A Model-Independent Backtracking Particle Filter Method in the PCRaster Python Framework

Rein Baarsma, 3235890

May 7, 2016

Abstract

Particle filters are an effective way to tackle data assimilation problems in the field of geosciences, especially when the model is either highly nonlinear or the measurement error can not be expressed as a Gaussian curve. The standard particle filter has been implemented into the PCRaster Python Framework which enables researchers to easily employ the method to data assimilation problems. However, the current particle filter method has no way to mitigate filter degeneracy. Spiller et al. (Physica D 237 (2008), 1498-1506) devised a way to repair a degenerated particle filter by reverting the filter to the last time it worked correctly and recalculate the posterior distribution with an increased sample size. In this thesis, a backtracking particle filter method has been constructed within the PCRaster Python Framework. The backtracking particle filter has been designed to be able to function with minimal intervention on the side of the model. The new method has been tested using two nonlinear stochastic models, showing an increase in efficiency. The method has some limitations when the stochastic forcing in the model is too high, but overall backtracking increases filter results. The backtracking algorithm is especially helpful when one of the update steps in the filter has unusually low observational error.

Utrecht University

Faculty of Geosciences

Heidelberglaan 2

3584CS Utrecht, the Netherlands

+31622352742

r.j.baarsma@students.uu.nl

Contents

1 Introduction

Numerical models in the field of geography have become increasingly complex as the computational power of computer systems has increased, as well as our understanding and information of the processes on the Earth's surface. This increase in complexity has brought with it challenges on the computational side as the models approximate the real situation much more accurately. The increase in computational strength has also given the opportunity to deal with uncertainty in our models, because running a model a large number of times with different parameters no longer necessarily requires much time. This creates the opportunity to use statistical analyses on the results of several model runs to study the integrity of our computer models.

Many geophysical systems show nonlinear behavior, particularly at small spatio-temporal scales as the turbulent behaviour of fluids can no longer be averaged out (van Leeuwen [2009]). Additionally, initial states of model runs are often quite difficult to estimate. Using nonlinear stochastic models these behaviors and uncertainties can be modeled in a Monte Carlo simulation as random variables, and the results of several model runs can yield an insight into the range and possibilities of the outcome. Although the power of current (super)computers is impressive, modeling nonlinear behavior in geophysical systems with their characteristic large state space can still be constrained by resources. This constraint is especially important when calculating time-sensitive information, for example atmospheric processes in order to predict the weather. After all, the prediction should show up before the weather does. In addition, highly nonlinear models like this are very prone to error accumulation because of the stochastic nature of (some of) the parameters. Because of this it has become necessary to use statistical methods to ensure the error in the results is as small as possible using as few model runs as possible.

A particle filter is a method that can be used to perform such statistical analyses, by feeding the model with observational data and using this data to update the model. Based on the principle of Bayesian updating, this method uses observational data to calculate how well individual model samples (or 'particles') are performing in a Monte Carlo simulation. Because the particle filter is a Monte Carlo-based technique it requires a large amount of computational power to perform all the model runs. In order to effectively use particle filters, they have to be implemented efficiently and statistical techniques have to be used to minimize the amount of particles that the filter requires in order to return adequate results (van Leeuwen [2003]).

Particle filters are computationally heavy algorithms, being that they are Monte Carlo simulations with an added statistical component. The Monte Carlo simulation propagates particles through time towards an update step, where the comparison is made between the model states of each particle and the observational data to assign a certain weight (Simon [2006]). It's imperative to use as few particles as possible while still maintaining a sufficient range of model results for comparison with measured data. A common problem to look out for is filter divergence, where too much particles accumulate error and stray from the measured data so when the comparison takes place only a small number of particles have any significant weight, while most particles are so far from the observation that they carry no weight at all, which means they hardly count towards the final result of the model. Computationally these particles are a waste, since the odds of them contributing to the final result are slim, but they are still ran for the entire duration of the simulation.

To combat filter divergence it is possible to resample the particles at the moment of comparison. Particles with a weight of (practically) 0 are deleted, and particles with a high weight are duplicated a number of times in accordance with their weight. When the weights of all particles are reset after this, the particles show approximately the same probabilistic trend as the weighted trend before resampling, where for instance a particle with a weight of 2η is now represented by two particles with a weight of 1η (η representing the average particle weight). The new particles can now be used for the simulation until the next comparison step. Even though the same amount of particles is used as in the standard particle filter, all particles that are used after each comparison step have a meaningful correlation with the measurement data and error is only accumulated until the next measurement comparison. This way more particles will be meaningful to the end result, and no computational effort is wasted on particles that calculate the wrong results (van Leeuwen [2009]).

Sometimes resampling is not enough, especially in large-scale applications. Filter divergence can still happen after resampling if the likelihood peak is very narrow, for instance when a large number of observations is present in the system (van Leeuwen [2003]). In this situation even though you resample after comparison with measurements, the error accumulation in the time between measurement steps is too large compared to the measurement error to prevent filter divergence, and the result has very few particles with large enough weight for resampling. Many different methods have been suggested to both prevent and combat this effect, including several resampling techniques. Another suggested solution to the problem is to go back to the time the filter last worked properly and run the simulation a second time for the same time period, this time with more particles. The particles will be reduced to the original number after the measurement comparison has been redone. This backtracking particle filter is best used for models where the reward for using as little particles as possible is very large, such as large state-space models that require a long time to run.

There are many other methods to combat filter divergence, particularly resampling schemes such as the Guided Sequential Importans Resampling scheme (van Leeuwen [2009]). Compared to these methods, backtracking is regarded as a brute force method. However, as the goal of this research is to implement a new method into an existing modelling framework, it is important for the method to be largely model independent. Most resampling methods require insight into the model state, which would force the modeler to write code specifically tailored to the model that the particle filter method is applied to. The aim is to create a backtracking particle filter method that is largely automated, requiring little to no extra input compared to a standard particle filter.

This thesis will aim to create a backtracking particle filter method for use in spatiotemporal modelling. The research goal of this thesis will be two-fold. The first research goal will be to create a backtracking particle filter and to analyse the performance of this filter when applying it to a nonlinear model. In order to analyse the performance for such a particle filter it will have to be compared to a different particle filter technique. It has to be studied how to find the optimal amount of particles necessary to find an accurate solution using both the standard particle filter and the backtracking particle scheme.

The second goal is to implement this particle filter into an existing modelling framework. In doing so it will be available for future research in such a way that no extensive programming experience is necessary to run a backtracking particle filter simulation on a model. The final product should strike a balance between ease of use and flexibility, for instance having the capability of easily switching resampling schemes or conditions for triggering the backtracking process. Since the goal of the backtracking scheme is to run a particle filter with as little computational resources as possible, the programming will have to be done in such a way that the calculations are made as fast as possible.

2 Background

2.1 The Standard Particle Filter

Particle filters are a Monte Carlo simulation method based on Bayesian statistics, so it is important to first explain Bayes' Theorem and its implications before explaining particle filters, as well as the Monte Carlo simulation technique. Then the statistical theory behind particle filters will be discussed, as well as the limitations of the technique and discuss some of the techniques used to deal with these limitations. Special attention will be given to the backtracking particle filter method, as the filter created in this thesis is based on the backtracking particle filter. The ideas in this section are more broadly applicable than just in the geophysical sciences, so examples will be given from both in- and outside the field.

2.1.1 Bayes' Theorem

The joint probability of two events occurring $Pr(A,B)$ can be expressed as

$$Pr(A,B) = Pr(A)Pr(B|A)(= Pr(B)Pr(A|B)) \tag{1}$$

where $Pr(A)$ is the probability of event A occurring both with and without event B and $Pr(B|A)$ is the probability of event B occurring given that event A happens. For instance, if you have seven coins in your wallet, the probability of a random draw getting you a German Euro coin is equal to the number of German coins (4/7) times the number of Euros among those German coins (1/4). From (1) the following can be inferred:

$$Pr(A|B) = \frac{Pr(A)Pr(B|A)}{Pr(B)} \tag{2}$$

which states that the conditional probability of an unobserved event A occurring given an observed event B is equal to the probability of event B occurring given that event A has occured or is occurring times the relative probability of A compared to B. This is Bayes' Theorem, and in essence it describes the process of updating a degree of belief in a hypothesis based on new evidence (Howson [1990], Gelb et al. [1974]). Imagine your new stove giving you a shock. You now want to know if your stove is faulty ($Pr(Faulty|Shock)$). You read that 99.5 percent of faulty stoves give consumers electrical shocks and that 1 in 1000 ovens is faulty, so $Pr(Shock|Faulty)$ is 0.995 and $Pr(Faulty)$ is 0.001. If the odds of a person getting shocked by their stove for whatever reason is 1 in 100, the probability that your stove is faulty given that it gave you a shock is 0.1. So even though the initial belief was that the probability of a faulty stove was 1 in 1000, when it shocks someone the probability becomes a 100 times higher. Still, because so little stoves are faulty, there is still 90% possibility our stove is fine.

In the above example of Bayesian inference the formal interpretation of the values used is as follows: our initial belief in the hypothesis of a faulty stove $Pr(Faulty)$ is called the *prior*, and the final result $Pr(Faulty|Shock)$ is called the *posterior*. The quotient $Pr(Shock|Faulty)/Pr(Shock)$ represents the support the shock event provides for the hypothesis that the stove is faulty. This quotient works as follows: as long as $Pr(B|A)/Pr(B)$ is greater than one (or $Pr(B|A) > Pr(B)$), event B confirms or supports event A. If $Pr(B|A) < Pr(B)$ event B undermines event A, and if $Pr(B|A) = Pr(B)$ event B is neutral with respect to A. In general this means that any event B provides support for event A as long as event B is rare and has a high dependence on event A. In the above example, event B (the shock) could not muster enough support for event A (a faulty stove) because getting a shock from a stove was too common an occurrence compared to a faulty stove to provide support for the hypothesis.

