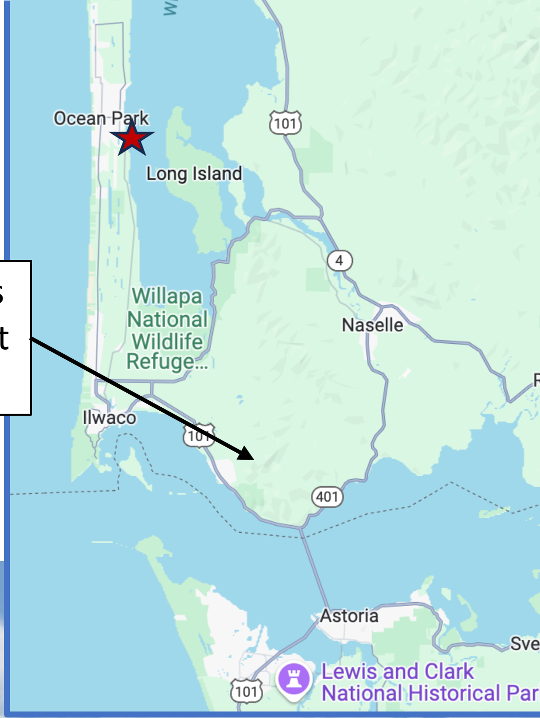
The use and Abuse of Random Numbers

Tim Mattson

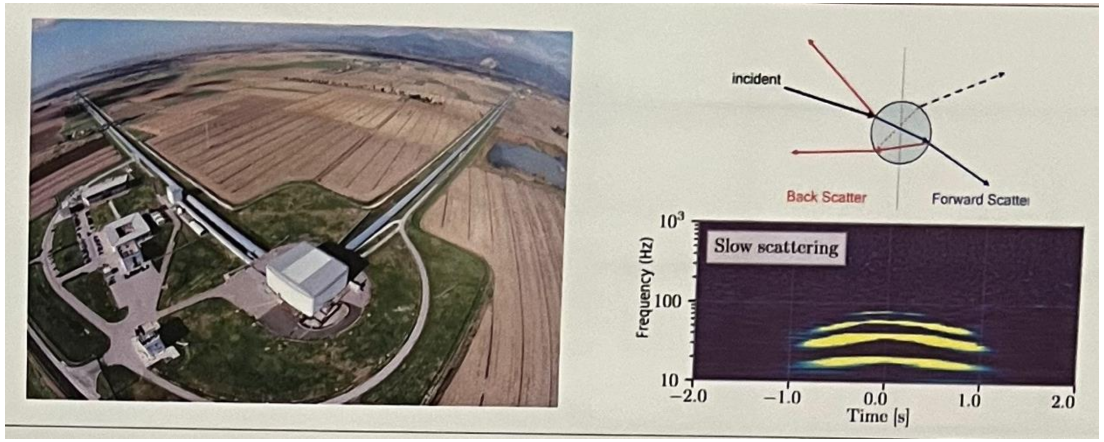
The view below is Looking Southeast from Here

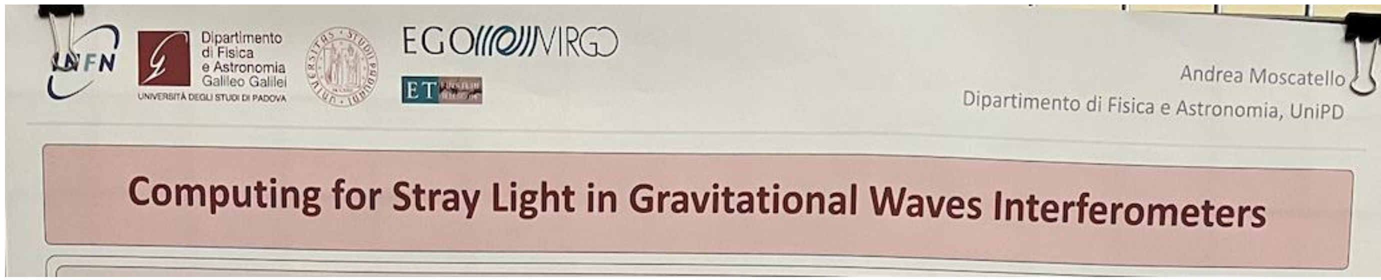


Solving problems with random samples

- For a large and important class of algorithms, we sample a problem domain and then use a statistical analysis over those samples to generate an answer. These are often called Monte Carlo algorithms.

- Algorithms based on random sampling are very common … for example, in high energy physics …
    - Monte Carlo generation of collisions events (e.g. Pythia)
    - Monte Carlo detector simulation (e.g. Geant4)
    - Monte Carlo digitization (e.g. electronics simulation)
    - Statistical analysis (e.g. significance tests, importance sampling)





Example: Monte Carlo Integration

- A simple and direct method to approximate definite integrals
- The definite integral for the integrand $f(\vec{x})$ over a d-dimensional domain $\vec{x}$ is:

$$I[f] = \int_0^1 f(\vec{x}) d\vec{x}$$

- The limits of the integral are normalized over a d-dimensional cube $[0,1]^d$
- Randomly sample points within $[0,1]^d$ over a uniform distribution to create a sequence $\{\vec{x}_i\}$.
- The empirical approximation to the definite integral is $I_N[f]$.

$$I_N[f] = \frac{1}{N}\sum_{i=1}^{N} f(\vec{x}_i) \qquad \lim_{N\to\infty} I_N[f] \to I[f]$$

- The rate of convergence … independent of the dimension, d … is $O(N^{-1/2})$
- This is a slow rate of convergence, but it is independent of the dimension of the integral. For integrals solved over grids over $[0,1]^d$ the rate of convergence is $O(N^{-k/d})$ where k is the order of the numerical quadrature method. Also, defining a grid over $[0,1]^d$ for large d results in a prohibitively large number of points to sample.

Monte Carlo integration is robust, easy to implement, and for higher dimension problems (any time k/d < ½) the rate of convergence (while still slow) beats traditional numerical quadrature methods.

Choosing the random samples

- For Monte Carlo algorithms to work, the random samples must be:
  - Distributed according to the statistics required of the problem … that is, uniformly distributed or as samples of a predefined distribution (e.g., Gaussian, Poison, etc.).
  - Each Sample must be unpredictable given knowledge of other samples.

- A sequence of such numbers are called *Random Numbers.*

- We can generate a sequence of Random numbers from natural processes (e.g. white noise from a thermocouple), but not by any algorithm running on a deterministic machine (i.e., a computer).

```
// function returns a
// random number

int random()
{
  return 3;
}
```

The best we can do on a computer is produce numbers that appear to be random … that lack correlations between numbers or other features in the sequence that make the numbers predictable. We call these **Pseudo-Random numbers**.

Monte Carlo Methods require high quality pseudo-random numbers

Pseudo-Random Numbers

- High Quality Pseudo-Random numbers are indistinguishable from true Random numbers.

- They are generated by deterministic algorithms which means they can generate the same sequence between runs of a program (critical for validation purposes) … Reproducibility is your friend!!!!

- Pseudo-Random numbers, however, present their own challenges.
    - They aren't truly Random … the key is to use formal testing to show they are random enough.
    - It is depressingly easy to generate bad sequences of pseudo-random numbers and never know that your scientific results are garbage.
    - Its easy to write Pseudo-Random number generators but extremely difficult to write ones that are dependably random enough in all situations. Leave creating such generators to the pros … use libraries.

How to create sequences of Pseudo-Random Numbers

- We call the software that generates our pseudo-random numbers a random number generator or RNG

- There are at least two parts to an RNG … the algorithm and the parameters.

