Yan Zhou

vSMC – Parallel SMC in C++

Second edition (version 2.2.0)

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 OTHER SEQUENTIAL MONTE CARLO 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` to nonzero values before including any vSMC headers to make their existence known to the library, respectively.

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.

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.

The `StateMatrix` class deliberately does not provide a `resize` method. There are algorithms that change the sample size between iterations. However, such algorithms often change size through resampling or other methods, either deterministically or stochastically. An example of changing size of a sampler is provided in section 6.1.

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, and `particle.weight()` returns the type `Weight` object[5]. They are constructed with the same integer value N when the above constructor is invoked.

As a 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;
    std::normal_distribution<double> rnorm(0, 1);
    x = rnorm(rng);
}
```

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,

```
std::mt19937 rng;
void function(double &x)
{
    std::normal_distribution<double> rnorm(0, 1);
    x = rnorm(rng);
}
```

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

[5] More precisely, it is a `Particle<T>::weight_type` object, whose exact type 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 the class `Weight`

```
std::mt19937 rng1(s1); // For thread i_1 with seed s_1
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,

```
auto &rng1 = particle.rng(i1); // Called from thread i_1
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,

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

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.

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 template `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 subclass 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) {
    // sp is of type SingleParticle<T>
}
```

The above 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. A sampler can be initialized by user defined object that is 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
    // Initialize the particle system
};
```

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 function `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 callable objects that is convertible to the following types,

```
using move_type = std::function<std::size_t(std::size_t, Particle<T> &)>;
using mcmc_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 callable object that is 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 its dimension, and the third 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. 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,

```
1 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,

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

Details can be found in section 8.3.

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

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

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.

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 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.

2.4.2 *Implementations*

The complete program is shown in appendix A.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, PFMEval());
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 `PFMEval` that records X_{pos}^t and Y_{pos}^t 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[6] script that can be used to process the output

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

```
library(ggplot2)

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

obs <- read.table("pf.data", header = FALSE)
dat <- data.frame(
X = c(pf[["pos.0"]], obs[,1]),
Y = c(pf[["pos.1"]], obs[,2]))
dat[["Source"]] <- rep(c("Estimate", "Observation"), each = dim(obs)[1])
plt <- qplot(x = X, y = Y, data = dat, geom = "path")
plt <- plt + aes(group = Source, color = Source, linetype = Source)
plt <- plt + theme_bw() + theme(legend.position = "top")
pdf("pf.pdf")
print(plt)
dev.off()
```

The `print` 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 library does not provide direct support of changing the sample size. However, it is possible and an example is shown in section 6.1. 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. The last two columns show the importance sampling estimates of the positions recorded by the monitor named `"pos"`. The 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 constant 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>;
```

