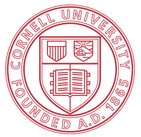
Use and Abuse of Random Numbers

Dan Riley (daniel.riley@cornell.edu)
Cornell University

(starting from slides by Tim Mattson of Intel)



Cornell University
Laboratory for Elementary-Particle Physics

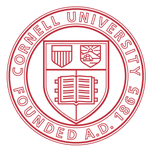
About Random Numbers

A sequence of numbers (with some distribution) where each member has no discernible correlation with the other numbers in the sequence

- Can't prove that a sequence is random, can only prove that one is not by finding a pattern
- As far as we know, some natural processes can provide "true" random number sequences (radioactive decay, thermal noise, dice, etc.)

Good random number sequences are critical for a variety of important techniques

- Monte Carlo methods use random sampling of spaces of alternatives to find statistically good answers
- Event and detector response simulation, fitting, significance tests, importance sampling



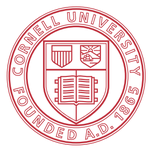
Pseudo-Random Numbers

Usually we don't actually want "true" random numbers

- "True" random numbers are expensive to collect
- Computers are deterministic—a given initial state and pre-defined sequence of instructions should always produce the same result
  - Without some auxiliary source of randomness, computers can't produce true randomness
  - Some computers do have built-in sources of thermal noise for cryptography
- Typically we want a reproducible sequence that is "random enough" wrt our process or procedure

Pseudo-random numbers are deterministic sequences

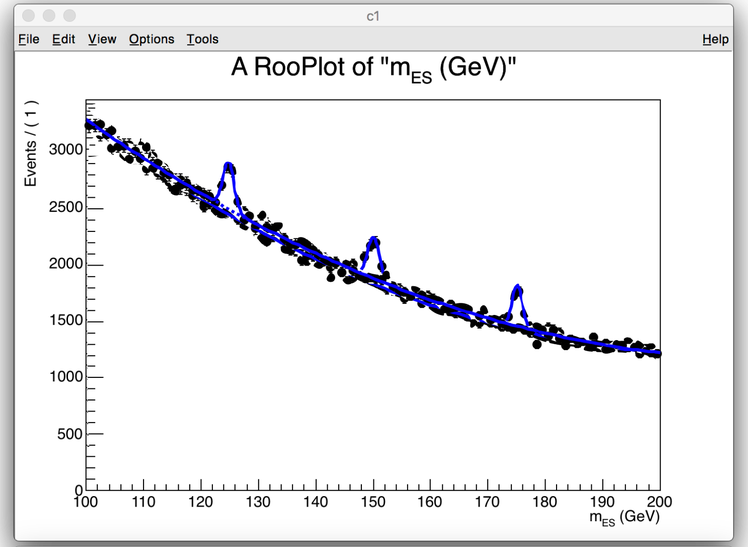
- Creating a good random number generator is hard—so don't make your own
- Speed and quality of randomness are somewhat correlated
- A random number generator has state—the amount state sets an upper bound on how long a sequence can be created before it starts repeating

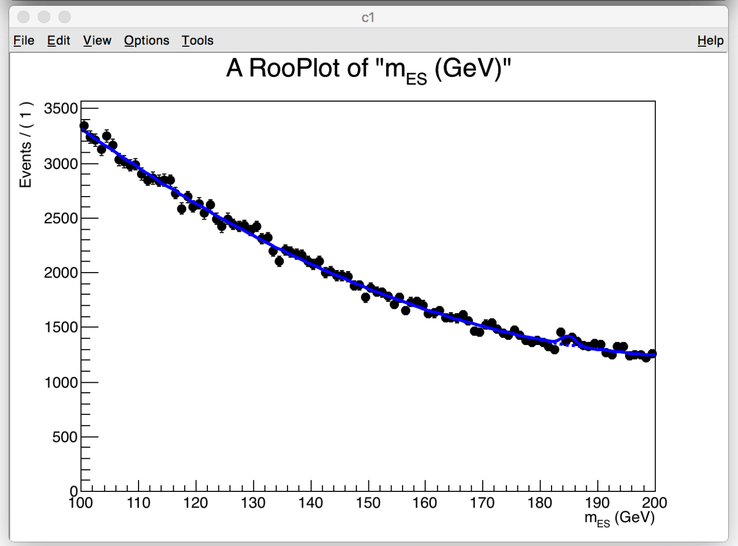


ToyMC for Significance Testing

What's the significance of an apparent signal?