Bayes theorem does not just work with simple probabilities but it can be used for probability density functions as well. The prior probability density is then updated with a likelihood derived from observation to form a posterior probability density function. So in these cases, we are no longer talking about an observed event providing information on a different unobserved event, but rather an observed quantity providing information on a larger unobserved system (Hobbs and Hooten [2015]).

A probability density function (or pdf) $[z]$ of a continuous variable z has the following characteristics (Gelb et al. [1974])

$$[z] \quad \geq \quad 0 \tag{3}$$

$$Pr(a \leq z \leq b) = \int_a^b [z]dz \tag{4}$$

$$\int_{-\infty}^{\infty} [z]dz = 1 \tag{5}$$

all of which are quite logical considering $[z]$ represents a probability. An important thing to note is that while the area under the curve (5) is 1, the range of $[z]$ is $[0, \infty >$ (Hobbs and Hooten [2015], Gelb et al. [1974]).

Going back to our stove, one could use Bayesian inference to research whether the stove's temperature gauge is working correctly. When you set the stove's temperature gauge to 200ºC, the expectation will be that the temperature is going to settle at around 200ºC but it won't be exactly that. You can test this by using an oven thermometer, but this oven thermometer has errors of its own, and you can't read the exact temperature from the dial. It is important to note that no updating statistic can work if there is a systemic error in the observational data. In this experiment, the prior probability density function could be taken as a Gaussian curve with a mean oven temperature T^o at 200ºC, and the likelihood from a series of measurements from the gauge temperature T^g is used to calculate a posterior probability density function of the oven temperature. From (2) we obtain

$$[T|T^g] = \frac{[T^o][T^g|T^o]}{[T^g]} \tag{6}$$

We have no clue as to the error in measurement of the temperature gauge, but as long as we take a broad prior probability density function $[T^o]$ that includes $[T^g]$ we can use the law of total probability to obtain

$$[T^g] = \int [T^g|T^o][T^o]dT^o \tag{7}$$

and express $[T^g]$ as a probability density within the model probability density $[T^o]$ (Hobbs and Hooten [2015]). In other words, this can work as long as the model is able to predict the results from the measurements, which is the first and most obvious assumption of updating using Bayesian statistics. The denominator in (6) becomes a normalization factor for $[T^o|T^g]$. The law of total probability works if the probability density of $T^g$is contained in the probability density of T^o. The normalization factor has an interesting implication for $[T^g|T^o]$, because it does not have to be a proper pdf, in that it does not have to integrate to 1 (van Leeuwen [2009]). The denominator in (6) ensures that $[T^o|T^g]$ is a proper pdf regardless of the form of $[T^g|T^o]$.

Another challenge when using the Bayesian model is selection of a prior. When little is known about the state space prior to the measurements it is imperative to select a prior that provides as little information to the posterior distribution as possible. It is impossible to have a completely noninformative prior, so prior selection is very important. A prior usually has to be a proper probability density function in order for the Bayesian method to be valid. An improper prior (one that does not integrate to 1 over its domain) can technically work, but unless mathematically proven this is very unlikely. With enough data the prior takes a backseat though, so when little is known about a parameter it makes sense to collect as much data as possible to create a sound posterior (Hobbs and Hooten [2015]).

On the other hand, an informative and well-justified prior can have a significantly useful effect on the posterior. It's also very rare to have a model of a parameter with completely unknown values. At the very least the range of values the parameter can take is known. There are very little geophysical phenomena that have not been studied at all, and one of the big advantages of the Bayesian approach over other statistical methods is its ability to incorporate results of previous studies into results. This makes it possible to draw meaningful conclusions even without superfluous amounts of data. The posteriors of older Bayesian studies can be used as priors in current research just as well as other types of data, which is a cornerstone of the Bayesian method of scientific reasoning(Hobbs and Hooten [2015]).

Using Bayesian statistics to view geophysical systems can be very useful, and a powerful tool in researching it. Bayesian statistics view unobserved values as random, and observed values can then be used to inform on the probability density function of that unobserved parameter (van Leeuwen [2009]). One could argue that unobserved values are not random but very much determined, and perhaps determined by several different values. Consider for instance the amount of water present in a watershed. This amount of water is at any point in time determined by several factors such as reservoir size, rainfall, evaporation, transpiration, upwelling and the amount of outflow through channels. So the amount can be determined. The amount of water in a watershed is difficult if not impossible to measure directly however not to mention constantly changing, so it is easier and more generally applicable to see it as a randomly distributed variable.

2.1.2 Particle Filters

Particle Filtering is a method used to update Monte Carlo simulations of a nonlinear (or linear) model with observational data using Bayesian statistics to achieve insight into a hidden state of the model (Hobbs and Hooten [2015]). This hidden state is a quality or quantity the model calculates, but is unobserved or unobservable in its realistic counterpart (the experiment or natural phenomenon the model mimics). A Monte Carlo simulation consists of many individual model runs, hereafter referred to as particles, that are sampled from the Bayesian prior probability density function. These particles are then used to make a prediction about the posterior distribution by running the particles through the model. This predicted posterior distribution is then compared to observational data to create a true posterior distribution. Particles are weighted according to their posterior probability. The observational data corresponds with a (partial) result of the model, so seeing which particles return results close to the observational data yields insight into the parameters that we can not compare to observations, i.e. the hidden states of the model (van Leeuwen [2009]).

Consider a stochastic model where the complete model state is denoted by ψ, where variables ν_x are expressed by probability density functions. The aim is to find the joint probability density of all variables conditional to the observations $[\psi|d] = [v_1|d, v_2|d, ..., v_n|d]$. The model is occasionally updated using observations d when they become available using Bayes' Theorem to obtain $[\psi|d]$. A prior probability density function $[\psi^0]$ is defined, where superscript 0 denotes that this is the prior pdf before the first time step, made up of any prior knowledge on the parameter values. From this prior pdf N particles are sampled. These particles have to be ran through the model up to the timestep where the observational data is made (t_u). The Bayesian model comparison at t_u can now be calculated by comparing each particle to the observations. Equation (6) becomes

$$[\psi(t_u)|d_u] = \frac{[d_u|\psi(t_u)][\psi(t_u)]}{[d_u]} \tag{8}$$

where the prior term$[\psi(t_u)]$ is actually the predicted posterior pdf of the model state, created by running the particles through the model up to timestep t_u. In formal statistical terms this process is expressed as sampling from $[\psi(t_u)|\psi_i(t_0)]$ for each particle i. The law of total probability can be used here to create (Hobbs and Hooten [2015])

$$[\psi(t_u)|d_u] = \frac{[d_u|\psi(t_u)][\psi(t_u)]}{\int [d_u|\psi(t_u)][\psi(t_u)]d\psi(t_u)} \tag{9}$$

It is important to note here that this implies that the model at time t_u can predict $[d_u]$. This may seem trivial but if erroneous assumptions are made about the range of parameter values or there is another fundamental flaw in the model, it is entirely possible to create a geophysical model that is unable to recreate realistic values. This limit to the particle filter method is the reason that testing the particle filter used in this thesis is not done with real world values, but rather with values generated from the model itself.

The probability density function of the model is approximated by the use of the particles in the form

$$[\psi] = \frac{1}{N}\sum_{i=1}^{N} \delta(\psi - \psi_i) \tag{10}$$

where N is the total number of particles and $\delta(\psi - \psi_i)$ is the Dirac delta function of the difference between the model state and the model state of particle i (Simon [2006], Nanako et al. [2007]). Practically this function describes a space where the probability of a model state existing is 0 except if the model state is equal to the model state of particle, where the probability is $1/N$. Using equation (10)'s definition of $[\psi]$ in equation (9) yields

$$[\psi|d] = \sum_{i=1}^{N} w_i \delta(\psi - \psi_i) \tag{11}$$

with w_i being the weight of particle i, calculated as

$$w_i = \frac{[d|\psi_i]}{\sum_{j=1}^{N}[d|\psi_j]} \tag{12}$$

so $[\psi|d]$ is no different from $[\psi]$ in that it is an accumulation of Dirac delta functions, except the peaks at every ψ_i are of different size according to their proximity to the measurement data. To calculate the top half of this equation, the observational data can be compared to the model states using

$$w_i = \exp\left(\frac{(\bar{x} - \psi_i)^2}{-2\sigma^2}\right) \tag{13}$$

where $\bar{x}$ is the measurement average, ψ_i is the model state of particle i and σ^2 is the measurement variance. The bottom half of equation 12 simply normalizes these values.

In the next iteration of the observational update at timestep t_v, the weights will be carried over to the calculation of the prior. So

$$[\psi(t_v)] = \sum_{i=1}^{N} \frac{[d(t_u)|\psi_i(t_u)]}{\sum_{j=1}^{N}[d(t_u)|\psi_j(t_u)]}\delta(\psi(t_v) - \psi_i(t_v)) \tag{14}$$

With the posterior pdf calculated in the particles, it is possible to calculate mean, median and mode model states, as well as functions of the model state. In the standard particle filter method this is all that is done with the particles. Note that in this method no particles are deleted and the model state of the particles is in no way affected by the calculation of posterior probability. The issue with this method is that after a couple of update steps, most weight will be assigned to a very limited number of particles, and most particles will carry little to no weight (Künsch [2013]). This problem is called *filter divergence*. Any sequential importance method has this problem. Given enough time, a particle filter of any model will degenerate (Doucet et al. [2000]). To ensure the statistical validity of the Monte Carlo simulation a large number of particles is required in the standard particle filter. This requires computational power and consequently time, most of which goes to calculations on particles that do not carry any weight at all in the posterior distribution.

