rTRNG: Advanced Parallel Random Number Generation in R

Riccardo Porreca
Introduction and Motivation

Monte Carlo simulations

simulated variables

b

5

X
Introduction and Motivation

Monte Carlo simulations

simulated variables

5

parallel simulation

© Mirai Solutions GmbH
Consistency with **full sequential simulation**: simulating only $S$, how can we keep $X$ same as the original $\{5,b\}$?
Introduction and Motivation

Limitation: conventional (Pseudo)RNGs based on deterministic recurrence are intrinsically sequential $r_i = f(r_{i-1}, r_{i-2}, \ldots, r_{i-k})$

- Key principles with parallel RNG
  - independent, non-overlapping streams
  - *fair-playing* – results independent of architecture, parallelization techniques, number of parallel processes
    $\Rightarrow$ no random seeding and individual RNGs per process

- Avoid inefficient *naïve* approaches
  - simulate full sequence and discard draws
  - storing relevant seeds

- Available approaches in R
  - parallel, rstream, rlecuyer
  - focus on independent sub-streams

devtools::install_github("miraisolutions/rTRNG", build_vignettes = TRUE)

Based on Tina’s Random Number Generator library by Heiko Bauke

“State of the art C++ pseudo-random number generator library for sequential and parallel Monte Carlo simulations”

http://numbercrunch.de/trng
https://github.com/rabauke/trng4

- collection of random number engines (PRNGs) and distributions
  - linear congruential, multiple recurrence, YARN, lagged Fibonacci, Mersenne-Twister
  - uniform, (truncated) normal, (two-sided) exponential, maxwell, cauchy, logistic, lognormal, pareto, power-law, tent, weibull, extreme value, gamma, beta, chi2, student-t, snedecor-F, rayleigh, bernoulli, (negative) binomial, hypergeometric, geometric, poisson, discrete

- compliant with ISO C++ standard for PRNGs and C++ STL

Package rTRNG

- usage of distributions and engines exposed to R
- C++ library and headers available to other R projects using C++
rTRNG: Distributions and Engines

- Drawing from `r<dist>_trng(..., engine, parallelGrain)`
  - `runif_trng`, `rnorm_trng` (more to come)

- Engines: exposed as Reference Classes via Rcpp Modules
  - Conventional RNGs: `lagfib(2/4)(plus/xor)_19937_64`
    `mt19937(_64)`
  - Parallel RNGs: `lcg64(_shift)`
    `mrg2, mrg3(s), mrg4, mrg5(s)`
    `yarn2, yarn3(s), yarn4, yarn5(s)`
rTRNG: Distributions and Engines

- Drawing from distributions: `r<dist>_trng(..., engine, parallelGrain)`
  - `runif_trng`, `rnorm_trng` (more to come)
- Engines: exposed as Reference Classes via Rcpp Modules
  - Conventional RNGs: `lagfib(2/4)(plus/xor)_19937_64` `mt19937(_64)`
  - Parallel RNGs: `lcg64(_shift)`
    `mrg2, mrg3(s), mrg4, mrg5(s)`
    `yarn2, yarn3(s), yarn4, yarn5(s)`
- based on linear recurrences (linear feedback shift register)
  \[ r_i = a_1r_{i-1} + a_2r_{i-2} + \ldots + a_n r_{i-n} \mod m \]
- strong theoretical foundation about statistical properties (pseudo-noise) and transformations
- simple mathematical structure => manipulation of RNG streams
Set / Create / Manipulate Engines

Base-R-like usage: select and manipulate a global engine
help(TRNG.Random)

  TRNGkind(kind)
  TRNGseed(seed)
  TRNG.Random.seed()
  TRNGjump(steps)
  TRNGsplit(p, s)

Used as default engine by
r<dist>_trng

Create and manipulate individual reference engine objects
help(TRNG.Engine)

