vSMC – PARALLEL SEQUENTIAL MONTE CARLO IN C++

Yan Zhou

Version 2.1.0

CONTENTS

abstract

Sequential Monte Carlo is a family of algorithms for sampling from a sequence of distributions. Some of these algorithms, such as particle filters, are widely used in the physics and signal processing researches. More recent developments have established their application in more general inference problems such as Bayesian modeling.

These algorithms have attracted considerable attentions in recent years as they admit natural and scalable parallelization. However, these algorithms are perceived to be difficult to implement. In addition, parallel programming is often unfamiliar to many researchers though conceptually appealing, especially for sequential Monte Carlo related fields. A C++ template library is presented for the purpose of implementing general sequential Monte Carlo algorithms on parallel hardware.

1 INTRODUCTION

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

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

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

2 SEQUENTIAL MONTE CARLO

2.1 SEQUENTIAL IMPORTANCE SAMPLING AND RESAMPLING

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

$$\mathbb{E}_\pi[\varphi(X)] = \int \varphi(x)\pi(x)\,\mathrm{d}\,x = \int \frac{\varphi(x)\pi(x)}{\eta(x)}\eta(x)\,\mathrm{d}\,x = \mathbb{E}_\eta\Big[\frac{\varphi(X)\pi(X)}{\eta(X)}\Big] \tag{1}$$

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

$$\hat{\varphi}_1 = \frac{1}{N}\sum_{i=1}^N \frac{\varphi(X^{(i)})\pi(X^{(i)})}{\eta(X^{(i)})} \tag{2}$$

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

$$\hat{\varphi}_2 = \frac{\sum_{i=1}^N w^{(i)}\varphi(X^{(i)})}{\sum_{i=1}^N w^{(i)}} \tag{3}$$

or

$$\hat{\varphi}_3 = \sum_{i=1}^N W^{(i)}\varphi(X^{(i)}) \tag{4}$$

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

Sequential importance sampling (SIS) generalizes the importance sampling technique for a sequence of distributions $\{\pi_t\}_{t\ge0}$ defined on spaces $\{\prod_{k=0}^t E_k\}_{t\ge0}$. At time $t = 0$, sample $\{X_0^{(i)}\}_{i=1}^N$ from η_0 and compute the weights $W_0^{(i)} \propto \pi_0(X_0^{(i)})/\eta_0(X_0^{(i)})$. At time $t \ge 1$, each sample $X_{0:t-1}^{(i)}$, usually termed *particles* in the literature, is extended to $X_{0:t}^{(i)}$ by a proposal distribution $q_t(\cdot|X_{0:t-1}^{(i)})$. And the weights are recalculated by $W_t^{(i)} \propto \pi_t(X_{0:t}^{(i)})/\eta_t(X_{0:t}^{(i)})$ where

$$\eta_t(X_{0:t}^{(i)}) = \eta_{t-1}(X_{0:t-1}^{(i)})q_t(X_{0:t}^{(i)}|X_{0:t-1}^{(i)}) \tag{5}$$

and thus

$$\begin{aligned} W_t^{(i)} \propto \frac{\pi_t(X_{0:t}^{(i)})}{\eta_t(X_{0:t}^{(i)})} &= \frac{\pi_t(X_{0:t}^{(i)})\pi_{t-1}(X_{0:t-1}^{(i)})}{\eta_{t-1}(X_{0:t-1}^{(i)})q_t(X_{0:t}^{(i)}|X_{0:t-1}^{(i)})\pi_{t-1}(X_{0:t-1}^{(i)})} \\ &= \frac{\pi_t(X_{0:t}^{(i)})}{q_t(X_{0:t}^{(i)}|X_{0:t-1}^{(i)})\pi_{t-1}(X_{0:t-1}^{(i)})}W_{t-1}^{(i)} \end{aligned} \tag{6}$$

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

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

$$\mathbb{E}\Big[\sum_{i=1}^M \bar{W}_t^{(i)}\varphi_t(\bar{X}_{0:t}^{(i)})\Big] = \mathbb{E}\Big[\sum_{i=1}^N W_t^{(i)}\varphi_t(X_{0:t}^{(i)})\Big] \tag{7}$$