- If you knew the mass beforehand, then the fit tells you the significance
- For an unknown mass, the fit gives a local significance, but the global significance depends on the probability of a fluctuation that size at any mass in the range tested
- One way to test—simulate a lot of background with no signal distributions and see how often a signal appears
- Perfect for parallelization, but good results depend on unbiased random numbers

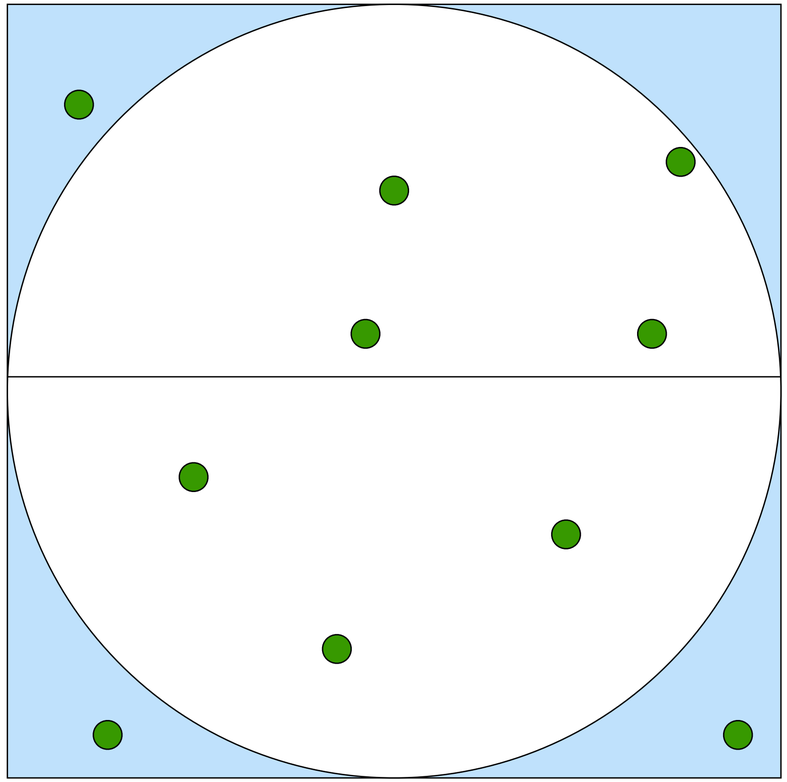




A Simple Monte Carlo Calculation

Sample a problem domain to

- Numerically integrate, estimate probabilities, find optimal values
- Example: numerical integration to find π by simulating throwing darts at circle bounded by a square
- Difference in rates is proportional to the ratio of the areas, which is proportional to π
- So randomly choose points, and count the fraction in the circle