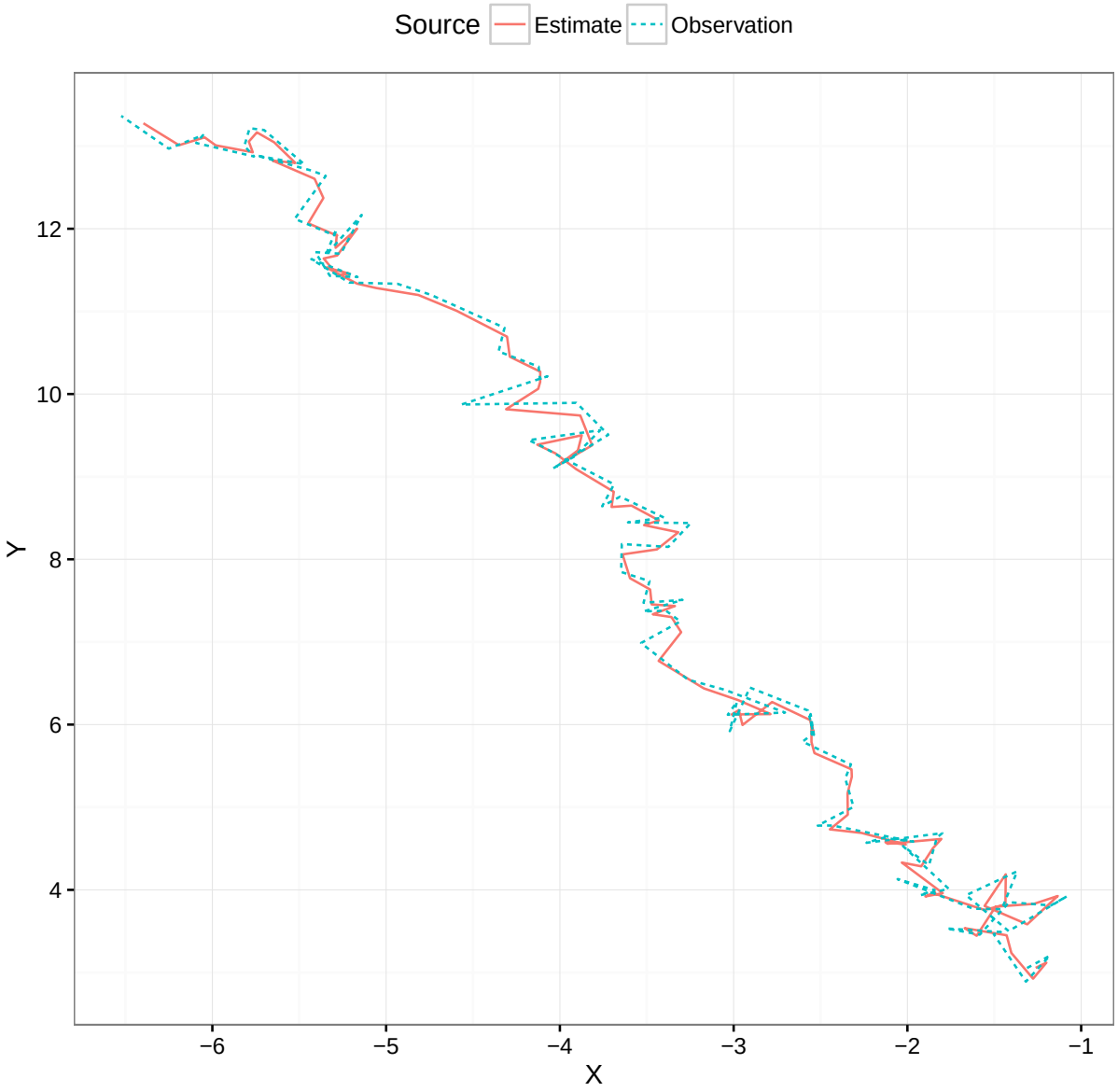

Figure 2.1 A simple particle filter

The numbers of particles and data points are also defined as constants in this simple example,

```
static constexpr std::size_t N = 1000; // Number of particles
static constexpr std::size_t n = 100;  // Number of data points
```

Last, 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;
```

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
{
```

```
3     public:
4     std::size_t operator()(Particle<PFState> &particle, void *param)
5     {
6         eval_param(particle, param);
7         eval_pre(particle);
8         std::size_t acc = 0;
9         for (auto sp : particle)
10            acc += eval_sp(sp);
11        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        w_.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        w_[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(w_.data());
36      }

37      private:
38      Vector<double> w_;
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 `w_` 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: `PFMEval`

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

```
1  class PFMEval
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 `PFMEval` 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 subclasses as below,

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

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[7] (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`.

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

3.1 CLONING OBJECTS

The `Sampler<T>` and `Particle<T>` objects have copy constructors, assignment operators, move constructors, and move assignment operators that behaves 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 to generating exactly the same random numbers in the next iteration. This is an undesired side effects. 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 its own RNG system 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.

A note on `weight_type`, it needs to provide the following method,

```
w.ess();           // Get ESS
w.set_equal();     // Set W^(i) = 1/N
w.resample_size(); // Get the 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 each computing node has been allocated N_r particles.

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

        return 1 / global;
    }

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

    double *resample_data()
    {
        if (rank == 0) {
            // Gather all normalized weights into a member data on this node
            // Say resample_weight_
            return resample_weight_.data();
```

```
18          } else {
19              return nullptr;
20          }
21      }

22      void set_equal()
23      {
24          // Set all weights to 1/∑N_r
25          // Synchronization
26      }

27      void set(const double *v)
28      {
29          // Set W^(i) = v_i for i = 1, ..., N_r
30          // Compute S_r = ∑_{i=1}^{N_r} W^(i)
31          // Gathering S_r, compute S = ∑S_r
32          // Broadcast S
33          // Set W^(i) = W^(i)/S for i = 1, ..., N_r
34      }
35  };
```

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. And then a pointer to this vector 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 see in section 2.3.1 that if class `T` is a subclass 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>;
```

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

    template <typename S>
    class single_particle_type : public PFStateSPBase<S>
    {
        public:
        using PFStateSPBase<S>::single_particle_type;

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

        // Return $\ell(X_t^{(i)}|Y_t)$
        double log_likelihood(std::size_t t);
    };

    void read_data(const char *param);

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

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);
```

```
9         NormalDistribution<double> norm_vel(0, 1);
10        sp.pos_x() = norm_pos(sp.rng());
11        sp.pos_y() = norm_pos(sp.rng());
12        sp.vel_x() = norm_vel(sp.rng());
13        sp.vel_y() = norm_vel(sp.rng());
14        w_[sp.id()] = sp.log_likelihood(0);

15        return 0;
16    }

17    void eval_post(Particle<PFState> &particle);

18    private:
19    Vector<double> w_;
20 };
```

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,

```
1  class PFStateSP
2  {
3      public:
4      double &pos_x() { return pos_x_; }
5      double &pos_y() { return pos_y_; }
6      double &vel_x() { return vel_x_; }
7      double &vel_y() { return vel_y_; }

8      double log_likelihood(double obs_x, double obs_y) const;

9      private:
10     double pos_x_;
11     double pos_y_;
12     double vel_x_;
13     double vel_y_;
14 };
```

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_; }
    double &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 PFMEval 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 PFMEval as the following,

```
class PFMEval
{
    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);
```

```
10     }
11 };
```

And it can be added to a sampler as,

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

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 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. Most of the macros have the prefix `VSMC_HAS_`, which specify if a feature is available to the library. A few have the prefix `VSMC_USE_`, which specify if a feature shall be used in the case that it is available. A handful remaining ones define some constants or types used by the library. These macros are listed in table 4.1.

