Yan Zhou

vSMC – Scalable Monte Carlo

Second edition (version develop)
April 22, 2016

CONTENTS

1 SEQUENTIAL MONTE CARLO

1.1 INTRODUCTION

Sequential Monte Carlo (SMC) methods are a class of sampling algorithms that combine importance sampling and resampling. They have been primarily used as "particle filters" to solve optimal filtering problems; see, for example, Cappé, Godsill, and Moulines (2007) and Doucet and Johansen (2011) for recent reviews. They are also used in a static setting where a target distribution is of interest, for example, for the purpose of Bayesian modeling. This was proposed by Del Moral, Doucet, and Jasra (2006b) and developed by Peters (2005) and Del Moral, Doucet, and Jasra (2006a). This framework involves the construction of a sequence of artificial distributions on spaces of increasing dimensions which admit the distributions of interest as particular marginals.

SMC algorithms are perceived as being difficult to implement while general tools were not available until the development of Johansen (2009), which provided a general framework for implementing SMC algorithms. SMC algorithms admit natural and scalable parallelization. However, there are only parallel implementations of SMC algorithms for many problem specific applications, usually associated with specific SMC related researches. Lee et al. (2010) studied the parallelization of SMC algorithms on GPUs with some generality. There are few general tools to implement SMC algorithms on parallel hardware though multicore CPUs are very common today and computing on specialized hardware such as GPUs are more and more popular.

The purpose of the current work is to provide a general framework for implementing SMC algorithms on both sequential and parallel hardware. There are two main goals of the presented framework. The first is reusability. It will be demonstrated that the same implementation source code can be used to build a serialized sampler, or using different programming models (for example, OpenMP and Intel Threading Building Blocks) to build parallelized samplers for multicore CPUs. The second is extensibility. It is possible to write a backend for vSMC to use new parallel programming models while reusing existing implementations. It is also possible to enhance the library to improve performance for specific applications. Almost all components of the library can be reimplemented by users and thus if the default implementation is not suitable for a specific application, they can be replaced while being integrated with other components seamlessly.

1.2 SEQUENTIAL IMPORTANCE SAMPLING AND RESAMPLING

Importance sampling is a technique which allows the calculation of the expectation of a function φ with respect to a distribution π using samples from some other distribution η with respect to which π is absolutely

continuous, based on the identity,

$$\mathbb{E}_\pi[\varphi(X)] = \int \varphi(x)\pi(x)\ \mathrm{d}\,x = \int \frac{\varphi(x)\pi(x)}{\eta(x)}\eta(x)\ \mathrm{d}\,x = \mathbb{E}_\eta\Big[\frac{\varphi(X)\pi(X)}{\eta(X)}\Big] \tag{1.1}$$

And thus, let $\{X^{(i)}\}_{i=1}^N$ be samples from η, then $\mathbb{E}_\pi[\varphi(X)]$ can be approximated by

$$\hat{\varphi}_1 = \frac{1}{N}\sum_{i=1}^N \frac{\varphi(X^{(i)})\pi(X^{(i)})}{\eta(X^{(i)})} \tag{1.2}$$

In practice π and η are often only known up to some normalizing constants, which can be estimated using the same samples. Let $w^{(i)} = \pi(X^{(i)})/\eta(X^{(i)})$, then we have

$$\hat{\varphi}_2 = \frac{\sum_{i=1}^N w^{(i)}\varphi(X^{(i)})}{\sum_{i=1}^N w^{(i)}} \tag{1.3}$$

or

$$\hat{\varphi}_3 = \sum_{i=1}^N W^{(i)}\varphi(X^{(i)}) \tag{1.4}$$

where $W^{(i)} \propto w^{(i)}$ and are normalized such that $\sum_{i=1}^N W^{(i)} = 1$.

Sequential importance sampling (SIS) generalizes the importance sampling technique for a sequence of distributions $\{\pi_t\}_{t\geq 0}$ defined on spaces $\{\prod_{k=0}^t E_k\}_{t\geq 0}$. At time $t = 0$, sample $\{X_0^{(i)}\}_{i=1}^N$ from η_0 and compute the weights $W_0^{(i)} \propto \pi_0(X_0^{(i)})/\eta_0(X_0^{(i)})$. At time $t \geq 1$, each sample $X_{0:t-1}^{(i)}$, usually termed *particles* in the literature, is extended to $X_{0:t}^{(i)}$ by a proposal distribution $q_t(\cdot|X_{0:t-1}^{(i)})$. And the weights are recalculated by $W_t^{(i)} \propto \pi_t(X_{0:t}^{(i)})/\eta_t(X_{0:t}^{(i)})$ where

$$\eta_t(X_{0:t}^{(i)}) = \eta_{t-1}(X_{0:t-1}^{(i)})q_t(X_{0:t}^{(i)}|X_{0:t-1}^{(i)}) \tag{1.5}$$

and thus

$$\begin{aligned} W_t^{(i)} \propto \frac{\pi_t(X_{0:t}^{(i)})}{\eta_t(X_{0:t}^{(i)})} &= \frac{\pi_t(X_{0:t}^{(i)})\pi_{t-1}(X_{0:t-1}^{(i)})}{\eta_{t-1}(X_{0:t-1}^{(i)})q_t(X_{0:t}^{(i)}|X_{0:t-1}^{(i)})\pi_{t-1}(X_{0:t-1}^{(i)})} \\ &= \frac{\pi_t(X_{0:t}^{(i)})}{q_t(X_{0:t}^{(i)}|X_{0:t-1}^{(i)})\pi_{t-1}(X_{0:t-1}^{(i)})}W_{t-1}^{(i)} \end{aligned} \tag{1.6}$$

and importance sampling estimate of $\mathbb{E}_{\pi_t}[\varphi_t(X_{0:t})]$ can be obtained using $\{W_t^{(i)}, X_{0:t}^{(i)}\}_{i=1}^N$.

However this approach fails as t becomes large. The weights tend to become concentrated on a few particles as the discrepancy between η_t and π_t becomes larger. Resampling techniques are applied such that, a new particle system $\{\bar{W}_t^{(i)}, \bar{X}_{0:t}^{(i)}\}_{i=1}^M$ is obtained with the property,

$$\mathbb{E}\Big[\sum_{i=1}^M \bar{W}_t^{(i)}\varphi_t(\bar{X}_{0:t}^{(i)})\Big] = \mathbb{E}\Big[\sum_{i=1}^N W_t^{(i)}\varphi_t(X_{0:t}^{(i)})\Big] \tag{1.7}$$

In practice, the resampling algorithm is usually chosen such that $M = N$ and $\bar{W}^{(i)} = 1/N$ for $i = 1, \dots, N$. Resampling can be performed at each time t or adaptively based on some criteria of the discrepancy. One popular quantity used to monitor the discrepancy is *effective sample size* (ESS), introduced by Liu and Chen (1998), defined as

$$\text{ESS}_t = \frac{1}{\sum_{i=1}^{N} (W_t^{(i)})^2} \tag{1.8}$$

where $\{W_t^{(i)}\}_{i=1}^N$ are the normalized weights. And resampling can be performed when ESS $\leq \alpha N$ where $\alpha \in [0, 1]$.

The common practice of resampling is to replicate particles with large weights and discard those with small weights. In other words, instead of generating a random sample $\{\bar{X}_{0:t}^{(i)}\}_{i=1}^N$ directly, a random sample of integers $\{R^{(i)}\}_{i=1}^N$ is generated, such that $R^{(i)} \geq 0$ for $i = 1, \dots, N$ and $\sum_{i=1}^N R^{(i)} = N$. And each particle value $X_{0:t}^{(i)}$ is replicated for $R^{(i)}$ times in the new particle system. The distribution of $\{R^{(i)}\}_{i=1}^N$ shall fulfill the requirement of Equation (1.7). One such distribution is a multinomial distribution of size N and weights $(W_t^{(i)}, \dots, W_t^{(N)})$. See Douc, Cappé, and Moulines (2005) for some commonly used resampling algorithms.

1.3 SMC SAMPLERS

SMC samplers allow us to obtain, iteratively, collections of weighted samples from a sequence of distributions $\{\pi_t\}_{t\geq 0}$ over essentially any random variables on some spaces $\{E_t\}_{t\geq 0}$, by constructing a sequence of auxiliary distributions $\{\tilde{\pi}_t\}_{t\geq 0}$ on spaces of increasing dimensions, $\tilde{\pi}_t(x_{0:t}) = \pi_t(x_t) \prod_{s=0}^{t-1} L_s(x_{s+1}, x_s)$, where the sequence of Markov kernels $\{L_s\}_{s=0}^{t-1}$, termed backward kernels, is formally arbitrary but critically influences the estimator variance. See Del Moral, Doucet, and Jasra (2006b) for further details and guidance on the selection of these kernels.

Standard sequential importance sampling and resampling algorithms can then be applied to the sequence of synthetic distributions, $\{\tilde{\pi}_t\}_{t\geq 0}$. At time $t-1$, assume that a set of weighted particles $\{W_{t-1}^{(i)}, X_{0:t-1}^{(i)}\}_{i=1}^N$ approximating $\tilde{\pi}_{t-1}$ is available, then at time t, the path of each particle is extended with a Markov kernel say, $K_t(x_{t-1}, x_t)$ and the set of particles $\{X_{0:t}^{(i)}\}_{i=1}^N$ reach the distribution $\eta_t(X_{0:t}^{(i)}) = \eta_0(X_0^{(i)}) \prod_{k=1}^t K_t(X_{t-1}^{(i)}, X_t^{(i)})$, where η_0 is the initial distribution of the particles. To correct the discrepancy between η_t and $\tilde{\pi}_t$, Equation (1.6) is applied and in this case,

$$W_t^{(i)} \propto \frac{\tilde{\pi}_t(X_{0:t}^{(i)})}{\eta_t(X_{0:t}^{(i)})} = \frac{\pi_t(X_t^{(i)}) \prod_{s=0}^{t-1} L_s(X_{s+1}^{(i)}, X_s^{(i)})}{\eta_0(X_0^{(i)}) \prod_{k=1}^t K_t(X_{t-1}^{(i)}, X_t^{(i)})} \propto \tilde{w}_t(X_{t-1}^{(i)}, X_t^{(i)}) W_{t-1}^{(i)} \tag{1.9}$$

where $\tilde{w}_t$, termed the *incremental weights*, are calculated as,

$$\tilde{w}_t(X_{t-1}^{(i)}, X_t^{(i)}) = \frac{\pi_t(X_t^{(i)}) L_{t-1}(X_t^{(i)}, X_{t-1}^{(i)})}{\pi_{t-1}(X_{t-1}^{(i)}) K_t(X_{t-1}^{(i)}, X_t^{(i)})} \tag{1.10}$$

If π_t is only known up to a normalizing constant, say $\pi_t(x_t) = \gamma_t(x_t)/Z_t$, then we can use the *unnormalized* incremental weights

$$w_t(X_{t-1}^{(i)}, X_t^{(i)}) = \frac{\gamma_t(X_t^{(i)})L_{t-1}(X_t^{(i)}, X_{t-1}^{(i)})}{\gamma_{t-1}(X_{t-1}^{(i)})K_t(X_{t-1}^{(i)}, X_t^{(i)})} \tag{1.11}$$

for importance sampling. Further, with the previously *normalized* weights $\{W_{t-1}^{(i)}\}_{i=1}^N$, we can estimate the ratio of normalizing constant Z_t/Z_{t-1} by

$$\frac{\hat{Z}_t}{Z_{t-1}} = \sum_{i=1}^{N} W_{t-1}^{(i)} w_t(X_{t-1}^{(i)}, X_t^{(i)}) \tag{1.12}$$

Sequentially, the normalizing constant between initial distribution π_0 and some target π_T, $T \geq 1$ can be estimated. See Del Moral, Doucet, and Jasra (2006b) for details on calculating the incremental weights. In practice, when K_t is invariant to π_t, and an approximated suboptimal backward kernel

$$L_{t-1}(x_t, x_{t-1}) = \frac{\pi(x_{t-1})K_t(x_{t-1}, x_t)}{\pi_t(x_t)} \tag{1.13}$$

is used, the unnormalized incremental weights will be

$$w_t(X_{t-1}^{(i)}, X_t^{(i)}) = \frac{\gamma_t(X_{t-1}^{(i)})}{\gamma_{t-1}(X_{t-1}^{(i)})}. \tag{1.14}$$

1.4 RELATED ALGORITHMS

Some other commonly used sequential Monte Carlo algorithms can be viewed as special cases of algorithms introduced above. The annealed importance sampling (AIS; Neal (2001)) can be viewed as SMC samplers without resampling. Particle filters as seen in the physics and signal processing literature, can also be interpreted as the sequential importance sampling and resampling algorithms. See Doucet and Johansen (2011) for a review of this topic.

2.1 CONVENTIONS

All classes that are accessible to users are within the name space `vsmc`. Class names are in `CamelCase` and function names and class members are in `small_cases`. In the remaining of this guide, we will omit the `vsmc::` name space qualifiers. We will use "function" for referring to name space scope functions and "method" for class member functions.

2.2 GETTING AND INSTALLING THE LIBRARY

The library is hosted at GitHub[1]. This is a header only C++ template library. To install the library just move the contents of the `include` directory into a proper place, e.g., `/usr/local/include` on Unix-alike systems. This library requires working C++11, BLAS and LAPACK implementations. Standard C interface headers for the later two (`cblas.h` and `lapacke.h`) are required. Intel Threading Building Blocks[2] (TBB), Intel Math Kernel Library[3] (MKL) and HDF5[4] are optional third-party libraries. One need to define the configuration macros `VSMC_HAS_TBB`, `VSMC_HAS_MKL` and `VSMC_HAS_HDF5`, respectively, to nonzero values before including any vSMC headers to make their existence known to the library.

2.3 CONCEPTS

The library is structured around a few core concepts. A sampler is responsible for running an algorithm. It contains a particle system and operations on it. A particle system is formed by the states $\{X^{(i)}\}_{i=1}^N$ and weights $\{W^{(i)}\}_{i=1}^N$. This system will also be responsible for resampling. All user defined operations are to be applied to the whole system. These are "initialization" and "moves" which are applied before resampling, and "MCMC" moves which are applied after resampling. These operations do not have to be MCMC kernels. They can be used for any purpose that suits the particular algorithm. Most statistical inferences requires calculation of $\sum_{i=1}^N W^{(i)} \varphi(X^{(i)})$ for some function φ. This can be carried out along each sampler iteration by a monitor. Table 2.1 lists these concepts and the corresponding types in the library. Each of them are introduced in detail in the following sections.

[1] https://github.com/zhouyan/vSMC
[2] https://www.threadingbuildingblocks.org
[3] https://software.intel.com/en-us/intel-mkl
[4] http://www.hdfgroup.org

Concept	Type
State, $\{X^{(i)}\}_{i=1}^N$	`T`, user defined
Weight, $\{W^{(i)}\}_{i=1}^N$	`Weight`
Particle, $\{W^{(i)}, X^{(i)}\}_{i=1}^N$	`Particle<T>`
Single particle, $\{W^{(i)}, X^{(i)}\}$	`SingleParticle<T>`
Sampler	`Sampler<T>`
Initialization	`Sampler<T>::init_type`, user defined
Move	`Sampler<T>::move_type`, user defined
MCMC	`Sampler<T>::mcmc_type`, user defined
Monitor	`Monitor<T>`

Table 2.1 Core concepts of the library

2.3.1 *State*

The library gives users the maximum flexibility of how the states $\{X^{(i)}\}_{i=1}^N$ shall be stored and structured. Any class type with a constructor that takes a single integer value, the number of particles, as its argument, and a method named `copy` is acceptable. For example,

```
1  class T
2  {
3      public:
4      T(std::size_t N);

5      template <typename IntType>
6      void copy(std::size_t N, IntType *index)
7      {
8          for (std::size_t i = 0; i != N; ++i) {
9              // Let a_i = index[i], set X^(i) = X^(a_i)
10         }
11     }
12 };
```

How the state values are actually stored and accessed are entirely up to the user. The method `copy` is necessary since the library assumes no knowledge of the internal structure of the state. And thus it cannot perform the last step of a resampling algorithm, which makes copies of particles with larger weights and eliminate those with smaller weights.

The `StateMatrix` class template

For most applications, the values can be stored within an N by d matrix, where d is the dimension of the state. The library provides a convenient class template for this situation,

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

where `Layout` is either `RowMajor` or `ColMajor`, which specifies the matrix storage layout; `Dim` is a non-negative integer value. If `Dim` is zero, then the dimension may be changed at runtime. If it is positive, then the dimension is fixed and cannot be changed at runtime. The last template parameter `T` is the C++ type of state space. The following constructs an object of this class,

```
StateMatrix<ColMajor, Dynamic, double> s(N);
```

where `Dynamic` is just an enumerator with value zero. We can specify the dimension at runtime through the method `s.resize_dim(d)`. Note that, if the template parameter `Dim` is positive, then this call results in a compile-time error.

To access X_{ij}, the value of the state of the i^{th} particle at the j^{th} coordinate, one can use the method `s.state(i,j)`. The method `s.data()` returns a pointer to the beginning of the matrix. If `Layout` is `RowMajor`, then the method `s.row_data(i)` returns a pointer to the beginning of the i^{th} row. If `Layout` is `ColMajor`, then the method `s.col_data(j)` returns a pointer to the beginning of the j^{th} column. These methods help interfacing with numerical libraries, such as BLAS.

Apart from `resize_dim`, the matrix can also be resized by the sample size or both. `s.resize(N, dim)` is the most general form. Let n be the minimum of the original and new sample sizes, and d be the minimum of the original and new dimensions. The n by d matrix at the upper left corner of the original matrix is preserved. If the matrix is enlarged in either direction, new values will be inserted and default initialized. Note that, if the template parameter `Dim` is positive, then this call results in a compile-time error. The method call `s.resize(N)` is equivalent to `s.resize(N, s.dim())` except that it never generate a compile-time error. For more sophisticated resizing, use the `copy` method, which will create a new set of states according to the index vector.

