Yan Zhou

vSMC – Scalable Monte Carlo

Second edition (version develop)

March 27, 2016

CONTENTS

1 SEQUENTIAL MONTE CARLO

1.1 INTRODUCTION

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

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

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

1.2 SEQUENTIAL IMPORTANCE SAMPLING AND RESAMPLING

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

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

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

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

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

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

or

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

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

Sequential importance sampling (SIS) generalizes the importance sampling technique for a sequence of distributions $\{\pi_t\}_{t\ge0}$ defined on spaces $\{\prod_{k=0}^t E_k\}_{t\ge0}$. 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 $t \ge 1$, 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 $i = 1, \ldots, N$. 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

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

where $\{W_t^{(i)}\}_{i=1}^{N}$ are the normalized weights. And resampling can be performed when $\mathrm{ESS} \leq \alpha N$ 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 $i = 1, \ldots, N$ 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 $(W_t^{(i)}, \ldots, W_t^{(N)})$. See Douc, Cappé, and Moulines (2005) for some commonly used resampling algorithms.

1.3 SMC SAMPLERS

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

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

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

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

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

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

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

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

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

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

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

is used, the unnormalized incremental weights will be

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

1.4 RELATED ALGORITHMS

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

2 BASIC USAGE

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,

```
class T
{
    public:
    T(std::size_t N);

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

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. The object containing the weights is returned by `particle.weight()`. Its type is `Particle<T>::weight_type`, whose definition depends on the type `T`. See section 3.2 for more details. If the user does not do something special as shown in that section, then the default type is the `Weight`. 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

```
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 $f(X^{(1)},\dots,X^{(N)})$. 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 derived class of `StateMatrix`, then it has two additional methods,

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

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

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

```
for (auto sp : particle) {
    // 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,

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

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

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

Details can be found in section 8.3.

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,

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

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

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

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 B.1.1. In this section we show the outline of the implementation.

The main program

```
Sampler<PFState> sampler(N, Multinomial, 0.5);
sampler.init(PFInit()).move(PFMove(), false).monitor("pos", 2, PFEval());
sampler.initialize(const_cast<char *>("pf.data")).iterate(n - 1);

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

`Sampler<PFState>` object is constructed first. Then the initialization `PFInit`, move `PFMove` and a monitor `PFEval` 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[5] script that can be used to process the output

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

```r
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`,

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

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
{
```

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

    return acc;
}

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

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

std::size_t eval_sp(SingleParticle<PFState> sp)
{
    NormalDistribution<double> norm_pos(0, 2);
    NormalDistribution<double> norm_vel(0, 1);
    sp.state(PosX) = norm_pos(sp.rng());
    sp.state(PosY) = norm_pos(sp.rng());
    sp.state(VelX) = norm_vel(sp.rng());
    sp.state(VelY) = norm_vel(sp.rng());
    w_[sp.id()] = sp.particle().value().log_likelihood(0, sp.id());

    return 0;
}

void eval_post(Particle<PFState> &particle)
{
```

```
        particle.weight().set_log(weight_.data());
    }

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

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,

```
class PFMove
{
    public:
    std::size_t operator()(std::size_t t, Particle<PFState> &particle);
    void eval_pre(std::size_t t, Particle<PFState> &particle);
    std::size_t eval_sp(std::size_t t, SingleParticle<PFState> sp);
    void eval_post(std::size_t t, Particle<PFState> &particle);

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

Monitor: `PFEval`

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

```
class PFEval
{
    public:
    void operator()(std::size_t t, std::size_t dim,
```

```
        Particle<PFState> &particle, double *r)
    {
        eval_pre(t, particle);
        for (std::size_t i = 0; i != particle.size(); ++i, r += dim)
            eval_sp(t, dim, particle.sp(i), r);
        eval_post(t, particle);
    }

