Framework for Spatio-Temporal Modelling

Supporting Deterministic and Stochastic Modelling, Data Assimilation and Model Calibration

Oliver Schmitz
Derek Karssenberg

Framework for Spatio-Temporal Modelling: Supporting Deterministic and Stochastic Modelling, Data Assimilation and Model Calibration

by Oliver Schmitz and Derek Karssenberg

Published 2011-03-22

Faculty of Geosciences, Utrecht University, The Netherlands

Table of Contents

List of Examples

Chapter 1. Introduction

The framework supports static models without a time component and temporals models wich simulate a dynamic processes. Both types can be used for Monte Carlo [2], particle filter [11] or Ensemble Kalman filter [5] simulations. With the PCRaster Python extension [14],[13] models can be extended by a spatial component.

Two steps in the model development cycle are the conversion of the conceptual model structure into computer code and the assimilation or calibration of the model with observational data [6]. This framework combines the tasks of model construction and optimisation in a single framework.

The framework does not require to use the PCRaster environmental modelling language even so the example scripts in the following sections solely make use of it.

1.1. Requirements

To use the framework you need to install Python 2.5 (http://www.python.org/) and NumPy (http://numpy.scipy.org/). To generate the output graphs for the particle filter you need to install graphviz (http://www.graphviz.org/).

Furthermore the framework directory must be added to the PYTHONPATH environment variable.

The examples in the demo directory require the PCRaster Python extension.

Chapter 2. Modelling frameworks

2.1. Quickstart

In addition to spatial functions a support frame is required to build spatio-temporal models. This section introduces two frameworks that ease the development of static and dynamic models.

The script below is the Python version of the hydrological runoff model shown in the demo of the PCRaster distribution (http://pcraster.geo.uu.nl/models/catsop/index.html). To run the script change to the demo directory (demo/deterministic) and execute **python2.5 runoff.py**.

Example 2.1. Python script `runoff.py` specifying a spatio-temporal model using the dynamic modelling framework.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# model for simulation of runoff
# 24 timesteps of 6 hours => modelling time one week

from PCRaster import *
from PCRaster.Framework import *

class RunoffModel(DynamicModel):
  def __init__(self, cloneMap):
    DynamicModel.__init__(self)
    setclone(cloneMap)

  def initial(self):
    # coverage of meteorological stations for the whole area
    self.rainZones = spreadzone("rainstat.map", scalar(0), scalar(1))

    # create an infiltration capacity map (mm/6 hours), based on the
    # soil map
    self.infiltrationCapacity = lookupscalar("infilcap.tbl", "soil.map")
    self.report(self.infiltrationCapacity, "infilcap")

    # generate the local drain direction map on basis of the elevation map
    self.ldd = lddcreate("dem.map", 1e31, 1e31, 1e31, 1e31)
    self.report(self.ldd, "ldd")

    # initialise timeoutput
    self.runoffTss = TimeoutputTimeseries("runoff", self, "samples.map", noHeader=False)

  def dynamic(self):
    # calculate and report maps with rainfall at each timestep (mm/6 hours)
    surfaceWater = timeinputscalar("rain.tss", self.rainZones)
    self.report(surfaceWater, "rainfall")

    # compute both runoff and actual infiltration
    runoff = accuthresholdflux(self.ldd, surfaceWater,\
         self.infiltrationCapacity)
    infiltration = accuthresholdstate(self.ldd, surfaceWater,\
         self.infiltrationCapacity)

    # output runoff, converted to m3/s, at each timestep
```

```
    logRunOff = runoff / scalar(216000)
    self.report(logRunOff, "logrunof")
    # sampling timeseries for given locations
    self.runoffTss.sample(logRunOff)

myModel = RunoffModel("mask.map")
dynModelFw = DynamicFramework(myModel, lastTimeStep=28, firstTimestep=1)
dynModelFw.run()
```

2.2. Deterministic Modelling

2.2.1. Static Modelling Framework

This section introduces to the usage of the static modelling framework. A static model is described by:

Z = f(Z, I, P)

with Z, the model state variables; I, inputs; P, parameter; and f defining the model structure. The static model framework is used to build models without temporal dependencies, like calculating distances between a number of gauging stations.

2.2.1.1. Static model template

The following script shows the minimal user class that fulfils the requirements for the static framework:

```
# userModel.py
from PCRaster.Framework import *

class UserModel(StaticModel):
  def __init__(self):
    StaticModel.__init__(self)

  def initial(self):
    pass
```

In the class of the user model the following method must be implemented:

`initial()` This method contains the static section of the user model.

The model class can be executed with the static framework as follows:

```
# runScript.py

import userModel
from PCRaster.Framework import *

myModel = userModel.UserModel()
staticModel = StaticFramework(myModel)
staticModel.run()
```

To run the model execute **python2.5 runScript.py**. The script `runScript.py` creates an instance of the user model which is passed to the static framework afterwards. `staticModel.run()` executes the initial section of the user model.

2.2.1.2. Example

The following example shows the static version of the demo script. PCRaster operations can be used in the same way as in scripts without the modelling framework:

Example 2.2. Static version of the demo script.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# static model

from PCRaster import *
from PCRaster.Framework import *

class RunoffModel(StaticModel):
  def __init__(self, cloneMap):
    StaticModel.__init__(self)
    setclone(cloneMap)

  def initial(self):
    # coverage of meteorological stations for the whole area
    self.rainZones = spreadzone("rainstat.map", scalar(0), scalar(1))

    # create an infiltration capacity map (mm/6 hours), based on the
    # soil map
    self.infiltrationCapacity = lookupscalar("infilcap.tbl", "soil.map")
    self.report(self.infiltrationCapacity, "infilcap")

    # generate the local drain direction map on basis of the elevation map
    self.ldd = lddcreate("dem.map", 1e31, 1e31, 1e31, 1e31)
    self.report(self.ldd, "ldd")

myModel = RunoffModel("mask.map")
stModelFw = StaticFramework(myModel)
stModelFw.run()
```

Setting the map attributes (e.g. number of rows and columns and cellsize) is done by using `setclone` in the constructor of the model class.

PCRaster operations can have data from disk as input arguments, as is done in the `spreadzone` operation.

Note that the framework provides an additional report operation (`self.report`) whose behavior is dependent on the method in which it is used. It writes the data to disk with a filename conforming to the PCRaster conventions generated from the second argument, i.e. appending a ".map" suffix for the static framework (the name of the local drain direction map will become "ldd.map") and appending a time step when used in the dynamic framework. Storing data with a specific name or at a specific location is done using `report` instead of `self.report`.

2.2.2. Dynamic Modelling Framework

This section describes the usage of the dynamic modelling framework. In addition to spatial processes dynamic models include a temporal component. Simulating dynamic behaviour is done by iterating a dynamic section over a set of timesteps.

The state of a model variable at time t is defined by its state at t-1 and a function f [7]:

$Z_{1..m}(t) = f(Z_{1..m}(t\text{-}1), I_{1..n}(t), P_{1..l})$

The model state variables $Z_{1..m}$ belong to coupled processes and have feedback in time. $I_{1..n}$ denote the inputs to the model, $P_{1..l}$ are model parameters, and f transfers the model state from time step t-1 to t.

The dynamic modelling framework executes f using the following scheme (in pseudo code):

```
initial()
```

```
for each timestep:
  dynamic()
```

2.2.2.1. Dynamic model template

The following script shows the minimal user class that fulfils the requirements for the dynamic framework:

```
# userModel.py
from PCRaster.Framework import *

class UserModel(DynamicModel):
 def __init__(self):
  DynamicModel.__init__(self)

 def initial(self):
  pass

 def dynamic(self):
  pass
```

In the class of the user model the following methods must be implemented:

`initial()` This method contains the code to initialise variables used in the model.

`dynamic()` This method contains the implementation of the dynamic section of the user model.

Applying the model to the dynamic framework is done by:

```
# runScript.py

import userModel
from PCRaster.Framework import *

myModel = userModel.UserModel()
dynModel = DynamicFramework(myModel, 50)
dynModel.run()
```

To run the model execute **python2.5 runScript.py**. The script `runScript.py` creates an instance of the user model which is passed to the dynamic framework. The number of time steps is given as second argument to the framework constructor.

2.2.2.2. Example

An script for a dynamic model is given in the quick start section in Example 2.1. The model contains two main sections: The `initial` section contains operations to initialise the state of the model at time step 0. Operations included in this section are executed once. The `dynamic` section contains the operations that are executed consecutively each time step. Results of a previous time step can be used as input for the current time step. The dynamic section is executed a specified number of timesteps: 28 times in the demo script.