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

$$\text{ESS}_t = \frac{1}{\sum_{i=1}^{N}(W_t^{(i)})^2} \tag{8}$$

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

The common practice of resampling is to replicate particles with large weights and discard those with small weights. In other words, instead of generating a random sample $\{\bar{X}_{0:t}^{(i)}\}_{i=1}^N$ directly, a random sample of integers $\{R^{(i)}\}_{i=1}^N$ is generated, such that $R^{(i)} \ge 0$ for $i = 1, \dots, N$ and $\sum_{i=1}^N R^{(i)} = N$. And each particle value $X_{0:t}^{(i)}$ is replicated for $R^{(i)}$ times in the new particle system. The distribution of $\{R^{(i)}\}_{i=1}^N$ shall fulfill the requirement of Equation (7). One such distribution is a multinomial distribution of size N and weights $(W_t^{(i)}, \dots, W_t^{(N)})$. See Douc, Cappé, and Moulines (2005) for some commonly used resampling algorithms.

2.2 SMC SAMPLERS

SMC samplers allow us to obtain, iteratively, collections of weighted samples from a sequence of distributions $\{\pi_t\}_{t\ge0}$ over essentially any random variables on some spaces $\{E_t\}_{t\ge0}$, by constructing a sequence of auxiliary distributions $\{\tilde{\pi}_t\}_{t\ge0}$ on spaces of increasing dimensions, $\tilde{\pi}_t(x_{0:t}) = \pi_t(x_t)\prod_{s=0}^{t-1} L_s(x_{s+1}, x_s)$, where the sequence of Markov kernels $\{L_s\}_{s=0}^{t-1}$, termed backward kernels, is formally arbitrary but critically influences the estimator variance. See Del Moral, Doucet, and Jasra (2006b) for further details and guidance on the selection of these kernels.

Standard sequential importance sampling and resampling algorithms can then be applied to the sequence of synthetic distributions, $\{\tilde{\pi}_t\}_{t\ge0}$. At time $t-1$, assume that a set of weighted particles $\{W_{t-1}^{(i)}, X_{0:t-1}^{(i)}\}_{i=1}^N$ approximating $\tilde{\pi}_{t-1}$ is available, then at time t, the path of each particle is extended with a Markov kernel say, $K_t(x_{t-1}, x_t)$ and the set of particles $\{X_{0:t}^{(i)}\}_{i=1}^N$ reach the distribution $\eta_t(X_{0:t}^{(i)}) = \eta_0(X_0^{(i)})\prod_{k=1}^t K_t(X_{t-1}^{(i)}, X_t^{(i)})$, where η_0 is the initial distribution of the particles. To correct the discrepancy between η_t and $\tilde{\pi}_t$, Equation (6) is applied and in this case,

$$W_t^{(i)} \propto \frac{\tilde{\pi}_t(X_{0:t}^{(i)})}{\eta_t(X_{0:t}^{(i)})} = \frac{\pi_t(X_t^{(i)})\prod_{s=0}^{t-1} L_s(X_{s+1}^{(i)}, X_s^{(i)})}{\eta_0(X_0^{(i)})\prod_{k=1}^t K_t(X_{t-1}^{(i)}, X_t^{(i)})} \propto \tilde{w}_t(X_{t-1}^{(i)}, X_t^{(i)})W_{t-1}^{(i)} \tag{9}$$

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

$$\tilde{w}_t(X_{t-1}^{(i)}, X_t^{(i)}) = \frac{\pi_t(X_t^{(i)})L_{t-1}(X_t^{(i)}, X_{t-1}^{(i)})}{\pi_{t-1}(X_{t-1}^{(i)})K_t(X_{t-1}^{(i)}, X_t^{(i)})} \tag{10}$$

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

$$w_t(X_{t-1}^{(i)}, X_t^{(i)}) = \frac{\gamma_t(X_t^{(i)})L_{t-1}(X_t^{(i)}, X_{t-1}^{(i)})}{\gamma_{t-1}(X_{t-1}^{(i)})K_t(X_{t-1}^{(i)}, X_t^{(i)})} \tag{11}$$

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

$$\frac{\hat{Z}_t}{Z_{t-1}} = \sum_{i=1}^N W_{t-1}^{(i)} w_t(X_{t-1}^{(i)}, X_t^{(i)}) \tag{12}$$

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

$$L_{t-1}(x_t, x_{t-1}) = \frac{\pi(x_{t-1})K_t(x_{t-1}, x_t)}{\pi_t(x_t)} \tag{13}$$

is used, the unnormalized incremental weights will be

$$w_t(X_{t-1}^{(i)}, X_t^{(i)}) = \frac{\gamma_t(X_{t-1}^{(i)})}{\gamma_{t-1}(X_{t-1}^{(i)})}. \tag{14}$$

2.3 OTHER SEQUENTIAL MONTE CARLO ALGORITHMS

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

3 BASIC USAGE

3.1 CONVENTIONS

All classes that are accessible to users are within the name space `vsmc`. Class names are in `CamelCase` and function names, free or class methods, are in `small_cases`. In the remaining of this guide, we will omit the `vsmc::` name space qualifiers.