Choice between matrix storage layout If one needs to access the matrix frequently row by row or column by column, then the choice of the first template parameter `Layout` is obvious. The `copy` method, which is used by resampling algorithms, is more efficient with `RowMajor`. The difference is more significant with large sample size, high dimension or both. The performance of resizing, either explicitly or implicitly through `copy`, depends on both the matrix storage layout, the new sample size and dimension. If the dimension does not change, it is more efficient to use `RowMajor`. If the sample size does not change, it is more efficient to use the `ColMajor`. If both of them are changed, then the performance depends on the original and new sizes.

2.3.2 *Weight*

The vector of weights $\{W^{(i)}\}_{i=1}^N$ is abstracted in the library by the `Weight` class. The following constructs an object of this class,

```
Weight w(N);
```

There are a few methods for accessing the weights,

```
w.ess();          // Get ESS
w.set_equal();    // Set W^(i) = 1/N
```

The weights can be manipulated, given a vector of length N, say v,

```
w.set(v);         // Set W^(i) ∝ v^(i)
w.mul(v);         // Set W^(i) ∝ W^(i) v^(i)
w.set_log(v);     // Set log W^(i) = v^(i) + const.
w.add_log(v);     // Set log W^(i) = log W^(i) + v^(i) + const.
```

The method `w.data()` returns a pointer to the normalized weights. It is important to note that the weights are always normalized and all mutable methods only allow access to $\{W^{(i)}\}_{i=1}^N$ as a whole.

2.3.3 *Particle*

A particle system is composed of both the state values, which is of user defined type, say `T`, and the weights. The following constructs an object of class `Particle<T>`,

```
Particle<T> particle(N);
```

The method `particle.value()` returns the type `T` object. The object containing the weights is returned by `particle.weight()`. Its type is `Particle<T>::weight_type`, whose definition depends on the type `T`. See section 3.2 for more details. If the user does not do something special as shown in that section, then the default type is `Weight`. They are constructed with the same integer value N when the above constructor is invoked.

For Monte Carlo algorithm, random number generators (RNG) will be used frequently. The user is free to use whatever RNG mechanism as they see fit. However, one common issue encountered in practice is how to maintain independence of the RNG streams between function calls. For example, consider below a function that manipulates some state values,

```
void function(double &x)
{
    std::mt19937 rng;
```

```
4     std::normal_distribution<double> rnorm(0, 1);
5     x = rnorm(rng);
6 }
```

Every call of this function will give x exactly the same value. This is hardly what the user intended. One might consider an global RNG or one as class member data. For example,

```
1 std::mt19937 rng;

2 void function(double &x)
3 {
4     std::normal_distribution<double> rnorm(0, 1);
5     x = rnorm(rng);
6 }
```

This will work fine as long as the function is never called by two threads at the same time. However, SMC algorithms are natural candidates to parallelization. Therefore, the user will need to either lock the RNG, which degenerates the performance, or construct different RNGs for different threads. The later, though ensures thread-safety, has other issues. For example, consider

```
1 std::mt19937 rng1(s1); // For thread i_1 with seed s_1
2 std::mt19937 rng2(s2); // For thread i_2 with seed s_2
```

where the seeds $s_1 \neq s_2$. It is difficult to ensure that the two streams generated by the two RNGs are independent. Common practice for parallel RNG is to use sub-streams or leap-frog algorithms. Without going into any further details, it is sufficient to say that this is perhaps not a problem that most users bother to solve.

The library provides a simple solution to this issue. The method `particle.rng(i)` returns a reference to an RNG that conforms to the C++11 uniform RNG concept. It can be called from different threads at the same time, for example,

```
1 auto &rng1 = particle.rng(i1); // Called from thread i_1
2 auto &rng2 = particle.rng(i2); // Called from thread i_2
```

If $i_1 \neq i_2$, then the subsequent use of the two RNGs are guaranteed to be thread-safe. In addition, they will produce independent streams. If TBB is available to the library, then it is also thread-safe even if $i_1 = i_2$. One can write functions that process each particle, for example,

```
1 void function(std::size_t i)
2 {
3     auto &rng = particle.rng(i);
4     // Process the particle i using rng
5 }
```

And if later this function is called from a parallelized environment, it is still thread-safe and produce desired statistical results. The details of the RNG system are documented later in chapter 7.

Resize the particle system

The particle system can be resized. However, the library does not provide a simple `resize` method that works the same way as `std::vector`, etc. When a particle system is resized, it is often done through some deterministic or stochastic algorithms. At least some particles will be preserved and possibly replicated. The `Particle<T>` class has a few methods for resizing using different user input to determine the particles to be preserved. All these methods use `T::copy` to resize the state vector. Therefore, if any of them need to be used, the method call `copy(N, index)` need to be able to handle the situation where its first argument has a value other than its original size. After the resizing, the weights are set to be equal.

Resize by selecting according to a user supplied index vector

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

The input iterator `index` shall point to a length N vector. Each element is the parent index of the corresponding new particle. The new system is formed by $\bar{X}^{(i)} = X^{(a_i)}$ where a_i is the i^{th} element of the index vector.

Resize by selecting according to a user supplied mask vector

```
1 template <typename InputIter>
2 void resize_by_mask(size_type N, InputIter mask);
```

The input iterator `mask` shall point to a length n vector, where n is the original sample size. For each element, when converted to `bool`, if it is `true`, then the corresponding particle is preserved. Otherwise it is discarded. If there are more than N particles to be preserved, then only the first N particles are copied into the new system. If there are less than N particles to be preserved, then the vector of these particles are copied repeatedly until there are enough particles to fill the new system.

Resize by resampling

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

The details of the resampling operation function object is discussed in chapter 6. For now, it is sufficient to say that, to use any of the builtin schemes, one can call the method like the following,

```
particle.resize_by_resample(N, ResampleMultinomial());
```

This method produce a new particle system by using the specified resampling algorithm. Note that, this is different from resampling a particle system in that, all operations are local. That is, if the particle is in a distributed system, only local weights are used, and re-normalized.

Resize by uniformly selecting from all particles

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

This is equivalent to,

```
particle.weight().set_equal();
particle.resize_by_resample(N, ResampleMultinomial());
```

Resize by selecting a range of particles

```
void resize_by_range(size_type N, size_type first, size_type last);
```

This is the most destructive resizing method. Let $n = \text{last} - \text{first}$. If $n < N$, the particles in the range $[\text{first}, \text{last})$ are copied repeatedly until there are enough particles to fill the new system. Otherwise only the first N particles are copied. If `last` is omitted, then it is set to `size()`. If `first` is also omitted, then it is set to zero. Use this method only when the statistical properties of the new system is irrelevant. If one want to avoid the copying altogether, it might be more efficient to simply create a new system with the desired size. This method is a no-op if N is the same as the original size.

2.3.4 *Single particle*

It is often easier to define a function $f(X^{(i)})$ than $f(X^{(1)}, \ldots, X^{(N)})$. However, `Particle<T>` only provides access to $\{X^{(i)}\}_{i=1}^{N}$ as a whole through `particle.value()`. To allow direct access to $X^{(i)}$, the library uses a class `SingeParticle<T>`. An object of this class is constructed from the index i of the particle, and a pointer to the particle system it belongs to,

```
SingleParticle<T> sp(i, &particle);
```

or more conveniently,

```
auto sp = particle.sp(i);
```

In its most basic form, it has the following methods,

```
sp.id();       // Get the value i that sp was constructed with
sp.particle(); // Get a reference to the Particle<T> object sp belongs to
sp.rng();      // => sp.particle().rng(sp.id());
```

If `T` is a derived class of `StateMatrix`, then it has two additional methods,

```
sp.dim();    // => sp.particle().value().dim();
sp.state(j); // => sp.particle().value().state(sp.id(), j);
```

It is clear now that the interface of `SingleParticle<T>` depends on the type `T`. Later in section 3.3 we will show how to insert additional methods into this class.

A `SingleParticle<T>` object is similar to an iterator. In fact, it supports almost all of the operations of a random access iterator with two exceptions. First dereferencing a `SingleParticle<T>` object returns itself. The support of `operator*` allows the range-based for loop to be applied on a `Particle<T>` object, for example,

```
for (auto sp : particle) {
    // operations on sp
}
```

is equivalent to

```
for (std::size_t i = 0; i != particle.size(); ++i) {
    auto sp = particle.sp(i);
    // operations on sp
}
```

The above range-based loop does make some sense. However trying to dereferencing a `SingleParticle<T>` object in other contexts does not make much sense. Recall that it is an *index*, not a *pointer*. The library does not require the user defined type `T` to provide access to individual values, and thus it cannot dereference a `SingleParticle<T>` object to obtain such a value. Similarly, the expression `sp[n]` returns `sp + n`, another `SingleParticle<T>` object. For the same reason, `operator->` is not supported at all.

2.3.5 *Sampler*

A sampler can be constructed in a few ways,

```
Sampler<T> sampler(N);
```

constructs a sampler that is never resampled, while

```
Sampler<T> sampler(N, Multinomial);
```

constructs a sampler that is resampled every iteration, using the Multinomial algorithm. Other resampling schemes are also implemented, see chapter 6. Last, one can also construct a sampler that is only resampled when ESS $< \alpha N$, where $\alpha \in [0, 1]$, by the following,

```
Sampler<T> sampler(N, Multinomial, alpha);
```

If $\alpha > 1$, then it has the same effect as the first constructor, since ESS $\leq N$. If $\alpha < 0$, then it has the same effect as the second constructor, since ESS > 0.

In summary, if one does not tell the constructor which resampling scheme to use, then it is assumed one does not want to do resampling. If one specify the resampling scheme without a threshold for ESS, then it is assumed it need to be done at every step.

The method `sampler.particle()` returns a reference to the particle system. It is safe to resize a `Particle<T>` object contained within a `Sampler<T>` object. The method `sampler.size()` is only a shortcut to `sampler.particle().size()`.

A sampler can be initialized by user defined function objects that are convertible to the following type,

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

For example,

```
auto init = [](Particle<T> &particle, void *param) {
    // Process initialization parameter param
    // Initialize the particle system particle
};
```

is a C++11 lambda expression that can be used for this purpose. One can add it to a sampler by calling `sampler.init(init)`. Upon calling `sampler.initialize(param)`, the user defined `init` will be called and the argument `param` will be passed to it.

Similarly, after initialization, at each iteration, the particle system can be manipulated by user defined function objects that are convertible to the following types,

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

Multiple moves can be added to a sampler. The call `sampler.move(move, append)` adds a `move_type` object to the sampler, where `append` is a boolean value. If it is `false`, it will clear any moves that were added before. If it is `true`, then `move` is appended to the end of an existing sequence of moves. Each move will be called one by one upon calling `sampler.iterate()`. A similar sequence of MCMC moves can also be added to a sampler. The call `sampler.iterate()` will call user defined moves first, then perform the possible resampling, and then the sequence of MCMC moves.

Note that the possible resampling will also be performed after the user defined initialization function is called by `sampler.initialize(param)`. And after that, the sequence of MCMC moves will be called. If it desired not to perform mutations during initialization, then following can be used,

```
sampler.init(init).initialize(param);
sampler.move(move, false).mcmc(mcmc, false).iterate(n);
```

The above code also demonstrates that most methods of `Sampler<T>` return a reference to the sampler itself and thus method calls can be chained. In addition, method `sampler.iterate(n)` accepts an optional argument that specifies the number of iterations. It is a shortcut for

```
for (std::size_t i = 0; i != n; ++i)
    sampler.iterate();
```

2.3.6 *Monitor*

Inferences using a SMC algorithm usually require the calculation of the quantity $\sum_{i=1}^{N} W^{(i)}\varphi(X^{(i)})$ at each iteration for some function φ. One can define function objects that are convertible to the following type,

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

For example,

```
void eval(std::size_t iter, std::size_t d, Particle<T> &particle, double *r)
{
    for (std::size_t i = 0; i != particle.size(); ++i, r += dim) {
        auto sp = particle.sp(i);
        r[0] = /* φ_1(X^(i)) */;
        // ...
        r[d - 1] = /* φ_d(X^(i)) */;
    }
}
```

The argument `d` is the dimension of the vector function φ. The output is an N by d matrix in row major layout, with each row corresponding to the value of $\varphi(X^{(i)})$. Then one can add this function to a sampler by calling,

```
sampler.monitor("name", d, eval);
```

where the first argument is the name for the monitor; the second is the dimension to be passed to the user defined function; and the third is the evaluation function. At each iteration, after all the initialization, possible resampling, moves and MCMC moves are done, the sampler will calculate $\sum_{i=1}^{N} W^{(i)}\varphi(X^{(i)})$. This method has two optional arguments. The first is a boolean value `record_only`. If it is `true`, it is assumed that no summation is needed. For example,

```
void eval(std::size_t iter, std::size_t d, Particle<T> &particle, double *r)
{
```

```
    r[0] = /* φ_1({X^(i)}_{i=1}^N) */;
    // ...
    r[d - 1] = /* φ_d({X^(i)}_{i=1}^N) */;
}
```

In this case, the monitor acts merely as a storage facility. The second optional argument is `stage` which specifies at which point the monitoring shall happen. It can be `MonitorMove`, which specifies that the monitoring happens right after the moves and before resampling. It can also be `MonitorResample`, which specifies that the monitoring happens right after the resampling and before the MCMC moves. Last, the default is `MonitorMCMC`, which specifies that the monitoring happens after everything.

The output of a sampler, together with the records of any monitors it has can be output in plain text forms through a C++ output stream. For example,

```
std::cout << sampler;
```

We will see how this works later with a concrete particle filter example. If the HDF5 library is available, it is also possible to write such output to HDF5 format, for example,

```
hdf5store(sampler, file_name, data_name);
```

Details can be found in section 8.3. More sophisticated methods for retrieving monitor records are defined by the `Monitor<T>` class. An object of this class can be retrieved by `sampler.monitor("name")`. See the reference manual for details.

2.4 A SIMPLE PARTICLE FILTER

2.4.1 *Model and algorithm*

This is an example used in Johansen (2009). Through this example, we will show how to re-implement a simple particle filter in vSMC. It shall walk one through the basic features of the library introduced above.

The state space model, known as the almost constant velocity model in the tracking literature, provides a simple scenario. The state vector X_t contains the position and velocity of an object moving in a plane. That is, $X_t = (X^t_{\text{pos}}, Y^t_{\text{pos}}, X^t_{\text{vel}}, Y^t_{\text{vel}})^T$. Imperfect observations $Y_t = (X^t_{\text{obs}}, Y^t_{\text{obs}})^T$ of the positions are possible at each time instance. The state and observation equations are linear with additive noises,

$$X_t = AX_{t-1} + V_t$$
$$Y_t = BX_t + \alpha W_t$$

where

$$A = \begin{pmatrix} 1 & 0 & \Delta & 0 \\ 0 & 1 & 0 & \Delta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \qquad \alpha = 0.1$$

Initialization

Set $t \leftarrow 0$.

Sample $X_{\text{pos}}^{(0,i)}, Y_{\text{pos}}^{(0,i)} \sim \mathcal{N}(0, 4)$ and $X_{\text{vel}}^{(0,i)}, Y_{\text{vel}}^{(0,i)} \sim \mathcal{N}(0, 1)$.

Weight $W_0^{(i)} \propto \exp \ell(X_0^{(i)} | Y_0)$ where ℓ is the log-likelihood function.

Iteration

Set $t \leftarrow t + 1$.

Sample

$$X_{\text{pos}}^{(t,i)} \sim \mathcal{N}(X_{\text{pos}}^{(t-1,i)} + \Delta X_{\text{vel}}^{(t-1,i)}, 0.02) \qquad X_{\text{vel}}^{(t,i)} \sim \mathcal{N}(X_{\text{vel}}^{(t-1,i)}, 0.001)$$
$$Y_{\text{pos}}^{(t,i)} \sim \mathcal{N}(Y_{\text{pos}}^{(t-1,i)} + \Delta Y_{\text{vel}}^{(t-1,i)}, 0.02) \qquad Y_{\text{vel}}^{(t,i)} \sim \mathcal{N}(Y_{\text{vel}}^{(t-1,i)}, 0.001)$$

Weight $W_t^{(i)} \propto W_{t-1}^{(i)} \exp \ell(X_t^{(i)} | Y_t)$.

Repeat the Iteration *step until all data are processed.*

Algorithm 2.1 Particle filter algorithm for the almost constant velocity model.

and we assume that the elements of the noise vector V_t are independent Gaussian with variance 0.02 and 0.001 for position and velocity, respectively. The observation noise, W_t comprises independent, identically distributed t-distributed random variables with degree of freedom $\nu = 10$. The prior at time 0 corresponds to an axis-aligned Gaussian with variance 4 for the position coordinates and 1 for the velocity coordinates. The particle filter algorithm is shown in algorithm 2.1.

2.4.2 *Implementations*

The complete program is shown in appendix B.1.1. In this section we show the outline of the implementation.

The main program

```
1 Sampler<PFState> sampler(N, Multinomial, 0.5);
2 sampler.init(PFInit()).move(PFMove(), false).monitor("pos", 2, PFEval());

3 sampler.initialize(const_cast<char *>("pf.data")).iterate(n - 1);

4 std::ofstream output("pf.out");
5 output << sampler;
6 output.close();
```

`Sampler<PFState>` object is constructed first. Then the initialization `PFInit`, move `PFMove` and a monitor `PFEval` that records X^t_{pos} and Y^t_{pos} are added to the sampler. The monitor is named `"pos"`. Then it is initialized with the name of the data file `"pf.data"`, and iterated $n-1$ times, where n is the number of data points. At last, the output is written into a text file `"pf.out"`. Below is a short R[5] script that can be used to process the output

```
pf <- read.table("pf.out", header = TRUE)
print(pf[1:5,])
```

The `print(pf[1:5,])` statement shows the first five lines of the output,

```
  Size Resampled Accept.0      ESS    pos.0   pos.1
1 1000         1        0   2.9204 -1.21951 3.16397
2 1000         1        0 313.6830 -1.15602 3.22770
3 1000         1        0  33.0421 -1.26451 3.04031
4 1000         1        0  80.1088 -1.45922 3.37625
5 1000         1        0 382.8820 -1.47299 3.49230
```

The column `Size` shows the sample size at each iteration. The column `Resampled` shows nonzero values if resampling were performed and zero otherwise. For each moves and MCMC steps, an acceptance count will be recorded. In this particular example, it is irrelevant. Next the column `ESS` shows the value of ESS *before* resampling. The last two columns show the importance sampling estimates of the positions recorded by the monitor named `"pos"`. A graphical representation of the output is shown in figure 2.1.

