The Logistic Lattice in Random Number Generation

Neal R. Wagner[1]
Division of Mathematics, Computer Science, and Statistics
University of Texas at San Antonio
San Antonio, TX 78249, USA

Abstract. This paper describes pseudo-random number generators based on non-linear dynamics. The recommended generator uses a special "re-mapped" form of the logistic equation at each of a finite set of nodes arranged in a 1- or 2-dimensional ring. Each node receives small perturbations from adjacent nodes by means of diffusive coupling equations—the so-called *coupled map lattice*. The generator is notable as the first to use non-linear dynamics in a lattice with a continuous state at each node. Time and space are discrete, but the state is continuous, unlike the cellular automata that Wolfram used to produce random numbers. It is a new source of pseudo-random numbers, unlike other current sources. The paper gives evidence that this generator possesses good statistical properties and a very long period, except in cases of negligible probability.

1. Introduction

Wolfram [11] used cellular automata and chaos theory to obtain pseudo-random numbers, but there has been little work on chaos methods utilizing floating point numbers, and such work was mostly carried out forty years ago.

An approach using iterative chaotic equations and floating point numbers must overcome several problems [2], [3], [8]:

(i) *Unknown distribution.* Usually the distribution of numbers produced by these non-linear equations is not known and probably not expressible in terms of known functions.

(ii) *Basins of influence of short cycles.* A number that gets close to a short cycle, even an unstable one, stays close for awhile (depending on how close it was initially), so the equations exhibit decidedly non-random behavior.

(iii) *No detailed quantitative theory.* Most of the theoretical results about these equations are qualitative and relate to global behavior.

(iv) *Theory only for real numbers.* The existing theory applies to infinite precision real numbers. Floating point numbers often behave quite differently.

This paper handles item (i) using a variation of the logistic equation, for which the distribution is known. Item (ii) requires that one iterate an equation long enough for the number to fill with noise before sampling a random number. Items (iii) and (iv) can only be approached with a statistical analysis of experimental data and with assumptions that floating point numbers will behave in some respects like infinite precision numbers.

[1]This material is based in part upon work supported by the Texas Advanced Research Program.

There has been a great deal of recent work on pseudo-random number generators, of which [1], [6], and [7] are typical. This paper describes a completely new generator, not currently studied by other researchers. The discussion focuses first on cycle length because that is the basic requirement of a reasonable random number generator. With the approach in this paper, the sequence is known to be equidistributed in infinite precision (see below). Assuming one iterates to fill the number with noise, one expects good statistical properties as long as the cycle length is long.

2. The Logistic Equation

The iterative equation $f(x) = 4x(1-x), 0 \le x \le 1$, known as the *logistic* equation, is historically interesting as one of the earliest proposed sources of pseudo-random numbers. Ulam and von Neumann suggested its use in 1947 [9], partly because it had a known algebraic distribution, so that iterated values could be transformed to the uniform distribution. The equation was mentioned again in 1949 by von Neumann [10] and much later in 1969 by Knuth [5, Exercise 3.4.1-24], but it was never used.

2.1. *Behavior in Infinite Precision*

Iterates of the "tent" function $g(u) = 1 - 2|u - (1/2)|, 0 \le u \le 1$ produce a uniform distribution, and g can be transformed to f via $T(u) = \sin^2((\pi/2)u)$. Both T and its inverse $T^{-1}(v) = (2/\pi)\arcsin(\sqrt{v})$ preserve the cycle structure of iterates of f and g.

If one is working with real numbers (infinite precision) then for all starting values except for a set of measure zero, iterates of g are equidistributed (= uniformly distributed) [5], [10]. Similarly, except for a set of measure zero, iterates of f are distributed according to $(1/\pi)\int_0^v dx/\sqrt{x(1-x)}$, and the sequence $\{T^{-1}(f_n(v))\}$ is equidistributed, again for any v outside a set of measure zero.

Despite being equidistributed, the basins of influence of the short cycles of g produce significant non-random behavior. A starting value very close to a value in a short cycle will stay close to that cycle for awhile, though all cycles are unstable, meaning that the values will eventually diverge. The cycle structure of f and g is complicated, with cycles of every finite length, and with countably many distinct starting values leading into each cycle (except for the cycle $(3/4)$). In fact the number of cycles of length n is equal to or a little less than $\textbf{floor}((2^n - 1)/n)$, depending on the factorizations of $2^n \pm 1$.

