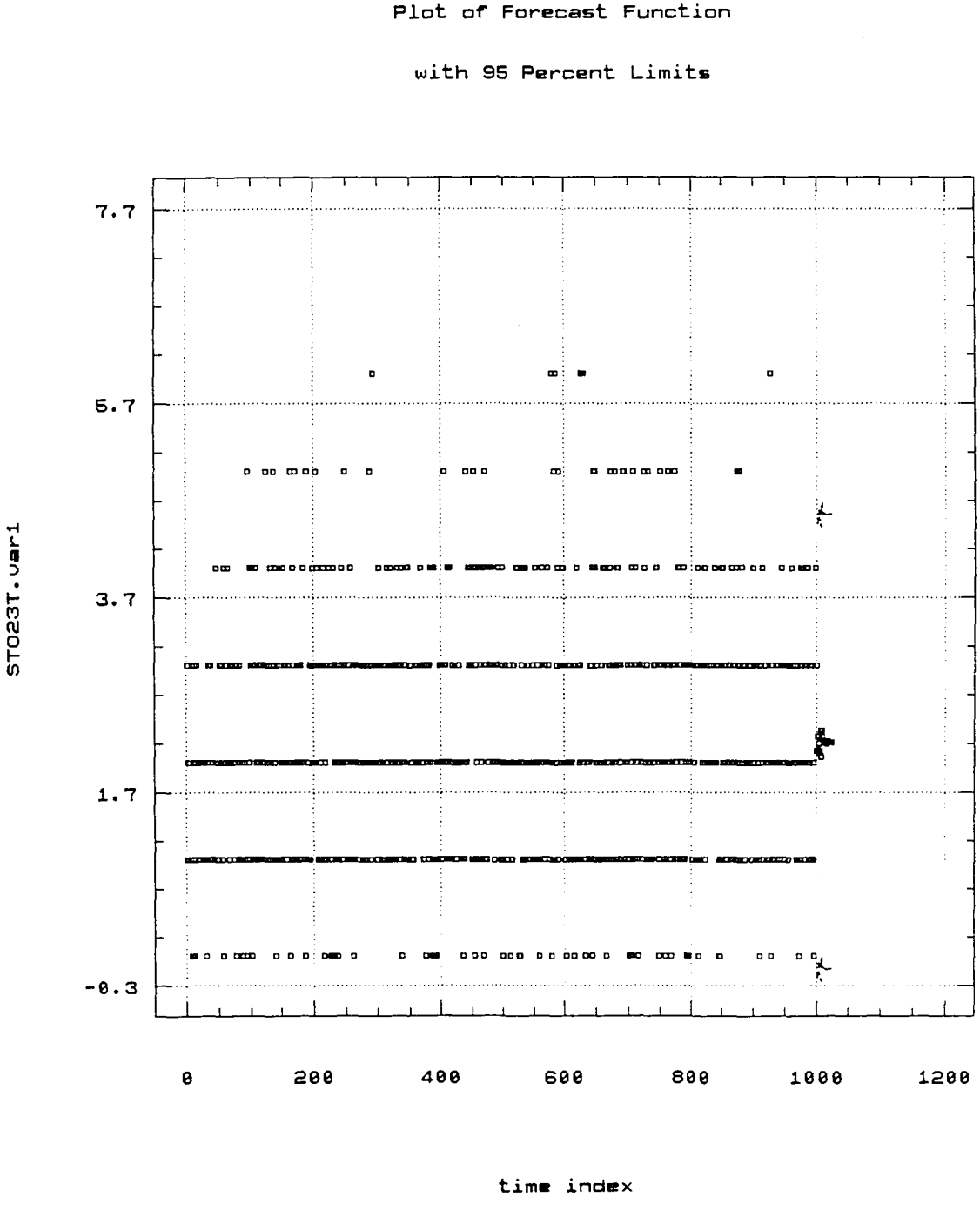


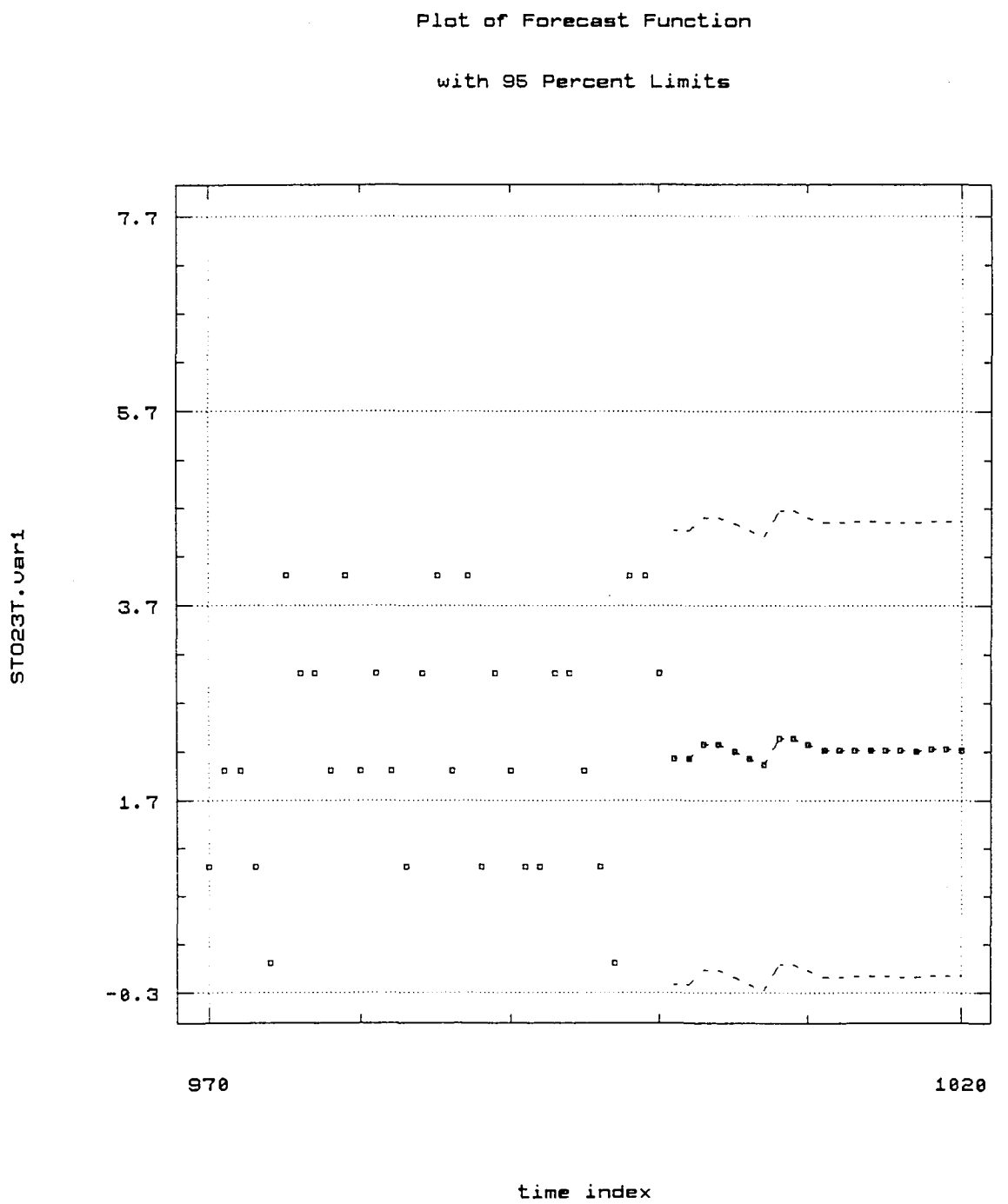

Vertical Time Plot
