vSMC

Scalable Monte Carlo

Yan Zhou

VERSION 3.0.0

CONTENTS

LIST OF TABLES

1 INTRODUCTION

In this chapter, we introduce the basic structure of Monte Carlo algorithms that can be implemented with this library. At a high level, almost all algorithms can be described as the following, from an implementation perspective. Let $\{E_t\}_{t\ge0}$ be a sequence of state spaces, and $\{X_t^i\}_{i=1}^{N_t} \in E_t^{N_t}$. A Monte Carlo algorithm iteratively applies some kernel $M_t : \prod_{k=0}^{t-1} E_k^{N_k} \to \prod_{k=0}^{t} E_k^{N_k}$, that draws $\{X_k^{1:N_k}\}_{k=1}^{t}$ given $\{X_k^{1:N_k}\}_{k=0}^{t-1}$ according to some probability law.

At first this may seem a very strange way to describe Monte Carlo algorithms. But later it will become clear that, we can implement many always by just defining proper kernels M_t.

2 CORE CONCEPTS

In this chapter, we introduce the core concepts of the library. There are five of them. At the center is a class that represent the state space, that is the value of the states $\{X_t^i\}_{i=1}^{N_t}$. This class may also contain other members that is specific for the model, such as the data. Next is the weights $\{W_t^i\}_{i=1}^{N_t}$. Together with the states, they form a particle system. A sampler operates on a particle system to produce samples iteratively. A sampler might also use some monitors to calculate estimates, $\varphi(\{X_t^i, W_t^i\}_{i=1}^{N_t})$ as it progresses. These concepts are all abstracted in the library. Now we introduce each of them. For brevity, we will drop the time dependent subscript t in the remaining of this chapter.

2.1 STATE

Let E be the state space, a state object, say `T`, shall be able to represent the states collection $\{X^i\}_{i=1}^{N}$ and everything it depends on. Let $d \ge 1$ such that $E \subseteq \mathcal{X}^d$ for some space $\mathcal{X}$. Then, we can represent the states as in an N by d matrix. In this situation, the library provides a default implementation,

```
template <MatrixLayout Layout, std::size_t Dim, typename T>
class StateMatrix;
```

where `Layout` is either `RowMajor` or `ColMajor`, which specifies the matrix layout, `Dim` is the dimension of X^i, i.e., d and `T` shall be a C++ type that represents values in $\mathcal{X}$. For example, if $E \subseteq \mathbb{R}^d$, then one can use the following class,

```
StateMatrix<RowMajor, d, double> state(N);
```

We shall note that, this is not a general purpose matrix class for use such as linear algebra. It is more of a container with a few additional methods. Below is some common operations can be done on `state`,

```
state.size();          // Get the sample size
state.dim();           // Get the dimension
state.resize(N);       // Set a new sample size
state.reserve(N);      // Reserve space for a new sample size
state.shrink_to_fit(); // Release memory no longer needed
state(i, j);           // The element at row i, column j
```

If the template parameter `Dim` is equal to zero, then it is assumed that the dimension is dynamic and may change later. In this case, there are additional methods,

```
StateMatrix<RowMajor, Dynamic, double> state(N, d);
state.resize(N, d);  // Set new sample size and dimension
state.resize_dim(d); // Set a new dimension without chaning the sample size
state.reserve(N, d); // Reserve space for new sample size and dimension
```

The enumerator `Dynamic` above has the value zero. Note that, these methods are only available when the template parameter `Dim` is zero. Attempting to call these methods when it is positive will result in compile-time errors.

There are two more methods, whose purpose will become clear later. The first is,

```
void duplicate(size_type src, size_type dst);
```

Let the value of `src` and `dst` be i and j, respectively. This method set the value of X^j to that of X^i. In other words, X^i is duplicated while X^j is eliminated. The other method is,

```
template <typename IntType, typename InputIter>
void select(IntType N, InputIter index);
```

where `index` is an N-vector, say $\{a_i\}_{i=1}^N$. This method select samples to form a new collection $\{\hat{X}^i\}_{i=1}^N$ such that $\hat{X}^i = X^{a_i}$. Note that, the sample size N does not have to be the same as the original. This is closely related to the selection step of SMC algorithms.

It is also possible to get pointers to the raw data,

```
state.data(); // Equivalent to &state(0, 0)
```

If `Layout` is `RowMajor`, then one can get pointers to the beginning of each X^i,

```
state.row_data(i); // Equivalent to &state(i, 0)
```

If `Layout` is `ColMajor`, then one can get pointers to the beginning of each dimension of X^i,

```
state.col_data(j); // Equivalent to &state(0, j)
```

They are necessary for easy interfacing with many numerical libraries, but the user shall be careful when using them.

2.2 WEIGHT

The weights $\{W^i\}_{i=1}^N$ is abstracted by the class `Weight`. For example,

```
Weight w(N);
w.size();          // Get the sample size
w.resize(N);       // Set a new sample size
w.reserve(N);      // Reserve space for a new sample size
w.shrink_to_fit(); // Release memory no longer needed.
w.data();          // Get a pointer to the beginning of the weight
```

One property of `Weight` is that, $\{W^i\}_{i=1}$ is always normalized. For example, after the construction or resizing, the weights are set to equal, i.e., $W^i = 1/N$, for $i = 1, \dots, N$. One can manually set the weights to equal by

```
w.set_equal();
```

The weights can be manipulated in variance way. Let v be an iterator pointing to an N-vector $\{v^i\}_{i=1}^N$. To set $W_i \propto v_i$,

```
w.set(v);
```

To set $\log W^i = v^i + \text{const.}$, call

```
w.set_log(v);
```

To set weights incrementally, call

```
w.mul(v);
w.add_log(v);
```

which sets $W^i \propto W^i v^i$ and $\log W^i = \log W^i + v^i + \text{const.}$, respectively. The value of ESS of the weights can be obtained by,

```
w.ess();
```

One may also draw an integer $1 \le k \le N$ according to the weights by calling,

```
w.draw(rng);
```

where `rng` is a C++11 RNG engine object. Last, one can obtain a pointer to the raw data,

```
w.data();
```

Again, this shall be used with care. Unlike `StateMatrix`, the member `data` always returns a pointer of type `double`. In other words, the `Weight` does not provide any means for user to change an individual W^i without changing the others. Conceptually, it is the relative weights that matter. And changing one of them is in fact changing $\{W^i\}_{i=1}^N$ as a whole.

2.3 PARTICLE

A particle system, abstracted by the class template,

```
template <typename T>
class Particle;
```

is essentially formed by three parts. The first is an object of type `T`, that abstract the states $\{X^i\}_{i=1}^N$. The second is a type `Weight` object that abstracts the weights $\{W^i\}_{i=1}^N$. And the third is a collection of C++11 RNG engines. There are some restrictions on the type `T`. The constructor of `Particle` is as the following,

```
template <typename... Args>
explicit Particle(size_type N, Args &&... args)
```

where the first argument is the sample size. This and all other arguments, are passed down to the constructor of type `T`. For example,

```
using T = StateMatrix<RowMajor, Dynamic, double>;
Particle particle(N, d);
```

will construct the `StateMatrix` object with `N` and `d` arguments. Therefore, `T` mush has an constructor that accepts an integer value as its first argument. The possible arguments of the constructor of `Particle` is thus the same as those of `T`. Second, `T` has to provide a `select` method similar to that of `StateMatrix`. The library does not really impose any restriction on the internal structure of `T`. And thus it cannot perform the selection by itself. However, for more complicated case, one can always define a class, say `StateType` to represent the space E, and use a one dimension `StateMatrix` as the type `T`. For example,

```
class StateType; // User defined type
using T = StateMatrix<RowMajor, 1, StateType>;
Particle<T> particle(N);
```

More usefully, one can create a new type by deriving from `StateMatrix`. Note that, though the `select` method of `StateMatrix` is written as a function template that can accept any integers and iterators as its argument, for a user defined type `T` only need to support the following signature,

```
ReturnType select(size_type N, const size_type *index);
```

where `size_type` is `T::size_type` if `T` has such a member type, and `std::size_t` otherwise.

The `Weight` type object is constructed with a single argument, N. To retrieve references to the type `T` and `Weight` objects, one can call,

```
particle.state();
particle.weight();
```

respectively. Last but not least, the `Particle` class also contains a collection of RNG engines. The method,

```
particle.rng(i);
```

returns a reference to an RNG engine, specific to the i^{th} particle. For $i \neq j$, if the following,

```
auto &rng1 = particle.rng(i);
auto &rng2 = particle.rng(j);
```

are called from two different threads, then `rng1` and `rng2` will be instances of two independent RNG engines. The details are in section 5.5. The `Particle` class also contains an RNG engine independent of any particles,

```
auto &rng = particle.rng();
```

We will revisit this topic later when we discuss multi-threaded implementations in the next chapter.

2.3.1 *Resize the particle system*

The sample size of can be obtained by,

```
particle.size();
```

and it can also be changed. However, in Monte Carlo algorithms, one does not change the sample size without arbitrarily. Which sample to preserve and possibly duplicated and which sample to be eliminated, has to be done according to some algorithms that produce desirable effects. There are a few methods to resize a particle system. There share two common properties. They take the new sample size, say N, as their first argument. The other is that in the end, the call the `select` method on the type `T` object, with N and an iterator, say `index`, that points to a N-vector $\{a_i\}_{i=1}^N$ as arguments. In addition they all call the `resize` method on the type `Weight` object with N as the argument. How the type `T` handles the call to the `select` method is up to the user. But usually it should behave similarly to that of `StateMatrix`. Below is descriptions of each method for resizing a particle system. They differ in how they generate the index vector $\{a_i\}_{i=1}^N$. We will let M denote the original sample size. Also, for clarity, in the mathematical description of the vectors, we are using indices starting with 1, while in the actual C++ program, the indices starts with zero, as usual.

Resize with given index vectors

```
template <typename InputIter>
void resize_by_index(size_type N, InputIter index);
```

This method take the index vector as its input and pass it directly to the `select` method of `T`. If `Index` is not convertible to a pointer of type `size_type`, then the index vector will be first copied to a temporary and a pointer to it will be passed. Recall that, `T` is not required to handle all types of iterators.

Resize with resampling algorithms

```
template <typename ResampleType>
void resize_by_resample(size_type N, ResampleType &&op)
```

This method generate the index vector according to a resampling algorithm. A resampling algorithm produce the number of replications of each particle in the original system. The function `op` shall accept a call as the following,

```
op(M, N, rng, w, rep);
```

where M and N are the original and new sample size, `rng` is a reference to an RNG engine, `w` is a pointer of type `double`, that points to the M-vector of normalized weights. And last, `r` is a pointer of type `size_type` that points to a M-vector of the number of duplicates of each particles, say

$\{r_i\}_{i=1}^M$. It is required that, the results shall satisfy $r_i \ge 0$ for $i = 1, \dots, M$ and $\sum_{i=1}^M r_i = N$. The index vector is generated such that,

$$\begin{aligned} a_i &= i && \text{if } r_i > 0, \text{ for } i = 1, \dots, \min\{M, N\} \\ \sum_{i=1}^N \mathbb{I}_{\{j\}}(a_i) &= r_j && \text{for } j = 1, \dots, M \end{aligned}$$

Resize with uniform selection

```
void resize_by_uniform(size_type N);
```

The index vector is generated such that, $\mathbb{P}(a_i = j) = 1/M$, for $i = 1, \dots, N$, $j = 1, \dots, M$. This is equivalent to multinomial resampling with equal weights.

2.3.2 *Clone the particle system*

The `Particle<T>` class has the usual special members, such as the copy constructor, assignment operator, etc. They work just as usual. For example,

```
auto new_particle = particle;
```

create a new particle system as an exact duplicate of the original. However, this "exactness" is often undesired. The duplicated particle system will have exactly the same states of RNG engines as the original. And therefore, any random samples generated from this new system, will be exactly the same as the original. This is hardly the desired effects in algorithms where duplicating a particle system into multiple copies is required. In this situation, one can use the `clone` method,

```
auto new_particle = particle.clone();
```

In contrast to the copy constructor, this will create a new particle system exactly the same as the original, except that all RNG engines within the new system is re-seeded.

2.4 SAMPLER

A sampler is formed by the particle system together with all the operations on it. It is abstracted by the class template,

```
template <typename T>
class Sampler;
```

The template parameter `T` is same as that of `Particle`. Its constructor takes arbitrary arguments,

```
template <typename... Args>
explicit Sampler(Args &&... args)
```

and all arguments are passed down to the constructor of `Particle`. For example,

```
using T = StateMatrix<RowMajor, Dynamic, double>;
Sampler<T> sampler(N, d);
```

There two main types of methods for this class. The first is to configure a sampler by adding operations on the particles systems to it. And the second is iterating the sampler. Any callable objects that is convertible to the following can be used as operations on the particle system.

```
using eval_type = std::function<std::size_t(std::size_t, Particle<T> &)>;
```

For example,

```
std::size_t eval(std::size_t iter, Particle<T> &particle);
```

Such a function can be attached to the sampler by the `eval` method,

```
Sampler<T> &eval(
    const eval_type &new_eval, SamplerStage stage, bool append = true);
```

For example,

```
sampler.eval(eval, SamplerInit | SamplerMove, true);
```

The first argument is the evaluation object. The sampler maintains a sequence of evaluation steps. If the third argument is true, then the new evaluation object will be appended to the existing sequence. Otherwise, the sequence will be cleared first. The second argument will be explained shortly. The sampler can also have an optional resampling step. This can be attached to the sampler by the following method,

```
Sampler<T> &resample_method(
    ResampleScheme scheme, double threshold = resample_threshold_always());
```

which uses a builtin resampling scheme of the library (see section 6.1). Or alternatively,

```
Sampler<T> &resample_method(const eval_type &res_eval,
    double threshold = resample_threshold_always());
```

In either case, the parameter `threshold` specifies the condition under which the resampling step will be actually performed. Let its value be α, resampling is performed if and only if ESS $< \alpha N$.

2.4.1 *Evaluation stages*

The sampler is iterated with two methods,

```
sampler.initialize();
sampler.iterate(n);
```

The first initialize the sampler, and the second iterate the sampler n times. In each case, the sequence of user defined evaluation objects will be called, such as the `eval` function defined earlier. The first argument passed to it is the iteration number. This number is reset to zero when `initialize` is called and incremented by one each time the sampler is iterated. The initialization is divided into the following stages,

`SamplerInit` Initialize the particle system $\{X_0^i, W_0^i\}_{i=1}^{N_0}$.

Resampling If ESS $< \alpha N$, resampling the particle system to $\{\hat{X}_0^i, \hat{W}_0^i\}_{i=1}^{\hat{N}_0}$.

`SamplerMCMC` Mutate the particle system to $\{\check{X}_0^i, \check{W}_0^i\}_{i=1}^{\check{N}_0}$.

and the iteration at step t, is divided into the following stages,

`SamplerMove` Move the particle system from $\{\check{X}_{t-1}, \check{W}_{t-1}\}_{i=1}^{\check{N}_{t-1}}$ to $\{X_t^i, W_t^i\}_{i=1}^{N_t}$.

Resampling If ESS $< \alpha N$, resampling the particle system to $\{\hat{X}_t^i, \hat{W}_t^i\}_{i=1}^{\hat{N}_t}$.

