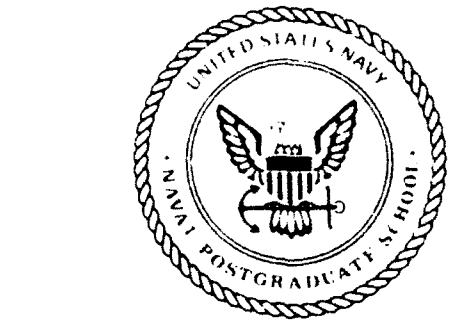


NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ON THE USE OF CHAOTIC DYNAMICAL SYSTEMS TO GENERATE PSEUDORANDOM BITSTREAMS

by

James E. Heyman

March 1993

Thesis Advisor: Jeffery J. Leader

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL *(If applicable)* 31	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943-5000	7b. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)*	10. SOURCE OF FUNDING NUMBERS Program Element No. / Project No / Task No / Work Unit Accession Number

11. TITLE *(Include Security Classification)*
ON THE USE OF CHAOTIC DYNAMICAL SYSTEMS TO GENERATE PSEUDORANDOM BITSTREAMS

12. PERSONAL AUTHOR(S) Heyman, James E.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(year, month, day)*	15. PAGE COUNT
Master's Thesis	From To	March 1993	102

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS *(continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUBGROUP	Chaotic dynamical systems and pseudorandom bitstream generators

19. ABSTRACT *(continue on reverse if necessary and identify by block number)*

There exist a variety of coding applications that require the generation of pseudorandom bitstreams. Such sequences must meet the conflicting requirements that they be reliably repeatable as well as unpredictable. That is, neither knowledge of a small sub-sequence nor an imperfect knowledge of the initial conditions (i.e. the key) will be sufficient to recover the entire sequence.

In this thesis we exploit the inherent unpredictability of a chaotic discrete dynamical system. Specifically, we develop a mapping of the Henon horseshoe attractor into the binary domain {0,1} and demonstrate that the sequences produced meet specified criteria of pseudorandomness.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS	Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Jeffery J. Leader	22b. TELEPHONE *(Include Area code)* (408) 656-2335	22c. OFFICE SYMBOL MA/Le

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

Approved for public release; distribution is unlimited.

On the Use of Chaotic Dynamical Systems to Generate Pseudorandom Bitstreams

by

James E. Heyman
Lieutenant, United States Navy
B.A., Macalester College, 1982

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
March 1993

Author: _____
James E. Heyman

Approved by: _____
Jeffery J. Leader, Thesis Advisor

_____
Mark Stamp, Second Reader

_____
Richard Franke, Chairman
Department of Mathematics

ABSTRACT

There exist a variety of coding applications that require the generation of pseudorandom bitstreams. Such sequences must meet the conflicting requirements that they be reliably repeatable as well as unpredictable. That is, neither knowledge of a small sub-sequence nor an imperfect knowledge of the initial conditions (i.e. the key) will be sufficient to recover the entire sequence.

In this thesis we exploit the inherent unpredictability of a chaotic discrete dynamical system. Specifically, we develop a mapping of the Hénon horseshoe attractor into the binary domain {0,1} and demonstrate that the sequences produced meet specified criteria of pseudorandomness.

TABLE OF CONTENTS

FIGURES

ACKNOWLEDGEMENT

This thesis is dedicated to my brother Tony,

the most chaotic person I know.

I. INTRODUCTION

A. APPLICATIONS OF PSEUDORANDOM BITSTREAMS

There are many applications in a variety of fields that require the use, and hence the generation, of pseudorandom binary sequences (PRBS). A familiarity with these is not central to this thesis but some will be used as examples and, as such, will be reviewed here. The first, and most commonly cited use, is in the field of cryptography where PRBS are used to encrypt and decipher communications. In the following example the ASCII representation of the letter "J" is encrypted and deciphered using a segment of a, for now mysteriously produced, pseudorandom sequence. The method used is to add the plaintext to the encrypting sequence bitwise modulo 2.[1] The ciphertext would then be transmitted and at the other end would be deciphered by adding the deciphering sequence again using mod 2 arithmetic.

```
  1001000    plaintext "J"
+ 0110010    encrypting sequence
  1111010    transmitted ciphertext "z"
+ 0110010    deciphering sequence
  1001000    plaintext "J"
```

It should be clear that the encrypting/deciphering sequence must be held by both the party sending the message and the receiving party. This however, as the masseuse would say, is the rub since there are only two basic ways to do this. The first is to distribute the actual key to all

[1] See Appendix A for a quick lesson in mod2 math.

parties involved. This is essentially the one time pad method and while it is provably a cryptographically strong approach (Shannon, 1949, pp 656-715) it has some obvious physical security and logistical disadvantages. The second is to distribute a sequence generating algorithm either in software or hardware and then pass the algorithm input that was used to encrypt the message, i.e. the key, along with the message or distribute a list of keys every couple of weeks or months (for example). Since the key itself isn't necessarily secure, ***the integrity of the system is dependent on the generator's inherent resistance to attack.***

Another example of using pseudorandom bitstreams is in an account/password scheme. Without implying that this is how it is done consider the common bank ATM which reads an account number off a card and then combines it with an individual's Personal Identification Number (PIN) to either grant or deny access to the account. If the account number and PIN represent two initial conditions for a generating scheme then a conceivable procedure would be to run off several thousand bits and then compare the next, say, 50 bits to the 50-length string that is stored in the bank's records. This can be easily done by adding the two 50 bit strings together bitwise mod2 so that if they match the sum is zero whereas an unsuccessful combination would give a non-zero sum. The strength of this scheme is dependent on the generated bitstream having a high sensitivity to initial

conditions so that a guessed PIN that differs from the correct PIN by just a little results in a different 50 bit sequence. The potential effectiveness comes from the enormity of the task of guessing which of the 2^{50} = 1,125,899,906,842,624 combinations is the right one. Of course, this is a somewhat contrived example since in reality the account number is fixed and all variability must come from the four to six digit PIN which is why the preferred method of hacking into an account is to find out someone's account number and then exhaust the 10^4 to 10^6 possible PINs.

Still another example of an application where pseudorandom bitstreams are used is spreading codes (NSA 1981 Ch. 3). These are used in two very different ways. The first is a communication scheme that repeats a given bit in the plaintext n times and then adds a bit from a pseudorandom bitstream to each of them. Although this reduces the effective transmission rate by a factor of the "chip rate" it also reduces the signal to noise ratio to the point that the signal actually looks like noise. Thus security is gained not so much by enciphering the message but rather by hiding the existence of the message itself. The other use of spreading codes is to allow multiple signals to be carried over a single frequency simultaneously which is a basic description of how the Global Positioning System (GPS) receiver, operating at a chip rate of 10 Mhz,

can sort out the signals for all the satellites in view even though they are broadcasting on the same frequency. Note that this highlights the need for strong autocorrelation properties which will keep the channels from overlapping.

B. HOW RANDOM IS ADMORN?

That the fourth word of this section's title is an anagram of the second word is pretty obvious. While it is tempting to say that "admorn" is "random" mixed up randomly, the permutation was determined by the roll of a die. So on one hand, before the mix-up there was no hint of which of the 6! possible spellings would occur and thus the spelling is unpredictable and therefore random. But on the other hand, the procedure was well defined and didn't depend on any processes that aren't covered by known physical laws so it would seem that it isn't random. One thing is for certain however, and that is that it cannot be both. Combined with the earlier discussion on applications this leads us to the somewhat fundamental question of what is meant by the term pseudorandom as a descriptor of a bitstream. A good starting point is the common intuitive notion of randomness which perhaps is best embodied in the common coin flip. Before we let a humble quarter decide such earth shattering events as which team shall choose whether to receive the kickoff we make some implicit, albeit basic, assumptions. Among these are that no matter who the flipper is heads are as likely as tails and that, no matter

how the flip has come up before, the next one is independent of the previous events. On the surface of it this would seem to fulfill the requirements that a random process be neither predictable nor repeatable. But it is here that we must make an important distinction as to what is meant by predictable (and also repeatable). Although it may seem haphazard a coin flip is, in fact, controlled by physical laws. The thumbnail is a certain distance from the coin centroid, the coin has a particular mass, the air a specific density and so on. As such, it is a deterministic process and thus not really random after all. But if true randomness requires a non-deterministic process can there ever be a truly random event? Without getting into the theology of a god who plays dice with the universe (a questionable example since the die toss is itself deterministic) we can limit our statement to say that the predictability of the coin flip may be beyond the ability of us mere mortals but we must differentiate between what we can predict and what is in fact predictable (Stewart, 1991, p. 286). This gives us the first clue as to what constitutes a pseudorandom process: ***not easily predictable but still repeatable.*** Which is quite similar to our basic working definition of chaos which describes a process as chaotic if it is "random" (i.e. non-predictable) and deterministic (i.e. repeatable). Granted that when viewed in the context of chaos the term "non-predictable" refers

more to a sensitivity to initial conditions, which will be discussed as part of the individual generating schemes, but for now we will proceed with this suggestive parallel between pseudorandomness and chaos.

C. GENERATORS VS. SEQUENCES

At this point it is important to distinguish between a sequence generator and the sequence generated. An ideal generator will be extremely sensitive to initial conditions, computationally simple, and will give rise to sequences that have long periods. As discussed earlier, the first of these requirements is tied to the unpredictability of the system. The second, whether the system is implemented in hardware or software, is necessary to achieve the high bit rates that some applications such as spreading codes require. The third criterion comes from two sources. The obvious one is that a coding scheme that is periodic within a single application is very weak. A slightly more subtle advantage comes from eliminating an inherent disadvantage of using linear or non-linear shift registers. Shift registers, by their nature, have a fixed period and the only way to get a longer period is to build a bigger shift register. But if a simple scheme were to have sufficiently long period then any practical length requirement could be met just by letting the generator run longer. The potential advantage of constructing a generator in this manner is to exploit the

central tenet of chaos: simple systems can produce complicated results.

Our proposed method of generating bitstreams is to use a discrete dynamical system that behaves in a chaotic manner and is at least thought to possess a strange attractor. We then map the orbit into the binary domain which results in a bitstream whose pseudorandomness can then be investigated via a series of tests. An obvious concern at this point is whether computing the orbit on a finite precision machine causes us to lose the chaoticity on which this entire scheme is dependent. Fortunately, by the shadowing lemma, this is not a problem since the overall effect of the finite precision is to push the orbits into a chaotic realm of their own (Peitgen, Jürgens, Saupe, 1992, ch. 10.8). The simplest way of explaining this is that the use of floating point numbers and the associated arithmetic induces a small error but this error will be picked up by the system's sensitivity to initial conditions. No problem. Granted the orbits will be different than if calculated with infinite precision but all of the chaoticity will be preserved. Whether or not the bitstream is chaotic is a question that can be answered, in short, in the affirmative. The proof of this is not terribly difficult but it is a tad irrelevant as it is enough to show, for coding purposes, that the sequence is pseudorandom in some sense. This distinction goes to the core of our work which is to show that the output of a

chaotic dynamical system can be mapped from the floating point domain to the binary domain in such a way that the resulting binary stream will be pseudorandom.

In the end we decided to use the Hénon horseshoe map because it is thought to be chaotic and, equally important, it is easy to calculate. This attractor will be described fully in chapter IV.

D. TESTS FOR PSEUDORANDOMNESS

In evaluating the pseudorandomness of binary sequences there are several criteria that need to be met. Some of these, such as sensitivity to initial conditions and period length, deal with the characteristics of the generation scheme itself. Others are concerned with the concept of how "random" the actual sequence is. For this latter group we will follow the approach used by R. Forré (1990) and base the discussion on the basic randomness postulates: runs, balance, and autocorrelation (Golomb, 1982, pp25-27). In addition we will also consider the linear complexity profile measure proposed by Reuppel (1986, ch. 4).

Although all of these tests are explicitly defined, what constitutes a passing grade is either not delineated or is described so closely that it is applicable only in theory. This is not as big a problem as it may seem since the entire question of how random is random enough is application dependent and a given application may have different requirements. A password/account scheme needs to

have a high sensitivity to initial conditions, a cryptographic sequence might require a high level of linear complexity while a spread spectrum application depends on a good autocorrelation profile to maintain channel separation.

Rather than get bogged down in trying to enumerate the technical requirements of specific uses we will use a two-pronged approach of describing a general but practical requirement. The first is to examine what these statistical tests result in when applied to a process that is reasonably assumed to be random (or at least non-repeatable). The second is to develop an understanding of the impact of each statistical consideration from the standpoint of someone trying to predict/guess the entire sequence based on the knowledge of only a portion of it. Rather than define these concepts abstractly we will describe them with two examples of bitstreams, one of which shows good pseudorandom properties while the other does not.

II. COIN FLIP SEQUENCE (THE GOOD)

A. METHODOLOGY

As an example of "randomness" we generated a binary sequence by flipping a coin 1000 times and assigning a 1 for heads and a 0 for tails.[2] The previous discussion of the deterministic nature of a coin flip is relevant to a point, in that if the height of the flip is limited to 4-6 inches then the process takes on the characteristics more of juggling in that the coin will spin a particular number of times. In that regime we found that heads could be made to come up almost 70% of the time. Above that height, however, it takes on the look of baton twirling in that it goes up spinning but how it comes down is for all practical purposes unpredictable and unrepeatable. Thus this limited range could be considered to be the "linear" regime of the process whereas a flip that goes above 6" will be said to enter the "non-linear" zone. Be that as it may, we believe that most people would accept, since we used the non-linear zone, that this represents a reasonably "random" process.

B. BALANCE

The first and most obvious consideration is balance. Golomb's first postulate says that, given a proposed pseudorandom sequence, the difference between the number of 1's and 0's should be no more than one (Golomb, 1982, p.25). Even he acknowledges that this is a little on the strict

---

[2] See Appendix D for the coin flip binary sequence.

side and refines this to say that the number should be "nearly equal". The value of this is obvious since if we consider a guess-the-sequence situation and the sequence was not balanced then we could simply guess all 1's or all 0's and make great headway. This concept also appeals to the intuitive sense that in flipping a coin there should be an equal probability of heads or tails.

The following figure displays the running mean of the coin flip sequence[3]. The graph depicts the expected central tendency pull and the final fraction that is heads is .486.

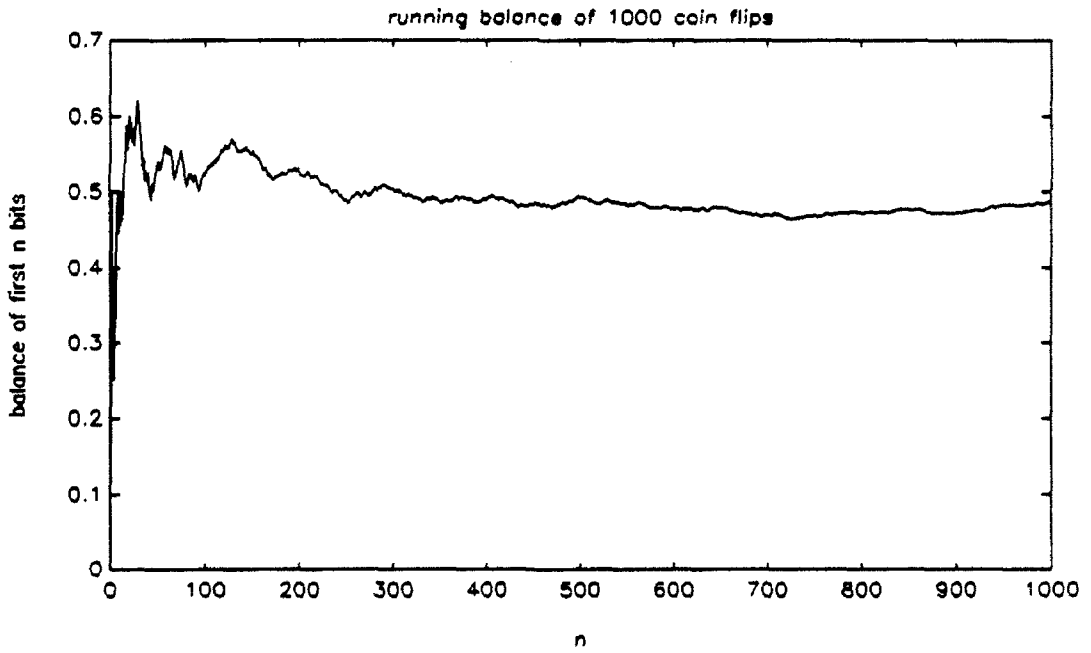


Figure 1: Coin flip balance

[3] See Appendix C for runbal.m program listing.

This still leaves open the question of how close to .5 the balance has to be. While different users will have different needs we will adopt a 95% confidence interval on the mean of a 1000 point random binary sequence. This results in a calculated mean, and thus the balance, between .47 and .53 being judged good enough. This is also consistent with the range of acceptability given by H. Fredrickson of the Naval Postgraduate School.[4]

C. AUTOCORRELATION

The next consideration is a sequence's autocorrelation. This is a measure of how well the sequence correlates to shifts of itself and so we shall start with a small digression into what is meant by correlation. For an individual correlation of two distinct sequences, we use the measure of the absolute value of agreements minus disagreements divided by the number of agreements plus disagreements (aka the total number of bits). The "obvious" way of measuring this might seem to be to take the number of agreements and divide by the total. The following example will serve to compare the two measures.

```
0101101100   original sequence
1111010110   guessed sequence
 x x   x x   agreements
```

There are four agreements and six disagreements which results in an "obvious" correlation of .4, but using our

---

[4] Classnotes from Fall 1992 MA4570 Cryptography class.

method the correlation is .2. Both of these methods share the property that if two sequences are similar then the correlation will be near one while if they are dissimilar they will be near zero. The benefit of the non-obvious (and correct) way is that, considering the prospect of guessing the sequence, there is a penalty for wrong guesses. Consider the example of trying to guess at a perfectly balanced sequence by guessing all 1's. Under the "obvious" measure these two sequences would have a correlation of .5, but there hasn't really been any progress in learning about the sequence. By using agreements minus disagreements the score would be zero and this reflects a discounting of lucky guesses. It's somewhat akin to playing billiards where uncalled shots do not count.

Back to autocorrelation. The autocorrelation is the correlation of every shift of a sequence with the sequence itself.[5] Ideally, when applied to an infinite length aperiodic sequence, this function should be two-valued and equal to one if the sequence is not shifted and zero otherwise (Golomb, 1982, p. 26). For the finite length sequences that we will work with the latter expectation is revised to "something small". In the following graph, the zeroth, or unshifted, lag is moved to the center. As exhibited, for a sequence with good autocorrelation property

---

[5] See Appendix C for autocorr.m program listing

we would expect to see a spike in the middle with the curve tapering off rapidly in each direction.

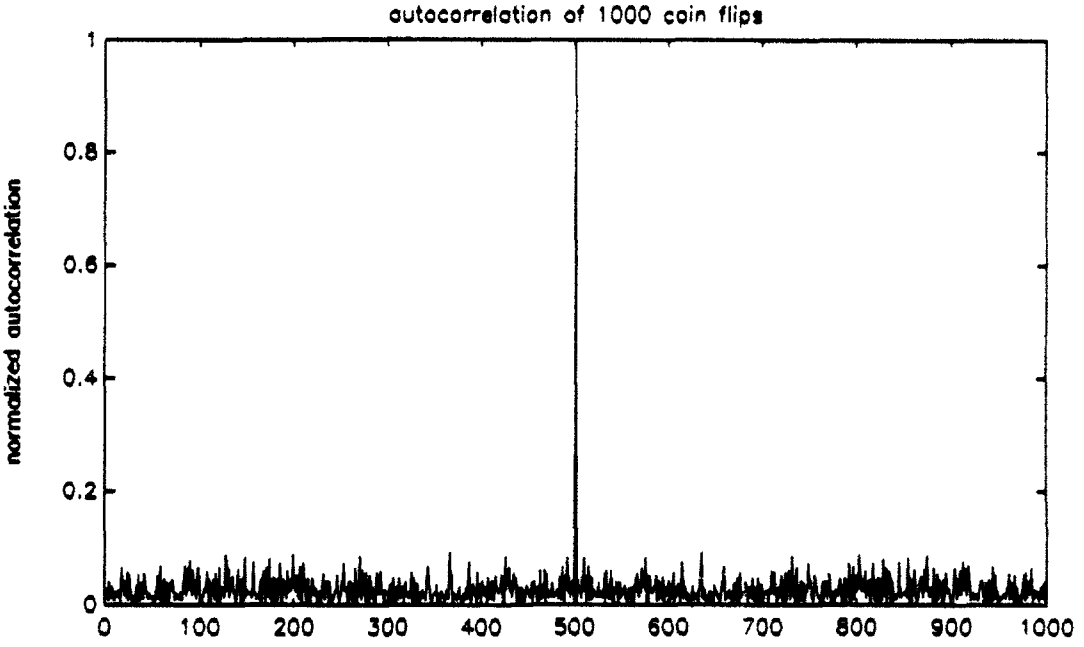


Figure 2: Coin flip autocorrelation

The exact shape and size of the sidelobes for a passing grade is once again dependent on the specific application but for general discussion we have adopted the standard that the sidelobes should be below a value of .1 when the spike at the zeroth lag is normalized to one.

D. RUNS

The runs property is one that is best described intuitively. The idea is that in a given sequence there should be no sub-sequences of length n that show up any more frequently than any other sequence of the same length. For

example, there are only four possible sub-sequences of length two (00, 01, 10, 11), corresponding to the decimal numbers (0, 1, 2, 3), and each should show up one fourth of the time. Precisely it is that in a sequence, 1/(2^i) of the runs should have length i. Once again this is a bit strict in practice since only full-length shift registers (FLSRs), of all (general) linear feedback shift registers, are guaranteed to have this property (Golomb 1982 p. 44). If one is not dealing with FLSRs then we suggest that passing this test perfectly would indicate that the sequence has a degree of regularity that is perhaps too high. Rather along the lines of having an event that happens, on average, 52 times a year actually occurring exactly once per week. We will examine the value of this test later in a guessing context but for now we state that a graph of runs that does not display any obvious peaks or valleys will be considered well balanced n-tupely speaking.[6] The following seven graphs indicate how performance on this test degrades as longer length runs are considered.

---

[6] See Appendix C for runs.m program listing.

