SANGOMA: Stochastic Assimilation for the Next Generation Ocean Model Applications EU FP7 SPACE-2011-1 project 283580

Deliverable 3.3:
Report on Implementation
Due date: 30/04/2014
Delivery date: 30/10/2015
Delivery type: Report, public

Lars Nerger Paul Kirchgessner
Alfred-Wegener-Institute, GERMANY

Arnold Heemink Nils van Velzen
Martin Verlaan M. Umer Altaf
Delft University of Technology, NETHERLANDS

Jean-Marie Beckers Alexander Barth
University of Liège, BELGIUM

Peter Jan Van Leeuwen Sanita Vetra-Carvalho
University of Reading, UK

Pierre Brasseur Jean-Michel Brankart Guillem Candille
CNRS-LEGI, FRANCE

Pierre De Mey
CNRS-LEGOS, FRANCE

Laurent Bertino Alberto Carrassi
NERSC, NORWAY

Contents

Chapter 1

Introduction

In work package 3 of SANGOMA existing data-assimilation methods are investigated and new methods are developed that allow for non-linear models and non-linear observation operators. The different methods are implemented as modules for the data-assimilation tool boxes used by the SANGOMA partners. One of the methods has to be implemented by all partners of SANGOMA that run a toolbox.

During the preparation of this task, the partners extensively discussed the possible methods for the common implementation. Candidates were the following methods:

- Equivalent weights particle filter (EWPF)
- Multivariate Rank Histogram Filter (MRHF)
- Gaussian Mixture Filter (GMF)

The EWPF is the research focus of the SANGOMA partner at University of Reading [van Leeuwen, 2010, van Leeuwen and Ades, 2013]. The MRHF has been developed by the Partner CNRS-LEGI at the time when the methods were discussed (meanwhile the method is published as Metref et al. [2014]). The Gaussian mixture filter is an extension of the Kalman filter that models non-Gaussian state distributions as sums of multiple Gaussian distributions.

The discussions involved the consideration of the method's features, implementation complexity, as well as readiness for a common implementation. The EWPF is a fully non-linear method that does not base on a modeling of distributions. However, the particle forecasting is rather complex as it involves a stochastic perturbation. In addition, the proposal density has to be applied in the form of a "nudging term" in the proposal step during the time stepping of the model and the equivalent weights step has to be implemented in a model-dependent way. The MRHF does imply a modeling of the distributions by an ensemble weighting scheme following the histogram of the ensemble distribution. At the time of the discussions, the MRHF was under development, and it still had to be checked whether a sufficient description of the method could be prepared to allow for a common implementation in the different assimilation tool boxes. The Gaussian mixture filter is less advanced as the distributions are modeling by a sum of Gaussian functions. However, it is also less skillful for the treatment of non-linear models and observation operators. As such, an implementation of the EWPF or MRHF would be preferable.

The discussions finally lead to the decision to use the EWPF method for the common implementation. Both the proposal step and equivalent weights step of the filter method can be implemented in a model-independent code that uses call-back routines for the model-specific operations. This strategy is consistent with the data model of SANGOMA.

The purpose of this deliverable is to report on the implementation of the EWPF as a common new data assimilation method in all toolboxes.

Chapter 2

The Equivalent Weights Particle Filter (EWPF)

Particle filters, like ensemble methods, are variants of the Monte Carlo methods in which the probability distribution of the model state given some observations is approximated by a number of particles; however, unlike Kalman filter-based ensemble methods, particle filters are fully non-linear data assimilation techniques. While particle filters are not a new concept, until very recently they have been deemed to be computationally unfeasible for large-dimensional systems due to the filter degeneracy problem. However, recently there has been a new development in the field and particle filter variants have emerged which have been shown to work for large dimensional systems with a limited number of particles. One such a method is the equivalent weights particle filter (EWPF) [Van Leeuwen, 2010, 2011, Ades and van Leeuwen, 2012], which by design avoids filter degeneracy by exploiting future observations. EWPF can be summarised by the following steps:

1. Before observation time k for each time step $0 \leq m < k$ and for each particle $j = 1, ..., N$:

   (a) Advect the model state in time

   $$\mathbf{x}_j^{(m)} = \mathcal{M}_m\left(\mathbf{x}_j^{(m-1)}\right) + \tilde{\boldsymbol{\beta}}_j^{(m)} + \boldsymbol{\Upsilon}\left[\mathbf{y}^{(k)} - \mathcal{H}_k\left(\mathbf{x}_j^{(m-1)}\right)\right] \tag{2.1}$$

   where $\boldsymbol{\Upsilon} = \mathbf{Q}(\mathbf{H}^{(k)})^{\mathsf{T}}\left[\mathbf{H}^{(k)}\mathbf{Q}(\mathbf{H}^{(k)})^{\mathsf{T}} + \mathbf{R}\right]^{-1}$ with $\mathbf{Q}$ being the covariance matrix of model errors, $\mathbf{R}$ the covariance matrix of observation errors, and $\mathbf{H}^{(k)}$ the linearised version of the observation operator $\mathcal{H}_k$.

   (b) Compute weights

   $$w_j^{(m)} = w_j^{(m-1)} \frac{p\left(\mathbf{x}_j^{(m)}|\mathbf{x}_j^{(m-1)}\right)}{q\left(\mathbf{x}_j^{(m)}|\mathbf{x}_j^{(m-1)}, \mathbf{y}^{(k)}\right)}, \tag{2.2}$$

   where both transition densities are assumed to be Gaussian and are calculated according to

$$\begin{aligned} p\left(\mathbf{x}_j^{(m)}|\mathbf{x}_j^{(m-1)}\right) &= \exp\left[-\frac{1}{2}\left(\boldsymbol{\Upsilon}\left[\mathbf{y}^{(k)} - \mathbf{H}^{(k)}\mathbf{x}_j^{(m)}\right] + \tilde{\boldsymbol{\beta}}_j^{(m)}\right)^{\mathsf{T}} \mathbf{Q}^{-1}\right. \\ &\quad \left.\left(\boldsymbol{\Upsilon}\left[\mathbf{y}^{(k)} - \mathbf{H}^{(k)}\mathbf{x}_j^{(m)}\right] + \tilde{\boldsymbol{\beta}}_j^{(m)}\right)\right] \quad (2.3) \\ q\left(\mathbf{x}_j^{(m)}|\mathbf{x}_j^{(m-1)}, \mathbf{y}^k\right) &= \exp\left[-\frac{1}{2}\left(\tilde{\boldsymbol{\beta}}_j^{(m)}\right)^{\mathsf{T}}\mathbf{Q}^{-1}\tilde{\boldsymbol{\beta}}_j^{(m)}\right]. \quad (2.4) \end{aligned}$$