2.2. *Behavior in Finite Precision*

In finite precision the behavior of f and g above is quite different from the theoretical results for infinite precision. Using a typical binary floating point unit, iterates of g from any starting value converge immediately to zero. (At each iteration, another significant bit becomes zero.) Iterates of f are more irregular, producing cycles as if successive values were chosen "at random." If successive values of f were chosen uniformly from n available numbers, then one would expect with probability 0.5 a duplicate value after about $1.18\sqrt{n}$ iterations and a cycle of length about $0.59\sqrt{n}$. In actuality the choices are not uniform, and the floating point numbers themselves are not uniformly distributed, but the behavior is not too far from cycles of the above lengths.

Table 1 shows the cycle lengths obtained by experiments using different kinds of hardware and precisions, as well as the percent of starting values leading into the cycle and the average initial run before the cycle starts. The cycle search in single precision is exhaustive

Hardware	Precision	Cycle Structure: Cycle length	Cycle Structure: Percent occurrence	Cycle Structure: Average initial run	Number of starting values
all types	single	**1**	**93.0%**	**2034**	4 194 305, all x such that $0.75 \leq x \leq 1$
		930	5.6%	340	
		431	1.0%	251	
		106	0.35%	244	
		205	0.1%	83	
		5	0.002%	31	
		4	0.0004%	7	
		3	0.00005%	2	
		1	0.00002%	0	
Weitek or Mips or Sparc	double	5 638 349	69.4%	54 000 000	7191
		1	**15.2%**	**10 000 000**	
		14 632 801	11.3%	8 500 000	
		10 210 156	1.5%	5 900 000	
		2 625 633	1.3%	3 800 000	
		2 441 806	1.2%	5 200 000	
		1 311 627	0.028%	240 000	
		960 057	0.014%	200 000	
VAX	double	86 058 517	84.0%	68 000 000	102
		1	**16.0%**	**44 000 000**	

Table 1: Cycles of the logistic equation (boldface = the cycle(0)).

because every cycle will eventually enter the range from $3/4$ to 1. Notable in these experiments are the small number of distinct cycles that appear, their short lengths, and the number of times the iteration falls into the cycle (0). (All cycle lengths are exact, but average initial runs have been rounded.)

The results in the table are hardware dependent. If the hardware differs in any way, or even in some cases if the computations are reordered or optimized, the results will be completely different in detail, but one still sees just a small number of short cycles. The entries for double precision do not use exhaustive search, so there are infrequently occurring cycles that are not listed. Each table entry refers to a unique cycle of the given length. It is possible to have two distinct cycles of the same length, but in this paper that only occurs with cycles of length 1: in Table 1, the cycle (0) (occurring 93% of the time), and the cycle $(3/4)$ (occurring 0.00002% of the time). The software checked that after getting a first cycle of a given length, each subsequent cycle of the same length had a number in common with the first one.

The frequent convergence to the cycle (0) can partly be explained by taking iterated inverse images under f from 0. In infinite precision one gets only countably many numbers leading into (0), so that the iteration falls into the cycle (0) with probability 0. Now switch to finite precision and consider the identity $f(0.5+\epsilon) = 1-4\epsilon^2$. If ϵ is small, then $4\epsilon^2$ will be very small, so that roundoff error will cause $1-4\epsilon^2$ to be exactly 1. Thus one gets a whole

Figure 1: The original and re-mapped logistic equation.

interval around 0.5 mapping to 1. Similarly there are two intervals mapping into the interval around 0.5, and so forth.

So far the logistic map sounds useless for pseudo-random number generation. For most starting values v, the sequence $\{T^{-1}(f_n(v))\}$ will be roughly equidistributed, but grossly non-random looking. Moreover, in finite precision there is a sizable probability of convergence to zero, and otherwise it may go into a fairly short cycle.

3. The Re-mapped Logistic Equation

The logistic equation yields numbers very close to 0 (on the positive side) and very close to 1. Available floating point numbers "pile up" near 0, but there is no similar behavior near 1. It is possible to restructure the equation so that values occurring near 1 are re-mapped to the negative side of 0. The following definition does this, mapping [-1, 1] to itself:

$$F_\beta = \begin{cases} 2|x|(2-|x|), \text{ for } |x| \le \beta, \\ -2(1-|x|)^2, \text{ for } \beta < |x| \le 1, \end{cases} \tag{1}$$

where $\beta = 1 - (1/\sqrt{2})$. A graph of the original (expanded by a factor of two) and re-mapped versions is shown in Figure 1.