  $new(), $new(seed), $new(string)
  $kind(), $name()
  $seed(seed)
  $.Random.seed()
  $jump(steps)
  $split(p, s)
  $toString()
  $copy()
  $show()
Conventional RNG Usage

Base-R-like usage: select and manipulate a **global engine**

```r
example(TRNG.Random)

# set a specific TRNG kind
TRNGkind("yarn2")
# seed the current engine
TRNGseed(12358)
# draw 10 random variates
runif_trng(10)

# full engine specification
engspec <- TRNG.Random.seed()
# [...]  
# restore the engine
TRNG.Random.seed(engspec)
```

Create and manipulate individual reference **engine objects**

```r
element(1)

# create a reference object
rng <- yarn$new()
# seed
rng$seed(12358)  # yarn$new(12358)
# draw from distr. using the engine
runif_trng(10, engine = rng)

# engine state representation
state <- rng$toString()
engspec <- rng$Random.seed()
# [...]  
# restore as (global) engine
rng <- yarn$new(state)
TRNG.Random.seed(engspec)

# reference vs. copy
rng_ref <- rng
rng_cpy <- rng$copy()
```
Advanced RNG Manipulation: jump(steps)

- **Advance** the internal state of the RNG by steps without generating all intermediate states

\[
\begin{align*}
  r_i &= a_1 r_{i-1} + a_2 r_{i-2} + \ldots + a_n r_{i-n} \mod m
\end{align*}
\]

- For LFSR sequences, achieved in \(O(n^3 \ln(\text{steps}))\)
**Advanced RNG Manipulation: jump(steps)**

- **Advance the internal state** of the RNG by steps without generating all intermediate states

```
$ r_i = a_1 r_{i-1} + a_2 r_{i-2} + \ldots + a_n r_{i-n} \mod m$
```

- **For LFSR sequences**, achieved in $O(n^3 \ln(\text{steps}))$

```
rng <- yarn2$new(12358)
runif_trng(15, engine = rng)
## [1] 0.5803 0.3394 0.2214 0.3694 0.5427
## [6] 0.0029 0.1340 0.3468 0.1218 0.9471
## [11] 0.3365 0.1289 0.3804 0.5507 0.4360
rng$seed(12358)
rng$jump(11); runif_trng(4, engine = rng)
## [1] 0.1289 0.3804 0.5507 0.4360
```
Advanced RNG Manipulation: split(p, s)

- Generate directly the $s$-th of $p$ decimated subsequences

- New RNG computed in polynomial time by calibrating the internal parameters => subsequence generated directly (no generation-time complexity)
Advanced RNG Manipulation: split(p, s)

- Generate **directly** the s-th of p decimated subsequences

  - New RNG computed in **polynomial time** by calibrating the internal parameters => subsequence generated directly (no generation-time complexity)

```r
TRNGkind("yarn2"); TRNGseed(12358)
runif_trng(15)
## [1] 0.5803 0.3394 0.2214 0.3694 0.5427
## [6] 0.0029 0.1240 0.3468 0.1218 0.9471
## [11] 0.3365 0.1289 0.3804 0.5507 0.4360
TRNGseed(12358)
TRNGsplit(5, 4); runif_trng(3)
## [1] 0.3694 0.1218 0.5507
```
rTRNG: R/C++ Projects

- **TRNG C++ library** and **headers** available in **C++ code** within other **R projects**

- **Full power** and **flexibility** for implementing high-performance parallel simulation / Monte Carlo algorithms

- Standalone C++ “scripts” sourced via `Rcpp::sourceCpp`
  
  ```
  // [[Rcpp::depends(rTRNG)]]
  ```

- **R packages** importing **rTRNG**
  
  DESCRIPTION
  
  Imports: rTRNG
  
  LinkingTo: rTRNG

  NAMESPACE
  
  importFrom(rTRNG, TRNG.Version)
  
  Makevars(.win)
  
  ?rTRNG::LdFlags
rTRNG: R/C++ Projects

- **TRNG C++ library and headers** available in C++ code within other R projects
- **Full power and flexibility** for implementing high-performance parallel simulation / Monte Carlo algorithms