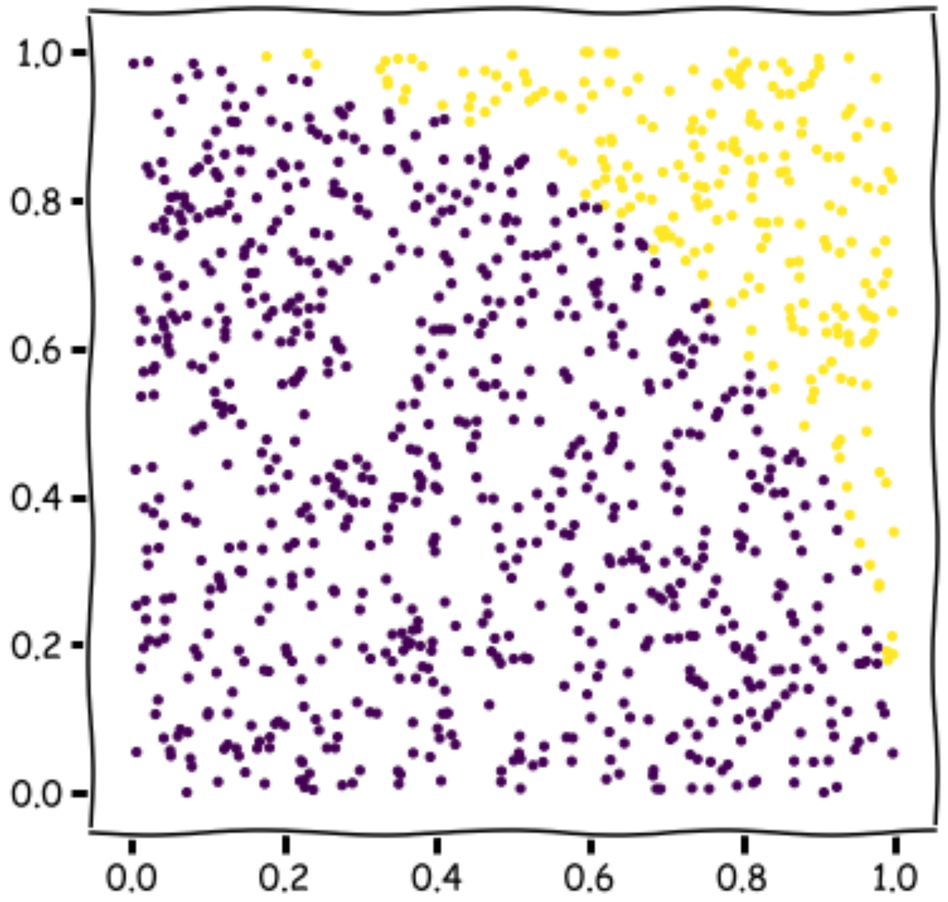


Jupyter notebook demos:

- https://github.com/dan131riley/RandomDemo.git

Python Illustration

```
import numpy as np
import matplotlib.pyplot as plt

def drawpi (points = 1000) :
    r = 1.0
    incircle = 0

    x = np.random.random(points)
    y = np.random.random(points)
    c = np.empty(points)

    for i in range(points) :
        if x[i]**2 + y[i]**2 <= r**2 :
            c[i] = 1.0
            incircle += 1
        else :
            c[i] = 2.0

    with plt.xkcd():
        plt.figure(figsize=(6,6))
        plt.scatter(x, y, c=c, s=10)
        plt.show()

    return 4*(incircle/points)
```



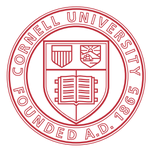
Linear Congruential Generator

LCG is easy to write, fast, adequate quality for *some* purposes

- With modulus ***m***, multiplier ***a***, increment ***c***, starting value X_0

$$X_{n+1} = (aX_n + c) \bmod m$$

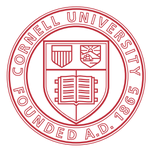
- Choice of constants is critical—$m = 10$, $X_0 = a = c = 7$ gives the sequence 7, 6, 9, 0, 7, 6, 9, 0, …
- If the constants are chosen carefully, the sequence length is set by the modulus
- For this example, using $a = 1366$, $m = 714025$, $c = 150889$ from a standard reference of reasonably good values
  - Note the relatively short cycle length—if my ToyMC has 200,000 events per try, only gives 3 independent tries!
- Note: LCG generally is not good enough for many simulation purposes!
  - Alternatives are nearly as fast, but more complicated, typically have more state and much longer sequence lengths (e.g., Mersenne Twister, MIXMAX)
  - But many naive implementations (including the default C libraries) are often LCGs.



