A tool set for random number generation on GPUs in R

Ruoyong Xu
University of Toronto

Patrick Brown
University of Toronto

Pierre L'Ecuyer
University of Montréal

Abstract

We introduce the R package **clrng** which leverages the **gpuR** package and is able to generate random numbers in parallel on a Graphics Processing Unit (GPU) with the **clRNG** (OpenCL) library. Parallel processing with GPU's can speed up computationally intensive tasks, which when combined with R, it can largely improve R's downsides in terms of slow speed, memory usage and computation mode. **clrng** enables reproducible research by setting random initial seeds for streams on GPU and CPU, and can thus accelerate several types of statistical simulation and modelling. The random number generator in **clrng** guarantees independent parallel samples even when R is used interactively in an ad-hoc manner, with sessions being interrupted and restored. This package is portable and flexible, developers can use its random number generation kernel for various other purposes and applications.

Keywords: GPU, **clrng** package, parallel computing, **clRNG** library.

1. Introduction

In recent years, parallel computing with R [R Core Team 2021] has become a very important topic and attracted lots of interest from researchers [see Eddelbuettel 2021b, for a review]. Although R is one of the most popular statistical software with many advantages, it has drawbacks in memory usage and computation mode aspects [Zhao 2016]. To be more specific, (1) R requires all data to be loaded into the main memory (RAM) and thus can handle a very limited size of data; (2) R is a single-threaded program, it can not effectively use all the computing cores of multi-core processors. Parallel computing is the solution to these drawbacks, for an overview of current parallel computing approaches with R, see CRAN Task View by Eddelbuettel [2021a] at `https://cran.r-project.org/web/views/HighPerformanceComputing.html`.

Graphics Processing Units (GPUs) have the potential to make an important contribution to parallel computing with R. GPUs can perform thousands of computations simultaneously, which makes them powerful for doing massively parallel computing, and they are relatively cheap compared to multicore CPU's. Although there have been a number of R packages developed which provide some GPU capability, they inevitably come with some limitations. Packages such as **gputools**, **gpumatrix**, **cudaBayesreg**, **rpud** (available on github), are no longer maintained, the popular **tensorflow** [Allaire, Eddelbuettel, Golding, and Tang 2016] package uses GPU via Python, which makes it difficult to include as a dependency for new R packages. All of these mentioned packages are restricted to R users with NVIDIA GPUs.

gpuR [Determan Jr. 2017] is the only R package with a convenient and flexible interface between R and GPUs, and it is compatible with many GPU devices. By utilizing the **ViennaCL** [Rupp, Tillet, Rudolf, Weinbub, Morhammer, Grasser, Jungel, and Selberherr 2016] library, it provides a bridge between R and non-proprietary GPUs through the OpenCL (Open Computing Language) backend, which when combined with **Rcpp** [Eddelbuettel and François 2011] gives a building block for other R packages.

Random number generation is critical in simulation-based statistical inference, machine learning and many other scientific fields. While most random number generators are sequential, the R packages **parallel**, **future** [Bengtsson 2021] and **rlecuyer** [Sevcikova, Rossini, and L'Ecuyer 2015] are able to generate random numbers in parallel on multicore CPUs. More specifically, **parallel** writes an interface for the **RngStreams**, a C++ library by L'Ecuyer, Simard, Chen, and Kelton [2002] which is based on a combined multiple-recursive generator (MRG) MRG32k3a. **future** and **rlecuyer** also uses the combined MRG algorithm for generating random numbers. For up-to-date review papers on the generation of random numbers on parallel devices, and GPUs in particular, see: L'Ecuyer [2015]; L'Ecuyer, Munger, Oreshkin, and Simard [2017]; L'Ecuyer, Nadeau-Chamard, Chen, and Lebar [2021].

The **clrng** package described here is currently the only R package that provides facilities for generating random numbers in parallel on GPU. Accomplishing this is complicated because each process must produce an independent (non-overlapping) sequence of random numbers, and in order to ensure reproducibility it should be possible to save and restore the current state of each stream of random numbers at any point. By leveraging the **gpuR** package, **clrng** provides an extensible framework for R package developers to make use of these facilities in C or OpenCL code, and the random number generators guarantee independent parallel samples even when R is used interactively in an ad-hoc manner, with sessions being interrupted and restored.

The remaining sections are organized as follows: In Section 2, we introduce streams and the use of streams in work-items on a GPU device for generating uniform random numbers, and the usage of `runifGpu()`. In Section 3, we introduce two non-uniform RNGs in **clrng**, as examples for users to develop other RNGs of interest on GPUs in R. In Section 4, we apply GPU-generated uniform random numbers in Monte Carlo simulation for Fisher's exact test, we introduce how the random numbers are used and how the algorithm is parallelized and implemented on GPU. Then we provide two real data examples to demonstrate the function usage and its R performance. In Section 5, we show a useful application of normal random numbers on GPU, using them to simulate batches of Gaussian random surfaces with Matérn covariance matrices simultaneously, the simulation also uses GPU-enabled functions from our other package **gpuBatchMatrix**. Finally, the paper concludes with a short summary and a discussion in Section 6.

2. Uniform random number generation

Uniform random number generators (RNGs) are the foundation for simulating random numbers from all types of probability distributions. L'Ecuyer [2012] summarized the usual two steps to generate a random variable in computational statistics: (1) generating independent and identically distributed (i.i.d.) uniform random variables on the interval $(0, 1)$, (2) applying transformations to these i.i.d. $U(0, 1)$ random variables to get samples from the desired

distribution. L'Ecuyer [2012] and Robert and Casella [2004] present many general transformation methods for generating non-uniform random variables, for example, the most frequently used inverse transform method, the Box-Muller algorithm [Box 1958] for Gaussian random variable generation, and so on, which are all built on uniform random variables.

clRNG is an OpenCL library for uniform random number generation, it provides four different RNGs: the MRG31k3p, MRG32k3a, LFSR113, and Philox-4×32-10 generators. These four RNGs use different types of constructions. The **clrng** package uses the MRG31k3p RNG, making it able to generate random numbers on GPUs. We choose MRG31k3p as the base generator for the following reasons: the original **RNGStreams** package [L'Ecuyer *et al.* 2002] was built with MRG32k3a, which was designed to be implemented in double precision, and not with 32-bit integers. The MRG31k3p generator was designed later, specifically for 32-bit integer arithmetic, so it runs faster on the 32-bit GPUs. It is also faster than Philox-4×32-10. The MRG31k3p generator was introduced in the paper [L'Ecuyer and Touzin 2000]. MRG31k3p was also statistically tested extensively and successfully, [see L'Ecuyer and Simard 2007].

In what follows, we will illustrate how to create streams and how to use streams to generate uniform random numbers.

2.1. Creating streams

When a RNG is called in parallel processes or successively called at several places in a program in R, random number generation would be more complicated because there is no guarantee that the streams (analogous to `.Random.seed` in R) do not overlap, and thus the generated sequences of random numbers is possible to have correlation. **clrng** uses multiple distinct streams that are used in work-items that executes in parallel on a GPU device. A popular way of obtaining multiple streams is to take an RNG with a long period and cut the RNG's sequence into very long disjoint pieces of equal length Z, and use each piece as a separate stream. Creating a new stream amounts to computing its starting point. Each of these streams can also be partitioned into substreams with equally-spaced starting points [L'Ecuyer *et al.* 2002; L'Ecuyer 2015], although this is not currently implemented in **clrng**. In general, a stream object contains three elements: the current state of the stream, the initial state of the stream (or seed), and the initial state of the current substream (by default it is equal to the seed). Streams are created sequentially in the way that whenever the user creates a new stream, the software automatically jumps ahead by Z steps to find its initial state, and the three states in the stream object are set to it.