- Some common algorithms (we'll talk about parameters later)
  - Linear Congruential Generator (LCG)
  - Lagged Fibonacci Generator
  - Mersenne Twister
  - XORshift generator
  - Wichmann-Hill generator

- There is no single "best" generator … the key is to pick the generator best suited to your needs.
  - LCG is easy to implement and has decent quality if you get the parameters right.
  - Wichmann-Hill is a family of independent generators … quite handy for parallel applications
  - XORshift is very efficient (3 shift and 3 XOR operations)

Random Numbers: Linear Congruential Generator (LCG)

- LCG: Easy to write, cheap to compute, portable, OK quality

```
random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
random_last = random_next;
```

- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I often use the following:
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025

If the ADDEND is zero, then we have a Multiplicative Linear Congruential Generator. (MLCG)

If you are careful in selecting the MULTIPLIER and PMOD, MLCG can be quite good.

LCG code

```
static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 1597;
double drandom ()
{
   long random_next;

   random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
   random_last = random_next;

   return  ((double)random_next/(double)PMOD);
}
```

Seed the pseudo random sequence by setting random_last

I often just pick a prime number that is less than PMOD.

Be careful with your random number generators

Famous PseudoRandom Generators: RANDU

- RANDU was a standard RNG from IBM. It was used heavily on their systems in the 1960s and 1970s.

- RANDU is a **Multiplicative Linear congruential** generator with multiplier, M, equal to 65539 and the modulus, mod, equal to 2^{31}. The seed (X_0) must be odd.

$$X_{n+1} = (M \cdot X_n)\% mod$$

- The following is Python code for RANDU

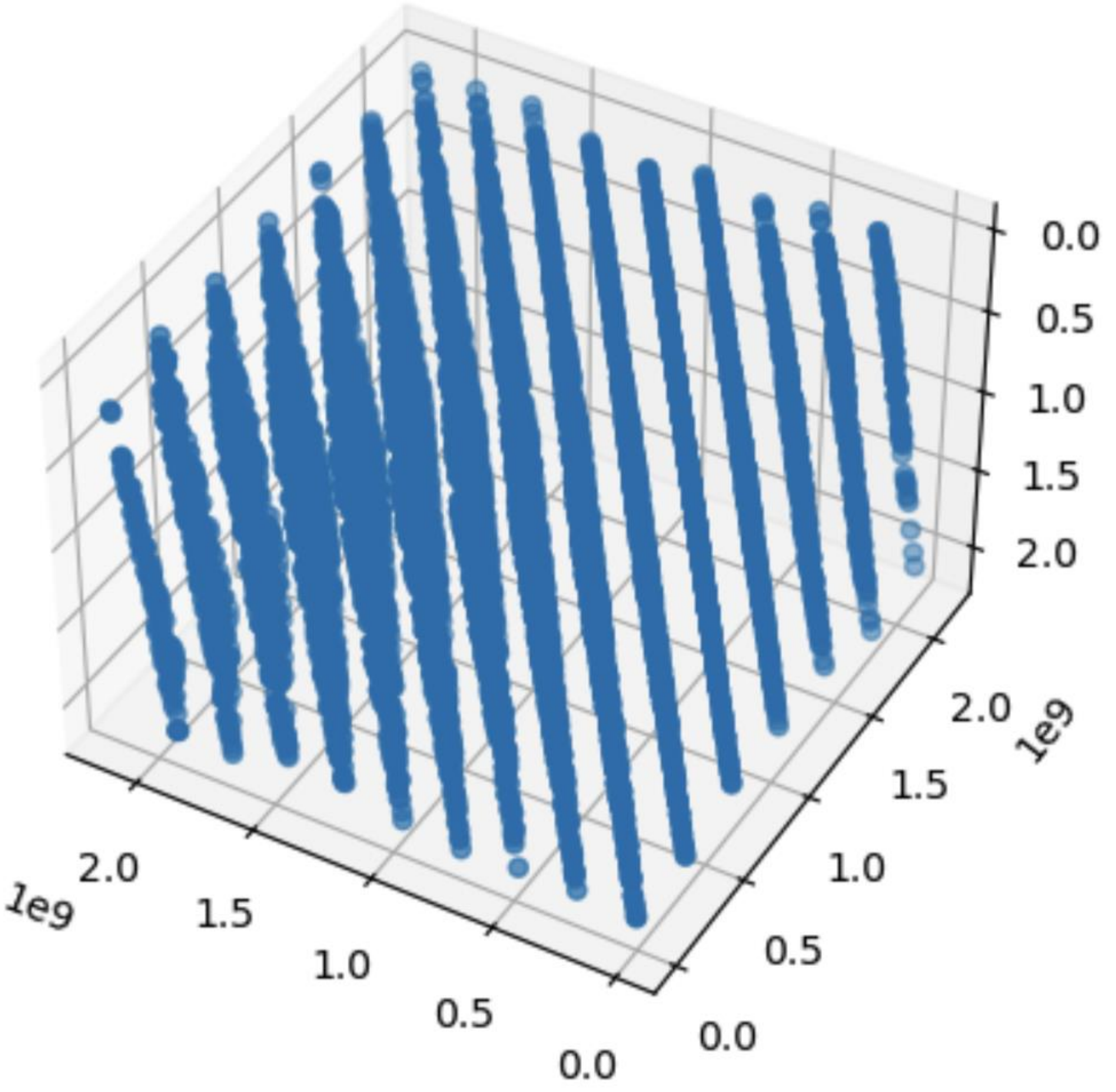
```
class RANDU:
  def __init__(self, seed=483647):
      self.seed = xval
      self.Mod  = 2_147_483_648
      self.Mult = 65539
      self.last = seed

   def random(self):
      self.last = (self.Mult*self.last)%self.Mod
      return self.last
```

RANDU generates integers ranging from 1 to (2^{31}-1)

- RANDU passes basic frequency tests (build a histogram of a sequence. Use a chi-squared test to verify numbers per bin are appropriately equal)

… but RANDU has problems

- It passes frequency tests, but those test overall statistics. They can't find local correlations.
- To look for such correlations, we can take consecutive blocks of three values and plot them as x,y,z coordinates in a 3D scatter plot.
- We see that the values fall along 15 hyperplanes. The generator exhibits local correlations between values.
- This means Monte Carlo results with this generator are suspect since such methods assume uniform random sampling.
- Problems with this generator were known as early as 1963.
- It wasn't until the 1990s that it was widely eliminated, though some Fortran compilers were found using it as late as 1999.



