Not Knowing Your Random Number Generator Could Be Costly: Random Generators - Why Are They Important

By Casimir C. “Casey” Klimasauskas

“Random numbers are too important to be left to chance.”

R. R. Coveyou

Introduction

Have you played a video game lately? Purchased something on the web? Trained a Neural Network? Used a Genetic Algorithm for optimization? Run software from Microsoft? Applied textures to a photograph? Played the stock market? If the answer is “yes” to any of these questions, Random Number Generator (RNG) have affected your life.

Can you trust your Random Number Generator? Can you test its performance? Does its performance really matter? In addition to some RNG related horror stories, this article presents guidelines for performing basic random number evaluations. It also provides numerous useful links and resources.

My interest in random numbers was re-kindled recently while determining whether a particular neural network prediction was better than chance or any of several other prediction algorithms. I was using a Monte-Carlo simulation in Microsoft Visual Basic (tied into Microsoft Excel). To ensure that the code was working correctly, I selected several outcomes for which I could compute the probability of occurrences and associated variance. With 1,000,000 trials, I expected three hits, yet I did not receive one. Even after repeating the experiment several more times, I never obtained the expected outcome, which was highly improbable.

Out of frustration, I modified the output of the Visual Basic random number generator by shuffling the numbers using a Bays-Durham Shuffle (a method for shuffling the order of the output of a random number generator as shown in figure 1). This reduced the correlation between successively generated numbers that interacted with my code, and I started approaching the expected number of hits (0-5 hits per run of 1,000,000,000 trials). Later testing verified that the Visual Basic RNG fails several common tests for random numbers. However, as discussed later, when the Visual Basic RNG is used in conjunction with a Bays-Durham shuffle, it passes more of the tests. This experience led me to start collecting RNGs, and methods for testing them. I am currently developing a software package that provides access to over 250 commonly used RNGs, and 13,000 lesser-known RNGs and variants. This software will be available shortly at *www.extremeet.com*.

Strange Encounters of the Random Kind

In one of my first neural network programs, a back-propagation algorithm to solve the exclusive-or problem, I employed the rand() function supplied with the Microsoft “C” compiler (circa 1986). It was also used to randomly select one of four training vectors (index = rand() & 3). I observed that the time to convergence was highly dependent on the initial random number seed. Working with larger networks and data sets, I noticed some training vectors were almost never selected, and that convergence became highly dependent on the initial random seed. Stanford students

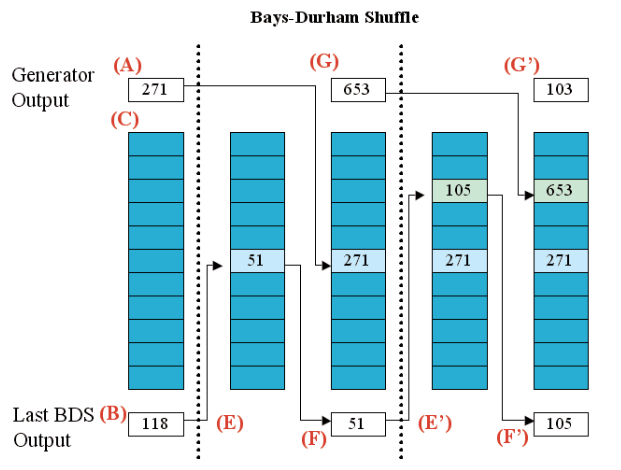


Figure 1 – Two cycles of a Bays-Durham Shuffle. Internal state consists of the last output of the Random Number Generator (A), the last output of the Bays-Durham Shuffle (B), and a table of random values (C). To generate a new output, an index (E) is created from the last output (B). The value in the table at this index becomes the next output (F). The value at the index is replaced with the current output of the generator (G), and the generator is updated to its next state.

Figure 1

once held competitions to see who could pick the random number seed that enabled their network to converge with the fewest iterations. The initial randomization of a neural network, and the random method employed for selecting training vectors has a major impact on convergence.

As I became interested in Genetic Algorithms (circa 1991), I became sophisticated enough to use Knuth's Lagged Fibonacci RNG as found in Press's "*Numerical Recipes in C: The Art of Scientific Computing*" [1], page 283. In one financial application, the results were substantially worse when the chromosome dimension (the length of the longest lag in the Knuth generator) was 55. My Genetic Algorithm (GA) was sensitive to some regularity of the RNG output structure. Picking a bad seed (usually even) could cause cycling in the RNG. I discovered a number of individuals with similar experiences. It turns out that the nature (algorithm and approach) of a RNG has a major impact on the performance of the Genetic Algorithm with which it is used.

Why Are Random Numbers Important?

RNGs are used in training neural networks, genetic algorithms, secure communications, online gambling, option pricing, stochastic optimization, image processing, video games and a host of other applications. One article indicated that Microsoft employs random numbers to generate test vectors for its software. George Marsaglia's (See: Important people) early work in random numbers focused on the gaming industry. Monte-Carlo simulation, based on random numbers, identifies opportunities in the options markets and develops hedging portfolios. Random numbers are an integral part of secure communication contributing one-time pad encryption keys, padding, and the like.

What is a Good Random Number Generator?

An RNG, or more precisely a Pseudo-Random Number Generator (PRNG) is an algorithm that generates number sequences that are indistinguishable from truly random sources such as the number of atoms that decay each second in a block of radium. This has several interesting implications. First, it means that occasionally, unlikely event combinations will occur. Second, there is no discernable structure in the sequence.

The best RNG for a particular application is one that produces the best quality results where quality is "in the eye of the beholder." For crypto-graphic purposes, quality may be measured by the computational energy required to deduce the next or prior key, if that is even possible. For simulation studies, it may be how well all or a portion of the distribution of outcomes match what is expected, or the degree of coverage of the space (fractal dimension). For stochastic optimization, using a genetic algorithm, it may be the degree to which the processes stochastically-based elements enables searching the entire space. There are some basic minimal qualities to watch for. An article planned for the July issue of Advanced Technology For Developers (*www.advancedtechnologyfor developers.com*) will discuss, in greater detail, each test described below, including source code for performing the tests.