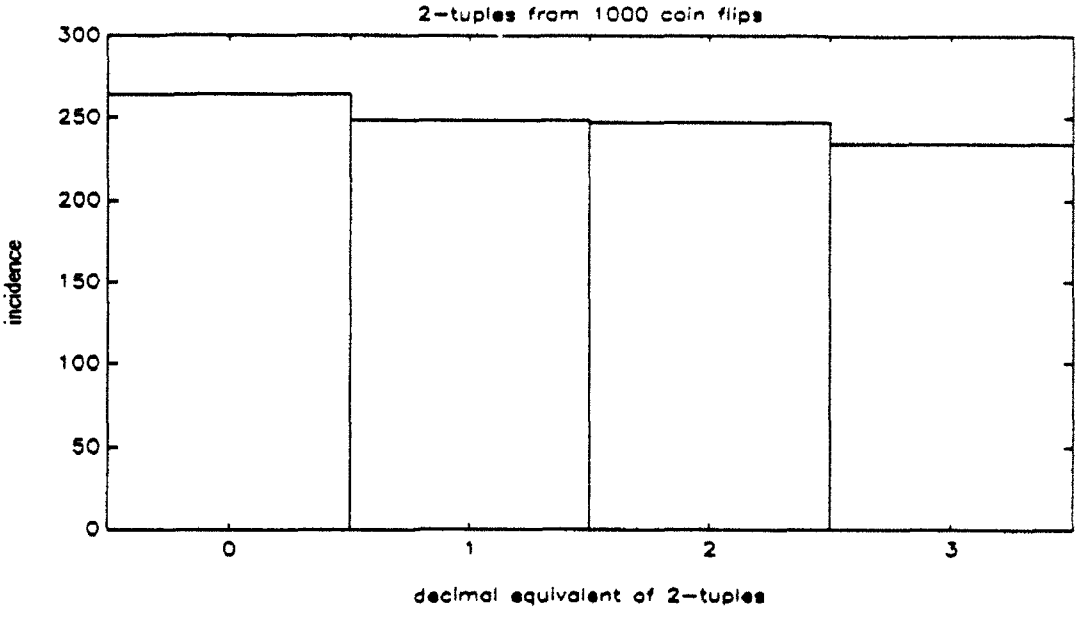


Figure 3: Coin flip runs of length 2

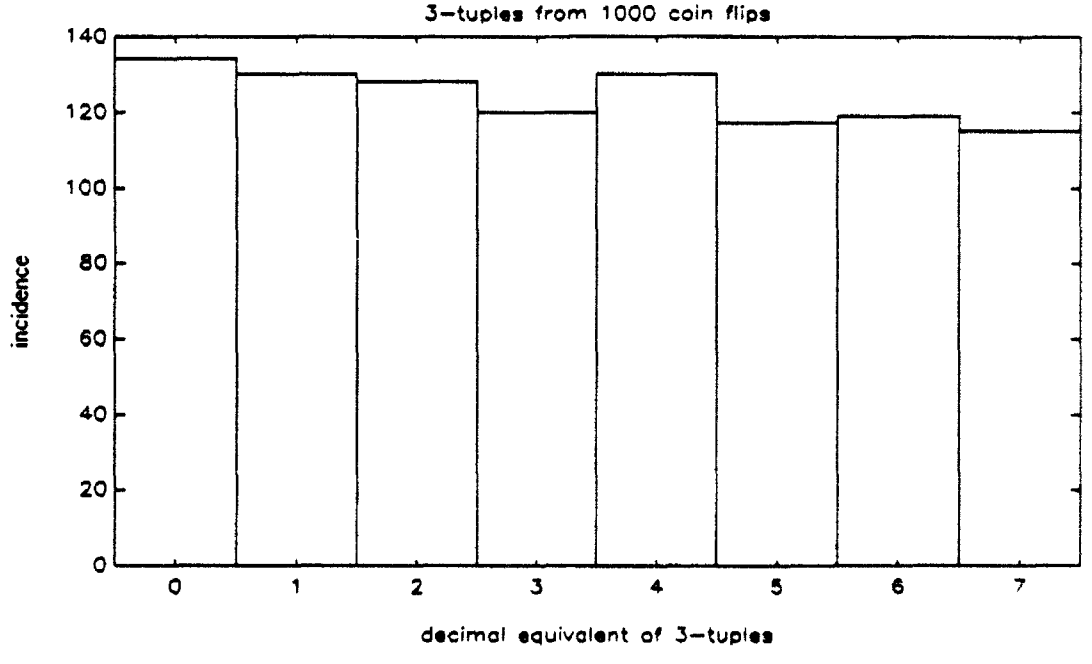


Figure 4: Coin flip runs of length 3

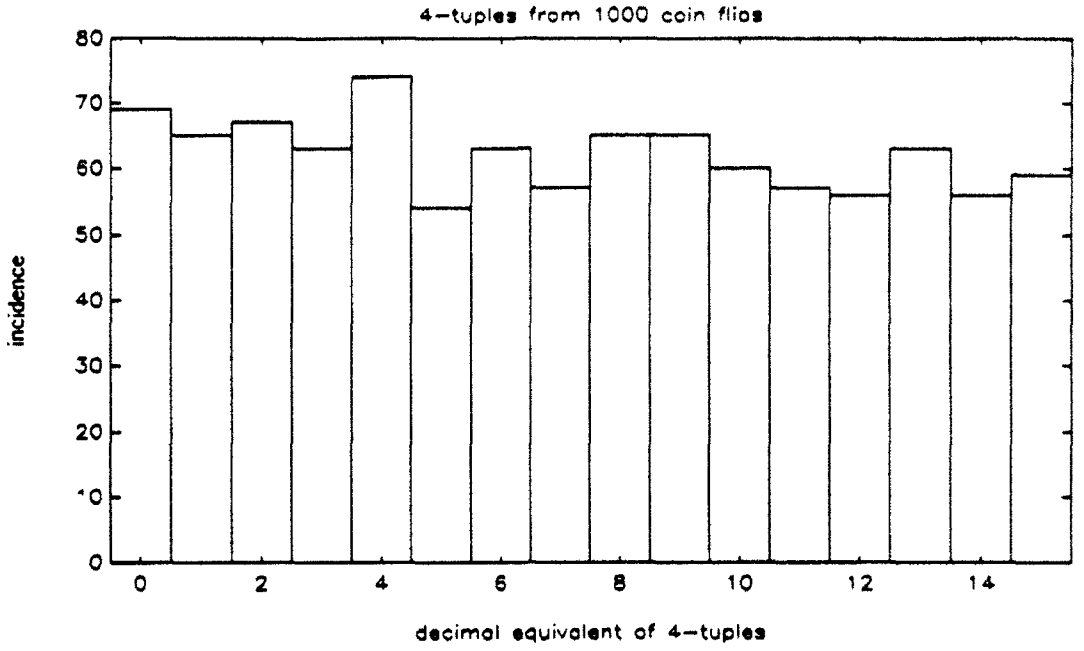


Figure 5: Coin flip runs of length 4

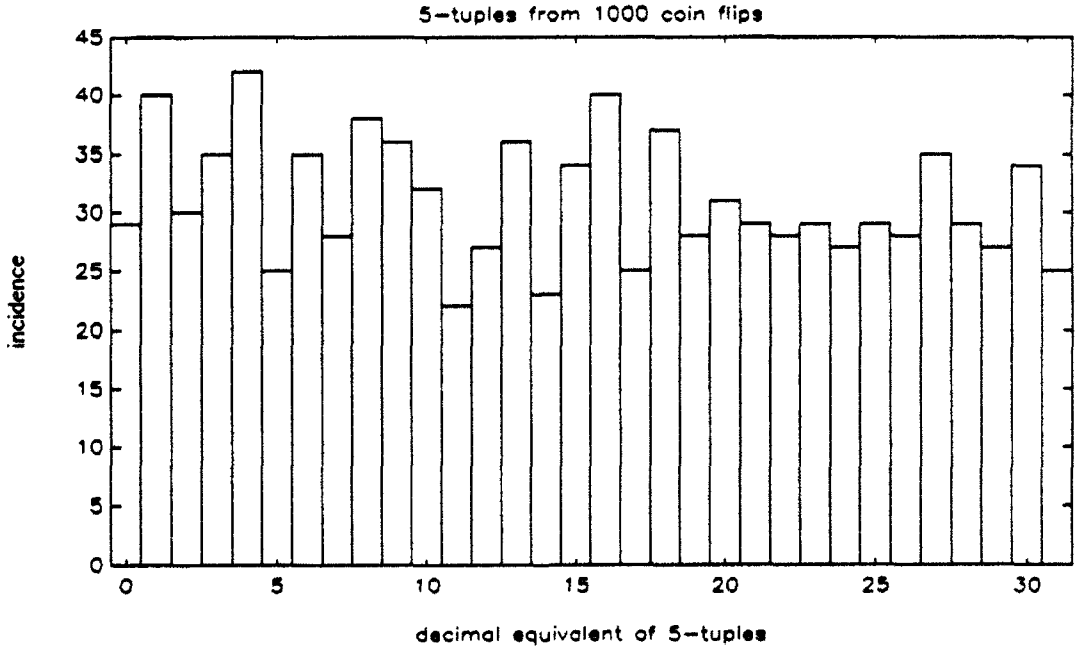


Figure 6: Coin flip runs of length 5

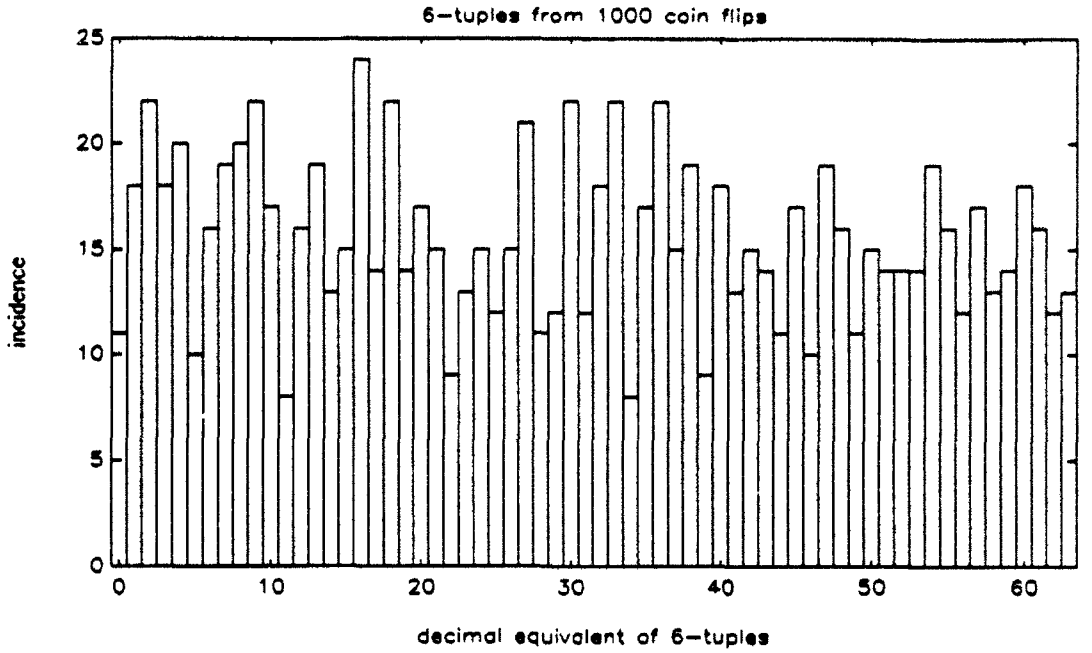


Figure 7: Coin flip runs of length 6

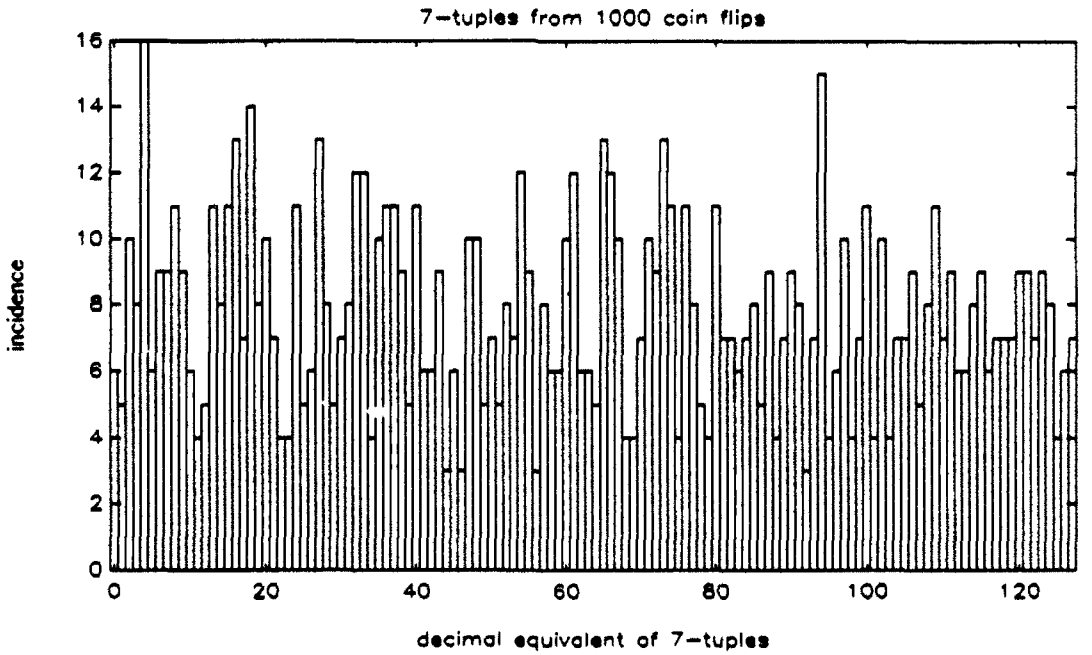


Figure 8: Coin flip runs of length 7

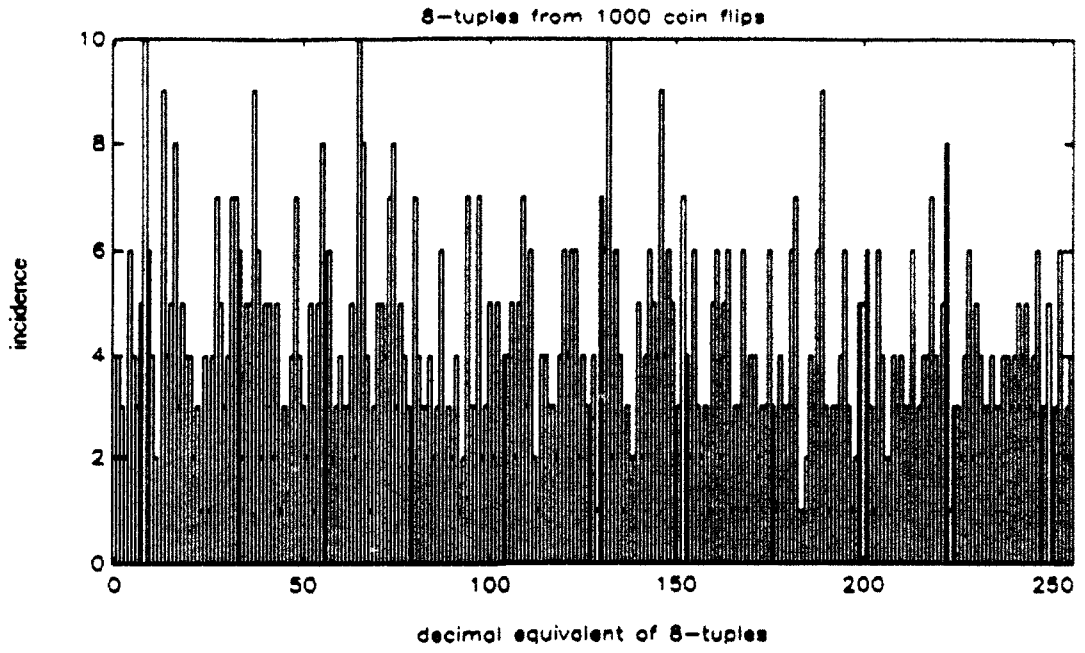


Figure 9: Coin flip runs of length 8

E. LINEAR COMPLEXITY

The final test is the linear complexity of the sequence, which is the minimum length linear feedback shift register[7] that could have produced the sequence.[8] The Berlekamp-Massey algorithm is based on the fact that, for an n stage shift register, the n+1st bit is determined by the 1st through nth bits. Likewise the n+2nd bit is determined by the 2nd through n+1st bits and so on. Doing this n times results in a solvable system of n equations in n unknowns which, when solved, gives the shift register function.

---

[7] See Appendix B for a quick lesson on shift registers.

[8] See Appendix C for lincomp.m program listing.

Since no more than 2m bits are required to determine a shift register generated sequence of length 2^m-1, in a pseudorandom sequence the first n bits should have a complexity of n/2 (Rueppel, 1986, p. 33). This ideal results in the profile roughly tracing the n/2 line as can be seen in the figure below (here we have used an algorithm which approximates the linear complexity). Note that there are several areas where the profile temporarily deviates from the ideal curve.

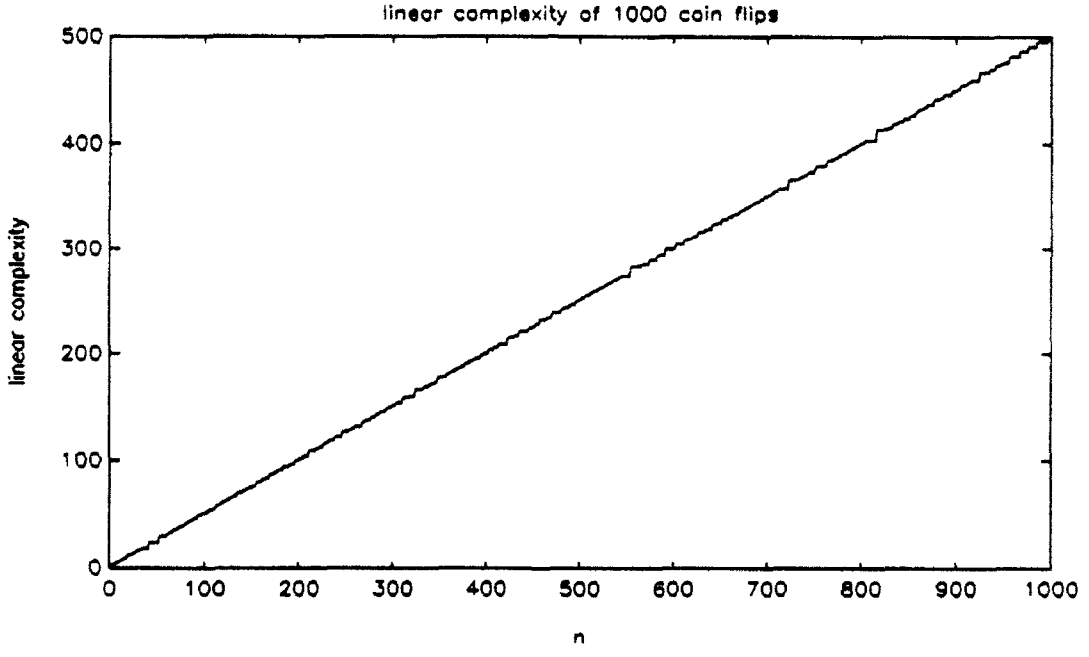


Figure 10: Coin flip linear complexity

This serves as an example of how in a truly random sequence there will be regions in which statistical clumping occurs. In these areas there are sub-sequences that can be produced by a shorter shift register. Eventually, however, the curve will return to the n/2 line in accordance with central tendency theory.

The process of calculating the linear complexity is actually based on finding the generating function and tracking how the degree of this function changes. However, it should be recognized that what is really found is a function that could have generated the sequence and thus serves more to simulate the actual generation process.

III. STACKED SINE WAVE SEQUENCE (THE BAD)

A. METHODOLOGY

By way of contrast we constructed another series by adding three scaled sine curves together, throwing in a small periodic sub-sequence, and then repeating most of the sequence to make it almost periodic. To convert the sequence from the rational domain to binary we set a split point and assigned a 1 to any number greater than that point and 0 to the rest.[9] Obviously we do not advocate this as a generation scheme (we are professionals, please don't try this at home) but the sequence created is instructive when tested in the same way as the coin flip sequence.

[9] See Appendix E for bad binary sequence.

B. BALANCE

The following figure displays the running mean of the bad sequence. Although the graph exhibits the desired, and anticipated, settling down to a mean value it turns out that the actual value is .545 which is outside of our acceptable range. This points out the need to look not only for convergence but also at the sequence's sample mean.

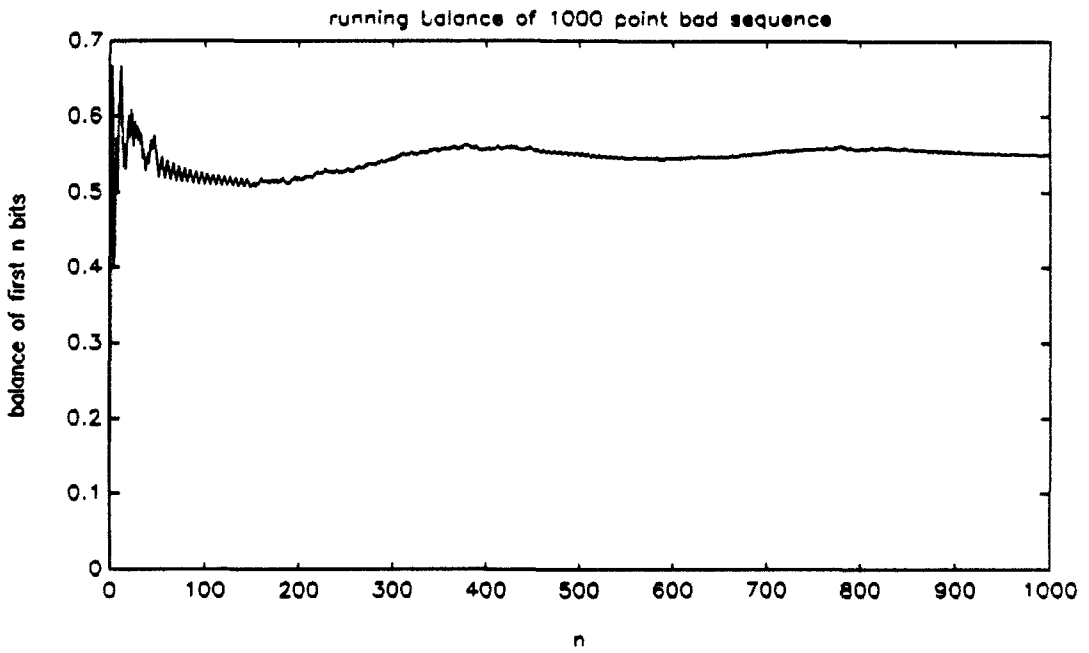


Figure 11: Bad sequence balance

C. AUTOCORRELATION

The autocorrelation test of this sequence gives a graph that is a fine example of what the graph looks like when the test is failed.

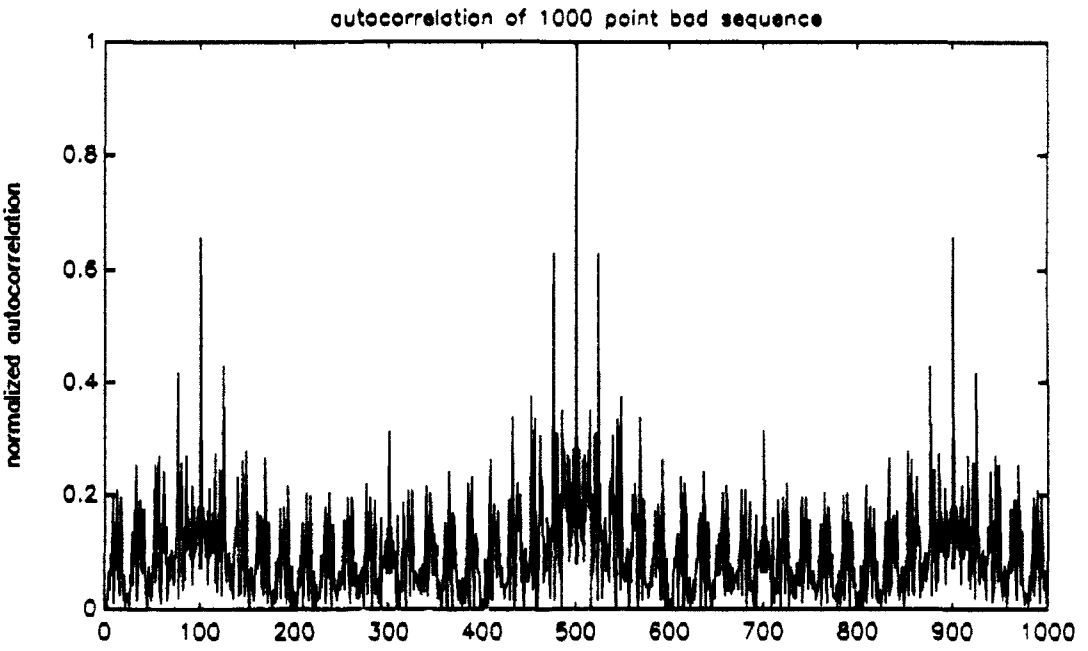


Figure 12: Bad sequence autocorrelation

There are two points of interest in the above graph. The first one is the large spike that goes over the .6 level. This is indicative of a sequence that is close to being periodic (which would result in a second spike of unit height). Slightly more subtle is that the general level of the sidelobes can be seen to be about .2 which signifies that, in general, this sequence looks similar to most shifts of itself.