2. At observation time k:

   (a) Calculate the maximum weight value for each particle

$$\begin{aligned} C_j &= -\log w_j^{(k-1)} + \frac{1}{2}\left[\mathbf{y}^{(k)} - \mathcal{H}_k\left(\mathcal{M}_k\left(\mathbf{x}_j^{(k-1)}\right)\right)\right]^{\mathsf{T}} \\ &\quad \left[\mathbf{H}^{(k)}\mathbf{Q}\left(\mathbf{H}^{(k)}\right)^{\mathsf{T}} + \mathbf{R}\right]^{-1}\left[\mathbf{y}^{(k)} - \mathcal{H}_k\left(\mathcal{M}_k\left(\mathbf{x}_j^{(k-1)}\right)\right)\right]. \end{aligned}$$

   Then choose a target weight C such that 80% (or any other suitable percentage) of particles can reach this weight, i.e. that 80% of C_j are less than C.

   (b) Find the deterministic particle analysis update (for the particles which can reach the target weight C) via

$$\check{\mathbf{x}}_j^{(k)} = \mathcal{M}_k\left(\mathbf{x}_j^{(k-1)}\right) + \alpha_j\boldsymbol{\Upsilon}\mathbf{d}_j^{(k)}, \quad (2.5)$$

   where

$$\begin{aligned} \boldsymbol{\Upsilon} &= \mathbf{Q}\left(\mathbf{H}^{(k)}\right)^{\mathsf{T}}\left[\mathbf{H}^{(k)}\mathbf{Q}\left(\mathbf{H}^{(k)}\right)^{\mathsf{T}} + \mathbf{R}\right]^{-1} \quad (2.6) \\ \alpha_j &= 1 - \sqrt{1 - \frac{b_j}{a_j}} \quad (2.7) \\ a_j &= \frac{1}{2}\left(\mathbf{d}_j^{(k)}\right)^{\mathsf{T}}\mathbf{R}^{-1}\mathbf{H}^{(k)}\boldsymbol{\Upsilon}\mathbf{d}_j^{(k)} \quad (2.8) \\ b_j &= \frac{1}{2}\left(\mathbf{d}_j^{(k)}\right)^{\mathsf{T}}\mathbf{R}^{-1}\mathbf{d}_j^{(k)} - C - \log w_j^{(k-1)} \quad (2.9) \\ \mathbf{d}_j^{(k)} &= \mathbf{y}^{(k)} - \mathcal{H}^{(k)}\left(\mathbf{x}_j^{(k)}\right). \quad (2.10) \end{aligned}$$

   (c) Perturb each particle with random perturbations

$$\mathbf{x}_j^{(k)} = \check{\mathbf{x}}_j^{(k)} + \mathbf{d}\boldsymbol{\beta}_j^{(k)} \quad (2.11)$$

   where the perturbation $\mathbf{d}\boldsymbol{\beta}_j^{(k)}$ is drawn from a mixture of uniform and Gaussian distributions, given by

$$\mathbf{d}\boldsymbol{\beta} \sim (1-\epsilon)\mathbf{Q}^{1/2}\mathcal{U}(-\gamma_U\mathbf{I}, +\gamma_U\mathbf{I}) + \epsilon\mathcal{N}(0, \gamma_N^2\mathbf{Q}). \quad (2.12)$$

   Choosing $\epsilon = 0.001/N$ ensures that we mainly sample from the uniform distribution, but the possibility to sample from the Gaussian distribution ensures the support of the proposal density is at least equal to the support of the model prior. Other parameters are chosen to be as follows:

$$\gamma_U = 10^{-5} \tag{2.13}$$

$$\gamma_N = \frac{2^{n/2}\epsilon\gamma_U^n}{\pi^{n/2}(1-\epsilon)}. \tag{2.14}$$

(d) Calculate the full weights at time k

$$w_j^{(k)} = w_j^{(k-1)} \frac{p\left(\mathbf{x}_j^{(k)}|\mathbf{x}_j^{(k-1)}\right) p\left(\mathbf{y}^{(k)}|\mathbf{x}_j^{(k)}\right)}{q\left(\mathbf{x}_j^{(k)}|\mathbf{x}_j^{(k-1)}, \mathbf{y}^{(k)}\right)}. \tag{2.15}$$

taking the final perturbation into the account using the transition density

$$q\left(\mathbf{x}_j^{(k)}|\mathbf{x}_j^{(k-1)}, \mathbf{y}^{(k)}\right) = (1-\epsilon)\mathbf{Q}^{1/2}\mathcal{U}(-\gamma_U\mathbf{I}, +\gamma_U\mathbf{I}) + \epsilon\mathcal{N}(0, \gamma_N^2\mathbf{Q}). \tag{2.16}$$

(e) Resample to obtain a full ensemble again, e.g. using universal resampling. After resampling the weights of the resampled particles are set to be equal, i.e. $w_j^{(k)} = 1/N$.

Chapter 3

Description of the EWPF code

This chapter describes the structure of the central generic routines of the EWPF. The calling interface to the routines '*proposal_step*' and '*equal_weight_step*' is documented as well as the interfaces of the call-back routines that are specific to the model and the observations that are assimilated. These routines have to be implemented by the user of the EWPF code.

