UNIVERSITY OF AGDER

# Model Calibration with Hierarchically Structured Bayesian Learning Automata

**Jan Gunnar Andreassen and Lars Magne Engedal**

**Supervisor**
Ole-Christoffer Granmo

*This Master's Thesis is carried out as a part of the education at the University of Agder and is therefore approved as part of this education.*

University of Agder
Faculty of Engineering and Science
Department of ICT, 25.05.2011

# Abstract

When using a hydrological model to estimate the amount of available resources, the accuracy of the estimates depends on the calibration of the model. That is, one needs to find appropriate values for the model parameters. Calibration of hydrological models requires the exploration of a significant search space, rendering traditional gradient descent techniques sub-optimal. The Bayesian learning automaton has emerged as a simple and computationally efficient addition to current, largely evolutionary, calibration techniques. Although particularly well suited for learning in stochastic environments, the automaton struggles with navigating huge action spaces.

To alleviate this limitation, we introduce a hierarchically structured variant of the Bayesian learning automaton, applying it to the field of model calibration and function optimization. Several variants of the automaton is implemented and empirically tested, as well as compared to competing calibration techniques from the literature.

The new hierarchically structured automaton shows great promise, improving on action space handling compared to earlier, non-hierarchical structures. Indeed, the computational complexity now grows logarithmically rather than linearly with the size of the action space. Our experiments show that this approach is a viable alternative to competing calibration techniques.

# Preface

This master thesis is submitted in partial fulfillment of the requirements for the degree Master of Science in Information and Communication Technology at the University of Agder, Faculty of Engineering and Science. The required hydrological model and set of measurements were contributed by head of development at Agder Energi, Bernt Viggo Matheussen.

The work was carried out under the supervision of Associate Professor Ole-Christoffer Granmo at the University of Agder, Norway. We wish to thank him for his excellent comments and suggestions throughout our last semester as master students.

Grimstad, May 2011

Jan Gunnar Andreassen Lars Magne Engedal

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

The world is a complex system and human beings have been trying to understand how it works for quite some time. While trying to solve such tasks, researchers and scientists create models to represent a part of the system they are looking into. Good models represent a close guess of an infinitely complex process, and often one needs to adjust (calibrate) some of the quantities (parameters) which make up a model.

It is not always a trivial task to determine what solution is the "best" among a multitude of candidates. "The terminology "best" solution implies that there is more than one solution, and the solutions are not of equal value." [1] This is usually because the solution space is very large and/or is generated by very complex models or functions. Novel approaches to narrowing down the list of candidates are always appreciated, as it increases the accuracy and performance of the solution finding process.

## 1.1 Background

Our problem area is the area of model parameter calibration and reinforcement learning, in this thesis related to optimization methods for calibration of hydrological models. Such models are used for estimations of the water balance in a modelled catchment area [2]. The model used for calibration in this thesis, the HBV model [3] [4], is used by hydrologists to simulate how snow, rain, water, soil and evaporation act together within an area. Such knowledge could be used to estimate the amount of water available for hydroelectric power production, or to estimate the time and size of floods in vulnerable areas.

To provide useful information about the hydrological conditions in a catchment area, the model needs to be carefully calibrated to that particular area. High precision model estimates may be achieved with the use of automatic optimization techniques such as Bayesian learning automata, providing the model users with a clear picture of the hydrological conditions in the modelled area. However, calibrating the hydrological model is not a trivial task, as the model typically contains more than twenty tunable parameters. These parameters often represent conceptual rather than measurable entities, and must therefore be estimated indirectly through measurements of the system (model) response [2]. As such, calibrating the model constitutes exploration of a significant search space, with each parameter constituting a dimension of complexity.

Bayesian learning automata are known for rapidly and accurately converging towards the optimal action [5], and are shown to outperform established contenders [6]. This, combined with a low computational complexity, makes them a good candidate for model calibration.

The challenges with using Bayesian learning automata [6] for calibration of high-dimensional models, is connected to the fact that the size of the action space representing the search space grows exponentially with the number of dimensions: As the parameter space of each dimension is divided into a discrete number of actions, the total amount of actions will be “actions to the power of dimensions” ($a^d$). Current implementations of Bayesian learning automata need to explore every action, both when selecting which action to perform and when updating probability distributions with feedback from the examined environment.

To make the Bayesian learning automaton able to explore larger action spaces, we investigate methods for hierarchically structuring the action space. This approach is explored in an attempt to reduce the number of actions needed to be explored before a selection can be made, as well as reducing the number of actions that needs to be updated with feedback from the environment.

## 1.2 Thesis Definition

The purpose of this thesis is to investigate how the Kalman Filter Bayesian Learning Automata (KF-BLA) can be organized hierarchically in order to deal with huge action spaces, typically experienced when optimizing high-dimensional functions. Ideally, by pursuing a hierarchical organization of the action space, the resulting KF-BLA based learning algorithm should scale logarithmically with the number of actions available rather than linearly. With respect to applications, the thesis will focus on calibration of the OHBV hydrological model, a particularly complex real-life optimization problem involving a high-dimensional parameter search space. A prototype will be developed to empirically evaluate the computational efficiency, variance and accuracy of the proposed learning algorithm. The algorithm will also be compared to competing calibration methods described in the literature. These include the SCE-UA method and the genetic algorithm.

## 1.3 Related Work

Research into calibration of conceptual hydrological models, amongst them the HBV model, has lead to a number of different calibration methods [7] [8] [9]. Many of these techniques rely on evolutionary computing concepts, mixed with directed search and/or random sampling, to evolve model parameters.

The SCE-UA algorithm [7] was introduced for calibration of model parameters for hydrological models. In the paper, it is shown how effective and efficient this technique is when applied to different theoretical problems of varying complexity. SCE-UA uses complex evolution and simplex directed search to evolve the model parameters or variables, depending on the optimization problem, into globally optimal solutions. The SCE-UA is used in several articles focusing on calibration of models. In [10] the NWSRFS-SMA model, in [11] the MIKE 11/NAM model, and in [12] the SAC-SMA and SNOW-17 models are calibrated with the SCE-UA algorithm.

Duan et. al.'s research [7] provides a foundation for other optimization techniques, which try to enhance the original algorithm. Vrugt et. al. [8] proposes extensions, which adds Markov Chain Monte Carlo (MCMC) sampling, giving name to the new algorithm SCEM-UA [8]. They also propose a way to handle multiple objective functions. Multi-objective SCEM-UA (MOSCEM-UA) [13] solves problems where hydrological models output many different outputs for each simulation, but still using the same parameter set. It tries to balance the total model efficiency with regards to all objective functions.

A number of articles describe use of the genetic algorithm for calibration of hydrological models. The SWAT-model is calibrated in [14], the Xinanjiang-model in [15], and in [16] the genetic algorithm is compared to the SCE-UA when calibrating the CRR-model.

Bayesian learning automata have been applied for solving stationary two-armed Bernoulli bandit problems [5]. As stated in the article, learning automata are able to learn the optimal action when interacting with unknown stochastic environments, and they combine rapid and accurate convergence with low computational complexity. The article shows that the Bayesian learning automaton is among the top performers in its field.

The research on Bayesian learning automata is expanded upon in [6], which introduces methods for tracking changes in the environment. That is, situations where the reward probabilities are changing with time. Indeed, the article shows that this scheme outperforms established top performers, both for stationary and non-stationary bandit problems. The algorithm proposed by Granmo and Berg forms the basis for development of a hierarchically structured BLA, and this thesis investigates how the scheme performs in a high-dimensional calibration setting.

## 1.4 Contributions

A new optimization method, "Hierarchically structured Kalman Filter Bayesian Learning Automata" (HKF-BLA), is proposed and compared to other optimization methods. These other optimization methods include model optimization by the continuous genetic algorithm with random sampling, and shuffled complex evolution, combining evolutionary concepts with traditional gradient descent based optimization techniques.

Our work extends the field of reinforcement learning with a new learning automaton. We provide a review of its components and algorithms, which constitute the proposed solution. The main advantage with this scheme is the improved navigation of the action space, both when selecting which action to perform on the environment, and when distributing feedback from the environment throughout the action space. The new navigation methods scale logarithmically instead of linearly with the size of the action space, and thus it enables exploration of much larger action spaces than previous solutions. The HKF-BLA is general and may be applied to other fields (it doesn't rely on domain knowledge), although in this case it is tested on a specific model.

We show how the new method compares to other methods of calibrating hydrological models. The different methods are compared with respect to how well they optimize the HBV model, using a data set supplied by

Agder Energy spanning about nine years of hydrological observations. This data set contains measurements of precipitation, temperate and runoff from a catchment area. The HBV model itself is treated as a "black box," and isn't analyzed in detail. It serves only as a practical application area for the optimization algorithms. This model is exciting because it constitutes a significant search space for the algorithms to explore.

Results of the model optimization process and key findings are presented. We present experiments, results and observations relating to the model optimization process and provide a discussion of advantages, disadvantages and other properties of the HKF-BLA itself and the HKF-BLA compared with other optimization algorithms. These experiments give more insight into the performance and accuracy of Bayesian learning automata as optimization algorithms.

Finally, we propose further research based on the findings in this thesis, for example other real life applications or empirical evaluations. One such example could be on-line automatic model calibration, or other similar complex optimization problems.

## 1.5 Assumptions and limitations

Regarding development of the solution, we assume that all prior research on learning automata is correct and valid, including the as yet unpublished algorithm 3.3 in section 3.1.4. We also assume that the HBV model is representative of models used for modelling hydrological processes, meaning that we assume the solution will work on similar models. The implementation of the HBV model that the solution is used on, is treated as a "black box," and is not analysed or improved upon in any way.

As regards the experiment setup and results, we assume that the data set from our test catchment contains accurate measurements, and that the catchment is neither extremely hard nor extremely easy to calibrate compared to other catchments. Thus we assume that the results we get from calibrating the model to this catchment is similar to the results we would get from calibrating other catchments.

The implementation of the SCE-UA algorithm is provided by its author, and is used as is. We implemented the GA ourselves using public libraries. Both algorithms are used for comparison purposes only, meaning that their concepts are explained without providing deeper analysis of their behaviour.

Finally, we do not perform a qualitative analysis of the parameter sets produced by the model, as this falls more under the field of hydrology than the field of optimization and reinforcement learning. Furthermore, we assume that the calibrated model parameters are static over the time span of the catchment data set.

## 1.6 Report Outline

In the following chapter we give an introduction to model calibration, along with a presentation of how to use optimization algorithms for calibration of hydrological models. To highlight the problem area, we provide a brief overview of the HBV model (section 2.1). Finally, in section 2.2, we introduce several commonly used objective functions (methods for comparing observed and estimated data).

Chapter 3 contains background information regarding optimization algorithms, with section 3.1 focusing on concepts related to reinforcement learning. These concepts lay the foundation for the solution in chapter 4. Algorithms used for comparison purposes are also introduced in chapter 3. The information presented in section 3.2 and 3.3 relates to the genetic algorithm (GA) and shuffled complex evolution (SCE-UA) respectively, and readers familiar with these concepts may skip those sections without loss of continuity.

In chapter 4 we present the components and algorithms constituting the HKF-BLA. First, in section 4.1 we explain how we hierarchically structured the action space of the learning automaton. Then, in section 4.2 we proceed with explaining the algorithm for action selection and in section 4.3 the algorithms for distributing rewards throughout the action space.

Several versions of the HKF-BLA are compared in chapter 5, followed by comparisons of the HKF-BLA with competing optimization algorithms. The different algorithms are compared with two different objective functions, the standard deviation (rmse), and the Lindström coefficient. The results are discussed in chapter 6, followed by conclusions in chapter 7. An example of a python implementation of the HKF-BLA can be found in appendix A.

Chapter 2

# Model calibration

Just like the control knobs on your stereo, which allows you to adjust settings like volume, balance, bass, and treble, a rainfall-runoff (RR) model is usually equipped with adjustable parameters. These parameters tune the model to a specific problem area and thus avoids the trouble of having to write a unique model for every problem encountered. The goal of the modeller is now to adjust the model parameters, such that the model matches the underlying conditions. This process of modifying parameters and verifying the results against observed records, is called model calibration.

The need for this calibration procedure comes from the fact that RR models are conceptual (not necessarily corresponding to physical reality), and therefore most of the RR model's parameters are not directly measurable. [2] Instead, the parameters have to be estimated. Additionally, the structure of the RR model is usually specified for a general catchment area rather than a concrete one. As such, it is possible to adapt the RR model to different catchment areas instead of writing a new model for each one.

An important observation is that the measured input and response values of the real world system may or may not differ from it's true input and response. Errors, inaccuracies or noise in the measurements would make the model produce a simulated response different from the actual response of the system. This problem is of interest for hydrologists, but in the context of this thesis, all measurements are assumed to be correct.

Traditionally, RR models have been manually calibrated. That is, an experienced hydrologist tunes the model parameters while observing how the calculated response of the model fits the measurements from the specific area. As RR models typically have one or two dozen interdependent parameters, this is an extremely time consuming process. A desire to model more catchment areas faster has lead to the development of automatic calibration procedures.

A strategy for an automatic calibration process is outlined in figure 2.1. To start the calibration process, some prior information is used to initialize the parameters required by the model. This prior information could be the educated guess of a hydrologist, or a set of randomly generated numbers (although in both cases constrained by each parameter's range).

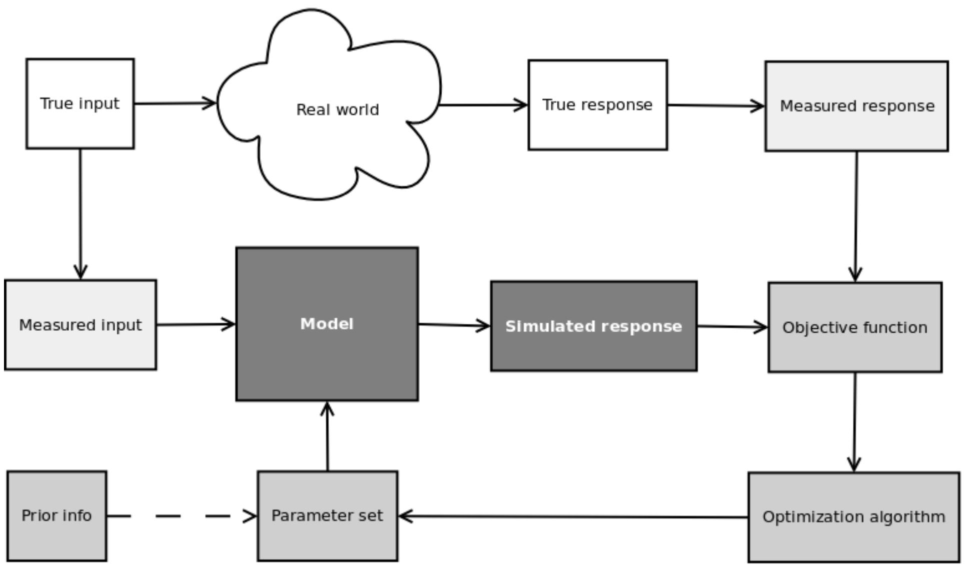


Figure 2.1: Strategy for model calibration

Assessing the fitness of the selected parameter set is done by running the measured input data through the model, resulting in the generation of a simulated response of the real world system being modelled. This simulated response is then compared to the measured response by means of an objective function, which output a score or similar fitness result for use by the optimization algorithm. The model used in this thesis, along with it's parameters, is presented in the next section, while different objective functions are discussed in section 2.2.

