Centrum Wiskunde & Informatica

Advanced RNGs

WISM454 Laboratory Class Scientific Computing, Jan-Willem Buurlage

February 15, 2019

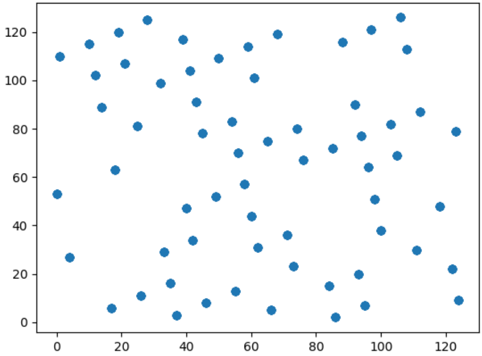
RNG

What makes a good RNG?

- Theoretical
  - Period
  - Uniformity
- Empirical
  - Tasks using RNGs, check if output as expected
  - Predictability
  - Invertibility (security!)
- Computational efficiency
- Space and time complexity
- Application specific requirements

Testing your RNG library

- Basic theory can tell you about e.g. the period, uniformity of your RNG without running it. Also: $d-$ dimensional equidistribution.
- E.g., if LCRNG is used to generate points in d dimensional space, confined to maximum of $\sqrt[d]{d!m}$ parallel hyperplanes.
  $(a, c, m) = (57, 53, 2^7 - 1)$, $\rightarrow$ 16 lines



Software packages ('battery of tests')

- There are software packages such as Diehard (1999) or TestU01 (2007)[1] which contain a collection of statistical tests for your RNGs.
- TestU01 is a C library, can be made part of your code!
- E.g. birthday paradox, rank of random binary matrices, play a game of craps, randomly place spheres in a box, . . . all follow a known distribution.

[1] http://simul.iro.umontreal.ca/testu01/tu01.html

Advanced RNGs

Overview of topics today:

1. Xorshift
2. Linear-feedback shift register
3. Mersenne twister

Xorshift

```
uint32_t xorshift(uint32_t x) {
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    return x;
}
```

Xorshift^[George Marsaglia (2003)]

- Consider $M = \{0, \ldots, m-1\}$, assume that our RNG is still of the form (for $x_i \in M$):
$$x_{i+1} = f(x_i),$$
- Generally, we want f to be be one-to-one.
- Usually not enough, since $f = \text{id}$ is one-to-one but certainly not random.
- Additionally require full period! (Although e.g. ‘f $=(+1)$‘ then still works)

Linear transitions

- Represent $x_i \in M$ as binary vector:

$$\mathbf{x}^{(i)} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{pmatrix}$$

  take f to be a linear function, i.e. given by transition matrix T:

$$\mathbf{x}^{(i+1)} = T\mathbf{x}^{(i)}.$$

- T represents one-to-one function iff invertible
- Limit to non-null vectors, i.e. full period consists of all $m-1$ non-zero integers in M.

Exercise

- Exercise: Prove that a non-singular T generates a non-zero sequence of full period for all non-zero seeds, if and only if the order of T is $2^n - 1$ (in group of non-singular $n \times n$ matrices)

Intermezzo: bitwise operations

- Bitwise operations are very efficient.

```
uint8_t x = 0b00010010; // 18
uint8_t y = 0b01010010; // 82
```

- Shifts:

```
x >> 2 // ~> 0b00000100
x << 2 // ~> 0b01001000
```

- Binary bitwise operations

```
x ^ y // XOR ~> 0b01000000
x | y // OR  ~> 0b01010010
x & y // AND ~> 0b00010010
```

- We have that `1 « k` is equal to 2^k, and that XOR is addition modulo two (i.e. addition in our vector space $(F_2)^n$)

How to choose T

- Left shift L (i.e. $Lx \equiv x \ll 1$), right shift R (i.e. $Rx \equiv x \gg 1$):

$$L = \begin{pmatrix} 0 & \dots & \dots & \dots & 0 \\ 1 & \ddots & \ddots & \ddots & \vdots \\ 0 & 1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & 1 & 0 \end{pmatrix}, R = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ \vdots & \ddots & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \dots & \dots & \dots & 0 \end{pmatrix}$$

- Note that $L^a x$, $R^a x$ equal to `x << a` and `x >> a` respectively.
- Clearly L and R singular.