Employing different resampling methods can prevent filter divergence. The main idea of most of these resampling methods is to take the weighted posterior and recreate this posterior using particles with equal weight. Doing this exactly is impossible unless there are an infinite number of particles, so this method adds some *sampling noise* to the prior. Because adding sampling noise means changing the shape of the prior, it is important not to resample too often (Spiller et al. [2008]). Van Leeuwen[2009] gives an overview of several resampling methods used in the field. *Probabilistic resampling* is a method where the new particles are sampled from the posterior pdf. Because the posterior is discretisized (see 11), the samples are exact copies of existing particles (Doucet [2001]). *Residual sampling* can be done by multiplying the weight of each particle by N, of wich the integer part yields the amount of copies that need to be made of that particle. The remaining particles are sampled from a probability density obtained from subtracting the integer part of each Nw_i from it so that all values are between 0 and 1, and using these new values as weights for the probability density. In that probability density, particles with a high chance of getting sampled had either a Nw_ivalue close to 1 or a value just under an integer (van Leeuwen [2003]).

In *stochastic universal sampling* all particles gain a section of the line $[0, 1]$ with a length proportional to their weight. The idea now is to overlay this line with a line of equal length where the line pieces have length $1/N$ rather than a weighted length. When a line piece of the second line end in a region of the first, the particle is chosen for resampling. A large line piece in the first line will be chosen multiple times, while the small pieces concordant with a lowly weighted particles will most probably not get chosen. In order to prevent a systemic problem with the order with which the particles in the first line are arranged, the second line is offset by a random amount between 0 and $1/N$ (Kitagawa [1996]).

After the resampling, the weights of all particles are reset to $1/N$ so the probability density of the particles mimics the posterior calculated in the weighting step.

This method is effective as long as the model has a strong enough nonlinear component to diverge the particles in between the updating steps. If the copied particles do not accumulate enough error to be considered unique at the next observation step, a phenomenon called *particle collapse* occurs (van Leeuwen [2009], Xiong et al. [2006]). In particle collapse situations, all the particles are roughly the same, so the probability density function becomes too narrow to yield credible information. The result is very similar to filter divergence, even though the effect itself is opposite; instead of all the weight being carried by one particle, all particles have the same amount of weight because they contain the same information.

In order to prevent particle collapse a method called *regularization* can be applied that will decrease the number of identical particles after resampling. The idea is to give the particles a 'jitter' after resampling, changing the parameter values that are used as variables in the particle filter in such a way that the particle is not identical to its root (Pham [2001]). Alternatively, a Merging Particle Filter can be used that creates new particles by mixing the states of succesful particles (Nanako et al. [2007]).

2.2 The Backtracking Particle Filter

The backtracking particle filter method is different from other methods in that it does not prevent filter divergence, but rather reacts to it. When filter divergence is detected, the backtracking particle filter goes back to the last time the filter worked correctly and tries again. Spiller et al. [2008] constructed several such filter for a two-dimensional point-vortex model. The model includes a 'tracer', of which the location is artificially measured and used as the observational data. These filters performed quite well compared to standard particle filter methods, which eventually lose track of the tracer in the point-vortex model.

At evaluation timestep t_e a backtracking filter evaluates the population of particles on its distance to the observations. If the particle population differs too much from the observations, the backtracking algorithm is triggered. Spiller et al. [2008] do this by comparing the population mean and covariance matrix of the particle cloud with the average from the observations and calculate a discrepancy factor. If the backtracking algorithm is triggered, the filter goes back α timesteps to the last evalutation timestep $t_{e-\alpha}$and doubles the amount of particles to recalculate the posterior at timestep t_e. It is important to note that a goal of effectively using a backtracking particle filter is to initiate the backtracking algorithm as few times as possible, so it is entirely possible that rerunning the filter from timestep $t_{e-\alpha}$ without doubling would yield a workable posterior. The backtracking algorithm is there to help the particle filter through timespans where the model adds an exceptionally strong noise to the particles between evaluation timesteps, or when the likelihood band from the observational data is very small.

It is also possible to evaluate the health of the particle cloud from the particle weights directly. When filter degeneracy occurs, there are not enough particles with significant weight to yield meaningful results. As such, a successful particle population should always have a number of particles with high weight. A sorted list of normalized weights of particles should not have most of its total value in the highest few particles, but rather a larger number of unique particles should have higher weight.

Spiller et al. [2008] tested three methods of doubling . In the first method, *cloud expansion*, the N particles used to create the failed posterior are reused, and N additional particles are created by copying each particle and adding independent mean zero Gaussian random variables (with small variance) to each of the observed states in the original N particles. This yields $2N$ particles that are then run through the model again.

The second method is called *directed doubling*. The $M = N/10$ particles with the highest weight are taken from the prior distribution at the time the particle filter last worked correctly. Now a line between the model states of each of these M particles and the observations is calculated, and N samples are taken on the lines between the observations and the model states. For each of the N new samples, a mirrored particle is created on the same line at the same distance from the model state, but away from the observations. This method is not used in the backtracking filter of this thesis, as it requires the algorithm to alter model states specifically, which makes this not generically applicable to models, and raises new research challenges such as how to perturb variables that are not directly measured.

The third method is to not perturb the posterior when doubling the amount of particles by resampling succesful particles, but rather perturb the observations prior to resampling. The observation Y_b at the backtracked time t_b is perturbed by a zero mean Gaussian $2N$ times, creating $2N$ realizations of Y_b'. These new observations are used to create $2N$ particles from the prior pdf. These particles are then propagated toward the time where the particle filter failed.

Spiller et al. [2008] tested their filter method (with each doubling method) against an Ensemble Kalman Filter and a standard particle filter, each realized with 500 particles. As expected, the Ensemble Kalman Filter performs less well than the particle filter methods, as the nonlinearity of their point-vortex model is too high for the EnKF to be effective. All particle filter methods that are used are effective, but when the time between observational updates gets high or the nonlinearity is very strong, the standard particle filter fails. In the study by Spiller et al. [2008], no significant difference between the different doubling methods has been observed.

2.3 PCRaster Python Framework

PCRaster is a grid-based modelling framework with a strict data type checking mechanism that lends itself very well for geospatial modelling. Spatial data is stored in a binary map format, that contains a header with the dimensions of the map and the type of data that is stored alongside the data of each cell. In the PCRaster format, cells can receive and transmit data from neighbouring cells. Analysis and manipulation of maps can be done using PCRaster operations, which can be either point operations where the value of the result is only dependent on the value in the input map(s), or neighbourhood operations where the values from any number of neighbouring cells in the input are also used in the calculation of the resulting cell. Alongside individual operations, operations can be scripted to perform a large number of calculations at once (de Jong and Karssenberg [2015]).

PCRaster map files can contain boolean, nominal, ordinal, scalar, directional or local drain directional data, and the PCRaster package contains a large scala of operators that can be used to manipulate, compare, analyse and generate these maps. Maps can be imported from and exported to other GIS formats. Along with the basic operators, PCRaster contains some functions dedicated to the field of hydrology and geomorphology such as flood wave propagation equations and visibility analysis.

PCRaster also contains a dynamic modelling module to calculate and store changes over time in attributes of maps that uses the same GIS database as other operations (Verstegen et al. [2012]). A PCRaster dynamic model is a set of scripted operations to maps, some of which are repeated every timestep using the results of the previous iteration. Specific to the dynamic modelling module in PCRaster is the time series data format, which can be stored as either a series of maps that shows the change of a certain map attribute over the modelled time period, or a table that shows the time change at a specific location.

To create more freedom of use PCRaster has been released as a Python module, enabling the map format of PCRaster to be read and written by Python operation. In addition to this a framework has been developed that enables the use of static and dynamic modelling in Python scripts, as well as Monte Carlo simulations, Particle Filtering and Ensemble Kalman Filtering (Karssenberg et al. [2010]). Python's Object Oriented Programming methods make it quite easy to implement new methods in the PCRaster Framework. The Monte Carlo simulation runs are governed by a class that incorporates the static or dynamic modelling class, and the Particle Filter class incorporates the Monte Carlo class. This enables the user to quite easily turn a dynamic model into a Monte Carlo simulation, simply by adding the required methods and initializing a Monte Carlo Model object.

The Python programming language makes it quite simple to implement these modelling schemes into research, because the language is easy to learn and read due to its intuitive syntax and intelligent use of functional whitespace. The Python language is therefore a very useful programming language for research outside of the field of computer science. Python aims to be easy to learn and use as well as powerful so programs can be built by the people who need to use them, rather than by independent professionals.

3 Methods

The particle filter method and stochastic models used to test the models were all created in the PCRaster Python framework. This framework already contains the methods to create and run a stochastic dynamic model, a Monte Carlo simulation, and a standard particle filter. The framework uses the operators from the PCRaster Python module, which allows advanced GIS operations to be performed. To create the backtracking particle filter, a new class was created within the framework. This method uses the functionality of the standard particle filter but adds and alters the function within the method in order to enable the filter to backtrack.

3.1 The PCRaster Python Framework

Python's Object Oriented Programming enables a wide range of modelling features. In Python, classes can be constructed. These classes can have methods and parameters associated with them, and child classes can be created that inherit these methods and parameters. To run a PCRaster model, a class is created with a parent from the PCRaster Python Framework. An object is created that initializes the class, and the 'run' method is called on this object.

A PCRaster Python model is created by using a class constructor, and adding the methods to this class that are specific to the model. For instance when creating a dynamic model, the DynamicModel class constructor is used from the PCRaster Python Framework. The new class inherits all the methods from the DynamicModel class, which includes a 'run' method. The new class now needs two methods, 'initial' and 'dynamic', that will be used by the 'run' method when it is called. The 'initial' method will be called once during the model run, and then the 'dynamic' method will be called for every timestep. When the model class is constructed, an instance of the class is called that determines the number of timesteps that are calculated, and then the 'run' method is called on that class.

From there, creating a Monte Carlo model is quite simple. The MonteCarloModel parent is added to the model class, the methods specific to the Monte Carlo method are added to the class. To run the Monte Carlo model, a dynamic model is initialized, and then a Monte Carlo model object is initialized using that dynamic model object. A particle filter is initialized in the same way using an instance of a Monte Carlo model.

