Physics 750: Exercise 14

Tuesday, November 7, 2017

In this exercise, we consider the problem of how to generate pseudo-random sequences. Since the computer is purely deterministic, the best we can hope for is to find a computational rule whose output is close to random. More precisely, the rule should produce numbers whose statistical properties are approximately those of numbers drawn from a random distribution.

Suppose we have a single-argument rule denoted F, then the sequence $X_0, X_1, X_2, \ldots$ corresponds to

$$\begin{aligned} X_0 & \\ X_1 &= F(X_0) \\ X_2 &= F(X_1) = F(F(X_0) = F^2(X_0) \\ &\vdots \\ X_n &= F(X_{n-1}) = \underbrace{F(F(\cdots F}_{n \text{ times}}(X_0))) = F^n(X_0) \end{aligned}$$

The initial value X_0 is called the seed. The smallest value p such that $F^p(X_0) = X_0$ is the *period* of F. Since the function will execute repeating cycles $X_0, X_1, \ldots, X_{p-1}, X_0, X_1, \ldots$, we clearly want p to be large. At the very least, it should be larger than the number of random numbers we need. Note that F could also depend on more than one previous value. A two-argument rule might produce a sequence $X_2 = F(X_0, X_1), X_3 = F(X_2, X_1), \ldots$, but would require two seeds, X_0 and X_1. In this case, the period is defined by $F(X_p, X_{p-1}) = X_0$ and $F(X_0, X_{p-1}) = X_1$.

In addition to having a long period, a suitable generator must produce numbers that are only weakly correlated with one another. Theoretical results is this area are rather scarce, so correlation is usually judged with empirical tests. In this exercise, you'll have a chance to implement a few such tests.

Typical generators are defined using a recursion rule restricted to a finite range of integers $\{0, 1, 2, \ldots, m-1\}$:

$$\begin{aligned} \text{Linear congruential:} &\quad X_{n+1} = (aX_n + c) \bmod m \\ \text{Quadratic congruential:} &\quad X_{n+1} = (aX_n^2 + bX_n + c) \bmod m \\ \text{Fibonacci:} &\quad X_{n+1} = (X_n + X_{n-1}) \bmod m \\ \text{Lagged Fibonacci:} &\quad X_{n+1} = (X_{n-j} + X_{n-k}) \bmod m \end{aligned}$$

A rather nice simplification is that with the choice of $m = 2^{32}$, each number fits into one machine word and there is no need to compute the modulus (the high bits are properly truncated during overflow).

There are some subtleties when it comes to implementation. The C++ language does not specify the bit sizes of the historic numeric types. Their sizes are platform-dependent. At most, their *relative* sizes are guaranteed: for example, `sizeof(int) <= sizeof(long int)`. But C++ now supports integer types of fixed width (`int8_t`, `int16_t`, `int32_t`, `int64_t`, and corresponding unsigned versions `uint8_t`, etc.). These are defined in the `cstdint` header file. This is an official part of the language as of C++11.

A generator defined modulo 2^{32} produces a sequence of numbers of type `uint32_t`. For most applications, you are likely to be interested in random integer or floating point numbers distributed uniformly in some arbitrary interval $[a, b]$ or $[a, b)$. In order to transform the `uint32_t`s into the types you want in the ranges you want, it is helpful to understand how to manipulate the underlying bit patterns. C++ has inherited all the standard bitwise operators from C. There is a unary operator ~ that takes the bitwise complement of a number, $\sim(b_{31} \cdots b_0)_2 = ([1 - b_{31}] \cdots [1 - b_0])_2$; in other words, all zeros are exchanged for ones and vice versa. There are also five binary operators, `&`, `|`, `^`, `<<`, and `>>`. The first three denote (bitwise) logical AND, OR, and XOR (exclusive or). The last two are the left and right bit shift operators.

As an illustration, consider `(72 & 184 == 8)` and `(72 | 184 == 248)` and `(72 ^ 184 == 170))`

	01001000		01001000		01001000
AND	10111000	OR	10111000	XOR	10111000
	00001000		11111000		11110000

In the examples above, the operations act on bits b and b' in the same position: b AND b' yields 1 if both b and b' are 1; b OR b' yields 1 if either b or b' are 1; and b XOR b' yields 1 if either b or b' are 1 but not both. The bit shift operators behave as follows:

$$(b_{31}b_{30}\cdots b_2b_0)_2 \ll n = (b_{32-n}\cdots b_2b_1\underbrace{0\cdots 00}_{n\text{ times}})_2$$

$$(b_{31}b_{30}\cdots b_1b_0)_2 \gg n = (\underbrace{0\cdots 00}_{n\text{ times}}b_{31}\cdots b_{n+1}b_n)_2$$