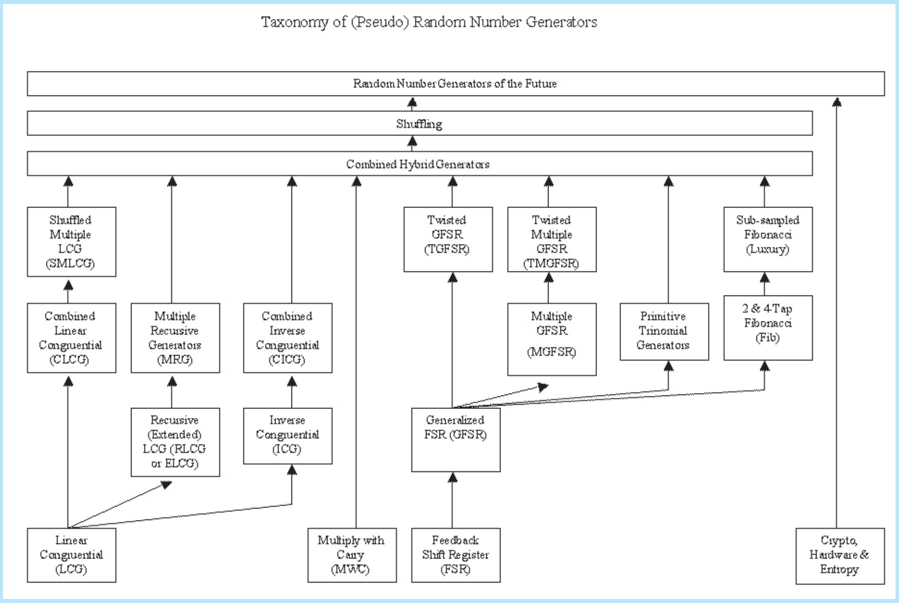
Another alternative to the tests described below is the Die Hard test suite by George Marsaglia accessible at: *http://stat.fsu.edu/~geo*.

Basic Statistical Tests: To even be considered, RNGs must pass these tests, which most easily do. The basic tests are stated in terms of real values in the range [0..1). For any RNG that produces integers, this transformation is: r = u / (1+max). Where u is the integer value, max the maximum value the generator takes, and r is the real uniformly distributed random number over the range 0 to 1. These limits, based on testing a block of 3,000,000,000

Some Terms Commonly and abbreviations commonly (miss-)used:

AWC	Add With Carry Lagged Fibonacci (or GFSR) generator combines terms by addition, including the prior carry.
Bays-Durham Shuffle	Method of post-processing the output of a RNG described by Knuth in "Numerical Recipes in C: The Art of Scientific Computing" [1] page 34.
CICG	Combined Inverse Congruential Generator.
CLCG	Combined Linear Congruential Generator. Combination of two or more LCGs.
Die Hard	Well-known and often referenced battery of tests for Random Number sequences developed by George Marsaglia.
DpRand	Double precision RNG by Nick Maclaren that combines a lagged Fibonacci generator similar to Knuth's with a Linear Congruential Generator.
ELCG	Extended Linear Congruential Generator. Also known as a Recursive Generator.
Entropy	Measure of a system's "randomness" often used when analyzing crypto-graphic quality generators.
FSR	Feedback Shift Register generator. Also called a Tausworthe generator or Linear Feedback Shift Register (LFSR) generator.
GFSR	Generalized version of the Feedback Shift Register, which includes Fibonacci generators.
ICG	Inverse Congruential Generator.
KISS	Combined hybrid generator developed by George Marsaglia.
LCG	Linear Congruential Generator.
Luxury	Sub-sampling method developed by Martin Luscher for improving structural problems with Marsaglia and Zaman's Subtract With Borrow lagged Fibonacci (GFSR) generator.
Mersenne Twister	Form of TGFSR developed by Matsumoto. Also known as mt19937 where the cycle length is 219937.
MGFSR	Multiple Generalized Feedback Shift Register generator.
Modulus	From a branch of mathematics known as "Finite Field Theory", many theoretical papers draw their conclusions from here. When computations are done "modulo the modulus", this is equivalent to taking the remainder after dividing by the modulus.
Mother	The "Mother-of-All" RNGs developed by George Marsaglia. At the time of its introduction, it had one of the longest period cycles of any generator proposed to date.
MRG	Multiple Recursive Generator.
MWC	Multiply with Carry generator.
Portable	Generator ported to another machines produces exactly the same results. In the strictest sense, this includes the ability to port to multiple parallel processor systems.
PRNG	Pseudo-Random Number Generator.
PTG	Primitive Trinomial Generator -- specialized sub-class of Generalized Feedback Shift Register generators.
RG	Recursive Generator -- A Linear Congruential Generator using multiple prior values as its state. Also called an Extended Linear Congruential Generator.
RNG	Random Number Generator. More correctly, Pseudo-Random Number Generator.
Seed	The starting value that initializes the state of a RNG consisting of a single number as in the case of LCGs, or more complex combined generators such as KISS.
SMLCG	Shuffled Multiple Linear Congruential Generator.
SWB	Subtract with Borrow lagged Fibonacci (or GFSR) generator that combines terms by subtracting them, and the prior borrow.
Tausworthe Generator	Also called a Feedback or Linear Feedback Shift Register Generator.
TGFSR	Twisted Generalized Feedback Shift Register generator. A GFSR with a post transform that "twists" the outputs.
TMGFSR	Twisted Multiple Generalized Feedback Shift Register. Multiple GFSRs with the Matsumoto "twist" or tempering transform.
Ultra	Hybrid lagged Fibonacci / Linear Congruential Generator by George Marsaglia.
Uniform RNG	RNG that produces numbers that uniformly cover everything between the generators minimum and maximum with equal probability.

Taxonomy of (Pseudo) Random Number Generators


Taxonomy of (Pseudo) Random Number Generators
Random Number Generators of the Future
Shuffling
Combined Hybrid Generators
Shuffled Multiple LCG (SMLCG)
Twisted GFSR (TGFSR)
Twisted Multiple GFSR (TMGFSR)
Sub-sampled Fibonacci (Luxury)
Combined Linear Congruential (CLCG)
Multiple Recursive Generators (MRG)
Combined Inverse Congruential (CICG)
Multiple GFSR (MGFSR)
Primitive Trinomial Generators
2 & 4-Tap Fibonacci (Fib)
Recursive (Extended) LCG (RLCG or ELCG)
Inverse Congruential (ICG)
Generalized FSR (GFSR)
Linear Congruential (LCG)
Multiply with Carry (MWC)
Feedback Shift Register (FSR)
Crypto, Hardware & Entropy

Linear Congruential Generator (LCG) – the simplest and unfortunately most widely employed RNG. LCGs are based on the relationship: $X_{t+1} = (A * X_t + B) \bmod M$. Substantial research has been done to find good values for A, B, and M such that the "cycle" length of the generator is (M-1). Sometimes, the result are shifted right to eliminate a portion of the low order bits. Example: Rand() function in Excel, Microsoft Visual C and most FORTRAN, C, C++, Java, Pascal compilers. A variant on the LCG wraps the prior carry into the next iteration.

