Real-time Data Assimilation and Error Correction

Software Manual

SEVENTH FRAMEWORK PROGRAMME

COLOPHON

Title
Real-time Data Assimilation and Error Correction

Report number
PREPARED 2014.049

Deliverable number
D3.6.3

Author(s)
Hutton, C.J. (UNEXE), Thompson, K. (UNEXE).

Quality Assurance
By L.S.Vamvakeridou-Lyroudia (UNEXE)

Document history

Version	Team member	Status	Date update	Comments
1	Hutton, C.J.	Draft	30/12/2012	
2	Hutton, C.J.	Draft	07/02/2013	
3	Thompson, K.	Draft	07/05/2013	
4	Hutton, C.J.	Draft	07/05/2013	

This report is:
PU = Public

Contents

1 Disclaimer

The computer software presented in this manual has been developed with appropriate effort made to ensure the functions provide a faithful representation of the algorithms they were designed to execute and therefore represent. Furthermore, the successful application of methods for model calibration, uncertainty quantification and sensitivity analysis presented herein is conditional on a number assumptions that, if not satisfied, may compromise the validity of their application. The author(s), therefore, assume no responsibility or liability for any results obtained, for any use made of the results, and for any litigation or damages that result from the use of the software.

2 Introduction

2.1 Introduction to PREPARED and Work Package 3.6

The European Commission funded project, PREPARED Enabling change aims to respond to the risks posed by climate change, and show that the water supply and sanitation systems of cities and their catchments can adapt and be resilient to the challenges of climate change. PREPARED aims to build the resilience of Urban Water Systems (UWS) in two primary ways:

- First, through optimisation of existing water supply and sanitation systems, to postpone investments in new infrastructure until investment risks are lower as more knowledge is available.
- Second, in the case where optimisation is not sufficient, PREPARED will provide guidance and produce frameworks to aid utilities in building more resilient water supply and sanitation systems

Building system resilience through optimisation of water supply and sanitation requires the identification and reduction of risk associated with UWS management. Numerical system models are widely applied to inform such management decisions, however such models are inherently complex, and contain multiple sources of system uncertainty that may compromise the quality of model predictions, and subsequently derived control decisions.

An essential and innovative aspect of PREPARED is the development of a software toolbox of methods to quantify and reduce system uncertainty through offline calibration and online data assimilation, to support real time modelling (Work Package 3.6). The toolbox is required to increase the technological capacity of existing water supply and sanitation systems to deal with uncertain changes to system inputs (e.g. rainfall, dry weather flow and water demand) resulting from climatic change. Such demands call for an integrated real time control strategy, supported by monitoring and modelling approaches, to provide decision support in the face of inherent system uncertainty.

Work package 3.6 has investigated methodologies for uncertainty quantification and reduction in UWS models. Existing methods for uncertainty quantification and data assimilation have been reviewed, and their suitability to UWS models evaluated, resulting in a review paper (Hutton et al. 2012).

The subsequently developed software, alongside the software manual(s), fulfils the requirements of PREPARED Deliverable 3.6.3, and is presented in two toolboxes: First, a toolbox of methods for offline calibration, uncertainty quantification and sensitivity analysis; Second, in this document, an online toolbox for real-time data assimilation and error correction.

2.2 Software Manual Structure

The software presented in this manual consists of two principal components: In Section 3, an Artificial Neural Network (ANN) class is presented that may be trained on a different time-series to help quantify and reduce uncertainty in real-time modelling; In Section 4 an ensemble data assimilation tool, particleFilter, is presented whereby an ensemble of models may be run to quantify uncertainty in model predictions. In each section, a description of the method and an example application are presented, alongside a description of the data members and member functions of each of the tools.

3 Artificial Neural Networks: Meta-modelling and Error Correction

3.1 Introduction

Artificial Neural Networks (ANNs) are a form of data-driven modelling structure. That is, the modelling structure does not represent a specific process-based model, or representation of a given system itself. However, they may be calibrated (trained) using a series of model derived, or measured system inputs and outputs as a means to predict (or forecast) future system outputs. Because of their potential to run at much shorter computational times than complex system models, they may be used for a number of purposes to facilitate real-time modelling:

- As a system meta-model to replicate the predictions of a physically-based system model (Corzo et al. 2009; Rao and Alvarruiz 2007).
- When trained on a residual error time-series, their predictions may be added to a physically-based model's predictions to 'correct' model forecasts (Abebe and Price 2003).
- To predict uncertainty bounds derived offline through calibration, that in real-time, are otherwise computationally demanding to reproduce (Shrestha et al. 2009).

The ANN specific part of the toolbox consists of the neural network model itself, (the **`ANN`** class and a neural network specific training algorithm in the **`annTraining`** class. It is also possible to apply the Bayesian based model calibration methods presented in the accompanying calibration toolbox for ANN training. As shown in the example below, however, the **`annTraining`** class would still be required as some functions are useful for training purposes.

As there are a potential range of applications of data-driven models to quantify and/or reduce uncertainty in model predictions, a generic example of ANN training is provided here. Further guidance on training and application within the context of hydroinformatics may be found within the research literature (Bowden et al. 2012; Elshorbagy et al. 2010; Maier and Dandy 2000; Maier and Dandy 2001).

3.2 Example Application

The example presented here is a function approximation problem, where the network is trained to approximate: $y = \sin(2x)\mathrm{e}^{-\mathrm{x}}$. Figure 1 shows the setup of the problem and the initialisation of the classes for training. Lines 7 and 8 declare the arrays to store the input and output data generated by the function that is to be approximated, and Line 9 declares the function to generate the data for training. At line 16, the training data set length is specified. On line 22, within the `main()` function, the data are generated for ANN training.