Before diving into the details of the implementation of `PFState`, etc., we will first define a few constants and types. The state space is of dimension 4. And it is natural to use a `StateMatrix` as the base class of `PFState`,

```
using PFStateBase = StateMatrix<RowMajor, 4, double>;
```

We define the following constants as the indices of each state component.

```
static constexpr std::size_t PosX = 0;
static constexpr std::size_t PosY = 1;
static constexpr std::size_t VelX = 2;
static constexpr std::size_t VelY = 3;
```

[5] http://r-project.org

State: `PFState`

As noted earlier, `StateMatrix` will be used as the base class of `PFState`. Since the data will be shared by all particles, we also store the data within this class. And methods will be provided to read the data from an external file, and compute the log-likelihood $\ell(X^{(i)})$, which accesses the data. Below the declaration of the class `PFState` is shown,

```
class PFState : public PFStateBase
{
    public:
    using PFStateBase::PFStateBase;

    // Return ℓ(X_t^(i)|Y_t)
    double log_likelihood(std::size_t t, size_type i) const;

    // Read data from an external file
    void read_data(const char *param);

    private:
    Vector<double> obs_x_;
    Vector<double> obs_y_;
};
```

Initialization: `PFInit`

The initialization step is implemented as below,

```
class PFInit
{
    public:
    std::size_t operator()(Particle<PFState> &particle, void *param)
    {
        eval_param(particle, param);
        eval_pre(particle);
        std::size_t acc = 0;
        for (auto sp : particle)
            acc += eval_sp(sp);
        eval_post(particle);
```

```
12          return acc;
13      }

14      void eval_param(Particle<PFState> &particle, void *param)
15      {
16          particle.value().read_data(static_cast<const char *>(param));
17      }

18      void eval_pre(Particle<PFState> &particle)
19      {
20          weight_.resize(particle.size());
21      }

22      std::size_t eval_sp(SingleParticle<PFState> sp)
23      {
24          NormalDistribution<double> norm_pos(0, 2);
25          NormalDistribution<double> norm_vel(0, 1);
26          sp.state(PosX) = norm_pos(sp.rng());
27          sp.state(PosY) = norm_pos(sp.rng());
28          sp.state(VelX) = norm_vel(sp.rng());
29          sp.state(VelY) = norm_vel(sp.rng());
30          weight_[sp.id()] = sp.particle().value().log_likelihood(0, sp.id());

31          return 0;
32      }

33      void eval_post(Particle<PFState> &particle)
34      {
35          particle.weight().set_log(weight_.data());
36      }

37      private:
38      Vector<double> weight_;
39  };
```

An object of this class is convertible to `Sampler<PFState>::init_type`. In the main method, `operator()`, `eval_param` is called first to initialize the data. Then `eval_pre` is called to allocated any resource this class need before calling any `eval_sp`. In this case, it allocate the vector `weight_` for storing weights computed

later. Next, the main loop initializes each state component with the respective Gaussian distribution, computes the log-likelihood and store them in the vector allocated in the last step. This is done by calling the `eval_sp` method. After all particles have been initialized, we set the weights of the system in `eval_post`. Later in section 2.5, it will become clear why we structured the implementation this way.

Move: `PFMove`

The move step is similar to the initialization. We show the declaration here,

```
1  class PFMove
2  {
3      public:
4      std::size_t operator()(std::size_t t, Particle<PFState> &particle);
5      void eval_pre(std::size_t t, Particle<PFState> &particle);
6      std::size_t eval_sp(std::size_t t, SingleParticle<PFState> sp);
7      void eval_post(std::size_t t, Particle<PFState> &particle);

8      private:
9      Vector<double> w_;
10 };
```

Monitor: `PFEval`

Last we define `PFEval`, which simply copies the values of the positions.

```
1  class PFEval
2  {
3      public:
4      void operator()(std::size_t t, std::size_t dim,
5          Particle<PFState> &particle, double *r)
6      {
7          eval_pre(t, particle);
8          for (std::size_t i = 0; i != particle.size(); ++i, r += dim)
9              eval_sp(t, dim, particle.sp(i), r);
10         eval_post(t, particle);
11     }

12     void eval_pre(std::size_t t, Particle<PFState> &particle) {}
```

```
13      void eval_sp(std::size_t t, std::size_t dim,
14          SingleParticle<PFState> sp, double *r)
15      {
16          r[0] = sp.state(PosX);
17          r[1] = sp.state(PosY);
18      }

19      void eval_post(std::size_t t, Particle<PFState> &particle) {}
20  };
```

2.5 SYMMETRIC MULTIPROCESSING

The above example is implemented in a sequential fashion. However, the loops inside `PFInit`, `PFMove` and `PFEval` clearly can be parallelized. The library provides basic support of multicore parallelization through its SMP module. Two widely used backends, OpenMP and TBB are available. Here we demonstrate how to use the TBB backend. First we will declare the implementation classes as derived classes,

```
1 class PFInit : public InitializationTBB<PFState>;
2 class PFMove : public MoveTBB<PFState>;
3 class PFEval : public MonitorEvalTBB<PFState>;
```

And remove `operator()` from their implementations. After these changes, the implementation will be parallelized using TBB. The complete program can be found in section The complete program is shown in appendix B.1.2.

It works as if `InitializationTBB<PFState>` has an implementation of `operator()` as we did before, except it is parallelized. Now it is clear that, method such as `eval_pre` and `eval_post` are called before and after the main loop. Method `eval_sp` is called within the loop and it need to be thread-safe if called with different arguments. This is the main reason we constructed the `NormalDistribution` objects within `eval_sp` instead of as member data, even though they are constructed in exactly the same way for each particle. This is because `NormalDistribution::operator()` is a mutable method and thus not thread-safe. If any of these member functions does not do anything, then it does not have to be defined in the derived class.

Apart from the three base classes we have shown here, there are also `InitializationOMP`, etc., for using the OpenMP backend. And `InitializationSEQ`, etc., for implementation without parallelization. The later works in exactly the same way as our implementation in the last section. It is often easier to debug a single-threaded program than a parallelized one. And thus one may develop the algorithm with the sequential backend and obtain optimal performance latter by only changing the name of a few base class names. This can usually be done automatically through a build system.

2.5.1 *Performance consideration*

The base classes dispatch calls to `eval_pre`, `eval_sp`, etc., through the virtual function mechanism. The performance impact is minimal for `eval_pre` and `eval_post`, since they are called only once in each iteration and we expect the computational cost will be dominated by `eval_sp` in most cases. However, the dynamic dispatch can cause considerable performance degenerating if the cost of a single call to `eval_sp` is small while the number of particles is large. Modern optimizing compilers can usually devirtualize the method calls in trivial situations. However, it is not always possible. In this situation, the library will need a little help from the user to make compile-time dispatch. For each implementation class, we will declare it in the following way,

```
1 class PFInit : public InitializationTBB<PFState, PFInit>;
2 class PFMove : public MoveTBB<PFState, PFMove>;
3 class PFEval : public MonitorEvalTBB<PFState, PFEval>;
```

The second template argument of the base class need to be exactly the same as the derived class. For interested users, this is called Curiously Recurring Template Pattern[6] (CRTP). This usage of the library's base classes also provides other flexibility. The methods `eval_pre` etc., can be either `const` or mutable. They can also be `static`.

[6] https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

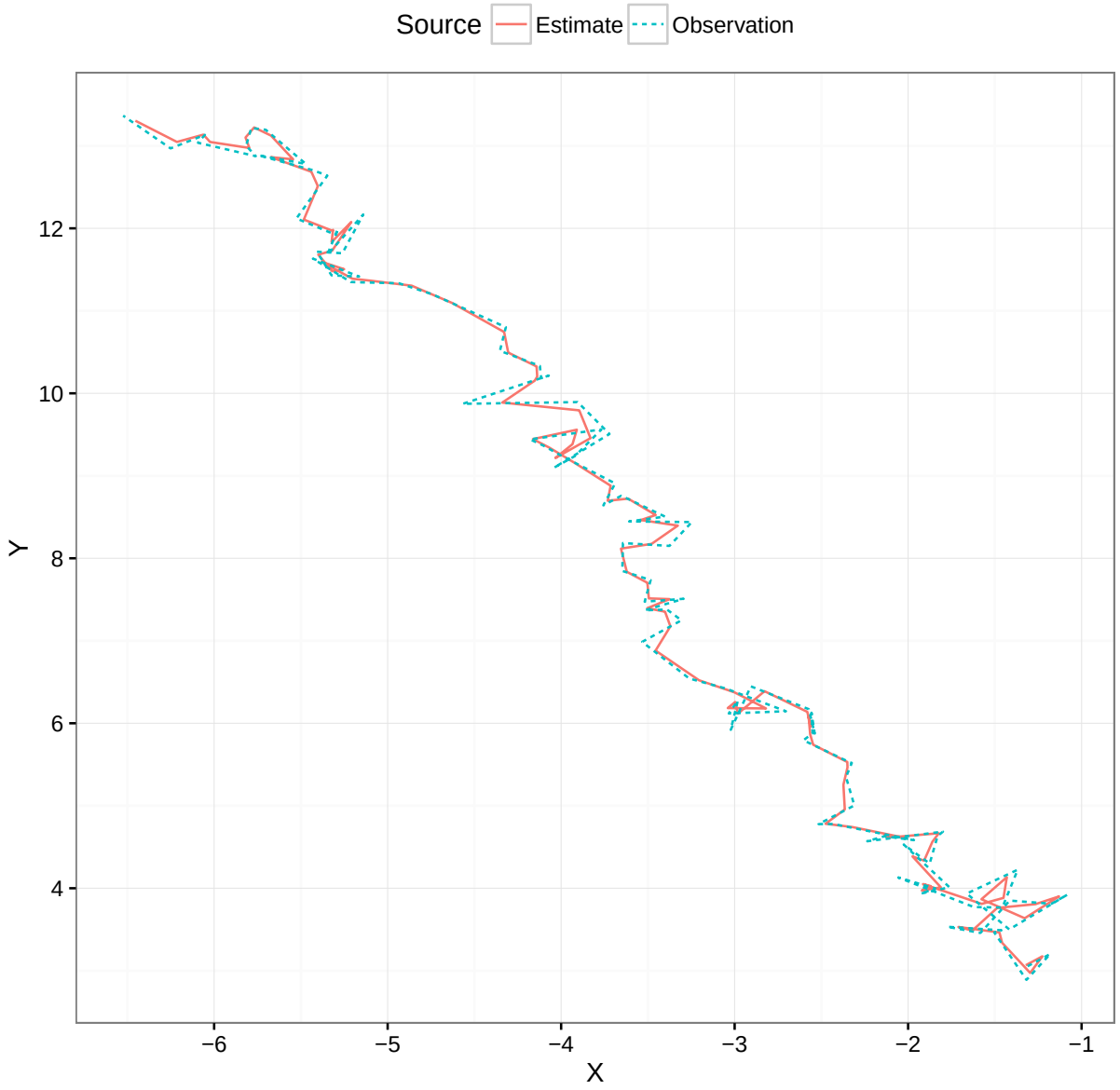

Figure 2.1 A simple particle filter

3.1 CLONING OBJECTS

The `Sampler<T>` and `Particle<T>` objects have copy constructors, assignment operators, move constructors, and move assignment operators that behave exactly the way as C++ programmers would expect. However, these behaviors are not always desired. For example, in Beskos et al. (2014) a stable particle filter in high-dimensions was developed. Without going into the details, the algorithm consists of a particle system where each particle is itself a particle filter. And thus when resampling the global system, the `Sampler<T>` object will be copied, together with all of its sub-objects. This include the RNG system within the `Particle<T>` object. Even if the user does not use this RNG system for random number generating within user defined operations, one of these RNG will be used for resampling by the `Particle<T>` object. Direct copying the `Sampler<T>` object will lead to multiple local filters start generating exactly the same random numbers in the next iteration. This is an undesired side effect. In this situation, one can clone the sampler with the following method,

```
auto new_sampler = sampler.clone(new_rng);
```

where `new_rng` is a boolean value. If it is `true`, then an exact copy of `sampler` will be returned, except it will have the RNG system re-seeded. If it is `false`, then the above assignemnt behaves exactly the same as

```
auto new_sampler = sampler;
```

Alternatively, the contents of an existing `Sampler<T>` object can be replaced from another one by the following method,

```
sampler.clone(other_sampler, retain_rng);
```

where `retain_rng` is a boolean value. If it is `true`, then the RNG system of `other_sampler` is not copied and the original is retained. If it is `false`, then the above call behaves exactly the same as

```
sampler = other_sampler;
```

The above method also supports move semantics. Similar `clone` methods exist for the `Particle<T>` class.

3.2 CUSTOMIZING MEMBER TYPES

The `Particle<T>` class has a few member types that can be replaced by the user. If the class `T` has the corresponding types, then the member type of `Particle<T>` will be replaced. For example, given the following declarations inside class `T`,

```
class T
{
    public:
    using size_type = int;
    using weight_type = /* User defined type */;
    using rng_set_type = RNGSetTBB<AES256_4x32>;
};
```

The corresponding `Particle<T>::size_type`, etc., will have their defaults replaced with the above types.

3.2.1 *Replacing `weight_type`*

The class for managing the weights needs to provide the following methods,

```
w.ess();           // Get ESS
w.set_equal();     // Set W^(i) = 1/N
w.resample_size(); // Get the sample size N.
w.resample_data(); // Get a pointer to normalized weights
```

For the library's default class `Weight`, the last two calls are the same as `w.size()` and `w.data()`. However, this does not need to be so. For example, below is the outline of an implementation of `weight_type` for distributed systems, assuming there are R computing nodes and the node with rank r has been allocated N_r particles. Let $\{W_r^{(i)}\}_{i=1}^{N_r}$ denote the weights at the node with rank r.

```
class WeightMPI
{
    public:
    double ess()
    {
        double local = /* sum_{i=1}^{N_r} (W_r^(i))^2 */;
        double global = /* Gather local from each all nodes */;
        // Broadcast the value of global

        return 1 / global;
    }

    std::size_t size() { return /* N_r */; }

    std::size_t resample_size() { return /* N = sum_{r=1}^{R} N_r */; }
```

```
13      const double *data()
14      {
15          return /* pointer to {W_r^(i)}_{i=1}^{N_r} */;
16      }

17      const double *resample_data()
18      {
19          if (rank == 0) {
20              // Gather all normalized weights into a member data on this node
21              // Say resample_weight_
22              return resample_weight_.data();
23          } else {
24              return nullptr;
25          }
26      }

27      void set_equal()
28      {
29          // Set all weights to 1/∑_{r=1}^{R} N_r
30          // Synchronization
31      }

32      void set(const double *v)
33      {
34          // Set W_r^(i) = v_i for i = 1, ..., N_r
35          // Compute S_r = ∑_{i=1}^{N_r} W_r^(i)
36          // Gathering S_r, compute S = ∑ S_r
37          // Broadcast S
38          // Set W_r^(i) = W_r^(i)/S for i = 1, ..., N_r
39      }
40  };
```

When `Particle<T>` performs resampling, it checks if the pointer returned by `w.resample_data()` is a null pointer. It will only generate the vector $\{a_i\}_{i=1}^{N}$ (see section 2.3.1) when it is not a null pointer, pass a pointer to this vector is passed to `T::copy`. Otherwise, a null pointer is passed to `T::copy`. Of course, the class `T` also needs to provide a suitable method `copy` that can handle the distributed system. By defining suitable `WeightMPI` and `T::copy`, the library can be extended to handle distributed systems.

3.3 EXTENDING `SingleParticle<T>`

The `SingleParticle<T>` can also be extended by the user. We have already seen in section 2.3.1 that if class `T` is a derived class of `StateMatrix`, `SingleParticle<T>` can have additional methods to access the state. This class can be extended by defining a member class template inside class `T`. For example, for the simple particle filter in section 2.4, we can redefine the `PFState` as the following,

```
1  using PFStateBase = StateMatrix<RowMajor, 4, double>;

2  template <typename T>
3  using PFStateSPBase = PFStateBase::single_particle_type<T>;

4  class PFState : public PFStateBase
5  {
6      public:
7      using PFStateBase::StateMatrix;

8      template <typename S>
9      class single_particle_type : public PFStateSPBase<S>
10     {
11         public:
12         using PFStateSPBase<S>::single_particle_type;

13         double &pos_x() { return this->state(0); }
14         double &pos_y() { return this->state(1); }
15         double &vel_x() { return this->state(2); }
16         double &vel_y() { return this->state(3); }

17         // Return ℓ(X_t^(i)|Y_t)
18         double log_likelihood(std::size_t t);
19     };

20     void read_data(const char *param);

21     private:
22     Vector<double> obs_x_;
23     Vector<double> obs_y_;
24 };
```

And later, we can use these methods when implement `PFInit` etc.,

```
class PFInit : public InitializeTBB<PFState, PFInit>
{
    public:
    void eval_param(Particle<PFState> &particle, void *param);

    void eval_pre(Particle<PFState> &particle);

    std::size_t eval_sp(SingleParticle<PFState> sp)
    {
        NormalDistribution<double> norm_pos(0, 2);
        NormalDistribution<double> norm_vel(0, 1);
        sp.pos_x() = norm_pos(sp.rng());
        sp.pos_y() = norm_pos(sp.rng());
        sp.vel_x() = norm_vel(sp.rng());
        sp.vel_y() = norm_vel(sp.rng());
        w_[sp.id()] = sp.log_likelihood(0);

        return 0;
    }

    void eval_post(Particle<PFState> &particle);

    private:
    Vector<double> w_;
};
```

It shall be noted that, it is important to keep `single_particle_type` small and copying the object efficient. The library will frequently pass argument of `SingleParticle<T>` type by value.