Finally, the parameter set is adjusted according to the strategy of the optimization algorithm, and the process starts over. The loop could continue for a fixed number of iterations, or be programmed to terminate upon stagnation. The different optimization algorithms used in the thesis are discussed in detail in chapter 3.

## 2.1 The model and its parameters

The rainfall-runoff model used in this thesis is the HBV model, developed by the Swedish Meteorological and Hydrological Institute. It consists of conceptual numerical descriptions of the hydrological processes in the modelled catchment area. The general water balance in the catchment is summarized in equation 2.1[17], with parameter names in table 2.1.

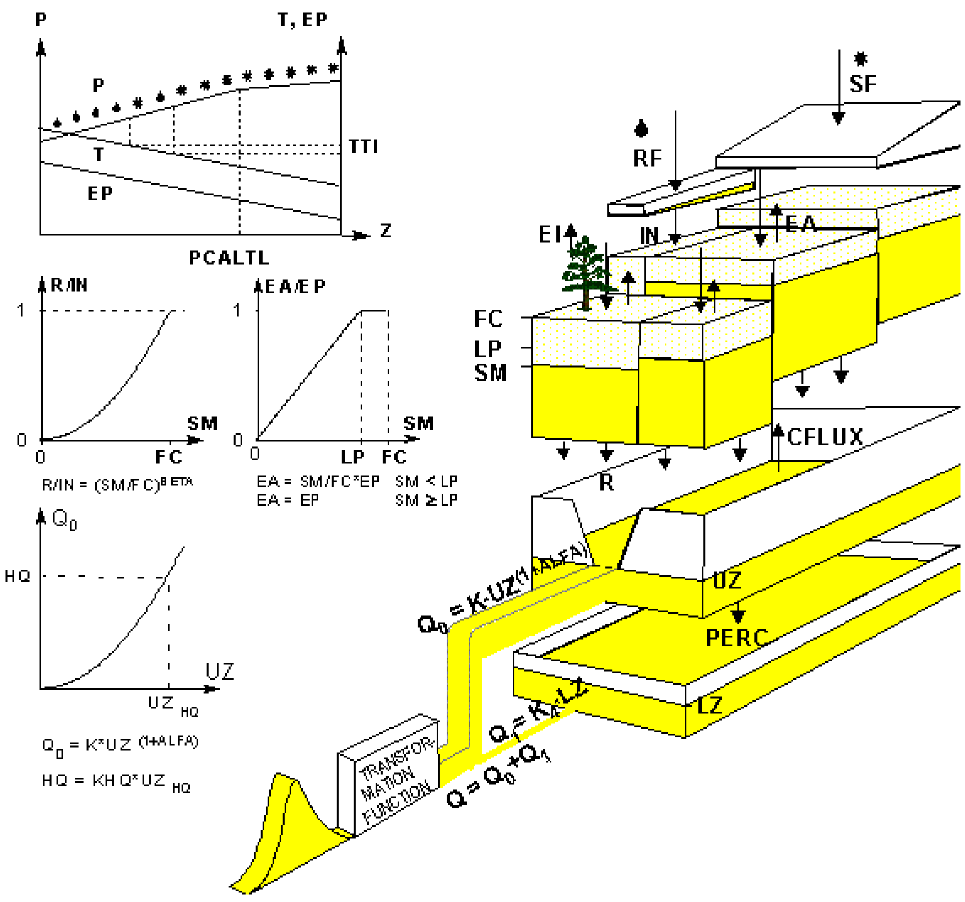
$$P - E - Q = \frac{d}{dt}(SP + SM + UZ + LZ + lakes) \tag{2.1}$$

| **Parameter** | **Name** |
|---|---|
| P | Precipitation |
| E | Evapotranspiration |
| Q | Runoff |
| SP | Snow pack |
| SM | Soil moisture |
| UZ | Upper groundwater zone |
| LZ | Lower groundwater zone |
| lakes | Lake volume |

Table 2.1: Water Balance Parameters

From the catchment area daily measurements of precipitation, air temperature and evapotranspiration (optional; estimates may be calculated from temperature) is input to the model. The air temperature is used for calculations of snow accumulation and melt, and may be omitted in snow free areas (if not used to calculate potential evaporation).

The model is composed of several subroutines, such as meteorological interpolation, snow accumulation and melt, evapotranspiration, soil moisture accounting, runoff generation and routing routines between sub-basins and lakes. The structure of the model is presented schematically in figure 2.2[17], showing the most important elements of the model.



P = Precipitation
T = Temperature
SF = Snow
RF = Rain
Z = Elevation
PCALTL = Threshold for altitude correction
TTI = Threshold temperature interval
IN = Infiltration
EP = Potential evapotranspiration
EA = Actual evapotranspiration
EI = Evaporation from interception
SM = Soil moisture storage
FC = Maximum soil moisture storage
LP = Limit for potential evapotranspiration

BETA = Soil parameter
R = Recharge
CFLUX = Capillary transport
UZ = Storage in upper response box
LZ = Storage in lower response box
PERC = Percolation
$K, K_4$ = Recession parameters
ALFA = Recession parameter
$Q_0, Q_1$ = Runoff components
HQ = High flow parameter
KHQ = Recession at HQ
$HQ_{UZ}$ = UZ level at HQ

Figure 2.2: HBV-model, with routines for snow(top), soil(middle) and response(bottom)

The snow accumulation and melt routine is designed to model the water holding capacity of snow and ice, which, based on temperature, delays runoff. This routine may use a hard threshold temperature TT, or an interval based threshold TTI, where precipitation inside the interval is assumed to be a mix of rain and snow. Different area types, such as open areas, forests and glaziers have different accumulation and melt functions.

The soil routine accounting models the water holding capacity of the soil in the catchment area. This routine is the main part controlling runoff generation, and is based on three parameters, BETA, LP and FC, shown in the middle of figure 2.2. BETA controls the increase in soil moisture from rainfall and snow melt, LP is the limit for potential evapotranspiration (water reentering the atmosphere) and FC is the maximum soil moisture storage in the model.

The response or runoff function transforms excess water from soil moisture and wet areas like lakes and rivers into runoff. The function is split into an upper non-linear zone(UZ) and a lower linear zone(LZ). These are the sources of the quick and slow runoff components $Q_0$ and $Q_1$ in the bottom of figure 2.2. The total runoff $Q$ calculated by the model is the value compared to the measured runoff from the catchment in order to determine the fitness of the parameter values.

## 2.2 Objective functions

The objective function is a central component to model calibration, as it is the means of comparing measured catchment response values with those simulated by the RR model. Since the simulated response values depends on the current model parameters, the objective function becomes a way of quantifying the fitness of the parameter set. This information is in turn used by the optimization algorithm to modify the parameters so that the model computes more accurate values.

### Standard deviation

One way of comparing two data sets, is by calculating their standard deviation, or root mean square error (2.2). In this function the squared difference at each data point is accumulated, then divided by the number of data points. This result is called the mean square error (MSE), and its root the RMSE. Taking the root also returns the unit of measurement back to the original. A perfect fit between the two data sets is indicated by an RMSE of zero (the two data sets are identical at all points), while an increasing RMSE value indicates an increasing distance between the two sets.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (x_{obs,i} - x_{sim,i})^2}{n}} \tag{2.2}$$

### Correlation

Correlation describes the dependence, or synchronicity, between two data sets 2.3. This function is slightly more complicated than the standard deviation. First, the product of each data set's deviation from its mean value is accumulated. Next, this value is divided by the root of the product of each data set's variance (the accumulated squared deviation from its mean value). A perfect correlation between two data sets will yield a score of 1. An opposite linear dependence is indicated by a score of -1, while an absence of any dependence is denoted by a score of 0.

$$CORR = \frac{\sum_{i=1}^{n} (x_{obs,i} - \bar{x}_{obs})(x_{sim,i} - \bar{x}_{sim})}{\sqrt{\sum_{i=1}^{n} (x_{obs,i} - \bar{x}_{obs})^2 \sum_{i=1}^{n} (x_{sim,i} - \bar{x}_{sim})^2}} \tag{2.3}$$

### The Nash-Sutcliffe coefficient

The Nash-Sutcliffe coefficient 2.4 can be explained as one minus the division of the mean squared error by the variance of the observations [18]. Thus two identical data sets will have a score of 1, while increasingly different data sets will have scores approaching $-\infty$.

$$R^2 = 1 - \frac{MSE}{Var(x_{obs})} = 1 - \frac{\sum_{i=1}^{n} (x_{obs,i} - x_{sim,i})^2}{\sum_{i=1}^{n} (x_{obs,i} - \bar{x}_{obs})^2} \tag{2.4}$$

### The Lindström coefficient

The Lindström coefficient 2.6 is an expansion of the Nash-Sutcliffe coefficient. Here the accumulated difference between predictions and observations are divided by the sum of the observations (2.5), before being weighted and subtracted from the Nash-Sutcliffe coefficient. See [9].

$$D_V = \frac{\sum_{i=1}^{n} (x_{sim,i} - x_{obs,i})}{\sum_{i=1}^{n} x_{obs,i}} \tag{2.5}$$

$$R_V^2 = R^2 - w|D_V| \tag{2.6}$$

# Chapter 3

# Optimization algorithms

Optimization is the process of making a solution better. Original concepts and ideas, both real life applications and theoretical problem solving, are improved by optimization methods developed in mathematics. It is widely known that the computer is well suited for intense computation, which has lead to many computer algorithms that solve different kinds of optimization problems.

A hydrological model is an example of a real life application, where calibration of this model is subject to optimization. In this thesis, the HBV [3] [4] model will be used as the hydrological model subject to calibration by the optimization algorithm. Calibrating the HBV model, with its 24 adjustable parameters, requires the exploration of a significant search space, which puts significant strain on the selected optimization method.

Different algorithms, GA [16] and SCE-UA [7] have been applied or introduced to solve the optimization problem of model calibration. These algorithms use traditional mathematical optimization techniques or an evolutionary process to perform the calibration. In some cases a method uses a combination of techniques, as we'll see in later sections.

## 3.1 Reinforcement Learning Algorithms

Most are familiar with the concept of learning by interaction, because it's inherent in human nature. Children and adults interact with an environment and interpret how their behaviour effects an outcome, cause and effect. Most of the time, there is not even a teacher or supervisor to guide the subject. Therefore, based on the outcome or a series of outcomes, the individual is able to gain knowledge about how action maps to outcome. Reinforcement learning algorithms encapsulates this concept of natural learning, describing components and how they interact, and transforms them into a computational approach to be used by computers and mathematicians.

### 3.1.1 Reinforcement Learning

Learning is an interactive and iterative process, in which the subject uncover how different actions result in specific rewards. Rewards in a philosophical sense may be discussed in length, but when considering computers it is confined to a numerical value. This numerical value represents how good a reward is, relative to other rewards. The main objective of reinforcement learning is to maximize these rewards over the entire learning period. This may seem intuitive, but we also know from real life experience that predicting the future, without substantial evidence and observations, is hard. To complicate things even further, the environment may be stochastic and actions may affect long-term reward, or even cause delayed reward. And all of this is done without any form of supervised instructions.

Since the agent doesn't get information from a teacher or supervisor, it has to gain experience about the environment by *exploration*. This exploration helps the agent build an internal model of how actions maps to response in the environment. But to maximize reward it has to *exploit* its experience. Therefore the agent is caught in a dilemma. Should it learn more and gain insight, or should it focus on what it already knows. This dilemma is referred to as exploration vs exploitation, and is a well known and well studied concept in reinforcement learning. $n$-armed bandit problem is an iconic example and a specific realization of the exploration vs exploitation dilemma.

Sutton and Barto defines a reinforcement learning algorithm as any method which solves a reinforcement learning problem [19]. As a result, there exists a wide variety of methods which conforms to this structure. They also identify four important sub-elements of reinforcement learning:

**Policy** The mapping between perceived states and actions taken. Determines behavior. "Stimulus-response rules" or associations.

**Reward function** Maps a state-action pair to a number, indicating the states immediate desirability value. The agent wants to maximize rewards in the long run. Reward and penalty may be compared with pleasure and pain in a biological system.

**Value function** *Accumulated reward* over the long run when starting from a given state. Rewards are directly linked to the environment, while value is estimated. Efficient value estimation is a key component in reinforcement learning algorithms.

**Model of the environment** The learning system produces a model of how it believes the environment will respond. The model is then used for planning how to proceed. This is the opposite of what trial-and-error learners do.

In comparison to reinforcement learning algorithms, *evolutionary* search methods such as, genetic algorithms, genetic programming, simulated annealing and other function optimization methods have been used to solve reinforcement learning problems, but these use reward as feedback and doesn't evaluate or consider a value function [19].

On the other hand, *supervised* learning rely on an expert guiding the subject. This is not always possible because it relies on creating examples of good behavior, which might is difficult, complex or simply impossible. For example, whenever the subject is supposed to learn something about an uncharted territory. The only way to achieve this is to explore or interact with the unknown environment. A stochastic environment may also lead to some of the same difficulties when considering supervised learning.

### Agent-Environment

The Agent-Environment framework is an abstraction which help define reinforcement learning problems. Its objective is to describe the necessary signals and how they interact. As in figure 3.1, the Agent constitutes the learner. For the purpose of Agent-Environment framework, the environment constitutes everything which is not the agent. An instance of a reinforcement learning problem is called a task, which also is a complete specification of an environment [19]. Environments may be stationary or non-stationary (dynamic), meaning the reward probability distribution might change with time.