3.1 The main routines called by user

The EWPF code is structured such that a user calls only two routines: '*proposal_step*' and '*equal_weight_step*' from their code. The '*proposal_step*' routine nudges particles at each model time step towards the next set of future observations. It also accumulates weight of each particle due to the nudging. The '*equal_weights_step*' routine updates particles which can achieve the deterministic move in space using the current observational information and then resamples updated particles to form a full set of particles with equal weights for the next assimilation window. The calling interfaces of both routines is documented below. Figures 5.1 and 5.2 in Chapter 5 show the schematic of the EWPF code separated into the two main routines that are called by the user.

proposal_step(weight,x_n,y,tt,obsVec,dt_obs) bind(C, name="proposal_step_")

```
use, intrinsic :: ISO_C_BINDING
use sangoma_base, only: REALPREC, INTPREC ! use sangoma defined precision for C-Bind
use user_base                            ! use user defined parameters
implicit none

integer(INTPREC), intent(in) :: tt                        ! current model time step
integer(INTPREC), intent(in) :: obsVec,dt_obs             ! number of next observation set & model
                                                          ! timesteps between obs
real(REALPREC), intent(in), dimension(Ny)     :: y        ! vector of the next set of observations
real(REALPREC), intent(inout), dimension(Nx,Ne) :: x_n    ! state matrix at the current time step n
real(REALPREC), intent(inout), dimension(Ne)  :: weight   ! vector containing weights for all particles
```

equal_weights_step(weight,x_n,y) bind(C, name="equal_weight_step_")

```
use, intrinsic :: ISO_C_BINDING
use sangoma_base, only: REALPREC, INTPREC ! use sangoma defined precision for C-Bind
use user_base                            ! use user defined parameters
implicit none

real(REALPREC), intent(inout), dimension(Ne)   :: weight ! vector holding particle weights
real(REALPREC), intent(out), dimension(Nx,Ne)  :: x_n    ! matrix of states for each particle
                                                         ! at current observation timestep n
real(REALPREC), intent(in), dimension(Ny)      :: y      ! vector of obs. data at current time step n
```

3.2 Model specific call-back routines

There are some model specific functions that each user has to implement in their system, they are:

- 'cb_H' - observation operator H;
- 'cb_HT' - transpose observation operator H^T;
- 'cb_solve_r' - inverse observation error covariance operator R^{-1};
- 'cb_solve_hqht_plus_r' - inverse innovation error covariance operator $(HQH^T + R)^{-1}$;
- 'cb_Qhalf' - square-root of model error operator $Q^{1/2}$.

Each of these routines need to be implemented as operators acting on a given input vector(s). Note that a half model error operator needs to be implemented since it is used to obtain random normal numbers with $Q^{1/2}$ covariance matrix in 'equal_weights_step' routine. Where we need to use the full Q matrix 'cb_Qhalf' is simply applied twice.
We list here the purpouse of these functions and suggested input/output variables for each of them.

cb_H(dim2,vecIn,vecOut)
Given *vecIn*, a vector or collection of vectors in state space, compute *vecOut* = $H($*vecIn*$)$, where H is a mapping from state space to observation space.

```
use sangoma_base   ! use sangoma-defined precision for C-Bind
use user_base      ! use user defined parameters
implicit none

integer(INTPREC), intent(in) :: dim2                          ! second dimension of the vector,
                                                              ! i.e. dim2=1 if only one particle
                                                              ! or state vector is used and
                                                              ! dim2 = Ne if the whole ensemble
                                                              ! of particles is used
real(REALPREC), intent(in), dimension(Nx,dim2) :: vecIn       ! input vector in state space to which to
                                                              ! apply the observation operator h, e.g. h(x)
real(REALPREC), intent(inout), dimension(Ny,dim2) :: vecOut   ! resulting vector in observation space
```

cb_HT(dim2,vecIn, vecOut)
Given *vecIn*, a vector or collection of vectors in observation space, compute *vecOut* = $H^T($*vecIn*$)$, where H^T is a mapping from observation space to state space.

```
use sangoma_base   ! use sangoma-defined precision for C-Bind
use user_base      ! use user defined parameters
implicit none

integer(INTPREC), intent(in) :: dim2                          ! second dimension of the vector,
                                                              ! i.e. dim2=1 if only one particle
                                                              ! or state vector is used and
                                                              ! dim2 = Ne if the whole ensemble
                                                              ! of particles is used
real(REALPREC), intent(in), dimension(Ny,dim2) :: vecIn       ! input vector in obs. space to which to
                                                              ! appl h, e.g. h^T(x)
real(REALPREC), intent(inout), dimension(Nx,dim2) :: vecOut   ! resulting vector in state space elements
```

cb_solve_r(dim2,vecIn,vecOut)
Given *vecIn*, a vector or collection of vectors in observation space, compute *vecOut* = $R^{-1}($*vecIn*$)$, where R is observation error covariance.

```
use sangoma_base   ! use sangoma defined precision for C-Bind
use user_base      ! use user defined parameters
implicit none

integer(INTPREC), intent(in) :: dim2                        ! a second dimension of the vecIn,vecOut
real(REALPREC), intent(in), dimension(Ny,dim2) :: vecIn     ! input vector in observation space
                                                            ! which to apply the inverse observation error
                                                            ! covariances R, e.g. R^{-1}(d)
real(REALPREC), intent(inout), dimension(Ny,dim2) :: vecOut ! resulting vector in observation space
```

cb_solve_hqht_plus_r(dim2,vecIn, vecOut)
Given *vecIn*, a vector or collection of vectors in observation space, compute *vecOut* = $(HQH^T + R)^{-1}($*vecIn*$)$, where R is observation error covariance, H is mapping from state space to observation space and Q is model error covariance.

```
use sangoma_base   ! use sangoma defined precision for C-Bind
use user_base      ! use user defined parameters
implicit none

integer(INTPREC), intent(in) :: dim2                       ! second dimension of the vector,
                                                           ! i.e. dim2=1 if only one particle
                                                           ! or state vector is used and
                                                           ! dim2 = Ne if the whole ensemle
                                                           ! of particles is used
real(REALPREC), intent(in), dimension(Ny,dim2) :: vecIn    ! vector in observation space to which to
                                                           ! apply the observation error covariances R,
                                                           ! e.g. (HQH^T+R)^{-1}(d)
real(REALPREC), intent(inout), dimension(Ny,dim2) :: vecOut! resulting vector in observation space
```

cb_Qhalf(dim2,vecIn, vecOut)
Given *vecIn*, a vector or collection of vectors in state space, compute *vecOut* = $Q^{1/2}($*vecIn*$)$, where Q is model error covariance.

```
use sangoma_base   ! use sangoma defined precision for C-Bind
use user_base      ! use user defined parameters
implicit none

integer(INTPREC), intent(in) :: dim2                       ! second dimension of the vector,
                                                           ! i.e. dim2=1 if only one particle
                                                           ! or state vector is used and
                                                           ! dim2 = Ne if the whole ensemle
                                                           ! of particles is used
real(REALPREC), intent(in), dimension(Nx,dim2) :: vecIn    ! vector in state space to which to apply
                                                           ! the squarerooted model error covariances
                                                           ! Q^{1/2}, e.g. Q^{1/2}(d)
real(REALPREC), intent(inout), dimension(Nx,dim2) :: vecOut! resulting vector in state spacen
```