Combined Linear Congruential Generator (CLCG) – combines two or more Linear Congruential Generators, usually by subtraction (exclusive OR is occasionally used). The modulus of each is selected so that they are different. With appropriate selection of A, B & M, the combined period is the product $(M_1-1)\ (M_2-1) \ldots$ An example is L'Ecuyer's CLCG4 [combines 4 LCGs] generator.

Shuffled Multiple LCG (SMLCG) – this variant on the Combined Linear Congruential Generator applies a Bays-Durham shuffle to one generators output prior to combining it with another generator. This tends to improve performance on more stringent statistical tests, producing a more random stream of results. Theoreticians shun this form of generator because the shuffle is very hard to analyze theoretically. Example: ran2 from [Press] page 282.

Recursive or Extended Linear Congruential Generators (RG or ELCG) – extend the recursion relationship to several prior states. An ELCG or RG is based on the relationship: $X_{t+1} = (A0 * X_t + A_1 * X_{t-1} * A_2 * X_{t-2} + \ldots + A_n * X_{t-n} + B) \bmod M$. Sometimes, as with Coveyou, the x's are multiplied together and there is only one coefficient. When multiplied, the result may be right-shifted (as with von Neumann's middle-square) to eliminate low order bits that tend to be less random. Examples of this are Marsaglia's Mother-of-all-RNG, Coveyou's Recursive Generator, Knuth's Recursive Generator ([Knuth], p. 108).

Multiple Recursive Generators (MRG) – combines two or more Recursive Generators by subtraction. Example: Rutger's MRGs.

Inverse Congruential Generators (ICG) – One flaws of LCGs for simulation is they exhibit a strong lattice structure. One approach for overcoming this is to take the "inverse" mod M of the current output. An Inverse Congruential Generator is based on the relationship: $X_{t+1} = (A * \mathrm{Inv}[X_t] + B) \bmod M$, where $(\mathrm{Inv}[z] * z) \bmod M = 1$. These tend to be computationally intensive, and do not fair well in empirical tests. The pLab project (Peter Hellekalek) has several examples of ICGs.

Combined Inverse Congruential Generators (CICG) –improving cycle lengths and statistical properties by combining multiple generators together. One example is from Peter Hellekalek, "Inversive Pseudorandom Number Generators: Concepts, Results and Links" at *http://random.mat.sbg.ac.at/generators/wsc95/inversive/inversive.html.*

Multiply with Carry (MWC) - Another class of generators is the Multiply with Carry which uses the recursion relationship: $X_{t+1} = (A * X_t + C_t) \bmod M$, $C_{t+1} = \mathrm{int}((A*X_t + C_t) / M)$. With properly chosen A and M, this generator has a period of A * M. (The "Die Hard" package (Marsaglia) contains a paper which describes the properties of this generator.)

Feedback Shift Register (FSR or LFSR) (Tausworthe) Generators – take a bit stream (typically represented by a 32-bit or 64-bit unsigned integer), treat it as a binary vector, and multiply it by a matrix (modulo 2) specifically selected to maximize the period. This transform, actually done through a series of shifts, masks, and exclusive Or's, is the basis for several crypto-graphic algorithms.

Generalized Feedback Shift Register (GFSR) – The FSR, often considered a linear recurrence on a polynomial, can expand out to multiple words. To work properly, it requires special initialization. From an implementation standpoint, it looks similar to a Fibonacci generator.

Twisted GFSR – "twists" the generators output by multiplying it by a matrix. The most famous example is Matsumoto's "twister" (recently revised). (See Makoto Matsumoto under "People".)

Multiple GFSR (MGFSR) – as with LCG's, multiple GFSRs or LFSRs can be combined to produce longer periods and better statistical properties. Example: taus88 by L'Ecuyer which combines three Tausworthe (FSR) generators together for a combined period of $(2^{31} - 1).(2^{29} - 1).(2^{28} - 1)$. Another example is L'Ecuyer's lfsr113 and lfsr258. (See Pierre L'Ecuyer under "People".) An interesting variant on this is Agner Fog's Ran-Rot generators. These perform a bit rotation on each component prior to combining them together. (See Agner Fog under "People").

Twisted Multiple Generalized Feedback Shift Registers (TMGFSR) – combines multiple GFSRs or LFSRs and "twists" the output. An example of a specially constructed LFSR with twisted output is L'Ecuyer's Poly96. (See Pierre L'Ecuyer under "People".)

Primitive Trinomial Generators (PTG) – Feedback Shift Registers, in a more general context, are Linear Congruential generators of polynomials (where each bit is raised to a power modulo 2). Richard Brent has evolved this into a real-valued generator with excellent statistical and empirical properties. (See Richard Brent under "People".)

2-Tap and 4-Tap Fibonacci Generators (Fib) – A special sub-class of GFSR generators that must be properly initialized for proper operation. Knuth's original RNG (for example, see [Press], page 283) is an example. Ziff's 4-tap generator ("Four-tap shift-register-sequence random-number generators" by Robert M. Ziff. *Computers in Physics*, Volume 12, Number 4, July/August 1998, pp 385f.). There is an implementation of this in the GNU Scientific Library.

Sub-sampled Fibonacci Generators (Luxury) – Fibonacci generators have flaws (regularities in their outputs - lattice structure). Martin Luscher's theoretical analysis of one class of lagged Fibonacci generators suggested by Marsaglia and Zaman, showed that it was highly chaotic in nature, with a strong lattice structure. With this insight, he suggested a method to eliminate the structural flaw by sampling the generator's output in blocks resulting in a modification to the Marsaglia and Zaman generator called "RanLux". It is quite popular in Monte-Carlo simulations in physics. An implementation by Loren P. Meissner is available at: *www.camk.edu.pl/~tomek/htmls.refs/ranlux.f90.html*

Crypto, Hardware, and Entropy-based Generators – Primarily targeted at establishing secure communication, crypto generators use an encryption algorithm (such as DES or TEA) to produce a stream of random numbers, feeding back the output as the next input. Hardware and Entropy-based generators us the inherent randomness of external events – disk interrupts, serial port interrupts, machine state at an interrupt, etc. – to create an "entropy pool". This pool is stirred and used as the basis for creating a random key (for one-time pad), pad-bytes, and so on. The goal is to producing a number that cannot be guessed or derived from looking at a sequence of outputs. Intel has created a "RNG chip" that samples noise on a chip component, transforms it, and creates a random bit stream. Intel has indicated it will integrate this into future chip-sets and mother boards. Examples include: the Pseudo-DES algorithm in [Press], page 300f, the TEA generator, and the entropy-based RNG in the Linux Kernel.

