Calibration, Uncertainty Quantification and Sensitivity Analysis

Software Manual
COLOPHON

Title
Calibration, Uncertainty Quantification and Sensitivity Analysis: Software Manual

Report number
PREPARED 2014.048

Deliverable number
D3.6.3

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

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

Document history

<table>
<thead>
<tr>
<th>Version</th>
<th>Team member</th>
<th>Status</th>
<th>Date update</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hutton, C.J.</td>
<td>Draft</td>
<td>30/12/2012</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Hutton, C.J.</td>
<td>Draft</td>
<td>07/02/2013</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Thompson, K.</td>
<td>Draft</td>
<td>07/5/2013</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Hutton, C.J.</td>
<td>Draft</td>
<td>07/5/2013</td>
<td></td>
</tr>
</tbody>
</table>

This report is:
PU = Public
Contents

1 Disclaimer 6

2 Introduction 7
  2.1 Introduction to PREPARED and Work Package 3.6 7
  2.2 Software Manual Structure 8

3 Model Calibration: From Theory to Application 9
  3.1 Probability Theory 9
  3.2 Running a model calibration 9
  3.3 Software Structure 10

4 Example Applications 13
  4.1 Running a Formal Bayesian Analysis 13
  4.2 Running an Informal Bayesian Analysis 16

5 Toolbox Classes: Member Function and Data Member Descriptions 19
  5.1 modelCalibration class 20
     5.1.1 Data Members 20
     5.1.2 Member Functions 20
  5.2 mcSampling class 22
     5.2.1 Data Members 22
     5.2.2 Member functions 22
  5.3 InformalLikelihoods class 24
     5.3.1 Data Members 24
     5.3.2 Member Functions 24
  5.4 informalBayesianAnalysis class 26
     5.4.1 Data Members 26
     5.4.2 Member Functions 27
  5.5 formalLikelihoods class 30
     5.5.1 Data Members 30
     5.5.2 Member Functions 30
  5.6 formalBayesianAnalysis class 33
     5.6.1 Data Members 33
     5.6.2 Member Functions 34
  5.7 ParameterAnalysis class 37
     5.7.1 Data Members 37
     5.7.2 Member Functions 37
  5.8 PredictionAnalysis class 39
     5.8.1 Data Members 39
     5.8.2 Member Functions 39
  5.9 genericFunctions class 41
5.9.1 Member Functions 41
5.10 Additional functionality in all classes 48

6 Software Installation 49
6.1 Example installation using the import library 49
6.2 Example installation by explicit methods 50

7 References 51
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 of 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 for application in UWS models evaluated (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, an online toolbox for real-time data assimilation and error correction.
2.2 Software Manual Structure

In section 3, some general theory for the probabilistic Bayesian approach to calibration is presented, alongside an overview of the calibration toolbox, and a general description of running a model calibration. Section 4 provides some code examples of how to use the toolbox to run a model calibration, and Section 5 provides a more detailed description of each class in the toolbox, its data members and member functions. Finally, section 6 provides guidance on software installation.
3 Model Calibration: From Theory to Application

3.1 Probability Theory

The software presented in this manual is designed for the calibration of numerical models within a probabilistic framework. In order to account for the multiple sources of system uncertainty that affect the accuracy of parameter calibration and model prediction, rather than calibrating a model to identify the Maximum Likelihood Estimate (MLE) of each model parameter, uncertainty in model parameters and predictions is quantified in the form of posterior (e.g. post calibration) Probability Density Functions (PDFs). The posterior uncertainty in model parameters ($\theta$), given a vector of observations used for model calibration ($Y$) is obtained via Bayes’ equation:

$$P(\theta|Y) = P(Y|\theta)P(\theta)$$

(1)

The second right hand term is the prior distribution of model parameters, which is typically chosen as a uniform distribution, the least informative distribution within a probabilistic framework. The first right hand term is the likelihood function. Solving Bayes’ equation analytically is typically intractable, and therefore some form of numerical sampling is required. Alongside the likelihood function, the way in which the posterior distribution is sampled may affect the thoroughness, and therefore validity of the model calibration. Within the probabilistic Bayesian framework, two approaches are provided in the toolbox for model calibration, which differ primarily by the Likelihood function chosen to derive posterior probabilistic information: Formal Bayesian Analysis and Informal Bayesian Analysis (See Hutton et al., (2012) and references therein for a discussion of these alternative methodologies.).

3.2 Running a model calibration

Regardless of the likelihood function chosen and the sampling procedure employed, both methods conform to a generic approach for first Parameter Sampling and second Uncertainty Analysis:

//Sampling

Define Prior Parameter Distributions (PDDs);

count = 0;

Do {
    Sample Parameter Set from prior Distributions;
    Run Model using the Parameter Set to Obtain Predictions;
}
Calculate Parameter Set Likelihood by Comparing Predictions to Observations;
Store Set of Parameters, Predictions and Likelihood;

\[
\text{count += 1;}
\]

} while(count < totalRuns);

//Uncertainty Analysis
Calculate Probabilities from Likelihoods;
Calculate Parameter Probability Density Functions;
Calculate Sensitivity Statistics;
Calculate Prediction Probability Density Functions;
Calculate Confidence and/or Prediction Intervals;
Output results to file;

3.3 Software Structure

The software in the calibration toolbox has been written in the programming language C++ as a group of classes, each with specific functions for Parameter Sampling and Uncertainty Analysis (Figure 1). The classes and contained functions have been separated such that alternative, user specified functions may be employed in place of those contained within the toolbox, whilst still enabling access to the functionality of other classes.

