Software practical — Uncertainty Quantification

Implementation of a C++ Particle Filter Library

Winter semester 2019

Submitted by: **Nils Friess**

March 5, 2020

Supervisors: Prof. Robert Scheichl, Gianluca Detommaso

Heidelberg University

Introduction

In this report we present a Particle Filter library written in C++. Before discussing the actual implementation we give the theoretical details of the following sections.

We remark here that the naming in these methods is ambiguous and varies from author to author and even in different publications of the same authors. We use – with some exceptions – the naming and notation used by Doucet and Johansen in [5] and highlight parts where the naming differs from other publications.

Mathematical Formulation of the Model

A Particle Filter is a Sequential Monte Carlo (SMC) method[1] that is used to estimate the state of a system that changes over time using only noisy and/ or partial observations of the system's state. This is done in a Bayesian framework where one attempts to construct the posterior probability density function (pdf) of the state based on the observations. We make the following assumptions:

(A1) A model describing the initial state and the evolution of the internal state in time is available in a probabilistic form.

(A2) A model that relates the observations to the internal state is available in a probabilistic form.

(A3) The observations are only available sequentially, not as a batch (i. e. , we assume that we receive new measurements sequentially in time).

Due to (A3) we aim at a recursive method that does neither require to store nor to reprocess all the previous information when a new observation becomes available. To formalise the first two assumptions we will use the notion of *hidden Markov models*.

Such models consist of the triplet

$$\begin{aligned} X_0 &\sim p(x_0), \\ X_n \mid (X_{n-1} = x_{n-1}) &\sim p(x_n \mid x_{n-1}), \\ Y_n \mid (X_n = x_n) &\sim p(y_n \mid x_n), \end{aligned}$$

where

- $n \in \mathbb{N}$ denotes discrete time;

[1] Some authors use the terms *Particle Filter* and *SMC method* synonymously. Doucet and Johansen develop in [5] a framework in which Particle Filters are only one specific method in the much broader class of SMC methods. They argue that this distinction allows for a better understanding of these methods. In this report, we are only interested in the *filtering problem* and we will introduce it without discussing the more general notion of SMC methods as given by Doucet and Johansen.

- X_n is the d_x-dimensional state of the system taking values in $\mathbb{R}^{d_x}$;
- $p(x_0)$ is the prior probability density function (pdf) of the system's state;[2]
- Y_n is the d_y-dimensional vector of observations which is assumed to be conditionally independent of all other observations given the state X_n;
- $p(y_n \mid x_n)$ is the conditional pdf of Y_n given $X_n = x_n$.

Assumptions (A1) and (A2) then state that all these pdfs are known. Our goal is now to estimate the distribution $p(x_n \mid y_{1:n})$, where $y_{1:n} := (y_1, y_2, \dots, y_n)$. This is often referred to as the *filtering problem* or *tracking*.[3] In a restrictive set of cases this distribution can be computed exactly (e. g. for linear Gaussian models or when the underlying state space of the Markov model is finite, cf. [5, Example 1 and 2]). In a more general nonlinear non-Gaussian setting, approximative methods such as particle filters are necessary.

[2] With abuse of notation we denote by $p(x)$ the pdf of the random variable X. For two random variables X and Y the corresponding (possibly different) density functions are denoted by $p(x)$ and $p(y)$ respectively; $p(x, y)$ denotes the joint pdf and $p(x \mid y)$ is the conditional pdf of X given $Y = y$.

[3] Note that Doucet and Johansen *do not* call this the filtering problem [5]. They reserve this term for the estimation of the joint distributions $p(x_{1:n} \mid y_{1:n})$. Since we are only concerned with estimating the marginal distribution $p(x_n \mid y_{1:n})$ we will still refer to this problem as filtering which is in accordance with many other publications, see for example [1, 4] or the more recent publication [10].

(Sequential) importance sampling

The central idea of Particle filters is to represent the posterior of the system $p(x_k \mid y_{1:k})$ at some time k as a weighted set of samples, so called *particles*, denoted by $\{x_k^{(i)}; w_k^{(i)}\}$. If we ignore for a moment the weights and assume that the samples are from the desired distribution, i. e. ,

$$x_k^{(i)} \sim p(x_k^{(i)} \mid y_{1:k}), \quad i = 1, \ldots, N$$

the Monte Carlo method approximates $p(x_k \mid y_{1:k})$ by the empirical measure[4]

$$\hat{p}(x_k \mid y_{1:k}) = \frac{1}{N} \sum_{i=1}^{N} \delta_{x_k^{(i)}}(x_k), \tag{1}$$

where $\delta_x(\cdot)$ denotes the Dirac delta centred at x. The expectation of a test function $f : \mathbb{R}^{d_x} \to \mathbb{R}$ given by

$$\mathbb{E}[f(x_k) \mid y_{1:k}] = \int f(x_k) p(x_k \mid y_{1_k}) \, \mathrm{d}x_k$$

is then estimated by

$$\mathbb{E}^{\mathrm{MC}}[f(x_k) \mid y_{1:k}] = \int f(x_k) \hat{p}(x_k \mid y_{1_k}) \, \mathrm{d}x_k = \frac{1}{N} \sum_{i=1}^{N} f(x_k^{(i)}).$$

It is well-known that the variance of the approximation error using this estimator decreases *independent of* d_x with a rate of $\mathcal{O}(N^{-1})$. However, often it is either impossible or practically intractable to sample from the posterior directly and thus, one often relies on a technique called *importance sampling*.

We start by choosing an *importance density* $q(x_k \mid y_{1:k})$ and draw N samples $x_k^{(i)}$, $i = 1, \ldots, N$ from it. If we would use these samples to approximate $p(x_k \mid y_{1:k})$ as in (1) the result would obviously not be accurate in general. To correct this bias we introduce *importance weights*

$$w_k^{(i)} \propto \frac{p(x_k^{(i)} \mid y_{1:k})}{q(x_k^{(i)} \mid y_{1:k})}, \tag{2}$$

that we require to be normalised such that $\sum_i w_k^{(i)} = 1$. We can now approximate the target density by

$$p(x_k \mid y_{1:k}) \approx \sum_{i=1}^{N} w_k^{(i)} \delta_{x_k^{(i)}}(x_k). \tag{3}$$

This technique is called *importance sampling*. Expectations of test functions can then be

[4] Again, we slightly abuse notation for the sake of simplicity and refrain from a rigorous measure-theoretic formulation.

estimated by

$$\mathbb{E}^{\mathrm{MC}}[f(x_k) \mid y_{1:k}] = \sum_{i=1}^{N} w_k^{(i)} f(x_k^{(i)}).$$

