Yan Zhou

vSMC – Scalable Monte Carlo

Release v2.3.0

CONTENTS

LIST OF TABLES

1 INTRODUCTION

In this chapter, we introduce the kinds of Monte Carlo algorithms readily supported by the library and introduce the terminologies used through out this document.

1.1 RESAMPLING

1.2 MARKOV CHAIN MONTE CARLO

1.3 SEQUENTIAL MONTE CARLO

2 BASIC USAGE

2.1 CONVENTIONS

All classes and functions 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`. We will use "function" for referring to name space scope functions and "method" for class member functions. For brevity, we will omit the "`vsmc::`" qualifier in the remaining of this document except in showing complete examples.

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 and BLAS implementations. Intel Threading Building Blocks[2] (TBB), Intel Math Kernel Library[3] (MKL) and HDF5[4] are optional third-party libraries. One need to define the configuration macros `VSMC_HAS_TBB`, `VSMC_HAS_MKL` and `VSMC_HAS_HDF5`, respectively, to nonzero values before including any vSMC headers to make their existence known to the library.

2.3 CONCEPTS

The library is structured around a few core concepts. A sampler is responsible for running an algorithm. It contains a particle system and operations on it. A particle system is formed by the states $\{X^{(i)}\}_{i=1}^N$ and weights $\{W^{(i)}\}_{i=1}^N$. This system will also be responsible for resampling. All user defined operations are to be applied to the whole system. These are "initialization" and "moves" which are applied before resampling, and "MCMC" moves which are applied after resampling. These operations do not have to be MCMC kernels. They can be used for any purpose that suits the particular algorithm. Most statistical inferences requires calculation of $\sum_{i=1}^N W^{(i)}\varphi(X^{(i)})$ for

[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

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.

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.

The `StateMatrix` class template

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

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

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

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

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

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

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

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

2.3.2 *Weight*

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

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

There are a few methods for accessing the weights,

```
w.ess();          // Get {\normalfont\textsc{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 $W^{(i)} \propto v^{(i)}$
w.mul(v);         // Set $W^{(i)} \propto W^{(i)} v^{(i)}$
w.set_log(v);     // Set $\log W^{(i)} = v^{(i)} + \text{const.}$
w.add_log(v);     // Set $\log W^{(i)} = \log W^{(i)} + v^{(i)} + \text{const.}$
```

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

2.3.3 *Particle*

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

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

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

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

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

Resize the particle system

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

Resize by selecting according to a user supplied index vector

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

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

Resize by selecting according to a user supplied mask vector

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

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

Resize by resampling

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

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

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

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

Resize by uniformly selecting from all particles

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

This is equivalent to,

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

Resize by selecting a range of particles

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

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

2.3.4 *Single particle*

It is often easier to define a function $f(X^{(i)})$ than $f(X^{(1)},\dots,X^{(N)})$. 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 `SingeParticle<T>`. An object of this class is constructed from the index i of the particle, and a pointer to the particle system it belongs to,

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

or more conveniently,

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

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

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

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

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

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

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

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

is equivalent to

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

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

2.3.5 *Sampler*

A sampler can be constructed in a few ways,

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

constructs a sampler that is never resampled, while

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

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

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

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

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

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

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

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

For example,

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

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

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

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

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

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

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

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

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

2.3.6 *Monitor*

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

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

For example,

```
void eval(std::size_t iter, std::size_t d, Particle<T> &particle, double *r)
{
    for (std::size_t i = 0; i != particle.size(); ++i, r += dim) {
        auto sp = particle.sp(i);
        r[0] = /* $\varphi_1(X^{(i)})$ */;
        // ...
        r[d - 1] = /* $\varphi_d(X^{(i)})$ */;
    }
}
```

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

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

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

```
void eval(std::size_t iter, std::size_t d, Particle<T> &particle, double *r)
{
    r[0] = /* $\varphi_1(\{X^{(i)}\}_{i=1}^N)$ */;
```

```
        // ...
        r[d - 1] = /* $\varphi_d(\{X^{(i)}\}_{i=1}^N)$ */;
    }
```

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

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

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

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

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

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

2.4 A SIMPLE PARTICLE FILTER

2.4.1 *Model and algorithm*

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

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

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

```
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 $\ell$ is the log-likelihood function.
Iteration
    Set $t \leftarrow t+1$.
    Sample
        $X_{\text{pos}}^{(t,i)} \sim \mathcal{N}(X_{\text{pos}}^{(t-1,i)} + \Delta X_{\text{vel}}^{(t-1,i)}, 0.02)$    $X_{\text{vel}}^{(t,i)} \sim \mathcal{N}(X_{\text{vel}}^{(t-1,i)}, 0.001)$
        $Y_{\text{pos}}^{(t,i)} \sim \mathcal{N}(Y_{\text{pos}}^{(t-1,i)} + \Delta Y_{\text{vel}}^{(t-1,i)}, 0.02)$    $Y_{\text{vel}}^{(t,i)} \sim \mathcal{N}(Y_{\text{vel}}^{(t-1,i)}, 0.001)$
    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.

where

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

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

2.4.2 *Implementations*

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

The main program

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

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

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

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

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

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

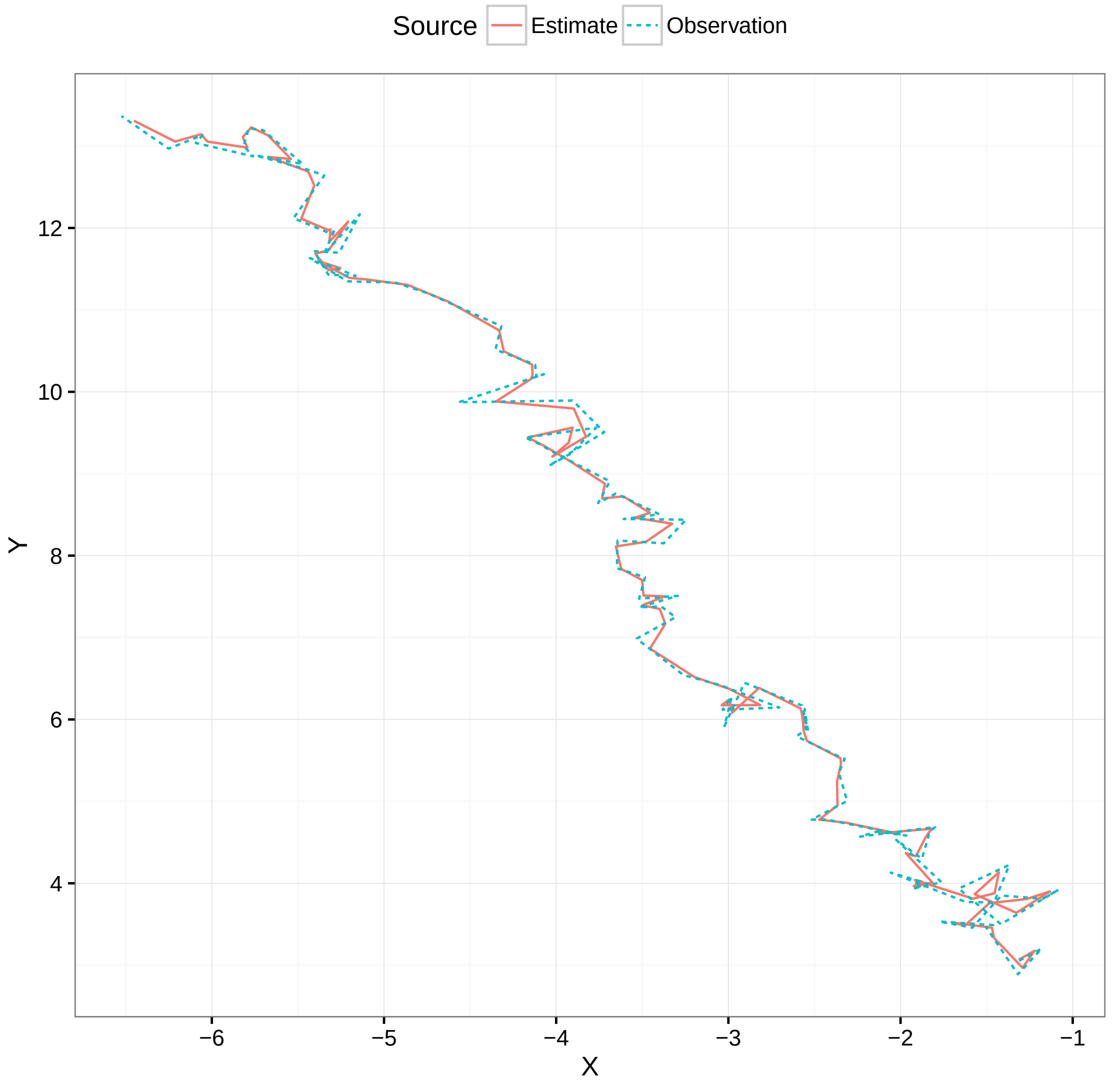

Figure 2.1 A simple particle filter

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

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

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

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

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 $\ell(X_t^{(i)}|Y_t)$
    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());
            weight_[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 `weight_` for storing weights computed later. Next, the main loop initializes each state component with the respective Gaussian distribution, computes the log-likelihood and store them in the vector allocated in the last step. This is done by calling the `eval_sp` method. After all particles have been initialized, we set the weights of the system in `eval_post`. Later in section 2.5, it will become clear why we structured the implementation this way.

Move: `PFMove`

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

