Is Pi Random?

And Related Matters

By Thomas R. Amoth

MTH490X, Prof. Robson

8/7/88

Introduction

Many articles have been written on the subject of normality of pi and other numbers (Wagon, 1985), (Stoneham, 1983) and seem to make the unwritten assumption that if pi is normal then it's random. But very little, if any, attention has been given to the question of whether such irrational numbers have a distribution of digits that is more uniform than would be produced by a random sequence or have some other subtle bias.

Clearly, pi and other such calculated numbers never were really random in an absolute sense since they can be calculated from well-known algorithms. Therefore a sufficiently sophisticated computer program looking for biases might detect the digit sequence as being pi and therefore be able to predict the digits which follow. This paper deals with not with such highly tailored tests but with more conventional statistical properties which would then dispel the notion pi and other numbers "look" random.

A chi-squared test on the first 2000 digits of pi yields chi-squared = 3.97 giving a probability of only 9% that a random sequence would be closer to perfectly uniform (Hald, 1958). The distribution of the first 10 million digits of pi (Wagon, 1985), yield chi-squared = 2.78 for a probability of 2.83% that a random sequence would have a smaller value. These values are biased enough toward too great a uniformity to be suspicious but not conclusive and amounted to a "teaser" that inspired this project.

One control in this experiment--a sequence of digits that acts truly random--might be found. Such a sequence could yield more insight into whether the behavior of pi is random. Another control would be to analyze rational numbers with denominators large enough so they won't repeat within the block being tested. These controls could show whether a given "apparent bias" occurs or doesn't occur in random sequences and serve as a check on statistical theory, intuition, and shortcomings of both. Other possibilities to investigate are digit sequences generated from a series of constantly changing rational numbers and the interleaving of such numbers.

This project has an interdisciplinary flavor. It involves mathematics via the formulas for pi and computer programming via implementation of the computation and analysis algorithms. It then uses statistics to analyze the results. I figured it did not involve the fourth of my favorite 4 fields--physics, but then I realized I might some day have to use statistics to decide if some data from a physics experiment was meaningful or just a "random fluctuation."

Statistical Analysis

The object is to find a more conclusive test--one showing a result having a much smaller probability of occuring by chance. One possible approach is to analyze more than one block and combine the results. Suppose 100 blocks are analyzed. Since the total number of digits in each block is fixed, there will be 100

constraints (on the 1000 counts of the number of digits 0 through 9 in each of the 100 blocks). Therefore, the chi-squared test can be handled using 900 as the number of degrees of freedom (i.e., the number of counts of digits that can be varied independently). Another simple example can be obtained directly from the above data: take the first 2000 digits away from the 10 million digit sample (presumably having a only tiny effect on chi-squared for the remaining 9.998 million digits). Then the value of chi-squared for the two samples combined would be 3.97 + 2.78 = 6.75 with 18 degrees of freedom yielding a probability of about 0.2% which strongly suggests such a bias is present.

That approach seemed promising, but the actual tests showed that pi doesn't exhibit such a bias, and the mean chi-squared over all blocks converges to 9 as expected for a random sequence (see graph, "Mean Chi-squared of pi Blocks") with the vertical axis labeled, "Chi-squared (9 deg. of freedom)").

It then seemed plausible that there might be something different about the first block for a variety of block sizes but the remaining blocks of any given block size would appear random. The graph, "Chi-squared (1st pi block)" shows that the 3.97 value at 2000 digits (Log10(Digit#) = 3.3) is not far from both local and global minimums and is therefore misleading--at least in the first 40 thousand digits.

While chi-squared stays below the nominal value of 9 until a little over 2000 digits, it gets above 16 soon after that, so it certainly doesn't stay low all the time. Random data would have a chi-squared less than 16 about 94% of the time which seems a bit less biased than the 10 million digit case and would tend to cancel out the apparent bias of the latter.

A real phenomena would have to involve some sort of a systematic bias for which an unlimited amount of evidence could be gathered merely by computing more digits, and no such bias has been found. In short, if I'd started by testing chi-squared on 4000 digits and the 10 million digit case, I never would have suspicious enough about the randomness of pi to do this project.

For comparison, I generated a number of random walks (see program description). Some of these had "peculiarities" that "looked" like biases, and certainly any process that produced only graphs like those is not random. And I would believe that any process that produced two graphs like one that "looked" biased and no others was not random.

Statistical Conclusions

The excess 0's for pi exceeds 3 standard deviations at about 256 bits. In the first 120 thousand bits, the excess 0's stay within a band of about plus and minus 1/3 of a standard deviation at the 120 thousand bit point for a while, suddenly deviate in the negative direction to about 1 standard deviation, and finally come back to the band. This behavior doesn't look random, but statisticians balk at coming to conclusions on the small (256 bit) sample size of the first aberration, and the latter anomaly is

probably not statistically significant and is difficult to analyze statistically.

The graphs for the "Excess 0's in the square roots of 2, 3, and 5 resemble the random walks I generated much better than the one for pi does. Thus while I have obtained no statistically convincing evidence, neither is the evidence convincing that pi is random. Therefore there is nagging doubt about pi's randomness and much more than 8 weeks of research would be required to resolve this question.

Pi Formulas

The old standard formula that goes back a couple centuries is 16 arctan(1/5) - 4 arctan(1/239) which can be readily derived by using the formula for the tangent of the sum of two angles. First determine that arctan(1/5) + arctan(1/5) = arctan(5/12); similarly, 4 arctan(1/5) = arctan(120/119). Use the tangent of the sum of angles formula one last time to determine that arctan(120/119) = pi/4 + arctan(1/239), using -pi/4 = -arctan(1). Solving for pi yields that old standard formula.

The April issue of Mathematics Magazine (Castellanos, 1988) gives a history of calculation of pi by describing a tremendous variety of other pi formulas ranging from very crude ones to minor improvements of the above. The better formulas in this general category converge by an essentially fixed number of digits for each term or iteration, so their execution time is essentially proportional to the square of the number of digits. I dreamed up a variation based on the fact that pi/4 is incredibly close to 22 arctan(1/28) and the remainder is the arctan of 1744507482180328366854565127/98646395734210062276153190241239 (using the rational number capabilities of LISP) or about 1/56546.78936. But even if some way could be found to compute the arctan of that 28 digit numerator over the 32 digit denominator efficiently, this formula could only yield a 2 to 1 improvement over the arctan(1/5) version.

The June issue gives the modern formulas that are related to elliptic functions and the arithmetic-geometric mean which uses square-root extraction in such a way as to double the number of digits of accuracy on each iteration. But the state-of-the art formulas (Borwein, 1988) involve the extraction of fourth-roots and are based on research by Ramanujan. That article (in the Scientific American) actually gives a complete sequence of operations on 1/4 of a page that will--without doing any of those formulas more than once--compute pi to 2 billion places if the calculations are performed with that much accuracy.

Arithmetic Algorithms

Traditionally, arithmetic operations on long numbers (other than addition and subtraction and similar simple operations) took an amount of time proportional to the product of the number of digits in the 2 operands. This relation applies to multiplication, using the conventional, simple algorithms, and related

operations have similar execution times.

In the last two decades, algorithms have been developed to to do a variety of such operations in an running time proportional to n times some power of log n which are ultimately based on applying the Fast Fourier Transform (FFT) to integer arithmetic (where n = number of digits). The standard algorithm is the Schonhage-Strassen (Aho, et.al., 1974) (hereafter abbreviated SS). The FFT can be used within a number system known in algebraic number theory as a "ring" (as well as the conventional use with complex numbers) if there are high-order roots of unity in that ring and it is possible to divide by the total number of separate data items being transformed (which is usually made to be a power of 2). Splitting the large integer up into a number of blocks and doing arithmetic modulo an odd number (to permit the above division) makes it possible to use the FFT technique.

I have slowly begun to realize the Schonhage-Strassen algorithm has some additional efficiencies built-in that are not at all obvious on a first (admittedly difficult) reading. The modulus is chosen to be 1 greater than a power of 2 and the roots of unity are themselves powers of 2. On a binary computer, multiplication by a power of 2 can be done by shifting and is considerably faster than ordinary multiplication--and this difference is far greater for a multiple precision number as for the blocks in SS. SS seemed to be using larger blocks than would be needed to meet these conditions. At first I thought this was to make that power of 2 divisor be equal to unity, but that would require the number of blocks to be at least as large as the number of bits per block which seems to be backwards from the way SS is designed. Nevertheless, there are enough subtle design points that I would give careful thought to changing the algorithm.

Such efficient algorithms are known for calculating algebraic functions (such as finding the roots of a polynomial), converting between different bases (as between binary and decimal), and elliptic functions (complete elliptic integrals)--which led to an efficient pi formula. According to Borwein (1988), even this formula for pi has been improved recently. However, Brent (1976) gives an algorithm for calculating trig, exponential and inverses of these functions in log n times the time to do a multiplication. Such an algorithm would make it possible to calculate e efficiently.

I haven't had time to implement any of the above during the 8 week course, due in part to complexity and also because Schonhage-Strassen itself used a simpler algorithm for efficient multiplication whose execution time is the log base 2 of 3 (about 1.6) power of the number of digits. As of this writing, I just got reciprocal and square root routines essentially working whose efficiency depends only on the speed of the multiplication routine they call. I hope to be able to use this with the 1.6th-power multiplication algorithm written by Russell Ruby and calculate square roots to hundreds of thousands of digits in reasonable time. I also hope to use the Kolmogorov-Smirnoff statistical test as suggested by Prof. Ed Waymire in the OSU Math Department on several irrational numbers and may try to implement an efficient

binary to decimal conversion routine. But that is the most that can be expected during the 8 week course.

I have determined how to calculate the gcd (greatest common divisor) efficiently. Presumably other functions such as Bessel functions can be calculated efficiently since all simpler functions can as shown by the article by Brent, although it is probably very difficult to discover how to do any particular function. Apparently, numerical analysis problems such as numerical integration or differential equation simulation could not be done efficiently because great accuracy would require evaluating the function a number of times that is roughly proportional to (or a polynomial of) the number of digits. But this observation is just a special case of the view that no conventional algorithm can have "log n times a multiplication" efficiency. So the question of whether an efficient algorithm exists for doing any numerical operation is equivalent to the question of existence of an unconventional algorithm that converges quadratically or at a similar rate.

Program Description

I wrote the program to do the calculations in c (which, for the uninitiated, is most simply described as a competitor of Pascal). While LISP has considerable capabilities to do multiprecision arithmetic, it is apparently not possible to improve those algorithms, and I felt I didn't want to be constrained by the operating environment and any other limitations it imposed--particularly in regard to efficiency. Whereas c is apparently the most efficient of the common high-level languages, and would permit easy interfacing to assembly routines later, if desired since it operates on data in a matter that is close to the way assembly language programs do but retains good machine-independence. For example, c can step through arrays using pointers (machine addresses--like assembly language programs would) rather than indexing as a Pascal program would (unless the Pascal programmer "cheats"). The main incompatibility between different systems I'm aware of is the number of bits in type "int" (integer).

My c program produced output data files in Ascii (character/text) format that were essentially just tables of numbers. I chose to make the file be in character rather than binary format based on the recommendation of one consultant in the CS lab who said there were sometimes incompatibility problems with transferring binary files. An ascii file also has the advantage of being human readable and immediately compatible with a far greater variety of programs.

In particular, I downloaded these text files to an IBM-PC then took them home to my IBM compatible and "imported" them to the spreadsheet "Quattro" which is essentially a compatible competitor of "Lotus 1-2-3" (Trademarks of Borland Int'l and Lotus Development Corp., respectively). In the case of the data giving the number of binary 0's in pi, this file was just two columns: bit number and number of 0's up to that point.

Once the data was in my spreadsheet, I entered formulas to do arithmetic on those two columns of numbers to obtain the number of excess 0's. Spreadsheets make this easy because you just have to enter one formula and then make lots of copies of it. I then entered other formulas representing the points corresponding to +1 and -1 standard deviation at each bit number (row of spreadsheet). These two formulas together became the parabolas facing sideways which you see on the graphs giving the "number of excess zeros" in pi.

Then I told Quattro to graph the results. There were 5 columns: bit # (which became the X-axis), number of zeros (not used directly for graphing) and the three columns of results calculated by Quattro which became the three lines printed on the graph. I then produced other graphs by creating versions of the spreadsheet with this data by entering other formulas to get, for example, the "log(bit #)" versus "excess 0's (in standard deviations)" plots. Thus, the c program did the heavy computational work, and Quattro was able to do arithmetic operations on that data to produce a variety of graphs.

