WORK DRAFT 'BREEZE - A CHAOS BASED PSEUDO RANDOM NUMBER GENERATOR BY HOPPING BETWEEN LOGISTIC MAP PSEUDO ORBITS'

ANDREAS BRIESE

1. Summary

Logistic maps (LM) expose a well studied phenomenom. The recursive call of the LM with

$$f(x_n) = \tau x_{n-1}(1 - x_{n-1}) \; with \; \{\tau \in R \mid (3.83, 4.0]\}; \{x \in R \mid (0, 1)\} \tag{1}$$

will get into chaotic state at $0 < x < 1$ and $3.56995 < \tau < 3.82843$ or $3.82843 < \tau <= 4.0$ and in a theoretically infinite space of real numbers the LM will conserve the chaotic state ad infinitum (elsewhere called an 'orbit of the LM'). Each orbit is unique and subsequently deterministic to the seed x_0. This makes LM an interesting candidate for deterministic Chaos Based Random Number Generation (CBPRNG).

Unfortunately in computational reality due to the double float IEEE754 representation and the resulting precision loss from rounding the single LM orbit might degrade rapidly and the period length of such a computed LM is limited in length and might even be very short (Li (2003, 2004), Arroyo (2009), Persohn and Povinelli (2012)).

The proposed 'breeze' CBPRNG uses a number of interacting LMs with $\{\tau \in R \mid (3.83, 4.0]\}$ to prevent orbit degradation. Random number output gained from the intermediate result mantissa in IEEE74 double float representation of the LM orbits repeatedly passed NIST Test Suite for randomness while the speed of the Go/Golang breeze implementation outclasses Go's standard library random implementations and implementations of the Multiply-with-Carry method and Salsa20 in Go.

See https://github.com/AndreasBriese/breeze for the detailed results and source code.

2. Coding chaos

2.1. **Chaos in logistic maps.** Logistic maps are well studied. They exhibit chaotic behavior if τ is set to values greater than 3.57 and smaller or equal to 4.0. At τ near 3.82 chaotic (up-and-down) output is suspended, but with τ greater 3.83 output is unpredictable and within the range of 0 to 1. Breeze uses $\{\tau \in R \mid (3.83, 4.0]\}$ therefore.

$$f(x_n) = \tau x_{n-1}(1 - x_{n-1}) \; with \; \{\tau \in R \mid (3.83, 4.0]\}; \{x \in R \mid (0, 1)\} \tag{2}$$

When exploring a single logistic map, they exhibit *orbits* of $x_n \in R$ determined by the starting x_1. In theory these orbits should be non-overlapping and expose an endless period but if translated to finite space of computational double float representation the *pseudo orbits* caused by rounding are overlapping and periods are limited in length and

Date: 2014/12/18.

might be very short (Li (2003, 2004), Arroyo (2009), Persohn and Povinelli (2012)). The starting points of such degradation leading to a short periods had been called *pathological seeds* by Persohn and Povinelli (2012).

To prevent the before mentioned degradation of pseudo orbits the new family of CBPRNG uses singular calculations of multiple logistic maps for random number output instead of exploring the *pseudo orbits*. These multiple maps are combined in such a way, that the outcome of one equation is used for the next calculation of another logistic map with $\tau_n \neq \tau_{n+1}$ after 'mirroring it at 1' by calculation $x' = 1 - x$. This results in 'hopping' between pseudo orbits of the logistic maps with any computation cycle. In finite space of double float computation this is turning both calculations into potential *'one-way-functions'* because of the rounding errors in IEEE double float:

$$x_{n-1} \cong \frac{x_n}{\tau(1-x_n)} \; with \; x_n = \tau x_{n-1}(1-x_{n-1}) \tag{3}$$

$$x \cong 1-(1-x) \; for \; small \; x \; (x \lll 0 \; and \; x \; < \; 2^{23} \; in \; particular) \tag{4}$$

The multiple logistic maps in breeze are:

$$breeze128(6 \; maps : f_{1..6} \; with \; \tau_{1..6})$$
$$f_i(x_{i,n}) = \tau_i(1-x_{i+1,n-1})(1-(1-x_{i+1,n-1})); \qquad i \in [1..6]$$
$$breeze256(12 \; maps : f_{1..12} \; with \; \tau_{1..12})$$
$$f_i(x_{i,n}) = \tau_i(1-x_{i+1,n-1})(1-(1-x_{i+1,n-1})); \qquad i \in [1..12]$$
$$breeze512(24 \; maps : f_{1..24} \; with \; \tau_{1..24})$$
$$f_i(x_{i,n}) = \tau_i(1-x_{i+1,n-1})(1-(1-x_{i+1,n-1})); \qquad i \in [1..24]$$