Concept	Class
Weight, $\{W^{(i)}\}_{i=1}^N$	`Weight`
State, $\{X^{(i)}\}_{i=1}^N$	`T`, user defined
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 1 Core concepts of the library

3.2 GETTING AND INSTALLING THE LIBRARY

The library is hosted at GitHub[1]. One can download the stable releases[2] or get the development branch from the GitHub repository. This is a header only C++ template library. To install the library just move the contents of the `include` directory into a proper place, e.g., `/usr/local/include` on Unix-alike systems. This library requires working C++11, BLAS and LAPACK implementations. Standard C interface headers for the later two (`cblas.h` and `lapacke.h`) are required[3]. Intel Threading Building Blocks[4] (TBB), Intel Math Kernel Library[5] (MKL) and HDF5[6] are optional third-party libraries. One need to define the configuration macro `VSMC_HAS_TBB`, `VSMC_HAS_MKL` and `VSMC_HAS_HDF5` to nonzero values before including any vSMC headers to make their existence known to the library, respectively. Doxygen[7] generated reference manual can be access online[8].

3.3 CONCEPTS

The library is based on a few core concepts. A sampler is responsible for running an algorithm. It is composed by a particle system and operations on it. A particle system is formed by the states $\{X^{(i)}\}_{i=1}^N$ and

[1] `https://github.com/zhouyan/vSMC`

[2] `https://github.com/zhouyan/vSMC/releases`

[3] If MKL is used, its `mkl_cblas.h` and `mkl_lapacke.h` headers will be used

[4] `https://www.threadingbuildingblocks.org`

[5] `https://software.intel.com/en-us/intel-mkl`

[6] `http://www.hdfgroup.org`

[7] `http://www.stack.nl/~dimitri/doxygen/`

[8] `http://zhouyan.github.io/vSMCDoc/master/`

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" which are applied after resampling[9]. Most statistical inferences are done through $\sum_{i=1}^N W^{(i)}\varphi(X^{(i)})$. This can be carried out along each sampler iteration by a monitor. Table 1 summarizes these concepts and the corresponding classes in the library. Each of them are introduced in detail in the following sections.

3.3.1 *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 is acceptable. For example,

```
class State
{
  public:
  State(std::size_t N);
};
```

How the state values are actually stored and accessed are entirely up to the user.

For most applications, the states can be stored within an N by d matrix, where d is the dimension of the state. Let X_{ij} denote the value of the state of the ith particle at coordinate j. In this case, the library provides a convenient class template,

```
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 X_{ij}. The following constructs an object of this class,

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

where `Dynamic` is just a enumerator with value zero. We can specify the dimension at runtime through the method call `sm.resize_dim(d)`. Note that, if the template parameter `Dim` is positive, then the call results in a compile-time error. To access X_{ij}, one can call the method `sm.state(i, j)`. The method call `sm.data()` returns a pointer to the beginning of the matrix. If `Layout` is `RowMajor`, then the method call `sm.row_data(i)` returns a pointer to the beginning of row i. If `Layout` is `ColMajor`, then the method call

[9] These operations do not have to be MCMC kernels. They can be used for any purpose that suites the particular algorithm.

sm.col_data(j) returns a pointer to the begninning of column j. These methods facilitate the interfacing with numerical libraries, such as BLAS. There are a few additional methods for accessing the state values. See the reference manual for details.

3.3.2 *Weight*

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

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

There are a few methods for accessing the weights,

```
w.ess();         // Get ESS = $1/\sum_{i=1}^{N}(W^{(i)})^2$
w.set_equal(); // Set $W^{(i)} = 1/N$
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 $\log W^{(i)} = v^{(i)} + const.$
w.add_log(v);  // Set $\log W^{(i)} = \log W^{(i)} + v^{(i)} + const.$
```

where the argument v is an input iterator which can be advanced at least N times. The method call 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.

3.3.3 *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 call particle.value() returns the type T object, and particle.weight() returns the type Weight object. They are constructed with the same integer value N when the above constructor is invoked.

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

```
void function(double &x)
{
  std::mt19937 rng;
  std::normal_distribution<double> rnorm;
  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;
  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 1
std::mt19937 rng2(s2); // For thread 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 call `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,

```
particle.rng(i); // Call from thread i
particle.rng(j); // Call from thread j
```

If $i \neq j$, then the above calls are guaranteed to be thread-safe. If TBB is available to the library, then it is also thread-safe even if $i = j$. In addition, each instance of the RNG generates independent streams. Therefore, one can write functions that process each particle, for example,

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

The details of the RNG system are documented later in section 7.

