Randomness in Scientific Computing

Ramses van Zon

PHY1610H, Winter 2024

Today's class

Today we will discuss:

- Randomness, why you want it.
- How to make it or fake it.
- Applications: Monte Carlo

Why Randomness?

Why Randomness?

- To simulate some physical phenomenon that has noise.

  E.g. Brownian motion, Nyquist noise.

  On the level of their description, this is real randomness.
- To perform averages or integrals in systems with many degrees of freedom.

  E.g. Stat. Phys. computations, path integral calculations.

  Here, the main objective is to get the converged answer quickly.
- To estimate a parameter's distribution from using data (MCMC).
- To test a statistical method.

Creating Randomness

Sources of randomness

True Random Number Generators

- Lava lamps.
- Radioactive decay.
- Various quantum processes.
- Atmospheric noise.
- Random computer hardware noise signals (thermals noise).

Generally slow, expensive, impossible to reproduce for debugging. Hard to characterize underlying distribution.

Pseudo Random Number Generators

- Come up with a algorithm that produces random numbers
- But wouldn't such an algorithm would be deterministic?
- Only has to **act** random, i.e., give fair and uncorrelated sequence.

Pseudo Random Number Generators (PRNG)

Recipe:

- Define some ‘state’, initialized by some ‘seed’ value(s).
- Produce a number from this state.
- Advance the state determistically.
- As long as the numbers produces behave as if they are
  - independent
  - identically distributed
  - according to a predefined distribution (eg uniform)

  we will be satisfied.

Depends a lot on the way the states are advanced. **Must test.**

Distributions are transformations

- Suppose we had a way to draw random values of a continuous variable x that is uniformly distributed between 0 and 1.
- Let's say that for any value x that is drawn, we were to compute a value $y = f(x)$, where f is a deterministic function.
- The values of y are also randomly distributed, but with a non-uniform distribution (unless $f(x) = x$).

So we can turn a uniformly distributed random variable into a non-uniformly distributed variable by applying a function.

If we want a specific non-uniform distribution, we just need to figure out the function. For many common cases, this is already done.

So our main focus is first to find uniformly distributed variables.

All pseudo random numbers are discrete

Despite the illusion of continous variables that floating point numbers give, there are only a finite number of bits, and thus a discrete set of values.

In fact, routines that give pseudo random floating point numbers are usually based on drawing a random integer number and dividing it by the largest possible generated integer.

From a random integer of n bits, we just need each bit to be uniformly distributed, with a chance of 50% of a 0 and 50% of a 1.

Warning: most PRNGs give lower bits that are more correlated than the higher bits.

Example: Coin Toss

The following class can produce a ‘random’ 1’s and 0’s representing heads and tails:

```
// badcoin.h
class BadCoin {
 public:
  // method to set the starting seed
  void start(unsigned int seed) {
   state = seed;
  }
  // method to toss the coin (1:head, 0:tail)
  int toss() {
   state++; // update state
   return state%2; // using lowest bit...
  }
 private:
  unsigned int state; // internal state
};
```

```
#include <iostream>
#include "badcoin.h"
int main()
{
  BadCoin coin;
  coin.start(13); //seed
  // toss the coin 20 times
  for (int i = 0; i < 20; i++)
    std::cout << coin.toss() << '\n';
  return 0;
}
```

What does this give?

- Is it fair? Independent samples? Period?

Testing for randomness

Suppose we have drawn N samples using our PRNG.

Let's look at two tests:

① Fairness: histogram counting the occurance of values

$$h_x = \sum_{i=1}^{N} \delta_{x x_i}$$

Here x is one of the possible random numbers (here ± 1), and x_i are samples produced by our PRNG ($\delta_{ii} = 1, \delta_{i,j\neq i} = 0$).