For comparison, I generated "random walks" (which is what the excess 0's of pi should be if it is truly random) entirely by using Quattro. Since I wanted as many bits as practical, I used Quattro's random-number generator 4 times in each line and therefore only plotted every fourth step in the random walk. Each random number was compared with .5 and the results of these 4 comparisons were added together and 2 subtracted from the sum. This formula gives a random number for each line of the spreadsheet between -2 and 2 that looks like 4 random steps of -1/2 or +1/2 with 50% probability for each step. Each formula except the first was added to the one above to produce a cumulative effect and therefore a random walk with every fourth step displayed on the graph.

The random-number generator appears to be an "unpredictable" one in the sense that there is no way to recreate a set of random numbers it generated; the usual method for creating such numbers is to look at the system (time-of-day) clock or other "unpredictable" event and do some complicated transformation on it. Since I didn't save a copy of the disk file for each random walk, there is no way I can print additional copies of those random walks with the computer.

Other Areas to Investigate

Relationships between statistical and mathematical properties might be made by classifying all real numbers according to the way they are "defined," (i.e., how are the digits "chosen") and whether they behave periodically, chaotically, randomly, or some combination thereof. The existence of (partial?) correlations between these two ways of classify real numbers could prove interesting.

Works Cited

Aho, Hopcroft, Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley 1974. 270-276 (Schonhage-Strassen integer-multiplication algorithm).

Bhattacharya, Rabi N., Edward C. Waymire. Stochastic Processes With Applications. 1987 (unpublished at this time).

Brent, Riachard P. "Fast Multiple-Precision Evaluation of Elementary Functions." Journal of the Association for Computing Machinery 23 (April 1976) 242-251. (Describes algorithms of order log n times the time of a multiplication, and gives references for Euler's constant and the Gamma function.)

Borwein, Jonathan M. and Borwein, Peter B. "Ramanujan and Pi." Scientific American 258 (Feb. 1988) 112-117.

Feller, William. An Introduction to Probability Theory and Its Applications. John Wiley & Sons 1968.

Hald, A. Statistical Tables and Formulas. New York: John Wiley, 1958.

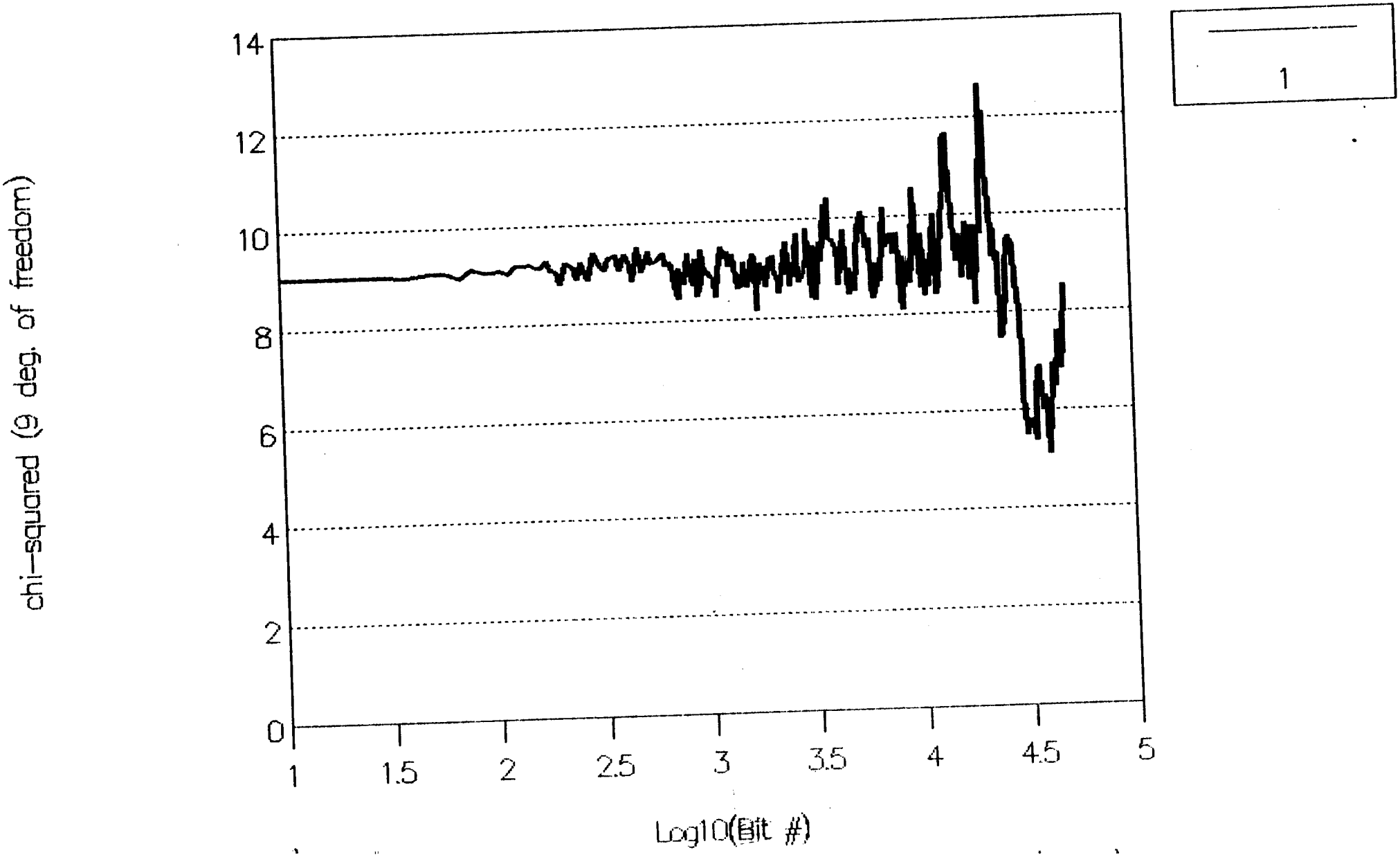
Knuth, Donald E. The Art of Computer Programming. Addison-Wesley 1981. 92-109 (spectral test), 290-297 (Fourier transform algorithms).

Miller, L.H. "Table of percentage points of Kolmogorov statistics." J. Amer. Statist> Asoc. 51 (1956), 111-121.

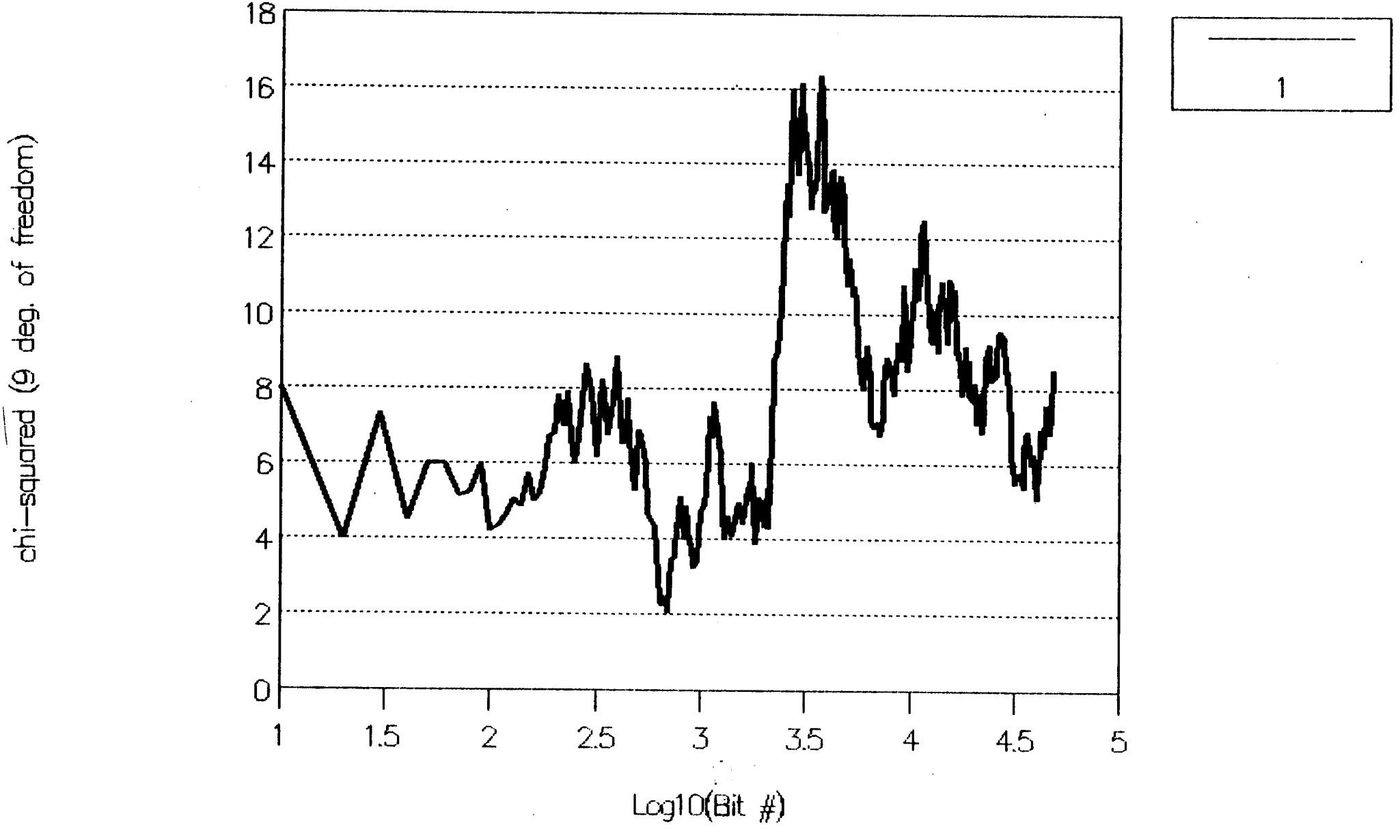
Stoneham, R. G. "On a sequence of (j,epsilon)-normal approximations to pi/4 and the Brouwer conjecture." Acta Arithmetica 42 (1983) 265-279.

Wagon, Stan. "Is pi Normal?" The Mathematical Intelligencer 7 (1985, No. 3) p65-67 (gives number of each digit in first 10 million digits of pi).

