Randomness of Rods on a Ring Systems

Will Dabney

August 30, 2006

Abstract

The generation of random numbers is an application of immediate importance for anyone who requires a secure means by which to generate cryptographic keys [Ellison, 1995]. The majority of non-hardware based methods for generating pseudo-random numbers have an unacceptably low level of randomness for cryptography. Thus, it is currently common practice to rely on hardware-based random number generators. The randomness in these hardware-based random number generators stems from the unpredictability of the underlying complex physical system, specifically random bits are harvested from a measurable property of the device [Davis et al., 1994]. We analyze an existing method for generating random numbers based on a model of a physical system, N-beads on a ring [Cooley and Newton, 2005], and show how the randomness of the system varies with the masses of the beads.

1 Introduction

Random numbers are numbers that are unpredictable. How unpredictable the numbers are is the measure of randomness. While there are many ways to measure randomness of a sequence of numbers, there are also theoretical definitions of randomness which cannot be measured. We will go into more details of the methods we use to measure randomness in this paper. However, the subject of measuring randomness reliably is a complex and highly discussed topic which this paper does not attempt to explore fully. Fortunately, others have already researched this aspect [Maurer, 1991][Soto, 1999].

To clarify our meaning when referring to different types of random numbers we use the terms *true-random number* and *pseudo-random number* [Ellison, 1995]. A true-random number is one which is impossible to guess regardless of the resources available; while a pseudo-random number is impractical to guess but can be guessed given sufficient resources. While the existence of true-random numbers can be argued against, we will use this term for a more relaxed definition of impossible such that it would be impossible to guess the numbers given modern technology. Another common use is for random bits, for which we say the probability of guessing the next bit correctly is no better than $1/2 + \epsilon$ for a suitably chosen ϵ which depends upon the method used [Ellison, 1995].

There are many methods that can be used to generate random numbers or harvest random bits [Ellison, 1995]. Generally these can be broken down into two categories. Hardware-based random number generators, such as those based on quantum effects in circuits [Jennewein et al., 2000] or air turbulence in disk-drives [Davis et al., 1994], make use of measurable physical properties of external systems to harvest random bits. Software-based random number generators, such as discrete log [Patel and Sundaram, 1998] or a mathematical model of a chaotic system [Cooley and Newton, 2005].

Hardware-based methods tend to have significant advantages in achieving a high degree of randomness. These methods also suffer from a host of problems such as being slow and the potential for undetected hardware failure, but newer approaches are improving on these problems [Davis et al., 1994]. Software-base random number generators are more likely to be classified as pseudo-random than true-random, but have the advantage of being capable of generating numbers much faster than their hardware counter-parts.

The problem for software-based pseudo-random number generators is that we must assume that the "adversary," the person attempting to guess the numbers, has possession of the software source-code and thus knows how the generator works. This presents a difficult problem, which any cryptographically strong software-based pseudo-random number generator must overcome.

Random numbers have a variety of uses, many of which do not depend on a high degree of randomness. These uses, for which the randomness required is very low, are not our concern. In those situations many efficient algorithms exist to provide low amounts of randomness quickly. Much more interesting are the applications for random numbers that have a high degree of randomness. These range from generating one-time pads to choosing port numbers for communication. In these situations it is very important for the user that a highly motivated adversary be unable to predict the next random number, even if they possess all previous numbers and access to the system. It is these requirements that make software based random number generators difficult.

The goal, in designing a pseudo-random number generator which meets these requirements, is to make it *infeasible* for an adversary to determine with any degree of certainty the next random number in the sequence. When the potential adversary is assumed to have knowledge of the system and previous random numbers, this is referred to as *cryptographically strong* [Aiello et al., 1998].

Some success has been achieved through the use of chaotic physical systems [Bernstein and Lieberman, 1990]. From this, a mathematical model of a chaotic physical system has been used for pseudo-random number generation [Cooley and Newton, 2004]. We follow on this method, using a mathematical model of a chaotic system to generate pseudo-random numbers, and continue by testing the randomness of this generator. We explore the question of whether the randomness from a model of a chaotic system will remain cryptographically strong. To measure randomness we use entropy [Shannon and Weaver, 1949], autocorrelation [Soto, 1999], and a universal statistical test for randomness [Shannon and Weaver, 1949].

The chaotic system we explore is that of N-beads on a frictionless ring [Glashow and Mittag, 1997] [Cooley and Newton, 2005]. Specifically we analyze the randomness of this system for $N = 3$. The chaotic properties of the system, and thus its randomness, come from the unpredictability of the iterative impacts of the beads without knowing the exact starting parameters. This raises the important question of whether the system is divergent or convergent for starting parameters which vary only slightly. We show that the system is divergent varying parameters.