3.3.1 *Compared to custom state type*

One can also write a custom state type. For example,

```
class PFStateSP
{
    public:
    double &pos_x() { return pos_x_; }
```

```
    double &pos_y() { return pos_y_; }
    double &vel_x() { return vel_x_; }
    double &vel_y() { return vel_y_; }

    double log_likelihood(double obs_x, double obs_y) const;

    private:
    double pos_x_;
    double pos_y_;
    double vel_x_;
    double vel_y_;
};
```

And the `PFState` class will be defined as,

```
using PFStateBase = StateMatrix<RowMajor, 1, PFStateSP>;

class PFState : public PFStateBase
{
    public:
    using PFStateBase::StateMatrix;

    double log_likelihood(std::size_t t, std::size_t i) const
    {
        return this->state(i, 0).log_likelihood(obs_x_[t], obs_y_[t]);
    }

    void read_data(const char *param);

    private:
    Vector<double> obs_x_;
    Vector<double> obs_y_;
};
```

The implementation of `PFInit`, etc., will be similar. Compared to extending the `SingleParticle<T>` type, this method is perhaps more intuitive. Functionality-wise, they are almost identical. However, there are a few advantages of extending `SingleParticle<T>`. First, it allows more compact data storage. Consider a situation where the state space is best represented by a real and an integer. The most intuitive way might be the following,

```
class S
{
    public:
    double &x() { return x_; }
    int &u() { return u_; }

    private:
    double x_;
    int u_;
};

class T : StateMatrix<RowMajor, 1, S>;
```

However, the type `S` will need to satisfy the alignment requirement of `double`, which is 8-bytes on most platforms. However, its size might not be a multiple of 8-bytes. Therefore the type will be padded and the storage of a vector of such type will not be as compact as possible. This can affect performance in some situations. An alternative approach would be the following,

```
class T
{
    public:
    template <typename S>
    class single_particle_type : SingleParticleBase<S>
    {
        public:
        using SingleParticleBase<S>::SingleParticleBase;

        double &x() { return this->particle().x_[this->id()]; }
        double &u() { return this->particle().u_[this->id()]; }
    };

    private:
    Vector<double> x_;
    Vector<int> u_;
};
```

By extending `SingleParticle<T>`, it provides the same easy access to each particle. However, now the state values are stored as two compact vectors.

A second advantage is that it allows easier access to the raw data. Consider the implementation `PFEval` in section 2.4.2. It is rather redundant to copy each value of the two positions, just so later we can compute

weighted sums from them. Recall that in section 2.3.6 we showed that a monitor that compute the final results directly can also be added to a sampler. Therefore, we might implement `PFEval` as the following,

```
class PFEval
{
    public:
    void operator()(std::size_t t, std::size_t dim,
        Particle<PFState> &particle, double *r)
    {
        cblas_dgemv(CblasRowMajor, CblasTrans, particle.size(), dim, 1,
            particle.value().data(), particle.value().dim(),
            particle.weight().data(), 1, 0, r, 1);
    }
};
```

And it can be added to a sampler as,

```
sampler.monitor("pos", 2, PFEval(), true);
```

This is only possible if the `PFState` was implemented with contiguous storage of the states. For this particular case, the performance benefit is small. But the possibility of accessing compact vector as raw data allows easier interfacing with external numerical libraries. If we have implemented `PFState` with the alternative approach shown earlier, the above direct invoking of `cblas_dgemv` will not be possible.

The library has a few configuration macros. All these macros can be overwritten by the user by defining them with proper values before including any of the library's headers. All configurations macros are listed in table 4.1. There are some additional macros for RNG related functionalities. They will be discussed in chapter 7.

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 shall be used, if available. For example, if `VSMC_HAS_MKL` is defined to a non-zero value, but the it is desirable not to use MKL's vector math functions, then one can define `VSMC_USE_MKL_VML` to zero to prevent the library to use this individual component. All other macros define either types or constants that are used by the library.

Another important difference between macros with prefixes `VSMC_HAS` and `VSMC_USE` is that, the former will affect the interface while the later only affect internal implementations.

Macro	Default	Description
VSMC_HAS_INT128	Platform dependent	Support for 128-bits integers
VSMC_HAS_SSE2	Platform dependent	Support for SSE2 intrinsic functions
VSMC_HAS_AVX2	Platform dependent	Support for AVX2 intrinsic functions
VSMC_HAS_AES_NI	Platform dependent	Support for AES-NI intrinsic functions
VSMC_HAS_RDRAND	Platform dependent	Support for RDRAND intrinsic functions
VSMC_HAS_X86	Platform dependent	Support for x86 platform
VSMC_HAS_X86_64	Platform dependent	Support for x86-64 platform
VSMC_HAS_POSIX	Platform dependent	Support for POSIX platform
VSMC_HAS_OMP	Platform dependent	Support for OpenMP 3.0 or higher
VSMC_HAS_OPENCL	0	Support for OpenCL 1.2 or higher
VSMC_HAS_TBB	0	Support for TBB 4.0 or higher
VSMC_HAS_TBB_MALLOC	VSMC_HAS_TBB	Support for TBB scalable memory allocation
VSMC_HAS_HDF5	0	Support for HDF5 1.8.6 or higher
VSMC_HAS_MKL	0	Support for MKL 11.0 or higher
VSMC_USE_OMP	VSMC_HAS_OMP	Use OpenMP parallelization outside the SMP module
VSMC_USE_TBB	VSMC_HAS_TBB	Use TBB parallelization outside the SMP module
VSMC_USE_TBB_TLS	VSMC_HAS_TBB	Use TBB thread-local storage classes (TLS)
VSMC_USE_MKL_CBLAS	VSMC_HAS_MKL	Use mkl_cblas.h instead of cblas.h
VSMC_USE_MKL_LAPACKE	VSMC_HAS_MKL	Use mkl_lapacke.h instead of lapacke.h
VSMC_USE_MKL_VML	VSMC_HAS_MKL	Use MKL vector mathematical functions (VML)
VSMC_USE_MKL_VSL	VSMC_HAS_MKL	Use MKL statistical functions (VSL)
VSMC_USE_ACCELERATE	Platform dependent	Use Mac OS X Accelerate framework for BLAS. Ignored if VSMC_USE_MKL_BLAS is defined to a non-zero value.
VSMC_INT64	Platform dependent	The 64-bit integer type used by x86 intrinsics
VSMC_INT128	Platform dependent	The 128-bit integer type
VSMC_CBLAS_INT_TYPE	int	The default integer type of BLAS routines
VSMC_ALIGNMENT	32	Default alignment for scalar types
VSMC_ALIGNMENT_CACHE	64	Default cache line alignment for scalar types
VSMC_ALIGNMENT_MIN	16	Minimum alignment for all types
VSMC_ALIGNED_MEMORY_TYPE	Platform dependent	The type of AlignedMemory
VSMC_CONSTRUCT_SCALAR	0	Should Allocator::construct zero out scalar types

Table 4.1 Configuration macros

5.1 CONSTANTS

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

```
auto pi_f = const_pi<float>();
auto pi_d = const_pi<double>();
auto 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 5.1.

5.2 VECTORIZED OPERATIONS

The library provides a set of functions for vectorized mathematical operations. For example,

```
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());
```

performs addition for vectors. It is equivalent to

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

The functions defined are listed in table 5.2 to 5.7. For each function, the first parameter is always the length of the vector, and the last is a pointer to the output vector (except `sincos` which has 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 5.2, 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.

5.3 PACK AND UNPACK VECTORS

The vectorized operations in the last section only operates on contiguous vectors. The library provides three functions to pack general vector into such storage,

```
// dst[i] = src[i * stride], i = 1 to n
template <typename RandomIter, typename IntType, typename OutputIter>
inline void pack_s(
    std::size_t n, RandomIter src, IntType stride, OutputIter dst);

// dst[i] = src[index[i]], i = 1 to n
template <typename RandomIter, typename InputIter, typename OutputIter>
inline void pack_i(
    std::size_t n, RandomIter src, InputIter index, OutputIter dst);

// Pack all src[i] with static_cast<boo>(mask[i]) is true, i = 1 to n
template <typename InputIterSrc, typename InputIterMask, typename OutputIter>
inline void pack_m(
    std::size_t n, InputIterSrc src, InputIterMask mask, OutputIter dst);
```

There are also three corresponding unpack functions,

```
// dst[i * stride] = src[i], i = 1 to n
template <typename InputIter, typename IntType, typename RandomIter>
inline void unpack_s(
    std::size_t n, InputIter src, IntType stride, RandomIter dst);

// dst[index[i]] = src[i], i = 1 to n
template <typename InputIterSrc, typename InputIterIndex, typename RandomIter>
inline void unpack_i(
    std::size_t n, InputIterSrc src, InputIterIndex index, RandomIter dst);

// dst[j] = src[i], where mask[j] is the i-th element of mask such that
// static_cast<bool>(mask[j]) is true, i = 1 to n
template <typename InputIterSrc, typename InputIterMask, typename OutputIter>
inline void unpack_m(
    std::size_t n, InputIterSrc src, InputIterMask mask, OutputIter dst);
```

These functions guarantee that the three assertions in the following program will never fail,

```
pack_s(n, src, stride, tmp);
unpack_s(n, tmp, stride, dst);
for (std::size_t i = 0; i != n; ++i)
    assert(src[i * stride] == dst[i * stride]);
```

```
pack_i(n, src, index, tmp);
unpack_i(n, tmp, index, dst);
for (std::size_t i = 0; i != n; ++i)
    assert(src[index[i]] == dst[index[i]]);

pack_m(n, src, mask, tmp);
unpack_m(n, tmp, mask, src);
for (std::size_t i = 0; i != n; ++i)
    if (mask[i])
        assert(src[i] == dst[i]);
```

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 5.1 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)`	$y_i = \lvert a_i \rvert$
`fma(n, a, b, c, y)`	$y_i = a_i b_i + c_i$

Table 5.2 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 5.3 Power and root functions

Function	Operation
`exp(n, a, y)`	$y_i = \mathrm{e}_i^a$
`exp2(n, a, y)`	$y_i = 2_i^a$
`exp10(n, a, y)`	$y_i = 10_i^a$
`expm1(n, a, y)`	$y_i = \mathrm{e}_i^a - 1$
`log(n, a, y)`	$y_i = \ln a_)$
`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 5.4 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 5.5 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 5.6 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 5.7 Special functions

6.1 BUILTIN ALGORITHMS

The library supports resampling in a more general way than the algorithm described in chapter 1. 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 2.4.2. 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>
```

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

```
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 $i = 0, \ldots, \min\{N, M\}$. 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 `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 $U_1 \le U_2 \le \ldots \le U_N$.

An obvious method is to generate the random numbers first and then sort them. However, no sorting algorithm has cost $O(M)$, while it is possible to generate such an ordered sequence with cost $O(M)$ 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 $O(N + M)$, unless the random sequence has a greater cost.

6.3 ALGORITHMS WITH INCREASING DIMENSIONS

Recall section 1.2, 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 $O(1)$. 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 $O(1)$, 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]];
        }
```

```
31         return rs;
32     }

33     private:
34     Vector<double> varphi_; // the values of φ(X_t^(i))
35     ResampleIndex index_;
36 };

37 Sampler<State> sampler(N, Multinomial); // Always resampling
38 sampler.particle.value().resize_dim(n + 1);
39 // configure the sampler
40 sampler.initialize(param);
41 sampler.iterate(n);
42 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 $i = 1, \ldots, N$. 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.

The library has a comprehensive RNG system to facilitate implementation of Monte Carlo algorithms. A set of counter-based RNGs developed in Salmon et al. (2011) are re-implemented in the library with some extensions. In addition wrappers for high performance RNGs in MKL are also provided, such that they can be used as C++11 RNG engines. Some continuous distributions, including all those in the standard library, are implemented. Some of them are considerably faster than their standard library counter-parts.

Section 7.1 introduces the library's interface for vectorized random number generating. Section 7.2 discusses the simple benchmark procedures later used in this chapter. Section 7.3 details the counter-based RNGs implemented in the library. Section 7.4 briefly discusses the non-deterministic RNGs using RDRAND instructions and section 7.5 shows how to use RNGs in the MKL library as C++11 engines. Section 7.7 discusses the distributions implemented by this library. Section 7.8 discusses how to seed counter-based RNGs.

7.1 VECTORIZED RANDOM NUMBER GENERATING

Before we discuss other features, we first introduce a generic function `rng_rand`, which provides vectorized random number generating.

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

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

The first is equivalent to generating random integers through a loop. For example,

```
rng_rand(rng, n, r.data());
// is equivalent to
for (std::size_t i = 0; i != n; ++i)
    r[i] = rng();
```

The results of the two will always be exactly the same unless RNG is non-deterministic. The second version of `rng_rand` is also similar to the loop,

```
rng_rand(rng, distribution, n, r.data())
// is similar to
```

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

However, the results will not always be exactly the same, though the vectorized version will generate random numbers with the same distribution as expected.

The advantage of using `rng_rand` is that, if `RNGType` or `DistributionType` are classes defined by this library, then vectorized implementations might be used instead of using the loop. For some distributions, the performance gain can be significant. Consider the following dummy example,

```
std::size_t n = 10000;
RNG rng_vsmc;
MKL_SFMT19937 rng_mkl;
std::normal_distribution<double> rnorm_std;
NormalDistribution<double> rnorm_vsmc;
Vector<double> r(n);

// Method 1
for (std::size_t i = 0; i != n; ++i)
    r[i] = rnorm_std(rng_vsmc);

// Method 2
for (std::size_t i = 0; i != n; ++i)
    r[i] = rnorm_vsmc(rng_vsmc);

// Method 3
rng_rand(rng_vsmc, rnorm_vsmc, n, r.data());

// Method 4
rng_rand(rng_mkl, rnorm_vsmc, n, r.data());
```

On the author's computer, on average method 1 costs 102 cycles to generate one standard Normal random number. Method 2 costs 61 cycles while method 3 costs only 14 cycles. The last, when the RNG engine is an MKL RNG wrapper, the MKL library's routines are used and it costs only 10 cycles.

7.2 PERFORMANCE MEASUREMENT

Some of the following sections provide performance data. All performance data 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 (Apple version 7.3.0), the GNU GCC (version 5.3.0), and the Intel C++ compiler (2016 update 2). The version information is shown below,When multiple compilers are tested, they are labeled "LLVM", "GNU", and "Intel", respectively in the tables. If only results from one compiler is shown, then unless stated otherwise, it is the LLVM clang compiler. The operating system is Mac OS X 10.11.4.

For the performance of RNGs, we measure two situations. The first is using the random integers one by one,

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

See section 7.7.1 for details of `UniformBitsDistributions`. In short, it generates 64 random bits each time `rbits(rng)` is executed. The second is vectorized performance,

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

See section 7.1 for details of `rng_rand`. In both cases, we repeat the experiments 100 times, each time with the number of elements n chosen randomly between 5,000 and 10,000. The performance is measured in cpB. The two performance data are labeled "Loop" and "`rng_rand`", respectively.

For the performance of distributions, we measure four situations. Take the Normal distribution as an example, first, if the distribution is available in the standard library or the Boost[1] library, we measure the following case,

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

Second, we measure the performance of the vSMC 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,

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

	Loop			rng_rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
mt19937	3.17	3.41	4.16	3.36	3.45	4.87
mt19937_64	1.68	1.50	2.09	1.82	1.43	2.10
minstd_rand0	5.14	5.79	8.13	5.15	5.71	7.52
minstd_rand	4.06	5.90	7.02	4.08	5.91	6.34
ranlux24_base	7.13	7.73	8.37	7.65	7.79	8.04
ranlux48_base	4.39	4.34	4.62	4.71	4.29	4.87
ranlux24	74.0	73.3	90.0	73.7	73.8	78.8
ranlux48	172	155	210	174	154	190
knuth_b	17.4	25.4	16.6	14.3	25.3	17.6

Table 7.1 Performance of standard library RNG

```
1 rng_rand(rng, rnorm_vsmc, n, r.data());
```

For all the three above, the RNG is `ARSx8` (section 7.3.1). The last measurement is the situation when RNG is `MKL_SFMT19937` (section 7.5).

```
1 MKL_SFMT19937 rng_mkl;
2 rng_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 performance is measured in cpE. The four performance data are labeled "STD/Boost", "vSMC", "`rng_rand`" and "MKL", respectively.

Note that these performance data, especially those for RNGs, are for reference only. The actual performance can differ considerably given different compiler, operating system and hardware.

7.3 COUNTER-BASED RNG

The standard library provides a set of RNG engines (performance data in table 7.1). Unfortunately, none of them are suitable for parallel computing without considerable efforts. The development by Salmon et al. (2011) made high performance parallel RNG much more accessible.

The RNGs introduced in the paper use deterministic functions f_k, such that, for a sequence $\{c_i = i\}_{i\geq 0}$, the sequence $\{y_i = f_k(c_i)\}_{i\geq 0}$ appears as random. In addition, for $k_1 \neq k_2$, 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,

Class	Counter bits	Key bits
`AES128x1, ARS128x2, AES128x4, AES128x8`	128	128
`AES192x1, ARS192x2, AES192x4, AES192x8`	128	192
`AES256x1, ARS256x2, AES256x4, AES256x8`	128	256
`ARSx1, ARS256x2, 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 7.2 Counter-based RNG; Each RNG engine may have suffix _64

the RNG has a period 2^b. And there can be at most 2^d independent streams. Table 7.2 lists all counter-based RNGs implemented in this library, along with the bits of the counter and the key. They all conform to the C++11 uniform RNG engine concept and output 32-bits integers. For 64-bits integer output, a suffix `_64` may be appended to the corresponding RNG engine names. For example, `Threefry4x64` and `Threefry4x64_64` generate the same 256-bits random integers internally. The only difference is that `operator()` of the former returns 32-bits integers while the later returns 64-bits integers.