Due to assumption (A3) ideally we would like a recursive formula to update the weights at each step. To obtain such a formula we consider the full posterior $p(x_{0:k} \mid y_{1_k})$ and express it in terms of the posterior at the previous time step and the known pdfs $p(y_k \mid x_k)$ and $p(x_k \mid x_{k-1})$:

$$\begin{aligned} p(x_{0:k} \mid y_{1:k}) &\propto p(y_k \mid x_{0:k}, y_{1:k-1}) p(x_{0:k} \mid y_{1:k-1}) \\ &= p(y_k \mid x_k) p(x_k \mid x_{0:k-1}, y_{1:k-1}) p(x_{0:k-1} \mid y_{1:k-1}) \\ &= p(y_k \mid x_k) p(x_k \mid x_{k-1}) p(x_{0:k-1} \mid y_{1:k-1}), \end{aligned}$$

where we used Bayes' theorem and the properties of the system described earlier (see [1] for a more detailed derivation). If in addition we choose an importance density that factorises such that

$$q(x_{0:k} \mid y_{1:k}) = q(x_k \mid x_{0:k-1}, y_{1:k}) q(x_{0:k-1} \mid y_{1:k-1})$$

the weights (2) can be written as

$$\begin{aligned} w_k^{(i)} &\propto \frac{p(y_k \mid x_k^{(i)}) p(x_k^{(i)} \mid x_{k-1}^{(i)}) p(x_{0:k-1}^{(i)} \mid y_{1:k-1})}{q(x_k^{(i)} \mid x_{0:k-1}^{(i)}, y_{1:k}) q(x_{0:k-1}^{(i)} \mid y_{1:k-1})} \\ &= \frac{p(y_k \mid x_k^{(i)}) p(x_k^{(i)} \mid x_{k-1}^{(i)})}{q(x_k^{(i)} \mid x_{0:k-1}^{(i)}, y_{1:k})} w_{k-1}^{(i)}. \end{aligned}$$

Since we are only interested in estimating the filtered posterior $p(x_k \mid y_{1:k})$ we choose an importance density $q(x_k \mid x_{0:k-1}, y_{1:k}) = q(x_k \mid x_{k-1}, y_k)$ that only depends on x_{k-1} and y_k. Then, it suffices to only keep $x_k^{(i)}$ in memory while the path $x_{0:k-1}^{(i)}$ and history of observations $y_{1:k-1}$ can be discarded. The weights can recursively be computed by

$$w_k^{(i)} \propto \frac{p(y_k \mid x_k^{(i)}) p(x_k^{(i)} \mid x_{k-1}^{(i)})}{q(x_k^{(i)} \mid x_{k-1}^{(i)}, y_{1:k})} w_{k-1}^{(i)}. \tag{4}$$

This is usually referred to as *sequential* importance sampling. We summarise the results up to this point in Algorithm 1. Note that since at time $k = 0$ no observation or previous state is available, we sample from the (known) prior and weigh all particles equally.

Using this approach alone, however, leads to *degeneracy* of the particles. It can be shown that the variance of the weights can only increase at every step [4, Proposition 1] which implies that the algorithm will eventually produce a single non-zero weight $w^{(i)} = 1$, carrying all the statistical information. This is visualised in Figure 1 where we plotted a

Algorithm 1: Sequential importance sampling

Input: n observations $y_1, y_2, \ldots, y_n$; number of particles N

Sample $x_0^{(i)} \sim p(x_0^{(i)})$;

Set weights $w_0^{(i)} = 1/N$;

for $k = 1, 2, \ldots, n$ **do**

  Sample $x_k^{(i)} \sim q(x_k^{(i)} \mid x_{k-1}^{(i)}, y_k)$. ;

  Compute unnormalised weights $\tilde{w}_k^{(i)}$ according to (4) ;

  Normalise $w_k^{(i)} = \tilde{w}_k^{(i)} / \sum_j \tilde{w}_k^{(j}$;

end

histogram of the weights after the first few time steps using the model described later in Example 2. One can clearly see that after a few steps almost all of the weights are zero. To account for this problem, we introduce another technique called *resampling*.

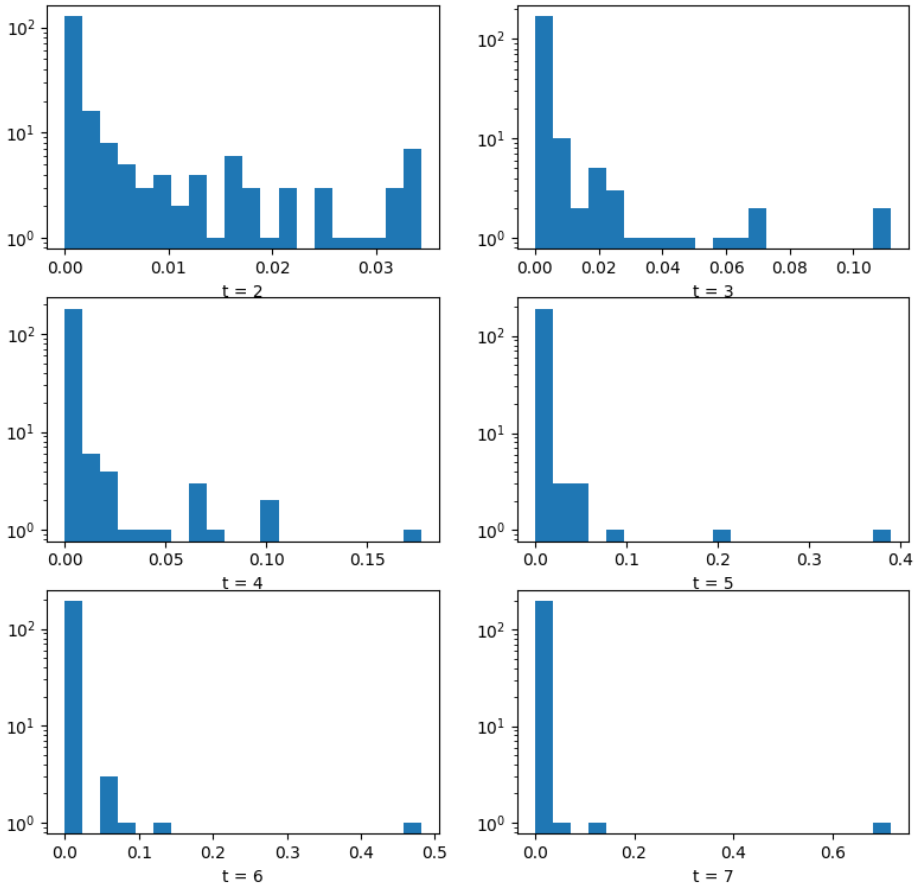

Figure 1: A histogram of 200 weights after just a few iterations. Almost all of the weights are zero at $k = 7$ which demonstrates the degeneracy of the particles.

Resampling

We present two resampling strategies in this section. The overall goal of all resampling methods is to remove particles with negligible weights with a high probability and replicate those with high weights. After resampling, the future particles are more concentrated in domains of higher posterior probability, which entails improved estimates. It can, of course, happen that a particle with a low weight at time k has a high weight at time $k+1$, in which case resampling could be wasteful. It should also be mentioned that if particles have (unnormalised) weights with a small variance, resampling might be unnecessary. This is discussed briefly at the end of this section.