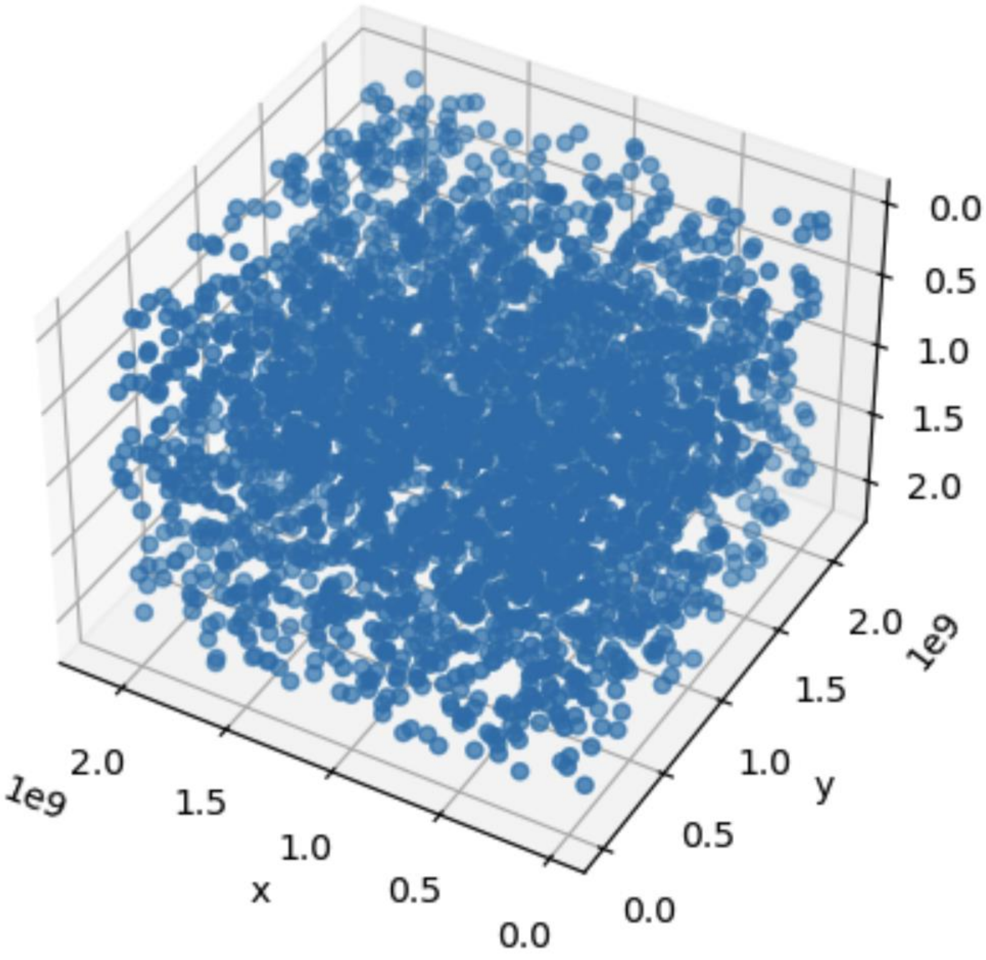
Each point in the plot (x, y, z)) is three consecutive values from RANDU. The points all fall into 15 planes.

Fixing RANDU

- We can fix this generator by more carefully selecting the Multiplier and the modulus value.

- Looking up values from a rigorous (peer reviewed) mathematical work* I updated the values in RANDU:

```
class MultLCG:
   def __init__(self, seed=483647):
       self.seed = xval
       self.Mod  = 2_147_483_647
       self.Mult = 1_583_458_089
       self.last = seed

    def random(self):
       self.last = (self.Mult*self.last)%self.Mod
       return self.last
```



- This new generator passed my frequency tests and removed the local correlations

*Pierre L'Ecuyer, "Tables of Linear Congruential Generators of different sizes and good lattice structure", Mathematics of Computation, Vol 68, Numb 225, jan 1999, pp 249-260

Plot generation code

If you are curious, here is the code I used to make the plots

```
from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt
```

```
randuTest = RANDU()
nvals = 30000     # make this divisible by three
seq=np.zeros(nvals,dtype=int)
for i in range(nvals):
   seq[i] = randuTest.random()
```

Generate a sequenced of pseudo random numbers using our RANDU generator

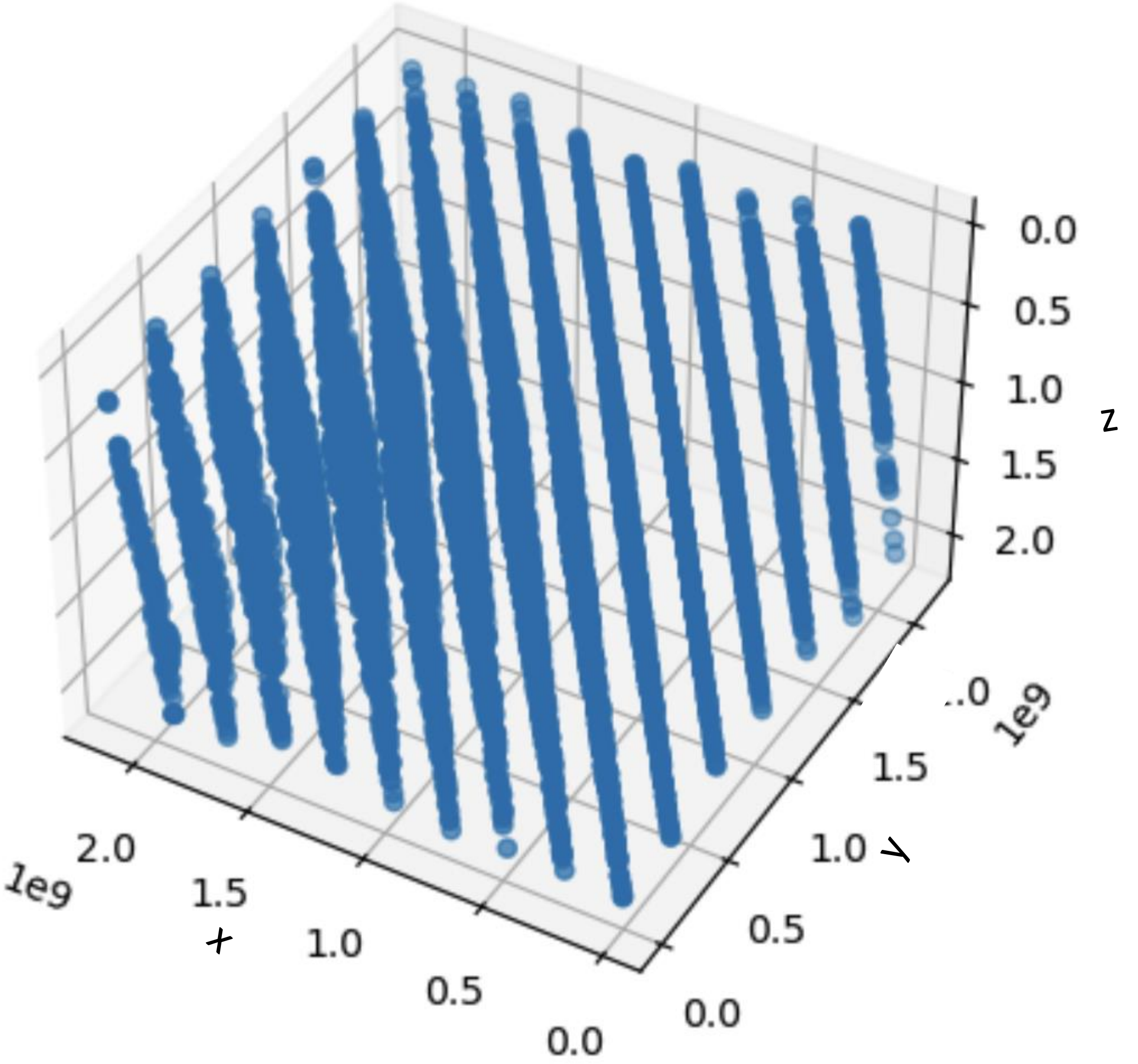
```
x = np.zeros(nvals//3)
y = np.zeros(nvals//3)
 z = np.zeros(nvals//3)
iseq = 0; i=0
while iseq < (nvals):
   x[i] = seq[iseq]
   y[i] = seq[iseq+1]
   z[i] = seq[iseq+2]
   iseq = iseq + 3
   i = i + 1
```

Gather consecutive values in the sequence by triples into three distinct sequences for plotting

```
ax = plt.axes(projection ='3d')
ax.scatter(x, y, z, 'blue')
ax.view_init(-140, 60)
plt.show()
```