```
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

3 ADVANCED USAGE

3.1 CLONING OBJECTS

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

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

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

```
auto new_sampler = sampler;
```

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

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

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

```
sampler = other_sampler;
```

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

3.2 CUSTOMIZING MEMBER TYPES

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

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

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

3.2.1 *Replacing* `weight_type`

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

```
w.ess();           // Get $\text{\normalfont\textsc{ess}}$
w.set_equal();     // Set $W^{(i)} = 1/N$
w.resample_size(); // Get the sample size $N$.
w.resample_data(); // Get a pointer to normalized weights
```

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

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

        return 1 / global;
    }

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

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

    const double *data()
    {
        return /* pointer to $\{W_r^{(i)}\}_{i=1}^{N_r}$ */;
    }

    const 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 $1 / \sum_{r=1}^R N_r$
        // Synchronization
    }

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

```
            // Broadcast $S$
            // Set $W_r^{(i)} = W_r^{(i)} / S$ 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, pass a pointer to this vector is passed to `T::copy`. Otherwise, a null pointer is passed to `T::copy`. Of course, the class `T` also needs to provide a suitable method `copy` that can handle the distributed system. By defining suitable `WeightMPI` and `T::copy`, the library can be extended to handle distributed systems.

3.3 EXTENDING `SingleParticle<T>`

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

```
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_; }
    int &u() { return u_; }

    private:
    double x_;
    int u_;
};

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

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

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

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

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

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

A second advantage is that it allows easier access to the raw data. Consider the implementation `PFEval` in section 2.4.2. It is rather redundant to copy each value of the two positions, just so later we can compute weighted sums from them. Recall that in section 2.3.6 we showed that a monitor that compute the final results directly can also be added to a sampler. Therefore, we might implement `PFEval` as the following,

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

And it can be added to a sampler as,

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

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

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. 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. These are listed in table 4.1. The second type has a prefix `VSMC_USE_`. These macros specify that a certain feature or third-party library can be used, if available. These are listed in table 4.2. 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. The third type changes the behavior of the library, for example by setting default value of some template parameter or the choosing among different implementations of a certain feature. They are listed in table 4.3.

Macro	Default	Description
`VSMC_HAS_INT128`	Platform dependent	Support for 128-bits integers
`VSMC_HAS_SSE2`	Platform dependent	Support for SSE2 intrinsic functions
`VSMC_HAS_AVX2`	Platform dependent	Support for AVX2 intrinsic functions
`VSMC_HAS_AES_NI`	Platform dependent	Support for AES-NI intrinsic functions
`VSMC_HAS_RDRAND`	Platform dependent	Support for RDRAND intrinsic functions
`VSMC_HAS_X86`	Platform dependent	Support for x86 platform
`VSMC_HAS_X86_64`	Platform dependent	Support for x86-64 platform
`VSMC_HAS_POSIX`	Platform dependent	Support for POSIX platform
`VSMC_HAS_OMP`	Platform dependent	Support for OpenMP 3.0 or higher
`VSMC_HAS_OPENCL`	`0`	Support for OpenCL 1.2 or higher
`VSMC_HAS_TBB`	`0`	Support for TBB 4.0 or higher
`VSMC_HAS_TBB_MALLOC`	`VSMC_HAS_TBB`	Support for TBB 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

Table 4.1 Configuration macros for feature availability

Macro	Default	Description
`VSMC_USE_OMP`	`VSMC_HAS_OMP`	Use OpenMP parallelization outside the SMP module
`VSMC_USE_TBB`	`VSMC_HAS_TBB`	Use TBB parallelization outside the SMP module
`VSMC_USE_TBB_TLS`	`VSMC_HAS_TBB`	Use TBB thread-local storage classes (TLS)
`VSMC_USE_MKL_CBLAS`	`VSMC_HAS_MKL`	Use `mkl_cblas.h` instead of `cblas.h`
`VSMC_USE_MKL_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.