D. RUNS

As mentioned earlier, besides differentiating between full length and non-full length shift registers, the principal rationale for using the runs tests is an intuitive feeling that there should be no potentially exploitable bias in the occurrence of arbitrary length sub-sequences.

The following bar charts depict the run incidence up to 8-tuples. Note that although none are as flat as the coin flip case the severe degradation into sharp peaks and valleys does not occur until the 5-tuple case.

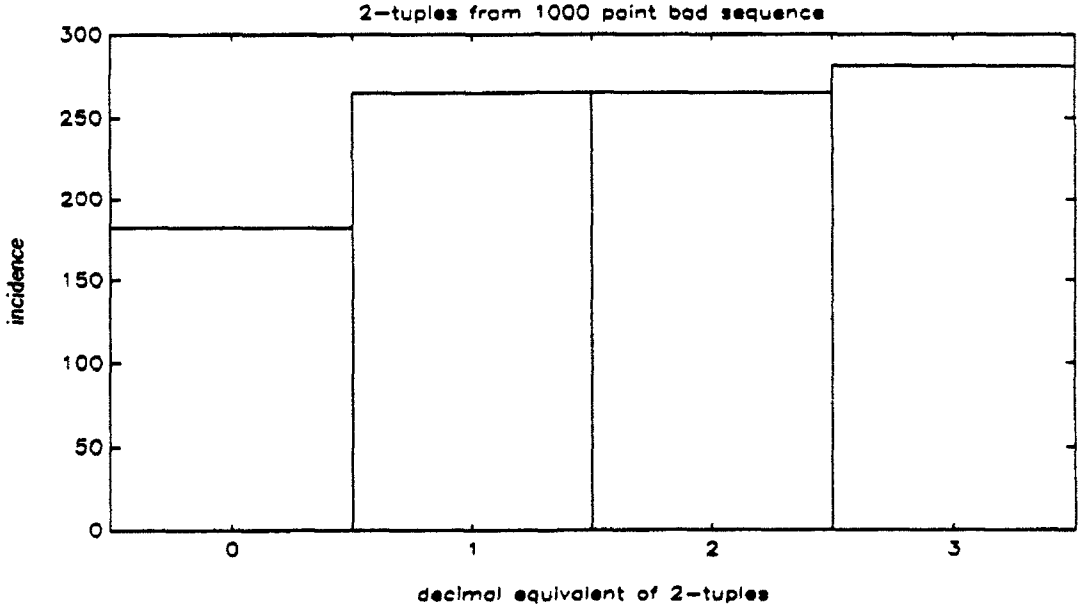


Figure 13: Bad sequence runs of length 2

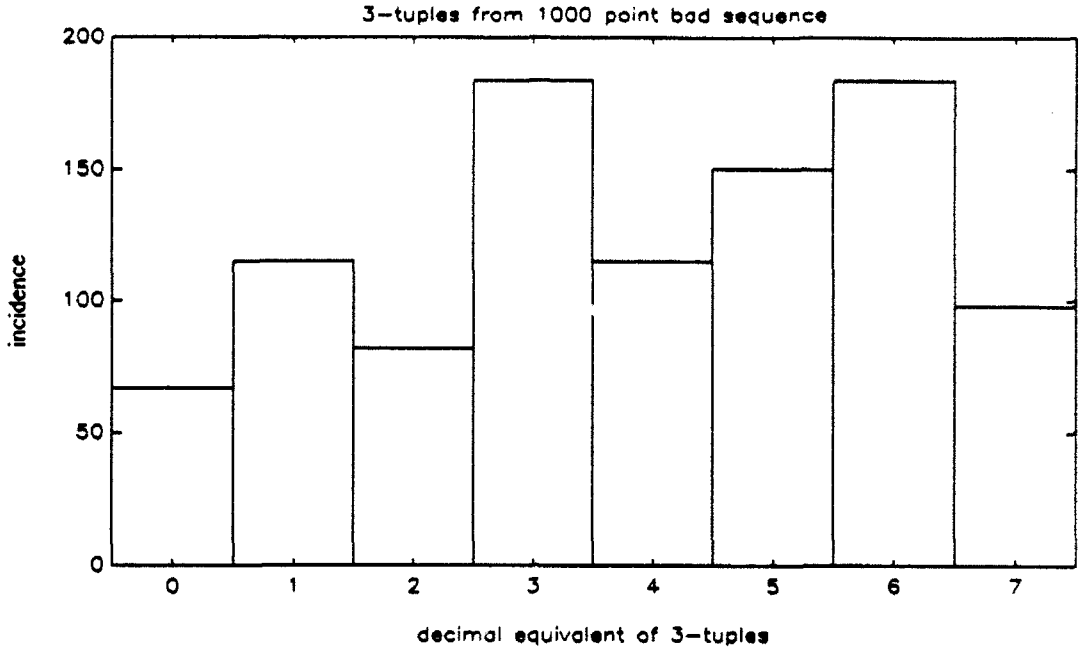


Figure 14: Bad sequence runs of length 3

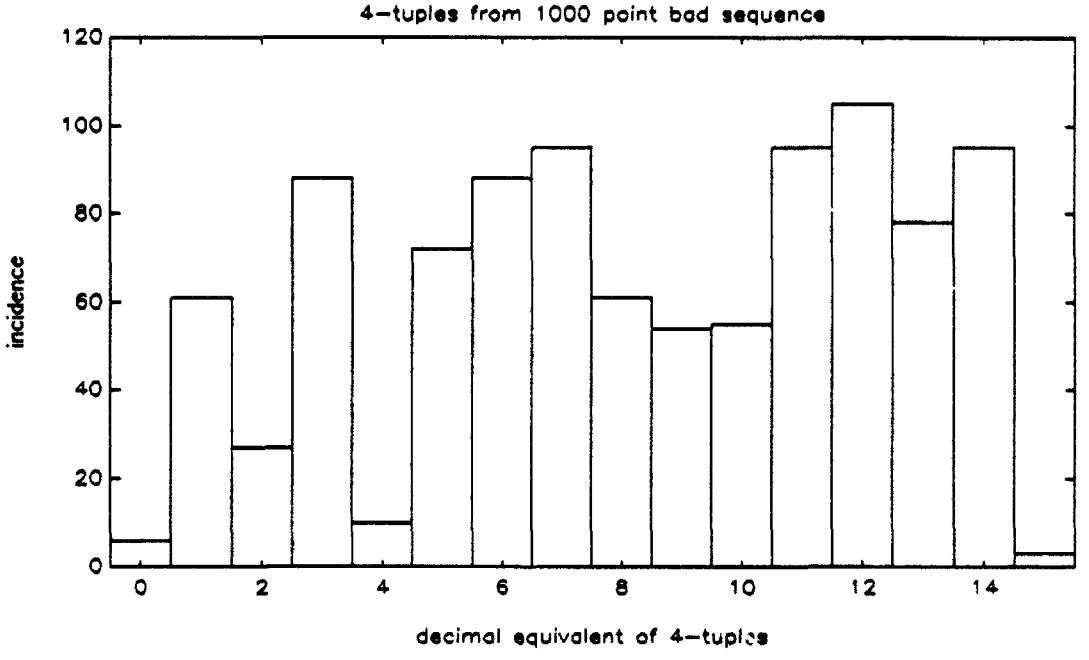


Figure 15: Bad sequence runs of length 4

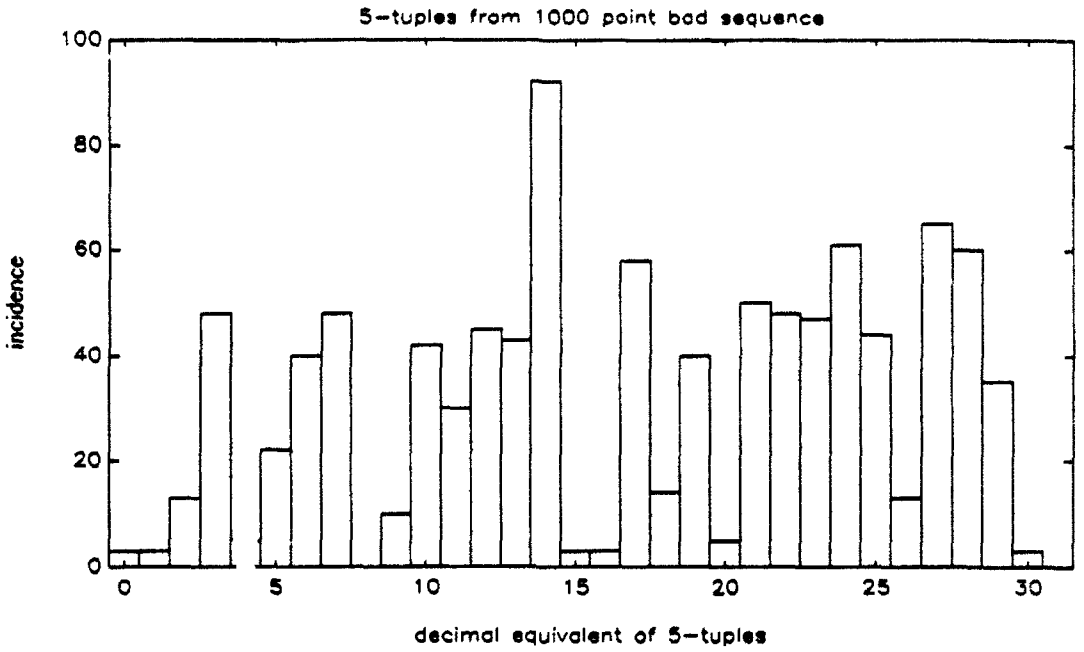


Figure 16: Bad sequence runs of length 5

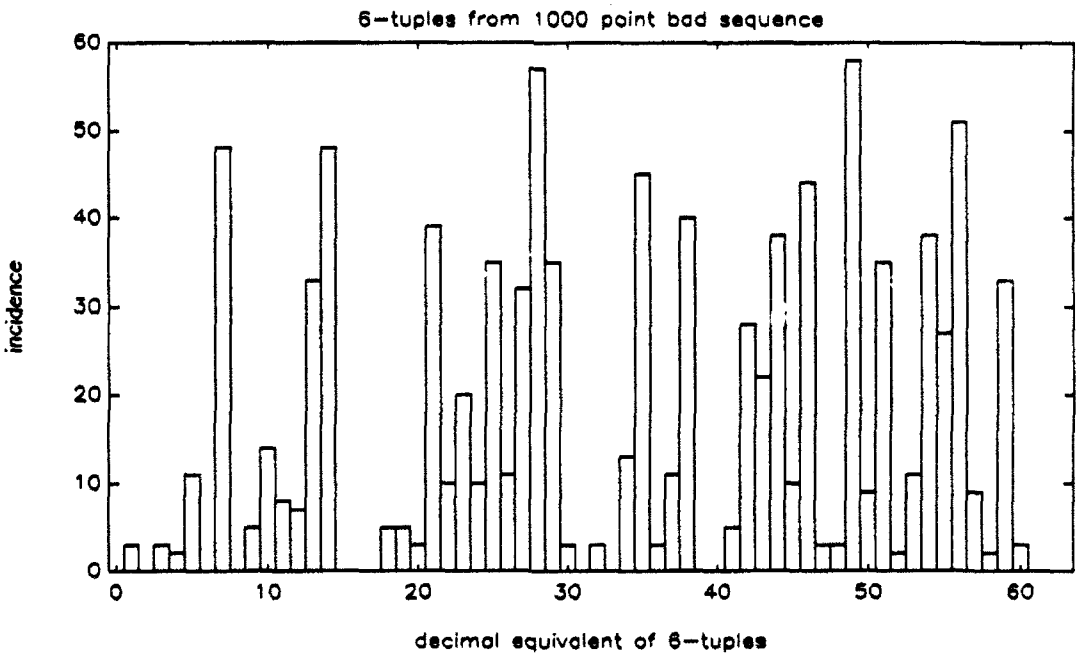


Figure 17: Bad sequence runs of length 6

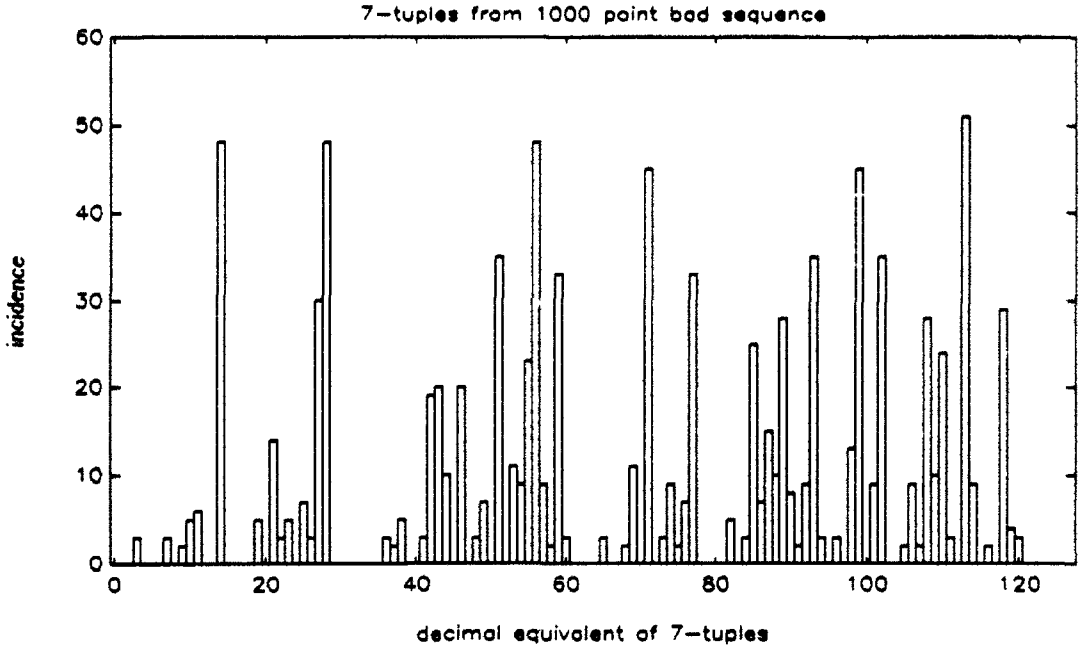


Figure 18: Bad sequence runs of length 7

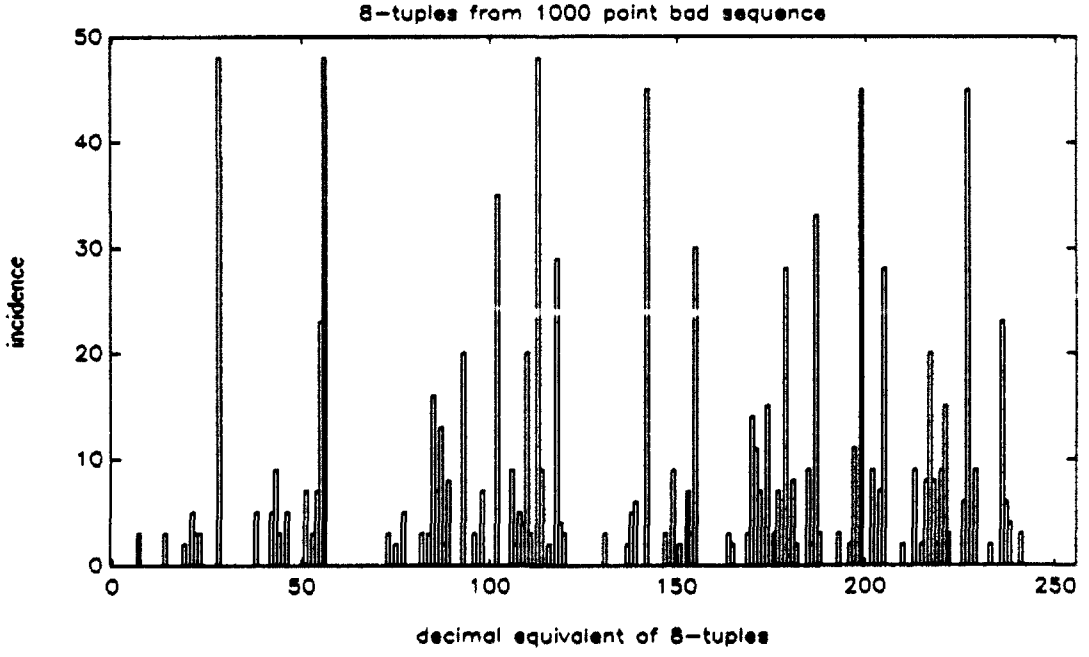


Figure 19: Bad sequence runs of length 8

The results of the runs tests can best be understood by summarizing which runs show up most frequently in the longer tests.

Table 1: Most frequent n-tuples in decreasing order

5-tuples	6-tuples	7-tuples	8-tuples
01110	110001	1110001	01110001
11011	011100	0111000	00111000
11000	111000	0011100	00011100
11100	001110	0001110	11100011
10001	000111	1100011	11000111

By examining the above table it would seem that the most common sub-sequences are 000 and 111. For a slightly different slant on the same tests one could look at those sequences that do not show up at all. There are two 5-tuples, fourteen 6-tuples, 56 7-tuples, and 161 8-tuples that do not occur. A quick look at the earlier bar charts reveal that in many of these there are longer strings of all 0's and all 1's. Admittedly the point of this paper is not how to attack an enciphering scheme but it seems somewhat obvious that guessing 111000111000... (or a shift thereof) might prove fruitful. In fact, the correlation of the actual sequence to the periodic sequence just described is .09. Although this is below the level of the sidelobes of

the coin flip autocorrelation and much below the sidelobes of the bad sequence autocorrelation it is not as innocuous as it might seem. By way of anecdotal evidence, a typical cross correlation of two 1000 long binary sequences generated by the MATLAB random number generator is about .035. We will put stricter bounds on acceptable sequence cross correlations in Chapter IV but for now will suggest that .09 is high enough to conclude that progress has been made in guessing the original sequence.

E. LINEAR COMPLEXITY

As opposed to the linear complexity of the earlier case the following graph indicates a number of problems.

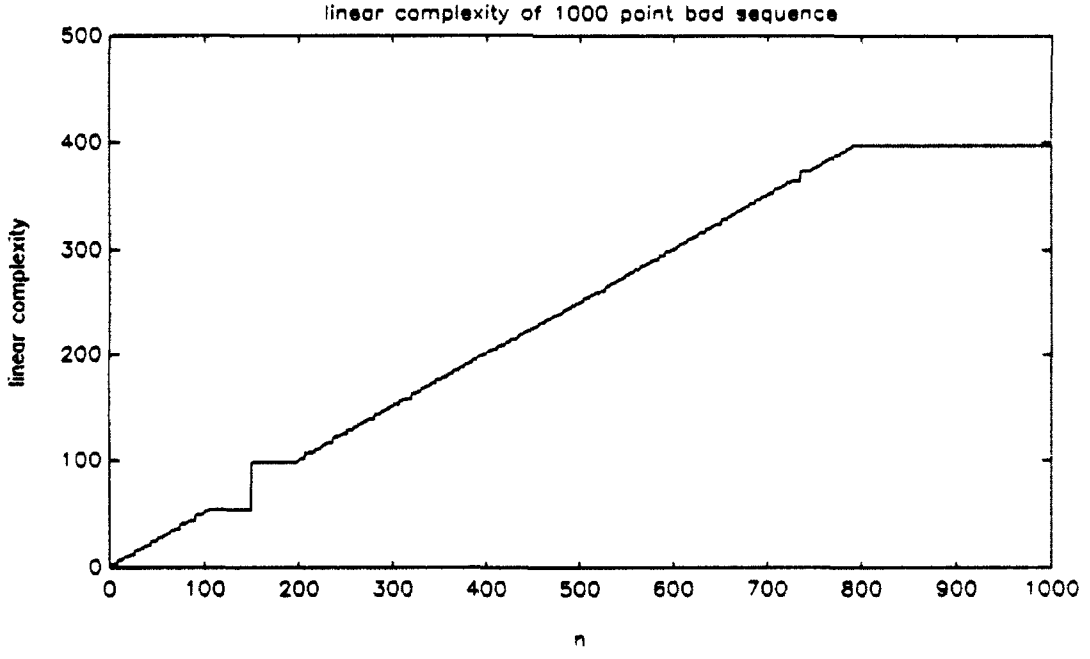


Figure 20: Bad sequence linear complexity

The first are the flattened portions for n between 100 and 200. This indicates the possible presence of a periodic sub-sequence and, as such, the linear complexity ceases to increase with the overall sequence length. Once the sub-sequence passes the curve returns to the n/2 line. The other conspicuous troublespot occurs at n=800. The sudden and permanent flattening of the linear complexity curve signifies that the entire sequence has been simulated before all 1,000 datapoints have been used.

IV. HÉNON GENERATED BITSTREAM (THE CHAOTIC)

A. METHODOLOGY

Our proposed method for generating pseudorandom bitstreams is built on one characterization of chaos, namely, that simple systems can produce complicated results. Specifically, it is based on the Hénon horseshoe mapping, described mathematically as:

$$x_{i+1} = y_i + 1 - 1.4x_i^2$$
$$y_{i+1} = .3x_i$$

This system of equations has properties that make it particularly applicable in light of the coding requirements described earlier (Peitgen, Jürgens, Saupe, 1992, p. 671). The first of these is that it has an attractor and thus the orbits form a pattern not unlike how iron filings follow the "lines of force" of a magnet. This permits the discussion of the ensemble of orbits as having common and specific mean, median, and other statistical properties. By way of example, the following two graphs highlight this fact by showing that, although the orbits come from two different initial conditions and thus follow different paths, they end up looking the same after very few iterations.[10]

---

[10] See Appendix C for henreal.m program listing.