Plot x,y,z points and view at an angle chosen to show the parallel hyperplanes

x, y, and z are triplets of consecutive values



Numbers are ints in units of billions

Key Lesson from the RANDU mess

- Maintain a healthy level of skepticism for any default, built-in Random number generator.
- Run your own tests to make sure the numbers are random enough.
- Insist on knowing:
  - Which method the generator is using (e.g. LCG, lagged Fibonacci, Mersenne Twister, etc.)
  - That the period of the generator is sufficient for your problem.
  - That the parameters used in the generator are good and from a reputable source
- I often write my own generator if I can't verify the details of built-in generators, but that is dangerous. It is best to find (and use) a reputable library.
  - Scalable Parallel Random Number Generators (SPRNG) (sprng.org) from Michael Mascagni (University of Florida and NIST)

Lets explore Monte Carlo methods and pseudo random number generators with a classic problem

Monte Carlo methods in Popular Culture

SMBC by Zack Weinersmith

… famous for the best high-level explanation of Quantum Computing EVER published:

https://www.smbc-comics.com/comic/the-talk-3

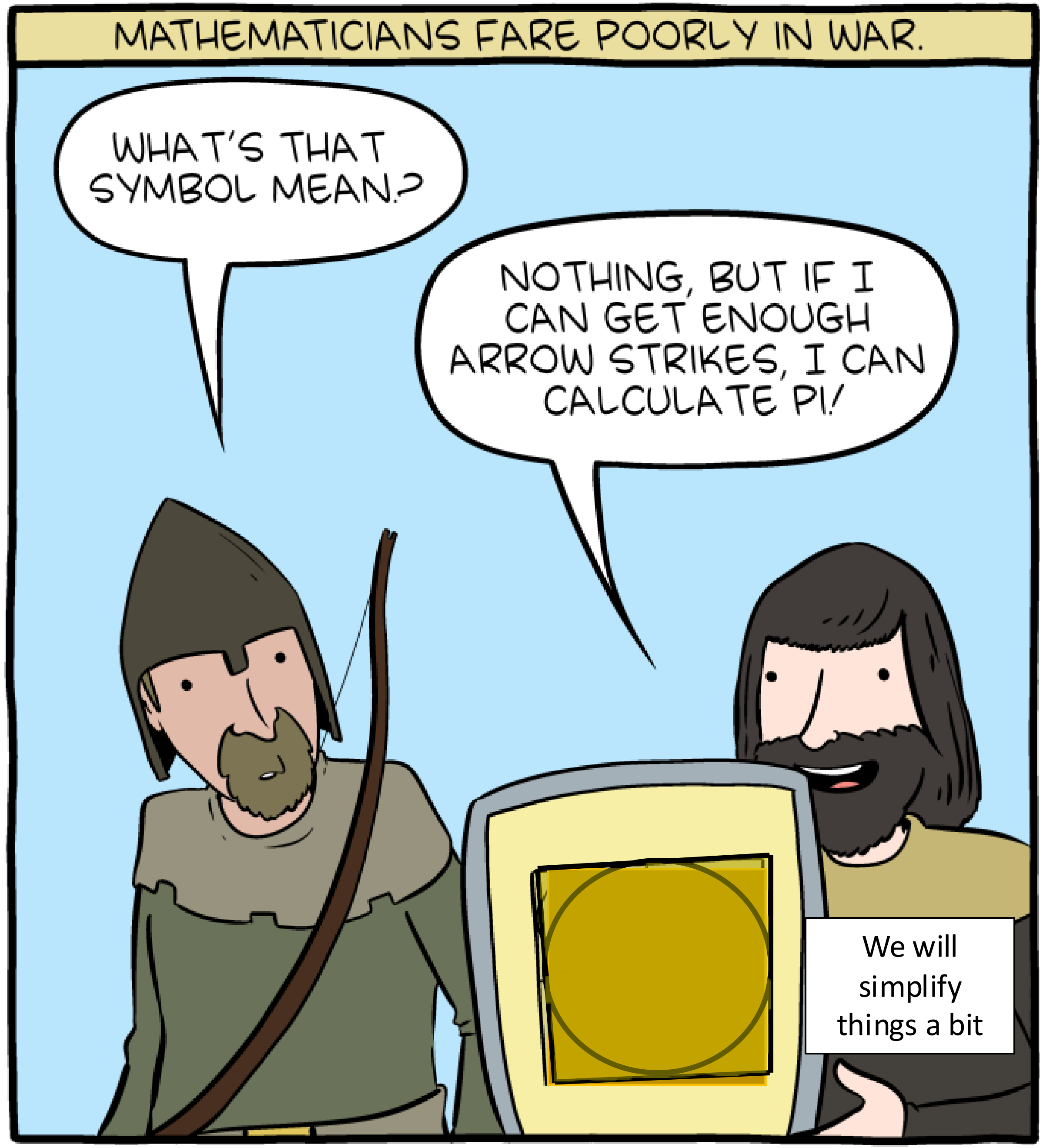
https://www.smbc-comics.com/comic/math-and-war

Monte Carlo methods in Popular Culture

SMBC by Zack Weinersmith

… famous for the best high level explanation of Quantum Computing EVER published:

https://www.smbc-comics.com/comic/the-talk-3



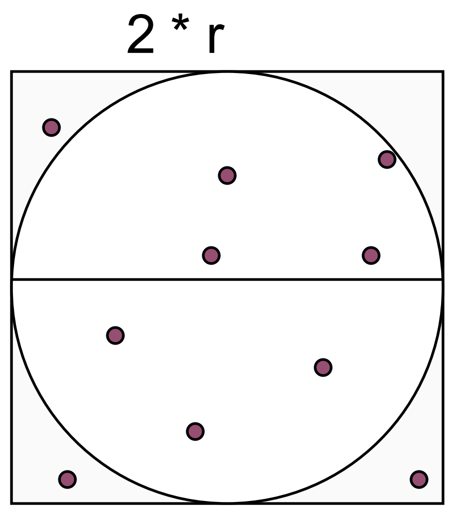
We will simplify things a bit

https://www.smbc-comics.com/comic/math-and-war

Monte Carlo Calculations

Using random numbers to solve problems

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing π with a digital dart board:



N= 10	π = 2.8
N=100	π = 3.16
N= 1000	π = 3.148

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:

  $A_c = r^2 * \pi$

  $A_s = (2*r) * (2*r) = 4 * r^2$

  $P = A_c/A_s = \pi /4$

- Compute π by randomly choosing points; π is four times the fraction that falls in the circle