The agent is able to interact with its environment, and this interaction happens at discrete steps $t = 0, 1, 2, 3, \dots, n$. These steps may be related to time or independent of time. At step $t$, the agent perceives the state of the environment as $s_t \in \mathcal{S}$, with $\mathcal{S}$ as a set of possible states. The agent has to chose an action $a_t$ from all available actions for the state $s_t$, $a_t \in \mathcal{A}(s_t)$. Now, the agent is able to exert this action on the environment, and in step $t+1$ it receives a reward $r_{t+1} \in \Re$ as

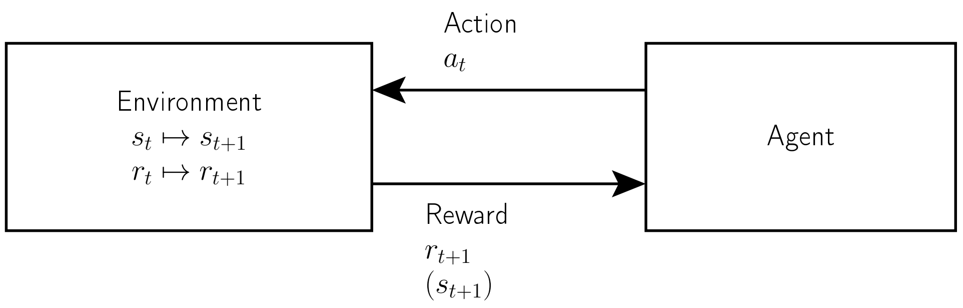


Figure 3.1: Agent and environment

a consequence. The *policy* at step $t$ is denoted as $\pi_t(s, a)$, which is the probability of selecting $a_t = a$ if $s_t = s$. This framework abstracts the reinforcement learning problem into three signals, passing between the agent and its environment [19]:

**The actions** Choices made by the agent.

**The states** Basis on which the choices are made.

**The rewards** The agent's goal.

To summarize, reinforcement learning defines automatic learning and decision-making described as a computational approach, with emphasis on interaction with an environment and learning from this interaction. This exposes the challenge between exploration and exploitation: Should the agent gain new information about its environment or make the most of its current information?

### 3.1.2 Bayesian Learning Automata

Before we look at the inner workings and motivation behind the Bayesian learning automata family of reinforcement learning methods, we'll give a brief refresher on Bayes' theorem, Bayesian inference and reasoning, probability distributions and conjugate priors.

Equation 3.1 is the famous Bayes' theorem, which simply put shows the connection between the conditional probability $A$ given $B$ (posterior probability) and the conditional probability of $B$ given $A$ (likelihood) and the probability of $A$ (prior probability). In other words, Bayes' theorem shows the connection between how old beliefs should be changed or updated in light of new evidence.

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)} \tag{3.1}$$

> "This simple equation underlies all modern AI systems for probabilistic inference."[20]

Sometimes there exists a relationship between the different probabilities in Bayes' theorem. When considering the probability distributions for the prior and posterior in Bayes' theorem, if they are in the same family, then they are said to be conjugate distributions. The prior distribution is said to be a conjugate prior for the likelihood function. In practice, this means that the integral in

$$p(\theta|x) = \frac{p(x|\theta)\,p(\theta)}{\int p(x|\theta)\,p(\theta)\,\mathrm{d}\theta} \tag{3.2}$$

will be of some known form, which is easier to deal with mathematically. In some cases, it reduces the calculations from very complex formulas and difficult integrals, down to updating hyper-parameters[1] of the conjugate probability distribution.

The Bayesian learning automaton is a reinforcement learning method, which uses Bayesian reasoning to guide its belief about a system. The automaton was introduced by Granmo in [5] and used it to solve different two-armed Bernoulli bandit (TABB) problems. As mentioned in section 3.1.1, $n$-armed bandit problems from a good testbed for investigation of reinforcement learning methods. The method showed several interesting properties such as being self-correcting and having guaranteed convergence to pulling the optimal arm. In addition, previous approaches that used Bayesian reasoning suffered from unbounded computation, which made them difficult and impractical to use in practice. Granmo's novel solution made the Bayesian reasoning computationally efficient, by leverage mathematical properties of conjugate priors, such as the *beta probability distribution*.

Following in algorithm 3.1 is pseudo-code of the Bayesian Learning Automaton, with the beta distribution

$$f(x;\alpha,\beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{\int_0^1 u^{\alpha-1}(1-u)^{\beta-1}\,\mathrm{d}u}, x \in [0,1] \tag{3.3}$$

[1]As is done in the Bayesian Learning Automata

**Algorithm 3.1** Bayesian Learning Automaton

```
Ensure: $\alpha_1 = \beta_1 = \alpha_2 = \beta_2$
 1: loop
 2:   Draw a value $x_1$ randomly from beta distribution $f(x_1, \alpha_1, \beta_1)$ with
      parameters $\alpha_1, \beta_1$.
 3:   Draw a value $x_2$ randomly from beta distribution $f(x_2, \alpha_2, \beta_2)$ with
      parameters $\alpha_2, \beta_2$.
 4:   if $x_1 > x_2$ then
 5:     $Arm\ i \leftarrow$ Pull $Arm$ 1.
 6:   else
 7:     $Arm\ i \leftarrow$ Pull $Arm$ 2.
 8:   end if
 9:   Receive either Reward or Penalty as a result of pulling $Arm\ i$.
10:   Increase $\alpha_i$ with 1 upon Reward, and increase $\beta_i$ with 1 upon Penalty.
11: end loop
```

### 3.1.3 The Kalman Filter

A Kalman filter is simply an *optimal recursive data processing algorithm*[21]. The reason filters such as the Kalman filter exists is because some things are difficult to measure exactly. Therefore the quantities have to be inferred by observation, which is often noisy and inaccurate. As the definition described, Kalman filters have several nice features when it comes to dealing with observations and estimation.

**Optimal** Incorporates all available information into a single measure, even though the measurements may have different noise, errors or uncertainties.

**Recursive** Doesn't require all previous data to be stored and processed when updating with new measures.

**Data Processing** Is usually implemented in software, running on a processing unit, as opposed to electrical or mechanical filters.

Here is an example which explains the main concepts of the Kalman filter and how they work together. Suppose you are out driving with a friend and as the passenger, you fall asleep. After a while you wake up at some unknown rest area and want to know the current location of the car. You look around, look at the time $t_1$ and guess the location to be $z_1$. As imaginable, this measure has quite high uncertainty $\sigma_{z_1}^2$, but it's good enough for you at this instance. With these points, one is able to create a conditional probability function $f_{x(t_1)|z(t_1)}(x|z_1)$ which gives the probability of being at location $x$ based on the measure $z_1$. The currently best estimate of position is $\hat{x}(t_1) = z_1$.

You ask your friend, the driver, where he thinks the car is located. He has access to a Global Positioning System (GPS) and gives the GPS location $z_2$ as his estimate. Since the location hasn't changed, $z_2$ is also said to happen at $t_2 \approx t_1$. This measure on the other hand has very low uncertainty $\sigma^2_{z_2}$ (at least compared to the manual observation).

Now we combine the two estimates into a new and better estimate, a Gaussian density with mean $\mu$ and the corresponding uncertainty $\sigma^2$, with the equations 3.4 and 3.5 respectively. Our new best estimate given this density is $\hat{x}(t_2) = \mu$.

$$\mu = \frac{\sigma^2_{z_2}}{\sigma^2_{z_1} + \sigma^2_{z_2}} z_1 + \frac{\sigma^2_{z_1}}{\sigma^2_{z_1} + \sigma^2_{z_2}} z_2 \tag{3.4}$$

$$\frac{1}{\sigma^2} = \frac{1}{\sigma^2_{z_1}} + \frac{1}{\sigma^2_{z_2}} \tag{3.5}$$

As suggested, the new estimate will have less uncertainty and will also incorporate all information available at that time. The next logical step is to introduce how to add new measures from the following time step to the old optimal estimate. One of the strengths of the Kalman filter is its recursive feature, only requiring the previous optimal estimate as starting point, yet allowing for new updates to be added.

Back to the traveling example. After the break, you head out towards the trip destination. The cars position is estimated to change at approximately some average speed $v$, give or take some error $\sigma^2_n$. Your best guess is the speed limit of the road. The location probability density function estimated at the rest stop will flatten and stretch out as time goes along. This is because for every time step, $x$ will move on average some distance $v$. As time passes, error accumulates and the Gaussian density, used to predict location $x$, flattens. Just before the car stops at time $t_3$, the best estimate of your current location $x$ at time $t_3^-$ will be

$$\begin{aligned} \hat{x}(t_3^-) &= \hat{x}(t_2) + v[t_3 - t_2] \\ \sigma^2_x(t_3^-) &= \sigma^2_x(t_2) + \sigma^2_n[t_3 - t_2] \end{aligned} \tag{3.6}$$

When the car stops, a measurement $z_3$ of the actual location is taken. $z_3$ is combined with the *prediction* of $\hat{x}(t_3^-)$ into the actual best estimate at the current time $t_3$

$$\begin{aligned}\hat{x}(t_3) &= \hat{x}(t_3^-) + K(t_3)[z_3 - \hat{x}(t_3^-)] \\ \sigma_x^2(t_3) &= \sigma_x^2(t_3^-) - K(t_3)\sigma_x^2(t_3^-)\end{aligned} \tag{3.7}$$

with $K(t_3)$

$$K(t_3) = \sigma_x^2(t_3^-)/[\sigma_x^2(t_3^-) + \sigma_{z_3}^2] \tag{3.8}$$

and again, we see that $\hat{x}(t_3) = \mu$ is the optimal estimate of our location $x$ at time $t_3$.

For a more complete primer, excellent additional information about Kalman filters, mathematical properties and engineering applications, see [21].

### 3.1.4 Tracking reward in dynamic environments

The tracking capability of Kalman filters are well-known and powerful[21] [20]. As a consequence, the Kalman filter and its tracking abilities pose as a good candidate for working with dynamic (non-stationary) environments. In these dynamic environments, the state of the reward probabilities change over time. This causes a major problem for the original BLAs intended for stationary environments with static reward probabilities.

Kalman filters do probabilistic reasoning over time, which suggest an iterative process. In section 3.1.3 it was introduced how different measures could be incorporated into a single optimal measure and how to incorporate new measures over time, in a recursive way.

Algorithm 3.2 contains pseudo-code for the Kalman filter Bayesian Learning Automaton. This algorithm was introduced by Granmo et al. [6] to solve non-stationary bandit problems. The KF-MANB serves as a foundation for our work and it is therefore repeated here. We'll walk through some of the important parts and show how it uses the Kalman filters.

The algorithm requires starting conditions, such as number of bandit arms $q$, observation noise $\sigma_{ob}^2$ and transition noise $\sigma_{tr}^2$. All normal distributions belonging to the arms,

$$\mu = \{\mu_1, \mu_2, \cdots, \mu_q\}$$
$$\sigma = \{\sigma_1, \sigma_2, \cdots, \sigma_q\}$$

are initialized to standard values $A$ for the former and $B$ for the latter. The algorithm continues as follows: for each iteration $N$, it starts by drawing a random variable $x_j$ from the normal distribution corresponding to $Arm_j$, denoted as $\mathcal{N}(\mu_j[N], \sigma_j[N])$. When all arms have been pulled, the results are compared, and the index $i$ of the maximum $x_j$ is found. Now the actual $Arm_i$ is pulled, which gives the reward $\tilde{r_i}$. The algorithm is logically partitioned into two distinct parts; a selection part and an update part. Steps 2-6 selects an action and pulls the resulting *Arm*. Steps 7-15 updates the arm-distributions. The selected arm $Arm\ j = Arm\ i$ is updated with the Kalman filter equations

$$\mu_j[N+1] = \frac{(\sigma_j^2[N] + \sigma_{tr}^2) * \tilde{r_i} + \sigma_{ob}^2 * \mu_j[N]}{\sigma_j^2[N] + \sigma_{tr}^2 + \sigma_{ob}^2} \tag{3.9}$$

and

$$\sigma_j^2[N+1] = \frac{(\sigma_j^2[N] + \sigma_{tr}^2) * \sigma_{ob}^2}{\sigma_j^2[N] + \sigma_{tr}^2 + \sigma_{ob}^2} \tag{3.10}$$

such that the reward from pulling $Arm\ i$ is incorporated into the distribution. Here we can see how each $(\mu_j[N], \sigma_j[N])$ tuple is updated to $(\mu_j[N+1], \sigma_j[N+1])$, using the reward $\tilde{r_i}$ as an observation in the Kalman equations.

For all the arms that did not get pulled $Arm\ j \neq Arm\ i$, the algorithm adds uncertainty $\sigma_{tr}^2$ to these arms, as seen in the formulas:

$$\begin{aligned} \mu_j[N+1] &= \mu_j[N] \\ \sigma_j^2[N+1] &= \sigma_j^2[N] + \sigma_{tr}^2 \end{aligned} \tag{3.11}$$

This comes from the fact these arms have not been checked and since their reward distribution is non-stationary the actual reward becomes more uncertain over time. The added uncertainty adds incentive for investigating the unexplored arms.

The derivation of the formulas in 3.9 and 3.10 comes from Kalman filters, working with Gaussian distributions and the fact that Gaussian is self conjugate when considering a Gaussian likelihood function. For complete derivations and examples of the multivariate case, see the chapter on “Probabilistic reasoning over time” in *Artificial Intelligence* [20].

**Algorithm 3.2** KF-MANB

```
Require: Number of bandit arms $q$
Require: Observation noise $\sigma_{ob}^2$
Require: Transition noise $\sigma_{tr}^2$
Ensure: $\mu_1[1] = \mu_2[1] = \cdots = \mu_q[1] = A$
Ensure: $\sigma_1[1] = \sigma_2[1] = \cdots = \sigma_q[1] = B$ #Typically, A can be set to 0, with B being sufficiently large.
1: for $N = 0$ to ... do
2:   for $j = 0$ to $q$ do
3:     draw $x_j$ randomly from the associated normal distribution $f(x_j; \mu_j[N], \sigma_j[N])$ with the parameters $(\mu_j[N], \sigma_j[N])$
4:   end for
5:   pull the Arm $i$ whose drawn value $x_i$ is the largest one:
         $$i = \arg\max_{j \in \{1,\dots,q\}} x_j$$
6:   receive a reward $\tilde{r_i}$ from pulling Arm $i$, and update parameters as follows:
7:   for $j = 0$ to $q$ do
8:     if Arm $i$ then
9:       $\mu_i[N+1] = \frac{(\sigma_i^2[N]+\sigma_{tr}^2)*\tilde{r_i}+\sigma_{ob}^2*\mu_i[N]}{\sigma_i^2[N]+\sigma_{tr}^2+\sigma_{ob}^2}$
10:      $\sigma_i^2[N+1] = \frac{(\sigma_i^2[N]+\sigma_{tr}^2)*\sigma_{ob}^2}{\sigma_i^2[N]+\sigma_{tr}^2+\sigma_{ob}^2}$
11:    else
12:      $\mu_j[N+1] = \mu_j[N]$
13:      $\sigma_j^2[N+1] = \sigma_j^2[N] + \sigma_{tr}^2$
14:    end if
15:  end for
16: end for
```

Algorithm 3.3 is an extended version of the KF-MANB originally proposed in [6], which introduces the concept of distributing the reward between actions, much like a *ripple effect*. This algorithm is the result of a cooperation project between the University of Agder and Agder Energy taking place after the publication of [6], and it has not yet been part of published work.