Example LCG Implementation

for example purposes only…

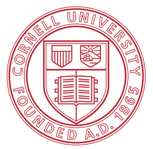
```
class LCG:
    def __init__(self, seed = 1, Multiplier = 1366, Addend = 150889, Pmod = 714025):
        self.multiplier = Multiplier
        self.addend = Addend
        self.pmod = Pmod
        self.last = seed

    def random(self):
        self.last = (self.multiplier * self.last + self.addend) % self.pmod
        return self.last/self.pmod

lcgdef = LCG()
lcgdef.random()
0.213234830713210 3
```



Calculating π Demo

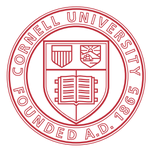
```
def calcpi(lcg, num_trials = 10000):
   incircle = 0

    for i in range(num_trials) :
        x = lcg.random()
        y = lcg.random()
        if x*x + y*y <= 1.0 :
            incircle += 1

    return 4*(incircle/num_trials)

import math

for decade in range(8) :
    trials = 10**decade
    pi = calcpi(lcgdef, trials)
    print(f'{trials:8d} trials pi = {pi:.5f} deviation {abs(pi-math.pi):.5f}')
```

Trials	PI	Deviation
1	4.00000	0.85841
10	3.20000	0.05841
100	2.92000	0.22159
1000	3.06800	0.07359
10000	3.12320	0.01839
100000	3.14524	0.00365
1000000	3.14205	0.00046
10000000	3.14158	0.00002



A Bad Choice of LCG Parameters

```
lcgdef.setseed(0)

trials = 5*10**6
pi = calcpi(lcgdef, trials)
print(f'Ok:  {trials:8d} trials pi = {pi:.5f} deviation {abs(pi-math.pi):.5f}')

lcgbad = LCG(seed = 1, Multiplier = 65539, Addend = 0, Pmod = 2**31)

pi = calcpi(lcgbad, trials)
print(f'Bad: {trials:8d} trials pi = {pi:.5f} deviation {abs(pi-math.pi):.5f}’)


Ok:   5000000 trials pi = 3.14158 deviation 0.00002
Bad:  5000000 trials pi = 3.14216 deviation 0.00057
```

