Modern pRNGs

A deterministic finite automata with $\mathcal{S}$ states (**state set**), with **transition** (f) and **output** functions (g),

$$f : \mathcal{S} \to \mathcal{S}$$

$$g : \mathcal{S} \to (0, 1)$$

and initial state (**seed**) of s_0.

The $i+1$-th state is $s_{i+1} = f(s_i)$ and the output is $u_{i+1} = g(s_{i+1})$.

The sequence of states is periodic, the period length (ρ) is the smallest $j > 0$ such that

$$s_{i+j} = s_i \qquad \text{for some} \quad i \geq 0$$

If k is the number of bits required to represent $\mathcal{S}$, then a **"well-designed"** pRNG has a period "near" 2^k

Modern pRNGs

f, g, and k are the important parts for this discussion. In terms of our Lehmer PMMLCG:

$$f(x_{i+1}) = a\,x_i \bmod m$$

$$g(x_i) = \frac{x_i}{m}$$

$$k = 32 \quad \rho = 2^{31} - 1$$

Linear Recurrence Generators

Arithmetic modulo some $m \geq 2$. State is a k dimensional vector

$$\vec{x}_i = \begin{bmatrix} x_{i,0} \\ x_{i,1} \\ \vdots \\ x_{i,k-1} \end{bmatrix} \qquad x_{i,*} \in Z_m = \{0, 1, 2, \ldots, m-1\}$$

transition function

$$\vec{x}_i = \boldsymbol{A}\vec{x}_{i-1} \bmod m$$

with $\boldsymbol{A}$ a $k \times k$ matrix, $a_{i,j} \in Z_m$.

If m is prime and $\boldsymbol{A}$ chosen correctly

$$\rho \approx m^k - 1$$

Linear Recurrence Generators

The output function g consists of two computations:

$$\vec{y}_i = \begin{bmatrix} y_{i,0} \\ \vdots \\ y_{i,w-1} \end{bmatrix} = \boldsymbol{B}\vec{x}_i \bmod m \qquad y_{i,*} \in Z_m = \{0, 1, 2, \ldots, m-1\}$$

$$u_i = \frac{1}{2m^w} + \sum_{t=1}^{w} \frac{y_{i,t-1}}{m^t}$$

where $\boldsymbol{B}$ is $w \times k$ matrix with $b_{i,j} \in Z_m$.

Linear Recurrence Generators

LRG example for

$$m = 7 \qquad w = 4 \qquad \vec{y}_i = \begin{bmatrix} 3 \\ 0 \\ 2 \\ 5 \end{bmatrix}$$

$$u_i = \frac{1}{2m^w} + \sum_{t=1}^{w} \frac{y_{i,t-1}}{m^t} = \frac{1}{2(7^4)} + \frac{3}{7^1} + \frac{0}{7^2} + \frac{2}{7^3} + \frac{5}{7^4} \approx 0.4366930$$

Linear Recurrence Generators

LRG example for

$$m = 7 \qquad w = 4 \qquad \vec{y}_i = \begin{bmatrix} 3 \\ 0 \\ 2 \\ 5 \end{bmatrix}$$

$$u_i = \frac{1}{2m^w} + \sum_{t=1}^{w} \frac{y_{i,t-1}}{m^t} = \frac{1}{2(7^4)} + \frac{3}{7^1} + \frac{0}{7^2} + \frac{2}{7^3} + \frac{5}{7^4} \approx 0.4366930$$

Suppose

$$\vec{y}_i = \vec{0} \ \ (\text{all } y_{i,*} = 0)$$

$$u_{min} = \frac{1}{2(7^4)} \approx 0.0002082466$$

the smallest possible u from this LRG pRNG.