The centre of the toolbox in the class modelCalibration, which contains data members for storing the calibration results from Parameter Sampling for subsequent Uncertainty Analysis. The user specifies before calibration the number of (typically best performing) parameter sets, or samples to retain for posterior analysis (tSamples). For each set, the parameters, predictions and likelihood are stored in modelCalibration. These arrays are then passed as function arguments as required for uncertainty analysis. Underlying much of the toolbox is the genericFunctions class that contains algorithms commonly used in many of the other classes.

Parameter Sampling

Parameter Sampling first requires a sample from the prior distributions for each model parameter, which in the mcSampling class, are drawn randomly from user specified uniform priors for each parameter. Following sampling, the model in question is run for a time series of driving conditions using the sampled parameter set to derive a vector of model predictions comparable to the user supplied vector of observations. Both formal
Bayesian and Informal Bayesian Likelihoods are provided in the `formalLikelihoods` and `informalLikelihoods` classes respectively, to derive posterior likelihoods for each parameter set, and associated vector of model predictions.

**Figure 1.** Overview of the software and class structure within the Bayesian Framework. The software is generally split into classes concerned with sampling and following sampling, uncertainty analysis. Two routes are available for sampling: formal Bayesian approach, or an informal Bayesian approach.
Uncertainty Analysis

Following Parameter Sampling, the parameter sets and associated predictive data are then passed to either formalBayesAnalysis or informalBayesAnalysis depending on the likelihood function uses in Parameter Sampling. Within the above functions parameterAnalysis is called upon to calculate posterior parameter PDFs and run the parameter sensitivity analysis, whilst the predictionAnalysis class is called to calculate the posterior prediction PDFs for each observation, and, where required, confidence intervals and prediction intervals.
4 Example Applications

The application of the toolkit to run both a formal and informal Bayesian analysis is demonstrated for a simple, linear model. The example code is available with this manual, which may be used as a starting point for users to adapt the toolkit to their own applications. The observations have been derived by perturbing model predictions with a known noise distribution, which also serves to demonstrate the ability of the formal Bayesian analysis to infer the unknown error structure parameters, alongside those of the actual model.

4.1 Running a Formal Bayesian Analysis

The application of the toolkit to calibrate a linear model using a formal Bayesian analysis is demonstrated in the code snippets from Figure 2 through to Figure 5, which is a general structure for Monte Carlo calibration that may be applied to most models. On lines 6-11 the relevant header files for the classes to be called during calibration are specified (note: the genericFunctions class is used here to set up and perturb the linear model predictions to demonstrate calibration, and does not need to be declared explicitly otherwise). The linear model function and function to generate data in order to calibrate the linear model are declared on line 12 and line 16, respectively. Two vectors are then declared on lines 18 and 19 to store the vector of observations used in calibration that need to be supplied by the user, and the associated model predictions, each time the model is run.

Figure 2. First C++ code snippet showing the example application of linear model calibration using the formal Bayesian Analysis

Figure 3 shows the start of the main function, where the model and parameters are set up prior to sampling. The number of observations, parameters, and retained parameter sets for posterior analysis, are declared on lines 28 to 30. On line 31 the generateData() function is called, which generates 1000 observations from a linear model with an intercept of 0.6 and a
The third parameter passed is the standard deviation of a Gaussian noise, which is added to the linear model predictions to generate “observations”.

Lines 34 to 38 specify the prior uniform parameter distributions for the three parameters that will be inferred during calibration: the intercept (a) and the gradient (b), which are the model parameters, and the third parameter, which is the standard deviation of the Gaussian likelihood function used in calibration. Finally, objects of the relevant classes are then created from lines 41 to 57. This is done by declaring a pointer to the desired class type and calling the appropriate factory function to assign a value to this pointer. The factory function creates an instance of the class in memory and returns a valid pointer to that class. Class members may now be accessed through the class pointer. Three of these classes have to be initialised: on line 43 the `modelCalibration::initialise()` function is initialised to create storage for the calibration results; `formalLikelihood::initialise()` passes the observations to the likelihood function; and `mcSampling::initialise()` passes the prior parameter ranges for parameter sampling.

Figure 3. Second C++ code snippet showing the example application of linear model calibration using the formal Bayesian Analysis: set up prior to parameter sampling.
Figure 4. Third C++ code snippet showing the example application of linear model calibration using the formal Bayesian Analysis: Parameter Sampling.

Figure 4 shows the main parameter sampling loop. On line 67 a sample is drawn from the uniform priors for each model parameter. On lines 70 to 72 the linear model is run by passing the parameters from mcSampling object Psamp, which are then stored in the array predictions. On lines 75 to 78 the formal likelihood is calculated associated with the model predictions and parameter set. First, on line 76 the standard deviation is set for each observation, using setStd() of the Plike object. The standard deviation, sampled as a parameter, is passed to the function, which may be multiplied by each observation to set an observation specific standard deviation, and therefore account for heteroscedastic errors. As the final argument is set to zero in the function, this does not occur here. The negative log likelihood for the sample is calculated on line 78, and on line 81, the result alongside the sample parameters and predictions are passed to addSample() in the Pcal object. This function determines whether the sample likelihood is large enough to be retained as one of the retainedParameterSets.