Macro	Default	Description
`VSMC_INT64`	Platform dependent	The 64-bits integer type used by x86 intrinsics functions
`VSMC_HAS_INT128`	Platform dependent	Support for 128-bits integers
`VSMC_INT128`	Platform dependent	The 128-bits integer type
`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	If we are using x86
`VSMC_HAS_X86_64`	Platform dependent	If we are using x86-64
`VSMC_HAS_POSIX`	Platform dependent	If we are on a POSIX platform
`VSMC_HAS_OMP`	Platform dependent	Support for OpenMP 3.0 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 or higher
`VSMC_USE_MKL_CBLAS`	`VSMC_HAS_MKL`	Use MKL header `mkl_cblas.h` instead of the standard `cblas.h`
`VSMC_USE_MKL_LAPACKE`	`VSMC_HAS_MKL`	Use MKL header `mkl_lapacke.h` instead of the standard `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_CBLAS_INT_TYPE`	`int`	The default integer type of BLAS routines
`VSMC_ALIGNMENT`	`32`	Default alignment of `AlignedAllocator`

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. Note that all functions has a prefix `const_`, which is omitted in the table.

5.2 VECTORIZED OPERATIONS

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

```
std::size_t n = 1000;
vsmc::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.

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. The order of the input parameters are as they appear in the mathematical expressions. For example, `fma` perform the operation $ab + c$, then the function shall be called as,

```
fma(n, a, b, c, y);
```

where `a`, `b`, `c` are input parameters, and some of them, not all, can be scalars instead of pointers. And `y` is the output parameter, which has to be pointer to a length n vector.

Function	Value	Function	Value	Function	Value
`pi`	π	`pi_2`	2π	`pi_inv`	$1/\pi$
`pi_sqr`	π^2	`pi_by2`	$\pi/2$	`pi_by3`	$\pi/3$
`pi_by4`	$\pi/4$	`pi_by6`	$\pi/6$	`pi_2by3`	$2\pi/3$
`pi_3by4`	$3\pi/4$	`pi_4by3`	$4\pi/3$	`sqrt_pi`	$\sqrt{\pi}$
`sqrt_pi_2`	$\sqrt{2\pi}$	`sqrt_pi_inv`	$\sqrt{1/\pi}$	`sqrt_pi_by2`	$\sqrt{\pi/2}$
`sqrt_pi_by3`	$\sqrt{\pi/3}$	`sqrt_pi_by4`	$\sqrt{\pi/4}$	`sqrt_pi_by6`	$\sqrt{\pi/6}$
`sqrt_pi_2by3`	$\sqrt{2\pi/3}$	`sqrt_pi_3by4`	$\sqrt{3\pi/4}$	`sqrt_pi_4by3`	$\sqrt{4\pi/3}$
`ln_pi`	$\ln \pi$	`ln_pi_2`	$\ln 2\pi$	`ln_pi_inv`	$\ln 1/\pi$
`ln_pi_by2`	$\ln \pi/2$	`ln_pi_by3`	$\ln \pi/3$	`ln_pi_by4`	$\ln \pi/4$
`ln_pi_by6`	$\ln \pi/6$	`ln_pi_2by3`	$\ln 2\pi/3$	`ln_pi_3by4`	$\ln 3\pi/4$
`ln_pi_4by3`	$\ln 4\pi/3$	`e`	e	`e_inv`	$1/\mathrm{e}$
`sqrt_e`	$\sqrt{\mathrm{e}}$	`sqrt_e_inv`	$\sqrt{1/\mathrm{e}}$	`sqrt_2`	$\sqrt{2}$
`sqrt_3`	$\sqrt{3}$	`sqrt_5`	$\sqrt{5}$	`sqrt_10`	$\sqrt{10}$
`sqrt_1by2`	$\sqrt{1/2}$	`sqrt_1by3`	$\sqrt{1/3}$	`sqrt_1by5`	$\sqrt{1/5}$
`sqrt_1by10`	$\sqrt{1/10}$	`ln_2`	$\ln 2$	`ln_3`	$\ln 3$
`ln_5`	$\ln 5$	`ln_10`	$\ln 10$	`ln_inv_2`	$1/\ln 2$
`ln_inv_3`	$1/\ln 3$	`ln_inv_5`	$1/\ln 5$	`ln_inv_10`	$1/\ln 10$
`ln_ln_2`	$\ln \ln 2$				

Table 5.1 Mathematical constants. Note: All functions are prefixed by `const_`.

Function	Operation	Function	Operation	Function	Operation
`add`	$a + b$	`sub`	$a - b$	`sqr`	a^2
`mul`	ab	`abs`	$\lvert a \rvert$	`fma`	$ab + c$
`inv`	$1/a$	`div`	a/b	`sqrt`	$\sqrt{a}$
`invsqrt`	$1/\sqrt{a}$	`cbrt`	$\sqrt[3]{a}$	`invcbrt`	$1/\sqrt[3]{a}$
`pow2o3`	$a^{2/3}$	`pow3o2`	$a^{3/2}$	`pow`	a^b
`hypot`	$\sqrt{a^2 + b^2}$	`exp`	e^a	`exp2`	2^a
`exp10`	10^a	`expm1`	$\mathrm{e}^a - 1$	`log`	$\ln(a)$
`log2`	$\log_2(a)$	`log10`	$\log_{10}(a)$	`log1p`	$\ln(a + 1)$
`cos`	$\cos(a)$	`sin`	$\sin(a)$	`sincos`	$\sin(a)$ and $\cos(a)$
`tan`	$\tan(a)$	`acos`	$\arccos(a)$	`asin`	$\arcsin(a)$
`atan`	$\arctan(a)$	`acos`	$\arccos(a)$	`atan2`	$\arctan(a/b)$
`cosh`	$\cosh(a)$	`sinh`	$\sinh(a)$	`tanh`	$\tanh(a)$
`acosh`	$\operatorname{arc}\cosh(a)$	`asinh`	$\operatorname{arc}\sinh(a)$	`atanh`	$\operatorname{arc}\tanh(a)$
`erf`	$\operatorname{erf}(a)$	`erfc`	$\operatorname{erfc}(a)$	`cdfnorm`	$1 - \operatorname{erfc}(a/\sqrt{2})/2$
`lgamma`	$\ln \Gamma(a)$	`tgamma`	$\Gamma(a)$		

Table 5.2 Vectorized mathematical operations

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 other statistical properties, in practice, such an algorithm can be decomposed into three steps. First, a vector $\{r_i\}_{i=1}^N$ is generated such that $\sum_{i=1}^N r_i = M$. Then a vector $\{a_i\}_{i=1}^M$ is generated such that, $\sum_{i=1}^M \mathbb{I}_{\{j\}}(a_i) = r_j$. 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. Below is the signature,