`SamplerMCMC` Mutate the particle system to $\{\check{X}_t^i, \check{W}_t^i\}_{\check{N}_t}$.

In each of these stages, one or more `eval_type` objects are called upon to perform the necessary sampling and calculation. In the *Resampling* stage, the evaluation object set by `resmaple_method` will be used to used to perform the resampling. The builtin algorithms will use the `select` method of the state type `T` to perform the selection after generating the index vector. In all other three types of stages, all objects in the sequence of evaluations objects set by the `eval` method of `Sampler` is checked. Recall that, the each evaluation object is attached to the sampler by a call to the following method,

```
Sampler<T> &eval(
    const eval_type &new_eval, SamplerStage stage, bool append = true);
```

At each stage, the value of the second parameter will be check. For example, at the `SamplerInit` stage, this object will be called if and only if the value of `SamplerInit & stage` is non-zero. The other two stages, `SamplerMove` and `SamplerMCMC` is similar. The last one is so named because in many algorithms, the mutation step is performed by some MCMC kernel. It is not universally so. Clearly, the evaluation objects at that stage, if any, do not have to perform MCMC type mutations.

2.5 MONITOR

Let $\varphi_t(\{X_t^i, W_t^i\}_{i=1}^{N_t})$ be some test function with values in $\mathbb{R}^d$. It is often of interest to monitor its value as the algorithm progresses. In the library, this is done through the class `Monitor<T>`. It has the following constructor,

```
Monitor(std::size_t dim, const eval_type &eval, bool record_only = false,
    MonitorStage stage = MonitorMCMC);
```

The first parameter `dim` is the dimension of φ_t, d. The second is a user defined callback function. More specifically,

```
using eval_type =
    std::function<void(std::size_t, std::size_t, Particle<T> &, double *)>;
```

For example,

```
void varphi(std::size_t t, std::size_t d, Particle<T> &particle, double *r);
```

When this function is called. The first argument passed to it is t, the iteration number. The second is d, the dimension. The third is a reference to the particle system at iteration t. And the last is a pointer to a vector for output.

The third parameter of the constructor of `Monitor`, `record_only` determines how shall the function above behave. If it is true, then the function `varphi` shall return the value of φ_t directly, and the output parameter `r` points to a d-vector. In this case, `Monitor` merely record the values of φ_t at each iteration. On the other hand, if `record_only` is false, then it is assumed that φ_t takes the following form,

$$\varphi_t(\{X_t^i, W_t^i\}_{i=1}^{N_t}) = \sum_{i=1}^{N_t} W_t^i \varphi_t^i(X_t^i).$$

And the output parameter `r` is an N by d row major matrix, with the d-vector value $\varphi_t^i(X_t^i)$ written into each row of this matrix. And each time `varphi` is called, `Monitor` will compute and store the result of the summation above.

The last parameter of the constructor of `Monitor` specifies when shall the calculation of φ_t be carried out. Possible values are,

`MonitorMove` The evaluation happens after the `SamplerInit` or `SamplerMove` stage, i.e., right before the possible resampling.

`MonitorResample` The evaluation happens right after the *Resampling* stage.

`MonitorMCMC` The evaluation happens after the `SamplerMCMC` stage, i.e., at the end of the initialization or iteration.

A monitor can be attached to a sampler. For example,

```
Monitor<T> mon(d, eval, false, MonitorMCMC);
sampler.monitor("name", mon);
```

Later, this monitor can be retrieved by,

```
const auto &mon = sampler.monitor("name");
```

One can also detach the monitor from the sampler by,

```
sampler.monitor_clear("name");
```

One can retrieve the results in various ways. Every time a monitor being evaluated, it record two values. The first is the iteration number t, at which it was evaluated. The other is the value of φ_t. One can retrieve these values using the following methods,

```
mon.iter_size();  // Total number of evaluations
mon.index(j);     // The iteration number of the j-th evaluation
mon.record(i, j); // The value of the i-th component of the result
                  // at the j-th evaluation
```

Note that, the monitor does not have to be attached to a sampler before it starting initialization. It can also be detached before the sampler finishing all iterations. Therefore, the iteration numbers of a monitor being evaluated are not necessarily the sequence $0, 1, \dots, n$, where n is the total number of iterations.

2.6 EXAMPLE

3 SYMMETRIC MULTIPROCESSING

4 ELEMENTARY MATHEMATICAL FUNCTIONS

4.1 CONSTANTS

The library defines some mathematical constants in the form of constant expression functions. For example, to get the value of π with a desired precision, one can use the following,

```
constexpr float pi_f = const_pi<float>();
constexpr double pi_d = const_pi<double>();
constexpr long double pi_l = const_pi<long double>();
```

The compiler will evaluate these values at compile-time and thus there is no performance difference from hard-coding the constants in the program, while the readability is improved. All defined constants are listed in Table 4.1.

4.2 VECTORIZED FUNCTIONS

The library provides a set of vectorized elementary mathematical functions. For example, to perform additions of two vectors,

```
std::size_t n = 1000;
Vector<double> a(n), b(n), y(n);
// Fill vectors a and b
add(n, a.data(), b.data(), y.data());
```

This is equivalent to

```
for (std::size_t i = 0; i != n; ++i)
    y[i] = a[i] + b[i];
```

The functions defined are listed in Tables 4.2 to 4.8. For each function, the first parameter is always the length of the vector, and the last is a pointer to the output vector (except for `sincos` and `modf`, which have two output parameters). For all functions, the output is always a vector. If there are more than one input parameters, then some of them, but not all, can be scalars. For example, for the function call `fma(n, a, b, c, y)` in Table 4.2, the input parameters are `a`, `b`, and `c`. Some of them, not all, can be scalars instead of pointers to vectors. The output parameter `y` has to be a pointer to a vector. Therefore, there are seven versions of this function for each type of the value. Note that, mixed precision is not allowed. For example,

```
Vector<double> a(n);
Vector<double> b(n);
Vector<double> y(n);
fma(n, a.data(), b.data(), 2, y.data());
```

will cause compile-time error because the third argument is of type `int` while the others are of `double` precision. The correct call shall be,

```
fma(n, a.data(), b.data(), 2.0, y.data());
```

Without any third-party libraries, these functions do not provide performance gain compared to the simple loop. When MKL or Apple Accelerate framework is present, some functions can have substantial performance improvement when all input arguments are vectors. The performance of vectorized random number generating introduced later in section 5.1 heavily depends on these functions.

Function	Value	Function	Value
`const_pi`	π	`const_pi_2`	2π
`const_pi_inv`	$1/\pi$	`const_pi_sqr`	π^2
`const_pi_by2`	$\pi/2$	`const_pi_by3`	$\pi/3$
`const_pi_by4`	$\pi/4$	`const_pi_by6`	$\pi/6$
`const_pi_2by3`	$2\pi/3$	`const_pi_3by4`	$3\pi/4$
`const_pi_4by3`	$4\pi/3$	`const_sqrt_pi`	$\sqrt{\pi}$
`const_sqrt_pi_2`	$\sqrt{2\pi}$	`const_sqrt_pi_inv`	$\sqrt{1/\pi}$
`const_sqrt_pi_by2`	$\sqrt{\pi/2}$	`const_sqrt_pi_by3`	$\sqrt{\pi/3}$
`const_sqrt_pi_by4`	$\sqrt{\pi/4}$	`const_sqrt_pi_by6`	$\sqrt{\pi/6}$
`const_sqrt_pi_2by3`	$\sqrt{2\pi/3}$	`const_sqrt_pi_3by4`	$\sqrt{3\pi/4}$
`const_sqrt_pi_4by3`	$\sqrt{4\pi/3}$	`const_ln_pi`	$\ln\pi$
`const_ln_pi_2`	$\ln 2\pi$	`const_ln_pi_inv`	$\ln 1/\pi$
`const_ln_pi_by2`	$\ln\pi/2$	`const_ln_pi_by3`	$\ln\pi/3$
`const_ln_pi_by4`	$\ln\pi/4$	`const_ln_pi_by6`	$\ln\pi/6$
`const_ln_pi_2by3`	$\ln 2\pi/3$	`const_ln_pi_3by4`	$\ln 3\pi/4$
`const_ln_pi_4by3`	$\ln 4\pi/3$	`const_e`	e
`const_e_inv`	$1/\mathrm{e}$	`const_sqrt_e`	$\sqrt{\mathrm{e}}$
`const_sqrt_e_inv`	$\sqrt{1/\mathrm{e}}$	`const_sqrt_2`	$\sqrt{2}$
`const_sqrt_3`	$\sqrt{3}$	`const_sqrt_5`	$\sqrt{5}$
`const_sqrt_10`	$\sqrt{10}$	`const_sqrt_1by2`	$\sqrt{1/2}$
`const_sqrt_1by3`	$\sqrt{1/3}$	`const_sqrt_1by5`	$\sqrt{1/5}$
`const_sqrt_1by10`	$\sqrt{1/10}$	`const_ln_2`	$\ln 2$
`const_ln_3`	$\ln 3$	`const_ln_5`	$\ln 5$
`const_ln_10`	$\ln 10$	`const_ln_inv_2`	$1/\ln 2$
`const_ln_inv_3`	$1/\ln 3$	`const_ln_inv_5`	$1/\ln 5$
`const_ln_inv_10`	$1/\ln 10$	`const_ln_ln_2`	$\ln\ln 2$

TABLE 4.1 Mathematical constants

Function	Operation
`add(n, a, b, y)`	$y_i = a_i + b_i$
`sub(n, a, b, y)`	$y_i = a_i - b_i$
`sqr(n, a, y)`	$y_i = a_i^2$
`mul(n, a, b, y)`	$y_i = a_i b_i$
`abs(n, a, y)`	$y_i = \|a_i\|$
`fma(n, a, b, c, y)`	$y_i = a_i b_i + c_i$

TABLE 4.2 Arithmetic functions

Function	Operation
`inv(n, a, y)`	$y_i = 1/a_i$
`div(n, a, b, y)`	$y_i = a_i/b_i$
`sqrt(n, a, y)`	$y_i = \sqrt{a_i}$
`invsqrt(n, a, y)`	$y_i = 1/\sqrt{a_i}$
`cbrt(n, a, y)`	$y_i = \sqrt[3]{a_i}$
`invcbrt(n, a, y)`	$y_i = 1/\sqrt[3]{a_i}$
`pow2o3(n, a, y)`	$y_i = a_i^{2/3}$
`pow3o2(n, a, y)`	$y_i = a_i^{3/2}$
`pow(n, a, b, y)`	$y_i = a_i^{b_i}$
`hypot(n, a, b, y)`	$y_i = \sqrt{a_i^2 + b_i^2}$

TABLE 4.3 Power and root functions

Function	Operation
`exp(n, a, y)`	$y_i = \mathrm{e}^{a_i}$
`exp2(n, a, y)`	$y_i = 2^{a_i}$
`exp10(n, a, y)`	$y_i = 10^{a_i}$
`expm1(n, a, y)`	$y_i = \mathrm{e}^{a_i} - 1$
`log(n, a, y)`	$y_i = \ln a_i$
`log2(n, a, y)`	$y_i = \log_2 a_i$
`log10(n, a, y)`	$y_i = \log_{10} a_i$
`log1p(n, a, y)`	$y_i = \ln(a_i + 1)$

TABLE 4.4 Exponential and logarithm functions

Function	Operation
`cos(n, a, y)`	$y_i = \cos(a_i)$
`sin(n, a, y)`	$y_i = \sin(a_i)$
`sincos(n, a, y, z)`	$y_i = \sin(a_i), z_i = \cos(a_i)$
`tan(n, a, y)`	$y_i = \tan(a_i)$
`acos(n, a, y)`	$y_i = \arccos(a_i)$
`asin(n, a, y)`	$y_i = \arcsin(a_i)$
`atan(n, a, y)`	$y_i = \arctan(a_i)$
`acos(n, a, y)`	$y_i = \arccos(a_i)$
`atan2(n, a, y)`	$y_i = \arctan(a_i/b_i)$

TABLE 4.5 Trigonometric functions

Function	Operation
`cosh(n, a, y)`	$y_i = \cosh(a_i)$
`sinh(n, a, y)`	$y_i = \sinh(a_i)$
`tanh(n, a, y)`	$y_i = \tanh(a_i)$
`acosh(n, a, y)`	$y_i = \operatorname{arc}\cosh(a_i)$
`asinh(n, a, y)`	$y_i = \operatorname{arc}\sinh(a_i)$
`atanh(n, a, y)`	$y_i = \operatorname{arc}\tanh(a_i)$

Table 4.6 Hyperbolic functions

Function	Operation
`erf(n, a, y)`	$y_i = \operatorname{erf}(a_i)$
`erfc(n, a, y)`	$y_i = \operatorname{erfc}(a_i)$
`cdfnorm(n, a, y)`	$y_i = 1 - \operatorname{erfc}(a_i/\sqrt{2})/2$
`lgamma(n, a, y)`	$y_i = \ln \Gamma(a_i)$
`tgamma(n, a, y)`	$y_i = \Gamma(a_i)$

Table 4.7 Special functions

Function	Operation
`floor(n, a, y)`	$y_i = \lfloor a_i \rfloor$
`ceil(n, a, y)`	$y_i = \lceil a_i \rceil$
`trunc(n, a, y)`	$y_i = \operatorname{sgn}(a_i)\lfloor \lvert a_i \rvert \rfloor$
`round(n, a, y)`	$y_i =$ nearest integer of a_i
`modf(n, a, y, z)`	$y_i = \operatorname{sign}(a_i)\lfloor \lvert a_i \rvert \rfloor, z_i = \operatorname{sign}(a_i)\lvert a_i - y_i \rvert$

Table 4.8 Rounding functions

5 RANDOM NUMBER GENERATING

The library has a comprehensive RNG system to facilitate implementation of Monte Carlo algorithms. Similar to the standard library header `<random>`, there are mainly two parts of this system. The first is a set of RNG engines that generate random integers. The other is a set of distribution generators. The former are documented in sections 5.2 to 5.4, and the later in section 5.6. Apart from these, the library also provides facilities for vectorized random number generating (section 5.1) and using multiple RNGs in parallel programs (section 5.5).

5.1 VECTORIZED RANDOM NUMBER GENERATING

Before we discuss other features, we first introduce a generic function `rand`, which provides vectorized random number generating. There are two versions. The first operates on RNG engines and generates random integers,

```
template <typename RNGType>
inline void rand(
    RNGType &rng, std::size_t n, typename RNGType::result_type *r);
```

The effect of the function call,

```
rand(rng, n, r);
```

is equivalent to the loop,

```
for (std::size_t i = 0; i != n; ++i)
    r[i] = rng();
```

The results will always be the same unless a non-deterministic RNG is used. For some RNGs implemented in the library, the vectorized version may have considerable performance advantage.

The second version of `rand` is for generating distribution random numbers,

```
template <typename RNGType, typename DistributionType>
inline void rand(RNGType &rng, const DistributionType &distribution,
    std::size_t n, typename DistributionType::result_type *r);
```

For example,

```
NormalDistribution<double> normal;
rand(rng, normal, n, r);
```

This is similar to the following loop,

```
for (std::size_t i = 0; i != n; ++i)
    r[i] = normal(rng);
```

Depending on the type of `rng` and the distribution (including its parameters), the vectorized version may have superior performance. However, the results will not be exactly the same as using a loop.