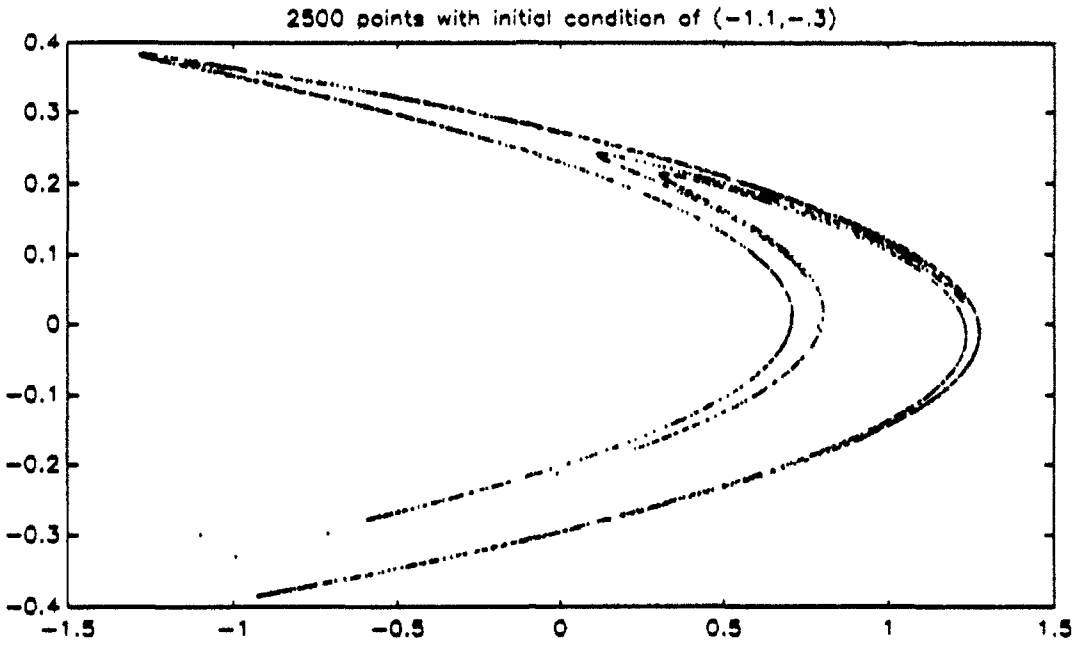


Figure 21: Hénon attractor #1

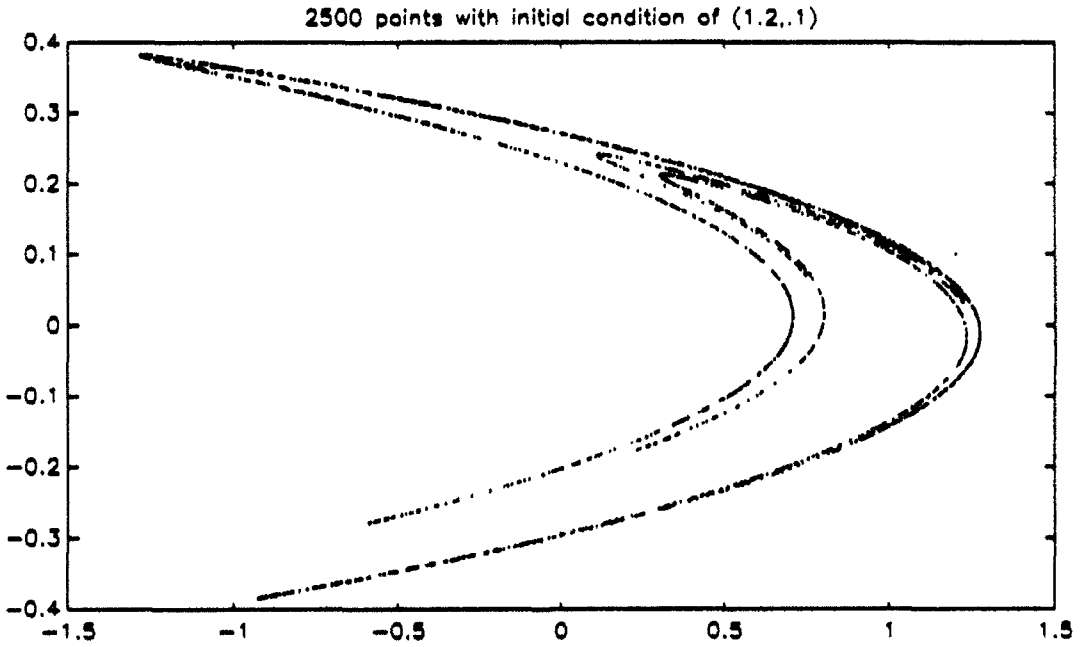


Figure 22: Hénon attractor #2

The second is that the system is chaotic and thus highly unpredictable. This sensitivity to initial conditions can be seen in the following picture which shows that even when they differ by as little as .00001 the subsequent orbits diverge significantly after a small number of iterations.

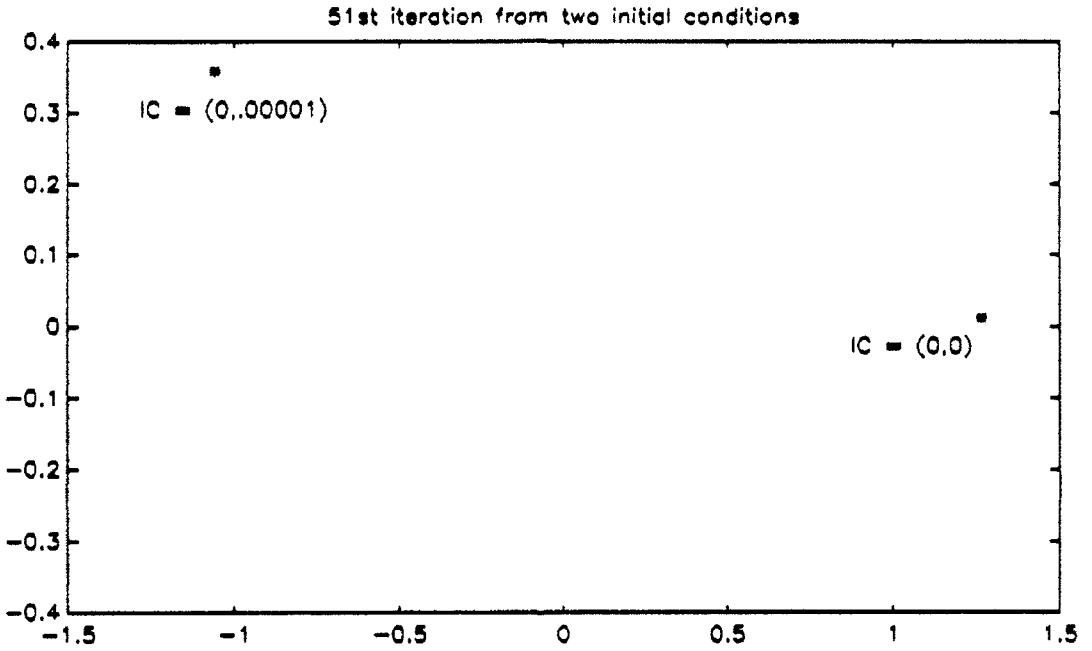


Figure 23: Sensitivity to initial conditions I

Another way of viewing this is to calculate the difference between the abscissas of the two orbits used above. It should be noted that the ordinates behave similarly.

The following graph shows that although the orbits start near each other they quickly separate.

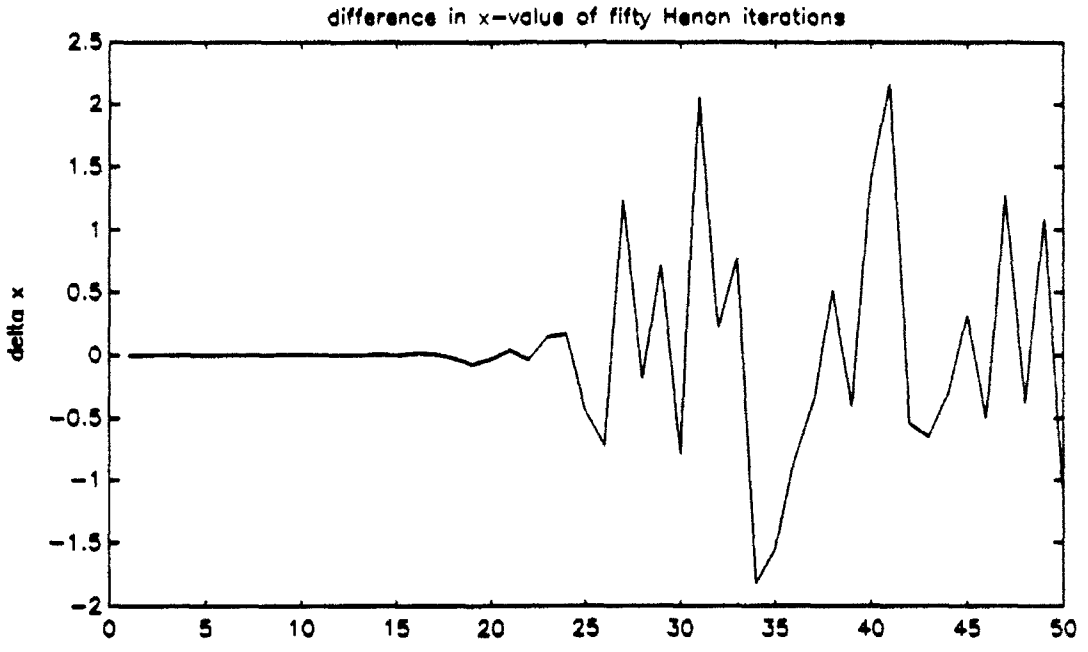


Figure 24: Sensitivity to initial conditions II

A third property is that the Hénon attractor is thought to be strange. This strangeness, if it is present, is not very important because the geometry of the actual attractor is not as significant as the actual dynamics on the attractor. It does result in the attractor having a fractal dimension, which is not essential for our purposes (thus the name of this paper), but, as will be seen, its effect on the period of a computed orbit is the cherry on the whipped cream.

Our approach will be to determine the median value of the abscissas of the orbits on the Hénon attractor and then use that value as a split point for subsequent iterations, assigning a 1 for x-values above that point and a 0 for those below.[11] Note that this is not the same as the geometric median value of the attractor itself but rather relies on the orbits always following some distribution along the attractor.

One of the drawbacks of working with non-linear systems is that they are very hard to analyze. That being the case we continue our graphical approach towards describing and evaluating these generating functions. The statistical properties of the sequences themselves will be measured using the tools developed earlier.

The first task was to determine if the orbits on the attractor have a single median to speak of and if so what its value is. The method used was to generate 100,000-point sequences based on 20,000 initial conditions. In Hénon's original paper he shows that the quadrilateral with corners (-1.33,.42), (1.32, .133), (1.245,,-.14), (-1.06,-.5) constitutes a trapping region (Hénon, 1976). The initial points were chosen using the MATLAB random number function and tested for inclusion in the quadrilateral.[12] Rather

---

[11] See Appendix C for henon.m program listing.

[12] See Appendix C for init.m program listing.

than accepting these initial conditions on face value the following graph indicates their uniform distribution.

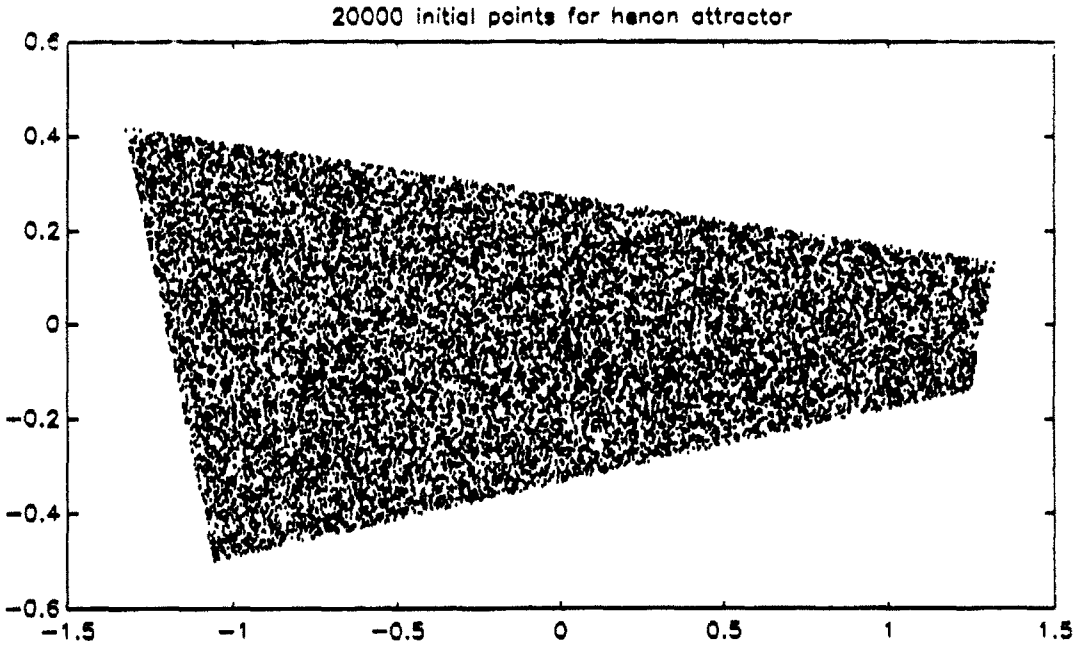


Figure 25: Distribution of initial points

To determine the median of each of these orbits, for the sake of computational time, we had to go outside of MATLAB and feed these initial conditions into a C++ program which iterated the orbit and calculated the median.[13]

[13] See Appendix C for medbatch.cpp program listing.

The following bar chart summarizes the distribution of the 20,000 medians.

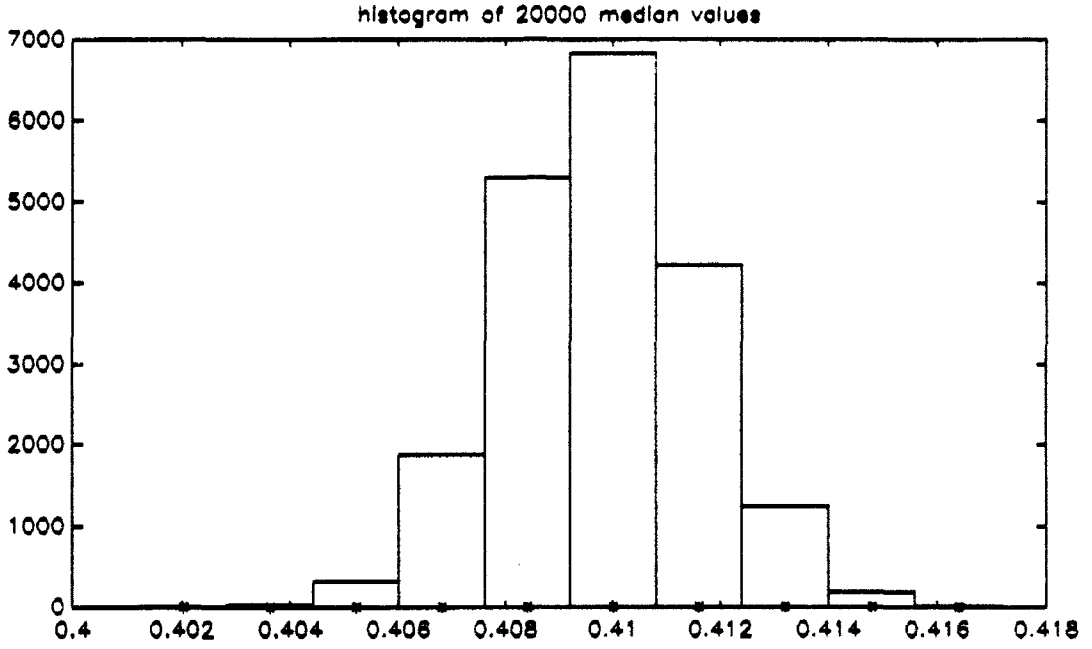


Figure 26: Determination of Hénon median value

To ten digit precision the mean value of the medians is .4097889545 while the standard deviation is .0017937950. Assuming that the median should be good to four decimal places the error analysis with these numbers and a sample size of 20,000 results in z = 3.94. In plain terms, for a given orbit, we are in excess of 99% confident that the calculated median rounded to four decimal places will be .4098. Based on this, from here on, we will state that this is the median value of the distribution on the attractor itself. It should be noted that this value differs from the

one calculated by Forré (1990) who came up with a median of .39912. This can perhaps be attributed to differences in experimental procedure but, beyond that, is not much of an issue since the correlation between two 1,000 point sequences that differ only by these split points is .99.

The second requirement of the generator is that it have a high sensitivity to initial conditions. This is, in some respects, a crossover point between the generator and the binary sequence as what we are truly concerned about is that the latter be sufficiently different given a small change in the initial conditions. The process was to generate 100 sequences of length 10,000. For each of these sequences the initial conditions were perturbed by increasing intervals ($\sqrt{2}x10^{-1}$, $\sqrt{2}x10^{-2}$, ...) and the correlation between the new sequence and the original sequence was calculated at selected lengths.[14] The results are summarized in Table 2.

Table 2: Small perturbation cross correlations

Δ I.C. $\sqrt{2}$ x	100	500	1000	2000	5000	10000
10^{-1}	.0460	.0072	.0075	.0006	.0000	.0010
10^{-2}	.0810	.0135	.0066	.0014	.0006	.0014
10^{-3}	.1720	.0255	.0119	.0101	.0002	.0008

[14] See Appendix C for sensitiv.m program listing.

Table 2 (cont.)

Δ I.C. $\sqrt{2}$ x	100	500	1000	2000	5000	10000
10^{-4}	.1958	.0443	.0298	.0144	.0054	.0027
10^{-5}	.2520	.0528	.0292	.0196	.0067	.0029
10^{-6}	.3022	.0598	.0302	.0179	.0089	.0049
10^{-7}	.3672	.0722	.0359	.0185	.0071	.0045
10^{-8}	.4312	.0863	.0482	.0245	.0118	.0066
10^{-9}	.4612	.0905	.0459	.0206	.0088	.0045
10^{-10}	.5130	.0940	.0508	.0238	.0086	.0044
10^{-11}	.5906	.1181	.0610	.0298	.0115	.0056
10^{-12}	.6388	.1206	.0553	.0300	.0108	.0055
10^{-13}	.6866	.1400	.0692	.0339	.0137	.0061
10^{-14}	.7528	.1408	.0730	.0389	.0115	.0041
10^{-15}	.8128	.1765	.0953	.0477	.0262	.0177
10^{-16}	.8884	.4298	.3706	.3417	.3226	.3166
10^{-17}	.9784	.9004	.8898	.8851	.8814	.8807
10^{-18}	.9966	.9836	.9817	.9809	.9807	.9803

Two rules come out of this chart; the first is that the minimum spacing between allowable initial conditions should be 10^{-15}. The second being that in a coding application we should throw away at least the first two thousand points to assure that the correlation of two sequences with initial conditions differing by the minimum separation remains below .05 which we adopt as a general level of satisfactory correlation. Neither of these pose a practical problem since even in MATLAB that number of points takes less than four seconds to generate on a 486/33 personal computer and the restricted spacing still allows for a keyspace of 1.5×10^{30} different initial pairs inside the trapping quadrilateral.

The final requirement of the generator is that it have sufficiently long period. This too is an area in which the use of a strange attractor comes in handy. By its very nature, if calculated with infinite precision, there will be uncountably many non-periodic orbits and countably many periodic orbits. Not only are the vast majority of orbits non-periodic but the probability of landing on a periodic orbit is actually zero! Which is fine but leaves open the question of what happens when the iterations are done on a finite precision machine. In this case we defined periodicity by a subsequent point coming within a certain tolerance of a particular earlier point. Our methodology was to iterate an initial point 1,000 times, store the next

iteration and then check subsequent iterations against this stored point. This continues until the two points are within a given tolerance of each other. This procedure can be pictured by drawing a circle with a radius of the tolerance around the stored point and going until another point falls within that circle. Before turning the computer loose on this problem it is instructive to try to anticipate the effect of varying the tolerance. As the order of distribution of points on the attractor is haphazard in the plane one might expect that tightening the tolerance by a factor of 10 would decrease the size of the circle by a factor of 100 and thus increase the period by a similar factor. On the other hand, even though the Hénon attractor has been stretched and folded to lie in the plane it is still essentially a linear map in the following sense. The graph can be viewed as a continuous curve and as such, the effect of decreasing the interval by a factor of 10 should increase the period linearly by a similar amount. Actually, the answer is between these two and reflects the fact that the dimension of the horseshoe attractor is neither 1 nor 2 but has been estimated numerically to be 1.28 (Peitgen, Jürgens, Saupe, 1992, p. 670).

Our attempts to numerically test for the period length followed the above procedure with the exception that instead of using a circle (euclidean norm) we used a square

(infinity or max norm) to speed up the computations.[15] Alas, this was not enough to avoid running into the wall of inadequate computer power. Tests were run on a variety of machines but none were fast enough or capable of running non-stop long enough to complete runs with tolerances beyond 10^{-9}. In the tests to that point however it became evident that the period could be approximated by the formula 1/tolerance. Based on this we extrapolate that the period of an orbit, and thus the underlying binary sequence, calculated using our minimum spacing of 10^{-15} will be long enough for any practical use. In terms of previously discussed applications this is approximately the period of a full length linear shift register of length 50 and is sufficient to keep the GPS generation rate requirement of 10Mhz satisfied for a little over three years (the actual GPS period is about 225 days). Having demonstrated that our proposed generator meets the basic requirements we now move onto validating the generator by testing the binary sequence by the now familiar means.

B. BALANCE

As usual we begin our conversation on balance. It should be obvious that since the split point was set equal to the calculated median the balance test should be easy to pass. However, since the actual initial conditions used in

---

[15] See Appendix C for period.for program listing.

the test were different than those to calculate the median this also gives a good double check as to the validity of the claim on the median value. The following histograms detail the distribution of the balance statistic over 1,000 sequences of lengths 100, 500, 1000, 2000, 5000, and 10000, and clearly show that the majority of the sequences, whether long or short, fall within the range of acceptable balances of .47 to .53.[16]

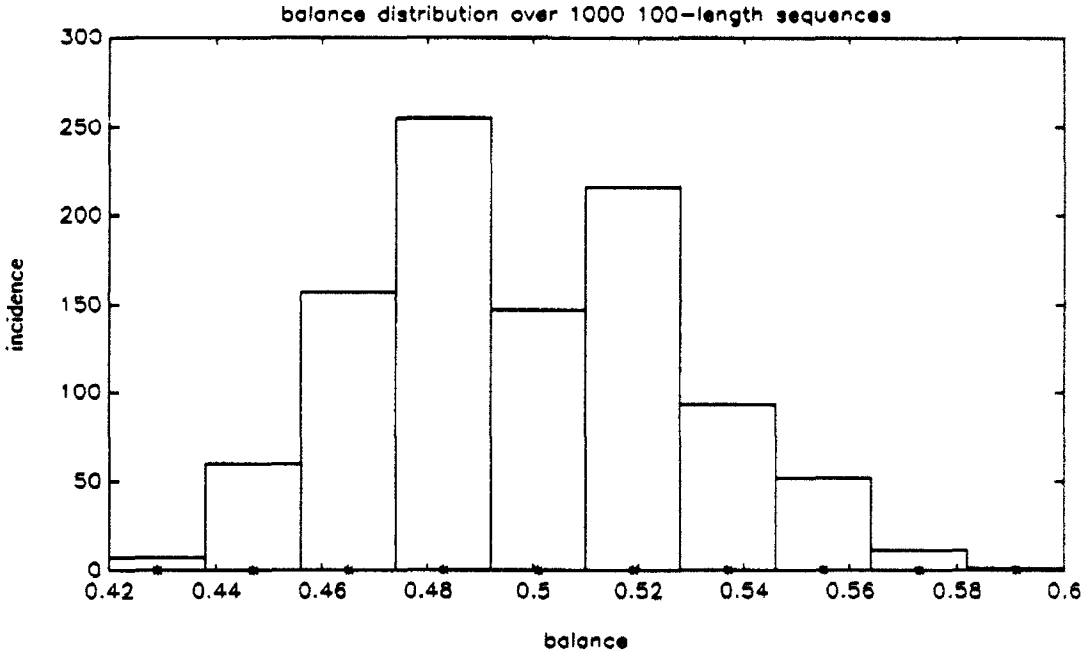


Figure 27: Balance over 100-length sequences

[16] See Appendix C for baltest.m program listing.