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

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

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

2.5 SYMMETRIC MULTIPROCESSING

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

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

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

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

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

2.5.1 *Performance consideration*

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

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

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

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

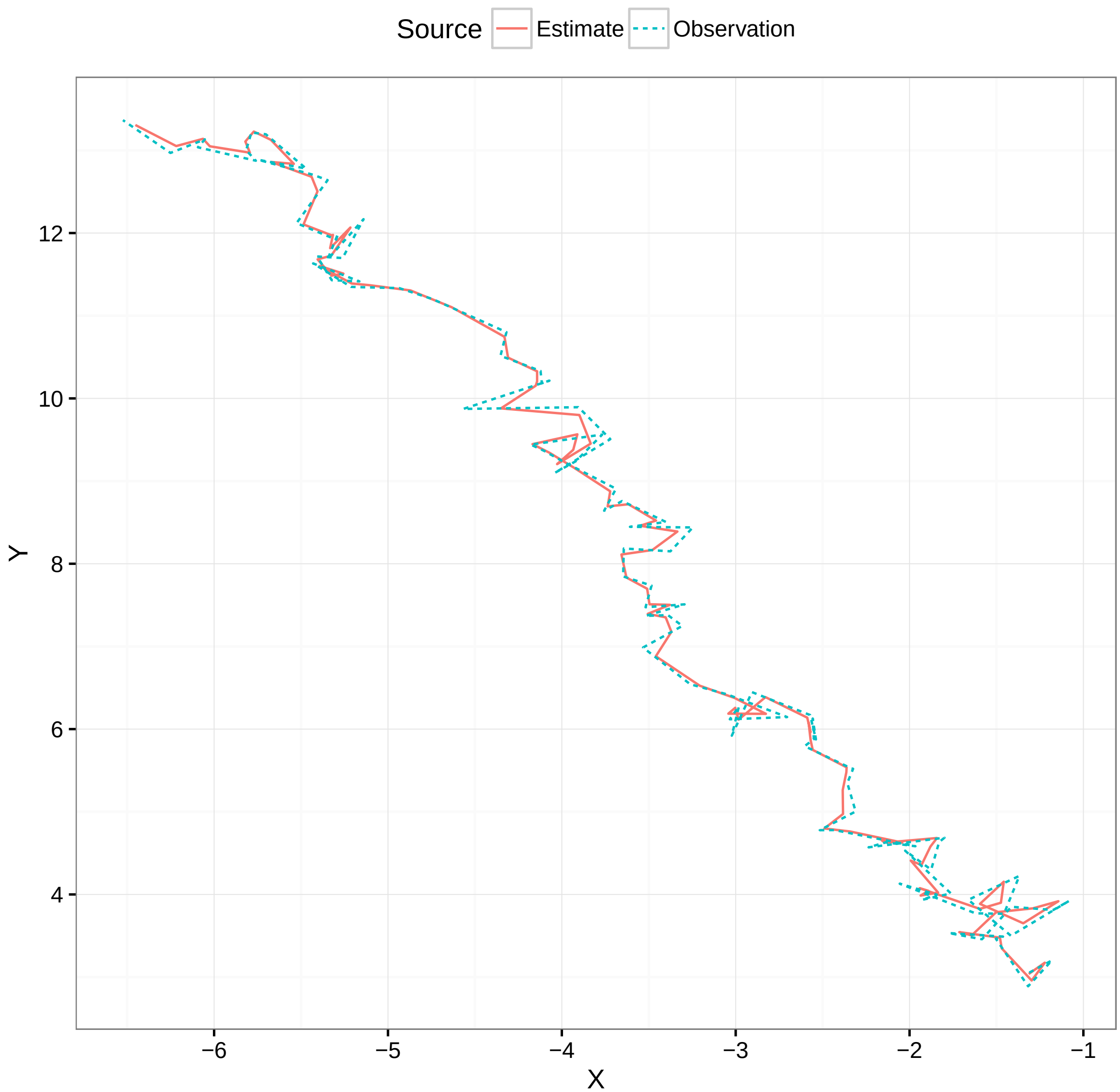

Figure 2.1 A simple particle filter

3.1 CLONING OBJECTS

The `Sampler<T>` and `Particle<T>` objects have copy constructors, assignment operators, move constructors, and move assignment operators that 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();
```

```
        } else {
            return nullptr;
        }
    }

    void set_equal()
    {
        // Set all weights to $1/\sum N_r$
        // Synchronization
    }

    void set(const double *v)
    {
        // Set $W^{(i)} = v_i$ for $i = 1, \dots, N_r$
        // Compute $S_r = \sum_{i=1}^{N_r} W^{(i)}$
        // Gathering $S_r$, compute $S = \sum S_r$
        // Broadcast $S$
        // Set $W^{(i)} = W^{(i)}/S$ for $i = 1, \dots, N_r$
    }
};
```

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

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

template <typename T>
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);
```

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

        return 0;
    }

    void eval_post(Particle<PFState> &particle);

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

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

3.3.1 *Compared to custom state type*

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

```
class PFStateSP
{
    public:
    double &pos_x() { return pos_x_; }
    double &pos_y() { return pos_y_; }
    double &vel_x() { return vel_x_; }
    double &vel_y() { return vel_y_; }

    double log_likelihood(double obs_x, double obs_y) const;

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

And the `PFState` class will be defined as,

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

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

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

    void read_data(const char *param);

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

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

```
class S
{
    public:
    double &x() { return x_; }
    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 `PFEval` in section 2.4.2. It is rather redundant to copy each value of the two positions, just so later we can compute weighted sums from them. Recall that in section 2.3.6 we showed that a monitor that compute the final results directly can also be added to a sampler. Therefore, we might implement `PFEval` as the following,

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

```
    }
};
```

And it can be added to a sampler as,

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

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.

4 CONFIGURATION MACROS

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

There are three types of configuration macros. The first type has a prefix `VSMC_HAS`. These macros specify a certain feature or third-party library is available. The second type has a prefix `VSMC_USE`. These macros specify that a certain feature or third-party library shall be used, if available. For example, if `VSMC_HAS_MKL` is defined to a non-zero value, but the it is desirable not to use MKL's vector math functions, then one can define `VSMC_USE_MKL_VML` to zero to prevent the library to use this individual component. All other macros define either types or constants that are used by the library.

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

Table 4.1 Configuration macros

5 MATHEMATICAL OPERATIONS

5.1 CONSTANTS

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

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

The compiler will evaluate these values at compile-time and thus there is no performance difference from hard-coding the constants in the program, while the readability is improved. All defined constants are listed in table 5.1.

5.2 VECTORIZED OPERATIONS

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

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

performs addition for vectors. It is equivalent to

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

The functions defined are listed in table 5.2 to 5.7. For each function, the first parameter is always the length of the vector, and the last is a pointer to the output vector (except `sincos` which has two output parameters). For all functions, the output is always a vector. If there are more than one input parameters, then some of them, but not all, can be scalars. For example, for the function call `fma(n, a, b, c, y)` in table 5.2, the input parameters are `a`, `b`, and `c`. Some of them, not all, can be scalars instead of pointers to vectors. The output parameter `y` has to be a pointer to a vector.

5.3 PACK AND UNPACK VECTORS

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

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

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

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

There are also three corresponding unpack functions,

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

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

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

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

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

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

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

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

Table 5.1 Mathematical constants

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

Table 5.2 Arithmetic functions

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

Table 5.3 Power and root functions

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

Table 5.4 Exponential and logarithm functions

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

Table 5.5 Trigonometric functions

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

Table 5.6 Hyperbolic functions

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

Table 5.7 Special functions

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

The first step determines the statistical properties of the resampling algorithm. The library defines all algorithms discussed in Douc, Cappé, and Moulines (2005). Samplers can be constructed with builtin schemes as seen in section 2.4.2. In addition, samplers can also be constructed with user defined resampling operations. A user defined resampling algorithm can be any type that is convertible to `Sampler<T>::resample_type`, following function call,