Table 4.2 Configuration macros for feature usability

Macro	Default	Description
`VSMC_INT128`	Platform dependent	The 128-bit integer type
`VSMC_ALIGNMENT`	`32`	Default alignment for scalar types
`VSMC_ALIGNMENT_CACHE`	`64`	Default cache line alignment for scalar types
`VSMC_ALIGNMENT_MIN`	`16`	Minimum alignment for all types
`VSMC_ALIGNED_MEMORY_TYPE`	Platform dependent	The type of `AlignedMemory`
`VSMC_CONSTRUCT_SCALAR`	`0`	Should `Allocator::construct` zero out scalar types

Table 4.3 Configuration macros for library behavior

5 MATHEMATICAL OPERATIONS

5.1 USING THE BLAS LIBRARY

The library requires a working BLAS implementation. Internally, the library use the standard C interface, such as `cblas_dgemv`. Though this interface has been standardized for years, it is not universally available. If the `cblas.h` header is not available or the runtime library does not have the required functions defined, then one should define the configuration macro `VSMC_HAS_CBLAS` to zero. By default, this macro is defined to one if either `VSMC_USE_MKL_CBLAS` or `VSMC_USE_ACCELERATE` (see table 4.2) are non-zero. The user can also manually define this macro to zero if it is undesirable to include the C interface header.

When the C interface is not available, the library declares the standard Fortran subroutines in C itself, such as `dgemv_`. If there should be no underscore appending the function name, then one can define the macro `VSMC_BLAS_NAME_NO_UNDERSCORE`. Alternatively, one can define the macro `VSMC_BLAS_NAME` directly. For example, if the name C function name has an underscore in front of the Fortran subroutine name,

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

5.2 CONSTANTS

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

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

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

5.3 VECTORIZED OPERATIONS

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

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

```
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. Therefore, there are $2^3 - 1$ versions of this function for each type of the value. Note that, mixed precision is not allowed. For example,

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

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

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

Without any third-party libraries, these functions does not provide performance gain compared to the simple loop. When MKL or Apple Accelerate framework is present, some functions can have substantial performance improvement when all input arguments are vectors. Vectorized random number generating introduced later (section 7.1) performance heavily depends on these functions.

5.4 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,

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

Table 5.2 Arithmetic functions

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

Table 5.3 Power and root functions

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

Table 5.4 Exponential and logarithm functions

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

Table 5.5 Trigonometric functions

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

```
// 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 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
// 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]);
```

6.1 BUILTIN ALGORITHMS

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

The first step determines the statistical properties of the resampling algorithm. The library defines all algorithms discussed in Douc, Cappé, and Moulines (2005). Samplers can be constructed with builtin schemes as seen in section 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

`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

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

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

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

6.2 USER DEFINED ALGORITHMS

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

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

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

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

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

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

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

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

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

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

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

6.3 ALGORITHMS WITH INCREASING DIMENSIONS

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,

$$\varphi_t(X^{(i)}_{0:t}|X^{(i)}_{0:t-1}) = \varphi_t(X^{(i)}_t, \varphi_{t-1}(X^{(i)}_{0:t-1})),$$
$$\phi_t(X^{(i)}_{0:t}|X^{(i)}_{0:t-1}) = \phi_t(X^{(i)}_t, \varphi_{t-1}(X^{(i)}_{0:t-1}), \phi_{t-1}(X^{(i)}_{0:t-1})),$$

such that $q(\cdot|X^{(i)}_{0:t}) = q(\cdot|\varphi(X^{(i)}_{0:t-1}))$ and $W_t(X^{(i)}_{0:t}) = W_t(\phi(X^{(i)}_{0:t}))$. In this case, only the values of $\varphi_t(X^{(i)}_{0:t})$ and $\phi_t(X^{(i)}_{0:t})$ 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 $X^{(i)}_t \in \mathbb{R}$, and ϕ_t is the same as φ_t),

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

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

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

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

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

        return rs;
    }

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

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

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

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

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

7 RANDOM NUMBER GENERATING

The library has a comprehensive RNG system to facilitate implementation of Monte Carlo algorithms. Similar to the standard library header `<random>`, there are mainly two parts of this system. The first is a set of RNG engines that generate random integers. The other is a set of distribution generators. The former are documented in sections 7.2 to 7.4, and the later in section 7.6. Apart from these, the library also provides facilities for vectorized random number generating (section 7.1) and using multiple RNGs in parallel programs (section 7.5).

7.1 VECTORIZED RANDOM NUMBER GENERATING

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

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

The effect of the function call,

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

is equivalent to the loop,

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

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

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

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

For example,

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

This is similar to the following loop,

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

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

7.1.1 *Performance measurement*

All performance results shown later in this chapter is measured in single core cycles per bytes (cpB) for RNGs, or cycles per element (cpE) for distributions (`double` precision). The processor used to measure the performance is an Intel Core i7-4960HG CPU. Three compilers are tested. The LLVM clang (version 3.8), the GNU GCC (version 6.1), and the Intel C++ compiler (version 2016 update 2). When multiple compilers are tested, they are labeled "LLVM", "GNU", and "Intel", respectively in tables. If only results from one compiler are shown, then unless stated otherwise, it is the LLVM clang compiler. The operating system is Mac OS X (version 11.1). Depending on the CPU, the compiler, and the operating system, performance may vary considerably. The library does not attempt to optimize for any particular platform. However, for any given platform, the relative performance advantage to alternatives, such as the standard library, is usually substantial.

For RNG engines, we measure the performance of generating random bits instead of the raw output from the engines. All internal usages of RNGs in the library first transfer the raw output to unsigned integers uniform on the set $\{0, \dots, 2^W - 1\}$, $W \in \{32, 64\}$ (see section 7.6.1). Direct use of the raw output from RNG engines is rare in applications.

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

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

The second is the vectorized performance,

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

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

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

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

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

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

The third is the vectorized performance,

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

For all the three above, the RNG is `ARSx8` (section 7.2.1). The last is when RNG is `MKL_SFMT19937` (section 7.4),

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

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

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

7.2 COUNTER-BASED RNG

The standard library provides a set of RNG engines (performance data in table 7.1). Unfortunately, none of them are suitable for parallel computing without considerable efforts. To illustrate the problem, consider the situation where there are two threads that need to generate random numbers. And thus two RNG engine instances need to be created for each thread. Let the random numbers generated by them be $\{r_i^1\}_{i>0}$ and $\{r_i^2\}_{i>0}$. To ensure that $r_i^1 \ne r_i^2$ most of the time for $i > 0$, it is usually sufficient to set the seeds $s_1 \ne s_2$. However, this is not a sufficient condition for the two sequences not to overlap. It is entirely possible that, $r_i^1 = r_{i+k}^2$ for all $i > \max\{0, -k\}$ for some integer k even if s_1 and s_2 are chosen randomly. If $|k|$ is larger than or close to the total number of random numbers required, then this situation might not be an issue. However, there is no easy way to ensure such a condition, as long as the random number are generated with a recursion $y_i = f(y_{i-1})$.

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

```
std::mt19937 rng;