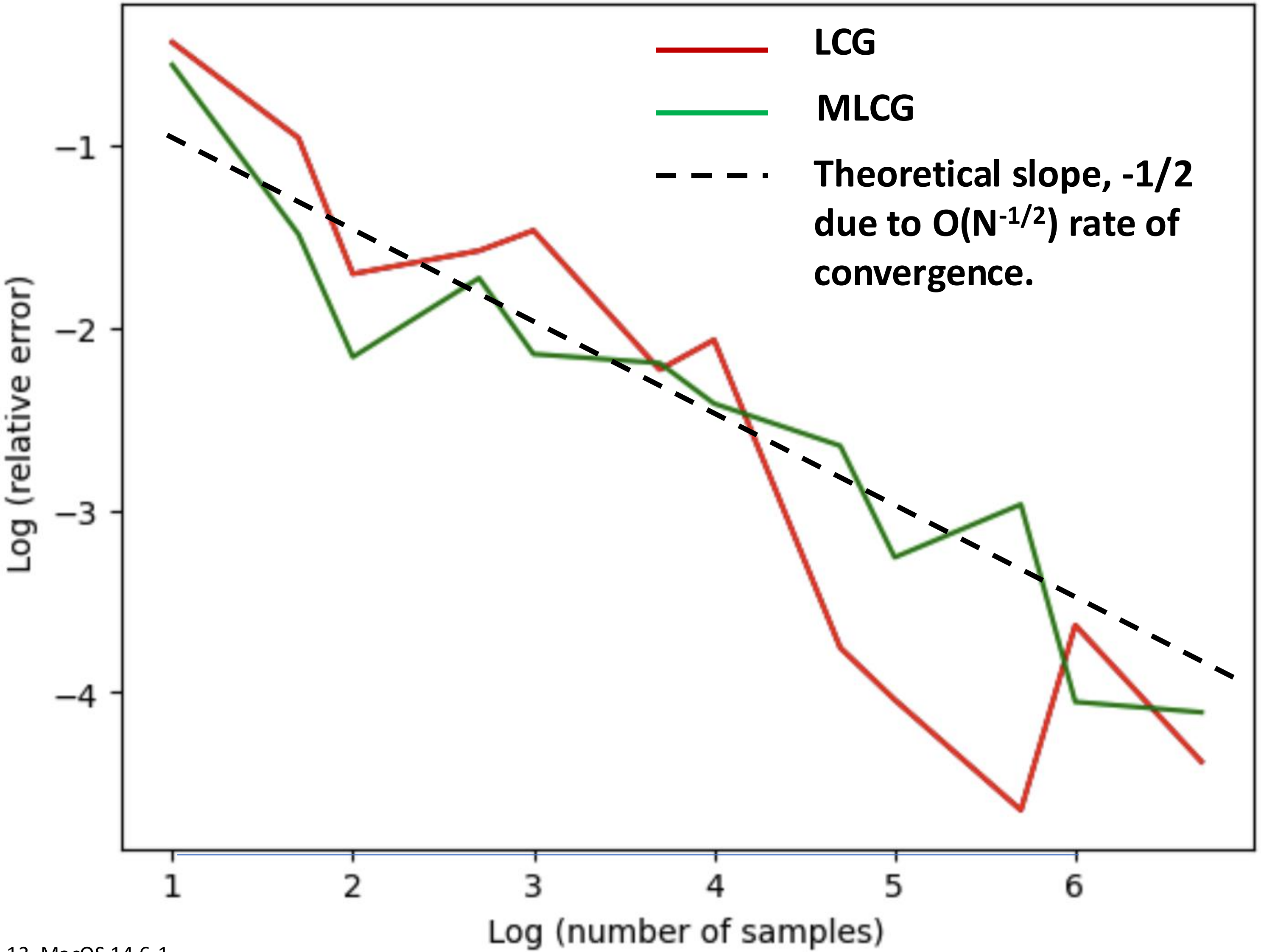
Monte Carlo algorithms: estimating π

```
#include random.h
static long num_trials = 10000;
int main ()
{
   long i;      long Ncirc = 0;       double pi, x, y;
   double r = 1.0;   // radius of circle. Side of squrare is 2*r
   seed(-r, r);          // The circle and square are centered at the origin
   for(i=0;i<num_trials; i++)
   {
      x = drandom();        y = drandom();
      if ( x*x + y*y) <= r*r)   Ncirc++;
    }

    pi = 4.0 * ((double)Ncirc/(double)num_trials);
    printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```

Single thread results π Monte Carlo: LCG and MLCG

LCG: Linear Congruential Generator $\quad$ ranNext = $(M_{LCG}$*ranLast + A)%mod_{LCG}

MLCG: Multiplicative Linear Congruential Generator. $\quad$ ranNext = $(M_{MLCG}$*ranLast)%mod_{MLCG}




```
// LCG parameters
static long MULTIPLIER  = 2416;
static long ADDEND      = 37441;
static long PMOD        = 1771875;
static long SEED        = 7919;

// MLCG parameters
static long MULTIPLIER  = 1583458089;
static long PMOD        = 2147483647;
static long SEED        = 7325973;
```

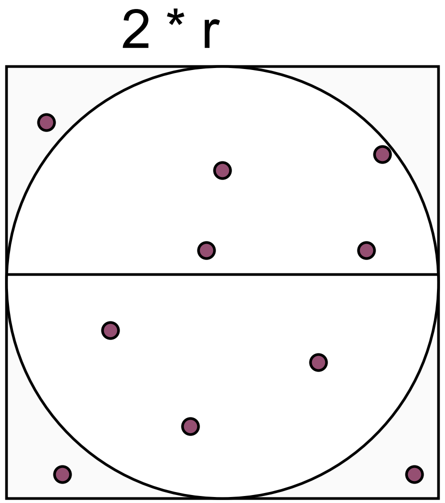
Apple M2 chip, gcc-13, MacOS 14.6.1

… let’s go parallel

Monte Carlo Calculations

Using random numbers to solve problems

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing π with a digital dart board:



N= 10	π = 2.8
N=100	π = 3.16
N= 1000	π = 3.148

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:

  $A_c = r^2 * \pi$

  $A_s = (2*r) * (2*r) = 4 * r^2$

  $P = A_c/A_s = \pi /4$
- Compute π by randomly choosing points; π is four times the fraction that falls in the circle

Parallel Programmers love Monte Carlo algorithms

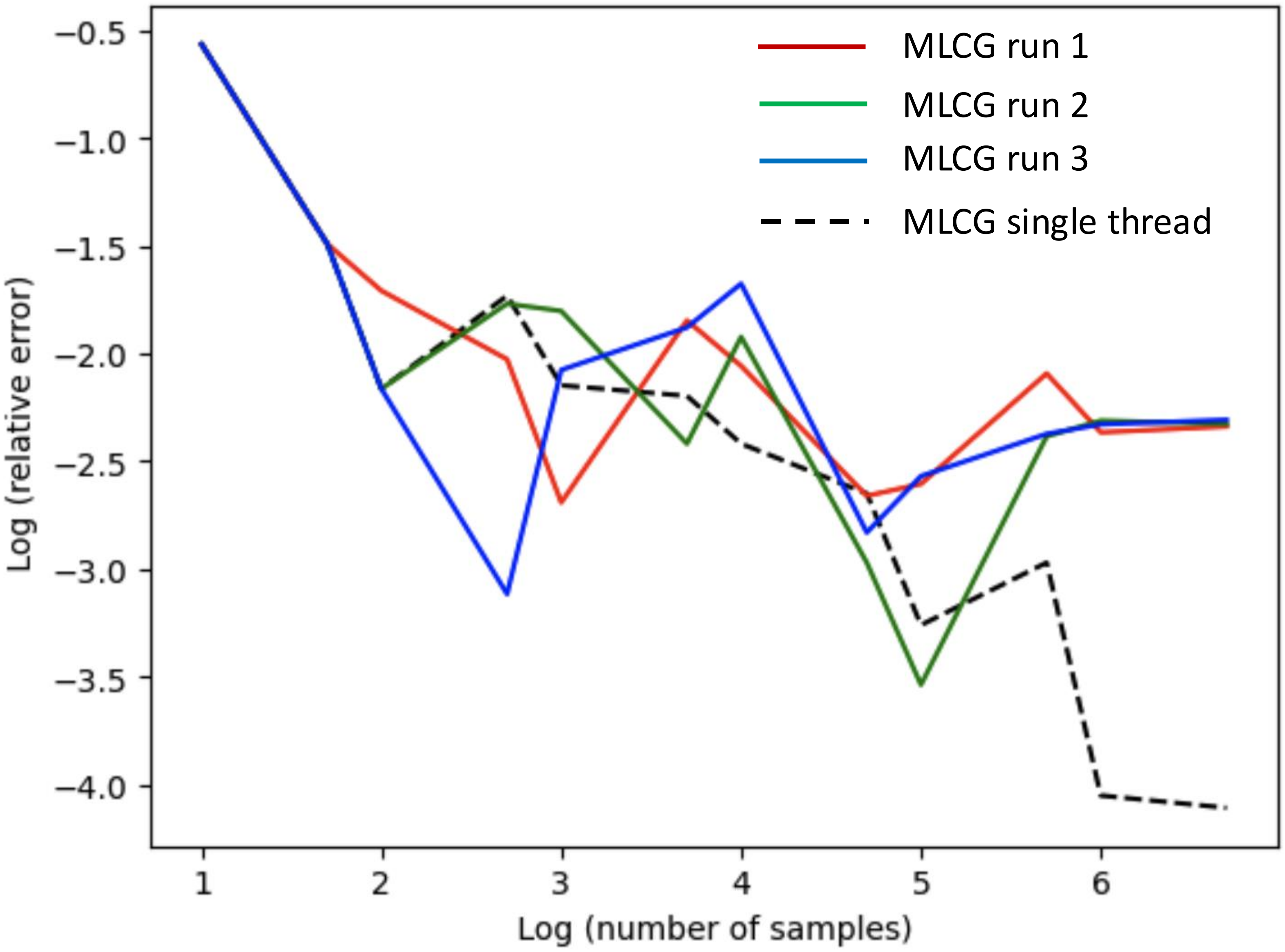
```
#include “omp.h”
static long num_trials = 10000;
int main ()
{
   long i;      long Ncirc = 0;       double pi, x, y;
   double r = 1.0;   // radius of circle. Side of squrare is 2*r
   seed(0,-r, r);  // The circle and square are centered at the origin
   #pragma omp parallel for private (x, y) reduction (+:Ncirc)
   for(i=0;i<num_trials; i++)
   {
      x = random();        y = random();
      if ( x*x + y*y) <= r*r)   Ncirc++;
    }