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

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

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

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

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

Table 6.1 Resampling schemes

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

6.1 RESIZING A SAMPLER

The library does not direct support varying sample size algorithms. However, it is possible. Below is a program within which the sample size is changed through a Multinomial resampling algorithm.

```
// 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(N, M, rng, weight, rep.data());
resample_trans_rep_index(N, M, rep.data(), idx.data());
Particle<T> particle(M);
particle.weight().set_equal();
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 vectors of replication numbers `rep` and indices `idx` are generated with the library's functions. Particles in the original sampler are copied into a temporary particle system `particle`. And then this temporary is moved into the original sampler. It is important to note that, if the `T` type object `particle.value()` has any other non-static member data other than the states, they also need to be copied into the temporary first. Note that `sampler.size()` is only a shortcut for `sampler.particle().size()`. The sampler object itself does not store size information, nor does it need to.

6.2 HIGH PERFORMANCE RESAMPLING FOR ALGORITHMS WITH INCREASING DIMENSIONS

Recall section 1.2, in general an SIS algorithm operates on increasing dimensions. Assume that the storage cost of a single particle at the marginal ($X_t^{(i)}$) is of order $O(1)$. At each iterations t, the path $X_{0:t-1}^{(i)}$ is extended to $X_{0:t}^{(i)}$. The resampling algorithms operate on the space $\prod_{k=0}^{t} E_k$. When the proposal $q_t(\cdot|X_{0:t-1}^{(i)}) = q_t(\cdot|X_{t-1}^{(i)})$ and only the marginal $\eta_t(X_t)$ is of interest, one can only resample $\{X_t^{(i)}\}_{i=1}^N$. This leads to $O(N)$ cost for resampling. This is the typical case for SMC algorithms. However, there are situations where resampling $\{X_{0:t}^{(i)}\}_{i=1}^N$ is necessary. In this case, the cost of resampling at iteration t is $O(tN)$. And total resampling cost to obtain $\{X_{0:t}^{(i)}\}_{i=1}^N$, is $O(t^2N)$.

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

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

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

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

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

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

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

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

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

```
using StateBase = StateMatrix<vsmc::ColMajor, vsmc::Dynamic, double>;
class State : StateBase
{
    public:
    StateBase(std::size_t N) : StateBase(N), varphi_(N) {}

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

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

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

        return rs;
    }

    private:
```

```
    Vector<double> varphi_; // the values of
    ResampleIndex index_;
};

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

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

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

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

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 note that, the RNGs introduced in the paper use deterministic functions 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 $k_1 \ne k_2$, f_{k_1} and f_{k_2} will generate two sequences that appear statistically independent. Compared to more conventional RNGs which use recursions $y_i = f(y_{i-1})$, 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-bit counter but output 64-bit integers. And thus it has a 2^{1024} period. Such period length may seem 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 instructions have a latency of seven or eight cycles, while they can be issued at every cycle. Therefore better performance can be achieved if multiple 128-bit random integers are generated at the same

Class	Result type	Bits	
		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	std::uint32_t	64	64
Philox2x64	std::uint64_t	128	128
Philox4x32	std::uint32_t	128	128
Philox4x64	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 counter-based RNGs

```
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. Type aliases 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.

The type aliases `ARS`, `AES128`, `AES192` and `AES256` are the respective engines with `std::uint32_t` as `ResultType` and other template parameters taking their default values. Similarly, `ARS_64`, `AES128_64`, `AES192_64` and `AES256_64` are type aliases with `std::uint64_t` as `ResultType`.

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 7.2. Type aliases 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. The type aliases `Philox` and `Philox_64` are defined similarly to `ARS` and `ARS_64`, respectively.

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 7.2. Type aliases 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. The type aliases `Threefry` and `Threefry_64` are defined similarly to `ARS` and `ARS_64`, respectively.

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

```
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>>;
```

The type alias `ThreefrySSE2` and `ThreefrySSE2_64` are defined similarly to `ARS` and `ARS_64`, respectively.

If AVX2 intrinsics are supported, then the following 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 ThreefryEngineAVX2 =
    CounterEngine<ThreefryGeneratorAVX2<ResultType, K, Rounds>>;
```

The type aliases `ThreefryAVX2` and `ThreefryAVX2_64` are defined similar to `ARS` and `ARS_64`, respectively.

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. The preference is in the order listed in table 7.3. The user can define the configuration macro `VSMC_RNG_TYPE` to override the choice made by the library.

7.3 NON-DETERMINISTIC RNG

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

Class	Availability
ARS	VSMC_HAS_AES_NI
ThreefryAVX2	VSMC_HAS_AVX2
ThreefrySSE2	VSMC_HAS_SSE2
Threefry	Always available

Table 7.3 Default RNG

Class	Result type	MKL BRNG
MKL_MCG59	unsigned	VSL_BRNG_MCG59
MKL_MCG59_64	unsigned MKL_INT64	VSL_BRNG_MCG59
MKL_MT19937	unsigned	VSL_BRNG_MT19937
MKL_MT19937_64	unsigned MKL_INT64	VSL_BRNG_MT19937
MKL_MT2203	unsigned	VSL_BRNG_MT2203
MKL_MT2203_64	unsigned MKL_INT64	VSL_BRNG_MT2203
MKL_SFMT19937	unsigned	VSL_BRNG_SFMT19937
MKL_SFMT19937_64	unsigned MKL_INT64	VSL_BRNG_SFMT19937
MKL_NONDETERM	unsigned	VSL_BRNG_NONDETERM
MKL_NONDETERM_64	unsigned MKL_INT64	VSL_BRNG_NONDETERM
MKL_ARS5	unsigned	VSL_BRNG_ARS5
MKL_ARS5_64	unsigned MKL_INT64	VSL_BRNG_ARS5
MKL_PHILOX4X32X10	unsigned	VSL_BRNG_PHILOX4X32X10
MKL_PHILOX4X32X10_64	unsigned MKL_INT64	VSL_BRNG_PHILOX4X32X10

Table 7.4 MKL RNG

7.4 MKL RNG

The MKL library provides some high performance RNGs. The library implements a wrapper class `MKLEngine` that makes them accessible as C++11 generators. They are listed in table 7.4. 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,

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

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

The second implementation is `RNGSetVector`. It is an array of RNGs with length N. It has memory cost $O(N)$. 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 = RNGSetScalar<RNG>;
};
```

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

7.6 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 a drop-in replacement of `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 7.5 lists all the additional distributions implemented. The library also implement the multivariate Normal distribution. Its usable is shown by the following example,

```
double mean[2] = { /* the mean vector */ };
double cov[4] = { /* the covariance matrix */ };
double chol[3];
double r[2];
// 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 (`norm2`), where the dimension is specified as a template parameter is more efficient. The distribution accept the lower triangular of the Choleskey decomposition of the covariance matrix as input, instead of the covariance matrix itself. The input matrix shall be in packed format, with row major storage layout. A function `cov_chol` is provided to compute this decomposition from a covariance matrix stored in various format. The output is suitable as direct input to the distribution. See section 8.2 for details.