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

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

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

```
std::mt19937 rng;

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

	Loop			`rand`		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
`mt19937`	2.94	1.90	3.89	3.16	1.97	4.51
`mt19937_64`	1.63	1.40	1.89	1.89	1.34	1.87
`minstd_rand0`	5.04	5.49	6.79	5.06	5.53	6.26
`minstd_rand`	3.94	4.88	5.94	3.95	5.14	5.38
`ranlux24_base`	6.42	7.01	8.14	6.37	7.22	7.53
`ranlux48_base`	4.17	3.87	5.33	4.15	3.88	5.15
`ranlux24`	67.99	63.80	84.23	68.21	63.88	71.88
`ranlux48`	154.89	138.96	182.23	157.99	140.58	165.29
`knuth_b`	15.85	22.65	13.65	12.39	22.53	13.95

Table 7.1 Performance of standard library RNG

This not only requires managing the RNGs explicitly, but also knowing the number of threads at compile-time, which prevents it to be used in any applications using dynamic parallelization. And similar to the first solution, it cannot be used when threads are created implicitly.

The development by Salmon et al. (2011) made high performance parallel RNG much more accessible. The RNGs introduced in the paper use deterministic functions f_k, such that, for a sequence $\{c_i = i\}_{i\ge0}$, the sequence $\{y_i = f_k(c_i)\}_{i\ge0}$ appears random. In addition, for $k_1 \ne k_2$, f_{k_1} and f_{k_2} will generate two sequences that appear statistically independent. Compared to more conventional RNGs which use recursions $y_i = f_k(y_{i-1})$, these counter-based RNGs are much easier to setup in a parallelized environment. If c, the counter, is an unsigned integer with b bits, and k, the key, is an unsigned integer with d bits. Then for each k, the RNG has a period 2^b. And there can be at most 2^d independent streams. Table 7.2 lists all counter-based RNGs implemented in the library, along with the bits of the counter and the key. They all output 32-bits unsigned integers uniform on the set $\{0, \dots, 2^{32} - 1\}$. For 64-bits output, a suffix `_64` may be appended to the corresponding RNG engine names. For example, `Threefry4x64` and `Threefry4x64_64` both generate the same 256-bits random integers internally. The only difference is that `operator()` of the former returns 32 or those 256 bits each time it is executed, while the later returns 64 bits.

All RNGs in table 7.2 are actually type aliases. More generally the library defines the following class template as the interface,

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

Class	Counter bits	Key bits
`AES128x1`, `ARS128x2`, `AES128x4`, `AES128x8`	128	128
`AES192x1`, `ARS192x2`, `AES192x4`, `AES192x8`	128	192
`AES256x1`, `ARS256x2`, `AES256x4`, `AES256x8`	128	256
`ARSx1`, `ARS256x2`, `ARSx4`, `ARSx8`	128	128
`Philox2x32`	64	32
`Philox2x64`	128	64
`Philox4x32`	128	64
`Philox4x64`	256	128
`Threefry2x32`	64	64
`Threefry2x64`	128	128
`Threefry4x32`	128	128
`Threefry4x64`	256	256
`Threefry8x64`	512	512
`Threefry16x64`	1024	1024

Table 7.2 Counter-based RNG

where `ResultType` shall be an unsigned integer type and `Generator` is the class that actually implement the algorithm. See the reference manual for details of the generator type. For most users, those implemented in the library are sufficient. They are introduced in the next few sections. A few configuration macros of these generators are listed in table 7.3 and will be referred to later.

7.2.1 *AES-NI instructions based RNG*

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

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

The corresponding RNG engine is,

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

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

Table 7.3 Configuration macros for counter-based RNG

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

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

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

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

template <std::size_t Rounds>
```

```
using AES256KeySeq =
    internal::AESKeySeq<Rounds, internal::AES256KeySeqGenerator>;

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

and correspondingly four RNG engines,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The engine `ARS` is the library's default RNG if AES-NI instructions are supported. Aliases for block sizes 1, 2, 4 and 8 are defined for all four algorithms, as well as both 32- and 64-bits versions. These aliases are listed in table 7.2. The performance of these engines depends on a few factors, such as CPU types, compilers, operating systems, etc. See tables 7.4 to 7.7 for performance data. In any

	Loop			`rand`		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
AES128x1	1.05	5.57	4.56	0.77	4.29	4.04
AES128x2	1.25	3.28	3.00	0.70	2.74	2.50
AES128x4	0.85	1.74	1.75	0.65	1.31	1.38
AES128x8	1.22	1.26	1.48	0.85	0.85	0.84
AES128x1_64	0.86	5.74	4.41	0.76	4.56	4.03
AES128x2_64	0.80	2.62	2.80	0.70	2.40	2.50
AES128x4_64	0.75	1.55	1.63	0.65	1.31	1.38
AES128x8_64	1.15	1.11	1.26	0.85	0.83	0.84

Table 7.4 Performance of `AES128Engine`

case, the performance is good enough even for the most demanding applications. The library does not attempt to optimize the algorithm for any particular platform. In realistic applications, the performance of RNG is unlikely to become a bottle neck. Note that, the best performance is obtained with the vectorized `rand` function (see section 7.1).

7.2.2 *Philox*

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

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

The corresponding RNG engine is,

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

	Loop			rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
AES192x1	1.73	5.64	5.20	0.90	4.99	4.65
AES192x2	1.44	3.09	3.62	0.82	2.63	2.77
AES192x4	0.98	1.90	1.88	0.76	1.46	1.52
AES192x8	1.31	1.42	1.44	0.94	0.99	0.93
AES192x1_64	1.09	5.54	5.25	0.89	4.98	4.63
AES192x2_64	0.91	2.84	3.10	0.82	2.62	2.77
AES192x4_64	0.88	1.69	1.77	0.76	1.45	1.52
AES192x8_64	1.24	1.22	1.32	0.94	0.95	0.93

Table 7.5 Performance of `AES192Engine`

	Loop			rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
AES256x1	1.97	6.90	5.78	1.10	5.61	5.19
AES256x2	1.32	3.38	3.53	0.95	2.95	3.05
AES256x4	1.13	2.15	2.03	0.88	1.70	1.68
AES256x8	1.40	1.42	1.57	1.02	1.01	1.02
AES256x1_64	1.36	6.88	5.61	1.09	5.69	5.23
AES256x2_64	1.06	3.24	3.40	0.95	3.00	3.05
AES256x4_64	1.02	1.85	1.93	0.88	1.63	1.68
AES256x8_64	1.33	1.40	1.44	1.04	1.08	1.02

Table 7.6 Performance of `AES256Engine`

	Loop			rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
ARSx1	0.70	3.57	3.32	0.47	2.88	2.50
ARSx2	0.95	2.17	2.35	0.44	1.58	1.73
ARSx4	0.62	1.41	1.44	0.40	0.96	1.00
ARSx8	0.63	1.06	1.09	0.34	0.62	0.69
ARSx1_64	0.54	2.99	3.15	0.46	2.46	2.49
ARSx2_64	0.47	2.07	2.20	0.43	1.59	1.73
ARSx4_64	0.47	1.31	1.31	0.40	0.96	1.00
