

**ACCU
2023**

WHAT IS A RANDOM NUMBER AND WHY SHOULD I CARE?

FRANCES BUONTEMPO

A random number

```
int GetRandom()  
{  
    return 59;  
    //A perfectly randomly  
    // picked number  
}
```

<http://stackoverflow.com/questions/4195958/how-do-i-scale-down-numbers-from-rand>

What is random?

- A single number is not random, a sequence of numbers might be (Knuth)
- They still have properties
 - Mean (expectations)
 - Variance
- Chaotic != random
- Pseudo-random numbers



It is impossible to prove definitively whether a given sequence of numbers is random.

DILBERT By SCOTT ADAMS

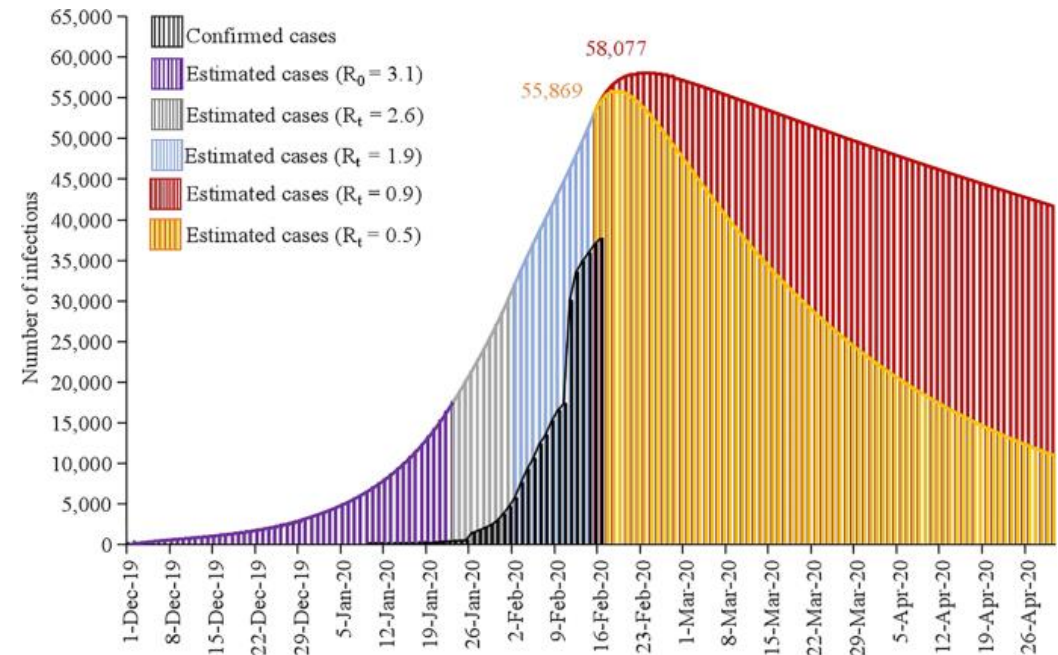


"Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin."

John von Neumann

Why should you care?

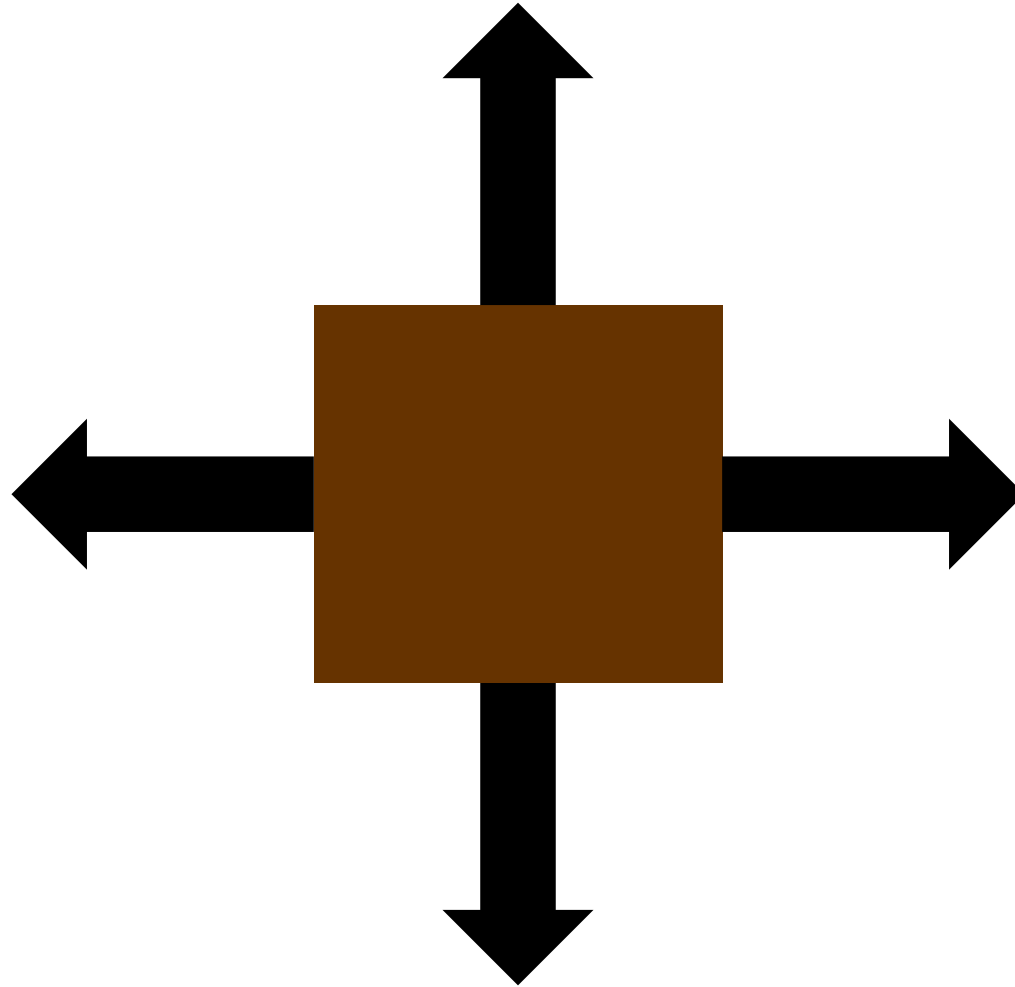
- People get it wrong
- We need randomness for
 - Games
 - Simulations, e.g. Covid-19 modeling
 - ...
- How to test
 - Code using random numbers
 - Random number generators



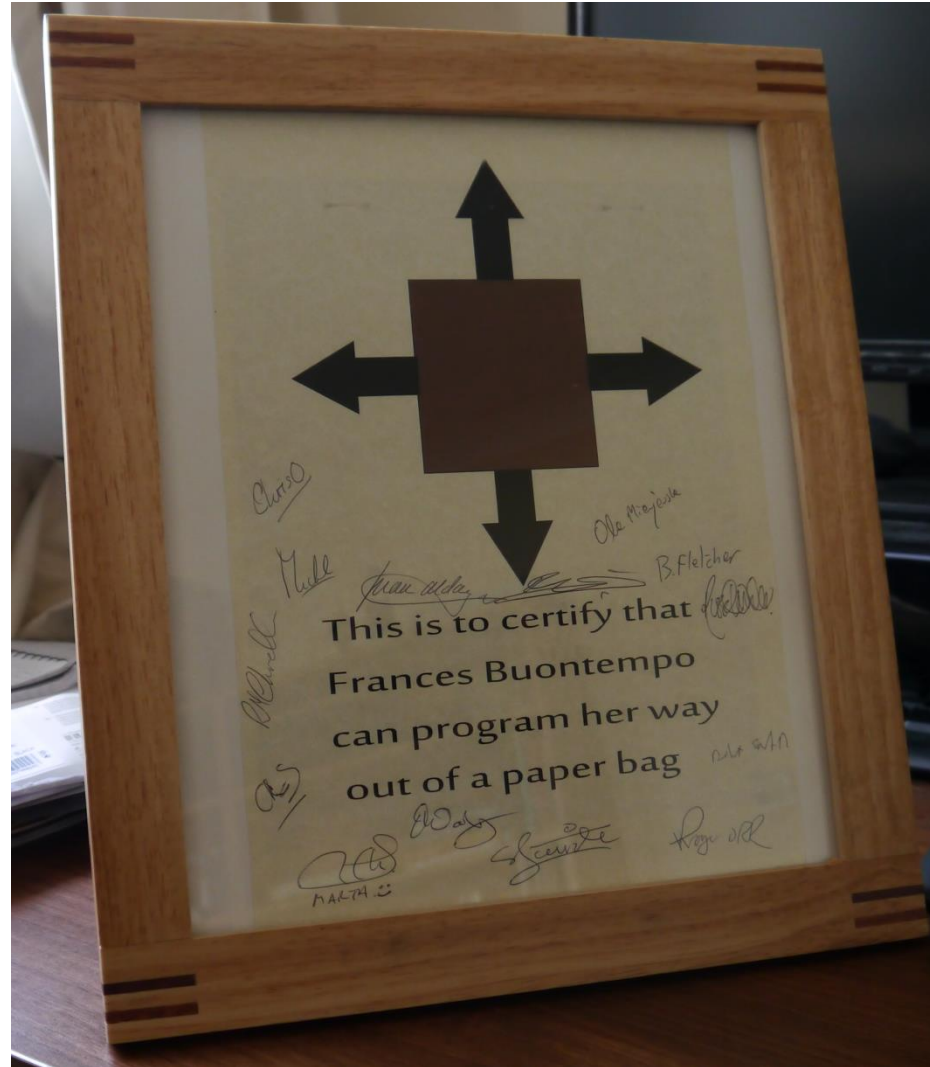
<https://www.nature.com/articles/s41421-020-0148-0>