This version contains two differences from the original KF-MANB used to solve multi-armed bandit problems. In step 8 a distance between the selected *Arm i* and the currently updating *Arm j* is calculated. This distance is multiplied with the dependence noise $\sigma_d^2$ and added to the old observation noise $\sigma_{ob}$, creating a new observation noise $\sigma_{ob}$. This represents the increase in uncertainty when moving away from the arm *Arm i*, the actual arm pulled. In steps 10-11 we see the familiar equations 3.9 and 3.10. The second difference between the algorithms is evident here. All arm-distributions are updated with the Kalman equations, instead of just the distribution belonging to the selected arm.

**Algorithm 3.3** KF-MANB with reward spread

```
Require: Number of bandit arms $q$
Require: Observation noise $\sigma_{ob}^2$
Require: Transition noise $\sigma_{tr}^2$
Require: Dependence noise $\sigma_d^2$
Ensure: $\mu_1[1] = \mu_2[1] = \cdots = \mu_q[1] = A$
Ensure: $\sigma_1[1] = \sigma_2[1] = \cdots = \sigma_q[1] = B$ #Typically, A can be set to 0, with B being sufficiently large.
 1: for $N = 0$ to ... do
 2:   for $j = 0$ to $q$ do
 3:     draw $x_j$ randomly from the associated normal distribution $f(x_j; \mu_j[N], \sigma_j[N])$ with the parameters $(\mu_j[N], \sigma_j[N])$
 4:   end for
 5:   pull the Arm $i$ whose drawn value $x_i$ is the largest one:
          $$i = \arg\max_{j \in \{1,\dots,q\}} x_j$$
 6:   receive a reward $\tilde{r_i}$ from pulling Arm $i$, and update parameters as follows:
 7:   for $j = 0$ to $q$ do
 8:     calculate distance between Arm $j$ and Arm $i$ as dist
 9:     $\sigma_{ob} \leftarrow \sigma_{ob} + dist * \sigma_d$
10:     $\mu_j[N+1] = \frac{(\sigma_j^2[N]+\sigma_{tr}^2)*\tilde{r_i}+\sigma_{ob}^2*\mu_j[N]}{\sigma_j^2[N]+\sigma_{tr}^2+\sigma_{ob}^2}$
11:     $\sigma_j^2[N+1] = \frac{(\sigma_j^2[N]+\sigma_{tr}^2)*\sigma_{ob}^2}{\sigma_j^2[N]+\sigma_{tr}^2+\sigma_{ob}^2}$
12:   end for
13: end for
```

## 3.2 Continuous Genetic Algorithm

"Practical Genetic Algorithms" [1] is used as a basis for implementing the GA-based optimization method. It describes how the continuous genetic algorithm can be applied to optimizations involving real numbers.

The main difference between the continuous and the classic binary genetic algorithms lies in the contents of the chromosomes, also called individuals. In the continuous genetic algorithm, the values constituting the search space are used directly. There is no coding to binary values and decoding afterwards, and therefore it is no longer necessary to consider how many bits are needed to accurately present a value. However, although the values are said to be continuous, the precision of the algorithm is still limited by the internal precision of the computer system. Before any evolution may take place, the user has to define the chromosome (genotype), which forms the layout of the population and the basis for candidate solutions (individuals, phenotypes). It is also critical to define a fitness function, which is used to evaluate the candidate solutions according to the problem domain. In our case any of the objective functions defined in section 2.2 may serve as fitness functions.

### Initialization

The initial population must be generated before any evolution can take place. The population should be a representative sample of the entire solution space, and is normally achieved by randomly generating individuals, within the bounds of the parameter ranges. It's possible to seed the population with individuals containing an educated guess of the parameters.

### Selection

The evolution is driven forward by selecting different candidate solutions (individuals) for breeding and creating new, possibly better, offspring. There exist different strategies for selecting which individuals are allowed to mate. The easiest strategy is to naively select the fittest individuals, breed between them, and create a new generation. Usually, a variation of this is used in practice. A stochastic element is introduced, such that there exists a probability of choosing a lesser fit individual from the mating pool. The reasoning behind this is to avoid premature convergence and to keep the population diversity high. Roulette Wheel selection and Tournament selection are examples of popular and well studied stochastic selection methods. As an example, see this algorithm which describes Tournament selection.

- Choose k individuals from the population at random.
- From the pool of k individuals, select the fittest individual.

Variations of this algorithm may include a decreasing probability of selection, based on pool rank, instead of just picking the fittest individual from the tournament pool.

### Reproduction

After a pool of parents is derived from the selection process, the next step is to recombine individuals into a new generation of candidate solutions. The reproduction operators (genetic operators) are crossovers and mutations, which are methods inspired by biology. Crossovers aim to produce children which inherit their parents' traits and characteristics. Mutations exist to provide genetic diversity, allowing the candidate solutions to explore the search space. Population diversity helps the algorithm avoid local optima. Figure 3.2 shows how a single-point crossover operator uses two parent genes and produces two child genes. Here the stochastic element is the randomly selected crossover point. Another approach, which is especially suited for continuous variables, is the blend crossover. This crossover blends the parameters from the parent genes one by one, as per formula 3.12. Here $p_{new}$ is the new parameter, $\beta$ is a random number between zero and one, $p_m$ is the parameter in the mother gene, and $p_f$ is the parameter in the father gene.

$$p_{new} = \beta p_m + (1 - \beta) p_f \tag{3.12}$$

A simple mutation algorithm is described in the following example.

- Pick a point in the gene.
- Randomly modify with in the bounds (for binary representation, this means flipping a bit).

When using a continuous chromosome, the mutated number is selected within the range bounds of the parameter, at random.

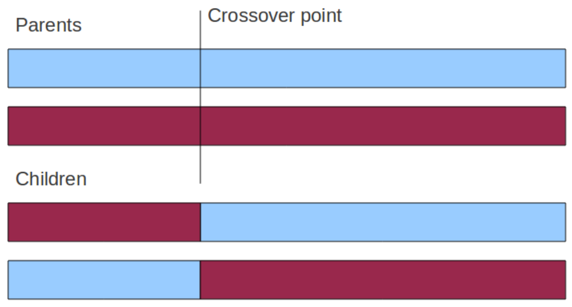


Figure 3.2: Example of a one-point crossover

### Termination

The evolution repeats every generation, until some criteria is met. These criteria may be based on anything measurable, both regarding the population or individuals. It is also possible to combine different termination criteria. Following are some examples:

- Fitness has reached a plateau (there is no more gain between successive generations).
- There exists a solution that satisfies a predefined criterion.
- A fixed number of generations are exceeded.
- Solutions are checked manually.
- Some combination of the above.

### Assembling the parts

The genetic algorithm starts by a definition of the variables constituting the search space, which together form a chromosome, or individual. Next, a cost or fitness function is needed, to be able to calculate the relative value of each individual. The fitness value is the deciding factor on whether an individual is better or worse. Other genetic algorithm parameters also need to be set, such as the size of the population, selection of mating schemes, crossovers, mutators etc. Now the initial population can be generated.

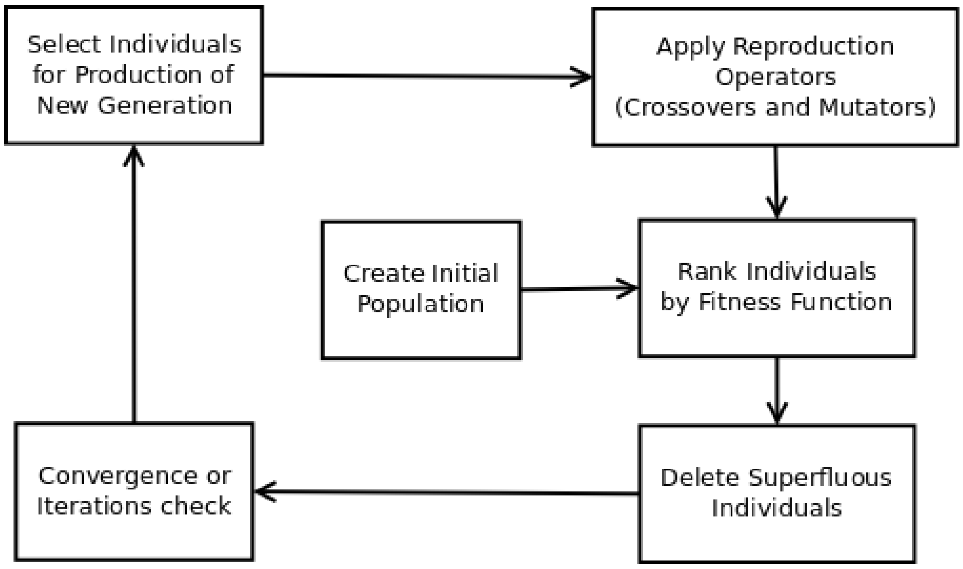

Select Individuals for Production of New Generation
Apply Reproduction Operators (Crossovers and Mutators)
Create Initial Population
Rank Individuals by Fitness Function
Convergence or Iterations check
Delete Superfluous Individuals

Figure 3.3: Overview of the Genetic Algorithm

An overview of the relationship between the different parts of the genetic algorithm is provided in figure 3.3, an the process may be summarized as follows: After creating the initial population, the fitness of each individual is calculated, and this cost determines each individual's rank in the mating scheme. The mating scheme is often called the selection method or simply the selector. Those individuals selected to mate are sent through the mating process, consisting of a crossover and a mutator. The role of the crossover is to produce new individuals based on the traits of their predecessors, while the mutator provides a small chance of introducing new traits to the individuals. The population with the new individuals are ranked again, and the weakest individuals are deleted (killed off) to keep the population size constant. Finally, there may be a convergence check; either to see if the algorithm has reached its goal, or to see whether it has stagnated or is still improving. In some cases, the genetic algorithm is set to run a fixed number of times, and then terminate.

## 3.3 Shuffled Complex Evolution

The Shuffled Complex Evolution (SCE-UA) algorithm was developed by Qingyun Duan as part of his doctorate study at the University of Arizona. It uses a combination of different techniques, such as competitive evolution and directed search, and is both efficient and effective at finding global optimums. A brief explanation of its main concepts and pseudo-code follows. For full explanation, see Duan's paper on the algorithm [7]. A Matlab implementation of the algorithm is located at [22].

The main components of the algorithm is:

**Complexing** All candidate points are divided into $p$ complexes $A^1, \cdots, A^p$ of size $m$, using the formula

$$A^k = \left\{x_j^k, f_j^k | x_j^k = x_{k+p(j-1)}, f_j^k = f_{k+p(j-1)}, j = 1, \cdots, m\right\}$$

**Evolution** Within each complex, parent points are selected to create a simplex and the downhill simplex method is applied to move these points towards the optimal point.

**Shuffling** After the evolution of all complexes, all points from all complexes are collected in a single container, and sorted in increasing order. This ensures that the next time complexes are created, they will not contain the same points as they did the last time, thus avoiding a single complex converging on a local optimum.

In algorithm 3.4, pseudo-code and main structure of the SCE-UA is presented. Step 4 contains the complexing part and the composition of step 3 and 6 constitutes the shuffling. Step 5, which does the evolution of each complex is detailed in a separate algorithm.

**Algorithm 3.4** Shuffled Complex Evolution Method

```
Require: Dimensions $n$
Require: Complexes $p, p \geq 1$
Require: Points in each complex $m, m \geq n+1$
Ensure: Sample size $s = pm$
1: Sample $s$ points from parameter space $\Omega$ and evaluate at each point
2: repeat
3:     $D \leftarrow$ order $s$ by increasing function value
4:     $C \leftarrow$ partition $D$ into $p$ complexes of $m$ points
5:     Evolve each complex $C^k, k = 1, \cdots, p$ (CCE method)
6:     $D \leftarrow$ Gather all points from complexes
7: until Convergence criteria is met
```

The CCE method, presented in algorithm 3.5, does the actual evolution of the points and it's easy to see parallels with concepts presented in section 3.2. All points in a complex are weighted with a triangular probability distribution, to ensure that points with low value (better score/fitness) have a greater chance of being selected as parents. The algorithm selects $q$ parents for the evolution process and these $q$ parents form the simplex used by the downhill simplex method by Nelder and Mead [23]. Step 3 in the CCE method is essentially the downhill simplex method with a small difference. The expansion step is replaced with a mutation step (inspired by genetic algorithms). This means that if reflection and contraction fails, a new point is randomly generated. This point is constrained by the bounds of the complex and the parameter space $\Omega$.

**Algorithm 3.5** Competitive Complex Evolution Method

```
Require: Dimensions $n$
Require: Points in each complex $m$
Require: Reflection $\alpha$ and contraction $\beta$ coefficients
Ensure: Parent size $q, 2 \leq q \leq m$
Ensure: $\alpha \geq 1, \beta \geq 1$
1: Generate triangular probability distributions $\rho$
2: Select $q$ parents according to $\rho$
3: Evolve the worst point by reflection, contraction or mutation
```

# Chapter 4

# Hierarchically structured Kalman Filter Bayesian Learning Automata

The KF-MANB (algorithm 3.3) introduced at the end of section 3.1.4 would ideally be able to calibrate a hydrological model (typically consisting of about 20 or more tunable parameters). However, if each parameter should have a resolution of $n$ actions (or arms), the total number of actions constituting the KF-MANB would, as explained in section 1.1, be $n$ to the power of dimensions (that is, $n$ to the power of the number of parameters subject to calibration). As the KF-MANB needs to explore every action both when selecting which action to perform (see 3.3 line 2) and when distributing rewards throughout the action space (see 3.3 line 7), it becomes clear that the number of actions quickly become to large for any practical purpose.

To alleviate this restriction, several KF-MANBs can be run in parallel, each automaton taking care of 2-3 parameters while maintaining a decent resolution. This is however not an optimal solution, as many automata running in parallel will create a synchronization problem: When the action space is split between several KF-MANBs, they are no longer guaranteed to find the optimal arm/action [5]. Thus, to reduce the number of automata needed to cover the total action space, and thereby reducing the synchronization problem, ways have to be found to more efficiently navigate the action space of each automaton. More efficient methods, both for selecting which actions to perform on the model and for distributing rewards, would allow more parameters (a larger action space) to be put into each automaton. To facilitate improved methods for navigation of each automaton's action space, we explored a hierarchically structured version of the KF-MANB, introduced in the following section.

Some familiarity with binary trees is assumed, such as the concepts of root nodes, leaf nodes, parent nodes, sibling nodes and child nodes. Also, knowledge of basic navigation of binary trees is assumed. This could for example be collecting all leaf nodes, or all parents of the current collection, or navigate from leaf to root, or indeed from leaf to parent to parent's sibling, etc.

## 4.1 Hierarchical structure

To reduce the computational complexity of sampling all actions and updating all actions, we devised a hierarchical layout of the action space. Using a *divide-and-conquer* approach, we are able to sample and update a *path* through the structure (action tree), instead of the entire action space. It also enables easy distribution of reward to other parts of the tree, limiting the need to update every specific node.