All RNGs in table 7.2 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 implement the algorithm. `Generator` has at least two member types, `ctr_type` and `key_type`, the types of the counter and key. In addition, four methods are required,

```
// The size of output in bytes
static constexpr std::size_t size();

// Reset the key of the generator
```

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 7.3 Configuration macros for counter-based RNG

```
4 void reset(const key_type &key);

5 // Increment counter and generate a new random buffer
6 template <typename ResultType>
7 void operator()(ctr_type &ctr,
8     std::array<ResultType, size() / sizeof(ResultType)> &buffer);

9 // Increment counter and generate n new random buffers
10 template <typename ResultType>
11 void operator()(ctr_type &ctr, std::size_t n,
12     std::array<ResultType, size() / sizeof(ResultType)> *buffer);
```

The operators are not restricted to increment the counter only once for each random buffer. The only restriction is that `size()` is divisible by `sizeof(ResultType)`. This rule is enforced at compile-time. In the remaining of this section, we introduce a few generators implemented by the library. A few configuration macros of these RNGs are listed in table 7.3.

7.3.1 *AES-NI instructions based RNG*

The AES-NI instructions based RNGs in Salmon et al. (2011) are implemented in a more general form,

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

3 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 sequence of round keys; `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 this 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 =
    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 expression,

	Loop			rng_rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
AES128x1	1.08	6.05	5.65	0.78	4.55	4.97
AES128x2	1.22	3.12	3.63	0.70	2.60	3.10
AES128x4	0.85	1.71	2.14	0.66	1.31	1.71
AES128x8	1.51	1.26	2.00	1.08	0.84	1.13
AES128x1_64	0.87	5.66	5.18	0.78	4.51	4.81
AES128x2_64	0.88	2.82	3.74	0.77	2.60	3.33
AES128x4_64	0.76	1.70	2.03	0.66	1.42	1.75
AES128x8_64	1.17	1.18	1.69	0.85	0.87	1.12

Table 7.4 Performance of AES128Engine

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

shall define a type that has a static constant expression member data `value` that is the $\texttt{I}^{\text{th}}$ 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 as supported. Aliases for block sizes 1, 2, 4 and 8 are defined for all four algorithms, as well as both 32- and 64-bits versions. These aliases are listed in table 7.2.

The performance of these engines depends on a few factors, such as CPU types, compilers, operating systems, etc. For example, the theoretical peak performance for `ARSx8` is 0.32 cpB on recent CPUs. In realistic situations, depending on the compiler, on the same computer values ranging from 0.35 to 1.2 cpB were observed by the author. In any case, such 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 `rng_rand` function (see section 7.1). The performance data of all these RNGs are in table 7.4 to 7.7.

	Loop			rng_rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
AES192x1	1.74	6.20	6.55	0.96	5.46	5.76
AES192x2	1.40	4.07	3.77	0.85	3.45	3.19
AES192x4	0.99	2.39	2.30	0.76	1.85	1.85
AES192x8	1.32	1.72	1.51	0.96	1.21	0.95
AES192x1_64	1.09	6.11	5.34	0.92	5.52	4.69
AES192x2_64	0.92	3.28	3.49	0.83	3.02	3.13
AES192x4_64	1.10	2.15	1.80	0.95	1.80	1.57
AES192x8_64	1.34	1.31	1.34	1.00	1.02	0.95

Table 7.5 Performance of AES192Engine

	Loop			rng_rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
AES256x1	1.94	7.49	5.85	1.07	6.56	5.23
AES256x2	1.37	3.95	4.35	1.00	3.44	3.74
AES256x4	1.16	2.37	2.04	0.91	1.90	1.68
AES256x8	1.42	1.59	1.77	1.04	1.16	1.03
AES256x1_64	1.31	7.05	5.95	1.07	6.44	5.52
AES256x2_64	1.08	4.08	3.40	0.99	3.78	3.07
AES256x4_64	1.36	2.15	2.22	1.19	1.85	1.94
AES256x8_64	1.38	1.47	1.49	1.05	1.16	1.05

Table 7.6 Performance of AES256Engine

7.3.2 *Philox*

The Philox algorithm in Salmon et al. (2011) is implemented in a more general form,

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

	Loop			rng_rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
ARSx1	0.70	3.77	4.12	0.55	3.03	3.13
ARSx2	1.07	2.18	2.76	0.47	1.65	2.08
ARSx4	0.56	1.76	1.68	0.40	1.19	1.17
ARSx8	0.67	1.18	1.48	0.34	0.70	0.93
ARSx1_64	0.55	3.09	4.21	0.55	2.56	3.39
ARSx2_64	0.47	2.74	2.26	0.43	2.20	1.79
ARSx4_64	0.48	1.33	1.58	0.40	0.98	1.20
ARSx8_64	0.50	1.07	1.09	0.35	0.70	0.79

Table 7.7 Performance of `ARSEngine`

```
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>>;

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>;
```

The default vector length and the number of rounds can be changed by configuration macros listed in table 7.3. Type aliases are also defined, as listed in table 7.2. The template parameter `Counstants` is similar to that of `ARSEngine`. It defines the round constants of the algorithm. The defaults are taken from the paper. See the reference manual for details. The performance data is in table 7.8.

	Loop			rng_rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
Philox2x32	3.66	6.95	3.99	2.31	4.78	2.51
Philox4x32	3.10	13.9	10.5	2.02	2.09	3.99
Philox2x64	2.02	4.08	3.17	1.07	2.37	1.73
Philox4x64	1.95	4.85	6.56	1.03	2.11	1.26
Philox2x32_64	3.04	6.54	3.19	2.21	4.61	2.30
Philox4x32_64	3.22	13.8	5.19	2.22	2.16	2.83
Philox2x64_64	1.73	3.97	3.13	1.06	2.49	2.25
Philox4x64_64	1.60	5.19	7.07	0.96	2.29	1.80

Table 7.8 Performance of PhiloxEngine

7.3.3 *Threefry*

The Threefry algorithm in Salmon et al. (2011) is implemented in a more general form,

```
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;

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>>;

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>;
```

	Loop			rng_rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
Threefry2x32	6.51	5.99	22.9	5.33	5.53	4.49
Threefry4x32	7.94	4.02	8.02	6.62	3.65	7.31
Threefry2x64	6.48	3.60	3.95	5.48	2.89	2.75
Threefry4x64	3.85	2.20	4.34	3.29	1.86	3.79
Threefry8x64	3.68	1.78	3.20	2.44	1.47	2.55
Threefry16x64	3.32	5.08	8.31	2.85	4.82	7.86
Threefry2x32_64	5.53	6.41	23.4	5.37	6.20	4.50
Threefry4x32_64	7.27	4.91	8.61	6.41	4.48	7.87
Threefry2x64_64	5.10	3.73	3.39	4.69	3.07	2.82
Threefry4x64_64	3.41	2.31	4.82	3.01	1.99	4.31
Threefry8x64_64	3.10	1.85	3.22	2.46	1.57	2.60
Threefry16x64_64	2.87	5.65	8.97	2.54	5.25	8.54

Table 7.9 Performance of ThreefryEngine

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

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

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

The default vector length and the number of rounds can be changed by configuration macros listed in table 7.3. Type aliases are also defined, as listed in table 7.2. The template parameter `Counstants` is similar to that of `ARSEngine`. It defines the round constants of the algorithm. The defaults are taken from the paper. See the reference manual for details. The performance data is in table 7.9.

7.3.4 *Default RNG*

Note that, not all RNGs defined by the library is available on all platforms. The library also defines a type alias `RNG` which is one of the RNGs listed in table 7.2. The preference is in the order listed in table 7.10. The user can define the configuration macro `VSMC_RNG_TYPE` to override the choice made by the library.

Class	Availability
ARS	VSMC_HAS_AES_NI
Threefry	Always available

Table 7.10 Default RNG

	Loop			rng_rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
MKL_MCG59	1.54	0.84	1.22	0.40	0.36	0.40
MKL_MCG59_64	0.72	0.71	1.01	0.37	0.36	0.37
MKL_MT19937	1.37	0.82	1.02	0.34	0.37	0.33
MKL_MT19937_64	0.60	0.62	0.93	0.31	0.32	0.34
MKL_MT2203	1.37	0.78	1.06	0.26	0.28	0.28
MKL_MT2203_64	0.61	0.71	0.87	0.26	0.34	0.24
MKL_STMT19937	1.36	0.75	0.95	0.21	0.21	0.19
MKL_STMT19937_64	0.65	0.65	0.90	0.21	0.21	0.20
MKL_NONDETERM	33.2	33.9	34.8	32.3	34.5	33.8
MKL_NONDETERM_64	33.0	33.1	33.3	33.6	33.4	33.4
MKL_ARS5	1.57	0.92	1.25	0.38	0.34	0.37
MKL_ARS5_64	0.87	0.83	0.92	0.36	0.36	0.31
MKL_PHILOX4X32X10	1.87	1.29	1.40	0.67	0.68	0.62
MKL_PHILOX4X32X10_64	1.10	1.28	1.27	0.65	0.78	0.61

Table 7.11 Performance of non-deterministic RNG

7.4 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 **??**.

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 7.12 MKL RNG; Each RNG engine may have suffix _64

7.5 MKL RNG

The MKL library provides some high performance RNGs. The library implements a wrapper class `MKLEngine` that makes them accessible as C++11 generators. They are listed in table 7.12. Note that, MKL RNGs performs 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 7.13.

7.6 MULTIPLE RNG STREAMS

Earlier in section 2.3.3 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,

```
1 RNGSet<RNG> rng_set(N); // A set of N RNGs
2 rng_set.resize(n);      // Change the size of the set
3 rng_set.seed();         // Seed each RNG in the set with Seed::instance()
4 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 $O(N)$. 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 $i \neq j$.

	Loop			rng_rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
MKL_MCG59	1.54	0.84	1.22	0.40	0.36	0.40
MKL_MCG59_64	0.72	0.71	1.01	0.37	0.36	0.37
MKL_MT19937	1.37	0.82	1.02	0.34	0.37	0.33
MKL_MT19937_64	0.60	0.62	0.93	0.31	0.32	0.34
MKL_MT2203	1.37	0.78	1.06	0.26	0.28	0.28
MKL_MT2203_64	0.61	0.71	0.87	0.26	0.34	0.24
MKL_STMT19937	1.36	0.75	0.95	0.21	0.21	0.19
MKL_STMT19937_64	0.65	0.65	0.90	0.21	0.21	0.20
MKL_NONDETERM	33.2	33.9	34.8	32.3	34.5	33.8
MKL_NONDETERM_64	33.0	33.1	33.3	33.6	33.4	33.4
MKL_ARS5	1.57	0.92	1.25	0.38	0.34	0.37
MKL_ARS5_64	0.87	0.83	0.92	0.36	0.36	0.31
MKL_PHILOX4X32X10	1.87	1.29	1.40	0.67	0.68	0.62
MKL_PHILOX4X32X10_64	1.10	1.28	1.27	0.65	0.78	0.61

Table 7.13 Performance of MKL 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,

```
1 std::size_t eval_pre(SingleParticle<T> sp)
2 {
3     auto &rng = sp.rng();
4     // using rng to initialize state
5     // do some computation, likely far more costly than TLS
6 }
```

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,

```
1 class T
2 {
```

Class	Parameters
`LaplaceDistribution`	location `a`; scale `b`
`LevyDistribution`	location `a`; scale `b`
`ParetoDistribution`	shape `a`; scale `b`
`RayleighDistribution`	scale `sigma`

Table 7.14 Random number distributions

```
3     using rng_set_type = RNGSetScalar<RNG>;
4 };
```

will replace the type of the RNG set contained in `Particle<T>`. Note that, `Particle<T>` itself does not use any RNG in the set.

7.7 DISTRIBUTIONS

The library also provides implementations of some common distributions. They all conforms to the C++11 random number distribution concepts. Some of them are the same as those in the C++11 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. Table 7.14 lists all the additional distributions implemented. As stated in section 7.1, they support vectorized random number generating. In the following sections we introduce each distributions included in the library.

7.7.1 *Uniform bits distribution*

The class template,

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

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

```
1 RNG rng;
2 UniformBitsDistribution<std::uint32_t> dist;
3 dist(rng); // Return 32-bits random integers
```

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

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

where $K = \lceil W/M \rceil$. To see that it produces the desired results, note that there is an one-to-one mapping between $(r_0, \ldots, r_{K+1})$ and $\sum_{k=0}^{K-1} r_k^{kM}$, and $2^{KM} \equiv 0 \mod 2^W$. Or a more intuitive way is that, for $r_i - r_{\min}$ to be i.i.d. random integers on the set $S_M = \{0, \ldots, 2^M - 1\}$, the RNG shall produce M independent random bits. And U is formed by the lower W bits of each KM random bits produced by the RNG. 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` is used, and hence lower performance.

7.7.2 *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 for this purpose,

```
1 template <typename RealType>
2 class U01CCDistribution;

3 template <typename RealType>
4 class U01CODistribution;

5 template <typename RealType>
6 class U01OCDistribution;

7 template <typename RealType>
8 class U01OODistribution;

9 template <typename RealType>
10 class U01Distribution;
```

The last `U01Distribution` is used by all other distributions discussed later. If the configuration macro `VSMC_RNG_U01_USE_FIXED_POINT` is zero, then it behaves similarly to `std::generate_canonical` and thus `std::uniform_real_distribution`, otherwise it is the same as `U01CODistribution`. We now discuss the details of each distribution and the implication of configuration macros.

First of all, 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. If `RealType` is `long double` or the RNG produce random integers with at least 64 bits, then 64-bits integers are used. If the configuration macro `VSMC_RNG_U01_USE_64BITS_DOUBLE` is defined to be an non-zero value and `RealType` is `double`, then 64-bits integers are also used. Otherwise, 32-bits random integers are used. Below, we let W be the number of bits of the random integers, and M be the number of significant bits (including the implicit one) of `RealType`, which is usually 24 for `float`, 53 for `double`. The situation for `long double` is more complicated. On x86 and some other platforms, it is 64. We also denote the input random integers as U and the output random real numbers as Y. Intermediate integer V and real X might also be used.

U01CCDistribution

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

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

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. Consider the following dummy example,

```
1 std::size_t n = 10000;
2 RNG rng;
3 std::uniform_real_distribution<double> runif_std;
4 U01CODistribution<double> runif_vsmc;
5 Vector<double> r(n);

6 // Method 1
7 for (std::size_t i = 0; i != n; ++i)
```

```
    r[i] = runif_std(rng);

// Method 2
for (std::size_t i = 0; i != n; ++i)
    r[i] = runif_vsmc(rng);

// Method 3
rng_rand(rng, runif_vsmc, n, r.data());
```

On the author's computer, on average method 1 costs 29 cycles to generate one standard uniform random number. Method 2 costs 6.1 cycles while method 3 costs only 2.4 cycles. The best performance is obtained when the output of RNG are either 32 or 64 random bits. All RNGs in this library satisfy this condition.

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. In fact, if the RNG produces 32 random bits, then the results are identical to the standard library for `float`. If it produces 64 random bits, then the results are also identical for `double` or `long double`.

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[2] 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.

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

7.7.3 *Distributions using the inverse method*

The following class templates implement distributions using the inverse method. Their distribution functions are shown in the comments on each class template.

```
// F(x) = 1/π arctan((x−a)/b) + 1/2
template <typename RealType>
class CauchyDistribution;

// F(x) = 1 − e^{−λx}
template <typename RealType>
class ExponentialDistribution;

// F(x) = exp{− exp(−(x−a)/b)}
template <typename RealType>
class ExtremeValueDistribution;

// F(x) = 1/2 + 1/2 sgn(x − a)(1 − exp{−|x−a|/b})
template <typename RealType>
class LaplaceDistribution;

// F(x) = (1 + exp{−(x−a)/b})^{−1}
template <typename RealType>
class LogisticDistribution;

// F(x) = 1 − (b/x)^a
template <typename RealType>
class ParetoDistribution;

// F(x) = 1 − exp{−x²/(2σ²)}
template <typename RealType>
class RayleighDistribution;

// F(x) = (x−a)/(b−a)
template <typename RealType>
class UniformRealDistribution;

// F(x) = 1 − exp{−(x/b)^a}
```

Distribution	STD/Boost	vSMC	rng_rand	MKL
Cauchy(0,1)	92.1	72.5	14.5	7.82
Exponential(1)	69.7	60.5	8.83	6.56
ExtremeValue(0,1)	120	66.7	14.0	15.5
Laplace(0,1)	77.8	74.5	9.35	9.01
Logistic(0,1)	–	37.7	13.3	12.5
Pareto(1,1)	–	95.0	11.9	10.4
Rayleigh(1)	–	73.7	11.6	9.28
UniformReal(-0.5,0.5)	31.3	12.7	4.25	2.50
Weibull(1,1)	145	77.8	9.79	8.29

Table 7.15 Performance of distributions using the inverse method

```
26 template <typename RealType>
27 class WeibullDistribution;
```

The performance data of these distributions is in table 7.15.

7.7.4 *Normal and related distribution*

The class template

```
1 // f(x) = 1/sqrt(2πσ²) exp{-(x-μ)²/(2σ²)}
2 template <typename RealType>
3 class NormalDistribution;
```

implements the Normal distribution with the Box-Muller method (Box and Muller 1958). It is also used to implement the Log-Normal and Levy distributions. Below in the comments, z is a the standard Normal random number and x is the target distribution random number,

```
1 // x = e^(m+sz)
2 template <typename RealType>
3 class LognormalDistribution;

4 // x = a + b/z²
5 template <typename RealType>
6 class LevyDistribution;
```

The performance data of these distributions is in table 7.16.

The library also implements the multivariate Normal distribution.

Distribution	STD/Boost	vSMC	rng_rand	MKL
Levy(0,1)	–	67.5	20.2	15.8
Lognormal(0,1)	124	79.7	18.8	13.1
Normal(0,1)	101	58.1	13.8	9.47

Table 7.16 Performance of Normal and related distributions

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

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

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

Otherwise, if `Dim` is positive integer, it can be constructed with static size,

```
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 it is a zero vector. The parameter `chol` is a $d(d+1)/2$-vector, where d is 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 generate such a matrix. Alternatively, one can use the covariance functionalities in the library. See section 8.2, which also provides a concrete example of using the multivariate Normal distribution.