3.3 User defined parameters in 'user_base' module

In the module 'user_base' module a user can define all the parameters for the EWPF, specifically

- Dimension parameters:
  - *Ne* - number of particles
  - *Nx* - dimension of the state vector
  - *Ny* - dimension of the observation vector
- Parameters specific to the EWPF step for mixture density:

  - *efac = 0.001/real(Ne)*
  - *ufac = 0.00001*

  More information about these parameters and their values can be found in Ades and van Leeuwen [2012].

- Percentage of particles that can reach target weight: *keep*. Usually *keep = 0.8* is a good choice. See Ades and van Leeuwen [2012] on how the choice of *keep* affects EWPF results. Using *keep* we define number of particles that will be kept and moved in the equal weights step as *Ne_keep = nint(keep*Ne)*.

- Nudging parameters in *Bprime* routine:

  - *freetime* - a fraction of time between consequent observations sets, i.e. last and next that no nudging is performed. That is using `freetime = 0.6` will mean there will be no nudging until more than 60% of model timesteps between last and next observations have been realised. *freetime* is compared against *tau*, which is the fraction of time passed between two analyis times and given by

  $$tau = 1.0 - (t_{nextObs} - t_{current})/dt_{obs}.$$

  - *nudgefac* - parameter between 0 and 1 scaling the nudging factor.

Chapter 4

Implementation of the EWPF code in the different toolboxes

The implementation has been performed by the partners of the project into the different toolboxes (EMPIRE, NERSC EnKF toolbox, OAK, OpenDA, PDAF). Four of the implementations use the Fortran implementation of the EWPF, while the NERSC EnKF toolbox is coded for Matlab. This chapter describes the different implementations.

4.1 Implementation into EMPIRE

The EWPF has been implemented into EMPIRE (Employing MPI for Researching Ensembles) and full details, including EMPIRE code in FORTRAN, are available online www.met.reading.ac.uk/ darc/empire/doc/html/.

There are a number of particle and ensemble filters implemented in EMPIRE. A user can select the type of filter to be used by changing settings in 'pf_parameters.dat'. In particular for the EWPF, one has to select 'filter = 'EW''. The 'pf_parameters.dat' data file is also used to select other options for the filter such as observation generation, number of observations (if generated), run twin experiments or not. In addition, some of the parameters in 'user_base.F90' are defined in this file such as 'efac', 'ufac', 'keep', and 'nudgefac'.

When run, the first thing EMPIRE does is to call 'configure_model' routine which returns dimensions of the state and next set of observations as well as the time at which next observations are available. However, a user can and needs to use the 'configure_model' routine to load in memory other data such as data necessary for call-back routines 'cb_Qhalf' and 'cb_solve_r'. Further, if $\mathbf{HQH}^T + \mathbf{R}$ does not change over time then the user also can factorise it in 'configure_model' to save computational time later. After the assimilation of the first observations has been completed, EMPIRE calls 'reconfigure_model' which returns the time at which next observations are available and their dimension. This continues until all observations are assimilated.

4.1.1 Knowledge gained at EMPIRE

Being that EWPF was developed at UREAD by Van Leeuwen [2010, 2011] we have substantial knowledge in the algorithm and its nuances, such as the importance of nudging term and model error operator. Further UREAD developed and provided the code compliant with Sangoma data model in WP1 as well as implemented it in our own toolbox EMPIRE. Thus from this cross-implementation of one common tool in most of Sangoma toolboxes we mainly and importantly gained knowledge on how to organise code and work with partners who have drastically different systems, from languages that they use to data models. For example, we originally wrote the code in Fortran, but then released a Matlab version too since some partners preferred Matlab language and also since many tools in Sangoma are both available in Fortran and Matlab. Thus both versions of the code are compliant with Sangoma data model and have exactly the same structure and implementation instructions.

Further, from the point of view of the algorithm development, it has been very interesting to learn what other partners thought about the EWPF as a non-linear data assimilation method and developing discussion on comparisons between various ensemble data assimilation methods.

Finally, the EWPF is an important part of EMPIRE toolbox and will remain part of it in the future.

4.2 Implementation into NERSC assimilation system

The EnKF Matlab toolbox presently distributed on the webpage enkf.nersc.no would have needed too extensive structural changes to include the EWPF. NERSC has instead implemented the EWPF into a newer Matlab package for data assimilation education and research, designed to accommodate Particle Filters as much as Ensemble Kalman Filters.

The Data Assimilation and Ocean Forecasting group at NERSC is thus developing a Data Assimilation Toolbox in which different up-to-date data assimilation methods are implemented. The ultimate goal is to provide and maintain a stable and documented version of the Toolbox and make it available on enkf.nersc.no. At present the Toolbox is in its β-version, with different formulations of stochastic and square-root filters are implemented and a bootstrap particle filter. Necessary tests for stability of the code, its careful debugging and documentation are in order to move toward version α. The Toolbox is presently coded in Matlab but we plan a Python version after the end of Sangoma.

At the moment the Toolbox contains 4 models: Lorenz '63, Lorenz '96, a linear advection and a Quasi-Geostrophic model. It is possible to play with various models and observational setups, including observational interval, error covariances, degree of nonlinearity of the models and observational type. The EWPF has been implemented in the Toolbox and is now among the methods available for testing and benchmarking.

