Table of Contents

1 Simulation and Random Number Generation
   • Simulation of Physical Systems
   • Creating Fake Data Sets for Stress Tests
   • Parameter Estimation with Monte Carlo

2 Pseudo-Random Number Generators (PRNGs)
   • Linear Congruential Generators
   • Seeding the RNG
   • The Mersenne Twister
   • The Xorshift Algorithm
   • Juking the Stats: Benford’s Law

3 Sampling from Arbitrary PDFs
   • Inversion Method
   • Acceptance/Rejection Method
   • Generating Gaussian and Poisson Random Numbers
“Monte Carlo” methods are a broad set of techniques for calculating probabilities and related quantities using sequences of random numbers.

- Simulate physical systems with models of noise and uncertainty
- Simulate data with known inputs to stress-test your analysis ("data challenges"). Can be quite extensive...
- Perform calculations that cannot be done analytically or with a deterministic algorithm. E.g., function minimization, or many high-dimensional integrals
- **Inverse Monte Carlo**: estimate best-fit parameters with uncertainties using many simulated data sets – avoid explicit and difficult uncertainty propagation

All this depends upon the generation of (pseudo-)random numbers. This means **you MUST understand how random number generators (RNGs) work!**
Example Simulation from U of R Faculty

Physics of granular materials which become rigid with increasing density ("jamming" transition) [1]:

Universality of Jamming Criticality in Overdamped Shear-Driven Frictionless Disks

Daniel Vågberg,¹ Peter Olsson,¹ and S. Teitel²

¹Department of Physics, Umeå University, 901 87 Umeå, Sweden
²Department of Physics and Astronomy, University of Rochester, Rochester, New York 14627, USA

(Received 18 December 2013; revised manuscript received 4 August 2014; published 3 October 2014)

We investigate the criticality of the jamming transition for overdamped shear-driven frictionless disks in two dimensions for two different models of energy dissipation: (i) Durian’s bubble model with dissipation proportional to the velocity difference of particles in contact, and (ii) Durian’s “mean-field” approximation to (i), with dissipation due to the velocity difference between the particle and the average uniform shear flow velocity. By considering the finite-size behavior of pressure, the pressure analog of viscosity, and the macroscopic friction $\sigma/p$, we argue that these two models share the same critical behavior.
Example “Data Challenge”

The Laser Interferometer Gravitational Wave Observatory (LIGO) is (in)famous for carrying out extensive data challenges [2]

Very important to conduct end-to-end “stress tests” in background-dominated analyses. Above: fake binary merger injected into LIGO data stream, 2011
Example of Inverse Monte Carlo

From paper on discovery of cosmic-ray “hot spots” [3]:

PRL 101, 221101 (2008)

Discovery of Localized Regions of Excess 10-TeV Cosmic Rays


(Milagro Collaboration)
Table of Contents

1 Simulation and Random Number Generation
   • Simulation of Physical Systems
   • Creating Fake Data Sets for Stress Tests
   • Parameter Estimation with Monte Carlo

2 Pseudo-Random Number Generators (PRNGs)
   • Linear Congruential Generators
   • Seeding the RNG
   • The Mersenne Twister
   • The Xorshift Algorithm
   • Juking the Stats: Benford’s Law

3 Sampling from Arbitrary PDFs
   • Inversion Method
   • Acceptance/Rejection Method
   • Generating Gaussian and Poisson Random Numbers
Pseudo-Random Numbers

- We need to generate sequences of random numbers to model noise and uncertainty.

- Computers are not random, they are deterministic. So how do we get random sequences of numbers?
- Answer: we don’t. We get *pseudo*-random sequences and try to use them in clever ways.
Pseudo-Random Number Generators (RNGs)

Linear Congruential Generator

- An old but popular technique of generating pseudo-random number sequences is the linear congruential generator (LCG)
- A sequence of values \( x_i \) is generated using the recurrence relation

\[
x_{n+1} = (ax_n + c) \mod m
\]

- Generate integers in \([0, m - 1]\). The longest sequence with no repeating values, called the period of the RNG, is at most \(m\).
- Note: if \(m\) is an unsigned integer (\(\text{uint32}_t\) on most systems) then the period will be \(2^{32} \approx 4 \times 10^9\). \((2^{64} \approx 10^{18}.\) Most real simulations need orders of magnitude more numbers than this!
- Hull-Dobell Theorem: the full period is achieved iff \(c\) and \(m\) are co-prime, \(a - 1\) is divisible by all prime factors of \(m\), and \(a - 1\) is a multiple of 4 if \(m\) is a multiple of 4.
Choosing the Random Seed