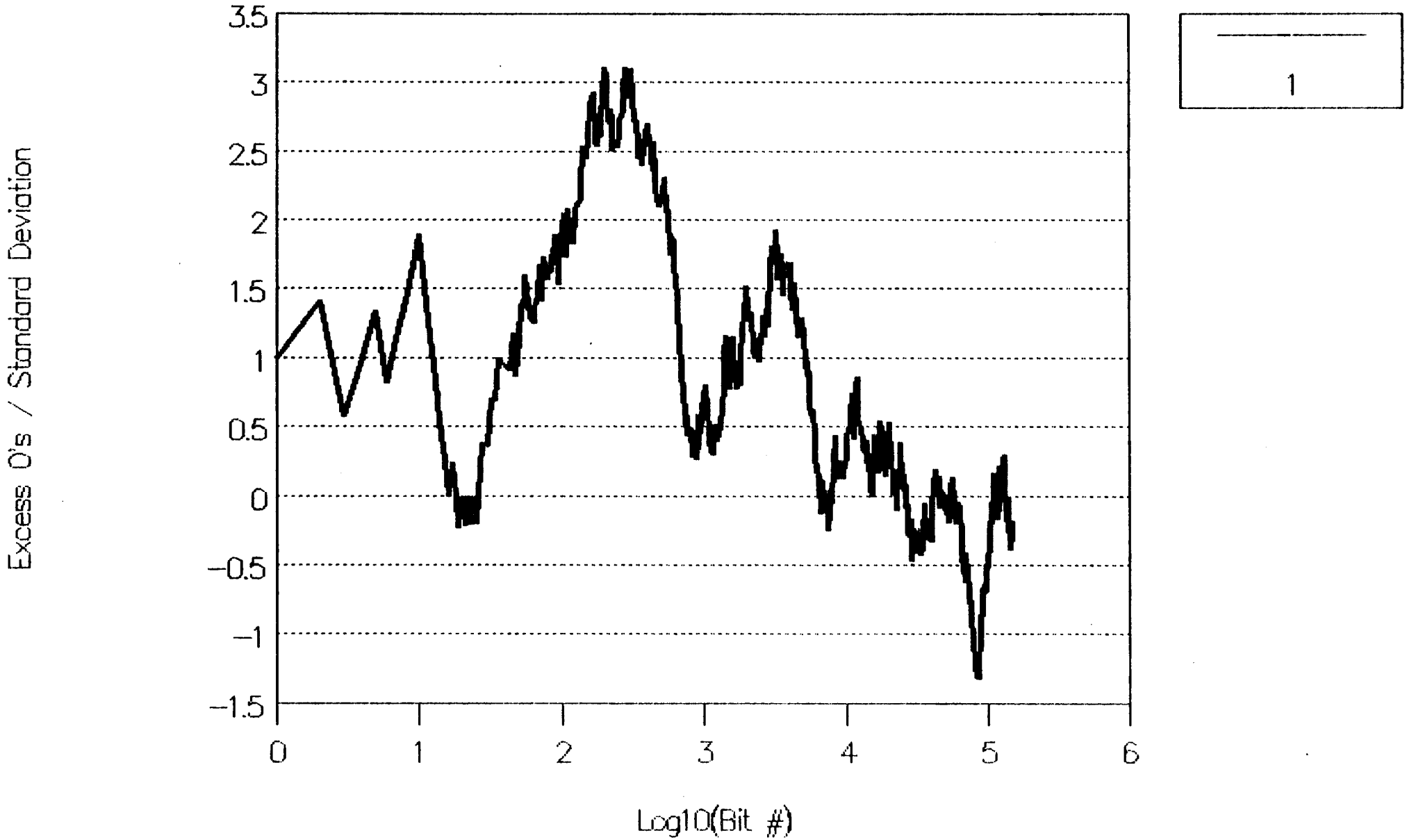
Mean Chi-squared of all blocks



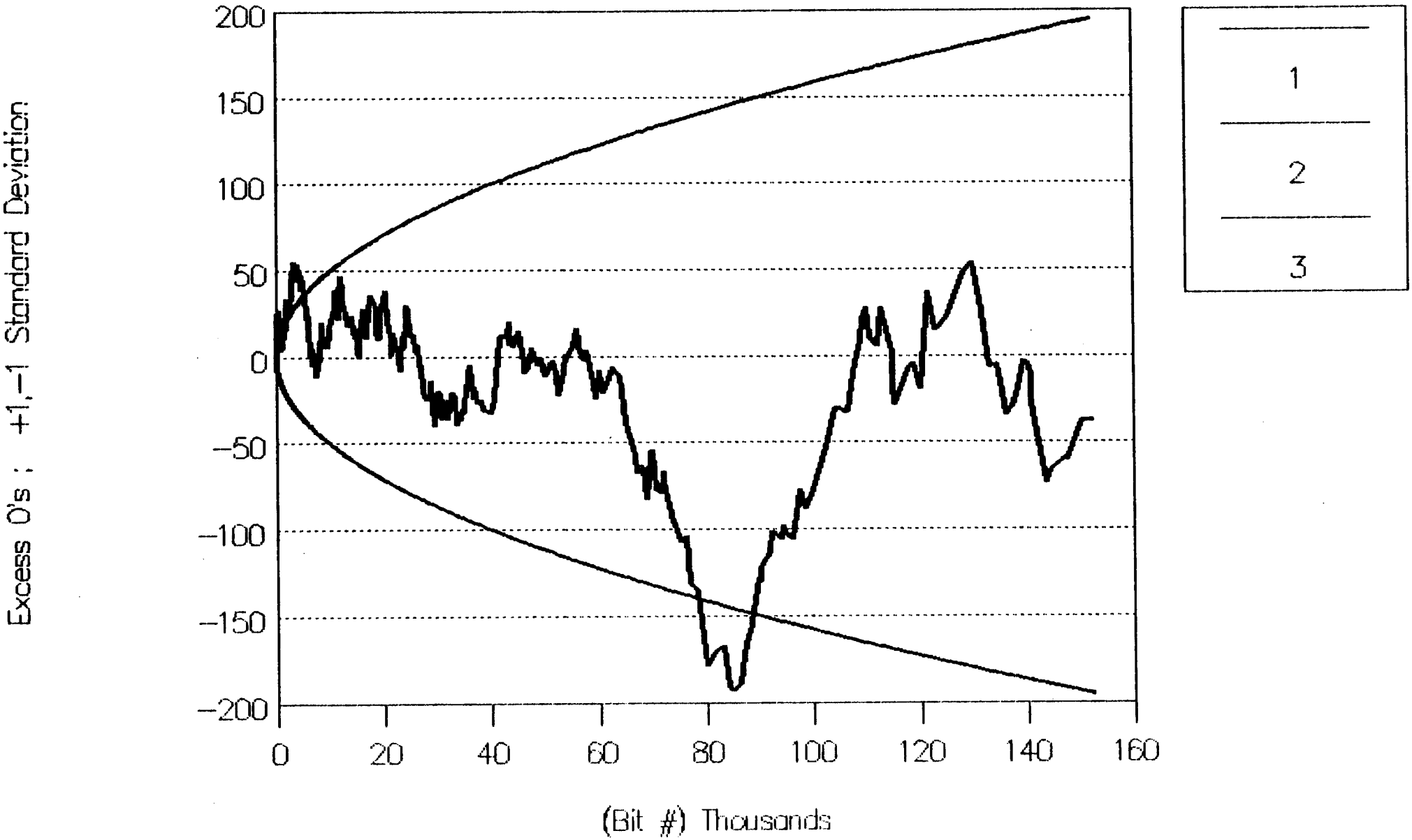
Chi-squared (1st pi block)