As above, we denote by $\{x^{(i)}; w^{(i)}\}_{1\leq i\leq N}$ the set of particles with their associated weights at some time k (which is omitted in the notation). We assume that the weights have already been normalised such that $\sum_i w^{(i)} = 1$. We further denote by $\{\tilde{x}^{(i)}; \tilde{w}^{(i)}\}_{1\leq i\leq N}$ the particles and weights after resampling took place. We require the particles $\tilde{x}^{(i)}$ to be weighted equally which implies, since we also require the weights to be normalised that $\tilde{w}^{(i)} = 1/N$.

The use of resampling to improve importance sampling was originally introduced by Gordon, Salmond and Smith in [6]. The resampling methods presented here are two of the most popular amongst the literature, see [3]. The most simple resampling strategy, called *multinomial resampling* is not discussed here due to its poor performance compared to other techniques. It is mentioned because it is the method introduced in [6] as part of the *bootstrap filter* that uses the prior as the proposal density (c. f. Examples 1 and 2).

Both methods presented in the following are based on drawing samples from the point mass distribution $\sum_{j=1}^{N} w^{(j)}\delta_{X^{(j)}}$. In practice, this is achieved by repeated uses of the inversion method, which itself uses the empirical cumulative distribution function (cdf) associated with the weights. This is based on the following fact:

Claim. If U is a uniform random variable on $(0,1]$ then $X = F^{-1}(U)$ has distribution F, where F is the cdf of X and $F^{-1}(t) = \min\{x \mid F(x) = t\}$ is the inverse cdf.

Proof. Let $U \sim \mathcal{U}(0,1]$. Then

$$\begin{aligned} P(F^{-1}(U) \leq x) &= P(\min\{x \mid F(x) = U\} \leq x) && \text{(definition of } F^{-1}\text{)} \\ &= P(U \leq F(x)) \\ &= F(x) && \text{(definition of distribution of } U\text{).} \end{aligned}$$

□

The inversion method can be explained visually as follows. We plot the empirical cdf of the weights and sample from $U \sim \mathcal{U}(0,1]$. We denote the actual value of the sample by u.

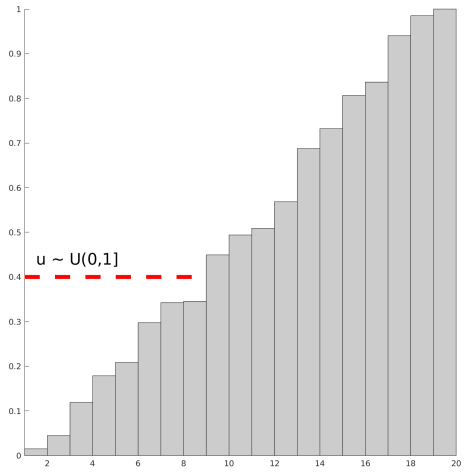

Figure 2: Visualisation of the inversion method. The bars represent the empirical cumulative distribution function associated with a set of 19 weights. The dashed line is horizontally drawn from the value of u at the vertical axis until it "hits" a bar. The index of the bar yields the generated sample (in this case the new sample is therefore 9).

We then draw a horizontal line from the coordinate $(0, u)$ to the right until it intersects one of the bars, see Figure 2. The index of the bar that is intersected determines the new sample.

In our case, we do not draw just one sample U but we generate N different samples $\{U_i\}_{1 \leq i \leq N}$ in such a way that they are sorted in ascending order. For every of these samples (from lowest to highest) we look for the intersected bar and add its index to a list. This list then corresponds to the indices of the particles that should be resampled. Consider the following example: Suppose we had five particles and the list of indices after the resampling reads $\{1, 3, 4, 4, 5\}$. Then, particles $X^{(1)}, X^{(3)}, X^{(5)}$ should be resampled, particle $X^{(4)}$ should even be duplicated. Particle $X^{(2)}$, however, will be dropped. In other words, the particles after the resampling are

$$\tilde{X}^{(1)} = X^{(1)},\ \tilde{X}^{(2)} = X^{(3)},\ \tilde{X}^{(3)} = X^{(4)},$$
$$\tilde{X}^{(4)} = X^{(4)},\ \tilde{X}^{(5)} = X^{(5)}.$$

The two strategies presented in the following only differ in the way the U_is are generated. We summarise the results in Algorithm 2.

Algorithm 2: Resampling using the empirical cdf

Input: N samples $U_i \sim \mathcal{U}(0,1]$ sorted in ascending order; list of weights
Result: List of indices I that represent that particles to be resampled
C = cumsum(weights) ; // Generate the empirical cdf as list of accumulated sums[5]
I = zeros(N);
$i, j = 0$;
while $i < N$ **do**
  if $U_i < C_j$ **then** // Found intersecting bar
    $I_i = j$;
    $i = i + 1$;
  else
    $j = j + 1$;
  end
end

Systematic Resampling

This algorithm separates the sample space into N divisions. One random offset, drawn from a $\mathcal{U}[0,1)$ distribution, is used to choose where to sample from for all divisions. This guarantees that every sample is exactly $1/N$ apart.

In other words, the U_is are generated by sampling $\tilde{U} \sim \mathcal{U}[0,1)$ and defining

$$U_i = \frac{\tilde{U} + i - 1}{N} \quad \text{for } i = 1, \ldots, N\,.$$

Stratified Resampling

This algorithm is similar to the previous one, its aim is to make selections relatively uniformly across the particles. We start by partitioning the $(0,1]$ interval into N disjoint sets, $(0,1] = (0,1/N] \cup (1/N, 2/N] \cup \cdots \cup ((N-1)/N, 1]$. The U_is are then drawn independently in each of the sub-intervals:

$$U_i \sim \mathcal{U}((i-1)/N, i/N].$$

We mentioned earlier that resampling might be unnecessary if the weights are sufficiently uniform and we would like to have a criterion allowing us to check whether resampling should be performed. To that end the effective sampling size (ESS) is often

[5] Here, the function `cumsum` is assumed to work in the same way as, for example, MATLAB's or NumPy's `cumsum` function. That is, `cumsum([1,2,3,4])` should return `[1,3,6,10]`.

used which can be estimated using

$$ESS \approx \left(\sum_{i=1}^{N} \left(w_t^{(i)} \right)^2 \right)^{-1} . \tag{5}$$

For more details, see [1, p. 179]. If the variance of the weights is maximal, i. e. , if all but one of the weights are zero, the value of ESS is 1. If, however, the weights all have the same value $w_t^{(i)} = 1/N$ the value of ESS is N, since

$$\left(\sum_{i=1}^{N} \left(\frac{1}{N} \right)^2 \right)^{-1} = \left(N \frac{1}{N^2} \right)^{-1} = N .$$

Therefore, we will only resample if ESS is below a certain threshold, e. g. $N/2$.

We have now gathered everything we need to implement a particle filter, since essentially particle filters are simply a combination of sequential importance sampling and one of the resampling strategies (or, of course, any other resampling algorithm not presented here). Consequently, these methods are sometimes called *Sequential Importance Sampling with Resampling* abbreviated by SIR or SIS/R. Other popular names include *Bootstrap filter*, *Monte Carlo filter*, *Survival of the fittest* or *Condensation algorithm*.