10
8
6
4
2
0
ST016T.var1
0
200
400
600
800
1000
Time

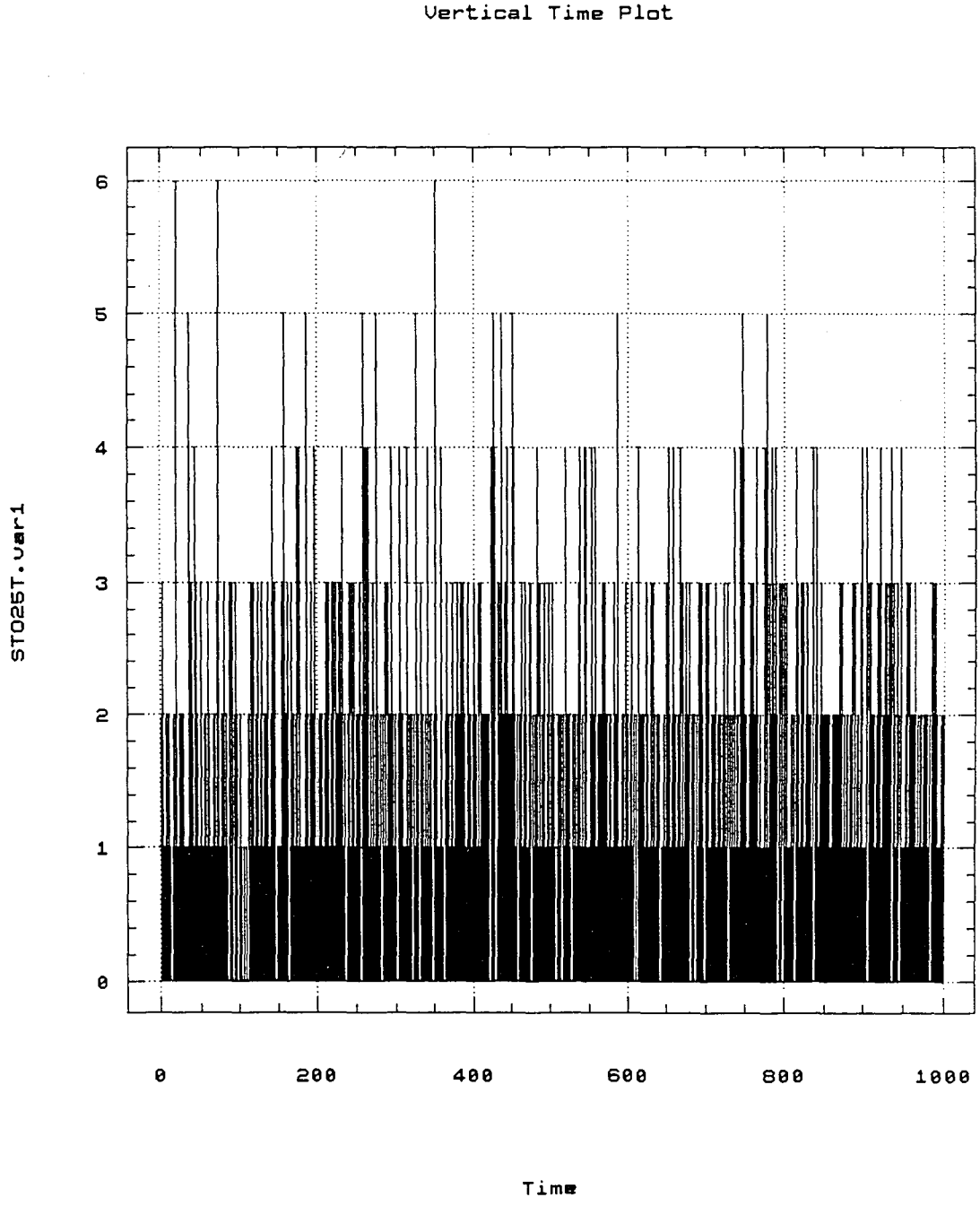

Plot of Forecast Function
with 95 Percent Limits
STO16T.var1
9
7
5
3
1
-1
0
200
400
600
800
1000
1200
time index