Figure 5. Fourth C++ code snippet showing the example application of linear model calibration using the formal Bayesian Analysis: Uncertainty Analysis
Figure 5 shows the final stage in calibration, uncertainty analysis. First, on line 88, the negative log likelihoods are converted to normalised probabilities. Then, on lines 91-92, the formal Bayesian analysis object is initialised, by passing the calibration results, observations, and required confidence interval. On line 95 the uncertainty analysis is run. The first argument in `Pbayes->runAnalysis()` is a function that samples from the chosen error model in order to calculate the prediction intervals; the second argument is the number of parameters this function takes, which in this case is one, the Standard Deviation of the Gaussian error model; and the final argument is the number of samples taken from this distribution in order to generate the prediction intervals. The function has been written as such, so that users can employ their own error models during parameter sampling, but also still make use of the `formalBayesianAnalysis()` by passing a function that samples from their chosen distribution.

Finally, lines 98 to 104 write out the results of the calibration to file, and lines 108 to 111 release the allocated memory by calling the `destroy()` function of each class that was created. This should be done whenever a class is no longer needed.

### 4.2 Running an Informal Bayesian Analysis

Running an informal Bayesian analysis proceeds in much the same way as the formal Bayesian analysis presented in section 4.1.

```cpp
1 // model calibration.cpp example application
2 #include "stofcs.h"
3 4 //header files of calibration classes.
5 #include "calibrationTools.h"
6 7 8 //Linear model function, with gradient (a) and intercept (b).
9 void linearModel(double a, double b, double x, double &y);
10 11 //function to generate 'data' using the linear model.
12 //Observations perturbed with random normal noise, with standard deviation (std).
13 void generateData(int length, double a, double b, double std);
14 15 double *observations; //vector to store the observations generated for use in model calibration
16 double *predictions; //vector to store the latest model predictions
17 18 int nObs; //number of observations used for calibration
19 int parameters; //total number of parameters.
20 int retainedParameterSets; //number of parameter sets retained for posterior analysis.

Figure 6. First C++ code snippet showing the example application of linear model calibration using the informal Bayesian Analysis.

The only difference between the formal Bayesian approach in Figure 7 is that here, `initialiseNSE()` is run to initialise the informal likelihood function, and the `informalBayesianAnalysis()` function is initialised.
Figure 7. Second C++ code snippet showing the example application of linear model calibration using the informal Bayesian Analysis.

During parameter sampling (Figure 8) the informal likelihood is calculated on lines 70-71.

Figure 8. Third C++ code snippet showing the example application of linear model calibration using the informal Bayesian Analysis.

The principal difference between the practical coding of the formal and informal Bayesian approaches for calibration comes in posterior analysis (Figure 9), where the behavioural thresholds in the informal Bayesian analysis need to be specified by the user (lines 83 to 89) and then passed to the `runAnalysis()` function of the `informalBayesianAnalysis` class pointed by PIbayes.
Figure 9. Running the analysis.
5 Toolbox Classes: Member Function and Data Member Descriptions

This section provides more detailed information regarding each Class’s functionality. Many class functions do not need to be called explicitly to run a full calibration by the user, as they are utilised already within other classes. For example, the genericFunctions class is used by a number of other classes, and contains some algorithms routinely called in both Formal and Informal Bayesian analysis. However, a more thorough class description is provided to facilitate extension and development of the methods provided.

For most classes, information is passed by pointer or by reference within each function call. In addition, the data members of most classes are created in an initialise() function. The arguments for such functions are the fully capitalised names of specific data members.

In the following description, where data members are arrays and classes, the integer parameters used to dimension these arrays are shown in square brackets. This is to help prevent errors when using an accessing array values, and so users can access individual results, independently of the provided "output to file" functions. The abstract interface for these classes is defined in “calibrationToolbox.h”.
5.1 modelCalibration class

modelCalibration is the central point of the calibration toolbox, the data members of which store the model calibration results for subsequent posterior analysis.

5.1.1 Data Members

- **int tPar:** total number of parameters in calibration
- **int tObs:** total number of observations used in calibration
- **int tSamples:** total number of samples retained from sampling for posterior analysis
- **double **par [tPar][tSamples]:** store for parameter sets of each sample
- **double **pred[tPred][tSamples]:** store for the predictions made by each parameter set (sample)
- **double *like[tSamples]:** store for the likelihood associated with each sample.

5.1.2 Member Functions

**initialise**

```cpp
void initialise(int TPAR, int TOBS, int TSAMPLES);
```

Description: initialises the modelCalibration class and storage arrays for sampling.

Arguments:

- **int TPAR:** input to tPar
- **int TOBS:** input to tObs
- **int TSAMPLES:** input to tSamples

**addSample**