- Note that the LCG is deterministic. If you start from the same \( x_0 \), a value known as the seed, you always get the same sequence.
- The choice of seed can affect the performance of the LCG; i.e., a poor choice could lead to a period \( \ll m \).
- Determinism is great for debugging, but if you generate the same numbers over and over you aren’t getting a pseudo-random sequence
- **Common mistake**: accidentally hardcoding the seed into your simulation code
- **Solution 1**: use system clock to choose \( x_0 \) via a call to `time(0)`; returns time in seconds since 00:00 UT, 1 Jan 1970 (Unix epoch).
  - Be careful to use the lowest-order bits of the time, including milliseconds. If you just use the seconds, what happens on a computing cluster if multiple jobs start simultaneously?

Good enough for physics simulations, but not cryptography
Choosing the Random Seed

- **Solution 2:** use the reservoir of random bits in computer’s entropy pool, accessible in `/dev/random`. Could be noise measured in a resistor, or clock drift [4], or a peripheral device connected to a source of quantum randomness.

- However you generate the seed, make sure you *always save the seed value* so you can regenerate the sequence later for checks!
Known Issues to Watch For

- The LCG is fast but has some known problems
- Many RNGs can produce hidden long-range correlations between values in the sequence.
- **Ex.:** if you generate $n$-dimensional points with the LCG, the points will lie on $(n!m)^{1/n}$ hyperplanes [5].
- Clearly random numbers shouldn’t do that.
- Could this affect your simulation? Maybe. Depends on your application.
Alternatives to the LCG

Mersenne Twister

- Most popular RNG currently in use is an algorithm called the Mersenne Twister [6], which uses the matrix linear recurrence relation

\[ x_{k+n} = x_{k+m} \oplus (x^u_k \mid x^l_{k+1})A \]

with \( \mid = \) bitwise OR and \( \oplus = \) bitwise XOR.

- For \( n = \) degree of recurrence, \( w = \) word size in bits, and \( 0 \leq r \leq w - 1 = \) bits in lower bitmask, the algorithm requires that the period length

\[ 2^{nw-r} - 1 \]

is a Mersenne prime – a prime number of the form \( 2^n - 1 \).

- The MT implementation in Python and C++ (Boost, ROOT) has period \( 2^{19937} - 1 \approx 4 \times 10^{6001} \).
Alternatives to the LCG
Xorshift Algorithms

- Another class of RNG is called Xorshift ("XOR-shift"), which depends on a combination of XOR and bit shift operations [7].
- These are extremely fast because XOR and shifting are simple CPU instructions. Example: a $2^{128} - 1$ period algorithm

```c
#include <cstdint>

// State variables; start s.t. not all = 0
uint32_t x, y, z, w;

uint32_t xorshift128() {
    uint32_t t = x ^ (x << 11);
    x = y; y = z; z = w;
    return w = w ^ (w >> 19) ^ t ^ (t >> 8);
}
```
Human-Generated Random Numbers

How good are you at generating random numbers?

Example

Without over-thinking this, take a minute to write down as many random values between 1 and 100 as you can.

What does the distribution of numbers look like?

How would you tell if this is really a random sequence? Is it easy to predict parts of the sequence (auto-correlation)?

Do we need to specify more information to answer this question?
Benford’s Law

- If you are like most people, you didn’t repeat numbers enough (remember the demon in the cartoon...)
- Also, your “random” sequence is probably uniform between 1 and 100
- However, in many sources of data the values follow a distribution known as Benford’s Law: 1 is the leading digit 30% of the time, 2 is the leading digit 18% of the time, etc.
- If you pick a number randomly from the logarithmic number line, it will roughly follow Benford’s Law

This rule can be used to detect fraudulent numbers in elections, accounting (stock prices), and scientific papers.
# Table of Contents

1. **Simulation and Random Number Generation**
   - Simulation of Physical Systems
   - Creating Fake Data Sets for Stress Tests
   - Parameter Estimation with Monte Carlo

2. **Pseudo-Random Number Generators (PRNGs)**
   - Linear Congruential Generators
   - Seeding the RNG
   - The Mersenne Twister
   - The Xorshift Algorithm
   - Juking the Stats: Benford’s Law

3. **Sampling from Arbitrary PDFs**
   - Inversion Method
   - Acceptance/Rejection Method
   - Generating Gaussian and Poisson Random Numbers
Generating Arbitrary Random Numbers

