Use and Abuse of Random Numbers

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

Cornell University
Laboratory for Elementary-Particle Physics

About this Talk

Mostly lecture

- Haven't thought of good exercises that would actually teach about the topic
- There is a Jupyter notebook with most of the figures (and a possible exercise)
    - https://github.com/dan131riley/RandomDemo.git

What we'll cover

- Train wreck stories—things that can go wrong if you aren't careful
    - For illustration purposes, will look at a bad pseudo-random number generator that used to be commonly used
- How to pick a good random number generator
    - Standard test suites
    - Understanding requirements
- Some standard or recommended pseudo-RNGs
    - With a little bit of theory

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
- Some natural processes appear to 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
  - Physical processes take time
- 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, primarily for cryptography
- Typically we want a reproducible sequence that is "random enough" wrt our process or procedure
  - Reproducibility can be important for debugging and regression testing

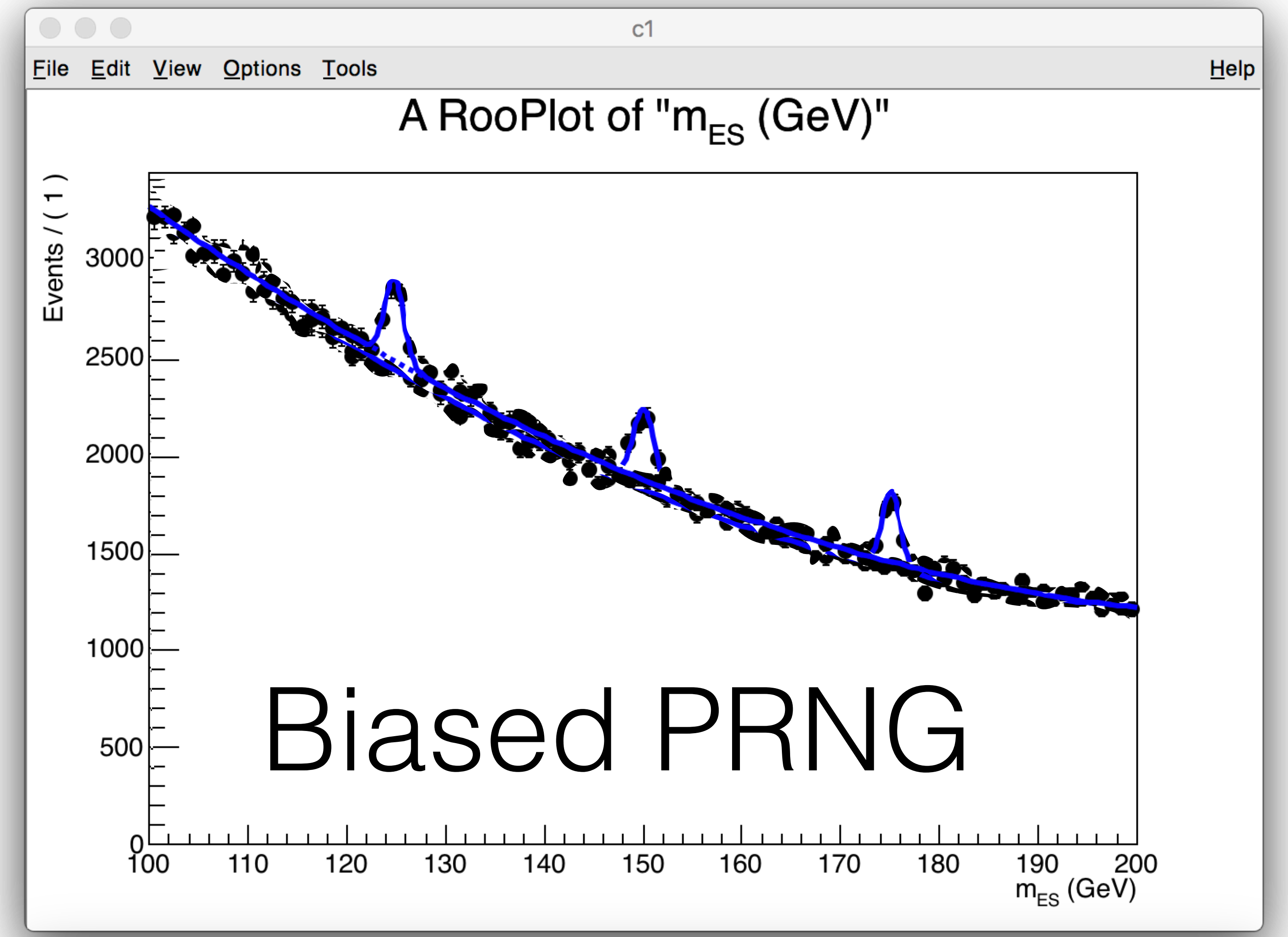
Pseudo-random numbers are deterministic sequences