On lines 25-26 and 29-30 instances of the **annTraining** class and the neural network model class **ANN** are declared; this is done by declaring a pointer to the desired interface class type and calling the corresponding factory function. The factory function creates an instance of the class in memory and returns a pointer to that class. Finally, on line 34 the network is created by specifying the number of nodes in each layer, the number of hidden layers and also, in the final two arguments, the codes for the hidden and output layer activation functions (see section 6.3.2).

```
1  #include "stdafx.h"
2
3  //include the toolbox header
4  #include "dataAssimilationToolbox.h"
5
6  //model training data generation
7      double **outputs;
8      double **inputs;
9      void generateData();
10
11     //network inputs and outputs
12     int inputNodes = 1;
13     int outputNodes = 1;
14
15     //number of training observations
16     int tObs = 100;
17
18
19 int _tmain(int argc, _TCHAR* argv[])
20 {
21     //generate data for ANN training
22     generateData();
23
24     //initialise an object of the neural network training class and get a pointer
25     IannTraining* train;
26     train = getAnnTraining();
27
28     //initialise an object of the neural network class and get a pointer
29     IANN* ann;
30     ann = getAnn();
31
32
33     //create the network structure and activation functions.
34     ann->createNetwork(1,7,1,1,1,3);
```

Figure 1. Code snippet showing example problem of neural network training.

In Figure 2 on line 38 the **train** object is initialised by specifying the number of input nodes, output nodes, data training length, and passing the inputs and outputs of the function to be approximated – e.g. the data used to conduct training. On line 41 the function **convertData()** is called, which converts the input data to fall within the range specified by the last two arguments in the function call. Such a conversion is conducted to help train the neural network by ensuring the input data range falls on that of the activation function. On line 44 the parameters that control the training performance are set, prior to execution of the function **trainNetwork()** on line 48, which trains the neural network passed as an argument (by reference), setting the weights of the network to the optimal set found during training. Finally, on lines 52 and 55 the optimal neural network is written to file, followed by the predictions of the neural network, to provide a visual performance of the training algorithm.

```
    //initialise the neural network data for training.
    train->initialiseData(inputNodes, outputNodes, tObs, inputs, outputs);

    //convert the input data as necessary for use in the ANN.
    train->convertData(*ann,-2,2);

    //set the parameters required for neural network training.
    train->setParameters(0.2,0.9,1000,700,2);

    //trains the neural network passed to the function,
    //sets the weights of thatr ann as the optimal weights.
    train->trainNetwork(*ann);


    //output the weights and data conversion factors to file.
    ann->writeWeightFile("weightfile.txt");

    //output predictions to file.
    train->writePredictions(*ann, "predictions.txt");

    return 0;
}
```

Figure 2. Second code snippet showing an example problem of neural network training

3.3 ANN class description

An instance of the **ANN** class is a self-contained neural network, which can be trained using the **annTraining class**, or via another means, such as the accompanying calibration toolbox. Functions are contained for initiating and running the network.

3.3.1 Data Members

```
//number of neurons in each layer, and number of hidden layers.
```
`int nInput:` number of input nodes in the input layer of the neural network

`int nHidden:` number of hidden nodes in a given hidden layer of the neural network

`int nLayers:` number of layers of hidden nodes in the neural network.

`int nOutput:` number of output nodes in the output later of the network.

```
//store for neurons.
```
`double* inputNeurons:` store for input neurons

`double** hiddenNeurons:` store for hidden neurons

`double* outputNeurons:` store for output neurons

`double** hiddenErrorGradients:` store for hidden layer error gradients

`double* outputErrorGradients:` store for output layer error gradients

`double** wInputHidden:` store for weights between the input and first hidden layer

`double** wHiddenOutput:` store for the weights between the final hidden layer and output neuron layer.

`double*** wHiddenHidden:` store for weights between hidden layers

`double** dInputHidden:` store for change in network weights between input and hidden layer

`double** dHiddenOutput:` store for change in network weights between hidden and output layer

`double*** dHiddenHidden:` store for change in network weights between hidden layers

`double** doldInputHidden:` store for change in network weights from the last iteration between input layer and the first hidden layer

`double** doldHiddenOutput:` store for change in network weights from the last iteration between hidden layer and the output layer

`double*** doldHiddenHidden:` store for change in network weights from the last iteration between hidden layers

`int actuatorHidden: index specifying the actuator for the hidden neurons:`

1 = sigmoid function
2 = tanh function
3 = linear function
4 = binary function

`int actuatorOutput: index specifying the actuator for the output neurons:`

1 = sigmoid function
2 = tanh function
3 = linear function
4 = binary function

Stores for the conversion factors of the input variables by z-score transformation:

`double * mInput [nInput]:` mean of the input variable

`double * sdInput [nInput]:` standard deviation of the input variable

`double * mulInput [nInput]:` store for the multiplier of the input variable

`double * mOutput [nOutput]`: mean of the output variable

`double * sdOutput [nOutput]`: standard deviation of the output variable

`double * mulOutput [nOutput]`: store for the multiplier of the output variable

3.3.2 *Member Functions*

createNetwork

```
void createNetwork(int NINPUT, int NHIDDEN, int NLAYERS, int NOUTPUT, int
ACTUATORHIDDEN, int ACTUATOROUTPUT);
```

Description: initialises the **ANN** class data members and structure

Arguments:

`int NINPUT`: input to `nInput`

`int NHIDDEN`: input to `nHidden`

`int NLAYERS`: input to `nLayers`

`int NOUTPUT`: input to `nOutput`

`int ACTUATORHIDDEN`: input to `actuatorHidden`

`int ACTUATOROUTPUT`: input to `actuatorOutput`

feedForwardTrain

```
void feedForwardTrain(double *inputs);
```

Description: runs the neural network by feeding the specified inputs through the network. Used by **annTraining** as the `double *inputs` need to be in converted form.

