SIMULATION SYSTEMS

PART 1: RANDOM NUMBER GENERATION

Prof. Yasser Mostafa Kadah

Introduction to Simulation

- Simulating the process can be useful as a validation of a model or a comparison of two different models
- If we have reason to trust our model, then simulation can further be used to explore how interventions in the model might affect its behavior
- Simulations are also useful if the long-term behavior of the model is hard to analyze by first principles.
  - In such cases, we can look at how a model evolves and watch for particularly interesting but unexpected properties

Random Number Generation

- It may seem perverse to use a computer, that most precise and deterministic of all machines conceived by the human mind, to produce "random" numbers
  - More than perverse, it may seem to be a conceptual impossibility.
  - After all, any program produces output that is entirely predictable, hence not truly "random"
- One sometimes hears computer-generated sequences termed *pseudo-random*, while the word *random* is reserved for the output of an intrinsically random physical process

Random Number Generation

- A working definition of randomness in the context of computer-generated sequences is to say that the deterministic program that produces a random sequence should be different from, and — in all measurable respects — statistically uncorrelated with, the computer program that uses its output
  - pragmatic point of view is thus that randomness is in the eye of the beholder
  - What is random enough for one application may not be random enough for another.

Uniform Deviates

- Uniform deviates are just random numbers that lie within a specified range, typically 0.0 to 1.0 for floating-point numbers, or 0 to $(2^{32}-1)$ or $(2^{64}-1)$ for integers
  - Within the range, any number is just as likely as any other
- New high-performance methods are now available
  - Expect to get "perfect" deviates in no more than a dozen or so arithmetic or logical operations per deviate, and fast, "good enough" deviates in many fewer operations than that

Uniform Deviates

- Many out-of-date and inferior methods remain in general use
  - Never use a generator principally based on a linear congruential generator (LCG) or a multiplicative linear congruential generator (MLCG)
  - Never use a generator with a period less than 2^{64}, or any generator whose period is undisclosed
  - Never use a generator that warns against using its low-order bits as being completely random.
    - That was good advice once, but it now indicates an obsolete algorithm (usually a LCG).
  - Never use the built-in generators in the C and C++ languages, especially rand and srand
    - These have no standard implementation and are often badly flawed.

Uniform Deviates

- Indications that a generator is “over-engineered”, and therefore wasteful of resources:
  - Avoid generators that take more than (say) two dozen arithmetic or logical operations to generate a 64-bit integer or double precision floating result
  - Avoid using generators (over-)designed for serious cryptographic use
  - Avoid using generators with period > 10^{100}. You really will never need it, and, above some minimum bound, the period of a generator has little to do with its quality
- An acceptable random generator must combine at least two (ideally, unrelated) methods. The methods combined should evolve independently and share no state. The combination should be by simple operations that do not produce results less random than their operands.

Uniform Deviates: History

- Recurrence relation:

$$\boxed{I_{j+1} = aI_j + c \pmod{m}}$$

- Here m is called the *modulus*, a is a positive integer called the *multiplier*, and c (which may be zero) is nonnegative integer called the *increment.*
- For c≠0, this equation is called a linear congruential generator (LCG)
- When c ≠0, it is sometimes called a multiplicative LCG or MLCG

Uniform Deviates

$$I_{j+1} = aI_j + c \pmod{m}$$

- LCG must eventually repeat itself, with a period that is obviously no greater than m
  - If m, a, and c are properly chosen, then the period will be of maximal length, i.e., of length m.
  - In that case, all possible integers between 0 and m-1 occur at some point, so any initial “seed” choice of I_0 is as good as any other
- Ex: a=3, c=1, m=5, I_0=1: 1,4,3,0,1,.....

Uniform Deviates

- LCGs and MLCGs have additional weaknesses: When m is chosen as a power of 2 (e.g., RANDU), then the low-order bits generated are hardly random at all.
  - In particular, least significant bit has a period of at most 2, the second at most 4, the third at most 8, and so on!
- An elegant number-theoretical test of m and a, the spectral test, was developed to characterize the density of planes in arbitrary dimensional space
- The field's long preoccupation with LCGs was somewhat misguided!!

Recommended Methods for Use in Combined Generators

- To be recommendable for use in a combined generator, we require a method to be understood theoretically to some degree, and to pass a reasonably broad suite of empirical tests
  - Diehard battery of statistical tests or NIST-STS test suite

A) 64-bit Xorshift Method

- In just three XORs and three shifts (generally fast operations), it produces a full period of 2^{64}-1 on 64 bits