In the clRNG library, the MRG31k3p RNG's entire period of length approximately 2^{185} is divided into approximately 2^{51} non-overlapping streams of length $Z = 2^{134}$. The state (and seed) of each stream is a vector of six 31-bit integers. This size of state is appropriate for having streams running in work-items on GPU cards, while providing a sufficient period length for most applications. The initial state of the first stream (also called "initial seed of the package" or "initial seed of the stream creator" in clRNG) for the MRG31k3p RNG is by default $(12345, 12345, 12345, 12345, 12345, 12345)$.

`setBaseCreator()` sets the initial state of the first stream that is created, it plays a role in **clrng** like the function `SetPackageSeed()` in **RngStreams** and `clrngSetBaseCreatorState()` in **clRNG**. **clrng** is able to create streams both on the host and on the GPU device. The `createStreamsCpu()` function does the former. The R output below creates 4 streams on the

host.

```
R> # creating streams on CPU
R> library("clrng")
R> setBaseCreator(rep(12345, 6))
R> myStreamsCpu <- createStreamsCpu(4)
R> t(myStreamsCpu)
```

```
##        [,1]       [,2]       [,3]       [,4]
##  [1,] 12345  336690377  502033783  739421137
##  [2,] 12345  597094797 1322587635 1475938232
##  [3,] 12345 1245771585 1964121530  730262207
##  [4,] 12345   85196284 1949818481 1630192198
##  [5,] 12345  523477687 1607232546  324551134
##  [6,] 12345 2094976052 1462898381  795289868
##  [7,] 12345  336690377  502033783  739421137
##  [8,] 12345  597094797 1322587635 1475938232
##  [9,] 12345 1245771585 1964121530  730262207
## [10,] 12345   85196284 1949818481 1630192198
## [11,] 12345  523477687 1607232546  324551134
## [12,] 12345 2094976052 1462898381  795289868
```

`setBaseCreator()` has a single argument, the state of the first stream which defaults to a vector of 12345's. The argument of `createStreamsCpu()` is the number of streams to create. Normally users would create many more than the four presented here, as most GPU's have thousands of work items. The default number of streams is 1024.

We move streams to the GPU by converting them to a '`vclMatrix`'.

```
R> myStreamsGpu = vclMatrix(myStreamsCpu)
```

Equivalently, `createStreamsGpu()` creates streams directly on the GPU and returns a '`vclMatrix`', which makes it slightly more efficient when the number of streams is large.

There are a few important notes on the usage of these above functions.

- Calling `setBaseCreator()` creates a hidden object `.Random.seed.clrng`, which (unlike the standard `.Random.seed`) controls the creation of new streams (not random number). We allow users to call `setBaseCreator()` at any time. `setBaseCreator()` should be called once before `createStreamsCpu()` and `createStreamsGpu()` if they would like to select their own initial seeds, or never, in which case streams take the package's default initial seed. Otherwise, there is a small probability of producing streams which overlap.
- Users may switch between `createStreamsCpu()` and `createStreamsGpu()` in the same program, calling either will read and update `.Random.seed.clrng`. This is not recommended as it causes unnecessary data transfer between host and device.
- If an R session is restarted during a task, as long as users saved the R environment (which Rstudio does automatically), the initial states (seeds) of the newly created streams will carry on from the last streams' seed from the previous R session.

- Objects on the GPU are lost when an R session is restarted, even if the workspace is saved. The object `myStreamsGpu` will remain in the workspace if the session is restarted, but the GPU memory containing `myStreamsGpu`'s data will no longer be accessible and an error will occur if the object is accessed. To retain streams when a session is restarted the streams should be copied to the CPU with `as.matrix`.

Unlike R's standard random number generators, the streams need to be explicitly specified when generating random numbers with `clrng`. It would be possible to hide streams from the user by storing them as a hidden object, which the **rlecuyer** package does to some extent, we have chosen not to do so for two reasons. First, the streams would need to be stored on the CPU and transferred to and from the GPU every time they were used. Second, computations might be distributed across multiple GPU devices or multiple computers, and being explicit about which streams are used where ensures reproducibility and avoids the possibility that the same streams might be used on different devices without the user being aware.

2.2. Generating uniform random numbers

The streams created sequentially are used by work-items (the GPU analogy of CPU cores) on a GPU to generate random numbers. In **clrng** each work-item takes one distinct stream to generate random numbers, so the number of streams created should always equal (or exceed) the maximum number of work items likely to be used. The main part of the kernel (functions for execution on the GPU) for generating uniform random numbers is shown in Listing 2.2. Kernels are written in the C-like OpenCL language, in which `__kernel` declares a function as a kernel, and the `__global` prefix to the pointer kernel arguments specifies that they point to global memory space accessible to all work items. Users can set the argument `verbose=2` in the random number generator to print out the kernel, which is slightly more complex than the code in Listing 2.2.

Here the pointers `streams` and `out` refer to the streams and the output matrix respectively, which are stored in global memory. `Nrow` and `Ncol` represent row and column number of the matrix `out` respectively. `Npad` is the "internal" number of columns, and `out` is an `Nrow` by `Ncol` submatrix of a larger `Nrow` by `Npad` matrix.

Each stream's current state is copied to the private memory of each work-item by the function `streamsToPrivate()`, in which `g1` and `g2` point to the first three and second three elements of stream states. The object `startvalue` gives the position of the work item's stream in the `streams` object and is computed with the `NpadStreams` object defined in a macro not shown here.

The function `clrngMrg31k3pNextState()` generates an uniform random integer between 1 and 2147483647, and then scaled to be in the interval $(0, 1)$ by multiplying it by a constant `mrg31k3p_NORM_cl` (defined to be $1/2147483648$). If to generate random integers, `temp` is not scaled. At the end of generating random numbers, streams are transferred back to global memory through the function `streamsFromPrivate()`.

```
__kernel void mrg31k3pMatrix(
  __global int* streams,
  __global float* out,
  int Nrow, int Ncol, int Npad){

int Drow, Dcol;
```

```
uint g1[3], g2[3];
double temp;
const int startvalue = (get_global_id(0) +
get_global_size(0) * get_global_id(1)) * NpadStreams;

streamsToPrivate(streams,g1,g2,startvalue);

for(Drow = get_global_id(0); Drow < Nrow;
    Drow += get_global_size(0)){

    for(Dcol = get_global_id(1); Dcol < Ncol;
        Dcol += get_global_size(1)){

        temp = fact * clrngMrg31k3pNextState(g1, g2);
        out[Drow * Npad + Dcol] = temp;

  }//Dcol
}//Drow

streamsFromPrivate(streams,g1,g2,startvalue);

}//kernel
```

Now we use the 4 streams created in section 2.1 to generate a vector of double-precision i.i.d. $U(0,1)$ random numbers with `runifGpu()`. To view the generated random numbers, we need to convert them to R vectors or matrices, by doing this, the random numbers are moved from the global memory of the GPU device to the host.

```
R> sim_1 = runifGpu(n = 8, streams = myStreamsGpu, Nglobal = c(2,
+    2))
R> as.vector(sim_1)

## [1] 0.735 0.842 0.614 0.216 0.110 0.870 0.649 0.170
```

The arguments of the `runifGpu()` are described as follows:

- `n`: Either a scalar giving the number of samples to generate or a vector of length 2 specifying the size of a matrix to be output.
- `streams`: Streams for random number generation, which must be stored on the GPU as a `vclMatrix` object.
- `Nglobal`: Number of work items, by default it is set as $(64, 8)$.
- `type`: `double`, `float` or `integer`, the format of generated random numbers, defaults to `double`.
- `verbose`: Print extra information if verbose > 1.