4.3 Implementation into OAK

The EWPF implementation in OAK is based on the reference Fortran code. OAK uses so far an off-line coupling between the model and the assimilation routines. However, the EWPF implementation requires a model state update at every time step, which is not practical with an off-line coupling. To integrate the EWPF scheme, we implemented an on-line coupling between the model and OAK. As in PDAF, subroutines for initialization and cleaning-up have to be added near the beginning and near the end of the main program of the model and a subroutine for the analysis inside the time-loop of the model. The online-coupling supports an already MPI parallelised model using sub-domains, but the number of processors must be equal to the number of sub-domains times the ensemble size. For the EWPF and the standard global assimilation scheme, the analysis and the proposal step are performed only by the first MPI process (master). Only the local analysis step is fully parallelised.

The implementation of all call-back routines were quite straight-forward, except for the routine `cb_solve_hqht_plus_r` which applies the inverse of the matrix $\left[\mathbf{H}^{(k)}\mathbf{Q}(\mathbf{H}^{(k)})^{\mathsf{T}} + \mathbf{R}\right]$ to a given vector. As this matrix is symmetric and positive defined we use the conjugate gradient algorithm to solve this matrix equation.

In order to activate the EWPF scheme, the parameter `schemetype` has to be set to the value 2 in the OAK configuration file. The parameters that can be used to tune the EWPF can also be set in this file by defining a section as follows.

```
# percentage of kept particles
EWPF.keep = 0.8
# normal standard deviation of 1
EWPF.nstd = 1.0
# normal mean of 0
EWPF.nmean = 0.0
# parameter for uniform distribution in equal_weight_step
EWPF.ufac = 0.00001
# parameter for uniform distribution in equal_weight_step
EWPF.efacNum = 0.001
# parameter in Bprime routine of freetime i.e. time with no nudging
EWPF.freetime = 0.6
# nudging strength parameter
EWPF.nudgefac = 0.9
```

These are also the default values of these parameters (as defined in the module `user_base.f90`). At any analysis cycle the weight can be saved by adding an entry for diagnostics:

```
Diag001.weigthf = 'file.nc#weightf'
Diag001.weigtha = 'file.nc#weighta'
```

Using these lines, one can save the weights before (weightf) and after the analysis (weighta) into the NetCDF file `file.nc` as the NetCDF variables `weightf` and weigtha respectively.

4.3.1 Knowledge gained at OAK

For the OAK toolbox, the adapted data model was suitable to integrate the provided Fortran analysis scheme into OAK. The data model makes intensive use of call-back routines using only essential parameters that the EPWF analysis scheme needs to know about (which is indeed a good programming practice). However, to implement the call-back routine, access to additional variables (sometimes specific to OAK) are necessary. One solution could be to make all these variables global. This is illustrated in the following code where the call-back function representing the observation operator needs the location of the observations, but this information is not necessary for the EWPF routine `equal_weight_step` and therefore not a parameter passed to this routine.

```
module oak
  ! location of the observations needed for the observation operator
  real :: obsgrid(...)

contains
  subroutine ewpf_analysis(x...)
    call equal_weight_step(...,cb_H,...)
  end subroutine

 ! observation operator
 subroutine cb_H(x,Hx)
    ! needs obsgrid and available though global scope
 end subroutine
```

However, global variables are avoided in OAK due to various reasons (implicit coupling of subroutines, potential bugs due to mutable global variables, increased difficulty to write unit tests, inflexibility and potential concurrency issues), at least for new code added to OAK. The solution to this problem was to use nested call-back functions, which is possible in Fortran 2003:

```
module oak
contains
  ! location of the observations is explicitly passed as argument

  subroutine ewpf_analysis(x, obsgrid...)
    call equal_weight_step(...,cb_H,...)

  contains
   ! observation operator
   subroutine cb_H(x,Hx)
     ! needs obsgrid and available though local scope
   end subroutine
  end subroutine
end module
```

Nested subroutines can access variables of the parent subroutine. Alternative ways to avoid global variables are discussed in **?** which include a reverse communication interface (relatively complex to implement) and Fortran 2008 objects (not compatible with the adopted data model).

4.4 Implementation into OpenDA

The EWPF reference implementation in SANGOMA is written in Fortran90. Our goal was to not reimplement the algorithm but to directly use the reference implementation. OpenDA is written in Java except for some computational intensive parts. These parts are written in C and a bit in Fortran90. This code is called the native part of OpenDA. To implement the EWPF algorithm we have included the whole Fortran90 toolset from SANGOMA in the native part of OpenDA. All SANGOMA routines are used as is, which allows easy upgrading in case the SANGOMA tools are improved/updated in the future. We use JNA (`https://en.wikipedia.org/wiki/Java_Native_Access`) to interface between the Java and the SANGOMA tools. In order to add the EWPF to OpenDA we have

- added the SANGOMA Fortran90 tools and EWPF code to the native part of OpenDA. We do not use the make system of the SANGOMA tools but the OpenDA make system to compile the SANGOMA tools into a dynamic link library,
- implemented a Fortran90 module (`oda_EWPF_wrapper.f90`) that implements the user routines needed by the SANFOMA EWPF implementation and some interface routines that are actually called from Java. These interface routines are needed to pass and initialize the user routines,
- added a Java implementation for interfacing with the routines in `oda_EWPF_wrapper.f90`. The EWPF method fits within the ensemble-based data assimilation framework of OpenDA. Therefore our java code only needs to implement the "`analysis`" method. The analysis calls, indirectly using `oda_EWPF_wrapper.f90`, the SANGOMA EWPF routines `proposal_step` and `equal_weight_step` respectively at nudging and analysis times.

The user can use the EWPF method by selecting it in the OpenDA configuration file in the following way:

```
<algorithm className="org.openda.algorithms.particleFilter.EWPF">
<workingDirectory>./algorithm</workingDirectory>
<configString>EWPF_config.xml</configString>
</algorithm>
```

The observation handling in OpenDA is very strict when it comes to matching model times with observation times. This is a small complication for the EWPF algorithm where we match model results to observations in the future. In a similar way, the ensemble-based data assimilation framework in which the EWPF is implemented prefers to run from observation time to the next observation time not stopping the model in between. The solution is to add to the OpenDA observation component an option to virtually create observations at the nudging time instances with an observed value corresponding to the "real" observation with is used at the re-weighting step. In this way, using these virtual/fake observations, the EWPF algorithm fits nicely in the time stepping and interpolation framework.

The code is available as part of OpenDA.