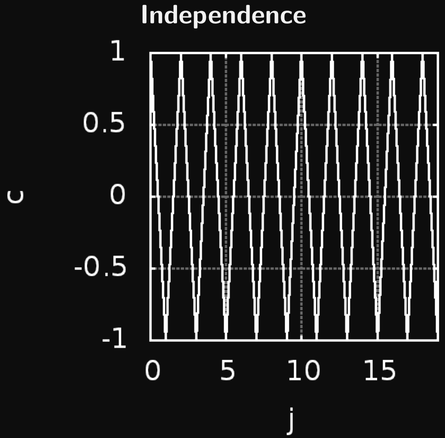
② Independence: look at correlations between samples:

$$c_j = \langle x_i x_{i+j} \rangle = \frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{x})(x_{i+j} - \bar{x})$$

If independent: $\mathcal{O}(1/\sqrt{N})$ for $j > 0$

Test results (N=20)

Good!



Bad!

Try again

Old version

```
// badcoin.h
class BadCoin {
 public:
  // method to set the starting seed
  void start(int seed) {
   state = seed;
  }
  // method to toss the coin (1:head, 0:tail)
  int toss() {
   state++; // update state
   return state%2; // using lowest bit...
  }
 private:
  unsigned int state; // internal state
};
```

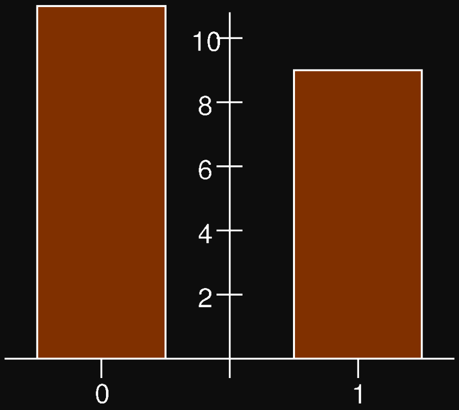
New version

```
// improvedcoin.h
class ImprovedCoin {
 public:
  // method to set the starting seed
  void start(int seed) {
   state = seed;
  }
  // method to toss the coin (1:head, 0:tail)
  int toss() {
   state=100+100*sin(state+1);//update state
   return state%2; // using lowest bit...
  }
 private:
  unsigned int state;
};
```

Difference lies in a more complex update of the state.

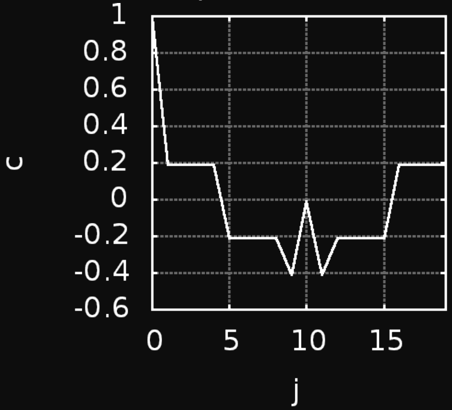
“Improved” test results (N=20)

Fairness



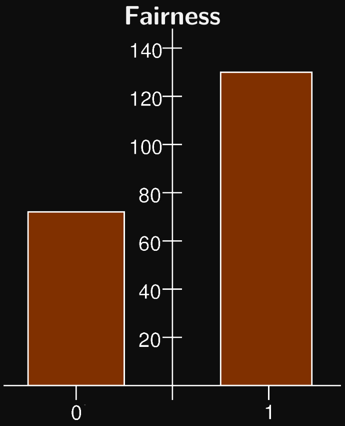
Less fair

Independence

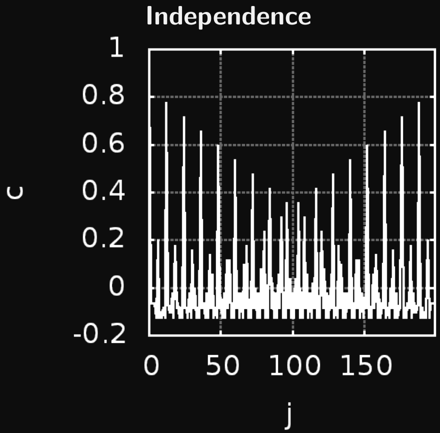


Better?