Implementation

In this section we present an implementation of a particle filter in C++. The particle filter itself is implemented as a dependency-free[6] header-only generic library that, while being easy to set up and use, is versatile and can be used with a wide variety of problems. This is demonstrated by three examples, two of which are based on the same problem but are using different prior and proposal distributions.

The code can be found at `github.com/nilsfriess/ParticleFilter`. The GitHub repository contains a CMake [9] project containing the actual library in the folder `libs/smcpf` and three examples located in the folder `apps`. Information about the dependencies of the individual examples and instructions on how to build and run them can be found in the `README.md` file inside the root folder of the repository. The `data` folder contains sample data to use with the examples and scripts that were used to generate the sample data.

The library consists of the following classes (and their respective header files)

- Particle (`particle.hh`)
- ParticleFilter (`particlefilter.hh`)
- Model (`model.hh`)
- History (`history.hh`)

All of these classes lie in a namespace `smcpf` and are templated to allow for arbitrary particle types, e. g. the `Particle` class, that holds the value and weight of a single particle is of the following form

```
template <class PT>
class Particle {
private:
  PT m_value;
  double m_weight;
...
};
```

where the particle type `PT` could take values of some finite set, be a real number (i. e. , a `double`) or a n-dimensional vector etc. The `ParticleFilter` class implements the algorithms introduced above. It takes the following template parameters

[6] The library can be configured to run some parts in parallel. In that case the program has to be linked against Intel's Threading Building Blocks (TBB) library [7]. If the parallel capabilities are not used, the library depends only on the C++ standard library.

```
template <class PT, class OT,
          size_t N,
          typename... Args>
class ParticleFilter { ... }
```

where `PT` and `OT` denote the type of particle and observation, respectively and `N` is the number of particles. The last template parameter `Args` is *parameter pack* that can hold an arbitrary number of additional arguments of any type. The use for such arguments is explained in the description of the `Model` class below.

The library can be configure to run parts of it in parallel, e. g. the particle evolution (see below). To enable this feature, one has to define the prepocessor constant `PF_USE_PARALLEL` before the header file `particlefilter.hh` is included, e. g. by

```
#define PF_USE_PARALLEL
#include <particlefilter.hh>
```

Internally, this uses the C++17 `execution` header from the C++ standard library that depends on Intel's TBB library [7]. This means, if `PF_USE_PARALLEL` is set, the TBB header files have to be available for the compiler and the program has to be linked against the library `tbb`. Not using the parallel capabilities of the library does neither change the way it has to be used nor does it limit its capabilities. It only affects how the internal algorithms are run.

Apart from these compile-time parameters, to construct a `ParticleFilter` one also needs to provide an instance of a `Model`, a resampling strategy[7], a resampling threshold and an initial seed for the random number generator (rng). Additionally, a boolean parameter that specifies whether some information from every time step should be held in memory has to be given (e. g. for debugging or plotting, see examples below). Only the first parameter is mandatory, i. e. the signature of the constructor of the `ParticleFilter` class is given by

```
ParticleFilter(
    Model<PT, OT, Args...> *t_model,
    bool t_enable_history = false,
    ResamplingStrategy t_strategy = ResamplingStrategy::RESAMPLING_SYSTEMATIC,
    double t_threshold = 0.5,
    double t_seed = 0)
```

The value of `t_strategy` can either be the default `RESAMPLING_SYSTEMATIC` or `RESAMPLING_NONE`, both of which are defined in the enumeration `ResamplingStrategy`. The value of `t_threshold` is used to decide when to actually perform resampling. At each time step an estimate of the effective sampling size is computed using (5). The particles

[7] At the moment, only systematic resampling is implemented.

are only resampled if this value is below `t_threshold` × `N`. Thus, `t_threshold` should take values between zero and one (since ESS takes values between 1 and `N`), where a value of zero implies that the particles are never resampled and one leads to resampling being performed at each step. Before explaining the methods defined inside the `ParticleFilter` we discuss the `Model` class.

This class is implemented as an abstract base class (sometimes called *interface*), meaning that the class itself cannot be instantiated. Therefore, in order to define a model, a class that is derived from `Model` has to be implemented. The class is also templated with the following parameters

```
template <class PT,
          class OT,
          typename... Args>
class Model { ... }
```

where `PT` and `OT` are again the particle and observation type. To explain the usage of the parameter pack `Args` we first discuss the virtual functions of the class:

```
virtual PT zero_particle() = 0;

virtual void sample_prior(Particle<PT> &t_particle) = 0;

virtual double update_weight(const Particle<PT> &t_particle_before_sampling,
                             const Particle<PT> &t_particle_after_sampling,
                             const OT &t_observation,
                             Args... t_args) = 0;

virtual PT sample_proposal(const Particle<PT> &t_particle,
                           const OT &t_observation,
                           Args... t_args) = 0;
```

All these methods are *pure virtual* methods meaning that a model class that derives from this class must implement all four of them. They all get automatically called by the `ParticleFilter`. The first method should return a value of type `PT` that represents zero, i. e. , in the one dimensional case where `PT = double` this should be 0, if `PT` is e. g. two dimensional, this method should return the 2D zero vector and analogously for higher dimensions and other particle types. The method `sample_prior` is used to initialise the set of particles. It has to set the value of the given particle `t_particle` using the `set_value` method of the particle. This is different from the `update_weight` and `sample_proposal` methods that *do not* alter the particle themselves. Rather they should return the value the the particle's weight should get multiplied by and the particle's new value, respectively (see examples below).

The first parameter of the `update_weight` method is the particle before `sample_proposal` is called and the second after it is called. This is useful, since the developer cannot specify the order in which these two methods are called. However, some models require the value of the particle before it has been updated (cf. Example 2) and some after the sampling step (cf. Example 1 and 3). The type of the last parameter `t_args` of both methods is specified by the template parameter pack `Args`. It can be used to supply an arbitrary number of additional parameters of arbitrary types to these methods. This can be used if the proposal pdf and weight update function depend on additional parameters like the current time step, which is the case in all of the following examples. The types provided as `Args` to the `Model` class and those provided to the `ParticleFilter` must match, otherwise the program will not compile. The following simple example is used to demonstrate how a model can be defined.

Example 1. This example has been studied in a number of publications before, see for example [1, 6, 8]. The implementation can be found in the file `apps/example1/main.cc`. Let

$$
\begin{aligned}
p(x_0) &= \mathcal{N}(0, 0.5) \\
p(x_n \mid x_{n-1}) &= \mathcal{N}(x_n; h_n(x_{n-1}; n), \sigma_{\text{sys}}) \\
p(y_n \mid x_n) &= \mathcal{N}(y_n; \frac{{x_n}^2}{20}, \sigma_{\text{obs}})
\end{aligned}
$$

where

$$
h_n(x_{n-1}; n) = \frac{1}{2} x_{n-1} + \frac{25 x_{n-1}}{1 + x_{n-1}^2} + 8 \cos(1.2n) \tag{6}
$$

and $\mathcal{N}(\mu, \sigma)$ denotes a Gaussian distribution with mean μ and variance σ and $\mathcal{N}(x; \mu, \sigma)$ denotes a Gaussian pdf with mean μ and variance σ evaluated at x. We choose $\sigma_{\text{sys}} = 10$ and $\sigma_{\text{obs}} = 1$.