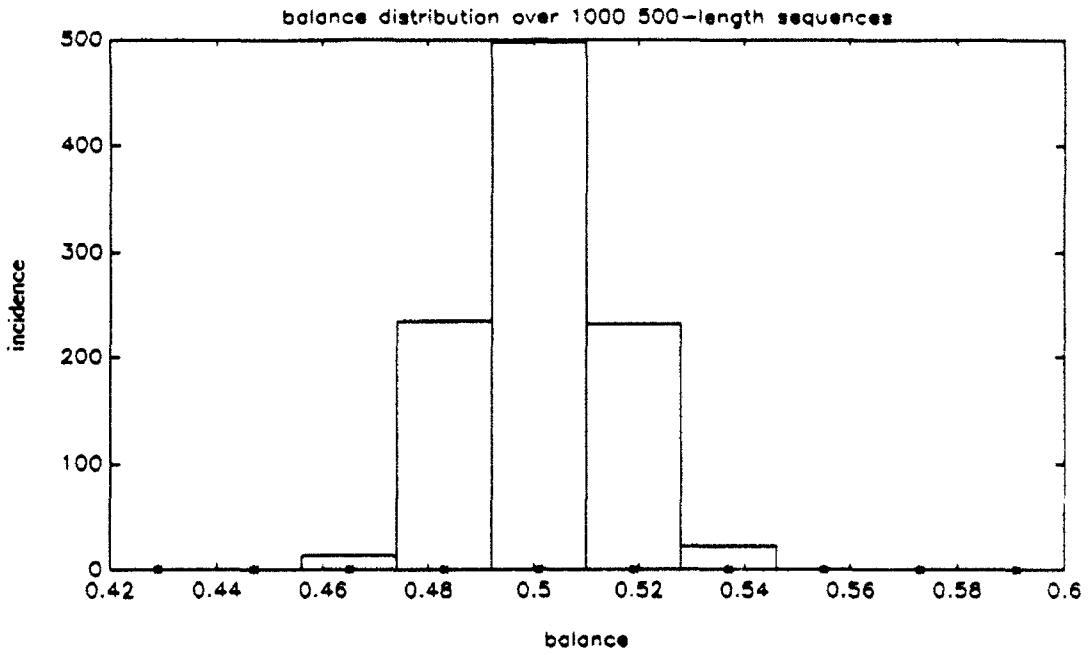


Figure 28: Balance over 500-length sequences

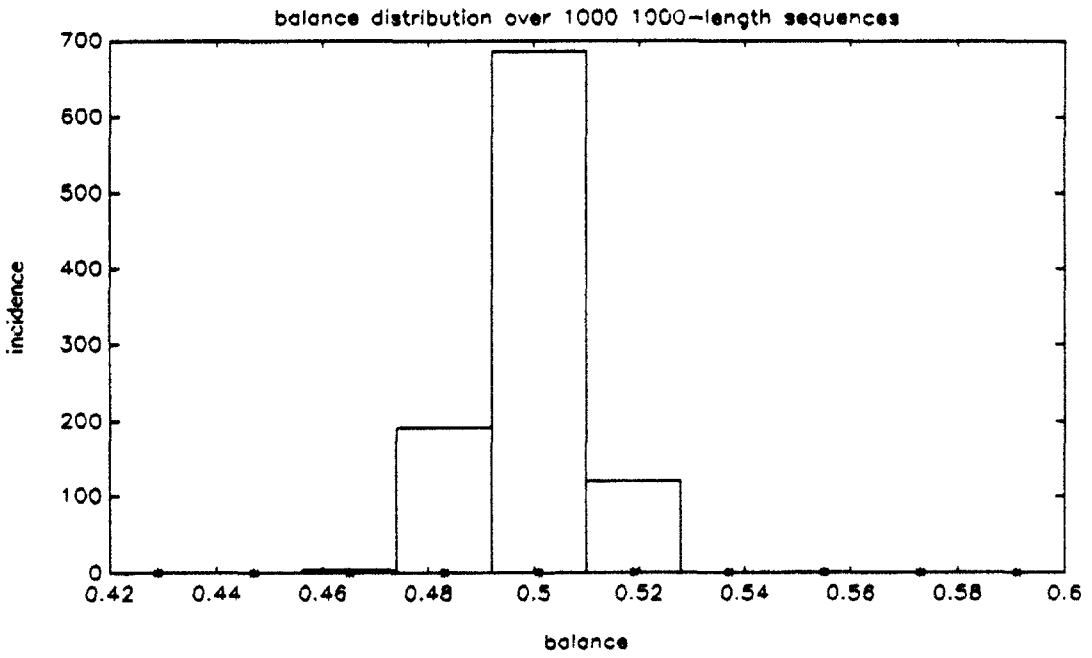


Figure 29: Balance over 1000-length sequences

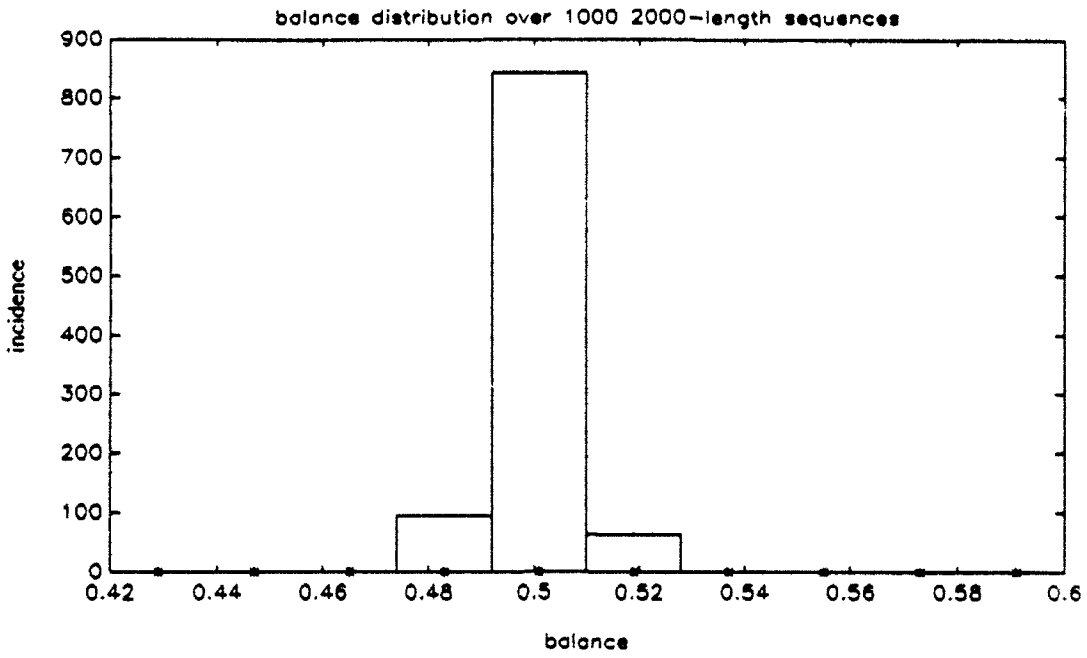


Figure 30: Balance over 2000-length sequences

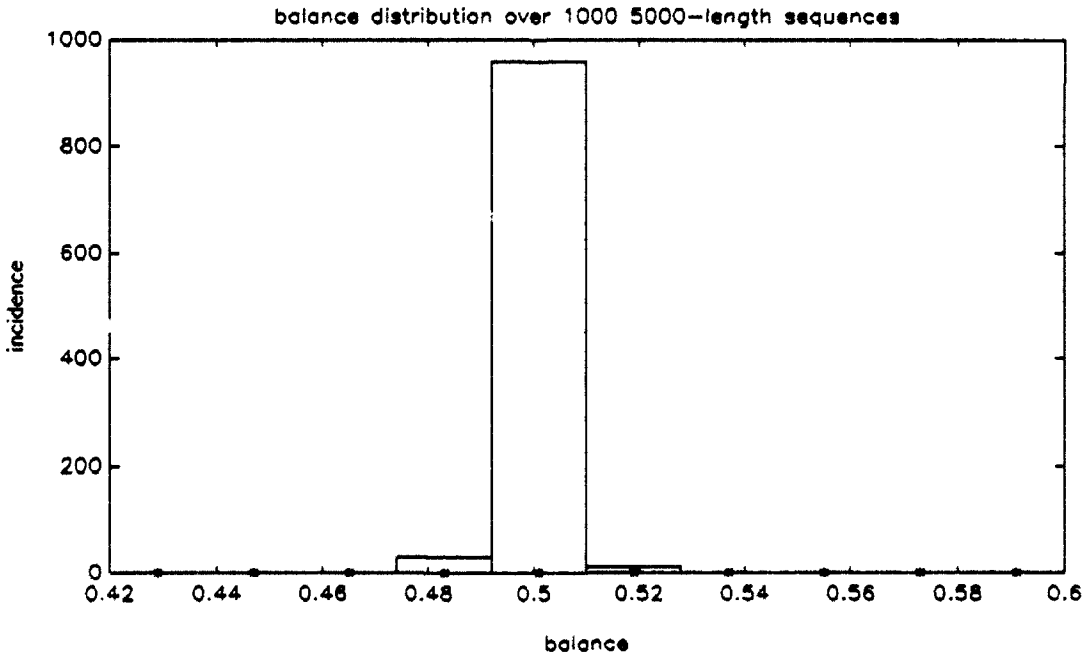


Figure 31: Balance over 5000-length sequences

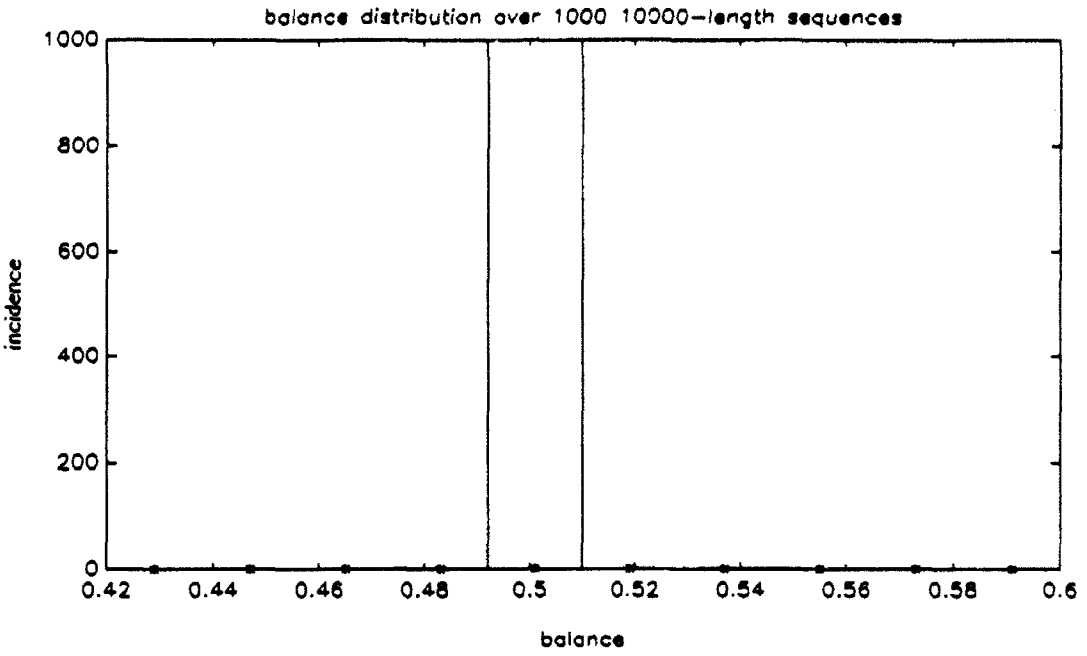


Figure 32: Balance over 10000-length sequences

By way of summary, the following graph combines the minimum, maximum, and mean values of the balance statistic over each of the sub-sequences mentioned earlier.

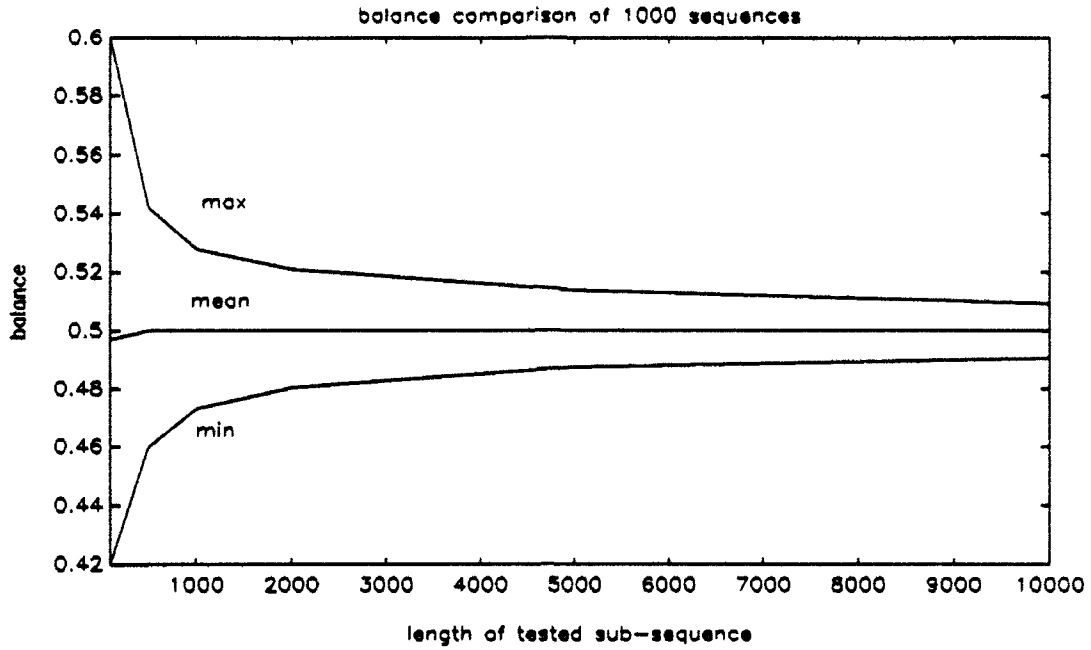


Figure 33: Summary of balance test

Although the above graph indicates that this generation scheme does produce bitstreams with good balance characteristics, the incidence of minimum and maximum balances occurring in the shorter sequences that are outside of the acceptable range raise the question of whether this is a function of being in the early part of a generated sequence or if it is part of a normal distribution of balances that should be expected. To test this anecdotally, we generated a sequence of length 100,000 and then calculated the balance of the 1,000 non-overlapping sub-sequences of length 100. The following graph summarizes the results and shows that approximately two-thirds of the

balances fall inside the acceptable range. This is a little below the expected value of 80% that is indicated by the mean of .4937 and standard deviation of .0238 but as this was a "quick and dirty" check this finding is by no means out of bounds.

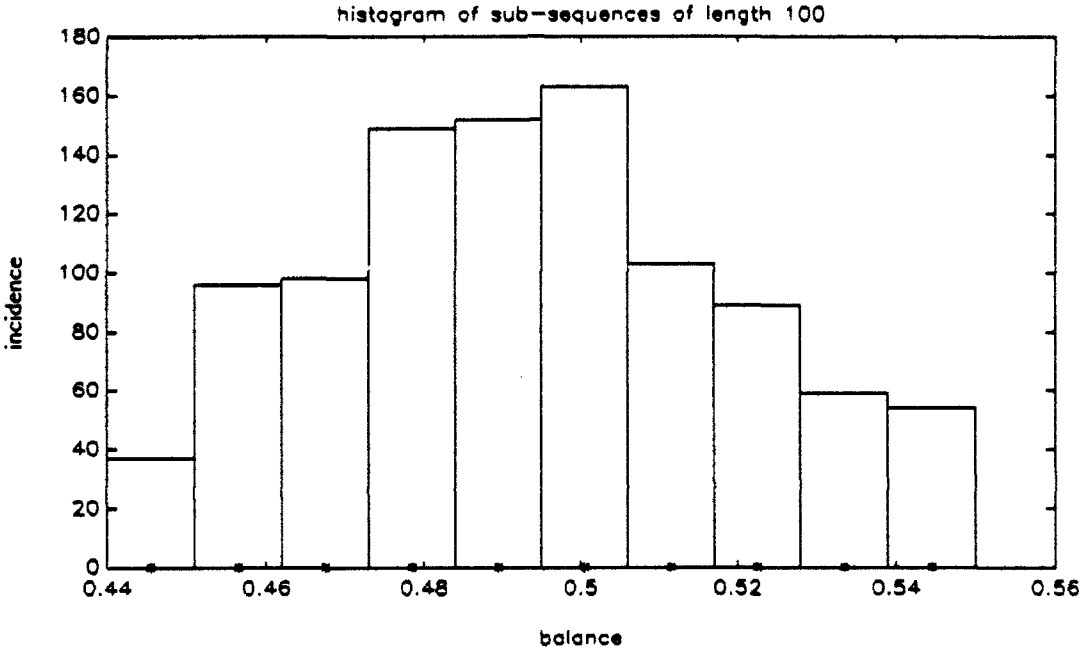


Figure 34: Balance of short sequences

From the above discussion, it is clear that longer sequences have better balance than shorter sequences. This, however, is in line with central tendency and, perhaps more importantly, is consistent with what we consider to be the normal intuitive view of how a flipped coin's balance would vary with the number of flips considered. As such, the overall conclusion is that this generation scheme passes the balance test.

C. AUTOCORRELATION

As mentioned at the outset, our procedure to determine whether a generator passes a particular test is to compare the results from the generator with the results from the coin flip and the stacked sine wave example. The basic idea is that the graph of a successful test will bear a striking resemblance to the former and will not look at all like the latter.

This first graph displays a typical (based on our experience) autocorrelation profile of a 1,000 point bitstream generated by the Hénon scheme.

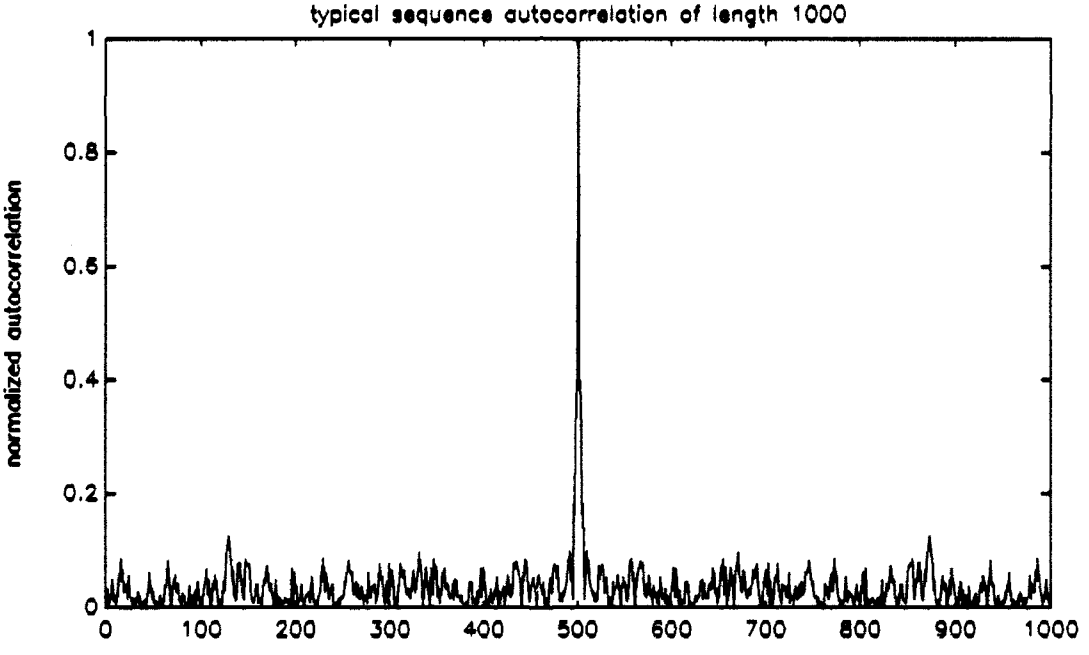


Figure 35: Typical Hénon autocorrelation

It should be readily apparent that this looks quite similar to the autocorrelation graph of the coin flip[17] and not at all like the respective stacked sine wave picture.[18] Including more graphs of single sequences will add little since they really do all look alike. An inability to prove this is, alas, one of the drawbacks of this graphic statistical approach but be that as it may the following graph shows the numerical average of five autocorrelation curves which are based on five sequences with initial conditions determined by the roll of a ten sided die. By rights, the concept of an averaged autocorrelation isn't overly meaningful but in this case, since the averaged graph looks like the earlier individual graph, it helps to highlight that Figure 35 is truly typical.

---

[17] See Figure 2.

[18] See Figure 12.

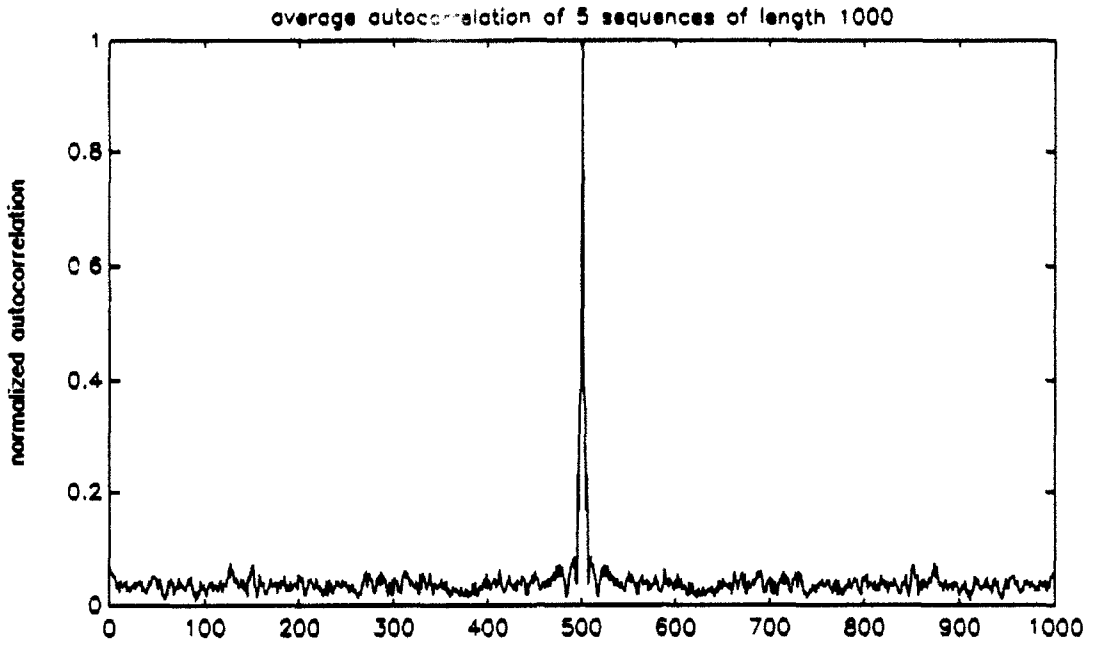


Figure 36: Average Hénon autocorrelation

Not much should be made of the fact that the sidelobes have shrunk a bit below the .1 level as this is indicative of some statistical skewing due to averaging over the small sample size.

For completeness, the final graph in this section shows that the good autocorrelation properties continue to hold for relatively long sequences.

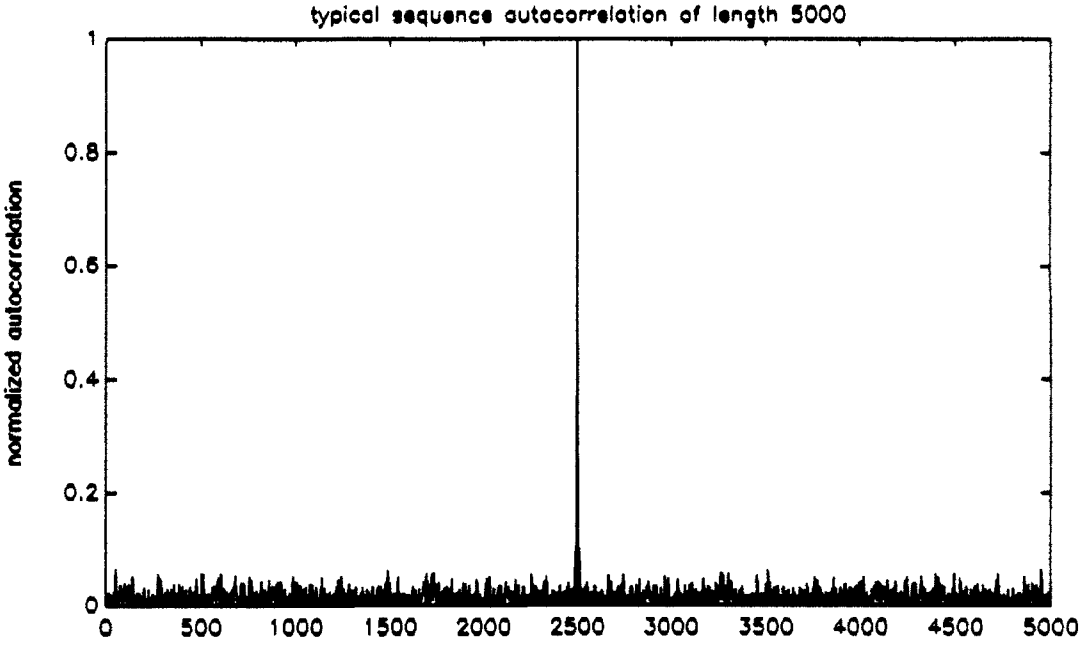


Figure 37: Typical long Hénon autocorrelation

D. RUNS

The following series of graphs represent the results of the runs test when applied to a typical binary sequence generated by the Hénon scheme. It is noteworthy that these profiles, particularly those for the longer n-tuples, are qualitatively invariant over all the sequences tested.