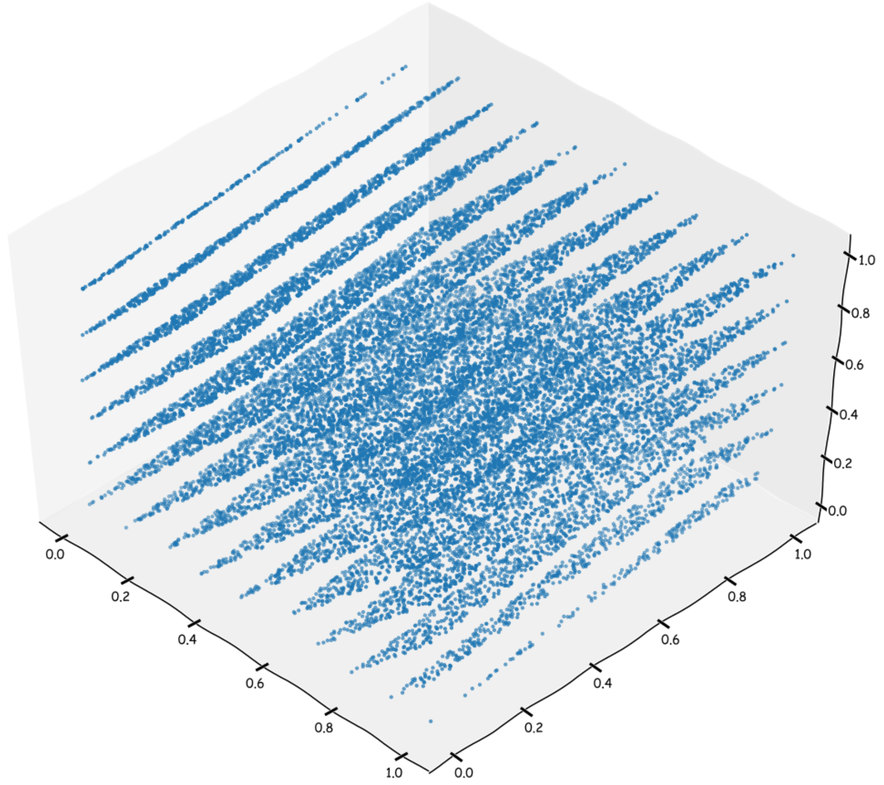


What's wrong with it?

LCG's generate tuples that lie on parallel hyperplanes

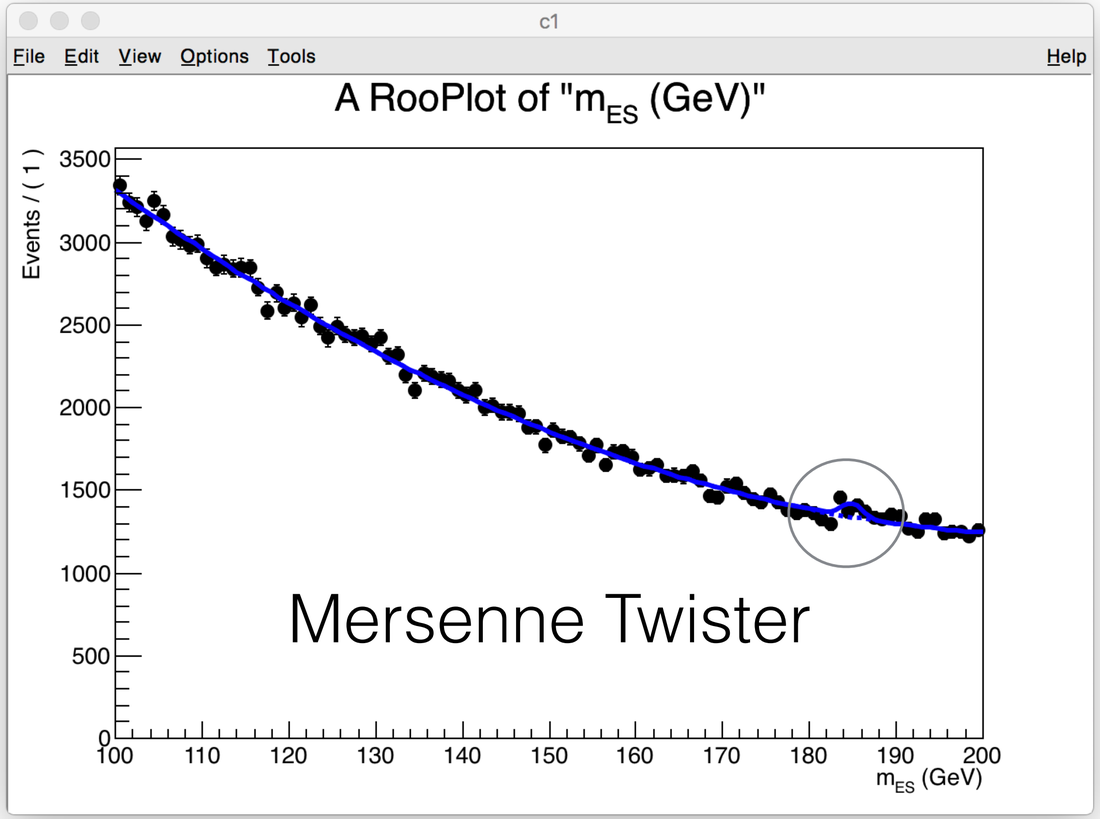
- # of dimensions of the tuple depends on the choice of parameters
- This notoriously bad choice, IBM's RANDU, was widely used in the 60s and 70s
  - RANDU generates correlated 3-tuples