```
template <typename IntType, typename RNGType>
void resample(std::size_t M, std::size_t N, RNGType &rng,
const double *weight, IntType *replication);
```

The last parameter is the output vector $\{r_i\}_{i=1}^N$. The builtin schemes are implemented as classes with `operator()` conforms to the above signature. For example, `ResampleMultinomial` implements the multinomial resampling algorithm.

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

```
template <typename IntType1, typename IntType2>
void resample_trans_rep_index(std::size_t M, std::size_t N,
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 $r_i > 0$. 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.

6.1 RESIZING A SAMPLER

Now, we provide an example of changing sampler size,

```
// sampler is an existing Sampler<T> object
auto N = sampler.size();
```

```
auto &rng = sampler.particle().rng();
auto weight = sampler.particle().weight().data();
Vector<std::size_t> rep(N);
Vector<std::size_t> idx(M);
ResampleMultinomial resample;
resample(M, N, rng, weight, rep.data());
resample_trans_rep_index(M, N, rep.data(), idx.data());
Particle<T> particle(M);
for (std::size_t i = 0; i != M; ++i) {
    auto sp_dst = particle.sp(i);
    auto sp_src = sampler.partice().sp(idx[i]);
    // Assuming T is a subclass of StateMatrix
    for (std::size_t d = 0; d != sp_dst.dim(); ++d)
    sp_dst.state(d) = sp_src.state(d);
}
// Copy other data of class T if any
sampler.particle() = std::move(particle);
```

The library has a comprehensive RNG system to facilitate implementation of Monte Carlo algorithms.

7.1 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 an 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())
    seed_txt >> Seed::instance(); // Read seed from a file
else
    Seed::instance().set(101);    // The default seed
seed_txt.close();
// The program
std::ofstream seed_txt("seed.txt");
seed_txt << Seed::instance();     // Write the seed to a file
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 the `modulo` method of `SeedGenerator`. For example,

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

where n is the number of processes and r is the rank of the current node. After this call, all seeds generated will belong to the equivalent class $s \equiv r \mod n$. Therefore, no two nodes will ever generate the same seeds.

7.2 COUNTER-BASED RNG

The standard library provides a set of RNG classes. 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 than it was before. In the author's personal opinion, it is the most significant development for parallel Monte Carlo algorithms in recent memory. See the paper for more details. Here, it is sufficient to mention that, the RNG introduced in the paper use a deterministic function f_k, such that, for any sequence $\{c_i\}_{i>0}$, the sequence $\{y_i\}_{i>0}$, $y_i = f_k(c_i)$, 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 $y_i = f(y_{i-1})$, these counter-based RNGs are much easier to setup in a parallelized environment.

If c, the counter, is an unsigned integer with b bits, and k, the key, is an unsigned integer with d bits. Then for each k, the RNG has a period 2^b. And there can be at most 2^d independent streams. Table 7.1 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 concept. All RNGs in Salmon et al. (2011) are implemented along with a few additions. Note that, the actual period of an RNG can be longer. For example, `Philox4x64` has a 256-bits counter but output 64-bits integers. And thus it has a 2^{1024} period. Such period length may seems very small compared to many well known RNGs. For example, the famous Mersenne-Twister generator (`std::mt19937`) has a period $2^{19937} - 1$. However, combined with 2^{256} independent streams, only the most demanding programs will find these counter-base RNGs insufficient.