```cpp
void addSample(double likelihood, double * parameters, double *predictions);
```
**Description:** Compares the likelihood of the sampled parameter set with those in storage to determine whether to retain the parameter set. Note: the function works based on the assumption that the higher the likelihood, the better the simulation. This is valid for most informal likelihoods, and formal negative log-likelihoods.

**Arguments:**

- `double likelihood`: the Likelihood of the parameter set.
- `double * parameters [tPar]`: vector storing the parameter set.
- `double *predictions [tObs]`: vector storing predictions associated with the parameter set.

**return[Data Member]**

- `double* returnLike();` //returns like
- `int returnSamp();` //returns tSamples
- `double** returnPar();` //returns Par
- `int returnTpar();` //returns Tpar
- `double** returnPred();` //returns Pred
- `int returnTobs();` //returns tObs

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

**Arguments:**

None.
5.2 mcSampling class

mcSampling provides functions for running a Monte Carlo – or random sampling – procedure from specified uniform prior distributions, for model calibration.

5.2.1 Data Members

**int tPar:** number of model parameters to calibrate.

**double *parMax [tPar]:** vector of maximum values for the uniform prior distributions for each model parameter.

**double *parMin [tPar]:** vector of minimum values for the uniform prior distributions for each model parameter.

**double *par [tPar]:** vector storing the most recently sampled model parameter set

5.2.2 Member functions

**initialise**

```c
void initialise(int TPAR, double *PARMIN, double *PARMAX);
```

**Description:** initialises an mcSampling object for calibration

**Arguments:**

- **int TPAR:** input to `int tPar`
- **double *PARMIN:** input to `double *parMin`
- **double *PARMAX:** input to `double *parMAX`

**sample**

```c
void sample();
```

**Description:** samples from the uniform priors of each model parameter, storing the results in *par.
Arguments: none

Return[Data Member]

double returnpar(int); //returns par value at index
double* returnparpointer(); //returns *par
double* returnParMin(); //returns *parMax
double* returnParMax(); //returns *parMin

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

Arguments:

None.
5.3 InformalLikelihoods class

informalLikelihoods provides three informal likelihood functions to calculate posterior probabilities within the informal Bayesian framework. See Smith et al. (2008) for further information regarding these functions. A new object of this class should be created each time a different informal likelihood function is used.

5.3.1 Data Members

int tObs: total number of observations used in calibration

double obsMean: mean of the observations vector

double denom: the denominator used in the respective informal likelihood function

double *observations [tObs]: store for observations used to calculate the informal likelihood

double exp: exponent used to determine between different likelihood functions

double numerator: the numerator for the specified likelihood function

5.3.2 Member Functions

initialiseNSE

void initialiseNSE(double *OBSERVATIONS, int & TOBS, double EXP);

Description: initialises the Nash-Sutcliffe Efficiency (NSE) based likelihood function when EXP = 2, and the Normalised Absolute Error (NAE) based likelihood when EXP = 1.

Arguments:

double *OBSERVATIONS: input to *observations

int & TOBS: input to tObs

double EXP: input to exp

runNSE

void runNSE(double *predictions, double & result);
**Description**: runs the NSE or NAE likelihood function, depending on the exponent set in initialiseNSE().

**Arguments**:

- `double *predictions [tObs]`: vector of model predictions
- `double & result`: return value of the likelihood

---

**initialiseNSSE**

```c
void initialiseNSSE(double *OBSERVATIONS, int & TOBS);
```

**Description**: initialises the Normalised Sum of Square Errors (NSSE) based likelihood function

**Arguments**:

- `double *OBSERVATIONS`: input to *observations
- `int & TOBS`: input to tObs

---

**runNSSE**

```c
void runNSSE(double *predictions, double & result);
```

**Description**: runs the NSSE based likelihood function.

**Arguments**:

- `double *predictions [tObs]`: vector of model predictions
- `double & result`: return value of the likelihood
5.4 **informalBayesianAnalysis class**

informalBayesianAnalysis provides functions to run an informal Bayesian Analysis, calculate parameter uncertainty, parameter correlation, parameter sensitivity, sensitivity of calibration to the behavioural threshold chosen during calibration, and confidence intervals for observations used in calibration, and functions to output summary information to space delimited text files.

5.4.1 **Data Members**

- **int tSamples**: total number of samples (parameter sets) used in posterior analysis
- **int tObs**: number of observations used in calibration
- **int tPar**: number of parameters in a parameter set
- **int tThresh**: number of sample (behavioural) thresholds used in analysis
- **double **pars [tPar][tSamples]**: store for each parameter set
- **double *obs [tObs]**: vector of observations
- **double *like [tSamples]**: store for likelihood associated with each parameter set
- **double *parmax [tPar]**: store for minimum value of uniform prior distributions for each parameter
- **double *parmin [tPar]**: store for maximum value of uniform prior distributions for each parameter
- **double *thresholds [tThresh]**: vector of decimal values denoting the fraction of the total number of retained runs that defines each behavioural threshold investigated
- **double * mlePar [tPar]**: vector storing the maximum likelihood parameter set
- **double maxLike**: maximum likelihood value
- **double **pred [tObs][tSamples]**: predictions associate with each observation from each parameter set (model run).
- **int *parRank [tSamples]**: stores the rank of each parameter set from maximum to minimum likelihood
- **parameterAnalysis * pA [tThresh]**: vector of parameterAnalysis.h objects for each threshold
- **predictionAnalysis *prA [tThresh]**: vector of predictionAnalysis.h objects for each threshold
- **double ci**: confidence interval required for prediction bounds (decimal: e.g. 0.95 gives 95% prediction bounds).
5.4.2 Member Functions

**initialise**