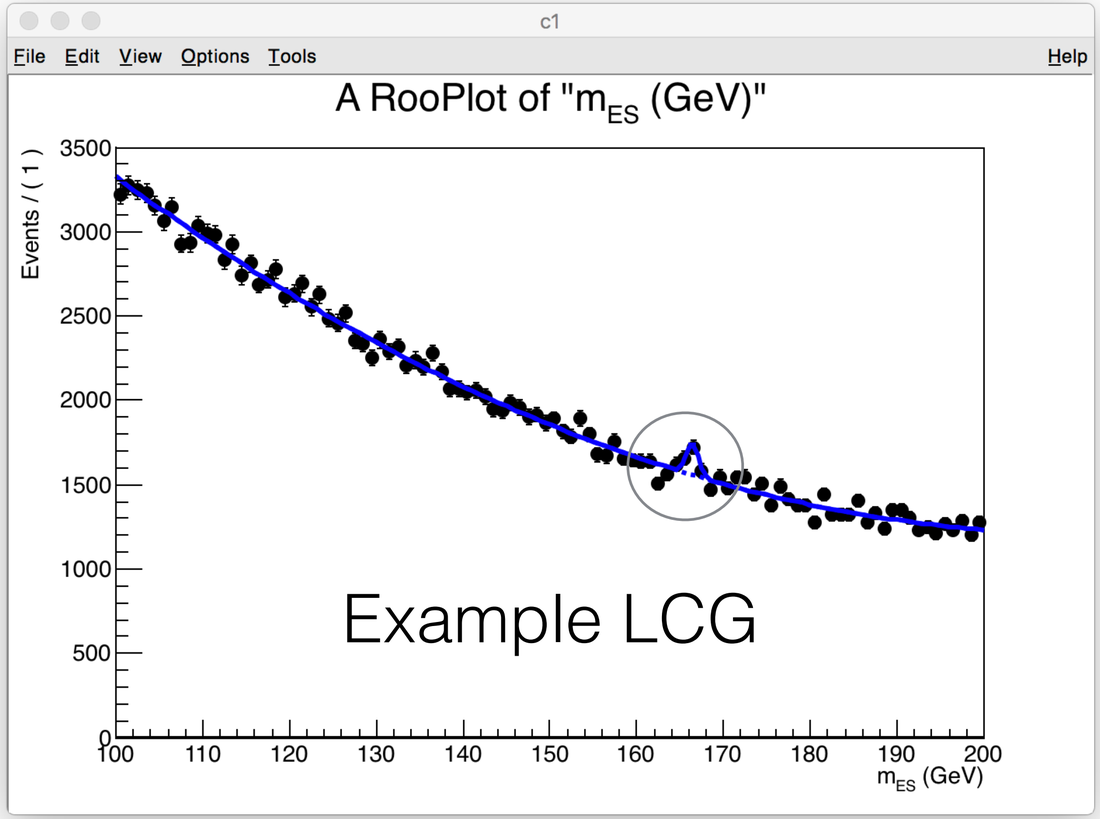
```
from mpl_toolkits.mplot3d import Axes3D

def draw3d(lcg, points = 20000):
    v = np.empty((3, points))

    for i in range(points) :
        for j in range(3) :
            v[j][i] = lcg.random()

    with plt.xkcd():
        fig = plt.figure(figsize=(20,16))
        ax = fig.add_subplot(111, projection='3d')
        ax.view_init(elev=50, azim=-45)
        ax.scatter(v[0], v[1], v[2], s=10, zdir='y')
        plt.show()

draw3d(lcgbad)
```



Anecdotal signal scanning

Manually scanned some background-only toy MC events looking for fake signals (note: not a scientific tests)

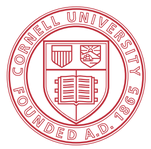




Multi-processing with Random Numbers

C++ LCG implementation to illustrate potential multi-processing and threading issues:

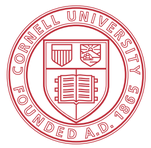
```
static uint32_t random_last = 0;  // internal state
double lcg_rand()
{
  static constexpr uint32_t kMultiplier  = 1366;
  static constexpr uint32_t kAddend      = 150889;
  static constexpr uint32_t kPmod        = 714025;

  random_last = (kMultiplier * random_last + kAddend) % kPmod;
  return ((double)random_last/(double)kPmod);
}
```



Parallel π

Calculating π this way is embarrassingly parallel, so make it parallel:

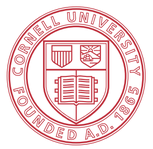
```
std::atomic<long> Ncirc{0}; // updates are “critical”

// #pragma omp parallel reduction (+:Ncirc)
tbb::parallel_for(tbb::blocked_range<size_t>(0, num_trials),
  [&](const tbb::blocked_range<size_t>& range){
    long Nlocal{0};
    for(auto i = range.begin(); i != range.end(); ++i) {
      const auto x{lcg_rand()}; const auto y{lcg_rand()};
      if ((x*x + y*y) <= 1.0) Nlocal++;
    }
    Ncirc += Nlocal;
  }
);
double pi = 4.0 * ((double)Ncirc/(double)num_trials);
std::cout << num_trials << " trials, pi " << pi << std::endl;
```



Some technological notes

I'm using standard C++14 and Intel Threading Building Blocks (TBB)

- **Tim's version used a simple for loop with an OpenMP**
  #pragma omp parallel reduction (+:Ncirc)
- **Mine uses std::atomic and tbb::parallel_for with a lamba expression**
  - std::atomic is not guaranteed to be lock free on all platforms and compilers so treat it like an OMP critical section!
- **OpenMP is very common in the HPC community**
  - works across languages
  - easier and possibly more efficient for simple cases and existing code
  - but doesn't support the latest C++ standards
  - complex uses change the language semantics!
- **TBB is C++-only, compatible with the latest C++ standards**
  - the ROOT project and the CMS experiment (and other LHC experiments) have adopted C++11 (or later) with TBB for threading
  - TBB is open source, can be implemented in standard C++17—not locked to a vendor
- **Concepts are similar**

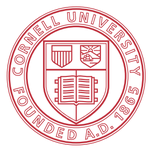


Trials with 4 threads

```
1000000 trials, pi 3.13155 error 0.01004
1000000 trials, pi 3.1338  error 0.00778
1000000 trials, pi 3.1324  error 0.00918
1000000 trials, pi 3.13148 error 0.01011
1000000 trials, pi 3.13253 error 0.00906
1000000 trials, pi 3.13451 error 0.00708
1000000 trials, pi 3.13293 error 0.00866
1000000 trials, pi 3.1335  error 0.00809
1000000 trials, pi 3.13416 error 0.00742
```

Same program, run the same way

- Different results every time!
- Error is much too large for 1000000 trials
- Problem 1: lcg_rand() isn't thread safe!

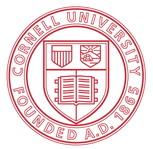


LCG Race

```
static uint64_t random_last = 0;  // internal state
double lcg_rand()
{
  random_last = (kMultiplier * random_last + kAddend) % kPmod;
  return ((double)random_last/(double)kPmod);
}
```

`random_last` is shared state across threads

- Race condition between using the old value and setting the new value can result in values being reused
- Race condition thus biases the results
- Change so each thread has its own copy:
  `thread_local` static long random_last = 0;

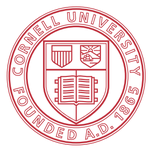


With thread_local, 4 threads

```
1000000 trials, pi 3.14021 error 0.00138
1000000 trials, pi 3.14047 error 0.00112
1000000 trials, pi 3.14076 error 0.00083
1000000 trials, pi 3.14047 error 0.00112
1000000 trials, pi 3.14024 error 0.00135
1000000 trials, pi 3.14052 error 0.00106
1000000 trials, pi 3.14014 error 0.00145
1000000 trials, pi 3.14044 error 0.00114
1000000 trials, pi 3.14051 error 0.00108
```

Same program, run the same way

- More consistent, but still different results every time!
- Error is smaller, but still too large for 1,000,000 trials
- Two more problems: sequence reuse and TBB
  - TBB doesn't guarantee a consistent mapping of tasks to threads
  - Every thread is starting with the same seed, so we're doing the same 250,000 trials 4 times!