Arguments:

`double *inputs[nInput]`: vector of inputs to the neural network.

actiationFunction

```
double activationFunction(int code, double value);
```

Description: returns the result of passing `double value` through the specified activation function

Arguments:

`int code:` Code signifying the activation function required:

1 = sigmoid function
2 = tanh function
3 = linear function
4 = binary function

`double value:` value passed to the activation function.

getOutputGradient

```
double getOutputGradient(int code, double observed, double predicted);
```

Description: returns the gradient of the output activation function

Arguments:

`int code:` Code signifying the activation function required:

1 = sigmoid function
2 = tanh function
3 = linear function
4 = binary function

`double observed:` observed value of the output node used to determine the residual error

`double predicted:` predicted value at the output node

getHiddenGradient

```
double getHiddenGradient(int code, double weightsum, double neuronValue);
```

Description: returns the gradient of the hidden activation function

Arguments:

`int code:` Code signifying the activation function required:

1 = sigmoid function
2 = tanh function
3 = linear function
4 = binary function

`double weightsum:` sum of the products of weights combined with output gradients linked to the hidden node in question.

`double neuronValue:` value of the hidden neuron.

writeWeightFile

```
void writeWeightFile(std::string filename);
```

Description: writes a file specifying the neural network architecture that can later be called for further application.

Arguments:

`std::string filename:` text (.txt) file(including full extension) where the weight file is stored

callWeightFile

```
void callWeightFile(std::string filename);
```

Description: calls a file specifying the neural network architecture that was written in the format of the output file written by **`writeWeightFile().`** Note: also runs **`createNetwork()`** so that the neural network is set up for application

Arguments:

`std::string filename:` text (.txt) file(including full extension) where the weight file is stored

feedForwardRun

```
void feedForwardRun(double *inputs, double *outputs);
```

Description: runs the neural network to produce predictions at the output node.

Arguments:

`double *inputs [nInput]:` pre-converted vector of inputs corresponding to the input neurons.

`double *outputs[nOutputs]:` vector where the output neuron predictions are returned to the user. Note: these are re-converted after running through the network back to the scale of the input variables.

batchRun

```
void batchRun(double **inputs, double **outputs, int length);
```

Description: repetitively calls **`feedForwardRun()`** `int length` times to derive a series of outputs from the neural network.

Arguments:

`double **inputs [nInput][length]: array storing the inputs to the neural network`

`double **outputs [nOutput][length]: array storing the outputs from the neural network.`

`int length:` number of times the neural network is run – e.g. the number of input vectors for which output vectors are required

3.4 annTraining class description

This class provides a calibration, or training algorithm, for a neural network when an instance of **`ANN`** is passed to an instance of **`annTraining`**. The training algorithm is a sequential back-propagation algorithm, that may be initialised multiple times from initial starting points to improve the probability of identifying a global optimum in weight space.

The user may specify the learning rate, momentum, number of initial starting points, and number of loops through the training dataset for each starting point.

3.4.1 Data Members

`int tInputs:` total number of input neurons, and therefore input data vectors used for training

`int tOutputs:` total number of output neurons, and therefore output data vectors used for training

`int tLength:` length of the training data time-series

`double ** inputs [tInputs][tLength]:` input array used to store in the training data

`double ** outputs [tInputs][tLength]:` output array used to store in the training data

`double **rInputs [tInputs][tLength]:` array to store the input data in a randomised order for more efficient training

`double **rOutputs [tInputs][tLength]:` array to store the output data in a randomised order for more efficient training

Stores for the conversion factors of the input variables by z-score transformation:

`double * meanInput [tInputs]:` mean of the input variable

`double * stdevaInput [tInputs]:` standard deviation of the input variable

`double * multiInput [tInputs]:` store for the multiplier of the input variable

`double * meanOutput [tOutputs]:` mean of the output variable

`double * stdevaOutput [tOutputs]:` standard deviation of the output variable

`double * multiOutput [tOutputs]:` store for the multiplier of the output variable

`double *optWeights [total weights]:` store for the optimal network weights found during training

`double ** optPredictions [tInputs][tLength]:` store for the optimal predictions of the optimal neural network

`int itt:` total number of random starting points in weight space from which to run the sequential back propagation algorithm

`int loops:` total number of loops through the training data set for each random starting point in weight space

`double learningRate:` the learning rate of the back propagation algorithm

`double momentum:` the momentum used in the back propagation algorithm

`double wRange:` the maximum difference from zero (in both positive and negative directions) over which the weights are initialised. E.g. if `wRange = 2`. Range of weight initialisation is [-2, 2].

3.4.2 Member functions

initialseData

```
void initialiseData(int TINPUTS, int TOUTPUTS, int TLENGTH, double
**INPUTS, double **OUTPUTS);
```

Description: initialises the data for the **`annTraining`** `class.`

Arguments:

`int TINPUTS:` input to `int tInputs.`

`int TOUTPUTS:` input to `int tOutputs.`

`int TLENGTH:` input to `int tLength.`

`double **INPUTS:` input to `double ** inputs.`

`double **OUTPUTS:` input to `double ** inputs.`

convertData

```
void convertData(ANN &nn, double MIN, double MAX);
```

Description: Converts the input and output data within the range specified in the function call suitable for use in the neural network. The range should be selected in consideration of the activation function and ranger over which network weights are initialised.

Arguments:

`ANN &nn: reference to an` object of the **`ANN`** class. The function **`createNetwork()`** within the **`ANN`** class must be called prior to **`convertData().`**

`double MIN:` lower bound of the range over which the data are converted.