```c
void initialise(double **PARS, double *LIKE, int TSAMPLES, int TPAR,
                 double *PARMIN, double * PARMAX, double **PRED, int & TOBS, double *Obs,
                 double CI);
```

**Description:** initialises pointers to arrays, and data storage for informal Bayesian analysis

**Arguments:**

- `double **PARS [tPar][tSamples]`: pointer to initialise `double **pars`
- `double *LIKE [tSamples]`: pointer to initialise `double *like`
- `int TSAMPLES`: initialises `tSamples`.
- `int TPAR`: initialises `tPar`.
- `double *PARMIN [tPar]`: initialises `double *parmin`.
- `double * PARMAX [tPar]`: initialises `double *parmax`.
- `double **PRED [tObs][tSamples]`: initialises `double **pred`;
- `int & OBSERVATIONS`: initialises `int observations`.
- `double *Obs [tObs]`: vector of observations used in calibration.
- `double CI`: initialises `double ci`.

**runAnalysis**

```c
void runAnalysis(int TTHRESH, double *thrs);
```

**Description:** runs an informal Bayesian analysis

**Arguments:**

- `int TTHRESH`: input to `tThresh`
double *thrs [tThresh]: vector of thresholds for use in informal Bayesian analysis. These need to be in the range (0-1], and specify a fraction of the total number of retained samples (tSamples).

outputTables

void outputTables(std::string filename);

**Description:** produces a space delimited file containing output tables in different formats describing calibration results and model parameter sensitivity

**Arguments:**
std::string filename: a filename to where the table is stored

outputPDFCDF

void outputPDFCDF(std::string filename);

**Description:** produces a space delimited file containing posterior PDFs and CDFs for all parameters, for each behavioural threshold

**Arguments:**
std::string filename: a filename to where the table is stored

outputPredInt

void outputPredInt(std::string filename);

**Description:** outputs the informal Bayesian prediction intervals for each behavioural threshold specified.

**Arguments:**
std::string filename: a filename to where the table is stored

sortPAR

void sortPar();
**Description:** sorts the parameters sets from largest likelihood to smallest

**Arguments:** none.

**sortPred**

```c
void sortPred();
```

**Description:** sorts the predictions sets from largest likelihood to smallest

**Arguments:** none.
5.5 formalLikelihoods class

formalLikelihoods provides a Gaussian likelihood function for formal Bayesian Analysis, as well as capability to account for heteroscedastic errors by setting the error variance for each observation. A new object of this class should be created each time a different informal likelihood function is used.

5.5.1 Data Members

int tObs: number of observations used in calibration.

double *observations [tObs]: vector of observations use in the likelihood function.

double *std [tObs]: vector of standard deviations, one for each observation that can be used to account for Heteroscedasticity in the residuals.

5.5.2 Member Functions

initialise

void initialise(double *OBSERVATIONS, int TOBS);

Description: initialise a formalLikelihoods object.

Arguments:

double *OBSERVATIONS [tObs]: input to observations.

int TOBS: input to tObs.

negLogGaussLF

void negLogGaussLF(double *predictions, double & result);

Description: calculates the negative log likelihood for a vector of predictions.

Arguments:

double *predictions [tObs]: predictions used to calculate the negative log likelihood.

double & result: return value of the negative log likelihood.
捏下样本的高斯似然函数，用于在预测分析类中计算预测区间。

**Arguments:**

- `double *par [1]`: 参数用于高斯似然函数。在这个例子中，一个维度为1的向量，包含高斯似然函数的标准差。
- `double & result`: 返回值为从高斯分布中采样的值。

```c
(*f1)(double *, double &) sampGaussLF();
```

**Description:** 返回指向stat_sampGaussLF函数的指针给函数调用者。这是由formalBayesianAnalysis类的runAnalysis函数所需的。

**Arguments:**

None.

```c
void normLogProb(double *normprob, double *loglike, int tSamples);
```

**Description:** 将一个负对数似然函数向量转换为一个正常化概率向量。

**Arguments:**

- `double *normprob [tSamples]`: 返回正常化概率的向量。
- `double *loglike [tSamples]`: 对数似然函数向量。
- `int tSamples`: 样本数。
double *par [1]: parameters used in the Gaussian likelihood function. In this case, a vector of dimension 1, containing the standard deviation of the Gaussian likelihood function.

**setStd**

*void setStd(double *predictions, double STD, double code)*;

**Description:** sets the standard deviation for each prediction for use in the negative log likelihood

**Arguments:**

*double *predictions [tObs]:* vector of model predictions associated with the vector of observations

*double STD:* sampled standard deviation used in the Gaussian Likelihood, used as input to populate *std

*double code:* set as zero for the same standard deviation for each prediction; set as one to set the standard deviation as the product of STD and the prediction value.
5.6 **formalBayesianAnalysis class**

formalBayesianAnalysis provides functions to run an informal Bayesian Analysis, calculate parameter uncertainty, parameter correlation, parameter sensitivity, and both confidence intervals for the model, and prediction intervals based on the chosen formal likelihood function applied.

5.6.1 **Data Members**

- **int tSamples:** total number of samples retained from sampling for posterior analysis
- **int tPar:** number of parameters calibrated
- **int tObs:** number of observations used in calibration
- **double **pars [tPar][tSamples]:** store for each parameter set
- **double *like [tSamples]:** store for likelihood associated with each parameter set
- **double *parmax [tPar]:** store for minimum value of uniform prior distributions for each parameter
- **double *parmin [tPar]:** store for maximum value of uniform prior distributions for each parameter
- **double *thresholds [tThresh]:** vector of decimal values denoting the fraction of the total number of retained runs that defines each behavioural threshold investigated
- **double * mlePar [tPar]:** vector storing the maximum likelihood parameter set
- **double maxLike:** maximum likelihood value
- **double **pred [tObs][tSamples]:** predictions associate with each observation from each parameter set (model run).
- **int *parRank [tSamples]:** stores the rank of each parameter set from maximum to minimum likelihood
- **parameterAnalysis * pA [1]:** vector of parameterAnalysis.h objects for each threshold
- **predictionAnalysis *prC [1]:** vector of predictionAnalysis.h objects for producing confidence intervals for each prediction
- **predictionAnalysis *pr{ [1]:** vector of predictionAnalysis.h objects for producing prediction intervals for each prediction
- **double ci:** confidence interval required for prediction bounds (decimal: e.g. 0.95 gives 95% prediction bounds).
- **double *obs [tObs]:** vector of observations
5.6.2 Member Functions

**initialise**