The object returned is either a `vclMatrix` or `vclVector`, depending on whether `n` is two- or one-dimensional. Reusing `myStreamsGpu` will produce a vector different form `sim_1`, as the current states of the streams in `myStreamsGpu` have advanced by two positions (each of the 4 work item generated 2 random numbers).

As mentioned earlier, streams on the GPU do not remain in the memory when an R session is restarted. There are two ways to reproduce results, depending on how the program calls streams creator. The simpler way to reproduce is just to have a single program in an R script file with the initial seed for the creator set at the beginning. But it is important that the program always creates the streams in exactly the same order, for example, like the toy R program for demonstration below, we can reproduce the random matrix `sim_mat` just by keeping a record of the initial seed `c(11,22,33,44,55,66)`.

```
R> library("clrng")
R> setBaseCreator(c(11, 22, 33, 44, 55, 66))
R> sim_mat <- matrix(0, nrow = 10, ncol = 6)
R> for (i in 1:10) {
+    if (i%%2 == 0) {
+        streams1 <- createStreamsGpu(4)
+        sim_mat[i, ] <- as.vector(clrng::rnormGpu(n = 6, streams = streams1,
+            Nglobal = c(2, 2)))
+    } else if (i == 3) {
+        streams2 <- createStreamsGpu(4)
+        sim_mat[i, ] <- as.vector(runifGpu(n = 6, streams = streams2,
+            Nglobal = c(2, 2)))
+    } else {
+        streams3 <- createStreamsGpu(8)
+        sim_mat[i, ] <- as.vector(rexpGpu(n = 6, streams = streams3,
+            Nglobal = c(4, 2)))
+    }
+}
R> sim_mat
```

Were we to replace the `for` loop above with a parallel equivalent (i.e. `mcmapply`), the order of stream creation could change with each program execution and the result would not be reproducible. For more complicated applications, we recommend users to save the matrix of streams (current states and initial states) on the CPU in a file as a `.rds` object. And later recall the saved streams for regenerating results, the streams will start from their current states after they are loaded. This is a safer way than the previous one for reproducing results in simulations. Below shows the code that saves streams to a data file called `myStreams.rds` on CPU and then load it back and transfer it to a 'vclMatrix' `streams_saved` on GPU, and using it to (re)generate some Normal random numbers.

```
R> saveRDS(as.matrix(myStreamsGpu), "myStreams.rds")
R> # Load the streams object as streams_saved
R> streams_saved <- vclMatrix(readRDS("myStreams.rds"))
R> clrng::rnormGpu(n = 6, streams = streams_saved, Nglobal = c(2,
+    2))
```

3. Some non-uniform random number generation

3.1. Normal random number generation

We apply the Box-Muller transformation to $U(0,1)$ random numbers to generate standard normal random numbers. As shown in Algorithm 1, Box-Muller algorithm takes two independent, standard uniform random variables U_1 and U_2 and produces two independent, standard Gaussian random variables X and Y, where R and Θ are polar coordinate random variables. The Box-Muller algorithm is a very good choice for Gaussian transform on GPU's compared to other transform methods [Howes and Thomas 2007], because this algorithm has no branching or looping, which are the things GPU's are poor at.

Algorithm 1: Box-Muller algorithm.

1, Generate U_1, U_2 i.i.d. from $U(0,1)$;
2, Define

$$R = \sqrt{-2 * \log U_1},$$
$$\Theta = 2\pi * U_2,$$
$$X = R * \cos(\Theta),$$
$$Y = R * \sin(\Theta);$$

3, Take X and Y as two independent draws from $N(0,1)$;

Listing 3.1 shows a fragment of the kernel that generates standard Gaussian random numbers. The kernel has work items operating in pairs with shared local memory, the U_1 and U_2 are generated in parallel and stored in the two-dimenstional vector `part` below. The `get_local_id(1)` command will return either zero or one, for the first and second item in the pair respectively. As the work-items in a work-group proceed at different rates, `barrier(CLK_LOCAL_MEM_FENCE)` ensures correct ordering of memory operations to local memory, so that Gaussian random numbers $(X_1, Y_1), \ldots, (X_n, Y_n)$ are generated in pairs correctly, errors such as $(X_n, Y_{(n-1)})$ or $(X_{(n-1)}, Y_{(n)})$ are avoided.

```
__kernel void mrg31k3pMatrix(
  __global int* streams,
  __global double* out,
  int Nrow, int Ncol, int Npad, int NpadStreams){

int Drow, Dcol;
uint g1[3], g2[3];
int startvalue = (get_global_id(0) * get_global_size(1) +
  get_global_id(1)) * NpadStreams;
const double fact[2] = { mrg31k3p_NORM_cl, TWOPI * mrg31k3p_NORM_cl };
const double addForSine[2] = { 0.0, -PI_2 };
local double part[2];

streamsToPrivate(streams,g1,g2, startvalue);

for(Drow=get_global_id(0); Drow < Nrow; Drow += get_global_size(0)){
```

```
  for(Dcol=get_global_id(1); Dcol < Ncol; Dcol += get_global_size(1)){

    part[get_local_id(1)] = fact[get_local_id(1)] *
      clrngMrg31k3pNextState(g1, g2);
    barrier(CLK_LOCAL_MEM_FENCE);
    out[Drow * Npad + Dcol] = sqrt( -2.0*log(part[0]) ) *
      cos(part[1] + addForSine[get_local_id(1)] );
    barrier(CLK_LOCAL_MEM_FENCE);

  }//Dcol
}//Drow
streamsFromPrivate(streams,g1,g2,startvalue);
}//kernel
```

Below we generate a large-size matrix of 100 million double-precision Gaussian random numbers, and compare the run-time between using `stats::rnorm()` and `rnormGpu()`. The best performance we have seen using `rnormGpu()` is above over 100 times faster with 512×128 work-items than using the standard (and single-threaded) `stats::rnorm()`. The difference in elapsed time becomes larger when matrix size increases.

```
R> streams <- createStreamsGpu(512 * 128)
R> system.time(rnormGpu(c(10000, 10000), streams = streams, Nglobal = c(512,
+    128), type = "double"))

##    user  system elapsed
##   0.053   0.000   0.053

R> system.time(matrix(rnorm(10000^2), 10000, 10000))

##    user  system elapsed
##   7.006   0.777   7.782
```

3.2. Exponential random number generation

The exponential random variates are produced by applying the inverse transform method on i.i.d. $U(0,1)$ random numbers. The random variable $X \sim \text{Exponential}(\lambda)$ has cumulative distribution function $F_X(x) = 1 - e^{-\lambda x}$ for $x \geq 0$ and $\lambda > 0$. The inverse of $F_X(\cdot)$ is $F_X^{-1}(y) = -(1/\lambda)\log(1-y)$, for $0 \leq y < 1$. The kernel is not shown because it is mostly same as the kernel for uniform and normal random numbers, except for the part that does the inverse transform.