The following operators can be used to set, clear, toggle, and test bits.

```
const uint8_t mask = 0x8F; // 10001111
uint8_t i = 0;
i |= (1 << 4); // set bit 4                          // i == 00010000
i |= mask; // set the highest and lowest four bits // i == 10011111
i &= ~(1 << 7); // clear the highest bit           // i == 00011111
i ^= (1 << 0); // toggle bit zero                  // i == 00011110
i ^= mask; // toggle mask bits                     // i == 10010001
if (i & (1 << n)) ... // test if the nth bit is set
```

1. Use the `curl` command to download from the class website everything you'll need for this exercise.

   ```
   $ WEBPATH=http://www.phy.olemiss.edu/~kbeach/
   $ curl $WEBPATH/courses/fall2017/phys750/src/exercise14.tgz -O
   $ tar xzf exercise14.tgz
   $ cd exercise14
   ```

   In `exercise14` directory you will find a file `rand.hpp` that contains templated class definitions for the generator `linear_congruential`. The template parameters fix the values of a and c. Instantiations of the class are *function objects* that produce integers in the range $\in \{0, 1, \ldots 2^{32} - 1\}$ when called.

   ```
   // a = 1664525, c = 1013904223
   // seeded with 18462
   linear_congruential<1664525,1013904223> F(18462);
   uint32 x = F();
   uint32 y[100]'
   generate(y,y+100,F);
   vector<uint32> z(100);
   generate(z.begin(),z.end(),F);
   ```

   The code above puts a random number into `x` and 100 random numbers into the C array `y` and the vector `z`. (`std::generate` is defined in the STL header `algorithm`. It assigns `F()` to each element of a list.)

   There is another class `rnd`, which serves as a wrapper for the generator. This class provides methods for generating numbers other than 32-bit integers.

```
rnd< linear_congruential<1664525,1013904223> > R(18462);
const double x = R(); // double from [0,1)
const int i = R.randInt(3); // integer from {0,1,2,3}
const int j = R.randInt(1,6); // integer from {1,2,3,4,5,6}
```

2. Class `linear_congruential` implements the recursion $X_{n+1} = aX_n + c \bmod 2^{32}$. Write a class `quadratic_congruential` for $X_{n+1} = aX_n^2 + bX_n + c \bmod 2^{32}$.

3. Write a class `lagged_fibonacci` $X_{n+1} = X_{n-j} + X_{n-k} \bmod 2^{32}$ $(j < k)$. This is a little tricky. At every step, the last $k + 1$ values have to be stored, and the generator must be seeded with at least $k + 1$ initial values. (Either use a `template` for general j and k, or fix the values $j = 25, k = 56$.)

   The lagged fibonacci scheme is sensitive to the initial values. They need to be quite random already. (What happens if all the initial values are zero? What happens if all initial seeds are even?) I suggest you use a single seed X_0 and generate $X_1, \ldots, X_k$ using a linear congruential generator.

4. Write a 32-bit generator `middle16_bits` in which each successive value is the square of the middle 16 bits of the previous value. For example, if $a_n = 0\text{x}2\text{A}8\text{B}3\text{D}65$, then a_{n+1} is computed as follows:

$$a_n = 00101010\,\underbrace{1000101100111101}_{\text{middle 16 bits}}\,01100101$$
$$t = \text{mid}_{16}(a_n) = 0000000000000000\,\underbrace{1000101100111101}_{\text{middle 16 bits}}$$
$$a_{n+1} = t^2 = 01001011101110110100110010001001$$

   Multiplication and the bit shift operators are all you need.

*5. Write a 32-bit generator `middle32_bits`. This is similar to the previous question, but the order of operations is reversed. Each a_n is first squared (into a 64-bit type). The middle 32 bits are then assigned to a_{n+1}.

*6. The class `rnd` provides a method `rand23` that returns a single-precision floating point number in the interval $[0, 1)$.

```
const float x = R.rand23();
```

   The method relies on the `convert32_t` type defined in `bitconvert.h`. (You can see how this works by looking over `bit_test.cpp`). `convert32_t` is a just a `union` on various floating point and integral types that have the same 32-bit width. A `union` is something like a `struct`, but it allows the same underlying bit pattern to be variously interpreted.

   The `float` has one sign bit, an 8-bit exponent, and a 23-bit mantissa:

$$(\pm)^s \times 2^{(e_1 \cdots e_8)_2 - 127} \times (1.m_1 m_2 \ldots m_{23})_2.$$

   `rand23` returns a `float` in the range $+2^0(1.000\cdots00)_2 - 1$ to $+2^0(1.111\cdots11)_2 - 1$. Since the generator produces 32-bit values, its output is bit shifted 9 places so that the 23 least significant bits remain filled. The most significant 9 bits are then set to $127 = 001111111_2$.

   The same trick can be employed for a `double`, which has the following bit layout:

$$(\pm)^s \times 2^{(e_1 \cdots e_{11})_2 - 1023} \times (1.m_1 m_2 \ldots m_{52})_2.$$

   Write the corresponding method `rand52`. Fill the `convert64_t` type with two 32-bit integers. Set the 12 high bits to $1023 = 01111111111_2$.