```c
void initialise(double **PARS, double *LIKE, int TSAMPLES, int TPAR,
double *PARMIN, double * PARMAX, double **PRED, double *OBS, int & TOBS,
double CI);
```

**Description:** initialise a formal Bayesian analysis object.

**Arguments:**
- `double **PARS`: input to pars
- `double *LIKE`: input to like
- `int TSAMPLES`: input to tSamples
- `int TPAR`: input to tPar
- `double *PARMIN`: input to parmin
- `double * PARMAX`: input to parmax
- `double **PRED`: input to pred
- `double *OBS`: input to obs
- `int & TOBS`: input to tObs
- `double CI`: input to ci;

**runAnalysis**

```c
void runAnalysis(void (*f)(double *,double & result),int errPar, int tErrSamp);
```

**Description:** runs the formal Bayesian analysis

**Arguments:**
void (*f)(double *, double & result): function passed to runAnalysis to sample from the given likelihood function used (e.g. if the Gaussian likelihood is used, sampGaussLF is passed)

int errPar: the number of error model parameters. These parameters should be the last parameters in the parameter vector(s), passed to formalBayesAnalysis

int tErrSamp: the total number of samples made from the assumed distribution to calculate the prediction intervals.

outputTables

void outputTables(std::string filename);

Description: output summary information, including optimal parameters, and sensitivity analysis

Arguments:

std::string filename: a filename (and extension) to where the file is to be stored.

outputPDFCDF

void outputPDFCDF(std::string filename);

Description: outputs the PDFs and CDFs for each parameter.

Arguments:

std::string filename: a filename (and extension) to where the file is to be stored.

outputPredInt

void outputPredInt(std::string filename);

Description: outputs the prediction intervals and confidence intervals for the observations

Arguments:
std::string filename: a filename (and extension) to where the file is to be stored.

**sortPAR**

```cpp
void sortPar();
```

**Description:** sorts the parameters sets from largest likelihood to smallest

**Arguments:** none.

**sortPred**

```cpp
void sortPred();
```

**Description:** sorts the predictions sets from largest likelihood to smallest

**Arguments:** none.
5.7 ParameterAnalysis class

ParameterAnalysis creates summary statistical information post calibration, including information on model sensitivity and parameter correlation.

5.7.1 Data Members

- int bins: number of bins used to construct the parameter PDFs
- int tSamples: number of samples used to construct distributions
- int tPar: number of parameters
- int coeffs: total number of parameter interactions
- double **CDF [tPar][bins]: store for parameter CDFs
- double **PDF [tPar][bins]: store for parameter PDFs
- double **binCent [tPar][bins]: store for central parameter value associated with each bin in the CDFs and PDFs
- double *maxProb [tPar]: maximum probability parameter set
- double *meanProb [tPar]: mean probability parameter set
- double *stdProb [tPar]: parameter standard deviation
- double *cdfDiff [tPar]: store for normalised aerial differences between posterior CDF and prior uniform CDF
- double *parMin [tPar]: minimum range of prior uniform distribution
- double *parMax [tPar]: maximum range of prior uniform distribution
- double *corrCoeff [coeffs]: store for correlation coefficients between parameters
- int *pc1 [coeffs]: store of first parameter index associated with correlation coefficients
- int *pc2 [coeffs]: store of second parameter index associated with correlation coefficients

5.7.2 Member Functions

runAnalysis
void runAnalysis(double **par, double *prob, int & TSAMPLES, int & TPAR, double * parmin, double * parmax, int BINS){

Description: runs a parameter analysis for a supplied sample of parameters and probabilities, calculating posterior PDFs and CDFs, parameter correlation coefficients and related summary information

Arguments:

double **par [tPar][tSamples]: array of parameters used in analysis

double *prob [tSamples]: array of probabilities associated with each parameter set.

int & TSAMPLES: entry to tSamples.

int & TPAR: number of parameters within each parameter set

double * parmin: vector of minimum values for the uniform prior distributions for each model parameter.

double * parmax: vector of maximum values for the uniform prior distributions for each model parameter.

int & BINS: number of bins used to create the PDFs and CDFs, controls how well resolved the parameter range is resolved
5.8  PredictionAnalysis class

predictionAnalysis calculates probability distributions, cumulative distribution functions and confidence intervals for those data used in calibration.

5.8.1  Data Members

int  tObs:  the number of observations used in calibration

int  tSamples:  the number of samples and therefore predictions at each observation point used to construct the confidence intervals

double  *ciU [tObs]:  the upper confidence interval for an observation

double  *ciL [tObs]:  the lower confidence interval for an observation

int  bins:  the number of bins used to construct the PDF and CDF

double  **PDF [tObs][bins]:  store for the PDF of each observation

double  **CDF [tObs][bins]:  store for the CDF of each observation

double  **binCent [tObs][bins]:  store for the values associated with the centre of each bin used to construct the PDFs and CDFs

double  *predMin [tObs]:  store for minimum prediction for each observation

double  *predMax [tObs]:  store for maximum prediction for each observation

5.8.2  Member Functions

 initialise

 initialise(double **PRED, double * PROB, int & TSAMPLES, int & TOBS){

 Description:  initialises a predictionAnalysis object. Must be called prior to runAnalysis

 Arguments:

 double  **PRED:

double  * PROB, int & TSAMPLES, int & TOBS
runAnalysis
unAnalysis(\texttt{double **pred, double * prob, int & LENGTH, int & TOBS, double & CI, int BINS});

\textbf{Description:} runs a predictions analysis for a supplied sample of predictions and probabilities, calculating posterior PDFs and CDFs, and confidence intervals.

\textbf{Arguments:}

\texttt{double **pred [observations][length]}: observations used to calculate the predictions intervals

\texttt{double * prob [length]}: probabilities associated with the predictions

\texttt{int & LENGTH}: input to \texttt{int length}

\texttt{int & TOBS}: input to \texttt{int tObs}

\texttt{double & CI}: required confidence interval percentage (decimal: e.g. 0.95 for 95\% confidence intervals)

\texttt{int BINS}: input to \texttt{int bins}
5.9 genericFunctions class

The generic functions class provides a series of generic algorithms utilised in both Formal and Informal Bayesian Analysis. The class has no data members; rather all data is passed by reference to the member functions of genericFunctions.h.

5.9.1 Member Functions

**MAE**