3.3.4 *Single particle*

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

```
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(); // A reference to the Particle<T> object
sp.rng();      // => sp.particle().rng(sp.id());
```

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

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

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

3.3.5 *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 method. Other resampling schemes are also possible, see section 6. Last, one can also construct a sampler that is only resampled when ESS $< \alpha N$, $\alpha \in [0,1]$, by the following,

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

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. More advanced constructors will be discussed in section 6

The method call `sampler.particle()` returns a reference to the particle system. A sampler can be initialized by user defined object that is convertible to the following type,

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

For example,

```
auto init = [](Particle<T> &particle, void *param) { /* Process 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 function `init` will be called and the argument `param` will be passed to it.

Similarly, after initialization, at each iteration, the particle system can be manipulated by users given callable objects that is convertible to the following types,

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

Multiple moves can be added to a sampler. The call `sampler.move(move, append)` adds a `move_type` object to the sampler, where `append` is a boolean value. If it is false, the call will clear any moves that were added before. If it is true, then `move` is appended to the end of a 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 it, the sequence of MCMC moves will be called. If it desired no to perform mutations during initialization, then following can be used,

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

The above snippet 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();
```

3.3.6 *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. One can define callable object that is convertible to

```
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, 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, second its dimension, and the third the evaluation function. Then 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 call has two optional arguments. First is a boolean value `record_only`. If it is true, it is assumed that no summation is needed. For example,

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

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,

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

Details can be found in section 8.3.

3.4 a simple particle filter

3.4.1 *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, $X_t = (X^t_{\mathrm{pos}}, Y^t_{\mathrm{pos}}, X^t_{\mathrm{vel}}, Y^t_{\mathrm{vel}})^T$. Imperfect observations $Y_t = (X^t_{\mathrm{obs}}, Y^t_{\mathrm{obs}})^T$ of the positions are possible at each time instance. The state and observation equations are linear with additive noises,

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

where

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

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

3.4.2 *Implementations*

We first show the main program.

```
Initialization
    Set $t \leftarrow 0$.
    Sample $X^{(0,i)}_{\text{pos}}, Y^{(0,i)}_{\text{pos}} \sim \mathcal{N}(0,4)$ and $X^{(0,i)}_{\text{vel}}, Y^{(0,i)}_{\text{vel}} \sim \mathcal{N}(0,1)$.
    Weight $W^{(i)}_0 \propto \exp \ell(X^{(i)}_0 | Y_0)$ where $\ell$ is the likelihood function.
Iteration
    Set $t \leftarrow t + 1$.
    Sample
            $X^{(t,i)}_{\text{pos}} \sim \mathcal{N}(X^{(t-1,i)}_{\text{pos}} + \Delta X^{(t-1,i)}_{\text{vel}}, 0.02)$        $X^{(t,i)}_{\text{vel}} \sim \mathcal{N}(X^{(t-1,i)}_{\text{vel}}, 0.001)$
            $Y^{(t,i)}_{\text{pos}} \sim \mathcal{N}(Y^{(t-1,i)}_{\text{pos}} + \Delta Y^{(t-1,i)}_{\text{vel}}, 0.02)$        $Y^{(t,i)}_{\text{vel}} \sim \mathcal{N}(Y^{(t-1,i)}_{\text{vel}}, 0.001)$
    Weight $W^{(i)}_t \propto W^{(i)}_{t-1} \exp \ell(X^{(i)}_t | Y_t)$.
Repeat the Iteration step until all data are processed.
```

Algorithm 1 Particle filter algorithm for the almost constant velocity model.