Linear Recurrence Generators

LRG example for

$$m = 7 \qquad w = 4 \qquad \vec{y}_i = \begin{bmatrix} 3 \\ 0 \\ 2 \\ 5 \end{bmatrix}$$

$$u_i = \frac{1}{2m^w} + \sum_{t=1}^{w} \frac{y_{i,t-1}}{m^t} = \frac{1}{2(7^4)} + \frac{3}{7^1} + \frac{0}{7^2} + \frac{2}{7^3} + \frac{5}{7^4} \approx 0.4366930$$

Suppose

$$\vec{y}_i = \vec{0} \ \ (\text{all } y_{i,*} = 0)$$

$$u_{min} = \frac{1}{2(7^4)} \approx 0.0002082466$$

the smallest possible u from this LRG pRNG.

Suppose

$$\vec{y}_i = \vec{6} \ \ (\text{all } y_{i,*} = 6 = m-1)$$

$$u_{max} = \frac{1}{2(7^4)} + \frac{6}{7^1} + \frac{6}{7^2} + \frac{6}{7^3} + \frac{6}{7^4} \approx 0.9997918$$

the largest possible u from this LRG pRNG.

Linear Recurrence Generators

LRG example for

$$m = 7 \qquad w = 4 \qquad \vec{y}_i = \begin{bmatrix} 3 \\ 0 \\ 2 \\ 5 \end{bmatrix}$$

$\vec{y}_i = \vec{0}$ (all $y_{i,*} = 0$)

$$u_{min} = \frac{1}{2(7^4)} \approx 0.0002082466$$

the smallest possible u from this LRG pRNG.

$\vec{y}_i = \vec{6}$ (all $y_{i,*} = 6 = m - 1$)

$$u_{max} = \frac{1}{2(7^4)} + \frac{6}{7^1} + \frac{6}{7^2} + \frac{6}{7^3} + \frac{6}{7^4} \approx 0.9997918$$

the largest possible u from this LRG pRNG.

Notice that

$$1 \approx u_{min} + (u_{max} - u_{min}) + u_{min} \Rightarrow 1 \approx u_{max} + u_{min}$$

... the interval is perfectly centered in $(0, 1)$.

Linear Recurrence Generators

LRG example for

$$m = 2 \qquad w = 51 \qquad \vec{y}_i = \text{can store its 51 bits in a 64b integer}$$

$$\text{64b integer} = \underbrace{0100110\ldots0001011}_{\text{51 bits of } \vec{y}_i} \underbrace{1}_{\star} \overbrace{00\ldots0}^{\text{0 pad}}$$

$$\begin{aligned} u_i &= \frac{1}{2m^w} + \sum_{t=1}^{w} \frac{y_{i,t-1}}{m^t} \\ &= \sum_{t=1}^{51} \left\{ \frac{y_{i,t-1}}{2^t} \right\} + \frac{1}{2^{51+1}} \\ &= \frac{0}{2^1} + \frac{1}{2^2} + \frac{0}{2^3} + \ldots + \frac{0}{2^{49}} + \frac{0}{2^{50}} + \frac{1}{2^{51}} + \left(\frac{1}{2^{52}} \right)^{\star} \end{aligned}$$

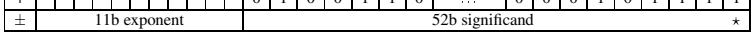
Linear Recurrence Generators

LRG example for

$$m = 2 \qquad w = 51 \qquad \vec{y}_i = \text{can store its 51 bits in a 64b integer}$$

$$\text{64b integer} = \underbrace{0100110\ldots0001011}_{\text{51 bits of } \vec{y}_i} \underbrace{1}_{\star} \overbrace{00\ldots0}^{\text{0 pad}}$$

These 52 bits can be bit-blasted into an IEEE-754 64b floating point representation's significand, which magically does all the $\frac{1}{2^r}$ math for us :)



Linear Recurrence Generators

The $\frac{1}{2m^w}$ is half the size of $\frac{1}{m^w}$, the **smallest** fractional part. $\frac{1}{2m^w}$ nudges u_i from $[0, 1)$ to $(0, 1)$.