In infinite precision, this re-mapped equation behaves exactly like the original, but with floating point numbers there is no longer any convergence to the cycle (0) of length 1. Unlike the original logistic equation, this version does not have an interval of numbers mapping into this cycle, so it is "well-tuned" to floating point numbers, since it requires and utilizes extra precision near 0 (on both sides).

Table 2 gives the cycles obtained by experimentation with the re-mapped equation. Notice that they are much longer on the average than the cycles of Table 1.

To transform numbers generated by $F_\beta(x)$ back to a uniform distribution one can use

$$S(x) = \begin{cases} (2/\pi)\arcsin(\sqrt{x/2}), \text{ for } 0 \le x < 1, \\ (2/\pi)\arcsin(\sqrt{-x/2}) + 0.5, \text{ for } -1 < x < 0, \end{cases} \tag{2}$$

An alternative to the remapped logistic equation is the "sawtooth" function: $h(x) =$

Hard-ware	Preci-sion	Cycle Structure			Number of starting values
		Cycle length	Percent occurrence	Average initial run	
all types	single	13753	89.9%	4745	8 388 609, all x such that $0.5 \leq x \leq 1$
		3023	5.4%	1150	
		2928	3.4%	670	
		1552	0.66%	355	
		814	0.6%	266	
		9	0.035%	191	
		1	0.00017%	14	
		3	0.000024%	1.5	
		1	0.000012%	1	
Weitek or Mips or Sparc	double	112 467 844	80.5%	105 000 000	1189
		61 607 666	5.7%	23 000 000	
		35 599 847	4.3%	19 000 000	
		1 983 078	3.6%	39 000 000	
		4 148 061	3.3%	60 000 000	
		15 023 672	2.5%	19 000 000	
		12 431 135	0.084%	5 500 000	
		705 986	0.084%	670 000	
M68040	double	73 573 097	80.6%	125 000 000	475
		38 326 216	16.8%	112 000 000	
		6 006 146	2.1%	34 000 000	
		5 195 797	0.42%	10 000 000	
Cray Y-MP	double	13 641 539	61.1%	27 500 000	3161
		1 281 377	20.5%	10 000 000	
		3 861 436	10.9%	7 000 000	
		1 630 338	2.9%	4 000 000	
		1 344 786	2.7%	2 800 000	
		6 034 850	1.1%	1 400 000	
		606 688	0.66%	2 000 000	
		782 542	0.095%	600 000	
		161 757	0.032%	200 000	
VAX	double	391 307 825	99.1%	284 000 000	110
		55 897 032	0.9%	45 000 000	

Table 2: Cycles of the re-mapped logistic equation.

(mx) **mod** $1 = (mx) -$ **floor**(mx), where m is not an integer (the only interesting case). The author carried out experiments in single and double precision, with m equal to 1.5, with m taking on two values close to π, and with $m = 3\ 141\ 592\ 653.7$. The resulting cycle lengths were very similar to those in Figure 2, except that round-off error produced short cycles for the large value of m. It seems likely that this equation, with some small non-integer m, would make an acceptable replacement for the remapped logistic equation in Section 4.

4. The Logistic Lattice

The *coupled map lattice* is a dynamical system with discrete time and space, and a continuous state. In recent years there has been a tremendous amount of research on these systems—hundreds of articles, of which [4] is typical. The models studied are usually in one or two dimensions, with "periodic boundary conditions," *i.e.*, the boundary wrapped in a circle or a torus. Researchers usually employ a logistic map at each lattice point and use *Laplacian* coupling. In this way they study a simple model for fluid mechanics which preserves the essential features of the partial differential equations, while remaining more tractable numerically. In one dimension the equations have the form:

$$x_{n+1}^{i} = f(x_n^i) + \nu[f(x_n^{i-1}) - 2f(x_n^i) + f(x_n^{i+1})], \text{ for } 0 \le i < m,$$

Here m is the number of nodes, and ν is a constant known as the *viscosity*. All subscripts are calculated modulo m. The computation proceeds from step n to step $n+1$. It is important that ν be very small so that the equation will continue to have (almost) the same distribution as the unmodified logistic equation. Also small values for ν give the "fully developed turbulence" that is desirable for pseudo-random number generation. Combining terms gives:

$$x_{n+1}^{i} = (1 - 2\nu)f(x_n^i) + \nu[f(x_n^{i-1}) + f(x_n^{i+1})], \text{ for } 0 \le i < m, \tag{3}$$