7.2.1 *AES-NI intrinsics based RNG*

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

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

		Bits	
Class	Result type	Counter	Key
AES128_*B*x32	std::uint32_t	128	128
AES128_*B*x64	std::uint64_t	128	128
AES192_*B*x32	std::uint32_t	128	192
AES192_*B*x64	std::uint64_t	128	192
AES256_*B*x32	std::uint32_t	128	256
AES256_*B*x64	std::uint64_t	128	256
ARS_*B*x32	std::uint32_t	128	128
ARS_*B*x64	std::uint64_t	128	128
Philox2x32*V*	std::uint32_t	64	64
Philox2x64*V*	std::uint64_t	128	128
Philox4x32*V*	std::uint32_t	128	128
Philox4x64*V*	std::uint64_t	256	256
Threefry2x32*V*	std::uint32_t	64	64
Threefry2x64*V*	std::uint64_t	128	128
Threefry4x32*V*	std::uint32_t	128	128
Threefry4x64*V*	std::uint64_t	256	256

Table 7.1 Counter-based RNG; *B*: either 1, 2, 4, or 8; *V*: either empty, SSE2, or AVX2.

time. This is specified by the template parameter `Blocks`. Larger blocks, up to eight, can improve runtime performance but this is at the cost of larger state size.

Four types of key sequences are implemented by the library, corresponding to the ARS algorithm in Salmon et al. (2011) and the AES-128, AES-192, and AES-256 algorithms. The following RNG engines are defined.

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

template <typename ResultType, std::size_t Blocks = VSMC_RNG_AES_BLOCKS>
using AES128Engine =
    AESNIEngine<ResultType, AES128KeySeq<ResultType, 10>, 10, Blocks>;

template <typename ResultType, std::size_t Blocks = VSMC_RNG_AES_BLOCKS>
```

Macro	Default
VSMC_RNG_AES_BLOCKS	4
VSMC_RNG_ARS_ROUNDS	5
VSMC_RNG_ARS_BLOCKS	4
VSMC_RNG_PHILOX_ROUNDS	10
VSMC_RNG_PHILOX_VECTOR_LENGTH	4
VSMC_RNG_THREEFRY_ROUNDS	20
VSMC_RNG_THREEFRY_VECTOR_LENGTH	4

Table 7.2 Configuration macros for the counter-based RNG

```
using AES192Engine =
    AESNIEngine<ResultType, AES192KeySeq<ResultType, 12>, 12, Blocks>;

template <typename ResultType, std::size_t Blocks = VSMC_RNG_AES_BLOCKS>
using AES256Engine =
    AESNIEngine<ResultType, AES256KeySeq<ResultType, 14>, 14, Blocks>;
```

The default template arguments can be changed by configuration macros listed in table 7.2. Shortcuts are also defined, as listed in table 7.1. For example, `ARS_4x32` is `ARSEngine` with result type `std::uint32_t`, four blocks, and the default number of rounds.

7.2.2 *Philox*

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

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

The default vector length and the number of rounds can be changed by configuration macros listed in table **??**. Shortcuts are also defined, as listed in table 7.1. For example, `Philox4x32` is `PhiloxEngine` with result type `std::uint32_t`, vector length four, and the default number of rounds.

7.2.3 *Threefry*

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

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

The default vector length and the number of rounds can be changed by configuration macros listed in table **??**. Shortcuts are also defined, as listed in table 7.1. For example, `Threefry4x32` is `ThreefryEngine` with result type `std::uint32_t`, vector length four, and the default number of rounds.

If SSE2 intrinsics are supported, then a more optimized version is also implemented. This implementation can have higher performance at the cost of larger state size. If AVX2 intrinsics are supported, then a even more optimized, with even larger state size version is also implemented.

```
template <typename ResultType, std::size_t K = VSMC_RNG_THREEFRY_VECTOR_LENGTH,
    std::size_t Rounds = VSMC_RNG_THREEFRY_ROUNDS>
using ThreefryEngineSSE2 =
    CounterEngine<ThreefryGeneratorSSE2<ResultType, K, Rounds>>;

template <typename ResultType, std::size_t K = VSMC_RNG_THREEFRY_VECTOR_LENGTH,
    std::size_t Rounds = VSMC_RNG_THREEFRY_ROUNDS>
using ThreefryEngineAVX2 =
    CounterEngine<ThreefryGeneratorAVX2<ResultType, K, Rounds>>;
```

7.2.4 *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 7.1. More specifically, if the AES-NI intrinsics are supported,

```
using RNG = ARS_4x32;
```

otherwise if AVX2 intrinsics are supported,

```
using RNG = Threefry4x32AVX2;
```

otherwise if SSE2 intrinsics are supported,

```
using RNG = Threefry4x32SSE2;
```

and last, on all other platforms,

```
using RNG = Threefry4x32;
```

This can be changed by the configuration macro `VSMC_RNG_TYPE`.

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 7.3 MKL RNG. Note: all classes can have a suffix _64.

7.3 NON-DETERMINISTIC RNG

If the RDRAND intrinsics are supported, the library also implement three RNGs, `RDRAND16`, `RDRAND32` and `RDRAND64`. They output 16-, 32-, and 64-bits random unsigned integers, respectively.

7.4 MKL RNG

The MKL library provides some high performance RNGs. The library implement a wrapper class `MKLEngine` that makes them accessible as C++11 generators. They are listed in table 7.3. 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.

7.5 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 $i \neq j$.

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

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

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

```
class T
{
    using rng_set_type = /* User defined type */;
};
```

will make the RNG set used by `Particle<T>` replaced by the user defined type.

7.6 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 an drop-in replacement for `std::normal_distribuiton`. This includes all of the continuous distributions defined in the standard library. Their benefits compared to the standard library will be discussed later. Table 7.4 lists all the additional distributions implemented.

The last, the library also implement the multivariate Normal distribution. Its usable is summarized by the following.