The N-beads on a ring system has been studied previously [Cooley and Newton, 2005, Cooley and Newton, 2004, Glashow and Mittag, 1997], and for this paper we follow these approaches closely. Our contribution is in the analysis of the randomness properties of this system and how they are affected by starting parameters.

2 N-Beads On A Ring

Consider a frictionless two-dimensional ring of length x. There are N beads of mass m_0, m_1, …, m_N respectively. The beads have initial positions on the ring given by their arc-lengths: $x = \sum x_k$. These beads slide without friction on the ring, and have initial velocities v_0, v_1, …, v_N with the requirement that the total momentum is a constant $P = \sum m_k v_k$. Following [Glashow and Mittag, 1997], we assume that $P = 0$.

As the beads slide on the ring they will inevitably collide, which causes a simple exchange of momentum in which only two beads are affected. Triple collisions are treated a pair of double collisions, such as 1 and 2 collide followed immediately by 2 and 3. The equations for calculating the changing velocities of the masses are simple, and can be organized compactly to describe each potential collision as a matrix product [Cooley and Newton, 2004].

This system has also been shown to be equivalent to a single billiard inside of a triangle which collides with the sides of the triangle [Glashow and Mittag, 1997]. In this situation the billiards collision with a side of the triangle can be shown to be equivalent to a collision of two beads in the first system. We initially used both the classical formulation of the system and the billiard in a triangle system to generate collisions. Code for generating random bits using both systems is found in Appendices A and B. We found that we were able to generate collisions efficiently with either approach, and eventually settled on using the billiard in a triangle system.

3 Generating Random Numbers

How do we use this system to generate random numbers? Hey this is kind of based on temperature / heat measures...

4 Measuring Randomness

We use Shannon's entropy equation to calculate the entropy of the sequence of random numbers generated [Shannon and Weaver, 1949]. This is shown in Equation 1. To measure the entropy of the random numbers generated with different relative starting masses, we varied the m_1 from 10^{-10} to 5×10^3. We then measured the entropy of a sequence of 5000 numbers generated from

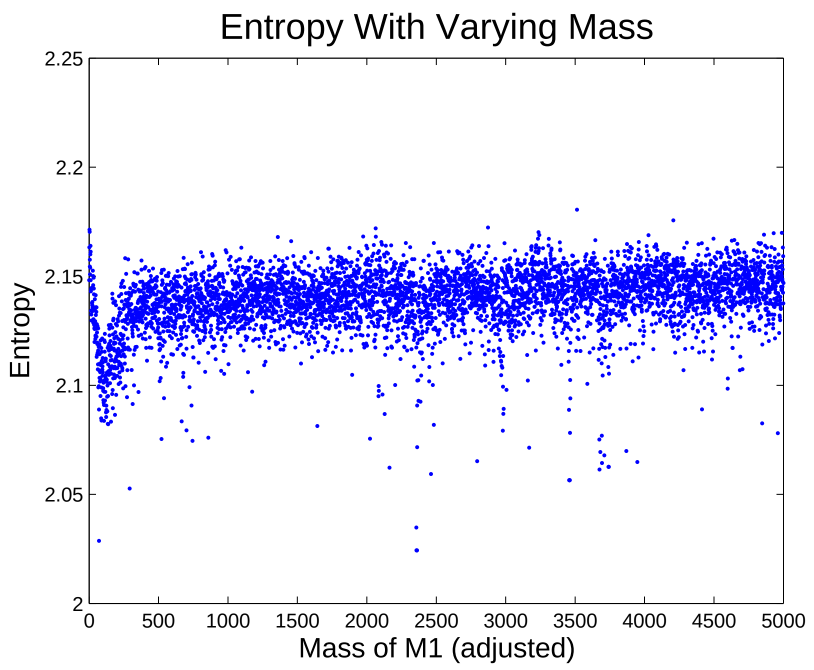

Figure 1: Entropy for random number sequences generated with billiard in triangle system.

these starting parameters. Figure 1 shows the resulting entropies for varying masses.

$$H = -\sum p_i \log p_i \tag{1}$$

How do we measure randomness... Really it's how do other people measure randomness. Now, how do we specifically use that to measure our randomness? Show the graphs for our measures. Give a brief evaluation of the randomness of the system, what does the graph immediately show?

5 Discussion

What does this mean? Really...?

Acknowledgements

This work was funded by the NSF......