```c
void MAE(double *vector1, double *vector2, int &vectorlength, double & result);
```

**Description:** Calculates the mean absolute error between two vectors of numbers

**Arguments:**

- `double *vector1`: first vector of numbers
- `double * vector2`: second vector of numbers
- `int & vectorlength`: length of vector1 and vector2
- `double & result`: mean absolute error stored

**bubbleSort**

```c
void bubbleSort(double *values, int length);
```

**Description:** sorts a vector of values from largest to smallest

**Arguments:**

- `double *values`: vector of numbers.
- `int length`: length of vector to be sorted.
**bubbleSortRank**

```c
void bubbleSortRank(double *values, int *rank, int length);
```

**Description:** stores in "rank" the vector indexes of the vector “values” from highest to lowest. E.g. if rank[0] = 63, then value[63] is the largest in the vector “values”.

**Arguments:**

- `double *values`: vector of numbers.
- `int *rank`: vector where the rank is stored of the highest value in “values”.
- `int length`: length of vector to be sorted.

**Normalise**

```c
void normalise(double *values, double *norms, int length);
```

**Description:** normalises the vector “values” and stores the result in vector “norms”

**Arguments:**

- `double *values`: vector of numbers.
- `double *norms`: vector where the normalised vector “values” are stored
- `int length`: length of vector to be sorted

**probDist**

```c
void probDist(double *par, double * prob, int & length, double & min, double & max, int & bins, double *PDF, double * binCent);
```

**Description:** computes the probability density function (PDF) of a vector of values using an associated vector of probabilities.
Arguments:

- `double *par`: vector of values for which the PDF is calculated.
- `double * prob`: vector of probabilities associated with the values in “par”.
- `int & length`: length of input vectors
- `double & min`: minimum value of par used to construct the PDF bins
- `double & max`: maximum value of par used to construct the PDF bins
- `int & bins`: number of bins used to construct the PDF
- `double *PDF`: probability associated with each bin. The vector is of length “bins”.
- `double * binCent`: vector storing the central values of each bin, derived from “par”. The vector is of length “bins”.

**cumDist**

```c
void cumDist(double *PDF, int & bins, double * CDF);
```

**Description:** computes the cumulative probability density function (CDF) from a probability density function, which may be derived from probDist().

Arguments:

- `double *PDF`: vector of probabilities
- `int & bins`: length of vector of probabilities
- `double * CDF`: vector storing the cumulative probabilities

**CDFDiff**