    pi = 4.0 * ((double)Ncirc/(double)num_trials);
    printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```

Embarrassingly parallel: the parallelism is so easy its embarrassing.

Add two lines and you have a parallel program.

π Monte Carlo with 8 threads: LCG and MLCG

MLCG: Multiplicative Linear Congruential Generator. ranNext = (M_{MLCG}*ranLast)%mod_{MLCG}



Run the same program the same way and get different answers!

That is not acceptable!

Issue: The MLCG generator is not threadsafe

// MLCG parameters

```
static long MULTIPLIER  = 1583458089;
static long PMOD        = 2147483647;
static long SEED        = 7325973;
```

Apple M2 chip, gcc-13, MacOS 14.6.1

Data Sharing and OpenMP: Threadprivate

- Makes global data private to a thread
  - Fortran: **COMMON** blocks
  - C: File scope and static variables, static class members
- Different from making them **PRIVATE**
  - with **PRIVATE** global variables are masked.
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using **COPYIN** or at time of definition (using language-defined initialization capabilities)

A Threadprivate Example (C)

Use threadprivate to create a counter for each thread.

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return (counter);
}
```

MLCG code: threadsafe version

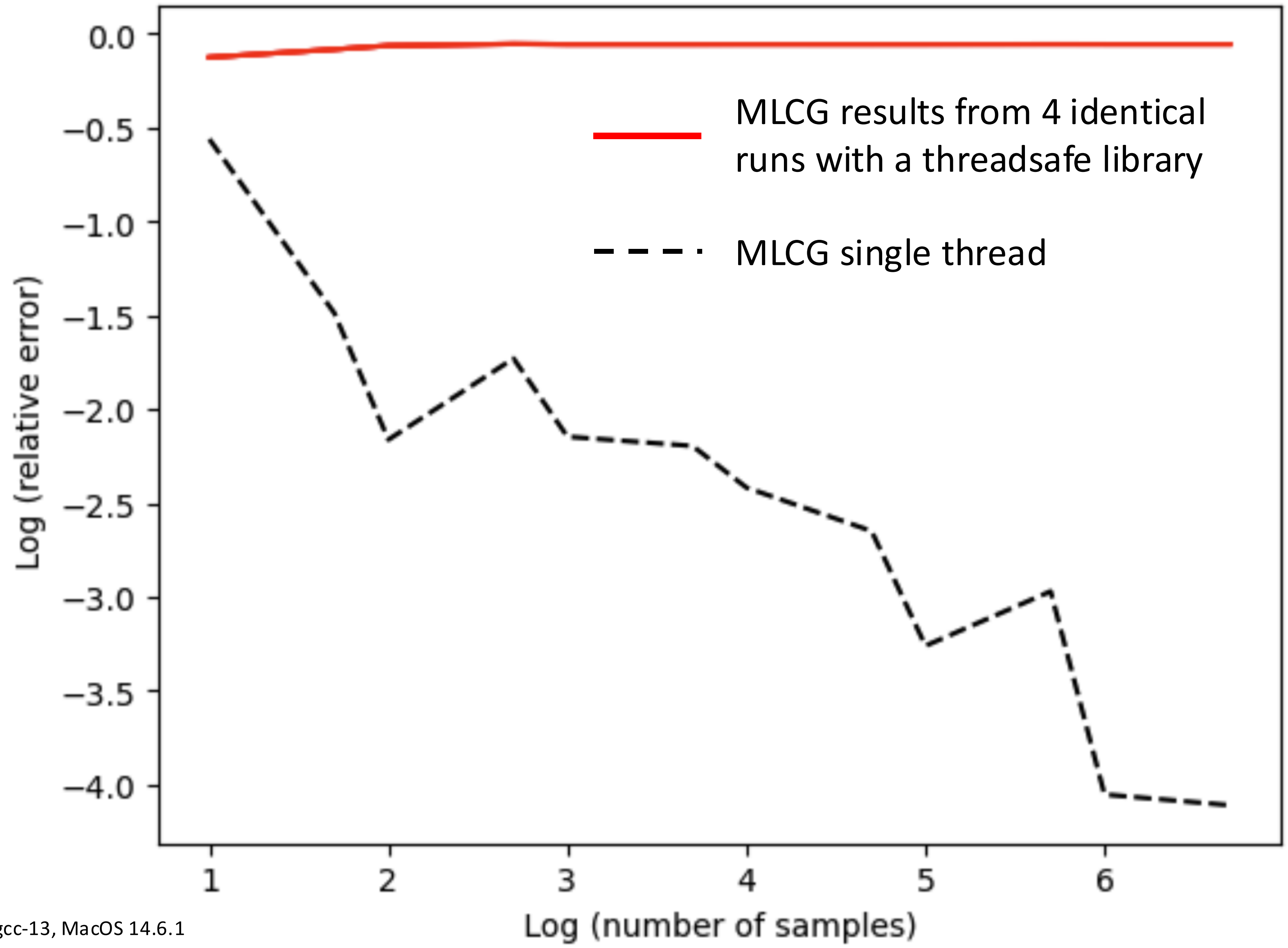
```
static long MULTIPLIER  = 1366;
static long PMOD        = 714025;
long random_last = 0;
#pragma omp threadprivate(random_last)
double random ()
{
   long random_next;

   random_next = (MULTIPLIER  * random_last)% PMOD;
   random_last = random_next;

  return  ((double)random_next/(double)PMOD);
}
```

random_last carries state between random number computations,

To make the generator threadsafe, make random_last threadprivate so each thread has its own copy.

π Monte Carlo with 8 threads: Threadsafe RNG library

MLCG: Multiplicative Linear Congruential Generator. ranNext = $(M_{MLCG}*ranLast)\%mod_{MLCG}$



The library is threadsafe … we get the same results from one run to the next, but the results are awful.

Why?

// MLCG parameters

