Sometimes one needs to mimic the exact behavior of R's `Distributions` within C++ code. The incredible **Rcpp** team has provided access to these distributions through `Rmath.h` (in the `R::` namespace), as well as through the `Rcpp::` namespace where there can be two forms: scalar as in R, and vectorized via Rcpp sugar. The behavior of these functions may not always exactly match what the user expects from the standard R behavior, particularly if attempting to use the functions in `Rmath.h`. In particular, the functions in `Rmath.h` are not vectorized. In what follows, I will use **Rcpp** to mimic the behavior of both the `rmultinom` and `rpois` functions available in base R so that this functionality and behavior is provided in native C++.

The multinomial distribution
generalizes the binomial distribution to `k` discrete outcomes instead of 2; consequently, it is parameterized in terms of `k` probabilities that must sum to 1. The base R function `rmultinom` used for generating multinomial data takes three arguments: `n` the number of simulated data sets to produce, `size`, the number of multinomial outcomes to sample for each data set, and `prob` a numeric vector of probabilities. The function returns a `k` $\times$ `n` integer matrix.

The following C++ code uses the `R::rmultinom` function available in `Rmath.h` to generate `size` multinomial outcomes. The `R::rmultinom` function relies on referencing a pointer to an `IntegerVector` to store the results. We create a helper function, `rmultinom_1`, that draws `size` multinomial outcomes from the multinomial distribution based on the probabilities in `prob`. We then do this `n` independent times in the function `rmultinom_rcpp`. To match the base R functionality, `rmultinom_rcpp` returns a `k` $\times$ `n` `IntegerMatrix`.

```
#include
using namespace Rcpp;

IntegerVector rmultinom_1(unsigned int &size, NumericVector
&probs, unsigned int &N) {
    IntegerVector outcome(N);
    rmultinom(size, probs.begin(), N, outcome.begin());
    return outcome;
}

// [[Rcpp::export]]
IntegerMatrix rmultinom_rcpp(unsigned int &n, unsigned int
&size, NumericVector &probs) {
    unsigned int N = probs.length();
    IntegerMatrix sim(N, n);
    for (unsigned int i = 0; i < n; i++) {
        sim(_,i) = rmultinom_1(size, probs, N);
    }
    return sim;
}
```

We now check if the `rmultinom` and `rmultinom_rcpp` functions produce the same results. We generate a vector of 200 probabilities that sum to 1. We will sample 500 multinomial outcomes and do this independently 20 times.

```
prob <- runif(200)
prob <- prob/sum(prob) # standardize the probabilities
size <- 500
n <- 20

set.seed(10)
sim_r <- rmultinom(n, size, prob)
set.seed(10)
sim_rcpp <- rmultinom_rcpp(n, size, prob)
all.equal(sim_r, sim_rcpp)
```

```
[1] TRUE
```

A benchmark of the functions suggests that the `rmultinom_rcpp` function is very slightly slower than the `rmultinom` function, but that is not really a concern for our purposes.

```
microbenchmark::microbenchmark(
    rmultinom(1000, size, prob),
    rmultinom_rcpp(1000, size, prob)
)
```

```
Unit: milliseconds
                              expr     min      lq    mean  median
uq     max neval cld
      rmultinom(1000, size, prob) 10.8676 10.9925 11.1737 11.0826
11.1898 13.9595   100  a
 rmultinom_rcpp(1000, size, prob) 11.0879 11.2072 11.4897 11.3341
11.6203 13.9920   100   b
```

The poisson distribution is a non-negative discrete distribution characterized by having identical mean and variance. The base R function `rpois` used for generating Poisson data takes two arguments: `n` the number of simulated values to produce, and `lambda`, a positive numeric vector. The `rpois` function cycles (and recycles) through the values in `lambda` for each successive value simulated. The function produces an integer vector of length `n`. We provide similar functionality using the `R::rpois` function available in `Rmath.h`. Note that we cycle through the values of `lambda` so that if the end of the `lambda` vector is reached before we have generated `n` values, then we restart at the beginning of the `lambda` vector.

```
#include
using namespace Rcpp;

// [[Rcpp::export]]
IntegerVector rpois_rcpp(unsigned int &n, NumericVector
&lambda) {
    unsigned int lambda_i = 0;
    IntegerVector sim(n);
    for (unsigned int i = 0; i < n; i++) {
        sim[i] = R::rpois(lambda[lambda_i]);
        // update lambda_i to match next realized value with
correct mean
        lambda_i++;
```

```
            // restart lambda_i at 0 if end of lambda reached
            if (lambda_i == lambda.length()) {
                lambda_i = 0;
            }
        }
        return sim;
    }
```

We now evaluate whether the `rpois` and `rpois` functions produce the same results. We generate a positive vector with 200 values for `lambda` and draw `length(lambda) + 5` independent Poisson values.

```
lambda <- runif(200, 0.5, 3)
set.seed(10)
pois_sim_r <- rpois(length(lambda) + 5, lambda)
set.seed(10)
pois_sim_rcpp <- rpois_rcpp(length(lambda) + 5, lambda)
all.equal(pois_sim_r, pois_sim_rcpp)
```

```
[1] TRUE
```

A benchmark of the two functions suggests the `rpois_rcpp` function may be slightly faster, but once again, that is not our primary concern here.

```
microbenchmark::microbenchmark(
    rpois(length(lambda) + 5, lambda),
    rpois_rcpp(length(lambda) + 5, lambda)
)
```

```
Unit: microseconds
                                  expr   min     lq    mean median
uq    max neval cld
      rpois(length(lambda) + 5, lambda) 7.412 7.7780 8.50529 7.9495
8.2165 60.014   100   b
 rpois_rcpp(length(lambda) + 5, lambda) 6.721 6.9965 7.37430 7.1645
7.4950 21.263   100  a …
```