Below is an example that produces a 2×4 matrix of Exponential random numbers with expectation equal to 1.

```
R> r_matrix <- rexpGpu(c(2, 4), rate = 1, myStreamsGpu, Nglobal = c(2,
+    2), type = "double")
R> as.matrix(r_matrix)

##      [,1]  [,2]  [,3]  [,4]
## [1,] 1.00 0.689 2.218 1.167
## [2,] 1.48 1.205 0.406 0.241
```

The arguments of the `rexpGpu()` are described as follows:

- `n`: A vector of length 2 specifying the row and column number if to create a matrix, or a number specifying the length if to create a vector.
- `streams`: Streams for random number generation, streams cannot be missing.
- `Nglobal`: Global index space. Default is $(64, 8)$.
- `type`: "double" or "float" format of generated random numbers.
- `verbose`: Print extra information if verbose > 1. Default is FALSE.

4. An application of uniform RNG: Fisher's simulation

The GPU-generated random numbers can be applied in suitable statistical simulations to accelerate computations. One application of GPU-generated random numbers in **clrng** is Monte Carlo simulation for Fisher's exact test. Fisher's exact test is applied for analyzing usually 2×2 contingency tables when one of the expected values in table is less than 5. Different from methods which rely on approximation, Fisher's exact test computes directly the probability of obtaining each possible combination of the data for the same row and column totals (marginal totals) as the observed table, and get the exact p-value by adding together all the probabilities of tables as extreme or more extreme than the one observed. However, when the observed table gets large in terms of sample size and table dimensions, the number of combinations of cell frequencies with the same marginal totals gets very large, [Mehta and Patel 2011, p. 23] shows a 5×6 observed table that has 1.6 billion possible tables. Calculating the exact P-values may lead to very long run-time and can sometimes exceed the memory limits of your computer. Hence, the option `simulate.p.value = TRUE` in `stats::fisher.test()` is provided, which enables computing p-values by Monte Carlo simulation for tables larger than 2×2.

The test statistic calculated for each random table is $-\sum_{i,j} \log(n_{ij}!), i = (1, \ldots, I), j = (1, \ldots, J)$, (i.e., minus log-factorial of table), where I and J are the row and column number of the observed table. This test statistic can also be independently calculated for a table by `clrng::logfactSum()`. Given an observed table and a number of replicates B, the Monte Carlo simulation for Fisher's exact test does the following steps:

1. Calculate the test statistic for the observed table.
2. In each iteration, simulate a random table with the same dimensions and marginal totals as the observed table using the `rcont2()` algorithm, compute and optionally save the test statistic from the random table.
3. Count the number of iterations (*Counts*) that have test statistics less or equal to the one from the observed table.
4. Estimate p-value using $\frac{1+Counts}{B+1}$.

Step 1 and step 2-3 are done on a GPU with two kernels enqueued sequentially. For step 2, `clrng::fisher.sim()` adapted the function `rcont2()` used by `stats::fisher.test()` for constructing random two-way tables with given marginal totals. The algorithm [see Patefield 1981] samples the entries row by row, one at a time, conditional on the values of the entries

already sampled. The conditional probabilities for the possible values of the next entry are updated dynamically each time a new entry is sampled. Then this entry is sampled by standard inversion of the cumulative distribution function, using one $U(0,1)$ random number. For an $I \times J$ table, this requires $(I-1)(J-1)$ random numbers (the last column and last row do not need to be sampled). Finally, one can compute the test statistic for this newly sampled table. On a GPU, this step can be replicated say n times in parallel by creating n distinct random streams and launching n separate work-items. Each work-item takes a random stream as input, performs all of Step 2 and 3, and returns the value of the test statistic on the GPU. Computing the p-value on the CPU (Step 4) is then a trivial operation. Step 2 of saving test statistics from the random tables is made optional, which can reduce the run-time. By the way, there are a lot of other more recent methods for sampling (larger) contingency tables, many of them use Markov Chain Monte Carlo, the use of streams and GPU would be quite different in that case. See for examples [Miller and Harrison 2013; Kayibi, Pirzada, and Chishti 2018; Dyer, Kannan, and Mount 1997]. Doing the `fisher.sim()` function on GPU opens up many possibilities for future work, the specific implementation we've done for the tables isn't necessarily the optimal one.

The arguments of the `clrng::fisher.sim()` are described as follows:

- `x`: A contingency table, a "vclMatrix" object.
- `N`: Number of simulation runs.
- `streams`: Streams for random number generation, streams cannot be missing.
- `type`: "double" or "float" of returned test statistics.
- `returnStatistics`: If TRUE, return all test statistics.
- `Nglobal`: Global index space. Default is set as $(64, 16)$.

Users request N number of replicates, while the actual number of replicates to be executed on GPU is larger (`ceiling(N/prod(Nglobal))*prod(Nglobal)`). We show the advantage of `clrng::fisher.sim()` by computing the p-values for two real data examples: one with a relatively big p-value and another with a very small p-value, and compare the run-time with using `stats::fisher.test()` for each of the data sets on two computers: one with a very good CPU and an ordinary GPU, the other is equipped with an excellent GPU and an ordinary CPU. The R outputs for testing on computer 2 is shown in the following.

4.1. Comparing run-time: Monthly birth anomolies

The 2-way contingency table in Table 1 comes from the 2018 Natality public use data from the Centers for Disease Control and Prevention's National Center for Health Statistics [for Health Statistics 2019]. The 2018 natality data file may be downloaded at `https://www.cdc.gov/nchs/data_access/VitalStatsOnline.htm`. Table 1 is a 12×12 table that shows frequencies for congenital anomalies of the newborn by birth month in 2018 within the United States. The column variables of these two tables represent the twelve categories of congenital anomalies of the newborn: 1) Anencephaly; 2) Meningomyelocele/Spina bifida; 3) Cyanotic congenital heart disease; 4) Congenital diaphragmatic hernia; 5) Omphalocele; 6) Gastrochisis; 7) Limb reduction defect; 8) Cleft lip with or without cleft palate; 9) Cleft palate alone; 10) Down syndrome; 11) Suspected chromosomal disorder; and 12) Hypospadias.

Table 1: Monthly counts of birth anomalies.

	Ane	Men	Cya	Her	Omp	Gas	Lim	Cle	Pal	Dow	Chr	Hyp
Jan	29	55	172	46	39	73	48	183	77	103	102	174
Feb	25	45	175	35	31	55	34	142	81	115	100	180
Mar	31	48	182	41	47	72	40	200	86	90	96	180
Apr	34	45	186	36	32	75	42	173	56	87	90	193
May	33	40	187	46	24	80	35	180	75	91	100	197
Jun	34	48	189	35	33	75	45	154	74	102	100	182
Jul	26	43	198	34	21	74	36	179	79	86	92	193
Aug	24	41	189	44	43	62	48	183	88	109	94	194
Sept	34	44	147	40	37	66	36	158	73	112	103	196
Oct	25	43	207	45	31	65	49	181	77	108	115	220
Nov	36	55	188	39	39	62	43	144	68	98	79	173
Dec	23	48	196	31	31	71	31	177	86	86	73	156

```
R> streams <- createStreamsGpu(n = 256*64)
R> month_gpu <- vclMatrix(month, type = "integer")
R> system.time(result_month <-
+  clrng::fisher.sim(month_gpu, 1e6,
+     streams=streams, type="double",
+     returnStatistics=TRUE,  Nglobal = c(256,64)))

##    user  system elapsed
##   0.337   0.008   0.344

R> unlist(result_month[c("threshold", "simNum", "counts")])

## threshold    simNum    counts
##    -47955   1015808    409429

R> result_month$p.value

## [1] 0.403
```