ID	a_1	a_2	a_3
A1	21	35	4
A2	20	41	5
A3	17	31	8
A4	11	29	14
A5	14	29	11
A6	30	35	13
A7	21	37	4
A8	21	43	4
A9	23	41	18

state: x (unsigned 64-bit)

initialize: $x \neq 0$

update:
$$x \leftarrow x \wedge (x \;>>\; a_1),$$
$$x \leftarrow x \wedge (x \;<<\; a_2),$$
$$x \leftarrow x \wedge (x \;>>\; a_3);$$

or
$$x \leftarrow x \wedge (x \;<<\; a_1),$$
$$x \leftarrow x \wedge (x \;>>\; a_2),$$
$$x \leftarrow x \wedge (x \;<<\; a_3);$$

can use as random: x (all bits) $\ast$

can use in bit mix: x (all bits)

can improve by: output 64-bit MLCG successor

period: $2^{64} - 1$

B) Multiply with Carry (MWC) with Base $b=2^{32}$

state:	x (unsigned 64-bit)
initialize:	$1 \le x \le 2^{32} - 1$
update:	$x \leftarrow a\ (x\ \ \&\ \ [2^{32} - 1]) + (x\ \ >>\ \ 32)$
can use as random:	x (low 32 bits) ⁎
can use in bit mix:	x (all 64 bits)
can improve by:	output 64-bit xorshift successor to 64 bit x
period:	$(2^{32}a - 2)/2$ (a prime)

ID	a
B1	4294957665
B2	4294963023
B3	4162943475
B4	3947008974
B5	3874257210
B6	2936881968
B7	2811536238
B8	2654432763
B9	1640531364

C) LCG Modulo 2^{64}

state:	x (unsigned 64-bit)
initialize:	any value
update:	$x \leftarrow ax + c \pmod{2^{64}}$
can use as random:	x (high 32 bits, with caution)
can use in bit mix:	x (high 32 bits)
can improve by:	output 64-bit xorshift successor
period:	2^{64}

ID	a	c (any odd value ok)
C1	3935559000370003845	2691343689449507681
C2	3202034522624059733	4354685564936845319
C3	2862933555777941757	7046029254386353087

D) MLCG Modulo 2^{64}

state:	x (unsigned 64-bit)
initialize:	$x \neq 0$
update:	$x \leftarrow ax \pmod{2^{64}}$
can use as random:	x (high 32 bits, with caution)
can use in bit mix:	x (high 32 bits)
can improve by:	output 64-bit xorshift successor
period:	2^{62}

ID	a
D1	2685821657736338717
D2	7664345821815920749
D3	4768777513237032717
D4	1181783497276652981
D5	702098784532940405

E) MLCG with m≫2^{32}, m Prime

state:	x (unsigned 64-bit)
initialize:	$1 \le x \le m-1$
update:	$x \leftarrow ax \pmod{m}$
can use as random:	$x \quad (1 \le x \le m-1)$ or low 32 bits ⋇
can use in bit mix:	(same)
period:	$m-1$

ID	m	a
E1	$2^{39}-7 = 549755813881$	10014146
E2		30508823
E3		25708129
E4	$2^{41}-21 = 2199023255531$	5183781
E5		1070739
E6		6639568
E7	$2^{42}-11 = 4398046511093$	1781978
E8		2114307
E9		1542852
E10	$2^{43}-57 = 8796093022151$	2096259
E11		2052163
E12		2006881

F) MLCG with $m \gg 2^{32}$, m Prime, and $a(m-1) \approx 2^{64}$

state:	x (unsigned 64-bit)
initialize:	$1 \leq x \leq m-1$
update:	$x \leftarrow ax \pmod{m}$
can use as random:	$x \quad (1 \leq x \leq m-1)$ or low 32 bits ⁎
can use in bit mix:	ax (but don't use both ax and x) ⁎
can improve by:	output 64-bit xorshift successor of ax
period:	$m-1$

ID	m	a
F1	$1148 \times 2^{32} + 11 = 4930622455819$	3741260
F2	$1264 \times 2^{32} + 9 = 5428838662153$	3397916
F3	$2039 \times 2^{32} + 3 = 8757438316547$	2106408

How to Construct Combined Generators

- The methods being combined should be independent of one another
- The output of the combination generator should in no way perturb the independent evolution of the individual methods, nor should the operations effecting combination have any side effects
- The methods should be combined by binary operations whose output is no less random than one input if the other input is held fixed.
  - For 32- or 64-bit unsigned arithmetic, this in practice means that only the + and ^ operators can be used.
  - Example of a forbidden operator: multiplication

Examples of Combined Generator

$$\boxed{\mathrm{Ran} = [\mathrm{A1}_l(\mathrm{C3}) + \mathrm{A3}_r] \wedge \mathrm{B1}}$$

- Combination and/or composition of four different generators. For the methods A1 and A3, the subscripts *l* and *r* denote whether a left- or right-shift operation is done first. The period of Ran is the least common multiple of the periods of C3, A3, and B1.
- Another Example: $\boxed{\mathrm{Ranq2} \equiv \mathrm{A3}_r \wedge \mathrm{B1}}$

Assignments

- Implement and compare the random number generators described in this part using the Diehard battery of tests