ARSx8_64	0.50	0.94	0.95	0.34	0.61	0.69

Table 7.7 Performance of `ARSEngine`

The default vector length and the number of rounds can be changed by configuration macros listed in table 7.3. There is no limit on the template parameter `K` or `Rounds`, nor any limitation on `T` except that it has to be an unsigned integer type. See Salmon et al. (2011) on the most general form of the algorithm. However, the library only provides default constants for 32- and 64-bits unsigned integer type `T` and `K` taking the values `2` or `4`. These four engines are defined as type aliases for convenience,

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

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

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

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

Type aliases for 32- and 64-bits `ResultType` are also defined, as listed in table 7.2. To use the engine with `K` taking values larger than four, or `T` being unsigned integer type with bits other than 32 or 64, one needs to provide a suitable trait class, `Constant`. It is similar to that of `ARSEngine`. See the

	Loop			`rand`		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
`Philox2x32`	7.03	10.97	3.41	5.93	9.15	2.17
`Philox4x32`	2.88	14.12	5.04	1.76	4.89	1.83
`Philox2x64`	1.86	3.20	2.46	0.98	1.97	1.48
`Philox4x64`	1.69	4.15	5.87	0.92	1.85	1.08
`Philox2x32_64`	6.86	11.15	7.85	5.93	9.35	5.97
`Philox4x32_64`	2.70	13.03	4.87	1.76	4.54	2.64
`Philox2x64_64`	1.63	3.11	3.22	1.00	2.00	2.33
`Philox4x64_64`	1.53	4.01	6.13	0.92	1.81	1.54

Table 7.8 Performance of `PhiloxEngine`

reference manual of `PhiloxConstants` for an example of how to define it. The performance data is in table 7.8.

7.2.3 *Threefry*

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

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

The corresponding RNG engine is,

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

The default vector length and the number of rounds can be changed by configuration macros listed in table 7.3. Similar to the implementation of the Philox algorithm, there is no limit on the template parameter `K` or `Rounds` as long as a suitable trait class `Constant` is provided. The library provides default constants for 64-bits unsigned integer type `T` and `K` taking the values `4`, `8` and `16`, taken from the skein[3] hash algorithm, for which the Threefish algorithm was originally developed for. Defaults for 32-bits `T` or `K` taking the value `2` are also provided, taken from Salmon et al. (2011). Type aliases for these configurations are defined for convenience,

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

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

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

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

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

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

Type aliases for 32- and 64-bits `ResultType` are also defined, as listed in table 7.2. The performance data is in table 7.9.

7.2.4 *Default* RNG

Note that, not all RNGs implemented by the library is available on all platforms. The library also defines two type aliases `RNG` and `RNG_64`, which are one of the RNGs listed in table 7.2. The preference is in the order listed in table 7.10. The user can define the configuration macro `VSMC_RNG_TYPE` to override the choice made by the library.

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

	Loop			rand		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
Threefry2x32	9.79	9.78	8.90	8.63	8.98	7.64
Threefry4x32	6.51	6.11	7.73	5.76	5.51	6.89
Threefry2x64	5.54	3.73	3.65	4.65	3.04	2.63
Threefry4x64	3.53	2.19	3.87	3.02	1.86	3.36
Threefry8x64	3.48	1.96	2.67	2.38	1.61	2.24
Threefry16x64	2.91	5.38	7.32	2.52	4.97	6.86
Threefry2x32_64	9.47	3.59	8.84	8.67	9.35	8.01
Threefry4x32_64	6.32	5.54	7.51	5.81	5.32	6.87
Threefry2x64_64	5.03	3.64	2.74	4.65	3.13	2.26
Threefry4x64_64	3.35	2.09	3.69	3.00	1.86	3.35
Threefry8x64_64	2.92	1.80	2.54	2.38	1.56	2.20
Threefry16x64_64	2.81	5.27	7.10	2.52	4.93	6.83

Table 7.9 Performance of `ThreefryEngine`

7.2.5 *Seeding counter-based RNG*

The singleton class template `SeedGenerator` can be used to generate distinctive seeds 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 two generators, `SeedGenerator<T1>` and `SeedGenerator<T2>`, will be independent. The second parameter is the type of the seed values. It can be any unsigned integer type. Classes such as `Particle<T>` will use the generator of the following type,

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

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

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

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

Table 7.10 Default RNG

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

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

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

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

7.3 NON-DETERMINISTIC RNG

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

	Loop			`rand`		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
`RDRAND16`	124.42	130.83	116.91	122.97	130.84	116.18
`RDRAND32`	59.14	64.43	60.02	57.67	63.33	59.04
`RDRAND64`	34.06	32.20	30.45	32.10	32.14	28.99

Table 7.11 Performance of non-deterministic RNG

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

Table 7.12 MKL RNG

RDRAND instruction may not return a random integer at all. The RNG engine will keep trying until it succeeds. One can limit the maximum number of trials by defining the configuration macro `VSMC_RNG_RDRAND_NTRIAL_MAX`. A value of zero, the default, means the number of trials is unlimited. If it is a positive number, and if after the specified number of trials no random integer is return by the RDRAND instruction, zero is returned. The performance data is in table 7.11.

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 engines. They are listed in table 7.12. Note that, MKL RNGs perform the best when they are used to generate vectors of random numbers. These wrappers use a buffer to store such vectors. And thus they have much larger state space than usual RNGs. Each RNG engines output by default 32-bits integers. Similar to the counter-based RNGs, 64-bits variants are also defined. The performance data is in table 7.13.

	Loop			`rand`		
RNG	LLVM	GNU	Intel	LLVM	GNU	Intel
`MKL_MCG59`	1.44	0.78	0.98	0.40	0.33	0.32
`MKL_MCG59_64`	0.74	0.67	0.87	0.39	0.34	0.32
`MKL_MT19937`	1.32	0.65	0.87	0.36	0.28	0.27
`MKL_MT19937_64`	0.63	0.55	0.76	0.32	0.29	0.27
`MKL_MT2203`	1.33	0.67	0.86	0.27	0.21	0.19
`MKL_MT2203_64`	0.57	0.57	0.75	0.22	0.22	0.19
`MKL_STMT19937`	1.20	0.70	0.85	0.18	0.19	0.17
`MKL_STMT19937_64`	0.58	0.52	0.74	0.19	0.17	0.17
