Contents

1 Exercises

1.1 Week 1

1.1.1 Setting up your programming environment

Follow the instructions in `code/week1` of the GitHub repository to set up a C++ development environment.

1.1.2 [LN 2.2] *Uniform distribution on the unit interval*

An RNG engine produces integers x_i in the set $\{0, 1, 2, \ldots, m-1\}$. To obtain reals ω_i that are distributed uniformly in $[0, 1]$ we can scale a number of different ways, such as:

- $\omega_i = \frac{x_i}{m-1}$
- $\omega_i = \frac{x_i}{m}$
- $\omega_i = \frac{x_i+1/2}{m}$

Discuss the advantages and disadvantages of these different scalings. Which one would prefer?

1.1.3 [LN 2.3] *Periods of poorly chosen iterations*

1. Compute the period of $(7x + 3)$ mod 10.
2. Compute the period of $5x$ mod 13 and of $7x$ mod 13.
3. Compute the period of $(3x + 4)$ mod 60. Note that, even for large m, the period can become small.

1.1.4 [LN 2.6]

1. Design an abstract class `rng`, that will serve as a base class for all RNG engines that you implement. What (pure virtual) functions should it have?
2. Design a class `lcrng` that inherits from `rng`. This class should allow a user to choose the parameters (a, c, m) and the seed for the RNG.
3. Predefine a number of LCRNGs (i.e., with specific parameters) in your library. One way to accomplish this is by making a class such as `park_miller` that inherits from `lcrng`, but has a simpler constructor that only takes the seed. You should at least predefine the LCRNGs that are listed in Table 2.1 of the lecture notes.
4. Implement Schrage's trick for the generators that it applies to. One straightforward way to do this (but definitely not the only correct way), is to have a `shrage_lcrng` class that inherits from `lcrng` with a different implementation for computing the next iterate (by overloading the member function that computes this).

1.2 Week 2

1.2.1 Building a library

Follow the instructions in `code/week2` of the GitHub repository to learn about dealing with multiple files.

1.2.2 [LN 2.9] *Implementation of the rejection method*

1. Design an abstract class `distribution`, that will serve as a base class for all the distributions that we implement. What (pure virtual) functions should it have?

2. Implement a uniform distribution class `uniform_distribution`, that is able to sample random reals in an interval $[a, b]$.

3. Design an interface that allows you to test your distribution. A quick (but inexact) way to do this is to observe the distribution optically, by e.g. looking at a histogram of the generated samples. You are free to choose how you want to realize this. For example, you can implement a C++ function that shows a histogram in your terminal, or even output the samples to a file and use your favorite environment (Python, MATLAB, Mathematica) to generate the plots.

4. Choose your favorite distribution f. Using the uniform distribution that you implemented before, realize f by implementing the rejection method. Can you design it in such a way that you only need to implement the rejection method once, and reuse this implementations for other choices of f?

1.2.3 (Hand-in) [LN 2.7] *Correctness of the inversion method*

Prove the following theorem:

> **Theorem 2.1 (Inversion method)**. Let f be a distribution function with cumulative distribution F. Let U be a random variable on Ω with uniform distribution $[0, 1]$. Then the random variable $X \equiv F^1(U)$ on Ω has distribution function f.

You may take $\Omega = [0, 1]$, and $U(\omega) = \omega$. If F is continuous, then we have that $F(x) = \int_{-\infty}^{x} f(y)dy$.

1.2.4 (Hand-in) [LN 2.8] *Inverse distribution functions*

Let $u \in [0, 1]$ and $\lambda, \sigma > 0$.

1. Verify that the cdf for an *exponential distribution*:

   $$F(x) = 1 - e^{-\lambda x}$$

   has inverse $F^{-1}(u) = -1/\lambda \log(1 - u)$.

2. Verify that the cdf for a *Cauchy distribution*:

   $$F(x) = 1/2 - 1/\pi \arctan(x/\sigma)$$

   has inverse $F^{-1}(u) = \sigma \tan(\pi(u - 1/2))$.

1.2.5 *Implementation of the inversion method*

1. Implement a Cauchy distribution `cauchy_distribution` by using the inverse of its cdf that you computed in the hand-in.