- All of the RNGs we have discussed will produce uniformly distributed random numbers:
  - LCG generates numbers between $[0, m]$
  - MT generates numbers between $[0, 1]$
- This is great for situations when you want a uniform distribution, but that does not correspond to most physical situations
- Luckily, there are several ways to convert a uniform distribution to an arbitrary distribution:
  1. Transformation or inversion method
  2. Acceptance/rejection method
- The transformation method is generally the most efficient technique, but it is only applicable in cases where the PDF you want is integrable and the CDF can be inverted
- Acceptance/rejection is less efficient but works for any PDF you will want to use for random draws
Transformation/Inversion Method

Given a PDF \( p(x|I) \) and its CDF \( F(x) = \int_{-\infty}^{x} p(x'|I) \, dx' \):

1. Generate a uniform random number \( u \) between \([0, 1]\)
2. Compute the value \( x \) s.t. \( F(x) = u \)
3. Take \( x \) to be the random draw from \( p(x|I) \)

In other words, from \( u \) and the invertible CDF \( F(x) \), the value \( x = F^{-1}(u) \) is distributed according to \( p(x|I) \).
Transformation/Inversion Method

Exponential Distribution

Example

The PDF of the exponential distribution is

\[
p(x | \xi) = \frac{1}{\xi} e^{-x / \xi}
\]

and the CDF is

\[
F(x) = P(X \leq x | \xi) = \int_0^x \frac{1}{\xi} e^{-x' / \xi} \, dx' = 1 - e^{-x / \xi}
\]

Therefore, given \( u \in [0, 1] \) we can generate \( x \) according to \( p(x | \xi) \) by inverting the CDF:

\[
u = F(x) = 1 - e^{-x / \xi}
\]

\[
x = F^{-1}(u) = -\xi \ln (1 - u) = -\xi \ln u
\]
Limits of the Inversion Method

- Inversion is very efficient and great if you can invert your CDF
- Unfortunately this condition is not fulfilled even for many basic 1D cases

Example

The CDF of the Gaussian distribution is

\[
F(x) = \int_{-\infty}^{x} p(x|\mu, \sigma) = \frac{1}{2} \left[ 1 + \text{erf} \left( \frac{x - \mu}{\sigma \sqrt{2}} \right) \right]
\]

The error function cannot be expressed in closed form, though there are numerical approximations to \( \text{erf} \) and \( \text{erf}^{-1} \) in scipy.

- A trick for complicated PDFs: express the CDF as a tabulated list of values \( (u, F(x)) \), “invert” it, and interpolate.
Acceptance/Rejection Method

Very old technique; modern form due to von Neumann. AKA “hit and miss,” it generates $x$ from an arbitrary $f(x)$ using a so-called instrumental distribution $g(x)$, where $f(x) < Mg(x)$ and $M > 1$ is a bound on $f(x)/g(x)$.

1. Sample $x$ from $g(x)$ and $u \in [0, 1]$.
2. Check if $u < f(x)/Mg(x)$
   - Yes: accept $x$
   - No: reject $x$, sample again

Very easy to implement, no limits on $f(x)$.

Calculation of $\pi$: uniformly generate $(x, y)$ pairs in box, count up points inside the circle. $\pi \approx 4N_{\text{circle}}/N_{\text{box}}$. 
Buffon’s Calculation of $\pi$

An early variant of the Monte Carlo approach can be seen in Buffon’s Needle (1700s), a method of calculating $\pi$

Given a needle of length $L$ dropped on a plane with parallel lines $d$ units apart, what is the probability the needle will cross a line if $L < d$?

- $x$ is center distance to nearest line; $x \sim U(0, d/2)$
- $\theta$ is angle between needle center line: $\theta \sim U(0, \pi/2)$
- Needle crosses line if $x \leq L \sin \theta/2$. Joint PDF:

$$P = \int_0^{\pi/2} d\theta \int_0^{L \sin \theta/2} dx \frac{4}{\pi d} = \frac{2L}{\pi d}$$
Acceptance/Rejection Method
Sampling from a 1D Distribution

Example

Suppose $f(x) = \frac{3}{8}(1 + x^2)$ for $-1 \leq x \leq 1$. (Aside: do you recognize this distribution?)