`MKL_NONDETERM`	31.51	31.70	29.55	30.68	31.86	29.21
`MKL_NONDETERM_64`	30.80	31.98	29.90	31.37	31.82	29.46
`MKL_ARS5`	1.28	0.86	1.02	0.30	0.32	0.30
`MKL_ARS5_64`	0.79	0.74	0.91	0.35	0.32	0.30
`MKL_PHILOX4X32X10`	1.70	1.10	1.24	0.61	0.60	0.55
`MKL_PHILOX4X32X10_64`	0.96	0.97	1.12	0.58	0.57	0.54

Table 7.13 Performance of MKL RNG

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 $\mathcal{O}(N)$. Many of the counter-based RNGs have small state size and thus for moderate

N, this cost is not an issue. The method calls `rng_set[i]` and `rng_set[j]` return independent RNGs if $i \neq j$. This implementation has the advantage that the behavior of an algorithm can be entirely deterministic even when the scheduling of parallel execution is dynamic, since each sample has its own RNG.

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

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

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

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

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

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

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

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

    private:
    Vector<RNGType> rng_;
};

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

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

7.6 DISTRIBUTIONS

The library provides implementations of some common distributions. Some of them are the same as those in the standard library, with `CamelCase` names. For example, `NormalDistribuiton` can be used as a drop-in replacement of `std::normal_distribuiton`. This includes all of the continuous distributions defined in the standard library. As stated in section 7.1, all the distributions defined in the library support vectorized random number generating. In the following sections we introduce each distributions included in the library.

7.6.1 *Uniform bits distribution*

The class template,

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

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

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

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

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

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

7.6.2 *Standard uniform distribution*

All continuous distributions are built upon the standard uniform distribution. And thus the performance and quality of the algorithm transferring random integers to random floating point numbers on the set $[0, 1]$ are of critical importance. The library provides five distributions, listed in table 7.14. They are all class template with a single template type parameter `RealType`, which is the floating point result type. For each distribution, the random integers produced by RNGs are transferred to 32- or 64-bits intermediate random integers through `UniformBitsDistribution` before they are further converted to floating numbers. The integer type depends on `RealType`, the range of the integers produced by the RNG, R, and the configuration macros `VSMC_RNG_U01_USE_64BITS_DOUBLE`. The exact relations are listed in table 7.15. In the remaining of this section, let W be the number of bits of the intermediate random integers, and M be the number of significant bits (including the implicit one) of `RealType`. We also denote the input random integers as U and the output random real numbers as X.

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

Table 7.14 Standard uniform distributions

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

Table 7.15 Intermediate integer types of standard uniform distributions

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

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

The minimum and maximum are 0 and 1, respectively.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.6.3 *Distributions using the inverse method*

Table 7.16 lists all the distributions implemented with the inverse method. The performance data of these distributions is in table 7.17. Note that, in this and other tables in the remaining of this section, we shortened the class name for brevity. For example, in the table the Cauchy distribution is listed as `Cauchy` while the full declaration of the class template is,

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

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

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

Table 7.16 Distributions using the inverse method

Distribution	STD/Boost	vSMC	`rand`	MKL
`Cauchy(0,1)`	80.74	44.39	13.68	7.33
`Exponential(1)`	67.02	58.97	7.71	5.90
`ExtremeValue(0,1)`	109.14	64.75	12.66	11.12
`Laplace(0,1)`	61.07	44.86	9.34	8.92
`Logistic(0,1)`	–	41.37	13.50	13.14
`Pareto(1,1)`	–	54.57	12.05	10.77
`Rayleigh(1)`	–	73.65	11.48	9.00
`UniformReal(-0.5,0.5)`	17.16	5.05	2.84	1.26
`Weibull(1,1)`	137.81	28.78	7.76	6.30

Table 7.17 Performance of distributions using the inverse method

Distribution	STD/Boost	vSMC	`rand`	MKL
`Levy(0,1)`	–	51.08	22.55	18.85
`Lognormal(0,1)`	86.62	74.47	18.12	13.15
`Normal(0,1)`	66.80	57.42	13.97	9.94

Table 7.18 Performance of Normal and related distributions

7.6.4 *Normal and related distribution*

The class template

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

implements the Normal distribution with PDF,

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

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

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

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

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

The performance data of these distributions is in table 7.18.

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

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

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

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

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

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

In either case, the parameter `mean` is the mean vector. If it is a null pointer, then it is assumed to be a zero vector. The parameter `chol` is a $d(d+1)/2$-vector, where d is the dimension. The vector is the lower triangular elements of the Cholesky decomposition of the covariance matrix, packed row by row. Libraries such as LAPACK has routines to compute such a matrix. Alternatively, one can use the covariance functionalities in the library. See section 8.2, which also provides a concrete example of using the multivariate Normal distribution.

7.6.5 *Gamma and related distribution*

The class template

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

implements the Gamma distribution with PDF,

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

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