3.2 The Particle Filter method

The theoretical particle filter method is explained in the Background section of this thesis. The particle filter as implemented in the PCRaster Python Framework is discussed here. In the framework, a particle filter run is a Monte Carlo run with several filter steps. The timesteps at which the filter is updated are defined when the particle filter object is initialized. At the update step, the particle states are compared to the state of measurement data. In the PCRaster Python particle filter, this comparison method has to be written by the modeller, as the type of data and the way the data is used is unique to each model.

When the particle filter runs, a Monte Carlo run is started for the timesteps until the first filter period. The prior distribution is described by the samples distribution a $t = 0$. The projected posterior distribution is calculated by propagating the samples through the dynamic model towards the first timestep. At this point, the Monte Carlo run is suspended. During suspension, weights are assigned to each particle based on the measurement data given. The posterior distribution is shown by the sample weights. This is the Bayesian update step. After this resampling takes place and the weights are reset. The posterior distribution of the particles is now. Two resampling methods are implemented in the framework: sequential importance filtering and residual sampling filtering (de Jong and Karssenberg [2015]). After resampling the filter step is complete, and the Monte Carlo simulation is resumed until the next filter time.

During sequential importance filtering, weights are normalized so the sum of all weights is 1, and a list of the cumulative weights is made which goes from 0 to 1. Then a random number between 0 and 1 is drawn from a uniform distribution. This number is compared to the list of cumulative weights, identifying the index of the first particle whos value is higher than the random number. This particle is then resampled. This process is repeated until the desired amount of particles is resampled. If a particle has a very low weight, it has a low chance to be resampled, whereas a particle with a high weight is likely to be picked a number of times.

In residual sampling the weights are again normalized but are then multiplied by the number of particles. Normalized weights have weight between 0 and 1, and a cumulative weight of 1. So the new weights have a value between 0 and N with a cumulative weight of N. Each particle with a value above 1 is resampled x times, where x is the integer part of its value. When this is done, a number of samples still needs to be taken. These samples are taken from the residual distribution. The integer part of each value is subtracted, so each value is between 0 and 1. This list is then used to sample the rest of the particles using the sequential importance sampling method above.

3.3 The Backtracking Particle filter

The backtracking algorithm was created based on the existing particle filter in the PCRaster Python Framework. The backtracking particle filter takes an extra argument when initialized that determines the amount of particles that are used when backtracking occurs.

Compared to the basic particle filter, the backtracking algorithm adds four extra steps to the particle filter process. The first occurs after the weights have been calculated during the update step, when the algorithm tests whether backtracking needs to occur. The evaluation takes the list of weights and sorts and normalizes this list. Then a list of cumulative weights is calculated. If 90% of the cumulative weight of the particles is contained in the highest 5% of total particles, the filter is considered to have failed, and backtracking is triggered.

The other three extra steps in the algorithm are part of the backtracking itself. First, the number of samples needs to be increased for the backtracking from N samples to M samples. We first create $n \cdot N$ new samples by copying the original samples (where n is the integer part of M/N). The rest of the particles are sampled randomly from all particles created. If the model fails at the first filter timestep, the entire sample cloud is reinitialized from the beginning. This is close to what Spiller et al. [2008] refer to as *cloud expansion*, without introducing any Gaussian noise to the particles. If the backtracking particle filter is applied to a model with strong dynamic stochastic forcing, adding noise to the particles is redundant. The method allows this to be done if the model has no stochastic forcing.

After this, the Monte Carlo simulation is run back up to the filter time step and the new posterior is calculated. This new posterior consists of M particles and has to be translated back to the original number of particles to continue the particle filter run. This is done by taking the N particles with the highest weight and using these as the new posterior. This posterior is used to resample N particles that are used in further calculation.

Simple population model		Complex snowfall model	
Parameter	Value	Parameter	Value
Initial Population	0.1	DEM	input map
Maximum Population	1.0	Degree Day Factor	0.01
Growth factor	$1.0 + U(0, 2)$	Temperature	input timeseries
Stochastic Forcing η	$1.0 + \phi * 0.05$	Stochastic Forcing η	$1.0 + \phi * 0.2$
Observation Std Dev	$Obs \cdot 0.01$	Temp. Lapse Rate	$0.04 + \phi * 0.00025$
		Precipitation	input timeseries spatially perturbed by $P = P_{ts}(1 + \phi * 0.5)$
		Observation Std Dev	$Obs > 0 \, : \, Obs * 0.15$ $Obs = 0 \, : \, 0.01$

Table 1: Model parameter values for the two models used in testing the Backtracking Particle Filter. For parameter values with a stochastic forcing component, $U(a, b)$ denotes a continuous uniform distribution between values a and b, and ϕ denotes a standard normal distribution with $\mu = 0$ and $\sigma = 1$.

3.4 The Dynamic Models

Two dynamic stochastic models were used in testing and creating the backtracking algorithm. At first a simple nonlinear point model was used. Using a point model makes it easy to control the model environment and force the backtracking conditions to trigger. After this, the model was tested using a more complex geophysical model that simulates snowfall and snowmelt. This model resembles the kind of models that the algorithm might be used for in the future.

3.4.1 Simple population growth model

The first is a nonlinear point model based on the population growth equation

$$P_{t+1} = [g \cdot P_t(P_{max} - P_t)]\,\eta \tag{15}$$

where P_t is the population at time t, $P_{max}$is the maximum population, g is a growth factor, and η is a stochastic forcing (see Table 1). To test the backtracking algorithm, the model was run once as a dynamic model, creating results for the Monte Carlo simulation to compare itself with. The goal of the Particle Filter is to correctly ascertain the growth factor g by comparing the populations at the filter timesteps. The weight of each particle at the filter update step is calculated using equation 13.

The measurement average is here taken as simply the model results from a sample dynamic model run, and the variance is imposed as a function of the measurement average (see Table 1)

The dynamic model run that creates the measurements is almost entirely deterministic. There is a small perturbation in the dynamic population equation to insert measurement error. The samples are initialized with a growth picked from a normal distribution around the value from the control run. When the model run was done that serves as the observational input for the model, the stochastic forcings used were slightly less pronounced than in Table 1, to ensure the measurements fall well within the scope of the particle filter.

3.4.2 Complex geophysical snowfall model

The snowfall model used is a demonstration model available from the PCRaster website. The model uses a small elevation model, a temperature measurement for each timestep and a precipitation measurement for each timestep to calculate the amount of snowfall and the amount of snowmelt, correcting the temperature using the elevation model and a temperature lapse rate.

The model takes input timeseries of temperature and precipitation, and calculates a local temperature map based on the elevation above the measuring point and the temperature lapse rate. Each sample in the particle filter run has a different temperature lapse rate taken from a uniform distribution (see Table 1). The precipitation is locally perturbed to create a noisy rainfall pattern. Snowcover is then calculated as

$$S_t = [S_{t-1} + S_f - M]\,\eta \tag{16}$$

, where S_t is the snowpack at time t, S_f is the amount of snowfall, M is snowmelt and η is a stochastic forcing (see Table 1). Snowfall S_f is equal to the precipitation as long as the the temperature is below 0, and snowmelt M is calculated as

$$M = \left\{ \begin{array}{c} T > 0\,;\, T \cdot d_f \\ T \leq 0\,;\, 0 \end{array} \right\} \tag{17}$$

where T is the temperature and d_f is the degree day factor.

The weight of the particles is calculated using the calculated snowcover and equation 13, except the model differentiates between several spatial zones, calculating a different average for each of these. Equation 13 becomes

$$w_i = \exp\left(\sum_{z=1}^{K} \frac{(\bar{x_z} - \psi_{i,z})^2}{-2\sigma_z^2} \right) \tag{18}$$

for K zones.

The temperature lapse rate is calculated during the sample initialization from a normal distribution. The precipitation imposed on the model during the run is perturbed using a normal distribution, creating stochasticity during the model run. Additionally, model uncertainty is introduced on the total amount of snow present after the full calculation by imposing a stochastic component.

While testing with this model initially particle collapse would occur consistently. No matter the amount of particles used, by the 3rd filter update the model would have one, sometimes two unique values for the temperature lapse rate. In order to combat this, a 'jitter' was introduced. When the model is resumed after resampling, a small stochastic force is added to the temperature lapse rate, ensuring eacht particle is unique.

4 Results

The backtracking particle framework was tested using the two models explained in the Methods section. For each of model, particle filter runs were made with 25, 50, 70, 100, 150, 200, 300, 500 and 1000 particles. The amount of auxiliary particles used in the backtracking filter were kept at double the amount of regular particles.

Which each model, the results of some runs is used to show the filter performance through time. For the rest of the runs, only the particle states of the final filter timestep are used. In the case of the backtracking particle filter, this can be a backtracked run, so the final filter timestep can have twice the number of particles in their population.

4.1 Backtracking Framework technical implementation

The backtracking particle filter algorithm could be implemented into the existing PCRasterPython Framework with relative ease. The base frameworks for running dynamic models and Monte Carlo simulations, on which the particle filter framework is dependent, were left unaltered as the basic methods to enable backtracking were already present.

Compared to the existing particle filter framework, the backtracking particle filter framework significantly changes only two methods, and adds two of its own. When an instance of the backtracking particle filter framework is initialized, folders need to be added in order to house the particles that are run during backtracking. For instance, a standard particle filter using 100 samples creates 100 folders, 1 to house each sample. A backtracking particle filter creates the same amount, as well as an amount of folders that are used to copy the particles into when backtracking. The 'run' method, which determines what actions are taken while the particle filter is ran had to be altered to allow for a test to see whether backtracking needs to occur, and if it does, to run the backtracking.

The backtracktest and backtracking methods are the two methods added to the particle filter framework. The backtracktest algorithm takes a list of weights, normalizes and sorts them, and calculates a list of cumulative weight. The resulting list starts with the highest weighted particle and ends with the lowest. The algorithm then looks at the value of the n$^{\text{th}}$number in the list, where $n = N/20$, and if it's value is above 0.9, meaning more than 90% of the total weight of the particles is carried in the n highest particles, the algorithm triggers backtracking.