Since both the particles and observations are one-dimensional we set `PT = double, OT = double`. Note that h defined in (6) depends on the index of the current time step n. Therefore, we provide one additional template parameter, i. e. , we set `Args = int`. Hence, the model could be defined as

```
class ExampleModel : public Model<double, double, int> {
```

Here, the zero particle is simple the value `0.0` and the prior is a Gaussian, centred at 0 with variance 0.5.

```
...
public:
  virtual double zero_particle() override { return 0.0; }

  virtual void sample_prior(Particle<double> &t_particle) override {
```

```
    t_particle.set_value(m_prior(m_gen));
  }
```

where `m_gen` and `m_prior` are defined as

```
...
private:
  std::mt19937 m_gen;
  std::normal_distribution<> m_prior{0, 0.5};
```

The class `std::mt19937` defines a pseudo rng based on the *Mersenne Twister* algorithm.

To implement the remaining two methods we first have to choose a proposal density. For simplicity, we use the prior $p(x_n \mid x_{n-1})$, i. e. , we implement a bootstrap particle filter. The recursive weight update formula (4) then simplifies to

$$w_k^{(i)} \propto w_{k-1}^{(i)} p(y_k \mid x_k^{(i)}),$$

and we obtain

```
  virtual double
  update_weight(const Particle<double> & /*t_particle_before_sampling*/,
                const Particle<double> &t_particle_after_sampling,
                const double &t_observation, int /*t_step*/) override {
    auto pval = t_particle_after_sampling.get_value();
    auto mean = (pval * pval) / 20.0;
    auto var = 1.0;

    return normal_pdf(t_observation, mean, var);
  }
```

The function `normal_pdf(double, double, double)` is defined in the file `helper.hh` and simply implements a Gaussian pdf. Note that the `update_weight` method does not return the new value of the weight but rather the value the weight should be multiplied with, i. e. , in our case $p(y_k \mid x_k^{(i)})$.

Since in this case the proposal is the prior, the `sample_proposal` method is a straightforward implementation of $p(x_k^{(i)} \mid x_{k-1}^{(i)})$ defined by the model above.

```
  virtual double sample_proposal(const Particle<double> &t_particle,
                                 const double & /*t_observation*/,
                                 int t_step) override {
    auto pval = t_particle.get_value();
    auto mean = pval / 2.0 + (25 * pval) / (1 + pval * pval) +
                8 * std::cos(1.2 * (t_step));
```

```
    auto var = 10.0;

    std::normal_distribution<> proposal{mean, var};
    return proposal(m_gen);
  }
```

With these four methods the model is fully defined and can be used to construct a `ParticleFilter`. In the file `apps/example1/main.cc` two additional methods

- `void load_observations(const std::string &t_filename)`
- `std::optional<std::pair<double, double>> next_observation()`

are defined. The first method loads observations from a `csv` file (comma-separated values) and the second method returns a new observation each time it is called until all observations have been read. An observation consists of a time index and the actual observation. The artificial observations can be generated using the Python script `data/example1/gen_obs_ex1.py` that generates 100 random samples $x_k \sim p(x_k \mid x_{k-1})$ where $x_0 \sim p(x_0)$. The observations are then generated by squaring these values, dividing them by 20 and perturbing them by additive Gaussian noise such that they are distributed according to $p(y_k \mid x_k)$. These artificial observations and the corresponding time steps are then written to a file that can be read by the method `load_observations`.

With this model we con now easily implement the actual particle filter:

```
int main() {
  constexpr size_t N = 400;

  typedef smcpf::ParticleFilter <double, // Type of particle
                                 double, // Type of observation
                                 N,      // Number of particles
                                 int>    // index of observation
                                 PF;
  ExampleModel model;

  PF pf(&model, true);
  ...
```

The observations are automatically loaded inside the constructor of the model and the initial set of particles is initialised inside the constructor of the `ParticleFilter`. To run the actual filtering we have to call the method `evolve`, defined as

```
void ParticleFilter::evolve(OT t_observation,  Args... t_args)
```

This method calls the `sample_proposal` and `update_weight` method of the model for each of the particles. If the template parameter `parallel` is set to true, this is done in parallel. This requires no extra care since the particles are all independent and altering the value or weight of one particles does not affect any of the other particles. In addition, this method checks whether resampling is necessary and possibly resamples the particles. Resampling, however, cannot be easily parallelized (there are some publications discussing the parallelization of resampling algorithms, see e. g. [10]).

The `evolve` method has to be called at each time step. Using the `next_observation` method of our model we do this as follows

```
while (auto obs = model.next_observation()) {
  pf.evolve(obs.value().second, obs.value().first);
}
```

Using the method

```
PT ParticleFilter::weighted_mean()
```

one can obtain a Monte Carlo estimate of the expectation of a random variable with pdf $p(x_k \mid y_{1:k})$ at each time step k. Since we set the `history` flag to `true`, these means are all held in memory and we can read them after all observations have been processed. To that end, the `ParticleFilter` class provides a method

```
template <class PTWriter, class ArgWriter>
void write_history(std::ostream &t_out,
                   PTWriter t_writer,
                   ArgWriter t_awriter,
                   char t_separator = ',')
```

that writes the contents of the history to the output stream `t_out` (e. g. a file stream) using the functors `PTWriter` and `ArgWriter`, which have to be implemented such that they convert values of types `PT` and `Args...` to `std::strings` (i. e. , they define how to print these values). For a detailed description on the usage of this method see the example files and the comments in the definition of the particle filter class.

Figure 3 shows the plot of one exemplar run together with the generated observations and simulated values using the model and particle filter described above.

The following two examples are based on the same problem and demonstrate how third-party libraries can be used to define arbitrary particle types.

Example 2 (Lotka Volterra using bootstrap filter). Given approximations of the population size of a certain species (the *predators*) we want to estimate the quantity of another species (the *preys*) and estimate their respective population sizes. The predators and

preys are assumed to follow the so called *Lotka-Volterra model*. This model is mainly described by the following system of differential equations:

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t} &= \alpha x - \beta xy, \\ \frac{\mathrm{d}y}{\mathrm{d}t} &= \delta xy - \gamma y, \end{aligned} \tag{7}$$

where

- x is the number of prey;
- y is the number of predator;
- $\alpha, \beta, \gamma, \delta$ describe the interaction of the predator and prey.

We assume that we can observe the predators (i.e. y) but not the number of prey. To this end, we will create artificial values for the predators and preys by solving the Lotka Volterra equations with some initial values. We then "observe" the number of predator by perturbing the exact computed values at discrete points in time by additive Gaussian noise. These observations are then used as inputs for the particle filter.

