Randomization

10.21.24

Learning Objectives

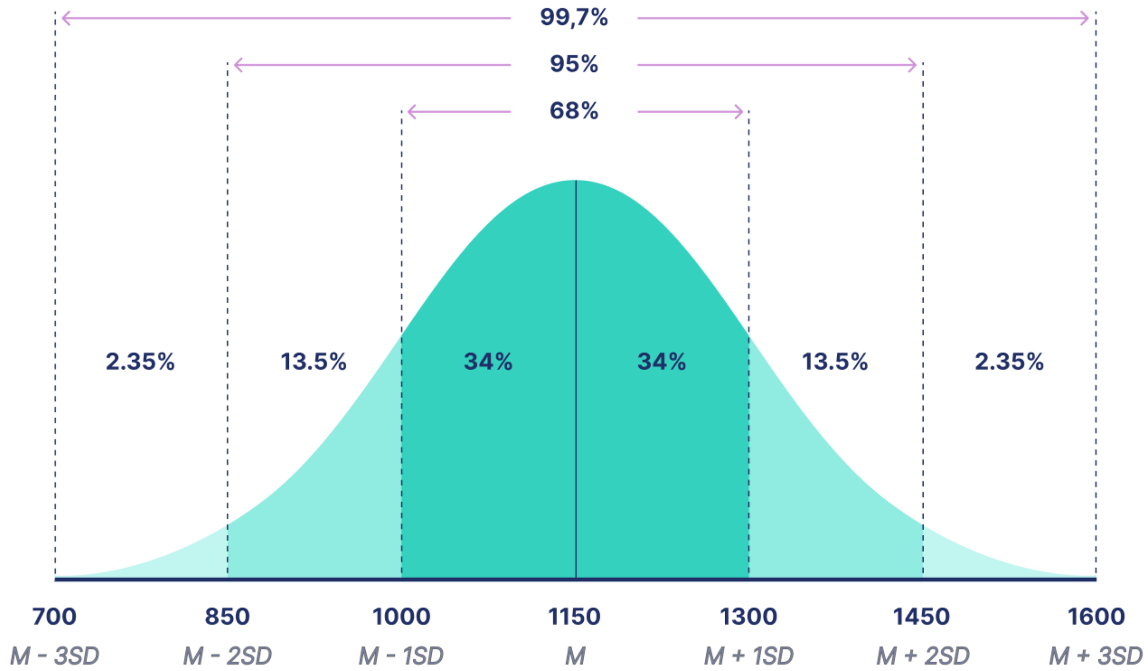
- Random Number Generation (RNG) algorithms
- Reproducible RNG streams via seeds
- Custom Distributional Sampling

Random Number Generators (RNG)

- Algorithmic (pseudo-random/ PRNG)

- Hardware (partially true-random)

What properties might we care about in a pseudo-RNG?



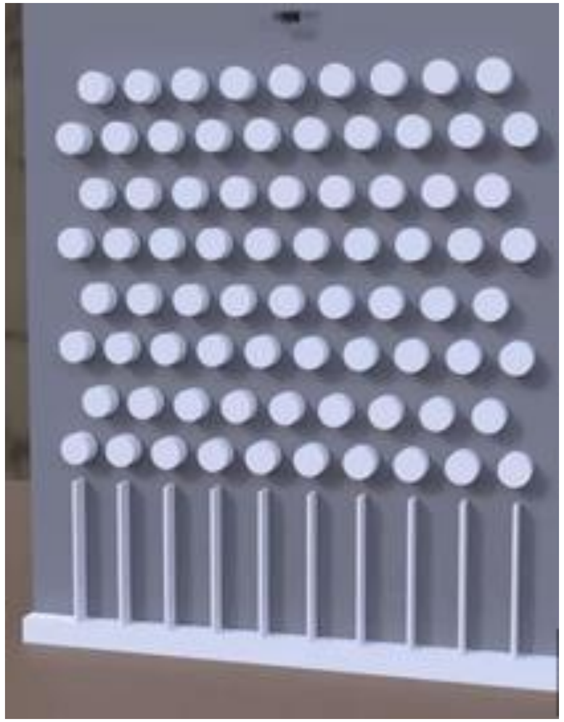
The tradeoff

Reproducibility

Independence

“True” random number generation

- “Cryptographically-Secure” RNG
- Acquire physical entropy and transform to useful distributions
- Each OS has internal algorithms to do this based upon logging local physical events
  - User input
  - Internal event timing
  - Electronic noise



Secure RNG with secrets

- secrets module calls the internal operating-system rng
  - Generates random byte-strings by calling **os.urandom**(nbyte)

1. Sampling: secrets**.choice**
2. Random integers: secrets**.randbelow** or secrets**.randbits**
3. Random addresses: secrets**.token_urlsafe**

Pseudo-randomness

- We want number streams that are:

1. Unbiased: Match theoretical distribution
2. Independent:
3. **Reproducible:** Numerically-stable as a fct. of previous values

- Even if you don't want (3), someone else does so it's a factor in PRNGs

How to create Pseudo-randomness?