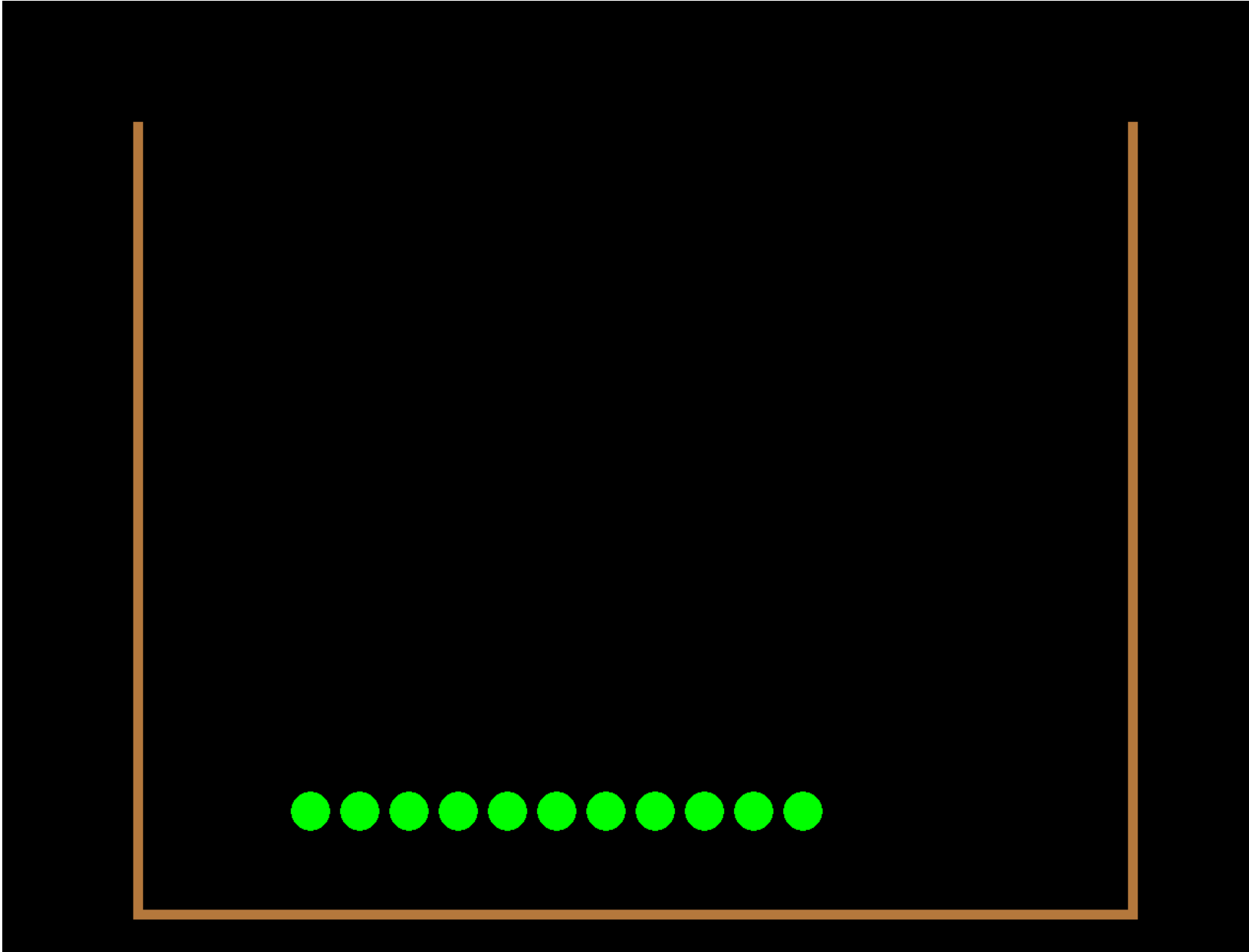
infectious disease dynamics SEIR (Susceptible, Exposed, Infectious, and Removed) model

Code your way out of a paper bag!



Award!





Did Fran do a demo?

- Race\Debug\SFMLRace.exe s

How do you generate random outcomes?

- C's `rand`
- Python's `random.random()`
- C#'s `Random.Next(1, 7)`
- C#'s `System.Security.Cryptography.RandomNumberGenerator`
- R's `runif(1)`, `rnorm(4)`

Always ask two questions

1. Default seed?

2. Range (), [] or []?

Under the hood



From <https://www.thestar.com/opinion/star-columnists/2021/10/25/this-is-nuts-after-squirrels-hid-hundreds-of-walnuts-under-the-hood-of-my-car-i-went-in-search-for-answers.html>

Generating random numbers

- Linear congruential generator (1958 by W. E. Thomson and A. Rotenber)
- $x_{i+1} = (Ax_i + c) \bmod M$
- If c is zero, it's a multiplicative linear congruential generator (MLCG) or multiplicative congruential generator (MCG).
 - Or a Lehmer RNG (1951)
 - $x_{i+1} = Ax_i \bmod M$
 - With e.g. $M = 2^{31} - 1 = 2,147,483,647$ (a Mersenne prime) and A is $7^5 = 16,807$

Cycle or Collapse

- $x_{i+1} = A x_i \bmod M$
- Up to $M-1$ if M is prime and we choose a suitable A
 - $A=7$ and $M=11$: **1**, 7, 5, 2, 3, 10, 4, 6, 9, 8, **1**, ...
 - $A=5$ and $M=11$: **1**, 5, 3, 4, 9, **1**, ...
- Why prime (or coprime)?
 - $A=4$ and $M=12$ starting with 1: 1, 4, 4, 4, ...
 - $4*1 \bmod 12 = 4$, then $4*4 \bmod 12 = 16 \bmod 12 = 4$, ...
 - $A=24$ and $M=12$ starting with 1: 1, 0, 0, 0, ...
 - $24*1 \bmod 12 = 0$, ...

Mersenne Twister

- $x_{k+n} = x_{k+m} \oplus \left((x_k^u | x_{k+1}^l) A \right) \quad k = 0, 1, \dots$
 - | concatenation of bit vectors
 - \oplus bitwise exclusive or
 - A is the “twist transformation”
 - $x A = \begin{cases} x \gg 1, \text{lowest bit of } x, x_0 = 0 \\ (x \gg 1) \oplus a, x_0 = 1 \end{cases}$
- Several seeds, to make bits for $x = (x_{w-1}, x_{w-2}, \dots, x_0,)$
- Matsumoto and Nishimura ACM Transactions on Modeling and Computer Simulation Vol 8 Issue 1 Jan. 1998 pp 3–30
 - <https://dl.acm.org/doi/10.1145/272991.272995>

Other engines are available

- Subtract with carry
 - AKA lagged Fibonacci: new term is “some combination” of any 2 previous terms
- Engine adaptors: generate pseudo-random numbers using another random number engine as entropy source.
 - `discard_block_engine`
 - discards some output of a random number engine
 - `independent_bits_engine`
 - packs the output of a random number engine into blocks of a specified number of bits
 - `shuffle_order_engine`
 - delivers the output of a random number engine in a different order

Is random a W.I.P?

- `minstd_rand0`
 - `std::linear_congruential_engine<std::uint_fast32_t, 16807, 0, 2147483647>`
 - Discovered in 1969 by Lewis, Goodman and Miller, adopted as "Minimal standard" in 1988 by Park and Miller
- `minstd::rand`
 - `std::linear_congruential_engine<std::uint_fast32_t, 48271, 0, 2147483647>`
 - Newer "Minimum standard", recommended by Park, Miller, and Stockmeyer in 1993
- Proposal P1932
 - The XorShift and Philox class generators are good candidates.

<https://wg21.link/P1932>
- Python's `numpy default_rng`
 - Now using PCG64 = permuted congruential generator, from 2014

<https://numpy.org/doc/stable/reference/random/index.html#random-quick-start>

P1932

Each of the C++11 random number generators has own advantages and disadvantages in terms of described criteria,

e.g. linear congruential generators, the simplest generators with 32-bit state, have a quite short generation period (2^{32}) and weak statistical properties

while Mersenne Twister 19937 generator has long generation period and strong statistical properties relying a big vector state underneath; in its turn, this state impacts on the effective support of parallel Monte Carlo simulations.

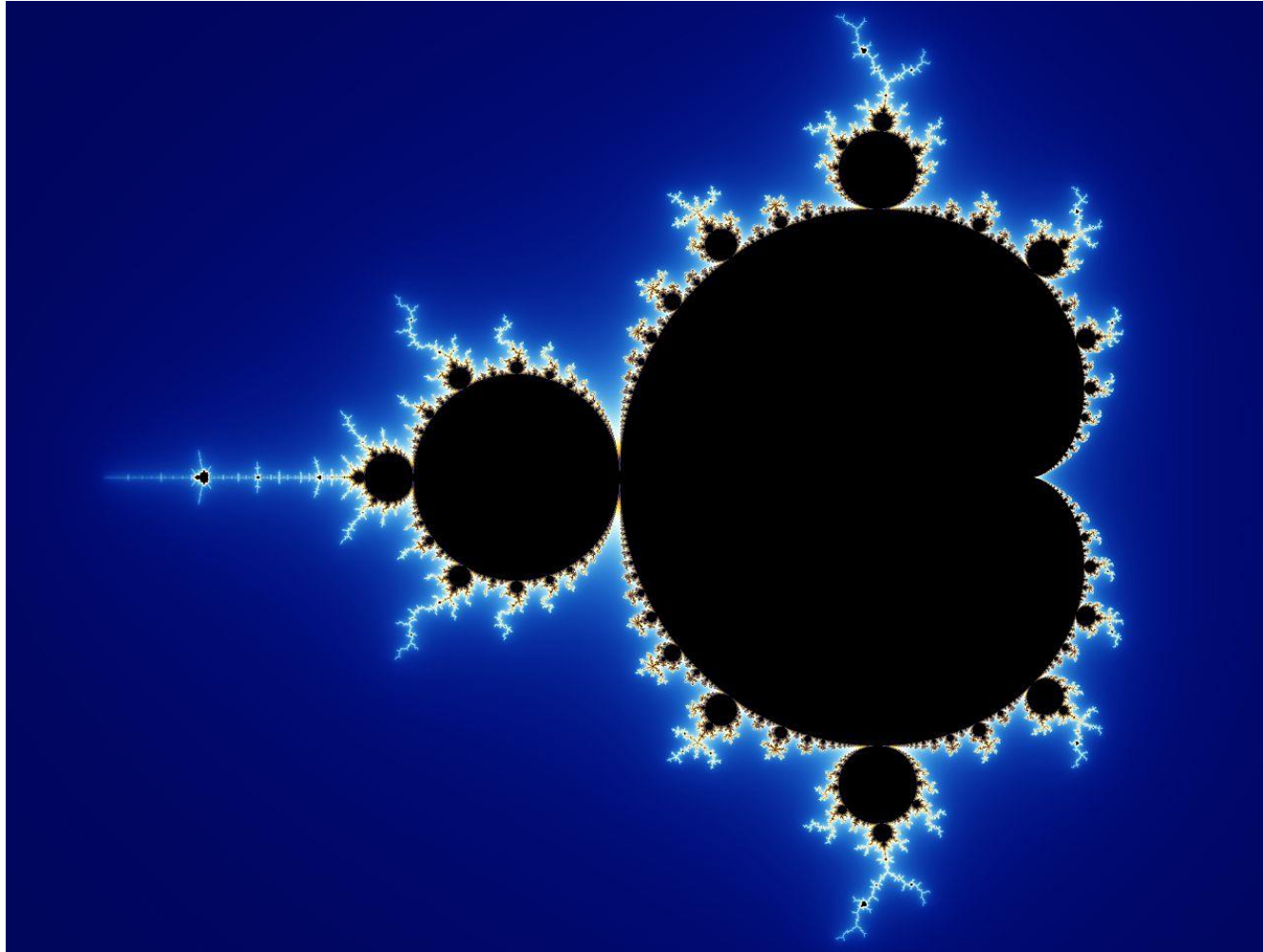
C++ random number generators do not support additional use cases such as quasi Monte Carlo simulations.