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
```

Class	Notes
UniformBits	No parameter, uniform on the set $\{0, \dots, 2^b - 1\}$, where b is the number of bits of the result type, which has to be an unsigned integer type.
U01	No parameter, uniform on $[0, 1)$
U01CC	No parameter, uniform on $[0, 1]$
U01CO	No parameter, uniform on $[0, 1)$
U01OC	No parameter, uniform on $(0, 1]$
U01OO	No parameter, 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 7.5 Random number distributions. Note: all class names have a suffix Distribution which is omitted in the table

```
rng_rand(rng, n, u.data()); // Same as above
norm(rng, n, r.data());     // Generate n Normal random numbers
normal_distribution(rng, n, r.data(), 0.0, 1.0);      // Same as above
normal_distribution(rng, n, r.data(), norm.param()); // Same as above
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 7.4.

7.8 RNG IN C AND OPENCL

The Philox and Threefry engines, together with a set of functions that transfer unsigned random integers to floating points numbers, are also implemented with only a subset of C99 features. They can be used in both C99 and OpenCL 1.2 device programs. See appendix B.1.5 for an example of such an OpenCL program. Here we provide a minimal example that generates two standard Normal random numbers using the Box-Muller method,

```
vsmc_threefry4x32 rng;
vsmc_threefry4x32_init(&rng, seed);

uint32_t u32[2];
u32[0] = vsmc_threefry4x32_rand(&rng);
u32[1] = vsmc_threefry4x32_rand(&rng);
```

```
double u01[2];
u01[0] = vsmc_u01_oc_u32d(u32[0]);
u01[1] = vsmc_u01_oc_u32d(u32[1]);
u01[0] = sqrt(-2 * log(u01[0]));
u01[1] *= 2;

double normal01[2];
normal01[0] = u01[0] * sin(M_PI * u01[1]);
normal01[1] = u01[0] * cos(M_PI * u01[1]);
```

For each engine type, there are two primary functions. One is used to initialize the state. These functions take the form of the engine name suffixed with `_init`. They take a pointer to the RNG and an unsigned integer as input. The other function is used to increment the counter and generate unsigned random integers. These functions take the form of the engine name suffixed with `_rand`. They take a pointer to the RNG as input and return an unsigned integer.

The functions that convert unsigned random integers to floating points uniform on $[0, 1]$ take the form `vsmc_u01_`*lr*`_u`*bitsfp*, where *l* specify the lower bound of the interval to be closed (`c`) or open (`o`); similarly *r* specify the upper bound of the interval; *bits* can be either `32` or `64`, which specifies the input type to be `uint32_t` or `uint64_t`, respectively; and last *fp* can be either `f`, `d`, or `l`, which specifies the output type to be `float`, `double`, or `long double`, respectively. Note that, if used in OpenCL device programs, the `long double` versions are not defined. And the `double` versions are only defined if the macro `VSMC_HAS_OPENCL_DOUBLE` is defined to a non-zero value.

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

8.1 ALIGNED MEMORY ALLOCATION

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

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

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

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

The method `aligned_malloc` shall behave similar to `std::malloc` with the additional alignment requirement. It shall return a null pointer if it fails to allocate memory. In any other case, including zero input size, it shall return a reachable non-null pointer. The method `aligned_free` shall behave similar to `std::free`. It shall be able to handle a null pointer as its input. The library provides a few implementations, listed in table 8.1. In addition, a type alias `AlignedMemory` is defined to be one of the class listed in the table, depending on the availability of those classes, with preference in the same order as they are listed. The user can define the configuration macro `VSMC_ALIGNED_MEMORY_TYPE` to override the choice made by the library.

The default alignment depends on the type `T`. If it is a scalar type (`std::is_scalar<T>`), then the alignment is `VSMC_ALIGNMENT`, whose default is 32. This alignment is sufficient for modern SIMD operations, such as AVX2. For other types, the alignment is the maximum of `alignof(T)` and `VSMC_ALIGNMENT_MIN`, whose default is 16.

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

Last, a type alias `Vector` is defined,

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

Table 8.1 Aligned memory allocation

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

This vector type is used throughout the library.

8.2 SAMPLE COVARIANCE

The library provides some basic functionality to estimate sample 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. This method has three optional parameters. The first is `cov_layout`, which specifies the storage layout of `cov`. The second is `cov_upper` and the third is `cov_packed`, both are `false` by default. If the later is `true`, a packed vector of length $d(d+1)/2$ is written into `cov`. If `cov_upper` is `true`, then the upper triangular is packed, otherwise the lower triangular is packed.

The estimated covariance matrix is often used to construct multivariate Normal distribution for the purpose of generating random walk proposals. The `NormalMVDistribution` in section 7.6 accepts the lower triangular of the Cholesky decomposition of the covariance matrix instead of itself. The following function will compute this decomposition,

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

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 STORE OBJECTS IN HDF5 FORMAT

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

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

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

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

This creates a `data.frame` similar to that shown in section 2.4.2. The `hdf5store` function is overloaded for `StateMatrix`, `Sampler<T>` and `Monitor<T>`. It is also overloaded for `Particle<T>` if an overload for `T` is available. Such an overload is automatically available if `T` is a derived class of `StateMatrix`. However, it may not be the most suitable one. Other types of objects can also be stored, see the reference manual for details.