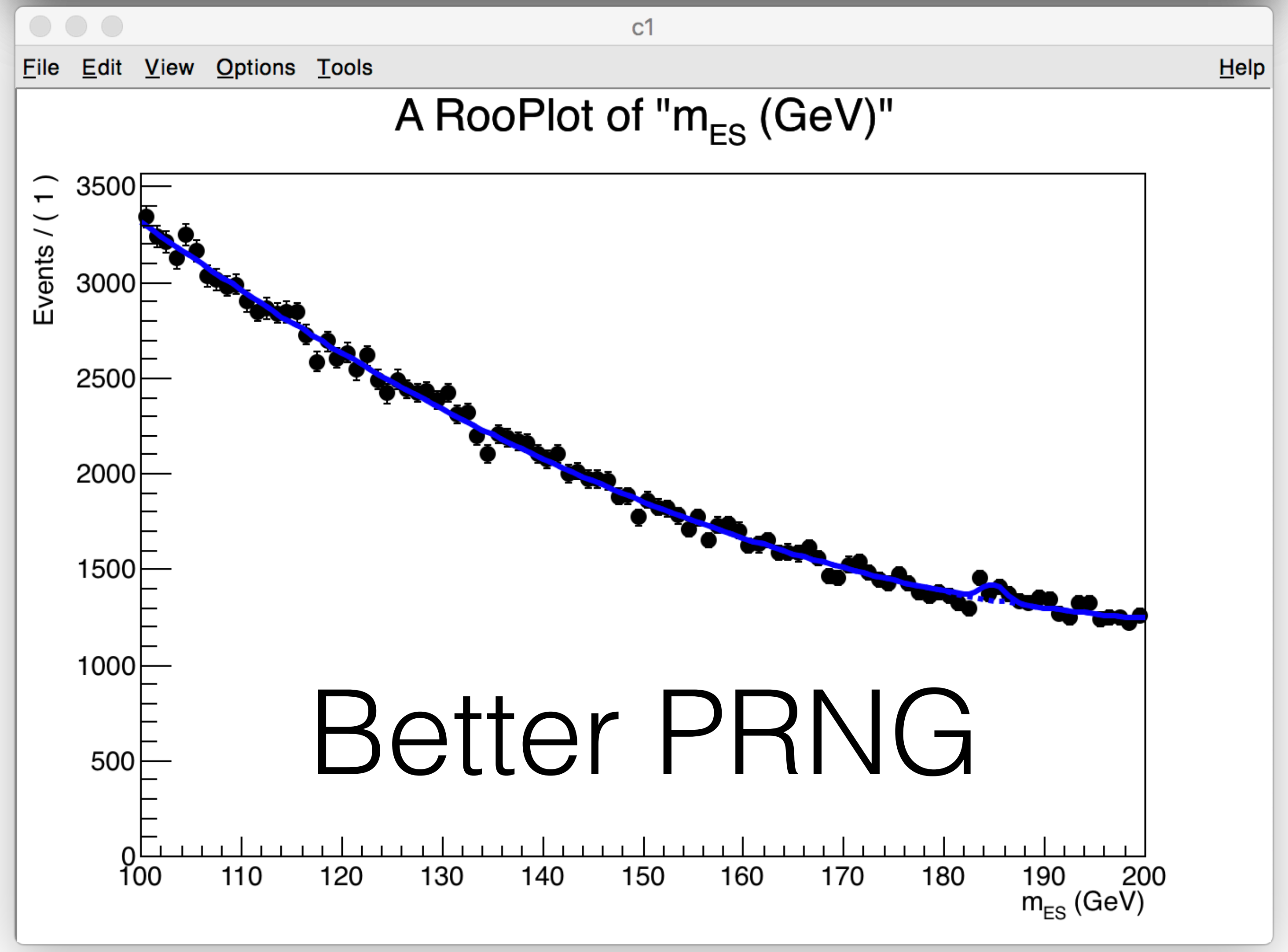
- Speed, memory consumption and quality of randomness are somewhat correlated
  - We'll look at this more towards the end of the talk
- A random number generator has state—the amount of state sets an upper bound on how long a sequence can be created before it starts repeating
  - Every PRNG has finite state, so they all repeat eventually
- Creating a good random number generator is hard