(An aside) $x^2 - 1$

1	0	0.25	0.5
0	-1	-0.9375	-0.75
-1	0	-0.12109	-0.4375
0	-1	-0.98534	-0.80859
-1	0	-0.02911	-0.34618
0	-1	-0.99915	-0.88016
-1	0	-0.00169	-0.22531
0	-1	-1	-0.94923
-1	0	-5.7E-06	-0.09896
0	-1	-1	-0.99021
-1	0	-6.6E-11	-0.01949
0	-1	-1	-0.99962
-1	0	0	-0.00076
0	-1	-1	-1
-1	0	0	-1.2E-06
0	-1	-1	-1
-1	0	0	-2.7E-12
0	-1	-1	-1
-1	0	0	0

$$z_{n+1} = z_n^2 - c$$

- Start with $z=0$, and pick a complex number c .
- Cycle, collapse (black), or continue growing (colour)...



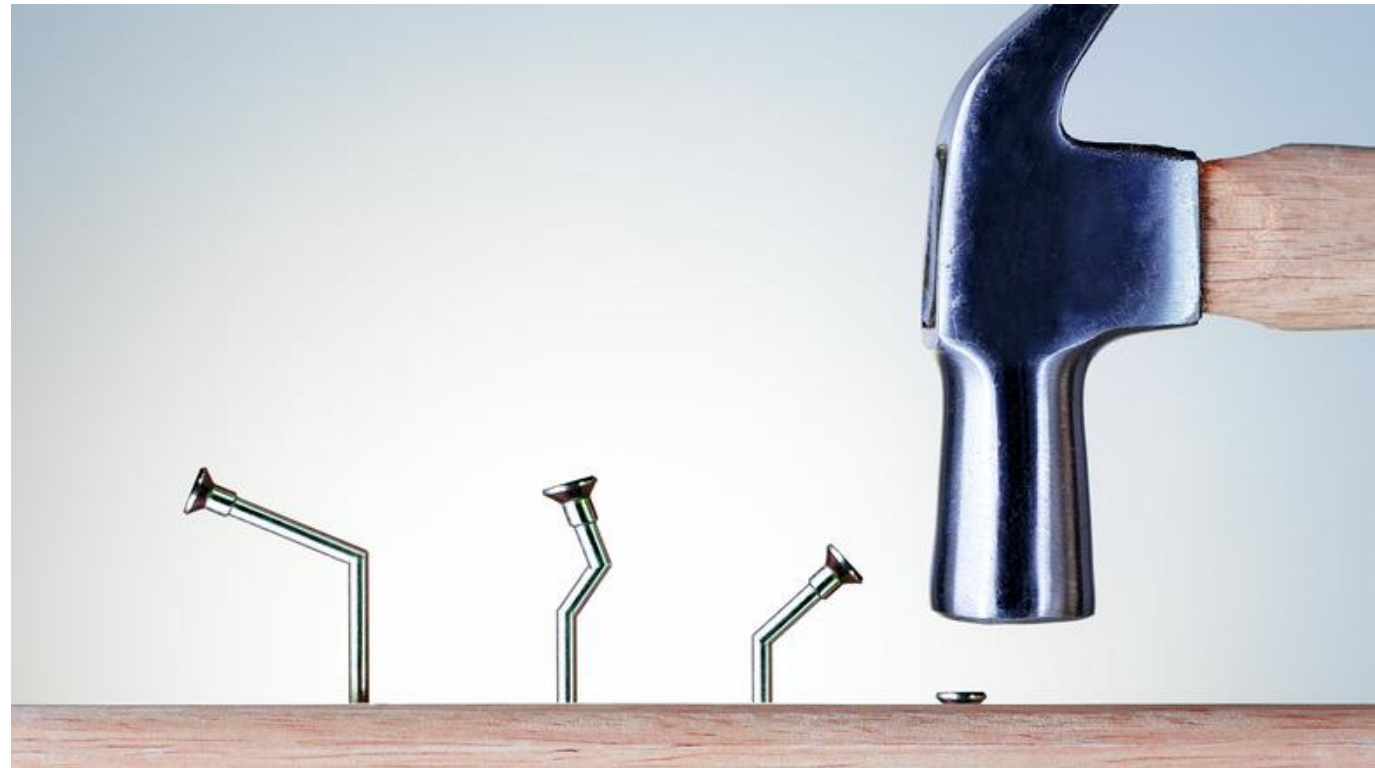
Created by Wolfgang Beyer with the program Ultra Fractal 3. - Own work, CC BY-SA 3.0,
<https://commons.wikimedia.org/w/index.php?curid=321973>

Or $2x^2 - 1$?

1	0	0.5	0.25
1	-1	-0.5	-0.875
1	1	-0.5	0.53125
1	1	-0.5	-0.43555
1	1	-0.5	-0.6206
1	1	-0.5	-0.22972
1	1	-0.5	-0.89446
1	1	-0.5	0.600119
1	1	-0.5	-0.27971
1	1	-0.5	-0.84352
1	1	-0.5	0.423054
1	1	-0.5	-0.64205
1	1	-0.5	-0.17554
1	1	-0.5	-0.93837
1	1	-0.5	0.761082
1	1	-0.5	0.158491
1	1	-0.5	-0.94976
1	1	-0.5	0.804093
1	1	-0.5	0.293132

Using random number generators

- Time to think
- Possible problems
- Common use cases
- Testing....



Seeds

- The number the recurrence relationship starts with x_i
- What should you use?
 - Current time?
 - Known number?
- **Same seed, different languages... (or same lang different os, etc)**

Threads

- Might not be thread safe... global state
 - “It is implementation-defined whether rand() is thread-safe.”
 - <https://en.cppreference.com/w/cpp/numeric/random/rand>
- Two threads... different results depending on order
 - Not just in C
- Don't use C's rand
 - <https://learn.microsoft.com/en-us/events/goingnative-2013/rand-considered-harmful>

Roll a die in Python

```
import random  
random.randint(1, 6) #alias for randrange(1, 6+1)
```

Always ask two questions

1. Default seed?
2. Range (), [] or []?

Never use %

- In C....

```
srand(time(NULL));
```

- <https://c-faq.com/lib/randrange.html>

```
rand() % N          /* POOR */
```

rand returns $RAND_MAX+1$ distinct values, which cannot always be evenly divided up into N buckets

```
M + rand() / (RAND_MAX / (N - M + 1) + 1)
```

gives numbers in the range $[M, N]$

Roll a die in C++

```
#include <iostream>
#include <random>
int main()
{
    std::default_random_engine generator; // probably a mt19937
    std::uniform_int_distribution<int> distribution(1, 6);
    const int count = 3;
    for (int i = 0; i < count; ++i)
    {
        std::cout << distribution(generator) << '\n';
    }
}
```

```
using default_random_engine = mt19937;
```

```
using mt19937 = mersenne_twister_engine<unsigned int, 32, 624, 397, 31,  
0x9908b0df, 11, 0xffffffff, 7, 0x9d2c5680, 15, 0xefc60000, 18, 1812433253>;
```

```
class mersenne_twister_engine : public mersenne_twister<_Ty, _Wx, _Nx,  
_Mx, _Rx, _Px, _Ux, _Sx, _Bx, _Tx, _Cx, _Lx>;
```

```
class mersenne_twister : public _Circ_buf<_Ty, _Nx> {  
    static constexpr _Ty default_seed = 5489U;  
    mersenne_twister() : _Dxval(_WMSK) {  
        seed(default_seed, static_cast<_Ty>(1812433253));  
    }  
}
```

Two questions

```
#include <iostream>
#include <random>
int main()
{
    std::random_device rd;
    std::default_random_engine generator{rd()};
    std::uniform_int_distribution distribution(1, 6); // don't need <int>
    const int count = 3;
    for (int i = 0; i < count; ++i)
    {
        std::cout << distribution(generator) << '\n';
    }
}
```

Could use time... but

```
#include <chrono>

std::default_random_engine generator{
    static_cast<unsigned int>(
        std::chrono::steady_clock::now().
            time_since_epoch().count())
};
```


A ~~warning~~ discussion about seeds

The current standard library does not provide any convenient way to use a `std::random_device` to properly (in the sense that each initial state is equally likely) seed a random engine.