In the backtracking method, first the sample numbers the filter uses are changed from $[1, N]$ to $[N, N+M]$ to make sure the calculations are put in the correct folders, and the correct number of samples is used by the monte carlo simulation. The original samples are then copied into the new folders used by the backtrack algorithm, copying the entire original population as often as it can, and any remaining particles are randomly sampled from the original population.

The backtracking algorithm then runs the Monte Carlo Simulation again, just as the simulation was run in the regular 'run' method, and calculates the weights based on the method from the model. The N highest weighted particles are identified and copied back to the folders between $[1, N]$. Before finishing, the algorithm undo's all the changes that were made in order to run the Monte Carlo simulation in the backtracking folders and puts the weights of the copied particles in the list used by the resampling method. The 'run' method then resumes as normal with the resampling.

On the user end, there is very little difference between a particle filter and a backtracking particle filter, as the only additional parameter a user needs to add to the code is the amount of particles to be used while

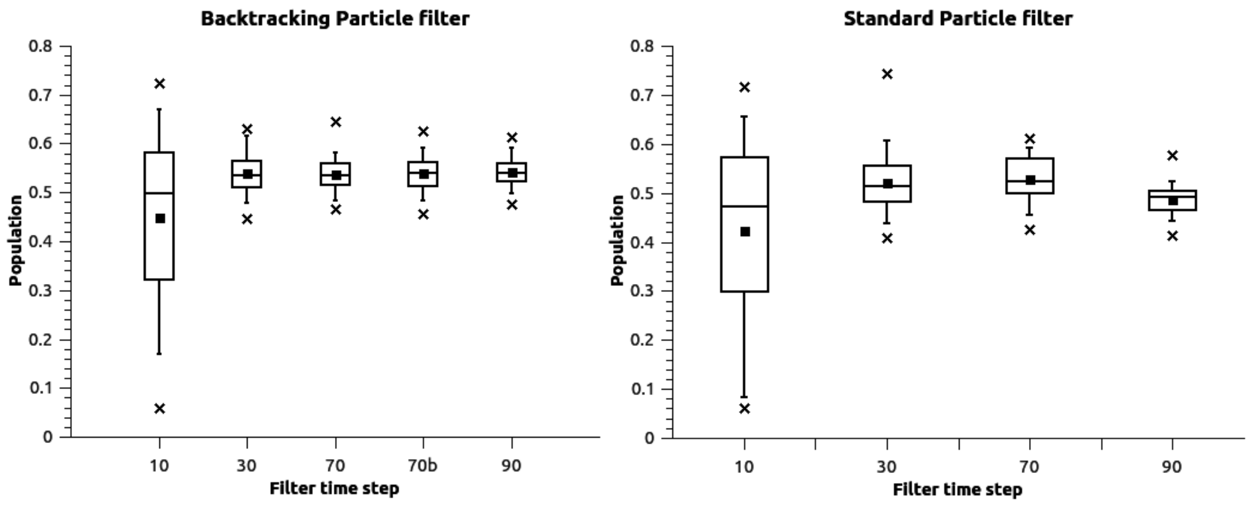

Figure 1: Population results of the Backtracking (left) and Standard (right) filter on the simple point model using 100 particles. The number of particles used while backtracking is 200. Backtracking triggers once during this run, after the 3rd filter update step. The results of the backtracking run are denoted with a 'b' on the horizontal axis. The box plot shows the 95th, 75th, 50th, 25th and 5th percentile values. The crosses mark the complete data range.

backtracking. The code is further outlined in Appendix A, where the code and its functionality is explained. The complete code of the backtracking method is available as a digital appendix to this thesis.

4.2 Simple point model results

Figures 1 through 4 show the results of the particle filter runs with the simple model. A filter run with 100 particles was chosen to show the progression of the particle cloud through the filter steps. The backtracking algorithm used 200 particles. Figure 1 shows the particle distribution of calculated model results directly. While both the backtracking and standard particle filter find similar results, it can be seen quite clearly that the backtracking particle filter has a much narrower range of values. This can be partly explained by luck, as the backtracking particle filter is already performing differently than the standard particle filter at filter timestep 30. The most distinctive difference between the two runs can be seen at filter step 90, however, showing a much narrower particle distribution than the standard filter.

Figure 2 shows the particle weight distribution as calculated by the user-defined 'particleweights' method in the filter method. The difference between the model runs can be seen most clearly here. Note that a large number of particles have near 0 weight in both model runs, which results in the strange shape of the box plots. While in figure 1 it is not immediately clear why backtracking occurred at filter timestep 7, here it can be seen directly. It can also be seen that both models perform quite well at filter timestep 30. Again, the most clear difference is at filter time step 70. The standard particle filter has clearly failed somehow. The backtracking particle filter has very high weighted particles at filter timestep 70.

The results from another 100 particle run is shown in Figure 3. This plot clearly shows that a backtracking particle filter run with 100 particles gets results much closer to the results obtained from a standard filter run with a 1000 particles than the standard particle filter, as the backtracking results are much closer to the 1 : 1 line in the graph.

Clearly, the backtracking particle filter performed markedly better in the 100 particle run with the simple

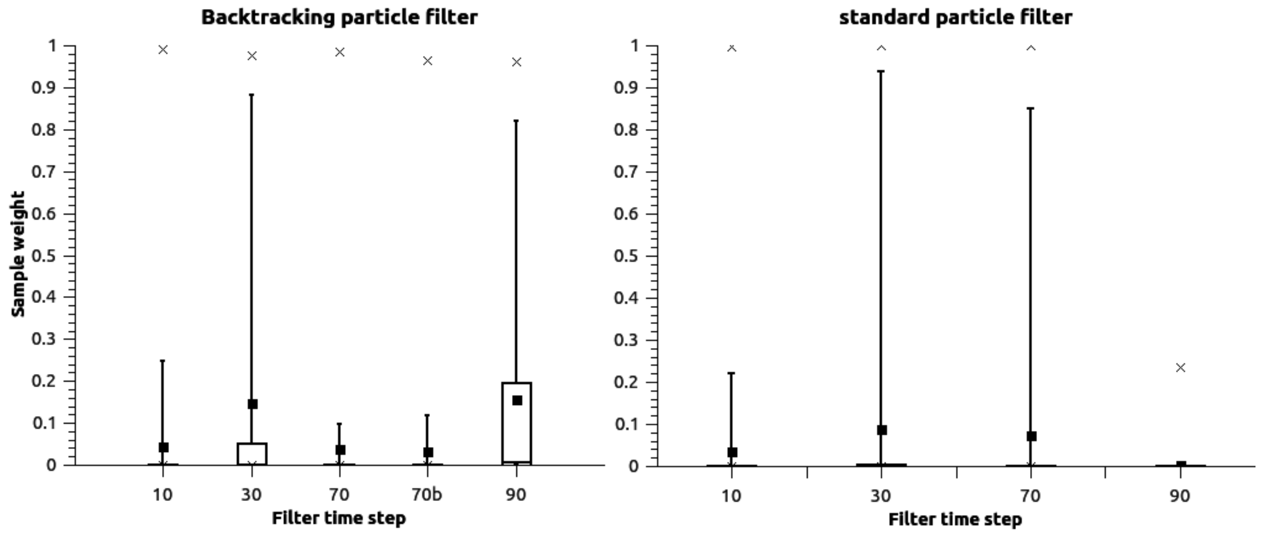

Figure 2: Particle weight distribution after each filter time step for the runs from Figure 1. The number of particles used while backtracking is 200. Backtracking triggers once during this run, after the 3rd filter period. The results of the backtracking run are denoted with a 'b' on the horizontal axis. The box plot shows the 95th, 75th, 50th, 25th and 5th percentile values. The crosses mark the complete data range.

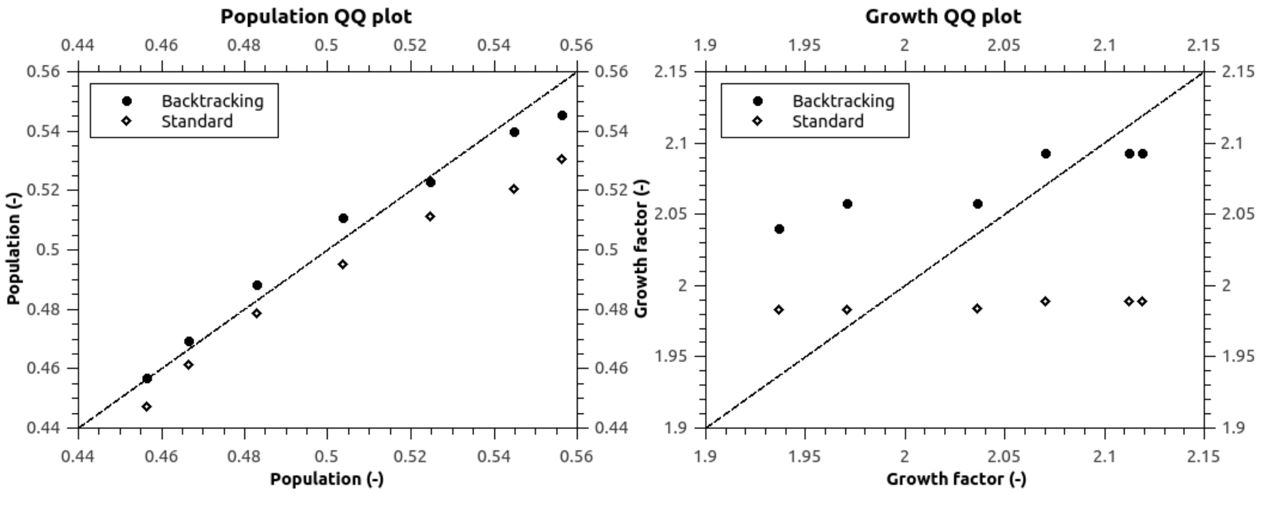

Figure 3: Comparison of the population (left) and growth factor (right) values in the particle filter of a run with 1000 particles (x-axis values) to standard and backtracking particle filter runs with 100 particles (y-axis values) at the final time step. The number of particles used while backtracking is 200. Backtracking triggers once during this run, after the 1st filter period. These results were obtained from a different run than the results from figures 1 and 2. The plot shows the 95th, 90th, 75th, 50th, 25th, 10th and 5th percentile values.