Combined Hybrid Generators – Using generators from two or more classes mitigates structural flaws. George Marsaglia, one of the pioneers of this empirical approach to improving the RNGs properties developed Combo (linear Congruential Generator with Coveyou's Recursive Multiplicative Generator), KISS (Linear Congruential, Feedback Shift Register, Recursive Generator), Super-Duper (Linear Congruential, Feedback Shift Register), Ultra (Lagged Fibonacci (GFSR), and Linear Congruential). Nick Maclaren's Dprand, combining a real-valued implementation of Knuth with a Linear Congruential Generator, improved the randomness of the low-order bits. (See Allan Miller's home page for an example of an implementation in Fortran-90.) From a theoretical perspective, Combined Hybrid Generators are difficult to analyze. Their popularity stems from empirical performance.

Shuffling – Method for breaking up serial correlation. The most famous method was suggested by Bays and Durham, known as the Bays-Durham Shuffle. [Press], page 279f, shows examples of the implementation, which is generally shunned by theorists due to difficulty with its analysis. On a practical side, it can improve the useful performance of bad generators. In the case of very good generators, it occasionally degrades the randomness of the resulting sequence.

random numbers collected from over 500 generators, are:

Average of 0.5 +/- 0.0004
Skew of 0 +/- 0.001
Serial Correlation of $< 10^{-7}$

If the generator cannot pass these tests, it does not meet the most basic requirements for random.

Specialized Statistical Tests: The entropy and Hurst exponent are two sophisticated tests that measure how the RNG performs. Entropy measures the degree to which the bits in the RNG, shifted through an 8-bit window, are uniformly distributed. Failing this test indicates that the generator only covers a limited number of bit-patterns. The Hurst exponent measures the degree of persistence or anti-persistence in trends. It is related to the relationship from Brownian Motion where the distance of a particle from its origin (d) as a function of time (T) is: d is proportional to $T^{1/2}$. A Hurst exponent of 0.5 indicates a random sequence; less than 0.5 is anti-persistent (having a tendency to reverse); greater than 0.5 is persistent. A very good discussion of the Hurst Exponent and how to calculate it can be found in Peters "Fractal Market Analysis", pp 54[2].

Entropy > 7.99
Hurst 0.5 +/- 0.03

Basic Simulation-Based Tests: These tests simulate a particular system, where the outcome is known, and the deviation from what is expected is measured. When testing long cycle generators, it is possible for an RNG to fail a test on part of the sequence and pass it on another. For this reason, failure is defined as a catastrophic failure, where the likelihood of the outcome is so small that it should virtually never occur. Your RNG should pass five simulation-based tests:

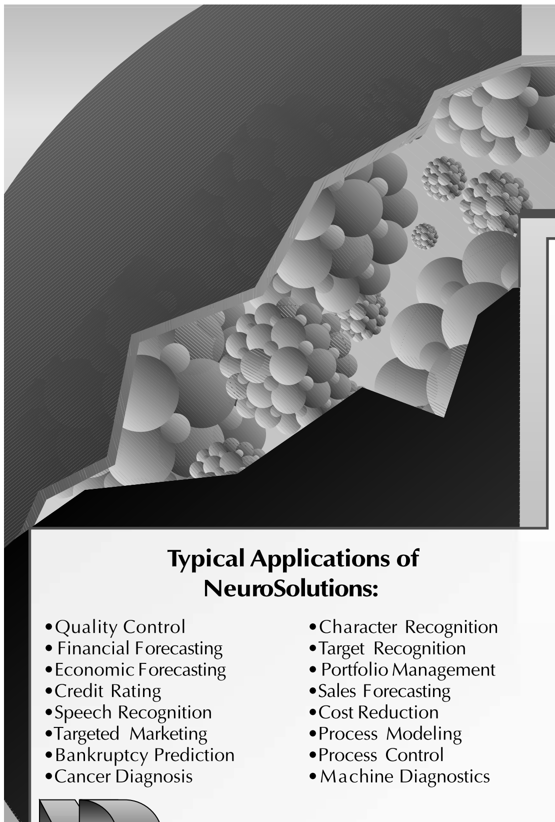
- Monte-Carlo estimate of PI,
- Simplified Poker Test,
- 1 & 2-dimensional Collision Test,
- 1-, 2-, and 3-dimensional Birthday Spacing test, and
- Roulette Test.

Several of these tests are described in detail by Knuth in "*The Art of Computer Programming: Volume 2 – Seminumerical Algorithms*", pp 61[3]. The specific variants used for the Collision Test and Birthday Spacing tests come from Pierre L'Ecuyer, "Software for Uniform Random Number Generation: Distinguishing the Good from the Bad". (Visit the website: *www.iro.umontreal.ca/~lecuyer/papers.html* and search for: "wsc01rng.pdf" to obtain a copy of the paper.)

Monte-Carlo estimate of PI: The area of a circle is PI * r^2. (PI = 3.14159265358979…) We can test the effectiveness of a RNG by using it to estimate PI. Take the last two numbers generated in sequence (Xt, Xt-1), convert them to real values between zero and one (Rt, Rt-1), and count (Mc) whether the point lies inside the unit circle. (i.e.: (Rt^2 + $Rt\text{-}1^2$) < 1). This is illustrated in figure 2. Since we are only dealing with one quadrant, the actual value of PI is 4 * Mc / T, where T = the number of trials. This approximates PI with most generators testing within +/- 0.01 of PI.

Simplified Poker Test: In this test, we deal five cards from either a fixed or an infinite deck. In each deal of five cards, we look for four different poker hands: 4-of-a-kind, 5-of-a-kind, and 4-aces. In the infinite deck, the next five Random Numbers specify the five cards. In the fixed 54-card deck, the next 54 random numbers become the next shuffle of the fixed deck. A block of 10,000 hands constitutes a run that repeats for 10 runs. After computing Chi-squared, the test compares the distribution of the 10 runs against the expected distribution. A p-value is computed and the test fails if the p-value > 0.85.

Collision Test: Also called the Hash Test, the basic concept is the construction of



NeuroSolutions™
The Premier Neural Network Development Environment

Version 4.0 Now Available!

NeuroSolutions provides the most powerful and flexible development environment for neural network design and application. **Download** a FREE evaluation copy from **www.nd.com** to see for yourself why thousands of neural network application developers and researchers have selected NeuroSolutions as their tool of choice.