Notice that the summation component is doing the (fractional) digit by digit arithmetic of a number ($0 < u_i < 1$) in a base m number system.

Linear Recurrence Generators $m = 2$

$m = 2$ represents some of our favorite modern LRGs:

1. Elements of $\boldsymbol{A}$ can be "packed" into big machine integers,
2. Arithmetic turns into bitwise operations on the big machine integers (everything is $\bmod 2$),
3. Big names: Linear Feedback Shift Register (LFSR), Generalized LFSR (GFSR), "twisted" GFSR, Mersenne "twister" and Shift-Register (in) Lookup Tables (LUT-SR) popular in FPGAs.
4. Some of the more popular ks:
   Mersenne twister (`MT19937`) has $k = 19937$
   WELL generators (`WELL`) is defined for multiple ks, up to $k = 44497$

Linear Recurrence Generators $m = 2$

$m = 2$ represents some of our favorite modern LRGs:

5 **Downside:** bigger ks mean more state space (**memory consumed for each stream**), and slower computations.

6 LRGs (it turns out Lehmer is a very simplified form) **ALL** fail statistical tests for bit-by-bit randomness so should never be used for cryptographic bit generators. **But we don't use the** x_i **like this** (recall $g : \mathcal{S} \to (0, 1)$) so they are quite nice for simulation.

7 It can be shown that two LRG (with $m = 2$) can have their $\vec{y}$ XOR'd together yielding an equally random sequence. The period (if constituent parts are chosen carefully) can be the product of the separate periods.

This approach has the potential for some parallelization and less computational effort determining initial stream "jump" factor. Two published examples have $\rho \approx 2^{113}$ and 2^{258}.

Linear Recurrence Generators $m > 2$

The traditional equation for $m > 2$ is

$$x_i = (a_1 x_{i-1} + a_2 x_{i-2} + \cdots + a_k x_{i-k}) \bmod m$$

The initial product can be written in standard form by using

$$\boldsymbol{A}\vec{x}_i = \begin{pmatrix} a_1 & 0 & \cdots & 0 \\ \vdots & a_2 & & \vdots \\ & \vdots & \ddots & \\ 0 & & \cdots & a_k \end{pmatrix} \begin{bmatrix} x_{i-1} \\ x_{i-2} \\ \vdots \\ x_{i-k} \end{bmatrix}$$

and $\bmod m$ is an element-by-element operation of each $\vec{x}_i$ component.

If $k = 1$ and $m = 2^{31} - 1$, $\boldsymbol{B} = \boldsymbol{I}_k$ we have our traditional Lehmer PMMLCG.

Streams in LRGs

Recall that jump multipliers for streams in Lehmer's PMMLCG could be calculated in a straight-forward manner.

If we want 14 streams, there will be $e = \left\lfloor \frac{\rho}{14} \right\rfloor$ elements in a stream with seed x_0, then stream $j = \{0, 1, 2, \ldots\}$ begins at

$$x_{j,0} = a^{je} x_0 \bmod m$$

Likewise for LRGs:

$$\vec{x}_{j,0} = (\boldsymbol{A}^{je} \bmod m)\vec{x}_0 \bmod m$$

Downside: the additional computational burden of such large states ($k = 19937$?!) and massive periods $\approx 2^k$ makes jumping ahead a long-winded task. If you need streams, "size" your pRNG correctly and then pay the (compute) price.

Non-Linear pRNGs

Two types:

- Combine two LRGs of different types ($m_1 = 2^{31-1}$, $m_2 = 2$)
- Make f, g, or both a non-linear function. eg: make f quadratic or cubic.