Formally, this model is defined by the following equations

$$\begin{aligned} p(\boldsymbol{x}_0) &= \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{I}), \\ p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}) &= \mathcal{N}(\boldsymbol{x}_t; M(\boldsymbol{x}_{t-1}; t), \boldsymbol{I}), \\ p(y_t \mid \boldsymbol{x}_t) &= \mathcal{N}(y_t; F(\boldsymbol{x}_t), 2), \end{aligned} \tag{8}$$

where $\boldsymbol{\mu}_0 = (4, 6)^T$, $\boldsymbol{I}$ denotes the 2×2 identity matrix, $M(x_{t-1}; t)$ is the solution of the Lotka Volterra equations from time $t-1$ to t with initial value x_{t-1} and $F(\boldsymbol{x})$ extracts the value of the predator from the vector $\boldsymbol{x} = (x_1, x_2)^T \in \mathbb{R}^2$, i. e. ,

$$F(\boldsymbol{x}) = F((x_1, x_2)^T) = x_2 \tag{9}$$

or equivalently

$$F(\boldsymbol{x}) = \boldsymbol{H}\boldsymbol{x}, \tag{10}$$

where $\boldsymbol{H} = (0, 1) \in \mathbb{R}^{1 \times 2}$ is an *observation matrix*. Additionally, we assume the values of $\alpha, \beta, \gamma, \delta$ are known to be

$$\alpha = \gamma = 1 \qquad \text{and} \qquad \beta = \delta = 0.1\,.$$

To define a model class, we proceed as in the previous example. Since the particles take values in $\mathbb{R}^2$ we cannot use `double` values. Since this example should also demonstrate how third-party libraries can be used, instead of using `std::vector` or `std::array` to

model the 2D vectors we use a vector class from the linear algebra library *Armadillo* [12]. The model can then be defined as

```
class LotkaVolterra : public smcpf::Model<arma::dvec, double, double> {
```

where the first parameter `arma::dvec` defines the particle type (i. e. , the two dimensional vector holding the value of the prey and predator, respectively), the second parameter is again the observation type and the additional `double` parameter denotes the type of the time steps.

As above, we need to override the four pure virtual methods of the `Model` class. Here, a zero particle is the zero vector, i. e. ,

```
  arma::dvec zero_particle() override { return {0, 0}; }
```

To sample from the prior we use the functor `BivariateGaussian` that can be found in the `functors.hh` file in the `apps/example2` folder. This functor represents a two-dimensional Gaussian distribution. Internally it uses a library called *StatsLib* [11] which provides a wide variety of statistical distributions and helper functions, in particular it provides classes and methods to sample from and evaluate multivariate normal distributions. After setting up the prior in the constructor of the `LotkaVolterra` class

```
    const arma::dvec mu{{4}, {6}};
    const arma::dmat sigma{{1, 0}, {0, 1}};
    m_prior = BivariateNormal(mu, sigma);
```

we can use it to define the `sample_prior` method

```
  virtual void sample_prior(smcpf::Particle<arma::dvec> &t_particle) override {
    t_particle.set_value(m_prior());
  };
```

As in the previous example, to define the remaining two methods we have to choose a proposal density. In this example, we again implement a bootstrap filter, i. e. , we use $p(x_t \mid x_{t-1})$ as the proposal and obtain

```
  virtual double update_weight(
      const smcpf::Particle<arma::dvec> & /*t_particle_before_sampling*/,
      const smcpf::Particle<arma::dvec> &t_particle_after_sampling,
      const double &t_observation, double /*t_time*/) override {
    return stats::dnorm(t_observation,
                        extract_predator(t_particle_after_sampling), 2);
  }

  arma::dvec sample_proposal(const smcpf::Particle<arma::dvec> &t_curr_particle,
                             const double & /* t_observation */,
```

```
                                double t_time) override {
    const auto mu = evolve(t_curr_particle, t_time);
    const arma::dmat cov = {{1, 0}, {0, 1}};

    return BivariateNormal(mu, cov)();
  }
```

The method `stats::dnorm` from the *StatsLib* library [11] used in the `weight_update` method provides the density function of a one-dimensional Gaussian. The method `extract_predator` implements the observation function $F(\boldsymbol{x})$ given in (9).

Inside the `sample_proposal` method the particle is evolved using the model's method `evolve` that simply solves the Lotka Volterra equations at the current time step using the `odeint` methods from the C++ library Boost [2].

The remaining methods of the class are again helper functions to process the previously generated observations. Setting up and running the particle filter is then done similarly as in the first example. Figure 4 shows the exact values of the predators and preys, respectively, and the observations that were given to the particle filter. The two dotted lines show the sample means of the filtered posterior, i. e. , the simulated values for the predators and preys.

The previous two examples both implement a bootstrap filter. The next example is based on the same model as Example 2 but uses a different proposal.

Example 3 (Lotka Volterra using optimal proposal)**.** Consider the model from the previous example defined in (8). In [4] Doucet, Godsill and Andrieu showed that for models of this form an optimal proposal density exists in the sense that the variance of the importance weights is minimised.

In general, this proposal is given by

$$q(x_k \mid x_{k-1}^{(i)}, y_k) = p(x_k \mid x_k^{(i)}, z_k)$$

Substituting this into (4) yields

$$w_k^{(i)} \propto w_{k-1}^{(i)} p(y_k \mid x_{k-1}^{(i)}).$$

As is illustrated in [4, Example 3] for models with nonlinear internal dynamics and linear measurements both $p(x_k \mid x_k^{(i)}, z_k)$ and $p(y_k \mid x_{k-1}^{(i)})$ are Gaussian. Our model obviously fulfils these requirements and we can define $p(x_k \mid x_k^{(i)}, z_k)$ and $p(y_k \mid x_{k-1}^{(i)})$ as follows

(see [4] for the derivation). First let

$$\Sigma_t = \left(\boldsymbol{I} + \frac{1}{2} \boldsymbol{H}^T \boldsymbol{H} \right)^{-1}$$
$$\boldsymbol{\mu}_t = \Sigma_t \left(M(\boldsymbol{x}_{t-1}; t) + \frac{1}{2} H^T y_t \right).$$

Then

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, y_t) = \mathcal{N}(\boldsymbol{x}_t; \boldsymbol{\mu}_t, \Sigma_t) \tag{11}$$

and

$$p(y_t \mid \boldsymbol{x}_{t-1}) = \mathcal{N}(y_t; \boldsymbol{H} M(\boldsymbol{x}_{t-1}; t), \boldsymbol{H}\boldsymbol{H}^T + 2). \tag{12}$$

Since this choice of proposal density minimises the variance of the weights which implies an increase of the effective sampling size compared to the previous approach, we expect resampling to be performed less often. This can be easily checked using the output files that the examples generate, since the history also saves whether resampling has been performed at a particular time step. And indeed, while the previous example requires resampling to be performed at around 20 steps (of a total of 50 steps), using the optimal proposal this number is reduced to five resampling steps (both exemplar runs used 5000 particles and a resampling threshold of 50%). This also has an effect on the speed of the particle filter since resampling is the only step that is not easily parallelized. Therefore, one wants to resample as few times as possible.

The code for this example can be found in the folder `apps/example3` but is essentially the same as in the previous example with the `update_weight` and `sample_proposal` methods altered to implement (12) and (11). For completeness, the results using this example are plotted in Figure 5.