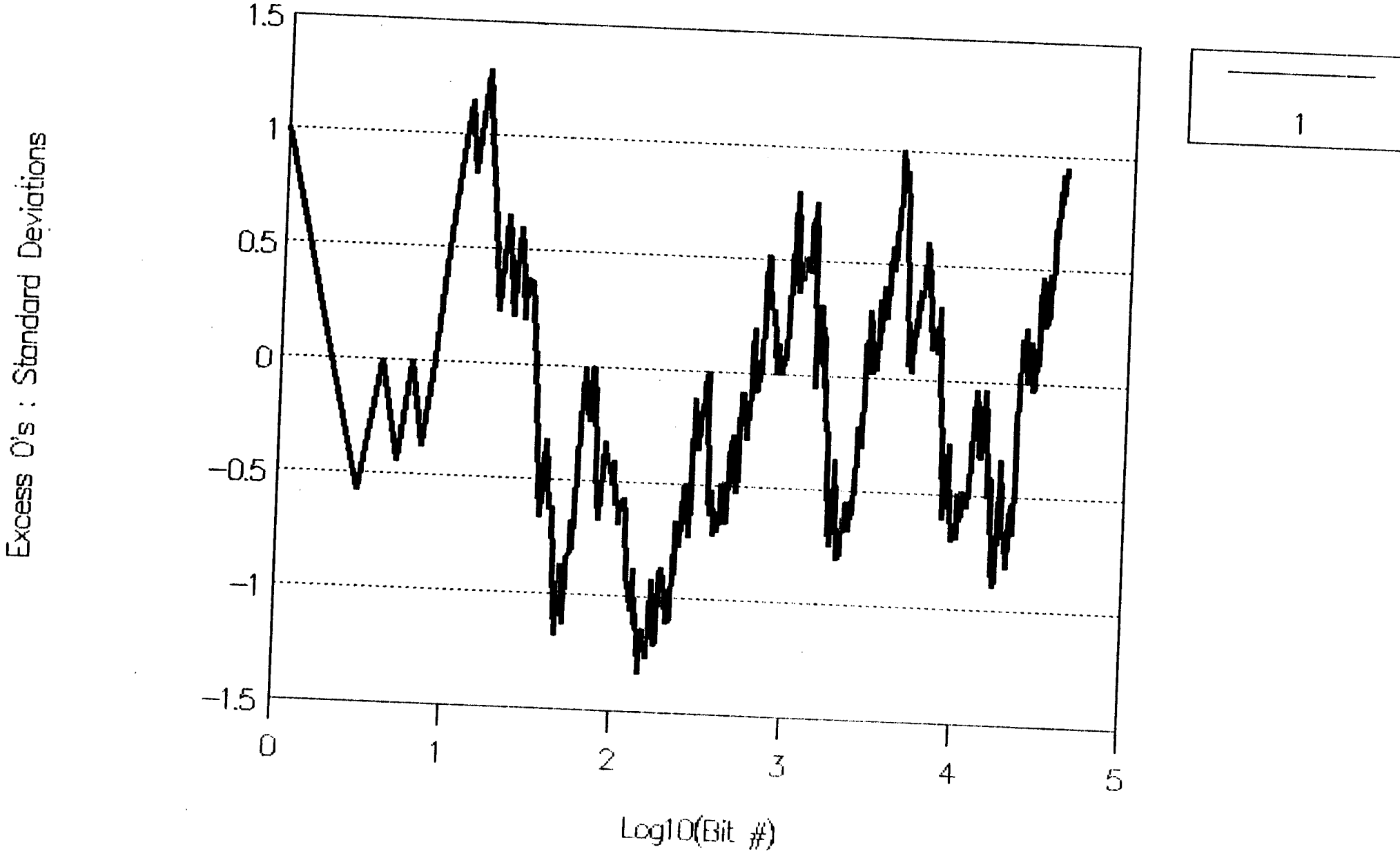
Excess 0's in Binary pi



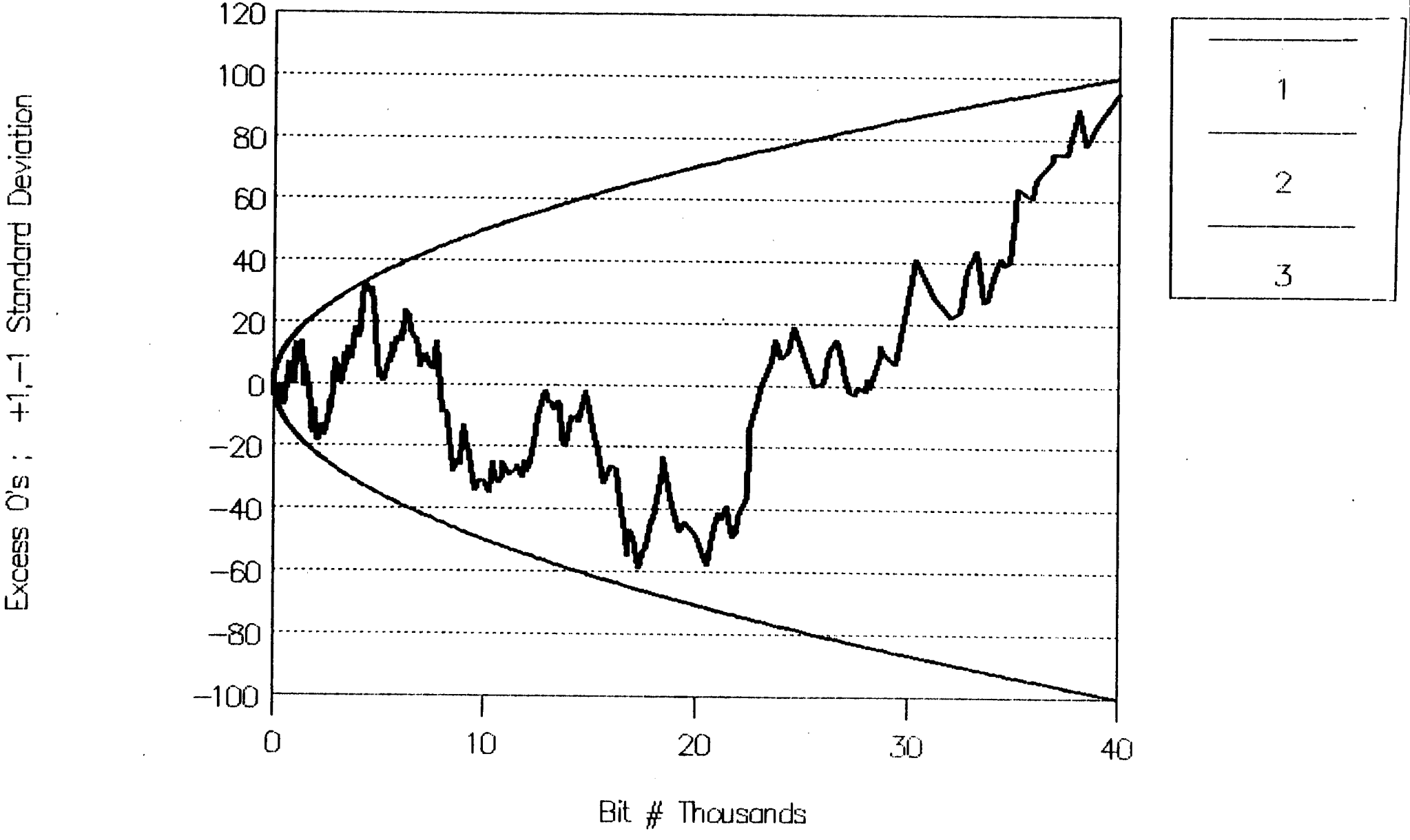
Excess 0's in Binary pi



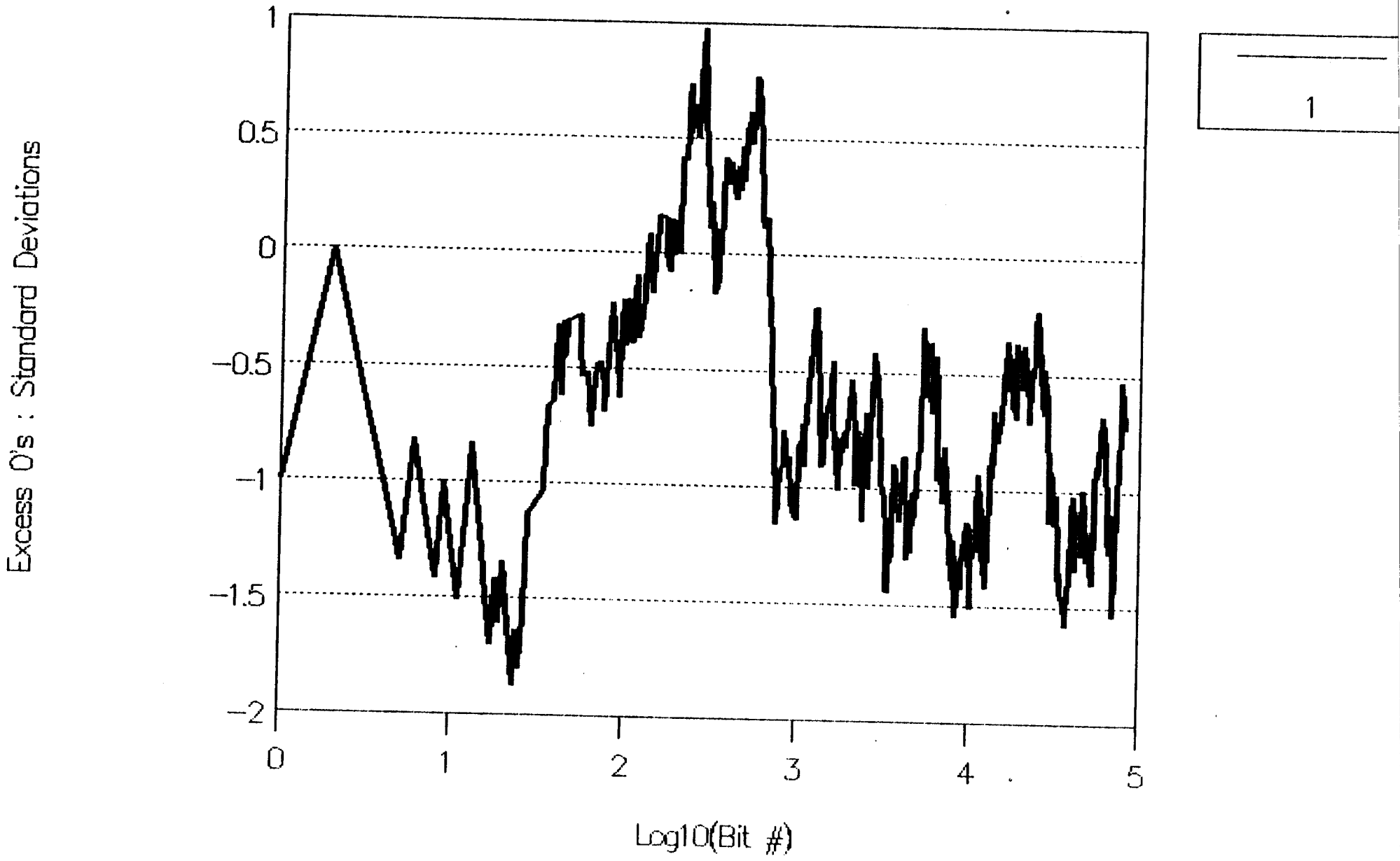
Excess 0's in Sqrt(2)



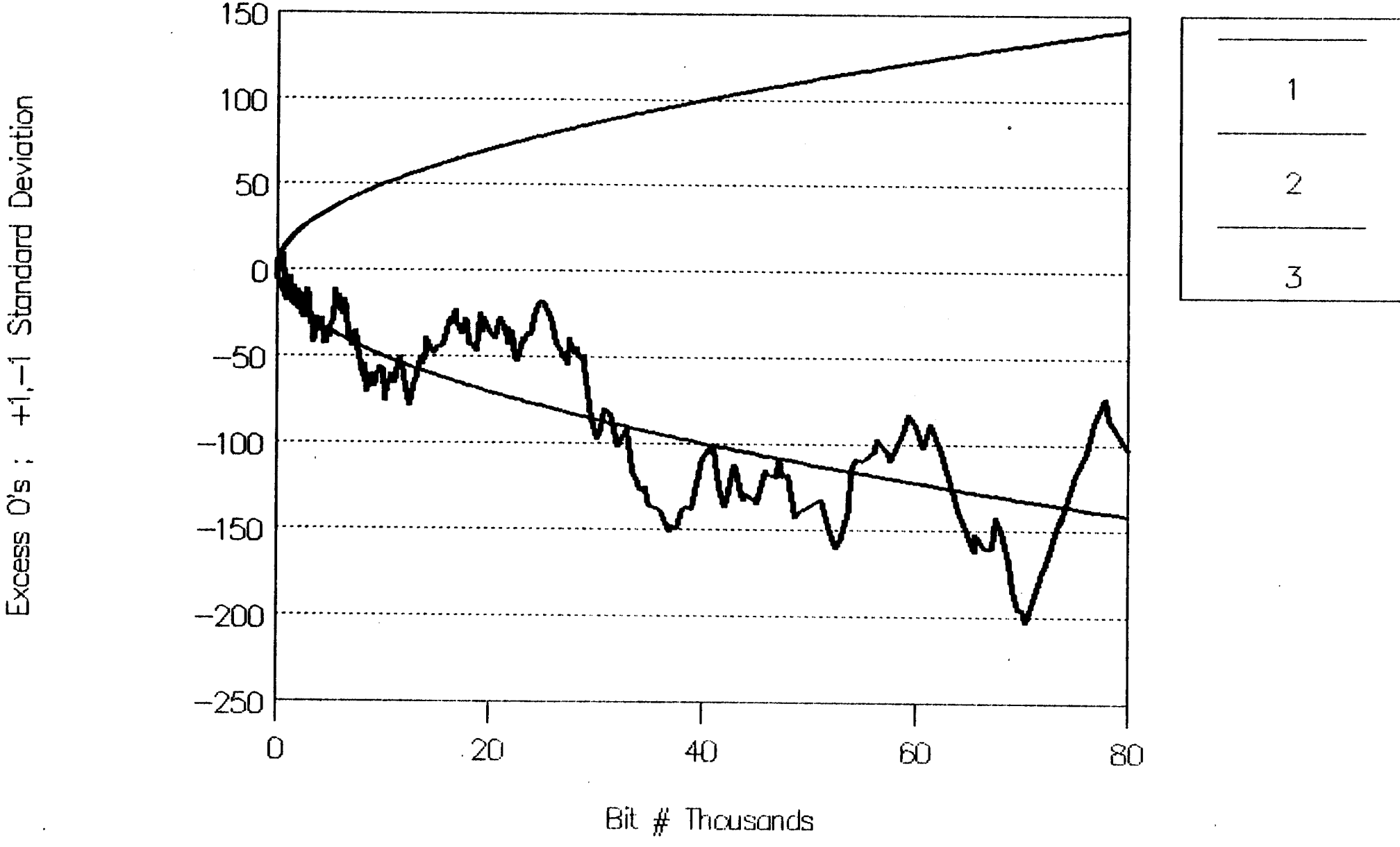
Excess 0's in Sqrt(2)



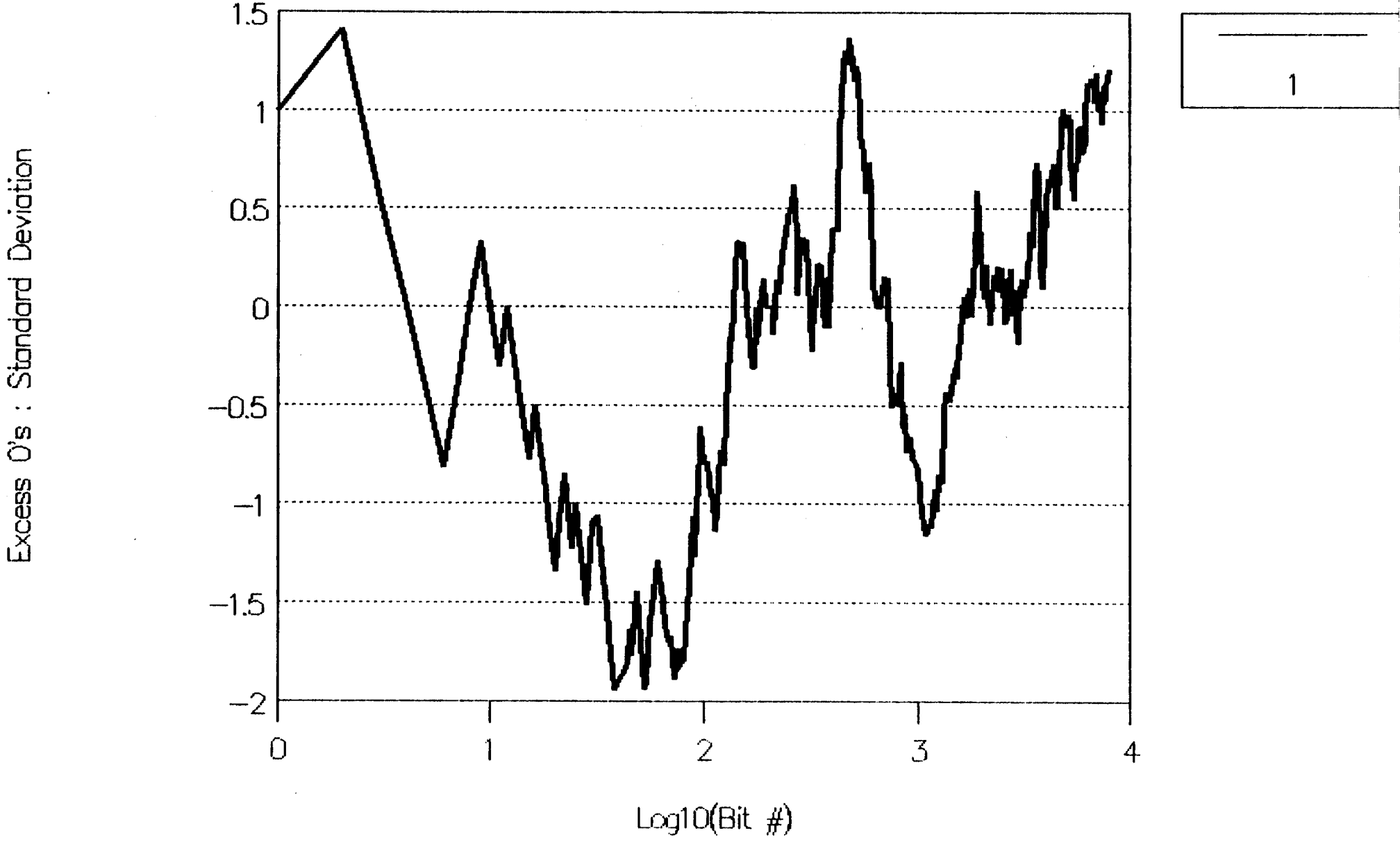
Excess 0's in Binary Sqrt(3)