```
static long MULTIPLIER  = 1583458089;
static long PMOD        = 2147483647;
static long SEED        = 7325973;
```

Apple M2 chip, gcc-13, MacOS 14.6.1

Pseudo Random Sequences

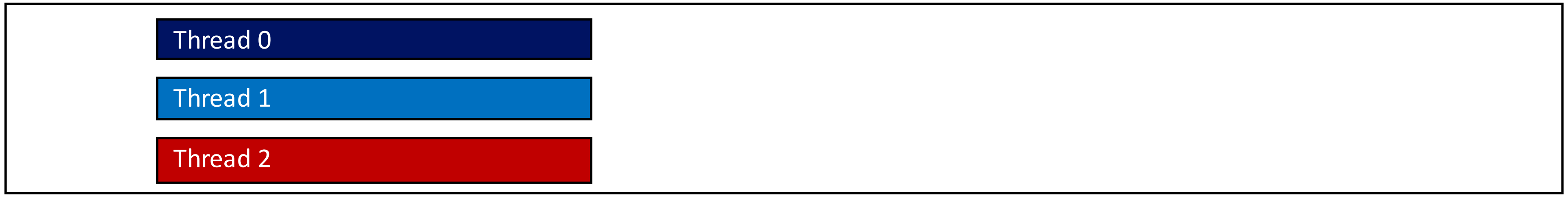
- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG

For LCG RNGs, if you pick the addend and multiplier correctly, the **period** equals the mod_{LCG} parameter

- In a typical problem, you grab a subsequence of the RNG period

Seed determines starting point

- IF each thread has the same seed, you just sample the same points over and over from each thread.

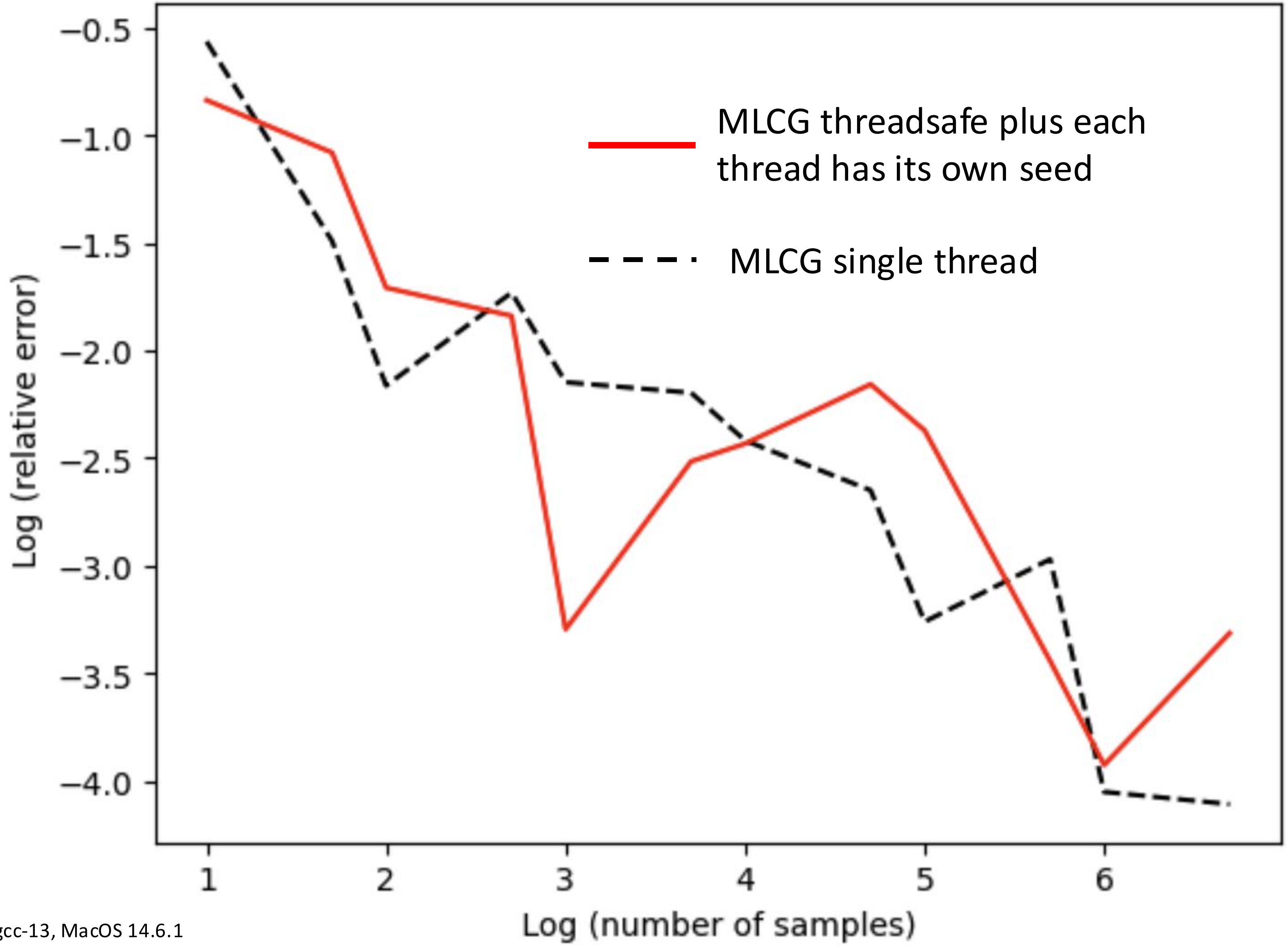


LCG: Linear Congruential Generator $ranNext = (M_{LCG}*ranLast + A)\%mod_{LCG}$

π Monte Carlo with 8 threads: Different seed per thread

MLCG: Multiplicative Linear Congruential Generator. ranNext = $(M_{MLCG}$*ranLast)%mod_{MLCG}

Seed(): Called inside parallel region by each thread to set ranLast = SEED * (thread_ID + 1)



Results are much better, but are erratic and degrade for larger cases

Why?

// MLCG parameters

```
static long MULTIPLIER  = 1583458089;
static long PMOD        = 2147483647;
static long SEED        = 7325973;
```

Apple M2 chip, gcc-13, MacOS 14.6.1

Pseudo Random Sequences

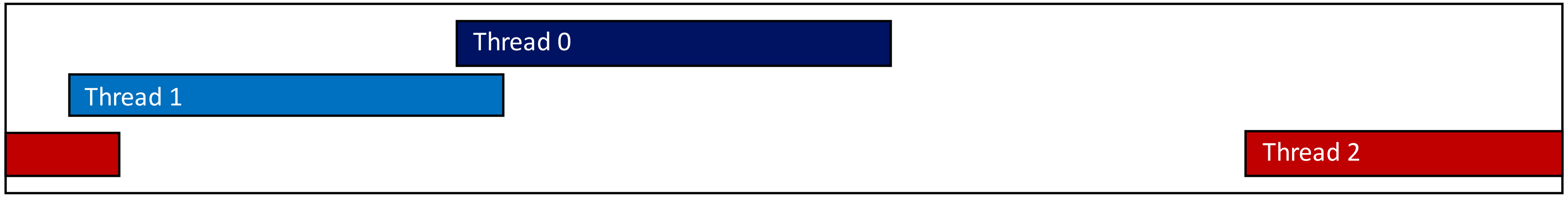
- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG

For LCG RNGs, if you pick the addend and multiplier correctly, the **period** equals the mod_{LCG} parameter

- In a typical problem, you grab a subsequence of the RNG period

Seed determines starting point

- Grab arbitrary seeds and you may generate overlapping sequences



- Overlapping sequences = over-sampling some points and bad statistics … lower quality or even wrong answers!

LCG: Linear Congruential Generator   ranNext = (M_{LCG}*ranLast + A)%mod_{LCG}

Parallel random number generators

- Multiple threads cooperate to generate and use random numbers.
- Solutions:
  - Pick a seed and hope for the best (a common approach)
  - Give each thread a separate, independent generator
  - Have one thread generate all the pseudo-random numbers.
  - Leapfrog … deal out sequence values “round robin” as if dealing a deck of cards.
  - Block method … pick your seed so each threads gets a distinct contiguous block.
- Other than “replicate and hope”, these are difficult to implement. Be smart … get a math library that does it right.

If done right, can generate the same sequence regardless of the number of threads …

Important for validation and debugging, but not needed for high quality results.

The state of the art is the Scalable Parallel Random Number Generators Library (SPRNG): http://www.sprng.org/ from Michael Mascagni’s group at Florida State University.

Leap Frog (skipping) Method (for MLCG)

- Interleave samples in the sequence of pseudo random numbers:
  - Thread i starts at the i^{th} number in the sequence
  - Stride through sequence, stride length = number of threads.
- Result … the same sequence of values regardless of the number of threads.