`double MAX:` upper bound of the range over which the data are converted.

setParameters

```
void setParameters(double LEARNINGRATE, double MOMENTUM, int ITT, int
LOOPS, double WRANGE);
```

Description: sets the main parameters of the training algorithm

Arguments:

`double LEARNINGRATE: input to double learningRate.`

`double MOMENTUM: input to double momentum`

`int ITT: input to int itt`

`int LOOPS: int loops`

`double WRANGE: input to double wRange`

trainNetwork

```
void trainNetwork(ANN &nn);
```

Description: trains the network based on the parameter settings specified previously in **`setParameters()`**

Arguments:

`ANN &nn: reference to an` object of the **ANN** class that is to be trained. One the function finishes, the network weights are set to the optimal set found during training.

initialiseWeights

```
void initialiseWeights(ANN &nn);
```

Description: called by **trainNetwork(),** the function initialises the weights within the range specified in **setParameters()** for each initial starting point in weight space (`int itt`).

Arguments:

`ANN &nn: reference to an` object of the **ANN** class that is to be trained. One the function finishes, the network weights are set to the optimal set found during training.

backPropTrain

```
double backPropTrain(ANN &nn, int onoff);
```

Description: runs through the vector of training data once, training the neural network sequentially.

Arguments:

`ANN &nn: reference to an` object of the **ANN** class that is to be trained. One the function finishes, the network weights are set to the optimal set found during training.

`int onoff:` If set to one, the network weights are changed through backpropagation after each sample of inputs and outputs. If set to zero, no training occurs, and the predictions (outputs) are stored in `double ** optPredictions;`

backPropagate

```
void backPropagate(ANN &nn, double *output);
```

Description: Back-propagates the output errors through the network.

Arguments:

`ANN &nn: reference to an` object of the **ANN** class that is to be trained. One the function finishes, the network weights are set to the optimal set found during training.

`double *output [tOutputs]:` vector of observations for each output node.

backPropagate

```
void updateWeights(ANN &nn)
```

Description: updates the weights of the network following **backPropagate().**

Arguments:

`ANN &nn: reference to an` object of the **ANN** class that is to be trained. One the function finishes, the network weights are set to the optimal set found during training.

passConversions

```
void passConversions(ANN &nn)
```

Description: passes the input and output data conversions to the specified object of the **ANN** class such that the trained network can be used for further application.

Arguments:

`ANN &nn: reference to an` object of the **ANN** class that is to be trained. One the function finishes, the network weights are set to the optimal set found during training.

storeWeights

```
void storeWeights(ANN &nn)
```

Description: stores the weights of the optimal model run in `double *optWeights` for later re-use

Arguments:

`ANN &nn: reference to an` object of the **`ANN`** class that is to be trained. One the function finishes, the network weights are set to the optimal set found during training.

setWeights

```
void setWeights(ANN &nn)
```

Description: sets the weights of the passed in **`ANN.h`** class to the optimal weights stored in `double *optWeights` following calibration/optimisation

Arguments:

`ANN &nn: reference to an` object of the **`ANN`** class that is to be trained. One the function finishes, the network weights are set to the optimal set found during training.

writePredictions

```
void writePredictions(ANN &nn, std::string filename)
```

Description: writes to file the optimal predictions, and observations for each sample used in calibration

Arguments:

`ANN &nn: reference to an` object of the **`ANN`** class that is to be trained. One the function finishes, the network weights are set to the optimal set found during training.

`std::string filename:` the filename (.txt) where the results will be written, along with the full extension

4 Ensemble Data Assimilation

4.1 Introduction

Real-world modelling problems often contain considerable uncertainty in, for example the system driving conditions, and the model structures themselves. As a consequence, methods have been developed that run an ensemble of models in real-time to propagate and account for different forms of uncertainty (Hutton et al. 2012a). For example, a rainfall-runoff system model can be run repeatedly, each run driven by a different input rainfall sampled from a pre-specified distribution that represents rainfall measurement uncertainty. Once propagated through the model, some measure of spread of the model predictions at a given point in the system (e.g. prediction range or distribution) can be used to quantify uncertainty in the model predictions/forecasts. As real-time observations become available this information may be used to judge which of the models performs best – i.e. which produce predictions closest to the observations. The observations (data) may then be assimilated to correct the model states and/or parameters in the ensemble, prior to re-propagation of the models in time to make subsequent forecasts. The aim is that repeated prediction and correction (assimilation) will keep the models on track to provide: first, good state estimates of the system at the time of the most recent observation; and second, by correcting the model states at the time of the observation, provide good initial conditions from which to make a system forecast.

The toolbox presented here, **`particleFilter`** provides functions for running an ensemble of models in real-time to quantify uncertainty in model predictions, and provide means of correcting both the states and parameters in the ensemble. Unlike offline calibration runs, data assimilation, by its nature, involves tight integration with the given system model in question, as system states and parameters have to be obtained, modified and reset within the model during simulation. Thus, some user defined **`set()`** and **`get()`** functions are required to use the toolbox, an example of which is detailed below. If code access is restricted, or if these functions do not exist within the model, then using most data assimilation methods will be limited.

4.2 Particle Fillter

The **`particleFilter`** toolbox contains methods for running an ensemble of models based on techniques developed under the general framework of Particle Filtering (See the following papers for further information on the theoretical and practical aspects of the methodology: Hutton et al. 2012a; Hutton et al. 2012b; van Leeuwen 2009). In brief, the general methodology is presented in Figure 3, and works as follows. The uncertainty in model states and parameters, $\mathbf{Y}$, is represented probabilistically. However, the probability of model states and parameters is represented by a discrete set of models, or particles (i), each with an associated weight w. When the ensemble of models, n_p, is propagated forwards to the observation time, t , the conditional probability of each particle's states and parameters

given the observations, $P(\mathbf{Y}_t|\mathbf{X}_t^i)$ is calculated. The probabilities for all particles are normalised to derive a weight for each particle:

$$w_t^i = \frac{P(\mathbf{Y}_t|\mathbf{X}_t^i)}{\sum_{i=1}^{n_p} P(\mathbf{Y}_t|\mathbf{X}_t^i)} \quad (1)$$

Once the particles have been weighted, in general it is required that some form of resampling is required, where the states and parameters of the particles are modified based on their relative weight – or effectively moved to better performing areas of state and/or parameter space. A wide range of methods are available to determine how to 'move' the particles in state and parameter space, as reviewed by van Leeurwen (2009). The toolbox provides one of the most popularly applied, Stochastic Universal Resampling, where particles are copied proportional to their weight. Finally, to maintain diversity in the ensemble of states and/or parameters, a kernel smoothing procedure is also included where particle states and parameters can be perturbed, following copying, to maintain diversity in the sample.