The initial section of the demo script is the same as in the static version. The dynamic section holds the operations for in- and output with temporal dependencies and the model processes. For time series input data the `timeinputscalar` assigns precipitation data for each time step to the `surfaceWater` variable. In the case that `rain0000.001` to `rain0000.028` hold the rainfall for each timestep instead you can replace the `timeinputscalar` operation by `surfaceWater = self.readmap("rain")`.

Output data is now reported as a stack of maps to disk. The function `self.report` will store the runoff with filenames `logrunof.001` up to `logrunof.028`.

For additional operations that can be used for example in conditional expressions like `self.currentTimestep` we refer to the code reference for this topic.

2.3. Stochastic Modelling and data assimilation

In the case that a model includes probabilistic rules or inputs variables and parameters that are given as spatial probability distributions, the model becomes stochastic [8]. The aim of stochastic modelling is to derive the probability distributions, which is done in the framework by Monte Carlo simulation.

The framework provides three different methods to support stochastic modelling and data assimilation: Monte Carlo simulation (e.g. [3], [2]), particle filter (e.g. [12], [11], [1]) and the Ensemble Kalman filter (e.g. [5],[10]).

2.3.1. Monte Carlo simulations

Monte Carlo simulations solve for a large number of samples the function f and compute statistics on the ensemble results. The framework supports this scheme by executing the following methods (in pseudo code):

```
premcloop()

for each sample:
  initial()
  if dynamic model:
    for each timestep:
      dynamic()

postmcloop()
```

The following additional methods must be implemented to use the framework in Monte Carlo mode:

`premcloop()` The premcloop can be used to calculate input parameters or variables that are both constant and deterministic. It is executed once at the beginning of the model run, the calculate variables can be used in all samples and time steps.

`postmcloop()` The postmcloop is executed after the last sample run is finished. It is used to calculate statistics of the ensemble, like variance or quantiles.

The `initial` and `dynamic` sections (the latter in case of a dynamic model for each time step) are executed for each Monte Carlo sample.

The framework generates samples directories named 1, 2, 3,..., N, with N the number of Monte Carlo samples. The methods `self.readmap()` and `self.report()` now read and store the data to and from the corresponding sample directory.

2.3.1.1. Static models

The Python script in Example 2.3 shows a static model which is executed within the Monte Carlo framework. The model simulates vegetation growth; 100 realisations are executed [8].

Example 2.3. Python script specifying a static model executed in the Monte Carlo framework.

```
from PCRaster import *
from PCRaster.Framework import *

class VegetationGrowthModel(StaticModel, MonteCarloModel):
  def __init__(self):
    StaticModel.__init__(self)
    MonteCarlo.__init__(self)
    setclone("clone.map")

  def premcloop(self):
    pass

  def initial(self):
```

```
    # spreading time for peat (years)
    peatYears = 0.1 + mapnormal() * 0.001
    # spreading time for other soil types (years)
    otherYears = 0.5 + mapnormal() * 0.02
    # number of years needed to move the vegetation front 1 m
    years = ifthenelse("peat.map", peatYears, otherYears)
    # time to colonization (yr)
    colTime = spread("distr.map", years)
    # colonized after 50 years?
    col = ifthen(colTime < 50)
    self.report(col, "col")

  def postmcloop(self):
    names = ["col"]
    mcaveragevariance(names, "", "")

myModel = VegetationGrowthModel()
staticModel = StaticFramework(myModel)
mcModel = MonteCarloFramework(staticModel, 100)
mcModel.run()
```

First, maps are created containing for each cell the time (years) needed for the plant to spread 1 m. The value and the error associated with this input parameter depend on the soil type. By using the function `mapnormal` each sample will generate an independent realisation of the input parameter `peatYears` and `otherYears`. The information is used to calculate a total spreading time map from the locations occupied with the plant on `distr.map`. Finally a Boolean map is generated containing all cells colonized within 50 years.

2.3.1.2. Dynamic models

The Python script in Example 2.4 shows a dynamic model which is executed within the Monte Carlo framework. The model simulates snow thickness and discharge for 180 time steps ([9]). A number of 10 realisations is executed.