```
#pragma omp single
{   nthreads = omp_get_num_threads();
     int id = omp_get_thread_num();
     iseed = PMOD/MULTIPLIER;     // just pick a seed
     pseed[0] = iseed;
     mult_n = MULTIPLIER;
     for (i = 1; i < nthreads; ++i)
    {
        iseed = (unsigned long long)((MULTIPLIER * iseed) % PMOD);
        pseed[i] = iseed;
        mult_n = (mult_n * MULTIPLIER) % PMOD;
    }

 }
 random_last = (unsigned long long) pseed[id];
```

One thread computes offsets and a strided multiplier (mult_n)

Each thread stores offset starting point into its threadprivate “random_last” value

Pseudo Random Sequences

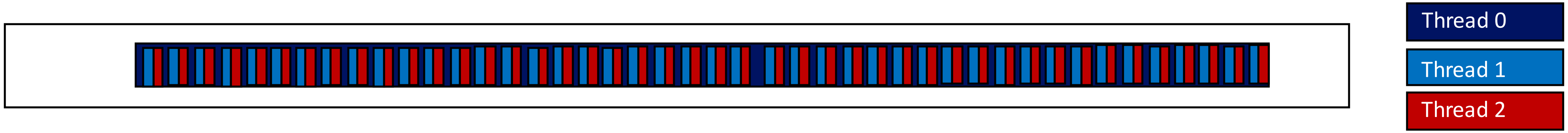
- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG

For LCG RNGs, if you pick the addend and multiplier correctly, the **period** equals the mod_{LCG} parameter

- In a typical problem, you grab a subsequence of the RNG period

Seed determines starting point

- Skip by the number of threads and start at seeds offset by a number of positions = to thread ID



- Parallel threads sample the same sub-sequence … you get the same answer regardless of the number of threads.

LCG: Linear Congruential Generator   ranNext = (M_{LCG}*ranLast + A)%mod_{LCG}

LeapFrog: Same sequence with many threads.

- We can use the leapfrog method to generate the same answer for any number of threads.
- These results are for Leapfrog with the MLCG generator (r_{new} = (Mult*r_{last})%Mod)

Samples	One thread	2 threads	4 threads
1000000	3.139852	3.139852	3.139852
5000000	3.140930	3.140930	3.140930
10000000	3.140884	3.140884	3.140884
50000000	3.141199	3.141199	3.141199
100000000	3.141348	3.141348	3.141348

- Used two streams of pseudo-random numbers … one for x and one for y. This was needed to make (x,y) pairs consistent as number of threads changed.
- Stream1 MCLG:
  - Mult: 1583458089
  - Mod: 2147483647
  - Seed: 2147483647
- Stream 2 MCLG:
  - Mult: 295397169;
  - Mod: 1073741789
  - Seed: 7727

Linear Congruential generators are fine, but there are many other ways to generate pseudo-random numbers

Commonly used RNGs

- **Combine multiple LCGs**
  - Long sequence length (with a good choice of relatively-prime multipliers)
  - Small state, great for skipping values (leapfrog)
  - Relatively slow(!)
  - No real theoretical grounding
  - Example: Wichmann-Hill (1982) combined 3 LCGs, expanded to 4 LCGs as tests became more stringent
- **Lagged Fibonacci Generator (LFG)**
  - $S_n = (S_{n-j}$ OP $S_{n-k})\%$ m. where OP is a binary operation. eg *addition*, *subtraction*, *multiplication*, or *exclusive-or* (often with added sprinkles)
  - Choice of binary operation defines a family of generators
  - Quality and sequence length determined by the lag k (0<j<k), large values can give very long sequences but require more memory for the generator state
  - Proper initialization is particularly important

Commonly used RNGs

- **Mersenne Twister (**Matsumoto & Nishimura, 1997) is a Lagged Fibonacci Generator with exclusive-or as the binary operation
  - Often recommended as a good tradeoff between speed and quality
  - Default generator for ROOT global gRandom
  - Can have very long sequence lengths
  - LeapFrog is possible but slow and not widely implemented
  - Independent sub-sequence algorithm not formally proved (or widely implemented)
  - Weak theoretical basis - fails some of the more stringent tests of the current TestU01 suite
- **RANLUX** (Marsaglia & Zaman 1991) An additive LFG with an additional "carry" term. Has interesting mathematical properties:
  - Equivalent to LCG with a very large multiplier
  - Fails some basic RNG tests, but has a large period (2^{48}).
  - High quality but can be relatively slow (up to ~50X slower than Mersenne Twister)
  - Lüscher (1994): with some additional constraints, dynamical system with Kolmogorov-Anosov mixing … with guarantees of ergodicity, coverage, asymptotic independence
  - Has been the standard generator for HEP …"Full" detector simulations, Lattice QCD…

LFG: Lagged Fibonacci Generator

HEP: High Energy Physics

Commonly used RNGs

- **MIXMAX generator:** G. Savvidy & N. Ter-Arutyunyan-Savvidy (1986):
  - A dynamical system of equations that rapidly approaches asymptotic mixing
  - Naive implementation hopelessly slow. K. Savvidy (2014) found tricks and optimizations that yield fast linear performance
  - When stored state is large (≥240 64-bit words):
    - Speed competitive with Mersenne Twister for a single iteration
    - Asymptotic mixing in ≈5 iterations
    - Sequence long enough ($>10^{4839}$) to allow guaranteed independent sub-sequences
    - Relatively efficient skipping (n logn)
    - Slow initialization
  - Displacing RANLUX as the HEP standard for high quality random numbers:

Commonly used RNG libraries

- C++ libraries
  - rand() - Avoid except for testing
  - boost
  - CLHEP - HEP standard package for RNGs
  - STL numerics library (includes LCG, mersenne twister, ranlux) - https://en.cppreference.com/w/cpp/numeric/random
- Python libraries
  - random
  - numpy.random
- Writing codes that require random numbers? Choose carefully and be sure to understand how to properly seed generators (and how to get either reproducible or distinct results when rerunning your application)

Let’s wrap this up … random numbers are supposed to be a boring technology you just use without thinking about it.

Conclusion

- You now know how to use (and abuse) pseudo-random numbers.

- It is shockingly easy to use them incorrectly … I lack detailed survey-data but based on anecdotal evidence, I suspect a large number of published papers using Monte Carlo methods are broken due to abuse of pseudo-random numbers.

- Important rules to follow:
  - Be careful with default, built-in random number generators.
  - Know the method your generator is using and confirm that the parameters are give you the period you need.
  - Use a quality (tested/validated) generator. They are fun to write, but it's a job best left to professional.
  - Don't be stupid about using generators in parallel. There are parallel generators "out there" (such as SPRNG). Use them. http://www.sprng.org/

Be careful. There is some extremely bad advice "out there". For example, from https://luscher.web.cern.ch/luscher/ranlux/ …

The ranlux generator is widely used in Monte Carlo simulation programs. Such simulations are often performed on parallel computers, where each MPI process runs a private copy of the generator (with different seeds).