The xorshift transition matrix

- But: $\mathrm{Id} + L^a$ and $\mathrm{Id} + R^a$ non-singular! However, sadly

$$T = (\mathrm{Id} + L^a)(\mathrm{Id} + R^b) \tag{1}$$

does not have the right order for any a, b.

- Instead choose

$$T = (\mathrm{Id} + L^a)(\mathrm{Id} + R^b)(\mathrm{Id} + L^c) \tag{2}$$

- Exercise: Verify experimentally that for (1) no a, b give T with required period for $n = 32$
- Exercise: Give all triples (a, b, c) for which (2) has full period

Standard `xorshift`

- The standard Xorshift RNG engine has:

$$T = (Id + L^5)(\mathrm{Id} + R^{17})(\mathrm{Id} + L^{13}).$$

```
uint32_t xorshift(uint32_t x) {
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    return x;
}
```

- There are versions with bigger state and/or more elaborate transition functions that outperform this basic version.

Linear-feedback shift register

- Xorshift is an example of a linear-feedback shift register

Let $a \in \{0, 1\}$

Definition

A (binary) sequence generated by a *shift register* is one satisfying an n-term recursion:

$$a^{(i+n)} = f\big(a^{(i)}, \ldots, a^{(i+n-1)}\big).$$

If f is linear, we speak of a linear-feedback shift register.

Mersenne twister

- Another example of LFSR
- period of $2^{19937} - 1$.
- slower, but k-equidistributed
- very popular

```cpp
#include <random>

auto rng = std::mt19937(seed);
std::cout << rng() << "\n";

// PS:
class mt19937 {
    uint32_t operator()() { ... }
};
```

mt19937

- From Wikipedia:

  The Mersenne Twister is the default PRNG for the following software systems: Microsoft Excel,[3] GAUSS,[4] GLib,[5] GNU Multiple Precision Arithmetic Library,[6] GNU Octave,[7] GNU Scientific Library,[8] gretl,[9] IDL,[10] Julia,[11] CMU Common Lisp,[12] Embeddable Common Lisp,[13] Steel Bank Common Lisp,[14] Maple,[15] MATLAB,[16] Free Pascal,[17] PHP,[18] Python,[19][20] R,[21] Ruby,[22] SageMath,[23] Scilab,[24] Stata.[25]

Expectations for your RNG library

- Required
  - Usable for randomized algorithms
  - LCRNG
  - Distributions: uniform int, uniform double, Gaussian
- Optional but expected
  - Xorshift
  - Tested with TestU01 / Diehard
  - Benchmarks (random numbers / second)
- Extra credits
  - Mersenne twister
  - Full test-suite based on e.g. TestU01
  - Personal statistical test
  - Other (personal?) engines
  - Extra distributions

C++

Polymorphism (I)

```
class rng {
  public:
    virtual int next() = 0;
};

class lcrng : public rng {
    ...
    int next() override {
        return ...;
    }
    ...
};
```

- Here, `rng` is the base class and `lcrng` is the derived class that inherits from the base class.