The structure is in essence a complete binary tree, with all nodes except the leaf nodes having two child nodes. This means that the tree is balanced, and has the same height for all leaf nodes. Tree height is the number of nodes one has to go through starting from the root node until reaching a leaf node. Each node contains a probability distribution ($\mu$,$\sigma$) which represent the expected reward received when doing that action on the environment. The tree implementation may be done in many different ways, e.g. linked-list type, array type or others. We'll use the array type referencing when talking about concrete nodes, because referencing by index is easier when explaining and has understandable mathematical properties. For example, a left child is always an odd index, a right child always even, a parent always $(index - 1)/2$, etc.

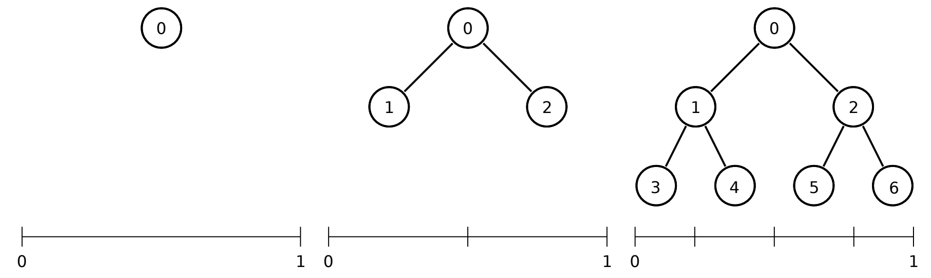
Assuming a complete binary tree, with height 2, the total number of nodes in the tree is $2^{2+1} - 1$. This comes from the generic formula $2^{h+1} - 1$. The number of leaf nodes, nodes at the lowest level of the tree, is $2^h$. The total number of leaf nodes constitutes the action space when dealing with action based learning automata and the discrete model parameter resolution when dealing with model calibration. As such, the number of nodes at the leaf level also needs to be the number of actions to the number of dimensions ($a^d$), giving equation 4.1.

$$\begin{aligned} 2^h &= a^d \\ h &= d * log_2(a) \end{aligned} \tag{4.1}$$

This equation is solved for $h$, showing that the number of actions per dimension should be chosen such that $h$ is an integer. If not, $h$ must be rounded upwards, to ensure the tree is at least large enough for the required number of actions (the actual resolution will then be higher than specified).

All nodes above the leaves are not action nodes in themselves, but rather required paths that lead to an action. This is similar to a decision tree used when illustrating or calculating conditional probabilities.

Since we are dealing with model calibration, it is necessary to look at how the action space maps to parameters and dimensionality. Actions are defined to be on the last level (leaf nodes). This is illustrated in figure 4.1c, with the four nodes (index 3-6) representing action nodes. The actions have to be mapped to the parameter space, such that each action may be used as input to the model calibration routine. The mapping may be done explicitly in the action selection routine, or in the routine that actually writes parameter values and runs the model. As shown by the figures in 4.1, for each level added, the parameter space is divided in two. The example figures assume that the parameter space is one-dimensional, thus showing a line being split into segments. The action tree could also be interpreted as representing a plane, cube or hyper-cube (depending on the number of parameters), and each level in the tree would divide one parameter in two. In any case, the selected action path is constructed of the "winning node" at each level and will be explained in detail in the next section.



(a) Root node (one action) (b) Two actions (c) Four actions

Figure 4.1: Hierarchical structure

## 4.2 Action selection routine

To illustrate the use of our hierarchical structure, let's consider the example shown in figure 4.1. At the top of this binary tree lives the root node. This is the starting point for the action selection algorithm. Following the root node is its children, a left and a right node. The action selection routine is a recursive routine and may be defined as in algorithm 4.1. The first step is to compare the contents of the children of the current node. Two random variables $\{x_l, x_r\}$ are drawn from the corresponding probability distributions located in the left and right node. Based on the size of $x_l$ or $x_r$, a new parent node is selected to be either the left or right node.

**Algorithm 4.1** Action selection routine

```
Require: $Parent$ = RootNode;
Ensure: $L$ = LeftChildNode $R$ = RightChildNode
1: while Parent has children do
2:     draw $x_l$ from $\mathcal{N}(L_\mu, L_\sigma)$
3:     draw $x_r$ from $\mathcal{N}(R_\mu, R_\sigma)$
4:     if $x_l > x_r$ then
5:         Parent is now the left node
6:     else
7:         Parent is now the right node
8:     end if
9: end while
```

The action selection routine loops until the current parent node doesn't have any children left. This happens when the current node is a leaf node. At this point, the algorithm has selected an action node, which may be used to interact with the environment. As can be seen from the algorithm, at each decision point the estimated reward distributions are sampled and the automaton follows the path of highest reward.

All these left or right choices represent a distinct path through the tree. We denote $C$ as a list of left or right choices, giving left the value 0 and right the value 1. To determine which section of the line we're at, we use the following formula $s = C_1\frac{1}{2} + C_2\frac{1}{4} + \cdots + C_n\frac{1}{2^h}$, where $C_n$ denotes the element at position $n$ in list $C$ and $h$ is the "resolution height" of the action-to-dimension space. The upper bound of the section is calculated by adding the section width $\frac{1}{2^h}$ to the lower bound. Thus, if we selected action node 4 from 4.1c this equals $C = [0, 1]$ and by our section formula $\left[s, s + \frac{1}{2^h}\right]$ and corresponds to the section $[0.25, 0.50]$. In cases where the tree is used to represent more than one parameter, each level in the tree can only split one parameter in two. As such, a decision must be made during implementation of the selection routine regarding which levels in the tree belongs to what parameters. One approach could be “interleaved,” where each parameter is split in two before going on to the next level of resolution. Another would be “non-interleaved,” in which the first $h$ levels of the tree corresponds to the first parameter, the next $h$ levels to the second parameter, etc.

## 4.3 Action update routines

After an action has been selected, the action is executed on the environment. In the case of model calibration, this reduces the action to a parameter set and the model is run with the specified parameter set. The model response is compared to the observed response with respect to an objective function. A reward is generated, with high reward given to an action (parameter set) which result in low error in the objective function.

### 4.3.1 The join method

The action selection routine assumes that the probability distribution contained in a node is a reflection of the probability distributions contained in its children nodes. Thus, the action update routines have to address this assumption when distributing rewards in the tree. All the update routines utilize the join method to achieve this effect.

The join method starts by assuming that the distribution in the left child ($X$) is more likely to provide a higher number than the distribution on the right ($Y$). Therefore, $X - Y$ should be greater than zero (see 4.2).

$$\begin{aligned} X \sim \mathcal{N}(\mu_l, \sigma_l^2) > Y &\sim \mathcal{N}(\mu_r, \sigma_r^2) \\ X > Y & \\ X - Y > 0 & \end{aligned} \tag{4.2}$$

The $\mu$ and $\sigma$ values for the distribution $X - Y$ is then calculated as in 4.3, as based on [24], chapter 7.

$$\begin{aligned} \mu &= \mu_l - \mu_r \\ \sigma &= \sqrt{\sigma_l^2 + \sigma_r^2} \end{aligned} \tag{4.3}$$

Next, the new $\mu$ and $\sigma$ values are used in the cumulative distribution function, as shown in 4.4, to obtain the probability that distribution $X$ in fact delivers a higher number than distribution $Y$. The probability that $Y$ provides the larger number is then simply $1 - \Pr(x \geq y)$.

$$\Pr(x \geq y) = F(0;\, \mu, \sigma^2) = \Phi\Big(\frac{0-\mu}{\sigma}\Big) \tag{4.4}$$

Finally, the new distribution $Z$, with its $\mu_{new}$ and $\sigma_{new}$, is calculated as per 4.5.

$$\begin{aligned} Z &= P_{x\geq y} * X + (1 - P_{x\geq y}) * Y \\ \mu_{new} &= P_{x\geq y} * \mu_l + (1 - P_{x\geq y}) * \mu_r \\ \sigma_{new} &= \sqrt{(P_{x\geq y} * \sigma_l)^2 + ((1 - P_{x\geq y}) * \sigma_r)^2} \end{aligned} \tag{4.5}$$

All these equations are compacted and summarized in algorithm 4.2.

**Algorithm 4.2** Node join method

```
Require: $Left$ = LeftNode, $Right$ = RightNode
Ensure: $\mu_l, \sigma_l$ from $Left$
Ensure: $\mu_r, \sigma_r$ from $Right$
1: $\mu_t \leftarrow \mu_l - \mu_r$
2: $\sigma_t \leftarrow \sqrt{\sigma_l^2 + \sigma_r^2}$
3: $t \leftarrow (0 - \mu_t)/\sigma_t$
4: $Pr(x \geq y) = \Phi(t)$
5: $\mu_n \leftarrow P_{x\geq y} * \mu_l + (1 - P_{x\geq y}) * \mu_r$
6: $\sigma_n \leftarrow \sqrt{(P_{x\geq y} * \sigma_l)^2 + ((1 - P_{x\geq y}) * \sigma_r)^2}$
7: return $\mu_n, \sigma_n$
```

### 4.3.2 Reward distribution schemes

Of the possible reward distribution schemes we've explored, we'll first look into the simplest of them, namely the Lazy action update routine. Then follows more and more involved routines, ending with the Complete action update routine.

With the reward in hand, it's time to update the hierarchy with the new information. As expected, the first step is to use the Kalman filter equations on the selected action and with the reward as the "observation". In the pseudo-code, this step is called KalmanUpdate. The concept of distance (introduced in the next reward distribution scheme) is not needed for this algorithm, as all updates are along the selected action path (meaning distance is zero), or a sibling to a path node (meaning distance is one). The next task is to propagate the reward update throughout the affected action path, starting from the selected node's sibling node. After the sibling node has been updated, both the selected node and the sibling node values are united with the join method, the result being stored in their parent node. After the parent node is updated, the selected node pointer is changed to point to the parent node and the loop repeats until there are no more parents available. This happens when the parent node reaches the root node of the hierarchy. Pseudo-code for this algorithm is presented in algorithm 4.3.

**Algorithm 4.3** "Lazy" action update routine

```
Require: $Main$ = SelectedNode
Require: $\tilde{r}$ = Reward
Ensure: $P$ = FindParent($Main$)
Ensure: $S$ = FindSibling($Main$)
1: KalmanUpdate($Main$, $\tilde{r}$, 0)
2: while $Main$ has $P$ do
3:     $S \leftarrow$ FindSibling($Main$)
4:     KalmanUpdate($S$, $\tilde{r}$, 1)
5:     $P \leftarrow$ FindParent($Main$)
6:     $P_\mu, P_\sigma \leftarrow$ Join($Main$, $S$)
7:     $Main \leftarrow P$
8: end while
```

The main point of the Lazy method is to only update affected nodes along the selected path through the tree. The first extension of this concept is to spread the reward information a little further, namely to the nearest neighbours of the selected action node. The Nearest Neighbours (NN) action update routine is presented in algorithm 4.4. It requires specification of the neighbourhood distance $k$, which is a number for how far the reward should be spread. This algorithm also introduces the concept of *distance*. Distance is used to "rebate" the reward, or to add uncertainty when updating neighbour nodes with an actual reward. Since the neighbours didn't actually execute their action on the environment, one has to add some uncertainty to the reward (observation).

The NN action update method starts out by locating the neighbours within a certain neighbourhood (geometric range dependent on the number of parameters represented by the tree). Once all the nodes which fall within this range have been identified, they are updated with the KalmanUpdate method. The equations use the reward as observation and a Distance factor as the distance between the selected node and the specific neighbour node. After all the neighbours[1] are updated, the algorithm continues in the same fashion as the Lazy method.

**Algorithm 4.4** "Nearest Neighbours" action update routine

```
Require: $Main$ = SelectedNode
Require: $\tilde{r}$ = Reward
Require: $k$ = Neighbour distance
Ensure: $P$ = FindParent($Main$)
Ensure: $S$ = FindSibling($Main$)
 1: $NN \leftarrow$ LocateNeighbours($Main$,$k$)
 2: for all $Node$ in $NN$ do
 3:     $D \leftarrow$ Distance($Main$, $Node$)
 4:     KalmanUpdate($Node$, $\tilde{r}$, D)
 5: end for
 6: while $Main$ has $P$ do
 7:     $S \leftarrow$ FindSibling($Main$)
 8:     KalmanUpdate($S$, $\tilde{r}$)
 9:     $P \leftarrow$ FindParent($Main$)
10:     $P_\mu, P_\sigma \leftarrow$ Join($Main$, $S$)
11:     $Main \leftarrow P$
12: end while
```

The NN algorithm only propagates information upwards in the tree along the selected action path, like the lazy algorithm. All the neighbourhood nodes receiving reward updates are not subjected to this process. The Nearest Neighbours Complete, or NN-C, algorithm 4.5 is looking to mitigate this inaccuracy by applying the join method to all affected nodes. This is achieved by locating the parents of the affected nodes, and recursively joining them until reaching the root node of the tree. The part of the algorithm distributing rewards to the neighbourhood nodes works as in the NN algorithm.

[1]The sibling node is *not* counted as a neighbour in this scheme. This is to avoid a double Kalman reward update on the sibling, one in the neighbour update and on in the join update.

**Algorithm 4.5** "Nearest Neighbours Complete" action update routine

```
Require: $Main$ = SelectedNode
Require: $\tilde{r}$ = Reward
Require: $k$ = Neighbour distance
 1: $NN \leftarrow$ LocateNeighbours($Main$,$k$)
 2: for all $Node$ in $NN$ do
 3:     $D \leftarrow$ Distance($Main$, $Node$)
 4:     KalmanUpdate($Node$, $\tilde{r}$, D)
 5: end for
 6: $P \leftarrow$ LocateParents($NN$)
 7: while $P$ not empty do
 8:     for all $Node$ in $P$ do
 9:         $Leftchild, Rightchild \leftarrow$ LocateChildren($Node$)
10:         $Node_\mu, Node_\sigma \leftarrow$ Join($Leftchild$, $Rightchild$)
11:     end for
12:     $P \leftarrow$ LocateParents($P$)
13: end while
```

The last reward distribution scheme is the Complete Algorithm 4.6, which spreads reward to all the leaf nodes and joins all the parent nodes recursively up to the root node in the tree. This scheme was implemented to mimic the behaviour of the reward distribution in the KF-MANB (algorithm 3.3), which updates all the actions in each iteration.