As shown in Figure 3, the methodology presented provides the user with the ability to update the model parameters after a given number of observations (sampFreq), as in many models, the parameter may not be expected to vary as frequently as the observations. In this case, the weight of a particle at the point of parameter update is the cumulative normalised weight determined across all of the previous observation points since the last update.

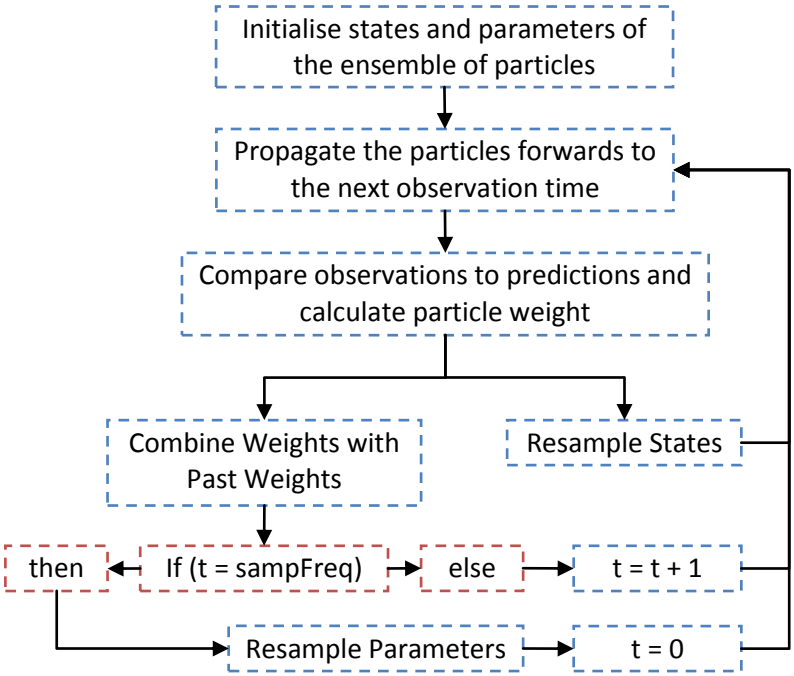

Figure 3. Flow diagram indicating the general steps in particle filtering.

4.3 Example Application

The Particle Filter and its integration with a model is illustrated with a simple linear reservoir state estimation problem. The outflow, **Q** from a linear reservoir is calculated by dividing the volume, **V** of the linear reservoir by a time constant **T**. The mass balance of the linear reservoir is then updated by adding the inflow **I**, and taking away **Q**, the flow leaving the reservoir. The linear reservoir is run, driven by a time series of inflows, to derive a vector of outflows **Q**. These are perturbed with Gaussian noise to derive a vector of outflow observations for use in the assimilation procedure.

An ensemble of models is then run, and the observations assimilated, to estimate the state of the system, **V** and the value of the time constant, **T**. The implementation of the particle filter for this joint state and parameter estimation problem is shown in Figure 4 to Figure 6.

```
1  #include "stdafx.h"
2
3  //include header
4  #include "dataAssimilationToolbox.h"
5
6
7  //prior ranges for the parameter value estimates
8  double *parmin; double *parmax;
9
10 //prior ranges from which the initial state conditions are initialised
11 double *statemin; double *statemax;
12
13 //function to generate the data used in assimilation from the model
14 void generateData(double noise);
15
16 //these are functions to be defined by the user
17 void setStatesParameters(int particle);
18 void getStates(int particle); //function to get the new states from the model
19 void runModel(double inputs, double outputs); //function to run the model
20
21 //Parameters
22 int tStates = 1;
23 int tParameters = 1;
24 int particles = 200;
25 int tPredictions = 1;
26
27 //number of total itterations.
28 int tObs = 10000;
```

Figure 4. Code snippet showing the application of the particle filter.

When initialising the particle filter prior values for the model states and parameters must be specified for each of the particles. These are sampled from uniform prior distributions, the ranges of which are stored in the arrays on lines 8 and 11, and used as input to a number of functions. The function `generateData()` on line 14 generates the data from the reservoir model for use in the assimilation problem. Three user defined functions are then specified on lines 17-19, which are responsible for exchanging the model states and parameters between the model and each of the particles. On lines 22 to 25, key parameters are specified: the number of states in the model, the number of parameters, the number of particles used in the ensemble, and the number of predictions used weight each particle. Finally, on line 28 the total observations used in this example is specified.

```
53 int _tmain(int argc, _TCHAR* argv[])
54 {
55
56     //Create the particle filter class and get a pointer
57     IparticleFilter* pf;
58     pf = getParticleFilter();
59
60     //generate data for the particlwe filter application
61     generateData(0.04);
62
63     //create storage for the model results.
64     createStorage();
65
66     //initialise the particle filter object
67     pf->initialise(particles, tStates, tParameters, tPredictions);
68
69     //initialise the parameters of the particle filter
70     pf->initialiseParameters(parmin, parmax);
71     pf->initialiseStates(statemin,statemax);
72
73     //set the frequency of sampling for the model parameters.
74     pf->setSampFreq(200);
75     pf->setParPert(0.05);
76
77     //run through the observations (e.g run through the timesteps).
78     int count = 1;
79     double *observations; observations = new double[tPredictions];
80
81     do {
82
83         //loop through the particles, propagating them to the next timestep
84         for (int i=0; i < pf->getTParticles(); i++){
85
86             setStatesParameters(i); //set the states and parameters in the model
```

Figure 5. Code snippet showing the application of the particle filter.