APPENDIX

A Billiard in a Triangle Code (C)

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define acot(x) (atan(1.0 / (x)))

struct _collision {
  double theta;
  double sine_theta;
  double position;
  short int current_side;
  short int previous_side;
  short int other_side;
} collision;


double angle_sum[3];
double length[3];

double getAngleSum(int side1, int side2)
{
  // 01, 10 -> 0; 12 21 -> 1; 02 20 -> 2
  // 0 + 1 = 1, 1 + 2 = 3, 2 + 0 = 2 : 1 3 2 -> 1 2 3 -> 0 1 2
  // add then subtract one
  // 01,10 -> 0; 20,02 -> 1;  12,21 -> 2
  return angle_sum[side1 + side2 - 1];
}
```

```
collision generateCollision(collision base)
{
  collision result;

  result.theta = getAngleSum(base.current_side, base.other_side)
      - base.theta;
  result.sine_theta = sin(result.theta);
  result.position = length[base.other_side] -
      ((base.position * base.sine_theta) /
                result.sine_theta);

  if(result.position < 0)
  {
    double base_theta_prime = 180 - base.theta;
    double base_sine_theta_prime = sin(base_theta_prime);
    result.theta = getAngleSum(base.current_side, base.previous_side)
        - base.theta;
    result.sine_theta = sin(result.theta);
    result.position = length[base.previous_side] -
        (((length[base.current_side] -
        base.position) * base_sine_theta_prime) / result.sine_theta);

    result.current_side = base.previous_side;
    result.previous_side = base.current_side;
    result.other_side = base.other_side;
  }
  else
  {
    result.current_side = base.other_side;
    result.previous_side = base.current_side;
    result.other_side = base.previous_side;
  }

  return result;
}


int main(int argc, char *argv[])
```

```
{
  float mass1 = 0, mass2 = 0, mass3 = 0, numberCollisions = 0;
  float tri_length1 = 0.0, tri_length2 = 0.0, tri_length3 = 0.0;
  float angle1 = 0.0, angle2 = 0.0, angle3 = 0.0;
  float start_loc = 0.0;
  int start_side = 0;
  float start_angle = 0.0;
  float[3] angles;

  // rndGen mass1 mass2 mass3 numCollisions
  if(argc < 8)
  {
    printf("Usage: RandGen Mass1 Mass2 Mass3 X1 X2 X3 NumberCollissions\n");
    exit(0);
  }

  mass1 = atoi(argv[1]);
  mass2 = atoi(argv[2]);
  mass3 = atoi(argv[3]);
  x1 = atoi(argv[4]);
  x2 = atoi(argv[5]);
  x3 = atoi(argv[6]);
  numberCollisions = atoi(argv[7]);

  double M = mass1 + mass2 + mass3;
  double L = x1 + x2 + x3;
  double cPi = mass1 * mass2 * mass3;
  double factor = sqrt(cPi/M);

  angle[0] = acot(factor / mass1);
  angle[1] = acot(factor / mass2);
  angle[2] = acot(factor / mass3);

  length[0] = L * sqrt((mass1*(mass2 + mass3))/(M*M));
  length[1] = L * sqrt((mass2*(mass1 + mass3))/(M*M));
  length[2] = L * sqrt((mass3*(mass1 + mass2))/(M*M));

  angle_sum[0] = angles[1] + angles[2];
  angle_sum[1] = angles[0] + angles[2];
  angle_sum[2] = angles[0] + angles[1];

