Randomness and Simulation

L. Olson

September 8, 2015

Department of Computer Science
University of Illinois at Urbana-Champaign
• randomness
• reproducibility
• designing an experiment
The Scientific Method as an Ongoing Process

1. **Make Observations**
   What do I see in nature?
   This can be from one's own experiences, thoughts, or reading.

2. **Think of Interesting Questions**
   Why does that pattern occur?

3. **Refine, Alter, Expand, or Reject Hypotheses**
   Relevant data can come from the literature, new observations, or formal experiments. Thorough testing requires replication to verify results.

4. **Gather Data to Test Predictions**
   General theories must be consistent with most or all available data and with other current theories.

5. **Develop Testable Predictions**
   If my hypothesis is correct, then I expect a, b, c,...

6. **Formulate Hypotheses**
   What are the general causes of the phenomenon I am wondering about?

7. **Develop General Theories**
   The scientific method
• How do I classify my method?
• Goal: determine how the error $|f(x) - p_n(x)|$ behaves relative to $n$ (and $f$).
• Goal: determine how the cost of computing $p_n(x)$ behave relative to $n$ (and $f$).
• for $f(x) = \frac{1}{1-x}$ we have

$$p_n = \sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots$$

• so

$$e_n = |f(x) - p_n(x)|$$

• Is $e_n \sim 1/n^r$?
• Is $e_n \sim 1/\sqrt{n}$?
• Is $e_n \sim 1/n!$?
• `mymethod()` takes $x$ seconds
• How long does it take in general?
• If the data input is of size $n$, how long should it take?
  • $n^2$?
  • $n!$?
  • $10^n$?
How to measure the impact of $n$ on algorithmic cost?

$\mathcal{O}(\cdot)$

Let $g(n)$ be a function of $n$. Then define

$$\mathcal{O}(g(n)) = \{ f(n) | \exists c, n_0 > 0 : 0 \leq f(n) \leq cg(n), \forall n \geq n_0 \}$$

That is, $f(n) \in \mathcal{O}(g(n))$ if there is a constant $c$ such that $0 \leq f(n) \leq cg(n)$ is satisfied.

- assume non-negative functions (otherwise add $|\cdot|$) to the definitions
- $f(n) \in \mathcal{O}(g(n))$ represents an asymptotic upper bound on $f(n)$ up to a constant
- example: $f(n) = 3\sqrt{n} + 2\log n + 8n + 85n^2 \in \mathcal{O}(n^2)$
Let $g(n)$ be a function of $n$. Then define

$$O(g(n)) = \{ f(n) \mid \exists c, n_0 > 0 : 0 \leq f(n) \leq cg(n), \forall n \geq n_0 \}$$

That is, $f(n) \in O(g(n))$ if there is a constant $c$ such that $0 \leq f(n) \leq cg(n)$ is satisfied.
Let $g(n)$ be a function of $n$. Then define

$$\Omega(g(n)) = \{ f(n) \mid \exists c, n_0 > 0 : 0 \leq cg(n) \leq f(n), \forall n \geq n_0 \}$$

That is, $f(n) \in \Omega(g(n))$ if there is a constant $c$ such that $0 \leq cg(n) \leq f(n)$ is satisfied.
Let $g(n)$ be a function of $n$. Then define

$$\Theta(g(n)) = \{ f(n) | \exists c_1, c_2, n_0 > 0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0 \}$$

Equivalently, $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$. 
randomness

- Randomness $\approx$ unpredictability
- One view: a sequence is random if it has no shorter description
- Physical processes, such as flipping a coin or tossing dice, are deterministic with enough information about the governing equations and initial conditions.
- But even for deterministic systems, sensitivity to the initial conditions can render the behavior practically unpredictable.
- We need random simulation methods

http://www.xkcd.com/221/

```c
int get_random_number()
{
    return 4; // chosen by fair dice roll.
    // guaranteed to be random.
}
```
randomness is easy, right?