In Figure 5 the start of the main() function is shown. On line 57 the particleFilter object is created; this is done by declaring a pointer to the desired interface class type and calling the corresponding factory function. The factory function creates an instance of the class in memory and returns a pointer to that class. On line 61 the function is called which generates the data from the model used in the assimilation problem. The observations are perturbed with a Gaussian noise – the standard deviation of which is specified as a function argument. On line 64 a function is called which creates storage of the results of the assimilation procedure across the observation time-series. This function is written in addition to the Particle Filter to show the results of the example, and is not essential to particle filter performance. On line 67 the **`particleFilter`** object is initialised by passing the variables specified on lines 22-25 from figure 4. On lines 70 and 71 the initial conditions for the model parameters and states are sampled from the uniform prior distributions specified as function arguments. These ranges are specified in the function **`generateData()`**. On line 74 the number of observations between parameter re-sampling is specified, and on line 75 the minimum parameter perturbation is also declared. Line 79 creates storage for the observations at each time-step, which are later used to calculate particle weights. On Line 81 the main loop running through the model simulation is started. For each observation, first all particles need to be propagated forwards to the next observation time. In the example, the states and parameters for each particle are passed from **`pf`**, the particle filter object, to the model using the user defined function **`setStatesParameters()`**. In Figure 6, the model is run using the set states and parameters, the input passed as the first function argument, and the result stored in the array passed as the second argument, on line 89. Finally, within the

particle loop the states of the model are retrieved and reset within the **pf** object using the function **getstates()**.

```
88              runModel(inputs[count],pf->getPredictions(0,1)); //run the model forwards
89
90              getStates(i); //get the new states at the next timestep
91
92          }
93
94          //assign the system observations from the store.
95          observations[0] = outputs[count];
96
97          //calculate the weights for the current timestep
98          pf->calculateWeights(observations, 0.04);
99
100         //update the cumulative particle weights.
101         pf->updateCumWeight();
102
103         //calculate statistics of the prediction before resampling
104         pf->calculateStatistics();
105
106         //determine particle copies
107         pf->stochUniResampling(pf->getWeightPtr(),pf->getCopiesPtr());
108
109         //copy states using the copies identified in stochastic universal resampling.
110         pf->copyStates();
111
112         //perturb states with random noise
113         pf->perturbStates(statemin, statemax);
114
115         //updates the parameters once pf.sampCount is equal to pf.sampFreq
116         pf->parameterUpdate(parmin, parmax);
117
118         setStorage(count);
119
120         count += 1;
121
122     }while(count < tObs);
123
124
125     //output results to file
126     outputResults("testfile.txt");
127
128     //release memory
129     pf->destroy();
130
131     return 0;
132 }
```

Figure 6. Code snipped showing particle filter application.

Once all the particles have been propagated forwards in time to the latest observation point, posterior weighting and resampling occurs. First, the observation for the latest time-step is obtained from the storage array on line 95, and passed to the function **calculateWeights()** on Line 98, along with the standard deviation used in the Gaussian error model to calculate particle weights. Once the weights for the current time-step are calculated, the cumulative weights for resampling the model parameters are updated, and the summary statistics of the ensemble distribution calculated on line 104. On line 107 the number of particle copies is calculated for each particle based on the current particle weights. The states are then copied on line 110, and on line 113 the states are perturbed with random noise. If **parPert** on line

75 (fig. 5) is not set equal to zero, then that fraction of that state range defined by the arguments passed to the function on line 113, is used as a minimum standard deviation from which to sample the perturbation. This prevents the particle range from collapsing to a single value, with a view towards maintaining diversity in the sample. The final particle filter function is called on line 116, which repeats the particle resampling, copying, and perturbation for the particle parameters if the sampling frequency has been reached. Finally, on line 118 and 126 the ensemble results are stored for each time-step, and a text file of all of the results output to a user specified location to evaluate the case study performance. As always, the object must be released using the destroy() function when it is no longer in use.

4.4 ParticleFilter class

4.4.1 Data Members

`int tParticles:` total number of particles (models) – ensemble members.

`int tStates:` total number of states in each particle.

`int tParameters:` total number of parameters in each particle.

`int tPredictions:` total number of predictions in each particle – e.g. predictions associated with the available observations.

`int sampFreq:` observation frequency at which the parameters are updated.

`int sampCount:` count used to determine when to update the parameters.

`double delta:` factor controlling the spread of the kernel smoothing function. Default = 0.8.

`double parPert:` specifies a minimum standard deviation when perturbing the model parameters following resampling. If `ParPert` is equal to zero, then the standard deviation of the parameter value across the ensemble of particles will be used. If set between zero and one, then that fraction of the range specified in the function call `perturbParameters()` will be used as a minimum perturbation.

`double statePert;` specifies a minimum standard deviation when perturbing the model states following resampling. If `statePert` is equal to zero, then the standard deviation of the state value across the ensemble of particles will be used. If set between zero and one, then that fraction of the range specified in the function call `perturbParameters()` will be used as a minimum perturbation.

`double **states [tParticles][tStates]:` array storing the states of the particles.

`double **parameters [tParticles][tParameters]:` array storing the parameters of the particles.

`double **predictions [tParticles][tPredictions]:` array storing the predictions of the particles.

`double *weight [tParticles]:` array storing the weights (probabilities) of the particles

`double *copies [tParticles]:` array storing the number of particle copies of each particle in the ensemble.

`double *cumWeight [tParticles]:` array storing the cumulative particle weight for use when re-sampling parameters.