```
double mean[2] = { /* the mean vector */ };
double cov[4] = { /* the covariance matrix */ };
double chol[3];
double r[2];
```

Class	Notes
`UniformBits`	No parameters, uniform on the set $\{0, \ldots, 2^b - 1\}$, where b is the number of bits of the result type, which has to be an unsigned integer type.
`U01`	No parameters, uniform on $[0, 1)$
`U01CC`	No parameters, uniform on $[0, 1]$
`U01CO`	No parameters, uniform on $[0, 1)$
`U01OC`	No parameters, uniform on $(0, 1]$
`U01OO`	No parameters, uniform on $(0, 1)$
`Laplace`	Parameters: location `a`; scale `b`
`Levy`	Parameters: location `a`; scale `b`
`Pareto`	Parameters: shape `a`; scale `b`
`Rayleigh`	Parameters: scale `sigma`

Table 7.4 Random number distributions. Note: all class names have a suffix `Distribution` which is omitted in the table

```
// Compute the lower triangular of the Cholesky decomposition
cov_chol(2, cov, chol);
RNG rng;
NormalMVDistribution<double, 2> norm2(mean, chol); // Bivariate Normal
NormalMVDistribution<double, Dynamic> normd(2, mean, chol); // Same as above
norm2(rng, r); // Generate a bivariate Normal
normd(rng, r); // Same as above
```

We shall mention here that the static form, where the dimension is specified as a template parameter is more efficient.

7.7 VECTORIZED RANDOM NUMBER GENERATING

The RNGs and distributions implemented by this library provides vectorized operations. For example,

```
std::size_t n = 1000;
RNG rng;
NormalDistribution<double> norm(0, 1);
Vector<RNG::result_type> u(n);
Vector<double> r(n);
rng(n, u.data());           // Generate n random unsigned integers
rng_rand(rng, n, u.data()); // Same as above
```

```
 8 norm(rng, n, r.data());     // Generate n Normal random numbers
 9 normal_distribution(rng, n, r.data(), 0.0, 1.0);     // Same as above
10 normal_distribution(rng, n, r.data(), norm.param()); // Same as above
11 rng_rand(rng, norm, n, r.data());                    // Same as above
```

Note that these functions will be specialized to use MKL routines if `rng` is one of the engines listed in table 7.3.

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 desired to allocate memory aligned by a certain boundary. The library provides the class template,

```
template <typename T, std::size_t Alignment = VSMC_ALIGNMENT,
    typename Memory = AlignedMemory>
class AlignedAllocator;
```

where the configuration macro `VSMC_ALIGNMENT` is defined to be `32` by default. For the requirement of the parameter type `Memory`, see the reference manual. It is sufficient to mention here that the default implementation works best if TBB is available. This class can be used as a drop-in replacement of `std::allocator<T>`. In fact, this library defines a type alias `Vector<T>` which is `std::vector<T, AlignedAllocator<T>>` if `T` is a scalar type, and `std::vector<T>` otherwise.

8.2 SAMPLE COVARIANCE ESTIMATING

The library provides some basic functionality to estimate sample variance. 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. The method has three additional optional parameters. The first is `cov_layout`, which specifies the covariance matrix storage layout. The second is `cov_upper` and the third `cov_packed`, both are `false` by default. If the later is `cov_packed`, a packed vector of length $d(d+1)/2$ is written into `cov`. If `cov_upper` is `false`, 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 scales. The `NormalMVDistribution` in section 7.6 accepts the lower triangular of the Cholesky decomposition of the covariance instead of the covariance matrix itself. The following function will compute this decomposition,

```
1 double chol[d * (d + 1) / 2];
2 cov_chol(d, cov, chol);
```

The output `chol` 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 STORING OBJECTS IN HDF5

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

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

create a HDF5 file with the sampler stored as a list. 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. Other types of objects can also be store, see the reference manual for details.

8.4 RAII CLASSES FOR MKL POINTERS

The library provides a few classes to manage MKL pointers. It provides Resource Acquisition Is Initialization (RAII) idiom on top of the MKL C interface. For example,

Class	MKL pointer type
`MKLStream`	`VSLStreamStatePtr`
`MKLSSTask`	`VSLSSTaskPtr`
`MKLConvTask`	`VSLConvTask`
`MKLCorrTask`	`VSLCorrTask`
`MKLDFTask`	`DFTaskPtr`

Table 8.1 RAII classes for MKL pointers

```
// VSLSSTaskPtr ptr;
// vsldSSNewTask(&ptr, &p, &n, &xstorage, x, w, indices);
MKLSSTask<double> task(&p, &n, &xstorage, x, w, indices);
// vsldSSEditMoments(ptr, mean, r2m, r3m, r4m, c2m, c3m, c4m);
task.edit_moments(mean, r2m, r3m, r4m, c2m, c3m, c4m);
// vsldSSCompute(ptr, estimates, method);
task.compute(estimates, method)
// vslSSDeleteTask(&ptr);
```

In the above snippets, `MKLSSTask` manages a `VSLSSTaskPtr` task pointer. All C functions that operates on the pointer, is also defined as methods in the class. Table 8.1 lists the classes defined by the library and their corresponding MKL pointers.

8.5 PROGRAM OPTIONS

The library provides some basic support of processing command line program options. Here we show a minimal example. The complete program is shown in appendix A.3. First, one need to allocated variables to store the options to be processed.

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

All types that support standard library I/O stream operations are supported. In addition, `std::vector<T>`, where `T` is a type that supports standard library I/O stream operations 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 option's value 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 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`). A string option's value can be grouped by quotes. For a vector option (`--vec`), all values are gather together and inserted into the vector.

8.6 PROGRAM PROGRESS

Sometime it is desirable to see how much progress of a program has been made. The library provide a `Progress` class for this purpose. Here we show a minimal example. The complete program is shown in appendix A.4.