represented by the following *Go* code:

```
func (l *Breeze128) roundTrip() {
  newstate1 := (1.0 - l.state1)
  newstate1 *= 4.0 * l.state1
  newstate2 := (1.0 - l.state2)
  newstate2 *= 3.999999999 * l.state2
  newstate3 := (1.0 - l.state3)
  newstate3 *= 3.99999998 * l.state3
  newstate4 := (1.0 - l.state4)
  newstate4 *= 3.99999997 * l.state4
  newstate5 := (1.0 - l.state5)
  newstate5 *= 3.999999 * l.state5
  newstate6 := (1.0 - l.state6)
  newstate6 *= 3.999997 * l.state6

  switch newstate1 * newstate2 * newstate3 * newstate4 * newstate5 * newstate6 {
  case 0:
    s1 := ((math.Float64bits(l.state1))<<11>>(12 + l.bitshift%7))
    s1 += ((math.Float64bits(l.state2))<<11>>(12 + l.bitshift%7))
    s1 += ((math.Float64bits(l.state5))<<11>>(12 + l.bitshift%7))
    s2 := ((math.Float64bits(l.state3))<<11>>(12 + l.bitshift%7))
    s2 += ((math.Float64bits(l.state4))<<11>>(12 + l.bitshift%7))
    s2 += ((math.Float64bits(l.state6))<<11>>(12 + l.bitshift%7))
    seed := [2]uint64{s1, s2}
    l.bitshift++
    l.seedr(seed)
  default:
```

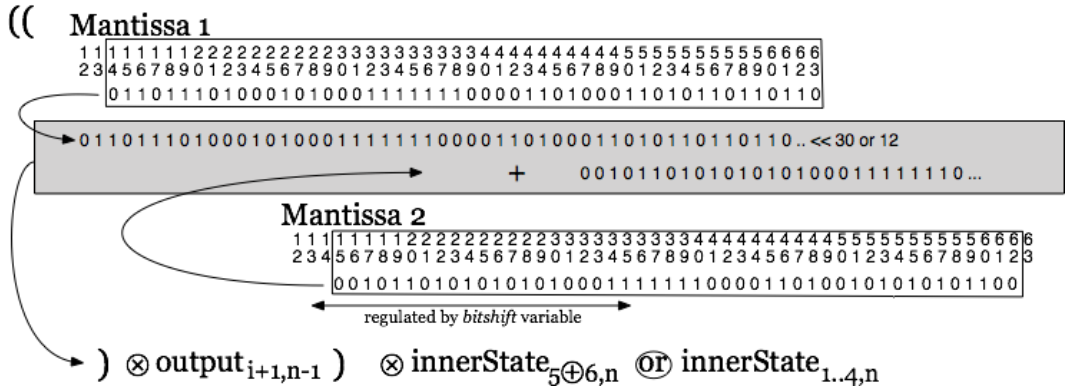
```
    l.state1 = 1.0 - newstate2
    l.state2 = 1.0 - newstate3
    l.state3 = 1.0 - newstate4
    l.state4 = 1.0 - newstate5
    l.state5 = 1.0 - newstate6
    l.state6 = 1.0 - newstate1
  }
  ...
}

Note: math.Float64bits() returns Little Endian bit representation of the float double.
      '^' means xor, '>>' '<<' bitwise shifting and '\%' means mod.
```

The switch statement leeds to a reseed of the generator from the previous states mantissa in case one of the (new)states rounded to Zero in the equation. The 'default:' case leeds to interchange of the (new)state values 'mirrored at 1'.
By splitting the newstate calculation (see code) 'Fused Multiply Add' (FMA) operation with different internal floating point representation in some chipsets should be prevented. The splitting should ensure that the intermediate results are always stored in 64 bit floating point words.

The theoretical entropy of logistic map orbits (isomorphic symmetric of $\frac{1}{2}$) in finite space of double float computation should be $\frac{L^{52}}{2} = L^{51}$ but the degradation of the map will reduce this in computational linear exhaustion of the maps pseudo orbits. Breeze approach of multiple logistic maps does not suffer of such degradation because any new cycle of computation may or may not be within the previous pseudo orbit (hopping effect) and the output randomness may therefore benefits from the full entropic potential of the finite space representation: $\mathbf{H} = log_2(2^{51}) = 51\ Sh.$

2.2. **Processing output from the logistic maps.** In contrast to other chaos based PRNG breeze uses the mantissa bits of the computation result from the logistic map equation as source of random bits. As discussed before the entropy of the 52 mantissa bits in double float IEEE754 representation should be $L^{51} or 51\ Sh.$ For latter crypto graphical use of the algorithm, encapsulation of the computation outcomes is useful to prevent leakage of any inner state of the generator. Therefore fractions of at least two outcome mantissa are added to form one 64 bit stream that is stored in a 64 unsigned integer. The underlying states are further obfuscated by bitwise xoring this variable by a previous 64 bit output and the 64 bitstream deriving from two different logistic map equations.