- Generate random $x \in [-1, 1]$ and $y \in [0, 0.75]$.
- If $y < f(x)$, populate the histogram with $x$. 
Acceptance/Rejection Method
Sampling from a 2D Distribution

**Example**

Suppose we want to sample from the 2D angular distribution

\[
\frac{1}{N} \frac{dN}{d \cos \theta d \varphi} = (1 + \cos \theta)(1 + \frac{1}{2} \cos 2\varphi)
\]

Generate triplets \((x, y, z)\), where \(x = \varphi \in [0, 2\pi]\), \(y = \cos \theta \in [-1, 1]\), and \(z \in [0, 3]\), keeping \((x, y)\) if \(z < f(x, y)\):

![Angular Distribution](image1)

![1,000,000 Events](image2)
Limitations of Acceptance/Rejection

Ideally you know $f_{\text{max}}$ or normalize $f(x) = p(x|I)$ to have a maximum of 1.

- If not, you’ll have to pre-scan the parameter space in advance.

If $f(x)$ ranges over many orders of magnitude, acceptance/rejection can be very inefficient as you’ll waste lots of time in low-probability regions.

Possible approaches:

- Subdivide $x$ into ranges with different $f_{\text{max}}$.
- Use importance sampling, where you generate random numbers according to a function that envelops the PDF you really want to sample.

Example implementation: vegas package in Python, an implementation of the adaptive Monte Carlo VEGAS multi-dimensional integration algorithm [8]
Monte Carlo Integration

- We can also solve integrals (esp. in several dimensions) with Monte Carlo. Mathematically, we approximate the integral by the average of the function of the interval of integration:

\[ I = \int_{a}^{b} f(x) \, dx \approx (b - a) \, E(f(x)) \]

- We take discrete samples of \( f \) and let the MC estimate converge to the true integral as the number of samples gets large:

\[
E(f(x)) = \frac{1}{N} \sum_{i=1}^{N} f(u_i) \rightarrow \frac{1}{b - a} \int_{a}^{b} f(u) \, du
\]

\[ I = I_{MC} = \frac{b - a}{N} \sum_{i=1}^{N} f(x_i) \]

- Error on the result given by the Central Limit Theorem:

\[ \sigma = \frac{\sqrt{V(f)}}{\sqrt{N}} \propto \frac{1}{\sqrt{N}} \]
Generating a Gaussian Random Number

How would you generate a Gaussian random number?

1. You can use inversion if you can numerically estimate \( \text{erf}^{-1} \).
2. You can use the acceptance/rejection method if you don’t mind wasting some calculations.
3. You can exploit the Central Limit Theorem. Sum 12 uniform variables, which approximates a Gaussian of mean \( 12 \times 0.5 = 6 \) and a variance of \( 12 \times (1/12) = 1 \). Subtract 6 to get a mean of zero. This takes even more calculation and isn’t exact.
4. Use the polar form of the binormal distribution

\[
p(x, y|I) = \frac{1}{2\pi} \exp \left\{ -\frac{1}{2} (x^2 + y^2) \right\}
\]

To generate two Gaussian random numbers at once.
Box-Müller Algorithm

Re-express the 2D Gaussian PDF in polar coordinates:

\[ p(x, y| l) \, dx \, dy = \frac{1}{2\pi} \exp \left\{ -\frac{1}{2} (x^2 + y^2) \right\} \, dx \, dy \]

\[ = \frac{1}{2\pi} \exp \left\{ -\frac{r^2}{2} \right\} \, r \, dr \, d\varphi \]

Then generate an exponential variable \( z = r^2 / 2 \), change variables to \( r \), and generate a uniform polar angle \( \varphi \):

\[ z = -\ln u_1 \text{ for } u_1 \sim U(0, 1) \]
\[ r = \sqrt{2z} \]
\[ \varphi = 2\pi u_2 \text{ for } u_2 \sim U(0, 1) \]

Then \( x = r \cos \varphi \) and \( y = r \sin \varphi \) are two normally-distributed random numbers. Very elegant! But due to the calls to trascendental functions (\( \sqrt{r} \), \( \log \), \( \cos \), etc.), numerical approaches could be faster in practice...
Generating a Poisson Random Variable

The best way to generate a Poisson random variable is to use inverse transform sampling of the cumulative distribution.

1. Generate \( u \sim U(0,1) \)
2. Sum up the Poisson PDF \( p(n|\lambda) \) with increasing values of \( n \) until the cumulative sum exceeds \( u \):

\[
  s_n = \sum_{k=0}^{n} \frac{\lambda^k e^{-\lambda}}{k!}, \quad \text{while } s_n < u
\]

3. Return the largest \( n \) for which \( s_n < u \).

This will work quite well until \( \lambda \) gets large, at which point you may start experiencing floating-point round-off errors due to the factor of \( e^{-\lambda} \). But for large \( \lambda \) you can start to use the Gaussian approximation.
References


References II