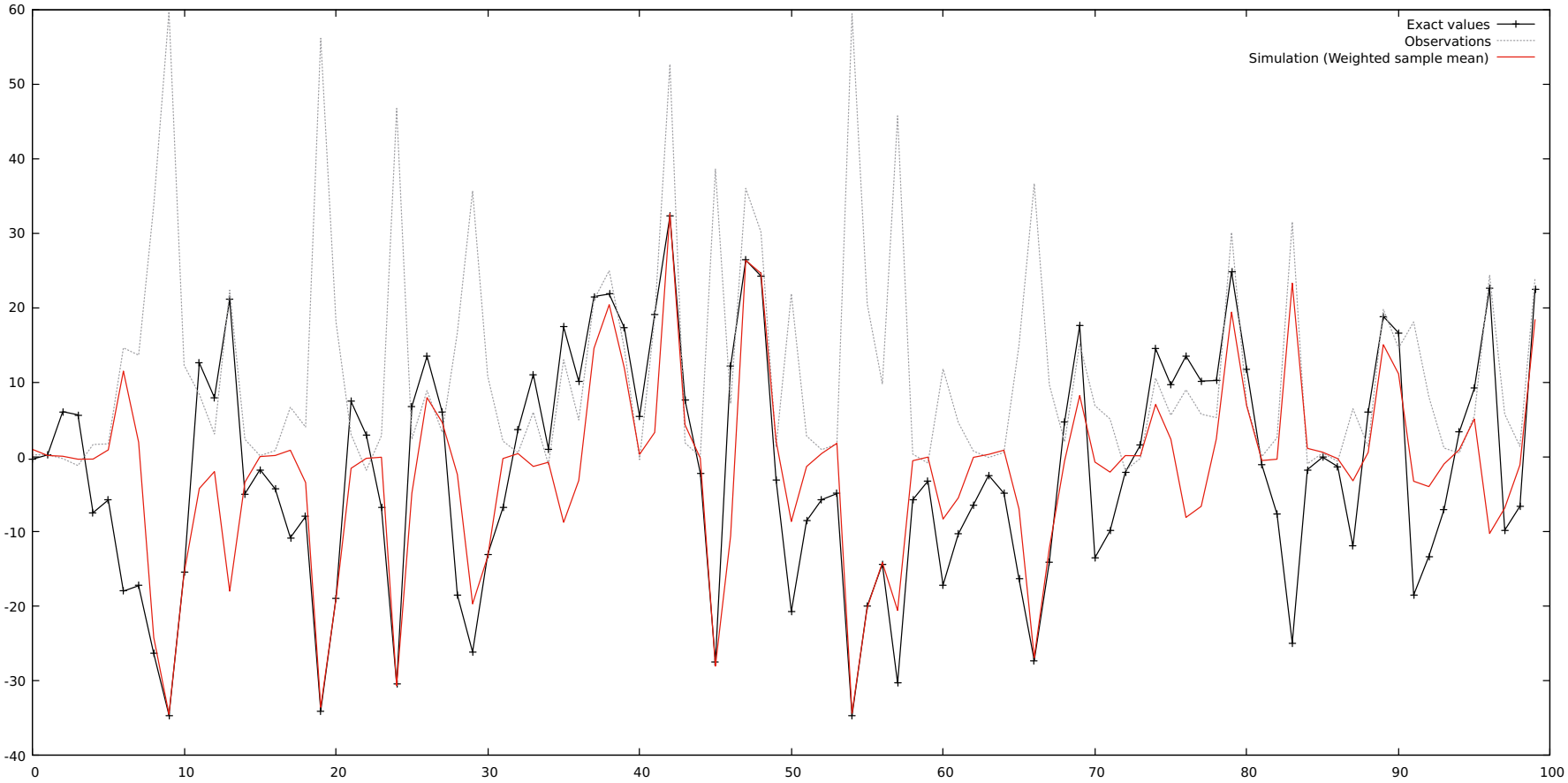

Figure 3: Results from one exemplar run using the model and particle filter described in Example 1. The black line with crosses shows the exact values, the dotted line shows the generated observations and the red line (the line without points) is the result using 400 particles, systematic resampling and a resampling threshold of 50%.

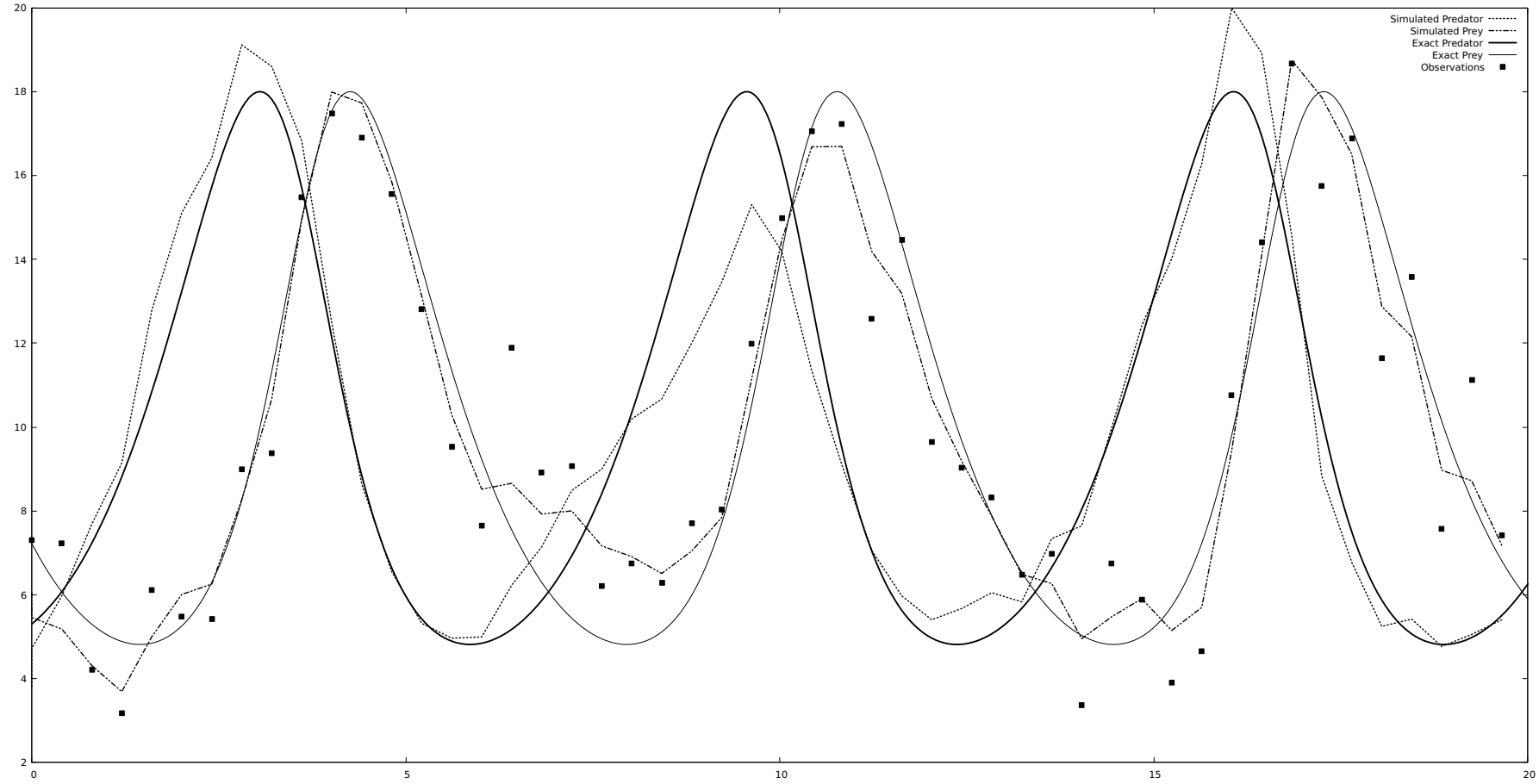

Figure 4: Results from one exemplar run using the model and particle filter described in Example 2. Here, 1000 particles and the default values of the particle filter were used.