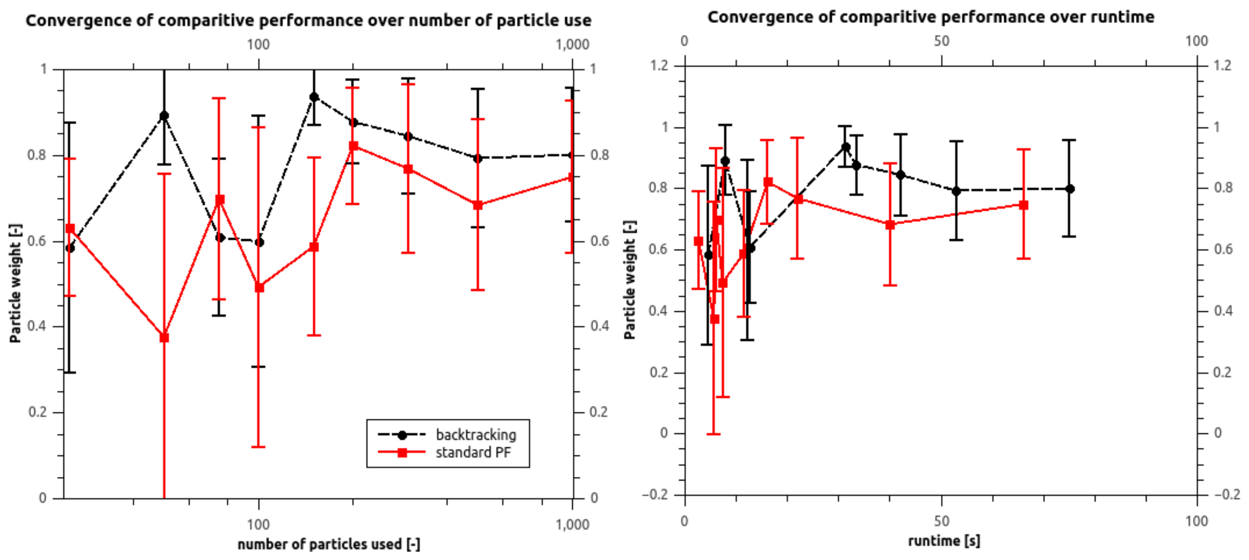

Figure 4: Summary of results gained using different amounts of particles on the simple point model. The line shows the mean particle weight of the 10% highest weighted particles, the error bar the standard deviation of this set. Plotted against the number of particles used (left) and the runtime in seconds (right). In the backtracking runs, the amount of particles used for backtracking are always twice the amount of regular particles. For particle filter runs with less than 100 particles, the runs were repeated a number of times to attain at least 100 datapoints.

model. The relative success of the backtracking algorithm on the simple model can also be seen in the particle weight distribution at the last filter timestep for particle filter runs with a different amount of particles (figure 4). Directly comparing the amount of particles used shows that with almost every particle filter run the backtracking particle filter performs better than the standard particle filter. This is not surprising since when backtracking occurs, the backtracking filter gets to use more particles compared to the standard particle filter. When plotting the particle weight distribution against computational time, the backtracking particle filter performs markedly better than the standard particle filter run. This is most obvious in the difference in the size of the error bars. Note that during the 1000 particle run with the backtracking particle filter, backtracking did not occur, and thus the result is very similar to the result of the standard particle filter. The fact that the backtracking particle filter still outperformed the standard particle filter with 1000 particle realizations does indicate that even with a high number of particles, luck is still a factor in determining the results of these particle filter runs.

4.3 Complex snowfall model results

Figures 5 through 8 show the results attained from the particle filter runs with the complex model. The results for the complex model are not as favorable for the backtracking particle filter as they are for the point model. The particles have a much wider distribution in the modeled data (figure 5) using the backtracking particle filter than from the standard particle filter. The particle weights (figure 6) are clearly much smaller from the backtracking particle filter than from the standard particle filter. This can again be explained by luck being a factor in running the particle filter, the standard particle filter run simply initializing with particles close to the measured values. The backtracking particle filter performance does not seem to improve after backtracking either.

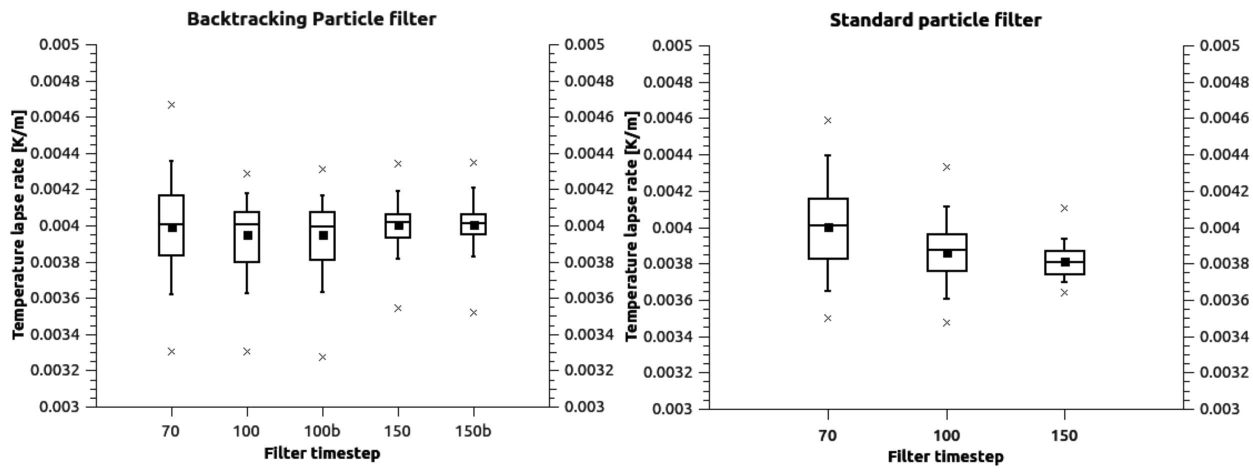

Figure 5: Temperature lapse rate results of the Backtracking (left) and Standard (right) filter on the complex snowfall model using 150 particles. The number of particles used while backtracking is 300. Backtracking is triggered twice during this run, after the 2nd and 3rd filter period. The results of the backtracking run are denoted with a ’b’ on the horizontal axis. The box plot shows the 95th, 75th, 50th, 25th and 5th percentile values. The crosses mark the complete data range.

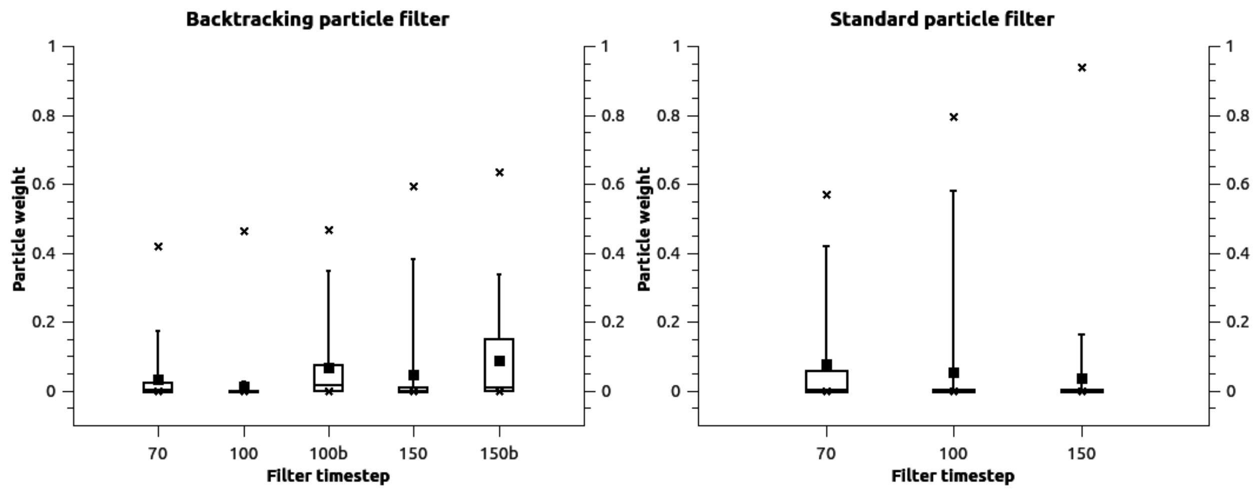

Figure 6: Particle weight distribution of the runs from Figure 5. The number of particles used while backtracking is 300. Backtracking is triggered twice during this run, after the 2nd and 3rd filter period. The results of the backtracking run are denoted with a ’b’ on the horizontal axis. The box plot shows the 95th, 75th, 50th, 25th and 5th percentile values. The crosses mark the complete data range.

