Centrum Wiskunde & Informatica

Random number generation II

WISM454 Laboratory Class Scientific Computing, Jan-Willem Buurlage

February 12, 2019

Organization

Recap

- We create a reusable software library for random number generation
- Later in the course, we build software on top of this. We will implement libraries first for Monte Carlo methods, and second for Genetic Algorithms
- Finally, we apply this generic software to some large-scale applications.
- The reports are supposed to document this entire process!

RNG

General distributions

- So far, we have focused on obtaining uniformly distributed numbers in some set, e.g.

$$M = \{0, 1, \ldots, m-1\},$$
$$I = [0, 1]$$

- Often, we want to draw numbers according to some other distribution function, e.g. Gaussian
- There are two methods for this, that are independent on the engine used:
  - Inversion method
  - Rejection method

Random variables

- Let Ω be some sample space, P a probability measure on Ω.
- Often, we will set $\Omega = [0, 1]$, and P to be uniform probability measure, i.e.:

$$P([a, b]) = b - a.$$

- Random variable: a function:

$$X : \Omega \to \mathbb{R}.$$

  This random variables 'realizes the desired distribution'.
- *Note:* When considering uniform distribution, we still consider a random variable X equal to the identity function (or translate-and-scale).
- Q: Say you want e.g. Gaussian distribution, what to choose for X?

Discrete case

- Assume $X(\Omega)$ countable, let $x \in X(\Omega)$.
- Discrete probability density function (distribution function, pdf):

$$f(x) \equiv P(X = x)$$

- Cumulative distribution function (cdf):

$$F(x) \equiv P(X \leq x) = \sum_{t \leq x} f(t)$$

Continuous case

- Let $X(\Omega) = \mathbb{R}$, and let $x \in \mathbb{R}$.
- Probability density function (pdf) or simply distribution function of X, is the $f : \mathbb{R} \to \mathbb{R}$ satisfying:
  1. $f(x) \geq 0$
  2. $\int_{-\infty}^{\infty} f(x)dx = 1$
  3. $\int_a^b f(x)dx = P(a \leq X \leq b)$
- Cumulative distribution function (cdf):

$$F(x) = P(X \leq x) = \int_{-\infty}^{x} f(y)dy.$$

Recap

- Concepts:
  - Random variable X
    - describes 'outcome of experiment' involving a random process
  - Distribution function $f(x)$:
    - the probability density of observed values of X at some point x
  - Cumulative distribution function $F(x)$:
    - probability of observing at most x
- Why is this relevant to our RNG library?
  - Our RNGs can generate uniform numbers in $[0, 1]$.
  - We want to generate random numbers according to a specific distribution.
  - For many distributions (Gaussian, Poisson, Binomial, ...) we know their pdf f.
  - How should we choose X to transform our generated numbers?

Example: uniform distribution on $[a, b]$

- As a simple example: transform uniform distribution $[0, 1]$ to a uniform distribution in $[a, b]$.
- Choose:

$$X(u) = a + (b - a)u, \qquad u \in [0, 1]$$

- Then:

$$f(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{if } x > b \end{cases} .$$

$$F(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{x-a}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{if } x > b \end{cases} .$$

The inversion method (idea)

- Q: Given f, find X.
- Observe that

$$u = \frac{x-a}{b-a}$$

$$u(b-a) + a = x$$

- This is a general scheme: compute and invert the cumulative distribution

The inversion method

Theorem

Let $f : \mathbb{R} \to [0,1]$ *be a distribution function with cdf* $F : \mathbb{R} \to [0,1]$*.*
Then $X \equiv F^{-1} : [0,1] \to \mathbb{R}$ *has distribution function* f*.*

- Here, the inverse → generalized inverse.

$$F^{-1}(u) = \inf\{x \in \mathbb{R} \mid F(x) \geq u\}.$$

- Needed because F is not *strictly* monotonically increasing.

Exercises

- Ex 2.7 Prove this theorem
- Ex 2.8 Apply to two relevant cases
- These are hand-in exercises, hand-in a LaTeX-ed solution in two weeks.

Limitations

- Sometimes F or F^{-1} not analytically computable or even expressable (e.g. Gaussian distribution!).
- Instead, we can fall back to numerical methods.

The rejection method

- Idea: Combine two RNGs to get desired distribution
- Let f be the desired distribution, and q some 'realizable' distribution, such that for some fixed $c \in \mathbb{R}$:

$$\forall x \in \mathbb{R} \; f(x) \leq cq(x).$$

- Intuitively: if we obtain a sample y according to the distribution q, then the relative probability of obtaining the same sample according to f would be:

$$r \equiv \frac{f(y)}{cq(y)}.$$

- Sample according q, then accept with probability $r \in [0, 1]$, i.e. obtain $u \in [0, 1]$ uniformly at random and compare with r.

The rejection method algorithm

```
auto u = lcsc::rng::uniform<double>(0.0, 1.0);
auto v = ...; // random variable with distribution q
while (true) {
    auto x = v(u.next());
    auto y = u.next();
    if (y <= (f(x) / (c * q(x)))) {
        return x;
    }
}
```

Exercises

- Ex 2.9 (After implementing RNGs). Implement rejection method.
- Ex 2.10 Use RNG to generate random permutations.

High-level overview of RNG library

Engine samples $m \in M$

$\downarrow$

Sample for **uniform distribution** $\mu \in [a, b]$

$\downarrow$

Sample for **general distribution** f

- Arrows are independent of specific methods!

