Software practical — Uncertainty Quantification

Implementation of a C++ Particle Filter Library

Winter semester 2019

Submitted by: **Nils Friess**

February 29, 2020

Supervisors: Prof. Robert Scheichl, Gianluca Detommaso

Heidelberg University

Introduction

{sec:intro}

In this report we present an implementation of a Particle Filter in C++. Before discussing the actual implementation we give the theoretical details of the method in this section. We start by giving a brief overview of the method before discussing the individual steps in more detail.

We remark here, that the naming in these methods is highly ambiguous and varies greatly from author to author and even in different publications of the same authors. We use – with some exceptions – the naming and notation used in [3] and highlight parts where the naming differs from other publications.

Overview of a Particle Filter

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 will be 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) The model describing the initial state and the evolution of the internal state in time is available in a probablistic form.

(A2) The 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 (ie. we assume that we receive new measurments 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*.

[1] Some authors use the terms *Particle Filter* and *SMC method* synonymously. Doucet and Johansen develop in [3] 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.

Such models consist of the triplet

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

where

- $n \in \mathbb{N}$ denotes discrete time;
- 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 fumction (pdf) of the system's state;[2]
- Y_n is the d_y-dimensional vector of observations which is assumed to be conditionally iindependent 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 $y_{1:n} \coloneqq (y_1, y_2, \ldots, y_n)$. This is often referred to as the *filtering problem* or *tracking*.[3]

1-step-predictor (Chapman-Kolmogorov)

In a restrictive set of cases this distribution can be computed exactly (e. g. when $p(y_t \mid y_t)$ is linear and the posterior of the system is Gaussian [1, p. 175] or when the underlying state space of the Markov model is finite, cf. [3, Example 1]). In a more general nonlinear non-Gaussian setting, approximative methods such as particle filters are necessary. Therefore, in practice a different approach is used.

(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

[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 Docuet and Johansen *do not* call this the filtering problem [3]. 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.

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]

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

where $\delta_x(\cdot)$ denotes the Dirac delta centered 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, in practice 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 $x_k^{(i)}$ 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*

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

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

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

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

[4] Again, we slightly abuse the notation for simplicity; the alternations required for a rigorous measure-theoretic formulation are straightforward.

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 a 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 pdf's $p(y_k \mid x_k)$ 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. If in addition we choose an importance density that factorizes 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

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

Since in our case, 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, merely $x_k^{(i)}$ is held in memory and the path $x_{0:k-1}^{(i)}$ and history of observations $y_{1:k-1}$ need not be stored. The weights can the recursively be computed by

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

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

Algorithm 1: Sequential importance sampling

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

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

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 weights according to (4). ;

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

end

{alg:sis}

that the variance of the weights can only increase at every step, which implies that the algorithm will eventually produce a single non-zero weight $w^{(i)} \approx 1$, carrying all the statistical information with the rest of the weights converging to zero. This is visualised in Figure 1 where we plotted a histogram of the weights after the first few time steps. One can clearly see that after a few steps almost all the weights are zero. To account for this problem, we introduce another technique called *resampling*.

Resampling

We are going to 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 t has a high weight at time $t + 1$, 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 $\{x^{(i)}, w^{(i)}\}_{1 \le i \le N}$ 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, i. e. $\sum_i w^{(i)} = 1$. We further denote by $\{\tilde{x}^{(i)}, \tilde{w}^{(i)}\}_{1 \le i \le N}$ 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 et al, see [4], laying the ground for Particle filters and SMC methods in general.