4.4.1 Knowledge gained at OpenDA

The EWPF is part of OpenDA and has been implemented using the the Sangoma fortran code directly (i.e. without any changes to the implementation code). Currently it is part of the development version but it will be part of the upcoming releases.

We have been experimenting with simple Lorenz models (mostly the 3-variable butterfly) to test the EWPF method, so far the results are not what we have expected, and this seems to be due to the addition of the noise in the proposal step where each particle is nudged towards future observations. Hence, we need to run more experiments and learn more about the nudging step for each model to see if and how we could unlock the potential of the algorithm.

4.5 Implementation into PDAF

PDAF [Nerger and Hiller, 2013] has an internal interface to all filter methods. The webpage (http://pdaf.awi.de/trac/wiki/AddFilterAlgorithm) describes the implementation of an additional filter in PDAF. Following this description, it is possible to add the SANGOMA EWPF as a new filter in PDAF.

Since a formulation of the EWPF is already included in the work version of PDAF, it was decided against the approach to add the SANGOMA EWPF as a completely new filter and instead to implement the SANGOMA routines as a new subtype of the existing EWPF filter. Thus, when the initialisation routine `PDAF_init` is called, the user has to provide the desired subtype. Currently three different formulations are available (see Table 4.1).

Subtype	Filter
0	Semi parallelized EWPF
1	Full parallelized EWPF
2	SANGOMA EWPF

Table 4.1: Available EWPF subtypes in PDAF

As described in the presentation of the EWPF algorithm, at each model time step in this assimilation method, the model states need to be changed either by the ‘*proposal step*’ or the ‘*equivalent weights step*’ of the EWPF. This change is done by adding a single call to the assimilation routine (see Fig. 4.1) in the time stepping routine of the model code.

```
CALL PDAF_assimilate_ewpf_si(outflag)
   INTEGER :: outflag     ! Status output flag of PDAF
```

Figure 4.1: Call to PDAF in the model code

Using this implementation concept, the call to PDAF from the model is identical for each subtype of the EWPF. The user just needs to provide the correct callback functions in the call, since they may differ for different implementations of the EWPF. The call-back routines have been implemented using default names defined inside `PDAF_assimilate_ewpf_si`. The implementation was straight forward and mainly we were able to use call-back routines of the already existing EWPF implementation.

In order to keep track of the timings, at the beginning of a new assimilation cycle, a variable `nsteps` is initialised internally in PDAF as the number of model steps between the current time step and the new observation time step. Using this variable, it is possible to distinguish whether the proposal or equal weights update needs to be done. It should always be possible to initialise the variable `nsteps` at the beginning of the assimilation interval, since by then the observations for the next assimilation time need to be available.

Inside the routine `PDAF_assimilate_ewpf_si` the filter routines themselves are called as is exemplified in Fig. 4.2. Here, the names of the callback routines used the PDAF-internal naming scheme.

To include the SANGOMA EWPF into this routine it is only necessary to call the SANGOMA EWPF routines inside of this routine and provide the necessary input variables and callback routines (Those variables starting with '`U_`'). The implementation supports ensemble parallelization, which uses already existing routines in PDAF.

To be able to modify the parameter values that are defined in the module `user_base` of the SANGOMA EWPF, the module was included with '`use`' in the routine `PDAF_assimilate_ewpf`. Then, the parameters, like the nudging strength of the proposal step can been adjusted when calling the initialization routine of PDAF.

4.5.1 Knowledge gained at PDAF

Overall, the inclusion of the EWPF code into PDAF was very easy as the data model is fully compatible the the interface standard used in PDAF. The routines '`proposal_step`' and '`equal_weight_step`' are integrated into the core library part of PDAF, in which the scope of variables is clearly separated from the scope of model- and observation-specific user routines like the call-back routine providing the observation operator.

Compared to other filter methods already available in PDAF, the requirement of the additional Fortran module '`user_base`' was a bit unfortunate. One could easily avoid this module, which is only used for the SANGOMA version of the EWPF by adding the parameters set in the module as arguments to the interface of the routines '`proposal_step`' and '`equal_weight_step`'. Here not the full list of parameters would be required for both routines, but only those used in the

```
! Increment time step counter
cnt_steps = cnt_steps + 1

cntloop:  IF (cnt_steps < nsteps) then
  ! *****************************************************************
  ! ***   At each time step without observations- Proposal Step    ***
  ! *****************************************************************
  CALL proposal_step(dim_ens, dim_p, dim_obs, weights, eofV, &
           observation, cnt_steps, nsteps, nsteps, &
           U_obs_op, U_adjoint_obs_op, U_sqrtQ, U_prodRinvA)

ELSEIF (cnt_steps == nsteps) THEN cntloop
  ! *****************************************************************
  ! ***   At observation time, apply the EWPF  Update              ***
  ! *****************************************************************
  CALL equal_weight_step(dim_ens, dim_p, dim_obs, weights, eofV, &
           observation, U_obs_op, U_adjoint_obs_op, U_prodRinvA, &
           U_solve_invHQHTpR, U_sqrtQ)

  !Resets timestep
  nsteps = steps

END IF cntloop
```

Figure 4.2: Extract of `PDAF_assimilate_ewpf`

respective routine. Obviously, this strategy would make the argument list a bit longer. However, it would avoid an additional Fortran module and would make it more explicit, which variables or parameters are required in the routines.

The SANGOMA EWPF implementation was compared to the previously existing EWPF in PDAF in numerical experiments using the Lorenz-96 model. The results were found tobe comparable, but not the same due to the use of different random numbers.

Chapter 5

Internal structure of the EWPF code

Chapter 3 explained the main routines of the EWPF code, which are directly called by the user code, and the model specific call-back routines. In this chapter we now document the internal structure of the EWPF routines and list the interfaces of all the internal routines that are collectively called by routines 'proposal_step' and 'equal_weights_step'. Figures 5.1 and 5.2 show the calling trees of all internal calls of these two routines.