The naïve approach that most people seem to use is the following.

```
template <typename EngineT>    //
requires (RandomNumberEngine (EngineT))
void seed_non_deterministically_1st (EngineT& engine)
{
    std::random_device rnddev {};
    engine.seed (rnddev ());
}
```

This code is severely flawed. If `EngineT` is `std::mt19937`, it has a state size of 19968 bits. However, if an unsigned int is 32 bits (as is the common case on many platforms today), then of the up to 2^{19968} states, at most 2^{32} (that is one 2^{-19936} -th) can possibly be chosen!

<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0205r0.html>

More weirdness

- The Mersenne Twister uses 624 32-bit integers to represent its internal state, plus a few more for housekeeping, so using one 32 bit value (e.g. from random device) seems odd.
- “Strangely enough, when you initialize the Mersenne Twister with a 32-bit seed (via `seed_seq`), **it can't ever generate 7, or 13 as its first output.** And **two different seeds produce 0.** Even more crazy, there are **twelve different 32-bit seeds that can produce the "random" numbers 1226181350 and 1563636090,** so those numbers show up twelve times more often than we'd expect.”

<https://www.pcg-random.org/posts/cpp-seeding-surprises.html>

Oh really?

```
std::seed_seq seeder{ 1080100664 };  
std::mt19937 rng(seeder);  
std::cout << rng() << '\n';
```

versus

```
std::mt19937 rng(1080100664);  
std::cout << rng() << '\n';
```

Warm up?

```
auto RandomlySeededMersenneTwister () {  
    std::mt19937 rng(std::random_device{} ());  
    rng.discard(700000);  
    return rng;  
}
```

<https://codereview.stackexchange.com/questions/109260/seed-stdmt19937-from-stdrandom-device>

700000 from "Improved long-period generators based on linear recurrences modulo 2", F. Panneton, P. L'Ecuyer, M. Matsumoto in ACM TOMS Volume 32 Issue 1, March 2006 Pages 1-16

But

“seed_seq initialization used by std::mt19937 performs a warm up”

<https://www.learncpp.com/cpp-tutorial/generating-random-numbers-using-mersenne-twister/>

```
mersenne_twister_engine() : _Mybase(default_seed, _Dx, _Fx) {}
```

```
explicit mersenne_twister_engine(result_type _Xx0) :  
    _Mybase(_Xx0, _Dx, _Fx) {}
```

```
template <class Seed_seq, Enable_if_seed_seq_t<Seed_seq,  
mersenne_twister_engine> = 0>
```

```
explicit mersenne_twister_engine(Seed_seq& _Seq) :  
    _Mybase(default_seed, _Dx, _Fx) {  
        seed(_Seq);  
    }
```

My head hurts!

- First, C++ standard randoms are not portable ("seed-stable")

- 2nd this is good enough:

```
std::random_device rd;  
std::mt19937 rng{ rd() };
```

- 3rd this might be slightly better:

```
std::random_device rd;  
std::seed_seq seeder{ rd() };  
std::mt19937 rng(seeder);
```

Roll two dice

Can they share the same generator?

```
rng1 = random.Random()  
rng2 = random.Random()  
rng1.randint(1, 6)  
rng2.randint(1, 6)
```

Roll two dice in C++

```
std::random_device rd;
std::default_random_engine generator1{ rd() };
std::default_random_engine generator2{ rd() };
    //Using time instead of rd would be bad!
std::uniform_int_distribution distribution(1, 6);
const int count = 3;
for (int i = 0; i < count; ++i)
{
    std::cout << distribution(generator1) << '\n';
    std::cout << distribution(generator2) << '\n';
}
```


Pick a card

```
import random  
random.randint(1, 52)
```

Always ask two questions

1. Default seed?
2. Range (), [] or []?

Shuffle the cards first!

```
import random
L = [1, 2, 3, 4, 5]
random.shuffle(L)
# [5, 3, 2, 4, 1]
```

Note that even for small $\text{len}(x)$, the total number of permutations of x can quickly grow larger than the period of most random number generators. This implies that most permutations of a long sequence can never be generated. For example, a sequence of length 2080 is the largest that can fit within the period of the Mersenne Twister random number generator.

How to shuffle 3 cards

```
for (int i = 0; i < cards.Length; i++)  
{  
    int n = rand.Next(cards.Length);  
    Swap(ref cards[i], ref cards[n]);  
}
```

<https://blog.codinghorror.com/the-danger-of-naivete/>

- 1 of 3 outcomes after first loop
- 1 of those goes to another of 3
- Finally last goes to another of 3 $\Rightarrow 3^3 = 27$ possible outcomes
- Should just be 6: 123, 132, 213, 231, 312, 321

A better way...

Knuth-Fisher-Yates shuffle algorithm

```
for (int i = cards.Length - 1; i > 0; i--)  
{  
    int n = rand.Next(i + 1);  
    Swap(ref cards[i], ref cards[n]);  
}
```

C++ shuffle kerfuffles

- Lives in the algorithm header
- ~~std::random_shuffle~~
 - Might use `swap(first[i], first[std::rand() % (i+1)])`;
- `std::shuffle` (a deck of cards, maybe)

```
std::random_device rd;  
std::mt19937 gen{ rd() };  
std::shuffle(deck.begin(), deck.end(), gen);  
//or  
std::ranges::shuffle(deck, gen);
```

Choices

- Choices v sample in python

```
things = ['red', 'red', 'red', 'red', 'blue', 'blue']
```

```
random.sample(things, k=3)
```

```
random.choices(things, k=3)
```

- **With or without replacement?**

- **Sample:** Return a k length list of unique elements chosen from the population without replacement.
- **Choice:** Return a k length list of elements chosen from the population with replacement.

Samples in C++

```
#include <algorithm>
#include <string>
#include <vector>
void sampling()
{
    using namespace std::string_literals;
    std::vector things = {"red"s, "red"s, "red"s, "red"s, "blue"s, "blue"s };
    std::random_device rd;
    std::default_random_engine gen{ rd() };
    std::ranges::shuffle(things, gen);

    std::vector result(things.begin(), things.begin() + 3); // Get the first 3 values
    for (auto thing: result)
        std::cout << thing << '\n';
}
```

But...

- If we want to sample 3 integers out of 1 billion without needs a vector with a billion values (memory-inefficient) and then we'd have to shuffle **all of them** (runtime-inefficient)

<https://www.gormanalysis.com/blog/random-numbers-in-cpp/>

- Suggests an `unordered_set` and select by index
- But things have repeats
 - `{"red"s, "red"s, "red"s, "red"s, "blue"s, "blue"s };`

Actually

```
using namespace std::string_literals;
auto things = { "red"s, "red"s, "red"s, "red"s, "blue"s, "blue"s };
std::vector<std::string> result;
std::sample(things.begin(), things.end(),
            std::back_inserter(result), 3,
            std::mt19937{ std::random_device{}() }); // C++17
// or
std::ranges::sample(things,
                    std::back_inserter(result), 3,
                    std::mt19937{ std::random_device{}() }); // C++20

for (auto thing : result)
    std::cout << thing << '\n';
```

R, S

- Knuth's algorithms
 - R reservoir sampling, sample without replacement, of k items from a population of unknown size n in a single pass over the items.
 - S randomly sampling n items from a set of M items, with equal probability, where $M \geq n$ and M , the number of items is unknown until the end.

Notes

This function may implement selection sampling or reservoir sampling.

Feature-test macro	Value	Std	Comment
<code>__cpp_lib_sample</code>	201603L	(C++17)	<code>std::sample</code>

Choices in C++

```
void choices()
{
    using namespace std::string_literals;
    std::vector things = { "red"s, "red"s, "red"s, "red"s, "blue"s, "blue"s };
    std::vector<double> weights(things.size(), 1.0/things.size());

    std::discrete_distribution<int> distribution(weights.begin(), weights.end());
    std::random_device rd;
    std::default_random_engine gen{ rd() };
    std::vector<std::string> result(3);
    std::generate(result.begin(), result.end(), [&]{ return things[distribution(gen)]; });

    for (auto thing : result)
        std::cout << thing << '\n';
}
```

How to test

- A dice game?
- Picking a card?
- Shuffling a card?



YOU DO NOT NEED TO TEST YOUR LANGUAGE'S RANDOM NUMBER GENERATOR



@fbuontempo

**BUT DO CHECK YOU ARE
USING THE RANDOM
NUMBER GENERATOR
CORRECTLY**

Testing

- Known seed v mock
- Send in 0? (Makes much of maths/simulation study easy)

Seeds

- A seed will give you a known sequence (on a compiler, platform, etc)
- Will it cover all the transitions?
- Fake out the random
 - Try return zero
- Have changes (win/lose) in a separate function to random
- What are you trying to test?



It's not as simple as using a predictable sequence to test it, because even simple changes to the code may use up the sequence in a different order and break tons of tests (which isn't helpful, because all that stuff didn't break).

I know I don't want to go change 50 tests because I reordered two lines.

My suggestion is that somewhere, your code should be *doing* something with the numbers.

You can test whether or not it does the right things with the right numbers

<http://wiki.c2.com/?UnitTestingRandomness>

Testing, testing

- <https://stackoverflow.com/questions/61047296/how-to-replace-the-call-to-random-randint-in-a-function-tested-with-pytest>

```
import random
```

```
...
```

```
def hit(self, enemy, attack):
```

```
    dmg = random.randint(self.weapon.damage_min,  
                          self.weapon.damage_max) +  
          self.strength // 4
```

```
...
```

Suggested approach

```
def test_player_hit_missed(monkeypatch,  
    monster,  
    hero):  
    monkeypatch.setattr('random.randint',  
        lambda a, b: -3)  
    hero.hit(monster, 'Scream')  
    assert monster.life == 55
```

What do you think?

- Send in values? (and `hit` does the hit)

Testing, ... what?

- <https://stackoverflow.com/questions/42788644/how-to-test-random-choice-in-python>

```
from random import shuffle
```

```
def getMaxIndices(lst):  
    '''  
    Return indices of max value. If max value appears more than once,  
    we chose one of its indices randomly.  
    '''  
    index_lst = [(i, j) for i, j in enumerate(lst)]  
    shuffle(index_lst)  
    index_lst.sort(key=lambda x: x[1])  
    max_index = index_lst.pop()[0]  
    return max_index
```

Suggested approach

```
@patch('random.shuffle', lambda x: x)
def test_get_max_Indices():
    max_index = getMaxIndices([4, 5, 6, 7, 8])
    assert max_index == 4
```

Alternatives?