```
template <typename RealType>
class ChiSquaredDistribution;

template <typename RealType>
class FisherFDistribution;

template <typename RealType>
class StudentTDistribution;
```

Distribution	STD/Boost	vSMC	`rand`	MKL
`Gamma(1,1)`	46.91	40.75	7.77	6.38
`Gamma(0.1,1)`	116.09	108.48	34.42	32.18
`Gamma(0.5,1)`	149.06	143.56	58.99	46.79
`Gamma(0.7,1)`	157.89	151.58	45.00	33.95
`Gamma(0.9,1)`	155.27	126.67	34.64	26.02
`Gamma(1.5,1)`	194.77	128.22	35.70	26.63
`Gamma(15,1)`	175.60	125.80	33.22	24.33

Table 7.19 Performance of Gamma distribution

Distribution	STD/Boost	vSMC	`rand`	MKL
`ChiSquared(2)`	47.40	43.50	7.76	6.34
`ChiSquared(0.2)`	118.34	113.04	34.45	32.12
`ChiSquared(1)`	148.01	147.80	60.26	46.66
`ChiSquared(1.4)`	157.34	157.19	45.12	34.06
`ChiSquared(1.8)`	156.95	131.37	34.75	26.11
`ChiSquared(3)`	194.27	132.80	35.48	26.91
`ChiSquared(30)`	175.42	132.15	33.87	24.74

Table 7.20 Performance of χ^2 distribution

They implement the χ^2-distribution, the Fisher's F-distribution, and the Student's t-distribution, respectively. The performance data of these distributions, for different parameters are in tables 7.19 to 7.22.

7.6.6 *Beta distribution*

The class template

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

Distribution	STD/Boost	vSMC	rand	MKL
FisherF(1,1)	297.83	321.61	141.49	112.18
FisherF(1,0.5)	270.66	300.03	126.03	104.85
FisherF(1,1.5)	303.26	341.12	136.33	97.66
FisherF(1,3)	339.16	317.82	112.51	91.63
FisherF(1,30)	316.72	313.00	106.89	87.74
FisherF(0.5,1)	271.46	301.39	126.14	105.44
FisherF(0.5,0.5)	244.30	280.94	109.63	96.42
FisherF(0.5,1.5)	278.05	323.27	119.76	89.43
FisherF(0.5,3)	308.79	294.08	95.27	82.27
FisherF(0.5,30)	291.52	295.48	91.44	80.04
FisherF(1.5,1)	300.82	331.97	134.41	97.18
FisherF(1.5,0.5)	275.38	311.72	119.38	89.55
FisherF(1.5,1.5)	305.81	349.47	128.79	79.49
FisherF(1.5,3)	338.16	327.47	105.71	74.71
FisherF(1.5,30)	320.67	321.94	100.56	70.67
FisherF(3,1)	340.75	313.84	111.08	90.55
FisherF(3,0.5)	315.19	291.68	95.90	82.92
FisherF(3,1.5)	349.25	320.84	105.00	74.05
FisherF(3,3)	379.47	309.88	80.60	66.62
FisherF(3,30)	352.89	311.23	75.35	63.08
FisherF(30,1)	321.07	311.52	107.43	87.30
FisherF(30,0.5)	295.64	293.28	91.73	80.41
FisherF(30,1.5)	327.78	323.60	101.41	70.91
FisherF(30,3)	355.12	307.18	76.58	64.13
FisherF(30,30)	339.48	304.53	70.56	59.71

Table 7.21 Performance of Fisher's F-distribution

Distribution	STD/Boost	vSMC	`rand`	MKL
`StudentT(2)`	102.81	128.86	36.56	29.79
`StudentT(0.2)`	171.87	193.97	64.68	60.00
`StudentT(1)`	209.09	232.05	96.15	76.90
`StudentT(1.4)`	217.53	250.00	96.01	62.82
`StudentT(1.8)`	218.20	222.99	68.56	52.72
`StudentT(3)`	238.16	223.01	64.46	52.83
`StudentT(30)`	217.51	219.29	59.04	49.43

Table 7.22 Performance of Student's t-distribution

implements the Beta distribution with PDF,

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

The specific algorithm used depends on the parameters. If $\alpha = 1/2$ and $\beta = 1/2$, or $\alpha = 1$ or $\beta = 1$, then the inverse method is used. If $\alpha > 1$ and $\beta > 1$, the method in Cheng (1978) is used. Otherwise, let $K = 0.852$, $C = -0.956$, and $D = \beta + K\alpha^2 + C$. If $\alpha < 1$, $\beta < 1$ and $D \le 0$, then Jöhnk's method (Devroye 1986, sec. 3.5) is used. In all other cases, one of the switching algorithms in Atkinson (1979) is used. Note that, there is no vectorized implementation at the moment for the switching algorithms. In other cases, the vectorized generating shall provide considerable speedup. The performance data is in table 7.23

Distribution	STD/Boost	vSMC	`rand`	MKL
`Beta(0.5,0.5)`	343.80	39.44	13.78	36.59
`Beta(1,1)`	87.06	11.82	2.66	8.95
`Beta(1,0.5)`	216.83	58.51	13.96	8.94
`Beta(1,1.5)`	375.56	56.12	15.41	8.98
`Beta(0.5,1)`	216.61	57.83	13.06	8.97
`Beta(1.5,1)`	368.19	54.64	15.15	8.90
`Beta(1.5,1.5)`	593.39	155.82	47.11	38.40
`Beta(0.3,0.3)`	305.74	131.63	52.68	33.30
`Beta(0.9,0.9)`	354.11	151.48	152.37	49.41
`Beta(1.5,0.5)`	456.83	179.56	179.85	59.94
`Beta(0.5,1.5)`	456.66	181.78	180.55	59.41

Table 7.23 Performance of Beta distribution

8 UTILITIES

The library provides some utilities for writing Monte Carlo simulation programs. For some of them, such as command line option processing, there are more advanced, dedicated libraries out there. The library only provides some basic functionality that is sufficient for 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 = AlignmentTrait<T>::value,
    typename Memory = AlignedMemory>
class Allocator;
```

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

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

The method `aligned_malloc` shall behave similar to `std::malloc` with the additional alignment requirement. It shall return a null pointer if it fails to allocate memory. In any other case, including zero input size, it shall return a reachable non-null pointer. The method `aligned_free` shall behave similar to `std::free`. It shall be able to handle a null pointer as its input. The library provides 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. Two container types are defined for convenience,

Class	Implementation	Availability
`AlignedMemoryTBB`	TBB memory allocation functions	`VSMC_HAS_TBB_MALLOC`
`AlignedMemorySYS`	Operating system libraries	`VSMC_HAS_POSIX`, or using MSVC
`AlignedMemoryMKL`	MKL memory allocation functions	`VSMC_HAS_MKL`
`AlignedMemorySTD`	Standard library	Always available

Table 8.1 Aligned memory allocation

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

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

8.2 SAMPLE COVARIANCE

The library provides some basic functionality to estimate sample covariance through the following class template,

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

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

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

It computes the sample covariance matrix Σ,

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

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

`layout` The storage layout of sample matrix x. It is assumed to be an n by p matrix.
`n` The number of samples.
`p` The dimension of each sample.
`x` The sample matrix. If it is a null pointer, then no computation is carried out.
`w` The weight vector. If it is a null pointer, then all samples are assigned weight 1.
`mean` Output storage of the mean. If it is a null pointer, then it is ignored.
`cov` Output storage of the covariance matrix. If it is a null pointer, then it is ignored.
`cov_layout` The storage layout of the covariance matrix.
`cov_upper` If `true`, then the upper triangular of the covariance matrix is packed, otherwise the lower triangular is packed. Ignored if `cov_pack` is `false`.
`cov_packed` If `true`, then the covariance matrix is packed.

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

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

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

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

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 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 8.2 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 8.2 lists the classes defined by the library and their corresponding OpenCL pointers.

8.5 PROCESS COMMAND LINE PROGRAM OPTIONS

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

```
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.

8.6 DISPLAY PROGRAM PROGRESS

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

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

When invoked, the program output something similar the following,

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

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

8.7 STOP WATCH

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

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

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

BIBLIOGRAPHY

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

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

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

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

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

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

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

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

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

A INTERFACING WITH C AND OTHER LANGUAGES

The library provides a set of C functions, declared in the header `vsmc.h`, and a companion runtime library. These functions expose a subset of the functionality of the C++ template library. The main purpose is for easier interfacing with other languages, instead of implementing algorithms in C itself. Of course, the later is possible and two complete C99 implementations of the simple particle filter in section 2.4 are shown in appendices B.1.3 and B.1.4. See the reference manual for complete list of the API. In this appendix, we introduce the conventions used in the API.

A.1 ARGUMENT TYPES

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

```
unsigned get();
```

is exposed to C as

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

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

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

A.2 CLASSES AND METHODS

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

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

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

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

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

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

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

corresponds to the C++ calls,

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

The memory can later be freed by

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

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

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

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

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

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

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

are translated to C as

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

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

Table a.1 C API types

Feature	Notes
SMP backends	Section 2.5
Resamling algorithms	Section 6.1
Ordered uniform random numbers	Section 6.2
Seeding	Section 7.2.5
Vectorized random number generating	Section 7.6
Aligned memory allocation	Section 8.1
Covariance matrix	Section 8.2
Process program command line options	Section 8.5
Display program progress	Section 8.6
Stop watch	Section 8.7
Registering C++11 RNG as MKL BRNG	Reference manual
Random walk	Reference manual

Table a.2 Features accessible from C

B SOURCE CODE OF COMPLETE PROGRAMS

B.1 A SIMPLE PARTICLE FILTER

B.1.1 *Sequential implementation*

```
#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);
        double x;
        double y;
        while (data >> x >> y) {
            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)
    {
        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.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());
        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> 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);
        double x;
        double y;
        while (data >> x >> y) {
            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 size_t PosX = 0;
static const size_t PosY = 1;
static const size_t VelX = 2;
static const size_t VelY = 3;