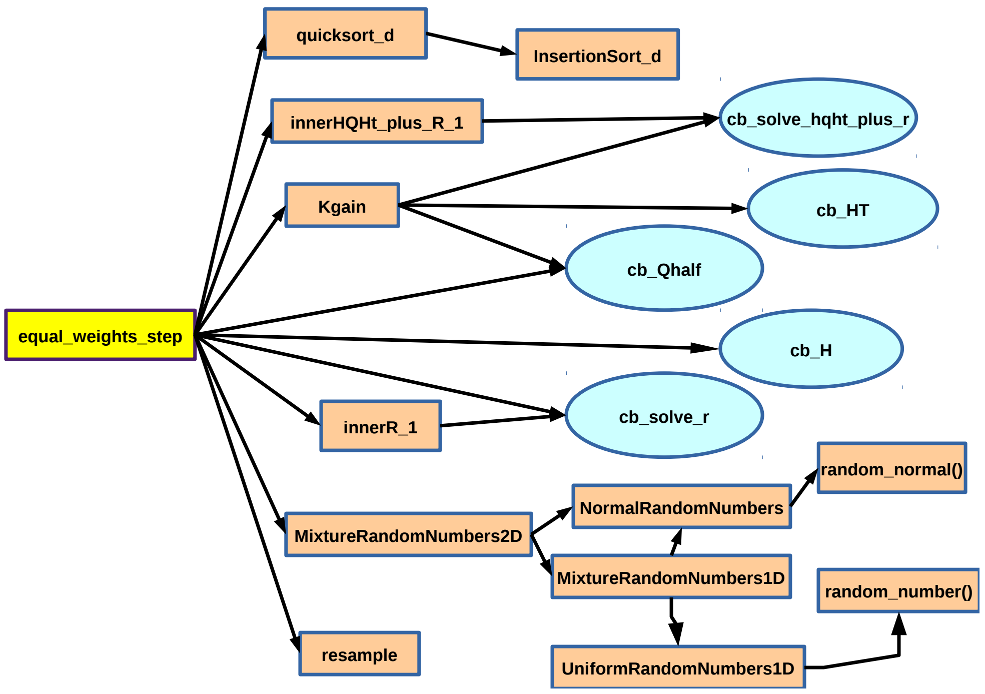

Figure 5.1: Schematic of the 'equal_weights_step' routine. Colour decoding: yellow routine is called from user code, peach routines are internal fully defined routines, blue oval routines are internal call-back routines that are model specific and need to be implemented by the user.

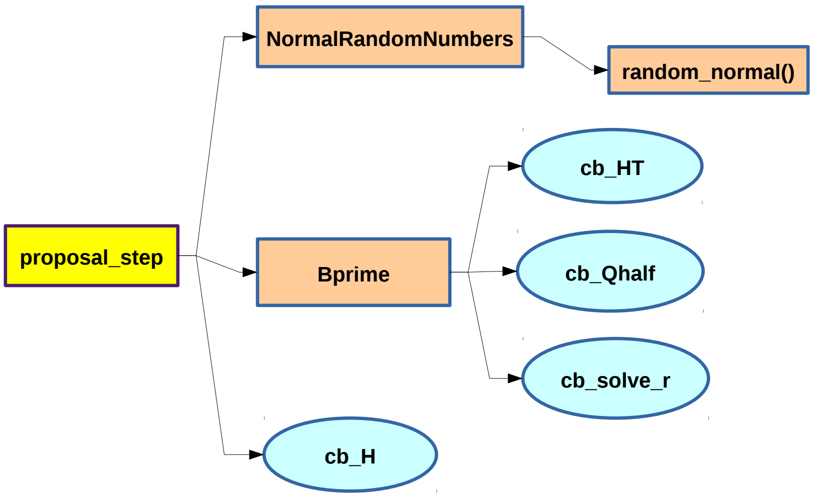

Figure 5.2: Schematic of the 'proposal_step' routine. The colors are as in Fig. 5.1.

Bprime(d,kgain,QHtR_1d,betan,tt,obsVec,dt_obs)
Compute the nudging term *QHtR_1d* and random forcing *betan* to nudge particles towards future observations. Also return *kgain* part of the nudging term which is used in weight adjustment due to the nudging.

```
integer(INTPREC),intent(in) :: tt                               ! current model tt
integer(INTPREC),intent(in) :: obsVec                           ! next observation time step
integer(INTPREC),intent(in) :: dt_obs                           ! model timesteps between observations
real(REALPREC), dimension(Ny,Ne), intent(in) :: d               ! a vector d = (y-H(x)) from proposal_filter
                                                                ! Note, that d is the difference between current
                                                                ! particle states and future observation
real(REALPREC), dimension(Nx,Ne), intent(out) :: kgain          ! kgain in the nudging
real(REALPREC), dimension(Nx,Ne), intent(out) :: QHtR_1d        ! QH^TR^{-1}*d
real(REALPREC), dimension(Nx,Ne), intent(out) :: betan          ! betan ~ N(0,Q)\\
```

resample(weight, x_n)
This subroutine takes the full state and particle array, *x_n*, and their corresponding weights, *weight*, and performs universal resampling to return *Ne* resampled particles with exactly equal weights of *1/Ne*. Importantly, both arrays *x_n* and *weight* are passed to *resample* sorted such that first *Ne_keep* particles are to be used to resample full *Ne* ensemble.

```
use sangoma_base ! use sangoma defined precision for C-Bind
use user_base    ! use user defined parameters
implicit none

real(REALPREC), intent(inout), dimension(Nx,Ne) :: x_n        ! ensemble particle matrix
real(REALPREC), intent(inout), dimension(Ne) :: weight        ! particle weights
```

Kgain(vecIn,vecOut)

Subroutine to apply the operator *Kgain* to a vector *vecIn* in observation space and return the vector *vecOut* in state space.

```
use sangoma_base   ! use sangoma defined precision for C–Bind
use user_base      ! use user defined parameters
implicit none

real(REALPREC), dimension(Ny,Ne), intent(in) :: vecIn   ! matrix of all  particle states in observation space
real(REALPREC), dimension(Nx,Ne), intent(out) :: vecOut ! resulting matrix in full space (of all particles)
```

innerR_1(dim2,vecIn,vecOut)