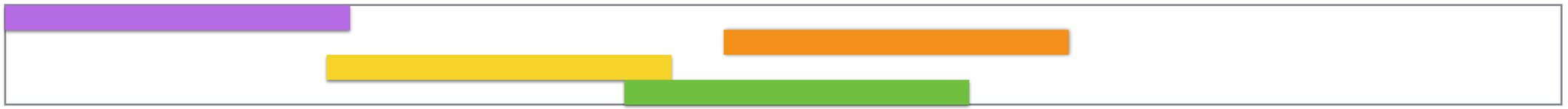


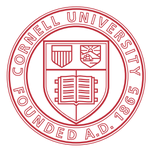
Overlapping Sequences

Typical single-thread usage uses a contiguous subsequence:

Our threaded example reuses a subsequence:

Generating seeds “randomly” can still lead to overlapping sequences that may bias the results (and you won’t know how much):

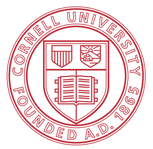




Parallel Random Number Generators

Possible solutions:

- **Every thread has its own unique generator**
  - These could be different generators from the same family, e.g. different LCG constants
  - Mersenne Twister has a “dynamic creator” scheme for this, but not formally proved
- **One thread generates all the random numbers**
  - Coordination overhead, blocking
- **Block methods, where every thread gets a well-defined subset of the sequence**
  - Sequence splitting allocates contiguous blocks so that each thread gets its own non-overlapping block
  - Leapfrog divides the sequence round-robin, so thread ***n*** gets the sequence of entries where the sequence number modulo the number of threads is ***n***
  - Block methods can be implemented efficiently for LCG and some other generators where jumping ahead is simple, less efficiently for others like Mersenne Twister
- **Most of these are tricky to implement correctly, so use a library**

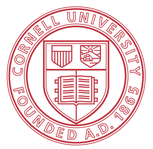


Not taking my own advice…

After some (painful) work…

- **Give each thread a different seed via leapfrog**
- **Specify the stride of the TBB loop so that only one task is created for each thread**
- **This isn't actually sufficient for complete reproducibility as the changing leapfrog stride changes the x,y pairings—need independent streams (use a library!)**
  - Achieved reproducibility for a given thread count, and comparable numerical performance, but not reproducibility with different thread counts

Steps	One Thread	Two Threads	Four Threads
1000	3.124	3.156	3.112
10000	3.128	3.1512	3.1428
100000	3.1456	3.13804	3.14488
1000000	3.14138	3.14102	3.14188

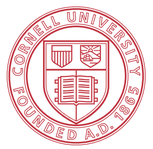


Independent Streams with TRandomMixMax

ROOT implementation of MIXMAX random number generator

- **MIXMAX is a relatively recent RNG with strong theoretical guarantees and *efficient* skipping**
- **Create # of independent streams > max # of threads**
  - TRandomMixMax::SetSeed() creates an independent stream, or in the constructor TRandomMixMax rng(streamid)

Steps	One Thread	Two Threads	Four Threads
1000	3.172	3.172	3.172
10000	3.1636	3.1636	3.1636
100000	3.14412	3.14412	3.14412
1000000	3.14095	3.14095	3.14095
10000000	3.14159	3.14159	3.14159



Summary

Pseudo-random numbers are widely used

- We depend on generators that are unbiased wrt the simulation process

Getting them right is tricky

- Pseudo-random numbers are deterministic and have correlations

Parallel programming creates new opportunities for biasing results

- For many applications, have to ensure that parts of the random number sequence are not unintentionally reused