```cpp
#include "pf.hpp"

int main()
{
    constexpr std::size_t N = 1000; // Number of particles
    constexpr std::size_t n = 100;  // Number of data points
    vsmc::Sampler<PFState> sampler(N, vsmc::Multinomial, 0.5);
    sampler.init(PFInit()).move(PFMove(), false).monitor("pos", 2, PFMEval());
    sampler.initialize(const_cast<char *>("pf.data")).iterate(n - 1);

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

    return 0;
}
```

A `Sampler<PFState>` object is constructed first. Then the initialization `PFInit`, move `PFMove` and a monitor `PFMEval` 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[10] script that can be used to process this

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

pdf("pf.pdf")
plot(obs[,1], obs[,2], xlab = "X", ylab = "Y")
lines(pf$pos.0, pf$pos.1)
dev.off()
```

The print statement shows the first five lines of the output,

```
Size Resampled Accept.0       ESS    pos.0   pos.1
1 1000         1        0   1.54765 -1.41510 3.03569
2 1000         1        0 159.69800 -1.22141 3.15391
3 1000         1        0 182.29200 -1.29763 2.95484
4 1000         1        0  17.90400 -1.48292 3.28054
5 1000         1        0 252.25700 -1.48763 3.43379
```

The column Size shows the sample size at each iteration. The library does not provide direct support of changing the sample size. However, it is possible and an example is shown in section 6. The column Resampled shows nonzero values if resampling were performed and zero otherwise. For each moves and MCMC steps, an acceptance count will be recorded. In this particular example, it is irrelevant. Next the column ESS shows the value of ESS. The last two columns show the importance sampling estimates of the positions recorded by the monitor named "pos". The graphical representation of the output is shown in figure 1.