Example 2.4. Python script `montecarlodyn.py` specifying a dynamic model executed in the Monte Carlo framework.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from PCRaster import *
from PCRaster.Framework import *

class SnowModel(DynamicModel, MonteCarloModel):
  def __init__(self):
    DynamicModel.__init__(self)
    MonteCarloModel.__init__(self)
    setclone("clone.map")

  def premcloop(self):
    dem = self.readmap("dem")
    self.ldd = lddcreate(dem, 1e31, 1e31, 1e31, 1e31)
    elevationMeteoStation = scalar(2058.1)
    self.elevationAboveMeteoStation = dem - elevationMeteoStation
    self.degreeDayFactor = 0.01

  def initial(self):
    self.snow = scalar(0)
    self.temperatureLapseRate = 0.005 + (mapnormal() * 0.001)
    self.report(self.temperatureLapseRate, "lapse")
```

```
    self.temperatureCorrection = self.elevationAboveMeteoStation\
         * self.temperatureLapseRate

  def dynamic(self):
    temperatureObserved = self.readDeterministic("tavgo")
    precipitationObserved = self.readDeterministic("pr")
    precipitation = max(0, precipitationObserved * (mapnormal() * 0.2 + 1.0))
    temperature = temperatureObserved - self.temperatureCorrection
    snowFall = ifthenelse(temperature < 0, precipitation, 0)
    self.snow = self.snow + snowFall
    potentialMelt = ifthenelse(temperature > 0, temperature\
         * self.degreeDayFactor, 0)
    actualMelt = min(self.snow, potentialMelt)
    self.snow = max(0, self.snow - actualMelt)
    rain = ifthenelse(temperature >= 0, precipitation, 0)
    discharge = accuflux(self.ldd, actualMelt + rain)
    self.report(self.snow, "s")
    self.report(discharge, "q")

  def postmcloop(self):
    names = ["s", "q"]
    mcaveragevariance(names, self.sampleNumbers(), self.timeSteps())
    percentiles = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
    mcpercentiles(names, percentiles, self.sampleNumbers(), self.timeSteps())

myModel = SnowModel()
dynamicModel = DynamicFramework(myModel, lastTimeStep=180, firstTimestep=1)
mcModel = MonteCarloFramework(dynamicModel, nrSamples=10)
mcModel.run()
```

To run the model execute **python2.5 montecarlo.py**.

In the `premcloop`, the local drain direction map is created from the digital elevation map `dem.map`. As the local drain direction map is used in the initial and dynamic section later on it is defined as a member variable of the snowModel class. If maps are reported in the `premcloop` they are written into the current working directory.

The `initial` section initialises for each realisation the state variables snow with an original value of zero. Also a realisation of the lapse rate is made from a probability distribution 0.005 + norm(0, 0.001). Each lapse rate is stored with `self.report` in the corresponding sample subdirectory.

The `dynamic` section calculates the snow height and the discharge. The temperature and precipitation values are obtained from disk with the `self.readDeterministic` operation. Random noise is added to the deterministic precipitation values in order to create independent realisations. The precipitation is diminished by the potential snowfall, which builds up the snow depth. Runoff and snowmelt compose the discharge in the catchment.

The `postmcloop` calculates statistics over all ensemble members. For snow and discharge, average and variance values are calculated. Furthermore the percentiles are calculated for both variables. The resulting maps of the `postmcloop` calculations are written to the current working directory.

2.3.2. Particle filter

The particle filter is a method to improve the model predictions. Observed values are hereby used at specific time steps to determine the best performing samples. The prediction performance of the ensemble is improved by continuing better performing and omitting bad samples.

The particle filter approximates the posterior probability density function from the weights of the Monte Carlo samples [9]:

$$p(x_t|Y_t) \approx \sum_{n=1}^{N} p_t^{(n)} \delta(x_t - x_t^{(n)})$$

with δ the Dirac delta function, Y_t the past and current observations at time t and x_t a vector of model components for which observation are available.

For Gaussian measurement error the weights are proportional to [10]:

Equation 2.1.

$$a_t^{(n)} = \exp(-[y_t - h_t(x_t^{(n)})]^T R_t^{-1}[y_t - h_t(x_t^{(n)})]/2)$$

with R_t the covariance matrix of the measurement error and h_t the measurement operator.

The weight of a sample is calculated by a normalisation of $a_t^{(n)}$:

$$p_t^{(n)} = a_t^{(n)} / \sum_{j=1}^{N} a_t^{(j)}$$

The particle filter framework executes the following methods using the scheme:

```
premcloop()

for each filter period:
  for each sample:
    if first period:
      initial()
    else:
      resume()
    for each timestep in filter period:
      dynamic()
    if not last filter period:
      suspend()
    updateWeight()

postmcloop()
```

The following additional methods must be implemented to use the framework:

`suspend()` The suspend section is executed after the time step precedent to a a filter timestep and used to store the state variables. This can be achieved with the `self.report()` method.

`resume()` The resume section is executed before the first time step of a filter period and intended to re-initialise the model after a filter time step. State variables can be obtained with the `self.readmap()` method.

`updateWeight()` The updateWeight method is executed at the filter moment and used to retrieve the weight of each sample. The method must return a single floating point value (i.e. the $a_t^{(n)}$).

Like in the Monte Carlo framework each sample output will be stored into a corresponding sample directory. Each sample directory contains a `stateVar` subdirectory that is used to store the state variables of the model. State variables not holding PCRaster data types must be stored into this directory by the user.

Two different algorithms are implemented in the filter framework. Sequential Importance Resampling and Residual Resampling (see e.g. [11]) can be chosen as selection scheme by using the adequate framework class. In Sequential Importance Resampling, a cumulative distribution function is constructed from the sample weights $p_t^{(n)}$. From this distribution N samples are drawn with replacement from a uniform distribution between 0 and 1.

In Residual Resampling, in the first step samples are cloned a number of times equal to $k_t^{(n)}$=`floor`($p_t^{(n)}$N) with N number of samples and `floor` an operation rounding the value to the nearest integer. In a second step, the residual weights $r_t^{(n)}$ are calculated according to:

$$r_t^{(n)} = \frac{p_t^{(n)} N - k_t^{(n)}}{N - \sum_{n=1}^{N} k_t^{(n)}}$$

and used to construct a cumulative distribution function. From this distribution a number of additional samples is drawn until a number of N samples is reached [9].

For each filter time step a comma separated file holding sample statistics is written to the current working directory. For each sample it contains its normalised weight, the cumulative weight up to that sample and the number of clones for that sample. A zero indicates that the sample is not continued. Furthermore a graphviz input file (http://www.graphviz.org/) holding the sample choice is generated.

2.3.2.1. Example

The script in Example 2.5 shows a dynamic model which is executed within the particle filter framework. The overall runtime of the model still accounts to 180 time steps, 10 realisations are executed. As three filter moments are chosen at the timesteps 70, 100 and 150 four periods in total are executed: from timestep 1-70, 71-100, 101-150 and 151-180.

Example 2.5. Python script `particlefilter.py` specifying a model executed in the particle filter framework.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from PCRaster import *
from PCRaster.Framework import *

class SnowModel(DynamicModel, MonteCarloModel, ParticleFilterModel):
  def __init__(self):
    DynamicModel.__init__(self)
    MonteCarloModel.__init__(self)
    ParticleFilterModel.__init__(self)
    setclone("clone.map")

  def premcloop(self):
    dem = self.readmap("dem")
    self.ldd = lddcreate(dem, 1e31, 1e31, 1e31, 1e31)
    elevationMeteoStation = scalar(2058.1)
    self.elevationAboveMeteoStation = dem - elevationMeteoStation
    self.degreeDayFactor = 0.01

  def initial(self):
    self.snow = scalar(0)
    self.temperatureLapseRate = 0.005 + (mapnormal() * 0.001)
    self.report(self.temperatureLapseRate, "lapse")
    self.temperatureCorrection = self.elevationAboveMeteoStation\
         * self.temperatureLapseRate

  def dynamic(self):
    temperatureObserved = self.readDeterministic("tavgo")
    precipitationObserved = self.readDeterministic("pr")
    precipitation = max(0, precipitationObserved * (mapnormal() * 0.2 + 1.0))
    temperature = temperatureObserved - self.temperatureCorrection
    snowFall = ifthenelse(temperature < 0, precipitation, 0)
    self.snow = self.snow + snowFall
    potentialMelt = ifthenelse(temperature > 0, temperature\
         * self.degreeDayFactor, 0)
    actualMelt = min(self.snow, potentialMelt)
    self.snow = max(0, self.snow - actualMelt)
```

```
    self.report(self.snow, "s")

  def postmcloop(self):
    names = ["s"]
    mcaveragevariance(names, self.sampleNumbers(), self.timeSteps())
    percentiles = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
    mcpercentiles(names, percentiles, self.sampleNumbers(), self.timeSteps())

  def updateWeight(self):
    modelledData = self.readmap("s")
    modelledAverageMap = areaaverage(modelledData, "zones.map")
    observedAverageMap = self.readDeterministic("obsAv")
    observedStdDevMap = ifthenelse(observedAverageMap > 0, observedAverageMap\
         * 0.4, 0.01)
    sum = maptotal(((observedAverageMap - modelledAverageMap) ** 2) / (-2.0\
         * (observedStdDevMap ** 2)))
    weight = exp(sum)
    weightFloatingPoint, valid = cellvalue(weight, 1, 1)
    return weightFloatingPoint

  def suspend(self):
    self.reportState(self.temperatureLapseRate, "lapse")
    self.reportState(self.snow, "s")

  def resume(self):
    self.temperatureLapseRate = self.readState("lapse")
    self.temperatureCorrection = self.elevationAboveMeteoStation\
         * self.temperatureLapseRate
    self.snow = self.readState("s")

myModel = SnowModel()
dynamicModel = DynamicFramework(myModel, lastTimeStep=180, firstTimestep=1)
mcModel = MonteCarloFramework(dynamicModel, nrSamples=10)
pfModel = SequentialImportanceResamplingFramework(mcModel)
#pfModel = ResidualResamplingFramework(mcModel)
pfModel.setFilterTimesteps([70, 100, 150])
pfModel.run()
```