Subroutine to take an observation array *y* and return in *w* an inner product of it scaled by observation error covariance R^{-1}, i.e. $w_i = y_i^T R^(-1) y_i$ for each input y_i.

```
use sangoma_base   ! use sangoma defined precision for C–Bind
use user_base      ! use user defined parameters
implicit none

integer(INTPREC), intent(in) :: dim2                   ! second dimension of the vector,
                                                       ! i.e. dim2=1 if only one particle
                                                       ! or state vector is used and
                                                       ! dim2 = Ne if the whole ensemle
                                                       ! of particles is used
real(REALPREC), dimension(Ny,Ne), intent(in) :: vecIn  ! input vector(s)
real(REALPREC), dimension(Ne), intent(out) :: vecOut   ! resulting inner product
```

innerHQHt_plus_R_1(dim2,vecIn,vecOut)

Subroutine to take an observation array *y* and return in *w* an inner product of it scaled by innovation error covariance $(HQH^T + R)^{-1}$, i.e. $w = y^T(HQH^T + R)^{-1}y$.

```
use sangoma_base   ! use sangoma defined precision for C–Bind
use user_base      ! use user defined parameters
implicit none

integer(INTPREC), intent(in) :: dim2                     ! second dimension of the vector,
                                                         ! i.e. dim2=1 if only one particle
                                                         ! or state vector is used and
                                                         ! dim2 = Ne if the whole ensemle
                                                         ! of particles is used
real(REALPREC), dimension(Ny,dim2), intent(in) :: vecIn  ! input vector
real(REALPREC), dimension(dim2), intent(inout) :: vecOut ! output vector
```

quicksort_d(a, idx_a, na)

Recursive subroutine to sort using the quicksort algorithm.

```
use sangoma_base, only: REALPREC, INTPREC
implicit none

integer(INTPREC), intent(in) :: na                        ! nr or items to sort
real(REALPREC), dimension(nA), intent(inout) :: a         ! vector to be sorted
integer(INTPREC), dimension(nA), intent(inout) :: idx_a   ! sorted indecies of a
```

NormalRandomNumbers(mean, stdev, n, k, phi)

Subroutine to generate, *phi*, an array of size $n \times k$ of normally distributed numbers with given *mean* and *stdev*.

```
use sangoma_base, only: REALPREC, INTPREC
use random
IMPLICIT NONE

integer(INTPREC), INTENT(IN) :: n                     ! first dimension of output vector
integer(INTPREC), INTENT(IN) :: k                     ! second dimension of output vector
real(REALPREC), INTENT(IN) :: mean                    ! mean of normal distribution
real(REALPREC), INTENT(IN) :: stdev                   ! Standard Deviation of normal distribution
real(REALPREC), dimension(n,k), INTENT(OUT) :: phi    ! n,k dimensional normal random numbers
```

MixtureRandomNumbers2D(mean, stdev, ufac, epsi, n, k, phi, uniform)
Subroutine to generate, *phi*, a collection (*k* of them) of vectors drawn from mixture (normal and uniform) density.

```
use random
use sangoma_base, only: REALPREC, INTPREC
implicit none

real(REALPREC), intent(in) :: mean                  ! Mean of normal distribution
real(REALPREC), intent(in) :: stdev                 ! Standard deviation of normal distribution
real(REALPREC), intent(in) :: ufac                  ! half-width of uniform distribution
                                                    ! that is centered on the mean
real(REALPREC), intent(in) :: epsi                  ! Proportion controlling mixture draw
                                                    ! if random_number > epsi then draw from
                                                    ! uniform, else normal
integer(INTPREC),intent(in) :: n,k                  ! first and second dimension of output vector
real(REALPREC), dimension(n,k), intent(out) :: phi ! n,k dimensional mixture random numbers
logical, dimension(k), intent(out) :: uniform       ! k dimensional logical with uniform(i) True if
                                                    ! phi(:,i) drawn from uniform.
                                                    ! False if drawn from normal
```

Chapter 6

Summary

This deliverable discussed the implementation of an equivalent weights particle filter (EWPF). Subroutines for the filter have been coded in Fortran and as Matlab scripts according to the data model and interface standard defined in Deliverable 1.3 of SANGOMA. Based on the documentation of the routines, the different partners of the project implemented the filter into their respective data assimilation toolboxes.

The implementations showed that the interfaces defined by the data model are well usable by the different toolboxes of the partners. However, a particular difficulty was found in the implementations which arises from a particular feature of the EWPF. Namely, the EWPF performs a nudging step (the 'proposal_step') after each time step of the model. This proposal_step utilizes the observational information from a future time to guide the ensemble of particles toward the observations assimilated next and ensures that no particle gets a negligible weight. However, to allow the toolboxes to compute they need to access the whole ensemble. If the model and toolbox are coupled offline, this implies that the ensemble of models have to write restart files after each time step, which are then read by the toolbox and modified in the proposal_step. This data exchange through files results in a significant overhead in computing time, compared to online-coupling in which the assimilation toolbox has direct access to the ensemble of particles in memory of the running ensemble. Such a strategy is followed natively by the EMPIRE and PDAF toolboxes, but also for OAK an online-coupling was implemented.

Overall, the implementations have shown that the data model is well suited for the implementation of filter methods, even if they have the complexity of the EWPF.

Bibliography

M. Ades and P. J. van Leeuwen. An exploration of the equivalent weights particle filter. *Quarterly Journal of Royal Meteorological Society*, pages n/a–n/a, 2012. doi: 10.1002/qj.1995.

S. Metref, E. Cosme, C. Snyder, and P. Brasseur. A non-gaussian analysis scheme using rank histograms for ensemble data assimilation. *Nonl. Proc. Geoph.*, 21:869–885, 2014.

L. Nerger and W. Hiller. Software for ensemble-based data assimilation systems - implementation strategies and scalability. *Computers & Geosciences*, 55: 110–118, 2013.

P. J. Van Leeuwen. Nonlinear data assimilation in qeosciences: an extremely efficient particle filter. *Quarterly Journal of the Royal Meteorological Society*, **136**:1991–1999, 2010.

P. J. van Leeuwen. Nonlinear data assimilation in geosciences: An extremely efficient particle filter. *Q. J. Roy. Meteor. Soc.*, 136:1991–1999, 2010.

P. J. Van Leeuwen. Efficient nonlinear data-assimilation in geophysical fluid dynamics. *Computers and Fluids*, **46**:52–58, 2011.

P. J. van Leeuwen and M. Ades. Efficient fully nonlinear data assimilation for geophysical fluid dynamics. *Computers & Geosciences*, 55:16–27, 2013.