We got 410825 cases whose test statistics are below the observed threshold based on an actual number of 1015808 simulations, so the p-value from `clrng::fisher.sim()` for this table is about 0.40443, which is close enough to the p-value from `stats::fisher.test()`. `clrng::fisher.sim()` takes about 0.31 seconds, **clrng** accounts for 2.17% of the elapsed time on CPU.

```
R> # using CPU
R> system.time(result_monthcpu <- stats::fisher.test(month, simulate.p.value = TRUE,
+    B = 1015808))
```

Table 2: Day-of-week birth anomaly data

	Ane	Men	Cya	Her	Omp	Gas	Lim	Cle	Pal	Dow	Chr	Hyp
Mon	30	34	173	37	23	80	49	191	83	122	109	216
Tue	60	121	383	80	83	131	71	349	146	164	168	352
Wed	51	106	417	92	73	145	72	333	136	179	196	351
Thu	60	86	362	69	74	120	85	326	132	220	187	359
Fri	52	94	347	87	59	123	68	323	145	170	166	345
Sat	52	63	323	67	64	135	73	316	170	189	188	357
Sun	49	51	211	40	32	96	69	216	108	143	130	258

```
##    user  system elapsed
##  14.859   0.014  14.874

R> result_monthcpu$p.value

## [1] 0.405
```

4.2. Comparing run-time: Day-of-week birth anomolies

Table 2 is a 7×12 table shows frequencies for congenital anomalies of the newborn by birth day of week in 2018 within the United States.

```
R> week_GPU <- gpuR::vclMatrix(week, type = "integer")
R> system.time(resultWeek <- clrng::fisher.sim(week_GPU, 10000000,
+    streams = streams, type = "double", returnStatistics = TRUE,
+    Nglobal = c(256, 64)))

##    user  system elapsed
##   1.938   0.017   1.954

R> unlist(resultWeek[c("threshold", "simNum", "counts")])

## threshold    simNum    counts
##    -54990  10010624      1350

R> resultWeek$p.value

## [1] 0.000135
```

The "week" table has a much smaller p-value: around 0.000125, which should require a larger number of simulations to get a more accurate p-value. With more than ten million simulations, we get 1205 cases and a p-value around 0 with `fisher.sim()`. `stats::fisher.test()` takes 92.45 seconds, while `fisher.sim()` takes about 1.95 seconds, the GPU elapsed time is decreased to about 2.10% of the time taken on CPU.

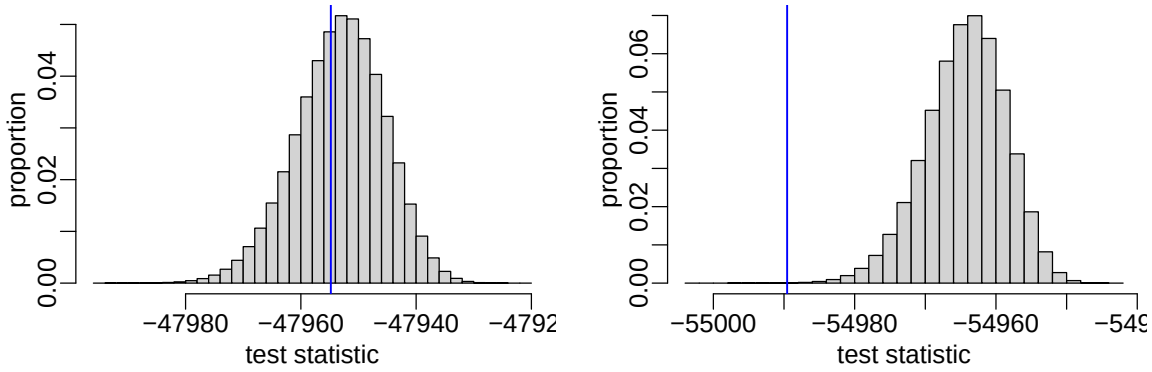
```
R> # using CPU
R> system.time(result_weekcpu <- fisher.test(week, simulate.p.value = TRUE,
+    B = 10010624))

##    user  system elapsed
##  91.311   0.186  91.432

R> result_weekcpu$p.value

## [1] 0.000127
```

4.3. A summary of the results

We summarized the comparison results in Table 3 and plot the test statistics in Figure 1.

Table 3: Summary of comparions of Fisher's test simulation on different devices. Computer 1 is equipped with CPU Intel Xenon W-2145 3.7Ghz and AMD Radeon VII. Computer 2 is equipped with VCPU Intel Xenon Skylake 2.5Ghz and VGPU Nvidia Tesla V100.

	Computer 1		Computer 2		
B	Intel 2.5ghz	AMD Radeon	Intel 3.7ghz	NVIDIA V100	Data
P-value					
1M	0.403804	0.403507	0.4035606	0.403507	month
10M	0.0001251	0.0001274	0.0001202	0.0001274	week
Run-time					
1M	49.33	2.15	14.8	0.32	month
10M	327.27	10.23	92	1.95	week



(a) Month data (b) week data

Figure 1: Approximate sampling distributions of the test statistics from the two examples Month and Week. The test statistic values of the observed tables are marked with a blue vertical line on each plot.

5. Simulating Gaussian Random Fields

Simulating Gaussian random fields (GRF's) is computationally intensive due the high dimensionality of the problem, a 100 by 100 grid has 10,000 cells and a 10,000 by 10,000 variance matrix. Writing $x_1, \ldots, x_n$ as point locations (i.e. centres of grid cells), the joint distribution of $U = (U_{x_1}, \ldots, U_{x_n})^\top$ is multivariate Normal (MVN). An isotropic random field has a variance matrix with entries which depend on the distances between point locations, and a GRF with geometric anisotoropy has entries depending on the direction as well as the distance. A geometrically anisotropic GRF with a Matérn [1960] correlation function is defined as

$$
\begin{aligned}
U &= [U(x_1), \ldots, U(x_n)]^\top \sim \mathrm{MVN}(0, \Sigma), \\
\Sigma_{ij} &= \mathrm{COV}[U(x_i), U(x_j)] = \sigma^2 \frac{2^{\kappa-1}}{\Gamma(\kappa)} \left(\sqrt{8\kappa}||d_{ij}||/\phi\right)^\kappa K_\kappa \left(\sqrt{8\kappa}||d_{ij}||/\phi\right), \\
d_{ij} &= \begin{bmatrix} 1 & 0 \\ 0 & \omega \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} (x_i - x_j)
\end{aligned}
$$

where $K_\kappa(\cdot)$ is a modified Bessel function of the second kind with order κ and $\Gamma(\cdot)$ is the standard gamma function. The model parameters are:

- σ, the (marginal) variance $\mathrm{var}(U_i) = \sigma$;
- κ, the shape parameter controlling the differentiability of $U(\cdot)$;
- ϕ, the range parameter controlling how quickly correlation decays with distance in the direction where correlation is strongest;
- $\omega \geq 1$, a parameter controlling how much faster correlation decays in the direction where correlation is weakest; and
- θ, the angle of rotation required to put the direction of maximum correlation on the x-axis.