Let's do more samples: N=200



Bad



Bad

Common RNG Types

Linear Congruential generators

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

The quality of the random numbers depends on the parameters (a, c, m).
Even the best ones are not very good, but they can be used as part of better generators.

Lagged-Fibonacci generator

$$x_i = (x_{i-j} \circ x_{i-k}) \bmod m$$

where $\circ$ can be any binary operator (add, mult, . . .). Requires a seed block from another PRNG.

Mersenne Twister

A complex variation of lagged-Fibonacci that stikes a great balance between speed and statistical tests.

. . .

Well Equidistributed Long-period Linear (WELL) generators

Other tests

- Check moments of distributions.
- Check that spacings between random points follow a Poisson integral if uniformly distributed.
- Examine sequences of 5 numbers. There are 120 ways to sort 5 numbers. The 120 ways should occur with equal probability.
- Parking circle test: randomly place unit circles in a 100 x 100 square. If the circle overlaps an existing one, try again. After 12,000 tries, the number of successfully "parked" circles should follow a certain normal distribution.
- Play 200,000 games of a dice game, counting the wins and number of throws per game. Each count follow a certain distribution.
- And many others. See, for example, the NIST test suite:
  https://csrc.nist.gov/projects/random-bit-generation
  and the TestU01 suite:
  http://simul.iro.umontreal.ca/testu01/tu01.html

Lesson: Don't do it yourself

What properties do we expect from a random number generator?

- We would like them from a given distribution (uniform, Gaussian).
- We would like them to be unpredictable.
- We would like them to be reproducible.
- We need them to be generated quickly.
- We need to have a long period.

It is not that easy to guess good PRNG algorithms and parameters.

There was a time when one was forced to implement PRNGs oneself, as standard ones were quite bad, but C++ has decent random number generators in its `<random>` standard library.

Using existing random numbers

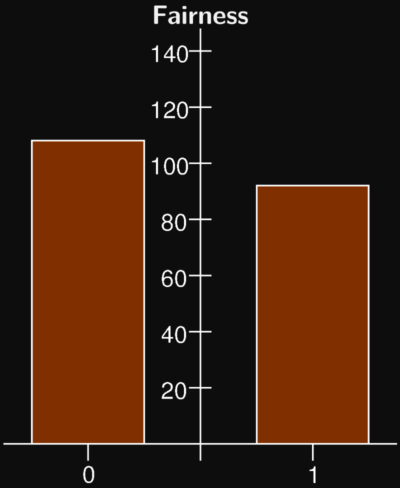
Previous way

```
// improvedcoin.h
class ImprovedCoin {
 public:
  // method to set the starting seed
  void start(int seed) {
   state = seed;
  }
  // method to toss the coin (1:head, 0:tail)
  int toss() {
   state = 100+100*sin(state+1); // update state
   return state%2; // using lowest bit...
  }
 private:
  unsigned int state;
};
```

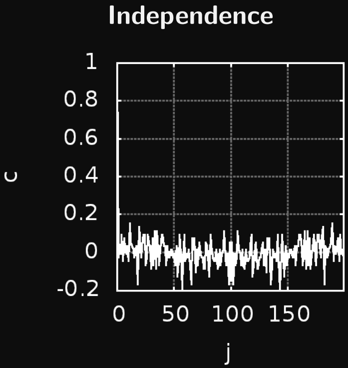
C++ way

```
// goodcoin.h
#include <random>
class GoodCoin {
 public:
   GoodCoin(): uniform(0,1) {}
   // method to set the starting seed
   void start(int seed) {
     engine.seed(seed);
   }
   // method to toss the coin (1:head, 0:tail)
   int toss() {
     return uniform(engine);//state in engine
   }
 private:
  std::uniform_int_distribution<int> uniform;
  std::mt19937 engine; // PNRG state
};
```

Test C++ way, N=200



Fair



Uncorrelated

About the random standard libary

The `<random>` library allows to produce random numbers using combinations of generators and distributions.