- Chaotic mixing
- Small differences in initial conditions become large
- Distant future becomes uncorrelated with past

Naïve Mixing Demo

- Logistic map is a simple model for RNG (too weak, nonuniform in practice)

$$x_{t+1} = 4x_t(1 - x_t)$$

- Linear Congruential Generator:

$$x_{t+1} = (ax_t + c)\ mod\ m$$

(PCG64: a=6364136223846793005, c=1, m=2^{64})

Modern Implementation

- For a discrete-system, you can’t mix forever.

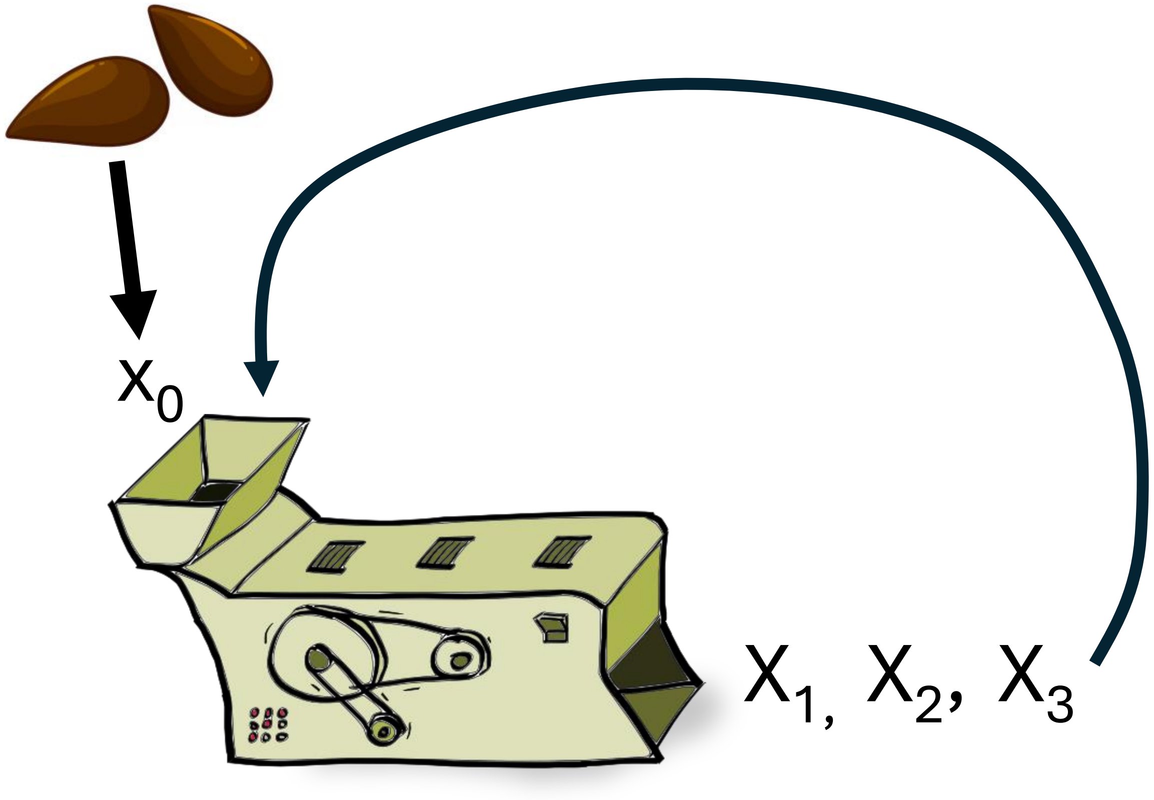
- Algorithm evolves as $x_{t+1} = f(x_t)$. Eventually you run out of unique values for x and the algorithm repeats.

- Marsenne Twister and PCG64: most popular RNGs
- random uses Twister, np.random uses PCG64 by default

- Bitwise definition ensures stability (no rounding involved)

Seeding an RNG

- Seed: determines the starting value of x_0.
- If you started with the same x_0, you'd get the same rng steam every time.
- Often true-random numbers used for seed.