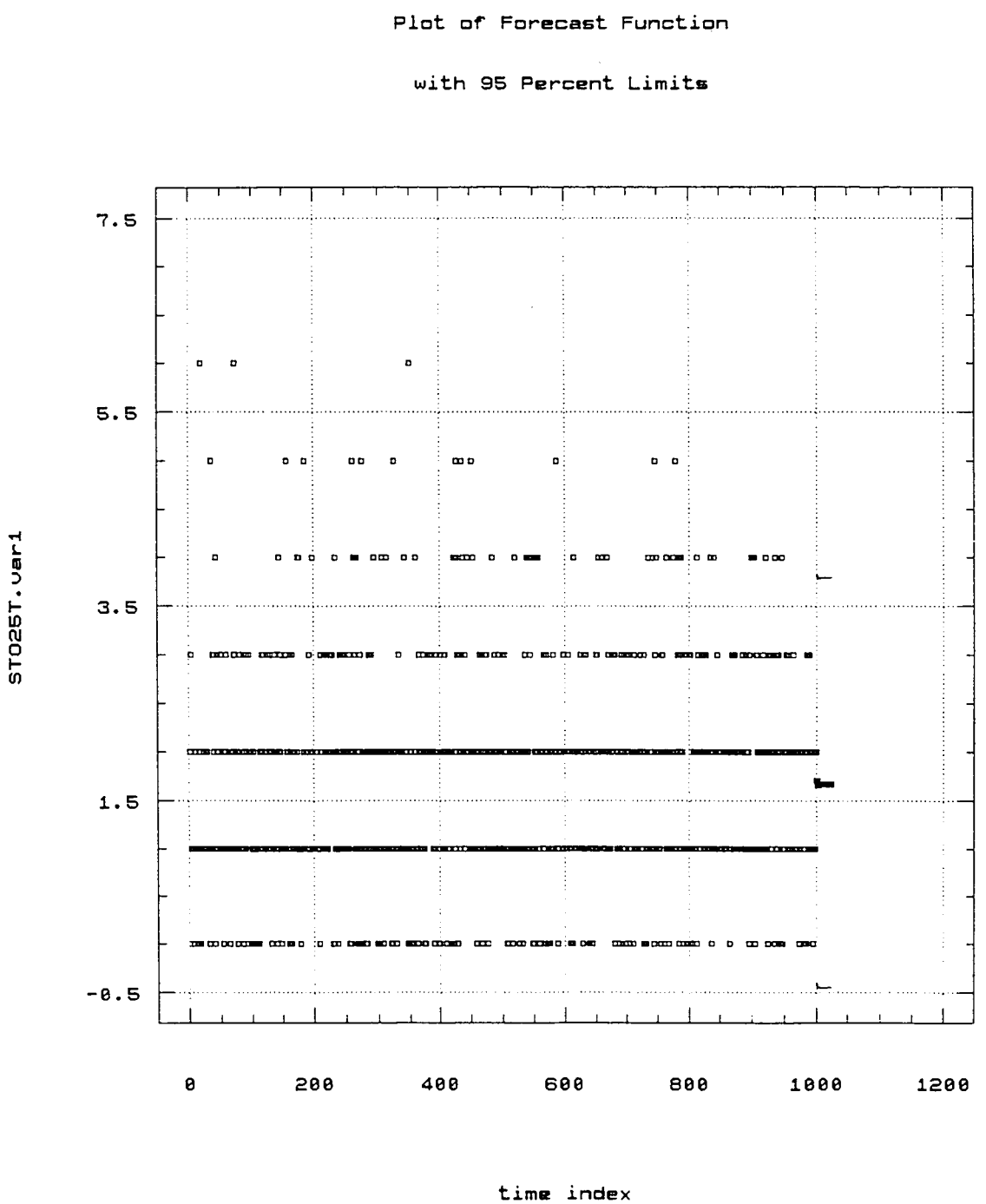

Plot of Forecast Function
with 95 Percent Limits
9
7
5
3
1
-1
ST016T.var1
970
1020
time index