8.4 RAII CLASSES FOR 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, below is a small program using the `MKLSSTask` class,

Class	MKL pointer type	Copyable
`MKLStream`	`VSLStreamStatePtr`	Yes
`MKLSSTask`	`VSLSSTaskPtr`	No
`MKLConvTask`	`VSLConvTask`	Yes
`MKLCorrTask`	`VSLCorrTask`	Yes
`MKLDFTask`	`DFTaskPtr`	No

Table 8.2 RAII classes for MKL pointers

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

In the above program, the `MKLSSTask` type object manages a `VSLSSTaskPtr` task pointer. The resources managed will be released when the object is destroyed. All C functions that operates on the pointer, is also defined as methods in the class. Table 8.2 lists the classes defined by the library and their corresponding MKL pointers. For those classes that are copyable, the copy constructor and assignment operator perform deep copy. It is safe to use the copy and the original independently. Those classes that are not copyable are movable.

8.5 RAII CLASSES FOR OPENCL POINTERS

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

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

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

Table 8.3 RAII classes for OpenCL pointers

In the above program, each class type object manages an OpenCL C type, such as `cl_platform`. The resources will be released when the object is destroyed. Note that, the copy constructor and assignment operator perform shallow copy. This is particularly important for `CLMemory` type objects. In appendix B.1.5 an OpenCL implementation of the simple particle filter example in section 2.4 is shown. Table 8.3 lists the classes defined by the library and their corresponding OpenCL pointers.

8.6 PROCESS COMMAND LINE PROGRAM OPTIONS

The library provides some basic support for processing command line options. Here we show a minimal example. The complete program is shown in appendix B.2. First, we allocate define to store values of options,

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

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

```
ProgramOptionMap option_map;
```

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

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

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

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

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

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

The results of the option processing is displayed below,

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

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

8.7 DISPLAY PROGRAM PROGRESS

Sometime it is desirable to see how much progress of a program has been made. The library provides a `Progress` class for this purpose. Here we show a minimal example. The complete program is shown in appendix B.3.

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

When invoked, the program output something similar the following,

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

The method `progress.start(n * n)` starts the printing of the progress. The argument specifies how many iterations there will be before it is stopped. The method `progress.message(ss.str())` direct the program to print a message. This is optional. Each time after we finish n iterations (there are n^3 total iterations of the inner-most loop), we increment the progress count by calling `progress.increment()`. And after everything is finished, the method `progress.stop()` is called. The `increment` method has an optional argument, which specifies how many steps has been finished. The default is one. For example, we can call `progress.start(n * n * n)` and `progress.increment(n)` instead.

8.8 STOP WATCH

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

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

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

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 INTERFACING WITH C AND OTHER LANGUAGES

B SOURCE CODE OF COMPLETE PROGRAMS

B.1 A SIMPLE PARTICLE FILTER

B.1.1 *Sequential implementation*

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

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;

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

```
        while (data.good()) {
            double x;
            double y;
            data >> x >> y;
            if (data.good()) {
                obs_x_.push_back(x);
                obs_y_.push_back(y);
            }
        }
        data.close();
    }

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

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

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

        return acc;
    }

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

```
    void eval_pre(vsmc::Particle<PFState> &particle)
    {
        auto &rng = particle.rng();
        const std::size_t size = particle.size();
        const double sd_pos = 2;
        const double sd_vel = 1;
        pos_x_.resize(size);
        pos_y_.resize(size);
        vel_x_.resize(size);
        vel_y_.resize(size);
        weight_.resize(size);
        vsmc::normal_distribution(rng, size, pos_x_.data(), 0.0, sd_pos);
        vsmc::normal_distribution(rng, size, pos_y_.data(), 0.0, sd_pos);
        vsmc::normal_distribution(rng, size, vel_x_.data(), 0.0, sd_vel);
        vsmc::normal_distribution(rng, size, vel_y_.data(), 0.0, sd_vel);
    }

    std::size_t eval_sp(vsmc::SingleParticle<PFState> sp)
    {
        sp.state(PosX) = pos_x_[sp.id()];
        sp.state(PosY) = pos_y_[sp.id()];
        sp.state(VelX) = vel_x_[sp.id()];
        sp.state(VelY) = vel_y_[sp.id()];
        weight_[sp.id()] = sp.particle().value().log_likelihood(0, sp.id());

        return 0;
    }

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

    private:
    vsmc::Vector<double> pos_x_;
    vsmc::Vector<double> pos_y_;
    vsmc::Vector<double> vel_x_;
```

```
    vsmc::Vector<double> vel_y_;
    vsmc::Vector<double> weight_;
};

class PFMove
{
    public:
    std::size_t operator()(std::size_t t, vsmc::Particle<PFState> &particle)
    {
        eval_pre(t, particle);
        std::size_t acc = 0;
        for (auto sp : particle)
            acc += eval_sp(t, sp);
        eval_post(t, particle);

        return 0;
    }

    void eval_pre(std::size_t t, vsmc::Particle<PFState> &particle)
    {
        auto &rng = particle.rng();
        const std::size_t size = particle.size();
        const double sd_pos = sqrt(0.02);
        const double sd_vel = sqrt(0.001);
        pos_x_.resize(size);
        pos_y_.resize(size);
        vel_x_.resize(size);
        vel_y_.resize(size);
        weight_.resize(size);
        vsmc::normal_distribution(rng, size, pos_x_.data(), 0.0, sd_pos);
        vsmc::normal_distribution(rng, size, pos_y_.data(), 0.0, sd_pos);
        vsmc::normal_distribution(rng, size, vel_x_.data(), 0.0, sd_vel);
        vsmc::normal_distribution(rng, size, vel_y_.data(), 0.0, sd_vel);
    }

    std::size_t eval_sp(std::size_t t, vsmc::SingleParticle<PFState> sp)
    {
        sp.state(PosX) += pos_x_[sp.id()] + 0.1 * sp.state(VelX);
```

```
        sp.state(PosY) += pos_y_[sp.id()] + 0.1 * sp.state(VelY);
        sp.state(VelX) += vel_x_[sp.id()];
        sp.state(VelY) += vel_y_[sp.id()];
        weight_[sp.id()] = sp.particle().value().log_likelihood(t, sp.id());

        return 0;
    }

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

    private:
    vsmc::Vector<double> pos_x_;
    vsmc::Vector<double> pos_y_;
    vsmc::Vector<double> vel_x_;
    vsmc::Vector<double> vel_y_;
    vsmc::Vector<double> weight_;
};

class PFEval
{
    public:
    void operator()(std::size_t t, std::size_t dim,
        vsmc::Particle<PFState> &particle, double *r)
    {
        eval_pre(t, particle);
        for (std::size_t i = 0; i != particle.size(); ++i, r += dim)
            eval_sp(t, dim, particle.sp(i), r);
        eval_post(t, particle);
    }