Seeding an RNG

- Both random and np.random start with a true-random seed by default.
- Specify values by: random**.seed**() or np.random**.seed**()

- Get/set current state by:

    np.random**.get_state**()

    np.random**.set_state**()

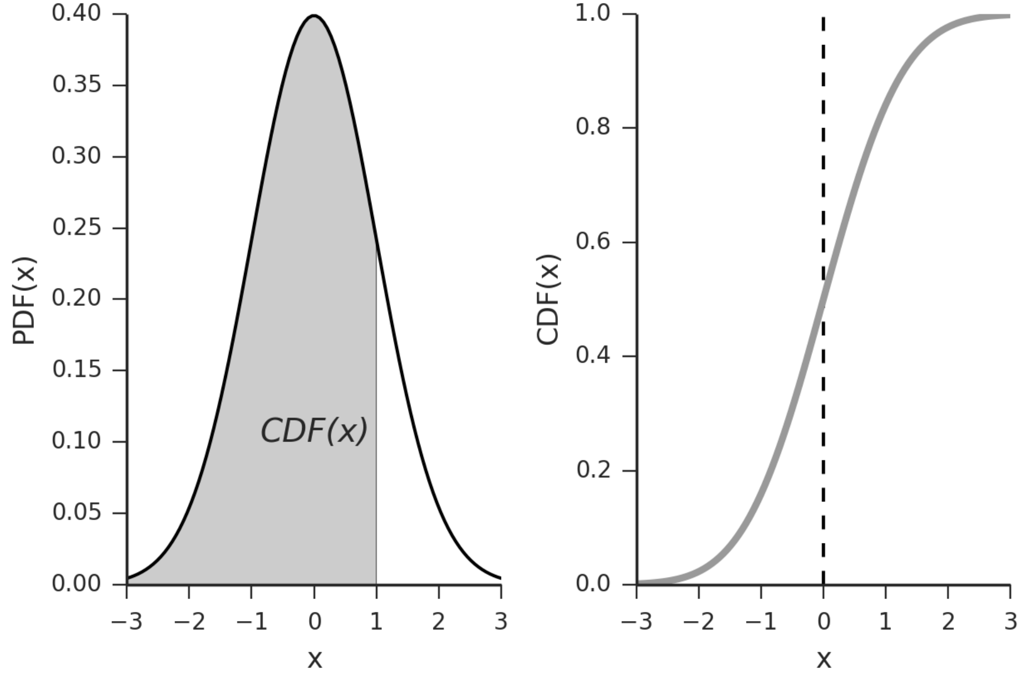
Pseudo RNG distributions

- Generally, try to approximate a **uniform distribution**
- Various algorithms can map a uniform onto an arbitrary distribution
- In 1d, we can define a surjection from (0,1) onto the support of a distribution using the CDF

Inverse Transform Sampling

For a target distribution $f(x)$: $(D \subseteq \mathbb{R})$ with CDF $F(x)$

1. Randomly generate $y \sim U([0,1])$
2. Set $x = F^{-1}(y)$



Sampling Distributions

- Numpy has builtin sampling from common distributions
  - (beta, T, F, etc.)

- Custom distributions: scipy.stats **rv_continuous** or **rv_discrete**

- Distributions are instances of the rv_continuous or rv_discrete classes

Sampling Distributions

- Continuous-Valued using an instance object:

```
from scipy.stats import rv_continuous, rv_discrete



class cust_samp(rv_continuous):
    def _pdf(self, x,sig):
        tmp=np.exp(-((x/sig)**2)/2);
        return tmp/(sig*np.sqrt(2*np.pi));

cust_gen = cust_samp(name='cust_samp');
sample=cust_gen.rvs(sig=1,size=500);
```

Sampling Distributions

- Discrete-Valued: Also instance, but simpler setup

```
Xset=[2,5,3,7];
Pset=[.2,.1,.3,.4];
cust2 = rv_discrete(name='cust2', values=(Xset,Pset))
```

The rng generator object

- Using the default RNG:

  rng=np.**random.default_rng**(**seed**)

- For specific rng, use np.random.Generator(rng_object)

  Call through methods: rng.random(), rng.choice() etc.

- Advantages: thread-safety, isolated states
- In parallel-settings can spawn a set of generators: np.random.Generator.spawn()

Learning Objectives

- Random Number Generation (RNG) algorithms
- Reproducible RNG streams via seeds
- Custom Distributional Sampling

Fin