5.1.1 *Performance measurement*

All performance results shown later in this chapter is measured in single core cycles per bytes (cpB) for RNGs, or cycles per element (cpE) for distributions (`double` precision). The processor used to measure the performance is an Intel Core i7-4960HG CPU. Three compilers are tested. The LLVM clang (version 3.8), the GNU GCC (version 6.1), and the Intel C++ compiler (version 2016 update 3). When multiple compilers are tested, they are labeled "LLVM", "GNU", and "Intel", respectively in tables. If only results from one compiler are shown, then unless stated otherwise, it is the LLVM clang compiler. The operating system is Mac OS X (version 11.1). Depending on the CPU, the compiler, and the operating system, performance may vary considerably. The library does not attempt to optimize for any particular platform. However, given accelerated vectorized mathematical functions (seed section 4.2), for any given platform, the relative performance advantage to alternatives, such as the standard library, is usually substantial.

For RNG engines, we measure the performance of generating random bits instead of the raw output from the engines. All internal usages of RNGs in the library first transfer the raw output to unsigned integers uniform on the set $\{0, \dots, 2^W - 1\}$, $W \in \{32, 64\}$ (see section 5.6.1). Direct use of the raw output from RNG engines is rare in applications.

Two usage cases are considered. The first is the performance of generating random integers one by one,

```
UniformBitsDistribution<std::uint64_t> rbits;
for (std:size_t i = 0; i != n; ++i)
    r[i] = rbits(rng);
```

The second is the vectorized performance,

```
rand(rng, rbits, n, r.data());
```

In both cases, we repeat the simulations 100 times, each time with the number of elements n chosen randomly between 5,000 and 10,000. The total number of cycles of the 100 simulations are recorded, and then divided by the total number of bytes generated. This gives the performance measurement in cpB. This experiment is repeated ten times, and the best results are shown. The two cases are labeled "Loop" and "`rand`", respectively.

For the performance of distributions, we measure four methods of drawing random numbers from the same distribution. The procedures is similar, except now we measure the performance in cpE. First, if the distribution is available in the standard library or the Boost[1] library, we measure the following case,

```
std::normal_distribution<double> rnorm_std(0, 1);
for (std::size_t i = 0; i = n; ++i)
    r[i] = rnorm_std(rng);
```

Second, we measure the performance of the library's implementation,

```
NormalDistribution<double> rnorm_vsmc(0, 1);
for (std::size_t i = 0; i = n; ++i)
    r[i] = rnorm_vsmc(rng);
```

The third is the vectorized performance,

```
rand(rng, rnorm_vsmc, n, r.data());
```

For all the three above, the RNG is `ARSx8` (section 5.2.1). The last is when RNG is `MKL_SFMT19937` (section 5.4),

```
MKL_SFMT19937 rng_mkl;
rand(rng_mkl, rnorm_vsmc, n, r.data());
```

In this case, not only the RNG itself is faster, the distribution might also use MKL routines. The four cases are labeled "STD", "vSMC", "`rand`" and "MKL", respectively.

[1]`http://www.boost.org`

5.2 COUNTER-BASED RNG

The standard library provides a set of RNG engines (performance data in Table C.1). Unfortunately, none of them are suitable for parallel computing without considerable efforts. To illustrate the problem, consider the situation where there are two threads that need to generate random numbers. And thus two RNG engine instances need to be created for each thread. Let the random numbers generated by them be $\{r_i^1\}_{i>0}$ and $\{r_i^2\}_{i>0}$. Because of the deterministic and recursive natural of the these RNGs, there exists $k \in \mathbb{Z}$, such that, $r_i^1 = r_{i+k}^2$ for all $i > \max\{0, -k\}$. If $|k|$ is larger than or close to the total number of random numbers required in each thread, then this situation might not be an issue. However, there is no easy way to ensure such a condition.

There are two standard solutions to this problem. The first is to use sub-streams for each thread. For example,

```
std::mt19937 rng;

// Thread k
std::mt19937 rng_k = rng;
rng_k.discard(n * k);
```

where n is the number of random numbers required by each thread. The k^{th} thread only uses the random numbers in the sub-stream $\{r_i\}_{nk<i\le n(k+1)}$. For this to work, the RNG needs a fast `discard` implementation, preferably with $\mathcal{O}(1)$ cost. In addition, one needs to manage RNGs explicitly for each thread, which prevents this method to be used in environments where threads are created implicitly, such as using TBB for parallelization.

The second solution is to use a leap-frog algorithm, such that each thread will use the elements $\{r_{iK+k}\}_{i>0}$ of the stream, where K is the total number of threads. This is not directly supported in the standard library, but can be emulated,

```
std::mt19937 rng;

// Thread k
std::mt19937 tmp = rng;
tmp.discard(k);
std::discard_block_engine<std::mt19937, K, 1> rng_k(tmp);
```

This not only requires managing the RNGs explicitly, but also knowing the number of threads at compile-time, which prevents it to be used in any applications using dynamic parallelization.

And similar to the first solution, it cannot be used when threads are created implicitly.

The development by Salmon et al. (2011) made high performance parallel RNG much more accessible. The RNGs introduced in the paper use bijection f_k, such that, for a sequence $\{c_i = i\}_{i \ge 0}$, the sequence $\{y_i = f_k(c_i)\}_{i \ge 0}$ appears random. In addition, for $k_1 \ne k_2$, f_{k_1} and f_{k_2} will generate two sequences that appear statistically independent. Compared to more conventional RNGs which use recursions $y_i = f_k(y_{i-1})$, these counter-based RNGs are much easier to setup in a parallelized environment. If c, the counter, is an unsigned integer with b bits, and k, the key, is an unsigned integer with d bits. Then for each k, the RNG has a period 2^b. And there can be at most 2^d independent streams. Another way is to view this kind of RNGs is that, it is one RNG with state $\{c, k\}$ for $0 \le c < 2^b$ and $0 \le k < 2^d$. And thus it has a period 2^{b+d}. And the division of the state into counter and key provides an easy way to setup a large number of sub-streams.

Of course, not any sequence of counters and keys are suitable. The RNGs are still deterministic. Since f_k for any given k is a bijection, the sequence of counter $\{c_i = f_k^{-1}(i)\}_{i \ge 0}$ will of course produce a very regular sequence $\{y_i = i\}_{i \ge 0}$. However, for regular sequences, such as $\{c_i = i\}_{i \ge 0}$ and $\{k_j = j\}_{j \ge 0}$, the resulting sequences $\{\{y_i\}_{i \ge 0}^j\}_{j \ge 0}$ appear random. See Salmon et al. (2011) for more details.

Table 5.1 lists all counter-based RNGs implemented in the library, along with the bits of the counter and the key. They all output 32-bits unsigned integers uniform on the set $\{0, \dots, 2^{32} - 1\}$. For 64-bits output, a suffix `_64` may be appended to the corresponding RNG engine names. For example, `Threefry4x64` and `Threefry4x64_64` both generate the same 256-bits random integers internally. The only difference is that `operator()` of the former returns 32 or those 256 bits each time it is executed, while the later returns 64 bits.

All RNGs in Table 5.1 are actually type aliases. More generally the library defines the following class template as the interface,

```
template <typename ResultType, typename Generator>
class CounterEngine;
```

where `ResultType` shall be an unsigned integer type and `Generator` is the class that actually implements the algorithm. See the reference manual for details of the generator type. For most users, those implemented in the library are sufficient. They are introduced in the next few sections. A few configuration macros of these generators are listed in Table 5.2 and will be referred to later.

Class	Counter bits	Key bits
`AES128x1`, `ARS128x2`, `AES128x4`, `AES128x8`	128	128
`AES192x1`, `ARS192x2`, `AES192x4`, `AES192x8`	128	192
`AES256x1`, `AES256x2`, `AES256x4`, `AES256x8`	128	256
`ARSx1`, `ARSx2`, `ARSx4`, `ARSx8`	128	128
`Philox2x32`	64	32
`Philox2x64`	128	64
`Philox4x32`	128	64
`Philox4x64`	256	128
`Threefry2x32`	64	64
`Threefry2x64`	128	128
`Threefry4x32`	128	128
`Threefry4x64`	256	256
`Threefry8x64`	512	512
`Threefry16x64`	1024	1024

TABLE 5.1 Counter-based RNG

Macro	Default
`VSMC_RNG_AES128_ROUNDS`	`10`
`VSMC_RNG_AES192_ROUNDS`	`12`
`VSMC_RNG_AES256_ROUNDS`	`14`
`VSMC_RNG_ARS_ROUNDS`	`5`
`VSMC_RNG_AES_NI_BLOCKS`	`8`
`VSMC_RNG_PHILOX_ROUNDS`	`10`
`VSMC_RNG_PHILOX_VECTOR_LENGTH`	`4`
`VSMC_RNG_THREEFRY_ROUNDS`	`20`
`VSMC_RNG_THREEFRY_VECTOR_LENGTH`	`4`

TABLE 5.2 Configuration macros for counter-based RNG

5.2.1 *AES-NI instructions based RNG*

The AES-NI[2] instructions based RNGs in Salmon et al. (2011) are implemented in the following generator,

```
template <typename KeySeqType, std::size_t Rounds, std::size_t Blocks>
class AESNIGenerator;
```

The corresponding RNG engine is,

```
template <typename ResultType, typename KeySeqType, std::size_t Rounds,
    std::size_t Blocks>
using AESNIEngine =
    CounterEngine<ResultType, AESNIGenerator<KeySeqType, Rounds, Blocks>>;
```

where `KeySeqType` is the class used to generate the sequences of round keys. The parameter `Rounds` is the number of rounds of AES encryption to be performed. See the reference manual for details of how to define the key sequence class. The AES-NI encryption instructions have a latency of seven or eight cycles, while they can be issued at every cycle. Therefore better performance can be achieved if multiple 128-bits random integers are generated at the same time. This is specified by the template parameter `Blocks`. Larger blocks, up to eight, might improve performance. But this is at the cost of larger state size. Without going into details, there are four types of sequence of round keys implemented by the library,

```
template <std::size_t Rounds>
using AES128KeySeq =
    internal::AESKeySeq<Rounds, internal::AES128KeySeqGenerator>;

template <std::size_t Rounds>
using AES192KeySeq =
    internal::AESKeySeq<Rounds, internal::AES192KeySeqGenerator>;

template <std::size_t Rounds>
using AES256KeySeq =
```

[2]https://en.wikipedia.org/wiki/AES_instruction_set

```
    internal::AESKeySeq<Rounds, internal::AES256KeySeqGenerator>;

template <typename Constants = ARSConstants>
using ARSKeySeq = internal::ARSKeySeqImpl<Constants>;
```

and correspondingly four RNG engines,

```
template <typename ResultType, std::size_t Rounds = VSMC_RNG_AES128_ROUNDS,
    std::size_t Blocks = VSMC_RNG_AES_NI_BLOCKS>
using AES128Engine =
    AESNIEngine<ResultType, AES128KeySeq<Rounds>, Rounds, Blocks>;

template <typename ResultType, std::size_t Rounds = VSMC_RNG_AES192_ROUNDS,
    std::size_t Blocks = VSMC_RNG_AES_NI_BLOCKS>
using AES192Engine =
    AESNIEngine<ResultType, AES192KeySeq<Rounds>, Rounds, Blocks>;

template <typename ResultType, std::size_t Rounds = VSMC_RNG_AES256_ROUNDS,
    std::size_t Blocks = VSMC_RNG_AES_NI_BLOCKS>
using AES256Engine =
    AESNIEngine<ResultType, AES256KeySeq<Rounds>, Rounds, Blocks>;

template <typename ResultType, std::size_t Rounds = VSMC_RNG_ARS_ROUNDS,
    std::size_t Blocks = VSMC_RNG_AES_NI_BLOCKS,
    typename Constants = ARSConstants>
using ARSEngine =
    AESNIEngine<ResultType, ARSKeySeq<Constants>, Rounds, Blocks>;
```

The first three are equivalent to AES-128, AES-192 and AES-256 block ciphers used in counter mode. The last is the ARS algorithm introduced by Salmon et al. (2011). The last template parameter `Constants` of `ARSKeySeq` and `ARSEngine` is a trait class that defines the constants of the Weyl's sequence. See Salmon et al. (2011) for details. The defaults are taken from the paper. To use an alternative pair of 64-bits integers as the constants, one can define and use a trait class as the following,

```
template <std::size_t>
struct NewWeylConstant;
```

```
template<>
struct NewWeylConstant<0>
{
    static constexpr std::uint64_t value = FIRST_CONSTANT;
};

template<>
struct NewWeylConstant<1>
{
    static constexpr std::uint64_t value = SECOND_CONSTANT;
};

struct NewConstants
{
    template <std::size_t I>
    using weyl = NewWeylConstant<I>;
};

using NewARS = ARSEngine<ResultType, Rounds, NewConstants>;
```

Alternative methods are also possible. The only requirement is that, the following statement,

```
template <std::size_t I>
using weyl = typename Constants::template weyl<I>;
```

shall define the type `weyl` such that it has a static constant expression member data `value` that is the I th Weyl constant. A few type aliases are defined for convenience. For example,

```
using ARSx8    = ARSEngine<std::uint32_t, VSMC_RNG_ARS_ROUNDS, 8>;
using ARSx8_64 = ARSEngine<std::uint64_t, VSMC_RNG_ARS_ROUNDS, 8>;
using ARS      = ARSEngine<std::uint32_t>;
using ARS_64   = ARSEngine<std::uint64_t>;
```

The engine `ARS` is the library's default RNG if AES-NI instructions are supported. Aliases for block sizes 1, 2, 4 and 8 are defined for all four algorithms, as well as both 32- and 64-bits output versions. These aliases are listed in Table 5.1. The performance of these engines depends on a few factors, such as CPU types, compilers, operating systems, etc. See Tables C.2 to C.5 for performance data. In any case, the performance is good enough even for the most demanding applications. The library does not attempt to optimize the algorithm for any particular platform. In realistic applications, the performance of RNG is unlikely to become a bottle neck. Note that, the best performance is obtained with the vectorized `rand` function (see section 5.1).