Compared to the script in Example 2.3 the three methods `suspend`, `resume` and `updateWeight` are added. The sections `initial`, `dynamic`, `premcloop` and `postmcloop` remain identical to the Monte Carlo version.

The state variables of the model are the snow height and the lapse rate. These variables are stored in the `suspend()` section with `self.reportState()` into the state variable directory. They are either cloned or replaced by the filter, or continued in the following filter period.

In the `resume()` method the lapse rate will now be set to either the same value as before the filter moment or to a new value cloned from another sample. The same procedure applies to the snow state variable. As the value of `self.temperatureCorrection` is dependent on the lapse rate it has to be re-initialised too.

To calculate the weigtht of a sample the model implements in `updateWeight` the Equation 2.1 [9]. For five meteorological stations in different elevation zones the average snow height values are compared. For the modelled data the zonal values are calculated with `areaaverage`, the observation values are read from disk. As the `observedAverageMap` contains missing values except at the measurement locations the `maptotal` operation yields the sum over the five elevation zones for the exponent of the weight calculation. The sample weight is afterwards extracted as individual floating point value.

2.3.3. Ensemble Kalman filter

The Ensemble Kalman Filter is a Monte Carlo approximation of the Kalman filter [4]. Contrary to the cloning of the particle filter the ensemble Kalman filter modifies the state variables according to:

Equation 2.2.

$x_t^{(n),+} = x_t^{(n),0} + P_t^0 H^T (H_t P_t^0 H_t^T + R_t)^{-1} (y_t^{(n)} - H_t x_t^{(n),0})$, for each sample n

where $x_t^{(n)}$ is a vector containing a realisation n at update moment t of model components for which observations are available. The superscript 0 indicates the prior state vector and superscript + indicates the posterior state vector calculated by the update. P_t^0 is the ensemble covariance matrix. $y_t^{(n)}$ is a realisation of the y_t vector holding the observations. R_t is the error covariance matrix and H_t the measurement operator ([4],[9]).

The execution scheme is similar to the one of the particle filter:

```
premcloop()

for each filter period:
  for each sample:
    if first period:
      initial()
    else:
      resume()
    for each timestep in filter period:
      dynamic()
    setState()
  setObservations()

postmcloop()
```

The user has to implement the `setState`, `setObservations` and the `resume` methods in the model class. As state variables (and eventually parameters) are modified the `setState` method now needs to return a vector (i.e. the $x_t^{(n),0}$ in Equation 2.2) holding the values instead of an individual value. In the `resume` section the updated values (i.e. the $x_t^{(n),+}$ in Equation 2.2) can be obtained with the `getStateVector` method.

For each update moment the user needs to provide the observed values y_t to the Ensemble Kalman framework with the `setObservations` method. The associated measurement error covariance matrix is set with `setObservedMatrices`. The measurement operator H_t can be set with the `setMeasurementOperator` method.