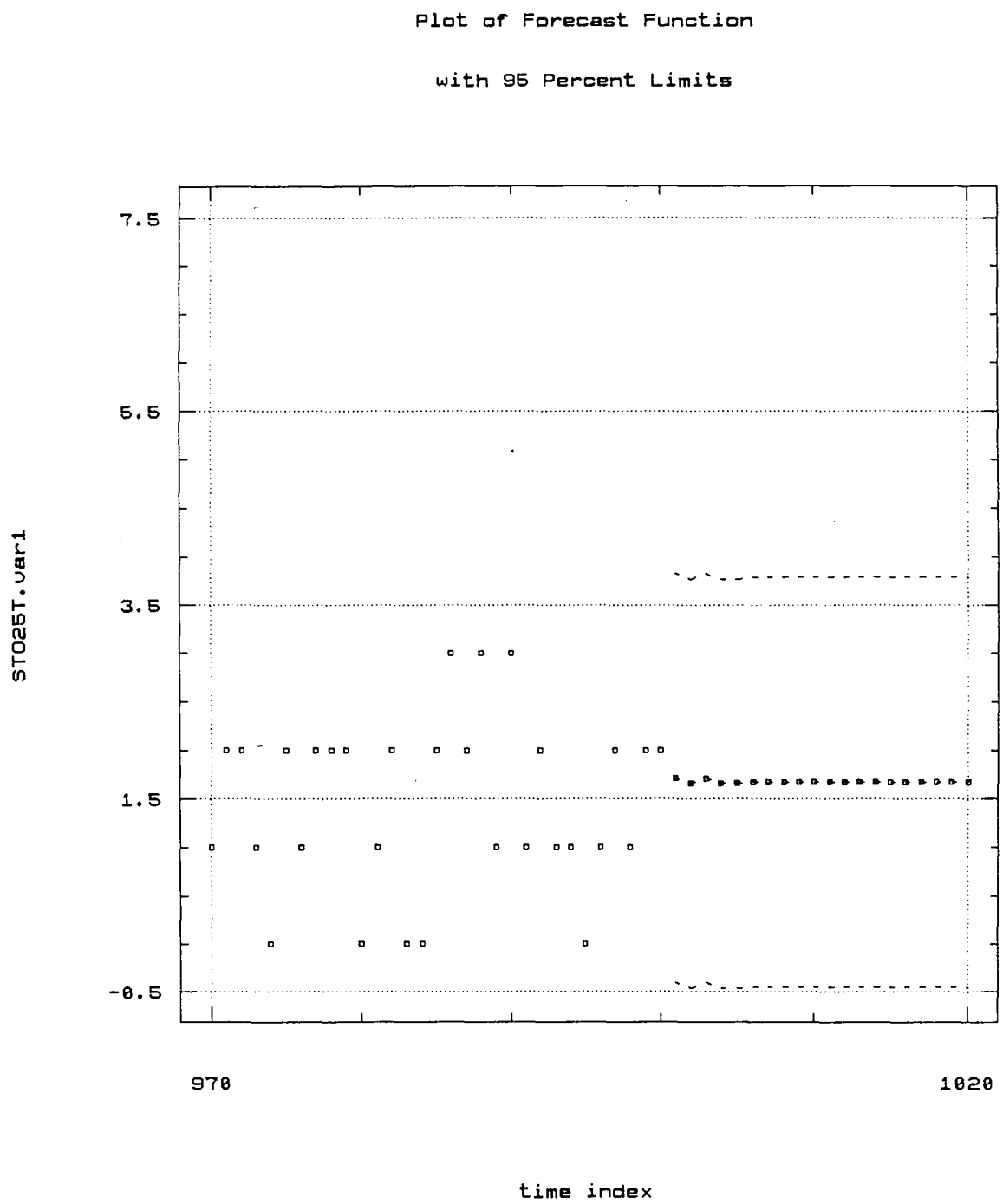

Vertical Time Plot
ST023T.var1
6
5
4
3
2
1
0
0
200
400
600
800
1000
Time


Plot of Forecast Function
with 95 Percent Limits
7.7
5.7
3.7
1.7
-0.3
ST023T.var1
0
200
400
600
800
1000
1200
time index


Plot of Forecast Function
with 95 Percent Limits
7.7
5.7
3.7
1.7
-0.3
ST023T.var1
970
1020
time index


Vertical Time Plot
STO25T.var1
6
5
4
3
2
1
0
0
200
400
600
800
1000
Time


Plot of Forecast Function
with 95 Percent Limits
7.5
5.5
3.5
1.5
-0.5
STO25T.var1
0
200
400
600
800
1000
1200
time index


Plot of Forecast Function
with 95 Percent Limits
7.5
5.5
3.5
1.5
-0.5
ST025T.var1
970
1020
time index

# VITA

Ms. Pei-chun Lin who is the daughter of Chun-Jeng Lin and Hsu-Chien Lin was born in Taiwan in 1970. She received a B.S. degree in Industrial Engineering from National Tsing-Hua University in 1992. After her graduation from the college, she entered the I.E. graduate program at Lehigh University.

# END

# OF

# TITLE