- Standalone C++ “scripts” sourced via `Rcpp::sourceCpp`
  
  ```cpp
  // [[Rcpp::depends(rTRNG)]]
  #include <Rcpp.h>
  #include <trng/yarn2.hpp>
  #include <trng/uniform_dist.hpp>
  using namespace Rcpp;
  using namespace trng;
  // [[Rcpp::export]]
  NumericVector exampleCpp() {
    yarn2 rng(12358);
    rng.jump(15);
    rng.split(5, 3); // 0-based index
    NumericVector x(3);
    uniform_dist<> unif(0, 1);
    for (int i = 0; i < 3; i++) {
      x[i] = unif(rng);
    }
    return x;
  }
  ```

- **R packages** importing rTRNG

  DESCRIPTION
  - Imports: rTRNG
  - LinkingTo: rTRNG

  NAMESPACE
  - importFrom(rTRNG, TRNG.Version)
  - Makevars(.win)
  - ?rTRNG::LdFlags
rTRNG: R/C++ Projects

- **TRNG C++ library** and **headers** available in **C++ code** within other **R projects**
- **Full power** and **flexibility** for implementing high-performance parallel simulation / Monte Carlo algorithms

```cpp
// [[Rcpp::depends(rTRNG)]]
#include <Rcpp.h>
#include <trng/yarn2.hpp>
#include <trng/uniform_dist.hpp>
using namespace Rcpp;
using namespace trng;

// standalone, consistent sub-simulation
NumericVector exampleCpp() {
  yarn2 rng(12358);
  rng.jump(15);
  rng.split(5, 3); // 0-based index
  NumericVector x(3);
  uniform_dist<> unif(0, 1);
  for (int i = 0; i < 3; i++) {
    x[i] = unif(rng);
  }
  return x;
}
```
Example: Parallel Sub-matrix Simulation

- Monte Carlo simulation of a matrix of i.i.d normal random variables

- Consistent (*fair-playing*), parallel simulation of any *subset* of the variables
  - combine *rTRNG* with *RcppParallel*
  - vignette("mcMat", package = "rTRNG")
Example: Parallel Sub-matrix Simulation

vignette("mcMat", package = "rTRNG")

```cpp
struct MCMatWorker : public Worker {
    RMatrix<double> M;
    const RVector<int> subCols;
    // constructor [omitted]
    // operator processing an exclusive range of row indices
    void operator()(std::size_t begin, std::size_t end) {
        trng::yarn2 r(12358), rj;
        trng::normal_dist<> normal(0.0, 1.0);
        r.jump((int)begin*M.ncol());
        for (IntegerVector::const_iterator jSub = subCols.begin();
             jSub < subCols.end(); jSub++) {
            int j = *(jSub-1); rj = r; rj.split(M.ncol(), j);
            for (int i = (int)begin; i < (int)end; i++) {
                M(i, j) = normal(rj);
            }
        }
    }
};

// [[Rcpp::export]]
NumericMatrix mcMatRcppParallel(const int nrow, const int ncol,
                                 const IntegerVector subCols) {
    NumericMatrix M(nrow, ncol);
    MCMatWorker w(M, subCols); parallelFor(0, M.nrow(), w);
    return M;
}
```
Take-away

- **State-of-the-art** parallel RNGs available to the R community
  - **Experiment/prototype** your parallel algorithm in R
    - **Base-R-like** behavior
    - Manipulation of random **engine objects**
  - Full potential by using TRNG library and headers in R/C++ projects and packages

- **rTRNG** package on our GitHub repo
  - [https://github.com/miraisolutions/rTRNG](https://github.com/miraisolutions/rTRNG)

- **Applied example**: credit default simulation
  - [https://github.com/miraisolutions/PortfolioRiskMC](https://github.com/miraisolutions/PortfolioRiskMC)
  - Presented at **R/Finance 2017** in Chicago