The NeuroSolutions Advantage
- Intuitive wizards simplify the model selection and creation tasks
- Choose from a comprehensive set of the latest neural architectures for improved model performance. Some examples include:
  ◦ MLP ◦ RBF ◦ SOM ◦ TDNN
  ◦ Fuzzy ◦ SVM ◦ PCA ◦ Recurrent
- Powerful genetic optimization takes the guess work out of setting parameters
- Flexible interface allows you to create your own custom components and neural architectures
- Quickly deploy your solution to a custom application and pay no licensing fees

NeuroSolutions Add-ons
- NeuroSolutions for Excel® – Easily create neural nets directly from Excel
- Custom Solution Wizard – Automatically create a custom neural net app

Other NeuroDimension Products/Services
- Trading Solutions – Use neural nets to increase your trading profits
- Genetic Server/Library – Embed GAs into your VB or C++ apps
- Interactive Book – Teach yourself neural net theory using over 200 live examples
- Course – Learn to apply neural nets through hands-on training

Special Offer Save $100
Use Offer Code 8977

Typical Applications of NeuroSolutions:

- Quality Control
- Financial Forecasting
- Economic Forecasting
- Credit Rating
- Speech Recognition
- Targeted Marketing
- Bankruptcy Prediction
- Cancer Diagnosis
- Character Recognition
- Target Recognition
- Portfolio Management
- Sales Forecasting
- Cost Reduction
- Process Modeling
- Process Control
- Machine Diagnostics

NeuroDimension • 1800 North Main Street • Gainesville, Florida 32609 • (352) 377-5144
FAX (352) 377-9009 • 1-800-ND-IDEAS • (1-800-634-3327) • info@nd.com • http://www.nd.com

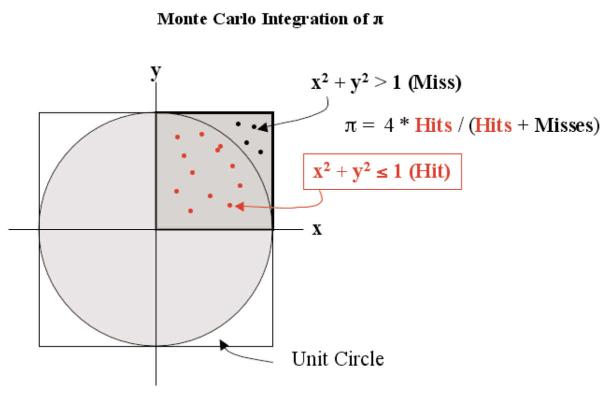
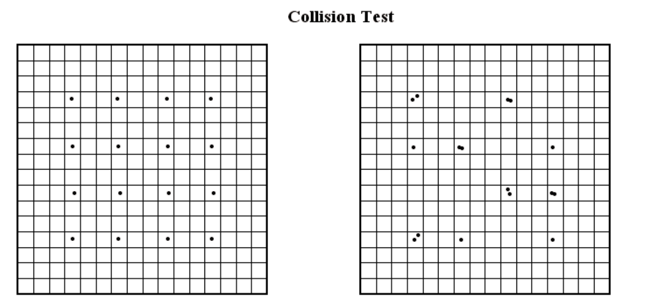


Figure 2 – The are of a circle is π r². For the unit circle, this is just π. We can estimate this by taking successively generated random numbers, scaling them into the range zero to one, and comparing the sum of their squares to 1. If less than 1, they are inside the circle. We count the number of trials and the number of "hits" inside the circle. The ratio of hits to total trials is _ π (since we are only doing the integration over a single quadrant). Monte Carlo simulation is one of the common ways to test random number generators.

Figure 2



3 a – Failure: Regular lattice results in NO collisions. Example: Visual Basic Rand() function.

3 b – Failure: Limited number of possible bit patterns. This often occurs in low-order bits. This can be an artifact of multiplying two numbers together. The low order bits take on only a subset of all possible values.

Figure 3 – The collision test measures structure in the random sequence. Given a certain size of hash table or incidence matrix, a purely random process will create a certain number of collisions. Too few or too many indicate structure in the sequence. In particular, the collision test shows up two defects: Strong lattice structure, and short cycles, particularly in the low order bits. These are illustrated above.

Figure 3

an enormous hash table ($k = 2^{24}$). The test generates random numbers and computes an index into this hash table. If the index comes from a uniformly distributed random number, we compute the expected probability colliding with another prior hash. We select a sample size ($n = 2^{16}$) much smaller than the hash table size. The expected number of collisions for uniformly distributed random numbers is: $L = n^2 / (2k)$. With too few collisions or too many collisions, we know there is substantial structure in the sequence we have selected (figure 3). Per design, the tests performed below each produce 128 collisions. A few RNGs, such as the one included with Visual Basic, fails because they have no collisions in the 1-dimensional test, and a large number of collisions in lower granularity in the 2-dimensional test. Others, such as Excel, fail the 2-dimensional test with too many collisions at higher resolutions. RandU, the original RNG in the IBM FORTRAN library fails both tests spectacularly. Going back to the story earlier in this article, the Visual Basic RNG failure shows up in the simulation test. Adding a Bays-Durham shuffle to this RNG enables it to pass the 2-dimensional collision test.

In the one-dimensional collision test, a single random variable between [0..1) is divided into 2^d bins. This is equivalent to using the high order d bits of the random number as a bin number. Initially, with all bins set to zero, the test generates much smaller 2^n random numbers with each converted into a bin index. The test determines the bin index for each number generated, sets the bin to one, and it counts the number of collisions. This particular test is designed so that the expected number of collisions is 128. Many hits indicate short-cycles in the RNG, while far fewer hits indicate a very strong lattice structure -- both are undesirable. The test is repeated for d = 22, 24, 26, 28, 30 bits and the corresponding 2^n ($n = d/2+4$) samples. The expected number of collisions for each test is 128 and each test repeats 5 times. By computing Chi-square, we compare the average number of collisions to the expected number. A p-value > 0.99 indicates test failure (generally catastrophic). Many of the better RNGs generate p-values < 0.10.

Important People in Random Numbers

Richard Brent – Professor of Computing Science, Oxford University, is a leader in the search for good RNGs based on primitive trinomials. The resulting RNGs have both good theoretical as well as empirical properties. Follow the link to "research interests", then "Random Numbers". Papers describing the theory and source code (GNU license) are available. *http://web.comlab.ox.ac.uk/oucl/people/richard.brent.html.*

Paul Coddington – A Senior Lecturer in the Computer Science department at the University of Adelaide. Paul's focus is on parallel high-performance computing, with an interest in simulation. He has written numerous papers on parallel RNGs and testing RNGs. *www.cs.adelaide.edu.au/~paulc*

Luc Devroye – A Professor at McGill University in Montreal has written several papers, and collected several resources on Random Numbers. Look at "Random Number Generation" under "Research Interests" at: *http://cgm.cs.mcgill.ca/~luc*. At the bottom of the page, you will find links to several RNGs.