This is the 'GO' code to process the output from float double Little Endian bit representation:




```
func (l *Breeze128) roundTrip() {
  ...
  l.bitshift = (l.bitshift + 1) % 20
  tmp := l.state[0]
  l.state[0] = l.state[1] ^ (((math.Float64bits(l.state1))<<30)
               + ((math.Float64bits(l.state2))<<12>>(13 + l.bitshift)))
  l.state[1] = l.state[2] ^ (((math.Float64bits(l.state2))<<30)
               + ((math.Float64bits(l.state3))<<12>>(13 + l.bitshift)))
  l.state[2] = l.state[3] ^ (((math.Float64bits(l.state3))<<30)
               + ((math.Float64bits(l.state4))<<12>>(13 + l.bitshift)))
  l.state[3] = l.state[4] ^ (((math.Float64bits(l.state4))<<30)
               + ((math.Float64bits(l.state1))<<12>>(13 + l.bitshift)))
  hop := (((math.Float64bits(l.state5))<<30)
         + ((math.Float64bits(l.state6))<<12>>(13 + l.bitshift)))
  l.bitshift++
  l.state[4] = (l.state[5] ^ (((math.Float64bits(l.state1))<<12)
               + ((math.Float64bits(l.state2))<<12>>(13 + l.bitshift)))) ^ l.state[2]
  l.state[5] = (l.state[6] ^ (((math.Float64bits(l.state1))<<30)
               + ((math.Float64bits(l.state3))<<12>>(13 + l.bitshift)))) ^ hop
  l.state[6] = (l.state[7] ^ (((math.Float64bits(l.state1))<<30)
               + ((math.Float64bits(l.state4))<<12>>(13 + l.bitshift)))) ^ l.state[1]
  l.state[7] = (l.state[8] ^ (((math.Float64bits(l.state2))<<30)
               + ((math.Float64bits(l.state1))<<12>>(13 + l.bitshift)))) ^ hop
  l.state[8] = (l.state[9] ^ (((math.Float64bits(l.state2))<<12)
               + ((math.Float64bits(l.state3))<<12>>(13 + l.bitshift)))) ^ l.state[3]
  l.state[9] = (l.state[10] ^ (((math.Float64bits(l.state2))<<30)
               + ((math.Float64bits(l.state4))<<12>>(13 + l.bitshift)))) ^ hop
  l.bitshift++
  l.state[10] = (l.state[11] ^ (((math.Float64bits(l.state3))<<30)
                + ((math.Float64bits(l.state2))<<12>>(13 + l.bitshift)))) ^ l.state[3]
  l.state[11] = (l.state[12] ^ (((math.Float64bits(l.state3))<<12)
                + ((math.Float64bits(l.state4))<<12>>(13 + l.bitshift)))) ^ l.state[1]
  l.state[12] = (l.state[13] ^ (((math.Float64bits(l.state3))<<30)
                + ((math.Float64bits(l.state1))<<12>>(13 + l.bitshift)))) ^ hop
  l.state[13] = (l.state[14] ^ (((math.Float64bits(l.state4))<<12)
                + ((math.Float64bits(l.state1))<<12>>(13 + l.bitshift)))) ^ l.state[2]
  l.state[14] = (l.state[15] ^ (((math.Float64bits(l.state4))<<30)
                + ((math.Float64bits(l.state2))<<12>>(13 + l.bitshift)))) ^ hop
  l.state[15] = (tmp ^ (((math.Float64bits(l.state4))<<30)
                + ((math.Float64bits(l.state3))<<12>>(13 + l.bitshift)))) ^ l.state[0]
}

Note: math.Float64bits() returns Little Endian bit representation of the float double.
      '^' means xor, '>>' '<<' bitwise shifting and '%' means mod.
```

2.3. **Initialization.** The main difference between the breeze variants is their key space. The multiple logistic maps are seeded each by $x_0 = \frac{1}{s}$ *with* $\{s \in \mathbb{N} \mid (1, 2^{21} \vee {}^{22})\}$.