**Algorithm 4.6** "Complete" action update routine

```
Require: $Main$ = SelectedNode
Require: $\tilde{r}$ = Reward
Require: $Nodes$ = All leaf nodes
 1: for all $Node$ in $Nodes$ do
 2:     $D \leftarrow$ Distance($Main$, $Node$)
 3:     KalmanUpdate($Node$, $\tilde{r}$, Distance)
 4: end for
 5: $P \leftarrow$ LocateParents($Nodes$)
 6: while $P$ not empty do
 7:     for all $Node$ in $P$ do
 8:         $Leftchild, Rightchild \leftarrow$ LocateChildren($Node$)
 9:         $Node_\mu, Node_\sigma \leftarrow$ Join($Leftchild$, $Rightchild$)
10:     end for
11:     $P \leftarrow$ LocateParents($P$)
12: end while
```

# Chapter 5

# Evaluation of algorithms

In this chapter the different Bayesian learning automata (BLA) are compared to each other, and the most promising approach is compared to other established optimization techniques. The first section describes how the different algorithms were tested. It introduces the model implementation used in the tests, along with the model parameters subject to calibration, as well as the catchment data set used with the model. It also summarizes the parameter settings of the evaluated algorithms, which are not to be confused with the parameters belonging to the model. The second section presents the results obtained using the standard deviation, or root mean square error, as the objective function with the BLA candidates. Similar results using the Lindström coefficient as the objective function are presented in the third section. In the last section the most promising BLA candidate is compared to other optimization techniques, using both the Lindström coefficient and the root mean square error as objective functions.

## 5.1 Test setup

The environment for testing of the different optimization algorithms is set up according to the strategy for model calibration described in chapter two. The model subject to calibration is in this case Agder Energy's implementation of the HBV model. As example of a real world system, they supplied us with a data set of observations from the Skjerka catchment.

The model parameters subject to calibration are listed in table B.1 i appendix B. Although some parameters can be calculated based on the geography of the actual catchment, we decided to calibrate as many parameters as possible, as this thesis focuses on optimization algorithms, and not on hydrology.

All the measurements in the data set are assumed to be correct, meaning that differences between the modelled and measured response are interpreted as inaccuracies or errors in the model. The data set contains about nine years of hydrological observations, and includes daily measurements of precipitation ($mm$), temperature ($°C$) and runoff ($m^3/s$). The first two are used as input values by the model, while the last is compared with the runoff response calculated by the model.

The comparison between calculated and measured runoff is handled by the objective function. We tested the optimization algorithms with two different objective functions: The standard deviation, or root mean square error (rmse), and the Lindström coefficient, both introduced in section 2.2. As the BLAs seemed to be sensitive to the ranges of the objective functions, we employed natural logarithms to transform the objective function values into scores ranging from zero to one. For the rmse function, which ranges from $\infty$ to 0, we used $e^{-rmse/50}$ as a transformation function. The Lindström coefficient, ranging from $-\infty$ to 1 was transformed with $e^{(lindstrom-1)}$. Additionally, the $w$ in the Lindström coefficient is set to 0.1, as recommended in [9].

The different optimization algorithms are summarized in table 5.1, and they will henceforth be referred to by their abbreviations. Each algorithm is tested 30 times, and due to limited amounts of processing power, each run is limited to 10000 model executions.

Table 5.1: Summary of algorithms and their abbreviations

| **Algorithm** | **Abbreviation** | **Reference** |
|---|---|---|
| Continuous Genetic Algorithm | GA | Section 3.2 |
| Shuffled Complex Evolution | SCE-UA | Section 3.3 |
| KF-BLA family | | |
| Flat with Reward Distribution (RD) | Flat | Algorithm 3.3 |
| Lazy RD | Lazy | Algorithm 4.3 |
| Nearest neighbour RD | NN | Algorithm 4.4 |
| Nn complete RD | NN-C | Algorithm 4.5 |
| Complete RD | Complete | Algorithm 4.6 |

All the hierarchically structured BLAs are set up with four dimensions (meaning four parameters) and 16 actions per dimension except the Complete version, which we had to limit to two dimensions, keeping the resolution of 16 actions (else we wouldn't have had the results in time). The Flat BLA, being the product of an earlier project, was left unaltered with two dimensions and 50 actions per dimension. The flat and NN versions can be set up with as much as 8 dimensions per BLA, but time did not allow testing of this setting.

All the above mentioned algorithms use the Kalman Filter, and we decided to use the same parameter settings for all of them. As such, the observation sigma, transition sigma and dependence sigma are set to 0.1, 0.0001 and 0.25, respectively. Initial mu and sigma values per action are set to 0 and 100. The NN and NN-C distributes rewards to the nearest neighbours of the chosen action, and the range was set to +-2 neighbours per dimension (giving a pool of $5^{dimensions}$ actions to update). All these values were set based on pre-experiment trials.

The GA was set up with a population size of 50, running for 200 generations, thus giving a total of 10000 model executions. A tournament selector with a pool size of two drove the selection process, followed by a blend crossover with crossover rate 90% and a uniform mutator with mutation rate 2%. The SCE-UA was set up with two complexes, and with a cutoff as close to 10000 model executions as possible. The algorithm settings are summarized in table 5.2.

Table 5.2: Summary of algorithm parameters used for test setups

| **Algorithm** | **Settings** |
|---|---|
| Shuffled Complex Evolution | Complexes: 2 |
| Continuous Genetic Algorithm | Blend crossover (90%)<br>Tournament selector (2 individuals)<br>Uniform mutator (2%) |
| KF-BLA family | Observation sigma ($\sigma^2_{ob}$) 0.1<br>Transition sigma ($\sigma^2_{tr}$) 0.0001<br>Dependence sigma ($\sigma^2_{d}$) 0.25<br>Initial sigma 100<br>Initial mu 0<br>Actions 16 (50 with flat)<br>Dimensions 4 (2 with complete and flat) |

## 5.2 Standard deviation tests

Following is a presentation of the results from the tests using the root mean square error (rmse) as objective function. Statistical data from the these tests are found in table 5.3, and are visualized in figure 5.1. A plot of the average development of the test runs is presented in figure 5.2.

In table 5.3, the results of the 30 test runs are summarized per algorithm with values for maximum, minimum and average results, as well as range (difference between maximum and minimum) and standard deviation (calculated assuming the 30 runs are samples of a larger population). As can be seen in the table, the Lazy algorithm is clearly the best in all these aspects, the NN algorithm taking second place. With minimizing the result being the objective, the Complete version comes in slightly ahead of the NN-C: The minimum and average results are better, but maximum result, range and standard deviation are worse. Last is the Flat version, with the worst performance in both minimum and average results. However, it is better than both the Complete and the NN-C versions with respect to range and standard deviation.

Table 5.3: Statistics on final score in rmse tests

| Algorithm | Minimum | Maximum | Range | Average | Std.Dev. |
|---|---|---|---|---|---|
| Lazy | 12.17 | 13.67 | 1.50 | 12.97 | 1.06 |
| NN | 12.49 | 15.31 | 2.82 | 13.60 | 1.99 |
| Complete | 12.76 | 18.84 | 6.08 | 14.49 | 4.30 |
| NN-C | 12.79 | 17.40 | 4.61 | 15.33 | 3.26 |
| Flat | 15.26 | 18.64 | 3.38 | 16.95 | 2.39 |

The statistical data is visualized in figure 5.1. Each algorithm has two columns, the range column showing minimum, maximum and average results. The stddev column shows the average results plus and minus the standard deviation. The most interesting observation seen in this figure is that the average for the Complete algorithm is closer to the minimum than the maximum, suggestion an uneven distribution of results in the range. The results for the other algorithms are more equally distributed. Also of interest is that, despite it's poor performance, the Flat version is quite a lot more consistent than the Complete and NN-C versions.

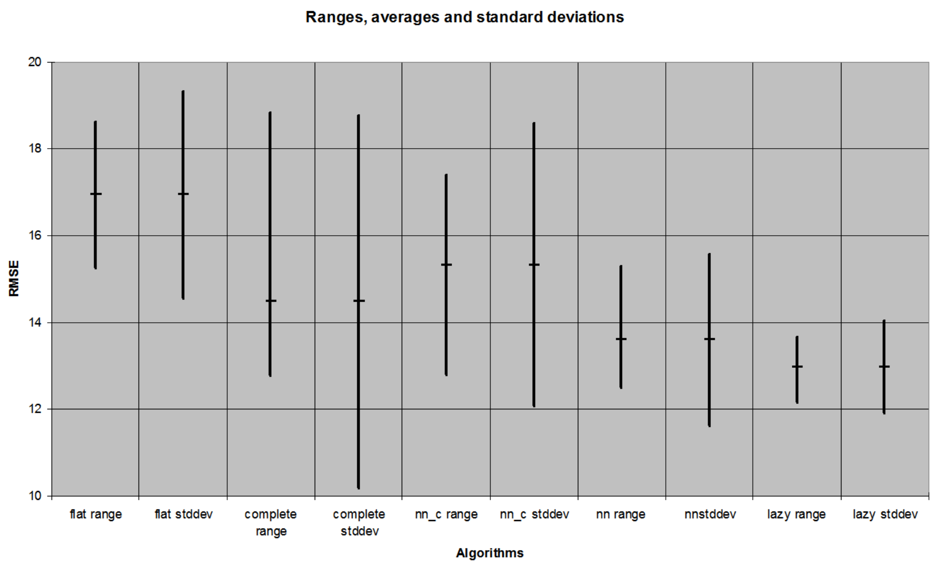


Figure 5.1: Statistics visualization for rmse tests

Figure 5.2 contains plots of the average development of each algorithm during the test runs. The plots show that for the first 1500-1600 iterations, the Complete algorithm performs better than the other candidates. However, it stagnates/converges pretty fast after that, and after about 2500 iterations the order of the candidates are the same as in the final results.

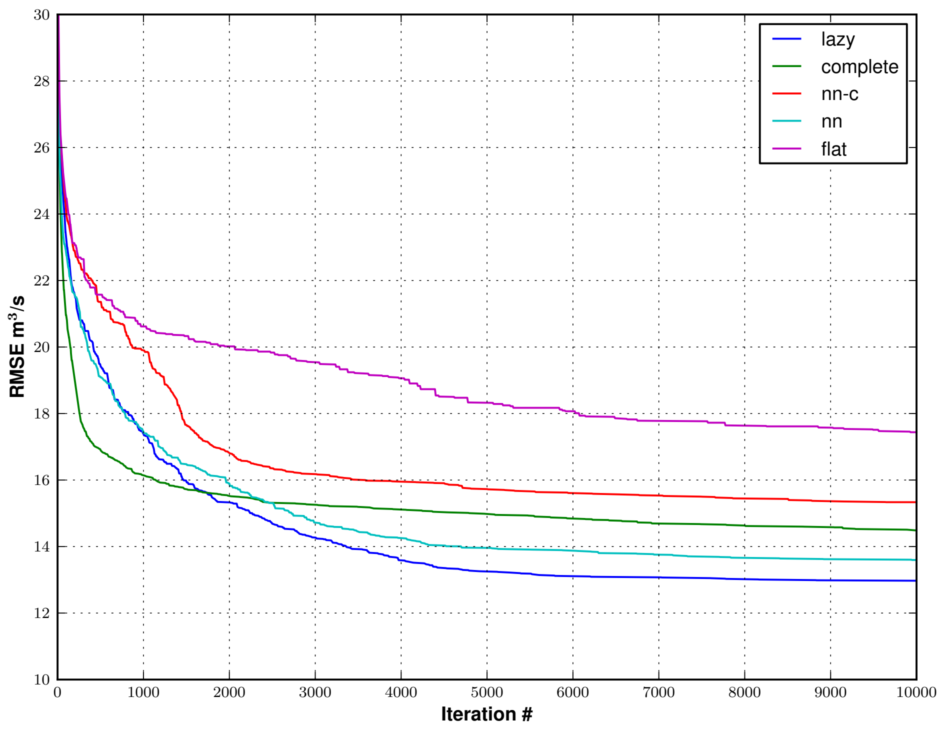


Figure 5.2: Development of rmse per BLA

## 5.3 Lindström tests

Next up is a presentation of the results of the tests using the Lindström coefficient as objective function. Statistical data from these tests are located in table 5.4, and are visualized in figure 5.3. A plot of the average development of the test runs is presented in figure 5.4.

In table 5.4, the results of the 30 test runs are summarized per algorithm in the same way as for the rmse results in the previous subsection. Thus the table contains values for maximum, minimum and average results, as well as range (difference between maximum and minimum) and standard deviation (calculated assuming the 30 runs are samples of a larger population). The obvious difference is that now we're after maximizing the result instead of minimizing it. As was the case with the rmse tests, the Lazy algorithm comes out on top in all categories, the NN version coming in second. This time however, the NN-C comes in ahead of the Complete algorithm, as it is better in both maximum and average results. The Flat version comes in last again, and again it is better than the Complete and NN-C in terms of range and standard deviation. Surprisingly, it also beats the Lazy and NN algorithm in these respects.

Table 5.4: Statistics on final score in Lindström tests

| Algorithm | Maximum | Minimum | Range | Average | Std.Dev. |
|---|---|---|---|---|---|
| Lazy | 0.8412 | 0.7926 | 0.0486 | 0.8191 | 0.0344 |
| NN | 0.8290 | 0.7754 | 0.0536 | 0.8073 | 0.0378 |
| NN-C | 0.8207 | 0.4944 | 0.3263 | 0.7486 | 0.2308 |
| Complete | 0.8089 | 0.5023 | 0.3066 | 0.7452 | 0.2168 |
| Flat | 0.7032 | 0.6938 | 0.0094 | 0.6985 | 0.0066 |

As with the rmse results, the data from table 5.4 is visualized in figure 5.3 with two columns per algorithm. The range column shows average, minimum and maximum results, while the stddev column show average results plus and minus the standard deviation. This time both the Complete and NN-C versions have averages lying closer to the maximum than the minimum result, suggesting that they are probably not as unstable as it might seem. The other surprise here is how incredibly stable the Flat version is, showing very small variations in the results.

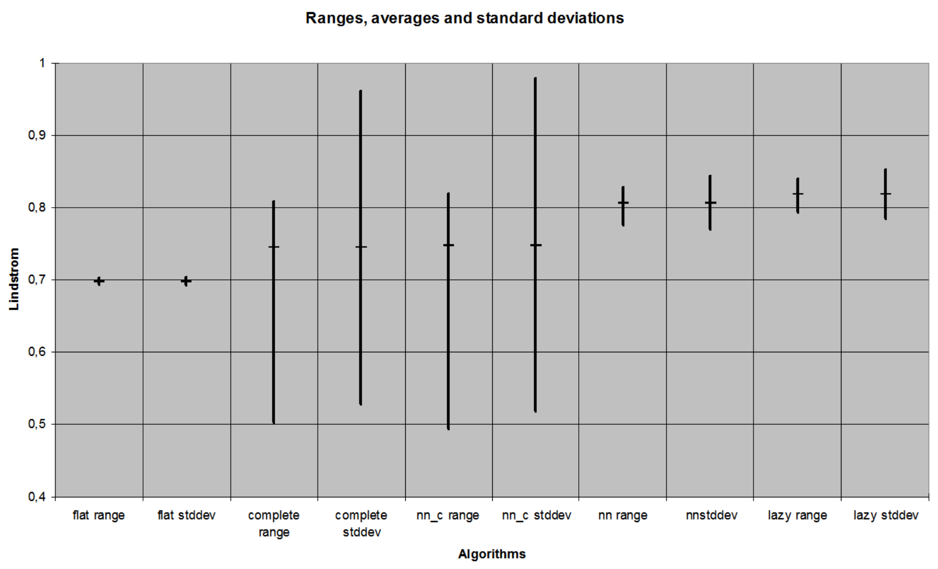


Figure 5.3: Statistics visualization for Lindström tests

Finally, in figure 5.4 the average development of each algorithm during the test runs is plotted. With this objective function, the Lazy and NN versions stay in the lead pretty much from start to end. The Complete version again performs very well in the beginning, but is overtaken by the NN-C around iteration 1100-1200. The Complete algorithm seems to be catching up towards the end, but ends slightly behind the NN-C.