`double *parMean [tParameters]:` store for mean parameter values of the ensemble.

`double *stateMean [tStates]:` store for the mean state values of the ensemble.

`double *parRanMin [tParameters]:` store for the minimum parameter value in the ensemble.

`double *parRanMax [tParameters]:` store for the maximum parameter value in the ensemble.

`double *parRan95L [tParameters]:` store for the lower 95% intervals of each parameter in the ensemble.

`double *parRan95U:` store for the upper 95% intervals of each parameter in the ensemble.

`double *stateRanMin:` store for the minimum state values in the ensemble.

`double *stateRanMax:` store for the maximum state values in the ensemble.

`double *stateRan95L:` store for the lower 95% intervals of each state in the ensemble.

`double *stateRan95U:` store for the upper 95% intervals of each state in the ensemble.

`genericFunctions gen:` object of generic functions class.

4.4.2 *Member Functions*

initialise

```
void initialise(int TPARTICLES, int TSTATES, int TPARAMETERS, int
TPREDICTIONS);
```

Description: initialises the particle filter object.

Arguments:

`int TPARTICLES: input to tParticles`

`int TSTATES: input to tStates`

`int TPARAMETERS: input to tParameters`

`int TPREDICTIONS: input to tPredictions`

initialiseStates

```
void initialiseStates(double *stateMin, double *stateMax);
```

Description: initialises the states of each particle, sampling from the uniform ranges specified.

Arguments:

`double *stateMin [tStates]:` minimum value for the uniform range.

`double *stateMax [tStates]:` maximum value for the uniform range.

initialiseParameters

```
void initialiseParameters(double *parMin, double *parMax);
```

Description: initialises the parameters of each particle, sampling from the uniform ranges specified.

Arguments:

`double *parMin [tParameters]:` minimum value for the uniform range.

`double *parMax [tParameters]:` maximum value for the uniform range.

stochUniResampling

```
void stochUniResampling(double *weight, double *copies);
```

Description: assigns a number copies to each particle proportional to particle weight, using the Stochastic Universal Re-sampling method.

Arguments:

`double *weight [tParticles]:` vector of particle weights.

`double *copies [tParticles]:` vector of particle copies populated by the function.

copyStates

```
void copyStates();
```

Description: copies a particle's states using the vector `double *copies;`

Arguments: None

copyParameters

```
void copyParameters();
```

Description: copies a particle's parameters using the vector `double *copies;`

Arguments: None

perturbStates

```
void perturbStates(double *stateMin, double *stateMax);
```

Description: perturbs the states of the particles using the kernel smoothing approach, ensuring that the perturbed states stay within the range specified in the function arguments.

Arguments:

`double *stateMin [tStates]:` minimum value for the uniform range.

`double *stateMax [tStates]:` maximum value for the uniform range.

perturbParameters

```
void perturbParameters(double *parMin, double *parMax);
```

Description: perturbs the parameters of the particles using the kernel smoothing approach, ensuring that the perturbed parameters stay within the range specified in the function arguments.

Arguments:

`double *parMin [tParameters]:` minimum value for the uniform range.

`double *parMax [tParameters]:` maximum value for the uniform range.

calculateWeights

```
void calculateWeights(double *observations, double std);
```

Description: calculates the particles' weights using a Gaussian likelihood function.

Arguments:

`double *observations [tPredictions]:` vector of observations of the same length as the predictions vector

`double std:` Standard Deviation assumed in the Gaussian error model.

updateCumWeight

```
void updateCumWeight();
```

Description: updates the cumulative weight of the particles across observation time steps.

Arguments: none.

resetCumWeight

```
void resetCumWeight();
```

Description: resets the cumulative weights of the particles

Arguments: none.

parameterUpdate

```
void parameterUpdate(double *parMin, double *parMax);
```

Description: updates the parameters when `sampFreq` equals `sampCount`, using stochastic universal resampling, and perturbing the parameters using the function `perturbParameters()`.

Arguments:

`double *parMin [tParameters]:` minimum value for the uniform range, used in function `perturbParameters()`.

`double *parMax [tParameters]`: maximum value for the uniform range, used in function `perturbParameters()`.

calculateStatistics

```
void calculateStatistics();
```

Description: Calculates summary statistics from the ensemble: ensemble mean, range and 95% uncertainty intervals for all states and parameters.

Arguments: none.

Set[Data member]

```
Void setSampFreq(int freq);              //sets sampFreq
void setParPert(double parPert);         //sets parPert
void setStatePert(double statePert);     //sets staePert
```

Description: A family of functions to set data members within the particleFilter object.

Arguments:

None.

get[Data member]

```
int getTParticles();                     //returns tParticles
double getPredictions(int i, int j);     //returns prediction at indices
double* getWeightPtr();                  //returns *weight
double* getCopiesPtr();                  //returns *copies
```

Description: A family of functions to return data members or in some cases pointers to data members of this object to the function caller.

Arguments:

None.

5 Software Installation

The software toolkit consists of a precompiled dynamic library (.dll) containing the functionality for the data assimilation classes. These data objects will be called upon from within the user's C++ environment when implementing a solution.

An import library (.lib) and header file (.h) are provided for ease of use. These must be made available to the C++ compiler. The header file contains purely abstract classes and virtual functions which refer to machine code within the precompiled library (.dll). An example application using the calibration toolkit from a user perspective in Visual Studio 2010 is provided in subsection 5.1.

5.1 Example installation using the import library

Create a new console application with default settings and copy the import library "ASSIMILATION_TOOLBOX_DLL_GEN.lib" and the header file "AssimilationToolbox.h" to the working directory. Add the import library to the "Additional Dependencies" field under Configuration Properties->Linker->Input in the project properties menu. Add the header file to the project by right clicking on header files and selecting Add->Existing Item from the menu.