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

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

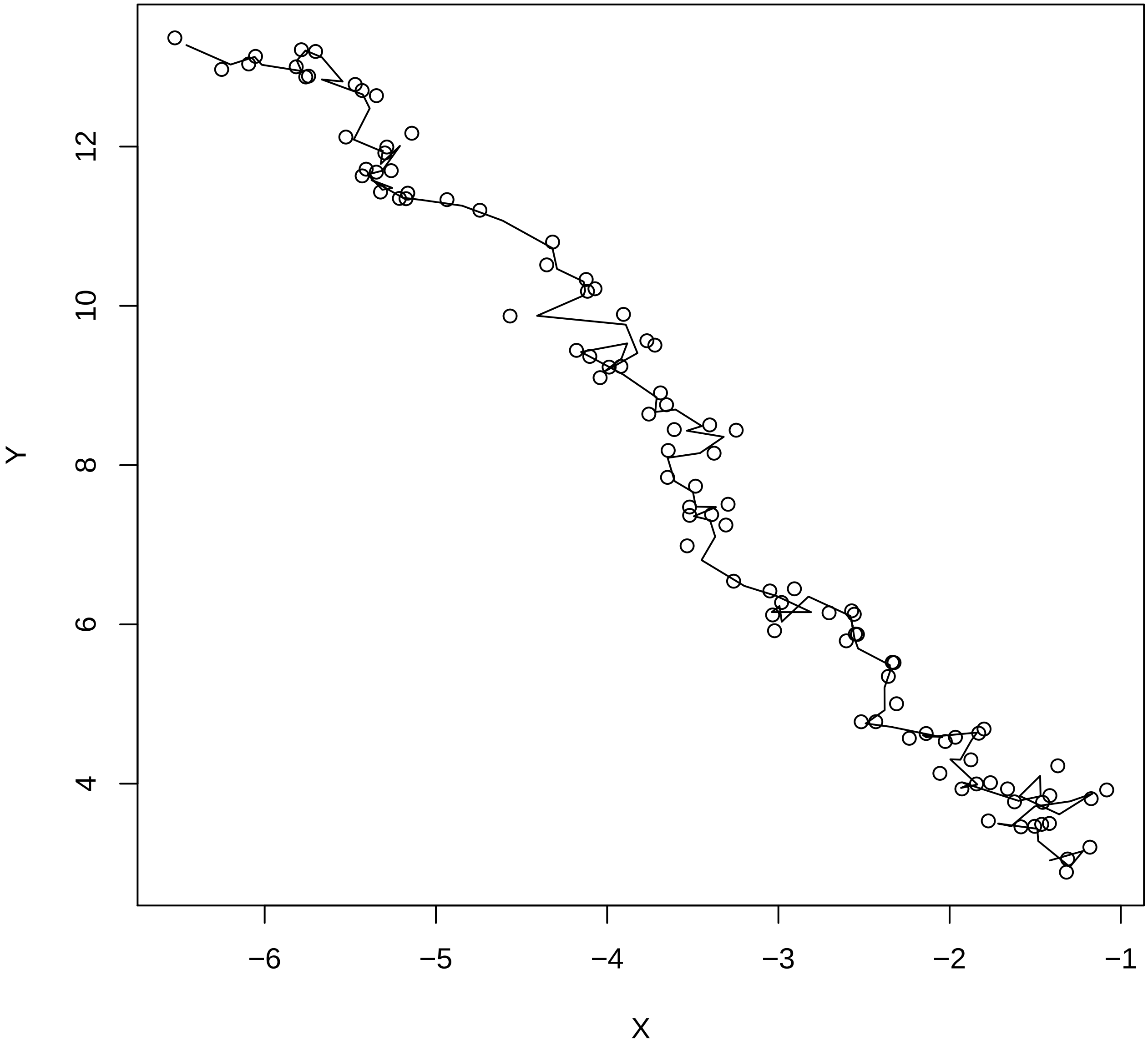

Figure 1 A simple particle system

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 definition of the class `PFState`

```
class PFState : public vsmc::StateMatrix<vsmc::RowMajor, 4, double>
{
    public:
    using base = vsmc::StateMatrix<vsmc::RowMajor, 4, double>;

    PFState(base::size_type N) : base(N) {}

    double log_likelihood(std::size_t iter, size_type i) const
    {
        double llh_x = 10 * (this->state(i, PosX) - obs_x_[iter]);
        double llh_y = 10 * (this->state(i, PosY) - obs_y_[iter]);
        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 *file)
    {
        if (!file)
            return;

        constexpr std::size_t n = 100; // Number of data points
        obs_x_.resize(n);
        obs_y_.resize(n);
        std::ifstream data(file);
        for (std::size_t i = 0; i != n; ++i)
            data >> obs_x_[i] >> obs_y_[i];
        data.close();
    }

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

The method `log_likelihood` accepts the iteration number (starting from zero at initialization) and the particle number as its input. It returns the value of $\ell(X^{(i)})$. The method `read_data` read the data input member data.

Initialization: `PFInit` The initialization step is implemented as below.

```
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 (std::size_t i = 0; i != particle.size(); ++i)
            acc += eval_sp(particle.sp(i));
        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)
    {
        w_.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());
        w_[sp.id()] = sp.particle().value().log_likelihood(0, sp.id());

        return 0;
    }

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

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

An object of this class is convertible to `Sampler<PFState>::init_type`. After initialization each state component with the respective Gaussian distribution, we compute the log-likelihood and store them in a vector. The class template `vsmc::Vector` is very similar to `std::vector`. See section 8.1 for details. After all particles have been initialized, we set the weights of the system.

The main method, `operator()` calls a few other methods to perform the tasks. Later in section 3.5 it will become clear why we structured the implementation this way.

Move: `PFMove` The move step is similar to the initialization. The implementation is as below,

```
class PFMove
{
    public:
    std::size_t operator()(std::size_t iter, vsmc::Particle<PFState> &particle)
    {
        eval_pre(iter, particle);
        std::size_t acc = 0;
        for (std::size_t i = 0; i != particle.size(); ++i)
            acc += eval_sp(iter, particle.sp(i));
        eval_post(iter, particle);

        return 0;
    }
    void eval_pre(std::size_t iter, vsmc::Particle<PFState> &particle)
    {
        w_.resize(particle.size());
    }

    std::size_t eval_sp(std::size_t iter, 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.state(PosX) += norm_pos(sp.rng()) + 0.1 * sp.state(VelX);
        sp.state(PosY) += norm_pos(sp.rng()) + 0.1 * sp.state(VelY);
        sp.state(VelX) += norm_vel(sp.rng());
        sp.state(VelY) += norm_vel(sp.rng());
        w_[sp.id()] = sp.particle().value().log_likelihood(iter, sp.id());

        return 0;
    }

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

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

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

```
class PFMEval
{
    public:
    void operator()(std::size_t iter, std::size_t dim,
        vsmc::Particle<PFState> &particle, double *r)
    {
        eval_pre(iter, particle);
        for (std::size_t i = 0; i != particle.size(); ++i, r += dim)
```

```
            eval_sp(iter, dim, particle.sp(i), r);
        eval_post(iter, particle);
    }

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

    void eval_sp(std::size_t iter, 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 iter, vsmc::Particle<PFState> &particle) {}
};
```

3.5 Symmetric Multiprocessing

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

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

And remove operator() from their implementations. After these changes, the implementation will be parallelized using TBB. 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. 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.

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.

3.5.1 *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 PFMEval : public MonitorEvalTBB<PFState, PFMEval>;
```

The second template argument of the base class need to be exactly the same as the derived class. For interested users, this is called Curiously Recurring Template Pattern[11] (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.