5.2.2 *Philox*

The Philox algorithm in Salmon et al. (2011) is implemented in the following generator,

```
template <typename T, std::size_t K = VSMC_RNG_PHILOX_VECTOR_LENGTH,
    std::size_t Rounds = VSMC_RNG_PHILOX_ROUNDS,
    typename Constants = PhiloxConstants<T, K>>
class PhiloxGenerator;
```

The corresponding RNG engine is,

```
template <typename ResultType, typename T = ResultType,
    std::size_t K = VSMC_RNG_PHILOX_VECTOR_LENGTH,
    std::size_t Rounds = VSMC_RNG_PHILOX_ROUNDS,
    typename Constants = PhiloxConstants<T, K>>
using PhiloxEngine =
    CounterEngine<ResultType, PhiloxGenerator<T, K, Rounds, Constants>>;
```

The default vector length and the number of rounds can be changed by configuration macros listed in Table 5.2. There is no limit on the template parameter `K` or `Rounds`, nor any limitation on `T` except that it has to be an unsigned integer type. See Salmon et al. (2011) on the most general form of the algorithm. However, the library only provides default constants for 32- and 64-bits unsigned integer type `T` and `K` taking the values `2` or `4`. These four engines are defined as type aliases for convenience,

```
template <typename ResultType>
using Philox2x32Engine = PhiloxEngine<ResultType, std::uint32_t, 2>;

template <typename ResultType>
using Philox4x32Engine = PhiloxEngine<ResultType, std::uint32_t, 4>;
```

```
template <typename ResultType>
using Philox2x64Engine = PhiloxEngine<ResultType, std::uint64_t, 2>;

template <typename ResultType>
using Philox4x64Engine = PhiloxEngine<ResultType, std::uint64_t, 4>;
```

Type aliases for 32- and 64-bits `ResultType` are also defined, as listed in Table 5.1. To use the engine with `K` taking values larger than four, or `T` being unsigned integer type with bits other than 32 or 64, one needs to provide a suitable trait class, `Constant`. It is similar to that of `ARSEngine`. See the reference manual of `PhiloxConstants` for an example of how to define it. The performance data is in Table C.6.

5.2.3 *Threefry*

The Threefry algorithm in Salmon et al. (2011) is implemented in the following generator,

```
template <typename T, std::size_t K = VSMC_RNG_THREEFRY_VECTOR_LENGTH,
    std::size_t Rounds = VSMC_RNG_THREEFRY_ROUNDS,
    typename Constants = ThreefryConstants<T, K>>
class ThreefryGenerator;
```

The corresponding RNG engine is,

```
template <typename ResultType, typename T = ResultType,
    std::size_t K = VSMC_RNG_THREEFRY_VECTOR_LENGTH,
    std::size_t Rounds = VSMC_RNG_THREEFRY_ROUNDS,
    typename Constants = ThreefryConstants<T, K>>
using ThreefryEngine =
    CounterEngine<ResultType, ThreefryGenerator<T, K, Rounds, Constants>>;
```

The default vector length and the number of rounds can be changed by configuration macros listed in Table 5.2. Similar to the implementation of the Philox algorithm, there is no limit on the template parameter `K` or `Rounds` as long as a suitable trait class `Constant` is provided. The library provides default constants for 64-bits unsigned integer type `T` and `K` taking the values `4`, `8` and `16`, taken from the skein[3] hash algorithm, for which the Threefish algorithm was originally

[3]`http://www.skein-hash.info`

developed for. Defaults for 32-bits `T` or `K` taking the value `2` are also provided, taken from Salmon et al. (2011). Type aliases for these configurations are defined for convenience,

```
template <typename ResultType>
using Threefry2x32Engine = ThreefryEngine<ResultType, std::uint32_t, 2>;

template <typename ResultType>
using Threefry4x32Engine = ThreefryEngine<ResultType, std::uint32_t, 4>;

template <typename ResultType>
using Threefry2x64Engine = ThreefryEngine<ResultType, std::uint64_t, 2>;

template <typename ResultType>
using Threefry4x64Engine = ThreefryEngine<ResultType, std::uint64_t, 4>;

template <typename ResultType>
using Threefry8x64Engine = ThreefryEngine<ResultType, std::uint64_t, 8>;

template <typename ResultType>
using Threefry16x64Engine = ThreefryEngine<ResultType, std::uint64_t, 16>;
```

Type aliases for 32- and 64-bits `ResultType` are also defined, as listed in Table 5.1. The performance data is in Table C.7.

5.2.4 *RNG and RNGMini*

Note that, not all RNGs implemented by the library is available on all platforms. The library also defines two type aliases `RNG` and `RNG_64`, which are one of the RNGs listed in Table 5.1. The preference is in the order listed in Table 5.3. The user can define the configuration macro `VSMC_RNG_TYPE` to override the choice made by the library.

5.2.5 *Seeding counter-based RNG*

The singleton class template `SeedGenerator` can be used to generate distinctive seeds sequentially. For example,

Alias	Class	Availability
`RNG`	`ARS`	`VSMC_HAS_AES_NI`
	`Threefry`	Always available
`RNG_64`	`ARS_64`	`VSMC_HAS_AES_NI`
	`Threefry_64`	Always available

TABLE 5.3 Default RNG

```
auto &seed = SeedGenerator<void, unsigned>::instance();
RNG rng1(seed.get()); // Construct rng1
RNG rng2(seed.get()); // Construct rng2 with another seed
```

The first argument to the template can be any type. For different types, different instances of `SeedGenerator` will be created. Thus, the seeds generated by two generators, `SeedGenerator<T1>` and `SeedGenerator<T2>`, will be independent. The second parameter is the type of the seed values. It can be any unsigned integer type. Classes such as `Particle<T>` will use the generator of the following type,

```
using Seed = SeedGenerator<NullType, VSMC_SEED_RESULT_TYPE>;
```

where `VSMC_SEED_RESULT_TYPE` is a configuration macro which is defined to `unsigned` by default.

One can save and set the seed generator using standard C++ streams. For example,

```
std::ifstream is("seed.txt");
if (is)
    is >> Seed::instance();    // Read seed from a file
else
    Seed::instance().set(101); // Set it manually
is.close();
// Using Seed
std::ofstream os("seed.txt");
os << Seed::instance();        // Write the seed to a file
os.close();
```

This way, if the simulation program needs to be repeated multiple times, each time it will use a different set of seeds. A single seed generator is enough for a single program. However, it is more difficult to ensure that each computing node has a distinctive set of seeds in a distributed system. A simple solution is to use the `modulo` method of `SeedGenerator`. For example,

```
Seed::instance().modulo(n, r);
```

where n is the number of processes and r is the rank of the current node. After this call, all seeds generated will belong to the equivalent class $s \equiv r \mod n$. Therefore, no two nodes will ever generate the same seeds. Note that, the seeds generated are not random at all. For any deterministic RNGs, the same seeds always produce identical streams. However, distinctive seeds does not always lead to independent streams. This seed generator is only suitable for counter-based RNGs.

5.3 NON-DETERMINISTIC RNG

If the RDRAND instructions are supported, the library also implements three RNGs, `RDRAND16`, `RDRAND32` and `RDRAND64`. They output 16-, 32-, and 64-bits random integers, respectively. The RDRAND instruction may not return a random integer at all. The RNG engine will keep trying until it succeeds. One can limit the maximum number of trials by defining the configuration macro `VSMC_RNG_RDRAND_NTRIAL_MAX`. A value of zero, the default, means the number of trials is unlimited. If it is a positive number, and if after the specified number of trials no random integer is return by the RDRAND instruction, zero is returned. The performance data is in Table C.8.

5.4 MKL RNG

The MKL library provides some high performance RNGs. The library implements a wrapper class `MKLEngine` that makes them accessible as C++11 engines. They are listed in Table 5.4. Note that, MKL RNGs perform the best when they are used to generate vectors of random numbers. These wrappers use a buffer to store such vectors. And thus they have much larger state space than usual RNGs. Each RNG engines output by default 32-bits integers. Similar to the counter-based RNGs, 64-bits variants are also defined. The performance data is in Table C.9.

5.5 MULTIPLE RNG STREAMS

Earlier in section **??** we introduced that `particle.rng(i)` returns an independent RNG instance. This is actually done through a class template called `RNGSet`. Three of them are implemented in the library. They all have the same interface,

Class	MKL BRNG
`MKL_MCG59`	`VSL_BRNG_MCG59`
`MKL_MT19937`	`VSL_BRNG_MT19937`
`MKL_MT2203`	`VSL_BRNG_MT2203`
`MKL_SFMT19937`	`VSL_BRNG_SFMT19937`
`MKL_NONDETERM`	`VSL_BRNG_NONDETERM`
`MKL_ARS5`	`VSL_BRNG_ARS5`
`MKL_PHILOX4X32X10`	`VSL_BRNG_PHILOX4X32X10`

TABLE 5.4 MKL RNG

```
RNGSet<RNG> rng_set(N); // A set of N RNGs
rng_set.resize(n);      // Change the size of the set
rng_set.seed();         // Seed each RNG in the set with Seed::instance()
rng_set[i];             // Get a reference to the i-th RNG
```

The first implementation is `RNGSetScalar`. As its name suggests, it is only a wrapper of a single RNG. All calls to `rng_set[i]` returns a reference to the same RNG. It is only useful when an `RNGSet` interface is required while the thread-safety and other issues are not important.

The second implementation is `RNGSetVector`. It is an array of RNGs with length N. It has memory cost $\mathcal{O}(N)$. Many of the counter-based RNGs have small state size and thus for moderate N, this cost is not an issue. The method calls `rng_set[i]` and `rng_set[j]` return independent RNGs if $i \neq j$. This implementation has the advantage that the behavior of an algorithm can be entirely deterministic even when the scheduling of parallel execution is dynamic, since each sample has its own RNG.

Last, if TBB is available, there is a third implementation `RNGSetTBB`, which uses thread-local storage (TLS). It has much smaller memory footprint than `RNGSetVector` while maintains better thread-safety. The performance impact of using TLS is minimal unless the computation at the calling site is trivial. For example,

```
std::size_t eval_pre(SingleParticle<T> sp)
{
    auto &rng = sp.rng();
    // using rng to initialize state
    // do some computation, likely far more costly than TLS
}
```

The type alias `RNGSet` is defined to be `RNGSetTBB` if TBB is available, otherwise defined to be `RNGSetVector`. It is used by the `Particle` class template. One can replace the type of RNG set used by `Particle<T>` with a member type of `T`. For example,

```
class T
{
    public:
    using rng_set_type = RNGSetScalar<RNG>;
};
```

will replace the type of the RNG set contained in `Particle<T>`. Below is a more advanced example of replacing `rng_set_type` when using OpenMP for parallelization.

```
template <typename RNGType>
class RNGSetOMP
{
    public:
    RNGSetOMP(std::size_t) : rng_(/* maximum number of OpenMP threads */)
    {
        seed();
    }

    void resize(std::size_t) {}

    void seed() { Seed::instance()(rng_.size(), rng_.begin()); }

    RNGType &operator[](std::size_t)
    {
        return rng_[omp_get_thread_num()];
    }

    private:
    Vector<RNGType> rng_;
};
```

```
class T
{
    public:
    using rng_set_type = RNGSetOMP<RNG>;
};
```

In this example, only a small number of RNG engines are created and it is (mostly) thread-safe. However, there are quite a few situations where this class is not suitable, with serialized nested parallel region being a primary one. The library does not provide the above implementation by default. There are too many cases that it can be misused.

5.6 DISTRIBUTIONS

The library provides implementations of some common distributions. Some of them are the same as those in the standard library, with `CamelCase` names. For example, `NormalDistribuiton` can be used as a drop-in replacement of `std::normal_distribuiton`. This includes all of the continuous distributions defined in the standard library. As stated in section 5.1, all the distributions defined in the library support vectorized random number generating. In the following sections we introduce each distributions included in the library.

5.6.1 *Uniform bits distribution*

The class template,

```
template <typename UIntType>
class UniformBitsDistribution;
```

is similar to the standard library's `std::independent_bits_engine`, except that it always generates full size random integers. That is, let W be the number of bits of `UIntType`, then the output is uniform on the set $\{0, \dots, 2^W - 1\}$. For example,

```
UniformBitsDistribution<std::uint32_t> rbits;
rbits(rng); // Return 32-bits random integers
```

Let r_{min} and r_{max} be the minimum and maximum of the random integers generated by `rng`. Let $R = r_{\text{max}} - r_{\text{min}} + 1$. Let r_i be consecutive output of `rng()`. If there exists an integer $M > 0$ such that $R = 2^M$, then the result is,

$$U = \sum_{k=0}^{K-1} (r_k - r_{\min}) 2^{kM} \bmod 2^W$$

where $K = \lceil W/M \rceil$. Unlike `std::independent_bits_engine`, the calculation can be vectorized, which leads to better performance. Note that, all constants in the algorithm are computed at compile-time and the summation is fully unrolled, and thus there is no runtime overhead. In the case $r_{\min} = 0$ and $M = W$, most optimizing compilers shall be able to generate instructions such that the distribution does exactly nothing and returns the results of `rng()` directly. If there does not exist an integer $M > 0$ such that $R = 2^M$, then `std::indepdent_bits_engine` will be used.

5.6.2 *Standard uniform distribution*

All continuous distributions are built upon the standard uniform distribution. And thus the performance and quality of the algorithm transferring random integers to random floating point numbers on the set $[0, 1]$ are of critical importance. The library provides five distributions, listed in Table 5.5. They are all class template with a single template type parameter `RealType`, which is the floating point result type. For each distribution, the random integers produced by RNGs are transferred to 32- or 64-bits intermediate random integers through `UniformBitsDistribution` before they are further converted to floating numbers. The integer type depends on `RealType`, the range of the integers produced by the RNG, R, and the configuration macros `VSMC_RNG_U01_USE_64BITS_DOUBLE`. The exact relations are listed in Table 5.6. In the remaining of this section, let W be the number of bits of the intermediate random integers, and M be the number of significant bits (including the implicit one) of `RealType`. We also denote the input random integers as U and the output random real numbers as X.

Distribution	Support
`U01CCDistribution`	$[0,1]$
`U01CODistribution`	$[0,1)$
`U01OCDistribution`	$(0,1]$
`U01OODistribution`	$(0,1)$
`U01Distribution`	$[0,1)$

TABLE 5.5 Standard uniform distributions

`RealType`	Conditions	Integer type
`float`	$\log_2 R \ge 64$	`std::uint64_t`
	Otherwise	`std::uint32_t`
`double`	$\log_2 R \ge 64$	`std::uint64_t`
	`VSMC_RNG_U01_USE_64BITS_DOUBLE`	`std::uint64_t`
	Otherwise	`std::uint32_t`
`long double`	Always	`std::uint64_t`

TABLE 5.6 Intermediate integer types of standard uniform distributions

`U01CCDistribution` This distribution produce random real numbers on $[0,1]$, with the lower and upper bounds inclusive. The specific algorithm is as the following,

$$\begin{aligned}
P &= \min\{W-1, M\} \\
V &= \begin{cases} U & \text{if } P+1 < W \\ \lfloor (U \bmod 2^{W-1})/2^{W-P-2} \rfloor & \text{otherwise} \end{cases} \\
Z &= (V \bmod 1) + V \\
X &= 2^{-(P+1)} Z
\end{aligned}$$

The minimum and maximum are 0 and 1, respectively.

`U01CODistribution` This distribution produce random real numbers on $[0, 1)$, with the lower bound inclusive and the upper bound never produced. The specific algorithm is as the following,

$$\begin{aligned}P &= \min\{W, M\}\\ V &= \lfloor U/2^{W-P} \rfloor\\ X &= 2^{-P}V\end{aligned}$$

The minimum and maximum are 0 and $1 - 2^{-P}$, respectively.

`U01OCDistribution` This distribution produce random real numbers on $(0, 1]$, with the upper bound inclusive and the lower bound never produced. The specific algorithm is as the following,

$$\begin{aligned}P &= \min\{W, M\}\\ V &= \lfloor U/2^{W-P} \rfloor\\ X &= 2^{-P}V + 2^{-P}\end{aligned}$$

The minimum and maximum are 2^{-P} and 1, respectively.