In two dimensions and after combining terms, the equation becomes:

$$\begin{aligned} x_{n+1}^{i,j} &= (1 - 4\nu)f(x_n^{i,j}) + \nu[f(x_n^{i,j-1}) + f(x_n^{i,j+1}) + f(x_n^{i-1,j}) + f(x_n^{i+1,j})], \\ &\quad \text{for } 0 \le i < m, 0 \le j < m, \end{aligned} \tag{4}$$

The author has carried out various experiments with these equations, using for f the remapped logistic equation of section 3 (equation (1)), and using $\nu = 10^{-6}$ in single precision, $\nu = 10^{-14}$ in double precision, and larger values of ν for smaller precisions that are software simulated. In one dimension the experiments employed ring sizes m of 3, 4, and 5, and in two dimensions they used $m = 3$, *i.e.*, a 3×3 torus. The first basic property to study is the cycle length. This is found for a sequence $\{x_n\}$ by looking for the first value of n for which $x_n = x_{2n}$. (The cycle length is then a divisor of n.) Even in single precision a cycle never occurred, so one must be satisfied with a search for simpler events. Iterate either $\{x_n^i\}$ or $\{x_n^{i,j}\}$ for $n = 1, 2, 3, \ldots$ looking for one of the following events. (Here the *notation* is 1-dimensional, but the same definitions apply in the two-dimensional case.)

- *full hit* (or *m-hit*) event: $\{x_n^j\} = \{x_{2n}^j\}$, for all $0 \le j < m$, where n is fixed and as small as possible. (This is what one is looking for: a cycle whose length is a divisor of n.)
- *k-hit* event: $\{x_n^j\} = \{x_{2n}^j\}$, for k values of j, $k < m$. (This is only of interest in saying how likely a full hit might be.)
- *single dup* event: $\{x_n^j\} = \{x_n^l\}$, for some $j \ne l$, and for some n.

Configuration	Precision	# of iterations	# of hits	# of dups
1-dim, ring of 5	single	26.1 $\times 10^9$	1531	3024
1-dim, ring of 5	double	39.7 $\times 10^9$	0	0
2-dim, ring of 3 $\times$ 3	single	4.1 $\times 10^9$	401	1667
2-dim, ring of 3 $\times$ 3	double	4.35 $\times 10^9$	0	0

Table 3: Experiments with coupled equations.

- *dup* event: There are disjoint subsets $J_1, J_2, \ldots, J_k$ of $\{0, 1, \ldots, m-1\}$, where each J_i has size at least 2, and where for each i and for each $j_1 \neq j_2$ in J_i, it is true that $\{x_n^{j_1}\} = \{x_n^{j_2}\}$. (Dup events are significant because certain dups lead to a stable duplicated state.)
- *full dup* event: A dup event with $k = 1$ and $J_1 = \{0, 1, \ldots, m-1\}$.
- *k-dup* event: There are k duplicated pairs.
- *stable dup* event: a dup event where the duplicated nodes remain duplicated for all subsequent iterations. (Of course a full dup is stable. For $m = 3$, any single dup is stable. For $m = 4$, any single dup of opposite nodes is stable, while a single dup of adjacent nodes is not stable.)

Table 3 summarizes the results of one set of experiments, using $m = 5$ in one dimension, $m = 3$ in two dimensions, and the remappped logistic equation. In single precision, only single hit and single dup events occurred. In double precision there were no hit or dup events at all.

After finitely many iterations, there must be a cycle. This could occur as a single full-hit event without any initial stable dup event, but this outcome has negligible probability. Instead with high probability (essentially 1), there will be some stable dup event, *i.e.*, duplications continuing with all subsequent iterations.

THEOREM.

(a) *In a 1-dimensional ring of size m, the smallest initial stable dup event involves at least $k =$* **floor**$((m-1)/2)$ *duplicated pairs.*

(b) *In a 3×3 torus, the smallest initial stable dup event involves at least 3 duplicated pairs.*

PROOF (*Sketch of (a)*). Start with a ring of m nodes. Suppose there is an initial stable dup with a minimum number of duplications. This means there are two nodes with the same value repeated indefinitely. (For this argument, start with any duplicate pair.) Since these nodes must get the same perturbations at each iteration, they must each have two neighbors with the same values repeated indefinitely, and so on. (Actually, the *sum* of the values of perturbations must be the same, but this sum cannot stay the same unless the individual values are the same. Also a given perturbation could be zero once or a few times, but cannot remain zero indefinitely.) The initial duplicate pair splits the ring into two remaining segments of length i and j, both $\leq m-2$ and both ≥ 0. Suppose x below marks the original duplicated pair. Additional duplicated pairs on both sides of the x's can occur in two ways. They can be *nested*, like $(\ldots yxz \ldots zxy \ldots)$ or they can be *alternating*, like $(\ldots yxz \ldots yxz \ldots)$. In case m is odd, one of i or j must be even (includes 0) and the other odd. Here the simplest stable dup must be nested, with $(m-1)/2$ duplicated pairs, and one node left over on the odd side. In case m is even, there are three cases. One can have both i and j odd and $(m/2)-1$ nested pairs. Or one can have both i and j even (one possibly 0), with $m/2$ nested pairs. Finally, one can have $i=j$ and $m/2$ alternating pairs. This completes the proof.