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

    void eval_sp(std::size_t t, std::size_t dim,
        vsmc::SingleParticle<PFState> sp, double *r)
    {
```

```
        r[0] = sp.state(PosX);
        r[1] = sp.state(PosY);
    }

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

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

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

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

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

    return 0;
}
```

B.1.2 *Parallelized implementation using TBB*

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

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

template <typename T>
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); }

        double log_likelihood(std::size_t t)
        {
            double llh_x = 10 * (pos_x() - obs_x(t));
            double llh_y = 10 * (pos_y() - 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);
        }

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

        double obs_y(std::size_t t)
        {
            return this->particle().value().obs_y_[t];
        }
    };
```

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

        std::ifstream data(param);
        while (data.good()) {
            double x;
            double y;
            data >> x >> y;
            if (data.good()) {
                obs_x_.push_back(x);
                obs_y_.push_back(y);
            }
        }
        data.close();
    }

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

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

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

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

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

        return 0;
    }

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

    private:
    vsmc::Vector<double> weight_;
};

class PFMove : public vsmc::MoveTBB<PFState, PFMove>
{
    public:
    void eval_pre(std::size_t t, vsmc::Particle<PFState> &particle)
    {
        weight_.resize(particle.size());
    }

    std::size_t eval_sp(std::size_t t, vsmc::SingleParticle<PFState> sp)
    {
        vsmc::NormalDistribution<double> norm_pos(0, std::sqrt(0.02));
        vsmc::NormalDistribution<double> norm_vel(0, std::sqrt(0.001));
        sp.pos_x() += norm_pos(sp.rng()) + 0.1 * sp.vel_x();
        sp.pos_y() += norm_pos(sp.rng()) + 0.1 * sp.vel_y();
        sp.vel_x() += norm_vel(sp.rng());
```

```
        sp.vel_y() += norm_vel(sp.rng());
        weight_[sp.id()] = sp.log_likelihood(t);

        return 0;
    }

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

    private:
    vsmc::Vector<double> weight_;
};

class PFEval : public vsmc::MonitorEvalTBB<PFState, PFEval>
{
    public:
    void eval_sp(std::size_t t, std::size_t dim,
        vsmc::SingleParticle<PFState> sp, double *r)
    {
        r[0] = sp.pos_x();
        r[1] = sp.pos_y();
    }
};

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

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

    vsmc::StopWatch watch;
    watch.start();
    sampler.initialize(const_cast<char *>("pf.data"));
```

```
    sampler.iterate(sampler.particle().value().data_size() - 1);
    watch.stop();
    std::cout << "Time (ms): " << watch.milliseconds() << std::endl;

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

    return 0;
}
```

B.1.3 *Sequential implementation in C*

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

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

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

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

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

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

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

static inline void pf_read_data(const char *param)
{
    if (!param)
        return;

    FILE *data = fopen(param, "r");
    int n = 0;
    while (1) {
        double x;
        double y;
        int nx = fscanf(data, "%lg", &x);
        int ny = fscanf(data, "%lg", &y);
        if (nx == 1 && ny == 1)
            ++n;
        else
            break;
    }
    vsmc_vector_resize(&pf_obs_x, n);
    vsmc_vector_resize(&pf_obs_y, n);
    fseek(data, 0, SEEK_SET);
    for (int i = 0; i < n; ++i) {
        fscanf(data, "%lg", &pf_obs_x.data[i]);
        fscanf(data, "%lg", &pf_obs_y.data[i]);
    }
    fclose(data);
}
```

```
static inline void pf_normal(
    vsmc_particle particle, double sd_pos, double sd_vel)
{
    vsmc_rng rng = vsmc_particle_rng(particle, 0);
    const int size = vsmc_particle_size(particle);
    vsmc_vector_resize(&pf_pos_x, size);
    vsmc_vector_resize(&pf_pos_y, size);
    vsmc_vector_resize(&pf_vel_x, size);
    vsmc_vector_resize(&pf_vel_y, size);
    vsmc_vector_resize(&pf_weight, size);
    vsmc_normal_distribution(rng, size, pf_pos_x.data, 0, sd_pos);
    vsmc_normal_distribution(rng, size, pf_pos_y.data, 0, sd_pos);
    vsmc_normal_distribution(rng, size, pf_vel_x.data, 0, sd_vel);
    vsmc_normal_distribution(rng, size, pf_vel_y.data, 0, sd_vel);
}

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

    pf_normal(particle, 2, 1);

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

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

    return 0;
}

static inline int pf_move(int t, vsmc_particle particle)
```

```
{
    pf_normal(particle, sqrt(0.02), sqrt(0.001));

    const int size = vsmc_particle_size(particle);
    for (int i = 0; i < size; ++i) {
        vsmc_single_particle sp = vsmc_particle_sp(particle, i);
        sp.state[PosX] += pf_pos_x.data[i] + 0.1 * sp.state[VelX];
        sp.state[PosY] += pf_pos_y.data[i] + 0.1 * sp.state[VelY];
        sp.state[VelX] += pf_vel_x.data[i];
        sp.state[VelY] += pf_vel_y.data[i];
        pf_weight.data[i] = pf_log_likelihood(t, &sp);
    }

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

    return 0;
}

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

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

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

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

    vsmc_sampler_save_f(sampler, "pf.out");

    vsmc_sampler_delete(&sampler);
    vsmc_vector_delete(&pf_obs_x);
    vsmc_vector_delete(&pf_obs_y);
    vsmc_vector_delete(&pf_pos_x);
    vsmc_vector_delete(&pf_pos_y);
    vsmc_vector_delete(&pf_vel_x);
    vsmc_vector_delete(&pf_vel_y);
    vsmc_vector_delete(&pf_weight);

    return 0;
}
```