2. Show that:

   $$\frac{1}{2\pi} \exp(-x^2/2) \leq \frac{1}{\pi(2 + x^2)}.$$

3. Implement a Gaussian distribution `normal_distribution` in two ways: by using the rejection method together with the Cauchy distribution that you implemented, and by using the rejection method together with the uniform distribution. You will need to cut the distributions off to make sure they have finite support.

4. Generate a large amount ($\geq 10^6$) of normally distributed numbers in both implementations, while timing the execution speed. You can do a rough timing of a program in your terminal by running e.g.:

   ```
   time ./test_normal_uniform
   time ./test_normal_cauchy
   ```

   How does the underlying distribution impact the runtime of your program? What would you roughly expect by analyzing the two auxiliary distributions?

1.2.6 [LN 2.10] *Random permutations of n points*

Implement a function:

```
std::vector<int> random_permutation(rng& engine, int n);
```

that generates a random permutation of the set $\{1, \ldots, n\}$.

1. Draw n numbers in the unit interval $[0, 1]$, using your `uniform_distribution` class. This gives you a sequence $(\omega_1, \omega_2, \ldots, \omega_n)$.
2. Define the permutation π as the permutation that sorts this sequence. You can find this permutation as follows:
   (a) Initialize the return list with $1, \ldots, n$ (hint: look up how to use `std::vector` and `std::iota`).
   (b) Sort this list using `std::sort`, but use a custom compare function, that sorts it according to the sequence of uniform samples that you generated.
3. Prove (on paper) for $n = 2$ and $n = 3$ that the generated permutation is uniformly distributed.

1.3 Week 3

1.3.1 Understanding header-only libraries

Look at the example code in `code/week3` of the GitHub repository. Make sure you understand the basics of templates, the ODR, and translation units. You can use this code as a starting point, or simply as inspiration for your own RNG library.

You are free to choose whether you want to employ templates for your engines and distributions, or to use fixed width numeric types. In any case, think about (and discuss in your report) some of the advantages and disadvantages of using templates.

From now on, maintain a `CMakeLists.txt` file in your code that allows it to be easily built by other programmers.

1.3.2 Full period for a linear generator T

Prove that a non-singular matrix 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).

1.3.3 Finding appropriate Xorshift parameters

Consider the linear generator functions:

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

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

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

Hint: Use the previous exercise.

1.3.4 Implementing Xorshift

1. Implement a class `xorshift` that inherits from `rng` and implements a Xorshift generator for a set of parameters defined by the user.
2. Predefine a number of Xorshift engines.

1.3.5 (Optional) Implementing MT19337

Research online how the Mersenne Twister is defined. Implement a class `mt19937` that implements it. You can compare with the `mt19337` implementation from the `<random>` library to ensure correctness.

1.4 Week 4

1.4.1 Using external libraries

Look at the example code in `code/week4` of the GitHub repository. Download and install the TestU01 library, and test your RNGs against some of its tests.

1.5 Week 6

1.5.1 An extension to our RNG library

Look at the example code in `code/week6` of the GitHub repository. Implement basic Monte Carlo integration (hit-or-miss and simple sampling) of a black box 1D function represented as an `std::function` object. Your Monte Carlo code should work for arbitrary functions and with any of the RNG engines you have implemented.

1.5.2 [LN 3.1] The error of the trapezoidal rule.

Show that the remainder:

$$R \equiv \int_a^b f(x)dx - \frac{b-a}{2}(f(a)+f(b))$$

can be expressed as

$$R \equiv \frac{(b-a)^3}{12} f''(\eta)$$

for some $\eta \in [a,b]$.

1.5.3 [LN 3.2] The error of the repeated trapezoidal rule

1. Show that the overall error for the repeated trapezoidal rule can be expressed as:

$$R(h) = -\frac{1}{12}h^3 \sum_{i=1}^{k} f''(\eta_i),$$

   with $\eta_i \in [x_{i-1}, x_i]$.

2. Show that this error is bounded by

$$|R(h)| \leq \frac{b-a}{12}h^2 \max_{x\in[a,b]} |f''(x)|.$$

1.5.4 [LN 3.4] Integrating a function with Monte Carlo.

Consider the integral:

$$I = \int_0^1 \sqrt{1-x^2}\, dx.$$

1. Compute this integral using *hit-or-miss* Monte Carlo.
2. Compute this integral using *simple sampling* Monte Carlo.
3. Investigate the behaviour of the remainder as a function of the number of 'shots' and 'sample points'.