In particular, the theorem says that a generator with at least $m = 7$ nodes in one dimension or 3×3 nodes in two dimensions requires three simultaneous duplicated pairs as the smallest initial stable dup event. In *single* precision, the results of Table 3 give estimates of very roughly 10^{-8} for the probability of a single hit at a given node or a single dup at a given pair of nodes. Because of the very small couplings involved, it is plausible to conjecture that actions at separate nodes are statistically independent of one another. Given this independence, one gets 10^{-25} as a very rough estimate for the probability of the first stable dup event, using single precision. Similar estimates for a full hit event before any stable dups give probability 10^{-56}, so one can discount the chance of this happening before a stable dup.

In double precision, the experiments in Table 3 produced no hit or dup events at all in 4.4×10^{10} iterations, so there are no estimates of their likelihood, but the probabilities are certainly very much smaller than in single precision. For practical purposes one can completely discount the chance of a stable dup or full hit event in double precision.

Extensive experiments were also run in software with simulated precisions of 8 and 12 significant bits, and were fully consistent with the above conclusions. Here as one would expect, the system eventually fell into an initial stable duplicated state. From such a state, further stable dups or a full hit were much more likely. Of course the system must terminate with a full hit and a cycle, but the full hit always involved only one or two separate values at all nodes.

5. Recommended Generator

The recommended generator, as coded in C in the Appendix, uses a coupled map lattice with at least $m = 7$ nodes in one dimension, or 3×3 nodes in two dimensions. It also uses the remapped logistic equation, and uses *decimation*, *i.e.*, values are sampled only after every 56 iterations, always from the same node. The "seed" for this random number generator is an array of m double-precision reals in the range from -1 to 1. Each separate array of m reals provides a different starting point for the generator. Some values are poor choices for a starting point: all the same value gives the equivalent of just a single logistic equation, and all 0's gives a fixed point. In practice, initializing the seed array using a auxiliary generator will work well, as shown in the Appendix. The author conjectures that the nodes are statistically independent of one another, so by sampling all nodes at once one would get a speedup by a factor of m.

As coded, the generator yielded good results when subjected to standard statistical tests. For example, the Kolmogoroff-Smirnov test was run with $10\ 000$ sets of 1000 numbers each (for $10\ 000\ 000$ iterations total), comparing the distribution of the $10\ 000$ test results with the expected distribution, as described in [5]. Good performance on statistical tests is now an old-fashioned way to certify a pseudo-random number generator [6], but this is the best one can do with the generator described here.

6. Conclusions

This article has presented a new pseudo-random number generator based on an interesting problem from non-linear dynamics—namely the use of a variation of the logistic equation at each node of a coupled 1-dimensional or 2-dimensional lattice. There is statistical and theoretical evidence that this generator provides excellent pseudo-random numbers.

Acknowledgments

The author thanks Kay Robbins, Steve Robbins, Dave Eberly, Hugh Maynard, Marianne Cain, Paul Putter, and Stan Zietz for help. For at least four years of CPU time on various machines, thanks are due to Drexel University, the University of Texas at San Antonio, Sun Microsystems Inc., Digital Equipment Corporation, NeXT Computer, Inc., and the University of Texas System Center for High Performance Computing. Finally support came from Harold Longbotham and his Nonlinear Signal Processing Lab, and from the Texas Advanced Research Program.

References

[1] S.L. Anderson. "Random number generators on vector supercomputers and other advanced architectures." *SIAM Review 32* (2, June 1990) 221251.

[2] P. Collet, and J.-P. Eckmann. *Iterated Maps on the Interval as Dynamical Systems.* Birkhaeuser, 1980.

[3] R.L. Devaney. *An Introduction to Chaotic Dynamical Systems, Second Edition.* Benjamin/Cummings, 1989.

[4] K. Kaneko. "Spatiotemporal chaos in one- and two-dimensional coupled map lattices." *Physica D 37* (1989) 6082.