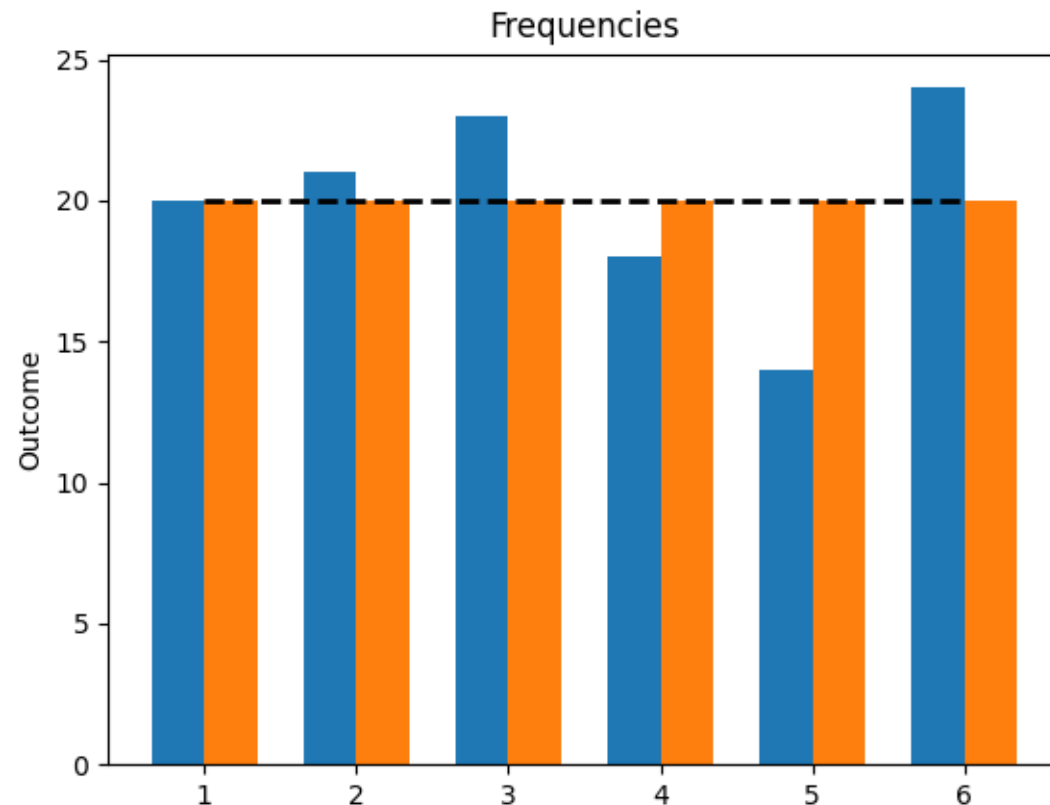
- Deterministic tests - verify code
 - Send in values? (and `hit` does the hit)
 - Never forget edge cases
- Random sequences have properties
 - What average do you expect?
 - Is your uniform distribution giving uniform results?
- Non-deterministic test - find bugs
 - Property based testing
 - Fuzzing
- Also, have you ever run your unit/integration/etc test in random order
 - Random is useful

Beyond uniform discrete

- Dice, cards,...
 - whole numbers (discrete)
 - Equally likely (uniform)
- Continuous (doubles etc)
 - <https://www.thusspakeak.com/ak/2014/06/01-WhatAreTheChancesOfThat.html>
- Non-uniform
 - Normal
 - Weighted
- Distributions of angles and directional statistics

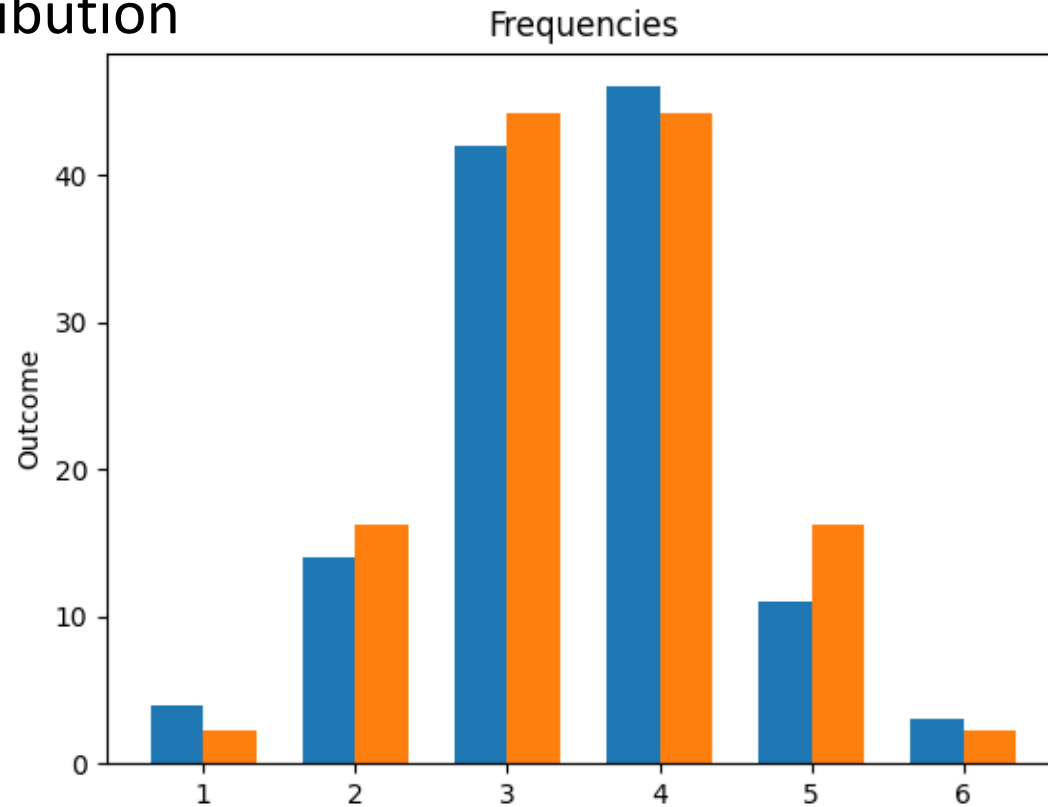
Uniform Distribution $x \sim U(a, b)$

- x is drawn from the uniform distribution with range $[a, b]$
- Roll a die 120 times,
 - how many of each number?
 - 1: 20,
 - 2: 21,
 - 3: 23,
 - 4: 18,
 - 5: 14,
 - 6: 24



Normal Distribution $X \sim N(3.5, 1)$

- The Bell curve or Gaussian distribution
- Bucket continuous numbers
- 1: 4,
- 2: 14,
- 3: 42,
- 4: 46,
- 5: 11,
- 6: 3