```
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) {
        // Do some computation
        progress.increment();
```

```
    }
}
progress.stop();
```

When invoked, the program output something similar the below

```
[  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, we increment the progress count by calling `progress.increment()`. And after everything is finished, the method `progress.stop()` is called.

8.7 TIMING

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 time is only accurate if the computation between `watch.start()` and `watch.stop()` is non-trivial.

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

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.

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

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

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,

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.

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 SOURCE CODE OF COMPLETE PROGRAMS

A.1 SEQUENTIAL IMPLEMENTATION OF A SIMPLE PARTICLE FILTER

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

static constexpr std::size_t N = 1000; // Number of particles
static constexpr std::size_t n = 100;  // Number of data points
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;

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

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

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

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

    void read_data(const char *param)
    {
        if (param == nullptr)
            return;

        obs_x_.resize(n);
```

```
26          obs_y_.resize(n);
27          std::ifstream data(param);
28          for (std::size_t i = 0; i != n; ++i)
29              data >> obs_x_[i] >> obs_y_[i];
30          data.close();
31      }

32      private:
33      vsmc::Vector<double> obs_x_;
34      vsmc::Vector<double> obs_y_;
35  };

36  class PFInit
37  {
38      public:
39      std::size_t operator()(vsmc::Particle<PFState> &particle, void *param)
40      {
41          eval_param(particle, param);
42          eval_pre(particle);
43          std::size_t acc = 0;
44          for (auto sp : particle)
45              acc += eval_sp(sp);
46          eval_post(particle);

47          return acc;
48      }

49      void eval_param(vsmc::Particle<PFState> &particle, void *param)
50      {
51          particle.value().read_data(static_cast<const char *>(param));
52      }

53      void eval_pre(vsmc::Particle<PFState> &particle)
54      {
55          w_.resize(particle.size());
56      }

57      std::size_t eval_sp(vsmc::SingleParticle<PFState> sp)
```

```
58      {
59          vsmc::NormalDistribution<double> norm_pos(0, 2);
60          vsmc::NormalDistribution<double> norm_vel(0, 1);
61          sp.state(PosX) = norm_pos(sp.rng());
62          sp.state(PosY) = norm_pos(sp.rng());
63          sp.state(VelX) = norm_vel(sp.rng());
64          sp.state(VelY) = norm_vel(sp.rng());
65          w_[sp.id()] = sp.particle().value().log_likelihood(0, sp.id());

66          return 0;
67      }

68      void eval_post(vsmc::Particle<PFState> &particle)
69      {
70          particle.weight().set_log(w_.data());
71      }

72      private:
73      vsmc::Vector<double> w_;
74  };

75  class PFMove
76  {
77      public:
78      std::size_t operator()(std::size_t t, vsmc::Particle<PFState> &particle)
79      {
80          eval_pre(t, particle);
81          std::size_t acc = 0;
82          for (auto sp : particle)
83              acc += eval_sp(t, sp);
84          eval_post(t, particle);

85          return 0;
86      }

87      void eval_pre(std::size_t t, vsmc::Particle<PFState> &particle)
88      {
89          w_.resize(particle.size());
```

```
90      }

91      std::size_t eval_sp(std::size_t t, vsmc::SingleParticle<PFState> sp)
92      {
93          vsmc::NormalDistribution<double> norm_pos(0, std::sqrt(0.02));
94          vsmc::NormalDistribution<double> norm_vel(0, std::sqrt(0.001));
95          sp.state(PosX) += norm_pos(sp.rng()) + 0.1 * sp.state(VelX);
96          sp.state(PosY) += norm_pos(sp.rng()) + 0.1 * sp.state(VelY);
97          sp.state(VelX) += norm_vel(sp.rng());
98          sp.state(VelY) += norm_vel(sp.rng());
99          w_[sp.id()] = sp.particle().value().log_likelihood(t, sp.id());

100         return 0;
101     }

102     void eval_post(std::size_t t, vsmc::Particle<PFState> &particle)
103     {
104         particle.weight().add_log(w_.data());
105     }

106     private:
107     vsmc::Vector<double> w_;
108 };

109 class PFMEval
110 {
111     public:
112     void operator()(std::size_t t, std::size_t dim,
113         vsmc::Particle<PFState> &particle, double *r)
114     {
115         eval_pre(t, particle);
116         for (std::size_t i = 0; i != particle.size(); ++i, r += dim)
117             eval_sp(t, dim, particle.sp(i), r);
118         eval_post(t, particle);
119     }