Agner Fog – The inventor and proponent of chaotic RNGs with random cycle lengths (ranrot generators). Go to: *www.agner.org* and follow the links to "Random Numbers". You will find source code and papers on the theory behind these generators. One of his key insights is monitoring the initial state of the generator, and when a "cycle" occurs, reset the generator. This creates a random cycle length.

Peter Hellekalek – An assistant Professor at the Institute of Mathematics at the University of Salzburg. As leader of the pLab project (A Server on the Theory and Practice of Random Number Generation), he has an excellent "news" page with the latest in news and insights into Random Numbers. His home page is: *http://random.mat.sbg.ac.at/team/peter.html*. Follow the links to pLab for more information about different types of RNGs and tests. The news page is: *http://crypto.mat.sbg.ac.at/news* and it also includes links to several other pages of interest.

Donald E. Knuth – The Renaissance man of computer science, and author of the Art of Computer Programming [Knuth] has developed a theoretically founded RNG (significantly improved since the original). Visit: *www-cs-faculty.stanford.edu/~knuth*, and follow the links to "Downloadable Programs". At the bottom of that page are links to access the source code for RanArray.

Pierre L'Ecuyer – A Professor at the University of Montreal is one of the more prominent and prolific researchers on Random Numbers. Visit his web page at: *www.iro.umontreal.ca/~lecuyer*. Follow the "Software" link. This has further links to a number of papers and software for creating and testing RNGs. Professor L'Ecuyer created a number of popular Linear Congruential Generators, Generalized Feedback Shift Register Generators, and special purpose generators for parallel computation and simulation. If you only look in one place, Pierre L'Ecuyer's web site is the place to go.

Martin Luscher – A Researcher at DESY/University in Hamburg. He wrote a seminal report (DESY 93-133), entitled: "A Portable High-Quality RNG for Lattice Field Theory Simulations". In this paper, he does an analysis of Marsaglia and Zaman's subtract with borrow fibonnaci generator, and using insights from theoretical physics demonstrates its firm grounding in chaos theory, and shows how the structural bias can be removed by sampling. The result of this is the "RanLux" generator. The idea of eliminating a portion of a random number sequence was originally proposed by Todd & Taussky in 1956. Luscher gave this a firm theoretical footing. These insights are also used by Knuth in his most recent RNG. RanLux source code is available at: *www.camk.edu.pl/~tomek/htmls.refs/ranlux.f.html.*

George Marsaglia – Recently retired from Florida State University, he created a number of "empirically-based" RNGs. He was a pioneer in mixing different generators types together to create composite generators with improved properties. He is also the author of the "Die Hard" suite for testing random numbers. His home page is: *http://stat.fsu.edu/~geo.*

Makoto Matsumoto – A Professor at the Department of Mathematics at Keio University, invented the Mersenne Twister, a very popular RNG. His home page (including links to the Mersenne Twister home page) is: *www.math.keio.ac.jp/~matumoto/eindex.html.*

Alan J. Miller - Honorary Research Fellow of CSIRO Mathematical & Information Sciences. He has translated several of the RNG algorithms into FORTRAN. Search his home page for "random": *http://members.ozemail.com.au/~milleraj.*

Harald Niederreiter – A Professor at the National University of Singapore, he is the primary proponent of Multiple Recursive Matrix methods. His home page is: *http://www.math.nus.edu.sg/~nied.*

The two dimensional spacing test uses the last two samples (non-overlapping) as x- and y- coordinates into a 2-dimensional cube, each axis divided into 2^d bins. This test is repeated for d = 11, 12, 13, 14, 15 bits (from each of two consecutive samples for a total of 22, 24, 26, 28, 30 bits) and the corresponding 2^n (n = d+4) samples. The test computes a Chi-square statistic along with a corresponding p-value. A p-value > 0.99 indicates test failure (generally catastrophic). Many good RNGs produce p-values < 0.10. It is important to remember, particularly when dealing with RNGs with long cycle lengths (much greater than 2^{64}), even a good generator will occasionally fail. Or, in the words of George Marsaglia, "p happens!". What we are looking for is catastrophic failures where the results are off the charts.

Birthday Spacings Test: We pick a year with k-days, and some number of people (n), and sort their birthdays into sequence. After computing the number of days between each birthday, and sorting these into sequence, we look at the distribution of spacings and use Chi-squared to compare them to the expected distribution (Poisson). An alternative test (the one used here) is to select k very large and n much smaller so that the number of expected collisions between spacings is approximately 1 (or two). This is illustrated in figure 4. An RNG fails (catastrophically) if the number of spacing collisions is much larger than expected (p > 0.99).

As with the collision test, we do 1, 2 and 3-dimensional tests, and a 3-dimensional test throwing away the high order 10-bits (testing low order bits). The expected number of collisions is Poisson distributed with mean: $L = N^3 / (4 * K)$, where N = the number of samples, and K = the number of days in a year.

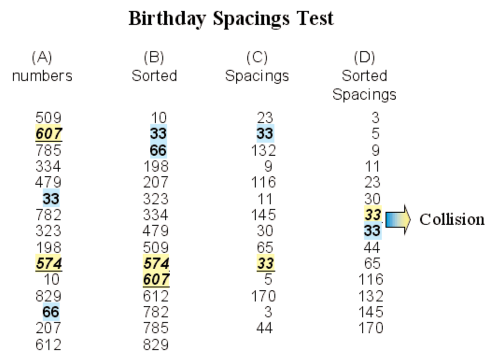
Roulette Test: This test specifically affects the mechanism found in a Genetic Algorithm's roulette wheel parent selection. The test assumes 10 genes with scores of 1, 1/(1+2), 1/(1+4), … , 1/(1+2*9) respectively. Associated with each score, we compute the probability of selecting any one of the bins. After repeating this for 100,000 spins, we compare the distribution of the outcomes to the expected distribution using Chi-square. Good generators typically give p-values of < 0.2 while a p-value of greater > 0.8 indicates a failure.

Advanced Simulation-Based Tests: George Marsaglia, now retired from Florida state University, developed a much more stringent series of simulation-based tests known as the "Die Hard" tests (after the battery). They became the standard by which RNGs are tested and ranked. Anyone with a new RNG must pass the Die Hard tests. A copy of the source code and executable for the Die Hard test is available at: *http://stat.fsu.edu/~geo*.