4 ADVANCED USAGE

5 MATHEMATICAL OPERATIONS

5.1 CONSTANTS

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

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

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

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

Table 2 Mathematical constants. Note: All functions are prefixed by `const_`.

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

5.2 VECTORIZED OPERATIONS

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

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

performs addition for vectors. If the input a and b are pointers to length n arrays, and the output y is a

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

Table 3 Vectorized mathematical operations

pointer to array of the same length, then this function call compute $y_i = a_i + b_i$ for $i = 1, \dots, n$. Either a and b can also be scalars. For example, if b is a scalar, then the operation performed is $y_i = a_i + b$. The functions defined are listed in table 3. For each function, the first parameter is always the length of the vector n, and the last is a pointer to the output y (except `sincos` which has two output pointers y and z). For all functions, the output is always a vector. If there are more than one input pointer, then some of them, but not all, can be scalars.

6 RESAMPLE

7 RANDOM NUMBER GENERATING

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

7.1 SEEDING

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

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

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

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

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

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

```
std::ifstream seed_txt("seed.txt");
if (seed_txt.good())
seed_txt >> Seed::instance(); // Read seed from a file
else
Seed::instance().set(101);    // The default seed
seed_txt.close();
// The program
std::ofstream seed_txt("seed.txt");
seed_txt << Seed::instance();     // Write the seed to a file
seed_txt.close();
```

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

A single seed generator is enough for a single computer program. However, it is more difficult to ensure that each computing node has a distinctive set of seeds in a distributed system. A simple solution is to use the `modulo` method of `SeedGenerator`. For example,

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

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

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

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

7.2 COUNTER-BASED RNG

The standard library provides a set of RNG classes. Unfortunately, none of them are suitable for parallel computing, at least without considerable user input. In addition, only the `std::mt19937` and `std::mt19937_64` have both high performance and desirable statistical properties.

The development by Salmon et al. (2011) made high performance parallel RNG much more accessible than it was before. In the author's personal opinion, it is the most significant development for parallel Monte Carlo algorithms in recent memory. See the paper for more details. Here, it is sufficient to mention that, the RNG introduced in the paper use a deterministic function f_k, such that, for any sequence $\{c_i\}_{i>0}$, the sequence $\{y_i\}_{i>0}$, $y_i = f_k(c_i)$, appears as random. In addition, for $k_1 \ne k_2$, f_{k_1} and f_{k_2} will generate two sequences that appear statistically independent. Compared to more conventional RNGs which use recursions $y_i = f(y_{i-1})$, these counter-based RNGs are much easier to setup in a parallelized environment.

If c, the counter, is an unsigned integer with b bits, and k, the key, is an unsigned integer with d bits.

Then for each k, the RNG has a period 2^b. And there can be at most 2^d independent streams. Table 4 lists all counter-based RNGs implemented in this library, along with the bits of the counter and the key. They all conform to the C++11 uniform RNG concept. All RNGs in Salmon et al. (2011) are implemented along with a few additions. Note that, the actual period of an RNG can be longer. For example, `Philox4x64` has a 256-bits counter but output 64-bits integers. And thus it has a 2^{1024} period. Such period length may seems very small compared to many well known RNGs. For example, the famous Mersenne-Twister generator (`std::mt19937`) has a period $2^{19937} - 1$. However, combined with 2^{256} independent streams, only the most demanding programs will find these counter-base RNGs insufficient.

Note that, not all RNGs defined by the library is available on all platforms. The library also defines a type alias `RNG` which is one of the RNGs listed in table 4. More specifically, if the AES-NI instructions are supported,

```
using RNG = ARS_4x32;
```

otherwise if AVX2 instructions are supported,

```
using RNG = Threefry4x32AVX2;
```

otherwise if SSE2 instructions are supported,

```
using RNG = Threefry4x32SSE2;
```

and last, on all other platforms,

```
using RNG = Threefry4x32;
```

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

7.3 NON-DETERMINISTIC RNG

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

7.4 MKL RNG

The MKL library provides some high performance RNGs. The library implement a wrapper class `MKLEngine` that makes them accessible as C++11 generators. They are listed in table 5. Note that, MKL RNGs performs best when they are used to generate vectors of random numbers. These wrappers use a buffer to store such vectors. And thus they have much bigger state space than usual RNGs.