`U01OODistribution` This distribution produce random real numbers on $(0, 1)$, with the lower and upper bounds never produced. The specific algorithm is as the following,

$$\begin{aligned}P &= \min\{W + 1, M\}\\ V &= \lfloor U/2^{W+1-P} \rfloor\\ X &= 2^{-(P-1)}V + 2^{-P}\end{aligned}$$

The minimum and maximum are 2^{-P} and $1 - 2^{-P}$, respectively.

`U01Distribution` It is now clear that the above four distributions actually produce "fixed point" instead of "floating point" numbers. The output X can be represented exactly by the target `RealType`. They have two advantages. First, when it is important that the lower or upper bound is never produced, to avoid underflow, overflow or other undefined behaviors in subsequent calculations, they provide such assurance. Second, they usually can be executed with only a couple of instructions by modern processors. And thus can have better performance.

The main drawback is accuracy. If `RealType` is `float` or `long double`, then the difference is minimal, since the intermediate random integers have more bits than the significant of the target floating point type. The situation is a bit more tricky in the case of `double` and the intermediate random integers are 32-bits. In this case, `U01CODistribution` can only produce 2^{32} distinctive values while `double` can represent much more values exactly within the range $[0, 1)$. In contrast, the standard library will use at least 53 random bits. This will not matter in most realistic applications. In fact, random numbers produced by `U01CODistribution` passes all tests in the TestU01[4] library that `std::uniform_real_distribution` would pass, for a good RNG. In other words, the quality of the RNG is the dominating factor.

However, there are situations where one do want the extra precision. For example, the library implement the Normal distribution using the standard Box-Muller method (Box and Muller 1958), for performance consideration. Better accuracy at the tail can only be archived by using procedures that can produce values closer to zero than 2^{-32}. In this case, there are two solutions. The first is to define the configuration macro `VSMC_RNG_U01_USE_FIXED_POINT` to zero, and thus `U01Distribution` is no longer the same as `U01CODistribution`. Instead, it behaves similarly to `std::generate_canonical`. More specifically,

$$\begin{aligned}
P &= \lfloor (W + M - 1)/W \rfloor \\
K &= \max\{1, P\} \\
X &= \sum_{k=0}^{K-1} U_k 2^{-(K-k)W}
\end{aligned}$$

The other solution is to define the configuration macro `VSMC_RNG_U01_USE_64BITS_DOUBLE` to a non-zero value, such that the intermediate random integers will always be 64-bits for `double` output. This configuration macro also affects all other four distributions discussed earlier.

5.6.3 *Distributions using the inverse method*

Table 5.7 lists all the distributions implemented with the inverse method. The performance data of these distributions is in Table C.10. Note that, in this and other tables in the remaining of this section, we shortened the class name for brevity. For example, in the table the Cauchy distribution is listed as `Cauchy` while the full declaration of the class template is,

```
template <typename RealType>
class CauchyDistribution;
```

[4] `http://www.iro.umontreal.ca/~simardr/testu01/tu01.html`

Distribution	Parameters	Support	CDF
`Cauchy`	`a, b`	$(-\infty,\infty)$	$\frac{1}{\pi}\arctan\left(\frac{x-a}{b}\right)+\frac{1}{2}$
`Exponential`	`lambda`	$[0,\infty)$	$1-\mathrm{e}^{-\lambda x}$
`ExtremeValue`	`a, b`	$(-\infty,\infty)$	$\exp\left\{-\exp\left(-\frac{x-a}{b}\right)\right\}$
`Laplace`	`a, b`	$(-\infty,\infty)$	$\frac{1}{2}+\frac{1}{2}\mathrm{sgn}(x-a)\left(1-\exp\left\{-\frac{\lvert x-a\rvert}{b}\right\}\right)$
`Logistic`	`a, b`	$(-\infty,\infty)$	$\left(1+\exp\left\{-\frac{x-a}{b}\right\}\right)^{-1}$
`Pareto`	`a, b`	$[a,\infty)$	$1-\left(\frac{b}{x}\right)^{a}$
`Rayleigh`	`sigma`	$[0,\infty)$	$1-\exp\left\{-\frac{x^2}{2\sigma^2}\right\}$
`UniformReal`	`a, b`	$[a,b)$	$\frac{x-a}{b-a}$
`Weibull`	`a, b`	$[0,\infty)$	$1-\exp\left\{-\left(\frac{x}{b}\right)^{a}\right\}$

TABLE 5.7 Distributions using the inverse method

5.6.4 *Normal and related distribution*

The class template

```
template <typename RealType>
class NormalDistribution;
```

implements the Normal distribution with PDF,

$$f(x)=\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\},$$

using the Box-Muller method (Box and Muller 1958). It is also used to implement the Log-Normal distribution, $X=\mathrm{e}^{m+sZ}$, where Z follows the standard Normal distribution,

```
template <typename RealType>
class LognormalDistribution;
```

and the Levy distribution, $X=a+b/Z^2$,

```
template <typename RealType>
class LevyDistribution;
```

The performance data of these distributions is in Table C.11.

Multivariate Normal distribution The library also implements the multivariate Normal distribution,

```
template <typename RealType, std::size_t Dim>
class NormalMVDistribution;
```

If `Dim` is zero (`Dynamic`), then the distribution can only be constructed with,

```
explicit NormalMVDistribution(std::size_t dim,
    const result_type *mean = nullptr, const result_type *chol = nullptr);
```

If `Dim` is a positive integer, it can only be constructed with,

```
explicit NormalMVDistribution(
    const result_type *mean = nullptr, const result_type *chol = nullptr);
```

In either case, the parameter `mean` is the mean vector. If it is a null pointer, then it is assumed to be a zero vector. The parameter `chol` is a $d(d+1)/2$-vector, where d is the dimension. The vector is the lower triangular elements of the Cholesky decomposition of the covariance matrix, packed row by row. Libraries such as LAPACK has routines to compute such a matrix. Alternatively, one can use the covariance functionalities in the library. See section 7.2, which also provides a concrete example of using the multivariate Normal distribution.

5.6.5 *Gamma and related distribution*

The class template

```
template <typename RealType>
class GammaDistribution;
```

implements the Gamma distribution with PDF,

$$f(x) = \frac{\mathrm{e}^{-x/\beta}}{\Gamma(\alpha)}\beta^{-\alpha}x^{\alpha-1}.$$

The specific algorithm used depends on the parameters. If $\alpha = 1$, it becomes the exponential distribution. If $0 < \alpha < 0.6$, it is generated through transformation of exponential power distribution (Devroye 1986, sec 2.6). If $0.6 \le \alpha < 1$, then rejection method from the Weibull distribution is used (Devroye 1986, sec. 3.4). If $\alpha > 1$, then the method in Marsaglia and Tsang (2000) is used. There are three related distributions,

```
template <typename RealType>
class ChiSquaredDistribution;

template <typename RealType>
class FisherFDistribution;

template <typename RealType>
class StudentTDistribution;
```

They implement the χ^2-distribution, the Fisher's F-distribution, and the Student's t-distribution, respectively. The performance data of these distributions, for different parameters are in Tables C.12 to C.15.

5.6.6 *Beta distribution*

The class template

```
template <typename RealType>
class BetaDistribution;
```

implements the Beta distribution with PDF,

$$f(x) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1 - x)^{\beta-1}$$

The specific algorithm used depends on the parameters. If $\alpha = 1/2$ and $\beta = 1/2$, or $\alpha = 1$ or $\beta = 1$, then the inverse method is used. If $\alpha > 1$ and $\beta > 1$, the method in Cheng (1978) is used. Otherwise, let $K = 0.852$, $C = -0.956$, and $D = \beta + K\alpha^2 + C$. If $\alpha < 1$, $\beta < 1$ and $D \leq 0$, then Jöhnk's method (Devroye 1986, sec. 3.5) is used. In all other cases, one of the switching algorithms in Atkinson (1979) is used. Note that, there is no vectorized implementation at the moment for the switching algorithms. In other cases, the vectorized generating shall provide considerable speedup. The performance data is in Table C.16

6 RESAMPLING

6.1 BUILTIN ALGORITHMS

The library supports resampling in a more general way than the algorithm described in chapter **??**. Recall that, given a particle system $\{W^{(i)}, X^{(i)}\}_{i=1}^N$, a new system $\{\bar{W}^{(i)}, \bar{X}^{(i)}\}_{i=1}^M$ is generated. Regardless of other statistical properties, in practice, such an algorithm can be decomposed into three steps. First, a vector of replication numbers $\{r_i\}_{i=1}^N$ is generated such that $\sum_{i=1}^N r_i = M$, and $0 \le r_i \le M$ for $i = 1, \dots, N$. Then a vector of indices $\{a_i\}_{i=1}^M$ is generated such that $\sum_{i=1}^M \mathbb{I}_{\{j\}}(a_i) = r_j$, and $1 \le a_i \le N$ for $i = 1, \dots, M$. And last, set $\bar{X}^{(i)} = X^{(a_i)}$.

The first step determines the statistical properties of the resampling algorithm. The library defines all algorithms discussed in Douc, Cappé, and Moulines (2005). Samplers can be constructed with builtin schemes as seen in section **??**. In addition, samplers can also be constructed with user defined resampling operations. A user defined resampling algorithm can be any type that is convertible to `Sampler<T>::resample_type`, following function call,

```
using resample_type = std::function<void(std::size_t, std::size_t,
    typename Particle<T>::rng_type &, const double *, size_type *)>;
```

where the first argument is N, the sample size before resampling; the second is M, the sample size after resampling; the third is a C++11 RNG type, the fourth is a pointer to normalized weight, and the last is a pointer to the vector $\{r_i\}_{i=1}^N$. The builtin schemes are implemented as classes with `operator()` conforms to the above signature. All builtin schemes are listed in Table 6.1

To transform $\{r_i\}_{i=1}^N$ into $\{a_i\}_{i=1}^M$, one can call the following function,

```
template <typename IntType1, typename IntType2>
void resample_trans_rep_index(std::size_t N, std::size_t M,
    const IntType1 *replication, IntType2 *index);
```

where the last parameter is the output vector $\{a_i\}_{i=1}^M$. This function guarantees that $a_i = i$ if $r_i > 0$, for $i = 0, \dots, \min\{N, M\}$. However, its output may not be optimal for all applications. The last step of a resampling operation, the copying of particles can be the most time consuming one, especially on distributed systems. The topology of the system will need to be taking into consideration to achieve optimal performance. In those situations, it is best to use

`ResampleScheme`	Algorithm
`Multinomial`	Multinomial resampling
`Stratified`	Stratified resampling
`Systematic`	Systematic resampling
`Residual`	Residual resampling
`ResidualStratified`	Stratified resampling on residuals
`ResidualSystematic`	Systematic resampling on residuals

Table 6.1 Resampling schemes

`ResampleMultinomial` etc., to generate the replication numbers, and manually perform the rest of the resampling algorithm.

6.2 USER DEFINED ALGORITHMS

The library provides facilities for implementing new resampling algorithms. The most common situation is that, a vector of random numbers on the interval $[0, 1)$ is generated, say $\{u_i\}_{i=1}^M$. The replication numbers are $r_i = \sum_{j=1}^M \mathbb{I}_{[v_{i-1},v_i)}(u_i)$, where $v_i = \sum_{j=1}^i W_i$, $v_0 = 0$. For example, the Multinomial resampling algorithm is equivalent to $\{u_i\}_{i=1}^M$ being i.i.d. standard uniform random numbers.

Alternatively, let $p_i = \lfloor MW_i \rfloor$, $q_i = MW_i - p_i$. One can perform resampling on the residuals, using weights proportional to $\{q_i\}_{i=1}^N$. The output size shall be $R = M - \sum_{i=1}^N p_i$. Let the replication numbers be $\{s_i\}_{i=1}^N$, then $r_i = p_i + s_i$.

The library provides the following class template for implementing such algorithms,

```
template <typename U01SeqType, bool Residual>
class ResampleAlgorithm;
```

where `U01SeqType` will be discussed later. The second parameter `Residual` determines if the resampling shall be applied to residuals.

The template template parameter `U01SeqType` shall be a class with default construct that defines the following an `operator()` that is compatible with the following,

```
template <typename RNGType>
void operator()(RNGType &rng, std::size_t N, double *r);
```

which will generate N random numbers within $[0, 1]$ such, say $\{U_i\}_{i=1}^N$, such that $U_1 \le U_2 \le \dots \le U_N$.

An obvious method is to generate the random numbers first and then sort them. However, no sorting algorithm has cost $\mathcal{O}(M)$, while it is possible to generate such an ordered sequence with cost $\mathcal{O}(M)$ by using order statistics. The library defines three such sequences. The first is equivalent to sorted i.i.d. random numbers. The second and the third are stratified and systematic, respectively. The builtin resampling algorithms are implemented using these sequences. For example,

```
using ResampleMultinomial = ResampleAlgorithm<U01SequenceSorted, false>;
using ResampleResidual = ResampleAlgorithm<U01SequenceSorted, true>;
```

If the user is able to define a new ordered random sequence, either through sorting or otherwise, then using the `ResampleAlgorithm` template, a new resampling algorithm can easily be implemented. Note that, the algorithm implemented by this class template always has a cost $\mathcal{O}(N + M)$, unless the random sequence has a greater cost.

6.3 ALGORITHMS WITH INCREASING DIMENSIONS

In general an SIS algorithm operates on increasing dimensions. Assume that the storage cost of a single particle at the marginal $(X_t^{(i)})$ is of order $\mathcal{O}(1)$. At each iterations t, the path $X_{0:t-1}^{(i)}$ is extended to $X_{0:t}^{(i)}$. The resampling algorithms operate on the space $\prod_{k=0}^t E_k$. When the proposal $q_t(\cdot|X_{0:t-1}^{(i)}) = q_t(\cdot|X_{t-1}^{(i)})$ and only the marginal $\eta_t(X_t)$ is of interest, one can only resample $\{X_t^{(i)}\}_{i=1}^N$. This leads to $O(N)$ cost for resampling. This is the typical case for SMC algorithms. However, there are situations where resampling $\{X_{0:t}^{(i)}\}_{i=1}^N$ is necessary. In this case, the cost of resampling at iteration t is $O(tN)$. And total resampling cost to obtain $\{X_{0:t}^{(i)}\}_{i=1}^N$, is $O(t^2N)$.

However, such cost is avoidable in some circumstances. Recall that, after generating the resampling index $\{a_i\}_{i=1}^N$, one set $\bar{X}_{0:t}^{(i)} = X_{0:t}^{(a_i)}$. Let $(\{X_0^{(i)}\}_{i=1}^N, \dots, \{X_t^{(i)}\}_{i=1}^N)$ be the marginals before resampling, and $(\{a_0^{(i)}\}_{i=1}^N, \dots, \{a_t^{(i)}\}_{i=1}^N)$ be the resampling index vectors at each iteration.

Then one can obtain $X_{0:t}^{(i)}$ through the following recursion,

$$
\begin{aligned}
b_t^{(i)} &= a_t^{(i)} \\
b_k^{(i)} &= a_k^{(b_{k+1}^{(i)})} \text{ for } k = t-1, \dots, 0 \\
\bar{X}_k^{(i)} &= X_k^{(b_k^{(i)})} \text{ for } k = t, \dots, 0
\end{aligned}
$$

Intuitively, only the resampling index vectors are resampled. The cost of the above recursion is $O(tN)$ instead of $O(t^2N)$. Not that it is likely to be much slower compared to directly copying particles at each iteration in the situation when only the marginals need to be resampled, in which case both has a cost $O(tN)$.

