R `sample` bug description

Kellie Ottoboni

February 17, 2017

What's going on?

`sample` generates simple random samples in two ways. If you are sampling fewer than half of items from $\{1, ..., n\}$ for $n > 10^7$, it calls the `sample2` C function in the file unique.c. Otherwise, it calls the `sample` C function in the file random.c. Each of these two functions handles two cases separately: when $n >= 2^{31}$ and when $n < 2^{31}$. They use the value 2^{31} because this is the maximum integer value. Thus, there are four places in the code that generate a random integer on the range $\{1, ..., n\}$. They are lines 1770 and 1780 in unique.c and lines 516, 524, 538, and 543 in random.c. Each essentially uses the same method; they differ in their data types and storage structures. We include one instance of it below:

```
R_xlen_t n = (R_xlen_t) dn;
double *x = (double *)R_alloc(n, sizeof(double));
double *ry = REAL(y);
for (R_xlen_t i = 0; i < n; i++) x[i] = (double) i;
for (R_xlen_t i = 0; i < k; i++) {
    R_xlen_t j = (R_xlen_t)floor(n * ru());
    ry[i] = x[j] + 1;
    x[j] = x[--n];
}
```

The first line converts the datatype of `dn`, the maximum population index, and renames it `n`. Second and fourth line create an array `x` which contains the indices $\{1, \ldots, \text{dn}\}$ stored as doubles. The third line changes the array `y`, an empty array of length k, to reals. The work starts at the fifth line: we sample a random uniform, multiply it by the maximum population index `n`, and take the floor to convert it to an integer of type R_xlen_t. We take this item out of `x`, add one, and put that item in the corresponding element of `y`. The last element of `x` takes its spot and we decrement `n`. We repeat this until `y` is full.

The question is, are we sampling from `x` uniformly? This depends on the values of `n` and the number of bits output by the random uniform function.

Random integer generation

The main function used to generate random uniforms is `unif_rand()`. It generates numbers on $[0, 1)$ with up to 32 bits of precision, depending on which PRNG the R user specifies.

Some PRNGs in R only return 25-bit integers, as evidenced by the cryptic comments

```
/* Our PRNGs have at most 32 bit of precision, and all have at least 25 */
```

on line 451 of random.c and

```
// more fine-grained  unif_rand() for n > INT_MAX
```

on line 1743 of unique.c.

This is insufficient to generate all possible integers on a range larger than 2^{25}. To compensate for this, they define the following function:

```
static R_INLINE double ru()
{
    double U = 33554432.0;
    return (floor(U*unif_rand()) + unif_rand())/U;
}
```

This function uses two PRNs from `unif_rand()` to get an integer with more bits of precision. First, they multiply a random uniform value on $[0, 1)$ by 2^{25} and take the floor. Assuming this random uniform has at least 25 bits (and is itself uniformly distributed), this will result in an integer between 1 and 2^{25} that truly is uniformly distributed. Then they add another random uniform $[0, 1)$ to it and divide by 2^{25}, resulting in a value on $[0, 1)$ that is uniformly distributed and has 50 bits of precision. This is possible because they store the output as a double. However, note that a double has 53 bits of precision (https://stat.ethz.ch/R-manual/R-devel/library/base/html/double.html), so even this procedure is insufficient to produce all doubles if the chosen PRNG only produces 25-bit integers.

PRNGS in R

R supplies the following PRNGs:

- Wichmann-Hill
- Marsaglia multiply-with-carry
- Super Duper LCG
- Mersenne Twister
- Knuth-TAOCP and Knuth-TAOCP-2002 (a 32-bit GFSR with lagged Fibonnaci sequences)
- L'Ecuyer CMRG

It isn't clear which, if any, of these returns 25-bit integers. Most seem to use 32 bits of precision. Perhaps the `ru()` function is a relic from earlier versions of R.

Back to random integers

The output of these random uniform functions takes on the values $U \in \{0, 2^{-w}, 2 \times 2^{-w}, ..., 1 - 2^{-w}\}$, where $w = 32$ for `unif_rand()` and $w = 50$ for `ru()`. Unless the scaling factor n is a power of two, $\lfloor nU \rfloor$ will be non-uniform: nU rounds down to certain integers more often than others.

> *Theorem:* If $n < 2^w$, then to first order, the ratio of the largest selection probability to the smallest selection probability is $1 + n2^{-w}$. (See Knuth (1997))

As discussed above, there are two regimes: we use `ru()` when $n >= 2^{31}$ and `unif_rand()` when $n < 2^{31}$. When we use `ru()`, the wordsize is at least $w = 50$ bits and at most $w = 53$ bits (this depends on the output of the PRNG we choose). The ratio of selection probabilities only becomes large (on the order of 10^{-3} for *very* large population sizes, say $\log_2(n) > 40$. When using `unif_rand()`, this ratio can get as large as 1.5 if $\log_2(n) \approx 31$. Even if $\log_2(n) = 20$, the ratio is about 1.0002. The problem is worst for large population sizes, just below the threshold that determines whether to use the integer or double version of the random number generator.