The standard isotropic Matérn model is obtained by setting $\omega = 1$. There are several alternative parameterisations of Matérn covariance functions in literature [see Haskard 2007]. In **gpuBatchMatrix**, we use the above form with the $\sqrt{8\kappa}$ term as it allows ϕ to be interpretable as the distance beyond which correlation is less than 0.14.

The conceptually simplest way of simulating a GRF is to multiply a vector of independent standard Normals by the Cholesky decomposition of Σ. This is the direct matrix decomposition method. Liu, Li, Sun, and Yu [2019] gives a comprehensive review on seven popular methods for GRF generation, all of these methods other than the matrix decomposition method, involve approximations to the random field and have specific requirements on the type of grid or covariance functions. The direct matrix decomposition method is exact, it works for all covariance functions and can generate random fields on arbitrary point locations, and is straightforward to implement.

Algorithm 2: Gaussian random fields simulation using the direct matrix decomposition method.

1, Calculate the covariance matrix Σ between locations;

2, Compute the Cholesky decomposition of $\Sigma = L \cdot D \cdot L^\top$, where L is a lower unit triangular matrix, and D is a diagonal matrix;

3, Generate on GPU a random matrix $Z = (Z_1, Z_2, \ldots, Z_n) \sim \mathrm{MVN}(0, I_n)$, where I_n is a $n \times n$ identity matrix;

4, Compute the random samples from U in batches using $U = L \cdot D^{\frac{1}{2}} \cdot Z$;

The direct method is detailed in Algorithm 2. It is computationally demanding for two reasons. First, evaluating the Bessel function is an iterative procedure that must be done for each of the $n^2/2$ entries of Σ. Second, the time cost of Cholesky decomposition of the covariance matrix is $\mathcal{O}(n^3)$, and is $\mathcal{O}(n^2)$ for the matrix-vector multiplication $L * Z$ [Liu *et al.* 2019]. Methods employing approximations gain benefits in computations while sacrificing exactness, we preserve the exactness and also largely reduce the computation burden by utilizing our other R package **gpuBatchMatrix**. **gpuBatchMatrix** address this issue since it computes batches of Matérn covariance matrices in parallel, and does Cholesky decomposition and matrix-matrix multiplication in batches in parallel on GPU.

Other R packages that offer simulation of GRF's such as **geoR** [Ribeiro Jr. and Diggle 2001], does not work for large number of locations. The **RandomFields** [Schlather, Malinowski, Menck, Oesting, and Strokorb 2015] package has several different methods for simulation of Gaussian fields, among which the circulant embedding, which is an improved method on direct matrix decomposition, and some variants of the method like the cut-off embedding [Gneiting, Ševčíková, Percival, Schlather, and Jiang 2006] are also exact and fast for isotropic GRF's, however, they work only on rectangular grids.

5.1. Simulating Gaussian random fields with Matérn covariances

The following shows an example, in which we simulate on GPU eight GRF's of Matérn covariance at one time with four sets of parameters, by taking advantage of the GPU-capabilities offered by **clrng** and **gpuBatchMatrix** together. Motivated by the classic Swiss rain dataset, we simluate random fields on a grid of points covering Switzerland.

Step 1, we create a grid of points covering Switzerland using the the `swissBorder` object from **geostatsp**. The grid is specified as having 90 cells in the horizontal direction.

```
R> library("geostatsp")
R> data("swissRain", package = "geostatsp")
R> myRaster = geostatsp::squareRaster(swissBorder, 90)
R> dim(myRaster)

## [1] 57 90  1
```

Step 2, create a matrix with four different parameter sets. The five parameters named are κ, ϕ, σ^2, ω and θ.

```
R> params = rbind(c(shape = 1.25, range = 50 * 1000, variance = 1.5,
+    anisoRatio = 1, anisoAngleRadians = 0), c(2.15, 60 * 1000,
+    2, 4, pi/7), c(0.6, 30 * 1000, 2, 2, pi/5), c(3, 30 *
+    1000, 2, 2, pi/7))

R> params

##      shape range variance anisoRatio anisoAngleRadians
## [1,]  1.25 50000      1.5          1             0.000
## [2,]  2.15 60000      2.0          4             0.449
## [3,]  0.60 30000      2.0          2             0.628
## [4,]  3.00 30000      2.0          2             0.449
```

Step 3, compute the Matérn covariance matrices using `gpuBatchMatrix::maternBatch()`, the returned Matérn covariance matrices are each of size 5130×5130 and are stacked by row in the output `maternCov`.

```
R> maternCov = gpuBatchMatrix::maternBatch(params, myRaster,
+    Nglobal = c(128, 64), Nlocal = c(16, 4))
R> dim(maternCov)

## [1] 20520  5130
```

Step 4, perform Cholesky decomposition on `maternCov`. The first argument in `cholBatch()` specifies the object to take Cholesky decomposition. Computed unit lower triangular matrices L_i's are stacked by row in `maternCov`. The diagonal values of each D_i are returned and stored in each row of `diagMat`. So if each batch Σ_i is of size $n \times n$, then each batch D_i is $1 \times n$.

```
R> diagMat = gpuBatchMatrix::cholBatch(maternCov, Nglobal = c(128,
+    8), Nlocal = c(32, 8))
```

$$\begin{bmatrix} \Sigma_1 \\ \Sigma_2 \\ \Sigma_3 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} L_1 \\ L_2 \\ L_3 \\ \vdots \end{bmatrix} \text{ and } \begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \end{bmatrix} \tag{1}$$

Step 5, create some streams on GPU and generate two standard Gaussian random vectors `zmatGpu`$= (Z_1, Z_2)$ using `clrng::rnormGpu()`, in which `c(nrow(materCov),2)` specifies the number of rows and columns of `zmatGpu`.

```
R> streamsForGrf <- createStreamsGpu(128 * 64)
R> zmatGpu = clrng::rnormGpu(c(nrow(maternCov), 2), streams = streamsForGrf,
+    Nglobal = c(128, 64), type = "double")
```

Step 6, compute $U = L * D^{(1/2)} * Z$ in batches, see the following illustration for the matrix operation that `multiplyLowerDiagonalBatch()` does. `maternCov`, `diagMat`, and `zmatGpu` correspond to the matrices L, D and $Z = (Z_1, Z_2)$ respectively.

```
R> simMat = gpuBatchMatrix::multiplyLowerDiagonalBatch(maternCov,
+    diagMat, zmatGpu, diagIsOne = TRUE, transformD = "sqrt",
+    Nglobal = c(128, 64, 2), Nlocal = c(8, 2, 1), NlocalCache = 1000)
```

$$\begin{bmatrix} L_1 \\ L_2 \\ L_3 \\ \vdots \end{bmatrix} * \begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \end{bmatrix} * \begin{bmatrix} Z_{11} & Z_{12} \end{bmatrix} = \begin{bmatrix} L_1 D_1 Z_{11} & L_1 D_1 Z_{12} \\ L_2 D_2 Z_{11} & L_2 D_2 Z_{12} \\ L_3 D_3 Z_{11} & L_3 D_3 Z_{12} \\ \vdots & \vdots \end{bmatrix} \tag{2}$$

Step 7, Finally, convert the results to a spatial `raster` object and plot them.

```
R> simRaster = raster::brick(myRaster, nl = ncol(simMat) * nrow(params))
R> values(simRaster) = as.vector(as.matrix(simMat))
```

A simple plot can be produced with

```
R> plot(simRaster)
```

and Figure 2 uses some facilities from the **mapmisc** [Brown 2021] package.