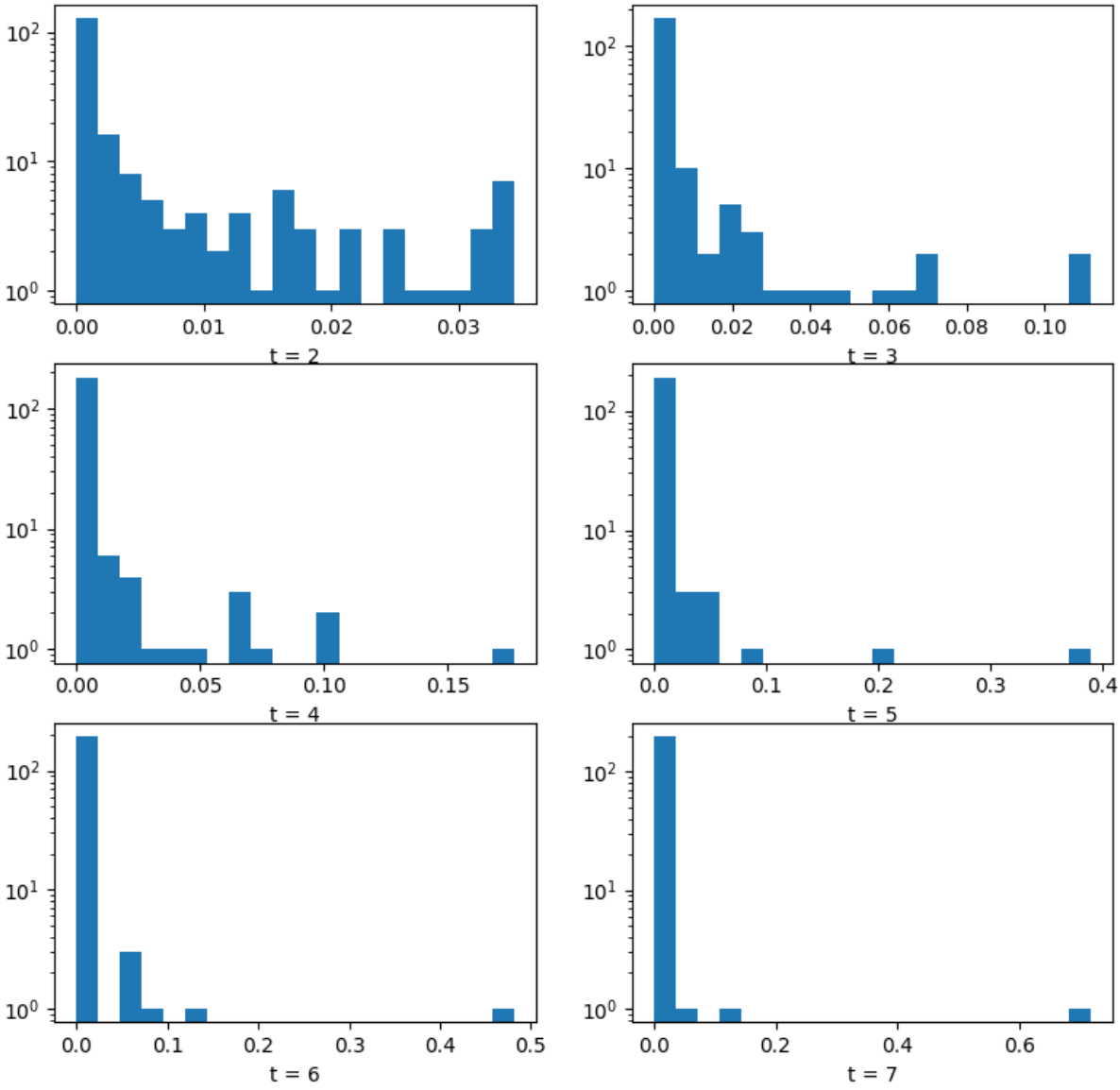

Figure 1: A histogram of 200 weights after just a few iterations. Almost all the weights are zero at $k = 7$ which demonstrates the degeneracy of the particles. {fig:weights}

The resampling methods presented here are two of the most popular amongst the literature, see [2]. The most simple resampling strategy, called *multinomial resampling* is not discussed here due to its poor performance compared to other techniques. It only is mentioned because it is the method introduced by Gordon et al. in [4] as part of the *bootstrap filter*, that uses the prior as the proposal density (c. f. Examples 1 and 2).

Both the 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 cumulative distribution function 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. 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