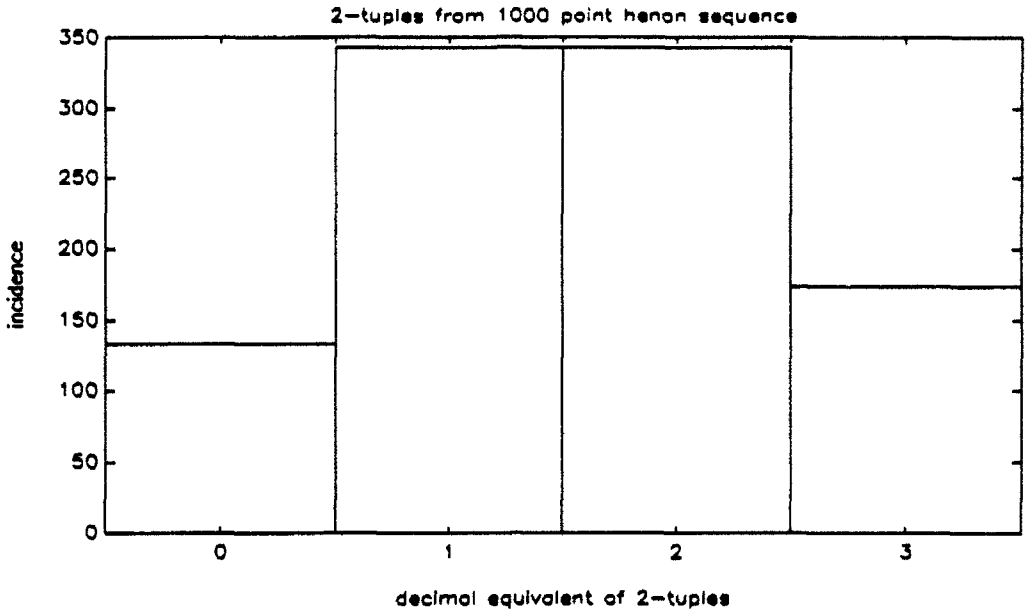


Figure 38: Hénon sequence runs of length 2

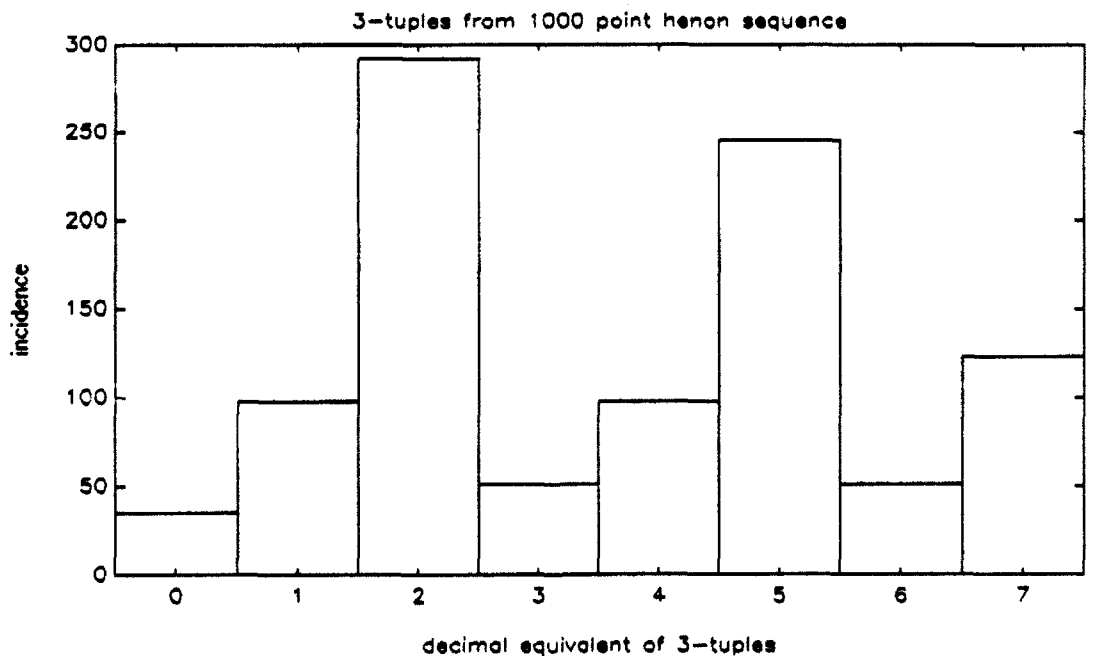


Figure 39: Hénon sequence runs of length 3

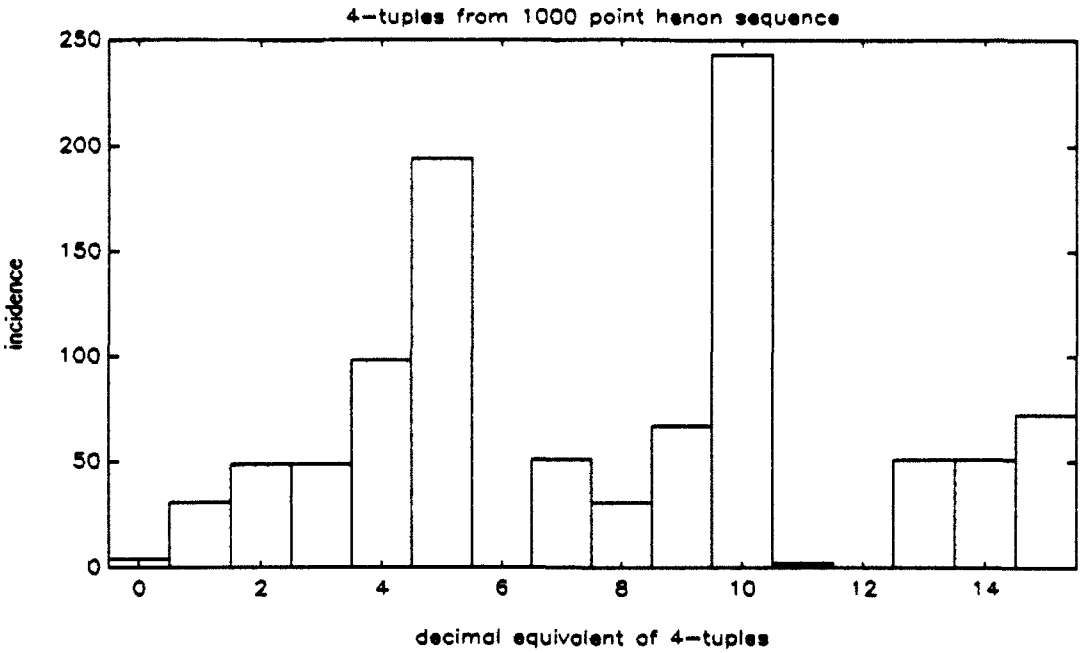


Figure 40: Hénon sequence runs of length 4

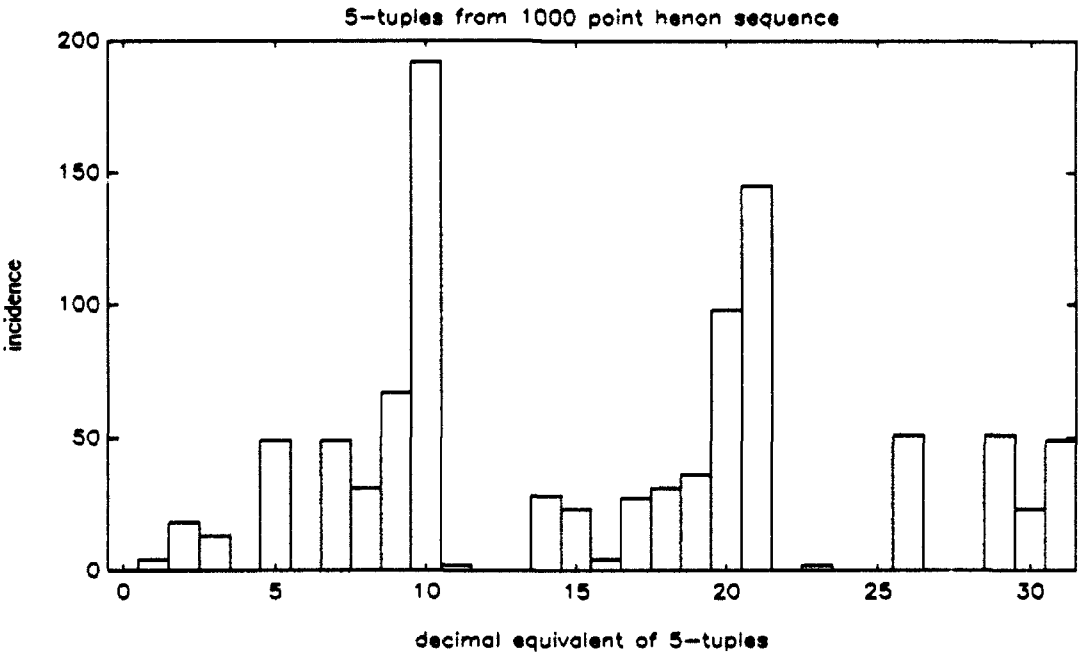


Figure 41: Hénon sequence runs of length 5

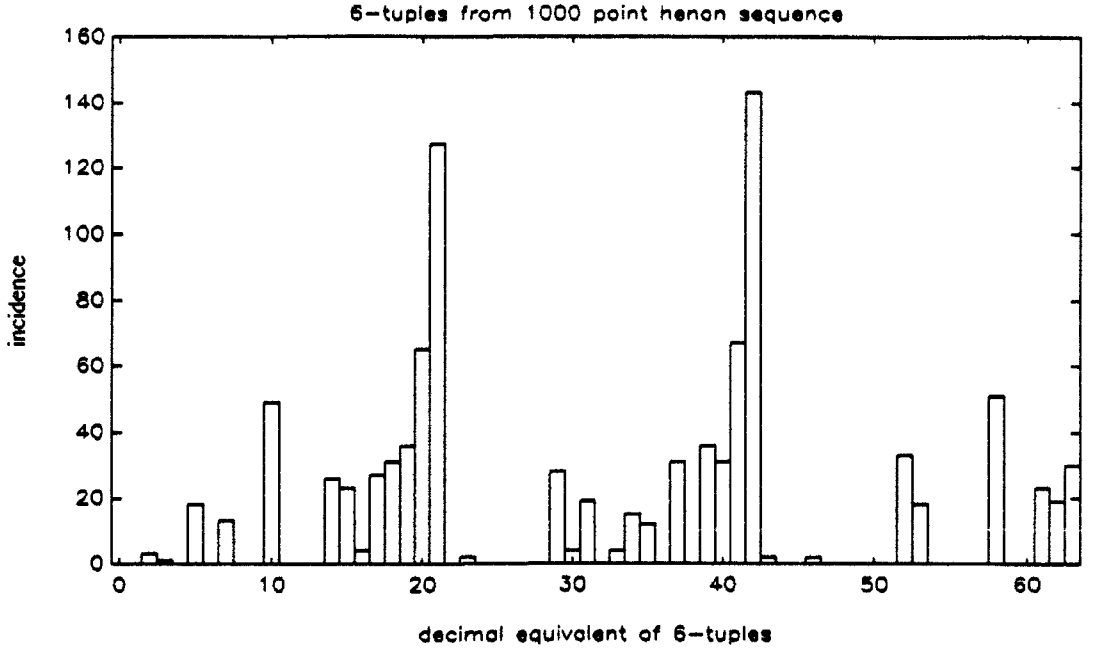


Figure 42: Hénon sequence runs of length 6

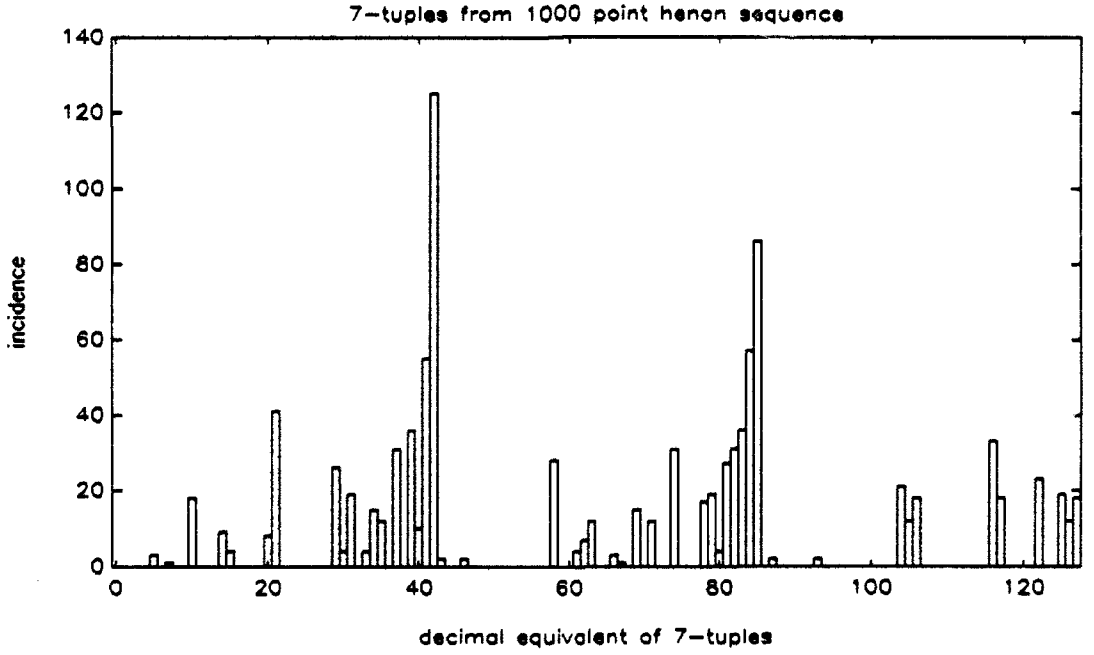


Figure 43: Hénon sequence runs of length 7

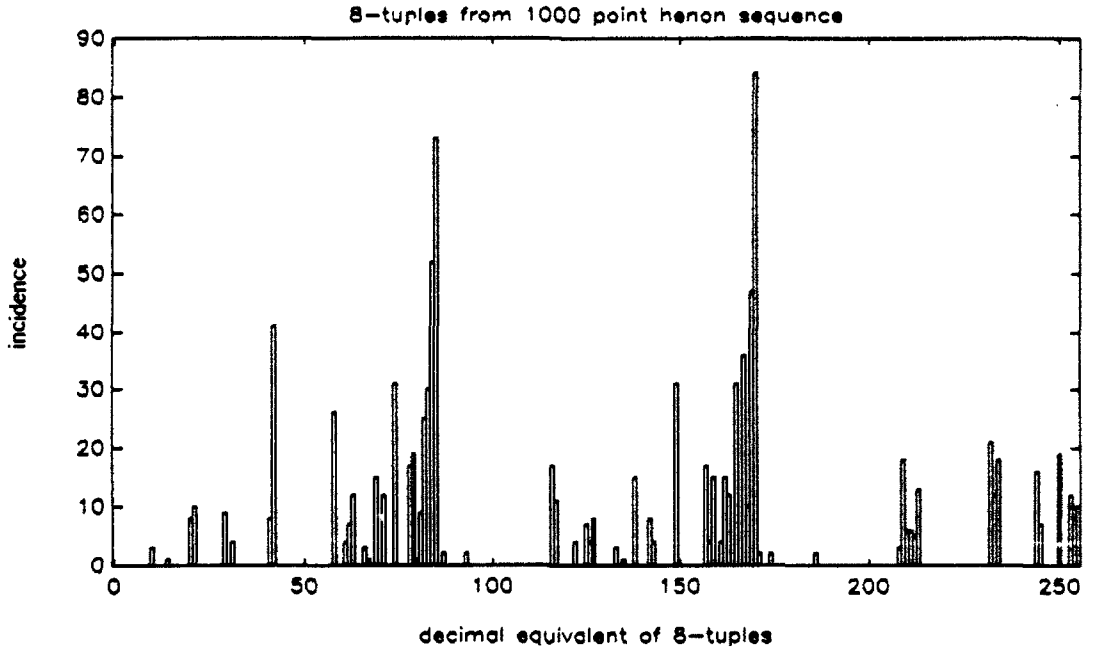


Figure 44: Hénon sequence runs of length 8

Although at first glance these results do not look too promising we will proceed with our usual analysis which concentrates on exploiting an apparent bias to make headway on guessing the sequence. In this case, as opposed to the coin flip, the bias is clear even in the first three graphs where the frequent decimal occurrences correspond to binary sequences with alternating 0's and 1's. The following table summarizes.

Table 3: Most frequent n-tuples in decreasing order

5-tuples	6-tuples	7-tuples	8-tuples
01010	101010	0101010	10101010
10101	010101	1010101	01010101
10100	101001	1010100	01010100
01001	010100	0101001	10101001
11010	111010	0010101	00101010

It is somewhat apparent that guessing the sequence 0101...01 should be a good start since the orbit, along the x-axis, tends to tick-tock except for when it doesn't in which case it tends to tock-tick. This is highlighted by the fact that the first two rows are precisely those n-tuples identified with such a tick-tock motion and comprise 34, 27, 21, and 16 percent of the respective total number of possible n-tuples. The following graph shows a sample of the orbit's trace which highlights this back and forth effect.

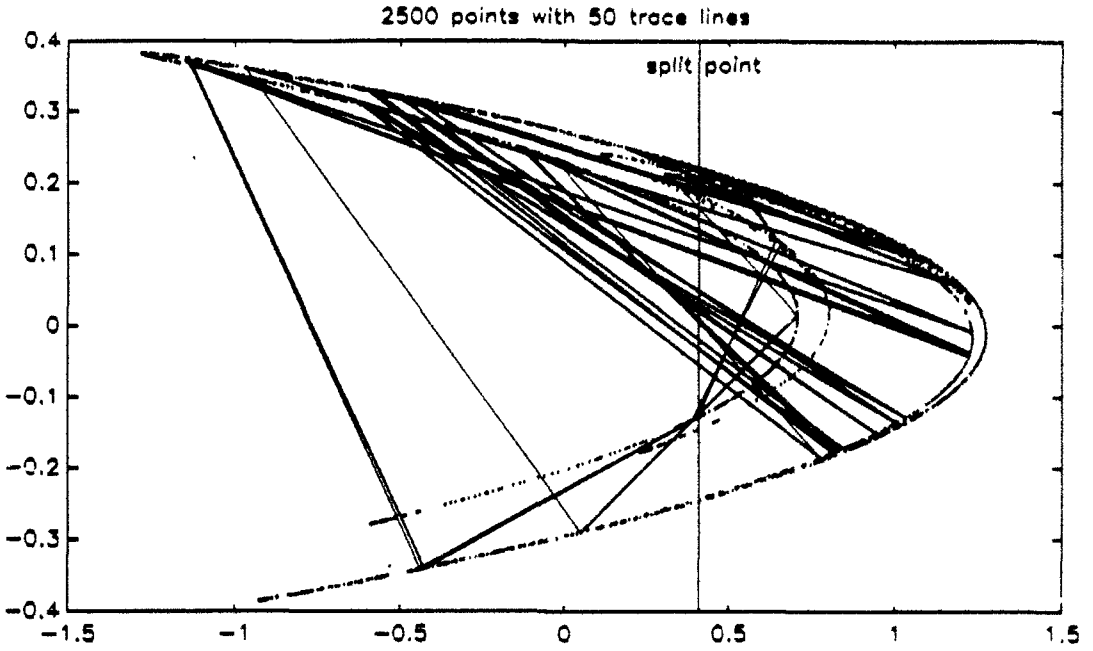


Figure 45: Tick-tock behavior of Hénon sequence

This underlying behavior, combined with the actual runs listed in Table 3, does not portend great things. However, as will be seen, the actual impact is far less than what might reasonably be expected at this point.

In order to make a valid statement about the generator it doesn't make sense to measure this guess against a single, albeit typical, sequence. In lieu of that approach we generated 1,000 sequences and checked the correlation of each against the 01... sequence.[19] This was done for sequences of length 8192, 1024, and 64. The resulting

[19] See Appendix C for is01bad.m program listing.

correlations are summarized in the following graphs. (Note the changing scales for comparisons between cases.)