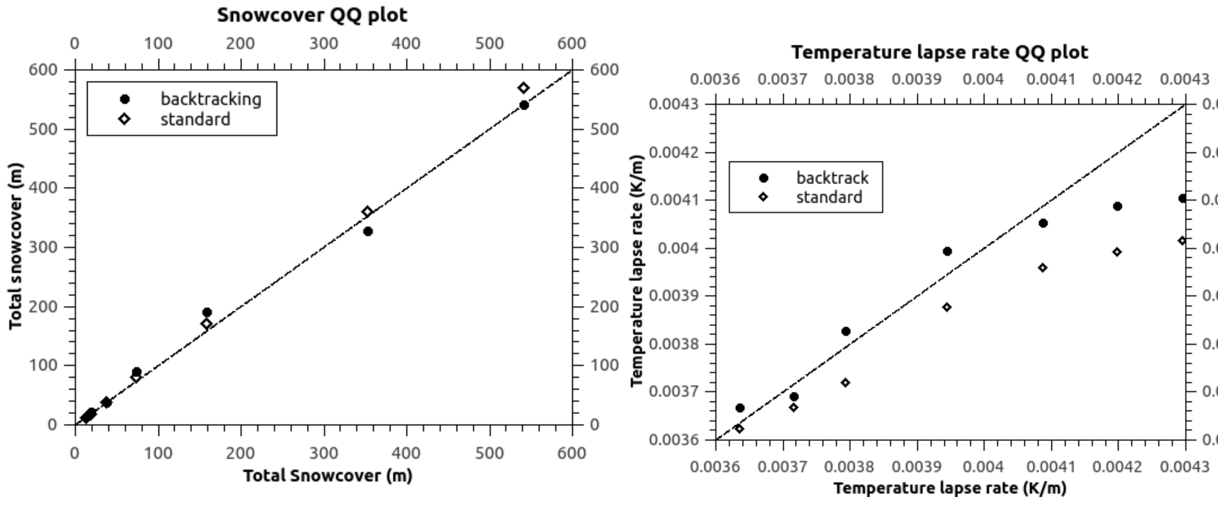

Figure 7: Comparison of the snowcover (left) and temperature lapse rate (right) values in the particle filter of a run with 1000 particles (x-axis values) to standard and backtracking particle filter runs with 100 particles (y-axis values) at the final time step. The number of particles used while backtracking is 200. Backtracking triggers once during this run, after the 1st filter period. These results were obtained from a different run than the results from figures 5 and 6. The plot shows the 95th, 90th, 75th, 50th, 25th, 10th and 5th percentile values.

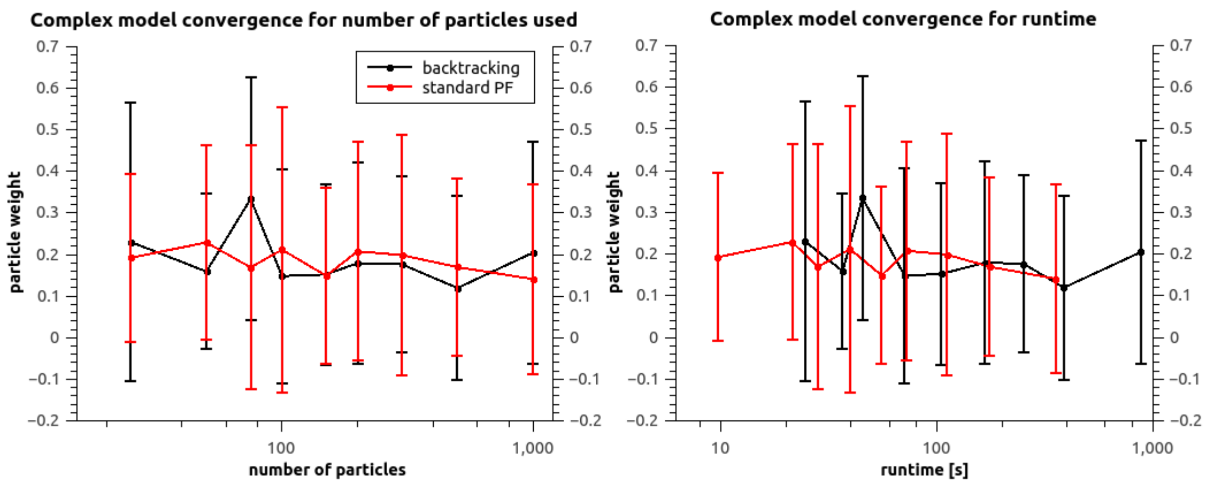

Figure 8: Summary of results gained using different amounts of particles using the complex snowfall model. The line shows the mean particle weight of the 10% highest weighted particles, the error bar the standard deviation of this set. Plotted against the number of particles used (left) and the runtime in seconds (right). In the backtracking runs, the amount of particles used for backtracking are always twice the amount of regular particles. For particle filter runs with less than 100 particles, the runs were repeated a number of times to attain at least 100 datapoints.

However, another run with 100 particles shows a clear difference in results from the quantile-quantile plots (figure 7). While it might not be obvious from a direct comparison of the model results with eachother, comparing the model results of the backtracking and standard particle filter run with 100 particles to a standard particle filter run with 1000 particles clearly shows that the backtracking particle filter is much better at attaining similar results to the 1000 particle run.

There is no clear difference in performance of the backtracking particle filter compared to the standard particle filter when comparing the results from the filter runs realized with different amounts of particles (figure 8). Also comparing the runtime shows no significant difference in performance. It is important to note that for the complex model, the number of particles seems to have no effect on the performance of the filter, regardless of which algorithm was chosen.

5 Discussion

Compared to the standard particle framework, the backtracking particle framework performs quite well. The backtracking algorithm is able to detect filter divergence quite well from the list of normalized particle weights, which makes it largely independent of the particle weight calculation used by the model. It is possible that other models have more strict or loose criteria for filter divergence. As it stands right now, the backtrack test is coded inside the framework, giving users no freedom to change the conditions. This feature could be useful for future use.

The method of cloud expansion is quite brute force, simply copying the whole particle cloud as much as they can and sampling random particles for the residual samples, but according to Spiller et al. [2008] the method of doubling used has very little effect on the performance of the backtracking algorithm. Cloud reduction, the method of going back to the original number of particles, is quite brute force as well. The backtracking algorithm simply takes the highest weighted particles and continues the particle filter with these. Statistically this is not the best method because this can actually cause particle collapse. However, a backtracking algorithm is only useful when dealing with a highly nonlinear system or a system with a large stochastic component, so it can be assumed that in between filter timesteps enough stochastic forcing will be put on the system to prevent filter collapse.

When testing the framework with the snowfall model, the backtracking algorithm did not perform as well compared to the standard particle framework. This could be because of the high stochastic forcing in the dynamic section of the model, compounded by the jitter introduced on the temperature lapse rate to prevent particle collapse. During runs without the jitter, whichever filter algorithm was used would collapse on a single temperature lapse rate value, and no other way could be found to abate this particle collapse.

It could be that running the complex model with more particles could yield more meaningful results. However, at 1000 particles the computational restraints of the system that was used to create these results was were already apparent. A backtracking run with 2000 particles was attempted, but this caused issues with the system memory.

The snowfall model uses three update steps, which might be too small a number to have meaningful results from the backtracking algorithm, because it is not always clear to see if the backtracking algorithm effectively deals with the detected filter divergence until later.

In general it is quite tricky to create the model conditions where the backtracking particle filter can be effectively compared to the standard particle filter. The stochasticity and uncertainty has to be high enough so the backtracking algorithm triggers, but if the backtracking algorithm triggers at every update step the filter is far too close to filter divergence to yield meaningful results. If the uncertainty is too low however, backtracking will not trigger at all and the results will be equal to the results from a standard particle filter run.

This raises a clear limitation to the backtracking algorithm. A backtracking algorithm is useful if you want to constrain the number of particles used in the filter, and at the same time want to prevent filter divergence. However, if your model is too stochastic and backtracking triggers after every filter step, it is better to simply use a standard particle filter with a higher number of particles.

There are a number of factors that have not been tested during the testing with this model. The number of particles used to backtrack has been kept at twice the regular amount during the entire testing phase. The backtracking algorithm can handle any number of particles above the regular amount, however, and it could yield useful results to test the effect of the amount of backtracking particles used on the results to see if there is a clear optimum in the results.

Another thing that has not yet been tested is a situation where the model error decreases over the model run, and the measurement error varies as well. This is a case that a backtracking particle filter has a high chance to have meaningful results. If the model error is exceptionally low at one update step, the backtracking particle filter can deal with this very effectively.

The two models used to test the backtracking algorithm are both quite simple and both have a small state space. It would be meaningful to test the backtracking algorithm on a model with a truly large state space to see whether the algorithm can cut down on computation time there as well.

There is still room for improvement in the backtracking algorithm as well. The current backtracking algorithm expands the particle cloud by simply copying every particle regardless of weight. Spiller et al. [2008] also tested a method called *directed doubling*, where the posterior calculated during the failed filter time step is used to select which particles will be used in the doubling. During their experiments the effect of the directed doubling was minimal, but perhaps when using the filter on other models it can yield very meaningful results.

The algorithm could also be altered to allow user-defined backtrack triggers catered specifically to the model used. The goal in this thesis was to create a generally applicable algorithm so this option was not explored, but it could make the algorithm more malleable to other types of models.

The current algorithm also doesn't explore cloud expansion on expanded clouds or recursive backtracking, as using it on a model that simply cannot recreate the measurements would create an infinite data-creation loop as more and more particles will be initialized. It can be useful to explore this option however, as this could pave the way for a fluid particle filter where the amount of particles needed to get results is no longer an input. This model would instead iterate toward the optimal number of particles automatically.

6 Conclusion

The backtracking particle framework works as intended, being able to detect filter divergence and reacting to it. Filter divergence is detected in a way that is largely independent of the way the particle weights are calculated by the user, using the list of normalized particle weights. The framework is able to be deployed on any model that can make use of the particle filter algorithm, and the user does not have to add any methods to the model class in order to run the backtracking particle filter. In this regard, the particle filter is quite user-friendly.

During testing on the simple point model, the backtracking particle filter consistently outperforms the standard particle filter, even when comparing runtime. The results of the snowfall model are not as conclusive, showing no difference in result between the standard particle filter and the backtracking algorithm. However, the results of the snowfall model also seem to be independent of particle cloud size, which indicates that the particle cloud sizes used were probably too small. Also, the 'jitter' introduced in the snowfall model to combat particle collapse could be a large factor in causing the poor performance of the backtracking algorithm. Computational and time restraints limit exploration into the exact cause.