2.3.3.1. Example

The script in Example 2.6 shows again the snow model which is now executed within the Ensemble Kalman filter framework. The overall runtime of the model still accounts to 180 timesteps, 10 realisations are executed. Again three filter moments are chosen at the timesteps 70, 100 and 150.

Example 2.6. Python script `kalmanfilter.py` specifying a model executed in the ensemble Kalman filter framework.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from PCRaster import *
from PCRaster.Framework import *
from numpy import *

class SnowModel(DynamicModel, MonteCarloModel, EnKfModel):
  def __init__(self):
    DynamicModel.__init__(self)
    MonteCarloModel.__init__(self)
    EnKfModel.__init__(self)
    setclone("clone.map")
```

```
def premcloop(self):
  dem = self.readmap("dem")
  self.ldd = lddcreate(dem, 1e31, 1e31, 1e31, 1e31)
  elevationMeteoStation = scalar(2058.1)
  self.elevationAboveMeteoStation = dem - elevationMeteoStation
  self.degreeDayFactor = 0.01

def initial(self):
  self.snow = scalar(0)
  self.temperatureLapseRate = 0.005 + (mapnormal() * 0.001)
  self.report(self.temperatureLapseRate, "lapse")
  self.temperatureCorrection = self.elevationAboveMeteoStation\
       * self.temperatureLapseRate

def dynamic(self):
  temperatureObserved = self.readDeterministic("tavgo")
  precipitationObserved = self.readDeterministic("pr")
  precipitation = max(0, precipitationObserved * (mapnormal() * 0.2 + 1.0))
  temperature = temperatureObserved - self.temperatureCorrection
  snowFall = ifthenelse(temperature < 0, precipitation, 0)
  self.snow = self.snow + snowFall
  potentialMelt = ifthenelse(temperature > 0, temperature\
       * self.degreeDayFactor, 0)
  actualMelt = min(self.snow, potentialMelt)
  self.snow = max(0, self.snow - actualMelt)
  self.report(self.snow, "s")

def postmcloop(self):
  names = ["s"]
  mcaveragevariance(names, self.sampleNumbers(), self.timeSteps())
  percentiles = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
  mcpercentiles(names, percentiles, self.sampleNumbers(), self.timeSteps())

def setState(self):
  modelledData = self.readmap("s")
  modelledAverageMap = areaaverage(modelledData, "zones.map")
  self.report(modelledAverageMap, "modAv")
  values = numpy.zeros(5)
  values[0] = cellvalue(modelledAverageMap, 5, 5)[0]
  values[1] = cellvalue(modelledAverageMap, 8, 14)[0]
  values[2] = cellvalue(modelledAverageMap, 23, 24)[0]
  values[3] = cellvalue(modelledAverageMap, 28, 12)[0]
  values[4] = cellvalue(modelledAverageMap, 34, 28)[0]
  return values

def setObservations(self):
  timestep = self.currentTimeStep()
  observedData = readmap(generateNameT("obsAv", timestep))
  values = numpy.zeros(5)
  values[0] = cellvalue(observedData, 1, 1)[0]
  values[1] = cellvalue(observedData, 3, 1)[0]
  values[2] = cellvalue(observedData, 11, 1)[0]
  values[3] = cellvalue(observedData, 18, 1)[0]
  values[4] = cellvalue(observedData, 40, 4)[0]

  # creating the observation matrix (nrObservations x nrSamples)
  # here without added noise
```

```
    observations = numpy.array([values,]*self.nrSamples()).transpose()

    # creating the covariance matrix (nrObservations x nrObservations)
    # here just random values
    covariance = numpy.random.random((5, 5))

    self.setObservedMatrices(observations, covariance)