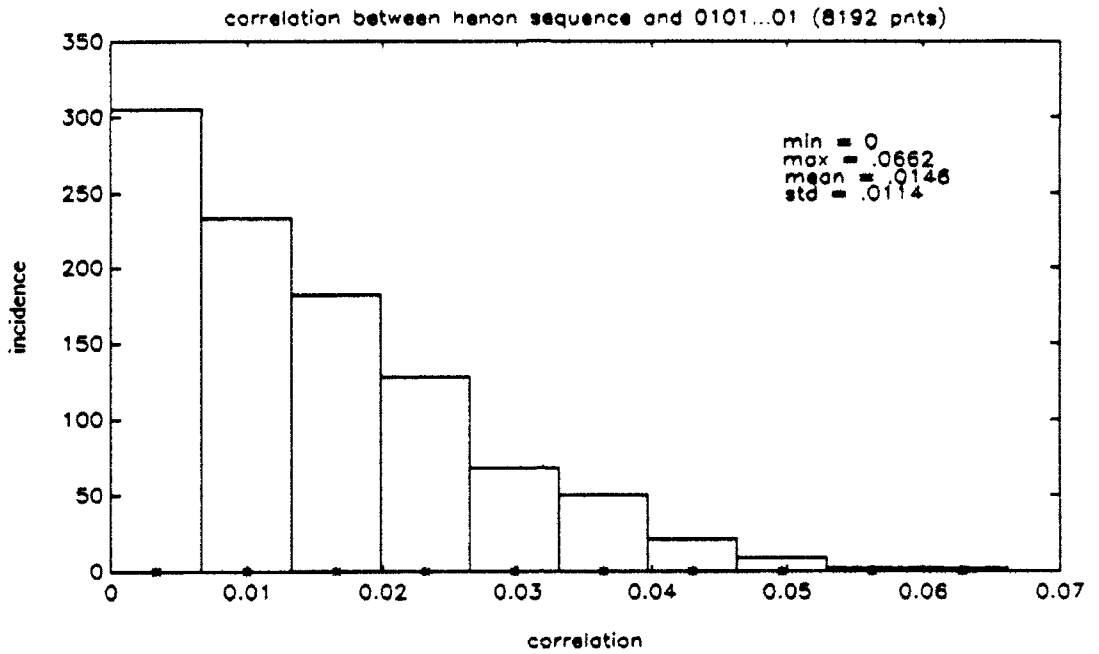


Figure 46: Hénon vs. 01...01 (8192 points)

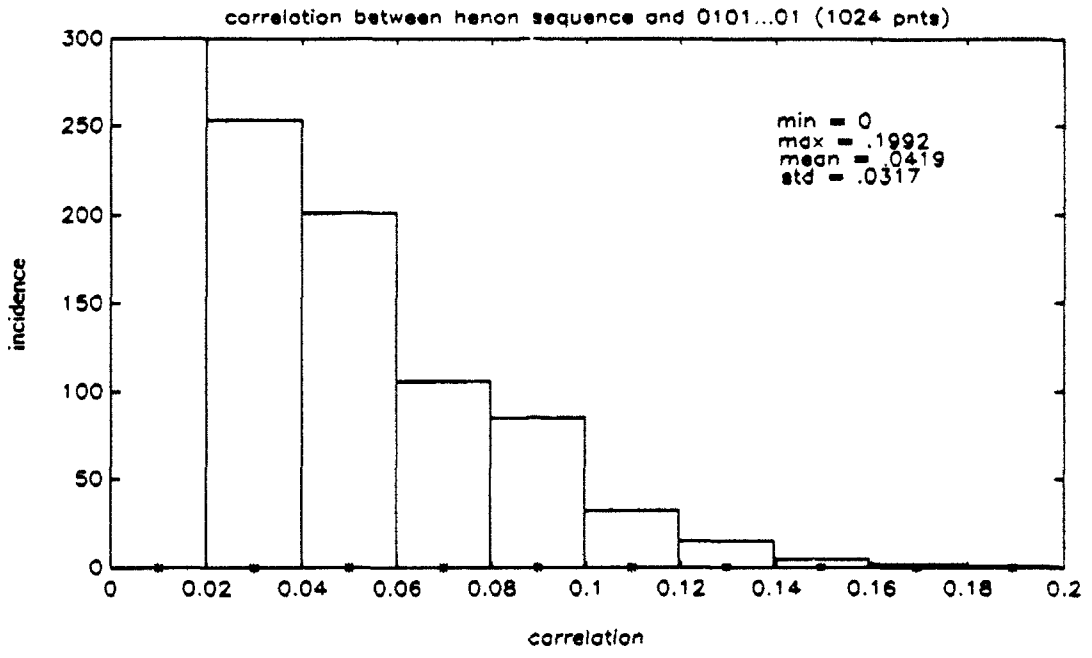


Figure 47: Hénon vs. 01...01 (1024 points)

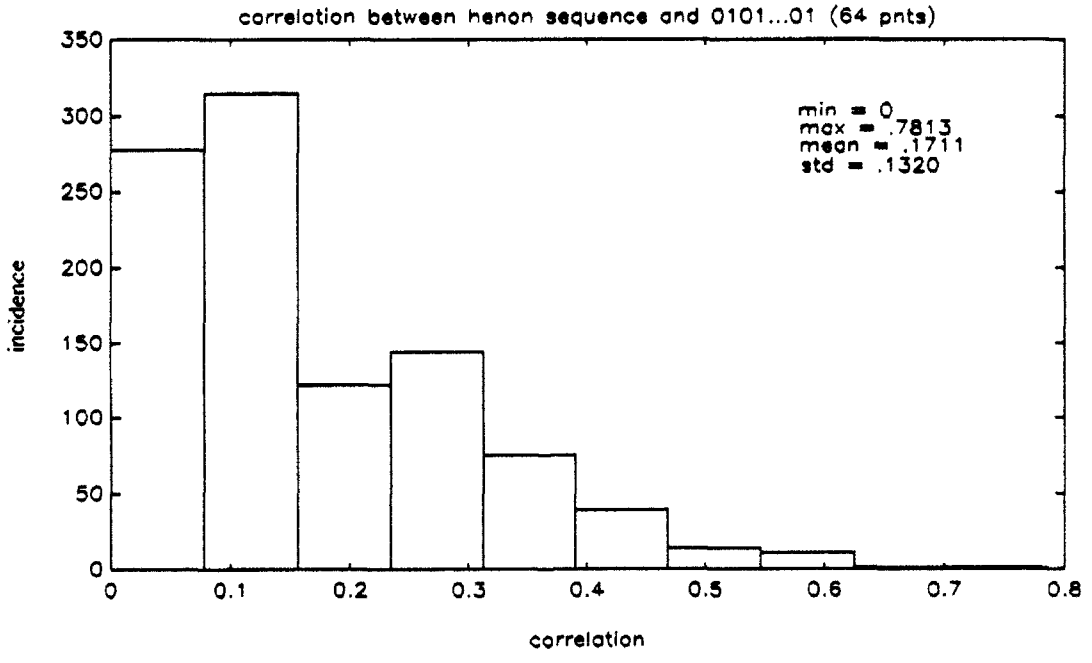


Figure 48: Hénon vs. 01...01 (64 points)

While it is clear that the alternating sequence doesn't make much progress on the longer sequences the mean value of .17 in the 64 point case is a little worrisome. By way of comparison we ran a similar experiment by guessing binary vectors produced using the MATLAB "rand" function which generates numbers using the standard linear congruential method.[20] The same length sequences were tested and the results are as follows.

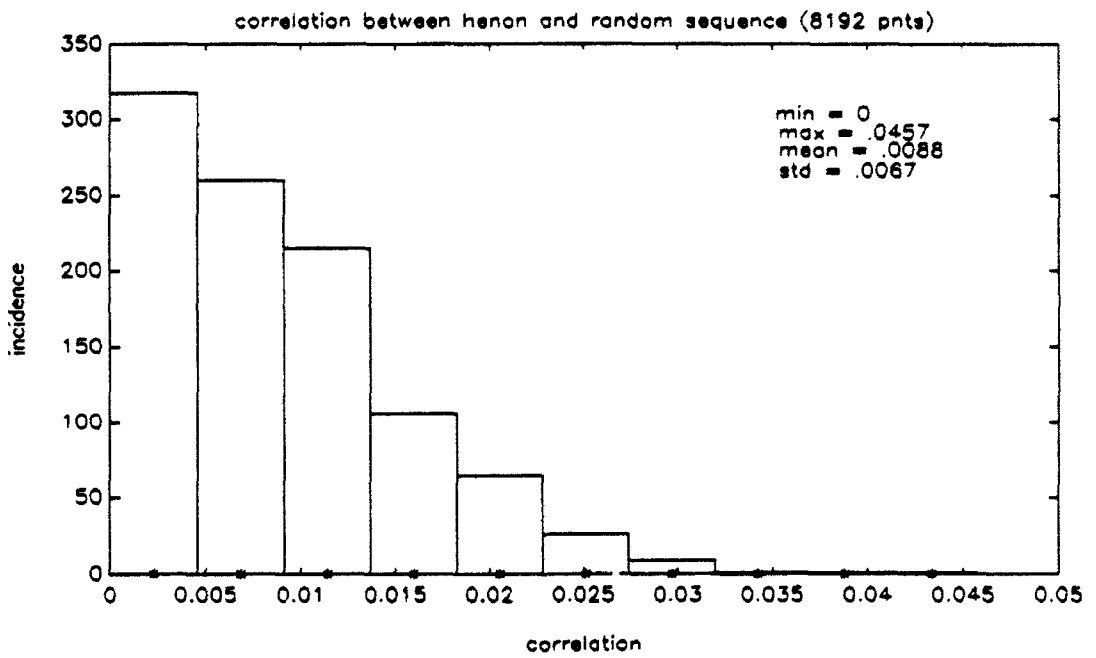


Figure 49: Hénon vs. random sequence (8192 points)

[20] See Appendix C for guess.m program listing.

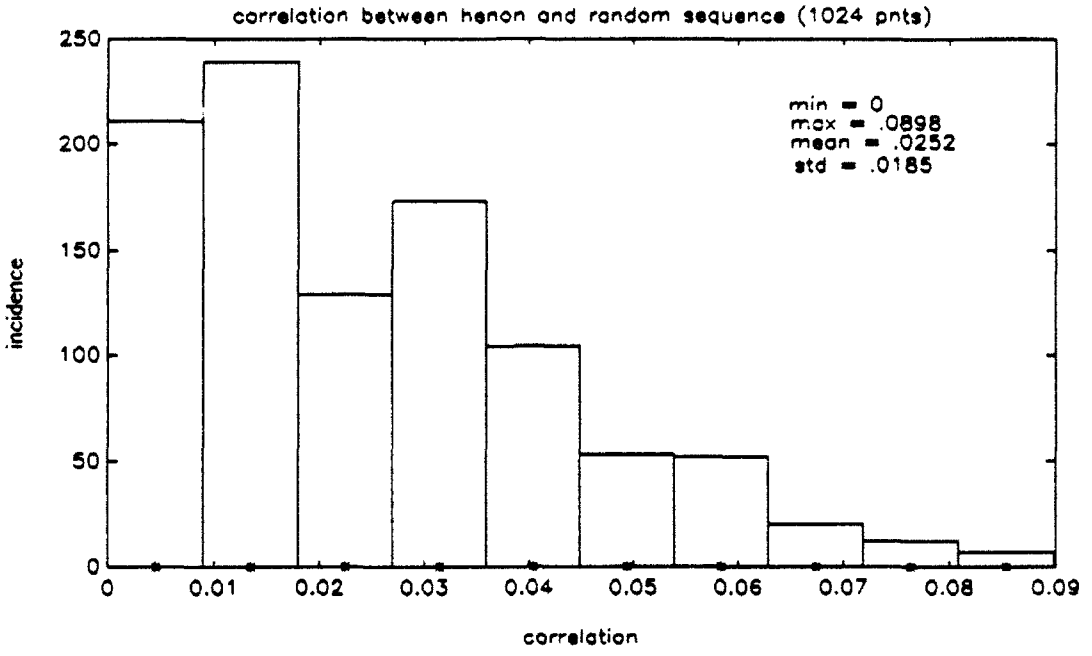


Figure 50: Hénon vs. random sequence (1024 points)

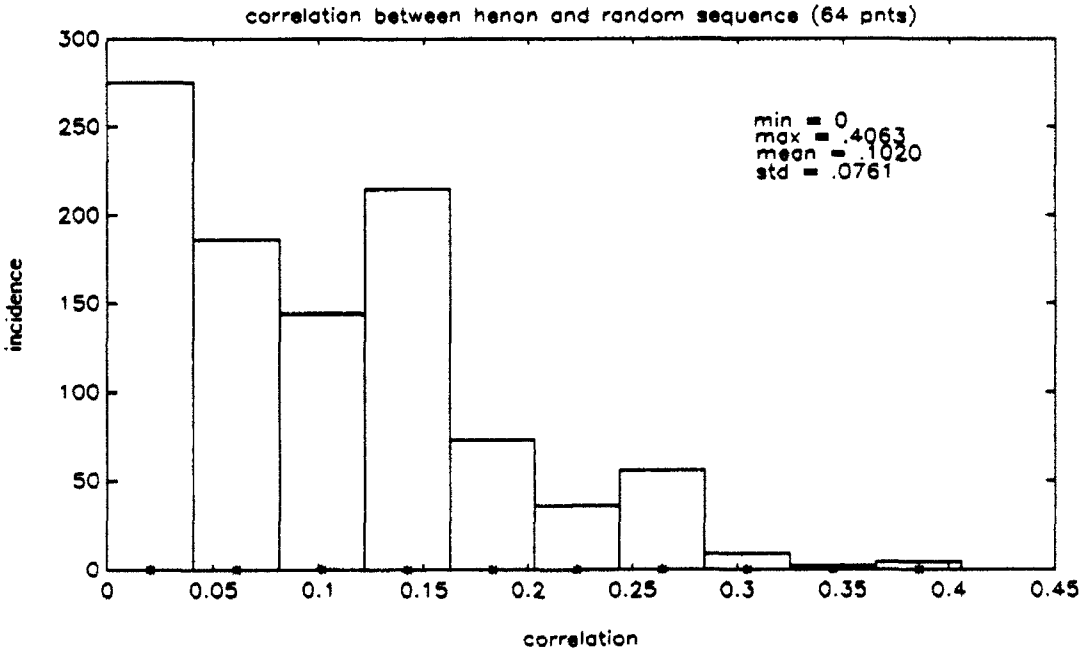


Figure 51: Hénon vs. random sequence (64 points)

The upshot of all this is that guessing the 0101...01 sequence is about 66% more effective than guessing at random. (Stamp has suggested a more sophisticated follow-on attack.[21]) When viewed in context of the applications discussed in the beginning of this paper this becomes a little less of a problem since a typical use, such as the bank ATM example given in Chapter 1, of a short sequence would result in a binary go/no-go decision which means that if the iterated sequence doesn't match the stored sequence perfectly then it doesn't matter much how far off it is. This is as opposed to the typical coding use of a longer sequence in which incremental knowledge is useful but which would appear not to be forthcoming from this guessing strategy as indicated in Figure 47.

Combining the above considerations we conclude that, in general, the runs test comes up neutral but that it's use for any specific application would have to be contingent on a closer look at this criterion.

E. LINEAR COMPLEXITY

In light of the mixed results of the runs property, the linear complexity takes on a special importance. This is due to the fact that if the generator really does produce a preponderance of 010101... sub-sequences then this could show up in a linear complexity that does not follow the optimal n/2 line. As can be seen from the following graph

---

[21] Private communication December 1992.

of a typical linear complexity profile of a 1,000 point sequence, this eventuality does not manifest itself.

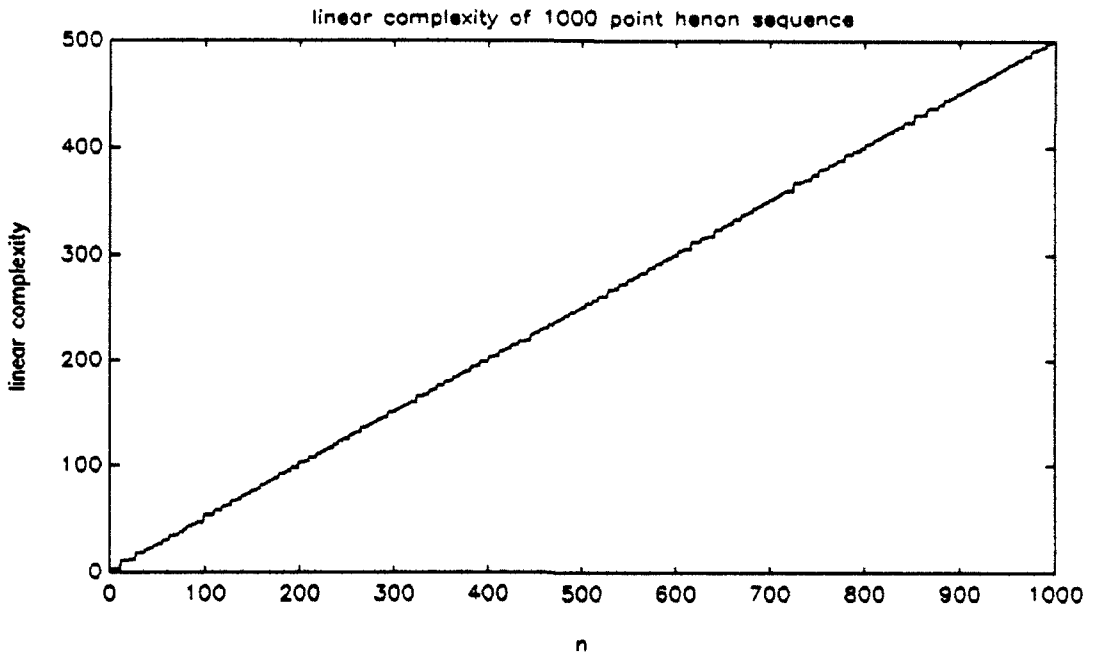


Figure 52: Hénon linear complexity

That the Hénon generation scheme passes this test with flying colors is hardly surprising. After all, the use of the Berlekamp-Massey algorithm in this case essentially represents a linear attack on a non-linear system. The question now becomes one of determining if there is a non-linear shift register that could duplicate the generator. This must remain an open question as, currently, there is no open source algorithm for determining quadratic or higher complexities.

V. CONCLUSIONS

First and foremost we believe that the proposed generating scheme based on the Hénon attractor is a sound and effective method for generating pseudorandom bitstreams. We base this conclusion on the fact that the algorithm displays a high degree of sensitivity to initial conditions, is simple so as to facilitate high bit rates, and produces bitstreams that fulfill the principal requirements for pseudorandomness. In this respect, we differ with Forré's (1990) conclusion that the resultant sequences display too much regularity in terms of the runs property to be of practical use. Our disagreement stems from, what we believe, is a common but excessive importance ascribed to the Golomb postulates for pseudorandomness per sé rather than an interpretation of what passing or failing a specific test actually means. It is for this reason that we have emphasized what information is gleaned about a sequence that "fails" a particular test rather than accepting the test results at face value as the final arbiter.

This is a critical difference because any deterministic system is going to have some type of regularity in its output and having a test that only says as much is not really adding anything to the discussion. The standard suite of tests were originally based on - and thus carry an implicit benchmark of - linear feedback shift registers. Reuppel recognized this shortcoming and advocated the use of

linear complexity as a measure of a sequence's unpredictability. Which is fine except there are sequences that fail the standard tests, have high linear complexity but are still cryptographically weak. The sequence 000...0001 is an example of such a beast. Fortunately, it is possible to identify such sequences by using Rueppel's linear complexity profile or the k-complexity proposed by Stamp (1992). So on one hand the bad news is that this sequence fails the standard tests. On the other hand, the good news is that it has high linear complexity. But on the third hand (the third hand?) it has bad k-complexity. Where does this end? It doesn't. ***It doesn't end because none of these tests have an inherent primacy.*** As such, the strongest statement that can be made is, as in our case, that the generator performs better than a shift register in some ways (e.g. long period) but not as well in others (e.g. runs) and consideration must be given to the practical advantages and disadvantages of one scheme over the other. In general though, we object to this blanket linking of a sequence's unpredictability and it's relation to a feedback shift register. As we have demonstrated by our construction, even though the sequence shares some characteristics of a FLSR generated sequence it was not produced in that manner. By extension, without denigrating the use of linear complexity analysis as a starting point, when analyzing a bitstream there is no reason to assume that

a shift register was used in the first place. We willingly stipulate that there is nothing special about the use of the Hénon map in that the resultant pseudorandomness of the bitstream is simply a byproduct of the chaoticity of the original system. As such, we suggest that for an arbitrary chaotic discrete dynamical system there exists a mapping into the binary domain such that the resulting sequence is pseudorandom. We leave as open questions both the analysis of the bitstream via symbolic dynamics and the determination of the exact relation between a system's chaoticity and a generated bitstream's pseudorandomness.

The ultimate measure of a sequence's pseudorandomness is whether having knowledge of some of the sequence allows the accurate prediction of bits yet to come in the absence of knowledge concerning the specific generator being used. This can also be viewed as how much effort does it take to describe the entire sequence. For example, the sequence 2, 4, 6, 8... is quite long but it can be fully described as "the even integers". By contrast, a truly random sequence can only be described by listing all the elements. Good pseudorandomness combines these two qualities by dictating that the sequence be based on simple rules but that it appear to the uninitiated to require a description approaching a complete list. Having covered the principal linear approaches in this paper, the next open question is to determine if there is a systematic way to determine if a

more complicated, perhaps non-linear, regularity exists beyond the usual scope of balance, autocorrelation, runs, and linear complexity. In as much as these systems cannot currently be analyzed directly and algorithms for complexities for orders higher than linear are not known we suggest that an alternate approach would be to utilize some form of non-algorithmic methodology such as neural nets. This is not to suggest that a single non-linear attack would be effective against all non-linear systems but based on preliminary unpublished work with neural nets we believe that there are methods to exploit the regularity present in non-linear deterministic systems.

APPENDIX A

This appendix is a brief review/lesson in $\mathbb{Z}_2$ and its associated mod2 arithmetic. Note that it is not meant to cover all the intricacies of Galois and extension fields but simply to lay down some basic rules that will give the reader some $\mathbb{Z}_2$ survival skills. For a complete treatment of this subject see Bloch's textbook on abstract algebra (1991).

Rule #1: $\mathbb{Z}_2$ consists only of the numbers 0 and 1.

The first step in dealing with mod2 math is to understand that mapping any integer to $\mathbb{Z}_2$ comes down to dividing the number by 2 and taking the remainder to be the answer. It seems obvious that the result must fall in the set {0,1} (which we denote as the "binary domain") but if not, a few moments of pondering should do the trick. For example:

$$7 = 3*2 + 1 \qquad\qquad 8 = 4*2 + 0$$

$$\therefore 7 = 1 \text{ mod2} \qquad\qquad \therefore 8 = 0 \text{ mod } 2$$

Which leads to the next two rules:

Rule #2: Any odd integer is equal to 1 mod2.

Rule #3: Any even integer is equal to 0 mod2.

Now that we are firmly in $\mathbb{Z}_2$ the next question is how to do arithmetic. The following are the addition, multiplication, and subtraction tables for $\mathbb{Z}_2$.

+	0	1
0	0	1
1	1	0

*	0	1
0	0	0
1	0	1

-	0	1
0	0	1
1	1	0

Rule #4: Anything added to itself is 0.

Rule #5: Subtraction is the same as addition.

In a nutshell, that is all there is to the world of $\mathbb{Z}_2$.

As a bridge to shift registers it is also necessary to understand how polynomials work in $\mathbb{Z}_2$. By and large, it is the same as with real numbers except that the coefficients can only be 0 or 1. This leads to some interesting results. For example, over the real numbers, the equation $x^2 + 1 = 0$ doesn't have a solution. But, in $\mathbb{Z}_2$, x=1 is a solution as follows.

$$x^2 + 1 = 0$$

for x=1 $\quad 1^2 + 1 = 0$

$$1 + 1 = 0$$

By Rule #4 $\quad 0 = 0$

Another way of looking at this is that if "limited" to the real numbers the above function is prime but when we move to $\mathbb{Z}_2$ it is factorable as

$$x^2 + 1 = (x+1)*(x+1)$$

since

$$\begin{aligned}(x+1)*(x+1) &= x^2 + 2x + 1 \\ &= x^2 + 0x + 1 \quad \text{by Rule \#3} \\ &= x^2 + 1\end{aligned}$$

Combining the above ideas there is a particular convenience unique to working with $\mathbb{Z}_2$ polynomials as demonstrated below.

$$\begin{aligned}x^3 + x + 1 &= 0 \\ x^3 &= -x - 1 \\ x^3 &= x + 1 \quad \text{by Rule \#5}\end{aligned}$$

Rule #6: In dealing with equations, feel free to put the "=" sign anywhere that is convenient.

APPENDIX B

This appendix serves as a short lesson in linear feedback shift registers, how they work, and the terminology involved with them. For a complete treatment the reader is directed to Golomb (1982) which is considered to be the seminal work on the subject.

A linear feedback shift register is a simple digital circuit which, on command, goes from one state to another. Their use in sequence generation is probably best shown by example.

Consider the $\mathbb{Z}_2$ function $f(x) = x^3 + x + 1$ and set it equal to 0,[22]

$$x^3 + x + 1 = 0$$

This can be written

$$x^3 = x + 1$$

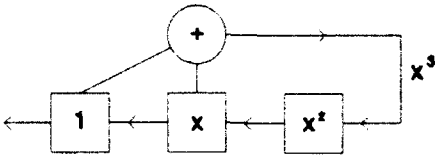
This equation can be used to describe the design and feedback of the shift register shown in Figure B1. Note that there is a box, actually called a "stage", for every power of x less than the polynomial degree. The real use of the polynomial designation is that it fully describes how the stages labeled "1" and "x" are added together and fed back along the line marked x^3 into the "x^2" stage.

---

[22] See Appendix A for a quick lesson in mod2 math.