7.7.5 *Gamma and related distribution*

The class template

```
// f(x) = e^{-x/β} / (Γ(α) β^α) x^{α-1}
template <typename RealType>
class GammaDistribution;
```

implements the Gamma distribution. The specific algorithms depends on the parameters. If $\alpha = 1$, it becomes the exponential distribution. Otherwise, if $\alpha < 0.6$, 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,

Distribution	STD/Boost	vSMC	rng_rand	MKL
Gamma(1,1)	88.1	58.3	11.3	6.63
Gamma(0.1,1)	184	169	38.4	38.9
Gamma(0.5,1)	229	192	61.5	61.0
Gamma(0.7,1)	278	203	49.1	43.6
Gamma(0.9,1)	260	143	33.1	28.2
Gamma(1.5,1)	281	143	33.4	27.5

Table 7.17 Performance of Gamma distribution

```
// GammaDistribution<RealType> rgamma(n / 2, 2);
// x = rgamma(rng);
template <typename RealType>
class ChiSquaredDistribution;

// ChiSquaredDistribution<RealType> rchi1(m);
// ChiSquaredDistribution<RealType> rchi2(n);
// x = (rchi1(rng) / m) / (rchi2(rng) / n);
template <typename RealType>
class FisherFDistribution;

// NormalDistribution<RealType> rnorm(0, 1);
// ChiSquaredDistribution<RealType> rchi(n);
// x = rnorm(rng) * std::sqrt(n / rchi(rng()));
template <typename RealType>
class StudentTDistribution;
```

They implement the χ^2-distribution, the Fisher F-distribution, and the Student's t-distribution, respectively. The performance data of these distributions, for different parameters is in table 7.17 to 7.20.

7.7.6 *Beta distribution*

The class template

```
// f(x) = Γ(α+β)/(Γ(α)Γ(β)) x^(α-1) (1-x)^(β-1)
template <typename RealType>
class BetaDistribution;
```

Distribution	STD/Boost	vSMC	rng_rand	MKL
ChiSquared(0.2)	199	126	31.8	33.6
ChiSquared(1)	233	171	59.1	51.2
ChiSquared(1.5)	236	169	48.7	35.3
ChiSquared(2)	79.2	44.2	8.75	6.73
ChiSquared(3)	280	147	35.9	32.2
ChiSquared(30)	251	147	30.0	26.0

Table 7.18 Performance of χ^2 distribution

implements the Beta distribution. The specific algorithm used depend 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 7.21

7.8 SEEDING

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

```
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 `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 seed_txt("seed.txt");
if (seed_txt.good())
```

Distribution	STD/Boost	vSMC	rng_rand	MKL
FisherF(0.2,0.2)	401	333	78.9	97.0
FisherF(0.2,1)	389	319	123	105
FisherF(0.2,1.5)	459	391	123	82.8
FisherF(0.2,2)	275	194	49.7	52.6
FisherF(0.2,3)	449	318	87.5	101
FisherF(1,0.2)	405	322	136	110
FisherF(1,1)	439	354	141	113
FisherF(1,1.5)	501	369	137	99.8
FisherF(1,2)	325	259	87.6	66.2
FisherF(1,3)	480	341	140	109
FisherF(2,0.2)	255	193	66.8	57.6
FisherF(2,1)	362	271	107	82.6
FisherF(2,1.5)	372	256	91.4	68.8
FisherF(2,2)	203	132	25.7	21.6
FisherF(2,3)	367	222	50.5	45.3
FisherF(3,0.2)	470	319	82.9	76.0
FisherF(3,1)	529	343	110	97.3
FisherF(3,1.5)	540	416	127	89.4
FisherF(3,2)	439	248	66.2	56.8
FisherF(3,3)	685	397	102	87.3

Table 7.19 Performance of Fisher's F-distribution

```
3     seed_txt >> Seed::instance(); // Read seed from a file
4 else
5     Seed::instance().set(101);    // The default seed
6 seed_txt.close();
7 // The program
8 std::ofstream seed_txt("seed.txt");
9 seed_txt << Seed::instance();     // Write the seed to a file
10 seed_txt.close();
```

This way, if the simulation program need to be repeated multiple times, each time is will use a different set of seeds.

A single seed generator is enough for a single computer 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

Distribution	STD/Boost	vSMC	rng_rand	MKL
StudentT(0.2)	294	214	60.5	69.9
StudentT(1)	354	254	88.9	72.8
StudentT(1.5)	396	255	83.0	59.0
StudentT(2)	201	129	43.0	27.8
StudentT(3)	389	268	64.7	55.6
StudentT(30)	371	248	83.2	55.3

Table 7.20 Performance of Student's t-distribution

Distribution	STD/Boost	vSMC	rng_rand	MKL
Beta(0.5,0.5)	445	51.9	12.6	48.2
Beta(1,1)	135	15.4	6.40	9.85
Beta(1,0.5)	307	65.4	12.9	11.3
Beta(1,1.5)	401	64.2	12.0	9.23
Beta(0.5,1)	288	61.4	11.8	9.37
Beta(1.5,1)	456	66.2	14.1	11.0
Beta(1.5,1.5)	692	176	48.1	41.1
Beta(0.3,0.3)	432	150	57.7	36.8
Beta(0.9,0.9)	483	169	169	48.3
Beta(1.5,0.5)	549	207	201	58.8
Beta(0.5,1.5)	546	200	200	57.7

Table 7.21 Performance of Beta distribution

the `modulo` method of `SeedGenerator`. For example,

```
1 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 (seed section 7.3. This seed generator is only suitable for counter-based RNGs.

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 most simple cases.

8.1 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 = Alignment<T>::value,
    typename Memory = AlignedMemory>
class Allocator;
```

which conforms to the `std::allocator` interface. The address of the pointer return 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 a few implementations, listed in table 8.1. In addition, a type alias `AlignedMemory` is defined to be one of the class listed in the table, depending on the availability of those classes, with preference in the same order as they are listed. The user can define the configuration macro `VSMC_ALIGNED_MEMORY_TYPE` to override the choice made by the library.

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 AVX2. For other types, the alignment is the maximum of `alignof(T)` and `VSMC_ALIGNMENT_MIN`, whose default is 16.

Some classes in the library are over-aligned to make efficient use of SIMD operations. Those classes' `operator new` and related methods are overloaded using `AlignedMemory`.

Last, a type alias `Vector` is defined,

Class	Notes
`AlignedMemoryTBB`	Use `scalable_aligned_malloc` and `scalable_aligned_free`. Defined if `VSMC_HAS_TBB_MALLOC` is defined to a non-zero value.
`AlignedMemorySYS`	Use `posix_memalign` and `free` on POSIX platforms. Use `_aligned_malloc` and `_aligned_free` if using MSVC. Defined if `VSMC_HAS_POSIX` is defined to a non-zero value, or the MSVC compiler is detected.
`AlignedMemoryMKL`	Use `mkl_malloc` and `mkl_free`. Defined if `VSMC_HAS_MKL` is defined to a non-zero value.
`AlignedMemorySTD`	Use `std::malloc` and `std::free`. Always defined.

Table 8.1 Aligned memory allocation

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

This vector type is used throughout the library.

8.2 SAMPLE COVARIANCE

The library provides some basic functionality to estimate sample covariance. For example,

```
constexpr std::size_t d = /* Dimension */;
using T = StateMatrix<RowMajor, d, double>;
Sampler<T> sampler(N);
// operations on the sampler
double mean[d];
double cov[d * d];
Covariance eval;
auto x = sampler.particle().value().data();
auto w = sampler.particle().weight().data();
eval(RowMajor, N, d, x, w, mean, cov);
```

The sample covariance matrix will be computed and stored in `cov`. The mean vector is stored in `mean`. Note that, if any of them is a null pointer, then the corresponding output is not computed. The sample `x` is assumed to be stored in an N by d matrix. The first argument passed to `eval` is the storage layout of this matrix. If `x` is a null pointer, then no computation will be done. If `w` is a null pointer, then the weight is assumed to be equal for all samples. This method has three optional parameters. The first is `cov_layout`, which specifies the storage layout of `cov`. The second is `cov_upper` and the third is `cov_packed`, both are

`false` by default. If the later is `true`, a packed vector of length $d(d+1)/2$ is written into `cov`. If `cov_upper` is `true`, then the upper triangular is packed, otherwise the lower triangular is packed.

The estimated covariance matrix is often used to construct multivariate Normal distribution for the purpose of generating random walk proposals. The `NormalMVDistribution` in section 7.7 accepts the lower triangular of the Cholesky decomposition of the covariance matrix instead of itself. The following function `chol` will compute this decomposition,

```
1 double chol_mat[d * (d + 1) / 2];
2 chol(d, cov, chol_mat);
3 NormalMVDistribution<double> normal_mv(d, nullptr, chol_mat); // zero mean
```

The output `chol_mat` is a packed vector in row major storage. This function also has three optional parameters, which are the same as those of `Covariance::operator()`, except that they are now used to specify the storage scheme of the input parameter `cov`.

8.3 STORE OBJECTS IN HDF5 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,

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

creates a 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,

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

This creates a `data.frame` similar to that shown in section 2.4.2. 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 available. Such an overload is automatically available if `T` is a derived class of `StateMatrix`. However, it may not be the most suitable one. Other types of objects can also be stored, see the reference manual for details.

8.4 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,

```
1 auto platform = cl_get_platform().front();
2 auto device = platform.get_device(CL_DEVICE_TYPE_DEFAULT).front();
```

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 8.2 RAII classes for OpenCL pointers

```
3 CLContext context(CLContextProperties(platform), 1, &device);
4 CLCommandQueue command_queue(context, device);
5 CLMemory buffer(context, CL_MEM_READ_WRITE, size);
6 std::string source = /* read source */;
7 CLProgram program(context, 1, &source);
8 program.build(1, &device);
9 CLKernel kernel(program, "kernel_name");
10 kernel.set_arg(0, buffer);
11 command_queue.enqueue_nd_range_kernel(kernel, 1, CLNDRange(), CLNDRange(N),
12     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 B.1.5 an OpenCL implementation of the simple particle filter example in section 2.4 is shown. Table 8.2 lists the classes defined by the library and their corresponding OpenCL pointers.

8.5 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 B.2. First, we allocate define to store values of options,

```
1 int n;
2 std::string str;
3 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 options, `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 gather together and inserted into the vector.

8.6 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. The complete program is shown in appendix B.3.

```
Progress progress;
progress.start(n * n);
for (std::size_t i = 0; i != n; ++i) {
    std::stringstream ss;
```

```
    ss << "i = " << i;
    progress.message(ss.str());
    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 `progress.start(n * n)` starts the printing of the progress. The argument specifies how many iterations there will be before it is stopped. The method `progress.message(ss.str())` 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 `progress.increment()`. And after everything is finished, the method `progress.stop()` is called. The `increment` method has an optional argument, which specifies how many steps has been finished. The default is one. For example, we can call `progress.start(n * n * n)` and `progress.increment(n)` instead.

8.7 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 `watch.start()` and `watch.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.

Cappé, Olivier, Simon J. Godsill, and Eric Moulines (2007). "An overview of existing methods and recent advances in sequential Monte Carlo". In: *Proceedings of the IEEE* 95.5, pp. 899–924.

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

Del Moral, Pierre, Arnaud Doucet, and Ajay Jasra (2006a). "Sequential Monte Carlo methods for Bayesian computation". In: *Bayesian Statistics 8*. Oxford University Press, pp. 1–34.

— (2006b). "Sequential Monte Carlo samplers". In: *Journal of Royal Statistical Society B* 68.3, pp. 411–436.

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.

Doucet, Arnaud and Adam M. Johansen (2011). "A tutorial on particle filtering and smoothing: Fifteen years later". In: *The Oxford Handbook of Non-linear Filtering*. Oxford University Press, pp. 1–37.

Johansen, Adam M. (2009). "SMCTC: sequential Monte Carlo in C++". In: *Journal of Statistical Software* 30.6, pp. 1–41.

Lee, Anthony et al. (2010). "On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods". In: *Journal of Computational and Graphical Statistics* 19.4, pp. 769–789.

Liu, Jun S. and Rong Chen (1998). "Sequential Monte Carlo methods for dynamic systems". In: *Journal of the American Statistical Association* 93.443, pp. 1032–1044.

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.

Neal, Radford M. (2001). "Annealed importance sampling". In: *Statistics and Computing* 11.2, pp. 125–139.

Peters, Gareth W (2005). "Topics in sequential Monte Carlo samplers". MA thesis.

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 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 2.4 are shown in appendices B.1.3 and B.1.4. See the reference manual for complete list of the API. In this appendix, we introduce the conventions used in the API.

A.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);
```

A.2 CLASSES AND METHODS

A few core classes are accessible from this API. They are listed in table A.1. Table A.2 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);
```

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 a.1 C types . Note that, all template parameter `T` are of type `StateMatrix<RowMajor, Dynamic, double>`

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);
```

Feature	Notes
SMP backends	Section 2.5
Resamling algorithms	Section 6.1
Ordered uniform random numbers	Section 6.2
Seeding	Section 7.8
Vectorized random number generating	Section 7.7
Aligned memory allocation	Section 8.1
Covariance matrix	Section 8.2
Process program command line options	Section 8.5
Display program progress	Section 8.6
Stop watch	Section 8.7
Registering C++11 RNG as MKL BRNG	Reference manual
Random walk	Reference manual

Table a.2 Features accessible from C

B SOURCE CODE OF COMPLETE PROGRAMS

B.1 A SIMPLE PARTICLE FILTER

B.1.1 *Sequential implementation*

```
1  #include <vsmc/vsmc.hpp>

2  static constexpr std::size_t PosX = 0;
3  static constexpr std::size_t PosY = 1;
4  static constexpr std::size_t VelX = 2;
5  static constexpr std::size_t VelY = 3;

6  using PFStateBase = vsmc::StateMatrix<vsmc::RowMajor, 4, double>;

7  class PFState : public PFStateBase
8  {
9      public:
10     using PFStateBase::PFStateBase;

11     double log_likelihood(std::size_t t, size_type i) const
12     {
13         double llh_x = 10 * (this->state(i, PosX) - obs_x_[t]);
14         double llh_y = 10 * (this->state(i, PosY) - obs_y_[t]);
15         llh_x = std::log(1 + llh_x * llh_x / 10);
16         llh_y = std::log(1 + llh_y * llh_y / 10);

17         return -0.5 * (10 + 1) * (llh_x + llh_y);
18     }

19     void read_data(const char *param)
20     {
21         if (param == nullptr)
22             return;

23         std::ifstream data(param);
```

```
24          while (data.good()) {
25              double x;
26              double y;
27              data >> x >> y;
28              if (data.good()) {
29                  obs_x_.push_back(x);
30                  obs_y_.push_back(y);
31              }
32          }
33          data.close();
34      }

35      std::size_t data_size() const { return obs_x_.size(); }

36      private:
37      vsmc::Vector<double> obs_x_;
38      vsmc::Vector<double> obs_y_;
39  };

40  class PFInit
41  {
42      public:
43      std::size_t operator()(vsmc::Particle<PFState> &particle, void *param)
44      {
45          eval_param(particle, param);
46          eval_pre(particle);
47          std::size_t acc = 0;
48          for (auto sp : particle)
49              acc += eval_sp(sp);
50          eval_post(particle);

51          return acc;
52      }

53      void eval_param(vsmc::Particle<PFState> &particle, void *param)
54      {
55          particle.value().read_data(static_cast<const char *>(param));
56      }
```

```
57      void eval_pre(vsmc::Particle<PFState> &particle)
58      {
59          weight_.resize(particle.size());
60      }

61      std::size_t eval_sp(vsmc::SingleParticle<PFState> sp)
62      {
63          vsmc::NormalDistribution<double> norm_pos(0, 2);
64          vsmc::NormalDistribution<double> norm_vel(0, 1);
65          sp.state(PosX) = norm_pos(sp.rng());
66          sp.state(PosY) = norm_pos(sp.rng());
67          sp.state(VelX) = norm_vel(sp.rng());
68          sp.state(VelY) = norm_vel(sp.rng());
69          weight_[sp.id()] = sp.particle().value().log_likelihood(0, sp.id());

70          return 0;
71      }

72      void eval_post(vsmc::Particle<PFState> &particle)
73      {
74          particle.weight().set_log(weight_.data());
75      }

76      private:
77      vsmc::Vector<double> weight_;
78  };

79  class PFMove
80  {
81      public:
82      std::size_t operator()(std::size_t t, vsmc::Particle<PFState> &particle)
83      {
84          eval_pre(t, particle);
85          std::size_t acc = 0;
86          for (auto sp : particle)
87              acc += eval_sp(t, sp);
88          eval_post(t, particle);
```

```
89          return 0;
90      }

91      void eval_pre(std::size_t t, vsmc::Particle<PFState> &particle)
92      {
93          auto &rng = particle.rng();
94          const std::size_t size = particle.size();
95          const double sd_pos = sqrt(0.02);
96          const double sd_vel = sqrt(0.001);
97          pos_x_.resize(size);
98          pos_y_.resize(size);
99          vel_x_.resize(size);
100         vel_y_.resize(size);
101         weight_.resize(size);
102         vsmc::normal_distribution(rng, size, pos_x_.data(), 0.0, sd_pos);
103         vsmc::normal_distribution(rng, size, pos_y_.data(), 0.0, sd_pos);
104         vsmc::normal_distribution(rng, size, vel_x_.data(), 0.0, sd_vel);
105         vsmc::normal_distribution(rng, size, vel_y_.data(), 0.0, sd_vel);
106     }

107     std::size_t eval_sp(std::size_t t, vsmc::SingleParticle<PFState> sp)
108     {
109         sp.state(PosX) += pos_x_[sp.id()] + 0.1 * sp.state(VelX);
110         sp.state(PosY) += pos_y_[sp.id()] + 0.1 * sp.state(VelY);
111         sp.state(VelX) += vel_x_[sp.id()];
112         sp.state(VelY) += vel_y_[sp.id()];
113         weight_[sp.id()] = sp.particle().value().log_likelihood(t, sp.id());

114         return 0;
115     }

116     void eval_post(std::size_t t, vsmc::Particle<PFState> &particle)
117     {
118         particle.weight().add_log(weight_.data());
119     }

120     private:
```

```
121     vsmc::Vector<double> pos_x_;
122     vsmc::Vector<double> pos_y_;
123     vsmc::Vector<double> vel_x_;
124     vsmc::Vector<double> vel_y_;
125     vsmc::Vector<double> weight_;
126 };

127 class PFEval
128 {
129     public:
130     void operator()(std::size_t t, std::size_t dim,
131         vsmc::Particle<PFState> &particle, double *r)
132     {
133         eval_pre(t, particle);
134         for (std::size_t i = 0; i != particle.size(); ++i, r += dim)
135             eval_sp(t, dim, particle.sp(i), r);
136         eval_post(t, particle);
137     }

138     void eval_pre(std::size_t t, vsmc::Particle<PFState> &particle) {}

139     void eval_sp(std::size_t t, std::size_t dim,
140         vsmc::SingleParticle<PFState> sp, double *r)
141     {
142         r[0] = sp.state(PosX);
143         r[1] = sp.state(PosY);
144     }

145     void eval_post(std::size_t t, vsmc::Particle<PFState> &particle) {}
146 };

147 int main(int argc, char **argv)
148 {
149     std::size_t N = 10000;
150     if (argc > 1)
151         N = static_cast<std::size_t>(std::atoi(argv[1]));

152     vsmc::Sampler<PFState> sampler(N, vsmc::Multinomial, 0.5);
```

```
153     sampler.init(PFInit()).move(PFMove(), false).monitor("pos", 2, PFEval());

154     vsmc::StopWatch watch;
155     watch.start();
156     sampler.initialize(const_cast<char *>("pf.data"));
157     sampler.iterate(sampler.particle().value().data_size() - 1);
158     watch.stop();
159     std::cout << "Time (ms): " << watch.milliseconds() << std::endl;

160     std::ofstream output("pf.out");
161     output << sampler;
162     output.close();

163     return 0;
164 }
```