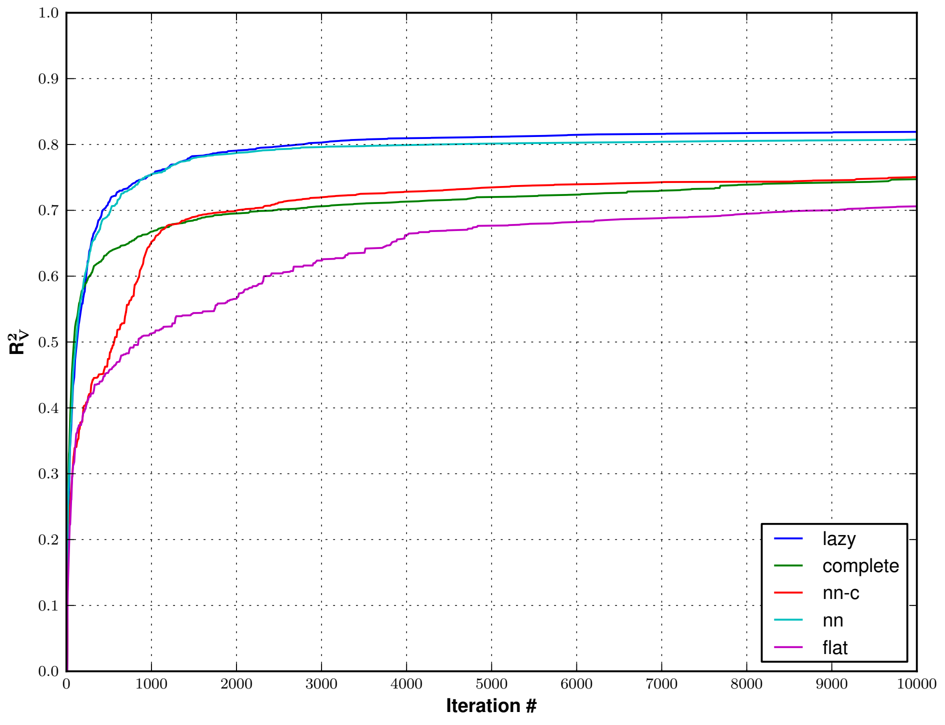


Figure 5.4: Development of Lindström coefficient per BLA

## 5.4 Comparison with other techniques

The last part of this evaluation contains comparisons of the HKF-BLA approach to other calibration techniques. Specifically, the most promising BLA candidate, the Lazy algorithm, is compared to the continuous genetic algorithm (GA) and the shuffled complex evolution (SCE-UA) algorithm.

### Standard deviation tests

Comparisons with the standard deviation as objective function is summarized in table 5.5, and visualized in figure 5.5. A plot of the average development in score is presented in figure 5.6.

Table 5.5 contains maximum, minimum and average scores from the 30 test runs, as well as values for range and standard deviation, calculated as in the previous sections. As can be seen in the table, the SCE-UA algorithm emerge as the best alternative in all categories. The GA narrowly beats the Lazy algorithm, having a smaller minimum score and a slightly smaller average. The Lazy however, seems to obtain more focused results: The maximum score, as well as range and standard deviation is better than with the GA.

Table 5.5: Comparison of final score in rmse tests

| Algorithm | Minimum | Maximum | Range | Average | Std.Dev. |
|---|---|---|---|---|---|
| SCE-UA | 11.54 | 12.85 | 1.31 | 12.20 | 0.93 |
| GA | 12.00 | 13.89 | 1.89 | 12.95 | 1.34 |
| Lazy | 12.17 | 13.67 | 1.50 | 12.97 | 1.06 |

The visualization of the test data in figure 5.5 shows how close the GA and the Lazy algorithm are in performance. As in the previous sections we have two columns per algorithm. The range column contains average, maximum and minimum scores, while the stddev column shows average scores plus and minus the standard deviation.

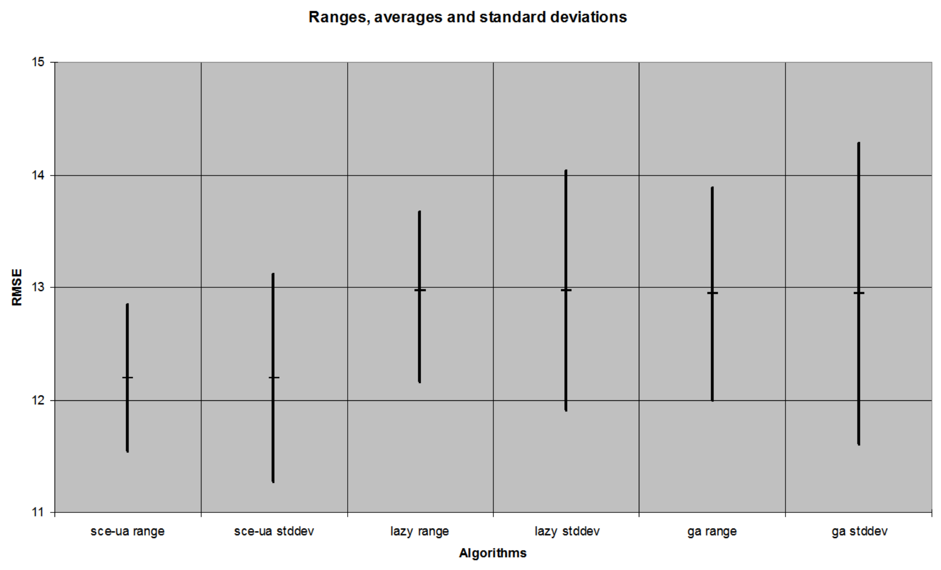


Figure 5.5: Statistics visualization for rmse comparisons

Figure 5.6 shows the average development of each algorithm during the tests runs. The SCE-UA takes the lead after about 1000 iterations. The Lazy algorithm and the GA follow each other pretty much the whole time, taking turns being in the lead.

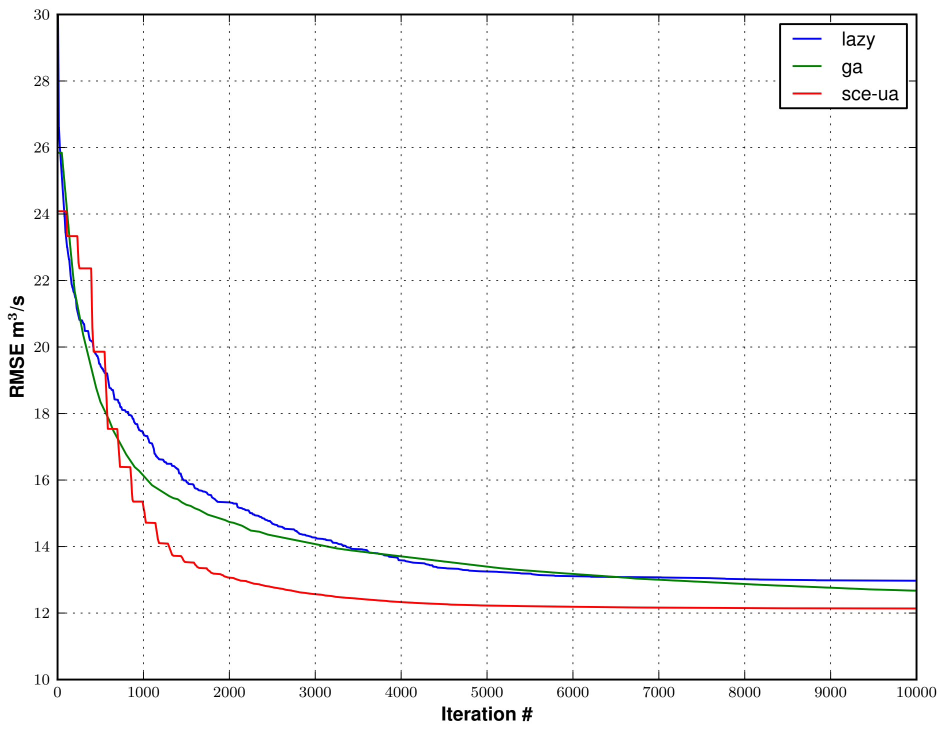


Figure 5.6: Comparison of rmse development

### Lindström coefficient tests

Table 5.6 shows statistical data from the comparison tests using the Lindström coefficient as objective function, the data being visualized in figure 5.7. The average development of the score is plotted in figure 5.8.

Similar to the previous tests, table 5.6 contains values for maximum, minimum and average scores from the 30 test runs. Range and standard deviation are calculated as before. As was the case with the rmse tests, the SCE-UA displays the best performance. This time, however, the Lazy algorithm comes out ahead of the GA. The maximum and average scores are better, while minimum, range and standard deviation are slightly worse. This is exactly the opposite of the situation in the rmse test.

The visualization of test data in figure 5.7 is similar to the visualization of the standard deviation tests, with two columns per algorithm. The range column shows average, maximum and minimum scores, and the stddev column shows average score plus and minus the standard

Table 5.6: Comparison of final score in Lindström tests

| Algorithm | Maximum | Minimum | Range | Average | Std.Dev. |
|---|---|---|---|---|---|
| SCE-UA | 0.8561 | 0.8264 | 0.0297 | 0.8412 | 0.0210 |
| Lazy | 0.8412 | 0.7926 | 0.0486 | 0.8191 | 0.0344 |
| GA | 0.8363 | 0.7964 | 0.0399 | 0.8163 | 0.0282 |

deviation. Again the Lazy algorithm and the GA are very close in performance, with the SCE-UA a little in front. This time the Lazy algorithm produces both higher and lower results than the GA, while the opposite was the case with the rmse tests.

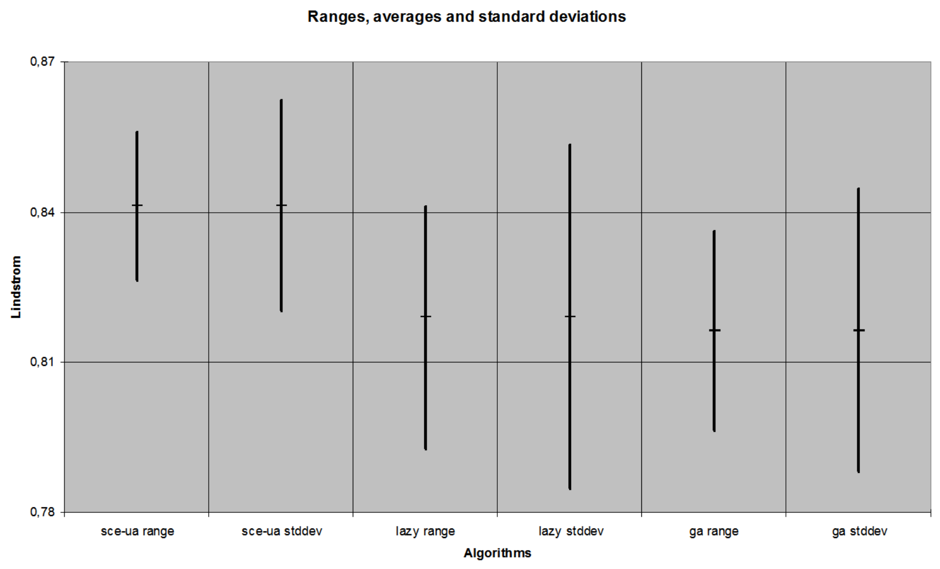


Figure 5.7: Statistics visualization for Lindström comparisons

Figure 5.8 shows the average development of each algorithm over the test runs, and similar to the rmse tests, the SCE-UA takes the lead after about 1000 iterations. This time, however, the Lazy algorithm stays in front of the GA during the entire development period, although the GA seems to catch up towards the end.

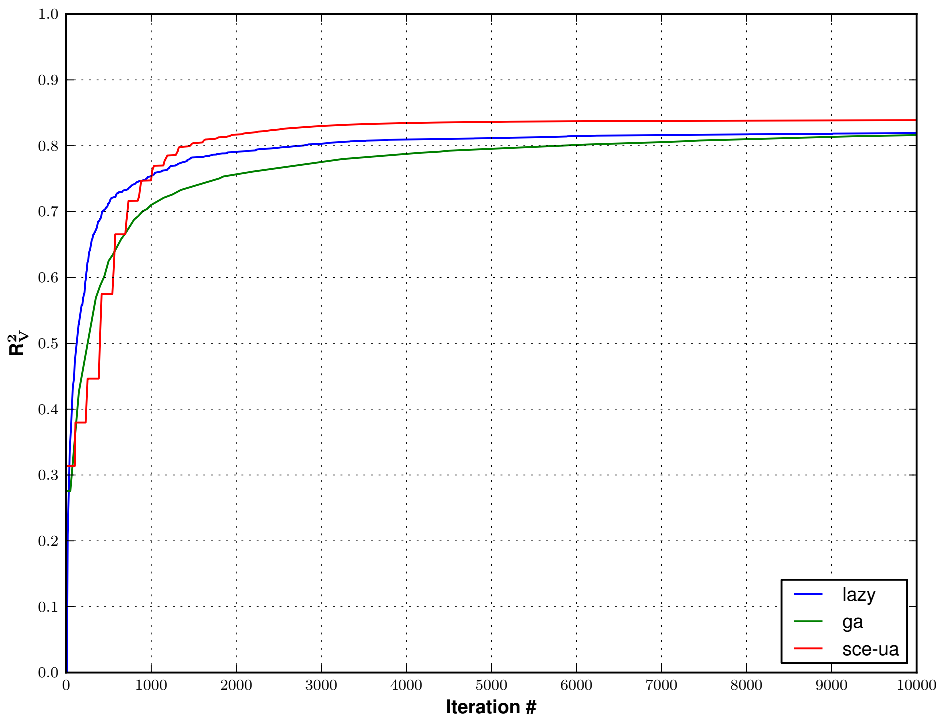


Figure 5.8: Comparison of Lindström development

# Chapter 6

# Discussion

The results presented in the previous chapter show that on average, all the hierarchically structured learning automata perform better than the Flat algorithm (in terms of providing a best, or most accurate, model estimate). The best HKF-BLA solution performs equally well as the genetic algorithm, but comes short of the performance of the SCE-UA algorithm.