This algorithm is useful in the following situation. Assume that there exist recursive functions,

$$
\begin{aligned}
\varphi_t(X_{0:t}^{(i)} | X_{0:t-1}^{(i)}) &= \varphi_t(X_t^{(i)}, \varphi_{t-1}(X_{0:t-1}^{(i)})), \\
\phi_t(X_{0:t}^{(i)} | X_{0:t-1}^{(i)}) &= \phi_t(X_t^{(i)}, \varphi_{t-1}(X_{0:t-1}^{(i)}), \phi_{t-1}(X_{0:t-1}^{(i)})),
\end{aligned}
$$

such that $q(\cdot|X_{0:t}^{(i)}) = q(\cdot|\varphi(X_{0:t-1}^{(i)}))$ and $W_t(X_{0:t}^{(i)}) = W_t(\phi(X_{0:t}^{(i)}))$. In this case, only the values of $\varphi_t(X_{0:t}^{(i)})$ and $\phi_t(X_{0:t}^{(i)})$ need to be resampled at every iteration. If they have storage costs at the order of $\mathcal{O}(1)$, then the total cost of resampling will still be $O(tN)$. See Beskos et al. (2014) for some examples of such algorithms.

The library provides the following class template for implementing such an algorithm.

```
template <typename IntType = std::size_t>
class ResampleIndex;
```

Its usage is demonstrated by the following example (assuming $X_t^{(i)} \in \mathbb{R}$, and ϕ_t is the same as φ_t),

```
  using StateBase = StateMatrix<ColMajor, Dynamic, double>;

  class State : StateBase
  {
      public:
      StateBase(std::size_t N) : StateBase(N), varphi_(N) {}
```

```
    template <typename IntType>
    void copy(std::size_t N, IntType *index)
    {
        // DO NOT CALL StateBase::copy
        for (std::size_t i = 0; i != N; ++i)
            varphi_[i] = varphi_[index[i]];
        index_.push_back(N, index);
    }

    // To be called during initialization
    void reset() { index_.reset(); }

    // Get the resampled state up to time n
    // Assuming that this->state(i, t) contains the marginal $X_t^{(i)}$
    State trace_back(std::size_t n)
    {
        // idxmat is an $N$ by $n + 1$ matrix, say $B$, such that
        // $B_{i,j} = b_j^{(i)}$
        auto idxmat = index_.index_matrix(ColMajor, n);
        State rs(*this);
        for (std::size_t j = 0; j <= n; ++j) {
            auto dst = rs.col_data(j);
            auto src = this->col_data(j);
            auto idx = idxmat.data() + j * this->size();
            for (std::size_t i = 0; i != this->size(); ++i)
                dst[i] = src[idx[i]];
        }

        return rs;
    }

    private:
    Vector<double> varphi_; // the values of $\varphi(X_t^{(i)})$
    ResampleIndex index_;
};
```

```
Sampler<State> sampler(N, Multinomial); // Always resampling
sampler.particle.value().resize_dim(n + 1);
// configure the sampler
sampler.initialize(param);
sampler.iterate(n);
auto state = sampler.particle().value().trace_back(n);
```

The method call `index_.push_back(N, index)` append a new resampling index vector to the history being recorded by `index_`. If called without the second argument, i.e., `index_.push_back(N)`, then it is assumed $a_i = i$ for $i = 1, \dots, N$. To retrieve $b_{t_0}^{(i)}$, where $t_0 \le t$, one can call `index_.index(i, t, t0)` If the last argument is omitted, it is assumed to be zero. If the second argument is also omitted, then it is assumed to be the iteration number of the last index vector recorded. It is of course more useful, and more efficient to retrieve an N by R matrix B, such that $R = t - t_0 + 1$, $B_{i,j} = b_j^{(i)}$. This is done by calling `index_.index_matrix(t, t0)`. Again, both arguments can be omitted, and the default values are the same as for `index_.index(i, t, t0)`.

The performance difference directly copy $X_{0:t}^{(i)}$ at each iteration and using the above implementation can be significant for moderate to large t. Of course, if $\eta_k(X_{0:t})$ is of interest for all $k \le t$, instead of only $\eta_t(X_{0:t})$ being of interest, then one is better off to copy all states at all iterations.

Note that, `ResampleIndex` is capable of dealing with varying sample size situations. Each call of `push_back` does not need to have the same sample size N. In addition, if such an index object need to be reused multiple times, one can use its `insert` method instead of `push_back`. See the reference manual for details.

7 UTILITIES

The library provides some utilities for writing Monte Carlo simulation programs. For some of them, such as command line option processing, there are more advanced, dedicated libraries out there. The library only provides some basic functionality that is sufficient for simple cases.

7.1 ALIGNED MEMORY ALLOCATION

The standard library class `std::allocator` is used by containers to allocate memory. It works fine in most cases. However, sometime it is desirable to allocate memory aligned by a certain boundary. The library provides the class template,

```
template <typename T, std::size_t Alignment = AlignmentTrait<T>::value,
    typename Memory = AlignedMemory>
class Allocator;
```

which conforms to the `std::allocator` interface. The address of the pointer returned by the `allocate` method will be a multiple of `Alignment`. The value of alignment has to be positive, larger than `sizeof(void *)`, and a power of two. Violating any of these conditions will result in compile-time error. The last template parameter `Memory` shall have two static methods,

```
static void *aligned_malloc(std::size_t n, std::size_t alignment);
static void aligned_free(void *ptr);
```

The method `aligned_malloc` shall behave similar to `std::malloc` with the additional alignment requirement. It shall return a null pointer if it fails to allocate memory. In any other case, including zero input size, it shall return a reachable non-null pointer. The method `aligned_free` shall behave similar to `std::free`. It shall be able to handle a null pointer as its input. The library provides three implementations, discussed below. The default argument `AlignedMemory` is an alias to one of them by default. It can be changed by defining the macro `VSMC_ALIGNED_MEMORY_TYPE`.

`AlignedMemoryTBB` This class uses `scalable_aligned_malloc` and `scalable_aligned_free` from the TBB library. This is the default method if `VSMC_USE_TBB_MALLOC` is true.

AlignedMemorySYS This class uses posix_memalign and free on POSIX platforms. It uses _aligned_malloc and _aligned_free if the compiler is MSVC. Otherwise, this class is not defined. If this class is define, it is the default method if VSMC_USE_TBB_MALLOC is false.

AlignedMemorySTD This class uses only std::malloc and std::free. It is the last resort method the library will use.

The default alignment depends on the type T. If it is a scalar type (std::is_scalar<T>), then the alignment is VSMC_ALIGNMENT, whose default is 32. This alignment is sufficient for modern SIMD operations, such as AVX. For other types, the alignment is the maximum of alignof(T) and VSMC_ALIGNMENT_MIN, whose default is 16. Two container types are defined for convenience,

```
template <typename T>
using Vector = std::vector<T, Allocator<T>>;

template <typename T, std::size_t N,
    std::size_t Alignment = AlignmentTrait<T>::value>
class alignas(Alignment) Array;
```

The first can be used as a drop-in replacement of std::vector<T> since it is merely a type alias with a different default allocator. The second can is derived from std::array<T, N> and thus can be a drop-in replacement in most situations.

7.2 SAMPLE COVARIANCE

The library provides a class template to estimate sample covariance,

```
template <typename RealType>
class Covariance;
```

At the time of writting, only float and double are supported. The class has the following operator as its interface,

```
void operator()(MatrixLayout layout, std::size_t n, std::size_t p,
    const RealType *x, const RealType *w, RealType *mean,
    RealType *cov, MatrixLayout cov_layout = RowMajor,
    bool cov_upper = false, bool cov_packed = false)
```

It computes the sample covariance matrix Σ,

$$\begin{aligned}\Sigma_{i,j} &= \frac{\sum_{i=1}^N w_i}{(\sum_{i=1}^N w_i)^2 - \sum_{i=1}^n w_i^2} \sum_{k=1}^N w_k(x_{k,i} - \bar{x}_i)(x_{k,j} - \bar{x}_j) \\ \bar{x}_i &= \frac{1}{\sum_{i=1}^N w_i} \sum_{k=1}^N w_k x_{k,i}\end{aligned}$$

where x is the n by p matrix of samples, and w is the n-vector of weights. Below we given detailed description of each of the parameters,

`layout` The storage layout of sample matrix x. It is assumed to be an n by p matrix.

`n` The number of samples.

`p` The dimension of each sample.

`x` The sample matrix. If it is a null pointer, then no computation is carried out.

`w` The weight vector. If it is a null pointer, then all samples are assigned weight 1.

`mean` Output storage of the mean. If it is a null pointer, then it is ignored.

`cov` Output storage of the covariance matrix. If it is a null pointer, then it is ignored.

`cov_layout` The storage layout of the covariance matrix.

`cov_upper` If `true`, then the upper triangular of the covariance matrix is packed, otherwise the lower triangular is packed. Ignored if `cov_pack` is `false`.

`cov_packed` If `true`, then the covariance matrix is packed.

The last three parameters specify the storage scheme of the covariance matrix. See any reference of BLAS or LAPACK for explanation of the scheme. Below is an example of the class in use,

```
using T = StateMatrix<RowMajor, Dynamic, double>;
Sampler<T> sampler(n, p);
// Configure and iterate the sampler
double mean[p];
double cov[p * (p + 1) / 2];
Covariance eval;
auto x = sampler.particle().value().data();
auto w = sampler.particle().weight().data();
eval(RowMajor, n, p, x, w, mean, cov, RowMajor, false, true);
```

One can later compute the Cholesky decomposition using LAPACK or other linear algebra libraries. Below is an example of using the covariance matrix to generate multivariate Normal proposals,

```
double chol[p * (p + 1) / 2];
double y[p];
LAPACKE_dpptrf(LAPACK_ROW_MAJOR, 'L', p, chol);
NormalMVDistribution<double, p> normal_mv(mean, chol);
normal_mv(rng, y);
```

7.3 STORE OBJECTS IN HDF5 5 FORMAT

If the HDF5 library is available (`VSMC_HAS_HDF5`), it is possible to store `Sampler<T>` objects, etc., in the HDF5 format. For example,

```
hdf5store(sampler, "pf.h5", "sampler", false);
```

creates an HDF5 file named `pf.h5` with the sampler stored as a list in the group `sampler`. If the last argument is `true`, the data is inserted to an existing file. Otherwise a new file is created. In R it can be processed as the following,

```
library(rhdf5)
pf <- as.data.frame(h5read("pf.h5", "sampler"))
```

This creates a `data.frame` similar to that shown in section **??**. The `hdf5store` function is overloaded for `StateMatrix`, `Sampler<T>` and `Monitor<T>`. It is also overloaded for `Particle<T>` if an overload for `T` is defined. The all follow the format as above. In addition, the following creates a new empty HDF5 file,

```
hdf5store(filename);
```

while the following create a new group named `dataname`,

```
hdf5store(filename, dataname, append);
```

7.4 RAII CLASSES FOR OPENCL POINTERS

The library provides a few classes to manager OpenCL pointers. It provides RAII idiom on top of the OpenCL C interface. For example, below is a small program,

Class	OpenCL pointer type
`CLPlatform`	`cl_platform_id`
`CLContext`	`cl_context`
`CLDevice`	`cl_device_id`
`CLCommandQueue`	`cl_command_queue`
`CLMemory`	`cl_mem`
`CLProgram`	`cl_program`
`CLKernel`	`cl_kernel`
`CLEvent`	`cl_event`

TABLE 7.1 RAII classes for OpenCL pointers

```
auto platform = cl_get_platform().front();
auto device = platform.get_device(CL_DEVICE_TYPE_DEFAULT).front();
CLContext context(CLContextProperties(platform), 1, &device);
CLCommandQueue command_queue(context, device);
CLMemory buffer(context, CL_MEM_READ_WRITE, size);
std::string source = /* read source */;
CLProgram program(context, 1, &source);
program.build(1, &device);
CLKernel kernel(program, "kernel_name");
kernel.set_arg(0, buffer);
command_queue.enqueue_nd_range_kernel(kernel, 1, CLNDRange(), CLNDRange(N),
    CLNDRange());
```

In the above program, each class type object manages an OpenCL C type, such as `cl_platform`. The resources will be released when the object is destroyed. Note that, the copy constructor and assignment operator perform shallow copy. This is particularly important for `CLMemory` type objects. In appendix **??**, an OpenCL implementation of the simple particle filter example in section **??** is shown. Table 7.1 lists the classes defined by the library and their corresponding OpenCL C pointer types.

7.5 PROCESS COMMAND LINE PROGRAM OPTIONS

The library provides some basic support for processing command line options. Here we show a minimal example. The complete program is shown in appendix **??**. First, we define variables to store values of options,

```
int n;
std::string str;
std::vector<double> vec;
```

All types that support standard library I/O stream operations are supported. In addition, for any type `T` that supports such operations, `std::vector<T, Alloc>`, is also supported. Then,

```
ProgramOptionMap option_map;
```

constructs the container of options. Options can be added to the map,

```
option_map
    .add("str", "A string option with a default value", &str, "default")
    .add("n", "An integer option", &n)
    .add("vec", "A vector option", &vec);
```

The first argument is the name of the option, the second is a description, and the third is a pointer to where the value of the option shall be stored. The last optional argument is a default value. The options on the command line can be processed as the following,

```
option_map.process(argc, argv);
```

where `argc` and `argv` are the arguments of the `main` function. When the program is invoked, each option can be passed to it like below,

```
./program_option --vec 1 2 1e-1 --str "abc" --vec 8 9 --str "def hij" --n 2 4
```

The results of the option processing is displayed below,

```
n: 4
str: def hij
vec: 1 2 0.1 8 9
```

To summarize these output, the same option can be specified multiple times. If it is a scalar option, the last one is used (`--str`, `--n`). The value of a string option can be grouped by quotes. For a vector option (`--vec`), all values are gathered together and inserted into the vector.

7.6 DISPLAY PROGRAM PROGRESS

Sometime it is desirable to see how much progress of a program has been made. The library provides a `Progress` class for this purpose. Here we show a minimal example.

```
Progress progress;
progress.start(n * n);
for (std::size_t i = 0; i != n; ++i) {
    progress.message("i = " + std::to_string(i));
    for (std::size_t j = 0; j != n; ++j) {
        // Do some computation
        progress.increment();
    }
}
progress.stop();
```

When invoked, the program output something similar the following,

```
[  4%][00:07][  49019/1000000][i = 49]
```

The method `start` starts the printing of the progress. The argument specifies how many iterations there will be before it is stopped. The method `message` direct the program to print a message. This is optional. Each time after we finish n iterations (there are n^3 total iterations of the inner-most loop), we increment the progress count by calling `increment`. And after everything is finished, the method `stop` is called.