Even the Die Hard tests are dated and more stringent and sophisticated tests targeted at specific classes of applications, particularly in the simulation area, are under development. Pierre L'Ecuyer is developing such a test suite.

Application Specific Requirements: Applications may have their own specific requirements. For example, electronic slot machines require that the RNG include a strong "entropy" component that prevents predicting what the next outcome will be, and yet must require very little memory on a processor with minimal computational capabilities. Someone pushing a button on the slot machine, generates a random number. Using a relatively simple RNG (like KISS), and letting it run continuously, accomplishes this. Crypto-graphic applications, on the other hand, require the ability to create a large number of streams, each with a very long cycle time. Simulation often requires that for any number of samples, the RNG sequence is "space filling" (high fractal dimension) in high dimensions.

Other application specific requirements include memory (some RNGs require megabytes of memory), execution time, creating multiple streams (independent and unique sequences determined by an initial "seed" or state), cycle length of each stream, creating the same sequence on any number of independent parallel processes, the ability to "leap" forward to a new starting point, and portability across multiple hardware platforms and compilers.


Birthday Spacings Test

(A) numbers	(B) Sorted	(C) Spacings	(D) Sorted Spacings
509	10	23	3
607	**33**	**33**	5
785	**66**	132	9
334	198	9	11
479	207	116	23
33	323	11	30
782	334	145	***33***
323	479	30	**33**
198	509	65	44
574	***574***	***33***	65
10	***607***	5	116
829	612	170	132
66	782	3	145
207	785	44	170
612	829		

Figure 4 – In this variant of the Birthday Spacings test, we start with a list of randomly generated numbers (A). These are sorted (B). Next, we take the difference between successive numbers (C). These are the birthday spacings. Finally, we sort the spacings (D) and look for collisions. The numbers highlighted in yellow and blue track a collision through this process. In a random sample, we would expect a specific number of collisions. Too few or many represents a second order structure.

Figure 4

Test Results

In one trial, a collection of over 500 generators was tested, using basic and specialized statistical tests and basic simulation tests. This resulted in one third (160) of the RNGs passing all tests. Again, it is important to note that this was done on a limited sample size (3,000,000,000), and only done once. In order to validate these results, the tests should be repeated multiple times with longer streams.

The generators were then ranked individually based on their performance in

Resources

"*Random Number Generation*", Chapter 4 of the Handbook on Simulation, Jerry Banks Ed., Wiley, 1998, 93--137. (Introductory tutorial on RNGs). This handbook won the best book award for an engineering-related handbook for 1998, by the Association of American Publishers. Look for "handsim.ps" on L'Ecuyer's publications page: *www.iro.umontreal.ca/~lecuyer/papers.html*. From my perspective, this is the single best overview of the field in one place.

eXtreme Emerging Technologies has a random number package supporting 250 commonly used RNGs, and over 13,000 variants. It also supplies a number of non-uniform distributions, and is callable from C, C++, Visual Basic, and Excel (through a supplied Visual Basic interface). They are located at: *http://www.eXtremeET.com*

The WWW Virtual Library: Random Numbers and Monte Carlo Methods at: *http://crypto.mat.sbg.ac.at/links* contains links to several other pages including both theory and source code. This is part of the pLab project led by Peter Hellekalek.

The Swarm User Guide: Resources for Random Number Generation provides links to code for several different RNGs in appendix "C.4": *www.santafe.edu/projects/swarm/ swarmdocs/userbook/swarm.random.sgml.appendix.html*

RNGs – Steve Park's old web-page (William and Mary College) that includes several generators and code for generating other distributions. *www.cs.wm.edu/~va/software /park/park.html*

The GNU Scientific Library contains a section that includes a number of RNGs structured to allow their transparent use. See: *http://sources.redhat.com/gsl*. For a description of the capabilities associated with Random Numbers, see: *http://sources.redhat.com/gsl/ref/gsl-ref_17.html*. Note that Linux itself has an interesting "hardware" RNG built into the kernel, which uses hardware interrupts and internal CPU state information to generate random numbers used for cryptographic purposes.

Glenn Rhoads' Home Page (PhD Student at Rutger's University). Check out the link to "Snippets" that takes you to a page with code snippets, including code for several RNGs. *http://remus.rutgers.edu/~rhoads.*

SourceBank has a fair collection of RNGs. See: *www.devx.com/sourcebank/directorybrowse.asp?dir_id=876*

Mathland has an interesting article and some great references that highlight the history of random numbers. Visit Ivars Peterson Mathland at: *www.maa.org/mathland/mathland_4_22.html*.

predicting PI, Hurst Exponent, 4-of-a-kind p-value, 5-of-a-kind p-value, Aces p-value, C1 p-value, C2 p-value, B1 p-value, B2 p-value, B3 p-value, and Roulette Wheel p-value. The top 20 generators in each category were averaged and re-ranked. Since they are based on a single test, these results cannot be considered definitive. Moreover, a generator that passes these tests still may not be the right one for an application.

What is interesting about these results is that the list contains two classic, well tested, empirically developed, generators by George Marsaglia (MotherXKissBDS and Ultra32), two theoretically derived and empirically validated generators by L'Ecuyer (LEcuyerMUBDS and Taus88BDS), and a theoretically derived and empirically improved generator by Matsumoto (TwisterBDS). Another interesting observation is that 16 of the top 20 generators used a post Bays Durham Shuffle.

A final observation and cautionary note: All of the generators commonly available and commonly used – IBM Fortran (RandU), Borland Pascal, Microsoft C, Excel, VB, Java – failed the tests! The list below shows the top 20 generators ranked as described above.