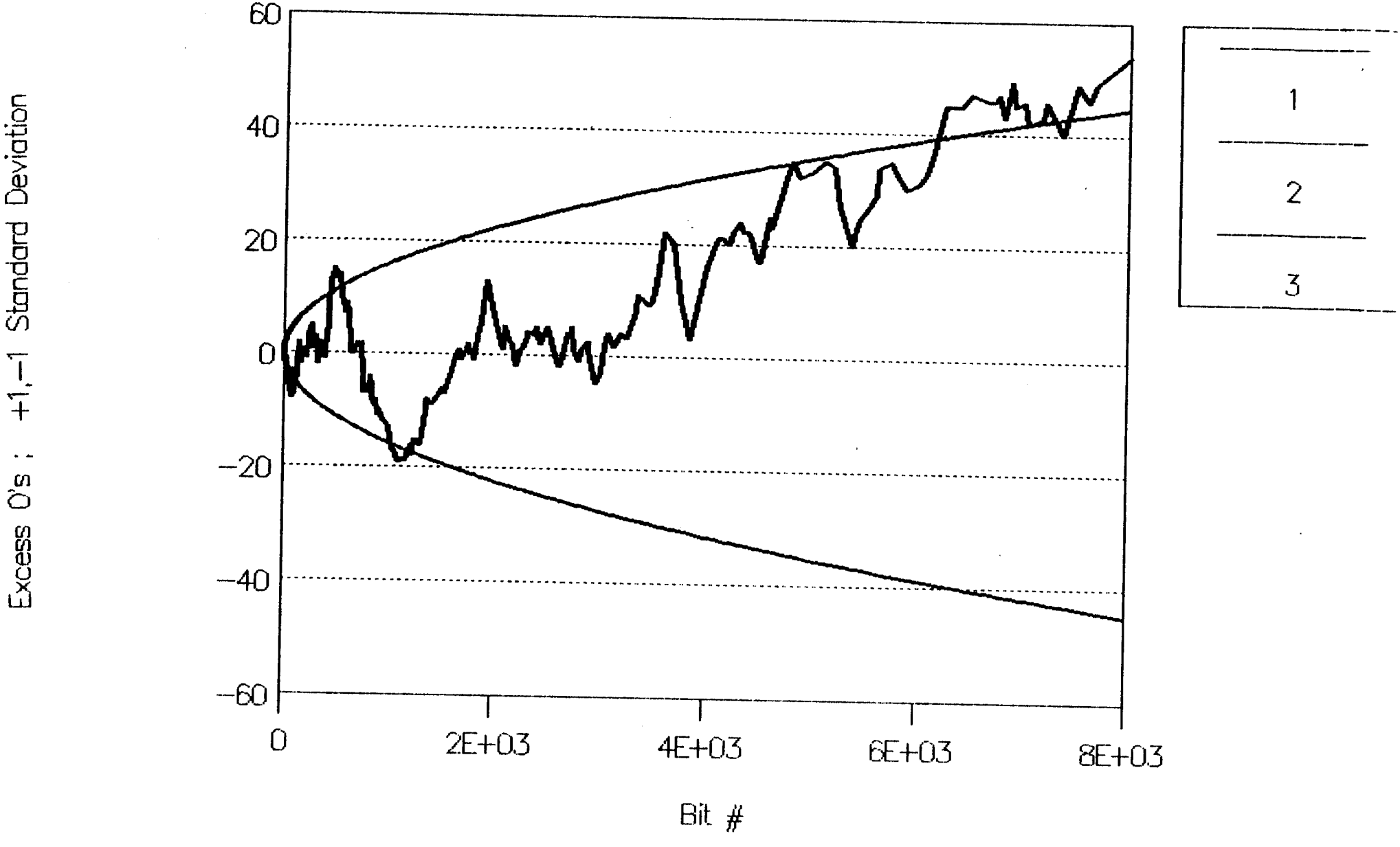
Excess 0's in Binary Sqrt(3)



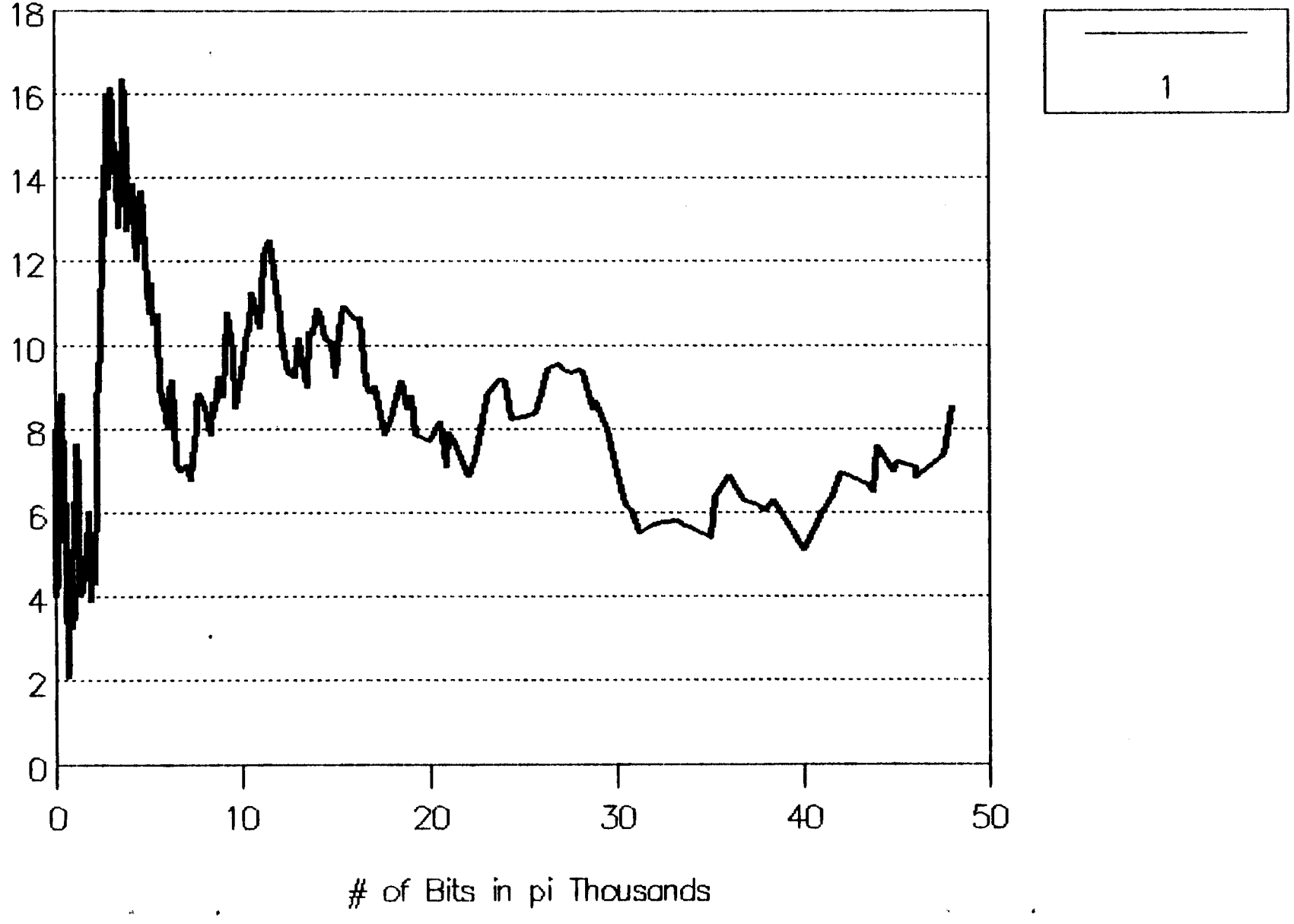
Excess 0's in Binary Sqrt(5)



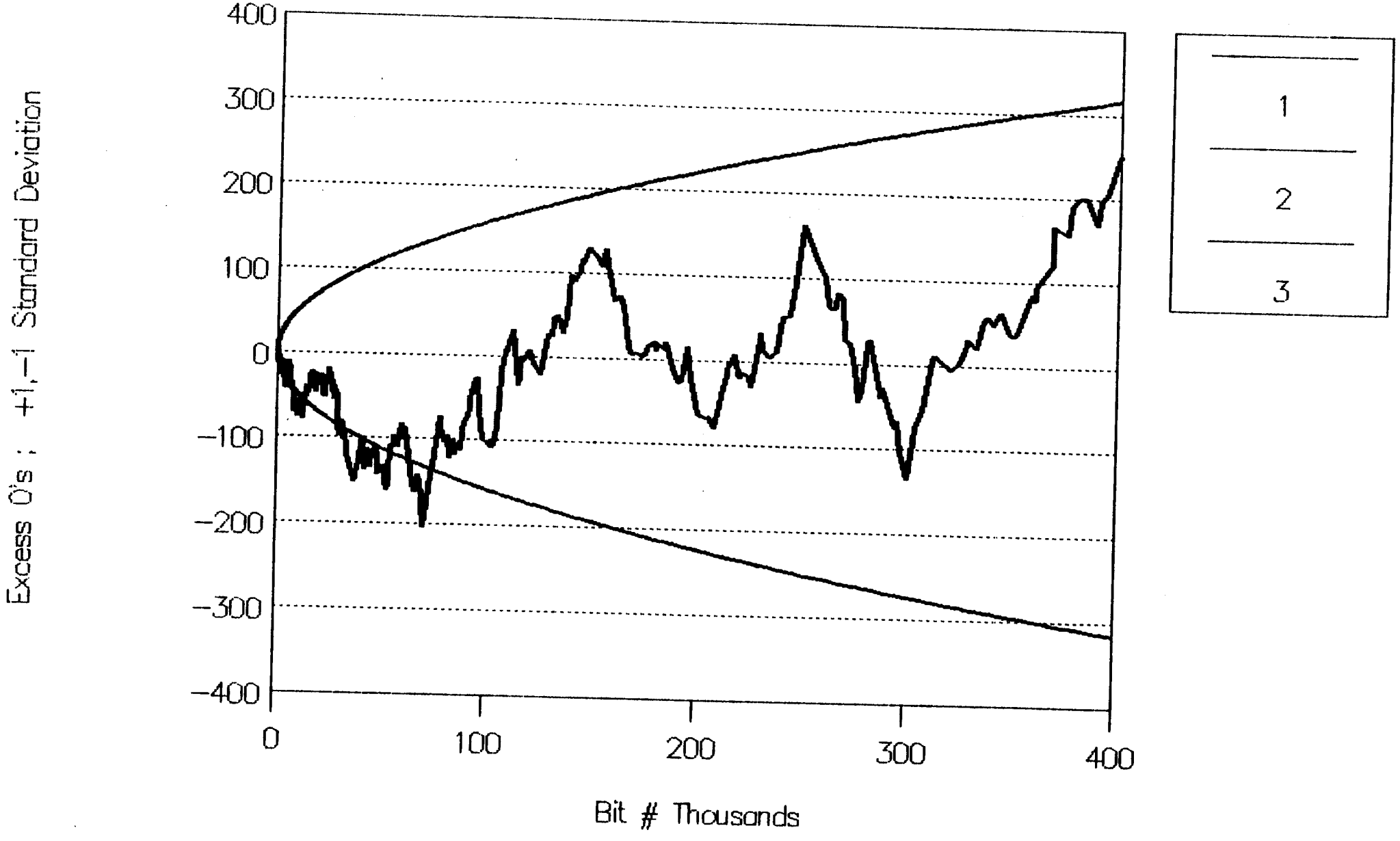
Excess 0's in Binary Sqrt(5)