6. Discussion

The package **clrng** has been created to make GPU-generated uniform and some non-uniform random numbers accessible for R users, it enables reproducible research in simulations by setting initial seeds in streams and saving and reloading the current states of streams. We further applied the GPU-generated random numbers in suitable statistical simulations, such as the Monte Carlo simulation for Fisher's exact test, and exact Gaussian spatial surfaces simulation, for which we are able to calculate quantiles for the normal distribution, our package **gpuBatchMatrix** adds additional help in the second application by taking the computational work for batches of Matérn covariance matrices, Cholesky decomposition and matrix multiplication on GPU. Comparison of performance between using **clrng** and using classic R on CPU for some real data examples has demonstrated significant improvement in execution time.

By leveraging the **gpuR** package, **clrng** provides a user-friendly interface that bridges R and OpenCL, users can use the facilities in our package without the need to know the complex OpenCL or even the C++ code. **clrng** is portable as its backend OpenCL supports multiple types of processors, and it is also flexible as its kernels can be incorporated or reconstructed in other R packages for other applications.

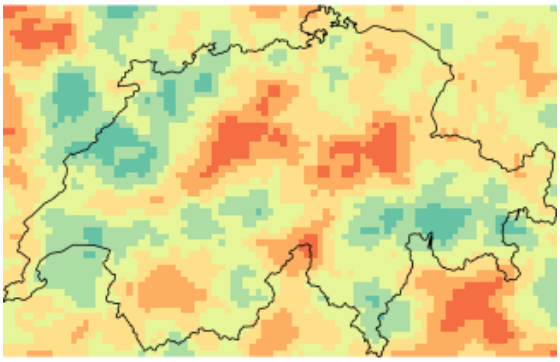
clrng is limited by the number and type of OpenCL RNGs used and the features of the **gpuR** package, as it depends upon the **gpuR** package. If developers wants to use our package to do something that's not supported in **gpuR**, for example "sparse" class objects, they would have to write OpenCL code to implement it.

Future work on **clrng** package could be: (1) Explore other RNGs, for example the MRG32k3a, LFSR113, and Philox-4×32-10 generators in clRNG library, and compare the R performance between using different GPU RNGs. (2) Now that the package can do Cholesky decomposition and matrix multiplication in batches on GPU, which is a motivation for us to work on parallel likelihood evaluations on GPU for Gaussian spatial models in the next step. (3) Create a "sparse" matrix class on GPU and use it for simulating Gaussian random fields with sparse correlation structures such as the Gaussian Markov random fields.

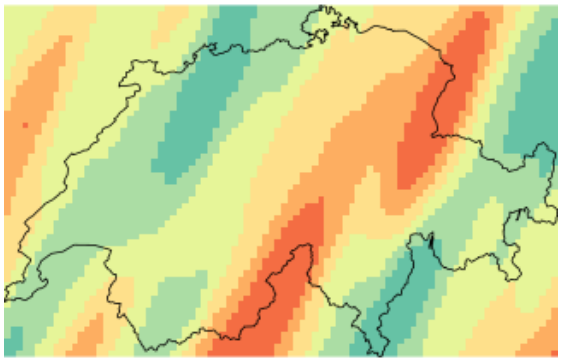
References

Allaire J, Eddelbuettel D, Golding N, Tang Y (2016). *tensorflow: R Interface to TensorFlow*. URL https://github.com/rstudio/tensorflow.

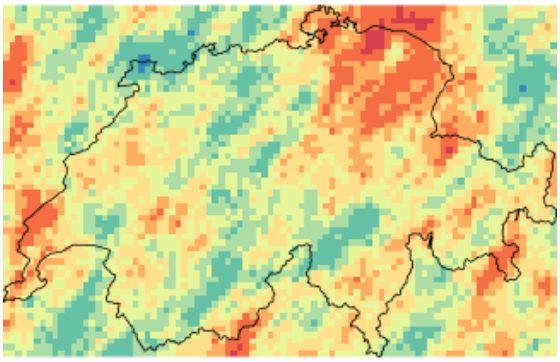
Bengtsson H (2021). "A Unifying Framework for Parallel and Distributed Processing in R using Futures." 10.32614/RJ-2021-048, URL https://journal.r-project.org/archive/2021/RJ-2021-048/index.html.



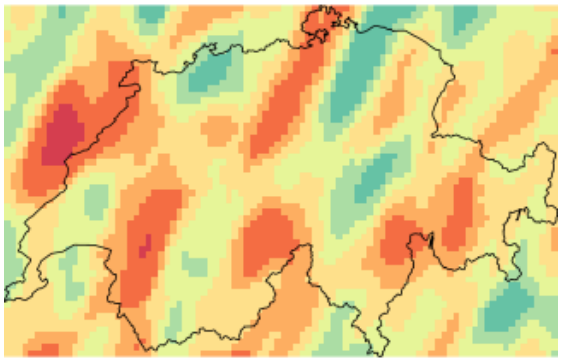
(a) parameter 1, simuation 1



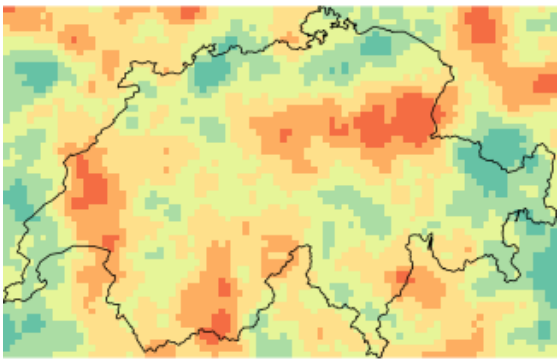
(b) parameter 2, simuation 1



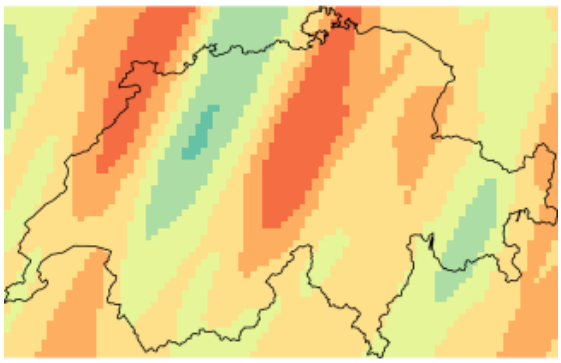
(c) parameter 3, simuation 1



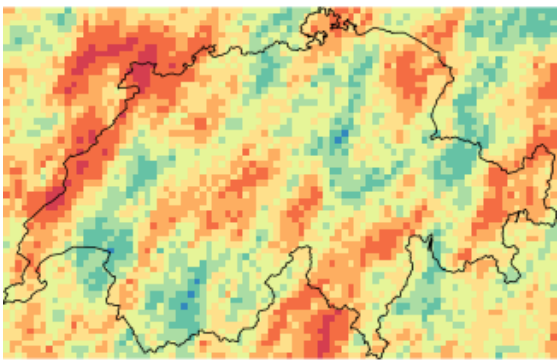
(d) parameter 4, simuation 1



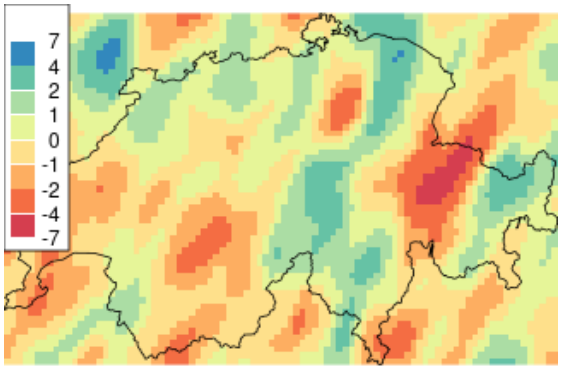
(e) parameter 1, simuation 2



(f) parameter 2, simuation 2



(g) parameter 3, simuation 2



(h) parameter 4, simuation 2

Figure 2: Simulated Gaussian random fields

Box GE (1958). "A note on the generation of random normal deviates." *Ann. Math. Statist.*, **29**, 610–611.