Fib33x13BDS Fib1279s418 Fib97x33BDS Fib63s31BDS Fib73a25BDS Fib7s5BDS Fib9689a5502 Fib33msc13BDS	Fib_rr_op_ss_BDS: Fibonacci generator. rr is the long lag, ss is the short lag. The op operator combines the two terms : x=xor, s=subtract, a=add, msc=multiply with carry, shift right 6. BDS indicates a Bays-Durham Shuffl e of the output.
MWC32o	32-bit Multiply with carry (A = 1893513180, M= 2^{32}).
MWC32d	32-bit Multiply with carry (A = 1965537969, M= 2^{32})
CasimirR7_5B D S	Consists of 7 each 16-bit Multiply with Carry generators wit h randomly selected multipliers and state reset, with generator selection based on a Bays-Durham Shuffle, sub-sampled based on a table of the first 5 odd primes, and a post Bays-Durham Shuffle.
MotherXKissBD S	The "Mother" Recursiv e Generator (Extended Linear Congruential Generator) Xor'd with the KISS RNG, and a pos t Bays-Durham Shuffle.
Rutgers10BDS	An LCG (A = 93167, M = 2^{32}-5)
LEcuyerMUBDS	A combined LCG developed by L'Ecuyer combining two LCG s (A=40014, M=2147483563) and (A=40692, M=2147483399) by Xor and performing a post Bays-Durham Shuffle.
TwisterBDS	Matusmoto's Twisted GFSR wit h a Bays-Durham Shuffle.
Ultra32	A lagged Fibonacci generator combined with a linear congruential generator invented by George Marsaglia.
RanLux1BDS	Martin Luscher's RanLux generator (sampled lagged fibonacci generator) with luxury level 1.
CICGMcBD S	A Combined Inverse Congruential Generator of Hellekalek, with a 16-bit Multiply with carry (A=30903, M= 2^{16}) generator, and a post Bays Durham Shuffle.
Taus88BDS	L'Ecuyer's multiple Generalized Feedback Shift Register with a post Bays Durham Shuffle.
CasimirM1933_7 BD S	1933 16-bit Multiply with Carry generators selected at random

Summary

Pseudo-Random Number Generators are an integral element of our everyday lives. For those of us who work with Neural Networks and Genetic Algorithms, they can have a substantive impact on our systems performance. A large number of high quality Random Number Generators – over 160 out of 500+ passed all of the tests described – are available. All of the commonly available RNGs – C, Visual Basic, Excel, Pascal, Java – failed these tests. A foolish builder builds a house on sand. Test and verify the randomness of your random number generator.

Casimir C. "Casey" Klimasauskas is President of Klimasauskas Group, a consulting firm that provides expert assistance and development in Neural Networks, Genetic Algorithms, and related emerging technologies. He can be reached at *klim@stargate.net* or visit his website at *www.klimasauskas.com*.

References

1. Press, William H., Teukolsky, Saul A., Vetterling, William T., Flannery, Brian R., Numerical Recipes in C: The Art of Scientific Computing, Second Edition, Cambridge University Press, 1988.
2. Peters, Edgar E., Fractal Market Analysis: Applying Chaos Theory to Investment & Economics, John Wiley & Sons, 1994.
3. Knuth, Donald E., The Art of Computer Programming: Volume 2 – Seminumerical Algorithms, Third Edition, Addison-Wesley, 1998.

STATISTICA
Data Miner enterprise system
The most comprehensive set of data mining tools on the market to:
✔ uncover hidden trends
✔ explain known patterns
✔ predict the future
STATISTICA
Neural Networks
also available
The most comprehensive NN application on the market
www.statsoft.com
StatSoft

Target Your Marketing to Professionals that Use your Products

Use a *PC AI* Web Banner Ad

Target your marketing to *PC AI* readers and website visitors.

PC AI readers are eager for the latest in intelligent agents, data mining, knowledge management, case based reasoning, fuzzy logic, intelligent applications, genetic algorithms, modeling and simulation decision support, e-commerce, and web technology.
Our readers are focused on using the latest technology to improve their bottom line.

PC AI

Quantities Limited

Back Issues Here

Buy any 6 issues for $30.00 Or $8.00/Issue (Foreign orders - add $4.00 for postage per copy)

or order online at www.pcai.com/pcai

Total amount enclosed $______________.

Send payment and coupon to:

PC AI Back Issue Request
POB 30130 Phoenix AZ 85046
Or Fax order to 602 971-2321
Or Call PC AI at 602 971-1869

Name ______________________
Address ____________________
____
City_________________ State ___
Zip ________________
Visa/MC/Amer#
____
Exp. Date ___________________
Signature____________________
Phone ______________________
E-mail ______________________

For a complete summary of the back issue contents visit: *http://www.pcai.com/web/issues/back_issue_summary.html*
Check out PC AI's 15 year cumulative index at *http://www.pcai.com/web/indexes/Cumulative_index.html*

Issue Theme (a sample of other topics)

1995

9 #1 Intelligent Tools
9 #2 Fuzzy Logic / Neural Networks
9 #3 Object Oriented Development (OOD)
9 #4 Knowledge-Based Systems
9 #5 AI Languages
9 #6 Business Applications

1996

10 #1 Intelligent Applications
10 #2 Object Oriented Development
10 #3 Neural Networks / Fuzzy Logic
10 #4 Knowledge-Based Systems
10 #5 Genetic Algorithm and Modeling
10 #6 Business Applications SOLD OUT

1997

11 #1 Intelligent Applications (Intelligent Web Search Engines)
11 #2 Object Oriented Development (Expert Systems on the Web)
11 #3 Neural Nets / Fuzzy Logic (Expert Systems)
11 #4 Knowledge-Based Systems (Data Mining)
11 #5 Data-Mining and Genetic Algorithm (Expert Systems)
11 #6 Business Applications (Neural Networks)

1998

12 #1 Intelligent Tools and Languages (Automated Agents)
12 #2 Object Oriented Development (Java Based AI)
12 #3 Neural Nets / Fuzzy Logic (Modeling)
12 #4 Knowledge-Based Systems (Modeling Methodology)

Issue Theme (a sample of other topics)

12 #5 Data Mining and Discovery (Knowledge Management)
12 #6 Business Applications (Neural Networks)

1999

13 #1 Intelligent Tools and Languages (Knowledge Verification)
13 #2 Rule and Object Oriented Development (Data Mining)
13 #3 Neural Nets & Fuzzy Logic (Searching)
13 #4 Knowledge-Based Systems (Fuzzy Logic)
13 #5 Data Mining (Simulation and Modeling)
13 #6 Business Applications (Machine Learning)

2000

14 #1 Intelligent Applications
14 #2 Intelligent Web Applications & Object Oriented Development
14 #3 Intelligent Web Portals, Neural Networks and Fuzzy Logic
14 #4 Knowledge Management, Expert Systems, Intelligent E-Business
14 #5 Data Mining, Modeling and Simulation, Genetic Algorithms

2001

15 #1 Intelligent Applications
15 #2 AI Web Apps, OOD, AI Lang
15 #3 Intelligent Business Rules & Fuzzy Logic (Petri Nets in Prolog, Knowledge for Sale)
15 #4 Knowledge Management and Decision Support (Brief History of AI)
15 #5 Data Mining, Modeling, Simulation and Analysis, Natural Language Processing
15 #6 AI to Combat Terrorism (Rule-Based Expert Systems, Hal - 2001, Multi-agent Network Planning)