In words, at every turn of the crank, three things occur:

- the contents of the two rightmost stages are shifted left
- the contents of the two leftmost stages are added together (mod2) and put into the rightmost stage
- the contents of the leftmost stage is output as the next bit in the sequence



initial fill	1	0	0	
	0	0	1	
	0	1	0	
	1	0	1	
	0	1	1	
	1	1	1	
	1	1	0	period 7
	1	0	0	

Figure B1

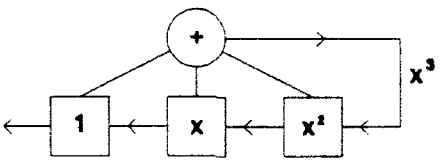
This process results in a bitstream of 10010111. In addition, however, there are some other tidbits to consider. The first of these is that all seven possible 3-tuples show up and then start to repeat. The second is that the period (7) is equal to 2^n-1 where n is the number of stages in the shift register and is also the degree of the generating polynomial. Finally, f(x) is prime. In general, factoring

a polynomial in $\mathbb{Z}_2$ is as big a headache as factoring a "regular" polynomial but in this case primeness is easy to determine if we consider the following. If f(x) is factorable then, being of degree 3, it must factor either into a monic polynomial and a quadratic or into three monic polynomials. In any event, there must be a monic polynomial. But as there are only two monic polynomials, namely (x) and (x+1), then either 0 or 1 is a root of f(x) and this can be tested by seeing if f(x) = 0 in either case:

$$f(0) = 0^3 + 0 + 1 \qquad f(1) = 1^3 + 1 + 1$$
$$= 0 + 0 + 1 \qquad = 1 + 1 + 1$$
$$= 1 \qquad = 1$$

Hence f(x) is prime. This is not a coincidence and, in fact, a subset of prime polynomials called primitive polynomials gives rise to Full Length Shift Registers (FLSR). These, in turn, generate sequences with maximal period 2^n-1 (assuming a non-zero initial fill) which are known as m-sequences (Koblitz, 1987, p. 36).

To better appreciate the benefits of primitive polynomials consider $f(x) = x^3 + x^2 + x + 1$ and the associated shift register and sequences.



initial fill	1	0	0	
	0	0	1	
	0	1	1	
	1	1	0	period 4
	1	0	0	
initial fill	1	0	1	
	0	1	0	period 2
	1	0	1	
initial fill	1	1	1	period 1
	1	1	1	

Note that, as opposed to the earlier case, the above shift register has the following traits:

- no single initial fill generates all seven 3-tuples
- the period is always less than 2^n-1
- the associated polynomial is not prime, and hence not primitive, since

$$x^3 + x^2 + x + 1 = (x+1)^3.$$

APPENDIX C

PROGRAM LISTINGS

RUNBAL.M

```
function rundown = runbal(x)
%function rundown = runbal(x)
%program to calculate the running balance of vector x.
%
%inputs:
%       x = vector to be analyzed
%
%outputs:
% rundown = output vector of running balances

len = length(x);
rundown = zeros(len,1);

for i = 1:len
        rundown(i) = mean(x(1:i));
end
```

AUTOCORR.M

```
function acorr = autocorr(x)
%function acorr = autocorr(x)
%uses fft to calculate the autocorrelation of vector x
%note that this method is equivalent to the traditional
%slide, rotate, calc (agree-disagree)/total but is much
%faster.  the cost of this speed is that the sign of the
%correlation is lost.  sequences of length 2^n are handled
%particularly quickly.

fx = fft(x-mean(x));
fxconj = (fx').';
acorr = abs(ifft(fx .* fxconj));
acorr = fftshift(acorr);
acorr = acorr / max(acorr);
```

RUNS.M

```
%this program takes as input an n-long binary vector x.
%it then steps through calculating the number of n-tuples
%(n=1,...8) that occur.  these are calculated and stored by
%converting the n-tuples to their decimal equivalent.  the
%output decveci contains the count for i-tuples.  results
%are best viewed using a bar-graph output.  note that this
%program is not written as a function due to the large
%number of output vectors.  for ease of programming, note
%that for n=1:7, this program doesn't check the last (8-n)
%n-tuples.

%input:
%    x = binary vector
%
%output:
%    decveci for i=2:8 = incidence of the 2^i-1 i-tuples
%
decvec2 = zeros(4,1);
decvec3 = zeros(8,1);
decvec4 = zeros(16,1);
decvec5 = zeros(32,1);
decvec6 = zeros(64,1);
decvec7 = zeros(128,1);
decvec8 = zeros(256,1);
n = length(x);

for i = 1:(n-7)
        decimal = [2 1] * x(i:i+1);
        decvec2(decimal+1) = decvec2(decimal+1) + 1;

        decimal = [4 2 1] * x(i:i+2);
        decvec3(decimal+1) = decvec3(decimal+1) + 1;

        decimal = [8 4 2 1] * x(i:i+3);
        decvec4(decimal+1) = decvec4(decimal+1) + 1;

        decimal = [16 8 4 2 1] * x(i:i+4);
        decvec5(decimal+1) = decvec5(decimal+1) + 1;

        decimal = [32 16 8 4 2 1] * x(i:i+5);
        decvec6(decimal+1) = decvec6(decimal+1) + 1;

        decimal = [64 32 16 8 4 2 1] * x(i:i+6);
        decvec7(decimal+1) = decvec7(decimal+1) + 1;

        decimal = [128 64 32 16 8 4 2 1] * x(i:i+7);
        decvec8(decimal+1) = decvec8(decimal+1) + 1;

end
```

LINCOMP.M

```
function profile = lincomp(x)
%function profile = lincomp(x)
%generates linear complexity profile of vector x using
%an approximation to the Berlekamp-Massey algorithm.
%
%input:  x = binary vector
%
%output: profile = linear complexity such that each element
                   profile(n) is the approximate shift
                   register length required for the first n
                   bits of x.

profile = zeros(length(x),1);
sigma = [1];
tau = [0 1];
n = 0;
for j = 0:length(x)-1
%calculate a
        a = 0;
        for i = 1:length(sigma)
        a = a + sigma(i)*x(j+2-i);
        end
        a = rem(a,2);
        sigmaold=sigma;
%calculate sigmanew = sigmaold + a*tau  (only if a ~= 0)
         if a == 1
                lens = length(sigma);
                lent = length(tau);
%even up lengths of sigma and tau so they can be added
                if lens < lent
                        sigma = [sigma zeros(1,lent-lens)];
                else
                        tau = [tau zeros(1,lens-lent)];
                end
                sigma = sigma + tau;
                tau = tau(1:lent);
                sigma = rem(sigma,2);
        end
%strip trailing zeros off of sigma
        sigma = sigma(1:max(find(sigma)));
%calculate new tau and n
        if (a==0 | n<0)
                tau = [0 tau];
                n = n + 1;
        else
                tau = [0 sigmaold];
                n = -n;
        end
profile(j+1) = max(length(sigma),max(profile));
end
```

HENREAL.M

```
function [x,y] = henreal(n,x0,y0)
%function [x,y] = henreal(n,x0,y0)
%
%program to generate n-length real sequences based on the
%Hénon horseshoe attractor.  initial points fit into the
%trapping quadrilateral described in Hénon (1976).
%
%inputs:
%       n = length of desired sequence
%      x0 = initial x value
%      y0 = initial y value
%
%outputs:
%       x = n by 1 real vector
%       y = n by 1 real vector
%
x = zeros(n,1);
y = zeros(n,1);
x(1) = x0;
y(1) = y0;

%routine to check if initial points are valid
A=[3.4074 1;-.1083 -1; -3.64 1; -.1562 1];
B=[-4.1119 -.2760 -4.6718 -.3344]';
if min((A*[x0;y0]) > B) == 0
        disp('initial point outside trapping region')
        return
end

%recursive generation of points
for i = 1:n-1;
        x(i+1) = y(i) + 1 - 1.4*x(i)^2;
        y(i+1) = .3 * x(i);
end
```

```
function x = henon(n,x0,y0)
%function x = henon(n,x0,y0)
%
%program to generate n-length binary sequences based on the
%Hénon horseshoe attractor.  initial points are checked
%against the quadrilateral of convergence. (Hénon 1976)
%
%inputs:
%       n = length of desired sequence
%      x0 = initial x value
%      y0 = initial y value
%
%outputs:
%       x = n by 1 binary vector
%
x(1) = x0;
y(1) = y0;
split = .4098;   %median x-value of henon attractor

%routine to check if initial points are valid
A=[3.4074 1;-.1083 -1; -3.64 1; -.1562 1];
B=[-4.1119 -.2760 -4.6718 -.3344]';
if min((A*[x0;y0]) > B) == 0
        disp('initial point outside convergence zone')
        return
end

x(2:n) = zeros(n-1,1);   %vectors are preallocated here to
                         %save time
y(2:n) = zeros(n-1,1);   %in case initial point is outside
                         %of zone

%recursive generation of points
for i = 1:n;
        x(i+1) = y(i) + 1 - 1.4*x(i)^2;
        y(i+1) = .3 * x(i);

%convert previous point to binary
        if x(i) <= split
                x(i) = 0;
        else
                x(i) = 1;
        end

end

%minor housekeeping to dump the last term
x = x(1:n);
```

INIT.M

```
function [x,y] = init(n,seed)
%function [x,y] = init(n,seed)
%generates vectors of n initial points for henon horseshoe
%function where seed initializes the built-in random
%number generator.

rand('uniform')
rand('seed', seed)

x = zeros(n,1);
y = zeros(n,1);
a = [3.4074 1; -.1083 -1; -3.64 1; -.1562 1];
b = [-4.1119; -.2760; -4.6718; -.3344];

for j = 1:n
        xx = 5;         %dummy values to get into while loop
        yy = 5;

        while min((a * [xx;yy]) > b) == 0
           xx = 2.66*rand -1.33;
           yy = rand - .5;
        end

        x(j) = xx; y(j) = yy;
end
```

MEDBATCH.CPP

```
/*  program to find the x-median value of the Hénon
    horseshoe attractor.  input is read in from "starts",
    100,000 points are calculated and the output sent to the
    file "medians".  program call uses the DOS redirection
    and is "balbatch <starts >medians".  no error checking
    is done on input. */

#include <iostream.h>
#include <stdlib.h>
#include "thesis.h"

double xold, xnew, yold, ynew;
float  huge xvalues[100001];
int main()

{
while (cin >> xold >> yold)     //read in initial points
   {
   xvalues[0] = xold;

   for (long i=1; i<100000; i++)        //calculate attractor
     {
     xnew = yold + 1 - 1.4*xold*xold;
     ynew = .3 * xold;
     xold = xvalues[i] = xnew;
     yold = ynew;
     }

//sort xvalues
   heapsort(100000,xvalues);

//output median value
   cout << (xvalues[49999]+xvalues[50000])/2 << endl;
   }
return 0;
}
```

SENSITIV.M

```
function corrmat = sensitiv(seed)
%function corrmat = sensitiv(seed)
%test for sensitivity to initial conditions.  averages over
%100 sequences.  both x&y are perturbed to make sure new
%initial points are inside domain.
%

rand('uniform')
rand('seed', seed)
corrmat = zeros(18,6);
corrtemp = zeros(18,6);
m=[100, 500, 1000, 2000, 5000, 10000];

for i = 1:100
   xbase = zeros(100,1);
   i
   while max(xbase(2:100)) == 0
       x0 = 2.66*rand - 1.33;
       y0 = rand -.5;
       xbase = henon(10000,x0,y0);
   end

  for j = -18:-1
      xoffset =
           henon(10000,x0-sign(x0)*10^j,y0-sign(y0)*10^j);
      for k = m
corrtemp(-j,find(k==m)) = (-1).^xbase(1:k)' *
                (-1).^xoffset(1:k) / k;
      end
   end
   corrmat = corrmat + corrtemp;
end
corrmat = abs(corrmat/100);
```

PERIOD.FOR

```
c     Note that with minor modifications this program can be
c     run in batch or autolog mode.

      double precision period, xold, xnew, x0, yold, ynew, y0
      double precision tolerance, distance

c     call excms ('filedef 5 disk period2 initial')
      write (6,*) 'enter tolerance'
      read (5,*) tolerance
      write (6,*) 'enter initial x-point'
      read (5,*) x0
      write (6,*) 'enter initial y-point'
      read (5,*) y0

      xold = x0
      yold = y0
      period = 0.0
      distance = 5.0

      do 10 i = 1,1000
           xnew = yold + 1.0 - 1.4*xold*xold
           ynew = .3 * xold
           xold = xnew
           yold = ynew
10    continue

      x0 = xold
      y0 = yold

      do while (tolerance .lt. distance)
           period = period + 1.0
           xnew = yold + 1.0 - 1.4*xold*xold
           ynew = .3 * xold
           xold = xnew
           yold = ynew
           distance = max(abs(xnew-x0), abs(ynew-y0))
      end do

      write (6,*) 'final distance is ', distance
      write (6,*) 'the period is ', period
      end
```

BALTEST.M

```
function [statmat,balmat] = baltest(m,seed)
%function [statmat,balmat] = baltest(m,seed)
%program to calculate the balance property of bitstreams
%generated with the henon scheme.  tests sequences of length
%10000 and five sub-sequences for percentage of 1s present.
%will test m different sequences and store the results in
%balmat who's rows are then analyzed with min, max, and mean
%which are then written into statmat.
%
%inputs:
%       m = number of sequences to test
%    seed = seed to generate the m initial conditions
%
%outputs:
%  balmat = m by 6 matrix where the columns are the balance
%           statistic for the first 100, 500, 1000, 2000,
%           5000, and 10000 points respectively

rand('uniform')
rand('seed', seed)
balmat = zeros(m,6);
statmat = zeros(3,6);
a = [3.4070 1; -.1083 -1; -3.64 1; -.1562 1];
b = [-4.1119; -.2760; -4.6718; -.3344];

for j = 1:m
        x0 = 2.66*rand - 1.33;
        y0 = rand - .5;
        while min((a * [x0;y0]) > b) == 0
                x0 = 2.66*rand - 1.33;
                y0 = rand - .5;
        end

        x = henon(10000,x0,y0);
        balmat(j,1) = norm(x(1:100))^2/100;
        balmat(j,2) = norm(x(1:500))^2/500;
        balmat(j,3) = norm(x(1:1000))^2/1000;
        balmat(j,4) = norm(x(1:2000))^2/2000;
        balmat(j,5) = norm(x(1:5000))^2/5000;
        balmat(j,6) = norm(x(1:10000))^2/10000;
end

%calculate statistics on balances
if m>1
        statmat(1,:) = min(balmat);
        statmat(2,:) = max(balmat);
        statmat(3,:) = mean(balmat);
else
        clear statmat
end
```

APPENDIX D

The following sequence is the result of the coin flip experiment. The 1's represent heads while the 0's are tails.

0 1 0 0 1 0 1 1 0 1 0 1 0 1 1 1 1 0 1 1 0 1 0 1 0 1 1 1 1 0
0 0 0 0 1 0 0 1 0 1 0 0 0 1 1 0 1 1 1 1 0 1 0 1 1 1 1 0 1 0
1 0 1 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 0 1 0 0 1 1
0 0 0 0 1 1 1 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 0 1 1
1 0 1 0 1 1 1 1 0 0 1 0 0 0 1 0 1 0 1 1 0 1 1 0 0 1 0 0 1 1
0 0 1 0 0 1 0 0 0 0 0 1 0 1 0 0 0 1 0 0 1 0 0 1 1 0 1 1 0 1
1 0 1 0 1 0 1 1 1 0 1 0 1 1 0 0 1 1 0 0 0 0 1 0 1 0 1 1 1 0
0 0 0 1 1 0 1 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 1 1 0 1 0
0 0 0 1 0 0 0 0 1 0 0 0 1 1 1 1 1 0 1 1 1 0 1 0 0 0 1 1 1 1
0 0 0 1 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 1 0 0 1 0 0 1 0 0 1
0 0 1 0 1 0 1 0 0 0 0 1 0 0 1 1 0 0 1 0 0 0 1 0 1 0 0 1 0 0
0 0 1 1 1 1 0 1 0 1 0 1 1 0 0 1 0 0 0 0 0 1 1 0 1 1 0 1 0 0
1 1 0 1 1 1 1 0 0 0 1 1 1 0 0 1 0 0 1 0 1 0 1 0 0 0 0 1 0 0
1 1 0 1 1 1 1 0 0 1 1 1 1 0 1 1 0 0 0 0 1 0 0 1 1 0 1 0 0 0
0 1 0 0 0 1 1 0 0 0 0 0 0 0 1 1 1 0 0 1 0 1 0 0 1 1 1 0 0 1
1 1 0 1 0 1 0 0 0 1 1 0 0 0 1 1 0 0 0 0 1 1 0 1 1 1 0 1 1 1
1 0 0 0 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 1 1 0 0 0 1 0 0 0 0
1 0 0 0 0 0 1 0 0 1 0 1 1 1 1 1 0 1 1 0 0 0 0 0 1 0 1 1 0 0
1 0 0 0 1 1 0 0 0 1 0 0 1 1 0 0 1 1 0 1 1 1 1 0 0 0 0 1 0 0
0 0 0 0 1 0 1 0 0 0 1 1 0 0 1 0 1 1 1 1 0 1 0 0 0 0 0 1 0 1
0 0 1 1 0 0 0 0 1 1 1 0 0 1 1 0 1 0 0 1 0 0 1 0 1 0 0 1 1 1
1 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 0 0 1 0 1 0 0 1 0 1 0 0 0 0
1 0 0 0 0 0 1 0 0 0 0 1 1 0 1 0 0 1 0 0 1 0 0 0 0 1 1 0 1 0
0 0 0 0 1 1 1 1 0 0 1 0 0 1 1 1 1 0 1 0 0 1 0 0 0 0 0 0 0 0
1 0 0 0 0 1 1 0 1 1 1 0 0 1 0 1 0 1 1 1 1 1 0 0 0 1 1 0 1 1
0 0 0 0 1 1 0 1 1 1 1 1 0 1 0 0 0 1 0 1 1 0 1 0 1 1 0 1 1 0
0 1 0 1 0 1 1 1 1 0 1 0 1 0 0 0 0 1 0 0 0 1 1 1 1 0 0 1 1 1
0 0 0 0 1 0 1 1 0 1 1 0 0 1 1 0 0 0 1 1 1 0 1 1 1 1 0 1 1 0
1 0 1 0 1 0 1 0 1 0 0 1 0 0 1 1 1 0 0 1 1 0 0 0 0 0 0 0 0 1
0 0 0 0 0 1 1 0 1 1 0 0 0 1 1 1 0 0 0 1 0 0 1 0 1 0 1 1 0 1
1 0 1 0 0 1 1 1 0 1 1 0 1 0 1 1 1 0 1 1 1 0 0 1 0 0 0 1 1 1
1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 1 1 1 1 0 1 1 0 0 1 1 0 0 1 1
0 0 0 0 1 1 1 0 0 1 1 0 1 1 0 1 0 0 0 1 1 1 1 1 1 0 0 1 0 0
1 0 1 1 1 1 0 1 1 0

APPENDIX E

The following sequence is the result of the stacked sine wave generation scheme.

```
0 1 1 0 0 1 1 0 1 1 1 1 0 0 0 1 0 1 1 1 0 1 1 0 0 1 1 0 1 0
1 0 1 0 0 1 0 0 1 1 0 1 1 1 0 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1
1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1
1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1
1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 0
1 1 0 0 1 1 0 1 1 1 0 0 1 0 1 0 1 0 1 0 1 1 0 0 1 1 0 0 1 1
0 1 1 0 0 0 1 0 1 0 1 1 1 0 1 1 0 0 1 1 0 0 1 1 0 1 1 0 1 0
1 0 1 0 1 1 1 0 1 1 0 0 1 1 0 1 1 1 0 0 1 0 1 0 1 0 1 0 1 1
0 0 1 1 0 0 1 1 0 1 1 1 0 0 1 0 1 0 1 1 1 0 1 1 0 0 1 1 0 1
1 1 0 1 1 0 1 0 1 0 1 0 1 1 1 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0
1 0 1 1 1 0 1 1 1 0 1 1 0 0 1 1 0 1 1 1 0 0 1 0 1 0 1 1 1 0
1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0 1 0 1 0 1 1 1 0 1 1 0 0 1 1
0 1 1 1 0 1 0 0 1 0 1 1 1 0 1 1 1 0 1 1 0 0 0 1 0 1 0 1 1 0
0 0 1 0 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 1 1 0 0 0 1 0 1 1 1
0 1 1 0 0 1 1 0 1 0 1 0 1 0 0 1 0 0 1 1 0 1 1 1 0 1 1 0 0 0
0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0
0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0
0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0
0 0 1 1 1 0 0 0 1 0 1 1 0 0 1 1 0 1 1 1 0 0 1 0 1 0 1 0 1 0
1 1 0 0 1 1 0 0 1 1 0 1 1 0 0 0 1 0 1 0 1 1 1 0 1 1 0 0 1 1
0 0 1 1 0 1 1 0 1 0 1 0 1 0 1 1 1 0 1 1 0 0 1 1 0 1 1 1 0 0
1 0 1 0 1 0 1 0 1 1 0 0 1 1 0 0 1 1 0 1 1 1 0 0 1 0 1 0 1 1
1 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0 1 0 1 0 1 1 1 0 1 1 0 0
1 1 0 1 1 1 0 1 1 0 1 0 1 1 1 0 1 1 1 0 1 1 0 0 1 1 0 1 1 1
0 0 1 0 1 0 1 1 1 0 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 0 1 0 1 0
1 1 1 0 1 1 0 0 1 1 0 1 1 1 0 1 0 0 1 0 1 1 1 0 1 1 1 0 1 1
0 0 0 1 0 1 0 1 1 0 0 0 1 0 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1
1 1 0 0 0 1 0 1 1 1 0 1 1 0 0 1 1 0 1 0 1 0 1 0 0 1 0 0 1 1
0 1 1 1 0 1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0
1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0
1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0
1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 0 1 1 0 0 1 1 0 1 1 1
0 0 1 0 1 0 1 0 1 0 1 1 0 0 1 1 0 0 1 1 0 1 1 0 0 0 1 0 1 0
1 1 1 0 1 1 0 0 1 1
```

REFERENCES

Bloch, Norman J., *Abstract Algebra with Applications*, Prentice-Hall, Inc, 1987.

Forré, Réjane, "The Hénon Attractor as a Keystream Generator", preprint, 1990.

Golomb, Solomon W., *Shift Register Sequences*, Aegean Park Press, 1982.

Hénon, Michel, "A two-dimensional mapping with a strange attractor", *Communications in Mathematical Physics*, 50:69-77, 1976.

Koblitz, Neal, *A Course in Number Theory and Cryptography*, Springer-Verlag, 1987.

National Security Agency, *Spread Spectrum Signals and Techniques Handbook*, 1981.

Peitgen, Heinz-Otto, Hartmut Jürgens, and Dietmar Saupe, *Chaos and Fractals: New Frontiers of Science*, Springer-Verlag, 1992.

Reuppel, Rainier A., *Analysis and Design of Stream Ciphers*, Springer-Verlag, 1986.

Shannon, Claude E., "Communication theory of secrecy systems", *Bell Systems Technical Journal*, 28:656-715, 1949.

Stamp, Mark, *A Generalized Linear Complexity*, Ph.D. Dissertation, Texas Tech University, Lubbock, TX, 1992.

Stewart, Ian, *Does God Play Dice? The Mathematics of Chaos*, Blackwell, 1991.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Department Chairman, Code MA/Fe Department of Mathematics Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Professor J. Leader, Code MA/Le Department of Mathematics Naval Postgraduate School Monterey, CA 93943-5000	3
5.	Professor M. Stamp Mathematical Sciences Worcester Polytechnic Institute 100 Institute Rd. Worcester, MA 01609-2280	1
6.	Fontana, Tony 1290 5th St. #1 Monterey, CA 93940	1
7.	Heyman, James 585 Laine St. #5 Monterey, CA 93940	5