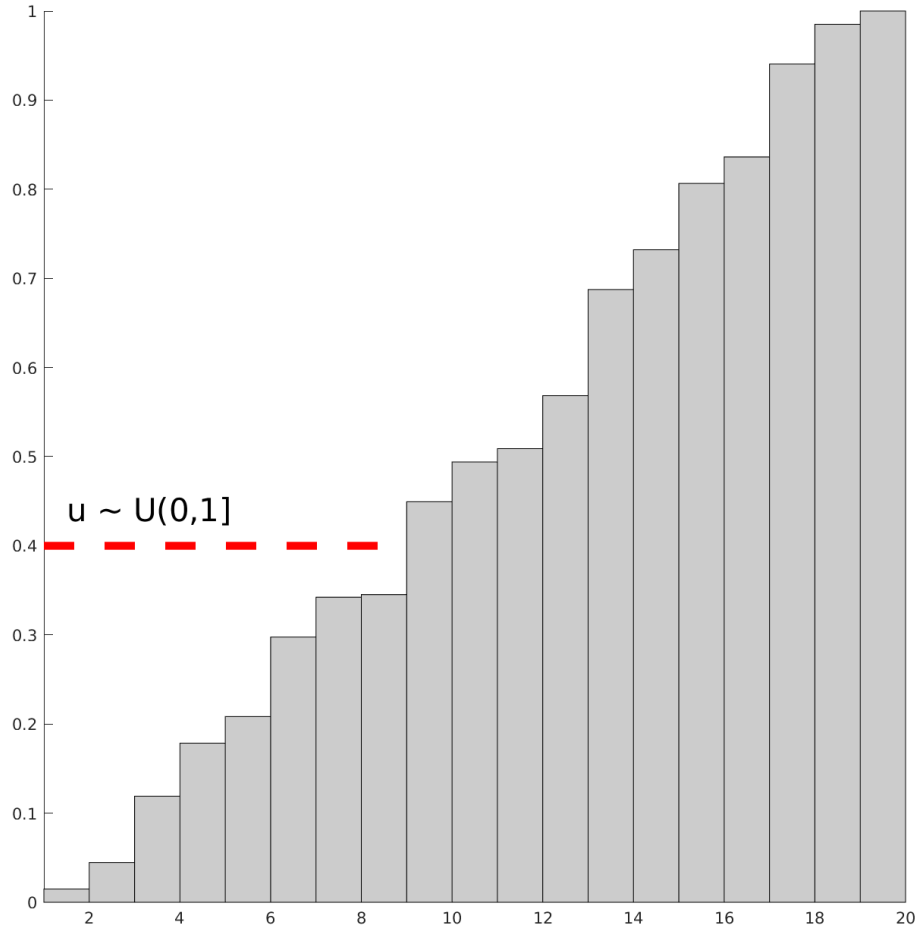

Figure 2: Visualisation of the inversion method. The bars represent the empirical cumulative distribution function associated with a set of 19 weights. The red line is horizontally drawn from the value of u at the vertical axis until it "hits" a bar. The index of the bar yields the generated sample (in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

{fig:ecdf}

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 cumulated sums`
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

{alg:resampling}

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]`.

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 used which can be estimated using

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

where the weights $w_t^{(i)}$ are assumed to be already normalised (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 some resampling strategies. Therefore, sometimes these methods are also called as*Sequential Importance Sampling with Resampling* abbreviated by SIS/R.

Implementation

In this section we discuss an implemenation of a particle filter in C++. The particle filter itself is implemented as a dependency-free[5] 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 [7] 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 namspace `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;
...
};
```

[5] The libary 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 [5].

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

```
template <class PT, class OT,
          size_t N,
          bool enable_history = false,
          bool parallel = false,
          typename... Args>
class ParticleFilter { ... }
```

where PT and OT denote the type of particle and observation, respectively, N is the number of particles, enable_history specifies whether some information from every timestep should be held in memory (e. g. for debugging or plotting, see below) and parallel specifies whether certain functions should run in parallel (see below). 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. As mentioned above, if parallel is set to true applications using the library have to be linked against Intel's TBB library [5]. Not using the parallel capabilities of the library does not change they way it has to be used but rather 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[6], a resampling treshhold and an initial seed for the random number generator (rng). 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,
    ResamplingStrategy t_strategy = ResamplingStrategy::RESAMPLING_SYSTEMATIC,
    double t_treshhold = 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_treshhold is used to decide when to actually perform resampling. At each

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

time step and estimate of the effective sampling size is computed using (5). The particles are only resampled if this value is below `t_treshhold` × `N`. Thus, `t_treshhold` 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 by 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 paramter 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.

{ex:1}

Example 1. This example has been studied in a number of publications before, see for example [1, 4, 6]. The implementation can be found in the file `apps/example1/main.cc`. Let

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

where

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

{eq:ex1:h}

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.[7] We choose $\sigma_{\text{sys}} = 10$ and $\sigma_{\text{obs}} = 1$.

sigma variance or sigma 2 std

[7] Later, we need multivariate Gaussian ditributions. The notation carrys over to the mutlidimensional case with σ replaced by Σ then denotes the covariance matrix.

Since both the particles and observations are one-dimensional we choose `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, centered 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_prior` and `m_gen` are defined as

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

The class `std::mt19937` defines a pseudo random number generator based on the popular *Mersenne Twister* algorithm.

To implement the remaining two methods we first have to choose a proposal density. For simplicity, we decided to 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

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

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 this function 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 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 with 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. 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`.

{ex:lv1}

Example 2 (Lotka Volterra using bootstrap filter)**.**

{ex:lv2}

Example 3 (Lotka Volterra using optimal proposal)**.**

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] 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.

[3] 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.

[4] 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.

[5] Intel. *TBB: Threading Building Blocks*. `https://software.intel.com/en-us/tbb`. Version 2020.1.

[6] 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.

[7] Kitware. *CMake*. `https://cmake.org/`.