B.1.2 *Parallelized implementation using TBB*

```
 1 #include <vsmc/vsmc.hpp>

 2 using PFStateBase = vsmc::StateMatrix<vsmc::RowMajor, 4, double>;

 3 template <typename T>
 4 using PFStateSPBase = PFStateBase::single_particle_type<T>;

 5 class PFState : public PFStateBase
 6 {
 7     public:
 8     using PFStateBase::StateMatrix;

 9     template <typename S>
10     class single_particle_type : public PFStateSPBase<S>
11     {
12         public:
13         using PFStateSPBase<S>::single_particle_type;

14         double &pos_x() { return this->state(0); }
15         double &pos_y() { return this->state(1); }
```

```
16          double &vel_x() { return this->state(2); }
17          double &vel_y() { return this->state(3); }

18          double log_likelihood(std::size_t t)
19          {
20              double llh_x = 10 * (pos_x() - obs_x(t));
21              double llh_y = 10 * (pos_y() - obs_y(t));
22              llh_x = std::log(1 + llh_x * llh_x / 10);
23              llh_y = std::log(1 + llh_y * llh_y / 10);

24              return -0.5 * (10 + 1) * (llh_x + llh_y);
25          }

26          private:
27          double obs_x(std::size_t t)
28          {
29              return this->particle().value().obs_x_[t];
30          }

31          double obs_y(std::size_t t)
32          {
33              return this->particle().value().obs_y_[t];
34          }
35      };

36      void read_data(const char *param)
37      {
38          if (param == nullptr)
39              return;

40          std::ifstream data(param);
41          while (data.good()) {
42              double x;
43              double y;
44              data >> x >> y;
45              if (data.good()) {
46                  obs_x_.push_back(x);
47                  obs_y_.push_back(y);
```

```
48              }
49          }
50          data.close();
51      }

52      std::size_t data_size() const { return obs_x_.size(); }

53      private:
54      vsmc::Vector<double> obs_x_;
55      vsmc::Vector<double> obs_y_;
56  };

57  class PFInit : public vsmc::InitializeTBB<PFState, PFInit>
58  {
59      public:
60      void eval_param(vsmc::Particle<PFState> &particle, void *param)
61      {
62          particle.value().read_data(static_cast<const char *>(param));
63      }

64      void eval_pre(vsmc::Particle<PFState> &particle)
65      {
66          weight_.resize(particle.size());
67      }

68      std::size_t eval_sp(vsmc::SingleParticle<PFState> sp)
69      {
70          vsmc::NormalDistribution<double> norm_pos(0, 2);
71          vsmc::NormalDistribution<double> norm_vel(0, 1);
72          sp.pos_x() = norm_pos(sp.rng());
73          sp.pos_y() = norm_pos(sp.rng());
74          sp.vel_x() = norm_vel(sp.rng());
75          sp.vel_y() = norm_vel(sp.rng());
76          weight_[sp.id()] = sp.log_likelihood(0);

77          return 0;
78      }
```

```
79      void eval_post(vsmc::Particle<PFState> &particle)
80      {
81          particle.weight().set_log(weight_.data());
82      }

83      private:
84      vsmc::Vector<double> weight_;
85  };

86  class PFMove : public vsmc::MoveTBB<PFState, PFMove>
87  {
88      public:
89      void eval_pre(std::size_t t, vsmc::Particle<PFState> &particle)
90      {
91          weight_.resize(particle.size());
92      }

93      std::size_t eval_sp(std::size_t t, vsmc::SingleParticle<PFState> sp)
94      {
95          vsmc::NormalDistribution<double> norm_pos(0, std::sqrt(0.02));
96          vsmc::NormalDistribution<double> norm_vel(0, std::sqrt(0.001));
97          sp.pos_x() += norm_pos(sp.rng()) + 0.1 * sp.vel_x();
98          sp.pos_y() += norm_pos(sp.rng()) + 0.1 * sp.vel_y();
99          sp.vel_x() += norm_vel(sp.rng());
100         sp.vel_y() += norm_vel(sp.rng());
101         weight_[sp.id()] = sp.log_likelihood(t);

102         return 0;
103     }

104     void eval_post(std::size_t t, vsmc::Particle<PFState> &particle)
105     {
106         particle.weight().add_log(weight_.data());
107     }

108     private:
109     vsmc::Vector<double> weight_;
110 };
```

```
111 class PFEval : public vsmc::MonitorEvalTBB<PFState, PFEval>
112 {
113     public:
114     void eval_sp(std::size_t t, std::size_t dim,
115         vsmc::SingleParticle<PFState> sp, double *r)
116     {
117         r[0] = sp.pos_x();
118         r[1] = sp.pos_y();
119     }
120 };

121 int main(int argc, char **argv)
122 {
123     std::size_t N = 10000;
124     if (argc > 1)
125         N = static_cast<std::size_t>(std::atoi(argv[1]));

126     vsmc::Sampler<PFState> sampler(N, vsmc::Multinomial, 0.5);
127     sampler.init(PFInit()).move(PFMove(), false).monitor("pos", 2, PFEval());

128     vsmc::StopWatch watch;
129     watch.start();
130     sampler.initialize(const_cast<char *>("pf.data"));
131     sampler.iterate(sampler.particle().value().data_size() - 1);
132     watch.stop();
133     std::cout << "Time (ms): " << watch.milliseconds() << std::endl;

134     std::ofstream output("pf.out");
135     output << sampler;
136     output.close();

137     return 0;
138 }
```

B.1.3 *Sequential implementation in C*

```c
#include <vsmc/vsmc.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

static const int PosX = 0;
static const int PosY = 1;
static const int VelX = 2;
static const int VelY = 3;

typedef struct {
    double *ptr;
    size_t size;
} pf_vector;

// Storage for data
static vsmc_vector pf_obs_x = {NULL, 0};
static vsmc_vector pf_obs_y = {NULL, 0};

// Temporaries used by pf_init and pf_move
static vsmc_vector pf_pos_x = {NULL, 0};
static vsmc_vector pf_pos_y = {NULL, 0};
static vsmc_vector pf_vel_x = {NULL, 0};
static vsmc_vector pf_vel_y = {NULL, 0};
static vsmc_vector pf_weight = {NULL, 0};

static inline double pf_log_likelihood(int t, const vsmc_single_particle *sp)
{
    double llh_x = 10 * (sp->state[PosX] - pf_obs_x.data[t]);
    double llh_y = 10 * (sp->state[PosY] - pf_obs_y.data[t]);
    llh_x = log(1 + llh_x * llh_x / 10);
    llh_y = log(1 + llh_y * llh_y / 10);

    return -0.5 * (10 + 1) * (llh_x + llh_y);
}
```

```
30 static inline void pf_read_data(const char *param)
31 {
32     if (!param)
33         return;

34     FILE *data = fopen(param, "r");
35     int n = 0;
36     while (1) {
37         double x;
38         double y;
39         int nx = fscanf(data, "%lg", &x);
40         int ny = fscanf(data, "%lg", &y);
41         if (nx == 1 && ny == 1)
42             ++n;
43         else
44             break;
45     }
46     vsmc_vector_resize(&pf_obs_x, n);
47     vsmc_vector_resize(&pf_obs_y, n);
48     fseek(data, 0, SEEK_SET);
49     for (int i = 0; i < n; ++i) {
50         fscanf(data, "%lg", &pf_obs_x.data[i]);
51         fscanf(data, "%lg", &pf_obs_y.data[i]);
52     }
53     fclose(data);
54 }

55 static inline void pf_normal(
56     vsmc_particle particle, double sd_pos, double sd_vel)
57 {
58     vsmc_rng rng = vsmc_particle_rng(particle, 0);
59     const int size = vsmc_particle_size(particle);
60     vsmc_vector_resize(&pf_pos_x, size);
61     vsmc_vector_resize(&pf_pos_y, size);
62     vsmc_vector_resize(&pf_vel_x, size);
63     vsmc_vector_resize(&pf_vel_y, size);
64     vsmc_vector_resize(&pf_weight, size);
65     vsmc_normal_rand(rng, size, pf_pos_x.data, 0, sd_pos);
```

```
66      vsmc_normal_rand(rng, size, pf_pos_y.data, 0, sd_pos);
67      vsmc_normal_rand(rng, size, pf_vel_x.data, 0, sd_vel);
68      vsmc_normal_rand(rng, size, pf_vel_y.data, 0, sd_vel);
69  }

70  static inline int pf_init(vsmc_particle particle, void *param)
71  {
72      pf_read_data((const char *) param);

73      pf_normal(particle, 2, 1);

74      const int size = vsmc_particle_size(particle);
75      for (int i = 0; i < size; ++i) {
76          vsmc_single_particle sp = vsmc_particle_sp(particle, i);
77          sp.state[PosX] = pf_pos_x.data[i];
78          sp.state[PosY] = pf_pos_y.data[i];
79          sp.state[VelX] = pf_vel_x.data[i];
80          sp.state[VelY] = pf_vel_y.data[i];
81          pf_weight.data[i] = pf_log_likelihood(0, &sp);
82      }

83      vsmc_weight_set_log(vsmc_particle_weight(particle), pf_weight.data, 1);

84      return 0;
85  }

86  static inline int pf_move(int t, vsmc_particle particle)
87  {
88      pf_normal(particle, sqrt(0.02), sqrt(0.001));

89      const int size = vsmc_particle_size(particle);
90      for (int i = 0; i < size; ++i) {
91          vsmc_single_particle sp = vsmc_particle_sp(particle, i);
92          sp.state[PosX] += pf_pos_x.data[i] + 0.1 * sp.state[VelX];
93          sp.state[PosY] += pf_pos_y.data[i] + 0.1 * sp.state[VelY];
94          sp.state[VelX] += pf_vel_x.data[i];
95          sp.state[VelY] += pf_vel_y.data[i];
96          pf_weight.data[i] = pf_log_likelihood(t, &sp);
```

```
97      }

98      vsmc_weight_add_log(vsmc_particle_weight(particle), pf_weight.data, 1);

99      return 0;
100 }

101 static inline void pf_eval(int t, int dim, vsmc_particle particle, double *r)
102 {
103     const int size = vsmc_particle_size(particle);
104     for (int i = 0; i < size; ++i) {
105         vsmc_single_particle sp = vsmc_particle_sp(particle, i);
106         *r++ = sp.state[PosX];
107         *r++ = sp.state[PosY];
108     }
109 }

110 int main(int argc, char **argv)
111 {
112     int N = 10000;
113     if (argc > 1)
114         N = atoi(argv[1]);

115     vsmc_sampler sampler = vsmc_sampler_new(N, 4, vSMCMultinomial, 0.5);
116     vsmc_sampler_init(sampler, pf_init);
117     vsmc_sampler_move(sampler, pf_move, 0);
118     vsmc_sampler_set_monitor(sampler, "pos", 2, pf_eval, 0, vSMCMonitorMCMC);

119     vsmc_stop_watch watch = vsmc_stop_watch_new();
120     vsmc_stop_watch_start(watch);
121     vsmc_sampler_initialize(sampler, (void *) "pf.data");
122     vsmc_sampler_iterate(sampler, pf_obs_x.size - 1);
123     vsmc_stop_watch_stop(watch);
124     printf("Time (ms): %lg\n", vsmc_stop_watch_milliseconds(watch));
125     vsmc_stop_watch_delete(&watch);

126     vsmc_sampler_print_f(sampler, "pf.out", '\t');
```

```
127     vsmc_sampler_delete(&sampler);
128     vsmc_vector_delete(&pf_obs_x);
129     vsmc_vector_delete(&pf_obs_y);
130     vsmc_vector_delete(&pf_pos_x);
131     vsmc_vector_delete(&pf_pos_y);
132     vsmc_vector_delete(&pf_vel_x);
133     vsmc_vector_delete(&pf_vel_y);
134     vsmc_vector_delete(&pf_weight);

135     return 0;
136 }
```

B.1.4 *Parallelized implementation using TBB in C*

```
 1 #include <vsmc/vsmc.h>
 2 #include <math.h>
 3 #include <stdio.h>
 4 #include <stdlib.h>

 5 static const int PosX = 0;
 6 static const int PosY = 1;
 7 static const int VelX = 2;
 8 static const int VelY = 3;

 9 // Storage for data
10 static vsmc_vector pf_obs_x = {NULL, 0};
11 static vsmc_vector pf_obs_y = {NULL, 0};

12 // Temporaries used by pf_init and pf_move
13 static vsmc_vector pf_pos_x = {NULL, 0};
14 static vsmc_vector pf_pos_y = {NULL, 0};
15 static vsmc_vector pf_vel_x = {NULL, 0};
16 static vsmc_vector pf_vel_y = {NULL, 0};
17 static vsmc_vector pf_weight = {NULL, 0};

18 static inline double pf_log_likelihood(int t, const vsmc_single_particle *sp)
19 {
20     double llh_x = 10 * (sp->state[PosX] - pf_obs_x.data[t]);
```

```
21     double llh_y = 10 * (sp->state[PosY] - pf_obs_y.data[t]);
22     llh_x = log(1 + llh_x * llh_x / 10);
23     llh_y = log(1 + llh_y * llh_y / 10);

24     return -0.5 * (10 + 1) * (llh_x + llh_y);
25 }

26 static inline void pf_read_data(const char *param)
27 {
28     if (!param)
29         return;

30     FILE *data = fopen(param, "r");
31     int n = 0;
32     while (1) {
33         double x;
34         double y;
35         int nx = fscanf(data, "%lg", &x);
36         int ny = fscanf(data, "%lg", &y);
37         if (nx == 1 && ny == 1)
38             ++n;
39         else
40             break;
41     }
42     vsmc_vector_resize(&pf_obs_x, n);
43     vsmc_vector_resize(&pf_obs_y, n);
44     fseek(data, 0, SEEK_SET);
45     for (int i = 0; i < n; ++i) {
46         fscanf(data, "%lg", &pf_obs_x.data[i]);
47         fscanf(data, "%lg", &pf_obs_y.data[i]);
48     }
49     fclose(data);
50 }

51 static inline void pf_normal(
52     vsmc_particle particle, double sd_pos, double sd_vel)
53 {
54     vsmc_rng rng = vsmc_particle_rng(particle, 0);
```

```
55      const int size = vsmc_particle_size(particle);
56      vsmc_vector_resize(&pf_pos_x, size);
57      vsmc_vector_resize(&pf_pos_y, size);
58      vsmc_vector_resize(&pf_vel_x, size);
59      vsmc_vector_resize(&pf_vel_y, size);
60      vsmc_vector_resize(&pf_weight, size);
61      vsmc_normal_rand(rng, size, pf_pos_x.data, 0, sd_pos);
62      vsmc_normal_rand(rng, size, pf_pos_y.data, 0, sd_pos);
63      vsmc_normal_rand(rng, size, pf_vel_x.data, 0, sd_vel);
64      vsmc_normal_rand(rng, size, pf_vel_y.data, 0, sd_vel);
65  }

66  static inline void pf_init_param(vsmc_particle particle, void *param)
67  {
68      pf_read_data((const char *) param);
69  }

70  static inline void pf_init_pre(vsmc_particle particle)
71  {
72      pf_normal(particle, 2, 1);
73  }

74  static inline int pf_init_sp(vsmc_single_particle sp)
75  {
76      sp.state[PosX] = pf_pos_x.data[sp.id];
77      sp.state[PosY] = pf_pos_y.data[sp.id];
78      sp.state[VelX] = pf_vel_x.data[sp.id];
79      sp.state[VelY] = pf_vel_y.data[sp.id];
80      pf_weight.data[sp.id] = pf_log_likelihood(0, &sp);

81      return 0;
82  }

83  static inline void pf_init_post(vsmc_particle particle)
84  {
85      vsmc_weight_set_log(vsmc_particle_weight(particle), pf_weight.data, 1);
86  }
```

```
87 static inline void pf_move_pre(int t, vsmc_particle particle)
88 {
89     pf_normal(particle, sqrt(0.02), sqrt(0.001));
90 }

91 static inline int pf_move_sp(int t, vsmc_single_particle sp)
92 {
93     sp.state[PosX] += pf_pos_x.data[sp.id] + 0.1 * sp.state[VelX];
94     sp.state[PosY] += pf_pos_y.data[sp.id] + 0.1 * sp.state[VelY];
95     sp.state[VelX] += pf_vel_x.data[sp.id];
96     sp.state[VelY] += pf_vel_y.data[sp.id];
97     pf_weight.data[sp.id] = pf_log_likelihood(t, &sp);

98     return 0;
99 }

100 static inline void pf_move_post(int t, vsmc_particle particle)
101 {
102     vsmc_weight_add_log(vsmc_particle_weight(particle), pf_weight.data, 1);
103 }

104 static inline void pf_eval_sp(
105     int t, int dim, vsmc_single_particle sp, double *r)
106 {
107     r[0] = sp.state[PosX];
108     r[1] = sp.state[PosY];
109 }

110 int main(int argc, char **argv)
111 {
112     int N = 10000;
113     if (argc > 1)
114         N = atoi(argv[1]);

115     vsmc_sampler sampler = vsmc_sampler_new(N, 4, vSMCMultinomial, 0.5);
116     vsmc_sampler_init_tbb(
117         sampler, pf_init_sp, pf_init_param, pf_init_pre, pf_init_post);
118     vsmc_sampler_move_tbb(sampler, pf_move_sp, pf_move_pre, pf_move_post, 0);
```

```
119     vsmc_sampler_set_monitor_tbb(
120         sampler, "pos", 2, pf_eval_sp, NULL, NULL, 0, vSMCMonitorMCMC);

121     vsmc_stop_watch watch = vsmc_stop_watch_new();
122     vsmc_stop_watch_start(watch);
123     vsmc_sampler_initialize(sampler, (void *) "pf.data");
124     vsmc_sampler_iterate(sampler, pf_obs_x.size - 1);
125     vsmc_stop_watch_stop(watch);
126     printf("Time (ms): %lg\n", vsmc_stop_watch_milliseconds(watch));
127     vsmc_stop_watch_delete(&watch);

128     vsmc_sampler_print_f(sampler, "pf.out", '\t');

129     vsmc_sampler_delete(&sampler);
130     vsmc_vector_delete(&pf_obs_x);
131     vsmc_vector_delete(&pf_obs_y);
132     vsmc_vector_delete(&pf_pos_x);
133     vsmc_vector_delete(&pf_pos_y);
134     vsmc_vector_delete(&pf_vel_x);
135     vsmc_vector_delete(&pf_vel_y);
136     vsmc_vector_delete(&pf_weight);

137     return 0;
138 }
```