Polymorphism (II)

- An abstract class is a class with at least one pure virtual function, like `rng` has:

  ```
  virtual int next() = 0;
  ```

  A non-abstract class is also called a concrete class
- Objects of an abstract type can not be manipulated by-value, because the representation of an `rng` is unknown. They have to be manipulated using references or pointers:

  ```
  void monte_carlo(rng& engine, ...) { ... }
  ```
- We call `monte_carlo` with a concrete RNG engine. At runtime, the function will call the `next` implementation of this concrete class_ (e.g. `lcrng`).

  ```
  auto r = lcrng{14239, 5205, (1 << 30) - 1};
  monte_carlo(r, ...)
  ```

Polymorphism (III)

- Abstract classes allow us to leave the choice of e.g. RNG engine to the user, and write our code independently of concrete realizations.

```
class uniform_real_distribution {
  public:
    uniform_real_distribution(rng& engine)
        : engine_(engine) {}

    float sample() {
        return (float)engine.next() /
            (engine.max() - 1);
    }

  private:
    rng& engine_;
}
```

Polymorphism (IV)

```
class abstract {
  public:
    virtual void f() = 0;
};

class concrete : public abstract {
  public:
    void f() override {}
};

abstract a; // ERROR
concrete b; // fine

void f(abstract& a) {
    a.f(); // fine
}
```

Polymorphism (V)

```
class abstract {
  public:
    virtual void f() = 0;

  protected:
    virtual void g() = 0;
    int x;

  private:
    void h() {}
    int y;
};
```

Polymorphism (VI)

- Access specifier, e.g. `public`:

```
class derived : public base ...
```

- `private` members of `base` are never visible to `derived`.
- access specifier specifies maximum visibility of inherited members

- E.g. `class derived : protected base` would make `public` and `protected` members of `base`, `protected` members of `derived`.
- For purposes other than inheriting, `protected` is like `private`.

RAII

```
class object {
  public:
    object() { std::cout << "Constructor\n"; }
    ~object() { std::cout << "Deconstructor\n"; }
};

void f() {
    object o;
}

f();

auto o = new object;
delete o;
```

Heap storage

- User defined types with heap storage

```
class rng_with_big_state : public rng {
  public:
    rng_with_big_state() {
        state_  = new State;
    }
    ~rng_with_big_state() {
        if (state_) { delete state_; }
    }
  private:
    State* state_;
};
```

- Hidden `new` and `delete`, safer user code:

```
void monte_carlo() {
    auto r = rng_with_big_state(); // or:
    rng_with_big_state r; // same thing
}
```

Polymorphism and RAII

- Derived classes inherit from base classes.
- Abstract classes versus concrete classes.
- Access specifiers
- RAII allows automatic resource management based on scopes
- These are very important concepts, crucial to understanding how to develop quality C++ software. Spend some time familiarizing yourself with these concepts!
- Any questions/comments on polymorphism and RAII?

Templates

- Up to now we have discussed runtime polymorphism (also virtual dispatch). Templates are 'compile time polymorphism' (static dispatch).

  ```
  struct distribution_u32 {
      uint32_t sample(rng& engine);
  };
  struct distribution_i32 {
      int32_t sample(rng& engine);
  };
  struct distribution_u64 {
      uint64_t sample(rng& engine);
  };
  struct distribution_f32 {
      float sample(rng& engine);
  };
  ```

- ... there must be an easier way

Enter templates!

```
template <typename T>
struct distribution {
    virtual T sample(rng& engine) = 0;
};

struct normal_distribution : distribution<float> {
    float sample(rng& engine) override {
        return ...;
    }
};
```

Fancy tricks!

```
template <typename T,
typename std::enable_if_t<std::is_floating_point_v<T>>>
struct normal_distribution : distribution<T> {
    normal_distribution(T mean, T stddev) { ... }

    T sample(rng& engine) override {
        return ...;
    }
};
```

Other examples

- Functions can be templates too, compile time values also allowed as template arguments.

```cpp
template <int D, typename T>
std::vector<std::array<T, D>> generate_points(
    int count, distribution<T>& f) {
    return ...;
}
```

- Many STL types are templates. We will revisit templates when we discuss the standard library next week!

Tutorial

Common things that are unclear

- ODR
- Calling base constructor
- Virtual functions / vtables

Concrete things to do

- Implement LCRNG, Xorshift engines
- Implement distributions:
  - Uniform
  - Gaussian (with rejection)
  - Something with inversion
- Write function to randomly permute an array
- Statistically test your generators