   Or, if you're feeling adventurous, fill the mantissa with the highest 26 bits from each of the `uint32_t`s.

**7. The method `randInt(n)` returns a number from $\{0, 1, \ldots, n\}$ by generating a real number $U \in [0, 1)$ and then calculating $\lfloor (n+1)U \rfloor$. ($\lfloor x \rfloor$ denotes the floor of x; that is, x rounded down to the nearest integer.) Rewrite the function so that the output $X \in \{0, 1, \ldots, 2^{32} - 1\}$ from the generator is bit-shifted to the right until its high bit is aligned with the most significant bit of n. Accept $X' \leftarrow X >> (31 - \log_2 n)$ if $X' \leq n$. Otherwise, generate another X and try again.

8. The program `walk1d.cpp` generates a sequence of random numbers

$$\underbrace{X_1, X_2}_{s_1}, \underbrace{X_3, X_4}_{s_2}, \underbrace{X_5, X_6}_{s_3}, \cdots,$$

which it reinterprets as a sequence of steps $s_i = \pm 1$. The positive or negative value is assigned based on the pair's relative ordering

$$s_i = \mathrm{sgn}(X_{2i-1} - X_{2i}) = \begin{cases} +1 & \text{if } X_{2i-1} > X_{2i} \\ -1 & \text{if } X_{2i-1} < X_{2i} \end{cases}$$

The sequence (s_i) defines a random walker in one dimension. After N steps, the walker is at position

$$R_N = \sum_{i=1}^{N} s_i.$$

Its distance from the origin is characterized by

$$R_N^2 = \sum_{i=1}^{N} \sum_{j=1}^{N} s_i s_j = \sum_{i=1}^{N} s_i^2 + \sum_{i \neq j} s_i s_j = N + \sum_{i \neq j} s_i s_j.$$

For purely random steps, the averages over many measurements should look like $\langle R_N \rangle = 0$ and $\langle R_N^2 \rangle = N$.

```
$ make walk1d
$ ./walk1d > walk1d.dat
$ gnuplot
gnuplot> plot "walk1d.dat" using 1:3 with lines, x
```

Write a new program `walk3d.cpp` that assigned steps $\boldsymbol{s} = (\pm 1, 0, 0)$, $(0, \pm 1, 0)$, or $(0, 0, \pm 1)$ in three-dimensional space according to the $3! = 6$ possible orderings of the triplets

$$\underbrace{X_1, X_2, X_3}_{\boldsymbol{s}_1}, \underbrace{X_4, X_5, X_6}_{\boldsymbol{s}_2}, \underbrace{X_7, X_8, X_9}_{\boldsymbol{s}_3}, \cdots$$

Again, $\langle \boldsymbol{R}_N \rangle = \boldsymbol{0}$ and $\langle \boldsymbol{R}_N \cdot \boldsymbol{R}_N \rangle = N$.

9. Write a program `moments.cpp` that computes the first several moments of a large random sequence. Compare the results for various generators to the exact values,

$$\int_0^1 dx\, x^n = \frac{1}{n+1}.$$

10. Write a program `dice.cpp` that simulates many ($> 100\,000$) rolls of a standard six-sided die. Keep a frequency histogram of the number of identical rolls in a row. The probability of any given face coming up is $p = 1/6$. The probability of rolling n identical numbers in a row is $p^{n-1}(1-p)$.

```
$ ./dice.cpp > dice.dat
$ gnuplot
gnuplot> p = 1.0/6.0;
gnuplot> plot "dice.dat", (1-p)*p**(x-1)
gnuplot> set logscale y
gnuplot> replot
```

11. Here is an algorithm due to MacLaren and Marsaglia that takes two random sequences and produces a single "more random" sequence.

    M1. Set X, Y equal to the next numbers if the sequences X_n, Y_n, respectively.

    M2. Set $j \leftarrow \lfloor kY/m \rfloor$, where m is the modulus used in the sequence Y_n; that is, j is a random integer value, $0 \le j < k$, determined by Y.

    M3. Output $V[j]$ and then set $V[j] \leftarrow X$.

    Implement this algorithm and construct a composite $Z = X \star Y$ from from two (rather bad) linear congruential generators $X_0 = 0$, $X_n = (5X_n + 3) \bmod 8$ and $Y_0 = 0$, $Y_n = (5X_n + 1) \bmod 8$. Compare (Z_n) to (X_n) and (Y_n).

    Note that mod 8 sequences can be called using an optional third template parameter:

    ```
    linear_congruential<5,3,8> X(0);
    linear_congruential<5,1,8> Y(0);
    ```

    Try plotting the coordinate pairs $(A_0, A_1), (A_2, A_3), (A_4, A_5), \ldots$ in `gnuplot` for each of $A = X, Y, Z$. Do the random numbers fall mainly in the plane?