B.1.5 *Parallelized implementation using OpenCL*

Host program

```
1 #include <vsmc/vsmc.hpp>

2 static constexpr std::size_t N = 10000; // Number of particles
3 static constexpr std::size_t n = 100;   // Number of data points
4 static constexpr std::size_t PosX = 0;
5 static constexpr std::size_t PosY = 1;
6 static constexpr std::size_t VelX = 2;
7 static constexpr std::size_t VelY = 3;
```

```
 8 static constexpr std::size_t G = 10240;
 9 static constexpr std::size_t L = 256;

10 typedef struct {
11     cl_float pos_x;
12     cl_float pos_y;
13     cl_float vel_x;
14     cl_float vel_y;
15 } cl_pf_sp;

16 using PFStateBase = vsmc::StateMatrix<vsmc::RowMajor, 1, cl_pf_sp>;

17 class PFState : public PFStateBase
18 {
19     public:
20     using size_type = cl_int;
21     using PFStateBase::StateMatrix;

22     void initialize(const vsmc::CLContext &context,
23         const vsmc::CLCommandQueue &command_queue,
24         const vsmc::CLKernel &kernel)
25     {
26         command_queue_ = command_queue;
27         kernel_ = kernel;

28         dev_data_ =
29             vsmc::CLMemory(context, CL_MEM_READ_WRITE | CL_MEM_HOST_READ_ONLY,
30                 sizeof(cl_pf_sp) * size());
31         dev_weight_ =
32             vsmc::CLMemory(context, CL_MEM_WRITE_ONLY | CL_MEM_HOST_READ_ONLY,
33                 sizeof(cl_float) * size());
34         dev_rng_set_ =
35             vsmc::CLMemory(context, CL_MEM_WRITE_ONLY | CL_MEM_HOST_READ_ONLY,
36                 sizeof(vsmc_threefry4x32) * size());
37         dev_index_ =
38             vsmc::CLMemory(context, CL_MEM_READ_ONLY | CL_MEM_HOST_WRITE_ONLY,
39                 sizeof(cl_int) * size());
40         dev_obs_x_ = vsmc::CLMemory(context,
```

```
41              CL_MEM_READ_ONLY | CL_MEM_HOST_WRITE_ONLY, sizeof(cl_float) * n);
42          dev_obs_y_ = vsmc::CLMemory(context,
43              CL_MEM_READ_ONLY | CL_MEM_HOST_WRITE_ONLY, sizeof(cl_float) * n);
44      }

45      void copy(std::size_t N, const cl_int *index)
46      {
47          command_queue_.enqueue_write_buffer(dev_index_, CL_TRUE, 0,
48              sizeof(cl_int) * N, const_cast<cl_int *>(index));

49          kernel_.set_arg(0, static_cast<cl_int>(size()));
50          kernel_.set_arg(1, dev_data_);
51          kernel_.set_arg(2, dev_index_);

52          command_queue_.enqueue_nd_range_kernel(kernel_, 1, vsmc::CLNDRange(),
53              vsmc::CLNDRange(G), vsmc::CLNDRange(L));
54          command_queue_.finish();
55      }

56      void copy_to_host()
57      {
58          command_queue_.enqueue_read_buffer(
59              dev_data_, CL_TRUE, 0, sizeof(cl_pf_sp) * size(), data());
60      }

61      void read_data(const char *param)
62      {
63          if (param == nullptr)
64              return;

65          vsmc::Vector<cl_float> obs_x(n);
66          vsmc::Vector<cl_float> obs_y(n);
67          std::ifstream data(param);
68          for (std::size_t i = 0; i != n; ++i)
69              data >> obs_x[i] >> obs_y[i];
70          data.close();

71          command_queue_.enqueue_write_buffer(
```

```
72              dev_obs_x_, CL_TRUE, 0, sizeof(cl_float) * n, obs_x.data());
73          command_queue_.enqueue_write_buffer(
74              dev_obs_y_, CL_TRUE, 0, sizeof(cl_float) * n, obs_y.data());
75      }

76      const vsmc::CLMemory &dev_data() const { return dev_data_; }
77      const vsmc::CLMemory &dev_weight() const { return dev_weight_; }
78      const vsmc::CLMemory &dev_rng_set() const { return dev_rng_set_; }
79      const vsmc::CLMemory &dev_obs_x() const { return dev_obs_x_; }
80      const vsmc::CLMemory &dev_obs_y() const { return dev_obs_y_; }

81      private:
82      vsmc::CLCommandQueue command_queue_;
83      vsmc::CLKernel kernel_;
84      vsmc::CLMemory dev_data_;
85      vsmc::CLMemory dev_rng_set_;
86      vsmc::CLMemory dev_weight_;
87      vsmc::CLMemory dev_index_;
88      vsmc::CLMemory dev_obs_x_;
89      vsmc::CLMemory dev_obs_y_;
90  };

91  class PFInit
92  {
93      public:
94      PFInit(const vsmc::CLCommandQueue &command_queue,
95          const vsmc::CLKernel &kernel)
96          : command_queue_(command_queue), kernel_(kernel)
97      {
98      }

99      std::size_t operator()(vsmc::Particle<PFState> &particle, void *param)
100     {
101         particle.value().read_data(static_cast<const char *>(param));

102         kernel_.set_arg(0, static_cast<cl_int>(particle.size()));
103         kernel_.set_arg(1, particle.value().dev_data());
104         kernel_.set_arg(2, particle.value().dev_rng_set());
```

```
105         kernel_.set_arg(3, particle.value().dev_weight());
106         kernel_.set_arg(4, particle.value().dev_obs_x());
107         kernel_.set_arg(5, particle.value().dev_obs_y());

108         command_queue_.enqueue_nd_range_kernel(kernel_, 1, vsmc::CLNDRange(),
109             vsmc::CLNDRange(G), vsmc::CLNDRange(L));
110         command_queue_.finish();

111         weight_.resize(particle.size());
112         command_queue_.enqueue_read_buffer(particle.value().dev_weight(),
113             CL_TRUE, 0, sizeof(cl_float) * particle.size(), weight_.data());
114         particle.weight().set_log(weight_.data());

115         return 0;
116     }

117     private:
118     vsmc::CLCommandQueue command_queue_;
119     vsmc::CLKernel kernel_;
120     vsmc::Vector<cl_float> weight_;
121 };

122 class PFMove
123 {
124     public:
125     PFMove(const vsmc::CLCommandQueue &command_queue,
126         const vsmc::CLKernel &kernel)
127         : command_queue_(command_queue), kernel_(kernel)
128     {
129     }

130     std::size_t operator()(std::size_t t, vsmc::Particle<PFState> &particle)
131     {
132         kernel_.set_arg(0, static_cast<cl_int>(t));
133         kernel_.set_arg(1, static_cast<cl_int>(particle.size()));
134         kernel_.set_arg(2, particle.value().dev_data());
135         kernel_.set_arg(3, particle.value().dev_rng_set());
136         kernel_.set_arg(4, particle.value().dev_weight());
```

```
137         kernel_.set_arg(5, particle.value().dev_obs_x());
138         kernel_.set_arg(6, particle.value().dev_obs_y());

139         command_queue_.enqueue_nd_range_kernel(kernel_, 1, vsmc::CLNDRange(),
140             vsmc::CLNDRange(G), vsmc::CLNDRange(L));
141         command_queue_.finish();

142         weight_.resize(particle.size());
143         command_queue_.enqueue_read_buffer(particle.value().dev_weight(),
144             CL_TRUE, 0, sizeof(cl_float) * particle.size(), weight_.data());
145         particle.weight().add_log(weight_.data());

146         return 0;
147     }

148     private:
149     vsmc::CLCommandQueue command_queue_;
150     vsmc::CLKernel kernel_;
151     vsmc::Vector<cl_float> weight_;
152 };

153 class PFEval : public vsmc::MonitorEvalTBB<PFState, PFEval>
154 {
155     public:
156     void eval_pre(std::size_t t, vsmc::Particle<PFState> &particle)
157     {
158         particle.value().copy_to_host();
159     }

160     void eval_sp(std::size_t t, std::size_t dim,
161         vsmc::SingleParticle<PFState> sp, double *r)
162     {
163         r[0] = sp.state(0).pos_x;
164         r[1] = sp.state(0).pos_y;
165     }
166 };

167 int main()
```

```
168 {
169     auto platform = vsmc::cl_get_platform().front();
170     std::string platform_name;
171     platform.get_info(CL_PLATFORM_NAME, platform_name);
172     std::cout << "Platform: " << platform_name << std::endl;
173     auto device = platform.get_device(CL_DEVICE_TYPE_DEFAULT).front();
174     std::string device_name;
175     device.get_info(CL_DEVICE_NAME, device_name);
176     std::cout << "Device:   " << device_name << std::endl;
177     vsmc::CLContext context(vsmc::CLContextProperties(platform), 1, &device);
178     vsmc::CLCommandQueue command_queue(context, device);
179     std::ifstream source_cl("pf_ocl.cl");
180     std::string source((std::istreambuf_iterator<char>(source_cl)),
181         std::istreambuf_iterator<char>());
182     source_cl.close();
183     vsmc::CLProgram program(context, 1, &source);
184     program.build(1, &device, "-I ../../include");
185     vsmc::CLKernel kernel_copy(program, "copy");
186     vsmc::CLKernel kernel_init(program, "init");
187     vsmc::CLKernel kernel_move(program, "move");
188     std::size_t pwgsm_copy = 0;
189     std::size_t pwgsm_init = 0;
190     std::size_t pwgsm_move = 0;
191     kernel_copy.get_work_group_info(
192         device, CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE, pwgsm_copy);
193     kernel_init.get_work_group_info(
194         device, CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE, pwgsm_init);
195     kernel_move.get_work_group_info(
196         device, CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE, pwgsm_move);
197     std::cout << "Kernel copy preferred work group size multiple: "
198               << pwgsm_copy << std::endl;
```

```
    std::cout << "Kernel init preferred work group size multiple: "
              << pwgsm_init << std::endl;
    std::cout << "Kernel move preferred work group size multiple: "
              << pwgsm_move << std::endl;

    vsmc::Sampler<PFState> sampler(N, vsmc::Multinomial, 0.5);
    sampler.particle().value().initialize(context, command_queue, kernel_copy);
    sampler.init(PFInit(command_queue, kernel_init));
    sampler.move(PFMove(command_queue, kernel_move), false);
    sampler.monitor("pos", 2, PFEval());

    vsmc::StopWatch watch;
    watch.start();
    sampler.initialize(const_cast<char *>("pf.data")).iterate(n - 1);
    watch.stop();
    std::cout << "Time (ms): " << watch.milliseconds() << std::endl;

    std::ofstream output("pf.out");
    output << sampler;
    output.close();

    return 0;
}
```

Device program

```
#include <vsmc/rngc/rngc.h>

typedef struct {
    float pos_x;
    float pos_y;
    float vel_x;
    float vel_y;
} pf_sp;

static inline float log_likelihood(const pf_sp *sp, float obs_x, float obs_y)
{
    float llh_x = 10 * (sp->pos_x - obs_x);
```

```
    float llh_y = 10 * (sp->pos_y - obs_y);
    llh_x = log(1 + llh_x * llh_x / 10);
    llh_y = log(1 + llh_y * llh_y / 10);

    return -0.5f * (10 + 1) * (llh_x + llh_y);
}

static inline void rnorm(vsmc_threefry4x32 *rng, float *r)
{
    uint32_t u32[4];
    u32[0] = vsmc_threefry4x32_rand(rng);
    u32[1] = vsmc_threefry4x32_rand(rng);
    u32[2] = vsmc_threefry4x32_rand(rng);
    u32[3] = vsmc_threefry4x32_rand(rng);

    float u01[4];
    u01[0] = vsmc_u01_co_u32f(u32[0]);
    u01[1] = vsmc_u01_co_u32f(u32[1]);
    u01[2] = vsmc_u01_co_u32f(u32[2]);
    u01[3] = vsmc_u01_co_u32f(u32[3]);

    u01[0] = sqrt(-2 * log(u01[0]));
    u01[1] = sqrt(-2 * log(u01[1]));
    u01[2] *= 2;
    u01[3] *= 2;

    r[0] = u01[0] * sinpi(u01[2]);
    r[1] = u01[0] * cospi(u01[2]);
    r[2] = u01[1] * sinpi(u01[3]);
    r[3] = u01[1] * cospi(u01[3]);
}

static inline float init_sp(
    pf_sp *sp, vsmc_threefry4x32 *rng, float obs_x, float obs_y)
{
    vsmc_threefry4x32_init(rng, get_global_id(0));

    float r[4];
```

```
    rnorm(rng, r);
    const float sd_pos = 2.0f;
    const float sd_vel = 1.0f;
    sp->pos_x = r[0] * sd_pos;
    sp->pos_y = r[1] * sd_pos;
    sp->vel_x = r[2] * sd_vel;
    sp->vel_y = r[3] * sd_vel;

    return log_likelihood(sp, obs_x, obs_y);
}

static inline float move_sp(
    pf_sp *sp, vsmc_threefry4x32 *rng, float obs_x, float obs_y)
{
    float r[4];
    rnorm(rng, r);
    const float sd_pos = sqrt(0.02f);
    const float sd_vel = sqrt(0.001f);
    sp->pos_x += r[0] * sd_pos + 0.1f * sp->vel_x;
    sp->pos_y += r[1] * sd_pos + 0.1f * sp->vel_y;
    sp->vel_x += r[2] * sd_vel;
    sp->vel_y += r[3] * sd_vel;

    return log_likelihood(sp, obs_x, obs_y);
}

__kernel void copy(int N, __global pf_sp *state, const __global int *index)
{
    int i = get_global_id(0);
    if (i >= N)
        return;

    state[i] = state[index[i]];
}

__kernel void init(int N, __global pf_sp *state,
    __global vsmc_threefry4x32 *rng_set, __global float *weight,
    const __global float *obs_x, const __global float *obs_y)
```

```
74 {
75     int i = get_global_id(0);
76     if (i >= N)
77         return;

78     pf_sp sp = state[i];
79     vsmc_threefry4x32 rng = rng_set[i];
80     weight[i] = init_sp(&sp, &rng, obs_x[0], obs_y[0]);
81     state[i] = sp;
82     rng_set[i] = rng;
83 }

84 __kernel void move(int t, int N, __global pf_sp *state,
85     __global vsmc_threefry4x32 *rng_set, __global float *weight,
86     const __global float *obs_x, const __global float *obs_y)
87 {
88     int i = get_global_id(0);
89     if (i >= N)
90         return;

91     pf_sp sp = state[i];
92     vsmc_threefry4x32 rng = rng_set[i];
93     weight[i] = move_sp(&sp, &rng, obs_x[t], obs_y[t]);
94     state[i] = sp;
95     rng_set[i] = rng;
96 }
```

B.2 PROCESS COMMAND LINE PROGRAM OPTIONS

```
1 #include <vsmc/vsmc.hpp>

2 int main(int argc, char **argv)
3 {
4     int n;
5     std::string str;
6     std::vector<double> vec;
```

```
    vsmc::ProgramOptionMap option_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);
    option_map.process(argc, argv);

    std::cout << "n: " << n << std::endl;
    std::cout << "str: " << str << std::endl;
    std::cout << "vec: ";
    for (auto v : vec)
        std::cout << v << ' ';
    std::cout << std::endl;

    return 0;
}
```

B.3 DISPLAY PROGRAM PROGRESS

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

int main()
{
    vsmc::RNG rng;
    vsmc::FisherFDistribution<double> dist(10, 20);
    std::size_t n = 1000;
    double r = 0;
    vsmc::Progress progress;
    progress.start(n * n);
    for (std::size_t i = 0; i != n; ++i) {
        std::stringstream ss;
        ss << "i = " << i;
        progress.message(ss.str());
        for (std::size_t j = 0; j != n; ++j) {
            for (std::size_t k = 0; k != n; ++k)
                r += dist(rng);
            progress.increment();
```

```
18          }
19      }
20      progress.stop();

21      return 0;
22  }
```