Lecture 4: Some limitations of R, and intro to C++

Ciaran Evans

Previously: Linear congruential generator

$$x_{n+1} = (ax_n + c) \bmod m$$

While a classic method, R does not use an LCG to generate its random numbers:

```
?.Random.seed
```

```
Details

The currently available RNG kinds are given below...
The default is "Mersenne-Twister".
```

What is the Mersenne Twister?

We won't go through all the details right now, but here is the core of the algorithm: we generate a sequence of integers by

$$x_k = x_{k-(n-m)} \oplus ((x^u_{k-n} | x^l_{k-(n-1)})A)$$

$$xA = \begin{cases} x \gg 1 & x_{[0]} = 0 \quad (x \text{ is even}) \\ (x \gg 1) \oplus a & x_{[0]} = 1 \quad (x \text{ is odd}) \end{cases}$$

Here, $\oplus$, $\gg$, and $|$ all represent **bitwise** operations on x

Example: bitwise shift

Consider a 4-bit integer:

$$3 = 0011$$

$$3 \gg 1 = 0001 = 1$$

$$3 \gg 2 = 0000 = 0$$

$$3 \ll 1 = 0110 = 6$$

An example in R

$$3 \gg 1 = 1$$

```
bitwShiftR(3, 1)
```

```
## [1] 1
```

$$3 \ll 1 = 6$$

```
bitwShiftL(3, 1)
```

```
## [1] 6
```

Another example in R

```
bitwShiftL(2147483647, 1)
```

```
## [1] -2
```

Wait. . . a left shift of 1 should *double* our number, right? How did we get -2??

Integers in R

Integers in R are *signed*, *32-bit* integers:

$$0 = 00000000000000000000000000000000$$

$$1 = 00000000000000000000000000000001$$

$$2147483647 = 2^{31} - 1 = 01111111111111111111111111111111$$

```
.Machine$integer.max
```

```
## [1] 2147483647
```

32-bit signed integers

$$2147483647 = 2^{31} - 1 = 01111111111111111111111111111111$$

$$? = 10000000000000000000000000000000$$

32-bit signed integers

$$2147483647 = 2^{31} - 1 = 01111111111111111111111111111111$$

$$-2147483648 = 10000000000000000000000000000000$$

$$-2147483647 = 10000000000000000000000000000001$$

$$-2147483646 = 10000000000000000000000000000010$$

etc.

$$-2 = 11111111111111111111111111111110$$

$$-1 = 11111111111111111111111111111111$$

Integers in R

$$2147483647 = 01111111111111111111111111111111$$

$$2147483647 \gg 1 = 11111111111111111111111111111110 = -2$$

What behavior do we *want*?

Random number generators produce numbers between 0 and 1 $\Longrightarrow$ we don't want any negative numbers. Instead, we want:

$$2147483647 = 01111111111111111111111111111111$$

$$2147483648 = 10000000000000000000000000000000$$

...

$$4294967295 = 11111111111111111111111111111111$$

That is, we want 32-bit **unsigned** integers (values between 0 and $2^{32} - 1$)

Types in R

6 basic scalar types in R:

- logical (`TRUE` or `FALSE`)
- double (for decimal numbers)
- integer (32-bit, signed)
- character (i.e., strings)
- complex (for complex numbers)
- raw (specifically for working with binary data)

Problem: No option for *unsigned* integers!

LCG in C++

Here is an implementation of the LCG in C++

```
arma::vec my_lcgC(int n, uint32_t x0,
                  uint64_t m = 4294967296,
                  uint32_t a = 1664525,
                  uint32_t c = 1013904223){
  arma::uvec x(n);

  x[0] = x0;
  for(int i = 1; i < n; i++){
    x[i] = (a*x[i-1] + c) % m;
  }

  arma::vec u = arma::conv_to<arma::vec>::from(x);
  return u/m;
}
```

How does this compare to R code?

Some data types in C++

- `int`: signed 32-bit integers
- `uint32_t`: unsigned 32-bit integers
- `uint64_t`: unsigned 64-bit integers
- `bool`: boolean (`true` or `false`)
- `double`: double-precision floating point number (for decimals)
- `arma::vec`: vector (of doubles) from Armadillo library
- `arma::uvec`: vector (of unsigned integers) from Armadillo library

Another example

```
double sumC(arma::vec x) {
  int n = x.n_elem;
  double total = 0;
  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total;
}
```

What is this code doing?

Comparing R and C++ speed

```
Rcpp::cppFunction('double sumC(arma::vec x) {
  int n = x.n_elem;
  double total = 0;
  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total;
}', depends = "RcppArmadillo")

x <- rnorm(10000)
bench::mark(
  sum(x),
  sumC(x)
)
```

```
## # A tibble: 2 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 sum(x)       15.4us   16.6us    60291.        0B        0
## 2 sumC(x)      11.9us   12.1us    82019.        0B        0
```

Comparing R and C++ speed

```
bench::mark(
  my_lcg(1000, 1),
  my_lcgC(1000, 1),
  check=F
)
```

```
## # A tibble: 2 x 6
##   expression             min   median `itr/sec` mem_alloc
##   <bch:expr>        <bch:tm> <bch:tm>     <dbl> <bch:byt>
## 1 my_lcg(1000, 1)  106.03us 108.98us     8944.    74.31KB
## 2 my_lcgC(1000, 1)   5.37us   6.03us   161443.     7.86KB
```

Some key points

- C++ can be faster than an equivalent implementation in R, especially loops/iteration
- C++ can be more general-purpose, and provides a wider variety of certain data types
- C++ *always* needs to know the type of an object
  - This is true for inputs, outputs, *and* any variables you create
- In C++, indexing begins at 0
- C++ needs a ; at the end of each line
- The Armadillo library provides many useful objects and functions that behave similarly to R counterparts

Your turn

Practice questions on the course website:

https://sta379-s25.github.io/practice_questions/pq_4.html

- Practice writing short functions in C++
- Start in class. You are welcome to work with others
- Practice questions are to help you practice. They are not submitted and not graded
- Solutions are posted on the course website