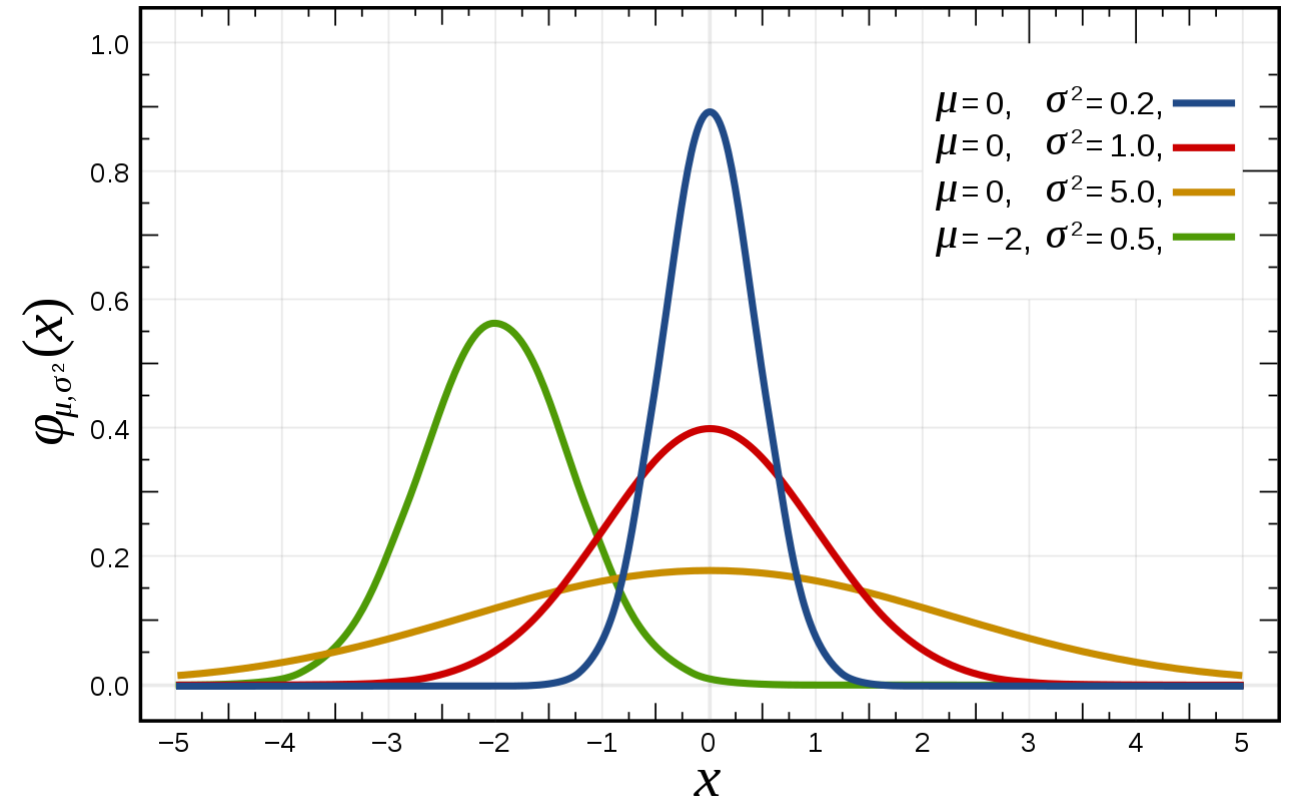


Gaussian pdf

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

μ mean

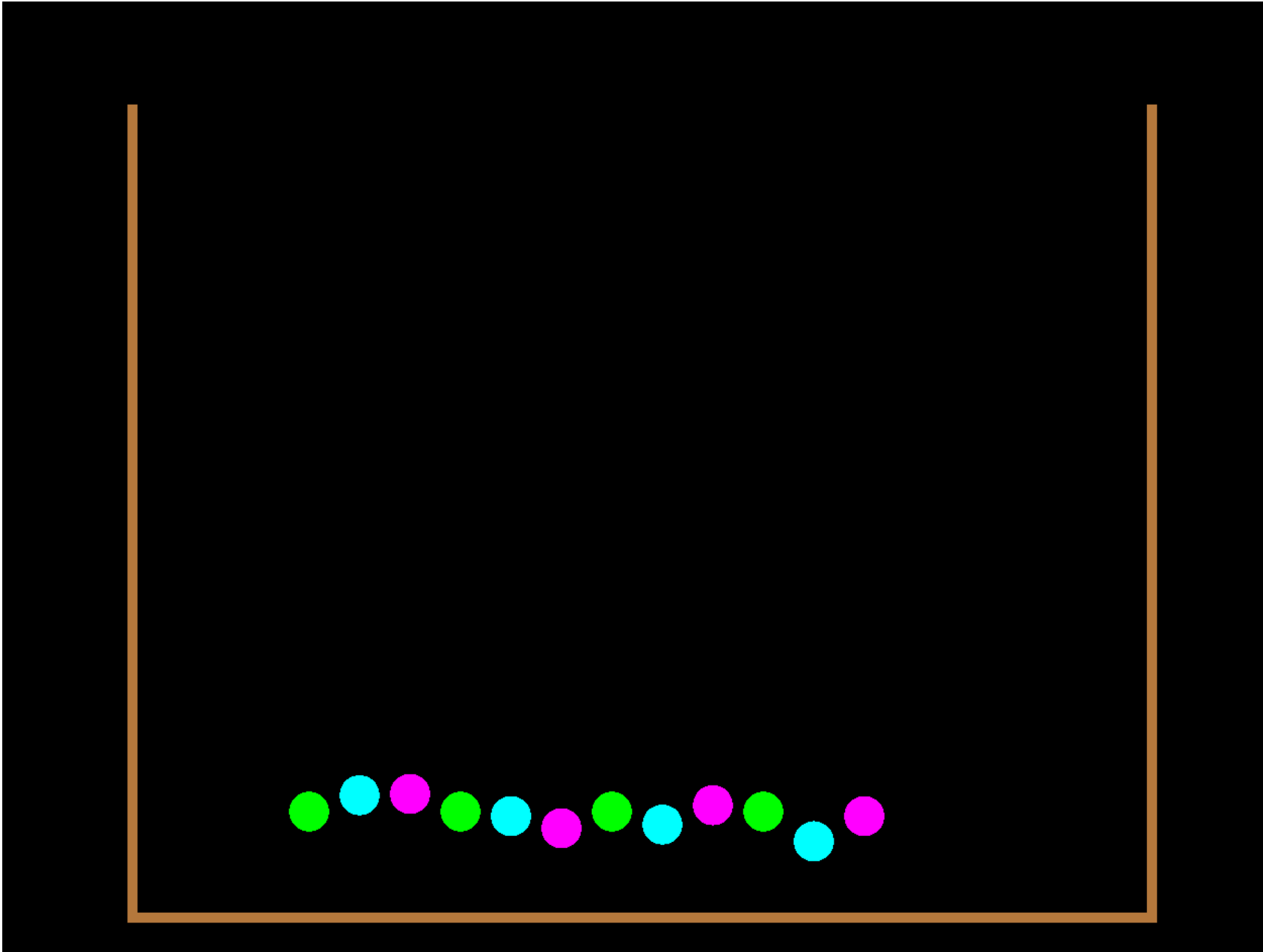
σ standard deviation



By Inductiveload - self-made, Mathematica, Inkscape, Public Domain,
<https://commons.wikimedia.org/w/index.php?curid=3817954>

Race!

- Two distributions and the steppers
 - Stepper, $y += 20.0f$; green blob
 - Uniform, on a float $[0, 40]$; cyan blob
 - Normal, $N \sim (20.0f, 10.0f)$; magenta blob
- Who will win?



@fbuontempo

Formal tests for randomness

- Does the sequence have a recognisable pattern?
- Can you get all the numbers in the range?...
- Diehard, dieharder



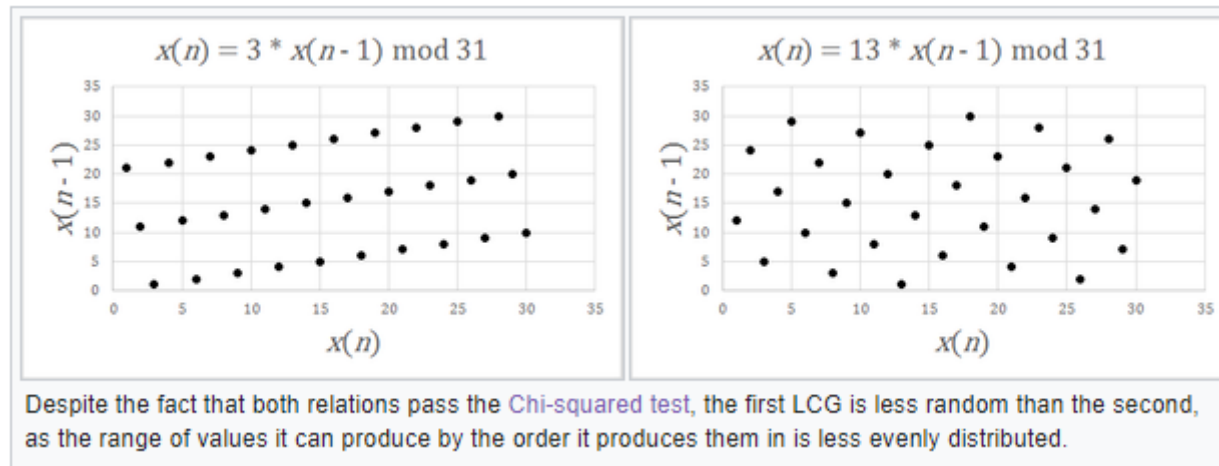
From IMDB

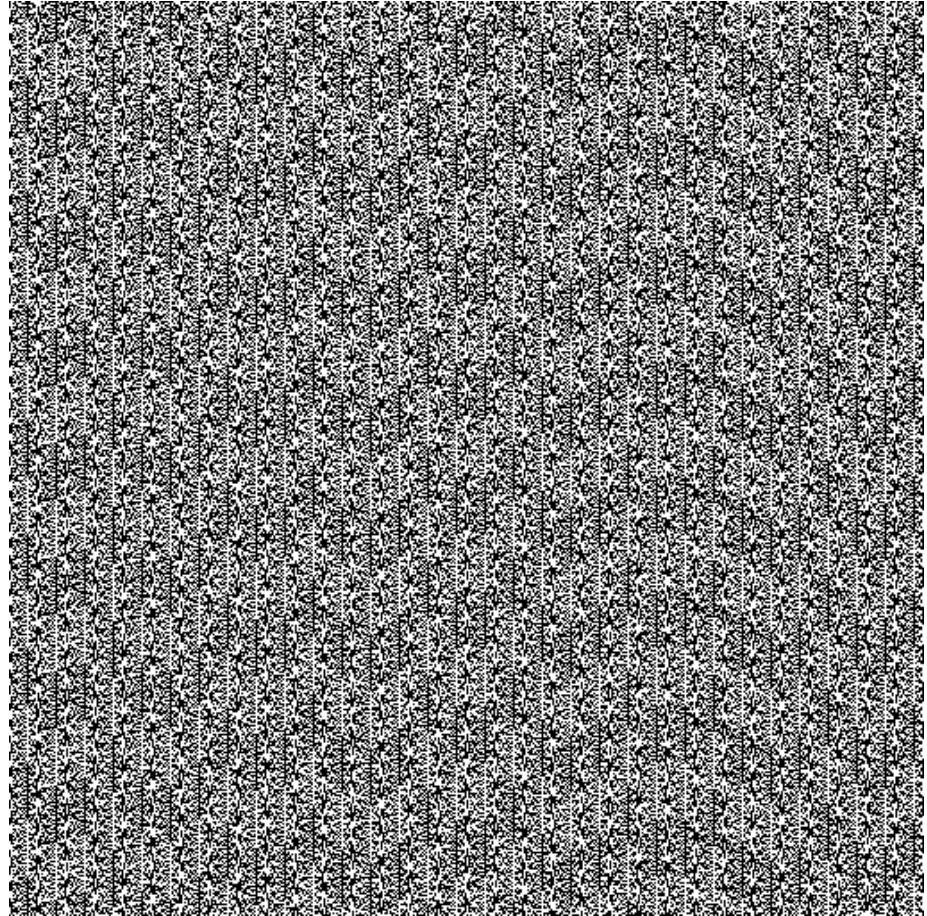
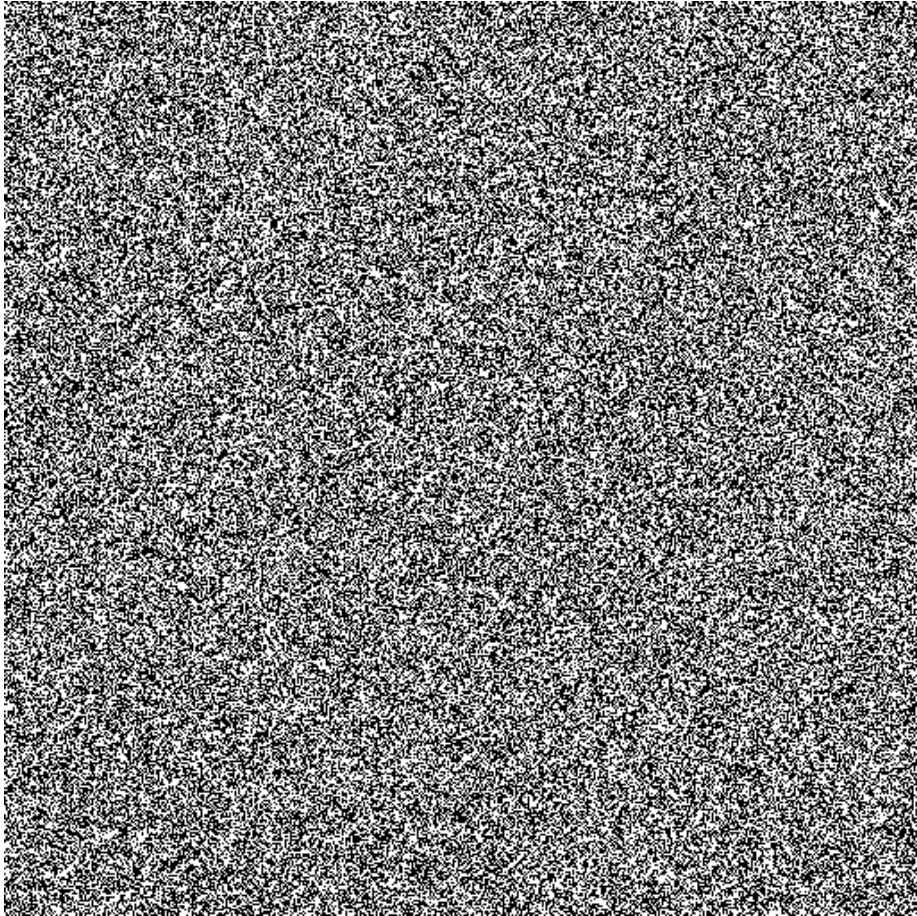
<https://www.imdb.com/title/tt0095016/>