Generators	Objects that generate uniformly distributed numbers.
Distributions	Objects that transform sequences of numbers generated by a generator into sequences of numbers that follow a specific random variable distribution, such as uniform, Normal or Binomial.

Distribution objects generate random numbers by means of their operator() member, which takes a generator object as argument:

```
std::mt19937_64 generator;
std::uniform_int_distribution<int> distribution(1,6);
int die_roll = distribution(generator);  // generates number in the range 1..6
```

Available generators

While there are ways to create your own, the library has a number of standard available generators:

default_random_engine	Default random engine
minstd_rand	Minimal Standard minstd_rand generator
minstd_rand0	Minimal Standard minstd_rand0 generator
mt19937	Mersenne Twister 19937 generator
mt19937_64	Mersenne Twister 19937 generator (64 bit)
ranlux24_base	Ranlux 24 base generator
ranlux48_base	Ranlux 48 base generator
ranlux24	Ranlux 24 generator
ranlux48	Ranlux 48 generator
knuth_b	Knuth-B

Some good PRNGs

The following have long periods, independent samples, a fair distribution, and pass most statistical tests:

- Mersenne twister: mt19937 and mt19937_64, in the C++ random library.
  Use this one if you need many billions of random numbers relatively fast.
- In the lagged-Fibonacci class: ranlux24 and ranlux48, in the C++ random library.
  Use this if speed is not an impediment and you need more statistical tests passed.
- r1279 (lagged-Fibonacci generator), in the GSL and boost::random.
- WELL generator

https://www.arxiv-vanity.com/papers/1005.4117

Tip

Employ two random number generators, and see if they give, statistically speaking, the same result. If they don't, one of them is bad for your application.

Monte Carlo

Monte Carlo Techniques

A collection of techniques whose unifying feature is the use of randomness. These applications of randomness generally fall into one of three categories:

- Adding randomness to otherwise-deterministic dynamics, and studying how the dynamics are changed.
- Generating samples from a given probability distribution, $P(x)$, usually a distribution that is complicated and can't be dealt with nicely in closed form (e.g. Markov Chain Monte Carlo).
- Estimating expectation values under this distribution, e.g.

$$\langle A(\mathbf{x})\rangle = \int P(\mathbf{x})A(\mathbf{x})d\mathbf{x}$$

where $\mathbf{x}$ is typically high dimensional.

These depend on having a good random number generator!

Monte Carlo example: traffic flow

Nagel-Schreckenberg traffic is a 1D toy model used to generate traffic-like behaviour. At each time step in the model, the following rules are applied to each car in the simulation:

1. If the velocity is below vmax, then increase v by 1 (try to speed up).
2. If the car in front of the given car is a distance d away, and v≥d, then reduce v to d-1 (don't want to hit the car).
3. Add randomness: if v>0 then with probability p the car reduces its speed by 1.
4. The car moves ahead by v steps (on a circular track).

The four rules boil down to

$$\begin{aligned} v &\leftarrow \min(v+1, v_{max}) \\ v &\leftarrow \min(v, d-1) \\ v &\leftarrow v-1 \text{ if } v \neq 0 \text{ with probability } p \\ x &\leftarrow x+v \end{aligned}$$