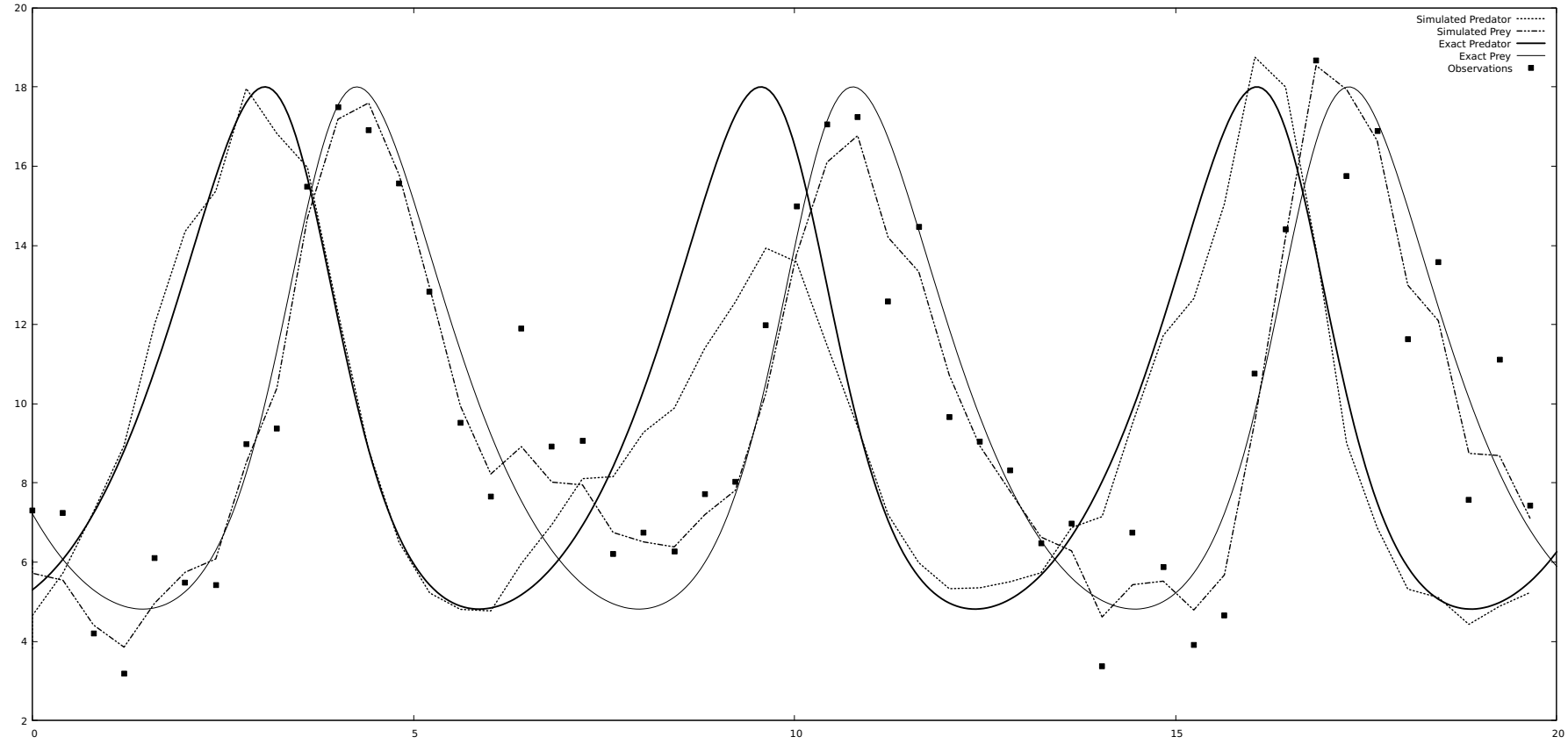

Figure 5: Results from one exemplar run using the model and particle filter described in Example 3. Here, 1000 particles and the default values of the particle filter were used.

References

[1] M. Arulampalam et al. "A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking". In: *IEEE Transactions on signal processing* 50.2 (2002), pp. 174–188.

[2] Boost. *Boost C++ Libraries*. `http://www.boost.org/`. Version 1.72.0. Last accessed 29. Feb 2020.

[3] R. Douc and O. Cappe. "Comparison of resampling schemes for particle filtering". In: *ISPA 2005. Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis, 2005.* Sept. 2005, pp. 64–69.

[4] A. Doucet, S. Godsill, and C. Andrieu. "On sequential Monte Carlo sampling methods for Bayesian filtering". In: *Statistics and Computing* 10.3 (2000), pp. 197–208.

[5] A. Doucet and A. M. Johansen. "A Tutorial on Particle Filtering and Smoothing: Fifteen years later". In: *The Oxford handbook of nonlinear filtering* (2011). Ed. by Dan Crisan and Boris Rozovskii, pp. 656–705.

[6] N. J. Gordon, D. J. Salmond, and A. Smith. "Novel approach to nonlinear/non-Gaussian Bayesian state estimation". In: *IEE proceedings F (radar and signal processing)*. Vol. 140. 2. IET. 1993, pp. 107–113.

[7] Intel. *TBB: Threading Building Blocks*. `https://software.intel.com/en-us/tbb`. Version 2020.1. Last accessed 29. Feb 2020.

[8] G. Kitagawa. "Monte Carlo Filter and Smoother for Non-Gaussian Nonlinear State Space Models". In: *Journal of Computational and Graphical Statistics* 5.1 (1996), pp. 1–25.

[9] Kitware. *CMake*. `https://cmake.org/`. Last accessed 29. Feb 2020.

[10] L. M. Murray, A. Lee, and P. E. Jacob. "Parallel Resampling in the Particle Filter". In: *Journal of Computational and Graphical Statistics* 25.3 (2016), pp. 789–805.

[11] K. O'Hara. *StatsLib: A C++ header-only library of statistical distribution functions*. `https://www.kthohr.com/statslib.html`. Version 3.0.0. Last accessed 29. Feb 2020.

[12] C. Sanderson and R. Curtin. "Armadillo: a template-based C++ library for linear algebra". In: *Journal of Open Source Software* 1.2 (2016), p. 26.