  def resume(self):
    vec = self.getStateVector(self.currentSampleNumber())
    modelledAverageMap = self.readmap("modAv")
    modvalues = numpy.zeros(5)
    modvalues[0] = cellvalue(modelledAverageMap, 1, 1)[0]
    modvalues[1] = cellvalue(modelledAverageMap, 3, 1)[0]
    modvalues[2] = cellvalue(modelledAverageMap, 11, 1)[0]
    modvalues[3] = cellvalue(modelledAverageMap, 18, 1)[0]
    modvalues[4] = cellvalue(modelledAverageMap, 40, 4)[0]
    oldSnowMap = self.readmap("s")
    self.zones = readmap("zones.map")
    newSnowCells = scalar(0)
    for i in range(1, 6):
      snowPerZone = ifthenelse(self.zones == nominal(i), oldSnowMap, scalar(0))
      snowCellsPerZone = ifthenelse(snowPerZone > scalar(0), boolean(1),\
         boolean(0))
      corVal = vec[i - 1] - modvalues[i - 1]
      newSnowCells = ifthenelse(snowCellsPerZone == 1, max(0,snowPerZone\
         + scalar(corVal)), newSnowCells)
    self.snow = newSnowCells


myModel = SnowModel()
dynamicModel = DynamicFramework(myModel, lastTimeStep=180, firstTimestep=1)
mcModel = MonteCarloFramework(dynamicModel, nrSamples=10)
ekfModel = EnsKalmanFilterFramework(mcModel)
ekfModel.setFilterTimesteps([70, 100, 150])
ekfModel.run()
```

In the `setState` of the Example 2.6 the average snow pack is calculated from the sample snow cover map. The map holding the average values is stored in the sample subdirectory in order to avoid recalculating the values in the `resume` section. The average value for each zone is extracted as an individual value and inserted into a numpy matrix. This matrix is returned to the framework.

In the `resume` section the array returned by the `getStateVector` now holds the updated state variables. In the following the correction factor for the snow values is calculated as the difference between the modelled averaged snow heights and the average snow heights returned by the Ensemble Kalman filter. For each zone this correction factor is applied to the snow pack cell values in order to obtain the new snow pack map.

Chapter 3. License

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Bibliography

[1] Arulampalam, M.S. and Maskell, S.. “A tutorial on particle filters for online nonlinear/non-Gaussian/Bayesian tracking”. 2002.

[2] Doucet, A., de Freitas, N., and Gordon, N.. “Sequential Monte Carlo Methods in Practice”. Springer. 2001.

[3] Doucet, A., Godsill, S., and Andrieu, C.. “On sequential Monte Carlo sampling methods for Bayesian filtering”. *Statistics and Computing*. 10. 197-208. 2000.

[4] Evensen, Geir. “The Ensemble Kalman Filter: theoretical formulation and practical implementation”. *Ocean Dynamics*. 53. 343-367. 2003.

[5] Evensen,G.. “Sequential data assimilation with a nonlinear quasi-geostrophic model using Monte Carlo methods to forecast error statistics”. *Journal of Geophysical Research*. 99. 10143-10162. 1994.

[6] Karssenberg, D and De Jong, K.. “Towards improved solution schemes for Monte Carlo simulation in environmental modeling languages”. NCG Nederlandse Commissie voor Geodesie, Netherlands Geodetic Commission. 2006.

[7] Karssenberg, D. and De Jong, K.. “Dynamic environmental modelling in GIS: 1. Modelling in three spatial dimensions”. *International Journal of Geographical Information Science*. 19. 559-579. 2005.

[8] Karssenberg, D. and De Jong, K.. “Dynamic environmental modelling in GIS: 2. Modelling error propagation”. *International Journal of Geographical Information Science*. 19. 623-637. 2005.

[9] Karssenberg, D., Schmitz, O., De Vries, L.M., Bierkens, M. F. P., De Jong, K., and Salamon, P.. “A software framework for construction of stochastic spatio-temporal models assimilated or calibrated with observational data.”. in prep..

[10] Simon, D.. “Optimal State Estimation: Kalman, H and Nonlinear Approaches”. Wiley-Interscience. 2006.

[11] Weerts, A. H. and El Serafy, G. Y. H.. “Particle filtering and ensemble Kalman filtering for state updating with hydrological conceptual rainfall-runoff models”. *Water Resources Research*. 42. 2006.

[12] Xiong, X., Navon, I. M., and Uzunoglu, B.. “A note on the particle filter with posterior Gaussian resampling”. *Tellus, Series A: Dynamic Meteorology and Oceanography*. 58. 456-460. 2006.

[13] Karssenberg, D., de Jong, K., and Van der Kwast, J.. “Modelling landscape dynamics with Python”. *International Journal of Geographical Information Science*. 2007.

[14] PCRaster . “PCRaster website”. *http://pcraster.geo.uu.nl/*. 2008.