Spectral characteristics

- Looking for repetitive patterns that are near each other
- Plotting $x(n)$ against $x(n-1)$

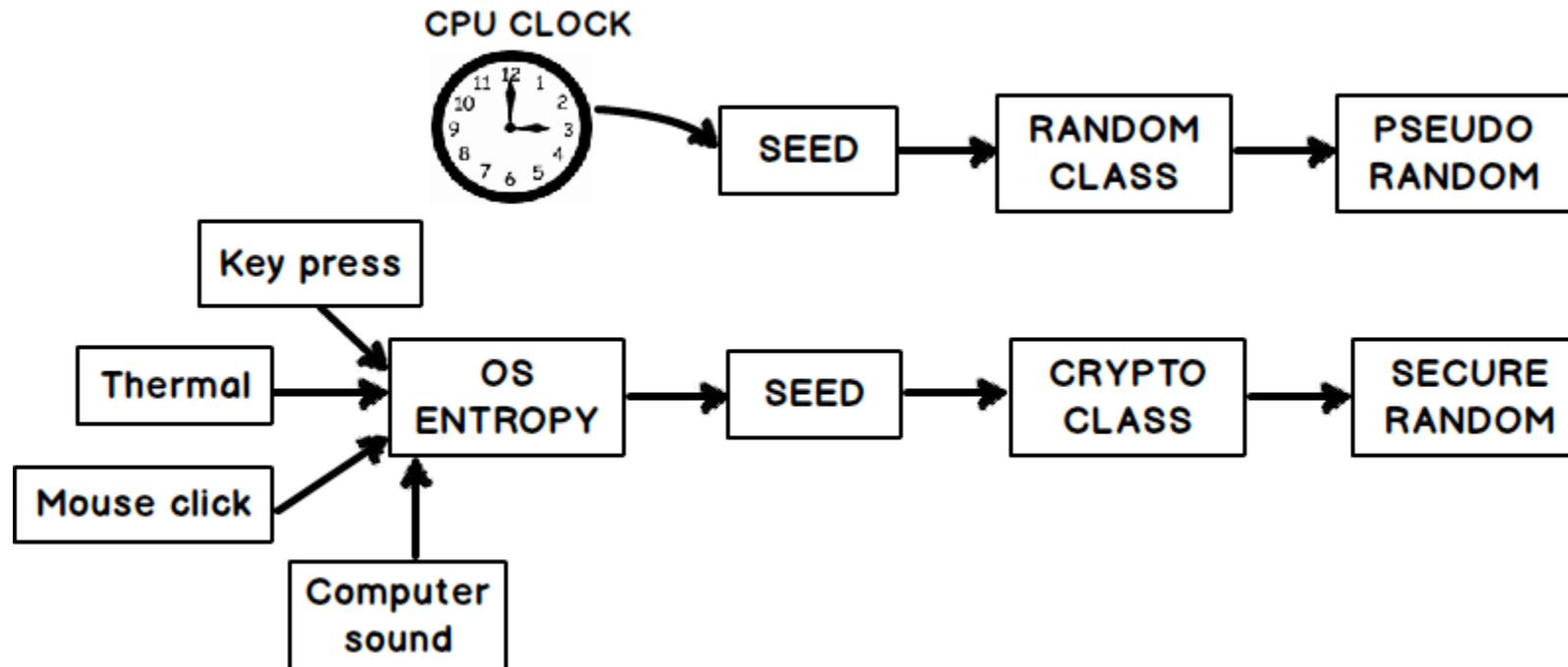
https://en.wikipedia.org/wiki/Spectral_test





<https://www.random.org/analysis/>

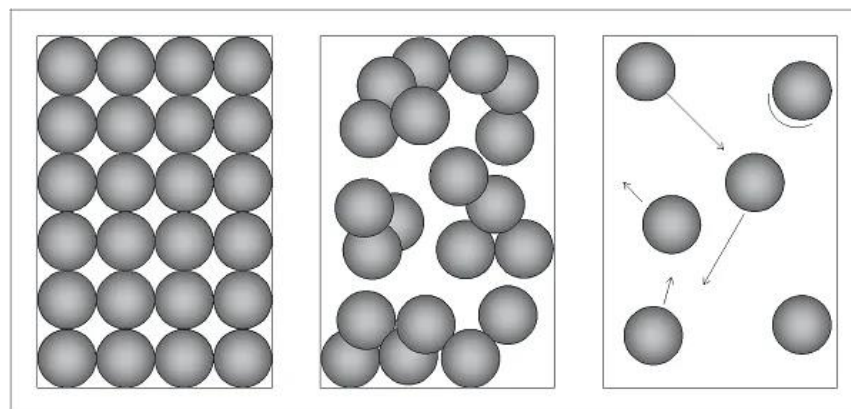
Pseudo v secure



From <https://stackoverflow.com/questions/2706500/how-do-i-generate-a-random-integer-in-c>

Entropy

- The number of possible microscopic arrangements or states of individual atoms and molecules of a system that comply with the macroscopic condition of the system. (Ludwig Boltzmann)
- The average level of "information", "surprise", or "uncertainty" inherent to the variable's possible outcomes. (Claude Shannon)



Information Entropy

$$H(X) = - \sum_{x \in X} p(x) \log p(x)$$

- Fair coin, two outcomes, $p(\text{head})=p(\text{tail})=0.5$
 - $H(X) = -(1/2 \log(1/2) + 1/2 \log(1/2)) = -(1/2 \times -1 + 1/2 \times -1) = 1$
- Unfair coin, two outcomes, $p(\text{head})=0, p(\text{tail})=1$
 - $H(X) = -(0 \times \log(0) + 1 \times \log(1)) = -(0 + 1/2 \times 0) = 0$

0: No surprise

1: Total surprise

Entropy

“I've seen winzip used as a tool to measure the randomness of a file of values before (obviously, the smaller it can compress the file the less random it is).”

<http://wiki.c2.com/?UnitTestingRandomness>

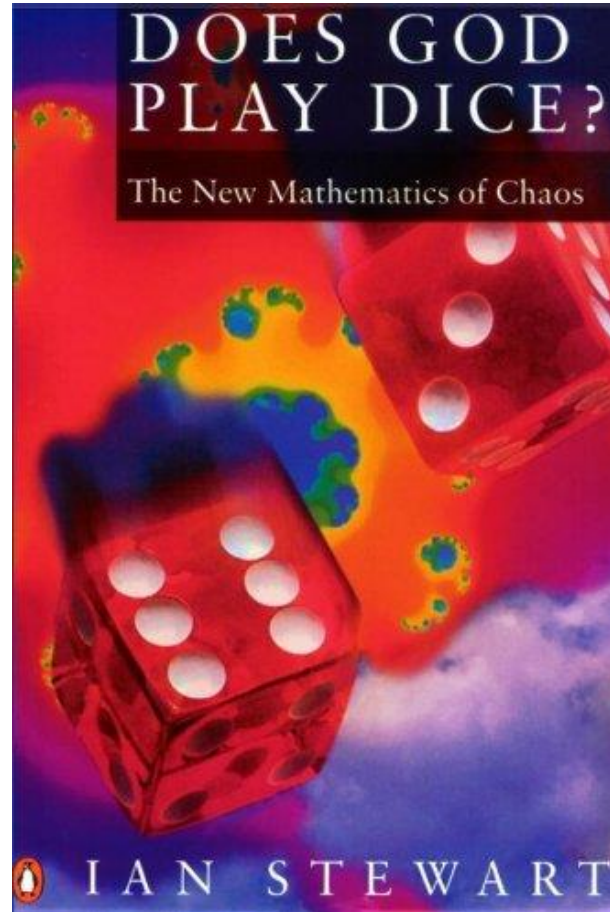
Cryptography

- NIST “provides guidelines and recommendations for generating random numbers for cryptographic use”
 - <https://csrc.nist.gov/projects/random-bit-generation>
- Also goes wrong
 - E.g. SSL, Key generation, RSA public key factoring
https://en.wikipedia.org/wiki/Random_number_generator_attack
- “the elliptic curve digital signature algorithm – ECDSA – demands that the random number used to sign a private key is only ever used once. If the random number generator is used twice, the private key is recoverable.”
https://www.theregister.com/2013/08/12/android_bug_batters_bitcoin_wallets/
 - The Java class SecureRandom (used by the vulnerable wallets) can generate collisions for the value r. (See <http://armoredbarista.blogspot.com/2013/03/randomly-failed-weaknesses-in-java.html>)

Integrated balanced homodyne detector

- 100-Gbit/s Integrated Quantum Random Number Generator Based on Vacuum Fluctuations
- “Quantum random number generation allows for the creation of truly unpredictable numbers due to the inherent randomness available in quantum mechanics.”