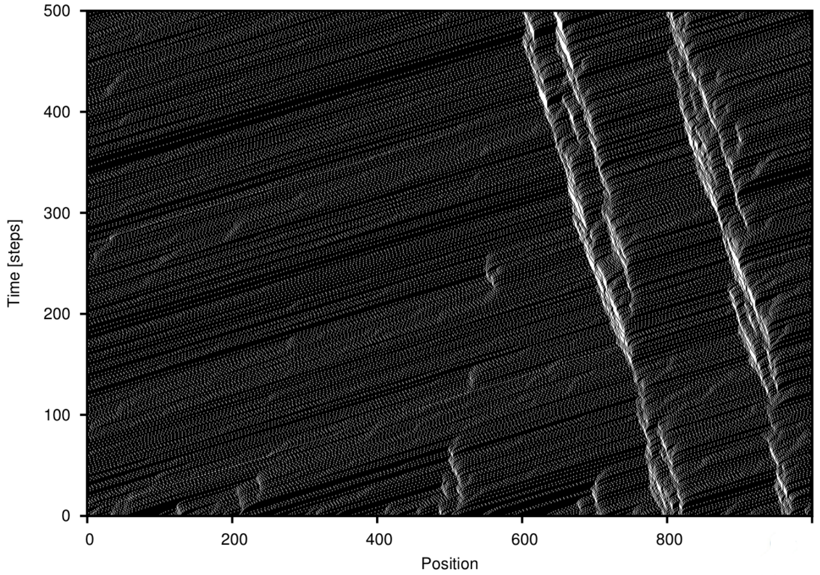
Monte Carlo example: traffic flow

numcars=200

gridsize=1000

p=0.13

vmax=5



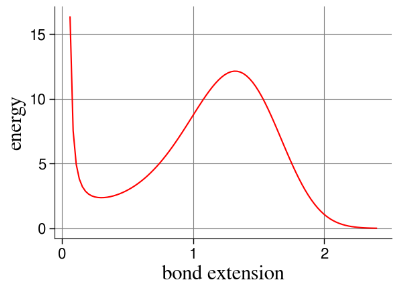
Note on Implementing Chance

$$v \leftarrow v - 1 \text{ if } v \neq 0 \text{ with probability } p$$

How do you do that?

- Draw a random number r using a PRNG with uniform distribution on $[0, 1)$.
- For any chosen value $p \in [0, 1)$, the chance that r is less than that value, is p itself.
- So if r is less than p, we will accept the move and decrease v if possible.
- If r is greater than or equal to p, we leave v as it is, i.e., we reject the move.

Monte Carlo Example: Chemical bond breakage



- In this model, the molecular bond between the atoms can break due to thermal fluctuations.
- Escape from the potential well corresponds to the bond breaking.

Model parameters

- the initial bond length
- the temperature (sets strength of thermal fluctuations)

Simulation parameters

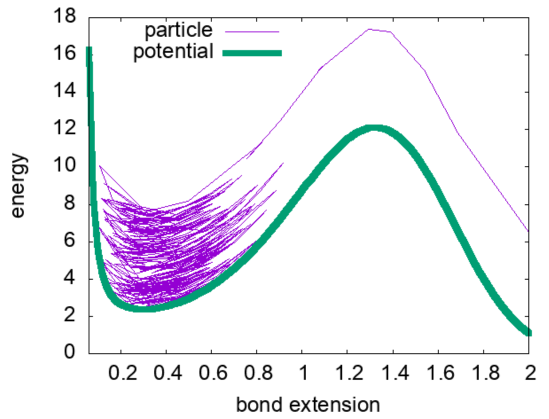
- the timestep
- maximum time to simulate
- random seed
- plus output parameters

Monte Carlo Example: Dynamics implementation

For each “timestep”:

1. Randomly perturb the bond extension, proportional to Δt.
2. Calculate the new energy of the system.
    - If the energy of the system goes down, keep the new position.
    - If the energy of the system goes up, keep the position if $r < \exp(-\Delta E/T)$, where r is a random number between 0 and 1, and T is the system temperature.
3. Repeat for all timesteps until escape.

Monte Carlo Example: results



initial bond extension: 0.6

temperature: 2.8

timestep: 0.0003

seed: 13

Breakage occured at $t = 5.33400000001951$.