7.7 STOP WATCH

Performance can only be improved after it is first properly benchmarked. There are advanced profiling programs for this purpose. However, sometime simple timing facilities are enough. The library provides a simple class `StopWatch` for this purpose. As its name suggests, it works much like a physical stop watch. Here is a simple example

```
StopWatch watch;
for (std::size_t i = 0; i != n; ++i) {
    // Some computation
    watch.start();
    // Computation to be benchmarked;
    watch.stop();
    // Some other computation
}
double t = watch.seconds(); // The time in seconds
```

The above example demonstrate that timing can be accumulated between loop iterations, function calls, etc. It shall be noted that, the timing is only accurate if the computation between `start` and `stop` is non-trivial.

Atkinson, A C (1979). “A family of switching algorithms for the computer generation of beta random variables”. In: *Biometrika* 66.1, pp. 141–145.

Beskos, A. et al. (2014). “A Stable Particle Filter in High-Dimensions”. In: *ArXiv e-prints*. arXiv: `1412.3501`.

Box, G E P and Mervin E Muller (1958). “A Note on the Generation of Random Normal Deviates”. In: *The Annals of Mathematical Statistics* 29.2, pp. 610–611.

Cheng, R. C. H. (1978). “Generating Beta variates with nonintegral shape parameters”. In: *Communications of the ACM* 21.4, pp. 317–322.

Devroye, Luc (1986). *Non-Uniform Random Variate Generation*. New York, NY: Springer New York.

Douc, Randal, Olivier Cappé, and Eric Moulines (2005). “Comparison of resampling schemes for particle filtering”. In: *Proceedings of the 4th International Symposium on Imange and Signal Processing and Analysis*, pp. 1–6.

Marsaglia, George and Wai wan Tsang (2000). “A simple method for generating Gamma variables”. In: *ACM Transactions on Mathematical Software* 26.3, pp. 363–372.

Salmon, John K. et al. (2011). “Parallel random numbers: As easy as 1, 2, 3”. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12.

A CONFIGURATION MACROS

The library has a few configuration macros. All macros have default values if left undefined, and can be overwritten by defining them with proper values before including any of the library headers.

There are three types of configuration macros. The first type has a prefix `VSMC_HAS_`. These macros specify a certain feature or third-party library is available. The second type has a prefix `VSMC_USE_`. These macros specify that a certain feature or third-party library can be used, if available. These two kinds of macros shall only be defined to integral values, with zero as false and all other values as true. The third type can be defined to arbitrary contents, and usually affect implementation choices made by the library.

A concrete example of the relations among these three types of configuration macros is the memory allocation functions used by the library. The details is in section 7.1. In short, they library has three implementations for aligned memory allocation. The preferred method is to use the `scalable_aligned_malloc` from TBB. It is defined through a class called `AlignedMemoryTBB`. This class is only defined if and only if `VSMC_HAS_TBB` is true, in which case the runtime library `tbbmalloc` will need to be linked as well (`-ltbbmalloc` on UNIX-alike systems). However, if it is desirable to define this class such that the user can use it with STL containers, but it is undesirable for the library itself to use it everywhere, then one can further define the macro `VSMC_USE_TBB_MALLOC` to zero. In this case, the class is still defined, but it will not be used by the library. Instead the system will now prefer operating system dependent memory allocation functions such as `posix_memalign`. Now, suppose this is still undesirable, and one want to use a memory allocation function not directly supported by the library. In this case, one can implement a aligned memory class (see section 7.1 for details), and then define the macro `VSMC_ALIGNED_MEOMRY_TYPE` to this class. For example,

```
class AlignedMemoryUsr
{
    public:
    static void *aligned_malloc(std::size_t n, std::size_t alignment);
    static void aligned_free(void *ptr);
};
```

```
#define VSMC_ALIGNED_MEMORY_TYPE ::AlignedMemoryUsr
// Include vSMC headers
```

In summary, the `VSMC_HAS_` macros affect what will be *defined* by the library. The `VSMC_USE_` macros affect what will be *used* by the library. And other macros give direct control of the library's implementations. In most situations, it is sufficient to use the first two to configure the features of the library. The third kind is for more advanced users. Below we details all configuration macros.

A.1 PLATFORM CHARACTERISTICS

The following macros have default values that depend on the compiler, operating system or CPU types. Their default values are therefore platform dependent. The library tries its best to determine the correct values. But in the case of errors, define these macros to override the default values.

`VSMC_HAS_X86` Determine if the CPU is an x86 compatible. In few rare occasions the library will use features known to exist on x86 CPUs, for example the little-endian memory layout, etc.

`VSMC_HAS_X86_64` Determine if the CPU is x86-64 compatible.

`VSMC_HAS_POSIX` Determine if the operating system is POSIX compatible.

`VSMC_HAS_AES_NI` Determine if the AES-NI instructions are supported by the CPU and the compiler. The RNGs based on these instructions in section 5.2.1 will are defined if this macro is true. One can include the following header directly to bypass this check,

```
#include <vsmc/rng/aes_ni.hpp>
```

`VSMC_HAS_RDRAND` Determine if the RDRAND instructions are supported by the CPU and the compiler. The RNGs based on these instructions in section 5.3 are only defined if this macro is true. One can include the following header directly to bypass this check,

```
#include <vsmc/rng/rdrand.hpp>
```

`VSMC_HAS_INT128` Determine if the compiler support 128-bits integer types.

VSMC_INTE128 The type of the 128-bits integer. Only have effects when VSMC_HAS_INT128 is true. The default is __int128.

A.2 THIRD-PARTY LIBRARIES

The following macros determine if certain third-party libraries are available. All macros with the prefix VSMC_HAS_ are define to 0 by default. All macros with the prefix VSMC_USE_ are defined to 1 if the corresponding VSMC_HAS_ macro is true. Otherwise it is defined to 0, unless stated otherwise.

VSMC_HAS_OMP Determine if OpenMP is supported.

VSMC_USE_OMP Allow the library to use OpenMP for parallelization. Currently this macro has no affect. The use of OpenMP for parallelization is explicitly through MoveOMP etc.

VSMC_HAS_OPENCL Determine if OpenCL (version 1.2 or higher) is supported.

VSMC_HAS_HDF5 Determine if the HDF5 library is available. The functions in section 7.3 are only defined if this macro is true. One can include the following header directly to bypass this check,

```
#include <vsmc/utility/hdf5.hpp>
```

VSMC_HAS_TBB Determine if the TBB library is available. The AlignedMemoryTBB class in section 7.1 and the RNGSetTBB class template in section 5.5 are only defined if this macro is true.

VSMC_USE_TBB Allow the library to use the TBB library for parallelization. Currently this macro has no affect. The use of TBB for parallelization is explicitly through MoveTBB etc.

VSMC_USE_TBB_MALLOC Allow the library to use AlignedMemoryTBB as the default memory allocation method.

VSMC_USE_TBB_TLS Allow the library to use RNGSetTBB as the default multiple RNG stream management method.

VSMC_HAS_MKL Determine if the MKL library is available. The RNGs in section 5.4 are only defined if this macro is true. One can include the following header directly to bypass this check,

```
#include <vsmc/rng/mkl.hpp>
```

VSMC_USE_MKL_CBLAS Allow the library to use the `mkl_cblas.h` header and assuming that corresponding runtime library will be linked for BLAS support.

VSMC_USE_MKL_VML Allow the library to accelerate the vector functions in section **??** using the VML component of MKL.

VSMC_USE_MKL_VSL Allow the library to use the VSL component of MKL.

VSMC_USE_ACCELERATE Allow the library to the Accelerate framework on Mac OS X for BLAS support. The `VSMC_USE_MKL_CBLAS` will take precedence over this macro. In addition, it also allows the library to use this framework to accelerate some vector functions in section **??**. The `VSMC_USE_MKL_VML` will take precedence over this macro for this purpose.

VSMC_USE_CBLAS Allow the library to the standard C interface of BLAS. The default value is `1` is either `VSMC_USE_CBLAS` or `VSMC_USE_ACCELERATE` is true. Otherwise the default value is zero. Manually define this macro to true if the `cblas.h` header is available and a compatible runtime library is linked. When this macro is tested to be false, the library will declare the BLAS Fortran routines in C itself (see below).

VSMC_BLAS_NAME and VSMC_BLAS_NAME_NO_UNDERSCORE These two macros determine how shall the BLAS Fortran routines be declared in C. The default behavior is to append an underscore to the function name. For example, `dgemv` in Fortran becomes `dgemv_` in C. If there should be no underscore, define the second macro. If the name mangling is more complicated, one can define the `VSMC_BLAS_NAME` macro directly. For example,

```
#define VSMC_BLAS_NAME(x) _##x
```

These macros only have effects if `VSMC_USE_CBLAS` is false.

VSMC_BLAS_INT The integer type of BLAS routines. The default is `MKL_INT` if `VSMC_USE_CBLAS` and `VSMC_USE_MKL_CBLAS` are both true. Otherwise it is `int`. If the BLAS interface use another integer type, such as the ILP64 interface of some implementations on LP64 platforms, then one should redefine this macro to the correct type.

A.3 RNG ENGINES

Some configuration macros of counter-based RNGs are discussed in section 5.2.

`VSMC_RNG_TYPE` The type of the alias `RNG`. The default is `ARS` if `VSMC_HAS_AES_NI` is true. Otherwise, it is `Threefry`.

`VSMC_RNG_64_TYPE` The type of the alias `RNG_64`. The default is `ARS_64` if `VSMC_HAS_AES_NI` is true. Otherwise, it is `Threefry_64`.

`VSMC_RNG_MINI_TYPE` The type of the alias `RNGMini`. The default is `Philox2x32`.

`VSMC_RNG_MINI_64_TYPE` The type of the alias `RNGMini_64`. The default is `Philox2x32_64`.

`VSMC_RNG_SET_TYPE` The type of the alias `RNGSet`. The default is `RNGSetTBB` if `VSMC_USE_TBB_TLS` is true. Otherwise, it is `RNGSetVector`. Note that, the two class templates have different default template argument. The class `RNGSetVector<>` is the same as `RNGSetVector<RNGMIni>` while the class `RNGSetTBB<>` is the same as `RNGSetTBB<RNG>`.

A.4 MEMORY ALLOCATION

`VSMC_ALIGNMENT` The default alignment for scalar types, such as `int`. More specifically, this affects types such that `std::is_scalar<T>` true. The default is `32`. This is sufficient for modern SIMD operations. This will affect the memory allocated by `Vector<T>` on the heap and `Array<T>` on the stack. The value must be a power of two and positive.

`VSMC_ALIGNMENT_MIN` The minimum alignment for all types. The default is `16`. The value must be a power of two and positive.

`VSMC_ALIGNED_MEMORY_TYPE` The default type of the `Memroy` template parameter of `Allocator` (see section 7.1). The default is `AlignedMemroyTBB` if `VSMC_USE_TBB_MALLOC` is true. Otherwise it is `AlignedMemorySYS` if `VSMC_HAS_POSIX` is true or use the MSVC compiler. Otherwise, it

is `AlignedMemorySTD`. To use other memory allocation libraries, one usually does not need to define a new class. Most such libraries provides proxies to transparently replace system allocation functions such as `posix_memalign` and `_aligned_malloc` etc. And thus it is sufficient to define this macro to `AlignedMemorySYS` and link to the proper libraries.

`VSMC_CONSTRUCT_SCALAR` Determine if scalar types shall be zero initialized upon allocation by `Allocator`. The default value is `0`. This departure from the behavior of `std::allocator`, with which the memory is always value initialized. For example,

```
std::vector<int> v(n);
```

will initialize all values to zero. In contrast,

```
Vector<int> v(n);
```

will leave the memory uninitialized. To get zero initialized vectors, one can either define this macro to `1`, or more efficiently,

```
Vector<int> v(n);
std::fill(v.begin(), v.end(), 0); // or
std::memset(v.data(), 0, sizeof(int) * v.size());
```

Most standard library implementations will pass the `std::fill` call to `std::memset`. It is strongly recommended that to leave this macro defined to zero. The `std::vector`, with which `Vector` is an alias to, calls the `construct` member of `Allocator` to initialize elements one by one, which is a huge waste of time. The library does not rely on zero initialization itself. Note that, this macro only affect *default initialization of scalars*. For class types etc., and other constructor of `std::vector`, such as

```
Vector<ClassType> c(n);
Vector<int> v(n, 2);
```

the behavior will be as expected. That is, the first will be default initialized and the second will be value initialized.

A.5 ERROR HANDLING

`VSMC_NO_RUNTIME_ASSERT` Determine if all runtime assertions shall be disabled. The default is `1` if `NDEBUG` is defined. Otherwise, it is `0`. Runtime assertions are hard errors. The program will

be terminated by `std::exit` upon failure of assertions. These are usually errors that will cause undefined behaviors.

`VSMC_NO_RUNTIME_WARNING` Determine if all runtime warnings shall be disabled. The default is `1` if `NDEBUG` is defined. Otherwise, it is `0`. Runtime warnings are soft errors. The program will not be terminated and will continue. These errors do not introduce undefined behaviors, but indicate possible misuse of library features.

`VSMC_RUNTIME_ASSERT_AS_EXCEPTION` Determine if all runtime assertions shall be turned into exceptions. The exception type is `RuntimeAssert`. The default value is `0`. It has no affect is `VSMC_NO_RUNTIME_ASSERT` is true.

`VSMC_RUNTIME_WARNING_AS_EXCEPTION` Determine if all runtime warnings shall be turned into exceptions. The exception type is `RuntimeWarning`. The default value is `0`. It has no affect is `VSMC_NO_RUNTIME_WARNING` is true.

B INTERFACING WITH C AND OTHER LANGUAGES

The library provides a set of C functions, declared in the header `vsmc.h`, and a companion runtime library. These functions expose a subset of the functionality of the C++ template library. The main purpose is for easier interfacing with other languages, instead of implementing algorithms in C itself. Of course, the later is possible and two complete C99 implementations of the simple particle filter in section **??** are shown in appendices **??** and **??**. See the reference manual for complete list of the API. In this appendix, we introduce the conventions used in the API.

B.1 ARGUMENT TYPES

For all C interfaces, `int` is always used for integer types, regardless of the C++ interfaces, except for memory functions. And `double` is always used for floating points. For example, the following method of `Seed`,

```
unsigned get();
```

is exposed to C as

```
int vsmc_seed_get(void);
```

And RNG methods where `unsigned` was expected in C++, an `int` is expected in C. If a function is a template, then only the specialization for `double` is exposed to C. In some cases, specializations for `int` and `unsigned char` are also exposed. The use of `int` as the universal integer type is for compatibility consideration. Again, the main purpose is for interfacing with other languages instead of implementing algorithms in C itself. For example, Fortran does not have native unsigned integer types. This is the main limitation of the C API. If integer values, such as sample size, larger than the largest value can be represented by `int` is needed. One can write their own C wrapper. Note that, when an unsigned integer type, such as `std::size_t` is expected in the C++ interface, and the input for the C function was a negative value, it is silently converted to a positive value through `static_cast`. Therefore `-1` will be treated as a very big number instead of emit an error. The user is responsible to check input argument ranges. This limitation is not so significant in practice. The types `int` and `double` are as portable as one would get for interfacing with other languages. For instance, R's core storage modes for atomic numeric vectors are either `int` or `double`. Similarly, wherever an iteration is expected by the C++ interface, pointers to `int` and `double` will expected by the C interface, with possible `const` qualifiers. Enumerators are also defined with the same values as in C++, but with a `vSMC` prefix. For example, `Multinomial` becomes `vSMCMultinomial` and the type `ResampleScheme` becomes `vSMCResampleScheme`. The other two enumerator types are `vSMCMatrixLayout` and `vSMCMonitorStage`. Last, when a function object is expected in the C++ interface, a function pointer will be expected in the C interface. For example, `Sampler::move_type` is translated to,

```
typedef int (*vsmc_sampler_move_type)(int, vsmc_particle);
```

B.2 CLASSES AND METHODS

A few core classes are accessible from this API. They are listed in table B.1. Table **??** lists additional features of the library exposed to the C API apart from the core classes listed above. With the exception of

```
typedef {
    double *state;
    int id;
} vsmc_single_particle;
```

all the C types in the table are only wrappers around a pointer. For example,

```
typedef {
    void *ptr;
} vsmc_sampler;
```

The pointer points to the address of a C++ object of the corresponding type.