B.1.4 *Parallelized implementation using TBB in C*

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

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

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

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

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

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

static inline void pf_read_data(const char *param)
{
    if (!param)
        return;

    FILE *data = fopen(param, "r");
    int n = 0;
    while (1) {
        double x;
        double y;
        int nx = fscanf(data, "%lg", &x);
        int ny = fscanf(data, "%lg", &y);
        if (nx == 1 && ny == 1)
            ++n;
        else
            break;
    }
    vsmc_vector_resize(&pf_obs_x, n);
    vsmc_vector_resize(&pf_obs_y, n);
    fseek(data, 0, SEEK_SET);
```

```
    for (int i = 0; i < n; ++i) {
        fscanf(data, "%lg", &pf_obs_x.data[i]);
        fscanf(data, "%lg", &pf_obs_y.data[i]);
    }
    fclose(data);
}

static inline void pf_normal(
    vsmc_particle particle, double sd_pos, double sd_vel)
{
    vsmc_rng rng = vsmc_particle_rng(particle, 0);
    const int size = vsmc_particle_size(particle);
    vsmc_vector_resize(&pf_pos_x, size);
    vsmc_vector_resize(&pf_pos_y, size);
    vsmc_vector_resize(&pf_vel_x, size);
    vsmc_vector_resize(&pf_vel_y, size);
    vsmc_vector_resize(&pf_weight, size);
    vsmc_normal_distribution(rng, size, pf_pos_x.data, 0, sd_pos);
    vsmc_normal_distribution(rng, size, pf_pos_y.data, 0, sd_pos);
    vsmc_normal_distribution(rng, size, pf_vel_x.data, 0, sd_vel);
    vsmc_normal_distribution(rng, size, pf_vel_y.data, 0, sd_vel);
}

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

static inline void pf_init_pre(vsmc_particle particle)
{
    pf_normal(particle, 2, 1);
}

static inline int pf_init_sp(vsmc_single_particle sp)
{
    sp.state[PosX] = pf_pos_x.data[sp.id];
    sp.state[PosY] = pf_pos_y.data[sp.id];
    sp.state[VelX] = pf_vel_x.data[sp.id];
```

```
    sp.state[VelY] = pf_vel_y.data[sp.id];
    pf_weight.data[sp.id] = pf_log_likelihood(0, &sp);

    return 0;
}

static inline void pf_init_post(vsmc_particle particle)
{
    vsmc_weight_set_log(vsmc_particle_weight(particle), pf_weight.data, 1);
}

static inline void pf_move_pre(int t, vsmc_particle particle)
{
    pf_normal(particle, sqrt(0.02), sqrt(0.001));
}

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

    return 0;
}

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

static inline void pf_eval_sp(
    int t, int dim, vsmc_single_particle sp, double *r)
{
    r[0] = sp.state[PosX];
    r[1] = sp.state[PosY];
}
```

```
int main(int argc, char **argv)
{
    int N = 10000;
    if (argc > 1)
        N = atoi(argv[1]);

    vsmc_sampler sampler = vsmc_sampler_new(N, 4, vSMCMultinomial, 0.5);
    vsmc_sampler_init_tbb(
        sampler, pf_init_sp, pf_init_param, pf_init_pre, pf_init_post);
    vsmc_sampler_move_tbb(sampler, pf_move_sp, pf_move_pre, pf_move_post, 0);
    vsmc_sampler_set_monitor_tbb(
        sampler, "pos", 2, pf_eval_sp, NULL, NULL, 0, vSMCMonitorMCMC);

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

    vsmc_sampler_save_f(sampler, "pf.out");

    vsmc_sampler_delete(&sampler);
    vsmc_vector_delete(&pf_obs_x);
    vsmc_vector_delete(&pf_obs_y);
    vsmc_vector_delete(&pf_pos_x);
    vsmc_vector_delete(&pf_pos_y);
    vsmc_vector_delete(&pf_vel_x);
    vsmc_vector_delete(&pf_vel_y);
    vsmc_vector_delete(&pf_weight);

    return 0;
}
```

B.1.5 *Parallelized implementation using OpenCL*

Host program

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

static constexpr std::size_t N = 10000; // 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;

static constexpr std::size_t G = 10240;
static constexpr std::size_t L = 256;

typedef struct {
    cl_float pos_x;
    cl_float pos_y;
    cl_float vel_x;
    cl_float vel_y;
} cl_pf_sp;

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

class PFState : public PFStateBase
{
    public:
    using size_type = cl_int;
    using PFStateBase::StateMatrix;

    void initialize(const vsmc::CLContext &context,
        const vsmc::CLCommandQueue &command_queue,
        const vsmc::CLKernel &kernel)
    {
        command_queue_ = command_queue;
        kernel_ = kernel;
```

```
        dev_data_ =
            vsmc::CLMemory(context, CL_MEM_READ_WRITE | CL_MEM_HOST_READ_ONLY,
                sizeof(cl_pf_sp) * size());
        dev_weight_ =
            vsmc::CLMemory(context, CL_MEM_WRITE_ONLY | CL_MEM_HOST_READ_ONLY,
                sizeof(cl_float) * size());
        dev_rng_set_ =
            vsmc::CLMemory(context, CL_MEM_WRITE_ONLY | CL_MEM_HOST_READ_ONLY,
                sizeof(vsmc_threefry4x32) * size());
        dev_index_ =
            vsmc::CLMemory(context, CL_MEM_READ_ONLY | CL_MEM_HOST_WRITE_ONLY,
                sizeof(cl_int) * size());
        dev_obs_x_ = vsmc::CLMemory(context,
            CL_MEM_READ_ONLY | CL_MEM_HOST_WRITE_ONLY, sizeof(cl_float) * n);
        dev_obs_y_ = vsmc::CLMemory(context,
            CL_MEM_READ_ONLY | CL_MEM_HOST_WRITE_ONLY, sizeof(cl_float) * n);
    }

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

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

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

    void copy_to_host()
    {
        command_queue_.enqueue_read_buffer(
            dev_data_, CL_TRUE, 0, sizeof(cl_pf_sp) * size(), data());
    }
```