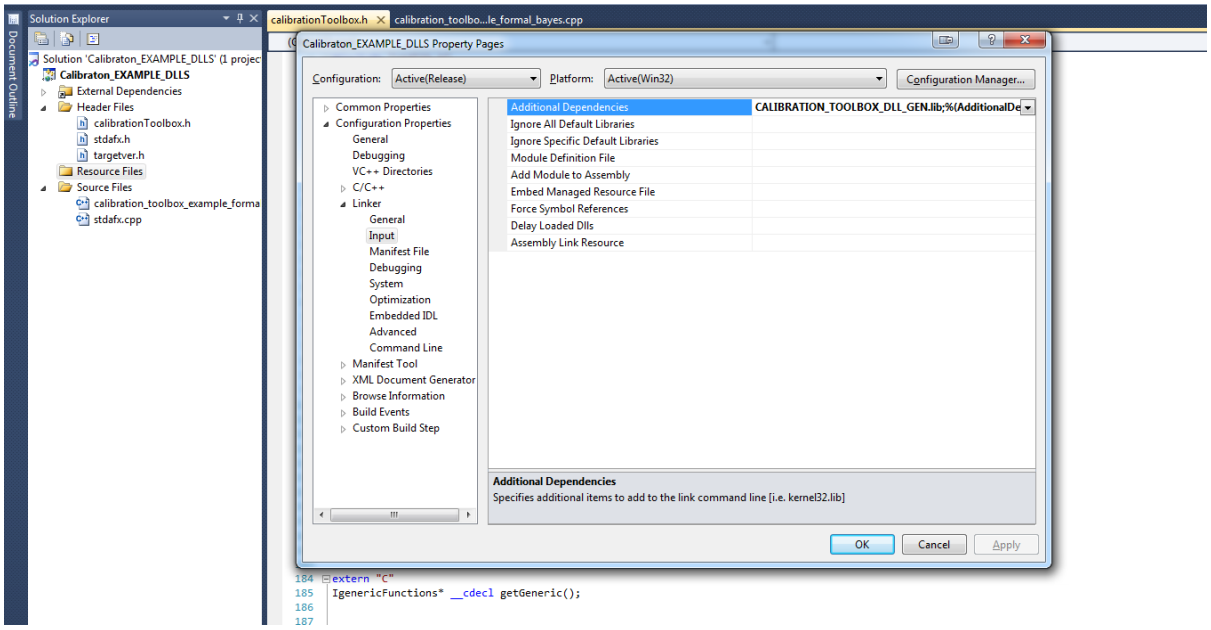

Figure 9. Linker Properies

Copy the dynamic library "ASSIMILATION_TOOLBOX_DLL_GEN.dll" to the debug and release directories that were created automatically by the project wizard. This file must always be present in the same directory as the executable file at runtime.

6 References

Abebe, A. J., and Price, R. K. (2003). "Managing uncertainty in hydrological models using complementary models." Hydrological Sciences Journal-Journal Des Sciences Hydrologiques, 48(5), 679-692.

Bowden, G. J., Maier, H. R., and Dandy, G. C. (2012). "Real-time deployment of artificial neural network forecasting models: Understanding the range of applicability." Water Resources Research, 48.

Corzo, G. A., Solomatine, D. P., Hidayat, de Wit, M., Werner, M., Uhlenbrook, S., and Price, R. K. (2009). "Combining semi-distributed process-based and data-driven models in flow simulation: a case study of the Meuse river basin." Hydrology and Earth System Sciences, 13(9), 1619-1634.

Elshorbagy, A., Corzo, G., Srinivasulu, S., and Solomatine, D. P. (2010). "Experimental investigation of the predictive capabilities of data driven modeling techniques in hydrology - Part 1: Concepts and methodology." Hydrology and Earth System Sciences, 14(10), 1931-1941.

Hutton, C. J., Kapelan, Z., Vamakeridou-Lyroudia, L. S., and Savic, D. (2012a). "Dealing With Uncertainty in Water Distribution Systems' Models: A Framework for Real-time Modelling and Data Assimilation." Journal of Water Resources Planning and Management-Asce.

Hutton, C. J., Vamvakeridou-Lyroudia, L. S., Kapelan, Z., and Savic, A. D. (2012b). "Quantifying and Reducing Urban Water System Model Prediction Uncertainty Through Sequential Monte Carlo Sampling." 10th International Conference on Hydroinformatics, HIC 2012, Hamburg, Germany.

Maier, H. R., and Dandy, G. C. (2000). "Neural networks for the prediction and forecasting of water resources variables: a review of modelling issues and applications." Environmental Modelling & Software, 15(1), 101-124.

Maier, H. R., and Dandy, G. C. (2001). "Neural network based modelling of environmental variables: A systematic approach." Mathematical and Computer Modelling, 33(6-7), 669-682.

Rao, Z. F., and Alvarruiz, F. (2007). "Use of an artificial neural network to capture the domain knowledge of a conventional hydraulic simulation model." Journal of Hydroinformatics, 9(1), 15-24.

Shrestha, D. L., Kayastha, N., and Solomatine, D. P. (2009). "A novel approach to parameter uncertainty analysis of hydrological models using neural networks." Hydrology and Earth System Sciences, 13(7), 1235-1248.

van Leeuwen, P. J. (2009). "Particle Filtering in Geophysical Systems." Monthly Weather Review, 137(12), 4089-4114.

Hutton, C. J., Kapelan, Z., Vamakeridou-Lyroudia, L. S., and Savic, D. (2012). "Dealing With Uncertainty in Water Distribution Systems' Models: A Framework for Real-time Modelling and Data Assimilation." *Journal of Water Resources Planning and Management-Asce*.

van Leeuwen, P. J. (2009). "Particle Filtering in Geophysical Systems." *Monthly Weather Review*, 137(12), 4089-4114.