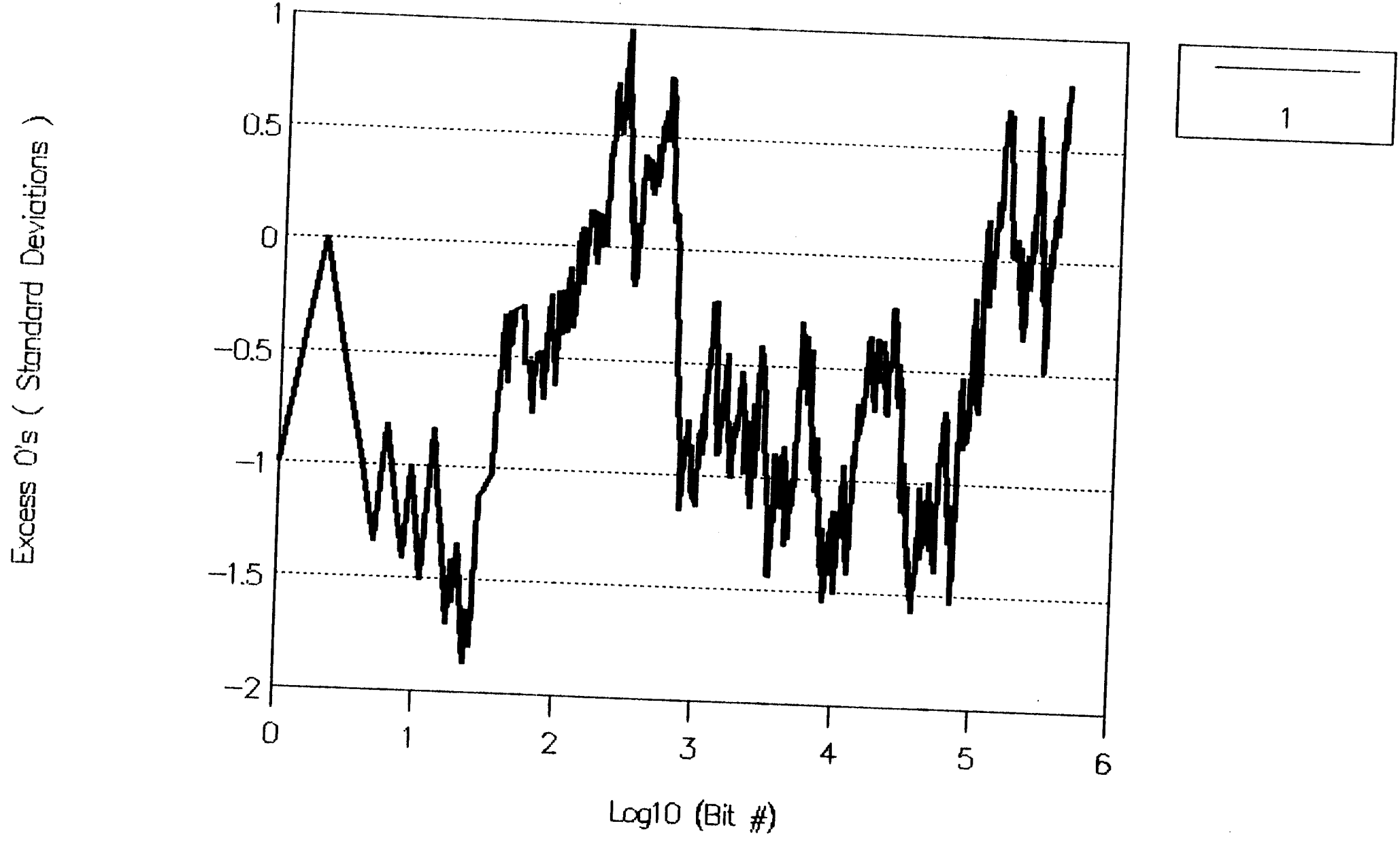
Chi-squared (1st pi block)



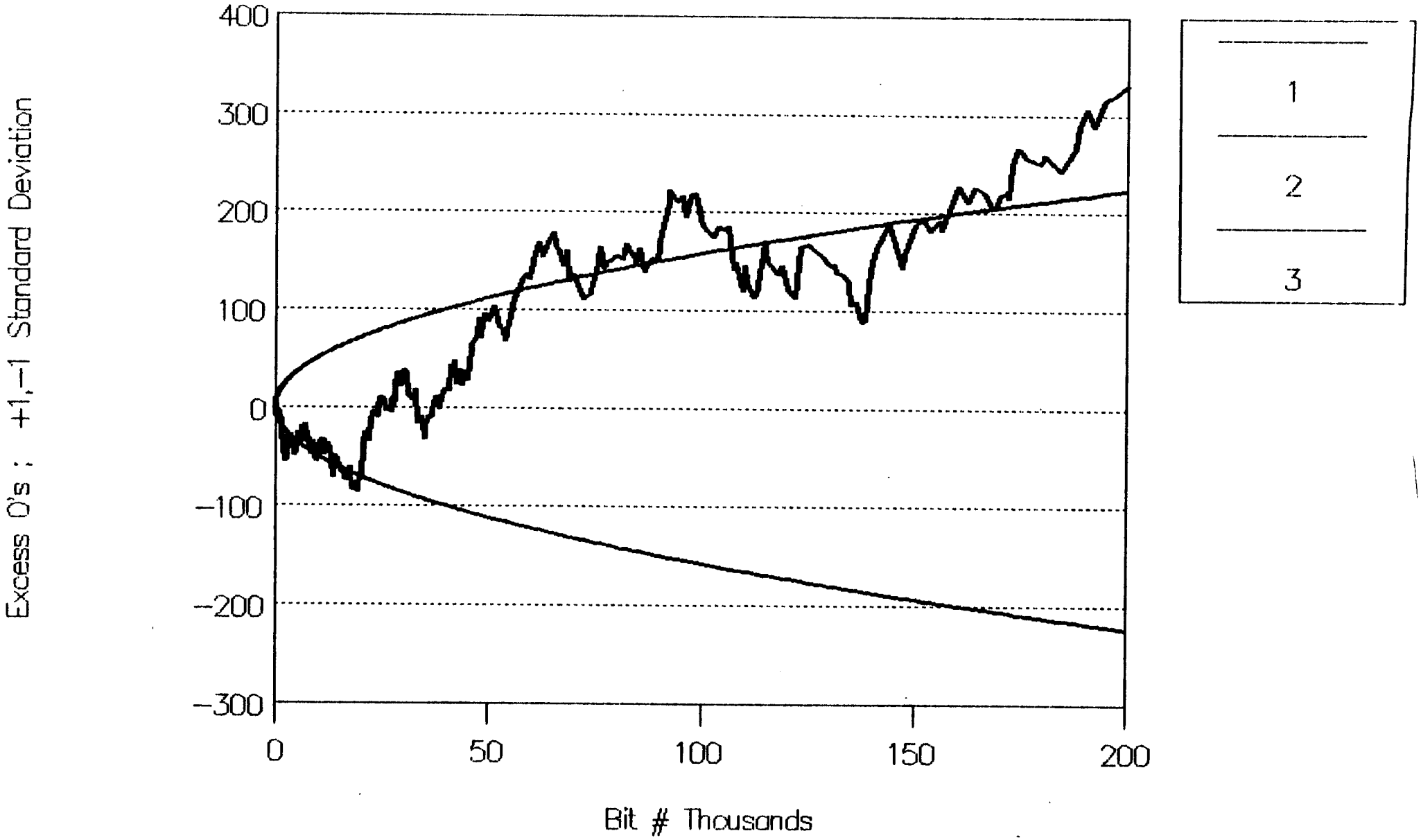
Excess 0's in Sqrt(3)



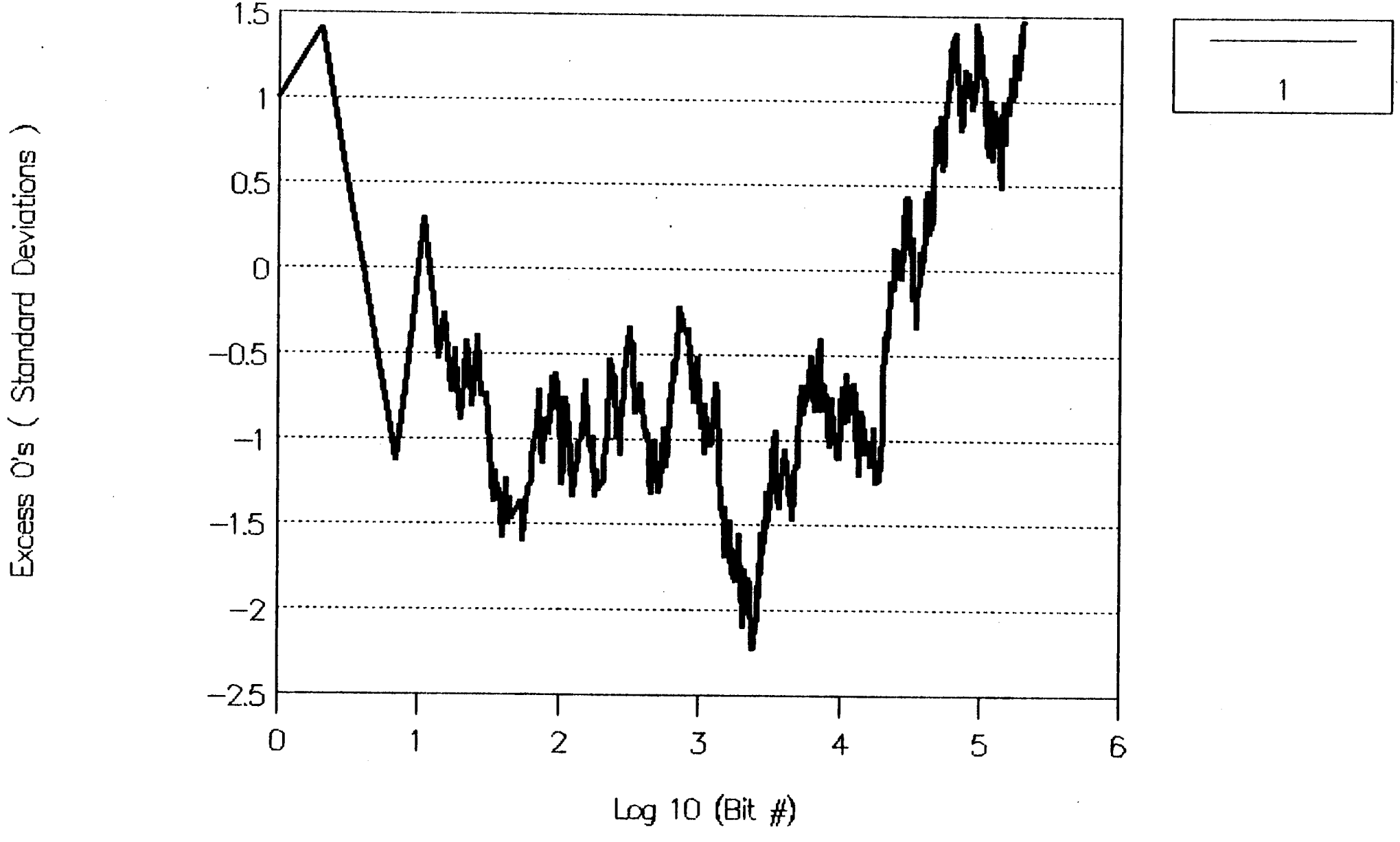
Excess 0's in Sqrt(3)



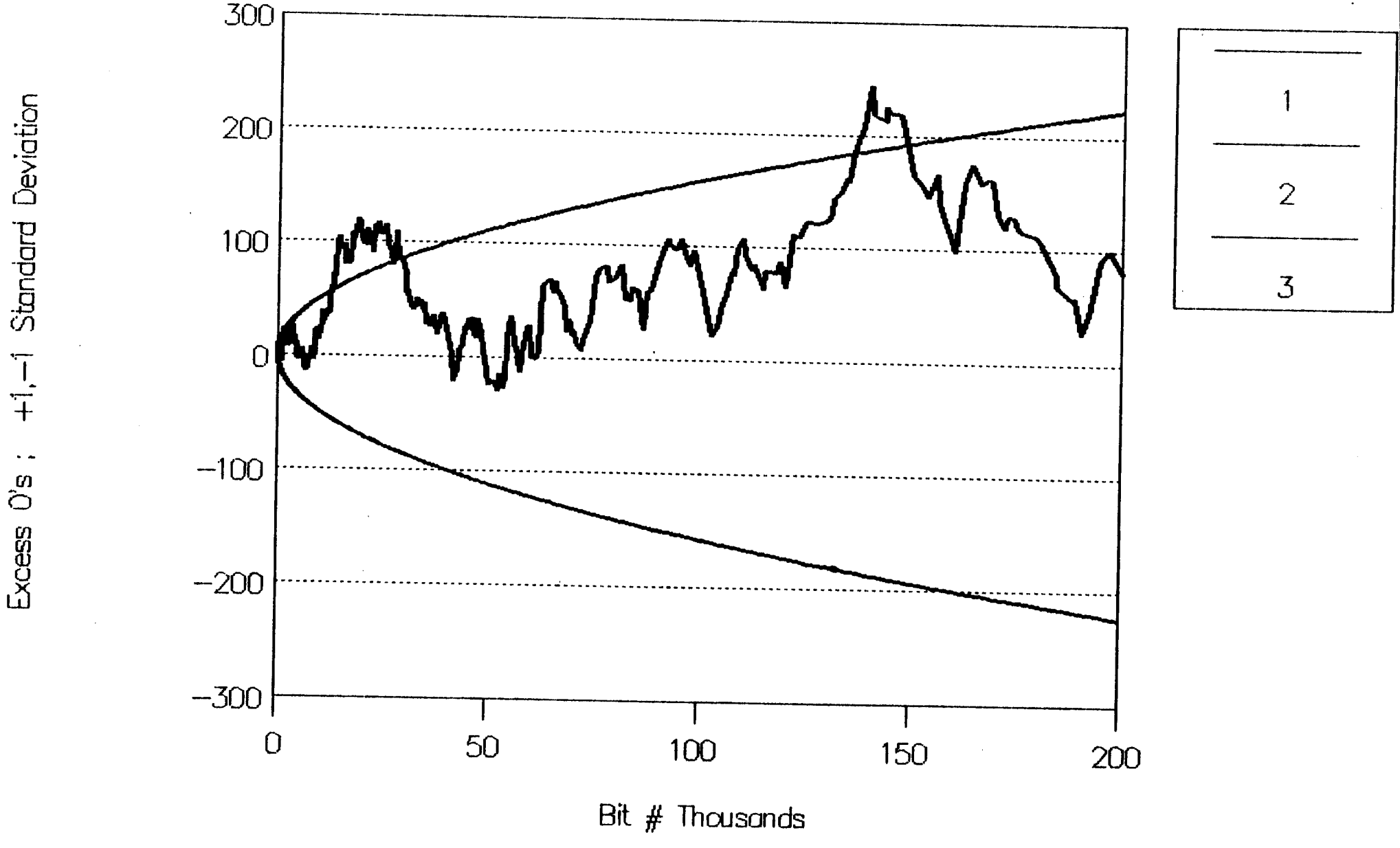
Excess 0's in Binary Sqrt(18)