```c
void CDFDiff(double *CDF, double *binCent, int & bins, double & area, double &parmin, double &parmax);
```

**Description:** calculates the normalised difference in area between two cumulative distribution functions.

Arguments:

- `double *CDF`: cumulative distribution probabilities
double *binCent: central values of each bin associated with the vector of cumulative probabilities.

int & bins: length of input vectors

double & maxDiff: return value for the maximum difference

double &parmin: minimum value, which alongside “parmax” is used to calculate the uniform prior distribution.

double &parmax: minimum value, which alongside “parmax” is used to calculate the uniform prior distribution.

coeffDeterm

void coeffDeterm(double *p1, double * p2, int &length, double & R2);

Description: computes the coefficient of determination between two vectors; e.g. the square of the Pearson’s product moment correlation coefficient

Arguments:

double *p1: first vector of values

double * p2: second vector of values

int &length: length of input vectors

double & R2: value where the coefficient of determination is returned from the function.

Max

void max(double *p1, int & length, double & maxVal, int & rank);

Description: computes the maximum value within a vector of numbers

Arguments:

double *p1: input vector of values

int & length: length of the input vector
double & maxVal: value where the maximum value of the input vector is returned from the function.

int & rank: value where the index of the maximum value within the input vector is returned from the function.

**Min**

void min(double *p1, int & length, double & maxVal, int & rank);

*Description:* computes the minimum value within a vector of numbers

*Arguments:*

double *p1: input vector of values

int & length: length of the input vector

double & maxVal: value where the minimum value of the input vector is returned from the function.

int & rank: value where the index of the minimum value within the input vector is returned from the function.

**initialiseRand**

void initialiseRand();

*Description:* initialises the random number generator using the computer clock.

*Arguments:* none.

**confInt**

void confInt(double *bin, double *CDF, int & length, double & ciL, double & ciU, double & ci);

*Description:* computes the confidence interval of a given percentage for a vector of values and associated cumulative probabilities

*Arguments:*
double *bin: input vector of values

double *CDF: cumulative probability associated with the input vector

int & length: length of the input vectors

double & ciL: value where the lower confidence interval is returned from the function

double & ciU: value where the upper confidence interval is returned from the function

double & ci: value where the required confidence interval is input to the function, which must be entered as a decimal: e.g. enter 0.95 for the 95% confidence interval.

sampleNormDist

sampleNormDist(double variance, double & result);

Description: generates a random number from a normal distribution, using the Box-Muller method.

Arguments:

double variance: required variance of the distribution from which to sample

double & result: value to which the resultant sample is added. Set as zero for the initial sample

calcMean

calcMean(double *vect1, int &length, double & mean);

Description: calculates the mean of a vector of numbers.

Arguments:

double *vect1: vector of numbers

int &length: length of vector of numbers

double & mean: return value for the mean
**calcStd**

calcStd(double *vect1, int &length, double &std);

*Description:* calculates the standard deviation of a vector of numbers.

*Arguments:*

- `double *vect1`: vector of numbers
- `int &length`: length of vector of numbers
- `double &std`: return value for the standard deviation

**randInt**

void randInt(double lower, double upper, double & result);

*Description:* calculates a random number over an interval.

*Arguments:*

- `double lower`: lower bound of the interval.
- `double upper`: upper bound of the interval.
- `double & result`: return value of the result.

**perturbKernSmooth**

void perturbKernSmooth(double *vect1, int length, double dirac);

*Description:* randomly perturbs a vector of values whilst maintaining the same mean and standard deviation of the original vector.

*Arguments:*

- `double *vect1 [length]`: vector of values to perturb
- `int length`: length of the vector
double dirac: factor that determines the strength of the perturbation

5.10 Additional functionality in all classes

destroy

void destroy();

Description: Deletes the class object, and releases memory that was dynamically allocated during its operation.

Arguments:

None.
6 Software Installation

The software toolkit consists of a precompiled dynamic library (.dll) containing the functionality for the Model Calibration class. 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 using the calibration toolkit from a user perspective in Visual Studio 2010 is provided in subsection 6.1.

6.1 Example installation using the import library

Create a new console application with default settings and copy the import library “CALIBRATION_TOOLBOX_DLL_GEN.lib” and the header file “calibrationToolbox.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.

Copy the dynamic library “CALIBRATION_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.
6.2 Example installation by explicit methods

To load the dynamic library at runtime, the header file “calibrationToolbox.h” containing the abstract classes and virtual functions should be included in the project. It is then necessary to load the eight factory functions, which will provide pointers to each of the eight classes as they are created at runtime. The eight functions are defined in the header file.

```c
#include "calibrationToolbox.h"

typedef IgenericFunctions* getGeneric();
ImcSampling* getMcSampling();
IPfSampling* getMcSampling();
IformalLikelihoods* getFormalLike();
liningLikelihoods* getInformalLike();
IformalBayes* getFormalBayes();
liningBayes* getInformalBayes();
ImodelCalibration* getModelCalibration();
```

Each factory function returns a pointer to a class defined in the interface (in “calibrationToolbox.h”). An example of dynamic loading is given in figure 10.

```c
int main()
{
    // Retrieve DLL handle.
    LPCSTR myDll = L"CALIBRATION_TOOLBOX_DLL_GEN.dll";
    HANDLE hDLL = LoadLibrary(myDll);
    if(hDLL == NULL)
    { //Library failed to load
        //Library loaded
    }
    p_getGeneric getGeneric = (p_getGeneric)GetProcAddress(hDLL, "getGeneric");
    p_getMcSampling getMcSampling = (p_getMcSampling)GetProcAddress(hDLL, "getMcSampling");
    p_getPfSampling getPfSampling = (p_getPfSampling)GetProcAddress(hDLL, "getPfSampling");
    p_getFormalLike getFormalLike = (p_getFormalLike)GetProcAddress(hDLL, "getFormalLike");
    p_getInformalLike getInformalLike = (p_getInformalLike)GetProcAddress(hDLL, "getInformalLike");
    p_getFormalBayes getFormalBayes = (p_getFormalBayes)GetProcAddress(hDLL, "getFormalBayes");
    p_getInformalBayes getInformalBayes = (p_getInformalBayes)GetProcAddress(hDLL, "getInformalBayes");
    p_getModelCalibration getModelCalibration = (p_getModelCalibration)GetProcAddress(hDLL, "getModelCalibration");
    //Now use the class factory functions as before!
    //Free the library
    FreeLibrary(hDLL);
    return 0;
}
```

Figure 10. Dynamic loading
7 References