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

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

        command_queue_.enqueue_write_buffer(
            dev_obs_x_, CL_TRUE, 0, sizeof(cl_float) * n, obs_x.data());
        command_queue_.enqueue_write_buffer(
            dev_obs_y_, CL_TRUE, 0, sizeof(cl_float) * n, obs_y.data());
    }

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

    private:
    vsmc::CLCommandQueue command_queue_;
    vsmc::CLKernel kernel_;
    vsmc::CLMemory dev_data_;
    vsmc::CLMemory dev_rng_set_;
    vsmc::CLMemory dev_weight_;
    vsmc::CLMemory dev_index_;
    vsmc::CLMemory dev_obs_x_;
    vsmc::CLMemory dev_obs_y_;
};

class PFInit
{
    public:
```

```
    PFInit(const vsmc::CLCommandQueue &command_queue,
        const vsmc::CLKernel &kernel)
        : command_queue_(command_queue), kernel_(kernel)
    {
    }

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

        kernel_.set_arg(0, static_cast<cl_int>(particle.size()));
        kernel_.set_arg(1, particle.value().dev_data());
        kernel_.set_arg(2, particle.value().dev_rng_set());
        kernel_.set_arg(3, particle.value().dev_weight());
        kernel_.set_arg(4, particle.value().dev_obs_x());
        kernel_.set_arg(5, particle.value().dev_obs_y());

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

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

        return 0;
    }

    private:
    vsmc::CLCommandQueue command_queue_;
    vsmc::CLKernel kernel_;
    vsmc::Vector<cl_float> weight_;
};

class PFMove
{
    public:
```

```
    PFMove(const vsmc::CLCommandQueue &command_queue,
        const vsmc::CLKernel &kernel)
        : command_queue_(command_queue), kernel_(kernel)
    {
    }

    std::size_t operator()(std::size_t t, vsmc::Particle<PFState> &particle)
    {
        kernel_.set_arg(0, static_cast<cl_int>(t));
        kernel_.set_arg(1, static_cast<cl_int>(particle.size()));
        kernel_.set_arg(2, particle.value().dev_data());
        kernel_.set_arg(3, particle.value().dev_rng_set());
        kernel_.set_arg(4, particle.value().dev_weight());
        kernel_.set_arg(5, particle.value().dev_obs_x());
        kernel_.set_arg(6, particle.value().dev_obs_y());

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

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

        return 0;
    }

    private:
    vsmc::CLCommandQueue command_queue_;
    vsmc::CLKernel kernel_;
    vsmc::Vector<cl_float> weight_;
};

class PFEval : public vsmc::MonitorEvalTBB<PFState, PFEval>
{
    public:
    void eval_pre(std::size_t t, vsmc::Particle<PFState> &particle)
```

```
    {
        particle.value().copy_to_host();
    }

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

int main()
{
    auto platform = vsmc::cl_get_platform().front();
    std::string platform_name;
    platform.get_info(CL_PLATFORM_NAME, platform_name);
    std::cout << "Platform: " << platform_name << std::endl;

    auto device = platform.get_device(CL_DEVICE_TYPE_DEFAULT).front();
    std::string device_name;
    device.get_info(CL_DEVICE_NAME, device_name);
    std::cout << "Device:   " << device_name << std::endl;

    vsmc::CLContext context(vsmc::CLContextProperties(platform), 1, &device);
    vsmc::CLCommandQueue command_queue(context, device);

    std::ifstream source_cl("pf_ocl.cl");
    std::string source((std::istreambuf_iterator<char>(source_cl)),
        std::istreambuf_iterator<char>());
    source_cl.close();
    vsmc::CLProgram program(context, 1, &source);
    program.build(1, &device, "-I ../../include");

    vsmc::CLKernel kernel_copy(program, "copy");
    vsmc::CLKernel kernel_init(program, "init");
    vsmc::CLKernel kernel_move(program, "move");
```

```
    std::size_t pwgsm_copy = 0;
    std::size_t pwgsm_init = 0;
    std::size_t pwgsm_move = 0;

    kernel_copy.get_work_group_info(
        device, CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE, pwgsm_copy);
    kernel_init.get_work_group_info(
        device, CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE, pwgsm_init);
    kernel_move.get_work_group_info(
        device, CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE, pwgsm_move);

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

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

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

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

    return 0;
}
```

Device program

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

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

static inline float log_likelihood(const pf_sp *sp, float obs_x, float obs_y)
{
    float llh_x = 10 * (sp->pos_x - obs_x);
    float llh_y = 10 * (sp->pos_y - obs_y);
    llh_x = log(1 + llh_x * llh_x / 10);
    llh_y = log(1 + llh_y * llh_y / 10);

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

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

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

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

```
    u01[3] *= 2;

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

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

    float r[4];
    rnorm(rng, r);
    const float sd_pos = 2.0f;
    const float sd_vel = 1.0f;
    sp->pos_x = r[0] * sd_pos;
    sp->pos_y = r[1] * sd_pos;
    sp->vel_x = r[2] * sd_vel;
    sp->vel_y = r[3] * sd_vel;

    return log_likelihood(sp, obs_x, obs_y);
}

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

    return log_likelihood(sp, obs_x, obs_y);
```

```
}

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

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

__kernel void init(int N, __global pf_sp *state,
    __global vsmc_threefry4x32 *rng_set, __global float *weight,
    const __global float *obs_x, const __global float *obs_y)
{
    int i = get_global_id(0);
    if (i >= N)
        return;

    pf_sp sp = state[i];
    vsmc_threefry4x32 rng = rng_set[i];
    weight[i] = init_sp(&sp, &rng, obs_x[0], obs_y[0]);
    state[i] = sp;
    rng_set[i] = rng;
}

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

    pf_sp sp = state[i];
    vsmc_threefry4x32 rng = rng_set[i];
    weight[i] = move_sp(&sp, &rng, obs_x[t], obs_y[t]);
    state[i] = sp;
```

```
    rng_set[i] = rng;
}
```

B.2 PROCESS COMMAND LINE PROGRAM OPTIONS

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

int main(int argc, char **argv)
{
    int n;
    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;
}
```

B.3 DISPLAY PROGRAM PROGRESS

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

int main()
{
```

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

    return 0;
}
```