Each C type objects can be created by an allocation function and destroyed by a deallocation function. For example,

```
vsmc_sampler sampler = vsmc_sampler_new(n, dim, vSMCMultinomial, 0.5);
```

corresponds to the C++ calls,

```
Sampler<T> sampler(n, Multinomial, 0.5);
sampler.particle.value().resize_dim(dim);
```

The memory can later be freed by

```
vsmc_sampler_delete(sampler);
sampler.ptr = NULL;
```

Most non-static methods can be accesses through C. The corresponding functions names takes the form *`class_method`*, where *`class`* is a type name, for example `vsmc_sampler`; and *`method`* is the name of the method. The first argument will be a *`class`* type object, and the following arguments are the same as in C++. When there are multiple overloading of a method, the most general form is provided. For example,

```
StateMatrix<RowMajor, Dynamic, double> s;
s.resize(n);
s.resize(n, dim);
```

are two overloading of the `resize` method. In C, only

```
vsmc_state_matrix s = vsmc_state_matrix_new(n, dim);
vsmc_state_matrix_resize(s, n, dim);
```

are accessible. In a few cases, when a getter and a setter are overloaded, `get` and `set` are inserted into the method name. For example,

```
double thres = sampler.threshold();
sampler.threshold(thres);
```

are translated to C as

```
double thres = vsmc_sampler_get_threshold(sampler);
vsmc_sampler_set_threshold(sampler, thres);
```

C++ types	C types
`StateMatrix<RowMajor, Dynamic, double>`	`vsmc_state_matrix`
`Weight`	`vsmc_weight`
`Particle<T>`	`vsmc_particle`
`SingleParticle<T>`	`vsmc_single_particle`
`Sampler<T>`	`vsmc_sampler`
`Monitor<T>`	`vsmc_monitor`
`RNG`	`vsmc_rng`

TABLE B.1 C API types

Feature	Notes
SMP backends	Section **??**
Resamling algorithms	Section 6.1
Ordered uniform random numbers	Section 6.2
Seeding	Section 5.2.5
Vectorized random number generating	Section 5.6
Aligned memory allocation	Section 7.1
Covariance matrix	Section 7.2
Process program command line options	Section 7.5
Display program progress	Section 7.6
Stop watch	Section 7.7
Registering C++11 RNG as MKL BRNG	Reference manual
Random walk	Reference manual

TABLE B.2 Features accessible from C

C PERFORMANCE DATA

This appendix shows all the tables of performance data.

	Loop			`rand`		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
`mt19937`	3.01	1.78	3.48	3.20	1.87	4.07
`mt19937_64`	1.62	1.34	1.67	1.88	1.25	1.69
`minstd_rand0`	4.99	5.27	6.54	5.02	5.28	6.10
`minstd_rand`	4.10	4.89	5.88	4.08	4.88	5.28
`ranlux24_base`	6.43	6.62	7.65	6.36	6.75	6.87
`ranlux48_base`	4.36	4.02	4.93	4.32	4.01	4.84
`ranlux24`	69.8	65.4	77.3	69.8	65.7	65.0
`ranlux48`	157	141	176	157	140	159
`knuth_b`	15.8	21.2	13.6	12.4	20.9	13.6

TABLE C.1 Performance of standard library RNG

	Loop			rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
AES128x1	1.05	5.15	4.57	0.77	3.98	4.04
AES128x2	1.26	2.79	3.01	0.70	2.32	2.51
AES128x4	0.85	1.71	1.75	0.65	1.30	1.38
AES128x8	1.23	1.24	1.40	0.85	0.83	0.84
AES128x1_64	0.86	5.00	4.42	0.76	3.95	4.04
AES128x2_64	0.80	2.53	2.81	0.70	2.33	2.51
AES128x4_64	0.75	1.54	1.63	0.65	1.30	1.37
AES128x8_64	1.15	1.11	1.26	0.85	0.83	0.84

TABLE C.2 Performance of AES128Engine

	Loop			rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
AES192x1	1.74	5.22	5.20	0.89	4.56	4.64
AES192x2	1.44	3.10	3.63	0.82	2.64	2.79
AES192x4	0.99	1.88	1.89	0.76	1.46	1.52
AES192x8	1.31	1.33	1.44	0.94	0.93	0.93
AES192x1_64	1.09	5.01	5.26	0.89	4.55	4.63
AES192x2_64	0.91	2.83	3.27	0.82	2.62	2.93
AES192x4_64	0.88	1.69	1.88	0.76	1.46	1.61
AES192x8_64	1.24	1.20	1.39	0.93	0.92	0.98

TABLE C.3 Performance of AES192Engine

	Loop			`rand`		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
`AES256x1`	1.97	6.38	5.80	1.08	5.21	5.23
`AES256x2`	1.32	3.37	3.54	0.95	2.93	3.06
`AES256x4`	1.13	2.01	2.05	0.88	1.62	1.69
`AES256x8`	1.40	1.48	1.64	1.02	1.01	1.02
`AES256x1_64`	1.36	6.20	5.66	1.07	5.15	5.24
`AES256x2_64`	1.06	3.13	3.46	0.94	2.94	3.08
`AES256x4_64`	1.02	1.85	1.94	0.88	1.60	1.68
`AES256x8_64`	1.33	1.28	1.45	1.05	1.00	1.02

Table C.4 Performance of `AES256Engine`

	Loop			`rand`		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
`ARSx1`	0.71	4.01	4.17	0.47	2.47	2.50
`ARSx2`	0.95	2.54	2.81	0.44	1.57	1.73
`ARSx4`	0.62	1.57	1.67	0.40	0.93	1.00
`ARSx8`	0.63	1.14	1.27	0.34	0.62	0.69
`ARSx1_64`	0.54	3.83	4.06	0.46	2.45	2.52
`ARSx2_64`	0.47	2.40	2.63	0.43	1.56	1.73
`ARSx4_64`	0.47	1.48	1.54	0.39	0.92	1.00
`ARSx8_64`	0.50	1.05	1.07	0.34	0.61	0.69

Table C.5 Performance of `ARSEngine`

	Loop			`rand`		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
`Philox2x32`	7.74	10.1	3.41	6.42	8.48	2.15
`Philox4x32`	3.08	13.2	5.05	1.87	4.52	1.83
`Philox2x64`	1.97	3.24	2.47	1.04	1.98	1.49
`Philox4x64`	1.79	4.06	5.94	0.96	1.77	1.09
`Philox2x32_64`	6.87	9.97	7.83	5.95	8.49	5.99
`Philox4x32_64`	2.71	13.2	4.90	1.80	4.49	2.67
`Philox2x64_64`	1.63	3.08	2.95	1.00	1.97	2.14
`Philox4x64_64`	1.54	3.96	6.13	0.92	1.81	1.55

TABLE C.6 Performance of `PhiloxEngine`

	Loop			`rand`		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
`Threefry2x32`	9.79	9.54	8.09	8.61	8.76	6.99
`Threefry4x32`	6.51	5.52	7.16	5.77	4.93	5.68
`Threefry2x64`	5.54	3.15	3.15	4.66	2.55	2.22
`Threefry4x64`	3.52	2.20	3.89	3.02	1.88	3.37
`Threefry8x64`	3.49	1.80	2.68	2.38	1.46	2.18
`Threefry16x64`	2.92	4.92	7.29	2.53	4.58	6.87
`Threefry2x32_64`	9.44	3.33	7.79	8.60	8.60	7.01
`Threefry4x32_64`	6.32	5.13	6.98	5.82	4.93	5.62
`Threefry2x64_64`	5.08	3.09	2.74	4.69	2.67	2.26
`Threefry4x64_64`	3.36	2.20	3.69	3.00	1.96	2.96
`Threefry8x64_64`	2.93	1.76	2.55	2.38	1.53	2.13
`Threefry16x64_64`	2.80	4.82	7.15	2.52	4.54	6.83

TABLE C.7 Performance of `ThreefryEngine`

	Loop			rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
RDRAND16	121	119	119	120	119	118
RDRAND32	60.3	59.8	59.6	58.8	59.8	58.8
RDRAND64	32.1	29.9	30.6	30.5	29.9	29.3

Table C.8 Performance of non-deterministic RNG

	Loop			rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
MKL_MCG59	1.30	0.75	0.97	0.34	0.32	0.32
MKL_MCG59_64	0.64	0.64	0.87	0.32	0.32	0.32
MKL_MT19937	1.12	0.63	0.88	0.28	0.27	0.28
MKL_MT19937_64	0.54	0.52	0.76	0.27	0.27	0.27
MKL_MT2203	1.12	0.63	0.86	0.19	0.19	0.19
MKL_MT2203_64	0.52	0.52	0.76	0.19	0.19	0.20
MKL_STMT19937	1.12	0.62	0.86	0.17	0.16	0.17
MKL_STMT19937_64	0.52	0.52	0.74	0.17	0.16	0.17
MKL_NONDETERM	30.1	29.2	29.4	29.4	29.0	29.1
MKL_NONDETERM_64	30.4	29.9	30.2	30.0	29.4	29.8
MKL_PHILOX4X32X10	1.51	1.02	1.25	0.55	0.54	0.55
MKL_PHILOX4X32X10_64	0.92	0.92	1.13	0.54	0.55	0.54
MKL_ARS5	1.35	0.79	1.02	0.32	0.30	0.30
MKL_ARS5_64	0.68	0.68	0.91	0.30	0.30	0.30

Table C.9 Performance of MKL RNG

Distribution	STD	vSMC	rand	MKL
Arcsine(0,1)	–	33.0	9.83	8.99
Cauchy(0,1)	80.2	43.4	13.7	7.27
Exponential(1)	67.2	59.0	7.68	5.72
ExtremeValue(0,1)	109	64.9	14.0	11.1
Laplace(0,1)	61.0	45.3	9.30	9.00
Logistic(0,1)	–	41.2	13.5	13.1
Pareto(1,1)	–	54.5	12.0	10.8
Rayleigh(1)	–	73.7	11.4	9.04
UniformReal(-0.5,0.5)	17.2	4.78	2.76	1.22
Weibull(1,1)	138	28.8	7.70	6.37

TABLE C.10 Performance of distributions using the inverse method

Distribution	STD	vSMC	rand	MKL
Levy(0,1)	–	51.1	22.1	18.8
Lognormal(0,1)	86.3	74.4	18.1	13.3
Normal(0,1)	66.7	57.4	13.9	10.3

TABLE C.11 Performance of Normal and related distributions

Distribution	STD	vSMC	rand	MKL
Gamma(1,1)	46.6	40.7	7.73	6.43
Gamma(0.1,1)	117	114	34.2	32.4
Gamma(0.5,1)	150	146	60.6	47.1
Gamma(0.7,1)	158	153	45.2	34.4
Gamma(0.9,1)	154	126	34.6	26.4
Gamma(1.5,1)	199	130	35.5	27.0
Gamma(15,1)	179	130	33.9	24.6

TABLE C.12 Performance of Gamma distribution

Distribution	STD	vSMC	rand	MKL
ChiSquared(2)	47.2	43.4	7.75	6.41
ChiSquared(0.2)	119	117	35.0	32.5
ChiSquared(1)	151	151	61.0	46.5
ChiSquared(1.4)	159	160	44.9	34.4
ChiSquared(1.8)	153	131	34.7	26.3
ChiSquared(3)	195	131	35.5	26.7
ChiSquared(30)	183	139	34.3	25.2

TABLE C.13 Performance of χ^2 distribution

Distribution	STD	vSMC	rand	MKL
`FisherF(1,1)`	306	336	143	112
`FisherF(1,0.5)`	266	299	126	105
`FisherF(1,1.5)`	301	325	135	97.2
`FisherF(1,3)`	337	315	111	89.0
`FisherF(1,30)`	317	315	107	87.1
`FisherF(0.5,1)`	269	297	125	105
`FisherF(0.5,0.5)`	243	278	109	96.0
`FisherF(0.5,1.5)`	276	307	119	89.3
`FisherF(0.5,3)`	308	296	94.0	81.2
`FisherF(0.5,30)`	290	292	89.8	78.6
`FisherF(1.5,1)`	301	323	134	96.6
`FisherF(1.5,0.5)`	273	301	118	89.2
`FisherF(1.5,1.5)`	308	331	129	79.7
`FisherF(1.5,3)`	345	316	104	73.0
`FisherF(1.5,30)`	319	314	99.0	69.4
`FisherF(3,1)`	338	316	111	89.2
`FisherF(3,0.5)`	312	290	94.8	81.2
`FisherF(3,1.5)`	352	324	105	73.4
`FisherF(3,3)`	380	305	79.5	64.1
`FisherF(3,30)`	358	303	74.3	61.0
`FisherF(30,1)`	348	338	109	88.4
`FisherF(30,0.5)`	308	291	90.1	78.5
`FisherF(30,1.5)`	331	315	100	69.8
`FisherF(30,3)`	360	304	74.2	61.3
`FisherF(30,30)`	343	303	69.2	57.6

TABLE C.14 Performance of Fisher's F-distribution

Distribution	STD	vSMC	rand	MKL
`StudentT(2)`	105	117	36.5	29.8
`StudentT(0.2)`	169	194	64.0	59.9
`StudentT(1)`	206	233	95.2	76.5
`StudentT(1.4)`	213	242	95.4	62.6
`StudentT(1.8)`	212	216	68.3	52.6
`StudentT(3)`	234	221	63.8	53.5
`StudentT(30)`	223	226	59.0	50.2

TABLE C.15 Performance of Student's *t*-distribution

Distribution	STD	vSMC	rand	MKL
`Beta(0.5,0.5)`	349	40.1	13.3	36.8
`Beta(1,1)`	87.3	11.8	2.63	9.01
`Beta(1,0.5)`	218	60.2	14.4	8.96
`Beta(1,1.5)`	382	59.9	15.7	9.08
`Beta(0.5,1)`	217	60.0	13.9	9.05
`Beta(1.5,1)`	375	58.7	15.4	9.14
`Beta(1.5,1.5)`	605	160	46.4	38.3
`Beta(0.3,0.3)`	318	136	52.8	33.3
`Beta(0.9,0.9)`	358	154	153	47.7
`Beta(1.5,0.5)`	463	184	183	59.0
`Beta(0.5,1.5)`	466	183	183	58.9

TABLE C.16 Performance of Beta distribution