C++

Second tour of C++

- We continue with our overview of the C++ language
- I don't expect you to become a fluent C++ programmer by only looking at these slides!
- Consider the C++ lectures as a summary of topics that you are supposed to familiarize yourself during this course
- Learning C++ is best done by consulting references (and writing a lot of code)!
  - *Bjarne Stroustrop* - The C++ Programming Language
  - *Scott Meyers* - Effective Modern C++
  - `https://en.cppreference.com`
  - ...

Arrays and pointers (I)

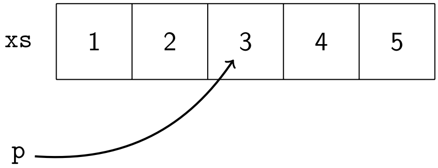
- Recap:

```
int xs[5] = {1, 2, 3, 4, 5};  // array of integers
int x = xs[3]; // x is now 4
int* p = &xs[3]; // pointer to 3rd element of xs
std::cout << p << "\n"; // ~> "0x1a2b3c4d"
std::cout << *p << "\n"; // ~> "4"!

++p; // increase pointer (not value)
std::cout << *p << "\n"; // ~> "5"!
```

- Operator &: address of
- Operator *: contents of (also called dereferencing).

Arrays and pointers (II)



- `xs` is the array containing data (stored as a pointer to the first element)
- `p` is free to point to any element, i.e. `p = xs` would make `p` point to the first element!
- Dealing with pointers is tricky business! In modern C++, they are avoided wherever possible.

References

```cpp
int x = 3;
int& y = x;
y = 4;
std::cout << x << "\n"; // ~> "4"

void f(int x, int& z) {
    z = x * x;
}

int z = 0;
f(x, z);
std::cout << z << "\n"; // ~> "9"
```

- A reference is like a pointer, but need not be dereferenced explicitely.
- It is like an alias, can only be initialized once
- Prefered over pointers

const-ness

```cpp
const int x = 3;
x = 5; // ERROR

int f(const int& x) {
    return x * x;
}
```

- A promise not to change a value, but merely use it
- In this case, we don't copy the `int`, but share a reference to it
- Important part of an interface!

Stack vs Heap memory

- A big motivation for using pointers is the difference between stack and heap memory.

```
int x = 3; // integer allocated on the stack

int* x = new int; // integer allocated on the heap
*x = 3;
delete x;
```

- Default: put on the stack. Limited space available. Automatically 'deallocates' when out of scope.
- Heap slower, but bigger and more flexible (dynamic deallocation)

Dangling pointers and references

- *Dangling pointers*: pointer to objects that are not valid.

```cpp
// references to objects that have gone out of scope
int& f() {
  int c = 3;
  return c;
}

auto& x = f();

// 'use after delete'
auto i = new int;
*i = 3;
auto j = i;
delete i;
std::cout << *j << "\n";
```

Classes and other user-defined types

- So far, we have only looked at built-in types
- New types can be created in two main ways, by classes and enumerations.
- Writing software for C++ is mostly about defining your own types, and operations on them! Simplest example is the `struct` from C

```
struct lcrng {
    int a;
    int c;
    int m;
};
```

- A LCRNG is defined by the three numbers a, c, m, it makes sense to define a type that groups them together.

Classes (II)

```
lcrng r;
r.a = 14239;
r.c = 5205;
r.m = (1 << 30) - 1;

int next(lcrng r, int x) {
    return (r.a * x + r.c) % r.m;
}

auto seed = 12345;
auto x1 = next(r, seed);
```

Classes (III)

```
// alternative initialization
lcrng r = {14239, 5205, (1 << 30) - 1};
auto r = lcrng{14239, 5205, (1 << 30) - 1};

// *member* function
struct lcrng {
    int next(int x_prev) {
        return ...;
    }

    int a;
    int c;
    int m;
};
```

Classes (IV)

```
class lcrng {
  public:
    lcrng(int a, int c, int m, int seed) :
        a_(a), c_(c), m_(m), x_(seed) {}

    int next() {
        return ...;
    }

  private:
    int a_;
    int c_;
    int m_;
    int x_;
};
```

Classes (V)

```
auto park_miller = lcrng(16807, 0, (1 << 31) - 1), seed);
for (int i = 0; i < samples; ++i) {
    std::cout << park_miller.next() << "\n";
}
```

- Note that when we are given `park_miller`, we can generate random numbers simply by calling `next` on it.

Polymorphism

- Another important application of references/pointers

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

- An RNG is anything implementing `next`..

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

Polymorphism (II)

- In many cases (e.g. obtaining real uniform distribution from integer one), we do not care about the specific engine!

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

    float sample() { ... }
}
```

- This is an example of polymorphism!

Namespaces

- Group your functions and types together under a single 'namespace', e.g.:

```
namespace lcsc {

class rng { ... };
void plot_histogram(const rng& engine) { ... }

} // namespace lcsc

lcsc::plot_histogram(lcsc::lcrng(...));
```

Summary

- Today we covered
  - arrays, pointers and references
  - `const` and its applications
  - stack versus heap memory
  - classes and structs
  - polymorphism
  - namespaces
- Over the next couple of weeks, we will apply all of these concepts to write a RNG library that we will use throughout the course.
- Example structure and interface available on the course website
- Strongly suggest to design your own interface!

Tutorial

This week

- Tutorial:
  - Demo of compiling with multiple files, see GitHub course page
  - Set up RNG structure with classes
  - Implement number of LCRNGs in this new system
  - Ex 2.9 (rejection method)
  - Ex 2.10 (random permutations)
- At home (hand-in exercises):
  - Ex 2.7 Prove *inversion of cdf* theorem
  - Ex 2.8 Apply to two relevant cases