Brown PE (2021). *mapmisc: Utilities for Producing Maps*. R package version 1.8.0, URL https://CRAN.R-project.org/package=mapmisc.

Determan Jr C (2017). *gpuR: GPU Functions for R Objects*. R package version 2.0.0, URL http://github.com/cdeterman/gpuR.

Dyer M, Kannan R, Mount J (1997). "Sampling contingency tables." *Random Structures & Algorithms*, **10**(4), 487–506.

Eddelbuettel D (2021a). "CRAN Task View: High-Performance and Parallel Computing with R." https://cran.r-project.org/web/views/HighPerformanceComputing.html. Version: 2021-11-08.

Eddelbuettel D (2021b). "Parallel computing with R: A brief review." *Wiley Interdisciplinary Reviews: Computational Statistics*, **13**(2), e1515.

Eddelbuettel D, François R (2011). "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08. URL https://www.jstatsoft.org/v40/i08/.

for Health Statistics NC (2019). "Natality 2018. Public use file." Annual internet product. 2019. Available at http://www.cdc.gov/nchs/data_access/VitalStatsOnline.htm.

Gneiting T, Ševčíková H, Percival DB, Schlather M, Jiang Y (2006). "Fast and exact simulation of large Gaussian lattice systems in R^2: exploring the limits." *Journal of Computational and Graphical Statistics*, **15**(3), 483–501.

Haskard KA (2007). *An anisotropic Matern spatial covariance model: REML estimation and properties*. Ph.D. thesis.

Howes L, Thomas D (2007). "Efficient random number generation and application using CUDA." *GPU gems*, **3**, 805–830.

Kayibi KK, Pirzada S, Chishti T (2018). "Sampling contingency tables." *AKCE International Journal of Graphs and Combinatorics*, **15**(3), 298–306. ISSN 0972-8600. doi:https://doi.org/10.1016/j.akcej.2017.10.001. URL https://www.sciencedirect.com/science/article/pii/S0972860017302396.

L'Ecuyer P (2015). "Random Number Generation with Multiple Streams for Sequential and Parallel Computers." In L Yilmaz, WKV Chan, I Moon, TMK Roeder, C Macal, MD Rossetti (eds.), *Proceedings of the 2015 Winter Simulation Conference*, pp. 31–44. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

L'Ecuyer P, Munger D, Oreshkin B, Simard R (2017). "Random Numbers for Parallel Computers: Requirements and Methods, with Emphasis on GPUs." *Mathematics and Computers in Simulation*, **135**, 3–17. Open access at http://dx.doi.org/10.1016/j.matcom.2016.05.005.

L'Ecuyer P, Nadeau-Chamard O, Chen YF, Lebar J (2021). "Multiple Streams with Recurrence-Based, Counter-Based, and Splittable Random Number Generators." In *Proceedings of the 2021 Winter Simulation Conference*. To appear, see https://www-labs.iro.umontreal.ca/~lecuyer/myftp/papers/wsc21rng.pdf.

L'Ecuyer P, Simard R (2007). "TestU01: A C Library for Empirical Testing of Random Number Generators." *ACM Transactions on Mathematical Software*, **33**(4), Article 22.

L'Ecuyer P, Simard R, Chen EJ, Kelton WD (2002). "An object-oriented random-number package with many long streams and substreams." *Operations research*, **50**(6), 1073–1075.

L'Ecuyer P, Touzin R (2000). "Fast Combined Multiple Recursive Generators with Multipliers of the Form $a = \pm 2^q \pm 2^r$." In JA Joines, RR Barton, K Kang, PA Fishwick (eds.), *Proceedings of the 2000 Winter Simulation Conference*, pp. 683–689. IEEE Press.

Liu Y, Li J, Sun S, Yu B (2019). "Advances in Gaussian random field generation: a review." *Computational Geosciences*, **23**(5), 1011–1047.

L'Ecuyer P (2012). "Random number generation." In JE Gentle, WK Härdle, Y Mori (eds.), *Handbook of computational statistics*, pp. 35–71. Springer, Berlin, Heidelberg.

Matérn B (1960). "Spatial variation, Technical Report." *Statens Skogsforsningsinstitut, Stockholm*.

Mehta CR, Patel NR (2011). "IBM SPSS exact tests." *Armonk, NY: IBM Corporation*, pp. 23–24.

Miller JW, Harrison MT (2013). "Exact sampling and counting for fixed-margin matrices." *The Annals of Statistics*, **41**(3), 1569 – 1592. doi:10.1214/13-AOS1131. URL https://doi.org/10.1214/13-AOS1131.

Patefield W (1981). "Algorithm AS 159: An Efficient Method of Generating Random R × C Tables with Given Row and Column Totals." *Applied Statistics*, **30**(1), 91–7.

R Core Team (2021). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org.

Ribeiro Jr P, Diggle P (2001). "geoR: a package for geostatistical analysis." *R-NEWS*, **1**(2), 15–18. ISSN 1609-3631. URL http://cran.R-project.org/doc/Rnews.

Robert CP, Casella G (2004). "Random variable generation." In *Monte Carlo Statistical Methods*, pp. 35–77. Springer.

Rupp K, Tillet P, Rudolf F, Weinbub J, Morhammer A, Grasser T, Jungel A, Selberherr S (2016). "ViennaCL—linear algebra library for multi-and many-core architectures." *SIAM Journal on Scientific Computing*, **38**(5), S412–S439.

Schlather M, Malinowski A, Menck PJ, Oesting M, Strokorb K (2015). "Analysis, Simulation and Prediction of Multivariate Random Fields with Package RandomFields." *Journal of Statistical Software*, **63**(8), 1–25. URL http://www.jstatsoft.org/v63/i08/.

Sevcikova H, Rossini T, L'Ecuyer P (2015). "Package 'rlecuyer'." URL https://cran.r-project.org/web/packages/rlecuyer/index.html.

Zhao P (2016). "R with Parallel Computing from User Perspectives." URL https://parallelr.com/2016/09/10/r-with-parallel-computing/.

Affiliation:

Ruoyong Xu, Patrick Brown
Department of Statistics
University of Toronto
700 University Ave., Toronto, ON M5G 1Z5, Canada
E-mail: ruoyong.xu@mail.utoronto.ca, patrick.brown@utoronto.ca

Pierre L'Ecuyer
Department of Computer Science and Operations Research
University of Montréal
Pavillon André-Aisenstadt, 2920 chemin de la Tour, Montréal, QC, Canada, H3T 1J4
E-mail: lecuyer@iro.umontreal.ca