Initialization seed derives from a 64 bit word cut into three s representing 21, 22, 21 bits.

That leads to a 128bit keyspace (two 64 bit words for six seedings) in breeze128, 256 bit (four 64 bit words for twelve seedings) in breeze256 and 512 bit (eight 64 bit words for 24 seedings) in breeze512.

3. Testing for random output

3.1. **NIST Test Suite.** The three variants of breeze had been tested by NIST Test Suite (NIST (2014)). 20 Sets of 100 output sequences of $8 * 10^6$ bit length deriving from each variant passed the test suite without warnings.

3.2. **Visual patterns.** Output from the PRNG was converted into RGBA-channels of 24bit png images (8000x4000 pixels scaled to 25%) underlaid with white and black backgrounds for visual pattern recognition and thoroughly inspected switching between consecutive images in 1sec intervals. Human visual sense is very fast in pattern recognition. Such patterns would indicate weakness in the random output by repeated sequence (short periods) or degradation of underlying entropy.

3.3. **Test for output doublets.** Degradation of the PRNG would lead to shortened periods, that are found repeatedly in the output. the breeze variants breeze128, breeze256 and breeze512 hold output arrays of 64 bit unsigned integers of length 16, 32 and 64, that are produced by one computation cycle.

A 2^{36} bit Bloom filter with 7 hash locations was used to check continuously for repetitions of consecutive pairs of PRNG variant output (256, 512, 1024 Byte) in 100 sets of 200 GB, 400 GB, 800 GB output length from breeze128, breeze256 and breeze512.

A single doublet was found in each of breeze128 and breeze512 total output sequences. They occurred after 174 GB output testing in the 98th sequence and after 728 GB of the 61rst sequence respectively.

3.4. **Results and discussion.** The NIST Test Suite was passed by the test sets without warnings and no indication of visible patterns in the generated images were observed.

Bloom filters are probabilistic approaches to test for inclusion into the filter (see http://en.wikipedia.org/ wiki/ Bloom_filter). The filter is 100 % accurate for non-inclusion response and if the bits corresponding to the hash values of a element to be checked isn't set, the filter does not include that value. But Bloom filters pose a false positive error (meaning that all corresponding bits are set even if the value is not included in the filter) that can be calculated by (4).

$$p = (1 - (\frac{1}{m})^{kn})^k \tag{5}$$

with m = No. of filter bits; k = No. of hashes (locations); n = No. of elements included.

Within the findings range (174GB: 679687500 $\leqslant$ element number, 728GB: 710937500 $\leqslant$ element number) the false positive probability for the findings was calculated to be 5.99e-09 and 8.12e-09 respectively - furthermore the error probability might be underestimated because only one hash function (sipHash) is used to compute seven locations in the Bloom filter and sipHash is not claimed to be collision resistant.

It is assumed that these two findings by the Bloom filter test were false positives and the fact that these findings were solitary and do not indicate a repetition in sequence (a short period longer than three elements) is supporting this statement.

To summarize, the performed test did not indicate weaknesses in any of the variants of the proposed chaos based pseudo random generator. Nonetheless prior to use in security sensible areas (i.e. used as stream cipher or other cryptographic purpose) further cryptanalysis by a third party, that underline it's security, might be considered.

4. Literature

Shujun Li (2003): Analyses and New Designs of Digital Chaotic Ciphers. Ph.D. thesis, School of Electronic and Information Engi- neering, Xian Jiaotong University, Xian, China. Available online at http://www.hooklee.com/pub.html.

Shujun Li (2004): When chaos meets computers. URL http://arxiv.org/ abs/nlin.CD/0405038, last revised in December 2005.

David Arroyo (2009): Framework for the analysis and design of encryption strategies based on discrete-time chaotic dynamical systems, Thesis, Universidad Politecnica de Madrid, Dpto. de Fisica y Mecanica Fundamentales y Aplicadas a la Ingenieria Agroforestal Area de conocimiento de Fisica Aplicada y Matematica Aplicada

K. J. Persohn, R. J. Povinelli (2012): Analyzing Logistic Map Pseudorandom Number Generators for Periodicity Induced by Finite Precision Floating-Point Representation http://povinelli.eece.mu.edu/ publications/ papers/ chaos2012.pdf

NIST Test Suite (2014): version sts-2.1.2 including changes of July 9, 2014; http://csrc.nist.gov/ groups/ ST/ toolkit/ rng/ index.html

Burton H. Bloom (1970), "Space/Time Trade-offs in Hash Coding with Allowable Errors", Communications of the ACM 13 (7): 422426, doi:10.1145/362686.362692