  return 0;
}
```

B N-Beads on a Ring Code (Matlab)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% matrixRndGen
% Arguments: numRods         The number of rods in the system
%            masses          An array containing the masses of the rods
%            arcLengths      An array containing the arc lengths of the rods
%            velocities      An array containing the absolute velocities
%            numCollisions   The number of collisions to simulate
%
% Returns:   An array, c, containing a list of all collisions.
%            Or, returns -1 if the total momentum of the system doesn't
%            equal zero.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [c d] = matrixRndGen (numRods, masses, arcLengths, velocities, numCollisi

% Ensure that the total momentum of the system is zero
if(sum(masses .* velocities) ~= 0)
    c=[-1];
    return;
end

% For our absolute positions on the ring:
location = cumsum([0; arcLengths(2:numRods)]);
ringLength = sum(arcLengths);
```

```
% Use a single array of all ones to calculate the constants
% This is an array of size numRods, with all ones.
unit_vector = zeros(numRods, 1) + 1;

% Calculate the constants used for adjusting velocities
% We form matrices containing the the correct values using
% Tensor products
m = unit_vector * masses' + masses * unit_vector'
mu = unit_vector * -masses' + masses * unit_vector'
mu_s = 2 * masses;

maxError = 0;
minError = 1000;
% Initialize the collisions list to zero
collisions = -1;
w = 1;
graph = location;
times = 0;
% Generate collisions
for i=1:numCollisions
    % Calculate the time till the next collision
    t = timeToCollision(velocities, arcLengths);
    % Remove times less than zero
    valid = t <= 0;
    t = t + (valid * realmax);

    % Choose the minimum, and keep the index (location) of it
    [t0, index] = min(t);

    % Update the arcLengths for all the rods
    % x_i' = x_i - t_0 * (v_i - v_i+1)
    % Using convolutions to do this quickly
    u = [-1; 1];
    vel = [velocities; velocities(1)];
    velDiff = conv(u, vel);
    velDiff = velDiff(2:numRods+1);
    minError = min(minError, abs(min(velDiff)));
    arcLengths = arcLengths - (t0 * velDiff);
```

```
        if(sum(arcLengths < 0) > 0)
            [o,p] = min(arcLengths);
            while(o < 0)
                arcLengths(p) = 0;
                [o,p] = min(arcLengths);
            end
        end

% Update positions on the ring
location = location + velocities .* t0;
% location = mod(location, ringLength);

graph = [graph location];
t1 = size(times);
times = [times; times(t1(1)) + t0];
% Now we should record the collision and change the velocities
% v' = M_ij * v
% Instead of using matrices here we can just update the velocities for
% the two colliding rods

% For multiple collisions occuring at the same time we must handle it
% by looking at arcLengths of 0, and velocity differences greater than
% 0, at the same rod.
s = sum((velDiff > 0) .* (arcLengths == 0)) > 0;
%if(s <= 0)
%    arcLengths
%    velDiff
%    t0
%    min(arcLengths)
%end
if(s <= 0)
    t0
end

while(s > 0)
    [y,j] = max((velDiff > 0) .* (arcLengths == 0));
```

```
            if(arcLengths(j) == 0)
                index = j;
                ip1_index = j + 1;
                if(ip1_index > numRods)
                    ip1_index = 1;
                end
                v_i = velocities(index);
                v_ip1 = velocities(ip1_index);
                velocities(index) = (v_i * mu(index, ip1_index) + v_ip1 * mu_s(ip1
                velocities(ip1_index) = (v_i * mu_s(index) + v_ip1 * mu(ip1_index,

                % index uniquely identifies the collision
                collisions = [collisions; index];

        end

        vel = [velocities; velocities(1)];
        velDiff = conv(u, vel);
        velDiff = velDiff(2:numRods+1);
        s =sum((velDiff > 0) .* (arcLengths == 0)) > 0;

    end
   maxError = max(maxError, abs(sum(masses .* velocities)));
  % graph = [graph arcLengths];
end
velocities
maxError
minError
c = graph;
d = times;
```

References

[Aiello et al., 1998] Aiello, W., Rajagopalan, S. R., and Venkatesan, R. (1998). Design of practical and provably good random number generators. *Journal of Algorithms*, 29.

[Bernstein and Lieberman, 1990] Bernstein, G. M. and Lieberman, M. A. (1990). Secure random number generation using chaotic circuits. *IEEE Transactions on Circuits and Systems*, 37.

[Cooley and Newton, 2004] Cooley, B. and Newton, P. K. (2004). Random number generation from chaotic impact collisions. *Regular and Chaotic Dynamics*, 9.

[Cooley and Newton, 2005] Cooley, B. and Newton, P. K. (2005). Iterated impact dynamics of n-beads on a ring. *SIAM Rev.*, 47(2):273–300.

[Davis et al., 1994] Davis, D., Ihaka, R., and Fenstermacher, P. (1994). Cryptographic randomness from air turbulence in disk drives. In *CRYPTO '94: Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, pages 114–120, London, UK. Springer-Verlag.

[Ellison, 1995] Ellison, C. (1995). P1363: Appendix e cryptographic random numbers. http://std.com/ cme/P1363/ranno.html.

[Glashow and Mittag, 1997] Glashow, S. L. and Mittag, L. (1997). Three rods on a ring and the triangular billiard. *Journal of Statistical Physics*, 87.

[Jennewein et al., 2000] Jennewein, T., Achleitner, U., Weihs, G., Weinfurter, H., and Zeilinger, A. (2000). A fast and compact quantum random number generator. *Review of Scientific Instruments*, 71:1675–1680.

[Maurer, 1991] Maurer, U. M. (1991). A universal statistical test for random bit generators. *Lecture Notes in Computer Science*, 537.

[Patel and Sundaram, 1998] Patel, S. and Sundaram, G. S. (1998). *An Efficient Discrete Log Pseudo Random Generator.* Springer.

[Shannon and Weaver, 1949] Shannon, C. E. and Weaver, W. (1949). *The Matematical Theory of Communications.* University of Illinois Press.

[Soto, 1999] Soto, J. (1999). Statistical testing of random number generators. In *Proceedings of the 22nd National Information Systems Security Conference.*