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

// Storage for data
static size_t pf_obs_size = 0;
static double *pf_obs_x = NULL;
static double *pf_obs_y = NULL;

// Temporaries used by pf_init and pf_move
static double *pf_pos_x = NULL;
static double *pf_pos_y = NULL;
static double *pf_vel_x = NULL;
static double *pf_vel_y = NULL;
static double *pf_weight = NULL;

static inline double pf_log_likelihood(
    size_t t, const vsmc_single_particle *sp)
{
    double llh_x = 10 * (sp->state[PosX] - pf_obs_x[t]);
    double llh_y = 10 * (sp->state[PosY] - pf_obs_y[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");
    size_t 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;
    }
    pf_obs_size = n;
    pf_obs_x = vsmc_malloc(n * sizeof(double), 32);
    pf_obs_y = vsmc_malloc(n * sizeof(double), 32);
    fseek(data, 0, SEEK_SET);
    for (size_t i = 0; i < n; ++i) {
        fscanf(data, "%lg", &pf_obs_x[i]);
        fscanf(data, "%lg", &pf_obs_y[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 size_t size = vsmc_particle_size(particle);
    pf_pos_x = vsmc_malloc(size * sizeof(double), 32);
    pf_pos_y = vsmc_malloc(size * sizeof(double), 32);
    pf_vel_x = vsmc_malloc(size * sizeof(double), 32);
    pf_vel_y = vsmc_malloc(size * sizeof(double), 32);
    pf_weight = vsmc_malloc(size * sizeof(double), 32);
    vsmc_rand_normal(rng, size, pf_pos_x, 0, sd_pos);
    vsmc_rand_normal(rng, size, pf_pos_y, 0, sd_pos);
    vsmc_rand_normal(rng, size, pf_vel_x, 0, sd_vel);
    vsmc_rand_normal(rng, size, pf_vel_y, 0, sd_vel);
}

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

    pf_normal(particle, 2, 1);

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

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

    return 0;
}

static inline size_t pf_move(size_t t, vsmc_particle particle)
{
    pf_normal(particle, sqrt(0.02), sqrt(0.001));

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

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

    return 0;
}

static inline void pf_eval(
    size_t t, size_t dim, vsmc_particle particle, double *r)
{
    const size_t size = vsmc_particle_size(particle);
    for (size_t 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)
{
    size_t n = 10000;
    if (argc > 1)
        n = (size_t) atoi(argv[1]);

    vsmc_sampler sampler = vsmc_sampler_new(n, 4, vSMCMultinomial, 0.5);

    vsmc_sampler_init(sampler, pf_init, 0);
    vsmc_sampler_move(sampler, pf_move, 0);

    vsmc_monitor pos = vsmc_monitor_new(2, pf_eval, 0, vSMCMonitorMCMC);
    vsmc_sampler_set_monitor(sampler, "pos", pos);
    vsmc_monitor_delete(&pos);

    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_size - 1);
    vsmc_stop_watch_stop(watch);
    printf("Time (ms): %lg\n", vsmc_stop_watch_milliseconds(watch));
    vsmc_stop_watch_delete(&watch);

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

    vsmc_sampler_delete(&sampler);
    vsmc_free(pf_obs_x);
    vsmc_free(pf_obs_y);
    vsmc_free(pf_pos_x);
    vsmc_free(pf_pos_y);
    vsmc_free(pf_vel_x);
    vsmc_free(pf_vel_y);
    vsmc_free(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 size_t PosX = 0;
static const size_t PosY = 1;
static const size_t VelX = 2;
static const size_t VelY = 3;

// Storage for data
static size_t pf_obs_size = 0;
static double *pf_obs_x = NULL;
static double *pf_obs_y = NULL;

// Temporaries used by pf_init and pf_move
static double *pf_pos_x = NULL;
static double *pf_pos_y = NULL;
static double *pf_vel_x = NULL;
static double *pf_vel_y = NULL;
static double *pf_weight = NULL;

static inline double pf_log_likelihood(
    size_t t, const vsmc_single_particle *sp)
{
    double llh_x = 10 * (sp->state[PosX] - pf_obs_x[t]);
    double llh_y = 10 * (sp->state[PosY] - pf_obs_y[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");
    size_t 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;
    }
    pf_obs_size = n;
    pf_obs_x = vsmc_malloc(n * sizeof(double), 32);
    pf_obs_y = vsmc_malloc(n * sizeof(double), 32);
    fseek(data, 0, SEEK_SET);
    for (size_t i = 0; i < n; ++i) {
        fscanf(data, "%lg", &pf_obs_x[i]);
        fscanf(data, "%lg", &pf_obs_y[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 size_t size = vsmc_particle_size(particle);
    pf_pos_x = vsmc_malloc(size * sizeof(double), 32);
    pf_pos_y = vsmc_malloc(size * sizeof(double), 32);
    pf_vel_x = vsmc_malloc(size * sizeof(double), 32);
    pf_vel_y = vsmc_malloc(size * sizeof(double), 32);
    pf_weight = vsmc_malloc(size * sizeof(double), 32);
    vsmc_rand_normal(rng, size, pf_pos_x, 0, sd_pos);
    vsmc_rand_normal(rng, size, pf_pos_y, 0, sd_pos);
    vsmc_rand_normal(rng, size, pf_vel_x, 0, sd_vel);
    vsmc_rand_normal(rng, size, pf_vel_y, 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 size_t pf_init_sp(vsmc_single_particle sp)
{
    sp.state[PosX] = pf_pos_x[sp.id];
    sp.state[PosY] = pf_pos_y[sp.id];
    sp.state[VelX] = pf_vel_x[sp.id];
    sp.state[VelY] = pf_vel_y[sp.id];
    pf_weight[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, 1);
}
```

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

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

    return 0;
}

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

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

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

    vsmc_sampler_init_smp_type pf_init = {
        pf_init_sp, pf_init_param, pf_init_pre, pf_init_post};
    vsmc_sampler_move_smp_type pf_move = {
        pf_move_sp, pf_move_pre, pf_move_post};
    vsmc_monitor_eval_smp_type pf_eval = {pf_eval_sp, NULL, NULL};

    vsmc_sampler sampler = vsmc_sampler_new(n, 4, vSMCMultinomial, 0.5);

    vsmc_sampler_init_smp(vSMCBackendTBB, sampler, pf_init, 0);
    vsmc_sampler_move_smp(vSMCBackendTBB, sampler, pf_move, 0);

    vsmc_monitor pos =
        vsmc_monitor_new_smp(vSMCBackendTBB, 2, pf_eval, 0, vSMCMonitorMCMC);
    vsmc_sampler_set_monitor(sampler, "pos", pos);
    vsmc_monitor_delete(&pos);

    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_size - 1);
    vsmc_stop_watch_stop(watch);
    printf("Time (ms): %lg\n", vsmc_stop_watch_milliseconds(watch));
    vsmc_stop_watch_delete(&watch);

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

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

```
#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;
}
```