The Devil in the Details

For all of these generators (LRG($m = 2$), LRG($m > 2$), Non-linear LRGs):

a. care must be taken in choosing $\boldsymbol{A}$ and $\boldsymbol{B}$ and m to obtain large periods, and minimize computational overhead;
the final generator may still fail empirical statistical tests

b. the function f typically "does the work" of scrambling the values (bits when $m = 2$) around; **this is unfortunate** since it makes the application of streams computationally burdensome:

$$\vec{x}_{j,0} = (\boldsymbol{A}^{je} \bmod m)\vec{x}_0 \bmod m$$

Counter Based pRNGs

Recall our original conceptual framework: a deterministic finite automata with $\mathcal{S}$ states (**state set**), with **transition** (f) and **output** functions (g),

$$f : \mathcal{S} \to \mathcal{S} \qquad g : \mathcal{S} \to (0,1) \qquad \text{seed of } s_0$$

The $i+1^{\text{th}}$ state is $s_{i+1} = f(s_i)$ and the output is $u_{i+1} = g(s_{i+1})$.

A complete reversal of philosophy: since stream applications need an efficient f to jump the state forward, let's just use a counter value. The **state** at i becomes the counter value i.

$$f(i) = i \qquad \text{No more } x_{i-1}, \boldsymbol{A}, \text{ or } a \text{ !}$$

Now the burden is on g to mangle the bits of $x_i = i$ around.

Counter Based pRNGs

A complete reversal of philosophy: since stream applications need an efficient f to jump the state forward, let's just use a counter value. The **state** at i becomes the counter value i.

$$f(i) = i \qquad \text{No more } x_{i-1}, \boldsymbol{A}, \text{ or } a \text{ !}$$

Now the burden is on g to mangle the bits of $x_i = i$ around.

We choose a **cryptographic (or at least cryptographic-like)** hashing or encryption function $\mathcal{H}$ (such as AES, SHA, MD5, ...) and make it our generator for a q length sequence $\{y_{i,b}\}_{b=1}^{q}$ of bits.

Counter Based pRNGs

$$f(x_i) = \text{state of } x_{i-1} + 1 \qquad \text{It's a counter!}$$

$$f(x_i) = i$$

$\mathcal{H}$ is our old $\boldsymbol{B}$, the first step in the output function g (j is stream index, s_0 is the seed for generation) ...

$$\{y_{i,b}\}_{b=1}^{q} = \mathcal{H}(s_0 | \, j | \, i) \quad \mathcal{H} \text{ is a hash of } s_0, \, j, \text{ and } i \text{ concatenated together}$$

or

$$\{y_{i,b}\}_{b=1}^{q} = \mathcal{H}(s_0 | \, j, i) \quad \mathcal{H} \text{ is encryption with key } s_0 | \, j \text{ of counter } i$$

$$u_i = \frac{1}{2^{q+1}} + \sum_{b=1}^{q} \frac{y_{i,b}}{2^b}$$

Counter Based pRNGs

$$\{y_{i,b}\}_{b=1}^{q} = \mathcal{H}(s_0 | \, j | \, i) \quad \mathcal{H} \text{ is a hash of } s_0,\ j, \text{ and } i \text{ concatenated together}$$

or

$$\{y_{i,b}\}_{b=1}^{q} = \mathcal{H}(s_0 | \, j, i) \quad \mathcal{H} \text{ is encryption with key } s_0 | \, j \text{ of counter } i$$

$$u_i = \frac{1}{2^{q+1}} + \sum_{b=1}^{q} \frac{y_{i,b}}{2^b}$$

Compare the u_i calculation to that of LRGs, the same nudging factor is there and the m has turned into q. By choosing $q > 0$, we can dictate the resolution of our u_i on $(0, 1)$.

Counter Based pRNGs

Downsides: slower than LRGs, **unknown if reported tests use CPU supported features for hashes**.

New hashing and "encryption" functions specifically for this application are being investigated; I'm still looking for consistent (good) published results.

The potential of having **FAST** stream setup with virtually **arbitrarily sized periods** will be a huge boon for large scale simulation.