[5] D.E. Knuth. *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms, Second Edition.* Addison-Wesley, 1981.

[6] S. Micali and C.P. Schnorr. "Efficient, perfect random number generators." *Advances in Cryptology, Lecture Notes in Computer Science 403.* Springer Verlag, 1990, 173198.

[7] S.K. Park and K.W. Miller. "Random number generators: good ones are hard to find." *Comm. ACM 31* (1988) 11921201.

[8] C. Preston. *Iterates of Maps on an Interval.* Lecture Notes in Mathematics, No. 999, Springer Verlag, 1983.

[9] S.M. Ulam and J. von Neumann. "On combination of stochastic and deterministic processes" (abstract). *Bull. Amer. Math. Soc 53* (1947) 1120.

[10] J. von Neumann. "Various techniques used in connection with random digits." *Applied Math. Series 12*, National Bureau of Standards, 1951, 3638.

[11] S. Wolfram. "Random sequence generation by cellular automata." *Advances in Applied Mathematics 7* (1986) 123169.

Appendix. Source for the recommended generator (ANSI C).

The header file "random.h"

```
#define NMAX 7  /* NMAX should be >= 7 */
double chaotic_uniform (double [NMAX]);
/************************************************************************
 * Generates uniformly distributed pseudorandom doubles in (0,1).         *
 * First initialize the array t. One could use an auxiliary generator ran() *
 *   that produces doubles in (0,1). Then invoke chaotic_uniform repeatedly.*
 * Thus the "seed" is an array t of NMAX doubles in (-1,1).  Like any       *
 *   seed for a RNG, the array t will change after each call.               *
 *   The applications program provides storage for the seed array t.        *
 * Typical usage by an application--to produce 100 random numbers:         *
 *      #include "random.h"                                                 *
 *      double ran(void);                                                   *
 *      double t[NMAX];                                                     *
 *      long i;                                                             *
 *      double result[100];                                                 *
 *      for (i = 0; i < NMAX; i++)                                          *
 *            t[i] = 2.0*ran() - 1.0;                                       *
 *      for (i = 0; i < 100; i++)                                           *
 *            result[i] = chaotic_uniform(t);                               *
 * The constant NMAX, the seed array t, and its initialization could all be *
 *   buried in the source file for unsophisticated users.                   *
 ************************************************************************/
```

The source file "random.c"

```
#include <math.h>                         /* Needed for asin, sqrt and fabs */
#include "random.h"                       /* Header file                    */
#define BETA 0.292893218813452476         /* Magic number used in f         */
#define NU 1.0e-14                        /* Viscosity constant in step     */
#define TWO_DIV_PI 0.636619772367581343   /* 2/Pi used in S                 */
#define NSTEP 28                          /* Half the # of steps to iterate */
/*======================================================================*/
static long mod (long i, long j)          /* If i is negative, then i%j     */
{                                         /* may be negative.  In general,  */
       long k = i%j;                      /* result is machine dependent.   */
       if (k < 0)  k = k + j;             /* Check your own architecture.   */
       return(k);
}
/*---------------------------------------------------Equation (1)---------*/
static double f (double x)                   /* Remapped logistic equation */
{
       double temp = fabs(x);
       if ( temp <= BETA ) return(2.0*temp*(2.0-temp));
       else return(-2.0*(1.0-temp)*(1.0-temp));
}
/*---------------------------------------------------Equation (3)---------*/
static void step (double t[], double tn[])         /* Coupled map lattice */
{
       long i;
       for (i = 0; i < NMAX; i++)  t[i] = f(t[i]);
       for (i = 0; i < NMAX; i++)
          tn[i] = (1.0-2.0*NU)*t[i] + NU*(t[mod(i-1,NMAX)]+t[mod(i+1,NMAX)]);
}
/*---------------------------------------------------Equation (2)---------*/
static double S (double x)                /* Change distribution to uniform */
{
       if (x >= 0) return(TWO_DIV_PI*asin(sqrt(x/2)));
       else return(TWO_DIV_PI*asin(sqrt(-x/2)) + 0.5);
}
/*----------------------------------------------------------------------*/
double chaotic_uniform (double t[NMAX])      /* Iterate step 2*NSTEP times */
{
       long i;
       double tn[NMAX];                      /* Extra copy of seed array t */
       for (i = 0; i < NSTEP; i++) {
              step(t, tn);
              step(tn, t);
       }
       return (S(t[0]));
}
/*======================================================================*/
```