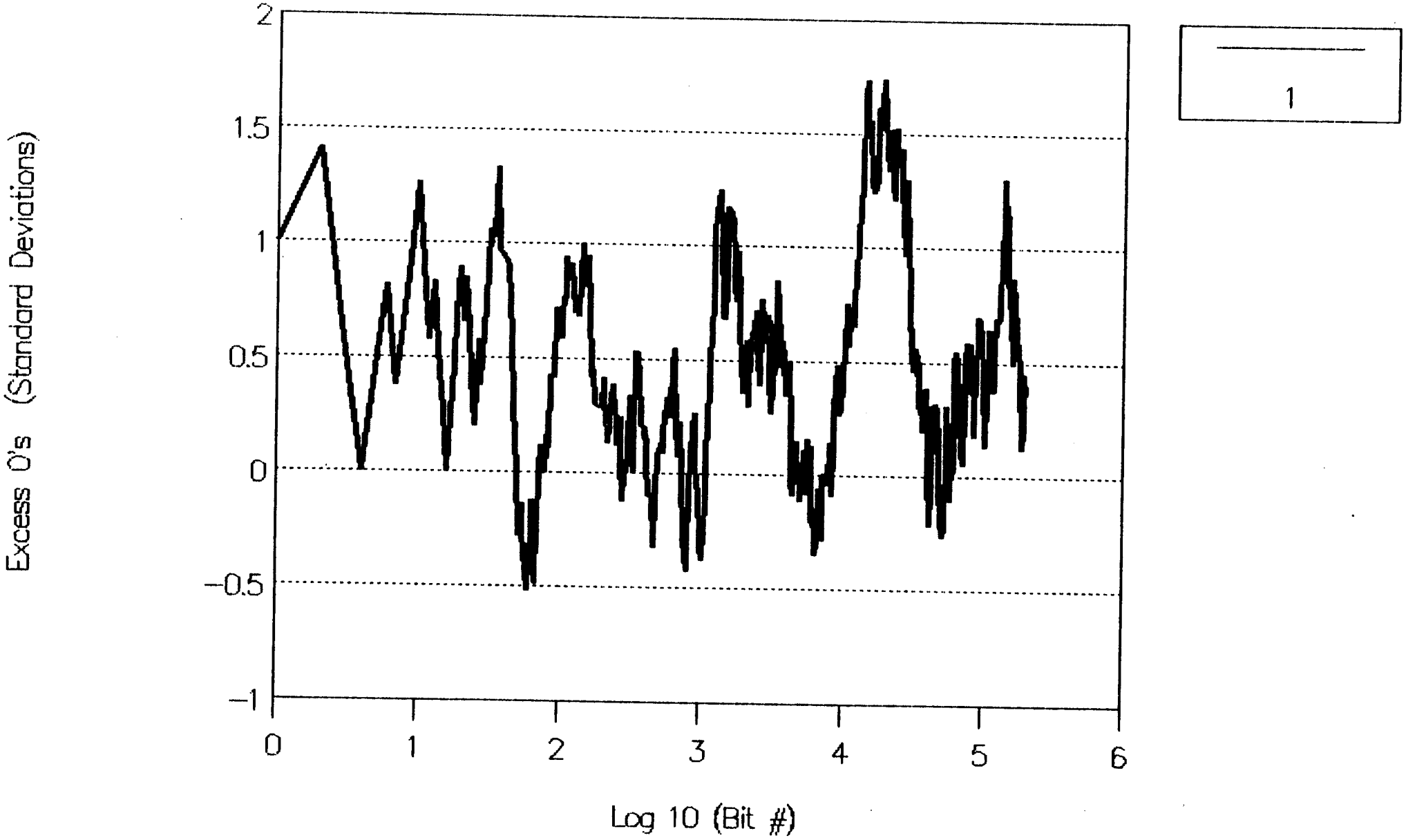
Excess 0's in Binary Sqrt(18)



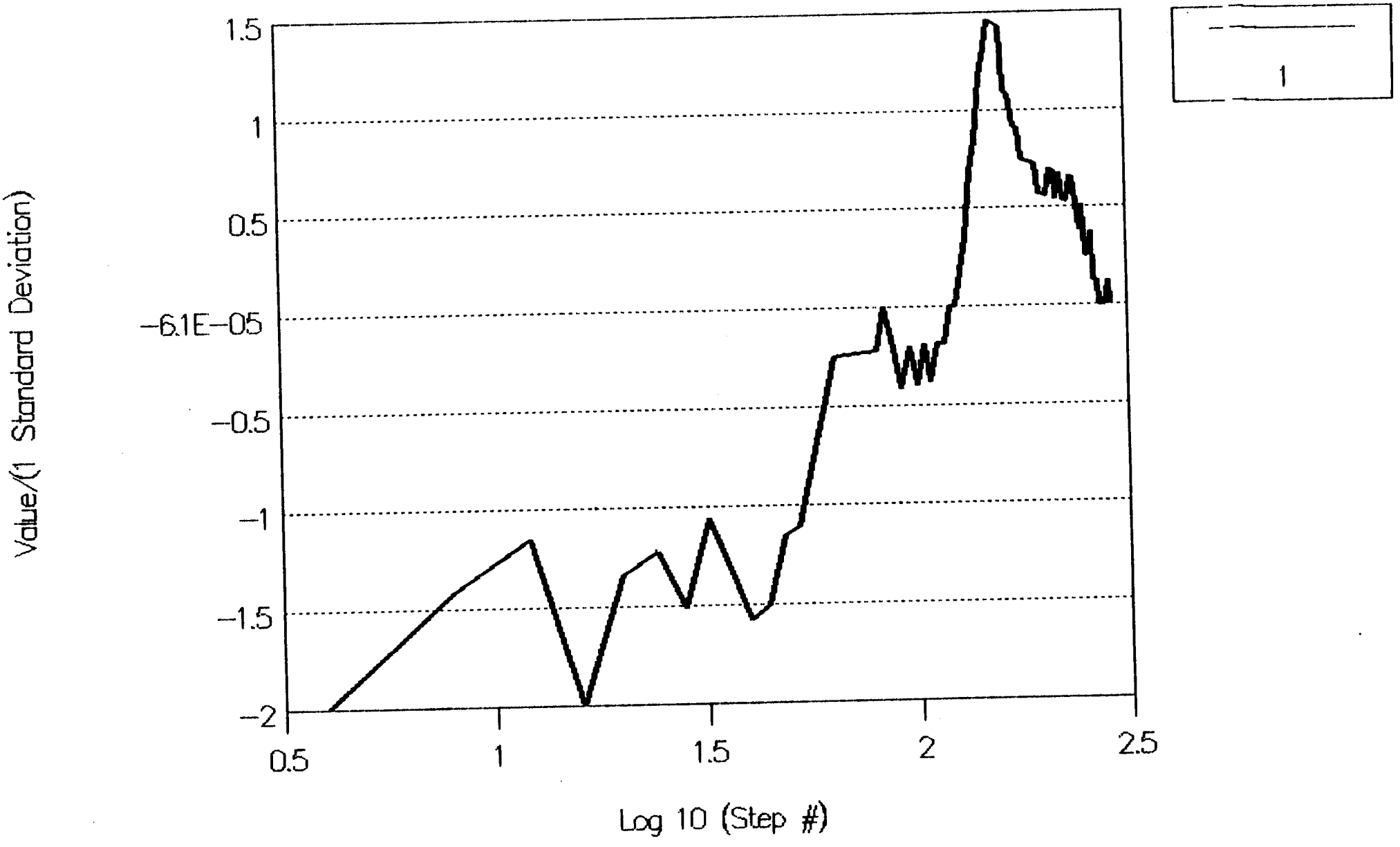
Excess 0's in Binary Sqrt(27)



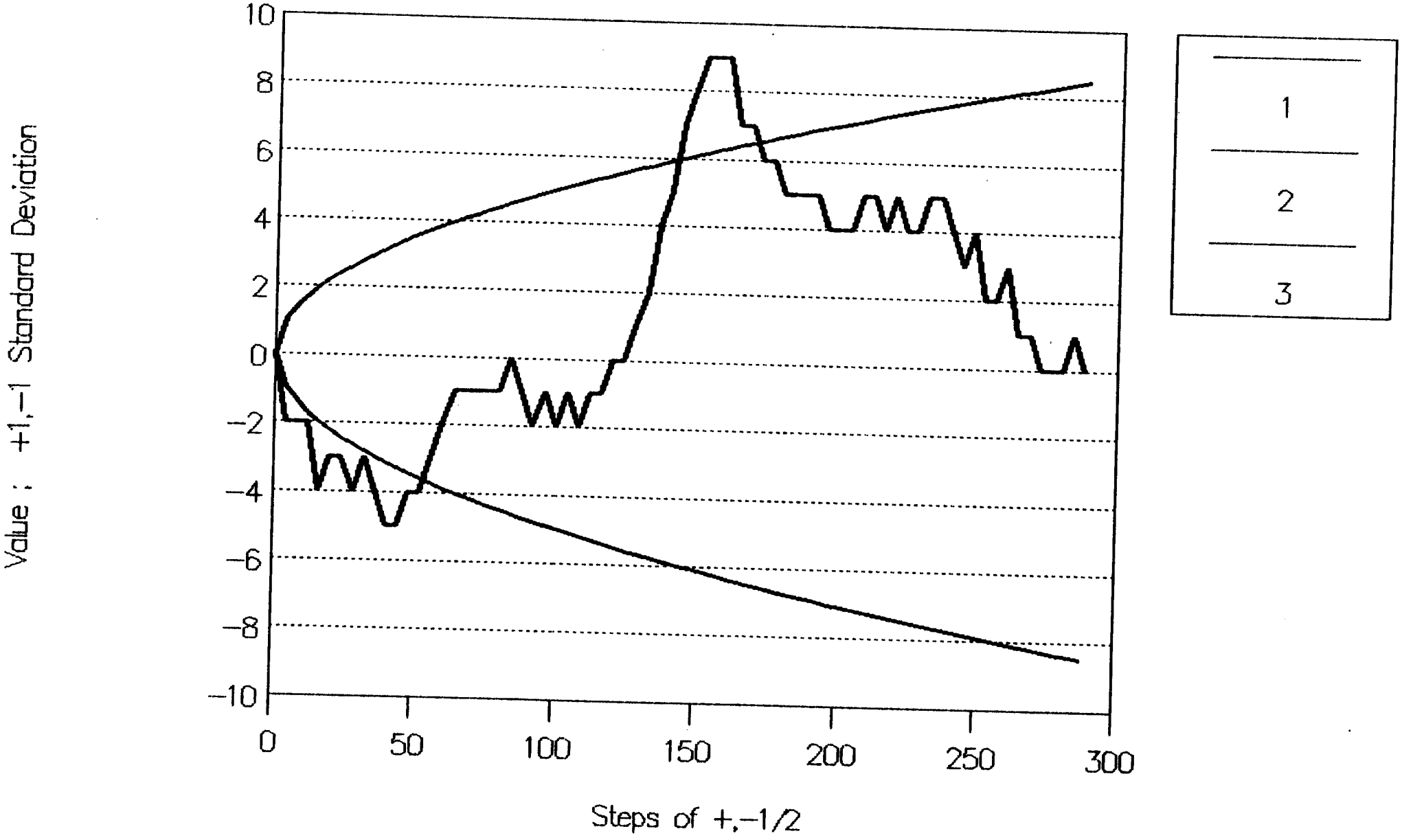
Excess 0's in Binary Sqrt(27)