Why it matters…

Toy Monte Carlo example: what's the significance of an apparent signal?

- If you know the mass beforehand, then the fit tells you the significance
- For an unknown mass, the fit gives a local significance
  - global significance depends on the probability of a fluctuation that size at **any** mass in the range tested (look elsewhere effect)
- Simulate a lot of background with no signal and see how often a signal appears
  - Perfect for parallelization, but good results depend on **unbiased** random numbers





Linear Congruential Generator

Linear Congruential Generators (LCGs) are fast and simple, and were widely used.

- 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
  - There are standard collections of reasonable values
- For our examples, we'll use $a = 1366$, $m = 714025$, $c = 150889$
  - Note the relatively short cycle length—if my ToyMC needs 200,000 random numbers per try, can only get 3 independent tries (folklore says less than 3).
- Many naive implementations (including many default C libraries) are still LCGs.

Notebook LCG Implementation

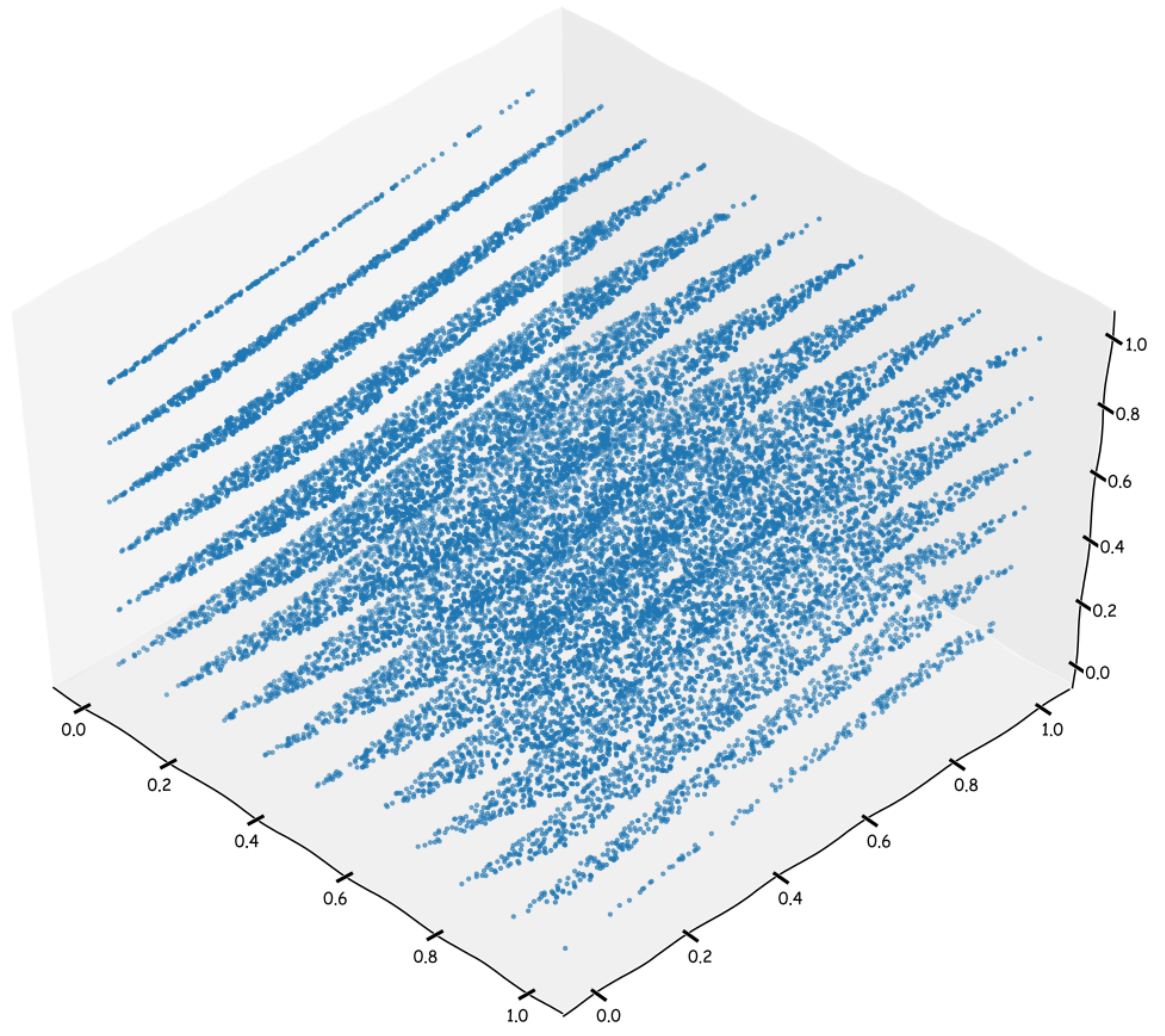
```
class LCG:
    def __init__(self, seed = 1, Multiplier = 1366, Addend = 150889, Pmod = 714025):
        """Create an LCG instance"""
        self.multiplier = Multiplier
        self.addend = Addend
        self.pmod = Pmod
        self.setseed(seed)

    def random(self):
        """Return a single random number between 0 and 1"""
        self.last = (self.multiplier * self.last + self.addend) % self.pmod
        return self.last/self.pmod

    def randv(self, size = 1):
        """Return a numpy array of random numbers between 0 and 1"""
        v = np.empty(size)
        for i in range(size):
            v[i] = self.random()

lcgdef = LCG()
lcgdef.random()
0.2132348307132103
```

An infamously bad choice

LCG's (and many related PRNGs) generate tuples that lie on parallel hyperplanes

- # of dimensions of the tuple depends on the choice of parameters
- One notorious choice, IBM's RANDU, was widely used in the 60s and 70s
    - RANDU generates correlated 3-tuples
    - Hyperplanes are widely spaced (spectral test)