At this point it is difficult to say if the backtracking particle filter will be useful for future research. The conditions necessary for a backtracking algorithm are quite strict. The model needs to have a high nonlinear or stochastic dynamic component, resulting in the possibility of the filter divergence unexpectedly occurring in between filter time steps. However, when these conditions are met there is little doubt that the backtracking particle filter algorithm will increase the accuracy of the results without exponentially increasing the computation time.

In the future the algorithm could be altered to be more useful in research by allowing users to input their own backtracking conditions and tests, implementing different ways of cloud expansion, improving on the cloud reduction algorithm, and exploring a way for the algorithm to automatically figure out the optimal sample size.

References

K. de Jong and D. Karssenberg. Pcraster documentation, 2015. URL `http://pcraster.geo.uu.nl/pcraster/4.1.0/doc/manual/index.html`.

A. Doucet. *Sequential Monte Carlo Methods in Practice*. Springer, 2001.

Arnaud Doucet, Simon Godsill, and Christophe Andrieu. On sequential monte carlo sampling methods for bayesian filtering. *Statistics and Computing*, 10:197–208, 2000.

A. Gelb, J.F. jr. Kasper, R.A. jr. Nash, C.F. Price, and A.A. jr. Sutherland. *Applied Optimal Estimation*. The M.I.T. Press, 1974.

N. Thompson Hobbs and Mevin B. Hooten. *Bayesian Models: A Statistical Primer for Ecologists*. Princeton : Princeton University Press, 2015.

Colin Howson. *Scientific Reasoning: The Bayesian approach*. La Salle, Illinois : Open Court, 1990.

Derek Karssenberg, Oliver Schmitz, Peter Salamon, Kor de Jong, and Marc F.P. Bierkens. A software framework for construction of process-based stochastic spatio-temporal models and data assimilation. *Environmental Modelling & Software*, 25:489–502, 2010.

G. Kitagawa. Monte carlo filter and smoother for non-gaussian nonlinear state space models. *Journal of Computational and Graphical Statistics*, 5(1):1–25, March 1996.

Hans R. Künsch. Particle filters. *Bernouilli*, 19(4):1391–1403, 2013.

S. Nanako, G. Ueno, and T. Higuchi. Merging particle filter for sequential data assimilation. *Nonlinear Processes in Geophysics*, 14(4):395–408, July 2007.

D. T. Pham. Stochastic methods for sequential data assimilation in strongly nonlinear systems. *Monthly Weather Review*, 129:1194–1207, May 2001.

D. Simon. *Optimal State Estimation*. John Wiley & Sons, 2006.

Elaine T. Spiller, Amarjit Budhiraja, Kayo Ide, and Chris K.R.T. Jones. Modified particle filter methods for assimilating lagrangian data into a point-vortex model. *Physica D*, 237:1498–1506, 2008.

P. J. van Leeuwen. A variance-minimizing filter for large-scale applications. *Monthly Weather Review*, 131: 2071–2084, 2003.

P. J. van Leeuwen. Particle filtering in geophysical systems. *Monthly Weather Review*, 137:4089–4114, 2009.

Judith A. Verstegen, Derek Karssenberg, Floor van der Hilst, and André Faaij. Spatio-temporal uncertainty in spatial decision support systems: A case study of changin land availability in bioenergy crops in mozambique. *Computers, Environment and Urban Systems*, 36:30–42, 2012.

X. Xiong, I. M. Navon, and B. Uzunoglu. A note on the particle filter with posterior gaussian resampling. *Tellus*, 58A:456–460, 2006.

A The Backtracking Particle Filter Code

The complete filter code is available in the digital appendix to this thesis. In this section only parts of the code will be explained, as most of the backtracking particle filter method is identical to the standard particle filter. Where the backtracking method diverges from the standard particle filter, this section will explain what the algorithm does. Information on the standard particle filter algorithm is available online at `http://pcraster.geo.uu.nl/pcraster/4.1.0/doc/python/pcraster/framework/PCRasterPythonFramework.html#particle-filter-method`.

The backtracking algorithm diverges from the standard particle filter at one key point in the 'run' function:

```
[...]
for sample in range(1, self._userModel().nrSamples() + 1):
    self._userModel()._setCurrentSample(sample)
    self._userModel()._d_inUpdateWeight = True
    fitnessValue = self._userModel().updateWeight()
    self._userModel()._d_inUpdateWeight = False
    assert type(fitnessValue) == float or type(fitnessValue) == int
    self._userModel()._d_particleWeights[sample - 1] = fitnessValue

# do backtracktest
self.doBacktrack = self._backtrackTest()
# if backtracktest == true initialize runbacktrack
    if self.doBacktrack == True:
      self._backtrack()
      # determine samples to clone
    samplesToClone = self._samplesToClone(self._particleWeights())
    assert sum(samplesToClone) == self._userModel().nrSamples()

[...]
```

When the weights have been calculated, the '_backtracktest' function is called on the list of weights. If this function returns True, the 'backtrack' function is called. This sequence encompasses the entire act of backtracking. The backtracktest function detects filter degeneracy:

```
def _backtrackTest(self):
  if hasattr(self._userModel(), 'backtrackTest'):
    return self._userModel().backtrackTest()
  else:
    weights = self._particleWeights()
    sortedWeights = sorted(weights, reverse=True)
    normalizedWeights = self._normaliseWeights(sortedWeights)
    topFivePercent = int(math.floor(len(normalizedWeights)/20.0))
    if sum(normalizedWeights[:topFivePercent]) > 0.9:
      return True
    else:
      return False
```

If a researcher has added their own backtracking method to their model, this method runs that and returns the result. Otherwise, it takes the list of weights that has just been calculated and sorts this from high to low. This list is then normalized to have a total weight of 1. The first 5 numbers in this list are added together. If this number is higher than 0.9, more than 90 percent of the total weight of particles is carried by the largest 5% of them, and the function returns True. Otherwise, it returns False. When the function returns True, the backtrack function triggers:

```
def _backtrack(self):
     # Do everything the run does, but in directories [N+1:M]
     nrOriginalSamples = self._userModel()._lastSampleNumber()
     firstAuxSample = self._userModel()._lastSampleNumber() + 1
     lastAuxSample = self._userModel()._lastSampleNumber() + self.nrAuxSamples
     self._userFramework()._setSampleNumbers(firstAuxSample, lastAuxSample)
```

First, settings of the the particle filter framework have to be set for the backtracking algorithm. The '_setSampleNumber' function changes the range of sample numbers the framework uses for running the monte carlo simulation.

```
# Copy samples into new sampledirs #
nr = 0
for sample in range(self._userModel()._firstSampleNumber(),
                    self._userModel()._lastSampleNumber() + 1):
  if nr < nrOriginalSamples:
    nr += 1
    dirNr = nr
  elif sample - self._userModel()._lastSampleNumber() > nrOriginalSamples:
    nr = 1
    dirNr = nr
  else:
    dirNr = math.floor(random.uniform(1, nrOriginalSamples + 1))
  source = "%d" % (dirNr)
  destination = "%d" % (sample)
  if os.path.isdir(destination):
    shutil.rmtree(destination)
  shutil.copytree(source, destination)
```

The particles are then copied into the folders that will be used by the backtracking algorithm. It first copies the entire original population of particles. Then it checks if there are enough of the new samples left to do it again. When it can't, it will take random samples until the total amount is reached.

```
# run particleFilter
lastPeriod = len(self._userModel()._d_filterTimesteps)
currentPeriod = self._userModel()._d_filterPeriod
self._runMonteCarlo(currentPeriod, lastPeriod)

if not currentPeriod == lastPeriod:
  # update the weights
  # calling the "objective fuction" for each sample
  auxSampleWeights = []
  for sample in range(self._userModel()._firstSampleNumber(),
                      self._userModel()._lastSampleNumber() + 1):
    self._userModel()._setCurrentSample(sample)
    self._userModel()._d_inUpdateWeight = True
    fitnessValue = self._userModel().updateWeight()
    self._userModel()._d_inUpdateWeight = False
    assert type(fitnessValue) == float or type(fitnessValue) == int
    auxSampleWeights.append(fitnessValue)
```

The algorithm then runs the monte carlo simulation on the particles for the period it is in the same way it did before backtracking in the 'run' method, but now for the new sample cloud. After this, it calculates the weight of each particle the same way it does in the 'run' method.

```
normalisedWeights = self._normaliseWeights(auxSampleWeights)
cumulativeWeights = self._cumulativeWeights(normalisedWeights)

samplesToClone = [0] * nrOriginalSamples
for i in range(0, nrOriginalSamples):
  lower = 0.0
  uniformReal = random.uniform(0.0, 1.0)
  for j in range(0, len(cumulativeWeights)):
    upper = cumulativeWeights[j]
    if uniformReal > lower and uniformReal <= upper:
      samplesToClone[j] += 1
      lower = upper

indexToClone = []
for i in range(len(samplesToClone)):
  sample = samplesToClone[i]
  if sample > 0:
    indexToClone += sample * [i]

for i in range(0, nrOriginalSamples):
  sampleToClone = indexToClone[i] + nrOriginalSamples + 1
  varname = "stateVar"
  cloneSource = os.path.join("%d" % sampleToClone, varname)
  cloneDestination = os.path.join("%d" % i, varname)
  if os.path.isdir(cloneDestination):
    shutil.rmtree(cloneDestination)
  shutil.copytree(cloneSource, cloneDestination)
  self._userModel()._d_particleWeights[i] = auxSampleWeights[indexToClone[i]]

# undo damage
```

```
self._userFramework()._setSampleNumbers(1, nrOriginalSamples)
```

Now it takes an amount of particles from the cloud equal to the number of original particles. These particles are sampled from a weighted distribution based on the particle weights calculated, just as in the sequential importance resampling method.