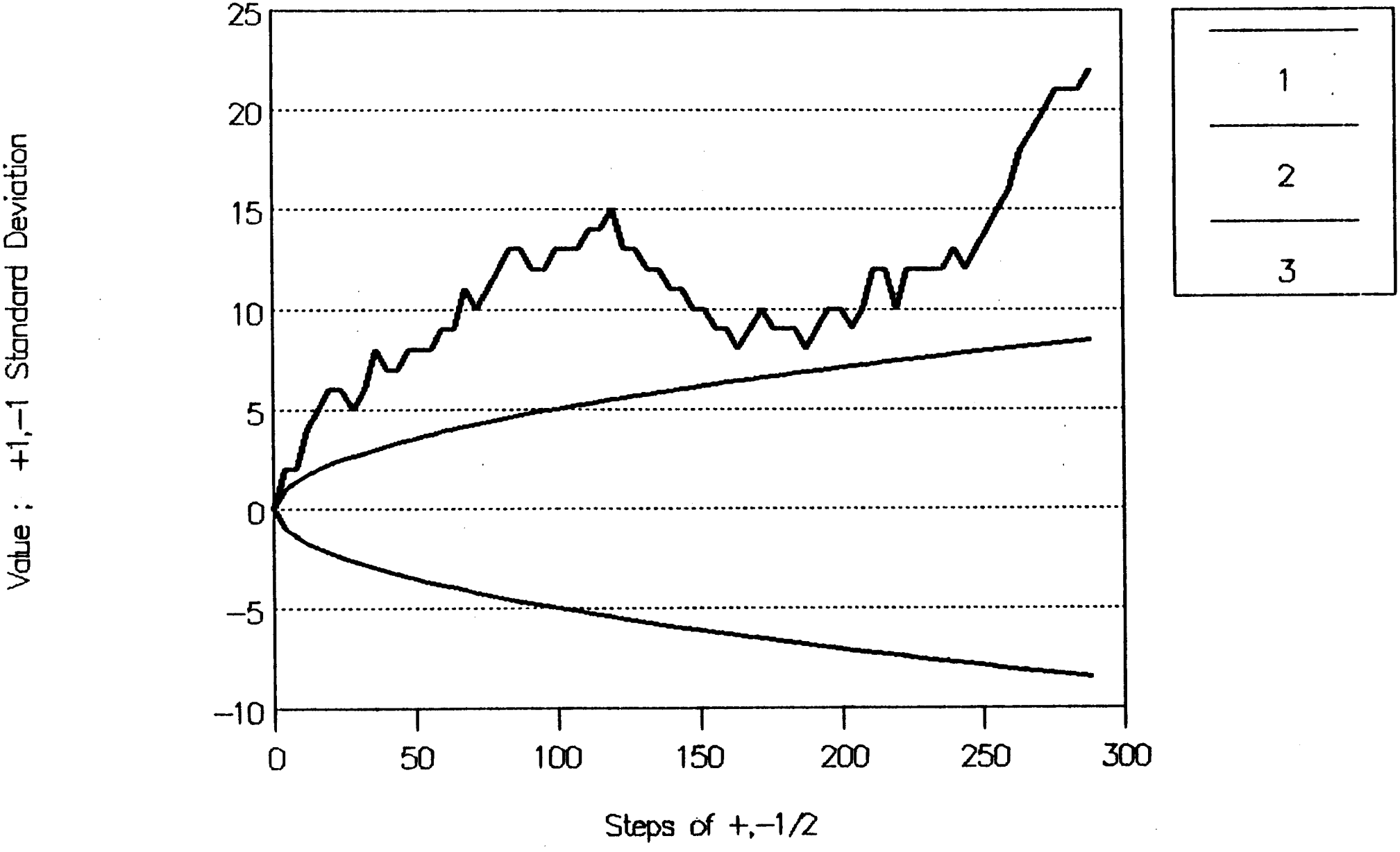
Random walk


1.5
1
0.5
-6.1E-05
-0.5
-1
-1.5
-2
0.5
1
1.5
2
2.5
Value/(1 Standard Deviation)
Log 10 (Step #)
1

Random walk

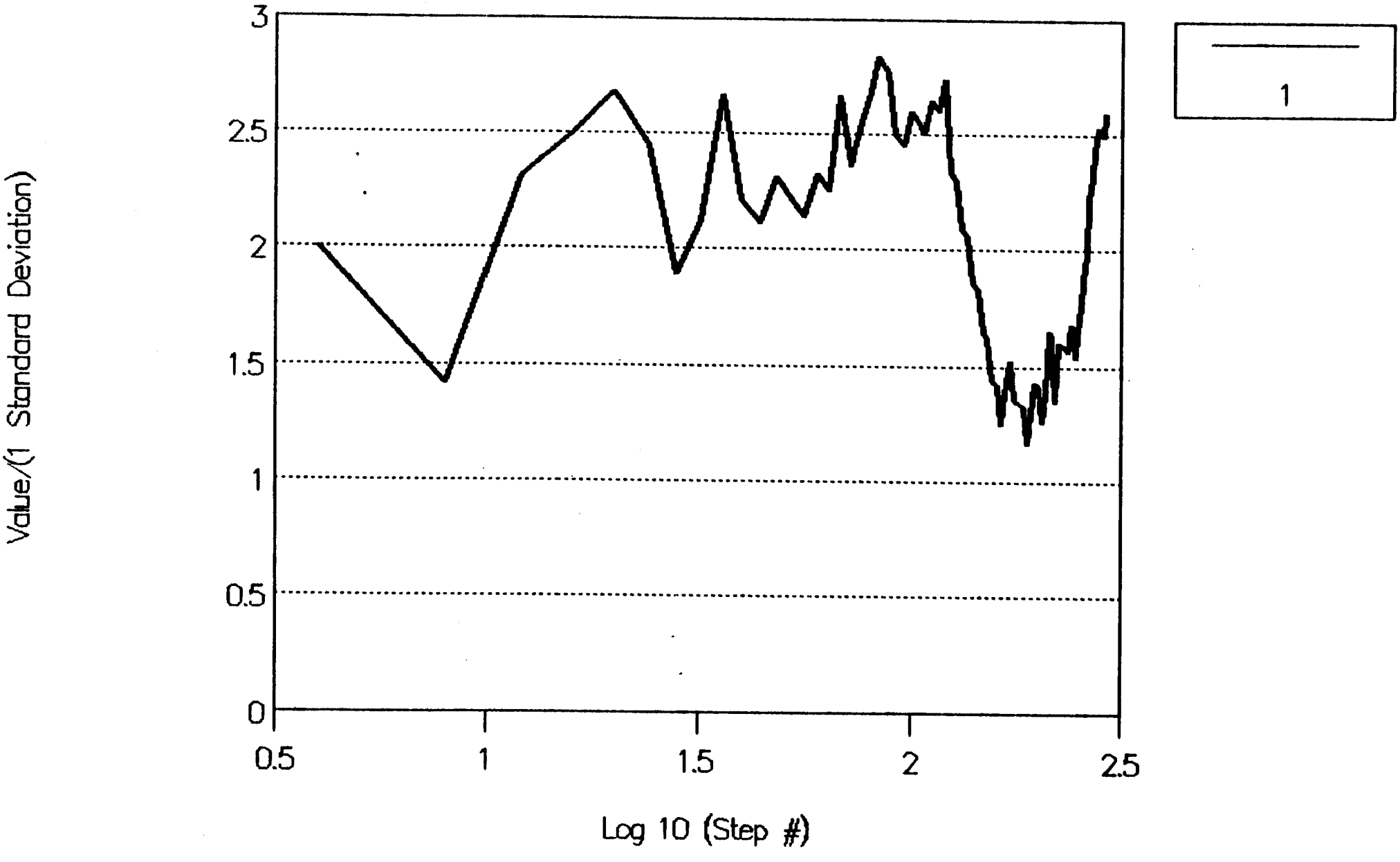

Value ; +1,-1 Standard Deviation
10
8
6
4
2
0
-2
-4
-6
-8
-10
0
50
100
150
200
250
300
Steps of +,-1/2
1
2
3

"Biased" Random walk

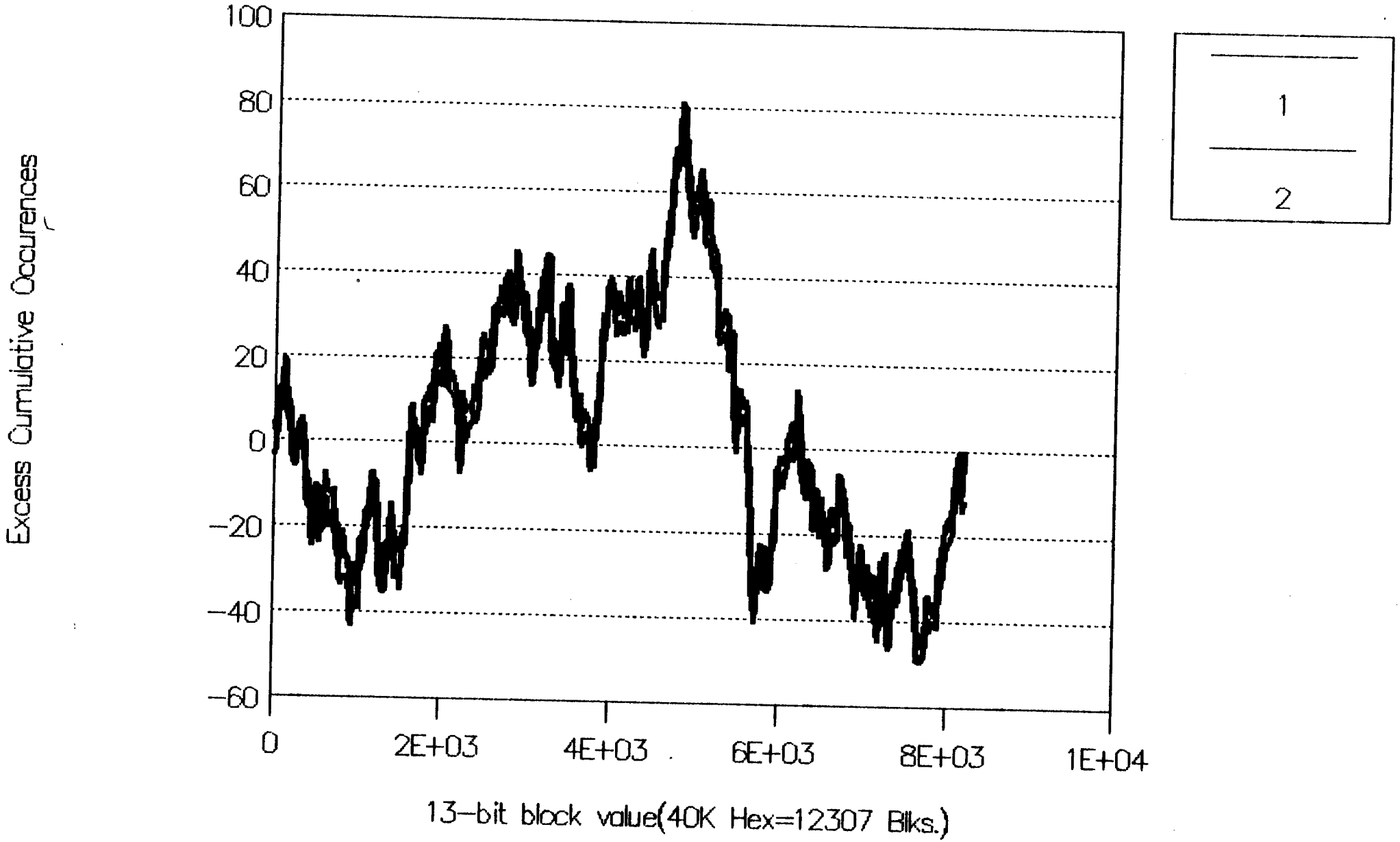


"Biased"

Random walk



K–S test: Sqrt(13), 13–bit blocks



K–S test: Sqrt(13), 13–bit blocks