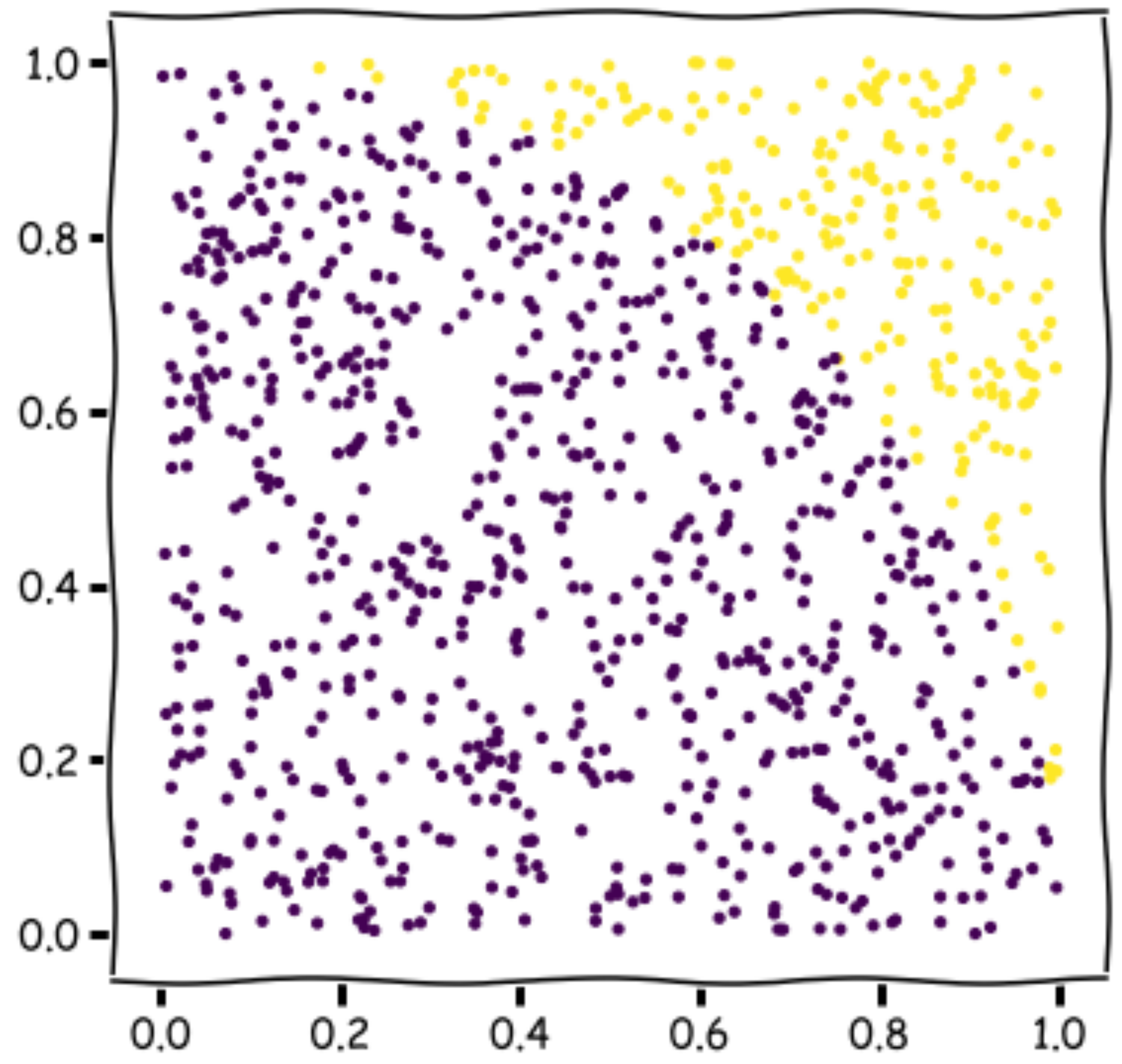
```
lcgbad = LCG(seed = 1, Multiplier = 65539,
             Addend = 0, Pmod = 2**31)

def draw3d(lcg, points = 20000):
    """Draw a 3d scatterplot of the given LCG"""
    v = lcg.randv(3*points)

    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[0:3*points:3],
                   v[1:3*points:3],
                   v[2:3*points:3], s=10, zdir='y')
        plt.show()

draw3d(lcgbad)
```



Tim’s Favorite Example — π

Monte Carlo integration to find π by simulating throwing darts at a circle bounded by a square

- Difference in rates is proportional to the ratio of the areas, which is proportional to π



Jupyter notebook demos:

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

Calculating π Demo

```
def calcpi(lcg, num_trials = 10000):
    """Calculate PI with the given LCG"""

    x = lcg.randv(num_trials)
    y = lcg.randv(num_trials)

    incircle = np.sum(x**2 + y**2 <= 1.0)

    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

But there’s a mystery here—haven’t we exhausted the PRNG state?

Random Numbers in Parallel

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;
```

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: is lcg_rand() 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 biases the results
- Solutions: give each thread its own local copy:

  thread_local static long random_last = 0;

  or externalize or encapsulate the state in an object per thread

  - Externalizing or encapsulating state is more flexible, especially for maintaining reproducibility

With thread_local, 4 threads

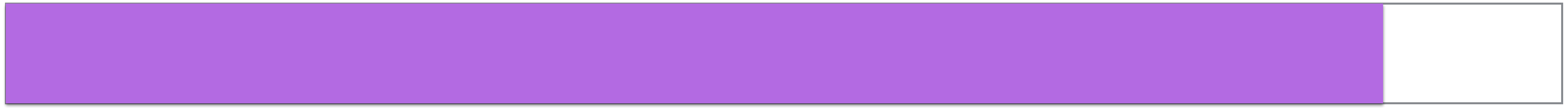
```
1000000 trials, pi 3.14021 error 0.00138
1000000 trials, pi 3.14047 error 0.00112
1000000 trials, pi 3.14076 error 0.00083
1000000 trials, pi 3.14047 error 0.00112
1000000 trials, pi 3.14024 error 0.00135
1000000 trials, pi 3.14052 error 0.00106
1000000 trials, pi 3.14014 error 0.00145
1000000 trials,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

Same program, run the same way

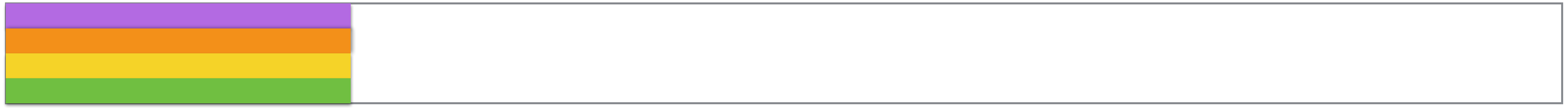
- More consistent, but still different results every time!
- Error is smaller, but still too large for 1,000,000 trials
- Two more problems: sequence reuse and TBB task/thread mapping
  - Every thread is starting with the same seed, so we're doing the same 250,000 trials 4 times!
  - TBB doesn't guarantee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

Overlapping Sequences

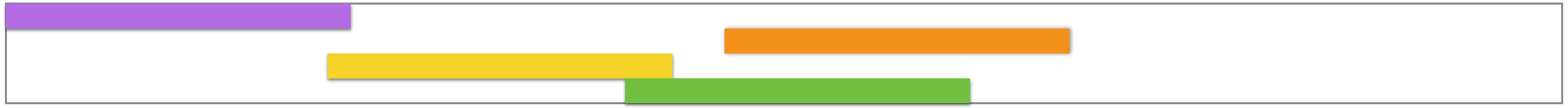
Typical single-thread usage uses a contiguous subsequence:



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 by 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
  - Potential for unexpected interactions
- One thread generates all the random numbers
  - Coordination overhead, blocking
- Block methods, where every thread gets a well-defined subset of a sequence
  - Sequence splitting allocates contiguous blocks so that each thread gets its own non-overlapping sub-sequence
  - 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
- Most of these are tricky to implement correctly
  - Getting reproducible results independent of the number of threads can be especially tricky

Recognizing a good PRNG

There are standard test suites with published results for most common PRNGs:

- TestU01 is the current standard
- Tests are largely empirical, from problems where PRNGs have been observed to fail
- Demands for quality PRNGs increase with computing power, so tests have also become more stringent to keep pace

Understanding requirements:

- PRNG quality
  - Can be measured according to standard empirical tests or application specific
- Sequence length
- Reproducibility
  - Usually simple for a sequential program, tricky for parallel programs
- Sub-sequence independence
  - For allocating a sub-sequence per thread

Some Standard PRNGs

How to make a good quality PRNG? A couple of common possibilities…

- **Combine multiple LCGs**
  - Long sequence length (with a good choice of relatively-prime multipliers)
  - Small state, fast skips
  - 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):** $S_n \equiv S_{n-j} \circledast S_{n-k}$, **where** $\circledast$ **can be 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, large values can give very long sequences but require more memory for the generator state