<https://journals.aps.org/prxquantum/abstract/10.1103/PRXQuantum.4.010330>



What have we learnt?

- There's no such thing as a random number
- C++ gives us choices
- Things change, so keep learning
- Seemingly arbitrary outcomes make for good games
 - And some useful stuff too
 - And fun
 - Time for one more demo?

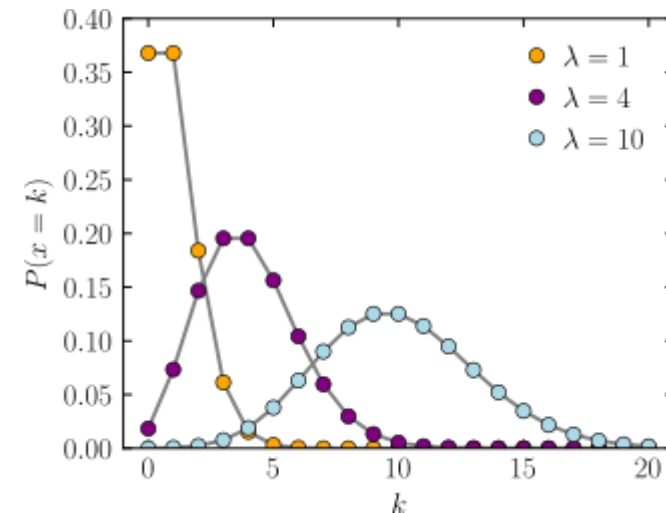
Jumpers

```
std::poisson_distribution<> distribution(0.03125); // 1 in 32
```

```
int jump = distribution(generator);
```

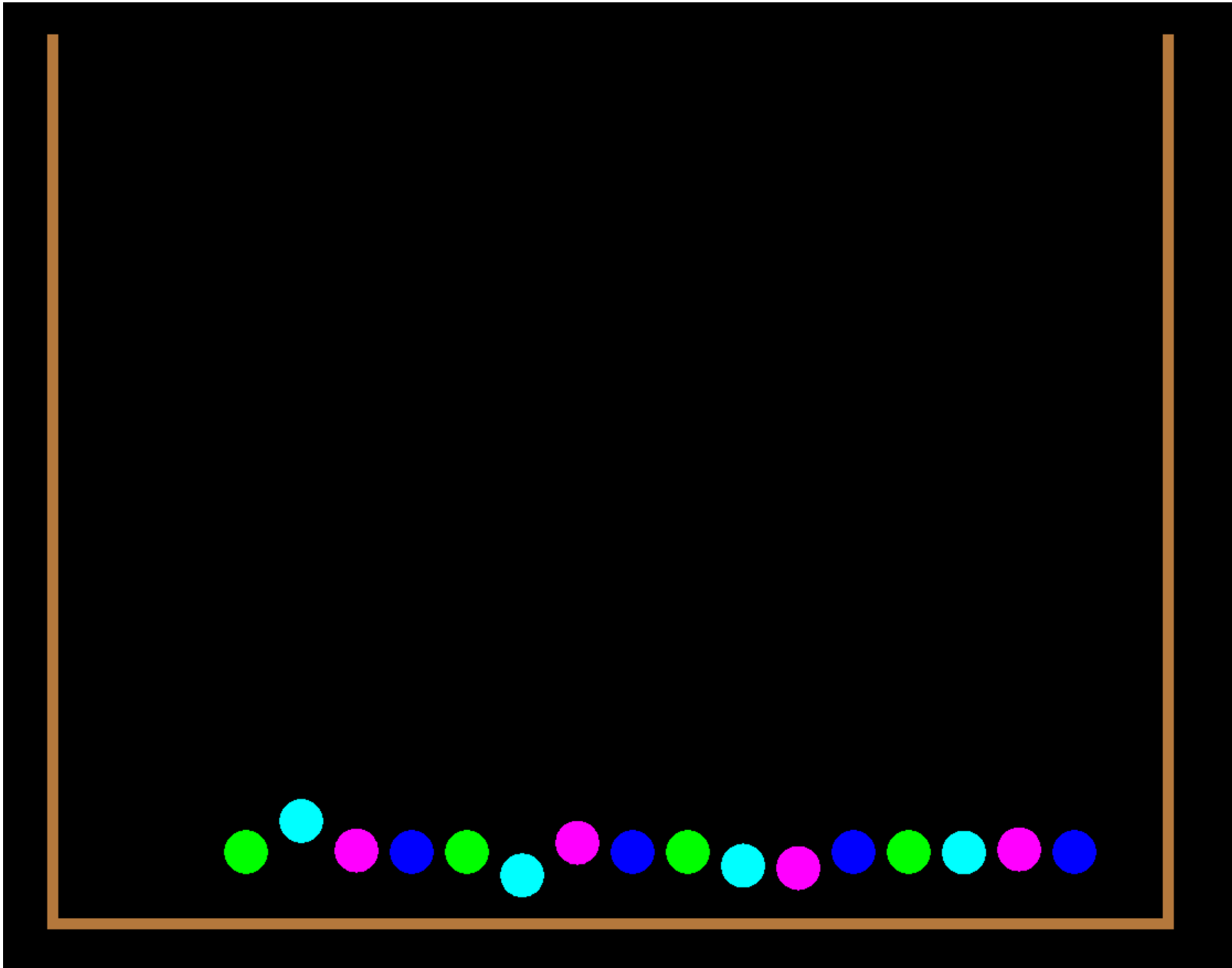
```
y += 20 + jump * 40.;
```

- Two distributions and the steppers
 - Stepper, $y += 20.0f$; green blob
 - Uniform, on a float $[0, 40]$; cyan blob
 - Normal, $N \sim (20.0f, 10.0f)$; magenta blob
 - Jumper, Poisson, lambda $1/32$; blue blob



```
>Race\Debug\SFMLRace.exe j
```

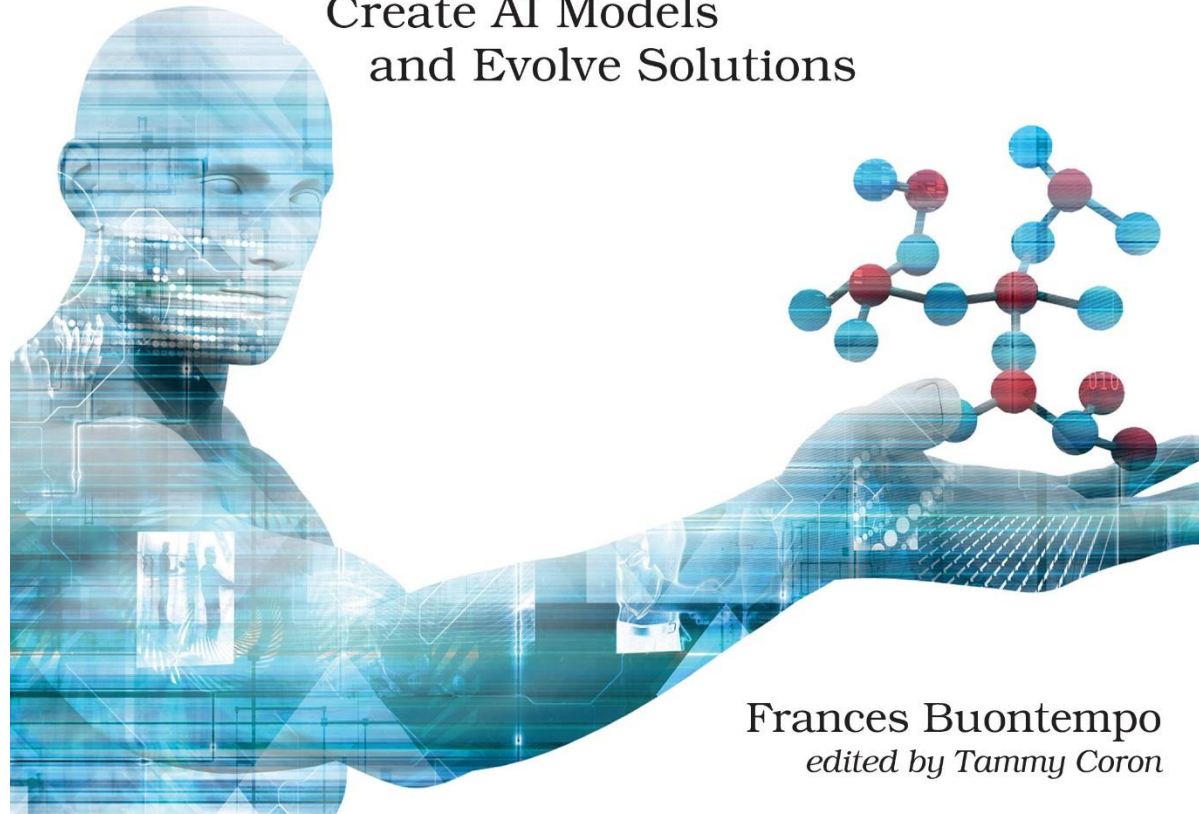
From https://en.wikipedia.org/wiki/Poisson_distribution



@fbuontempo

Genetic Algorithms and Machine Learning for Programmers

Create AI Models
and Evolve Solutions



Frances Buontempo
edited by Tammy Coron

C++ Book

35% discount code (good for all products in all formats): **ctwaccu23**

<http://mng.bz/1MPj>