- In May, 2008, Debian announced a vulnerability with OpenSSL: the OpenSSL pseudo-random number generator
  - the seeding process was compromised (2 years)
  - only 32,767 possible keys
  - seeding based on process ID (this is not entropy!)
  - all SSL and SSH keys from 9/2006 - 5/2008 regenerated
  - all certificates recertified
- Cryptographically secure pseudorandom number generator (CSPRNG) are necessary for some apps
- Other apps rely less on true randomness
• With unpredictability, true randomness is not repeatable
• ...but lack of repeatability makes testing/debugging difficult
• So we want repeatability, but also independence of the trials

```python
>>> np.random.seed(1234)
```
Computer algorithms for random number generations are deterministic

- ...but may have long periodicity (a long time until an apparent pattern emerges)
- These sequences are labeled *pseudorandom*
- Pseudorandom sequences are predictable and reproducible (this is mostly good)
Properties of a good random number generator:

Random pattern: passes statistical tests of randomness

Long period: long time before repeating
   Efficiency: executes rapidly and with low storage

Repeatability: same sequence is generated using same initial states
   Portability: same sequences are generated on different architectures
random number generators

- Early attempts relied on complexity to ensure randomness
- “midsquare” method: square each member of a sequence and take the middle portion of the results as the next member of the sequence
- ...simple methods with a statistical basis are preferable
linear congruential generators

- Congruential random number generators are of the form:

\[ x_k = (a x_{k-1} + c) \pmod{M} \]

where \( a \) and \( c \) are integers given as input.

- \( x_0 \) is called the seed

- Integer \( M \) is the largest integer representable (e.g. \( 2^{31} - 1 \))

- Quality depends on \( a \) and \( c \). The period will be at most \( M \).

Example

Let \( a = 13 \), \( c = 0 \), \( m = 31 \), and \( x_0 = 1 \).

\[
1, 13, 14, 27, 10, 6, \ldots
\]

This is a permutation of integers from 1, \ldots, 30, so the period is \( m - 1 \).
• IBM used Scientific Subroutine Package (SSP) in the 1960’s the mainframes.
• Their random generator, \texttt{rand} used \(a = 65539\), \(c = 0\), and \(m = 2^{31}\).
• arithmetic mod \(2^{31}\) is done quickly with 32 bit words.
• multiplication can be done quickly with \(a = 2^{16} + 3\) with a shift and short add.
• Notice (mod \(m\)):

\[
x_{k+2} = 6x_{k+1} - 9x_k
\]

...strong correlation among three successive integers
Matlab used $a = 7^5$, $c = 0$, and $m = 2^{31} - 1$ for a while.

- period is $m - 1$.
- this is no longer sufficient.
what’s used?

Two popular methods:

1. Method of Marsaglia (period \( \approx 2^{1430} \)).

   Initialize \( x_0, \ldots, x_3 \) and \( c \) to random values given a seed

   Let \( s = 2111111111x_{n-4} + 1492x_{n-3}1776x_{n-2} + 5115x_{n-1} + c \)

   Compute \( x_n = s \mod 2^{32} \)

   \( c = \text{floor}(s/2^{32}) \)

2. \texttt{rand()} in Unix uses \( a = 1103515245, c = 12345, m = 2^{31} \).

Even the Marsaglia method produces points in \( n - D \) on only a small number of hyperplanes.
linear congruential generators

- sensitive to $a$ and $c$

- be careful with supplied random functions on your system

- period is $M$

- standard division is necessary if generating floating points in $[0, 1)$. 
• produce floating-point random numbers directly using differences, sums, or products.

• Typical subtractive generator:

\[ x_k = x_{k-17} - x_5 \]

with “lags” of 17 and 5.

• Lags much be chosen very carefully

• negative results need fixing

• more storage needed than congruential generators

• no division needed

• very very good statistical properties

• long periods since repetition does not imply a period
If we need a uniform distribution over \([a, b)\), then we modify \(x_k\) on \([0, 1)\) by

\[(b - a)x_k + a\]
• sampling nonuniform distributions is much more difficult
• if the cumulative distribution function is invertible, then we can generate the non-uniform sample from the uniform:

\[
f(t) = \lambda e^{-\lambda t}, \quad t > 0
\]

thus

\[
y_k = -\log(1 - x_k)/\lambda
\]

where \( x_k \) is uniform in [0, 1).
• ...not so easy in general
quasi-random sequences

- For some applications, reasonable uniform coverage of the sample is more important than the “randomness”
- True random samples often exhibit clumping
- Perfectly uniform samples use a uniform grid, but do not scale well at high dimensions
- quasi-random sequences attempt randomness while maintaining coverage
quasi-random sequences

- quasi random sequences are not random, but give random appearance
- by design, the points avoid each other, resulting in no clumping