LFG example 1: Mersenne Twister

Mersenne Twister (Matsumoto & Nishimura, 1997):

- 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
- Skipping 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

LFG 2: RANLUX

Marsaglia & Zaman (1991) published RCARRY, an additive LFG with an additional “carry” term

- Not a good quality PRNG, failed many tests, but…

Some interesting mathematical properties:

- Equivalent to LCG with a **very** large multiplier
- Matrix representation: $a(i+1) = A \times a(i) \bmod 1$, where $a(i)$ is a vector of size k
  - Lüscher (1994): with some additional constraints, dynamical system with Kolmogorov-Anosov mixing ⇒ mixing theory provides guarantees of ergodicity, coverage, asymptotic independence
  - James (1994 CERNLIB implementation): throw away enough iterations to approach asymptotic independence (“luxury” level ≈ 15 iterations for a common choice of lags)
  - At high luxury levels, very high quality but relatively quite slow (×50 Mersenne Twister)

Has been the standard generator for HEP where highest quality is needed

- “Full” detector simulations, Lattice QCD…

MIXMAX

G. Savvidy & N. Ter-Arutyunyan-Savvidy (1986):

- Design a dynamical system family that rapidly approaches asymptotic mixing
- Naive implementation hopelessly slow
- K. Savvidy (2014) found tricks and optimizations that yield fast linear performance

Properties at high quality (state $\geq$ 240 64-bit words):

- Speed competitive with Mersenne Twister for a single iteration
- Asymptotic mixing in ≈ 5 iterations
- Sequence long enough ($\gtrsim 10^{4839}$) to allow guaranteed independent sub-sequences
- Relatively efficient skipping ($n \log n$)
- Slow initialization

Displacing RANLUX as the HEP standard for high quality random numbers:

- CLHEP default, CMS choice for both fast and full sim

Independent Streams with TRandomMixMax

ROOT implementation of MIXMAX random number generator

- **Create # of independent streams > max # of threads**
  - TRandomMixMax::SetSeed() creates an independent stream, or set 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 calculation they are used for

Generating high quality pseudo-random numbers efficiently is hard

- Pseudo-random number generators are deterministic and have correlations
- No PRNG is perfect for every application, and many are quite bad

Parallel programming creates new opportunities for biasing results

- Reusing parts of the sequence can introduce biases
- Reproducibility can be tricky

MIXMAX emerging as a good choice for many applications

- Can match Mersenne Twister for speed and RNG quality
- Can achieve very high quality much more efficiently than RANLUX
- Efficient skipping and astronomical sequence length simplify independent subsequences

Backup Slides

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

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: #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 historically hasn't supported 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 (and OpenMP is arguably simpler for an introductory course)

Independent Streams with TRandomMixMax

```
const int streams(8);  // maximum parallelism
const size_t chunksize = num_trials/streams;
std::atomic<long> Ncirc{0};
std::atomic<long> streamid{0};

tbb::parallel_for(size_t(0), num_trials, chunksize, [&](size_t j){
    TRandomMixMax rng(++streamid);
    long Nlocal{0};
    for (int i = 0; i < chunksize; ++i) {
      const auto x{rng.Rndm()}; const auto y{rng.Rndm()};
      if ((x*x + y*y) <= r*r) { ++Nlocal; }
    }
    Ncirc += Nlocal;
  }
);
```