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

Table 5 MKL RNG. Note: all classes can have a suffix _64.

7.5 MULTIPLE RNG STREAMS

Earlier in section 3.3.3 we introduced that `particle.rng(i)` returns an independent RNG instance. This is actually done through a class template called `RNGSet`. Three of them are implemented in the library. They all have the same interface,

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

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

The second implementation is `RNGSetVector`. It is an array of RNGs with length N. It has memory cost $O(N)$. Many of the counter-based RNGs have small state size and thus for moderate N, this cost is not an issue. The method calls `rng_set[i]` and `rng_set[j]` return independent RNGs if $i \neq j$.

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

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

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

Table 6 Random number distributions. Note: all class names have a suffix Distribution which is omitted in the table

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

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

will make the RNG set used by Particle<T> replaced by the user defined type. Alternatively, one can only replace the RNG type, for example

```
class T
{
    using rng_type = AES256_4x64;
};
```

will make the Particle<T> to use RNGSet<AES256_4x64> instead of RNGSet<RNG>, the default.

7.6 DISTRIBUTIONS

The library also provides implementations of some common distributions. They all conforms to the C++11 random number distribution concepts. Some of them are the same as those in the C++11 standard library, with CamelCase names. For example, NormalDistribuiton can be used as an drop-in replacement for std::normal_distribuiton. This includes all of the continuous distributions defined in the standard library. Their benefits compared to the standard library will be discussed later. Table 6 lists all the additional distributions implemented.

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

```
double mean[2] = { /* the mean vector */ };
double cov[4] = { / * the covariance matrix */ };
double chol[3];
double r[2];
// Compute the lower triangular of the Cholesky decomposition
cov_chol(2, cov, chol);
RNG rng;
NormalMVDistribution<double, 2> norm2(mean, chol); // Bivariate Normal
NormalMVDistribution<double, Dynamic> normd(2, mean, chol); // Same as above
norm2(rng, r); // Generate a bivariate Normal
normd(rng, r); // Same as above
```

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

7.7 VECTORIZED RANDOM NUMBER GENERATING

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

```
std::size_t n = 1000;
RNG rng;
NormalDistribution<double> norm(0, 1);
Vector<RNG::result_type> u(n);
Vector<double> r(n);
rng(n, u.data());           // Generate n random unsigned integers
rng_rand(rng, n, u.data()); // Same as above
norm(rng, n, r.data());     // Generate n Normal random numbers
normal_distribution(rng, n, r.data(), 0.0, 1.0);     // Same as above
normal_distribution(rng, n, r.data(), norm.param()); // Same as above
rng_rand(rng, norm, n, r.data());                    // Same as above
```

Note that these functions will be specialized to use MKL routines if rng is one of the engines listed in table 5.

7.8 RANDOM WALK

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 more simple cases.

8.1 ALIGNED MEMORY ALLOCATION

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

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

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

8.2 SAMPLE COVARIANCE ESTIMATING

8.3 STORING OBJECTS IN HDF5

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

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

create a HDF5 file with the sampler stored as a list. In R it can be processed as the following,

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

This creates a `data.frame` similar to that shown in section 3.4.2. Other types of objects can also be store, see the reference manual for details.

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

Table 7 RAII classes for MKL pointers

8.4 RAII CLASSES FOR MKL POINTERS

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

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

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

8.5 PROGRAM OPTIONS

The library provides some basic support of processing command line program options. Here is a minimal example

```
#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");
    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;
}
```

When invoked as the following,

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

The console output is as the following,

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

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

8.6 PROGRAM PROGRESS

Sometime it is desirable to see how much progress of a program has been made. The library provide a `Progress` class for this purpose. Below is a minimal example,

```
#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;
}
```

When invoked, the program output something similar the below

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

The method call `progress.start(n * n)` starting the printing of the progress. The argument specifies how many iterations there will be before it is stopped. The method call `progress.message(ss.str())` direct the program to print a message. This is optional. Each time after we finish *n* iterations, we increment the progress count by calling `progress.increment()`. And aftre everything is finished, the method `progress.stop()` is called.

8.7 x86 SIMD OPERATIONS

8.8 TIMING

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

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

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

REFERENCES

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

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

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

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

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

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

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

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

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

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

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