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

```
121     void eval_sp(std::size_t t, std::size_t dim,
122         vsmc::SingleParticle<PFState> sp, double *r)
123     {
124         r[0] = sp.state(PosX);
125         r[1] = sp.state(PosY);
126     }

127     void eval_post(std::size_t t, vsmc::Particle<PFState> &particle) {}
128 };

129 int main()
130 {
131     vsmc::Sampler<PFState> sampler(N, vsmc::Multinomial, 0.5);
132     sampler.init(PFInit()).move(PFMove(), false).monitor("pos", 2, PFMEval());
133     sampler.initialize(const_cast<char *>("pf.data")).iterate(n - 1);

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

137     return 0;
138 }
```

A.2 PARALLELIZED IMPLEMENTATION OF A SIMPLE PARTICLE FILTER

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

2 static constexpr std::size_t N = 1000; // Number of particles
3 static constexpr std::size_t n = 100;  // Number of data points

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

5 template <typename T>
6 using PFStateSPBase = PFStateBase::single_particle_type<T>;

7 class PFState : public PFStateBase
8 {
```

```
 9      public:
10      using PFStateBase::StateMatrix;

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

16          double &pos_x() { return this->state(0); }
17          double &pos_y() { return this->state(1); }
18          double &vel_x() { return this->state(2); }
19          double &vel_y() { return this->state(3); }

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

26              return -0.5 * (10 + 1) * (llh_x + llh_y);
27          }

28          private:
29          double obs_x(std::size_t t)
30          {
31              return this->particle().value().obs_x_[t];
32          }

33          double obs_y(std::size_t t)
34          {
35              return this->particle().value().obs_y_[t];
36          }
37      };

38      void read_data(const char *param)
39      {
```

```
40          if (param == nullptr)
41              return;

42          obs_x_.resize(n);
43          obs_y_.resize(n);
44          std::ifstream data(param);
45          for (std::size_t i = 0; i != n; ++i)
46              data >> obs_x_[i] >> obs_y_[i];
47          data.close();
48      }

49      private:
50      vsmc::Vector<double> obs_x_;
51      vsmc::Vector<double> obs_y_;
52  };

53  class PFInit : public vsmc::InitializeTBB<PFState, PFInit>
54  {
55      public:
56      void eval_param(vsmc::Particle<PFState> &particle, void *param)
57      {
58          particle.value().read_data(static_cast<const char *>(param));
59      }

60      void eval_pre(vsmc::Particle<PFState> &particle)
61      {
62          w_.resize(particle.size());
63      }

64      std::size_t eval_sp(vsmc::SingleParticle<PFState> sp)
65      {
66          vsmc::NormalDistribution<double> norm_pos(0, 2);
67          vsmc::NormalDistribution<double> norm_vel(0, 1);
68          sp.pos_x() = norm_pos(sp.rng());
69          sp.pos_y() = norm_pos(sp.rng());
70          sp.vel_x() = norm_vel(sp.rng());
71          sp.vel_y() = norm_vel(sp.rng());
72          w_[sp.id()] = sp.log_likelihood(0);
```

```
73          return 0;
74      }

75      void eval_post(vsmc::Particle<PFState> &particle)
76      {
77          particle.weight().set_log(w_.data());
78      }

79      private:
80      vsmc::Vector<double> w_;
81  };

82  class PFMove : public vsmc::MoveTBB<PFState, PFMove>
83  {
84      public:
85      void eval_pre(std::size_t t, vsmc::Particle<PFState> &particle)
86      {
87          w_.resize(particle.size());
88      }

89      std::size_t eval_sp(std::size_t t, vsmc::SingleParticle<PFState> sp)
90      {
91          vsmc::NormalDistribution<double> norm_pos(0, std::sqrt(0.02));
92          vsmc::NormalDistribution<double> norm_vel(0, std::sqrt(0.001));
93          sp.pos_x() += norm_pos(sp.rng()) + 0.1 * sp.vel_x();
94          sp.pos_y() += norm_pos(sp.rng()) + 0.1 * sp.vel_y();
95          sp.vel_x() += norm_vel(sp.rng());
96          sp.vel_y() += norm_vel(sp.rng());
97          w_[sp.id()] = sp.log_likelihood(t);

98          return 0;
99      }

100     void eval_post(std::size_t t, vsmc::Particle<PFState> &particle)
101     {
102         particle.weight().add_log(w_.data());
103     }
```

```
104     private:
105     vsmc::Vector<double> w_;
106 };

107 class PFMEval : public vsmc::MonitorEvalTBB<PFState, PFMEval>
108 {
109     public:
110     void eval_sp(std::size_t t, std::size_t dim,
111         vsmc::SingleParticle<PFState> sp, double *r)
112     {
113         r[0] = sp.pos_x();
114         r[1] = sp.pos_y();
115     }
116 };

117 int main()
118 {
119     vsmc::Sampler<PFState> sampler(N, vsmc::Multinomial, 0.5);
120     sampler.init(PFInit()).move(PFMove(), false).monitor("pos", 2, PFMEval());
121     sampler.initialize(const_cast<char *>("pf.data")).iterate(n - 1);

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

125     return 0;
126 }
```

A.3 PROCESSING COMMAND LINE PROGRAM OPTIONS

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

2 int main(int argc, char **argv)
3 {
4     int n;
5     std::string str;
```

```
    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;
}
```

A.4 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)
```

```
16                 r += dist(rng);
17             progress.increment();
18         }
19     }
20     progress.stop();

21     return 0;
22 }
```