Regarding the computational efficiency of the proposed solutions (measured as the number of model executions), our results show that the Lazy algorithm follows the same rate of improvement as the genetic algorithm and the SCE-UA. The NN algorithm shows about the same rate of improvement as the Lazy, while the NN-C is notably worse. The results for the Complete version are inconclusive. It improves fast in the rmse tests, but fails to do so in the Lindström tests.

The pattern seems to repeat itself when looking into the variance of the algorithms. The Lazy algorithm, and to a certain degree the NN algorithm, are both about as stable as the genetic algorithm, the SCE-UA showing a little less variance. The NN-C and the Complete algorithms delivers notably less stable results than all the others.

Of course, these results are only valid with the settings used in the experiment. A different model, or a different data set, or indeed different algorithm parameter settings might yield different results. The Lazy version of the HKF-BLA seems to be the most promising candidate solution, but more experiments are needed to verify this.

In general, when considering the HKF-BLAs, it seems that the results get worse the further the reward is distributed. This could be due to unforeseen inaccuracies in reward distribution when the distance between actions is large, or some kind of cascading effect that goes unnoticed when keeping reward distribution at a minimum. To determine whether this might be the case, we could either perform a mathematical analysis of the reward distribution routines, build a more sophisticated test environment, or both.

Based on our experiences with the KF-BLA family, we believe that several of their parameters are sensitive to properties of the explored environment. For example, it seems to us that the observation sigma needs to be set in relation to the value range of the objective function. Also, it seems the transition sigma must be set decades lower than the observation sigma, else the automata will explore randomly and never converge.

Another consideration is that the KF-BLAs are designed to keep optimizing for an unlimited number of iterations. They could also have been designed to optimize for a fixed number of iterations, set with an iterations parameter. This could provide a different way to adjust the explore/exploit balance, which is now controlled by the uncertainty parameters.

All things considered, we succeeded in reducing the required number of agents (down to 3-6 with the Lazy algorithm) as opposed to 8-12 with the KF-MANB. To achieve this improvement, we reduced the number of actions you need to sample before making a decision: In the KF-MANB, all actions had to be sampled, while now only about $log_2$ of the action space needs to be sampled.

Regarding reward distribution, the Complete version has twice the calculations compared to the KF-MANB (it has double the action space), while the Lazy algorithm on the other hand has about $2 * log_2$ of action space calculations. Thus both the select and the update routines can be said to scale logarithmically rather than linearly with the size of the action space, which is what we were hoping to accomplish with this project.

# Chapter 7

# Conclusion

Current agent implementations used for solving the multi-armed bandit problem struggle with covering huge action spaces, typically experienced when optimizing high-dimensional functions (such as when calibrating hydrological models). This is because the number of actions grow exponentially rather than linearly per added dimension. Current implementations therefore needs to be limited to 2-3 dimensions per agent, meaning you need to coordinate many agents running in parallel. This limitation is not really connected with the size of the action space, but rather with the fact that the current implementation needs to examine every action at every iteration both when selecting actions and updating action probabilities.

We therefore focused on reducing the number of agents needed to cover the total action space, in this case achieved by hierarchically structuring each agent's part of the action space. The number of actions needed to be explored for action selection is now growing logarithmically with the size of the tree, rather than linearly. Additionally, different methods of spreading information throughout the structure was explored, from total reward distribution as in the extended KF-MANB (Kalman filter based multi-armed normal bandit algorithm, see section 3.1.4 and [6]) to very simplistic in only spreading reward along the selected action path.

The hierarchically structured Kalman filter Bayesian learning automaton, or HKF-BLA (see chapter 4), is capable of dealing with much larger action spaces than the KF-MANB. This is not because the the number of actions per agent is smaller (actually it's twice the size), but rather because of the way the structure is navigated, both when selecting actions and when updating/spreading rewards throughout the tree. For example, the KF-MANB samples each action before selecting one to execute on the environment, while the HKF-BLA eliminates half the tree at each sampling point, making action selection a lot faster.

For model calibration, HKF-BLA is a big step forward compared to the KF-MANB, but it is not yet capable of outperforming the state of the art, the SCE-UA type algorithms. Still, it is capable of dealing with much larger action spaces than the current KF-MANB, and should therefore be able to solve more complex problems.

## Future work

Although we have made navigation of the action space much more efficient, we have met a limit regarding the size of the action space itself. It would therefore be interesting to explore methods for reducing the size of the action space. This could for example be done by reducing the number of actions per dimension and using the extended Kalman filter to provide multiple tops per action cell.

It would also be interesting to combine the HKF-BLA with a directed search algorithm (as is done in the SCE-UA with the downhill simplex method), to find the optimum value in each area the HKF-BLA finds to be promising. Another interesting idea from the SCE-UA would be to introduce complexes to the HKF-BLA, to be able to both explore and exploit at the same time.

Regarding the HKF-BLA itself, more empirical experiments are needed to further clarify the relations between its different parameters. Of particular interest would be to in more detail chart the relationship between the $\mu$, $\sigma_{obs}$ and $\sigma_{tr}$.

Perhaps the most interesting attribute of the HKF-BLA, is its ability to chart the action space. In contrast to the evolutionary approaches, which only produces a best parameter set, the HKF-BLA stores estimates of the expected performance for each action cell in the tree structure (which of course could be saved for future use). This information could be used to locate regions of attraction that should be explored further. Also, it should be much faster to produce new model estimates when new sets of measurements from the modelled catchment becomes available.

# Bibliography

[1] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms, Second Edition.* Wiley-Blackwell, 2004.

[2] H. V. Gupta, K. J. Beven, and T. Wagener, *Model Calibration and Uncertainty Estimation.* John Wiley & Sons, Ltd, 2006.

[3] S. Bergström, *Development and application of a conceptual runoff model for Scandinavian catchments.* Sveriges Meteorologiska och Hydrologiska Institut, 1976.

[4] S. Bergström, *The HBV model: Its structure and applications*, vol. 4. Sveriges Meteorologiska och Hydrologiska Institut, 1992.

[5] O.-C. Granmo, "A Bayesian Learning Automaton for Solving Two-Armed Bernoulli Bandit Problems," in *Proceedings of the 2008 Seventh International Conference on Machine Learning and Applications*, (Washington, DC, USA), pp. 23–30, IEEE Computer Society, 2008.

[6] O.-C. Granmo and S. Berg, "Solving Non-Stationary Bandit Problems by Random Sampling from Sibling Kalman Filters," in *Trends in Applied Intelligent Systems* (N. García-Pedrajas, F. Herrera, C. Fyfe, J. Benítez, and M. Ali, eds.), vol. 6098 of *Lecture Notes in Computer Science*, pp. 199–208, Springer Berlin / Heidelberg, 2010.

[7] Q. Duan, V. Gupta, and S. Sorooshian, "Shuffled complex evolution approach for effective and efficient global minimization," *Journal of optimization theory and applications*, vol. 76, no. 3, pp. 501–521, 1993.

[8] J. A. Vrugt, H. V. Gupta, W. Bouten, and S. Sorooshian, "A Shuffled Complex Evolution Metropolis algorithm for optimization and uncertainty assessment of hydrologic model parameters," *Water Resour. Res.*, vol. 39, pp. 1201–, Aug. 2003.

[9] G. Lindström, "A Simple Automatic Calibration Routine for the HBV Model," *Nordic Hydrology*, vol. 28, no. 3, pp. 153–168, 1997.

[10] P. Yapo, H. Gupta, and S. Sorooshian, "Automatic calibration of conceptual rainfall-runoff models: sensitivity to calibration data," *Journal of Hydrology*, vol. 181, no. 1-4, pp. 23–48, 1996.

[11] H. Madsen, "Automatic calibration of a conceptual rainfall-runoff model using multiple objectives," *Journal of Hydrology*, vol. 235, no. 3-4, pp. 276–288, 2000.

[12] T. Hogue, S. Sorooshian, H. Gupta, A. Holz, and D. Braatz, "A multistep automatic calibration scheme for river forecasting models," *Journal of Hydrometeorology*, vol. 1, no. 6, pp. 524–542, 2000.

[13] J. Vrugt, H. Gupta, L. Bastidas, W. Bouten, and S. Sorooshian, "Effective and efficient algorithm for multiobjective optimization of hydrologic models," *Water Resources Research*, vol. 39, no. 8, p. 1214, 2003.

[14] M. Muleta and J. Nicklow, "Sensitivity and uncertainty analysis coupled with automatic calibration for a distributed watershed model," *Journal of Hydrology*, vol. 306, no. 1-4, pp. 127–145, 2005.

[15] C. Cheng, C. Ou, and K. Chau, "Combining a fuzzy optimal model with a genetic algorithm to solve multi-objective rainfall-runoff model calibration," *Journal of Hydrology*, vol. 268, no. 1-4, pp. 72–86, 2002.

[16] T. Ndiritu *et al.*, "An improved genetic algorithm for rainfall-runoff model calibration and function optimization," *Mathematical and Computer Modelling*, vol. 33, no. 6-7, pp. 695–706, 2001.

[17] Sveriges Meteorologiska och Hydrologiska Institut, "The HBV Model," May 2011. [online] `http://www.smhi.se/sgn0106/if/hydrologi/hbv.htm`.

[18] J. Nash and J. Sutcliffe, "River flow forecasting through conceptual models part I – A discussion of principles," *Journal of Hydrology*, vol. 10, no. 3, pp. 282 – 290, 1970.

[19] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[20] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Prentice Hall, second ed., 2003.

[21] P. S. Maybeck, *Stochastic models, estimation, and control*, vol. 141 of *Mathematics in Science and Engineering*. Academic Press, 1979.

[22] Q. Duan, "Shuffled Complex Evolution (SCE-UA) Method," May 2011. [online] `http://www.mathworks.com/matlabcentral/fileexchange/7671`.

[23] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, 1965.

[24] G. Bhattacharyya and R. Johnson, *Statistical concepts and methods*, vol. 4. John Wiley, 1977.

Appendix A

# Python implementation of the Lazy algorithm

Example Python implementation of the Lazy algorithm. The tree is implemented as arrays, one for $\mu$ and one for $\sigma$. Navigation is done by indices.

The select routine uses the interleaved method to interprete the selected path through the tree (see section 4.2). Normalized minimum and maximum values for the chosen actions are calculated, and random normalized parameter values are drawn uniformly from these. Parameter values are de-normalized when applied to the model.

The update routine works as defined for the Lazy algorithm in section 4.3, updating the selected path with the help of the join method and the Kalman filter. For brevity reasons, implementation of the Kalman filter is left out.

```
import random
import math
import array as a
import numpy as np
import _norm_cdf

class HKFBLA:
    def __init__(self, height, dimensions, init_mu,
     init_sigma, observation_sigma,
     transition_sigma, dependence_sigma):
        self.height = height
        self.encoded_height = height*dimensions
        self.dimensions = dimensions
        self.tree_size = 2**(self.encoded_height+1)-1
        self.mu = a.array('f', [init_mu] * self.tree_size)
        self.sigma = a.array('f',
         [init_sigma] * self.tree_size)
        self.observation_sigma = observation_sigma
        self.transition_sigma = transition_sigma
        self.dependence_sigma = dependence_sigma
        self.selected_path = None
        self.scaling_vector = np.array([2**-i for i in
         xrange(1, self.height+1)])
        self.index = 0

    def select(self):
        index = 0
        path = []
        left = 2 * index + 1
        right = 2 * index + 2
        while left < self.tree_size:
            lv = random.gauss(self.mu[left],
             self.sigma[left])
            rv = random.gauss(self.mu[right],
             self.sigma[right])
            valg = 0
            if lv < rv:
                valg = 1
                index = right
            else:
                valg = 0
                index = left
            path.append(valg)
            left = 2 * index + 1
            right = 2 * index + 2

        self.index = index
        temp = np.array(path)
        self.selected_path = temp.reshape(
         self.height, self.dimensions).T
        min_vector = np.dot(
         self.selected_path, self.scaling_vector)
        max_vector = min_vector + 2**-self.height
        random_list = [random.uniform(min, max) for
         min, max in zip(min_vector, max_vector)]
    return random_list

def join_nodes(self, left, right):
    testmu = self.mu[left] - self.mu[right]
    testsigma = math.sqrt(
     self.sigma[left]**2 + self.sigma[right]**2)
    scaledz = - testmu / testsigma
    pxsy = _norm_cdf.norm_cdf(scaledz, 0.0, 1.0)
    new_mu = pxsy * self.mu[right] +
     (1-pxsy) * self.mu[left]
    new_sigma = math.sqrt((pxsy*self.sigma[right])**2
     + ((1-pxsy)*self.sigma[left])**2)
    return (new_mu, new_sigma)

def update(self, reward):
    i = self.index
    self.kalman_filter(reward, i, 0.0)
    parent = i
    while parent > 0:
        i = parent
        sibling = i
        if i % 2 == 0:
            sibling -= 1
        else:
            sibling += 1
        self.kalman_filter(reward, sibling, 1.0)
        parent = (i - 1) / 2
        self.mu[parent], self.sigma[parent] =
         self.join_nodes(i, sibling)
```

# Appendix B

# List of calibrated parameters

| **Parameter** | **Range** | **Unit** |
|---|---|---|
| rain corr | 0.3 – 3.0 | - |
| snow corr | 0.3 – 3.0 | - |
| max liquid in snow | 0.0 – 0.15 | - |
| threshold rain snow | -2.5 – 6.0 | $^\circ C$ |
| degree day factor | 0.5 – 10.0 | $mm/^\circ C/day$ |
| threshold melt | -2.5 – 6.0 | $^\circ C$ |
| threshold freeze | -2.5 – 6.0 | $^\circ C$ |
| refreeze efficiency | 0.0 – 0.5 | - |
| annual et | 0.0 – 3000.0 | $mm/year$ |
| precip grad | 0.9 – 1.4 | $mm/100m$ |
| delta temp grad | 0.0 – 1.0 | $^\circ C/100m$ |
| temp grad precip | -1.0 – -0.01 | $^\circ C/100m$ |
| field capacity | 10.0 – 1000.0 | $mm$ |
| lp | 0.3 – 1.0 | - |
| beta | 0.0 – 10.0 | - |
| max infil soil | 0.1 – 100.0 | $mm/hour$ |
| kuz2 | 0.000001 – 1.0 | - |
| kuz1 | 0.000001 – 1.0 | - |
| klz | 0.000001 – 1.0 | - |
| uz2 | 10.0 – 200.0 | $mm$ |
| uz1 | 10.0 – 200.0 | $mm$ |
| snow dist max | 0.05 – 0.9 | - |
| percolation | 0.02 – 20.0 | $mm/day$ |
| lake fraction | 0.0 – 1.0 | - |
| sol mlt | 0.0 – 50.0 | $mm/(day * W/m^2)$ |

Table B.1: Calibrated Parameters